计算机算法设计与分析

2015 年 3 月 15 日

目录

第一章　复杂性分析初步 1

第一节 空间复杂性 1

第二节 时间复杂性 5

第三节 渐进符号 11

习题 15

第二章　图与遍历算法 19

第一节 图的基本概念和性质 19

第二节 图的遍历算法 26

第三节 双连通图与网络可靠性 33

第三节 双连通图与网络可靠性 33

第四节 对策树 38

习题 43

第三章　分治算法 47

第一节 算法基本思想 47

第二节 排序算法 52

第三节 选择问题 61

第四节 关于矩阵乘法 63

第五节 快速 Fourier 变换 66

第六节 最接近点对问题 68

习题 71

第四章　贪心算法.....**83**

第一节 算法基本思想.....83

第二节 调度问题.....87

第三节 最优生成树问题.....93

第四节 单点源最短路径问题.....97

第五节 Huffman 编码.....99

习题.....102

第五章　动态规划算法.....**107**

第一节 算法基本思想.....107

第二节 多段图问题.....114

第三节 0/1 背包问题.....116

第四节 流水作业调度问题.....123

第五节 最优二叉搜索树.....127

习题.....130

第六章　回溯法.....**133**

第一节 算法基本思想.....133

第二节 定和子集问题和 0/1 背包问题.....136

第三节 n-皇后问题和旅行商问题.....141

第四节 图的着色问题.....145

第五节 回溯算法的效率分析.....147

习题.....153

第七章　分枝-限界算法.....159

第一节 算法基本思想.....159

第二节 0/1 背包问题的分枝限界算法.....162

第三节 电路板布线问题.....166

第四节 优先级的确定与 LC-检索.....169

第五节 旅行商问题的分枝限界算法.....175

习题.....178

第八章 NP-完全问题.....181

第一节 关于问题及算法的描述.....181

第二节 图灵机与确定性算法.....185

第三节 NP-类问题.....187

第四节 NP-完全问题.....194

第五节 证明新问题是 NP-完全问题的方法.....197

第六节 NP-困难问题.....207

习题.....209

第一章 复杂性分析初步

程序性能（program performance）是指运行一个程序所需的内存大小和时间多少。所以，程序的性能一般指程序的空间复杂性和时间复杂性。性能评估主要包含两方面，即性能分析（performance analysis）与性能测量（performance measurement），前者采用分析的方法，后者采用实验的方法。

考虑空间复杂性的理由

- ✓ 在多用户系统中运行时，需指明分配给该程序的内存大小；
- ✓ 想预先知道计算机系统是否有足够的内存来运行该程序；
- ✓ 一个问题可能有若干个不同的内存需求解决方案，从中择取；
- ✓ 用空间复杂性来估计一个程序可能解决的问题的最大规模。

考虑时间复杂性的理由

- ✓ 某些计算机用户需要提供程序运行时间的上限（用户可接受的）；
- ✓ 所开发的程序需要提供一个满意的实时反应。

选取方案的规则：如果对于解决一个问题有多种可选的方案，那么方案的选取要基于这些方案之间的性能差异。对于各种方案的时间及空间的复杂性，最好采取加权的方式进行评价。但是随着计算机技术的迅速发展，对空间的要求往往不如对时间的要求那样强烈。因此我们这里的分析主要强调时间复杂性的分析。

§1 空间复杂性

- **程序所需的空间**主要包括：指令空间、数据空间、环境栈空间。
- **指令空间**－用来存储经过编译之后的程序指令。大小取决于如下因素：
  - ✓ 把程序编译成机器代码的编译器；
  - ✓ 编译时实际采用的编译器选项；
  - ✓ 目标计算机。

注：所使用的编译器不同，则产生的机器代码的长度就会有差异；有些编译器带有选项，如优化模式、覆盖模式等等。所取的选项不同，产生机器代码也会不同。采用优化模式要多消耗时间。此外，目标计算机的配置也会影响代码的规模。例如，如果计算机具有浮点处理硬件，那么，每个浮点操作可以转化为一条机器指令。否则，必须生成仿真的浮点计算代码，使整个机器代码加长。

- **数据空间**－用来存储所有常量和变量的值。
  - ✓ 存储常量和简单变量；所需的空间取决于所使用的计算机和编译器，以及变量与常量的数目；
  - ✓ 存储复合变量；包括数据结构所需的空间及动态分配的空间；

- ✓ 计算方法：

  结构变量所占空间等于各个成员所占空间的累加；数组变量所占空间等于数组大小乘以单个数组元素所占的空间。例如

  double a[100]；所需空间为 100×8＝800

  int matrix[r][c]；所需空间为 4×r×c

C++基本数据类型（32 位字长机器）

类型名	说明	字节数	范围
char	字符型	1	-128～127
signed char	有符号字符型	1	-128～127
unsigned char	无符号字符型	1	0～255
short[int]	短整型	2	-32768～32767
signed short[int]	有符号短整型	2	-32768～32767
unsigned short[int]	无符号短整型	2	0～65535
int	整型	4	-2147483648～2147483647
Signed[int]	有符号整型	4	-2147483648～2147483647
unsigned[int]	无符号整型	4	0～4294967295
long[int]	长整型	4	-2147483648～2147483647
signed long[int]	有符号长整型	4	-2147483648～2147483647
unsigned long[int]	无符号长整型	4	0～4294967295
float	单精度浮点型	4	约 6 位有效数字
double	双精度浮点型	8	约 12 位有效数字
long double	长双精度浮点型	16	约 15 位有效数字

- **环境栈空间**－保存函数调用返回时恢复运行所需要的信息。当一个函数被调用时，下面数据将被保存在环境栈中：
  - ✓ 返回地址；
  - ✓ 所有局部变量的值、递归函数的传值形式参数的值；
  - ✓ 所有引用参数以及常量引用参数的定义（此部分参看附录 1）。
- **实例特征**：决定问题规模的那些因素，主要指输入和输出的数量或相关数的大小。如 对 n 个元素进行排序、n×n 矩阵的加法等， n 作为实例特征；两个 m×n 矩阵的加法，以 n 和 m 两个数作为实例特征。

注：一般情况下，指令空间的大小对于所解决的待定问题不够敏感。常量及简单变量所需要的数据空间也独立于所解决的问题，除非相关数的大小对于所选定的数据类型来说实在太大。这时，要么改变数据类型要么使用多精度算法重写

该程序，然后再对新程序进行分析。定长复合变量和未使用递归函数所需要的环境栈空间都独立于问题的规模。递归函数所需要的栈空间主要依赖于局部变量及形式参数所需要的空间。除此外，该空间还依赖于递归的深度（即嵌套递归调用的最大层次）。

- 程序所需要的空间可分为两部分：

$$S(P) = c + S_P(\text{实例特征})$$

  - **固定部分 c**，它独立于实例的特征。主要包括指令空间、简单变量、常量以及定长复合变量所占用的空间；
  - **可变部分 S_P**，主要包括复合变量所需的空间（其大小依赖于所解决的具体问题）、动态分配的空间（依赖于实例的特征）、递归栈所需的空间（依赖于实例特征）。

注：一个精确的分析还应当包括编译期间所产生的临时变量所需的空间，这种空间与编译器直接相关联，在递归程序中除了依赖于递归函数外，还依赖于实例特征。但是，在考虑空间复杂性时，一般都被忽略。

程序 1-1-1 利用引用参数

```
template < class T >
T Abc(T& a, T& b, T& c)
{
  return a+b+c+b*c\
    +(a+b+c)/(a+b)+4;
}
```

在程序 1-1-1 中，采用数据类型 T 作为实例特征。由于 a，b，c 都是引用参数，在函数中不需要为它们的值分配空间，但需保存指向这些参数的指针。若每个指针需要 2 个字节，则共需要 6 字节的指针空间。此时函数所需要的总空间是一个常量，而 S_{Abc}(实例特征)＝0。

若函数 Abc 的参数是传值参数，则每个参数需要分配 sizeof(T)的空间，于是 a，b，c 所需的空间为 3×sizeof(T)。函数 Abc 所需的其他空间都与 T 无关，故 S_{Abc}(实例特征)＝3×sizeof(T)。

如果使用 a，b，c 的大小作为实例特征，由于在传值参数的情形下，分配给每个变量 a，b，c 的空间均为 sizeof(T)，而不考虑这些变量中的实际值是多大，所以，不论使用引用参数还是使用传值参数，都有 S_{Abc}(实例特征)＝0。

程序 1-1-2 顺序搜索

```
template < class T >
  int SeqSearch(T a[], const T& x, int n)
```

```
{
  //在 a[0:n-1]中搜索 x,若找到则返回所在的位置，否则返回-1
 int i;
 for (i=0; i<n && a[i]=!x; i++);
 if (i==n) return -1;
 return i;
}
```

在程序 1-1-2 中，假定采用被查数组的长度 n 作为实例特征，并取 T 为 int 类型。数组中每个元素需要 4 个字节，实参 x 需要 4 个字节，传值形式参数 n 需要 4 个字节，局部变量 i 需要 4 个字节，整数常量-1 需要 2 个字节，所需要的总的数据空间为 18 个字节，其独立于 n，所以 $S_{SeqSearch}(n)=0$。这里，我们并未把数组 a 所需要的空间计算进来，因为该数组所需要的空间已在定义实际参数（对应于 a）的函数（如主函数）中分配，不能再加到函数 SeqSearch 所需要的空间上去。

再比较下面两个程序，它们都是求已知数组中元素的和。在第一个程序中采用累加的办法，而在第二个程序中则采用递归的办法。我们取被累加的元素个数 n 作为实例特征。在第一个函数中，a，n，i 和 tsum 需要分配空间，与上例同样的理由，程序所需的空间与 n 无关，因此，$S_{Sum}(n)=0$。在第二个程序中，递归栈空间包括参数 a，n 以及返回地址。对于 a 需要保留一个指针，对于 n 则需要保留一个 int 类型的值。如果是 near 指针（需占用 2 个字节），则返回地址需占用 2 个字节，n 需占用 4 个字节，那么每次调用 Rsum 函数共需 8 个字节。该程序递归深度为 n+1，所以需要 8(n+1)字节的递归栈空间。$S_{Rsum}(n)=8(n+1)$。

累加 a[0:n-1]

```
template<class T>
T Sum(T a[], int n)
{//计算 a[0:n-1]的和
  T tsum=0;
  for (int i=0; i<n; i++)
    tsum+=a[i];
  return tsum;
}
```

递归计算 a[0:n-1]

```
template<class T>
  T Rsum(T a[], int n)
  {//计算 a[0:n-1]的和
   if (n>0)
     return Rsum(a,n-1)+a[n-1];
   return 0;
  }
```

嵌套调用一直进行到 n=0。可见，递归计算比累加计算需要更多的空间。

```
Rsum(a, n)
    Rsum(a, n-1)
        .
            .
                .
                Rsum(a, 1)
                    Rsum(a, 0)
```

上述递归程序的嵌套调用层次

§2　时间复杂性

- **时间复杂性的构成**

一个程序所占时间 T(P)＝编译时间+运行时间。

编译时间与实例特征无关，而且，一般情况下，一次编译过的程序可以运行若干次，所以，人们主要关注的是运行时间，记做 t_P(实例特征)；如果了解所用编译器的特征，就可以确定程序 P 进行加、减、乘、除、比较、读、写等所需的时间，从而得到计算 t_P 的公式。令 n 代表实例特征（这里主要是问题实例的规模），则有如下的表示式：

$$t_P(n)=c_a ADD(n)+c_s SUB(n)+c_m MUL(n)+c_d DIV(n)+c_c CMP(n)+\cdots$$

其中，c_a，c_s，c_m，c_d，c_c 分别表示一次加、减、乘、除及比较操作所需的时间，函数 ADD，SUB，MUL，DIV，CMP 分别表示代码 P 中所使用的加、减、乘、除及比较操作的次数；

注：一个算术操作所需的时间取决于操作数的类型（int、float、double 等等），所以，有必要对操作按照数据类型进行分类；在构思一个程序时，影响 t_P 的许多因素还是未知的，所以，在多数情况下仅仅是对 t_P 进行估计。估算运行时间的方法主要有两种：A. 选一种或多种关键操作，确定这些关键操作所需要的执行时间（对于前面所列举的四种算术运算及比较操作，一般被看作是基本操作，并约定所用的时间都是一个单位）及每种关键操作执行的次数；B. 确定程序总的执行步数。

- **操作计数**

首先选择一种或多种操作（如加、乘和比较等），然后计算这些操作分别执行了多少次。关键操作对时间复杂性影响最大。

程序 1-2-1 寻找最大元素

```
template<class T>
int Max(T a[], int n)
```

```
{//寻找 a[0:n-1]中的最大元素
  int pos=0;
  for (int i=1; i<n; i++)
   if (a[pos]<a[i])
     pos=i;
  return pos;
}
```

这里的关键操作是比较。for 循环中共进行了 n-1 次比较。Max 还执行了其它比较，如 for 循环每次比较之前都要比较一下 i 和 n。此外，Max 还进行了其他的操作，如初始化 pos、循环控制变量 i 的增量。这些一般都不包含在估算中，若纳入计数，则操作计数将增加到一个常数倍。

程序 1-2-2 n 次多项式求值

```
template<class T>
 T PolyEval(T coeff[], int n, const T& x)
 {//计算 n 次多项式的值，coeff[0:n]为多项式的系数
 T y=1, value=coeff[0];
 for (int i=1; i<=n; i++)     //n 循环
   {//累加下一项
     y*=x;                    //一次乘法
     value+=y*coeff[i];      //一次加法和一次乘法
    }
 return value;
}                            //3n 次基本运算
```

Horner 法则：

$$P(x)=(\cdot\cdot\cdot(c_n*x+c_{n-1})*x+c_{n-2})*x+c_{n-3})*x+\cdot\cdot\cdot)*x+c_0$$

程序 1-2-3 利用 Horner 法则求多项式的值

```
template<class T>
 T Horner(T coeff[], int n, const T& x)
 {//计算 n 次多项式的值，coeff[0:n]为多项式的系数
   T value=coeff[n];
   for(i=1; i<=n; i++)              //n 循环
```

```
        T value＝value*x+coeff[n-i];  //一次加法和一次乘法
      return value;
    }                                  //2n 次基本运算
```

这里，关键操作是加法与乘法运算。在程序 1-2-2 中，for 循环的每一回都执行两次乘法运算和一次加法运算。所以程序的总的运算次数为 3n。 在程序 1-2-3 中，for 循环的每一回都执行乘法与加法运算各一次，程序总的运算次数为 2n。再考察下面两例：

程序 1-2-4 计算名次

```
template<class T>
void Rank(T a[], int n, int r[])
{//计算 a[0:n-1]中元素的排名
  for(int i=0; i<n; i++)
     r[i]=0;  //初始化
     //逐对比较所有的元素
  for(int i=1; i<n; i++)
    for(int j=0; j<i; j++)
      if(a[j]<=a[i]) r[i]++;
      else r[j]++;
}
```

程序 1-2-5 选择排序

```
template<class T>
 void SelectSort(T a[], int n)
 {//对数组 a[0:n-1]中元素排序
  for(int size=n; size>1; size--)
   { int j=Max(a,size);
   Swap(a[j],a[size-1]);
   }
 }
```

其中函数 Max 定义如程序 1-2-1 而函数 Swap 由下述程序给出

程序 1-2-6 交换两个值

```
template<class T>
inline void Swap(T& a, T& b)
{//交换 a 和 b
  T temp=a; a=b; b=temp;
}
```

在这两个程序中，关键操作都是比较。在 1-2-4 中，对于每个 i ，执行比较的次数是 i，总的比较次数为 1+2+ ・・・ + n-1= n-1)n/2。(在此，for 循环的额外开销、初始化数组 r 的开销、以及每次比较 a 中两个素时对 r 进行的增值开销都未列入其中。在程序 1-2-5 给出的排序中，首先找出最大元素，把它移到 a[n-1]，然后在余下的 n-1 个元素中再寻找最大的元素，并把它移到 a[n-2].如此进行下去，此种方法称为选择排序(SelectSort).从程序 1-2-1 中已经知道，每次调用 Max(a,size)需要执行 size-1 次比较，所以总的比较次数为 1+2+ ・・・ + n-1=(n-1)n/2.每调用一次函数 Swap 需要执行三次元素移动，所需要总的移动次数为 3(n-1)。

考虑**冒泡排序**（SubbleSort）。冒泡排序是从左向右逐个检查，对相邻的元

素进行比较，若左边的元素比右边的元素大，则交换这两个元素的位置。如数组[5,3,7,4,11,2]：比较 5 和 3，交换；比较 5 和 7，不交换；比较 7 和 4，交换；比较 7 和 11，不交换；比较 11 和 2，交换。这个行程称为一次冒泡，经一次冒泡后，原数组变为[3,5,4,7,2,11].可见，数组中最大的数被移动到最右的位置。下一次冒泡只对数组[3,5,4,7,2]进行即可。如此进行下去，最后将原数组按递增的顺序排列。

程序 1-2-7 一次冒泡

```
template<class T>
void Bubble(T a[], int n)void
{
  for(int i=0; i<n-1; i++)
    if(a[i]>a[i+1]) Swap(a[i],a[i+1]);
}
```

程序 1-2-8 冒泡排序

```
Template<class T>
BubbleSort(T a[], int n)
{
  for(int i =n; i >1; i --)
    Bubble(a, i );
}
```

在程序 1-2-7 中，for 循环的每一回都执行了一次比较和至多三次元素的移动，因而总的操作数为 4(n-1)。在程序 1-2-8 中，对于每个 i，调用函数 Bubble(a,i)需要执行 4(i-1)次操作，因而总的操作数为 2n(n-1).这里我们照例忽略了非关键操作。

分析程序 1-2-7 发现，在问题实例中，如果数组本身是递增的，则每一次冒泡都不需要交换数据，而如果数组是严格递减的，则第一次冒泡要交换数据 n-1 次。这里我们都取数组元素个数为实例特征，但操作数却是不同的。因而，人们往往还会关心最好的、最坏的和平均的操作数是多少。令 P 表示一个程序，将操作计数 $O_P(n_1,n_2,\cdots,n_k)$ 视为实例特征 $n_1,n_2,\cdots,n_k$ 的函数。令 $operation_P(I)$ 表示程序对实例 I 的操作计数，$S(n_1,n_2,\cdots,n_k)$ 为所讨论程序的具有实例特征 $n_1,n_2,\cdots,n_k$ 的实例之集，则

P 最好的操作计数为

$$O_P^{BC}(n_1,n_2,\cdots,n_k)=\min\{operation_P(I)\mid I\in S(n_1,n_2,\cdots,n_k)\}$$

P 最坏的操作计数为

$$O_P^{WC}(n_1,n_2,\cdots,n_k)=\max\{operation_P(I)\mid I\in S(n_1,n_2,\cdots,n_k)\}$$

P 平均的操作计数为

$$O_P^{AVG}(n_1,n_2,\cdots,n_k)=\frac{1}{|S(n_1,n_2,\cdots,n_k)|}\sum_{I\in S(n_1,n_2,\cdots,n_k)}operation_P(I)$$

P 操作计数的期望值为

$$O_P^{AVG}(n_1,n_2,\cdots,n_k)=\sum_{I\in S(n_1,n_2,\cdots,n_k)}p(I)\cdot operation_P(I)$$

其中，$p(I)$是可被成功解决的实例I出现的概率。

在前面的例子中，一般计算的都是程序的最坏操作计数。如在冒泡程序 Bubble 中即是如此。在顺序搜索 SeqSearch(T a[], const T& x, int n)中，取 n 作为实例特征，关键操作数是比较。此时，比较的次数并不是由 n 唯一确定的。若 n=100，x=a[0]，那么仅需要执行一次操作；若 x 不是 a 中的元素，则需要执行 100 次比较。当 x 是 a 中一员时称为成功查找，否则称为不成功查找。每回不成功的查找，需要执行 n 次比较。对于成功的查找，最好的比较次数是 1，最坏的比较次数为 n。若假定每个实例出现的概率都是相同的，则平均比较次数为

$$\frac{1}{n+1}\left(n+\sum_{i=1}^{n}i\right)=\frac{n}{(n+1)}+\frac{n}{2}$$

再考察插入排序算法：

程序 1-2-9 向有序数组插入元素

```
template<class T>
void Insert(T a[], int& n, const T& x)
{//向数组 a[0:n-1]中插入元素 x
 //假定 a 的大小超过 n
 int i;
 for(i=n-1; i>=0 && x<a[i]; i--)
   a[i+1]=a[i];
 a[i+1]=x;
 n++; //添加了一个元素
 }
```

程序 1-2-10 插入排序

```
template<class T>
void InsertSort(T a[], int n)
{//对 a[0:n-1]进行排序
 for(int i=1; i<n; i++) {
   T t =a[i];
   Insert(a, i, t);
  }
}
```

在程序 1-2-9 中假定：<u>a 中元素在 x 被插入前后都是按递增的顺序排列的</u>。我们取初始数组 a 的大小 n 作为实例特征，则程序 1-2-9 的关键操作是 x 与 a 中元素的比较。显然，最少的比较次数为 1，这种情况发生在 x 被插在数组尾部的时候；最多的比较次数是 n，发生在 x 被插在数组的首部或者数组的第一元素之后的时候。如果 x 最终被插在 a 的 i+1 处，i>=0,则执行的比较次数是 n-i。如果 x 被插在 a[0]之前的位置，则比较的次数为 n。假定 x 有相等的机会被插在任何一个可能的位置上，则平均比较次数是

$$\frac{1}{n+1}\left(n+\sum_{i=0}^{n-1}(n-i)\right)=\frac{1}{n+1}\left(n+\sum_{j=1}^{n}j\right)=n/2+n/(n+1)$$

在程序 1-2-10 中所执行的比较次数，最好情况下是 n-1，最坏情况下是 n(n-1)/2.

虽然在这些简单例子中，我们都给出了平均操作数，但是在一般情况下，平均操作数不是很容易求得的，操作数的数学期望值就更不容易求得了。

- **执行步数**

利用操作计数方法估计程序的时间复杂性注意力集中在“关键操作”上，忽略了所选择操作之外其他操作的开销。下面所要讲的统计执行步数（step-count）的方法则要统计程序/函数中所有部分的时间开销。执行步数也是实例特征的函数，通常做法是选择一些感兴趣的实例特征。如，若要了解程序的运行时间是如何随着输入数据的个数增加而增加的，则把执行步数仅看作输入数据的个数的函数。所谓程序步是一个语法或语意上的程序片断，该片段的执行时间独立于实例特征。例如 语句：return a+b+b*c+(a+b+c)/(a+b)+4；和 x=y;都可以作为程序步。一种直观的统计程序执行步数的方法是做执行步数统计表

矩阵加法程序执行步数统计表

语　　句	s/e	频率	总步数
Void Addm(T **a, …)	0	0	0
{	0	0	0
for(int i=0; i<rows; i++)	1	rows+1	rows+1
for(int j=0; j<cols; j++)	1	rows*(cols+1)	rows*cols+rows
c[i][j]=a[i][j]+b[i][j];	1	rows*cols	rows*cols
}	0	0	0
总　　计	2*rows*cols+2*rows+1		

其中 s/e 表示每次执行该语句所要执行的程序步数。一条语句的 s/e 就等于执行该语句所产生的 count 值的变化量。频率是指该语句总的执行次数。

实际统计中，可以通过创建全局变量 count 来确定一个程序或函数为完成其预定任务所需要的执行步数。将 count 引入到程序语句之中，每当原始程序或函数中的一条语句被执行时，就为 count 累加上该语句所需要的执行步数。

程序 1-2-11 矩阵加法与执行步数统计

```
template<class T>
void Addm(T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c
    for(int i=0; i<rows; i++){
```

```
        count++; //对应于上一条 for 语句  (共执行了 rows 步)
        for(int j=0; j<cols; j++){
          count++; //对应于上一条 for 语句(共执行了 rows×cols 步)
          c[i][j]=a[i][j]+b[i][j];
          count++; //对应于赋值语句       (共执行了 rows×cols 步)
        }
      count++; //对应 j 的最后一次 for 循环  (共执行了 rows 步)
      }
  count++; //对应 i 的最后一次 for 循环     (共执行了 1 步)
}
```

若取 rows 和 cols 为实例特征，则，程序 1-2-11 总的执行步数为 2×rows×cols＋2×rows＋1。据此，若 rows > cols，我们可以通过交换程序中 i 循环与 j 循环的次序而使程序所用时间减少。

§3 渐近符号

确定程序的操作计数和执行步数的目的是为了比较两个完成同一功能的程序的时间复杂性，预测程序的运行时间随着实例特征变化的变化量。设$T(n)$是算法 A 的复杂性函数。一般说来，当n单调增加趋于∞时，$T(n)$也将单调增加趋于∞。如果存在函数$\widetilde{T}(n)$，使得当$n\to\infty$时有$(T(n)-\widetilde{T}(n))/T(n)\to 0$，则称$\widetilde{T}(n)$是$T(n)$当$n\to\infty$时的渐近性态，或称$\widetilde{T}(n)$是算法 A 当$n\to\infty$的渐近复杂性。因为在数学上，$\widetilde{T}(n)$是$T(n)$当$n\to\infty$时的渐近表达式，$\widetilde{T}(n)$可以是$T(n)$中略去低阶项所留下的主项，所以它无疑比$T(n)$来得简单。

进一步分析可知，要比较两个算法的渐近复杂性，只要确定出各自的阶即可知道哪一个算法的效率高。换句话说，渐近复杂性分析只要关心$\widetilde{T}(n)$的阶就够了，不必关心包含在$\widetilde{T}(n)$中的常数因子。所以，对$T(n)$的分析又常常做进一步简化，即假设算法中用到的所有不同的运算（基本）各执行一次所需要的时间都是一个单位时间。

综上分析，我们已经给出了简化算法复杂性分析的方法和步骤，即只考虑当问题的规模充分大时，算法复杂性在渐近意义下的阶。为此引入渐近符号，首先给出常用的渐近函数。

常 用 的 渐 进 函 数

函数	名称	函数	名称
1	常数	n^2	平方
log n	对数	n^3	立方
n	线性	2^n	指数
n log n	n 倍 long n	n!	阶乘

在下面的讨论中，用 f(n)表示一个程序的时间或空间复杂性，它是实例特征 n（一般是输入规模）的函数。由于一个程序的时间或空间需求是一个非负的实数，我们假定函数 f(n)对于 n 的所有取值均为非负实数，而且还可假定 n>=0。

渐近符号 O 的定义：f(n)=0(g(n))当且仅当存在正的常数 c 和 n_0，使得对于所有的 n>=n_0，有 f(n)<=cg(n)。此时，称 g(n)是 f(n)的一个渐近上界。

函数 f 至多是函数 g 的 c 倍，除非 $n < n_0$。即是说，当 n 充分大时，在不计较相差一个非零常数倍的情况下，g 是 f 的一个上界函数。通常情况下，上界函数取单项的形式，如表 1 所列。

C_0 = 0(1)： f(n) 等于非零常数的情形。

3n+2 = 0(n)：可取 c＝4，n_0＝2 。

100n+6 = 0(n)：可取 c=101，n_0＝6 。

$10n^2+4n+3$ = 0(n^2)：可取 c=11,n_0=5 。

$6\times2^n+n^2$ = 0(2^n)：可取 c =7，n_0=4 。

$3\times\log n + 2\times n + n^2$ =0(n^2)．可取 c =2，n_0=5 。

$n\times\log n + n^2$= 0(n^2)．可取 c =2，n_0=3 。

3n+2 = 0(n^2)．可取 c =1，n_0=3 。

三点注意事项：

1． 用来作比较的函数 g(n)应该尽量接近所考虑的函数 f(n)．

3n+2=0(n^2) 是松散的界；3n+2=0(n) 是较好的界。

2． 不要产生错误界。

n^2+100n+6，当 n<3 时，n^2+100n+6<106n，由此就认为 n^2+100n+6＝0(n)，这是不对的。 事实上，对任何正的常数 c，只要 n>c-100 就有 n^2+100n+6>c×n。同理，$3n^2+4\times2^n$=0(n^2)是错误的界。

3.　f(n)=O(g(n))不能写成 g(n)=O(f(n))，因为两者并不等价。
实际上，这里的等号并不是通常相等的含义。按照定义，用集合符号更准确些：
O(g(n))={f(n)|f(n)满足：存在正的常数 c 和 n_0，使得当 n>=n_0时，f(n)<=cg(n)}
所以，人们常常把 f(n)=O(g(n))读作："f(n)是 g(n)的一个大 O 成员"。

大 0 比率定理：对于函数 $f(n)$ 和 $g(n)$，如果极限 $\lim_{n\to\infty}(f(n)/g(n))$ 存在，则 $f(n)=O(g(n))$ 当且仅当存在正的常数 c，使得 $\lim_{n\to\infty}(f(n)/g(n))\le c$.

例子　因为 $\lim_{n\to\infty}\frac{3n+2}{n}=3$，所以 $3n+2=O(n)$；

因为 $\lim_{n\to\infty}\frac{10n^2+4n+2}{n^2}=10$，所以 $10n^2+4n+2=O(n^2)$；

因为 $\lim_{n\to\infty}\frac{6*2^n+n^2}{2^n}=6$，所以 $6*2^n+n^2=O(2^n)$；

因为 $\lim_{n\to\infty}\frac{n^{16}+3*n^2}{2^n}=0$，所以 $n^{16}+3*n^2=O(2^n)$，

但是，最后一个不是好的上界估计，问题出在极限值不是正的常数。
下述不等式对于复杂性阶的估计非常有帮助：

定理 1.3.1. 对于任意给定的正实数 c、x 和 ε，有下面的不等式：

1）存在某个 n_0 使得对于任何 $n\ge n_0$，有 $(c\cdot\log n)^x<(\log n)^{x+\varepsilon}$。

2）存在某个 n_0 使得对于任何 $n\ge n_0$，有 $(\log n)^x<n$。

3）存在某个 n_0 使得对于任何 $n\ge n_0$，有 $(c+n)^x<n^{x+\varepsilon}$。

4）存在某个 n_0 使得对于任何 $n\ge n_0$，有 $n^x<2^n$。

5）对任意实数 y，存在某个 n_0 使得对于任何 $n\ge n_0$，有 $n^x(\log n)^y<n^{x+\varepsilon}$。

例子　根据定理 1，我们很容易得出：$n^3+n^2\log n=O(n^3)$；$n^4+n^{2.5}\log^{20}n=O(n^4)$；$2^n n^4\log^3 n+2^n n^5/\log^3 n=O(2^n n^5)$。

符号 Ω 的定义：$f(n)=\Omega(g(n))$ 当且仅当存在正的常数 c 和 n_0，使得对于所有的 $n\ge n_0$，有 $f(n)\ge c(g(n))$。此时，称 g(n)是 f(n)的一个渐近下界。

函数 f 至少是函数 g 的 c 倍，除非 $n<n_0$。即是说，当 n 充分大时，在不计

较相差一个非零常数倍的情况下，g 是 f 的一个下界函数。类似于大 O 符号，我们可以参考定理 1.3.1 所列的不等式，来估计复杂性函数的渐近下界，而且有下述判定规则：

大 Ω 比率定理：对于函数 $f(n)$ 和 $g(n)$，如果极限 $\lim_{n\to\infty}(f(n)/g(n))$ 存在，则 $f(n)=\Omega(g(n))$ 当且仅当存在正的常数 c，使得 $\lim_{n\to\infty}(f(n)/g(n))\ge c$.

符号 Θ 的定义：$f(n)=\Theta(g(n))$ 当且仅当存在正的常数 c_1,c_2 和 n_0，使得对于所有的 $n\ge n_0$，有 $c_1(g(n))\le f(n)\le c_2(g(n))$。此时，称 f(n) 与 g(n) 同阶。

函数 f 介于函数 g 的 c_1 和 c_2 倍之间，即当 n 充分大时，在不计较相差非零常数倍的情况下，g 既是 f 的下界，又是 f 的上界。

例子 $3n+2=\Theta(n)$; $10n^2+4n+2=\Theta(n^2)$。

$5\times 2^n+n^2=\Theta(2^n)$;

Θ 比率定理：对于函数 $f(n)$ 和 $g(n)$，如果极限 $\lim_{n\to\infty}(g(n)/f(n))$ 与 $\lim_{n\to\infty}(f(n)/g(n))$ 都存在，则 $f(n)=\Theta(g(n))$ 当且仅当存在正的常数 c_1,c_2，使得 $\lim_{n\to\infty}(f(n)/g(n))\le c_1, \lim_{n\to\infty}(g(n)/f(n))\le c_2$.

例子：$t_{sum}(n)=2n+3=\Theta(n)$；$t_{Add}(m,n)=2mn+2n+1=\Theta(mn)$；

$t_{SeqSeach}^{AVG}(n)=\alpha(n+7)/2+(1-\alpha)(n+3)=\Theta(n)$，其中 α 表示被查元素 x 在数组 a 中的概率。

比较大 O 比率定理和 Ω 比率定理，可知，Θ 比率定理实际是那两种情况的综合。对于多项式情形的复杂性函数，其阶函数可取该多项式的最高项，即

定理 1.3.2. 对于多项式函数 $a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+a_0$，如果 $a_m>0$，则

$$f(n)=O(n^m),\quad f(n)=\Omega(n^m),\quad f(n)=\Theta(n^m)$$

一般情况下，我们不能对每个复杂性函数直接去估计它们的渐进上界、渐进下界和“双界”，定理 1.3.1 和定理 1.3.2 给了一些直接确定这些界的阶函数（或

叫渐近函数）的参考信息。由这些信息可以给出多个函数经过加、乘运算组合起来的复杂函数的阶的估计。

小 o 符号定义： $f(n)=o(g(n))$ 当且仅当 $f(n)=O(g(n))$ 和 $g(n)\neq O(f(n))$。

一般情况下，$f(n)=o(g(n))$ 的充分必要条件是 $\lim_{n\to\infty}(f(n)/g(n))=0$ 。

最后给出折半搜索程序及算法复杂性估计，这里假定被查找的数组已经是单调递增的。

程序 1-3-1 折半搜索

```
template<class T>
int BinarySearch(T a[], const T& x, int n)
{//在 a[0]<=a[1]<=・・・<=a[n-1]中搜索 x
 //如果找到，则返回所在位置，否则返回 －1
 int left=0; int right=n-1;
 while(left<=right){
   int middle=(left+right)/2;
   if(x==a[middle]) return middle;
   if(x>a[middle]) left=middle+1;
   else right=middle － 1;
   }
 return －1; //未找到 x
}
```

while 的每次循环（最后一次除外）都将以减半的比例缩小搜索范围，所以，该循环在最坏的情况下需要执行 $\Theta(\log n)$ 次。由于每次循环需耗时 $\Theta(1)$，因此在最坏情况下，总的时间复杂性为 $\Theta(\log n)$。

习题 一

1. 试确定下述程序的执行步数，该函数实现一个 m×n 矩阵与一个 n×p 矩阵之间的乘法：

矩阵乘法运算

```
template<class T>
void Mult(T **a, T **b, int m, int n, int p)
```

```
{//m×n 矩阵 a 与 n×p 矩阵 b 相成得到 m×p 矩阵 c
  for(int i=0; i<m; i++)
    for(int j=0; j<p; j++){
      T sum=0;
      for(int k=0; k<n; k++)
        Sum+=a[i][k]*b[k][j];
      C[i][j]=sum;
    }
}
```

2． 函数 MinMax 用来查找数组 a[0:n-1]中的最大元素和最小元素，以下给出两个程序。令 n 为实例特征。试问：在各个程序中，a 中元素之间的比较次数在最坏情况下各是多少？

找最大最小元素 （方法一）

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{//寻找 a[0:n-1]中的最小元素与最大元素
  //如果数组中的元素数目小于 1，则还回 false
  if(n<1) return false;
  Min=Max=0; //初始化
   for(int i=1; i<n; i++){
      if(a[Min]>a[i]) Min=i;
      if(a[Max]<a[i]) Max=i;
      }
   return true;
}
```

找最大最小元素 （方法二）

```
template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{//寻找 a[0:n-1]中的最小元素与最大元素
  //如果数组中的元素数目小于 1，则还回 false
  if(n<1) return false;
  Min=Max=0; //初始化
   for(int i=1; i<n; i++){
```

```
        if(a[Min]>a[i]) Min=i;
        else if(a[Max]<a[i]) Max=i;
        }
    return true;
}
```

3. 证明以下关系式不成立：

1). $10n^2+9=O(n)$;

2). $n^2\log n=\Theta(n^2)$；

4. 证明：当且仅当$\lim\limits_{n\to\infty} f(n)/g(n)=0$时，$f(n)=o(g(n))$。

5. 下面那些规则是正确的？为什么？

1). $\{f(n)=O(F(n)),g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=O(F(n)/G(n))$；

2). $\{f(n)=O(F(n)),g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=\Omega(F(n)/G(n))$；

3). $\{f(n)=O(F(n)),g(n)=O(G(n))\}\Rightarrow f(n)/g(n)=\Theta(F(n)/G(n))$；

4). $\{f(n)=\Omega(F(n)),g(n)=\Omega(G(n))\}\Rightarrow f(n)/g(n)=\Omega(F(n)/G(n))$；

5). $\{f(n)=\Omega(F(n)),g(n)=\Omega(G(n))\}\Rightarrow f(n)/g(n)=\mathrm{O}(F(n)/G(n))$。

6). $\{f(n)=\Theta(F(n)),g(n)=\Theta(G(n))\}\Rightarrow f(n)/g(n)=\Theta(F(n)/G(n))$

6. 按照渐近阶从低到高的顺序排列以下表达式:

$$4n^2,\log n,3^n,20n,n^{2/3},n!$$

7. 1）假设某算法在输入规模是n时为$T(n)=3*2^n$. 在某台计算机上实现并完成该算法的时间是t秒. 现有另一台计算机, 其运行速度为第一台的 64 倍, 那么, 在这台计算机上用同一算法在t秒内能解决规模为多大的问题?

2）若上述算法改进后的新算法的时间复杂度为$T(n)=n^2$，则在新机器上用t秒时间能解决输入规模为多大的问题？

3）若进一步改进算法，最新的算法的时间复杂度为 $T(n)=8$，其余条件不变，在新机器上运行，在t秒内能够解决输入规模为多大的问题？

8. Fibonacci 数有递推关系：

$$F(n)=\begin{cases}1, & n=0\\ 1, & n=1\\ F(n-1)+F(n-2), & n>1\end{cases}$$

试求出 $F(n)$ 的表达式。

第二章 图与遍历算法

§1 图的基本概念和术语

- 无向图(undirected graph)

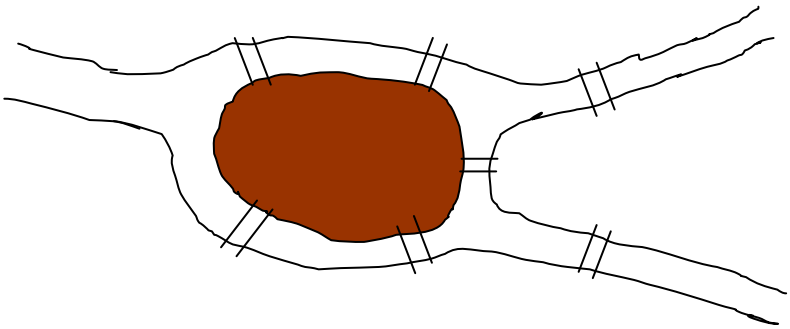


图 2-1-1 哥尼斯堡七桥

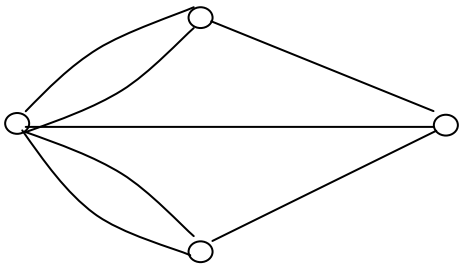


图 2-1-2 Euler 图

无向图，简称图，是一个用线（边）连接在一起的结点（顶点）的集合。严格地说，图是一个三元组 G=(V, E, I)，其中，V 是顶点的集合，E 是边的集合，而 I 是关联关系，它指明了 E 中的每条边与 V 中的每个顶点之间的关联关系：每条边必定连接两个而且只有两个顶点，它们称为该边的端点。有边相连的两个顶点称为相邻的。连接顶点 v 的边的条数称为 v 的度，记做 d(v). 图 G=(V, E, I)中顶点的度与边数有如下的 Euler 公式

$$\sum_{v\in V} d(v) = 2|E| \qquad (2.1.1)$$

由公式(2.1.1)可知，*图中奇度顶点的个数一定是偶数*。

没有重复边的图称为简单图，n 阶完全图是指具有 n 个顶点而且每两个顶点之间都有边连接的简单图，这样的图的边数为 n(n-1)/2.以下是 1～4 阶的完全图：

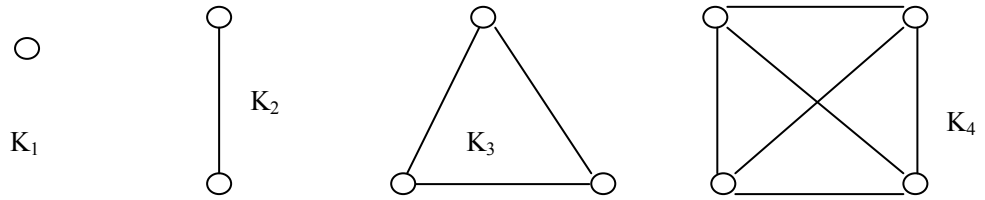


图 2-1-3 几个完全图

另一类特殊的图是偶图（也叫二部图），它的顶点集分成两部分 V_1，V_2，每部分中的顶点之间不相连（没有边连接）。

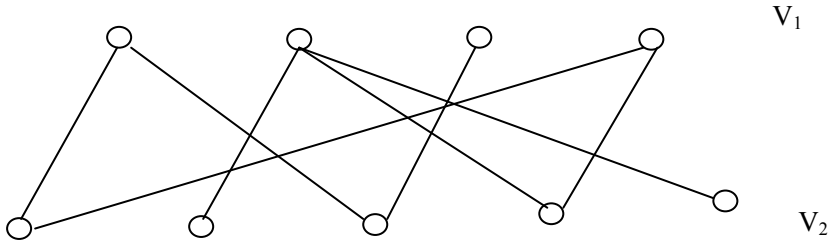


图 2-1-4　一个偶图

当然，k-部图，是指图的顶点集被分成 k 个部分，同部分的顶点之间没有边相连。

设图 G 的顶点集是 $V=\{v_1,v_2,\cdots,v_n\}$，边集是 $E=\{e_1,e_2,\cdots,e_m\}$，则顶点与顶点之间的邻接关系可以用如下矩阵表示：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \end{array} = A$$

称为邻接矩阵，其中

$$a_{ij}=\begin{cases} 1 & \text{如果}(v_i,v_j)\text{是}G\text{的一条边} \\ 0 & \text{否则} \end{cases}$$

顶点与边之间的关联关系可以用如下矩阵表示：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_m \end{array} \\ \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n1} & \cdots & b_{nm} \end{pmatrix} \end{array} = M$$

称为关联矩阵，其中

$$b_{ij} = \begin{cases} 1 & \text{如果 } v_i \text{ 是边 } e_j \text{ 的一个端点} \\ 0 & \text{否则} \end{cases}$$

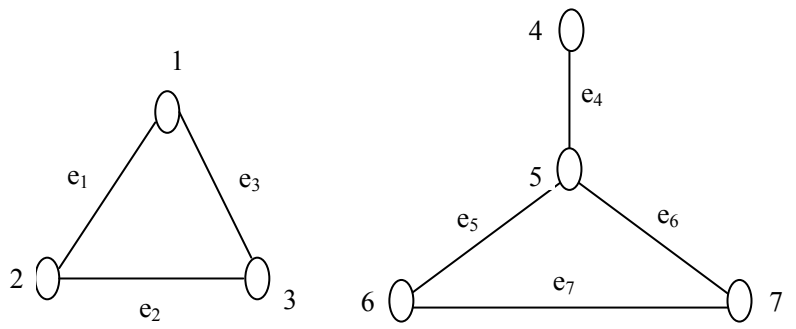


图 2-1-5 一个具有 7 个定点、7 条边的图

图 2-1-5 的邻接矩阵 A、关联矩阵 M 分别为：

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

图的另一个重要概念是路径，区分为途径、迹和路。

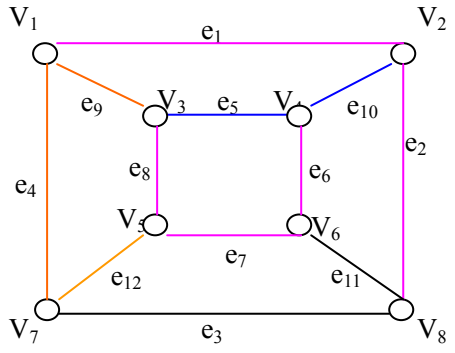
途径：顶点与边交叉出现的序列

$$v_0\, e_1\, v_1\, e_2\, v_2 \cdots e_l\, v_l \tag{2.1.2}$$

其中 e_l 的端点是 v_{i-1} 和 v_i 。 <u>迹</u>是指边不重复的途径，而顶点不重复的途径称为**路**。起点和终点重合的途径称为闭途径，起点和终点重合的迹称为闭迹，顶点不重复的闭迹称为**圈**。没有圈的图称为**森林**。

如果存在一条以 u 为起点、v 为终点的途径，则说顶点 u 可达顶点 v。如果图 G 中任何两个顶点之间都是可达的，则说图 G 是连通的。如果图 G 不是连通的，则其必能分成几个连通分支。图 2-1-6 是连通的，而图 2-1-5 不是连通的，它有

两个连通分支。



一条途径：

$v_1e_1v_2e_{10}v_4e_5v_3e_9v_1e_1v_2e_2v_8$

一条迹：

$v_1e_1v_2e_{10}v_4e_5v_3e_9v_1e_4v_7$

一条路：

$v_1e_1v_2e_{10}v_4e_5v_3e_8v_5e_{12}v_7$

图 2-1-6　立方体

不含圈的连通图称为**树**。森林的每个连通分支都是树，也就是说，森林是由树组成的。对于连通图，适当去掉一些边后（包括去掉零条边），会得到一个不含圈、而且包含所有顶点的连通图，它是一棵树，称为原图的**生成树**。一棵具有 n 个顶点的树的边数恰好为 $n-1$。所以，一个具有 k 个连通分支的森林恰好有 $n-k$ 条边。一个具有 n 个顶点的连通图至少有 n-1 条边；一个具有 n 个顶点，k 个连通分支的图至少有 $n-k$ 条边.

定理 2.1.1 如果 G 是具有 n 个顶点、m 条边的图，则下列结论成立：

1. 若 G 是树， 则 m = n-1；
2. 若 G 是连通图，而且满足 m = n-1，则 G 是树；
3. 若 G 不包含圈，而且满足 m = n-1，则 G 是树；
4. 若 G 是由 k 棵树构成的森林，则 m = n-k ；
5. 若 G 有 k 个连通分支，而且满足 m = n-k，则 G 是森林。

由图 G 的部分顶点和部分边按照它们在 G 中的关联关系构成的图称为 G 的子图，如果子图 H 的顶点集恰是 G 的顶点集，则 H 称为 G 的生成子图。可见，当 G 是连通图时，它的生成树是一种特殊的生成子图，它是 G 的既连通又边数最少的一个生成子图。一个连通图的生成树不是唯一的。

- **有向图**

描述单行道系统就不能用无向图，因为它不能指明各条路的方向。所谓有向图实际上是在无向图的基础上进一步指定各条边的方向。在有向图中每一条有向边可以用一对顶点表示：$e=(u,v)$，u 为始点，v 为终点，这条边是由顶点 u 指向顶点 v 的。

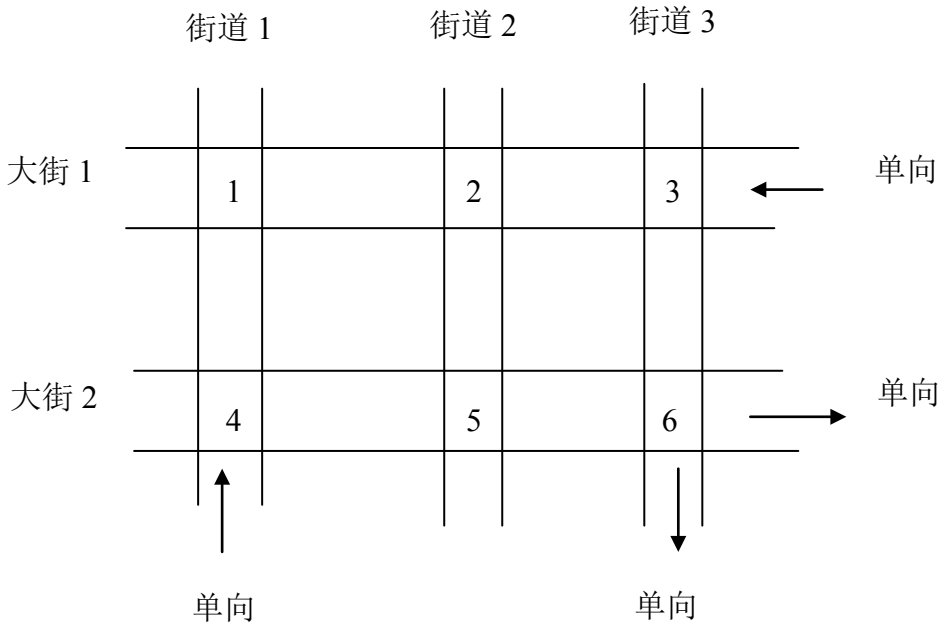


图 2-1-7 具有单行线的交通示意图

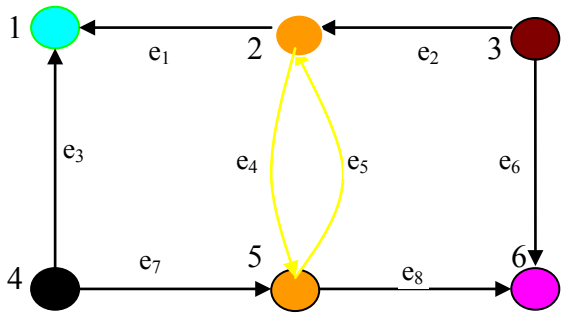


图 2-1-8
一个有向图及其双向连通分支

指向顶点 v 的有向边的条数称为顶点 v 的入度，记做 $d^-(v)$；而由顶点 u 出发的有向边的条数称为顶点 u 的出度，记做 $d^+(u)$。一个顶点 w 的出度与入度之和称为该顶点的度，记做 $d(w)$，即 $d(w)=d^+(w)+d^-(w)$。有向图 $G=(V,E,I)$ 的顶点的度和边数之间有如下的关系：

$$|E|=\sum_{v\in V}d^-(v)=\sum_{v\in V}d^+(v) \qquad (2.1.3)$$

类似地，有向图的表示也可以用邻接矩阵和关联矩阵，但是为指明边的方向，只用 0,1 两个元素是不够的。设有向图 G 的顶点集和边集分别是 $V=\{v_1,v_2,\cdots,v_n\}$ 和 $E=\{e_1,e_2,\cdots,e_m\}$，则邻接矩阵定义如下：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \end{array} = A$$

其中

$$a_{ij} = \begin{cases} 1 & \text{如果}(v_i, v_j)\text{是}G\text{的一条有向边} \\ 0 & \text{否则} \end{cases}$$

关联矩阵定义如下：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_m \end{array} \\ \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n1} & \cdots & b_{nm} \end{pmatrix} \end{array} = M$$

其中

$$b_{ij} = \begin{cases} 1 & \text{如果 } v_i \text{ 是有向边 } e_j \text{的始点} \\ -1 & \text{如果 } v_i \text{ 是有向边 } e_j \text{的终点} \\ 0 & \text{其它} \end{cases}$$

如，图 2-1-8 的关联矩阵为

$$M = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \end{pmatrix}$$

每个列上恰有一个 1 和一个 -1,代表一条有向边的始点和终点。

在有向图中，许多概念都可以通过无向图的相关概念加“有向”二字得到，如：有向边、有向途径、有向迹、有向路、有向圈，等等。有向图和无向图可以相互转化：将一个无向图的每条边都规定方向后，即得到有向图，其称为原无向图的一个定向图；将一个有向图的各条有向边的方向去掉，即得到一个无向图，其称为原有向图的基础图。

有向图中也有一些概念不能由无向图通过简单地附加“有向”一词而得到。如，连通，有向图 D 说是连通的是指其基础图是连通的。如果 D 中任意两个顶点都是相互有向可达的，则说有向图 D 是双向连通的（或叫强连通）。这里，顶点 u 可达顶点 v（有向可达）是指存在一条以 u 为起点、v 为终点的有向路。这里

的起点、终点不能互为替换，即，u 可达 v 不能保证 v 也可达 u。

有向图 2-1-8 就是连通的，但不是双向连通的，因为从任何顶点出发，都没有到达顶点 3 的有向路。不是双向连通的有向图可以分解成几个双向连通分支。图 2-1-8 共有 5 个双向连通分支，分别用不同的颜色标出。

- **赋权图**

一般的赋权图是指对图的每条边 e 赋予一个实数值 W(e)。如架设连接各城镇的交通路网，需考虑各段线路的修建费用；在运输网络中要考虑网络各段线路的容量，等等。

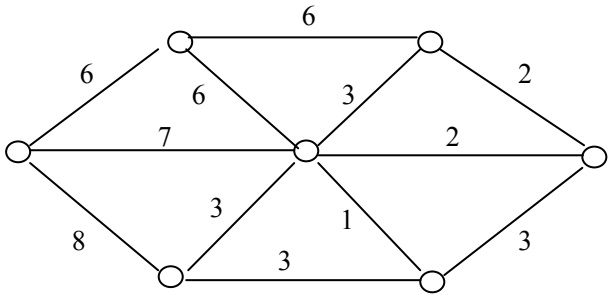
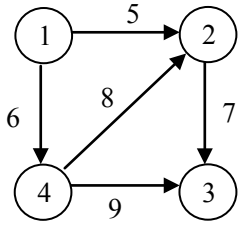


图 2-1-9　一个交通路网

注意，描述一个赋权图时，如果顶点v_i, v_j之间有一条边连接，而且权值为a，则其邻接矩阵中的(i, j)元素为a（而不再是 1），其余元素统一取为 0 或一个充分大的数，视问题而定。

- **图的邻接链表表示**

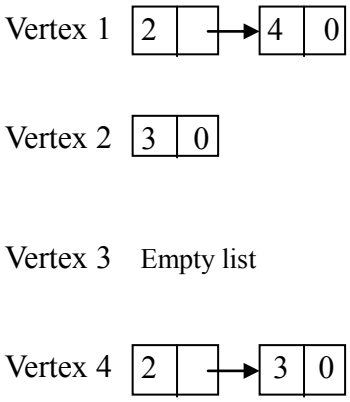
除了邻接矩阵表示和关联矩阵表示，在数据结构上也常常采用邻接链表的方法表示一个图。这是把邻接于每个顶点的点做成一个表连接到这个顶点上。图 2-1-10 给出了一个有向图(a)和它的邻接链表，其中，(b)是图 G 的邻接矩阵表示；(c)是各点的邻接表；(d)给出各点之间链接起来的结构。



(a)

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

(b)



(c)

		HEAD	NEXT
Vertices	1		5
	2		7
	3		0
	4		8
Edges	5	2	6
	6	4	0
	7	3	0
	8	2	9
	9	3	0

(d)

图 2-1-10 G 及其邻接链表

§2 图的遍历(搜索)算法

- **有根树与有向树**

树是一个没有圈的连通图。如果指定树的一个顶点为根，则这棵树称为有根树。在有根树中，邻接根的顶点称为根的儿子，而根称为这些儿子的父亲。对于不是根的顶点，除了它的父亲之外其它与之邻接的顶点都称为该顶点的儿子，该顶点也自然称为它的这些儿子的父亲。没有儿子的顶点称为叶顶点。

从根到每一个叶顶点都有一条唯一的路。这些路的最长者的长度称为该树的高度；不是根的顶点v不通过其父顶点而能到达的所有顶点同v一起生成一棵以v为根的子树，这棵子树的高度称为顶点v在原树中的高度。树的根到每个顶点v都有一条唯一的路，这条路的长度称为顶点v在树中的深度。如在图 2-2-1 的二叉树中，树的高度为 4，顶点 D,E 的深度都是 2；顶点 G 的高度为 1，深度为 3。可见，树中每个顶点的高度与深度的和不超过树的高度。

二叉树是这样一棵有根树，它的每个顶点至多有两个儿子。叶顶点深度都相同，而且除了叶顶点以外，每个顶点都恰有两个儿子的有根树称为完整的二叉树。高度为k的完整二叉树恰有$2^{k+1}-1$个顶点，它的所有顶点按照从上到下、从左到右的顺序可以编号为：$1,2,\cdots,2^{k+1}-1$。从这样编号的完整二叉树中的某一位置开始，删掉后面编号的所有顶点及与之关联的边所得到的二叉树称为一棵完全二叉树。

h 表示二叉树的高度

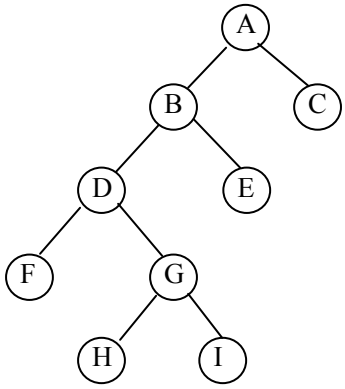
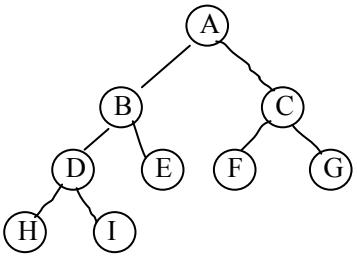


图 2-2-1 一棵二叉树



从一棵完整的二叉树中删掉标号为 2^h-i，$1\le i\le k$ 的 k 个顶点

图 2-2-2 一棵完全二叉树

从上到下、从左到右，将完整二叉树上的顶点编号为 $1,2,\ldots,2^{h+1}-1$

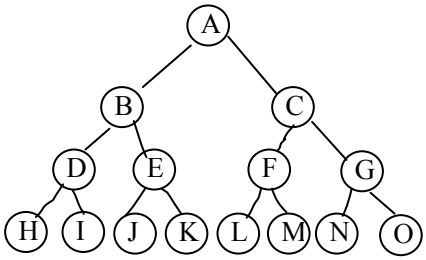


图 2-2-3 一个完整的二叉树

一个二叉树通常用两个数组 Lson 和 Rson 表示。假设二叉树的顶点标为 $1,2,\cdots,n$，则 Lson[i]=j 表示顶点 j 是顶点 i 的左儿子；同样，Rson[i]=j 表示顶点 j 是顶点 i 的右儿子。一个完全的二叉树可以用一个数组来表示：设 T 是一棵高度为 k 的完全二叉树，则数组的第一个元素是该树的根，第二个元素是根的左儿子，第三个元素是根的右儿子。一般地，第 i 个元素的左儿子是数组的第 $2i$ 个元素，第 i 个元素的右儿子是数组的第 $2i+1$ 个元素。反之，数组中第 $j(>1)$ 个元素的父亲是数组中的第 $\lfloor j/2 \rfloor$ 个元素。

一棵有向树是满足下述条件的无圈有向图：

1．有一个称为根的顶点，它不是任何有向边的终点；

2．除了根以外，其余每个顶点的入度均为 1；

3．从根到每个顶点有一条有向路。

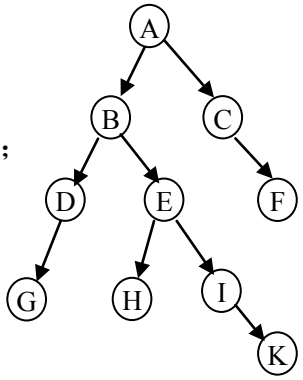


图 2-2-4 有向树

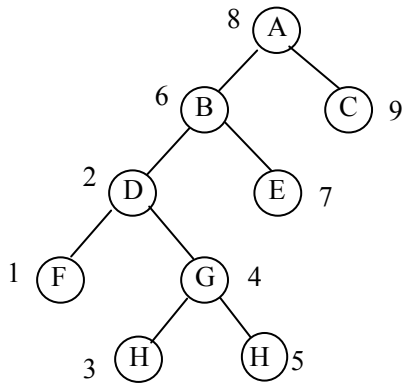
图 2-2-4 是一棵高度为 4 的有向树。对于有向树有类似于有根树的全部术语。

- **二叉树的搜索**

二叉树的搜索主要有先根次序搜索、中根次序搜索和后根次序搜索，各种遍历算法的伪代码写出如下，并在其后给出了搜索过程示意图。其中，Lson(T)表示以树 T 的左儿子为根的子树；Rson(T)表示以树 T 的右儿子为根的子树。

程序 2-2-1　二叉树的中根次序遍历算法伪代码

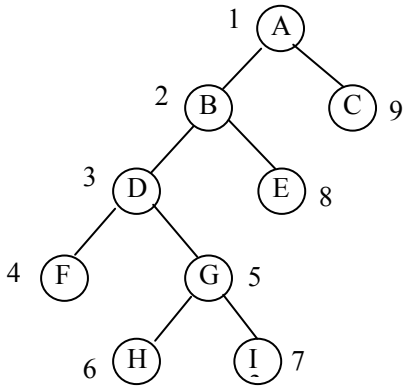
```
InOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
          //Lson,Data,Rson
  if T≠0 then
     InOrder(Lson(T));
     Visit(T);
     InOrder(Rson(T));
  end{if}
end{InOrder}
```



二叉树的中根次序遍历

程序 2-2-2　二叉树的先根次序遍历算法伪代码

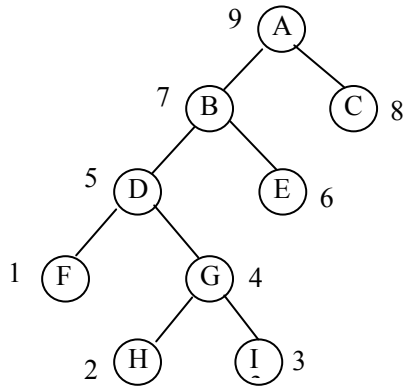
```
PreOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
           //Lson,Data,Rson
  if T≠0 then
     Visit(T);
     PreOrder(Lson(T));
     PreOrder(Rson(T));
  end{if}
end{PreOrder}
```



二叉树的先根次序遍历

程序 2-2-3　二叉树的后根次序遍历算法伪代码

```
PostOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
            //Lson,Data,Rson
     if T≠0 then
        PostOrder(Lson(T));
        PostOrder(Rson(T));
        Visit(T);
     end{if}
  end{PostOrder}
```



二叉树的后根次序遍历

- **一般树的遍历算法**

我们可以将二叉树的父与子之间的关系推广到一般树上，这样每个父亲的儿子可以有许多个，而且可以排出顺序。于是，关于二叉树的后两种遍历算法完全可以移置到一般的树上来，而中根优先次序遍历算法对于多根的情况不好确定根居于哪个位置，所以，不宜照搬。

树的先根次序遍历算法：

i. 若 T 为空，则返回；
ii. 访问 T 的根；
iii. 按树的先根次序遍历 T 的第一棵子树；
iv. 按树的先根次序依次遍历 T 的其余子树。

树的后根次序遍历算法：

v. 若 T 为空，则返回；
vi. 按树的后根次序遍历 T 的第一棵子树；
vii. 按树的后根次序依次遍历 T 的其余子树；
viii. 访问 T 的根。

- **一般图的遍历**

无论是二叉树还是一般的树，由于其不含有圈，所以属于同根的各个子树之间是相互独立的，遍历过程是对各个子树的分别遍历和对根遍历以及把这些遍历有机地组合起来。无论是什么顺序搜索，都不会出现重叠和死循环现象。一般的图没有这种独立性，所以上述方法不能施行。但是，上述方法的思想可以借鉴，于是产生了深度优先搜索方法和宽度优先搜索方法。

问题：*在一个给定的图 G=(V,E)中是否存在一条起于顶点 v 而终于顶点 u 的路径？遍历与某一起点 v 有路相通的所有顶点。*

➢ **宽度优先搜索算法（BFS）**

开始：起点 v 和一个空的待访队列 Q。

将 v 标记为已访问的顶点，然后开始检查其所有邻点，把其中未被访问的顶点依次放在待检查队列 Q 的尾部。用队列 Q 的首元素 u 替换 v（并从队列 Q 中去掉首元素 u），重复以上过程，直到队列 Q 空为止。

程序 2-2-4 由一点出发的宽度优先搜索算法伪代码

```
proc BFS(v) //宽度优先搜索 G，从顶点 v 开始执行，数组 visited 标示各
    //顶点被访问的序数；数组 s 将用来标示各顶点是否曾被放进待检查队
    //列，是则过标记为 1，否则标记为 0；计数器 count 计数到目前为止已
    //经被访问的顶点个数，其初始化为在 v 之前已经被访问的顶点个数
  1. AddQ（v，Q）；//首先访问 v,将 Q 初始化为只含有一个元素 v 的队列
  2. while Q 非空 do
  3.   u:=DelHead(Q); count:=count+1; visited[u]:=count;
  4.     for 邻接于 u 的所有顶点 w  do
  5.         if s[w]=0 then
```

```
6.            AddQ(w,Q); //将 w 放于队列 Q 之尾
7.            s[w]:=1;
8.          end{if}
9.       end{for}
10.    end{while}
11. end{BFS}
```

这里调用了两个函数：AddQ(w,Q)是将 w 放于队列 Q 之尾；DelHead(Q)是从队列 Q 取第一个元素，并将其从 Q 中删除。

定理 2.2.1 图 G 的宽度优先搜索算法能够访问 G 中由 v 可能到达的所有顶点。如果记 $t(\nu,\varepsilon)$和 $s(\nu,\varepsilon)$为算法 BFS 在任意一个具有ν个顶点和ε条边的连通图 G 上所花的最大时间和最大空间。则当 G 由邻接矩阵表示时有：

$$t(\nu,\varepsilon)=\Theta(\nu^2),\quad s(\nu,\varepsilon)=\Theta(\nu)$$

当 G 由邻接链表表示时；

$$t(\nu,\varepsilon)=\Theta(\nu+\varepsilon),\quad s(\nu,\varepsilon)=\Theta(\nu);$$

证明：除结点 v 外，只有当结点 w 满足 s[w]=0 时才被加到队列上，因此每个结点至多有一次被放到队列上。需要的队列空间至多是 $\nu-1$；visited 数组变量所需要的空间为 ν；其余变量所用的空间为$O(1)$，所以 $s(\nu,\varepsilon)=\Theta(\nu)$。

如果使用邻接链表，语句 4 的 for 循环要做 d(u)次，而语句 2～11 的 while 循环需要做 ν 次，因而整个循环做$\sum_{u\in V} d(u)=2\varepsilon$次$O(1)$操作，又 visited、s 和 count 的赋值都需要 ν 次操作，因而 $t(\nu,\varepsilon)=\Theta(\nu+\varepsilon)$。

如果采用邻接矩阵，则语句 2～11 的 while 循环总共需要做 ν^2次$O(1)$操作，visited、s 和 count 的赋值都需要 ν 次操作，因而 $t(\nu,\varepsilon)=\Theta(\nu^2)$。 证毕

由定理 2.2.1 可知，宽度优先搜索算法能够遍历图 G 的包含 v 的连通分支中的所有顶点。对于不连通的图，可以通过在每个连通分支中选出一个顶点作为起点，应用宽度搜索算法于每个连通分支，即可遍历该图的所有顶点。

程序 2-2-5 图的宽度优先遍历算法伪代码

```
proc BFT(G, ν) //count、s 同 BFS 中的说明，branch 是统计图 G 的
  //连通分支数
  count:=0; branch:=0;
  for i to n do
    s[i]:=0;  //将所有的顶点标记为未被访问
  end{for}
```

```
  for i to v do
    if s[i]=0 then
       BFS(i); branch:=branch+1;
    end{if}
  end{for}
end{BFT}
```

关于 BFT 算法的时间和空间复杂性与 BFS 同样估计（注意空间的差别）。

如果 G 是连通图，则 G 有生成树。注意到 BFS 算法中，由 4～8 行，将所有邻接于顶点 u 但未被访问的顶点 w 添加到待检测队列中。如果在添加 w 的同时将边(u,w)收集起来，那么算法结束时，所有这些边将形成图 G 的一棵生成树。称为图 G 的宽度优先生成树。为此，在 BFS 算法的第 1 行增加语句 T:={}，在第 7 行增加语句 T:=T∪{(u,w)}即可。

➢ **图的深度优先搜索**

深度优先搜索是沿着顶点的邻点一直搜索下去，直到当前被搜索的顶点不再有未被访问的邻点为止，此时，从当前被搜索的顶点原路返回到在它之前被访问的顶点，并以此顶点作为当前被搜索顶点。继续这样的过程，直至不能执行为止。

程序 2-2-6 图的深度优先搜索算法伪代码

```
proc DFS(v) //访问由 v 到达的所有顶点,计数器 count 已经初始化为 1;
    //数组 visited 标示各顶点被访问的序数，其元素已经初始化为 0。
1.  visited(v):=count;
2.  for 邻接于 v 的每个顶点 w  do
3.    if visited(w)=0 then
4.       count:=cout+1;
5.       DFS(w);
6.     end{if}
7.   end{for}
8. end{DFS}
```

对于连通图，深度优先算法也能产生一棵生成树，称为深度优先搜索树。读者可以在算法中添加语句使算法同时获得深度优先搜索树。

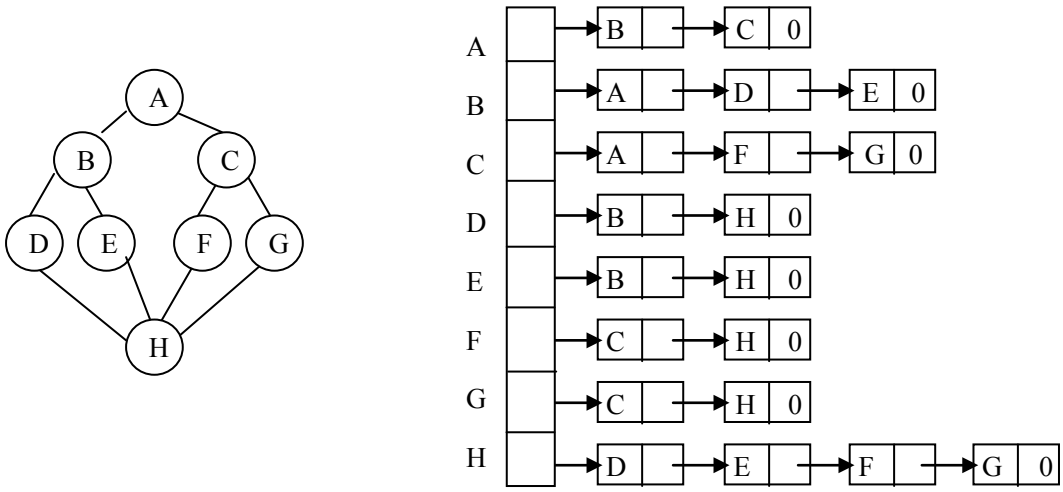


图 G 及其邻接链表

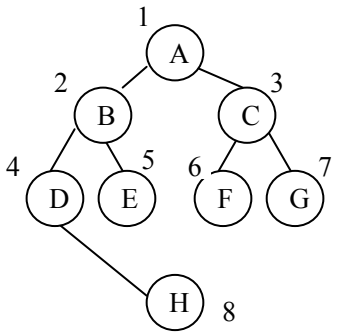


图 G 的宽度优先搜索树

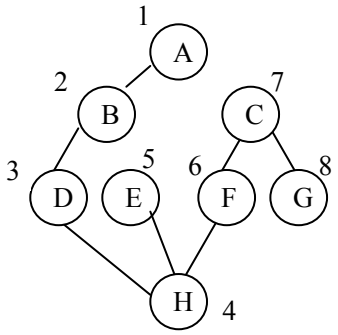


图 G 的深度优先搜索树

比较宽度优先和深度优先搜索算法，发现它们有很大不同。在 BFS 中，当搜索到某个顶点时,就要同时依次将该顶点的所有相邻顶点访问完;在 DFS 中，当搜索到某个顶点时，继续纵深搜索与该顶点相邻的其它一个未搜索过的顶点；前者用一个队列实现，而后者可以用一个栈来实现。当然，前者也可以用一个栈来实现，称为 D-搜索。然 D-搜索已经不同于 BFS 了。

§3 双连通与网络可靠性

通信网络的抽象模型可以是一个无向图:顶点代表通信站,边代表通信线路。图 2-3-1 和图 2-3-2 是两个通信网络示意图。直观可知，图 2-3-1 的通信网络可靠性较高,因为即使有一个网站出现问题,其它网站之间的通信仍能够继续进行。而图 2-3-2 所示的网络则不能，只要网站 F 发生故障，位于其左右两部分的网站之间就无法连通了。在图 2-3-2 中，象 F 这样的顶点称为割点。

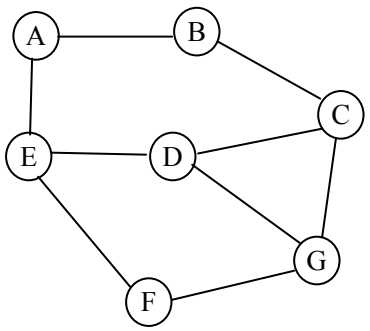


图 2-3-1

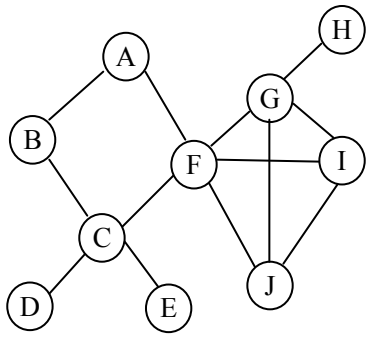


图 2-3-2

定义 连通无向图 G 中的顶点 v 称为割点，如果在 G 中去掉 v 及其关联的边，剩下的图就不再连通。

没有割点的连通图称为 2-连通的（也称为块）。图 G 中极大的 2-连通子图称为 G 的一个 2-连通分支。在图 2-3-2 中除了 F 以外，C 和 G 也都是割点。这个图有 5 个 2-连通分支（参见图 2-3-3）。图 2-3-1 是 2-连通的。

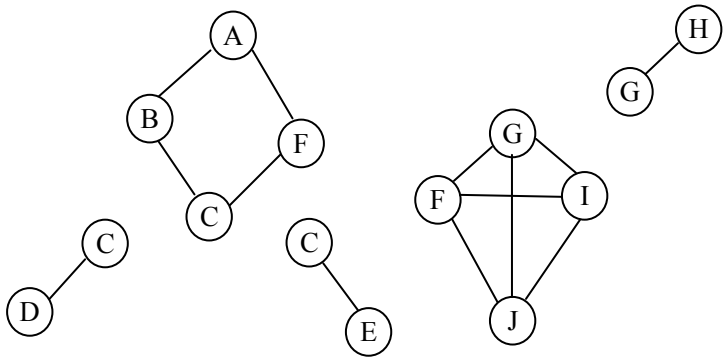


图 2-3-3 图 2-3-2 的 5 个 2-连通分支

就通信网络而言，当然希望没有割点。如果现有的网络有割点，则设法增加一些线路（当然希望尽量少的增加），使之成为 2-连通的。

添加边的算法：

E1： for 每个割点 u do

E2： 设 B_1，B_2，… ，B_k，是包含割点 u 的全部 2-连通分支

E3： 设 V_i 是 B_i 的一个顶点，且 $V_i \neq u$，$1 \leqslant i \leqslant k$。

E4： 将边(V_i, V_{i+1})添加到 G，$1 \leqslant i \leqslant k-1$。

E5：endfor

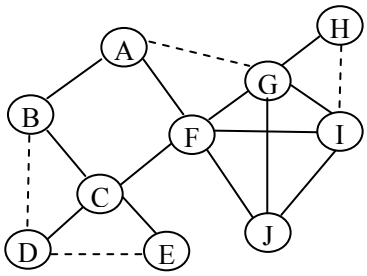


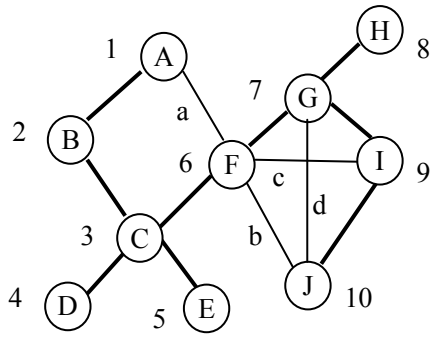
图 2-3-4 添加边使成为二连通图

问题：设计算法测试一个连通图是否 2-连通；若不是，算法将识别出割点。

解决方法：以深度优先遍历算法为基础，加以割点识别步骤。

当采用深度优先遍历算法时，顶点 v 被访问的序数称为 v 的深索数，记做 DFN(v).

假定图的邻接链表是按照顺时针方向的顺序构造的



a,b,c,d 是图 G 关于这棵生成树的余边

图 2-3-5 图 G 及其深度优先生成树

按照深度优先生成树将图中的顶点分层，使得上层是下层的祖先，而同层之间是兄弟关系：

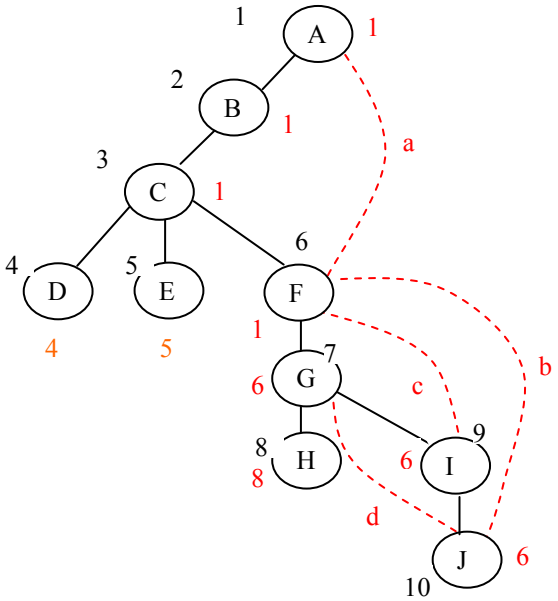


图 2-3-6 深度优先树的分层

1．关于深度优先生成树 T，图 G 的每一条边(u,v)的两个端点 u、v 之间，

或 u 是 v 的祖先，或 v 是 u 的祖先，即不是平辈关系。

2．树 T 的根是图 G 的割点当且仅当其在 T 中至少有两儿子；

3. 既不是根也不是叶的顶点 u 不是 G 的割点当且仅当 u 在 T 中的每个儿子 w 都至少有一个子孙（或 w 本身）关联着一条边 e，e 的另一个端点是 u 的某个祖先（e 一定是树 T 的余边）；

4．叶顶点不能是割点。

根据性质 3，4，深度优先生成树 T 的非根顶点 u 是 G 的割点当且仅当 u 至少有一个儿子 w，w 及其子孙都不与 u 的任何祖先相邻。

注意到 u 的深索数一定小于其子孙的深索数，所以深索数 DFN 并不能反映一个顶点是否是割点的情况。为此，我们递归地定义各个顶点 u 的最低深索数 L(u)：

定义 顶点 u 的最低深索数 L(u)定义为

$$L(u):=\min\{DFN(u),\ \min\{DFN(x)\mid(u,x)\text{是 T 的余边}\},\ \min\{L(w)\mid w\text{ 是 u 的儿子}\}\ \}$$

可见，如果 L(u)≠DFN(u)，则必定 L(u)< DFN(u)，因而 L(u)是 u 通过一条子孙路径(至多后跟一条 T 的余边)所可能到达的顶点的最低深索数。图 2-3-6 中，红色数字表示了各顶点的最低深索数。

结论：如果 u 不是深度优先生成树的根，则 u 是图 G 的割点当且仅当 u 有某个儿子 w，w 的最低深索数不小于 u 的深索数，即

$$L(w)\geqslant DFN(u),\ \text{for some son } w \text{ of } u$$

看来要想识别图 G 的所有割点，需先获得深度优先树 T，然后按后根优先次序遍历 T，计算出各个顶点的最低深索数。但是，从函数 L 的定义可知这两步工作可以同时完成。

程序 2-3-1 计算 DFN 和 L 的算法伪代码

```
proc DFNL(u,v) //一个深度优先搜索算法，u 是开始顶点。在深度优先
        //树中，若 u 有父亲，则 v 即是。数组 DFN 初始化为 0，变量
        //num 初始化为 1，n 是图 G 的顶点数。
        global  DFN[n]，L[n]，num，n
1.        DFN(u):=num; L(u):=num; num:=num+1;
2.        for 每个邻接于 u 的顶点 w  do
3.          if DFN(w)=0 then
4.              DFNL(w,u); //还未访问 w
5.              L(u):=min(L(u),L(w));
6.          else
7.            if w≠v then L(u):=min(L(u),DFN(w)); end{if}
```

```
8.          end{if}
9.        end{for}
10. end{DFNL}
```

为了得到图 G 的 2-连通分支，需对 DFNL 作一些修改。注意到，在第 4 行调用 DFNL 时，如果出现情况：

$$L(w) \geqslant DFN(u),$$

就可以断定 u 或是 T 的根，或是图 G 的割点。不论那种情况，都将边(u,w)以及对 DFNL 这次调用期间遇到的所有树边和余边加在一起（除了包含在子树 w 中其它 2-连通分支中的边以外），就构成图 G 的一个 2-连通分支。鉴于此，将 DFNL 做如下修改：

```
// 引进一个存放边的全程栈 S；
// 在 2 到 3 行之间加：
    2.1  if v≠w and DFN(w)<DFN(u) then
    2.2      将(u,w)加到 S 的顶部；
    2.3   end{if}
// 在 4 到 5 行之间加下列语句：
    4.1  if L(w)≥DFN(u) then
    4.2    print（’new biconnected component’）;
    4.3    loop
    4.4      从栈 S 的顶部删去一条边；
    4.5      设这条边是(x,y)
    4.6      print(“(”, x,“,”, y,“)”);
    4.7    until ((x,y)=(u,w) or(x,y)=(w,u));
    4.8  end{if}
```

如果 G 是有 n 个顶点 m 条边的连通图，且 G 用邻接链表表示，那么 DFN 的计算时间是 O(n+m).一旦算出 L[1:n],G 的割点就能在 O(n)时间识别出来，因此，识别全部割点总的时间不超过 O(n+m).

当 DFNL 增加了上述语句之后，其计算时间仍然是 O(n+m).

定理 2.3.1 设 G 是至少有两个顶点的连通图，则算法 DFNL 增加了语句 2.1-2.3 和 4.1-4.8 的算法能正确生成 G 的全部 2-连通分支。

证明：略。

§4 对 策 树

在一盘棋中，对弈各方都要根据当前的局势，分析和预见以后可能出现的局面，决定自己要采取的各种对策，以争取最好的结果。本节我们将用树的模型来描述对弈局势，并给出产生对策树的算法。先来分析拾火柴棍游戏。

在盘面上放 n 支火柴，由弈者 A 和 B 两人参加比赛。规则：两人轮流从盘上取走 1，2 或 3 支火柴，拿走盘中最后一支火柴者为负。

以盘中剩下的火柴数来表示该时刻的棋局。棋局序列 C_1，C_2，…，C_k 称为有效（棋局）序列，如果：

- C_1 是开始棋局；
- C_i 不是终止棋局，i＝1，2，…，k-1；
- 由 C_i 走到 C_{i+1} 按下述规则：若 i 是奇数，则 A 走一合法步骤；若 i 是偶数，则 B 走一合法步骤。

以 C_k 为终局的一个有效棋局序列 C_1，C_2，…，C_k 是此游戏的一盘战例。有限次博弈游戏的所有可能的实际战例可以用一棵树来表示（参见附页“对策树 1”）。

在圈 B 标志的地方表示 B 取胜；在方块 A 标志的地方表示 A 取胜。从图中可以看出，只要 A 第一步取 1 根火柴，则不论 B 如何应着，A 都有取胜的应着。否则，A 不保证取胜。那么怎样确定 A 的棋着呢？是否有一般的规律？对策树在决定采取什么对策，即确定弈者下一步应走哪步棋上是很有用的。事实上，走哪一步使自己获胜的机会最大，可以用一个估价函数 E(X)来评价，它是棋局 X 对于弈者价值大小的估计。设 E(X)是弈者 A 的估价函数，若棋局 X 能使 A 有较大的获胜机会，则 E(X)的值就高，若棋局 X 使得 A 有较大的失败可能，则 E(X)的值就低。

那些使得 A 获胜的终止棋局或不管 B 如何应着都保证 A 能获胜的棋局，则 E(X)取最大值。而对于保证 B 取胜的棋局，E(X)则取最小值

对于其对策树顶点较少的博弈游戏，例如 n=6 的拾火柴棍游戏，可以采用先对终局定义 E(X)，而后逐步确定各个棋局 X 的价值 V(X)的方法给出 A 在各步走棋参考。在 n＝6 的拾火柴棍游戏的对策树中，终止顶点是叶顶点，我们给出如下估值：

$$E(X)=\begin{cases}1 & \text{若X对于A是胜局}\\ -1 & \text{若X对于A是负局}\end{cases} \tag{2.4.1}$$

而对于其它顶点，需要给出对于 A 来说能够取胜的价值。例如，若已经知道顶点 b，c，d 的价值，则父顶点 a 的价值应当取它们的价值的最大值，当顶点 a 代表的棋局处应该 A 走棋时。因为从棋局 a 出发，A 下一着棋的走法应当导致其得胜可能性最大的下一步棋局。一般地，若 X 不是叶点，其有儿子 X_1，X_2，…，X_d，

则定义 X 的价值为

$$V(X)=\begin{cases}\max\limits_{1\le i\le d}\{V(X_i)\} & \text{若X是方形顶点}\\ \min\limits_{1\le i\le d}\{V(X_i)\} & \text{若X是圆形顶点}\end{cases} \qquad (2.4.2)$$

如此计算出对策树 1 各顶点的价值后，就很容易看出 A 要（想取胜）在其所处的各个棋局上应该采取的对策了。从图中可以看出 A 有三条必胜的路线。用(2.4.2)确定各个顶点价值的过程称为最大最小过程。

实际上，对策树 1 恰好列出了 n = 6 时，拾火柴棍游戏所有可能的棋局。从对策树根顶点出发到达叶顶点的每条路径恰是一个战例。但是对于较大规模的博弈，一般很难列出所有可能的棋局序列。比如，国际象棋，它的完整的对策的顶点数，据估计，将达到 10^{100}，即使用一台每秒能生成 10^{11} 个顶点，也需要 10^{80} 年以上的时间才能生成完整的对策树。所以，对于具有大规模对策树的博弈，不是采取考察其完整对策树的办法来确定弈者的对策。这种情况下，通常采用向前预测几步才决定走一步的策略，实际上是假想一组局面。这组局面也可以用部分对策树表示出来（参看文档“对策树 2”）。用估价函数估定这样的对策树（实际是子树）的叶顶点的值，然后根据公式(2.4.2)逐一确定其它顶点的价值。最后确定下一步该走的棋着。使用产生数级对策树确定下一步棋着的方法所导致的棋局的质量将取决于这两名弈者所采用的估价函数的功能和通过最大最小过程来确定当前棋局的价值 V(X)所使用算法的好坏。

因为 A、B 两弈者走棋总是交替进行的，在写递归算法时经常要区分 A、B，以确定是取最大值还是取最小值。为克服这个缺点，只要改变 B 者值的符号就可以。不妨假定弈者 A 是一台计算机， A 为了确定下一着棋，其应该有一个算法计算所有 V(X)。为产生一个递归程序，将公式(2.4.2)中的取最小部分也改成取最大，V(X)改为 V′(X)：

$$V'(X)=\begin{cases}\mathrm{e(X)} & \text{若X是所生成子树的叶顶点}\\ \max\limits_{1\le i\le d}\{-V'(X_i)\} & \text{若X不是所生成子树的叶顶点}\end{cases} \qquad (2.4.3)$$

其中，X_1，X_2,---，X_d 是 X 的所有儿子顶点。当 X 是叶顶点时，若 X 是 A 走棋的位置，则 e(X)=E(X);若是 B 走棋的位置，则 e(X)=-E(X)。通过对以 X 为根顶点、高为 l 的对策树的后根次序遍历，可以产生求取 V′(X)的递归算法。

程序 2-4-1 对策树的后根次序求值算法

```
VE(X, l)//通过至多向前看l着棋计算V′(X)，弈者A的估价
  //函数是e(X)。假定由任一不是终局的棋局X开始，此棋局的
  //合法棋着只允许将棋局X转换成棋局X1，X2，…，Xd.
    if X是终局或l=0  then  return(e(X))  end{if}
```

```
        ans:= -VE(X1, l-1); //遍历第一棵子树
        for  i  from  2  to  d  do
          ans:=max(ans,-VE(Xi, l-1));
        end{for}
    return(ans);
end{VE}
```

只要将偶数级顶点的价值改变一下符号，文档“对策树 2”中给出的各棋局的价值即是调用算法 VE 的计算结果，此时，$X=P_{11}$，$l=4$. 各棋局的值按如下次序依次确定：

P_{31}，P_{32}，P_{21}，P_{51}，P_{52}，P_{53}，P_{41}，P_{54}，P_{55}，P_{56}，P_{42}，P_{33}，…，P_{37}，P_{24}，P_{11}.

算法 VE 的目标是求取在棋局 P_{11} 情况下对于弈者 A 的 $V(P_{11})$ 值，以决定 A 下一步要采取的决策。但是，从这个例子的计算过程来看，为达此目标不必计算上述所有棋局的估价值。比如，当知道 $V(P_{41})=3$ 后，就知道 $V(P_{33})\geqslant 3$。因为 P_{33} 是求最大值点，所以，一旦知道 $V(P_{42})\leqslant 3$，就没有必要继续求以 P_{42} 为根子树其它顶点的估价值了(因为 P_{42} 是求最小值点，只要 P_{42} 有一个儿子结点的估价值不超过 3，就有 $V(P_{42})\leqslant 3$，所以，知道 $V(P_{42})\leqslant 3$ 可能是已经计算出 P_{42} 的某些儿子结点处的估价值)。如果给求最大值点 Y 赋一个值 $\alpha(Y)$，代表到目前为止所知道的该结点最大估值，那么，在求其儿子结点 X 的估价值时就有了参考，可以省掉不必要的计算。同样地，如果给求最小值点 Y 赋一个值 $\beta(Y)$，代表到目前为止所知道该结点最小估值，那么，在求其儿子结点 X 的估价值时就有了参考，可以省掉不必要的计算。比如，一旦判断出 $V(P_{36})\geqslant -1$，则不必计算以 P_{36} 的其它儿子结点为根的子树的顶点的价值。设 Y 是 X 的父亲结点，X 有儿子结点 X_1，X_2，…，X_d，上述减少计算的思路可以形成如下的规则：

- ✧ 若 Y 是取最大值的顶点，则一旦知道 $V(X_k)\leq\alpha(Y)$ 就不必再计算以 X_{k+1}，…，X_d 为根的子树的任何顶点的估价值；
- ✧ 若 Y 是取最小值的顶点，则一旦知道 $V(X_k)\geq\beta(Y)$ 就不必再计算以 X_{k+1}，…，X_d 为根的子树的任何顶点的估价值；

称之为 $\alpha-\beta$ 截断规则。采用 $\alpha-\beta$ 截断规则可以改进算法 VE。在算法 VE 中，各结点的估价值是用公式(2.4.3)计算的 V'，所以，每个结点都看作是求最大值点。于是，在定义一个结点的 B 值是该结点迄今最大可能值的情况下，$\alpha-\beta$ 截

断规则是：对于任一结点 X，设 B 是该结点父亲结点的 B 值，而且，D=-B，那么，如果 X 的估价值不小于 D 时，则可以停止生成 X 的其他儿子结点。

程序 2-4-2 使用$\alpha-\beta$截断规则的后根次序求值算法

```
VEB(X, l,D)//通过至多向前看l着棋，使用α-β截断规则和
    // 公式(2.4.3)计算 V′(X)，弈者 A 的估价函数是 e(X)。假定
    // 由任一不是终局的棋局 X 开始，此棋局的合法棋着只允许
    // 将棋局 X 转换成棋局 X1，X2，…，Xd.
    if X 是终局或l=0  then  return(e(X))  end{if}
    ans:=-VEB(X1，l-1，∞)；// V′(X)到目前可能的最大值
    for  i  from  2  to  d  do
      if  ans ≥ D then return（ans）end{if}//使用α-β截断规则
      ans:= max(ans,-VEB(Xi，l-1，-ans))；
    end{for}
  return(ans);
end{VEB}
```

算法 VEB 还可以进一步改进，即造成更大的截断。这只需要观察孙子结点的值达到多大才能影响到爷爷结点的值即可得到线索。仍采用公式(2.4.3)来计算 V′(X)，用$\mu(Y)$表示到迄今所知结点 Y 的最大的可能估价值，并设 X 是 Y 的儿子结点，X 的儿子结点 X_1，X_2，…，X_d ，要想结点 X 的值对 Y 的值产生影响，就必须要求 X 的每个儿子结点 X_i 满足：$V'(X_i)>\mu(Y)$。因为，若某个 $V'(X_i)\leqslant\mu(Y)$，

$$V'(Y)\geq\max\{\mu(Y),-V'(X\ )\},\text{而}V'(\mathrm{X})=\max_{1\leq j\leq d}\{-V'(X_j)\}\geq-V'(X_i)\geq-\mu(Y)$$

从而$-V'(X)\leqslant\mu(Y)$，所以，V′(Y)不受 V′(X)的影响。这样，我们可以给每个结点赋一个下界值 LB，只要判断出该结点的估价值不超过这个下界，就可以停止计算其父亲结点的估价值。这产生了下面的改进算法：

程序 2-4-3　$\alpha-\beta$截断算法

```
AB(X，l，LB，D) // LB 是 V′(X)的一个下界。//通过至多向前看l着棋，
    //使用α-β截断规则和公式(2.4.3) 计算 V′(X)，弈者 A 的估价
```

```
    //函数是 e(X)。假定由任一不是终局的棋局 X 开始，此棋局的合法
    //棋着只允许将棋局 X 转换成棋局 X1，X2，…，Xd.
    if X 是终局或l=0  then  return(e(X))  end{if}
    ans = LB;
    for  i  from  1  to  d  do
      if  ans ≥ D then return（ans）end{if} //使用α－β截断规则
      ans = max(ans,-AB(Xi，l-1，-D，-ans));
    end{for}
  return(ans);
end{AB}
```

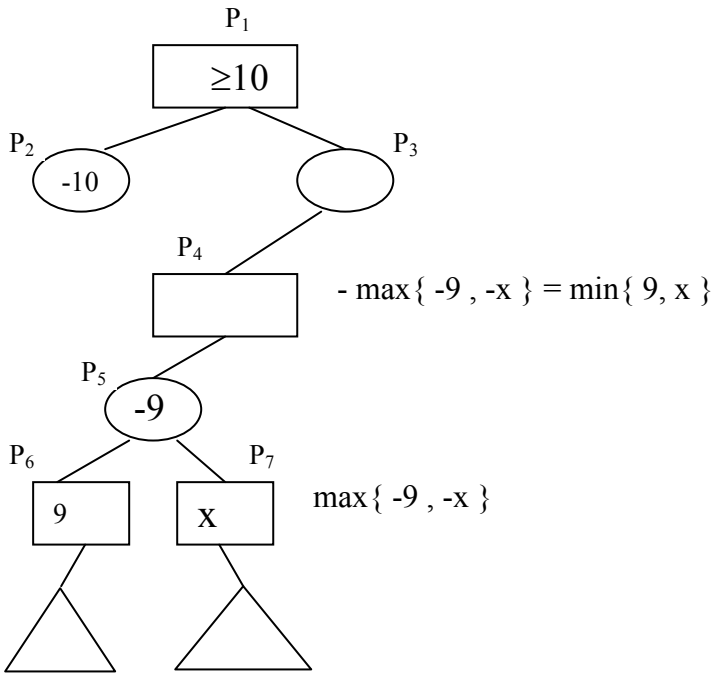


图 2-4-1 一棵假想的对策树

对图 2-4-1 这棵对策树调用算法 VEB(P_1, l, ∞)和调用算法 AB(P_1, l, -∞，∞)实际需要计算价值的结点数是不一样的。使用算法 VEB 需要计算 V′(P_7)， 但使用算法 AB 时不必。因为，当我们知道 V′(P_2)=-10 后， 就知道 10 是 V′(P_1)至少是 10，因而知道了 10 是 V′(P_4)的一个下界，进而知道 10 也是 V′(P_6)的下界。但是 V′(P_6)=9<10，所以计算 V′(P_5)的进程停止，即不需计算 V′(P_7).

习题 二

1.证明下列结论：

1）在一个无向图中，如果每个顶点的度大于等于 2，则该图一定含有圈；

2）在一个有向图 D 中，如果每个顶点的出度都大于等于 1，则该图一定含有一个有向圈。

2.设 D 是至少有三个顶点的连通有向图。如果 D 中包含有向的 Euler 环游（即是通过 D 中每条有向边恰好一次的闭迹），则 D 中每一顶点的出度和入度相等。反之，如果 D 中每一顶点的出度与入度都相等，则 D 一定包含有向的 Euler 环游。这两个结论是正确的吗？请说明理由。如果 G 是至少有三个顶点的无向图，则 G 包含 Euler 环游的条件是什么？

3. 设 G 是具有 n 个顶点和 m 条边的无向图，如果 G 是连通的，而且满足 m = n-1，证明 G 是树。

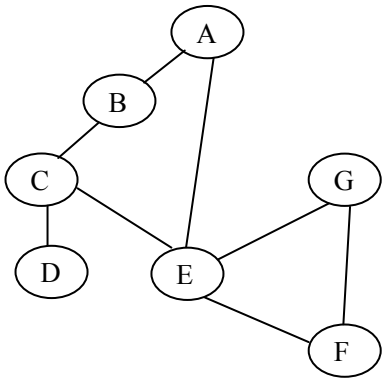
4. 假设用一个 n×n 的数组来描述一个有向图的 n×n 邻接矩阵，完成下面工作：

1） 编写一个函数以确定顶点的出度，函数的复杂性应为$\Theta(n)$；

2） 编写一个函数以确定图中边的数目，函数的复杂性应为$\Theta(n^2)$；

3） 编写一个函数删除边(i, j)，并确定代码的复杂性。

5.下面的无向图以邻接链表存储，而且在关于每个顶点的链表中与该顶点相邻的顶点是按照字母顺序排列的。试以此图为例描述讲义中算法 DFNL 的执行过程。



一个无向图 G

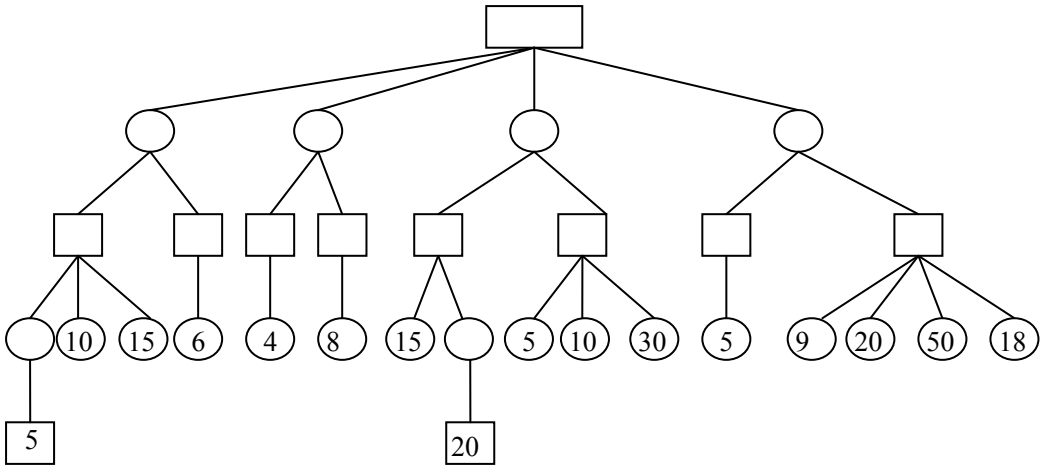
6.对图的另一种检索方法是 D-Search。该方法与 BFS 的不同之处在于将队列换成栈，即下一个要检测的结点是最新加到未检测结点表的那个结点。

1）写一个 D-Search 算法；

2）证明由结点 v 开始的 D-Search 能够访问 v 可到达的所有结点；

3）你的算法的时、空复杂度是什么？

7.考虑下面这棵假想的对策树：



1） 使用最大最小方法(2-4-2)式获取各结点的值；

2） 弈者 A 为获胜应该什么棋着？

3） 列出算法 VEB 计算这棵对策树结点的值时各结点被计算的顺序；

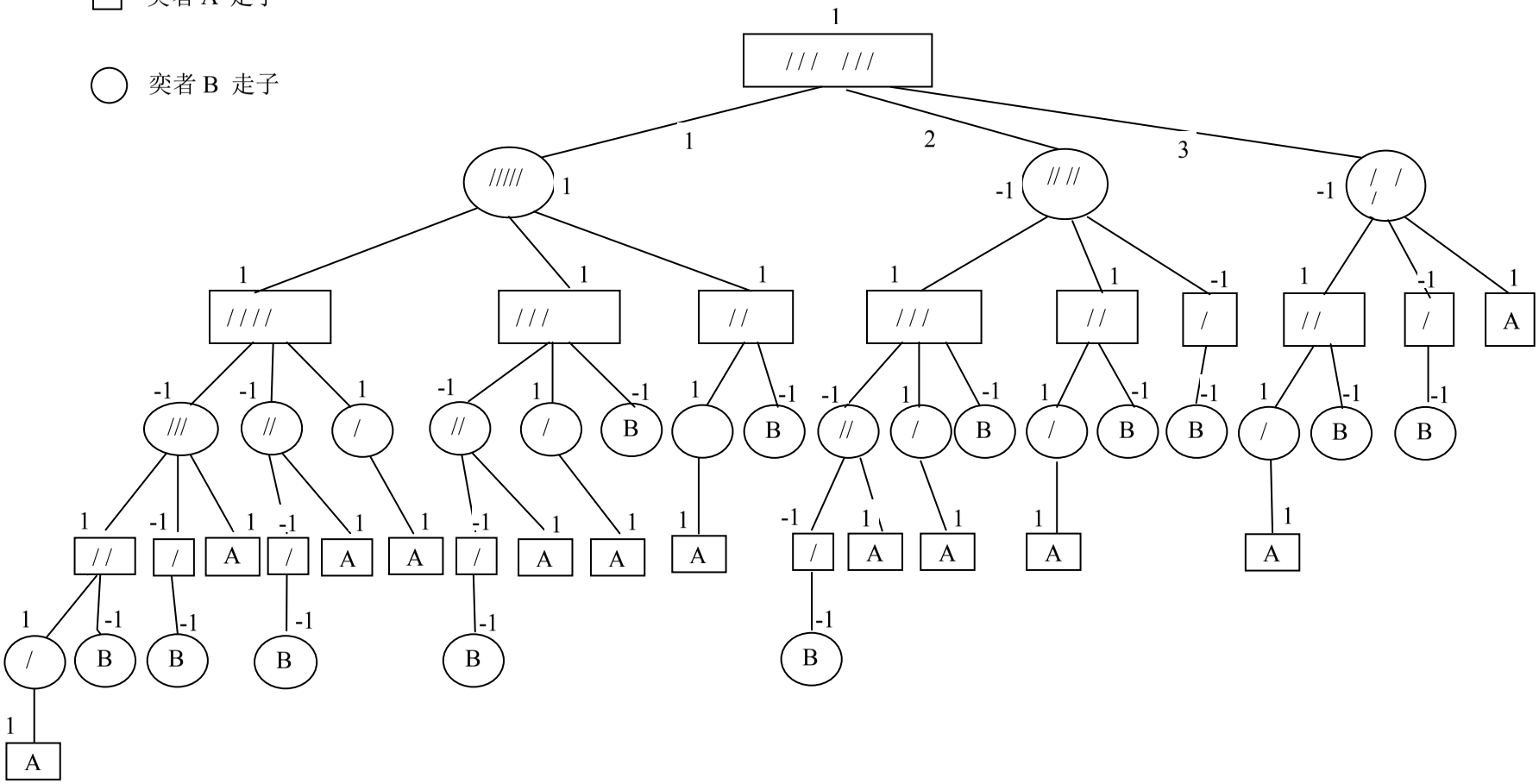
4） 对树中每个结点 X，用(2-4-3)式计算 V(X)；

5） 在取 X＝根，$l=10, LB=-\infty, D=\infty$ 的情况下，用算法 AB 计算此树的根的值期间，这棵树的哪些结点没有计算？

对策树 1　　n=6 时的拾火柴游戏状态树图

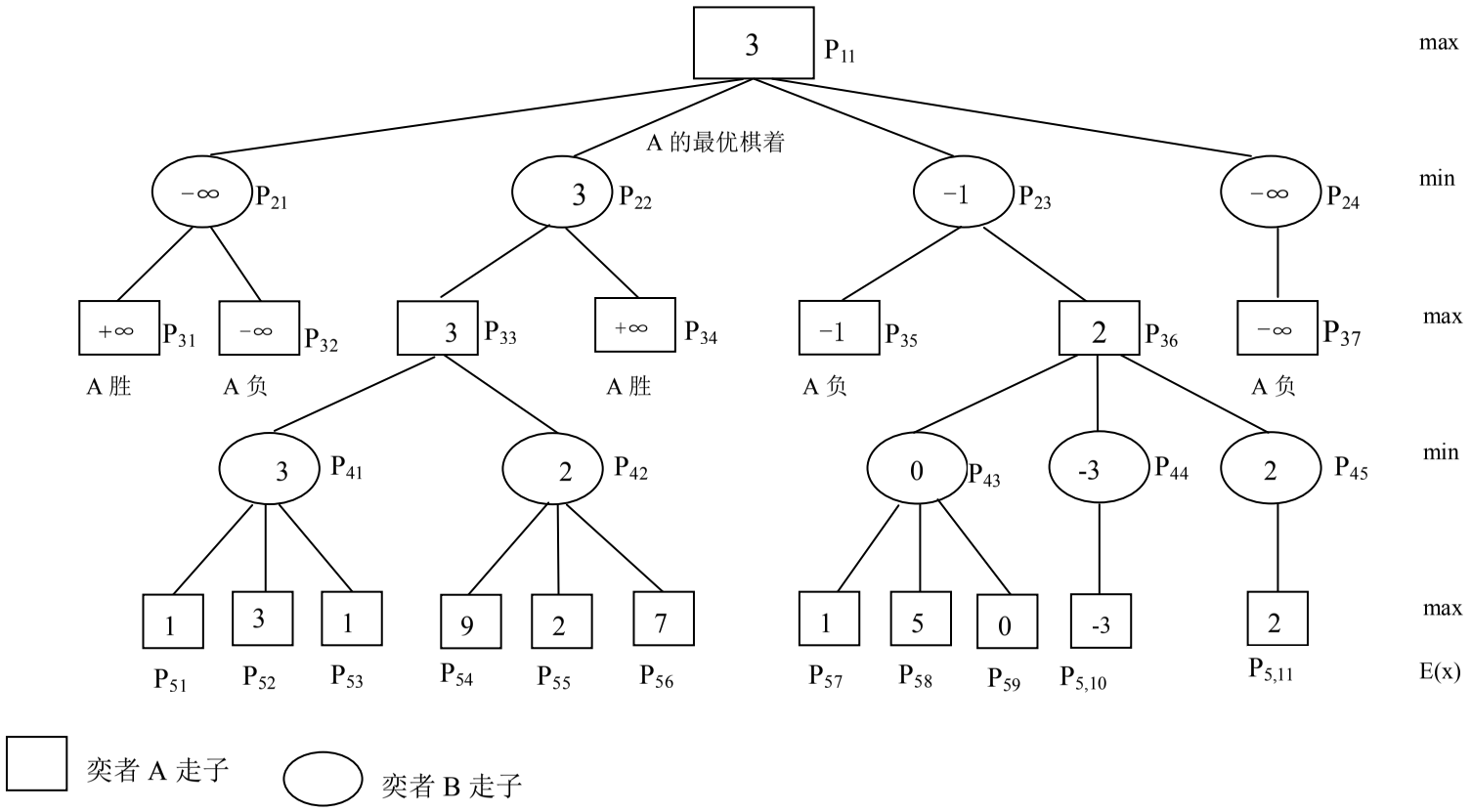
□ 奕者 A 走子

○ 奕者 B 走子



对策树 2

一盘假想博弈游戏的部分对策树



第三章　分 治 算 法

§1. 算法基本思想

先来分析折半搜索算法

程序 3-1-1 折半搜索

```
proc BiFind(a,n)
  //在数组 a[1..n]中搜索 x，数组中的元素满足 a[1] ≤ a[2] ≤ … ≤ a[n]。
  //如果找到 x，则返回所在位置（数组元素的下标），否则返回 －1
  global a[1..n], n;
  integer left,right,middle;
  left:=1; right:=n;
  while left ≤ right do
    middle:=(left+right)/2;
    if x=a[middle] then return(middle); end{if}
    if x>a[middle] then left:=middle+1;
    else right:=middle-1;
    end{if}
  end{while}
  return(－1); //未找到 x
end{BiFind}
```

while 的每次循环（最后一次除外）都将以减半的比例缩小搜索范围，所以，该循环在最坏的情况下需要执行 $\Theta(\log n)$ 次。由于每次循环需耗时 $\Theta(1)$，因此在最坏情况下，总的时间复杂度为 $\Theta(\log n)$。

折半搜索算法贯彻一个思想，即分治法。当人们要解决一个输入规模，比如 n，很大的问题时，往往会想到将该问题分解。比如将这 n 个输入分成 k 个不同的子集。如果能得到 k 个不同的可独立求解的子问题，而且在求出这些子问题的解之后，还可以找到适当的方法把它们的解合并成整个问题的解，那么复杂的难以解决的问题就可以得到解决。这种将整个问题分解成若干个小问题来处理的方法称为分治法。一般来说，被分解出来的子问题应与原问题具有相同的类型，这样便于算法实现（多数情况下采用递归算法）。如果得到的子问题相对来说还较大，则再用分治法，直到产生出不用再分解就可求解的子问题为止。人们考虑

和使用较多的是 k=2 的情形，即将整个问题二分。以下用 A[1..n]来表示 n 个输入，用 DiCo(p,q)表示用分治法处理输入为 A[p..q]的问题。

分治法控制流程

```
proc DiCo(p,q)
  global n，A[1..n];
  integer m,p,q;  // 1≤p≤q≤n
  if Small(p,q) then return(Sol(p,q));
  else m:=Divide(p,q); // p≤m<q
     return(Combine(DiCo(p,m),DiCo(m+1,q)));
  end{if}
end{DiCo}
```

这里，Small(p,q)是一个布尔值函数，用以判断输入规模为 q-p+1 的问题是否小到无需进一步细分即可容易求解。若是，则调用能直接计算此规模下子问题解的函数 Sol(p,q). 而 Divide(p,q)是分割函数，决定分割点；Combine(x,y)是解的合成函数。如果假定所分成的两个问题的输入规模大致相等，则 DiCo 总的计算时间可用下面的递归关系来估计：

$$T(n)=\begin{cases} g(n), & \text{当输入规模n比较小时直接求解}\mathrm{Sol}(n)\text{的用时} \\ 2T(n/2)+f(n), & f(n)\text{是}Combine\text{用时} \end{cases} \qquad (3.1.1)$$

例 3.1.1 求 n 元数组中的最大和最小元素

最容易想到的算法是直接比较算法：将数组的第 1 个元素分别赋给两个临时变量：fmax:=A[1]; fmin:=A[1]; 然后从数组的第 2 个元素 A[2]开始直到第 n 个元素逐个与 fmax 和 fmin 比较，在每次比较中，如果 A[i] > fmax，则用 A[i]的值替换 fmax 的值；如果 A[i] < fmin，则用 A[i]的值替换 fmin 的值；否则保持 fmax（fmin）的值不变。这样在程序结束时的 fmax、fmin 的值就分别是数组的最大值和最小值。这个算法在最好、最坏情况下，元素的比较次数都是 2(n-1)，而平均比较次数也为 2(n-1)。 如果将上面的比较过程修改为：

从数组的第 2 个元素 A[2]开始直到第 n 个元素，每个 A[i]都是首先与 fmax 比较，如果 A[i]>fmax,则用 A[i]的值替换 fmax 的值；否则才将 A[i]与 fmin 比较，如果 A[i] < fmin，则用 A[i]的值替换 fmin 的值。

这样的算法在最好、最坏情况下使用的比较次数分别是 n-1 和 2(n-1),而平均比较次数是 3(n-1)/2,因为在比较过程中，将有一半的几率出现 A[i]>fmax 情况。

如果采用分治的思想，可以构造算法，其时间复杂度在最坏情况下和平均用时均为 3n/2-2:

程序 3-1-2 **递归求最大最小值算法伪代码**

```
proc MaxMin（i,j,fmax,fmin）//A[1:n]是 n 个元素的数组，参数 i，j
  //是整数，1≤i≤j≤n，使用该过程将数组 A[i..j]中的最大最小元
  //分别赋给 fmax 和 fmin。
  global n, A[1..n];
  integer i, j;
  if i=j then
    fmax:=A[i]; fmin:=A[i]; //子数组 A[i..j]中只有一个元素
  elif i=j-1 then //子数组 A[i..j]中只有两个元素
    if A[i]<A[j] then
      fmin:=A[i]; fmax:=A[j];
    else fmin:=A[j]; fmax:=A[i];
    end{if}
  else
    mid:=⌊(i+j)/2⌋; //子数组 A[i..j]中的元素多于两个
    MaxMin(i, mid, lmax, lmin);
    MaxMin(mid+1, j, rmax, rmin);
    fmax:=max(lmax, rmax);
    fmin:=main(lmin, rmin);
  end{if}
end{Maxmin}
```

如果用 T(n)来表示 MaxMin 所用的元素比较次数，则上述递归算法导出一个递归关系式：

$$T(n)=\begin{cases}0 & n=1\\ 1 & n=2\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+2 & n>2\end{cases} \qquad (3.1.2)$$

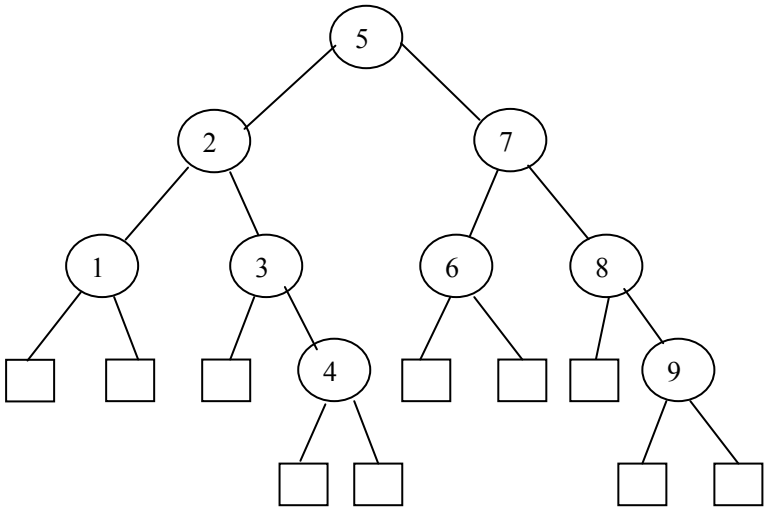
当 n 是 2 的方幂时，设 $n=2^k$，有

$$
\begin{aligned}
T(n) &= 2T(n/2)+2 \\
&= 2(2T(n/4)+2)+2 \\
&= 4T(n/4)+4+2 \\
&\cdots \\
&= 2^{k-1}T(2)+\sum_{1\le i\le k-1} 2^i \\
&= 2^{k-1}+2^k-2 \\
&= 3n/2-2
\end{aligned}
$$

无论是最好、最坏、还是平均情况，MaxMin 所用的比较次数都是 3n/2-2，比前面提到的算法(在最坏情况下)节省了 25%的时间。实际上，任何一种以元素比较为基础的找最大最小元素的算法，其元素比较次数的下界是$\lceil 3n/2 \rceil - 2$。从这种意义上来说，MaxMin 算法是最优的。然而，由于需要$\lfloor \log n \rfloor + 1$级的递归，每次递归调用需要将 i，j，fmax，fmin 和返回地址的值保留到栈中，需要多占用内存空间。而且由于这些值出入栈时也会带来时间开销，特别当 A 中元素的比较次数和整数 i 与 j 的比较次数相差无几时，递归求最大最小值算法未必比直接求最大最小值算法效率高。

例 3.1.2 搜索算法的时间下界

分析上节提到的折半搜索算法，我们已经知道其时间复杂度是$O(\log n)$。事实上，我们可以用一个二元比较树来分析折半搜索算法的时间复杂性。以下是 n=9 的二元比较树：



N=9 情况下，折半搜索的二元比较树

由图可见，当 x 在数组 A 中时，算法在圆形结点结束；不在 A 中时，算法在方形结点结束。因为 $2^3 \le 9 < 2^4$，所以比较树的叶结点的深度都是 3 或 4。因而元素比较的最多次数为 4。一般地有：

当 $n \in \left[2^{k-1}, 2^k\right)$ 时，成功的折半搜索至多做 k 次比较，而不成功的折半搜索或者做 k-1 次比较，或者做 k 次比较。

现在假设数组 A[1..n]满足：A[1]< A[2]< …< A[n]。要搜索元素 x 是否在 A 中。*如果只允许进行元素间的比较而不允许对它们进行其它的运算，则所设计的算法称为以比较为基础的算法。*

任何以比较为基础的搜索算法的执行过程都可以用一棵二元比较树来描述。每次可能的比较用一个内结点表示，对应于不成功的结果有一个外结点（叶结点）与之对应。线性搜索和折半搜索的二元比较树如下：

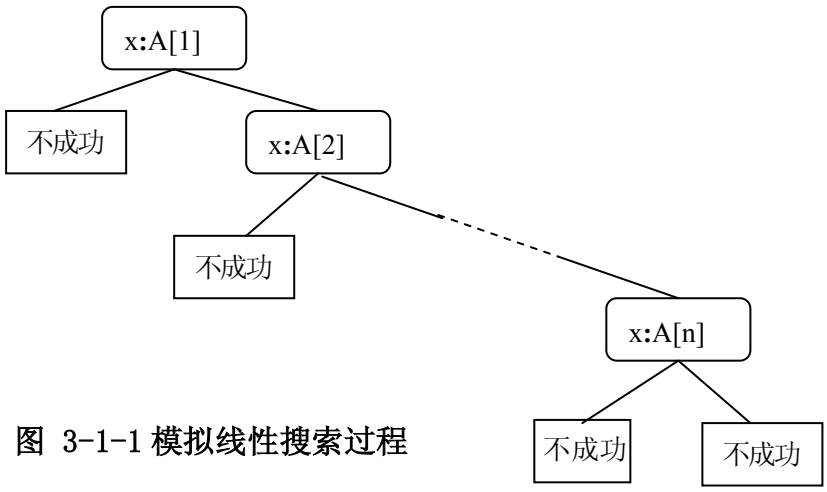


图 3-1-1 模拟线性搜索过程

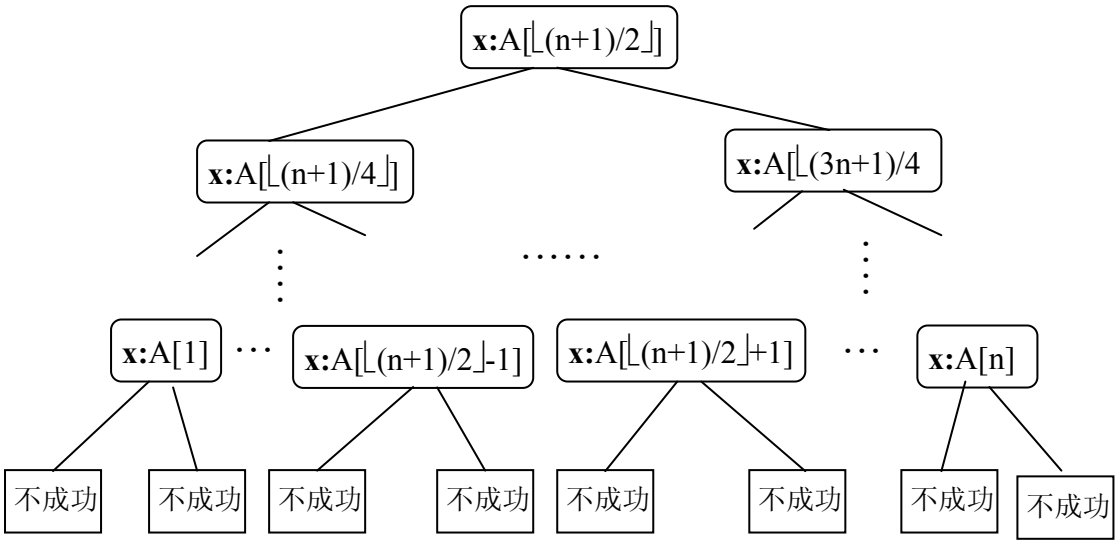


图 3-1-2 模拟折半搜索过程

定理 3.1.1 设数组 A[1..n]的元素满足 A[1]< A[2]< …< A[n]。则以比较为基础，判断 $x \in A[1..n]$ 的任何算法，在最坏情况下所需的最少比较次数 F(n)不小于⌈log(n+1)⌉。

证明 通过考察模拟求解搜索问题的各种可能算法的比较树可知，F(n)不大于树中由根到叶子的最长路径的距离，即树的高度。也就是说，在最坏情况下每种搜索算法的比较次数都是比较树的高度 k。对于每个二叉比较树，必有 n 个内结点与 x 在 A 中的 n 种可能的情况相对应。而每个内结点的深度都不会超过该树的高度减 1，即 k-1。因而，内结点的个数不超过 2^k-1，即 $n \le 2^k-1$。由此得 F(n)= k ≥⌈log(n+1)⌉。 证毕

定理 3.1.1 说明，任何一种以比较为基础的搜索算法，其最坏情况下所用时间不可能低于 $\Theta(\log n)$。不可能存在其最坏情况下时间比折半搜索数量级（阶）还低的算法。事实上，折半搜索所产生的比较树的所有叶结点都在相邻的两个层次上，而且这样的二叉树使得比较树的高度最低。因此，折半搜索是解决搜索问题在最坏情况下的最优算法。

§2. 排序算法

问题：已知 n 个元素的数组 A[1..n]，将 A 中元素按不降顺序排列。

- **归并排序算法**

先来分析插入排序算法

程序 3-2-1 向有序数组插入元素

```
proc Insert(a, n, x)
  //向数组 a[1..n]中插入元素 x
  //假定 a 的大小超过 n
  int i;
  for i from n by -1 to 1 do
    if x<a[i] then
      a[i+1]:=a[i];
  end{if}
  end{for}
  a[i+1]:=x;
end{Insert}
```

程序 3-2-2 插入排序

```
proc InSort(a, n)
    //对 a[1..n]进行排序
    for i from 2 to n do
        t:=a[i];
        Insert(a, i-1, t);
    end{for}
end{InSort}
```

将上述两个算法合并在一起，得到下述插入排序算法

程序 3-2-3 插入排序算法

```
proc InSort(a, n)
  for i from 2 to n do
    t:=a[i];
    integer j;
    for j from i-1 by -1 to 1 do
      if t<a[j] then a[j+1]:=a[j]; end{if}
    end{for}
    a[j+1]:=t;
  end{for}
end{InSort}
```

内层的 for 循环语句可能执行 i 次（$i=1,2,\cdots,n-1$），因此最坏情况下的时间是

$$\sum_{1\le i\le n-1} i = n(n-1)/2 = \Theta(n^2).$$

在这个算法中，大部分的时间都用在挪动元素上，随着已经排好顺序的数组的增长，被挪动的元素的个数也在增加，而且在整个过程中，很多元素不止一次被挪动。以下程序从某种程度上减少了这种浪费。这一算法的基本思想是采用分治的方法，将要排序的数组分成两部分，先对每部分进行排序，然后再将两部分已经排好序的子数组的元素按照从小到大的顺序交替地摆放在一个新的数组中。这一过程也许需要多次分解和组合，因而是一个递归过程。

程序 3-2-4 归并排序主程序伪代码

```
proc MergeSort(low, high) // A[low .. high]是一个全程数组，含有
   // high-low+1 个待排序的元素。
   integer low, high;
   if low < high  then
      mid:= ⌊(low+high)/2⌋  //求当前数组的分割点
      MergeSort(low, mid)  //将第一子数组排序
      MergeSort(mid+1, high)  //将第二子数组排序
      Merge(low, mid, high)  //归并两个已经排序的子数组
   end{if}
 end{MergeSort}
```

这里我们使用了辅助程序 Merge:

程序 3-2-5 合并过程伪代码

```
proc Merge(low, mid, high) //已知全程数组 A[low .. high], 其由
    //两部分已经排好序的子数组构成：A[low .. mid]和 A[mid+1 .. high]。
    //本程序的任务是将这两部分子数组合并成一个整体排好序的数组，
    //再存于数组 A[low .. high].
    integer h, i, j, k, low, mid, high;
    global A[low .. high];
    local B[low .. high]; //借用临时数组 B
    h:=low, i:=low, j:=mid+1;
    // h, j 是拣取游标, i 是向 B 存放元素的游标
    while h≤mid and j≤high do  //当两个集合都没有取尽时
       if A[h]≤A[j] then B[i]:=A[h], h:=h+1;
       else B[i]:=A[j], j:=j+1;
       end{if}
       i:＝i+1;
    end{while}
    if h>mid then
    //当第一子组元素被取尽，而第二组元素未被取尽时
      for k from j to high do
         B[i]:=A[k]; i:=i+1;
      end{for}
    else
    //当第二子组元素被取尽，而第一组元素未被取尽时
      for k from h to mid do
        B[i]:=A[k]; i:=i+1;
      end{for}
    end{if}
    //将临时数组 B 中元素再赋给数组 A
    for k from low to high do
      A[k]:=B[k];
    end{for}
  end{Merge}
```

可见，归并排序由分解与合并两部分组成，整个过程可用两棵树表示出来（参见本章附页“归并排序树”）。如果用 T(n)表示归并排序所用的时间，并假定合并过程所用时间与 n 成正比：cn，其中 c 是一个正数，则有

$$T(n)=\begin{cases} a & n=1 \\ 2T(n/2)+cn & n>1 \end{cases} \qquad (3.2.1)$$

其中，a 是一个常数。若 n 是 2 的方幂：$n=2^k$，直接推导可得

$$\begin{aligned} T(n) &= 2(2T(n/4)+cn/2)+cn \\ &= 4T(n/4)+2cn \\ &\cdots\cdots \\ &= 2^k T(1)+kcn \\ &= an+cn\log n \end{aligned}$$

对于一般的整数 n，我们可以假定 $2^k<n\le 2^{k+1}$，于是由 $T(n)\le T(2^{k+1})$，得

$$T(n)=O(n\log n)$$

- **以比较为基础的排序时间的下界**

类似于估计以比较为基础的搜索算法的时间下界，也可以用树来模拟以比较为基础的排序算法，在此我们考虑最坏情况下的时间下限，并假定数组中的元素互不相同。在树的内部结点上，算法执行一次比较，并根据比较的结果移向它的某一个子结点。由于每两个元素 A[i]和 A[j]的比较只有两种可能：A[i]<A[j]或 A[j]<A[i]，所以这颗树是二叉树。当 A[i]<A[j]时进入左分支，当 A[j]<A[i]进入右分支。各个叶结点表示算法终止。从根到叶结点的每一条路径与一种唯一的排列相对应。由于 n 个不同元素的不同排列共有 n！个，因此比较树有 n！个外部结点（参看本章附页“排序比较树”）。直接观察可知，由根到外结点路径即描述了该外结点所代表的排列生成过程，路径的长度即是经历的比较次数。因此，比较树中最长路径的长度（其是比较树的高）即是算法在最坏情况下所做的比较次数。要求出所有以比较为基础的排序算法在最坏情况下的时间下界，只需求出这些算法所对应的比较树的最小高度。如果比较树的高是 k，则该二叉树的外结点至多是 2^k 个。于是，$n!\le 2^k$。注意到

$$n!\ge n(n-1)\cdots(\lceil n/2\rceil)\ge (n/2)^{n/2-1} \qquad (3.2.2)$$

因而

$$k\ge (n/2-1)\log(n/2)=\Theta(n\log n) \qquad (3.2.3)$$

$T(n)\ge\Theta(n\log n)$，即 $\Theta(n\log n)$ 是以比较为基础的排序算法在最坏情况下的时间

下界。

从上式看出，归并排序是时间复杂度最低的排序算法（以比较为基础）。然而，仔细观察可以发现，归并排序有两个地方值得商榷：一是分解直到只有一个元素。事实上，当元素比较少时，直接进行排序，比如用插入排序算法，比起进一步分拆、合并手续要快得多。因为，在这种情况下，大量的时间都花在调用分解、合并函数上。所以，在归并排序算法中，对于*归并起点的规模* 应该有适当的限制，即加 Small(p,q)判断。二是辅助数组 B 的借用，虽然不可避免，但应该采用另一种方式，以避免数组 A 中的元素的频繁换位。为此，我们可以采用链表（该表中存储的是数组元素的下标），将数组 A 中的元素位置变动转化成链表值的变化。例如

LINK:

↓k=0　　↓k =2

位置	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
数据		50	10	25	30	15	70	35	55
指针	2	0	3	4	1	7	0	8	6

假定前 4 个已经排好，后 4 个也已经排好，链表如上，头指针分别是 2,5。下面是在此基础上将两个排好的子链表连接起来的过程。

q=2; r=5

```
A[2]<A[5]:  LINK[0]:=2; k:=i(=2);  i:=LINK[i](=3);
A[3]>A[5]:  LINK[k]:=5; k:=j(=5);  j:=LINK[j](=7);
A[3]<A[7]:  LINK[k]:=3; k:=i(=3);  i:=LINK[i](=4);
A[4]<A[7]:  LINK[k]:=4; k:=i(=4);  i:=LINK[i](=1);
A[1]>A[7]:  LINK[k]:=7; k:=j(=7);  j:=LINK[7](=8);
A[1]<A[8]:  LINK[k]:=8; k:=i(=1);  i:=LINK[i](=0);
                        k:=j(=8);  j:=LINK[j](=6)
                        k:=j(=6);  j:=LINK[j](=0)
```

Q = (10,　25,　30,　50)
　　↑i=2　↑i=4

R = (15,　35,　55,　70)
　　↑j=5

k: →2→5→3→4→7→1→8→6

```
if A[i]≤A[j] then
  LINK[k]:=i; k:=i; i:=LINK[i];
else
  LINK[k]:=j; k:=j; j:=LINK[j];
end{if}
```

指针移动过程　　**一般规则**

程序 3-2-6 使用链接的归并排序算法

```
proc MergeSortL(low, high, p) // Link 是全程数组 A[low..high]
   //的下标表, p 指示这个表的开始处。利用 Link 将 A 按非降顺序排列。
  global A[low..high]; Link[low..high];
  if high-low+1<16 then //设定子问题的最小规模 Small
    InSort(A,Link, low,high,p);
  else mid:=⌊(low+high)/2⌋;
    MergeSortL(low,mid,q); //返回 q 表
    MergeSortL(mid+1,high,r); //返回 r 表
    MergeL(q,r,p); 将表 q 和 r 合并成表 p
  end{if}
end{MergeSortL}
```

其中，合并程序 MergeL 是合并函数 Merge 的改进：

程序 3-2-7 使用连接表的合并程序

```
proc MergeL(q,r,p) // 由链接表 q 和 r 构造新的连接表。p、q、r 是
   //全程数组 Link[0..n]中两个表指针，这两个链表指出被划分的
   //两个子组的地址排序，而 p 指针指出两组归并后的地址排序。
   global n,A[1..n], Link[0..n];
   local integer i, j, k;
   i:=q; j:=r; k:=0; // 初始化，新表在 Link[0]处开始
   while i≠0 and j≠0 do //当两个表皆非空时
     if A[i]≤A[j] then
       Link[k]:=i; k:=i; i:=Link[i]; //加一个新元素到此表
     else Link[k]:=j; k:=j; j:=Link[j];
     end{if}
   end{while}
   if i=0 then
     Link[k]:=j;
   else Link[k]:=i;
   end{if}
   p:=Link[0];
end{MergeL}
```

例 3.2.1 考虑将数组 A=[50,10,25,30,15,70,35,55]按非降次序排列问题，采用改进的归并算法。这里主要说明链接表在合并函数 MergeL 被调用时的变化过程（参看本章附页“归并链接表”）。

● 快速排序算法

另一个利用分治法排序的例子是*快速排序*，是由计算机科学家 C.A.R.Hoare 提出的。基本策略是：将数组 A[1..n]分解成两个子数组 B[1..p]和 B[p+1..n]，使得 B[1..p]中的元素均不大于 B[p+1..n]中的元素，然后分别对这里的两个数组中的元素进行排序（非降的），最后再把两个排好序的数组接起来即可。一般的分解是从 A 中选定一个元素，然后将 A 中的所有元素同这个元素比较，小于或等于这个元素的放在一个子组里，大于这个元素的放在另一个子组里。这个过程叫做划分。

程序 3-2-8 划分程序伪代码

```
proc Partition(m,p) // 被划分的数组是 A[m,p-1]，
    //选定做划分元素的是 v:=A[m]。
    integer m, p, i;
    global A[m ..p-1];
    v:=A[m]; i:=m;
    loop
      loop i:=i+1; until A[i]>v; end{loop} //自左向右查
      loop p:=p-1; until A[p]≤v; end{loop} //自右向左查
      if i<p then
        Swap(A[i],A[p]); //交换 A[i]和 A[p]的位置
      else go to *;
      end{if}
    end{loop}
  *: A[m]:=A[p]; A[p]:= v; // 划分元素在位置 p
  end{Partition}
```

例 3.2.2 划分程序的执行情况： m=1, p=10 的情形

原数组：65 70 75 80 85 60 55 50 45 （＋∞）被划分成：(60, 45, 50, 55) , 65, (85, 80, 75, 70) （参看本章附页划分程序执行过程）。

程序 3-2-9 快速排序算法伪代码

```
proc QuickSort(p,q) //将数组 A[1..n]中的元素 A[p], A[p+1], ... , A[q]
   //按不降次序排列，并假定 A[n+1]是一个确定数，且大于 A[1..n]中所
   // 有的数。
   integer p,q;
   global n, A[1..n];
   if p<q then
      j:=q+1; Partition(p,j); // 划分后 j 成为划分元素的位置
      QuickSort(p,j-1);
      QuickSort(j+1,q);
   end{if}
 end{QuickSort}
```

由前面关于以比较为基础的排序算法在最坏情况下的时间下界可知，快速排序算法在最坏情况下的时间复杂性应不低于$\Omega(n\log n)$。事实上，在快速算法中元素比较的次数，在最坏情况下是$O(n^2)$。这是因为，在 Partition(m,p)的每一次调用中，元素的比较次数至多是 p-m.把 QuickSort 过程按照划分来分层，则第一层只调用 Partition 一次，即 Partition(1,n+1)，涉及的元素为 n 个；在第二层调用 Partition 两次，所涉及到的元素是 n-1 个，因为在第一层被选定的划分元素不在其中。若用 N_k 表示第 k 层调用 Partition 时所涉及的元素总个数，则 $N_k \le n-k+1$。这样在第 k 层调用 Partition 时发生的元素比较次数应不大于 n-k.注意到 $1\le k\le n$，因此 QuickSort 算法在最坏情况下总的元素比较次数不超过 n(n-1)/2，即 QuickSort 最坏情况下的时间复杂度为$O(n^2)$。

为了得到时间复杂性的平均值，我们不妨假设

1. 参加排序的 n 个元素互不相同；
2. Partition 中的划分元素 v 是随机选取的。

以 $C_A(n)$ 记时间复杂性的平均值。在上述假设条件下，调用 Partition(m,p)时，所取划分元素 v 是 A[m, p-1]中第 i($1\le i\le p-m$)小元素具有相等的概率，因而留下待排序的两个子组为 A[m..j-1]和 A[j+1..p-1]的概率是 1/(p-m)，$m\le j\le p-1$。由此得递归关系式

$$C_A(n) = n-1+\frac{1}{n}\sum_{1\le k\le n}(C_A(k-1)+C_A(n-k)) \qquad (3.2.4)$$

其中，$n-1$ 是 Partition 第一次被调用时所需要的元素比较次数，

$C_A(0)=C_A(1)=0$。将(3.2.4)式两端乘以 n 得

$$nC_A(n)=n(n-1)+2(C_A(0)+C_A(1)+\cdots+C_A(n-1)) \qquad (3.2.5)$$

用 n-1 替换(3.2.5)中的 n 得

$$(n-1)C_A(n-1)=(n-1)(n-2)+2(C_A(0)+C_A(1)+\cdots+C_A(n-2)) \qquad (3.2.6)$$

再用(3.2.5)减去(3.2.6)式得

$$\begin{aligned} C_A(n)/(n+1) &= C_A(n-1)/n+\frac{2(n-1)}{n(n+1)} \\ &\le C_A(n-1)/n+2/n \end{aligned} \qquad (3.2.7)$$

由递推关系式(3.2.7)，并注意到$C_A(0)=C_A(1)=0$，得

$$C_A(n)/(n+1)\le C_A(1)/2+2\sum_{2\le i\le n}1/i \qquad (3.2.8)$$

利用积分不等式

$$\sum_{2\le i\le n}1/i<\int_1^n\frac{dx}{x}=\ln n$$

得　$C_A(n)<2(n+1)\ln n=O(n\log n)$

看来快速排序与归并排序具有相同的平均时间复杂性。但是在实际表现中却是有所不同的。实验表明，快速排序一般要比归并排序效率更高些（参看《数据结构、算法与应用》，Sartaj Sahni 著，汪诗林 孙晓东 译 p.457）。

关于排序的算法我们已经接触了 5 种：冒泡排序、插入排序、选择排序、归并排序和快速排序，它们的时间复杂性列出如下：

算 法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
插入排序	n^2	n^2
选择排序	n^2	n^2
快速排序	n^2	n log n
归并排序	n log n	n log n

§3. 选 择 问 题

问题：已知 n 元数组 A[1..n]，试确定其中第 k 小的元素。

最容易想到的算法是采用一种排序算法先将数组按不降的次序排好，然后从排好序的数组中捡出第 k 个元素。但这样的算法在最坏情况下至少是 $O(n\log n)$。实际上，我们可以设计出在最坏情况下的时间复杂度为 $O(n)$ 的算法。为此，考察上节提到的算法 Partition。假设在一次划分中，划分元素 v 处于第 j 个位置。如果 k<j，则要找的第 k 小元素在新数组 A[1..j-1]中，而且是 A[1..j-1]的第 k 小元素；如果 k＝j，则划分元素 v 即是要找的第 k 小元素；如果 k>j，则要找的第 k 小元素在新数组 A[j＋1..n]中，而且是 A[j＋1..n]的第 k-j 小元素。

程序 3-3-1 采用划分的选择算法

```
proc PartSelect(A, n, k) //在数组 A[1..n]中找第 k 小元素 t，并将其存
    //放于位置 k，即 A[k]=t。而剩下的元素按着以 t 为划分元素的划分
    //规则存放。再令 A[n+1]=+∞.
    integer n, k, m, r, j;
    m:=1; r:=n+1; A[n+1]:= +∞;
    loop
      j:=r;
      Partition(m,j);
      case
        k=j : return // 返回 j,当前数组的元素 A[j]是第 j 小元素
        k<j : r:=j; // j 是新的下标上界
        else : m:=j+1; k:=k-j; //j+1 是新的下标下界
      end{case}
    end{loop}
end{PartSelect}
```

（参看本章附页 PartSelect 程序的执行过程）

这个算法在最坏情况下的时间复杂度是 $O(n^2)$。事实上，假定数组 A[1..n]中的元素互不相同，而且假定**划分元素是随机选取**的。注意，在每次调用 Partition(m, j)都要耗去 $O(j-m)$ 的时间。而下一次被划分的数组的元素个数至少比上一次减少 1。因而，从最初的划分中 m=1, j=n+1 开始，至多需要做 n-1 次划分。第一次划分，耗去时间至多为 n，第二次耗去时间至多是 n-1，… .所以，

PartSelect 在最坏情况下的时间复杂度为$O(n^2)$。但是，可以推出，PartSelect 的平均时间复杂度为$O(n)$。事实上，我们可以改进 PartSelect 算法，通过精心挑选划分元素 v，得到在最坏情况下的时间复杂度为$O(n)$的算法。

程序 3-3-2 改进的选择算法伪代码

```
proc Select(A, m, p, k) // 返回一个 i 值，使得 A[i]是 A[m..p]中第
   // k 小元素。r 是一个大于 1 的整数。
   global r;
   integer n, i, j;
   if p-m+1≤r then
     InSort(A, m, p);
     return(m+k-1);
   end{if}
   loop
      n:=p-m+1;
      for i to ⌊n/r⌋ do  //计算中间值
        InSort(A, m+(i-1)*r, m+i*r-1);
        //将中间值收集到 A[m..p]的前部:
        Swap(A[m+i-1], A[m+(i-1)*r+⌊r/2⌋-1]);
      end{for}
     j:=Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉);
     Swap(A[m], A[j]); //产生划分元素
     j:=p+1;
     Partition(m, j);
     case:
        j-m+1=k : return(j);
        j-m+1>k : p:=j-1;
        else   k:=k-(j-m+1); m:=j+1;
     end{case}
   end{loop}
end{Select}
```

这里，程序 Select 只在划分元素的选取上做了改进，其余部分沿用 PartSelect 的步骤。划分元素的选取方案是：取定正整数 r(>1),将原始数组按 r 个元素一段的原则分成⌊n/r⌋段（可能剩余 n-r*⌊n/r⌋个元素）。对每一段求取中

间元素，并把这⌊n/r⌋个中间元素搜集在数组 A[m..p]的前部（免去另开空间收存的操作）。现在调用程序

Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉)，

就产生了所要的划分元素。因为⌊n/r⌋一定小于 n，这样的递归过程是可行的。为了直接应用程序 PartSelect，将刚刚找到的划分元素放在数组 A[m..p]的首位。

我们以 r=5 来分析 Select 算法的时间复杂性。假设数组 A 中的元素都是互不相同的。由于每个具有 5 个元素的数组的中间值 u 是该数组的第 3 小元素，此数组至少有 3 个元素不大于 u;⌊n/5⌋个中间值中至少有⌈⌊n/5⌋/2⌉个不大于这些中间值的中间值 v。因而，在数组 A 中至少有

3*⌈⌊n/5⌋/2⌉ ≥ 1.5*⌊n/5⌋

个元素不大于 v。换句话说，A 中至多有

n-1.5*⌊n/5⌋= n-1.5*(n/5-e/5) ≤ 0.7n+1.2

个元素大于 v。同理，至多有 0.7n+1.2 个元素小于 v。这样，以 v 为划分元素所产生的新的数组至多有 0.7n+1.2 个元素。当 n≥24 时，0.7n+1.2≤0.75n=3n/4。

注意到程序 Select 中，从一层到下一层递归时，实际上相当于两次调用了 Select：一次体现在语句

j:=Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉);

另一次体现在 Partition(m, j)及后面的 case 语句组,其关键操作为Θ(n)次。主程序接着就要调用自身，执行规模不超过 3n/4 的选择问题。这两步涉及的数组规模分别是 n/5 和 ≤3n/4。程序中其它执行步的时间复杂度都至多是 n 的倍数。如果用 T(n)表示算法在数组长度为 n 的时间复杂度，则当 n≥24 时，有递归关系式

$$T(n) \le T(n/5) + T(3n/4) + cn \qquad (3.3.1)$$

其中 c 是常数。从递推关系式(3.3.1)出发，用数学归纳法可以证明

$$T(n) \le 20cn \qquad (3.3.2)$$

所以，在最坏情况下，Select 算法的时间复杂度是 $O(n)$。

§4. 关于矩阵乘法

假定 A, B 都是 n×n 矩阵，它们的 i 行 j 列元素分别记为 A(i, j)和 B(i, j)。如果用 S 和 C 分别记 A+B 和 A*B, 则有

$$S(i,j)=A(i,j)+B(i,j) \qquad 1\le i,j\le n$$
$$C(i,j)=\sum_{k=1}^{n}A(i,k)*B(k,j) \qquad 1\le i,j\le n \tag{3.4.1}$$

可见，矩阵加法运算的时间复杂度是$\Theta(n^2)$，而矩阵乘法的时间复杂度是$\Theta(n^3)$。后者是因为求每个元素$C(i,j)$都需要 n 次乘法和 n-1 次加法运算，而 C 共有n^2个元素。

如果用分治法解决矩阵的乘法问题，可以设想将矩阵分块，然后用分块矩阵乘法完成原矩阵的乘法运算。不妨假定 n 是 2 的方幂，$n=2^k$，将矩阵 A 和 B 等分成四块，于是

$$A*B=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix}$$

其中
$$C_{11}=A_{11}B_{11}+A_{12}B_{21} \qquad C_{12}=A_{11}B_{12}+A_{12}B_{22}$$
$$C_{21}=A_{21}B_{11}+A_{22}B_{21} \qquad C_{22}=A_{21}B_{12}+A_{22}B_{22} \tag{3.4.2}$$

如果用 T(n)记两个 n 阶矩阵相乘所用的时间，则有如下递归关系式：

$$T(n)=\begin{cases}b & n\le 2\\ 8T(n/2)+dn^2 & n>2\end{cases} \tag{3.4.3}$$

这是因为在计算 C 的块 C_{ij}时，需要计算 8 次 n/2 阶矩阵的乘法计算和 4 次 n/2 阶矩阵的加法计算，后者需要dn^2加法运算，这里 d 是一个常数。直接递推关系式(3.4.3)得

$$T(n)=bn^3/8+4d(n^2-16)/3$$

因为$b\ne 0$，所以$T(n)=\Theta(n^3)$。

虽然没有降低时间复杂度，但给我们一个启示。1969 年，斯特拉森(V.Strassen)发现了降低矩阵乘法时间复杂度的可能性。注意到，计算矩阵的加法比计算乘法的时间复杂度具有较低的阶$(n^2:n^3)$，而在用分块矩阵乘法时，既有矩阵加法又有矩阵乘法。如果能通过增加加法次数来减少乘法次数，则可能达到降低矩阵乘法的时间复杂度的目的。为此他设计了一个算法用以计算(3.4.2)式中的 C_{ij}，共用了 7 次乘法和 18 次加(减)法。令

$$P=(A_{11}+A_{22})(B_{11}+B_{22})$$
$$Q=(A_{21}+A_{22})B_{11}$$

$$
\begin{aligned}
R &= A_{11}(B_{12}-B_{22}) \\
S &= A_{22}(B_{21}-B_{11}) \\
T &= (A_{11}+A_{12})B_{22} \\
U &= (A_{21}-A_{11})(B_{11}+B_{12}) \\
V &= (A_{12}-A_{22})(B_{21}+B_{22})
\end{aligned}
\tag{3.4.4}
$$

则

$$
\begin{aligned}
C_{11} &= P+S-T+V \\
C_{12} &= R+T \\
C_{21} &= Q+S \\
C_{22} &= P+R-Q+U
\end{aligned}
\tag{3.4.5}
$$

由此得到的递推关系式为

$$
T(n)=\begin{cases} b & n\le 2 \\ 7T(n/2)+an^2 & n>2 \end{cases}
\tag{3.4.6}
$$

直接推导可得

$$
\begin{aligned}
T(n) &= an^2(1+7/4+(7/4)^2+\cdots+(7/4)^{k-2})+7^{k-1}T(2) \\
&= an^2\left(\frac{16}{21}\left(\frac{7}{4}\right)^{\log n}-\frac{4}{3}\right)+\frac{b}{7}(7)^{\log n} \\
&= an^2\left(\frac{16}{21}(n)^{\log\frac{7}{4}}-\frac{4}{3}\right)+\frac{b}{7}(n)^{\log 7} \\
&= \left(\frac{16a}{21}+\frac{b}{7}\right)n^{\log 7}-\frac{4a}{3}n^2 \\
&= \Theta\left(n^{2.81}\right)
\end{aligned}
\tag{3.4.7}
$$

从所得的结果看出，Strassen 算法的时间复杂度依赖于 2×2 矩阵的乘法运算所使用的乘法数。然而 Hoperoft 和 Kerr 在 1971 年已经证明：计算 2×2 矩阵的乘积，7 次乘法是必要的。因而，降低矩阵乘法运算的时间复杂度应改用其它分块的阶数，如采用 3×3 或 5×5 的块等。目前已知的最低的时间复杂度是 $O(n^{2.36})$。而目前所知道的矩阵乘法的最好下界仍是它的平凡下界 $\Omega(n^2)$。因此，到目前为止还无法确切知道矩阵乘法的时间复杂性。

Strassen 矩阵乘法采用的技巧也可以用于计算大整数的乘积。参看《计算机算法设计与分析》－王晓东编著，电子工业出版社，2001。

§5　快速 Fourier 变换

连续函数 $a(t)$ 的 Fourier 变换

$$A(f)=\int_{-\infty}^{\infty}a(t)e^{2\pi ift}dt$$

$A(f)$ 的逆变换为

$$a(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}A(f)e^{-2\pi ift}dt$$

N 个数据 $a=(a_0,a_1,\cdots,a_{N-1})$ 的离散 Fourier 变换

$$A_j=\sum_{0\le k\le N-1}a_k e^{2\pi ijk/N},\quad 0\le j<N$$

其逆变换为

$$a_k=\frac{1}{N}\sum_{0\le j\le N-1}A_j e^{-2\pi ijk/N},\quad 0\le k<N$$

这里，i 是虚数单位，N 是正整数。如果令 $\omega=e^{2\pi i/N}$，则 ω 是复数域上的 N 次本原单位根，此时，计算 A_j 相当于求多项式

$$a(x)=\sum_{0\le k\le N-1}a_k x^k$$

在 ω^j 处的值。根据 Horner 法则，需要做 $2(N-1)$ 次加法和乘法运算，因而，N 个数据的离散 Fourier 变换需要做 $2N(N-1)$ 次复数的加法和乘法运算。但是，我们可以设计一个分治算法，其时间复杂度为 $O(N\log N)$。

首先，若 ω 是 $N=2n$ 次本原单位根，则 ω^2 是 n 次本原单位根，而且，

$$\omega^{j+n}=-\omega^j,\ j=0,1,\cdots,n-1\text{。}$$

将多项式 $a(x)$ 的奇次项和偶次项分开，则

$$\begin{aligned}a(x)&=\left(a_1+a_3x^2+\cdots+a_{2n-1}x^{2(n-1)}\right)x+\left(a_0+a_2x^2+\cdots+a_{2(n-1)}x^{2(n-1)}\right)\\&=b(x^2)x+c(x^2)\end{aligned}$$

于是，

$$\begin{aligned}a(\omega^j)&=b(\omega^{2j})\omega^j+c(\omega^{2j})\\a(\omega^{j+n})&=-b(\omega^{2j})\omega^j+c(\omega^{2j})\end{aligned},\quad j=0,1,\cdots,n-1$$

其中，

$$b(y)=a_1+a_3y^2+\cdots+a_{2n-1}y^{n-1}$$
$$c(y)=a_0+a_2y+\cdots+a_{2(n-1)}y^{n-1}$$

这样，求 $N=2n$ 个数据的 Fourier 变换就归结为求两次具有 n 个数据的 Fourier 变换。

程序 3-5-1　快速 Fourier 变换

```
Proc FFT(N, a,w,A)
    # N=2^m,w 是 n 次单位根，a 是已知的 N 元数组，代表多项式 a(x)的系数，
    # A 是计算出来的 N 元数组，A[j]=a(w^j), j=0,1,…,N-1.
  real b[ ], c[ ];   int j;
  complex B[ ], C[ ], wp[ ];
  if N=1 then A[0]:=a[0];
  else
      n:=N/2;
      for j from 0 to n-1 do
          b[j]:=a[2*j+1];   c[j]:=a[2*j];
      end{for}
    end{if}
    FFT(n,b,w*w,B);
    FFT(n,c,w*w,C);
    wp[0]:=1;
    for j from 0 to n-1 do
      wp[j+1]:=w*wp[j];
      A[j]:= C[j]+B[j]*wp[j];   A[j+n]:= C[j]-B[j]*wp[j];
    end{for}
  end{FFT}
```

以 $T(N)$ 记算法 FFT 的时间复杂度，则

$$T(N)=\begin{cases} a, \text{if } N=1 \\ 2T(N/2)+cN, \text{if } N>1 \end{cases}$$

所以，算法的复杂度为 $T(N)=O(N\log N)$。

利用快速 Fourier 变换可以计算出两个多项式 $f(x),g(x)$ 的乘积，其时间复杂度为 $T(N)=O(N\log N)$，这里 N 是两个多项式的次数之和加 1。这可以分三步完成：第一步，采用 Fourier 变换计算出 $f(x),g(x)$ 在 $\omega^j=e^{2\pi ij/N}$ 处的值 $f(\omega^j),g(\omega^j)$，$j=0,1,\cdots,N-1$，这需要 $O(N\log N)$ 的计算量；第二步，构造 N 个数据

$A=\left[f(\omega^0)g(\omega^0),f(\omega^1)g(\omega^{1a}),\cdots,f(\omega^{N-1})g(\omega^{N-1})\right]$，它们实际上是乘积多项式 $a(x)=f(x)g(x)$ 在 $\omega^j=e^{2\pi ij/N}$，$j=0,1,\cdots,N-1$，处的值，这步需要 $O(N)$ 的计算量；第三步，利用 Fourier 变换的逆变换求出乘积多项式的系数 $a_k,k=0,1,\cdots,N-1$，这需要执行一次快速 Fourier 变换 FFT 即可，只不过那里的 ω 取为 $e^{-2\pi i/N}$，因而，这一步需要 $O(N\log N)$ 的计算量。

需要指出的是，快速 Fourier 变换需要本原单位根，被变换的数据的个数是 $N=2^m$，在复数域中任何 N 次本原单位根都是存在的，复数 $\omega=e^{2\pi i/N}$ 就是一个。但是计算机对复数的计算是近似的，因此，快速 Fourier 变换得到近似结果。象多项式乘积这样的问题（比如有理系数多项式）需要精确计算，这时我们需要能够进行精确计算的本原单位根，这需要我们考虑一般交换环上的本原单位根。

定义 3.5.1 设 R 是一个有单位元的交换环，n 是正整数，$\omega\in R$，

1）ω 称为一个 n 次单位根，如果 $\omega^n=1$；

2）ω 称为一个 n 次本原单位根，如果 ω 是 R 的可逆元，ω 是一个 n 次单位根，但对于 n 的任何素因子 p，$\omega^{n/p}-1$ 都不是环 R 的零因子。

n 次本原单位根具有如下性质：

I. 对于任何正整数 $0<l<n$，ω^l-1 都不是零因子；

II. $\sum\limits_{0\le j<n}\omega^{lj}=0$，$l=0,1,\cdots,n-1$；

III. 如果 ω 是 n 次本原单位根，则 ω^{-1} 也是；

IV. 如果 ω 是 $n=2m$ 次本原单位根，则 ω^2 是 m 次本原单位根，而且

$$\omega^{j+m}=-\omega^j,\ j=0,1,\cdots,m-1。$$

根据上述 4 条性质，我们有一般形式的 Fourier 变换和逆变换，快速 Fourier 变换，及应用 Fourier 变换设计求解其它问题的快速算法。

§6 最接近点对问题

已知空间中的 n 个点，如何找到最近的两个点即为最近点对问题。在直线上，最近点对问题可以通过一次排序和 1 维扫描来解决，这样做最好的时间复杂度为 $O(n\log n)$。实际上，可以设计一个分治算法来解决直线上的最近点对问题。

程序 3-6-1 求一维最近点对距离分治算法

```
proc ClosPair1(S, d)
  //S 是实轴上点的集合，参数 d 表示 S 中最近点对的距离
  global S,d;
  integer n;
  float m,p,q;
  n:=|S|;
  if n<2 then d:=∞; return(false); end{if}
  m:=S 中各点坐标的中位数;
  //划分集合 S
  S1:={x∈S| x≤m}; S2:={x∈S| x>m};
  ClosPair1(S1,d1);
  ClosPair1(S2,d2);
  p:=max(S1); q:=min(S2);
  d:=min(d1,d2,q-p);
  return(true);
end{ClosPair1}
```

以 T(n)记算法的时间复杂度，则有如下的递归关系式

$$T(n)=\begin{cases}O(1), & n<4;\\ 2T(n/2)+O(n), & n\geq 4\end{cases}$$

因为求中位数的时间复杂度可为 $O(n\log n)$ 。

上述解决一维最近点对问题的分治算法可以推广解决二维最近点对问题。这里有两个地方需要加工：

一是集合的划分，因为两个坐标，我们只能选取一个坐标作为划分的参考，比如选择集合 S 中各点的 x 坐标的中位数作为划分的标准

$$S1:=\{p\in S\mid x(p)\leq m_x\},\quad S2:=\{p\in S\mid x(p)>m_x\}$$

二是假设 $S1,S2$ 中最近点对的距离分别是 d_1,d_2，令 $d=\min\{d_1,d_2\}$，而且 d 不是 S 中最近点对的距离，则 S 中最近的点对 (p,q) 的两个点应该分布在 $S1,S2$ 两个集合中，譬如 $p\in S_1,q\in S_2$ 。与一维的情形不同，为找到这样的点对 (p,q)，需

要比照的点对有许多，最坏情况下会有$n^2/4$对。为此，需要通过分析缩小检查的范围。取定S_1中一点p，则S_2中使得点对(p,q)的距离不超过d的点q应该在下面的集合中

$$S_p=\{q\in S\mid m_x<x(q)\le m_x+d\ and\ y(p)-d\le y(q)\le y(p)+d\}$$

S_p处在一个$d\times 2d$的矩形区域中。因为S_p中任意两点的距离都不小于d，所以，S_p中至多有 6 个点。事实上，我们可以将这个矩形区域均分成 6 个子区域：纵向三等分，水平方向二等分，则每个区域中两点间的距离不超过：$\sqrt{(d/2)^2+(2d/3)^2}=5d/6$，因而，每个区域中至多有$S_p$的一个点。这样，为找到最近的点对$(p,q)$，其中$p\in S_1,q\in S_2$，只需检查至多$6\times\lceil n/2\rceil=3n+6$个点对。根据以上分析，可以设计一个分治算法，求解平面上最近点对问题。

程序 3-6-2 求一维最近点对距离分治算法

```
proc ClosPair2(S, d)
  //S 是平面上点的集合，但是已经按照y－坐标不降的次序排好，假定不同
  //点的x－坐标是不同的。参数 d 表示 S 中最近点对的距离,dist(p,q)表示
  //点对(p, q)之间的距离。
  global S,d;
  integer n;
  float m,p,q;
  n:=|S|;
  if n<2 then d:=∞; return(false); end{if}
  mx:=S 中各点x－坐标的中位数;
  //划分集合 S 成 S1 和 S2，它们中的点也都是按y－坐标不降地排列。
  S1:={p∈S| x(p)≤mx}; S2:={q∈S| x(q)>mx};
  ClosPair2(S1,d1);
  ClosPair2(S2,d2);
  dm:=min{d1,d2}; d:=dm;
  //检查距离直线 x=mx 不远于 dm 的两个条形区域中的点对
  P1:={p∈S1|mx-dm≤x(p)}; P2:={q∈S2| x(q)≤mx+dm};
```

```
    flag:=1;
    for i to |P1| do
      k:=flag;
      while y(P2[k])<y(P1[i])-dm do
        k:=k+1;
      end{while}
      flag:=k;
      for j from flag to |P2| do
        if y(P2[j])>y(P1[i])+dm then break;
        else d:=min{d,dist(p,P2[j])};
        end{if}
      end{for}
    end{for}
    return(true);
  end{ClosPair2}
```

分析这个算法的时间复杂度，假定为 T(n)。集合 S 划分成集合 S1 和 S2 只需要对集合 S 进行一次扫描即可，复杂度为Θ(n)次操作；程序本身的两次自调用复杂度为 2T(n/2)；形成集合 P1 和 P2 的复杂度为Θ(n)；在外部的 for 循环中包含两部分，while 循环和一个内部 for 循环。while 循环总的循环次数不会超过 n,因而复杂度为Θ(n)；每个内部 for 循环的循环次数不会超过 6，而外部循环次数不超过$\lceil n/2 \rceil$，故所有的内部 for 循环的循环次数总起来不超过 3n+6。程序的其它操作总起来都是常数，因而，

$$T(n) \le \begin{cases} a, & n < 4 \\ 2T(n/2) + cn, & n \ge 4 \end{cases}$$

其中，c 为正实数。据此可解出 $T(n) = O(n \log n)$ 。

习题 三

1. 编写程序实现归并排序算法 MergeSortL 和快速排序算法 QuickSort;

2. 用长分别为 100、200、300、400、500、600、700、800、900、1000 的 10 个数组的排列来统计这两种算法的时间复杂性；

3．讨论归并排序算法 MergeSort 的空间复杂性。

4．说明算法 PartSelect 的平均时间复杂性为 $O(n)$ 。

提示：假定数组中的元素各不相同，且第一次划分时划分元素v是第i小元素的概率为$1/n$。因为 Partition 中的 case 语句所要求的时间都是$O(n)$，所以，存在常数c，使得算法 PartSelect 的平均时间复杂度$C_A^k(n)$可以表示为

$$C_A^k(n) \le cn + \frac{1}{n}\left(\sum_{1\le i<k} C_A^{k-i}(n-i) + \sum_{k<i\le n} C_A^k(i-1)\right) \tag{1}$$

令$R(n) = \max_k(C_A^k(n))$，取$c \ge R(1)$，试证明$R(n) \le 4cn$。

附页

归并排序的 C++语言描述

```
#include<iostream.h>
template<class T>void MergeSort(T a[],int left,int right);
template<class T>void Merge(T c[],T d[], int l,int m,int r);
template<class T>void Copy(T a[],T b[],int l,int r);
void main()
{
    int const n(5);
    int a[n];
    cout<<"Input "<<n<<"numbers please:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    //for(int j=0;j<n;j++)
        //b[j]=a[j];
    MergeSort(a,0,n-1);
    cout<<"The sorted array is"<<endl;
    for(i=0;i<n;i++)
        cout<<a[i];
    cout<<endl;
}

template<class T>
void MergeSort(T a[],int left,int right) //
{
    if(left<right)
    {
        int i=(left+right)/2;
         T *b=new T[];
        MergeSort(a,left,i);
        MergeSort(a,i+1,right);
        Merge(a,b,left,i,right);
        Copy(a,b,left,right);
    }
```

```
}

template<class T>
void Merge(T c[],T d[],int l,int m,int r)
{
    int i=l;
    int j=m+1;
    int k=l;
    while((i<=m)&&(j<=r))
    {
        if(c[i]<=c[j])d[k++]=c[i++];
        else d[k++]=c[j++];
    }
    if(i>m)
    {
        for(int q=j;q<=r;q++)
            d[k++]=c[q];
    }
    else
        for(int q=i;q<=m;q++)
            d[k++]=c[q];
}

template<class T>
void Copy(T a[],T b[], int l,int r)
{
    for(int i=l;i<=r;i++)
        a[i]=b[i];
}
```

快速排序的 C++语言描述

```
#include<iostream.h>
template<class T>void QuickSort(T a[],int p,int r);
template<class T>int Partition(T a[],int p,int r);
void main()
```

```
{
    int const n(5);
    int a[n];
    cout<<"Input "<<n<<"numbers please:";
    for(int i=0;i<n;i++)
        cin>>a[i];
    QuickSort(a,0,n-1);
    cout<<"The sorted array is"<<endl;
    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

template<class T>
void QuickSort(T a[],int p,int r)
{
    if(p<r)
    {
        int q=Partition(a,p,r);
        QuickSort(a,p,q-1);
        QuickSort(a,q+1,r);
    }
}

template<class T>
int Partition(T a[],int p,int r)
{
    int i=p,j=r+1;
    T x=a[p];
    while(true)
    {
        while(a[++i]<x);
        while(a[--j]>x);
        if(i>=j)break;
        Swap(a[i],a[j]);
```

```
    }
    a[p]=a[j];
    a[j]=x;
    return j;
}

template<class T>
inline void Swap(T &s,T &t)
{
    T temp=s;
    s=t;
    t=temp;
}
```

附页： PartSelect 程序的执行过程

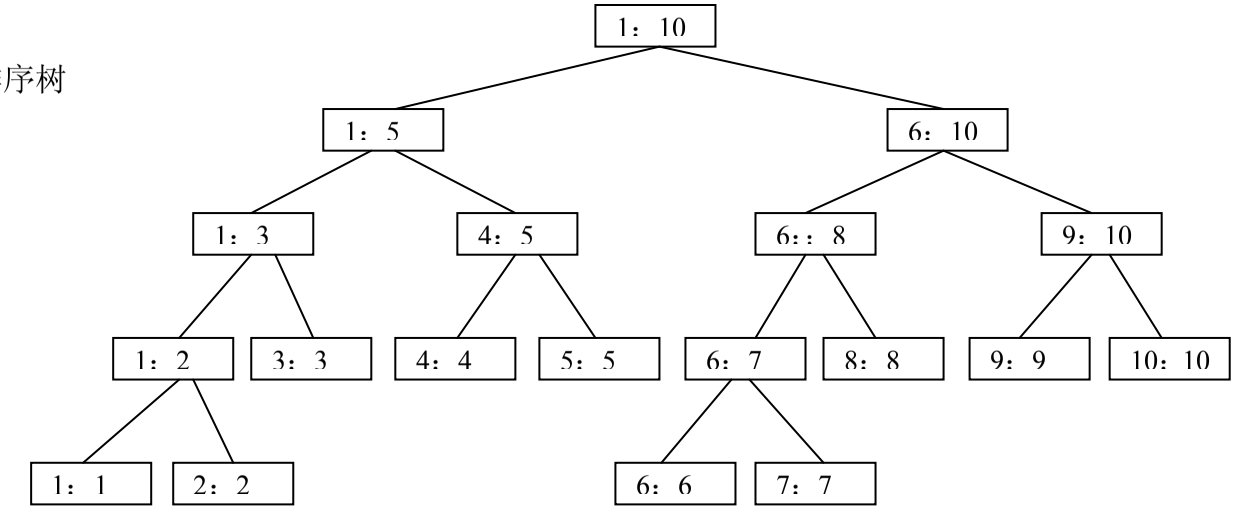
数组 A[1:9]=[65,70,75,80,85,60,55,50,45]　的划分过程

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	*i*	*p*
65	70	75	80	85	60	55	50	45	+∞	2	9
65	45	75	80	85	60	55	50	70	+∞	3	8
65	45	50	80	85	60	55	75	70	+∞	4	7
65	45	50	55	85	60	80	75	70	+∞	5	6
										6 > 5	
65	45	50	55	60	85	80	75	70	+∞	1	5
60	45	50	55	65	85	80	75	70	+∞		

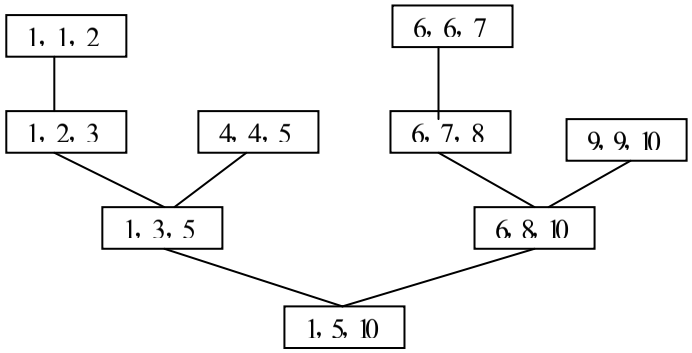
例子，数组 A[1:9]=[65,70,75,80,85,60,55,50,45]，求第 7 小元素

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	*i*	*p*
Partition(1,10) j=5												
Partition(6,10) j=9	60	45	50	55	65	85	80	75	70	+∞	10	9
Partition(6,9) j=6	60	45	50	55	65	70	80	75	85	+∞	7	6
Partition(7,9) j=8	60	45	50	55	65	70	80	75	85	+∞	9	8
Partition(7,8) j=7	60	45	50	55	65	70	75	80	85	+∞	8	7

附页：归并排序树

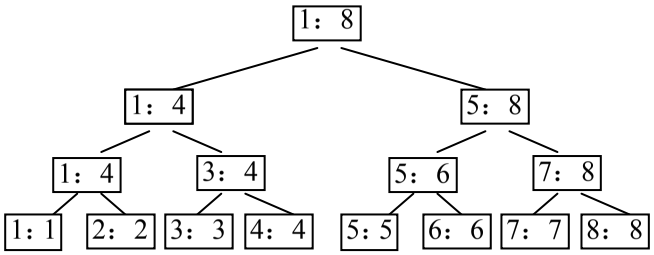


（含有 10 个元素数组的）MergeSort 调用（分拆）过程



Merge 调用（合并）过程

附页：链表归并过程



MergeSort 调用（分拆）过程

链表的归并过程

A = [50 , 10 , 25 , 30 , 15 , 70 , 35 , 55]　　数据表

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

Link=[0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0]　　初始化为零，逐步修改

q　r　p

1>2→ 2 ,　0 ,　1 ,　0 ,　0 ,　0 ,　0 ,　0 ,　0 [10,50]

3<4→ 3 ,　0 ,　1 ,　4 ,　0 ,　0 ,　0 ,　0 ,　0 [10,50], [25,30]

2<3→ 2 ,　0 ,　3 ,　4 ,　1 ,　0 ,　0 ,　0 ,　0 [10,25,30,50]

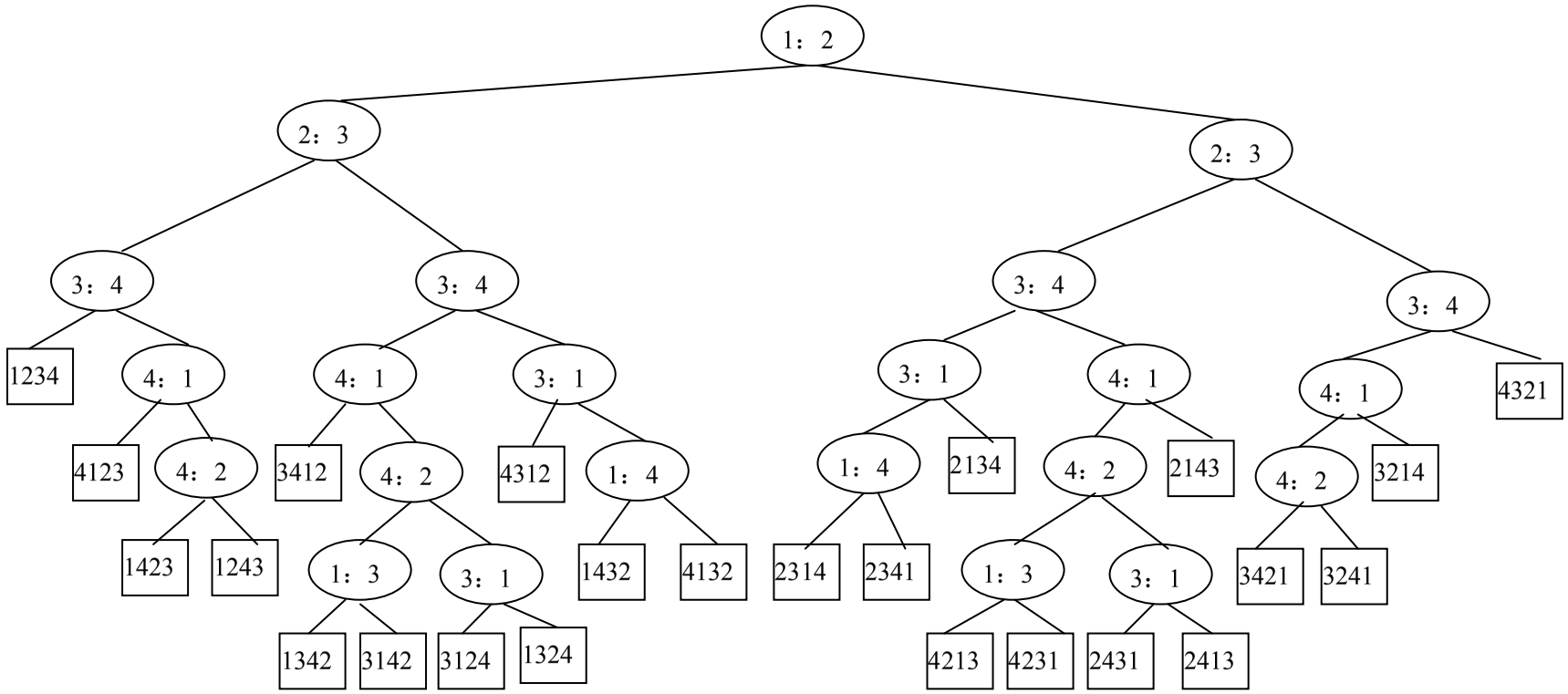
5<6→ 5 ,　0 ,　3 ,　4 ,　1 ,　6 ,　0 ,　0 ,　0 [10,25,30,50],[15,70]

7<8→7,　0,　3 ,　4 ,　1 ,　6 ,　0 ,　8,　0 [10,25,30,50],[15,70],[35,55]

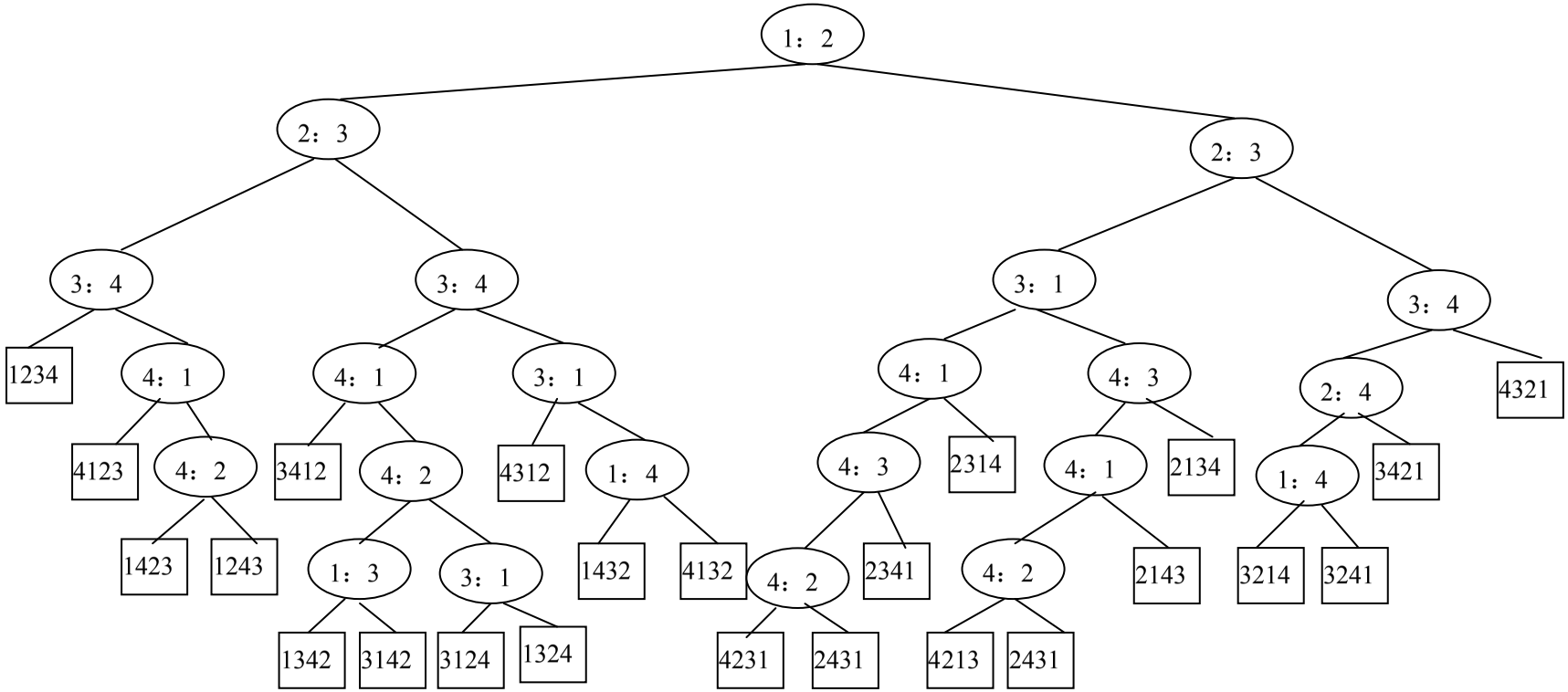
5<7→5,　0,　3 ,　4 ,　1 ,　7 ,　0 ,　8，　6 [10,25,30,50],[15, 35,55,70]

2<5→2,　8 ,　5 ,　4 ,　7 ,　3 ,　0 ,　1，　6 [10, 15,25,30, 35,50, 55,70]

附页：排序比较树



N=4 时的一棵比较树


1：2
2：3
2：3
3：4
3：4
3：1
3：4
1234
4：1
4：1
3：1
4：1
4：3
2：4
4321
4123
4：2
3412
4：2
4312
1：4
4：3
2314
4：1
2134
1：4
3421
1423
1243
1：3
3：1
1432
4132
4：2
2341
4：2
2143
3214
3241
1342
3142
3124
1324
4231
2431
4213
2431

N=4 时的另一棵比较树

第四章 贪心算法

§1. 贪心算法基本思想

找零钱 假如售货员需要找给小孩 67 美分的零钱。现在，售货员手中只有 25 美分、10 美分、5 美分和 1 美分的硬币。在小孩的催促下，售货员想尽快将钱找给小孩。她的做法是：先找不大于 67 美分的最大硬币 25 美分硬币，再找不大于 67－25＝42 美分的最大硬币 25 美分硬币，再找不大于 42－25＝17 美分的最大硬币 10 美分硬币，再找不大于 17－10＝7 美分的最大硬币 5 美分硬币，最后售货员再找出两个 1 美分的硬币。至此，售货员共找给小孩 6 枚硬币。售货员的原则是拿尽可能少的硬币找给小孩。

装载问题 有一艘大船用来装载货物。假设有 n 个货箱，它们的体积相同，重量分别是 $w_1, w_2, \cdots, w_n$，货船的最大载重量是 c。目标是在船上装更多个数的货箱该怎样装？当然，最简单的做法是“捡重量轻的箱子先往上装”，这是一种贪心的想法。如果用 $x_i = 1$ 表示装第 i 个货箱，而 $x_i = 0$ 表示不装第 i 个货箱，则上述问题是解优化问题：

求 $x_1, x_2, \cdots, x_n$，使得

$$\max \sum_{i=1}^{n} x_i \tag{4.1.1}$$

满足条件 $$\sum_{i=1}^{n} \mathrm{w}_i x_i \le c, \quad x_i = 0,1 \tag{4.1.2}$$

贪心算法，顾名思义，是*在决策中总是做出在当前看来是最好的选择*。例如找零钱问题中，售货员每捡一个硬币都想着使自己手中的钱尽快达到需要找钱的总数。在装载问题中，每装一个货箱都想着在不超重的前提下让船装更多的箱子。但是贪心方法并未考虑整体最优，它所做出的选择只是在某种意义上的局部最优。当然，在采用贪心算法时未必不希望结果是整体最优的。事实上，有相当一部分问题，采用贪心算法能够达到整体最优，如前面的找零钱问题以及后面将要讲到的单点源最短路径问题、最小生成树问题、工件排序问题等。为了更好理解贪心算法，我们将装载问题稍加推广，考虑可分割的背包问题。

背包问题 已知容量为 M 的背包和 n 件物品。第 i 件物品的重量为 w_i，价值是 p_i。因而将物品 i 的一部分 x_i 放进背包即获得 $p_i x_i$ 的价值。问题是：怎样装包使所获得的价值最大？即是如下的优化问题：

$$\max \sum_{1\le i\le n} p_i x_i \tag{4.1.3}$$

$$\begin{aligned} &\sum_{1\le i\le n} w_i x_i \le M \\ &0 \le x_i \le 1,\ p_i > 0, w_i > 0,\ 1 \le i \le n \end{aligned} \tag{4.1.4}$$

采用贪心算法，有几种原则可循：a）.每次捡最轻的物品装；b）.每次捡当前价值最大的物品装；c）.每次装包时既考虑物品的重量又考虑物品的价值，也就是说每次捡单位价值 p_i/w_i 最大的物品装。按原则 a)来装只考虑到多装些物品，但由于单位价值未必高，总价值可能达不到最大；按原则 b)来装，每次选择的价值最大，但同时也可能占用了较大的空间，装的物品少，未必能够达到总价值最大。比较合理的原则是 c)。事实上，按照原则 c）来装，确实能够达到总价值最大。

程序 4-1-1 背包问题贪心算法

```
proc GreedyKnapsack (p, w, M, x, n) //价值数组 p[1..n]、重量数组
   // w[1..n]，它们元素的排列顺序满足 p[i]/w[i]≥p[i+1]/w[i+1];
   // M 是背包容量，x 是解向量
  float p[1..n], w[1..n], x[1..n], M, rc;
  integer i, n;
  x:= 0;  // 将解向量初始化为零
  rc:= M; // 背包的剩余容量初始化为 M
  for i to n do
    if w[i] ≤ rc then
      x[i]:=1; rc:=rc-w[i];
    else break;
    end{if}
  end{for}
  if i≤n then
    x[i]:=rc/w[i];
  end{if}
 end{GreedyKnapsack}
```

定理 4.1.1 如果 $p[1]/w[1] \ge p[2]/w[2] \ge \cdots \ge p[n]/w[n]$，则 GreedyKnapsack 对于给定的背包问题实例生成一个最优解。

证明 设 $x=(x_1,x_2,\cdots,x_n)$ 是 GreedyKnapsack 所生成的解，但不是最优解。

因而必有某个x_i不为 1。不妨设x_j是第一个这样的分量。于是，当$1 \le i < j$时，$x_i = 1$；当$i = j$时，$0 \le x_i < 1$；当$j < i \le n$时，$x_i = 0$，而且$\sum w_i x_i = M$。因为x不是最优解，必存在解向量$y = (y_1, y_2, \cdots, y_n)$，使得$\sum p_i y_i > \sum p_i x_i$。设$k$是使得$y_k \ne x_k$的最小下标，则 $y_k < x_k$. 这是因为，当$k < j$时，$x_k = 1$，上述不等式自然成立；当$k \ge j$时，若 $x_k < y_k$，则由$x_1 = y_1, \cdots$，$x_{k-1} = y_{k-1}, x_{k+1} = 0, \cdots x_n = 0$，可推出

$$\sum_{i=1}^{k-1} w_i y_i + w_k y_k + \sum_{i=k+1}^{n} w_i y_i > \sum_{i=1}^{k-1} w_i x_i + w_k x_k = \sum_{i=1}^{n} w_i x_i = M$$

y不是可行解，矛盾。

因y是比x更优的解，不失一般性，可以假定$\sum_{i=1}^{n} w_i y_i = M$。于是

$$\sum_{i=1}^{k-1} w_i y_i + w_k y_k + \sum_{i=k+1}^{n} w_i y_i = \sum_{i=1}^{k-1} w_i x_i + w_k x_k \quad (+ \sum_{i=k+1}^{j} w_i x_i \text{ 当 } k < j \text{ 时})。$$

再由$y_k < x_k$，有$\sum_{i=k+1}^{n} w_i y_i \ge w_k (x_k - y_k) > 0$。现在取新的向量$z = (z_1, z_2, \cdots, z_n)$满足

$$z_1 = y_1, \cdots, z_{k-1} = y_{k-1}, z_k = x_k, \quad 0 \le z_{k+1} \le y_{k+1}, \cdots, 0 \le z_n \le y_n \quad ,$$

$$\sum_{k+1 \le i \le n} w_i (y_i - z_i) = w_k (z_k - y_k)$$

这样的向量z是存在的，而且是背包问题的可行解，因为

$$\begin{aligned}\sum_{1 \le i \le n} w_i z_i &= \sum_{1 \le i \le k-1} w_i y_i + w_k z_k + \sum_{k+1 \le i \le n} w_i z_i \\ &= \sum_{1 \le i \le k-1} w_i y_i + \sum_{k \le i \le n} w_i y_i \\ &= \sum_{1 \le i \le n} w_i y_i \le M\end{aligned}$$

至此，我们找到一个新的解向量z。以下证明它的总价值不小于y的总价值：

$$\begin{aligned}\sum_{1 \le i \le n} p_i z_i &= \sum_{1 \le i \le n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \sum_{k+1 \le i \le n} (y_i - z_i) w_i p_i / w_i \\ &\ge \sum_{1 \le i \le n} p_i y_i + \left((z_k - y_k) w_k - \sum_{k+1 \le i \le n} (y_i - z_i) w_i \right) p_k / w_k \\ &= \sum_{1 \le i \le n} p_i y_i\end{aligned}$$

中间的不等式是由于当$i>k$时有$p[k]/w[k]\geq p[i]/w[i]$而得。但是z与x的第一个不同分量的位置比y与x的第一个不同的分量的位置至少向后推迟了一个。以z代替y进行上面的讨论，我们又可以找到新的解向量z'，如此等等，由于分量的个数n有限，这样的过程必到某一步停止，最后找到解向量$y*$,它和x有相同的分量，又比x有更大的总价值，矛盾。这个矛盾源于x不是最优解的假设。证毕

贪心算法主要用于处理优化问题。每个优化问题都是由目标函数和约束条件组成。满足约束条件的解称为可行解，而那些使得目标函数取得最大（最小）值的可行解称为最优解。如背包问题是一个优化问题，式(4.1.3)中的函数是目标函数，而(4.1.4)式描述的要求是约束条件，这里优化是使目标函数取最大值。

贪心算法在每一步的决策中虽然没有完全顾忌到问题整体优化，但在局部择优中是朝着整体优化的方向发展的。为此，贪心算法首先要确定一个度量准则（称为贪心准则），每一步都是按这个准则选取优化方案。如背包问题的贪心准则是***选取单位价值 p/w 最大物品；***而装载问题的贪心的准则是***选取最轻的货箱；***找零钱问题所用的贪心准则是***选取面值最大的硬币。***对于一个给定的问题，初看起来，往往有若干种贪心准则可选，但在实际上，其中的多数都不能使贪心算法达到问题的最优解。如背包问题的下面实例：

n=3, M=20, p=(25, 24, 15), w=(18,15,10)

如果以价值最大为贪心准则,则贪心算法的执行过程是:首先考虑将物品 1 装包，此时获得效益值 25，包的剩余容量是 2。然后考虑将物品 2 装包，但物品 2 的重量 15 超出包的剩余容量，只能装入该种物品的 2/15，此时获得的总效益值为

25+24×2/15=28.2。

这样得到的可行解（1，2/15，0）并不是最优解。

事实上，如果以单位价值最大为贪心准则，则贪心算法的执行过程是：先计算出各个物品的单位价值(25/18, 24/15, 15/10)=(1.389, 1.6, 1.5)。首先考虑单位价值大的物品装包，即将物品 2 装包，此时获得效益值 24，背包的剩余容量是 5。然后考虑装物品 3，由于物品 3 的重量超出背包的剩余容量，只能装入该物品 5/10 =1/2, 至此背包已经装满,所得的总的效益值为 24+15/2=31.5。比前面的装法的效益值大。实践证明，选择能产生最优解的贪心准则是设计贪心算法的核心问题。以下给出贪心算法流程的伪代码。

程序 4-1-2 贪心算法抽象化控制流程

```
proc Greedy(A, n) // A[1:n]代表那个输入
    solution={}; //解向量初始化为空集
```

```
    for i to n do
        x:=Select(A);
        if Feasible(solution, x) then
            solution:=Union(solution, x);
        end{if}
    end{for}
    return(solution);
end{Greedy}
```

这里 Select(A)是按照贪心准则选取 A 中的输入项；Feasible(solution, x)是判断已知的解的部分 solution 与新选取的 x 的结合 Union(solution, x)是否是可行解。过程 Greedy 描述了用贪心策略设计算法的主要工作和基本控制流程。一旦给出一个特定的问题，就可将 Select，Feasible 和 Union 具体化并付诸实现。

§2. 调度问题

● 活动安排问题

我们首先从活动安排这一简单问题入手。该问题要求高效安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法使得尽可能多的活动能够兼容地使用公共资源。

问题： *已知 n 个活动 E={1, 2, …, n}，要求使用同一资源，第 k 个活动要求的开始和结束时间为 s_k, f_k, 其中 $s_k < f_k$, k=1, 2, …, n. 活动 k 与活动 j 称为相容的如果 $s_k > f_j$ 或者 $s_j > f_k$。活动安排问题就是要在所给的活动集合中选出最大(活动个数最多)的相容活动子集。*

解决这个问题的基本思路是在安排时应该将结束时间早的活动尽量往前安排，好给后面的活动安排留出更多的时间，从而达到安排最多活动的目的。据此，贪心准则应当是：*在未安排的活动中挑选结束时间最早的活动安排*。在贪心算法中，将各项活动的开始时间和结束时间分别用两个数组 s 和 f 存储，并使得数组中元素的顺序按结束时间非降排列：$f_1 \le f_2 \le \cdots \le f_n$。

算法 4-2-1 活动安排贪心算法伪代码

```
proc GreedyAction(s, f, n) // s[1..n]、f[1..n]分别代表 n 项活动的
    //起始时间和结束时间, 并且满足 f[1]≤ f[2]≤…≤ f[n]
    j:=1; solution:={1}; //解向量初始化
    for i from 2 to n do
```

```
      if si≥fj then
        solution:=solution ∪ {i}; // 将 j 加入解中
        j:=i;
      end{if}
    end{for}
    return(solution);
  end{GreedyAction}
```

例 4.2.1 待安排的 11 个活动的开始时间和结束时间按结束时间的非减次序排列如下：

表 1 一个作业安排表

	1	2	3	4	5	6	7	8	9	10	11
s[k]	1√	3	0	5√	3	5	6	8√	8	2	12√
f[k]	4	5	6	7	8	9	10	11	12	13	14

解集合为 {1,4,8,11}.容易证明算法 GreedyAction 所得到的解是最优解。

● 带期限的单机作业调度问题

为使问题简化，我们假定完成每项作业所用的时间都是一样的，如都是 1。带期限的单机作业安排问题陈述如下：

已知 n 项作业 E={1, 2, …, n}，要求使用同台机器完成（该台机器在同一时刻至多进行一个作业），而且每项作业需要的时间都是 1。第 k 项作业要求在时刻 f_k 之前完成，而且完成这项作业将获得效益 p_k，k=1, 2, …, n 。作业集 E 的子集称为相容的，如果其中的作业可以被安排由一台机器完成。带限期单机作业安排问题就是要在所给的作业集合中选出总效益值最大的相容子集。

这个问题可以考虑用贪心算法求解。容易想到贪心准则应该是：

尽量选取效益值大的作业安排

用两个数组 f 和 p 分别存放作业的期限值和效益值，并使得数组 p 中元素按照不增的顺序排列。按照上述贪心准则，算法由前向后逐个考察各个作业是否可以加入到解集 J 中即可。J 初始为空集，按照相容性要求，作业 i 可以加入 J 只要当前解集 J 中期限值不大于 f_i 的作业少于 f_i 个。由该算法取得的解集 J 具有性质：**若作业 k 不属于 J，则 J 中恰有 f_k 个期限不超过 f_k、而效益值不小于 p_k 的作业。**该算法在程序 4-2-2 中实现。可以证明算法 GreedyJob 获得的解集是带期限单机作业调度问题的最优解。下面的伪代码给出带期限作业调度问题贪心算法的概略描述，其中没有给出相容性检验的具体步骤。

程序 4-2-2 带限期作业调度的贪心算法（概略）

```
proc GreedyJob(f, p, n) //f[1..n]和p[1..n]分别代表各项作业的
   //限期和效益值，而且n项作业的排序满足：p1 ≥ p2 ≥ … ≥ pn
   local J;
   J:={1};//初始化解集
   for i from 2 to n do
      if J ∪ {i} 中的作业是相容的 then //此步验证需要认真设计
        J:= J ∪ {i}; // 将i加入解中
      end{if}
   end{for}
end{GreedyJob}
```

定理 4.2.1 算法 GreedyJob 对于单机作业调度问题总是得到最优解。

证明：假设贪心算法所选择的作业集 J 不是最优解，则一定有相容作业集 I，其产生更大的效益值。假定 I 是具有最大效益值的相容作业集中使得|I∩J|最大者，往证$I=J$。

反证法：若$I=J$不成立，则这两个作业集I和J之间没有包含关系。这是因为算法 GreedyJob 的特性和假定 I 产生的效益值比 J 的效益值更大。假设 a 是 J\I 中具有最大效益的作业，即 **J 中比 a 具有更大效益的作业(如果有的话)都应该在 I 中**。如果作业$b\in I\setminus J$且$p_b>p_a$，那么由算法中对J中作业的选取办法（相容性要求），J中至少有f_b个效益值$\geq p_b$的作业，其期限值$\leq f_b$。这些作业一定在I中，因而I中至少有f_b+1个作业，其期限值$\leq f_b$，这与I的相容性矛盾。所以，$I\setminus J$**中作业的效益值均不超过**p_a。

称区间$[k-1,k]$为时间片k，相容作业集I的一个调度表就是指定I中各个作业的加工时间片。如果I有一个调度表S将f_a时刻前的时间片安排的都是$I\cap J$中的作业，且$I\setminus J$中最早被安排的是作业b，在时间片k上，则$k>f_a$。前$k-1$个时间片上安排的都是J中的作业，这些作业的集A_1再添上作业a得到J中k个作业。由作业集J的相容性，A_1中至少有一个作业其期限值$\geq k$。将这个作业与作业b交换安排的时间片，得到新的调度表S_1。如果在调度表S_1中作业b安

排在时间片k_1，$k_1 > f_a$，则同理，可以将作业b再向前移，得到新的调度表S_2。如此做下去，必得到I的调度表S'，其在f_a时刻前的时间片安排有$I \setminus J$中作业b。令$I' = (I \setminus \{b\}) \cup \{a\}$，则$I'$也是相容的作业集。而且，由于$p_b \le p_a$，$I'$的效益总值不小于$I$的效益总值，因而$I'$具有最大的效益总值，但$|I' \cap J| > |I \cap J|$，与$I$的选取相悖。因而，$J = I$，$J$是最优解。 证毕

算法 GreedyJob 能够找到达到最大效益值的相容作业集，若按照作业期限的先后排序，就产生一个作业调度表。可以在算法中加建调度表的步骤：把被选中的作业安排在其期限允许的、而且期限晚于它的已安排作业之前的时间片上。如果用 J(r)表示安排在第 r 个时间片上的解集 J 中的作业，D(i)表示作业 i 的期限值，则相容性条件要求 D(J(r))≥r。根据上面分析，可以如下设计算法：

首先将作业 1 存入解数组 J 中，然后依次处理作业 2 到作业 n。假设已经处理了前 i-1 个作业，其中有 k 个作业 J(1),…,J(k)被选入 J 中，对它们的安排满足 D(J(1))≤ …≤ D(J(k))。为了检验作业集 J∪{i}的相容性，只需看能否找到按期限的非降次序插入作业 i 的适当位置，即，使得作业 i 在此处插入后有 D(J(r))≥r，1≤r≤k+1。找插入位置可以将作业 i 与 J 中作业依 D(J(k)),D(J(k-1)),…,D(J(1))的次序逐个比较。如果 D(i)≥D(J(j))>j 则选取作业 i 并将作业 i 插到作业 J(j)后面的位置上；如果 D(i)=D(J(j))=j，则作业 i 不能选入；如果 D(i)<D(J(j))且 D(i)≤j，则作业 i 不能选入；如果 D(i)<D(J(j))且 D(i)>j，则将作业 i 与作业 J(j-1)比较。如此下去，直到确定作业 i 不能被选入或可以选入并插入到 J 中适当位置为止。这样得到的作业集是相容的，而且作业是按期限的非降次序安排的。程序 4-2-3 给出了单机带期限作业调度贪心算法的伪代码。

程序 4-2-3 带期限作业调度问题贪心算法

```
proc GreedyJob(D,J,n,k)
  //D(1),…,D(n)是期限值，作业已按 p1≥ p2 ≥ … ≥ pn排序。J(i)是
  //最优解中的第 i 个作业。终止时，D(J(i))≤ D(J(i+1))，1≤i≤k
  integer D[0..n],J[0..n],i,k,n,r
  D(0):=0; J(0):=0;  //初始化
  k:=1; J(1):=1;  //计入作业 1，k 表示当前选择的作业个数
  for i from 2 to n do
  //按 p 的非增次序考虑作业，检查作业 i 可否被选入，并找出插入位置
     r:=k;
     while D(J(r))>D(i) and D(J(r))<>r do
```

```
        r:=r-1;
      end{while};
      //期限不晚于D(i)的作业个数小于D(i)时
      if D(J(r))≤ D(i) and D(i)>r then
        for j from k to r+1 by -1 do //给作业i腾出位置
          J(j+1):=J(j);
        end{for};
       J(r+1):=i; k:=k+1;
      end{if};
    end{for};
  end{GreedyJob}
```

例 4.2.2 设 n=7，$(p_1, p_2, \cdots, p_n)=(35,30,25,20,15,10,5)$，

$(d_1, d_2, \cdots, d_n)=(4,2,4,3,4,8,3)$，

算法 GreedyJob 的执行过程可描述如下：

作业	1；	2，1；	2，1，3；	2，4，1，3；	2，4，1，3，6；
期限值	4；	2，4；	2，4，4；	2，3，4，4；	2，3，4，4，8；

算法的关键操作是比较和相应移动作业的位置，都是围绕着判断作业集 J∪{i}的相容性。考察 J 外的作业 i 该加入 J 时，作业至多和 J 中的每个作业都比较一次。因而，J ∪ {i}的相容性判断最坏情况下需要|J|次比较。上述算法在最坏情况下的时间复杂度为$O(n^2)$。

为了避免上述算法反复移动作业的排序位置，可以考虑在算法中建立作业调度表，而且采取“将作业尽量安排在期限允许的最后面空闲时间片上”的策略。显然，所需要的时间片数不会超过 q:=min{n,max{D(i)|i=1,…,n}}。为了检验作业集 J 添加作业 i 是否还是相容的，只需检查时刻 k:=min{n,D(i)}前面是否还有空闲的时间片即可。例 4.2.2 按上述算法执行的情况如下：

作业	1；	2，1；	2，3，1；	4，2，3，1；	4，2，3，1，6；
期限值	4；	2，4；	2，4，4；	3，2，4，4；	3，2，4，4，8；
时间片	4；	2，4；	2，3，4；	1，2，3，4；	1，2，3，4，7；

其中，时间片 k 指区间[k-1,k]。

将时间片作为集合的元素，采用集合的并 Union 和查找 Find（参考附录“数据结构”）技术实现上述算法，可使时间复杂度达到$O(n\cdot\beta(2n,n))$，其中$\beta(2n,n)$

是 Ackermann 函数的逆函数。

程序 4-2-4 带期限作业调度快速算法

```
proc FastJob(D,J,n,q,k)
  //找最优解 J=J(1),…,J(k).假定 p1≥ p2≥ … ≥ pn.
  //q:=min{n,max{D(i)|i=1,2,…,n}}.
  integer q,D(n),J(n),F(0..q),P(0..q)
  for i from 0 to q do  //将集合树置初值
    F(i):=i; P(i):=-1;
  end{for}
  k:=0 //初始化
  for i to n do  //使用贪心规则
    j:=Find(min{n,D(i)});
    if F(j)≠0 then
      k:=k+1; J(k):=i;  //选择作业 i
      s:=Find(F(j)-1); Union(s,j);
      F(j):=F(s); P(s):=P(s)+P(j);
    end{if}
  end{for}
end{FastJob}
```

算法中 P(i)将作业链接到它的集合树上，同时指出该树上作业的个数。为了便于描述算法，还引进了 0 时间片[-1,0]。例 4.2.2 的执行情况在本章附页“快速作业调度算法”中描绘出来。

● 多机调度问题

设有 n 项独立的作业{1,2,…, n}，由 m 台相同的机器加工处理。作业 i 所需要的处理时间为 t_i。约定：任何一项作业可在任何一台机器上处理，但未完工前不准中断处理；任何作业不能拆分更小的子作业分段处理。*多机调度问题要求给出一种调度方案，使所给的 n 个作业在尽可能短的时间内由 m 台机器处理完。*

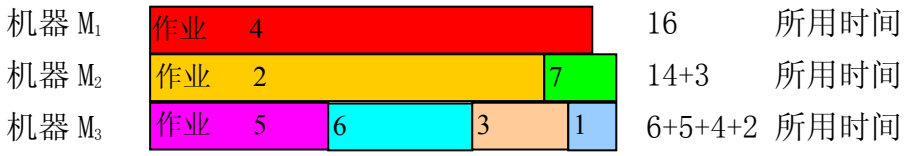
这是一个 NP 完全问题，到目前为止还没有一个有效的解法。利用贪心策略，有时可以设计出较好的近似解。可以采用贪心准则：*需要长时间处理的作业优先处理。*

例 4.2.3 设有 7 项独立的作业{1,2,3,4,5,6,7}，要由三台机器 M_1, M_2 , M_3 处理。各个作业所需要的处理时间分别为{2,14,4,16,6,5,3}.将作业标号按它们所需时间的长短排列：[4, 2, 5, 6, 3, 7,1].首先将作业 4 安排给机器 M_1 处理，

此时，机器 M_1 被占用的时间是 16，而机器 M_2 ，M_3 被占用的时间都是零，所以，应将作业 2 安排给机器 M_2 来处理，作业 5 应安排给机器 M_3 来处理，至此，机器 M_1，M_2 ，M_3 被占用的时间分别是 16、14、6。接下去应安排作业 6，因为机器 M_3 最早空闲，所以作业 6 应安排给机器 M_3。此时，机器 M_1，M_2 ，M_3 分别在 16、14、11 时刻开始空闲。所以下面将要安排的作业 3 安排给机器 M_3。此时，机器 M_1，M_2，M_3 分别在 16、14、15 时刻开始空闲。即将安排的作业 7 应安排给机器 M_2，此时，机器 M_1，M_2 ，M_3 分别在时刻 16、17、15 开始空闲。即将安排的作业 1 应安排给机器 M_3。如此全部作业均已安排完毕，所需要的时间是 17。这里采用的调度方案是：

将需要时间最长的未被安排作业首先安排给能够最早空闲下来的机器处理。

下面的图描述了上述例子的算法执行过程。



§3. 最优生成树问题

考虑无向图 G=(V, E)，不妨假定该图代表城市间的交通情况，顶点代表城市，边代表连接两个城市的可能的交通线路。现在将每条边赋予一个权值，这些权可以代表建造该条线路的成本、交通线的长度或其它信息，这样得到一个赋权图。在实际问题中，人们往往希望在结构上选择一组交通线，它们**连通所有的城市并且具有最小的建造成本。**这个问题相当于求取连通赋权图的具有最小权值的生成树。我们称之为最优生成树。贪心方法可以很好地解决此类问题。在此介绍两种算法：Prim 算法和 Kruskal 算法。

Prim 算法的基本思想：

1). 选择图 G 的一条权值最小的边 e_1，形成一棵两点一边的子树；

2). 假设 G 的一棵子树 T 已经确定；

3). 选择 G 的不在 T 中的具有最小权值的边 e，使得 $T\cup\{e\}$ 仍是 G 的一棵子树。

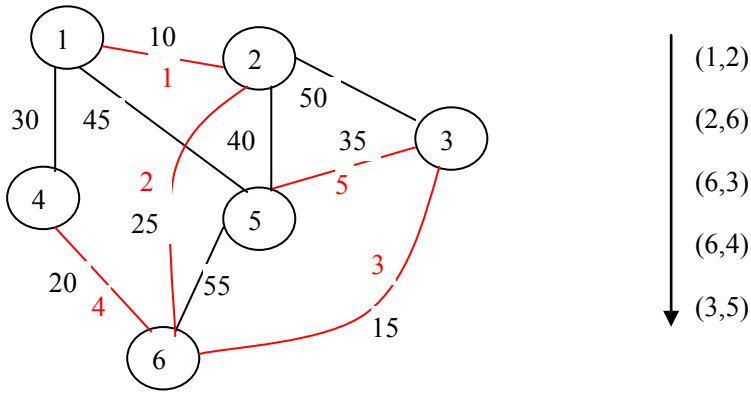
Kruskal 算法的基本思想：

1). 选择图 G 的一条权值最小的边 e_1；

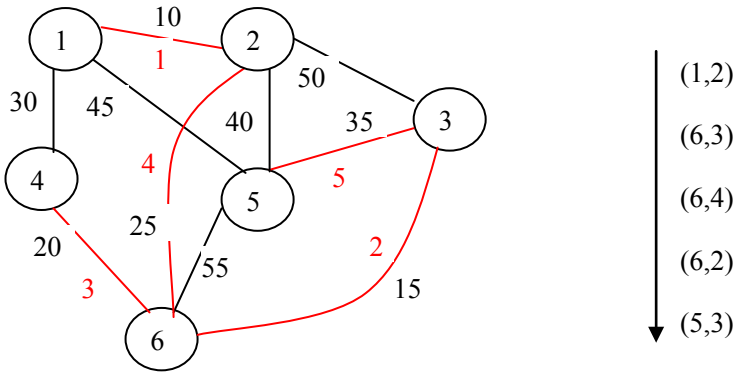
2). 假设已经选好 G 的一组边 $L=\{e_1, e_2, \cdots, e_k\}$；

3). 选择 G 的不在 L 中的具有最小权值的边 e_{k+1}，使得 $L\cup\{e_{k+1}\}$ 诱导出的 G 的子图不含 G 的圈。下面两个图给出 Prim 算法和 Kruskal 算法的最优生成树产生过程。

Prim
算 法



Kruskal
算法



程序 4-3-1 Prim 最小生成树算法

```
proc PrimTree(E,COST,n,T,mincost) //E 是图 G 的边集，COST 是 G 的带权
    //邻接矩阵。计算一棵最小生成树 T 并把它作为一个集合存放到二维
    //数组 T[1..n-1,1..2]中，这棵树的权值赋给 mincost
  real COST[1..n,1..n],mincost;
  integer NEAR[1..n],n,i,j,k,s,T[1..n-1,1..2];
  (k,s):= 权值最小的边;
  mincost:=COST[k,s];
  (T[1,1],T[1,2]):=(k,s); //初始子树
  for i to n do //将 NEAR 赋初值（关于初始子树 T）
     if COST[i,s]<COST[i,k] then
       NEAR(i)=s;
     else NEAR(i)=k;
     end{if}
  end{for}
  NEAR(k):=0,NEAR(s):=0;
```

```
13  for i from 2 to n-1 do  //寻找 T 的其余 n-2 条边
14    choose an index j such that
        NEAR(j)≠0 and COST[j,NEAR(j)] is of minimum value;
15    (T[i,1],T[i,2]):=(j,NEAR(j)); // 加入边（j，NEAR(j)）
16    mincost:=mincost+COST[j,NEAR(j)];
17    NEAR(j):=0;
18    for t to n do  //更新 NEAR（关于当前子树 T）
19      if NEAR(t)≠0 and COST[t,NEAR(t)]>COST[t,j] then
20        NEAR(t):=j;
21      end{if}
22    end{for}
23  end{for}
24  if mincost ≥ ∞ then
25    print( 'no spanning tree' );
26  end{if}
27 end{PrimTree}
```

这里，树 T 的元素为(T[i,1],T[i,2]),其中 T[i,1],T[i,2]代表第 i 条边的两个端点。NEAR(j)表示顶点 j 的以最小权的边邻接 T 的一个顶点。

PrimTree 算法的时间复杂度分析：

语句 3 需要 O(|E|)的时间；语句 6 至语句 11 的循环体需要时间为 O(n)；

语句 18 至 22 的循环体需要时间 O(n)；

语句 14 至 17 需要时间 O(n)；

语句 13 至 23 的循环体共需要时间 $O(n^2)$。

所以,PrimTree 算法的时间复杂度为 $O(n^2)$.

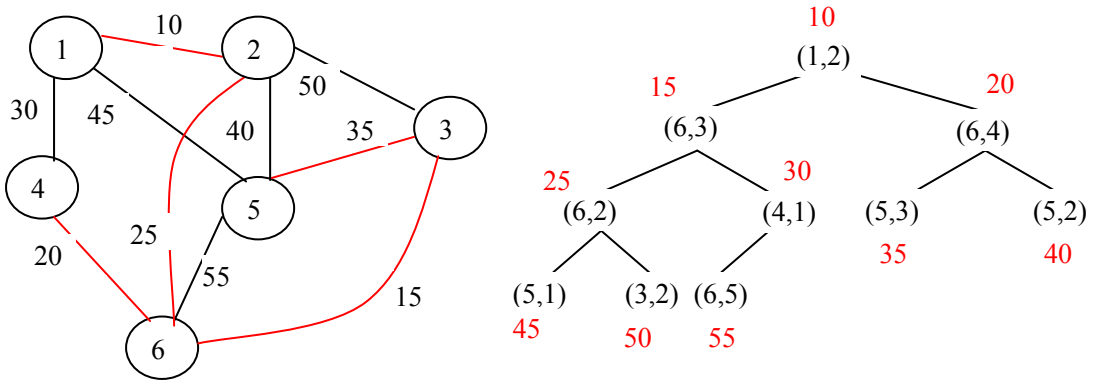
为叙述 Kruskal 算法，我们引进数据结构 min-堆：它是一个完全二叉树，其中每个结点都有一个权值，而且该二叉树满足：每个结点的权值都不大于其儿子的权值。min-堆的根具有最小的权值。

程序 4-3-2 Kruskal 最优生成树算法伪代码

```
proc KruskalTree(E,COST,n,T,mincost)//说明同算法 PrimTree
1  real mincost, COST[1..n,1..n];
2  integer Parent[1..n],T[1..n-1],n;
3  以带权的边为元素构造一个 min-堆;
4  Parent:=-1; //每个顶点都在不同的集合中;
```

```
i:= 0; mincost:= 0;
while i<n-1 and min-堆非空 do
    从堆中删去最小权边（u，v）并重新构造 min-堆
    j:= Find(u); k:= Find(v);
    if j≠k then  //保证不出现圈
      i=:i+1; T[i,1]:=u; T[i,2]:=v;
      mincost:=mincost+COST[u,v];
      Union(j,k); //把两个子树联合起来
   end{if}
end{while}
if i≠n-1 then
  print( 'no spanning tree' );
end{if}
return;
end{KruskalTree}
```

一个赋权图和它的
边的最小堆



定理 4.3.1 Kruskal 算法对于每一个无向连通赋权图产生一棵最优树。

证明：设 T 是用 Kruskal 算法产生的 G 的一棵生成树，而 T' 是 G 的一棵最优生成树且使得 $|E(T')\cap E(T)|$ 最大。用 E(T) 表示树 T 的边集，w(e) 表示边 e 的权值，而边集 E 的权值之和用 w(E) 表示。以下证明 E(T))=E(T'). 反假设 $E(T)\neq E(T')$，因为 $|E(T)|=|E(T')|$，所以 E(T) 与 E(T') 没有包含关系。设 e 是 $E(T)\backslash E(T')$ 中权值最小的边。将 e 添加到 T'中即得到 T'的一个圈：$e, e_1, e_2, \cdots, e_k$。因为 T 是树，诸 e_i 中至少有一个不属于 E(T)。不妨设 e_i 不属于 E(T)，则必然有 $w(e)\leq w(e_i)$。否则，由 $w(e)>w(e_i)$ 以及 E(T) 中比 e 权值小的边都在 T'中，e_i 同这

些边一起不含有圈，因而，按 Kruskal 算法，e_i将被选到 E(T)中，矛盾。在 T′中去掉边 e_i，换上边 e，得到 G 的一棵新的生成树 T″,这棵树有两个特点：

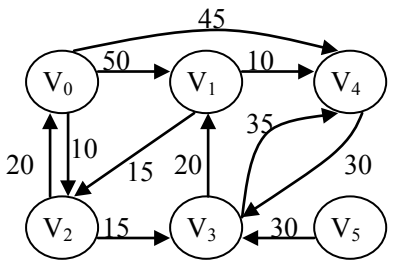
a). T″的权值不大于 T′的权值，因而与 T′有相等的权值；

b).|E(T″)∩E(T)| > |E(T′)∩E(T)|.

a)说明 T″也是一棵最优树，而 b)说明与 T′取法相悖。因此 E(T) = E(T′)，T 是最优生成树。　　证毕

§4 单点源最短路径问题

问题：已知一个赋权有向图 G=(V,E，w),求由 G 中某个指定的顶点 v_0出发到其它各个顶点的最短路径。



	路径	长度
(1)	v_0v_2	10
(2)	$v_0v_2v_3$	25
(3)	$v_0v_2v_3v_1$	45
(4)	v_0v_4	45

从 v_0 到其它各顶点的最短距离

对于一般的单点源最短路径问题，我们采用逐条构造最短路径的办法， 用

迄今已生成的所有路径长度之和为最小

作为贪心准则，这要求，每一条已被生成的单独路径都必须具有最小长度。假定已经构造了 k 条最短路径，则下面要构造的路径应该是下一条最短长度的最短度路径。现记这 k 条最短路径的终点之集为 S，为陈述方便，也将 v_0放于 S 中。如果 V\S 不是空集，则从 v_0到 V\S 中顶点的最短路径中应该有一条最短的，比如是 v_0到 v_{k+1}的最短路径 P：

$$P=v_0u_1\cdots u_{s-1}u_sv_{k+1} \tag{4.4.1}$$

显然，$P_1=v_0u_1\cdots u_{s-1}u_s$应是 v_0到 u_s的最短路径，因而由 S 的定义和选定的贪心准则，u_s应属于 S。同理，路径 P 上其它顶点 u_i也都在 S 中。所以，由 v_0出发的新的最短路径一定是某个已有的最短路径向前延伸一步。如果用 Dist(v_i)记从 v_0到 S 中顶点 v_i的最短路径的长度，而图 G 中的顶点 w 依其属于 S 与否分别记之为 S(w)=1 或 S(w)=0，则从 v_0出发，新的最短路径的长度应该是

$$D(S)=\min_{S(u)=1,S(w)=0}\{Dist(u)+COST(u,w)\} \tag{4.4.2}$$

满足(4.4.2)式的顶点 w 被选择加入 S，新的最短路径就是从 v_0出发到 w 的最短路径,而此时的 Dist(w)=D(S)，S 被更新为 $S'=S\cup\{w\}$,后者可以由更新 w 的（集

合）特征值来实现：S(w)=1(原来 S(w)=0).上述算法思想是 Dijkstra(迪杰斯特)提出的。

程序 4-4-1 Dijkstra 最短路径算法伪代码

```
proc DijkstraPaths(v,COST,n,Dist,Parent)//G 是具有 n 个顶点
   //{1,2,…,n}的有向图， v 是 G 中取定的顶点，COST 是 G 的邻接矩阵,
   //Dist 表示 v 到各点的最短路径之长度,Parent 表示各顶点在最短路径
   //上的前继。
    bool S[1..n]; float COST[1..n,1..n],Dist[1..n];
    integer u,v,n,num,i,w;
1     for i to n do     //将集合 S 初始化为空
2        S[i]:=0; Parent[i]:=v; Dist[i]:=COST[v,i];
3     end{for}
4     S[v]:=1; Dist[v]:=0; Parent[v]:=-1;//首先将节点 v 记入 S
5     for i to n-1 do    //确定由节点 v 出发的 n-1 条最短路
6        选取顶点 w 使得 Dist(w) = min_{S(u)=0} {Dist(u)}
7        S[w]:=1;
8        while S[u]=0 do
            //修改 v 通过 S 到达 S 以外的结点 u 的最小距离值
9           if Dist[u]> Dist[w]+COST[w,u] then
10             Dist[u]:=Dist[w]+COST[w,u];
11             Parent[u]:=w;
12          end{if}
13       end{while}
14    end{for}
15  End{DijkstraPath}
```

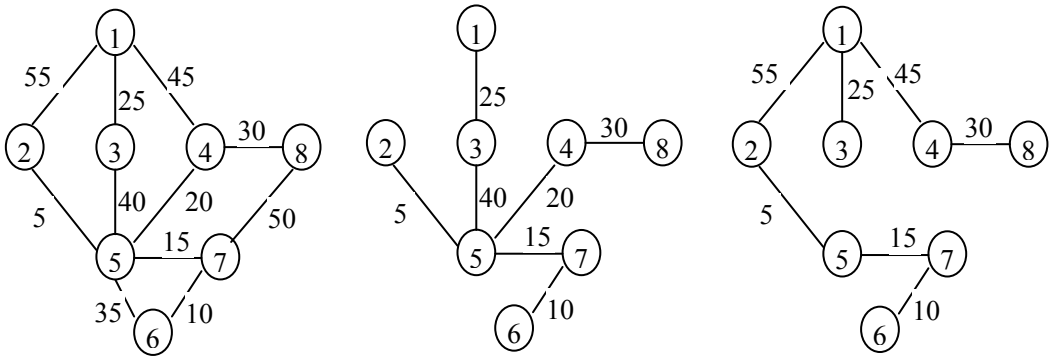
这里需要注意语句 6 和 10。根据(4.4.2)式，被选择的新的最短路径的长度应该等于语句 6 中的 $\min_{S(u)=0}\{Dist(u)\}$，这可由语句 10 中的 Dist 值更新语句看出。假设 S 变成 S′=S∪{w}后，下一次被选中的节点是 u，如果 Dist(u)在 S 变成 S′=S∪{w}后值发生了变化，则必然等于 Dist(w)+ COST(w,u)。事实上，假如在 w,u 的中间还有顶点 p，使得 w 通过 p 到 u 的距离比 COST(w,u)还小，则 v 到 p 的距离必然比 v 到 u 的距离更短，这与 u 被选中矛盾。所以，COST(w,u)是从 w 到 u 的最短路径的长度，进而 Dist(w)+ COST(w,u)是 v 到 u 的最短路径长度。因此 S 变

成 S′=S∪{w}后，S′之外各顶点 Dist 值的更新采用

Dist(u)=min{Dist(u),Dist(w)+ COST(w,u)}

必将使下一步选点得到 v 到 S′之外各顶点最短路径之最小的顶点，这保证了算法 DijstraPath 的正确性。对于每个不是 v 的顶点 u，可以通过回溯 Parent(u)得到由 v 到达 u 的最短路径。

算法的时间复杂度为$O(n^2)$。这是因为，在 for 循环中，求得元素 w 需要做$\Theta(n-i-1)$比较；而在 while 循环中考虑更新 Dist 的值时，也需要$\Theta(n-i-1)$操作，因而总的操作次数应该是$O(n^2)$。



赋权连通图 G　　G 的最优生成树　　G 的一棵单点源最短路径生成树

所有由 v 到达各顶点的最短路径并起来构成图 G 的一棵生成树，称为单点源最短路径树。上页的图给出了一个连通赋权图 G 及它的两棵生成树：最优生成树和单点源最短路径树。

§5 Huffman 编码

哈夫曼编码是用于数据文件压缩的一个十分有效的编码方法，其压缩率通常在 20％～90％之间。哈夫曼编码算法使用字符在文件中出现的频率表来建立一个 0,1 串，以表示各个字符的最优表示方式。下表给出的是具有 100,000 个字符文件中出现的 6 个不同的字符的出现频率统计。

表 4-5-1　字符出现频率表

不同的字符	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100

如果采用定长码，则每个字符至少需用三位，该文件总共需要 300,000 位；

如果采用上面所列变长码，则该文件需用

（45×1＋13×3＋12×3＋16×3＋9×4＋5×4）×1000＝224,000 位，

总码长减少了 25%。

一般的编码都是沿一个方向连续地写下 0,1 字串，比如从左到右、从上到下。解码时也是以同样的方向逐字地搜索。为了使解码唯一，必须使用一个规则。**前缀码**即是一种规则，它要求表示任何一个字符的 0,1 字串都不能是表示另一个字符的 0,1 字串的前部。比如，若用 01 表示字符 a，则表示其它字符的字串不能是具有形式：01…。

可以用一棵二叉树来表示前缀编码。用树叶代表给定的字符，并将给定字符的前缀码看作是从树根到代表该字符的树叶的一条道路。代码中每一位的 0 或 1 分别作为指示某节点到左儿子或右儿子的“路标”。

在一般情况下，若 C 是编码字符集，则表示其最优前缀码的二叉树中恰有 |C| 个叶结点，每个叶结点对应于一个字符。这样的二叉树的每个不是叶的结点恰有两个子结点，因而，该二叉树恰有 |C|-1 个内部结点。给定编码字符集 C 及其频率分布 f，即 C 中任一字符 c 以频率 f(c)在数据文件中出现。C 的一个前缀编码方案对应于一棵二叉树 T。字符 c 在树中的深度记为 $d_T(c)$。$d_T(c)$也是字符 c 的前缀码长。该编码方案的平均码长定义为

$$B(T)=\sum_{c\in C} f(c)d_T(c) \tag{4.5.1}$$

使平均码长达到最小的前缀编码方案称为 C 的一个最优编码。

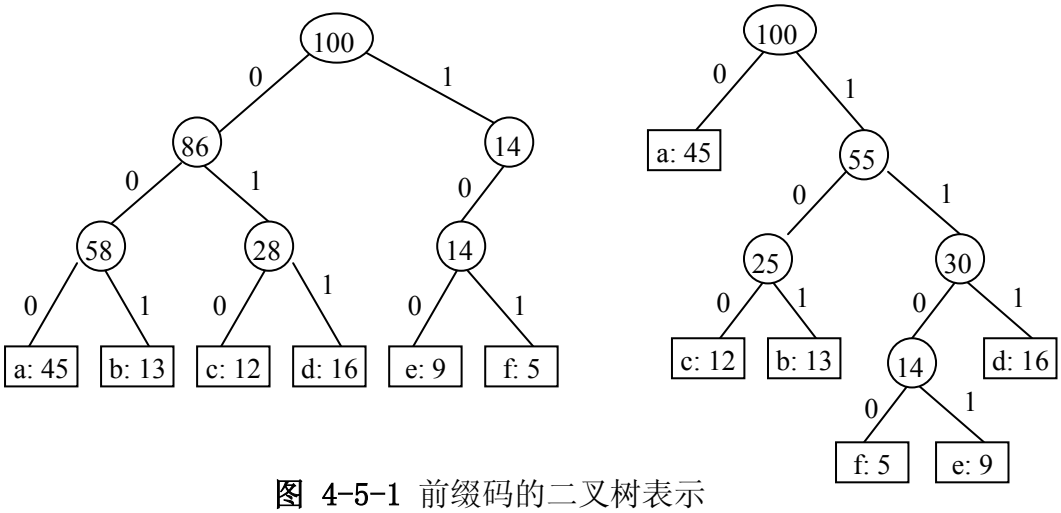


图 4-5-1 前缀码的二叉树表示

哈夫曼提出了一种构造最优前缀编码的贪心算法，称为哈夫曼编码。该算法以自底向上的方式构造表示最优前缀编码的二叉树 T。算法以 |C| 个叶结点开始，执行 |C|-1 次“合并”运算后产生最终所要求的二叉树 T。

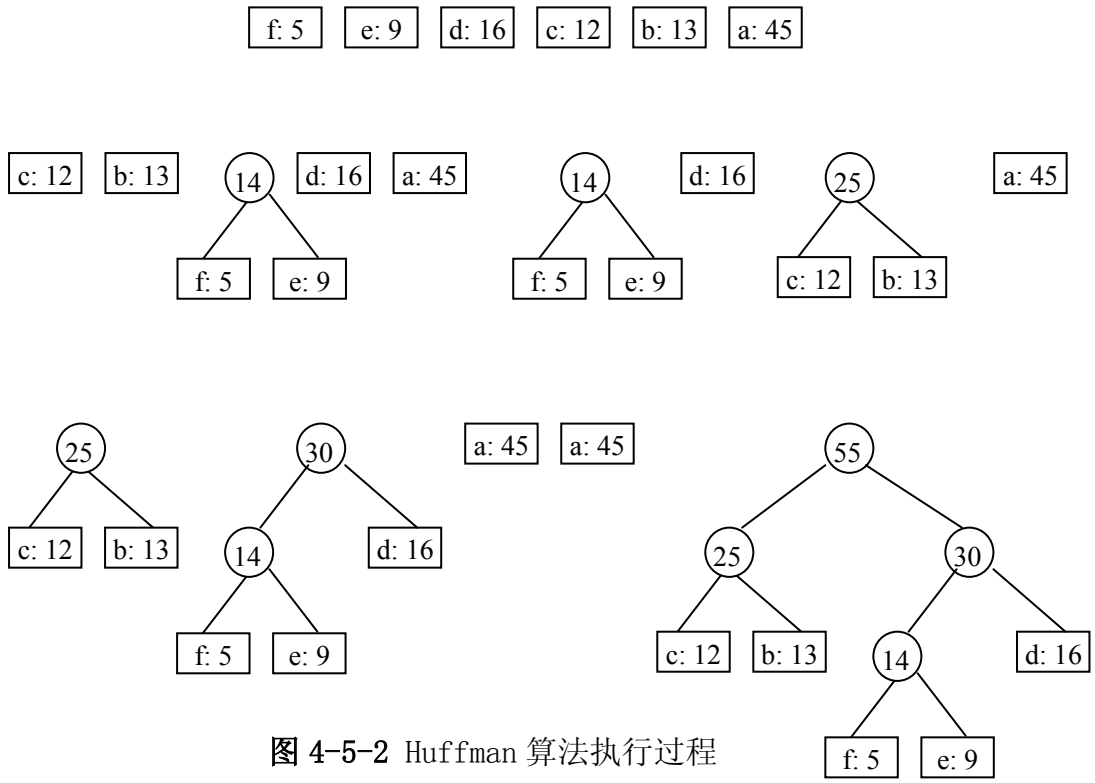


图 4-5-2 Huffman 算法执行过程

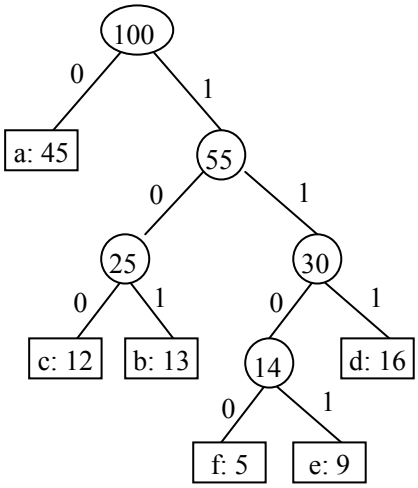


图 4-5-3 Huffman 编码树

Huffman 编码首先用字符集 C 中的每一个字符 c 的频率 f(c)初始化一个优先队列 Q，然后不断地从优先队列 Q 中取出具有最小频率的两棵树 x 和 y，将它们合并成一棵新树 z，z 的频率是 x 与 y 的频率之和。新树 z 以 x 为其左儿子，y 为其右儿子（当然，也可以 y 为左儿子，x 为右儿子）。经过至多 n-1 次合并后（假定 $|C| = n$）,优先队列中只剩下一棵树，即是所求的 Huffman 树。

设计 Huffman 算法可以用最小堆实现优先队列 Q。初始化最小堆需要 $O(n)$ 计

算时间。由于 DeleteMin 和 Insert 只需要$O(\log n)$时间，$n-1$次的合并共需要$O(n\log n)$计算时间。因此，关于n个字符的 Huffman 编码的计算时间为$O(n\log n)$。

定理 4.5.1 Huffman 编码产生最优前缀编码方案。

证明： 设 T 是一棵表示最优前缀编码的二叉树，编码的字符集为 C。我们先证明：对于频率最小的两个字符 x,y，可以将它们调换到最深叶顶点的位置而使新树 T′的平均码长不增加。事实上，假设 b,c 是两个最深的叶顶点，它们是兄弟，具有同样的深度。不妨设 f(b)≤f(c),f(x)≤f(y)。在树 T 中将节点 b 与节点 x 互换，得到一个新树 T′。此时

$$B(T)-B(T')=f(b)d_T(b)+f(x)d_T(x)-f(b)d_T(x)-f(x)d_T(b)$$
$$=(f(b)-f(x))(d_T(b)-d_T(x))\geq 0$$

如果在树 T′中将节点 y 与节点 c 互换，得到一棵新树 T″，同理可证 B(T′)≥B(T″)，但 T 是最有编码树，所以 B(T″)＝B(T)，说明 T″也是最优编码树。其次，如果令 T″中节点 x,y 的父节点代表一个新的字符 z，出现频率为

$$f(z)=f(x)+f(y),$$

则 T″去掉节点 x,y 后得到一个以{C\{x,y})∪{z}为字符集的最优前缀编码树。对于这棵树叶可以象对待树 T 那样进行调整，使得新树将具有最小频率的两个字符放在最深的叶子节点位置。如此继续下去，最后得到一棵只有一个节点的树，它是只有一个字符，出现率为 100%的最优编码树。以这棵树为根逐步将上述过程中摘掉的叶节点恢复出来，得到的就是 Huffman 树。所以 Huffman 树是最优树，即平均码长最短的树。 证毕

习题 四

1． 设有n个顾客同时等待一项服务。顾客i需要的服务时间为$t_i, 1\leq i\leq n$。应该如何安排n个顾客的服务次序才能使总的等待时间达到最小？总的等待时间是各顾客等待服务的时间的总和。试给出你的做法的理由（证明）。

2． 字符$a\sim h$出现的频率分布恰好是前 8 个 Fibonacci 数，它们的 Huffman 编码是什么？将结果推广到n个字符的频率分布恰好是前n个 Fibonacci 数的情形。Fibonacci 数的定义为：$F_0=1, F_1=1, F_n=F_{n-2}+F_{n-1}\ if\ n>1$

3． 设$p_1,p_2,\cdots,p_n$是准备存放到长为 L 的磁带上的 n 个程序，程序p_i需要的

带长为a_i。设$\sum_{i=1}^{n} a_i > L$，要求选取一个能放在带上的程序的最大子集合（即其中含有最多个数的程序）Q。构造Q的一种贪心策略是按a_i的非降次序将程序计入集合。

1) 证明这一策略总能找到最大子集Q，使得$\sum_{p_i \in Q} a_i \le L$。

2) 设Q是使用上述贪心算法得到的子集合，磁带的利用率可以小到何种程度？

3) 试说明 1)中提到的设计策略不一定得到使$\sum_{p_i \in Q} a_i / L$取最大值的子集合。

4． 写出 Huffman 编码的伪代码，并编程实现。

5． 已知n种货币$c_1, c_2, \cdots, c_n$和有关兑换率的$n \times n$表R，其中$R[i, j]$是一个单位的货币c_i可以买到的货币c_j的单位数。

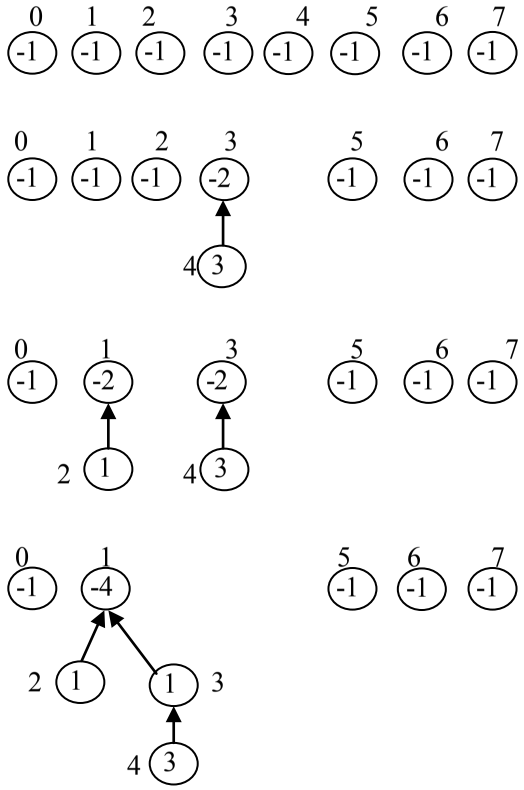
1）试设计一个算法，用以确定是否存在一货币序列$c_{i_1}, c_{i_2}, \cdots, c_{i_k}$使得：

$$R[i_1, i_2]R[i_2, i_3]\cdots R[i_k, i_1] > 1$$

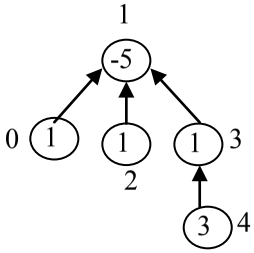
2） 设计一个算法打印出满足 1）中条件的所有序列，并分析算法的计算时间。

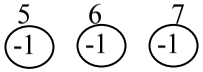
n=7, D=[4,2,4,3,4,8,3] 快速作业调度程序执行过程

作业	F[0]	F[1]	F[2]	F[3]	F[4]	F[5]	F[6]	F[7]	
无	0	1	2	3	4	5	6	7	考虑作业 1
{1}	0	1	2	3	3	5	6	7	考虑作业 2
{1,2}	0	1	1	3	3	5	6	7	考虑作业 3
{1,2,3}	0	1	1	1	3	5	6	7	考虑作业 4



{1,2,3,4}　0　0　1　1　3　5　6　7

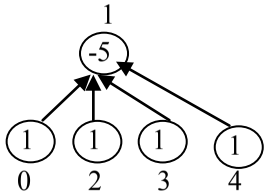


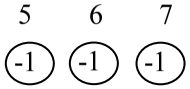


考虑作业 5

Find(min{7,D[5]})=Find(4)=1
F[1]=0,　作业 5 不被选择

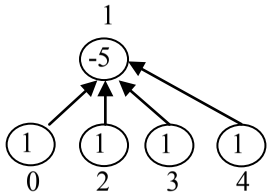
{1,2,3,4}　0,　0,　1,　1,　3　5　6　7

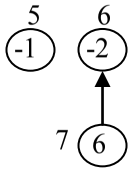




考虑作业 6

{1,2,3,4,6}　0,　0,　1,　1　3　5　6　6





考虑作业 7

Find(min{7,D[7]})=Find(3)=1
F[1]=0,　作业 7 不被选择

最优解 J={1, 2, 3, 4, 6}

第五章 动态规划算法

§1.动态规划算法的基本思想

动态规划方法是处理分段过程最优化问题的一类及其有效的方法。在实际生活中，有一类问题的活动过程可以分成若干个阶段，而且在任一阶段后的行为依赖于该阶段的状态，与该阶段之前的过程是如何达到这种状态的方式无关。这类问题的解决是多阶段的决策过程。20 世纪 50 年代，贝尔曼（Richard Bellman）等人提出了解决这类问题的“最优化原则”，从而创建了求解最优化问题的一种新的算法——动态规划算法。最优化原则指出，多阶段过程的最优决策序列具有性质：

无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。

这要求原问题计算模型的最优解需包含其（相干）子问题的一个最优解（称为***最优子结构性质***）。

动态规划算法采用最优化原则来建立递归关系式（关于求最优值的），在求解问题时有必要验证该递归关系式是否保持最优化原则。若不保持，则动态规划算法不适合求解该计算模型。在得到最优值的递归式之后，需要执行回溯以构造最优解。在使用动态规划算法自顶向下（Top-Down）求解时，每次产生的子问题并不总是新问题，有些子问题反复计算多次，动态规划算法正是利用了这种***子问题重叠性质***，对每一个子问题只计算一次，将其解保存在一个表格中，当再次要解此子问题时，只是简单地调用（用常数时间）一下已有的结果。

通常，不同的子问题个数随着输入问题的规模呈多项式增长，因此，动态规划算法通常只需要多项式时间，从而获得较高的解题效率。最优子结构性质和子问题重叠性质是计算模型采用动态规划算法求解的两个基本要素。

例 5.1.1．多段图问题

设 G=(V,E)是一个赋权有向图，其顶点集 V 被划分成 k>2 个不相交的子集 V_i: $1\le i\le k$，其中，V_1和 V_k分别只有一个顶点 s(称为源)和一个顶点 t(称为汇)，图 5-1-1 中所有的边(u,v)的始点和终点都在相邻的两个子集 V_i和 V_{i+1}中，而且 $u\in V_i$，$v\in V_{i+1}$。**多阶段图问题是：**求由 s 到 t 的最小成本路径（也叫最短路径）。

对于每一条由 s 到 t 的路径，可以把它看成在 k-2 个阶段做出的某个决策序列的相应结果：第 i 步决策就是确定 V_{i+1}中哪个顶点在这条路径上。假设

$$s,\ v_2,\ v_3,\ \cdots,\ v_{k-1},\ t$$

是一条由 s 到 t 的最短路径，再假定从源点 s(初始状态)开始，已经作出了到顶点 v_2的决策(初始决策)，则 v_2就是初始决策产生的状态。若将 v_2看成是原问题

的子问题的初始状态，这个子问题就是找一条由 v_2 到 t 的最短路径。事实上，路径 v_2，v_3，…，v_{k-1}，t 一定是 v_2 到 t 的一条最短路径。不然，设

$$v_2,\ q_3,\ \cdots,\ q_{k-1},\ t$$

是一条由 v_2 到 t 的比 v_2，v_3，…，v_{k-1}，t 更短的路径，则 s，v_2，q_3，…，q_{k-1}，t 是一条由 s 到 t 的比 s，v_2，v_3，…，v_{k-1}，t 更短的路径，与前面的假设矛盾。这说明多段图问题具有最优子结构性质。

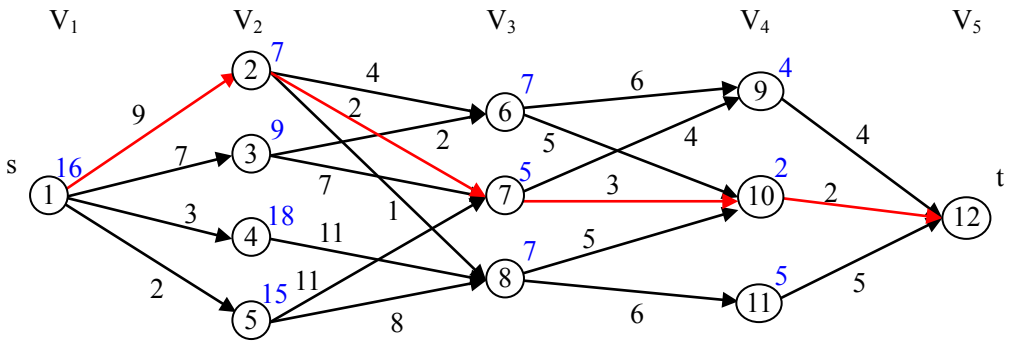


图 5-1-1 一个 5 段图

例5.1.2 0/1 背包问题

有 n 件物品，第 i 件重量和价值分别是 w_i 和 p_i，i=1，2，…，n。要将这 n 件物品的某些件装入容量为 c 的背包中，要求每件物品或整个装入或不装入，不许分割出一部分装入。0/1 背包问题就是要给出装包方法，使得装入背包的物品的总价值最大。这个问题归结为数学规划问题：

$$\max \sum_{1\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le n} w_i x_i \le c,\quad x_i \in \{0,1\}, i=1,2,\cdots,n \tag{5.1.1}$$

0/1 背包问题的上述计算模型具有最优子结构性质。事实上，若 $y_1, y_2, \cdots, y_n$ 是原问题的最优解，则 $y_2, \cdots, y_n$ 将是 0/1 背包问题的下述子问题：

$$\max \sum_{2\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{2\le i\le n} w_i x_i \le c - y_1 w_1,\quad x_i \in \{0,1\}, i=2,\cdots,n \tag{5.1.2}$$

的最优解。因为，若 $y'_2, \cdots, y'_n$ 是子问题(5.1.2)的最优解，且使得

$$\sum_{2\le i\le n} p_i y'_i > \sum_{2\le i\le n} p_i y_i \tag{5.1.3}$$

则 $y_1, y'_2, \cdots, y'_n$ 将是原问题(5.1.1)的可行解，并且使得

$$p_1 y_1 + \sum_{2 \le i \le n} p_i y'_i > \sum_{1 \le i \le n} p_i y_i \tag{5.1.4}$$

这与 $y_1, y_2, \cdots, y_n$ 是最优解相悖。

例5.1.3　矩阵连乘问题

给定 n 个数字矩阵 A_1，A_2，…，A_n，其中 A_i 与 A_{i+1} 是可乘的，i=1,2,…,n-1. 求矩阵连乘 $A_1A_2\cdots A_n$ 的加括号方法，使得所用的数值乘法运算次数最少。

考察两个矩阵相成的情形：C=AB。如果矩阵 A，B 分别是 p×r 和 r×q 矩阵，则它们的乘积 C 将是 p×q 矩阵，其(i, j)元素为

$$c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \tag{5.1.5}$$

i=1,…,p, j=1,…,q, 因而 AB 所用的数乘次数是 prq。如果有至少 3 个以上的矩阵连乘，则涉及到乘积次序问题，即加括号方法。例如 3 个矩阵连乘的加括号方法有两种：$(A_1A_2)A_3$ 和 $A_1(A_2A_3)$。设 A_1，A_2，A_3 分别是 $p_0 \times p_1$，$p_1 \times p_2$，$p_2 \times p_3$ 矩阵，则以上两种乘法次序所用的数乘次数分别为：

$p_0p_1p_2+p_0p_2p_3$ 和 $p_0p_1p_3+p_1p_2p_3$。

如果 p_0=10, p_1=100, p_2=5, p_3=50, 则两种乘法所用的数乘次数分别为：7500 和 75000。可见，由于加括号的方法不同，使得连乘所用的数乘次数有很大差别。对于 n 个矩阵的连乘积，令 P(n)记连乘积的完全加括号数，则有如下递归关系

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases} \tag{5.1.6}$$

由此不难算出 P=C(n-1)，其中 C 表示 Catalan 函数：

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n / n^{3/2}) \tag{5.1.7}$$

也就是说，P(n)是随 n 指数增长的，所以，我们不能希望列举所有可能次序的连乘积，从中找到具有最少数乘次数的连乘积算法。事实上，矩阵连乘积问题问题的上述计算模型具有最优子结构性质，我们可以采用动态规划的方法，在多项式时间内找到最优的连乘积次序。

用 A[i..j]表示连乘积 $A_iA_{i+1}\cdots A_j$。分析计算 A[1..n]的一个最优加括号方法。设这个加括号方法在矩阵 A_k 和 A_{k+1} 之间将矩阵链分开，即 $(A_1A_2\cdots A_k)(A_{k+1}\cdots A_n)$，$1 \le k < n$。依次，我们先分别计算 A[1..k]和 A[k+1..n]，然后将计算的结果相乘得到 A[1..n]。可见，A[1..n]的一个最优加括号方法所包含的矩阵子链 A[1..k]和 A[k+1..n]也必定采用了最优加括号方法。也就是说，矩阵连乘问题调的上述

计算模型具有最优子结构性质。

如上三个例子都具有最优子结构性质，这个性质决定了解决此类问题的基本思路是：

首先确定问题具有最优子结构性质的计算模型，然后寻找原问题的最优值和子问题的最优值之间的递推关系（自上而下），最后自底向上递归地构造出最优解（自下向上）。

最优子结构性质是最优化原理得以采用的先决条件。一般说来，分阶段选择策略确定最优解的问题往往会形成一个决策序列。Bellman 认为，利用最优化原理以及所获得的递推关系式去求解最优决策序列，可以使枚举数量急剧下降。

这里有一个问题值得注意：最优子结构性质提示我们使用最优化原则产生的算法是递归算法，但只是简单地使用递归算法可能会增加时间与空间开销。例如，下面的用递归式直接计算矩阵连乘积 A[1..n]的算法 RecurMatrixChain 的时间复杂度将是$\Omega(2^n)$：

程序 5-1-1 计算矩阵连乘的递归算法

```
  int RecurMatrixChain(int i, int j)
  {
     if (i==j) return 0;
     int u=RecurMatrixChain(i, i)
        +RecurMatrixChain(i+1,j)
        +p[i-1]*p[i]*p[j];
    s[i][j]=i;
    for(int k=i+1; k<j; k++){
      int t=RecurMatrixChain(i,k)
          +RecurMatrixChain(k+1,j)
          +p[i-1]*p[k]*p[j];
      if (t<u) {
        u=t;
        s[i][j]=k;}
      }
    return u;
}
```

如果用 T(n)表示该算法的计算 A[1..n]的时间，则有如下递归关系式：

$$T(n) \ge \begin{cases} O(1) & n=1 \\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & n>1 \end{cases}$$

$$T(n) \ge 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)=n+2\sum_{k=1}^{n-1}T(k),$$

可用数学归纳法直接证明：$T(n) \ge 2^{n-1}=\Omega(2^n)$，这显然不是我们所期望的。

注意到，在用递归算法自上向下求解具有最优子结构计算模型时，每次产生的子问题并不总是新问题，有些问题被反复计算多次。如果对每一个问题只解一次，而后将其解保存在一个表格中，当再次需要解此问题时，只是简单地用常数时间查看一下结果，则可以节省大量的时间。在矩阵的连乘积问题中，若用m[i][j]表示由第 i 个矩阵到第 j 个矩阵的连乘积 A[i..j]所用的最少数乘次数，则计算 m[1][n]时共有$\Theta(n^2)$个子问题。这是因为，对于$1 \le i < j \le n$，不同的有序对(i, j)对应于不同的子问题，不同子问题有$\binom{n}{2}+n=\Theta(n^2)$个。下面将会看到，用动态规划方法解此问题时，可在多项式时间内找到矩阵连乘积的最优加括号方法。

程序 5-1-2 求矩阵连乘最优次序的动态规划算法

```
void MatrixChain(int p, int n, int * *m, int * *s)
{
  for (int i=1; i<=n; i++) m[i][i]=0;
  for (int r=2; r<=n; r++){
    for (int i=1; i<=n-r+1; i++){
      int j=i+r-1; \\ r 是跨度
      m[i][j]= m[i+1][j]+p[i-1]*p[i]*p[j];
      s[i][j]=i;
      for (int k=i+1; k<j; k++){
        int t= m[i][k]+ m[k+1][j]+p[i-1]*p[k]*p[j];
        if (t< m[i][j]) {
           m[i][j]=t;
           s[i][j]=k; }
        }
      }
```

```
        }
    }
}
```

算法 MatrixChain 的主要计算量取决于程序中对 r, i 和 k 的三重循环，循环体内的计算量为 O(1)，三重循环的总次数是 $O(n^3)$，所以，算法的计算时间复杂性为 $O(n^3)$。

例 5.1.4 求以下 6 个矩阵连乘积最少数乘运算次数及所采用乘法次序。

$$A_1:30\times35;\ A_2:35\times15;\ A_3:15\times5;\ A_4:5\times10;\ A_5:10\times20;\ A_6:20\times25\quad,$$

即

P=[30, 35, 15, 5, 10, 20, 25]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000\\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125\\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375\end{cases}$$
$$=7125$$

一般的计算 $m[i][j]$ 以及 $s[i][j]$ 的过程如下图所示：

$m[i][j]$

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

跨度增加方向

$s[i][j]$

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

注意，上述算法只是明确给出了矩阵最优连乘次序所用的数乘次数 m[1][n]，并未明确给出最优连乘次序，即完全加括号方法。但是以 s[i][j]为元素的 2 维数组却给出了足够的信息。事实上，s[i][j]=k 说明，计算连乘积 A[i..j]的最佳方式应该在矩阵 A_k 和 A_{k+1} 之间断开，即最优加括号方式为(A[i..k])(A[k+1..j])。下面的算法 Traceback 按算法 MatrixChain 计算出的断点信息 s 指示的加括号方式输出计算 A[i..j]的最优次序。

程序 5-1-3　根据最优值算法构造最优解

```
Void Traceback(int i, int j, int * * s)
{
   if (i==j) return;
   Traceback(i, s[i][j], s);
   Traceback(s[i][j]+1, j, s);
   cout << "Multiply A" << i << ","  << s[i][j];
   cout << "and A" << (s[i][j] +1) << ","  << j << endl;
}
```

总结上面解矩阵连乘积问题，我们可以归纳出使用动态规划算法的基本步骤：

1.　分析最优解的结构

在这一步中，选定要解决的问题的一个模型，其具有最小子结构性质，这是选择动态规划算法的基础。

2.　建立递归关系

第一步的结构分析已经为建立递归关系奠定了基础。这一步的主要任务是递归地定义最优值，并建立该问题与其子问题最优值之间的递归关系。例如在矩阵连乘积问题中，递归关系为

$$m[i][j]=\begin{cases} 0 & i=j \\ \min\limits_{i\le k\le j}\{\ \mathrm{m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j]}\} & i<j \end{cases}$$

在 0/1 背包问题中，令 $g_j(X)$ 表示 0/1 背包问题：物品编号是 j, j+1, …, n，背包容量是 X 的最优值，则最优值递归关系是

$$g_j(X)=\max\left\{g_{j+1}(X), g_{j+1}(X-w_j)+p_j\right\} \qquad (5.1.8)$$

在多段图问题中的递归关系是

$$COST(i,j)=\min_{l\in V_{i+1},(j,l)\in E}\left\{c(j,l)+COST(i+1,l)\right\} \qquad (5.1.9)$$

这里 j 表示取 V_i 中的顶点 j。

3.　计算最优值

依据递归关系式可以自底向上的方式（或自顶向下的方式－备忘录方法）进行计算，在计算过程中保存已经解决的子问题答案。每个子问题只计算一次，而在后面需要时只要简单查一下，从而避免大量的重复计算。

4. 构造最优解

依据求最优值时记录的信息，构造出最优解（如矩阵连乘积）。

上述归纳的 4 个步骤是一个整体，必要时才分步完成，多数情况下是统一来完成的。

§2. 多段图问题

多段图是一种简单而经典的使用动态规划算法的模型，它既能有效地反映动态规划算法基本特征，而且在实际问题中有较多的应用。

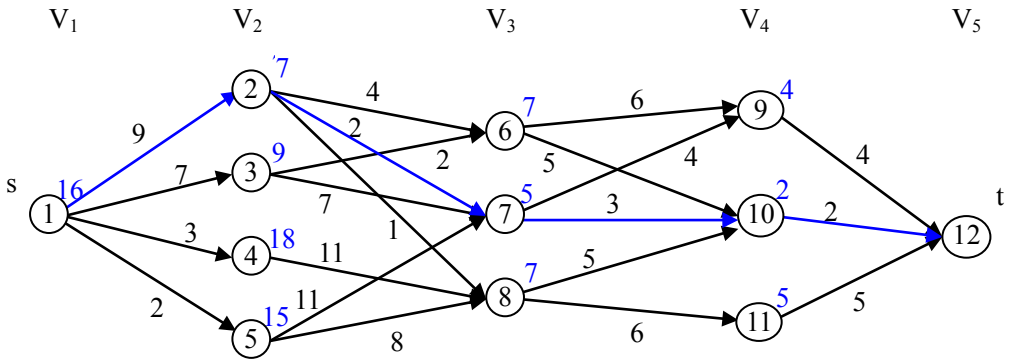


图 5-2-1 多段图的动态规划算法执行过程

最优值递推关系式为

$$COST(i,j)=\min_{l\in V_{i+1},(j,l)\in E}\{c(j,l)+COST(i+1,l)\} \qquad (5.2.1)$$

其中，$COST(i,j)$ 代表 i 段中的顶点 j 到汇点 t 的最短路径的长度。根据(5.2.1)式，我们可以由后向前逐步确定各阶段中的顶点到汇顶点 t 的最短路径。对于如上的 5 阶段网络图，蓝色字体标出了各顶点到汇顶点 t 的最短距离。

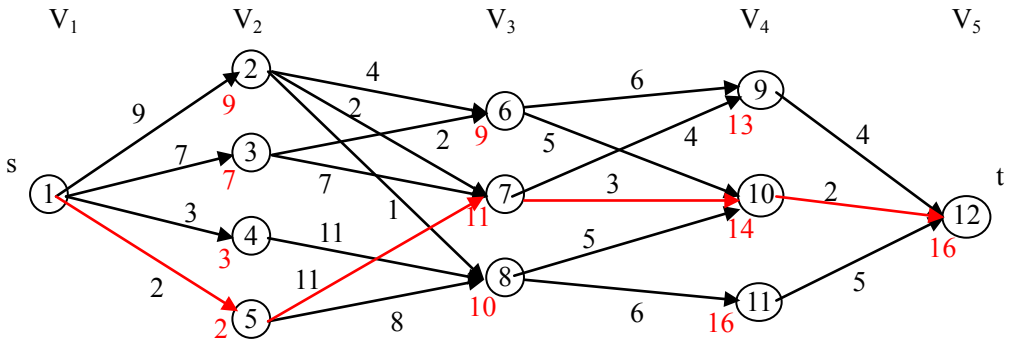


图 5-2-2 由前向后的处理方法（备忘录方法）

事实上，我们也可以由前向后逐步确定由源顶点 s 到各阶段中顶点的最短路径，此时的递归关系为

$$BCOST(i,j)=\min_{l\in V_{i-1},(l,j)\in E}\{BCOST(i-1,l)+c(l,j)\} \qquad (5.2.2)$$

其中，$BCOST(i,j)$代表源节点s到i段中的顶点j的最短路径的长度。上图中的红色字体标出了由源节点 s 到各顶点的最短距离。

程序 5-2-2　多段图的动态规划算法伪代码

```
    proc MultiGraph(E, k, n, P) //有 n 个顶点的 k 段图 G（按段序统
       // 一编号)，E 是边集，c(i, j)是边(i, j)的成本，P[1..k]是最小
       // 成本路径。
1      float COST[1..n]; integer D[1..n-1]，P[1..k], r, j, n;
2      COST[n]:=0;
3      for j from n-1 by -1 to 1 do
4        设 r 是这样一个顶点，(j, r)∈E 且使得 c(j, r)+COST[r]取最小
       值
5        COST[j]:= c(j, r)+COST[r];
6        D[j]:=r;  //指出 j 的后继
7      end{for}
8      P[1]:=1; P[k]:=n; //最短路径的起点为 s，终点为 t
9      for j from 2 to k-1 do
10       P[j]:=D[P[j-1]];  //最短路径上的第 j 个节点是第 j-1 节点的后
       继
11     end{for}
12   end{MultiGraph}
```

这里，D[j]将由 j 到汇顶点 t 的最短路径上 j 后面的顶点记录下来，P[j]则记录由源节点 s 到汇节点 t 的最短路径上处于第 j 阶段中的顶点。语句 10 是根据数组 D 中的信息逐段寻找到由源顶点 s 到汇顶点 t 的最短路径上的各个顶点。如果用邻接链表表示图 G，则语句 4 中 r 的确定可以在与 $d^+(j)$成正比的时间内完成。因此，语句 3－7 的 for 循环所需的时间是$\Theta(n+|E|)$。循环体 9－11 所需时间是$\Theta(k)\le n$，因而算法 MultiGraph 的时间复杂度是$\Theta(n+|E|)$。下面给出的备忘录算法的时间复杂度也是如此。

程序 5-2-3　多段图的备忘录算法伪代码

```
MemorizedMultiGraph(E, k, n, P) //有 n 个顶点的 k 段图 G（按段序统一
```

```
        //编号)，E 是边集，c(i, j)是边(i, j)的成本，P[1..k]是最小成本
        //路径。
1       real BCOST[1..n]; integer D[1..n-1], P[1..k], r, j, n;
2       BCOST[1]:=0;
3       for j from 2 to n do
4          设 r 是这样一个顶点，(r, j)∈E 且使得 BCOST(r)+ c(r, j)取最
           小值
5          BCOST[j]:= BCOST(r)+ c(r, j);
6          [j]:=r;   // 记录节点 j 的前继
7       end{for}
8       P[1]:=1; P[k]:=n;  // 最短路径的起点为节点 s，终点为节点 t
9       for j from k-1 by -1 to 2 do
10         P[j]:=D[P[j+1]];  // 最短路径上第 j 个节点是第 j+1 个节点
                                 的前继
11      end{for}
12   end{MemorizedMultiGraph}
```

§3. 0/1 背包问题

本节将使用动态规划的由前向后处理方法（也称备忘录算法）处理 0/1 背包问题：

$$\max \sum_{1\le i\le n} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le n} w_i x_i \le c, \quad x_i \in \{0,1\}, i = 1,2,\cdots,n \tag{5.3.1}$$

通过作出变量 x 的取值序列 $x_1, x_2, \cdots, x_n$ 来给出最优解。这个取值序列的对应决策序列是 $\underrightarrow{x_n, x_{n-1}, \cdots, x_1}$。在对 x_n 作出决策之后，问题处于下列两种状态之一：背包剩余的容量仍为 c，此时未产生任何效益；背包的剩余容量为 $c - w_n$，此时的效益值增长了 p_n。因而

$$m[n][c] = \max\{m[n-1][c], m[n-1][c-w_n] + p_n\} \tag{5.3.2}$$

一般地，如果记 0/1 背包子问题：

$$\max \sum_{1\le i\le k} p_i x_i$$

$$\text{s.t.} \sum_{1\le i\le k} w_i x_i \le X, \quad x_i \in \{0,1\}, i=1,2,\cdots,k \tag{5.3.3}$$

最优解的值为$m[k][X]$，则有

$$m[k][X]=\max\{m[k-1][X], m[k-1][X-w_k]+p_k\} \tag{5.3.4}$$

这里$0\le X\le c$，而且$k=1,2,\cdots,n$。为使(5.3.4)式能够有效地递推下去，需要延拓 k 和 X 的取值范围：$k=0,1,2,\cdots n; X\in(-\infty,+\infty)$；补充定义：

$$m[0][X]=\begin{cases}-\infty & if \quad X<0;\\ 0 & if \quad X\ge 0;\end{cases} \quad m[k][<0]=-\infty, k=1,2,\cdots n \tag{5.3.5}$$

事实上，我们的主要目的是计算出$m[n][c]$，根据(5.3.4)式，我们可能需要计算$m[n-1][c-w_n]$，而为了计算$m[n-1][c-w_n]$，可能需要计算$m[n-2][c-w_{n-1}-w_n]$，如此等等，在递推公式(5.3.4)中用到的 X 值可能为负数。另外，如果将$m[k][X]$看作 X 的函数，则是一个分段函数，而且当$X\ge\sum_{1\le i\le k} w_i$时取常值。所以将 X 的取值范围延拓至全体实数。

例 5.3.1　$n=3,(w_1,w_2,w_3)=(2,3,4),(p_1,p_2,p_3)=(1,2,5),c=6$。

$$m[0][X]=\begin{cases}-\infty & X<0\\ 0 & X\ge 0\end{cases},$$

$$m[1][X]=\begin{cases}-\infty & X<0\\ 0 & 0\le X<2\\ \max\{0,0+1\}=1 & X\ge 2\end{cases} \qquad m[0][X-w_1]+p_1$$

$$m[2][X]=\begin{cases}-\infty & X<0\\ 0 & 0\le X<2\\ 1 & 2\le X<3\\ \max\{1,0+2\}=2 & 3\le X<5\\ \max\{1,1+2\}=3 & X\ge 5\end{cases} \qquad m[1][X-w_2]+p_2$$

$$m[2][X]=\begin{cases}-\infty & X<0\\ 0 & 0\le X<2\\ 1 & 2\le X<3\\ 2 & 3\le X<4\\ \max\{2,0+5\}=5 & 4\le X<5\\ \max\{3,0+5\}=5 & 5\le X<6\\ \max\{3,1+5\}=6 & 6\le X<7\\ \max\{3,2+5\}=7 & 7\le X<9\\ \max\{2,3+5\}=8 & X\ge 9\end{cases}\qquad m[2][X-w_3]+p_3$$

由递推式(5.3.5)，$m[k-1][X]\le m[k][X]$，而且当 $X<w_k$ 时等式成立，这一事实可以写成

$$m[k][X]=\begin{cases}m[k-1][X], & X<w_k\\ \max\{m[k-1][X],m[k-1][X-w_k]+p_k\}, & X\ge w_k\end{cases}$$
$$k=1,2,\cdots,n$$

上述诸函数 $m[k][X]$ 具有如下性质：

1. 每个阶梯函数 $m[k][X]$ 都是由其跳跃点偶 $(b,m[k][b])$ 决定的。如果有 $r+1$ 个跳跃点：$b_0<b_1<\cdots<b_r$，各点的函数值分别是 $v_0,v_1,\cdots,v_r$，则

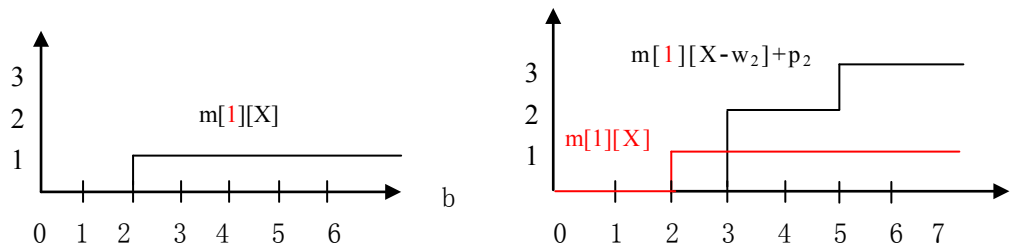
$$m[k][X]=v_i \quad if \quad b_i\le X<b_{i+1},\quad i=0,1,\cdots,r \tag{5.3.6}$$

这里约定 $b_{r+1}=+\infty$。

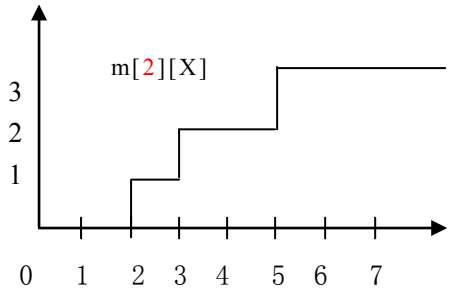
2. 令 S^k 是阶梯函数 $m[k][X]$ 的跳跃点偶的集合，则阶梯函数 $m[k-1][X-w_k]+p_k$ 的跳跃点偶之集去掉点偶(0, 0)后，恰是集合

$$S_a^i=\{(b,v)\,|\,(b-w_k,v-p_k)\in S^{k-1}\} \tag{5.3.7}$$

这是因为函数 $m[k-1][X-w_k]+p_k$ 的图象恰是由函数 $m[k-1][X]$ 的图象先向右平移 w_k，然后再向上移动 p_k 而得。下图给出了前面例子的阶梯函数图像示意：



$S^1=\{(0,0),\ (2,1)\}$　　$(w_2,\ p_2)=(3,\ 2)$　　$S^2_a=\{\ (3,2),\ (5,3)\}$



$S^2=\{(0,0),\ (2,1),\ (3,2),\ (5,3)\}$

根据递推公式(5.3.4)，S^k是从$S^{k-1}\cup S^k_a$中去掉那些点偶(b_i,v_i)：在$S^{k-1}\cup S^k_a$中存在点偶(b_k,v_k)使得$b_i\ge b_k$而且$v_i\le v_k$，此时我们说(b_k,v_k)淹没(b_i,v_i)。前面例子的诸S^k计算如下：

$$
\begin{aligned}
&S^0=\{(0,0)\};\quad (w_1,p_1)=(2,1), S^1_a=(w_1,p_1)+S^0=\{(2,1)\}\\
&S^1=\{(0,0),(2,1)\};\quad (w_2,p_2)=(3,2), S^2_a=(w_2,p_2)+S^1=\{(3,2),(5,3)\}\\
&S^2=\{(0,0),(2,1),(3,2),(5,3)\};\quad (w_3,p_3)=(4,5),\\
&\qquad\qquad S^3_a=(w_3,p_3)+S^2=\{(4,5),(6,6),(7,7),(9,8)\}\\
&S^3=\{(0,0),(2,1),(3,2),(4,5),(6,6),(7,7),(9,8)\}.
\end{aligned}
$$

在由S^2、S^3_a向S^3的归并过程中，点偶(5, 3)被(4, 5)淹没。

3．在处理实际问题时，由于要求$X\le c$，在计算$S^k_a=(w_k,p_k)+S^{k-1}$时应该去掉那些使得$b>c$的跳跃点偶(b,v)。根据前面提到的淹没规则，如果将S^k中的元素按照第一个分量的递增次序排列，则第二个分量也呈递增的次序，而且S^n的最后一个元素，设为(b,v)，的v值即是0/1背包问题(5.3.1)的最优值$m[n][c]$。

4．最优解的获得可以通过检查诸S^k来确定。设(b_{k_n},v_{k_n})是S^n的最后一个元素。若$(b_{k_n},v_{k_n})\in S^{n-1}$，则$x_n=0$．因为此时函数$m[n][X]$和函数$m[n-1][X]$在X≤c范围内的最大值一致，表明0/1背包问题(5.3.1)与其子问题

$$
\max\sum_{1\le i\le n-1}p_ix_i
$$

$$
\text{s.t.}\ \sum_{1\le i\le n-1}w_ix_i\le c,\quad x_i\in\{0,1\},i=1,2,\cdots,n-1 \qquad (5.3.8)
$$

有相同的最优值。若$(b_{k_n}, v_{k_n}) \notin S^{n-1}$，则$x_n = 1$。因为，此时函数$m[n][X]$在 X≤c 范围内的最大值是函数$m[n-1][X]$的最大值加$p_n$，相应地，0/1 背包问题(5.3.1)的最优值是其子问题

$$\max \sum_{1 \le i \le n-1} p_i x_i$$

$$\text{s.t.} \sum_{1 \le i \le n-1} w_i x_i \le c - w_n, \quad x_i \in \{0,1\}, i = 1,2,\cdots,n-1 \tag{5.3.9}$$

的最优值加p_n。注意到此时跳跃点偶一定具有形式

$$(b_{k_n}, v_{k_n}) = (w_n, p_n) + (b_{k_{n-1}}, v_{k_{n-1}}) \tag{5.3.10}$$

其中$(b_{k_{n-1}}, v_{k_{n-1}}) \in S^{n-1}$。于是，我们可以依$(b_{k_{n-1}}, v_{k_{n-1}}) \in S^{n-2}$与否而决定$x_{n-1}$取 0 或 1。如此等等，逐一确定$x_n, x_{n-1}, \cdots, x_1$的值。

在上述例子中，整理后的诸S^k为：

$$S^0 = \{(0,0)\};$$
$$S^1 = \{(0,0),(2,1)\};$$
$$S^2 = \{(0,0),(2,1),(3,2),(5,3)\};$$
$$S^3 = \{(0,0),(2,1),(3,2),(4,5),(6,6)\}.$$

(6, 6)是S^3的最后一个跳跃点偶，所以该 0/1 问题的最优值是 6。这个点偶不在S^2中，因而$x_3 = 1$；又 $(6,6)-(4,5)=(2,1) \in S^2$，据此判断$x_2$的取值。因为$(2,1) \in S^1$，所以$x_2 = 0$；最后由$(2,1) \notin S^0$知$x_1 = 1$，所以最优解为$(1,0,1)$.

为实现上述算法，可以用两个一维数组 B 和 V 来存放所有的跳跃点偶(b, v)，跳跃点偶集$S^0, S^1, \cdots, S^{n-1}$互相邻接地存放（将诸$S^i$中的全部元素统一编号，从$S^i$生成$S^{i+1}$时，元素也是递升地产生，而且使用淹没规则），并用指针 F(k)来指示集合 S^k，即 S^k的第一个元素的位置，而 F(n)-1 是S^{n-1}中最后一个元素的位置（这里不存放 S^n是由于我们只需要它的最后一个元素，而这个元素或者是 S^{n-1}的最后一个元素，此时 S^{n-1}与 S^n有相同的最后元素；或者具有形式(5.3.10)，而且v_{k_n}是满足$b_{k_{n-1}} + w_n \le c$的最大值。所以，确定 S^n的最后元素不必将 S^n的元素全部求出来。而且确定最优解的其它变量$x_{n-1}, \cdots, x_1$时也不使用数据 S^n）。因为产生 S^k仅使

用信息 S^{k-1} 和 (w_k, p_k)，所以不必存储 S_a^k。根据以上分析，我们不难给出求解 0/1 背包问题模型的动态规划算法。

程序 5-3-1　0/1 背包问题动态规划算法伪代码

```
DKnapsack(w, p, c, n, m) //数组 w, p 中的元素分别代表各件物品的
 //重量和价值，n 是物品个数，c 代表背包容量，m 代表跳跃点个数。
  float p[1..n], w[1..n], B[1..m], V[1..m], ww, pp, c;
  integer F[0..n], l, h, i, j, next;
  F[0]:=1; B[1]:=0; V[1]:=0;
  s:=1; t:=1; //S^0的首和尾
  F[1]:=2; next:=2; // B、V 中的第一个空位
  for i to n-1 do
     k:=s; //k 指示扫描 S^(i-1)的游标，它的起始位置为 s，终止位置为 t
     u:=最大指标 r，使得 s≤ r ≤ t, 而且 B[r]+w_i ≤ c;
     for j from s to u do
       (ww, pp):=( B[j]+w_i, V[j]+p_i); //S^i_a 中的下一个元素
        while k ≤ t && B[k] < ww do //从 S^(i-1)中取来元素归并
           B[next]:=B[k]; V[next]:=V[k];
           next:=next+1; k:=k+1;
        end{while}
        if k ≤ t && B[k]=ww then
          pp:=max(V[k], pp);
          k:=k+1;
        end{if}
        if pp > V[next-1] then
           (B[next], V[next]):=(ww, pp);
           next:=next+1;
        end{if}
         while k ≤ t && V[k] < V[next-1] do //清除
           k:=k+1;
         end{while}
      end{for}
      while k ≤ t do // 将 S^(i-1)剩余的元素并入 S^i
        (B[next], V[next]):= (B[k], V[k]);
        next:=next+1; k:=k+1;
```

```
        end{while}
      s:=t+1; t:=next-1; F[i+1]:=next; //为 S^{i+1}赋初值
    end{for}
    traceparts // 逐一确定 x_n,x_{n-1},…,x_1
  end{DKnapsack}
```

算法 DKnapsack 的主要工作是产生诸 S^i。在 i > 0 的情况下，每个 S^i 由 S^{i-1} 和 S^i_a 归并而成，并且 $|S^i_a| \le |S^{i-1}|$。因此 $|S^i| \le 2|S^{i-1}|$，在最坏情况下，没有序偶会被清除，所以

$$\sum_{1\le i\le n-1} |S^i| = \sum_{1\le i\le n} 2^{i-1} = 2^n - 1$$

由此知，算法 DKnapsack 的空间复杂度是 $O(2^n)$。由 S^{i-1} 生成 S^i 需要 $\Theta(|S^{i-1}|)$ 的时间，因此，计算 $S^0, S^1, \cdots, S^{n-1}$ 总共需要的时间为

$$\sum_{1\le i\le n-1} |S^{i-1}| \le \sum_{1\le i\le n-1} 2^{i-1} = 2^{n-1} - 1$$

算法 DKnapsack 的时间复杂度为 $O(2^n)$。

如果物品的重量 w_i 和所产生的效益值 p_i 都是整数，那么，S^i 中的元素 (b,v) 的分量 b 和 v 也都是整数，且 $b \le c, v \le \sum_{1\le k\le i} p_k$。又 S^i 中不同的元素对应的分量也都是不同的，故

$$|S^i| \le 1 + \sum_{1\le k\le i} p_k$$

$$|S^i| \le 1 + \min\left\{\sum_{1\le k\le i} w_k, c\right\}$$

此时，算法 DKnapsack 的时间复杂度为 $O(\min\left\{2^n, nc, n\sum_{i=1}^{n} p_k\right\})$。

附：从组合的角度来理解 0/1 背包问题的动态规划算法

先假定背包的容量是充分大的，考虑有 k 件物品往背包里装时，背包中物品总重量的各种可能出现的情况：

$k=0$: $W_0 = \{0\}$

$k=1$: $W_1 = \{0, w_1\}$

$k=2$: $W_2 = \{0, w_1, w_2, w_1+w_2\}$

$k=3$: $W_3 = \{0, w_1, w_2, w_1+w_2, w_3, w_1+w_3, w_2+w_3, w_1+w_2+w_3\}$

一般地，

$$W_k = \left\{ \sum_{i=1}^{k} x_i w_i \mid x_i \in \{0,1\} \right\} \tag{5.3.11}$$

而

$$W_{k+1} = W_k \cup \left(w_{k+1} + W_k \right), \qquad k = 1,2,\cdots,n-1 \tag{5.3.12}$$

递推关系式(5.3.10)正是我们计算诸S^k和S_a^k的依据，这只需将各种“重量”情况所对应的总价值带上，即产生元素对(b,v)，就能获得诸S^k和S_a^k。如上面的例子：$n=3,(w_1,w_2,w_3)=(2,3,4),(p_1,p_2,p_3)=(1,2,5),c=6$

$S^0=\{(0,0)\}$；　$(w_1,p_1)=(2,1)$,

$S_a^1=(w_1,p_1)+S^0=\{(2,1)\}$

$S^1=S^0\cup\left((w_1,p_1)+S^0\right)=\{(0,0),(2,1)\}$；　$(w_2,p_2)=(3,2)$,

$S_a^2=(w_2,p_2)+S^1=\{(3,2),(5,3)\}$

$S^2=S^1\cup\left((w_2,p_2)+S^1\right)=\{(0,0),(2,1),(3,2),(5,3)\}$；　$(w_3,p_3)=(4,5)$,

$S_a^3=(w_3,p_3)+S^2=\{(4,5),(6,6),(7,7),(9,8)\}$

$S^3=S^2\cup\left((w_3,p_3)+S^2\right)=\{(0,0),(2,1),(3,2),(4,5),(6,6),(7,7),(9,8)\}$　去掉被淹没的(5,3)

最后将S^3截断（根据约束条件），即去掉“重量”大于背包容量的点对，就得到所需要的点对集

$$S=\{(0,0),(2,1),(3,2),(4,5),(6,6)\}$$

得到最大价值是 6，因为$(6,6)\in S^3\setminus S^2$，应该是在前面的基础上增加了$(w_3,p_3)=(4,5)$而获得的，这个“基础”就是$(6,6)-(4,5)=(2,1)\in S^2$，所以，最优解中，$x_3=1$。再考虑$(2,1)\in S^2\cap S^1$的情况，在最优解中应有$x_2=0$。最后由$(2,1)\in S^1\setminus S^0$知，在最优解中$x_1=1$。所以，$(1,0,1)$就是上面背包问题的一个最优解。

§4. 流水作业调度问题

问题描述：已知 n 个作业{1，2，．．．，n}要在由两台机器 M_1 和 M_2 组成的流水线上完成加工。每个作业加工的顺序都是先在 M_1 上加工，然后在 M_2 上加工。M_1 和 M_2 加工作业 i 所需的时间分别为 a_i 和 b_i，$1\le i\le n$。流水作业调度问题要

求确定这 n 个作业的最优加工次序，使得从第一个作业在机器 M_1 上开始加工，到最后一个作业在机器 M_2 上加工完成所需的时间最少。

记 N={1, 2, . . . , n}，S 为 N 的子集合。一般情况下，当机器 M_1 开始加工 S 中的作业时，机器 M_2 可能正在加工 S 以外的作业，要等待时间 t 后才可用来加工 S 中的作业。将这种情况下流水线完成 S 中的作业所需的最短时间记为 T(S, t)，则流水作业调度问题的最优值即是 T(N, 0)。

流水作业调度问题的上述计算模型具有最优子结构性质。事实上，设 π 是 n 个流水作业的一个最优调度（实际上是作业的一个排序），其所需要的加工时间为 $a_{\pi(1)}+T'$，其中，T' 是在机器 M_2 的等待时间为 $b_{\pi(1)}$ 时，调度 π 安排作业 $\pi(2),\cdots,\pi(n)$ 所需的加工时间。若记 $S=N\setminus\{\pi(1)\}$，我们证明 $T'=T(S,b_{\pi(1)})$。

首先由 T' 的定义知 $T'\geq T(S,b_{\pi(1)})$。如果 $T'>T(S,b_{\pi(1)})$，设 π' 是作业集 S 开始在机器 M_1, 而机器 M_2 等待时间为 $b_{\pi(1)}$ 情况下的一个最优调度，则

$$\pi(1),\pi'(2),\cdots,\pi'(n)$$

是 N 的一个调度，该调度所需的时间为 $a_{\pi(1)}+T(S,b_{\pi(1)})<a_{\pi(1)}+T'$，这与 π 是 N 的最优调度矛盾。所以 $T'=T(S,b_{\pi(1)})$，说明流水作业调度问题的上述计算模型具有最优子结构性质。

可以如下建立流水作业调度问题的最优值递推关系式。容易看出

$$T(N,0)=\min_{1\leq i\leq n}\{a_i+T(N\setminus\{i\},b_i)\} \tag{5.4.1}$$

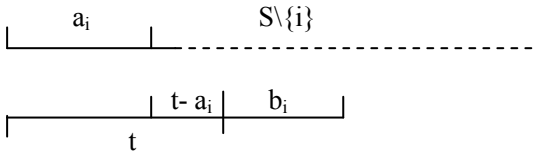
推广到一般情形：

$$T(S,t)=\min_{i\in S}\{a_i+T(S\setminus\{i\},b_i+\max\{t-a_i,0\})\} \tag{5.4.2}$$

其中，$\max\{t-a_i,0\}$ 这一项是由于机器 M_2 必须在 $\max\{t,a_i\}$ 时间之后才能开始加工 S 中的作业。因此，在机器 M_1 完成作业 i 之后，机器 M_2 还需等待

$$b_i+\max\{t,a_i\}-a_i=b_i+\max\{t-a_i,0\}$$

时间后才能开始加工 S\{i}中的作业。



按照递推关系(5.4.2)，我们容易给出流水调度问题的动态规划算法，但是其时

间复杂度将是$O(2^n)$。以下我们将根据这一问题的特点，采用 Johnson 法则简化算法，使得到时间复杂度为$O(n\log n)$。

设π是机器 M_1开始加工 S 中的作业而机器 M_2的等待时间为 t 时的任一最优调度。若在这个调度中，安排在最前面的两个作业分别是i和j，即$\pi(1)=i,\pi(2)=j$。则由递推关系式(5.4.2)得

$$T(S,t)=a_i+T\big(S\setminus\{i\},b_i+\max\{t-a_i,0\}\big)=a_i+a_j+T(S\setminus\{i,j\},t_{ij})$$

其中

$$\begin{aligned}t_{ij}&=b_j+\max\{b_i+\max\{t-a_i,0\}-a_j,0\}\\&=b_j+b_i+\max\{\max\{t-a_i,0\}-a_j,-b_i\}\\&=b_j+b_i+\max\{t-a_i-a_j,-a_j,-b_i\}\end{aligned}$$

注：上述推导过程中用到了关系式：

$$A+\max\{B_1,\cdots,B_l\}=\max\{A+B_1,\cdots,A+B_l\}$$

如果作业i和j满足

$$\min\{a_i,b_j\}\le\min\{a_j,b_i\}\tag{5.4.3}$$

则称作业i和j满足 Johnson 不等式；如果作业i和j不满足 Johnson 不等式，则交换作业i和j的加工次序后即满足 Johnson 不等式。在机器 M_1开始加工 S 中的作业而机器 M_2的等待时间为 t 时的调度π中，交换作业i和j的加工次序，得到作业集 S 的另一个调度π'，它所需的加工时间为

$$T'(S,t)=a_i+a_j+T(S\setminus\{i,j\},t_{ji})$$

其中，

$$t_{ji}=b_j+b_i+\max\{t-a_i-a_j,-a_i,-b_j\}$$

当作业i和j满足 Johnson 不等式(5.4.3)时，有

$$\max\{-a_i,-b_j\}\ge\max\{-a_j,-b_i\}$$

从而 $t_{ji}\ge t_{ij}$，可见$T'(S,t)\ge T(S,t)$。换句话说，当作业i和j不满足 Johnson 不等式时，交换它们的加工次序后，作业i和j将满足 Johnson 不等式，而且不会

增加加工时间。

由此可知，对于流水作业调度问题，必然存在一个最优调度π，使得作业$\pi(i)$和$\pi(i+1)$满足 Johnson 不等式：

$$\min\{a_{\pi(i)},b_{\pi(i+1)}\}\le\min\{a_{\pi(i+1)},b_{\pi(i)}\},\quad 1\le i\le n-1$$

一般地，可以证明，上述不等式与不等式

$$\min\{a_{\pi(i)},b_{\pi(j)}\}\le\min\{a_{\pi(j)},b_{\pi(i)}\},\quad 1\le i<j\le n \qquad (5.4.4)$$

等价。以上分析使我们不难理解关于流水作业调度问题的以下 Johnson 算法：

(1) 令 $AB=\{i\mid a_i<b_i\}$，$BA=\{i\mid a_i\ge b_i\}$；

(2) 将 AB 中作业依 a_i 的非减次序排列；将 BA 中作业依 b_i 的非增次序排列；

(3) AB 中作业接 BA 中作业即构成满足 Johnson 法则的最优调度。

程序 5-4-1 流水作业调度问题的 Johnson 算法

```
FlowShop(a,b,n,p)
// 给作业排序，数组 p 记录作业号的一个排列
float a[1..n],b[1..n];
integer c[1..n],d[1..n],p[1..n], n,j,k;
j:=1; k:=n;
for i to n do
  if a[i]<b[i] then
    d[j]:=i; c[j]:=a[i]; j:=j+1;
  else
    d[k]:=i; c[k]:=b[i]; k:=k-1;
  end{if}
end{for}
MergeSortL(c,1,k,q);
MergeSortL(c,k+1,n,r);
j:=q[0];
for i to k do
  p[i]:=d[j];
  j:=q[j];
end{for}
j:=r[0];
```

```
  for i to n-k do
    p[n-i+1]:=d[k+j];
    j:=r[j];
  end{for}
end{FlowShop}
```

上述算法的时间主要花在排序上，因此，在最坏情况下的时间复杂度为 $O(n\log n)$。空间复杂度为 $O(n)$ 更容易看出。

§5. 最优二叉搜索树

设 $S=\{x_1,x_2,\cdots,x_n\}$ 是一个有序集合，且 $x_1<x_2<\cdots<x_n$. 表示有序集合的二叉搜索树是利用二叉树的内节点存储有序集中的元素，而且具有性质：存储于每个节点中的元素大于其左子树中任一个节点中存储的元素，小于其右子树中任意节点中存储的元素。二叉树中的叶节点是形如 (x_i,x_{i+1}) 的开区间。在二叉搜索树中搜索一个元素 x，返回的结果或是在二叉树的内节点处找到：$x=x_i$；或是在二叉树的叶节点中：$x\in(x_i,x_{i+1})$，这里约定 $x_0=-\infty,x_{n+1}=+\infty$.

现在假定在第一种情况出现(即 $x=x_i$)的概率为 b_i；在第二种情况出现，即 $x\in(x_i,x_{i+1})$ 的概率为 a_i.显然

$$a_i\ge 0,\ 1\le i\le n, b_j\ge 0, 1\le i,j\le n;\ \sum_{i=0}^{n}a_i+\sum_{j=1}^{n}b_j=1 \tag{5.5.1}$$

集合 $\{a_0,b_1,a_1,\cdots,b_n,a_n\}$ 称为有序集 S 的存取概率分布。

在一个表示 S 的二叉搜索树 T 中，设存储元素 x_i 的顶点深度为 c_i，叶顶点 (x_i,x_{i+1}) 的深度为 d_i，则

$$p=\sum_{i=1}^{n}b_i(1+c_i)+\sum_{j=0}^{n}a_jd_j \tag{5.5.2}$$

表示在二叉搜索树 T 中做一次搜索所需的平均比较次数。p 也称为二叉搜索树 T 的平均路长。最优二叉搜索树问题是:

对于有序集 S 及其存取概率分布 $\{a_0,b_1,a_1,\cdots,b_n,a_n\}$，在所有表示 S 的二叉搜索树中，找出一棵具有最小平均路长的二叉搜索树。

为了采用动态规划算法，我们首先要考虑该问题的上述计算模型是否具有最优子结构性质。二叉搜索树 T 的一棵含有顶点 $x_i,x_{i+1},\cdots,x_j$ 和叶顶点 $(x_{i-1},x_i),(x_i,x_{i+1}),\cdots,(x_j,x_{j+1})$ 的子树 $\overline{T}$ 可以看作是有序集 $\{x_i,x_{i+1},\cdots,x_j\}$ 关于全集为实数区间 (x_{i-1},x_{j+1}) 的一棵二叉搜索树（T 自身可以看作是有序集 $S=\{x_1,x_2,\cdots,x_n\}$ 关于全集为整个实数区间 $(-\infty,+\infty)$ 的二叉搜索树）。

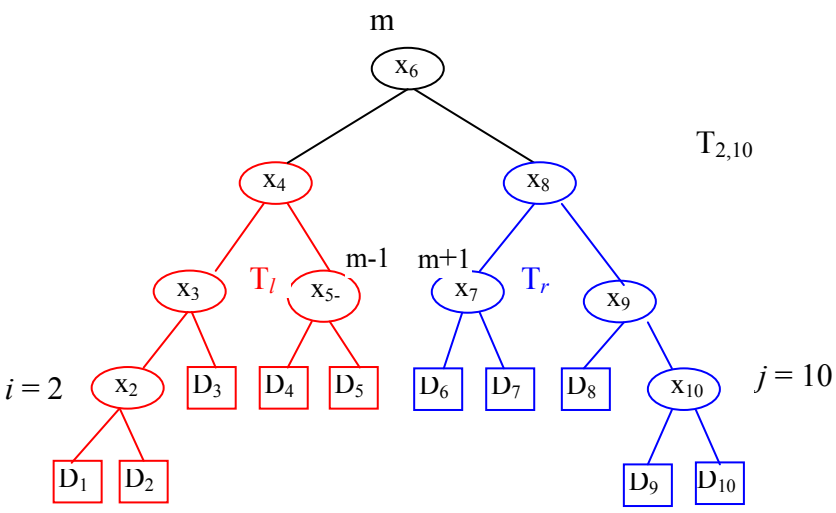


图 5-5-1 二叉搜索树的一棵子树

根据 S 的存取分布概率，x 在子树 $\overline{T}$ 的顶点处被搜索到的概率是

$$w_{ij}=\sum_{i-1\le k\le j}a_k+\sum_{i\le k\le j}b_k \tag{5.5.3}$$

于是 $\{x_i,x_{i+1},\cdots,x_j\}$ 的存储概率分布为 $\{\overline{a}_{i-1},\overline{b}_i,\cdots,\overline{b}_j,\overline{a}_j\}$，其中，$\overline{a}_h,\overline{b}_k$ 分别是下面的条件概率：

$$\overline{b}_k=b_k/w_{ij},\quad i\le k\le j;\qquad \overline{a}_h=a_h/w_{ij},\quad i-1\le h\le j \tag{5.5.4}$$

设 T_{ij} 是有序集 $\{x_i,x_{i+1},\cdots,x_j\}$ 关于存储概率分布 $\{\overline{a}_{i-1},\overline{b}_i,\cdots,\overline{b}_j,\overline{a}_j\}$ 的一棵最优二叉搜索树，其平均路长记为 p_{ij}. T_{ij} 的根顶点存储的元素是 x_m,其左子树 T_l 和右子树 T_r 的平均路长分别记为 p_l 和 p_r。由于 T_l 和 T_r 中顶点深度是它们在 T_{ij} 中的深度

减 1，如下推导

$$
\begin{aligned}
w_{ij} &= \sum_{i-1\le k\le j} a_k + \sum_{i\le k\le j} b_k \ , \\
p_{ij} &= \sum_{i-1\le k\le j} \bar{d}_k\, a_k \big/ w_{ij} + \sum_{i\le k\le j} (\bar{c}_k + 1)\, b_k \big/ w_{ij} \\
w_{ij} p_{ij} &= \sum_{i-1\le k\le j} \bar{d}_k a_k + \sum_{i\le k\le j} (\bar{c}_k + 1) b_k \\
w_{i,m-1} p_l &= \sum_{i-1\le k\le m-1} (\bar{d}_k - 1) a_k + \sum_{i\le k\le m-1} \bar{c}_k b_k \\
w_{m+1,j} p_r &= \sum_{m\le k\le j} (\bar{d}_k - 1) a_k + \sum_{m+1\le k\le j} \bar{c}_k b_k \\
w_{ij} p_{ij} &= w_{ij} + w_{i,m-1} p_l + w_{m+1,j} p_r \ (\because \bar{c}_m = 0)
\end{aligned}
\tag{5.5.5}
$$

由于 T_l 是有序集 $\{x_i, \cdots, x_{m-1}\}$ 的一棵二叉搜索树，故 $p_l \ge p_{i,m-1}$.若 $p_l > p_{i,m-1}$，则用 $T_{i,m-1}$ 替换 T_l 可得到平均路长比 T_{ij} 更小的二叉搜索树。这与 T_{ij} 是最优二叉搜索树矛盾。同理可证，T_r 也是一棵最优二叉搜索树。因此，最优二叉搜索树问题具有最优子结构性质。将(5.5.5)中的 p_l 换成 $p_{i,m-1}$， p_r 换成 $p_{m+1,j}$，则得到

$$
w_{ij} p_{ij} = w_{ij} + w_{i,m-1} p_{i,m-1} + w_{m+1,j} p_{m+1,j} \tag{5.5.6}
$$

采用上面的记号，则 n 元最优二叉搜索树问题即是求 $T_{1,n}$，其最优值为 $p_{1,n}$。由最优二叉搜索树的最优子结构性质，可以建立最优值 $p_{i,j}$ 的递推公式：

$$
w_{ij} p_{ij} = w_{ij} + \min_{i\le k\le j} \{ w_{i,k-1} p_{i,k-1} + w_{k+1,j} p_{k+1,j} \}, \quad i \le j \tag{5.5.7}
$$

初始时， $p_{i,i-1} = 0, \quad 1 \le i \le n$. 记 $w_{ij} p_{ij}$ 为 $m(i,j)$，则 $m(1,n) = w_{1,n} p_{1,n} \ = p_{1,n}$ 为所求最优值。由(5.5.7)得关于 $m(i,j)$ 的递推公式

$$
\begin{gathered}
m(i,j) = w_{ij} + \min_{i\le k\le j} \{ m(i,k-1) + m(k+1,j) \}, \quad i \le j \\
m(i,i-1) = 0, \quad i = 1,2,\cdots,n
\end{gathered}
\tag{5.5.8}
$$

据此，可以设计求解最优二叉搜索树问题的动态规划算法。

程序 5-5-1　最优二叉搜索树的动态规划算法

```
void OBSTree( int a, int b, int n, int **m, int **s, int **w)
{
    for (int i = 0; i < n; i++) {
```

```
        w[i+1][i] = a[i];
        m[i+1][i] = 0;
    }
    for (int r = 0; r < n; r++) {
        for (int i = 1; i <= n-r; i++) {
            int j = i+r;
            w[i][j] = w[i][j-1] + a[j] +b[j];
            m[i][j] = m[i+1][j];
            s[i][j] = i;
            for (int k =i + 1; k<= j; k++) {
                int t = m[i][k-1] + m[k+1][j];
                if (t < m[i][j]) {
                    m[i][j] = t;
                    s[i][j] = k; }
              }
              m[i][j]+=w[i][j];
        }
  }
}
```

算法 OBSTree 中用 s[i][j]保存最优子树的根顶点中元素。当 s[1][n] = k 时，x_k 为所求二叉搜索树的根顶点元素。其左子树为 T(1,k-1)，因此 i= s[1][k-1]即表示 T(1,k-1)的根顶点为 x_i。依次类推，容易由 s 记录的信息在 O(n)时间内构造出所求的最优二叉搜索树。算法中用到三个 2 维数组 m、s、w，故所需的空间为 $O(n^2)$. 算法的主要计算量在于计算

$$w_{ij} + \min_{i \le k \le j}\{m(i,k-1) + m(k+1,j)\}$$

对于固定的差 $r = j - i$，需要计算时间 $O(j-i+1) = O(r+1)$。因此算法所耗费的总时间为 $\sum_{r=0}^{n-1}\sum_{i=1}^{n-r} O(r+1) = O(n^3)$。

习题 五

1. 最大子段和问题：给定整数序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的子

段和的最大值：　$\max\left\{0, \max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k\right\}$

1）已知一个简单算法如下：

```
int Maxsum(int n,int a,int& besti,int& bestj)
{
   int sum = 0;
   for(int i=1;i<=n;i++){
     int suma = 0;
    for(int j=i;j<=n;j++){
       suma + = a[j];
       if(suma > sum){
         sum = suma;
         besti = i;
         bestj = j;
         }
      }
    }
    return sum;
}
```

试分析该算法的时间复杂性。

2）试用分治算法解最大子段和问题，并分析算法的时间复杂性。

3）试说明最大子段和问题具有最优子结构性质，并设计一个动态规划算法解最大子段和问题。分析算法的时间复杂度。（提示：令 $b(j)=\max_{1\le i\le j\le n}\sum_{k=i}^{j} a_k, j=1,2,\cdots,n$）

2.（双机调度问题）用两台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时所需要的时间是 a_i，若由机器 B 来处理，则所需要的时间是 b_i。现在要求每个作业只能由一台机器处理，每台机器都不能同时处理两个作业。设计一个动态规划算法，使得这两台机器处理完这 n 个作业的时间最短（从任何一台机器开工到最后一台机器停工的总的时间）。以下面的例子说明你的算法：

$n=6,(a_1,a_2,a_3,a_4,a_5,a_6)=(2,5,7,10,5,2),(b_1,b_2,b_3,b_4,b_5,b_6)=(3,8,4,11,3,4)$

3．考虑下面特殊的整数线性规划问题

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

$$\sum_{i=1}^{n} a_i x_i \le b, \quad x_i \in \{0,1,2\}, 1 \le i \le n$$

试设计一个解此问题的动态规划算法，并分析算法的时间复杂度。

4. 可靠性设计：一个系统由 n 级设备串联而成，为了增强可靠性，每级都可能并联了不止一台同样的设备。假设第 i 级设备 D_i 用了 m_i 台，该级设备的可靠性是 $g_i(m_i)$，则这个系统的可靠性是$\Pi g_i(m_i)$。一般来说 $g_i(m_i)$都是递增函数，所以每级用的设备越多系统的可靠性越高。但是设备都是有成本的，假定设备 D_i 的成本是 c_i，设计该系统允许的投资不超过 c，那么，该如何设计该系统（即各级采用多少设备）使得这个系统的可靠性最高。试设计一个动态规划算法求解可靠性设计问题。

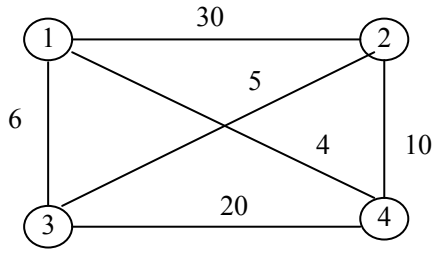
第六章　回 溯 法

§1. 回溯法的基本思想

回溯法有“通用的解题法”之称。应用回溯法求解问题时，首先应该明确问题的解空间。一个复杂问题的解决往往由多部分构成，即，一个大的解决方案可以看作是由若干个小的决策组成。很多时候它们构成一个决策序列。解决一个问题的所有可能的决策序列构成该问题的解空间。解空间中满足约束条件的决策序列称为*可行解*。一般说来，求解任何问题都有一个目标，在约束条件下使目标值达到最大（或最小）的可行解称为该问题的*最优解*。在解空间中，前 k 项决策已经取定的所有决策序列之集称为 k 定子解空间。0 定子解空间即是该问题的解空间。

例6.1.1　旅行商问题：某售货员要到若干个城市去推销商品。已知各个城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一遍，最后回到原驻地的路线，使得总的路程（或总旅费）最小。

用一个赋权图 G(V, E)来表示，节点代表城市，边表示城市之间的道路。图 6-1-1 中各边所带的权即是城市间的路程（或城市间的旅费）。



求赋权图 G 的具有最小权的 Hamilton 圈

图 6-1-1　一个赋权图

则旅行商问题归结为：**在带权图 G 中找到一条路程最短的周游路线，即权值之和最小的 Hamilton 圈。**

注：连通图 G 的一条 Hamilton 圈(或叫 Hamilton 回路)是指通过 G 的每个顶点恰好一次的圈。

如果假定城市 A 是驻地。则推销员从 A 地出发，第一站有 3 种选择：城市 B、C 或城市 D；第一站选定后，第二站有两种选择：如第一站选定 B，则第二站只能选 C、D 两者之一。当第一、第二两站都选定时，第三站只有一种选择：比如，当第一、第二两站先后选择了 B 和 C 时，第三站只能选择 D。最后推销员由城市 D 返回驻地 A。推销员所有可能的周游路线共有 3!=6 条，它们构成该旅行商问题实例的解空间，见图 6-1-2。

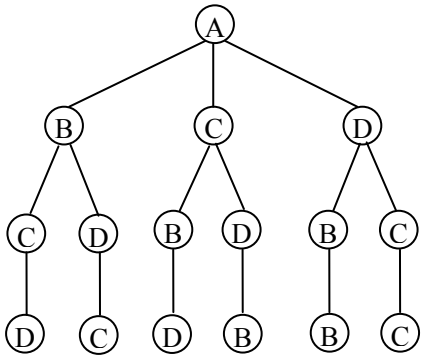


图 6-1-2 旅行商问题 6 种可能解

例 6.1.2 定和子集问题： 已知一个正实数的集合 $A=\{w_1,w_2,\cdots,w_n\}$ 和另一个正实数 M. 试求 A 的所有子集 S，使得 S 中的数之和等于 M。这个问题的解可以表示成 0/1 数组 $(x_1,x_2,\cdots,x_n)$，依据 w_i 是否属于 S， x_i 分别取值 1 或 0。故解空间中共有 2^n 个元素。它的树结构是一棵完整二叉树。

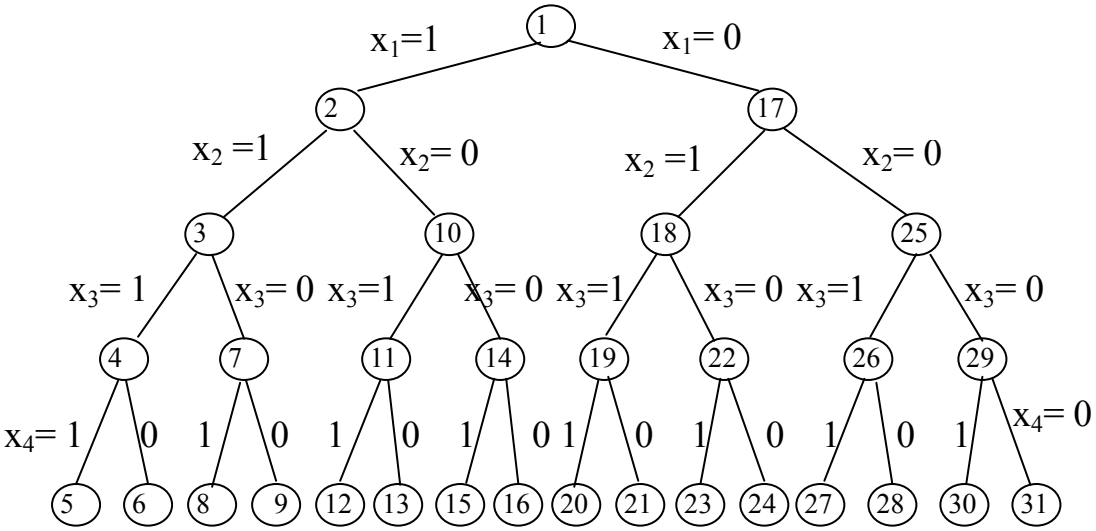


图 6-1-3 定和子集问题的解空间树

例6.1.3 4 皇后问题：在 4×4 棋盘上放置 4 个皇后，要使得每两个皇后之间都不能互相攻击，即任意两个皇后都不能放在同一行、同一列及同一对角线上。

将 4 个皇后分别给以 1 到 4 的编号，这个问题的解决就是要安排 4 个皇后的位置。因而，每个决策序列由 4 个决策组成：P_1，P_2，P_3，P_4，这里 P_k 代表安排第 k 个皇后的位置，每个皇后都有 16 种可能。该问题的解空间有 16^4 个决策序

列。这时的约束条件是:

任意两个皇后均不能位于同一行、同一列及同一个对角线上

注意到这个解空间比较大，从中搜索可行解较为困难。现在把约束条件中的“任意两个皇后均不在同一行” 也放在问题中考虑，即：将 4 个皇后放在 4×4 棋盘的不同行上（比如，第 i 个皇后确定放在第 i 行），约束条件是：

任意两个皇后均不能位于同一列、同一对角线上

这时的解空间中共有 4^4 个决策序列，因为此时每个皇后只有 4 个可能的位置可选。事实上，我们还可以用另一种方法描述，使得解空间进一步缩小。将问题陈述为：将 4 个皇后放在 4×4 棋盘的不同行、不同列上，使得

任意两个皇后均不能处在同一对角线上

这时的解空间应当由 4！个决策序列构成。因为此时每个决策序列实际上对应于{1, 2, 3, 4}的一个排列。我们可以用一棵排列树来描述解空间。

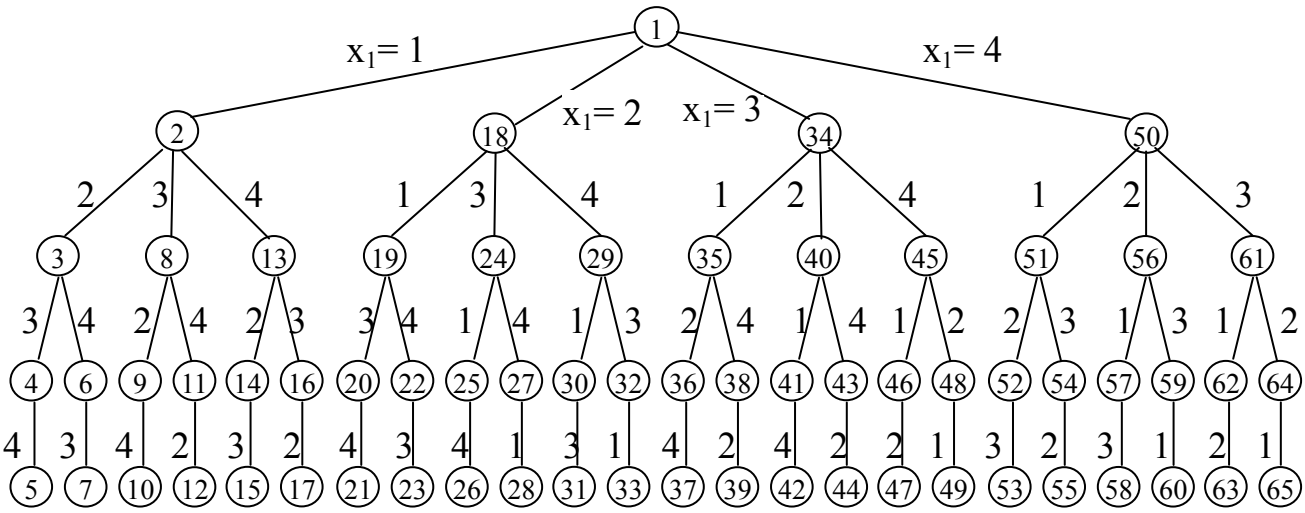


图 6-1-4 四皇后问题的解空间树

从例 6.1.3 来看，解空间的确定与我们对问题的描述有关。如何组织解空间的结构会直接影响对问题的求解效率。这是因为回溯方法的基本思想是通过搜索解空间来找到问题所要求的解。一般地，可以用一棵树来描述解空间，称为解空间树。当所给的问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为子集合树。此时，解空间有 2^n 个元素，遍历子集树的任何算法均需 $\Omega(2^n)$ 的计算时间，如例 2。当所给的问题是确定 n 个元素的满足某种性质的排列时，相应的解空间树称为排列树，此时，解空间有 $n!$ 个元素。遍历排列

树的任何算法均需$\Omega(n!)$的计算时间，如例 1 和例 3。本章只讨论能用这两类解空间树来描述解空间的问题。

为了叙述方便，引进一些关于状态空间树的术语。状态空间树上的每个节点对应求解问题的一个问题状态，它由一条从根到该节点的路径描述。由根到所有其它节点的路径描述了这个问题的状态空间。解状态是这样一些问题状态 S，对于这些问题状态，由根到 S 的那条路径确定了解空间的一个元组。答案状态是这样的一些解状态 S，对于这些解状态而言，由根到 S 的这条路径确定了这个问题的一个解（即可行解），解空间的树结构称为状态空间树。

确定了解空间的组织结构后，回溯法就从初始节点（状态空间树的根节点）出发，以深度优先的方式搜索整个状态空间树。这个开始节点就成为一个活节点，同时也成为当前的扩展节点。在当前扩展节点处，搜索向纵深方向移至一个新节点。这个新节点就成为一个新的活节点，并且成为当前的扩展节点。如果在当前的扩展节点处不能再向纵深方向搜索，则当前的扩展节点就成为死节点。此时应往回移动（回溯）至最近一个活节点处，并使这个活节点成为当前扩展节点。如此继续。回溯法就是以这种工作方式递归地在状态空间中搜索，直至找到要求的解或状态空间中已无活节点时为止。

事实上，当我们将问题的有关数据以一定的数据结构存储好以后（例如，旅行商问题存储赋权图的邻接矩阵、定和子集问题是存储已知的 n+1 个数、4 皇后问题用整数对(i, j)表示棋盘上各个位置，不必先建立一个解空间树），就搜索生成解空间树的一部分或全部，并寻找所需要的解。也就是说，对于实际问题不必生成整个状态空间树，然后在整个解空间中搜索，我们只需有选择地搜索。为了使搜索更加有效，常常在搜索过程中加一些判断以决定搜索是否该终止或改变路线。通常采用两种策略来避免无效的搜索，提高回溯法的搜索效率。其一是使用***约束函数***，在扩展节点处剪去不满足约束的子树；其二是用***限界函数***（后面将阐述）“剪去”不能达到最优解的子树。这两种函数统称为剪枝函数。总结起来，运用回溯法解题通常包括以下三个步骤：

1). 针对所给问题，确定问题的解空间；

2). 确定易于搜索的解空间结构；

3). 以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效的搜索。

§2. 定和子集问题和 0/1 背包问题

定和子集问题可以写成如下数学规划模型：确定所有向量 $x=(x_1,x_2,\cdots,x_n)$ 满足：

$$\begin{aligned} & x_i \in \{0,1\}, i = 1,2,\cdots,n \\ & s.t. \quad \sum_{1 \le i \le n} w_i x_i = M \end{aligned} \tag{6.2.1}$$

如果让w_i，p_i分别表示第 i 件物品的重量和价值，M 代表背包的容量，则 0/1 背包问题可以建立如下的数学模型：确定一个向量$x = (x_1, x_2, \cdots, x_n)$，满足

$$\begin{aligned} & \max \sum_{1 \le i \le n} p_i x_i \\ & s.t. \quad \sum_{1 \le i \le n} w_i x_i \le M \\ & x_i \in \{0,1\}, i = 1,2,\cdots,n \end{aligned} \tag{6.2.2}$$

这是一个优化问题，与定和子集问题比较，它多出目标函数，但只要求确定一个最优解；定和子集问题则往往要求确定所有可行解。

1．定和子集问题

用回溯法求解定和子集问题的过程也即是生成状态空间树的一棵子树的过程，因为,在搜索期间将剪掉不能产生可行解的子树（即不再对这样的子树进行搜索）。按照回溯算法的基本思想，搜索是采用深度优先的路线，算法只记录当前路径。假设当前扩展节点是当前路径的深度为 k 的节点，也就是说当前路径上，x_i，$1 \le i \le k$ 已经确定，算法要决定下一步搜索目标。此时，有两种情况：

$$\sum_{1 \le i \le k} w_i x_i = M \qquad 与 \qquad \sum_{1 \le i \le k} w_i x_i < M$$

前一种情况说明当前路径已经建立一个可行解，当前扩展节点即是一个答案节点。此时，应该记录已经获得的解（后面的变量取 0），停止对该路径的继续搜索，返回到前面最近活节点。后一种情况出现，算法要判断是否需要继续向前搜索。显然，只有当

$$\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge M$$

时才有必要继续向前搜索。如果集合$A = \{w_1, w_2, \cdots, w_n\}$中的元素是按不降的次序排列的，则上述继续搜索的条件还可以加强为

$$\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge M \quad and \quad \sum_{1 \le i \le k} w_i x_i + w_{k+1} \le M \tag{6.2.3}$$

(6.2.3)中两个不等式要求的条件记做 B_{k+1}，即当这两个不等式满足时，B_{k+1}=true；否则，B_{k+1}=false.

程序 6-2-1 定和子集问题的回溯算法

```
SumSubset(s,k,r) // 寻找 W[1..n]中元素和为 M 的所有子集。
    //W[1..n]中元素按不降次序排列，进入此过程时，X[1], . . . , X[k-1]
```

```
    // 的值已经选定，而且 Bk=true。记 s = Σ_{1≤j≤k-1} W[j]X[j]， r = Σ_{k≤j≤n} W[j]，
    //并假定 W[1] ≤ M, Σ_{1≤j≤n} W[j] ≥ M
1   global integer M, n;
2   global real W[1..n];
3   global bool X[1..n];
4   real r, s;
5   integer k, j;
    //由于 Bk =true，因此 s + W[k]≤ M
6   X[k]:=1; // 生成左儿子。
7   if s + W[k]= M then
8     print (X[j],j from 1 to k);
9   else
10    if s + W[k] + W[k+1] ≤ M then
11      SumSubset(s+W[k], k+1, r-W[k]);
12    end{if}
13  end{if}
14  if s + r - W[k]≥ M and s + W[k+1] ≤ M then
15    X[k]:=0; //生成右儿子
16    SumSubset(s,k+1,r-W[k]);
17  end{if}
18  end{SumSubset}
```

因为假定 W[1]≤M，$\sum_{1\le j\le n} W[j] \ge M$，所以程序开始执行时，$B_1$=true。同样，程序在递归执行过程中，总是以前提条件 B_k=true 开始处理深度为 k-1 的节点的儿子们的生成过程。左儿子生成由语句 6 完成，右儿子生成由语句 14～15 完成。在此过程中决定是否转到生成深度为 k 的节点的儿子节点的生成过程，由语句 10～12 和 14～16 完成。所以该算法所生成的子树的叶节点或是某个左儿子，此时找到一个可行解；或者是某个右儿子，此时由根到该节点的路径不能延伸为可行解。

例 6.2.1 n=6,M=30; W[1..6]=(5,10,12,13,15,18)。图 6-1-5 给出了由算法 SumSubset 所生成的解空间树的子树。

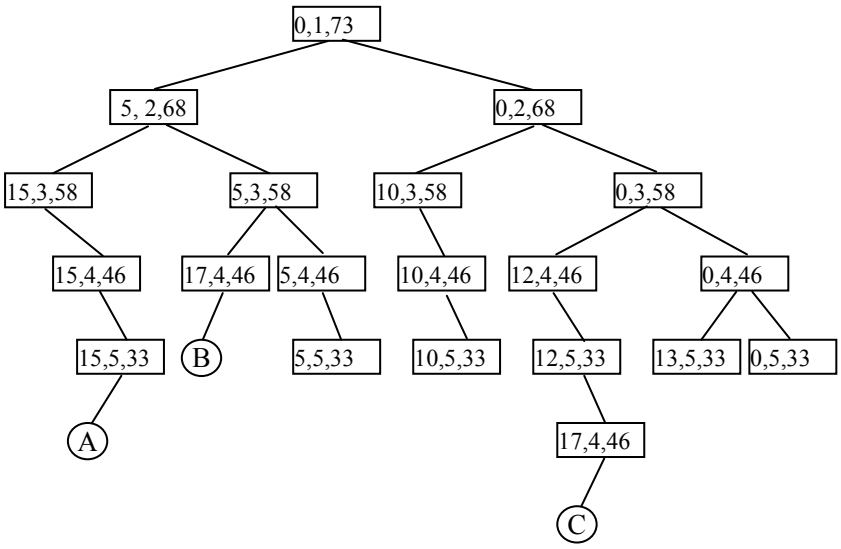


图 6-1-5 算法 SumSubset 所生成的解空间树的子树

2．0/1 背包问题

0/1 背包问题是一个优化问题,因此不仅可以使用约束函数，而且可以使用限界函数做剪枝处理。如果当前背包中的物品总重量是 cw,前面 k-1 件物品都已经决定好是否放入包中,那么第 k 件物品是否放入包中取决于不等式 $cw + w_k \le M$ 是否满足.这即是搜索的约束函数.

我们可以如下确定限界函数。回忆背包问题的贪心算法，当物品按照 $p_i / w_i \ge p_{i+1} / w_{i+1}$ 的规则排序时，最优解是

$$(x_1,\cdots,x_{l-1},x_l,x_{l+1},\cdots,x_n) = (1,\cdots,1,t,0,\cdots,0)$$

的形式,其中, $0 \le t \le 1$.假设当前背包中物品的总效益值和总重量分别为 cp、cw,如下方式确定限界函数:

程序 6-2-2 构造限界函数

```
proc BoundF(cp,cw,k,M) //在前 k-1 件物品装包决策已经作出的前提下，
  //考虑可能达到的最大效益值，返回一个上界。cp、cw 分别代表背包
  //中当前物品的价值和重量
 global n, p[1..n],w[1..n];
 integer k,i;
 real b,c,cp,cw,M;
 b:=cp; c:=cw;
 for i from k to n do
   c:=c+w[i];
```

```
    if c < M then
      b:=b+p[i];
    else
      return(b+(1+(M-c)/w[i])*p[i]);
    end{if}
  end{for}
  return(b);
end{BoundF}
```

这样确定的上界函数表明,从当前确定的 k 定子空间中寻求到的可行解所能达到的效益值以该函数的当前值为上界。所以,如果搜索最优解的算法在此之前已经知道大于这个界值的效益,则没有必要在当前确定的 k 定子空间中搜索.据此我们给出 0/1 背包问题的回溯算法.

程序 6-2-3 0/1 背包问题的回溯算法

```
proc BackKnap(M,n,W,P,fp,X)//M 是背包容量。n 件物品，数组 W[1..n]
  //和 P[1..n]分别表示物品的重量和价值，并按单位价值的不增顺序排
  //列下标。fp 是当前所知背包中物品所能达到的最大价值。X[1..n]中
  //每个元素值取 0 或 1。若物品 k 未放入背包,则 X[k]=0;否则,X[k]=1。
1   integer n,k,Y[1..n],X[1..n];
2   real M,W[1..n],P[1..n],fp,cw,cp;
    //cw、cp 分别代表当前背包中物品的重量和价值;
3   cw:=0; cp:=0; k:=1; fp:=-1;
4   loop
5     while k≤n and cw+W[k]≤M do //使用约束函数
6       cw:=cw+W[k]; cp:=cp+P[k]; Y[k]:=1; k:=k+1;
7     end{while}
8     if k>n then
9       fp:=cp; k:=n; X:=Y;  //修改解
10    else Y[k]:=0;
11    end{if}
12    while BoundF(cp,cw,k,M)≤fp do
13      while k≠0 and Y[k]≠1 do
14        k:=k-1;
15      end{while}
```

```
16        if k:=0 then return; end{if}
17        Y[k]:=0; cw:=cw-W[k]; cp:=cp-P[k];
18      end{while}
19      k:=k+1;
20    end{loop}
21  end{BackKnap}
```

算法采用一个大的循环搜索各条可能的路径。当一条路径的搜索到某一步不能继续往下搜索时,此时或是因为约束条件不满足,或是因为限界函数不满足。它们分别在语句 5~7 的子循环和语句 12~18 的子循环中体现。这是程序的两个主要部分，其余部分主要处理边界条件，如 k > n 的验证等。第一种情况出现时, 语句 8~11 给出部分处理，剩下的事情交给语句 12~19 来处理，给出了搜索退回的方案。

例 6.2.2　0/1 背包问题：n=8，M=110，W=[1，11，21，23，33，43，45，55]，P=[11，21，31，33，43，53，55，65]。解空间树的搜索情况参看本章附页：“0/1 背包问题解空间树搜索情况”。

§3. n-皇后问题和旅行商问题

在第一节已经给出了这两个问题的解空间. 如果用 $(x_1, x_2, \cdots, x_n)$ 表示解，则它是自然数 $\{1,2,\cdots,n\}$ 的一个排列。对于旅行商问题，我们可以假定 $x_1 = 1$，表示售货员的驻地是城市 1。对于旅行商问题，采用回溯法主要是给出约束函数和限界函数；而对于 n 皇后问题，主要任务是给出约束函数。对于旅行商问题可以假定赋权图是完全图，这只要将实际不相邻的两个顶点间用带有权 $+\infty$ 的边连接即可。

1. n-皇后问题

这里的约束条件是：*任何两个皇后都不能位于同一对角线上*。如果用 (i, j) 表示棋盘上第 i 行与第 j 列交叉处的位置，则在同一条平行于主对角线的直线上的两点 (i, j) 和 (k, l) 满足关系式 $j - l = -(i - k)$；而处于同一条平行于副对角线的直线上的两点 (i, j) 和 (k, l) 则满足关系式 $j - l = i - k$. 将这两个关系式略作变形即得两种情况的统一表示：

$$|j - l| = |i - k| \qquad (6.3.1)$$

反之，如果棋盘上的两点(i,j)和(k,l)满足关系式(6.3.1)，则它们一定位于同一条平行于对角线的直线上。所以，如果$(x_1,x_2,\cdots,x_n)$是 n 皇后问题的一个可行解，则对任何$i \neq k$，等式

$$|x_i - x_k| = |i-k| \tag{6.3.2}$$

均不得成立，这即是约束条件。如果假定前面的 k-1 个皇后位置已经排好，现在要安排第 k 个皇后的位置，必须使得$x_i \neq x_k$，而且等式(6.3.2)对于$i=1,2,\cdots,k-1$都不得成立。

算法可以这样设计：对于x_k所有可能的取值，验证上述条件。对于每个取值x_k，这要用一个 bool 函数 Place 来完成。

程序 6-3-1 皇后问题放置函数

```
Place(k)//如果第 k 个皇后能放在第 X[k]列,则返回 true,否则返
  //回 false。X 是一个全程数组，进入此过程时已经设置了 k-1 个值，
  //检验第 k 个皇后的位置 X[k]是否与前 k-1 个皇后的位置有冲突。
    global X[1..k]; integer i,k;
    i:=1;
   while i<k do
      if X[i]=X[k] or |X[i]-X[k]|=|i-k| then
        return (false);
      end{if}
      i:=i+1;
   end{while}
   return(true);
 end{Place}
```

使用函数 Place 能使求 n-皇后问题的回溯算法具有简单的形式.

程序 6-3-2 求 n-皇后问题可行解的回溯算法

```
proc nQueens(n)
 integer k,n,X[1..n];
 X[1]:=0; k:=1; //k 是当前行,X[k]是当前列
 while k>0 do
```

```
      X[k]:=X[k]+1; //转到下一列
      while X[k]<n and Place(k)=false do
         X[k]:=X[k]+1;
      end{while}
      if X[k]≤n then
         if k=n then
           Print(X);
         else k:=k+1; X[k]:=0; //转到下一行
         end{if}
      else k:=k-1; //回溯
      end{if}
    end{while}
  end{nQueens}
```

程序 nQueens 处理 4-皇后问题的执行过程参看本章附页“4 皇后解空间树搜索情况”.

2.旅行商问题

用 0,1,…,n-1 代表图的 n 个顶点,一个周游$i_1i_2\cdots i_ni_1$用数组$(i_1,i_2,\cdots,i_n)$表示，它是旅行商问题的一个可行解。如果可行解的前 k-1 个分量$x_1,x_2,\cdots,x_{k-1}$已经确定，则判定$x_1x_2\cdots x_{k-1}x_k$能否形成一条路径，只需做 k-1 次比较:

$$x_k \neq x_1, x_k \neq x_2, \cdots, x_k \neq x_{k-1}, \tag{6.3.3}$$

此即构成旅行商问题的约束条件。用 w(i,j)记边(i,j)的权值。cl 记当前路径$x_1x_2\cdots x_{k-1}$的长度,即$cl=\sum_{1\le i\le k-2} w(x_i,x_{i+1})$。如果当前知道的最短周游的线路长度为$fl$，则当

$$cl + w(x_{k-1},x_k) > fl \tag{6.3.4}$$

时，$x_1x_2\cdots x_{k-1}x_k$不会是最短周游路线的一部分，在状态空间树中，相应的一枝被剪掉，(6.3.4)即是旅行商问题的限界条件。此外，当$k=n$时，若

$$cl + w(x_{k-1},x_k) + w(x_k,x_1) < fl \tag{6.3.5}$$

则算法需要更新fl，令

$$fl = cl + w(k-1,k) + w(k,1) \tag{6.3.6}$$

程序 6-3-3 旅行商问题的约束条件函数

```
NextValue(k) //从节点 1 出发的路径,如果 X[k]是前面某个节点 X[i](i<k)
   //则返回 false，否则返回 true。X 是一个全程数组，进入此过程时前
   //k-1 个值已经确定，而且第 k 个值 X[k]已经选出备检。
   global X[1..k]; integer i，k;
   i:=1;
   while i<k do
     if X[k] = X[i] then
        return (false);
     end{if}
     i:=i+1;
   end{while}
     return(true);
end{NextValue}
```

使用函数 NextValue 能使求旅行商问题的回溯算法具有简单的形式

程序 6-3-4 求旅行商问题的回溯算法

```
proc BackTSP(n,W) // 求图 G 的从顶点 0 出发的最短周游（Hamilton 圈）
   //路径，W 是 G 的邻接矩阵。X[k]是当前路径的第 k 个顶点，cl 是当前
   //路径的长度，fl 是当前所知道的最短周游长度.
    in teger  k,n,X[1..n];
    real W[1..n,1..n],cl,fl;
    for i to n do
      X[i]:=0;
    end{for}
    k:=2; cl:=0; fl:=+∞;
    while k>1 do
       X[k]:=X[k]+1 mod n; //给 X[k]预分配一个值
       for j from 1 to n do
         if NextValue(k)=true then
            cl:=cl+W(X[k-1],X[k]); break;
         end{if}
```

```
        X[k]:= X[k]+1 mod n; //重新给 X[k]分配一个值。
      end{for}
      if fl ≤ cl or k=n and fl < cl+W[X[k],1] then
         cl:=cl-W(X[k-1],X[k]); k:=k-1; //回溯
      elif k=n and fl ≥ cl+W(X[k],1) then
        fl:=cl+W(X[k],1);
        cl:=cl-W(X[k-1],X[k]); k:=k-1; //回溯
      else k:=k+1;//继续纵深搜索
      end{if}
    end{while}
  end{BackTSP}
```

旅行商问题的例子

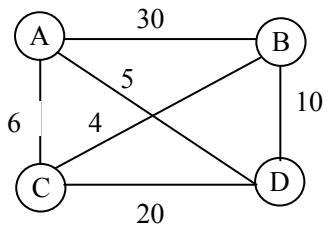
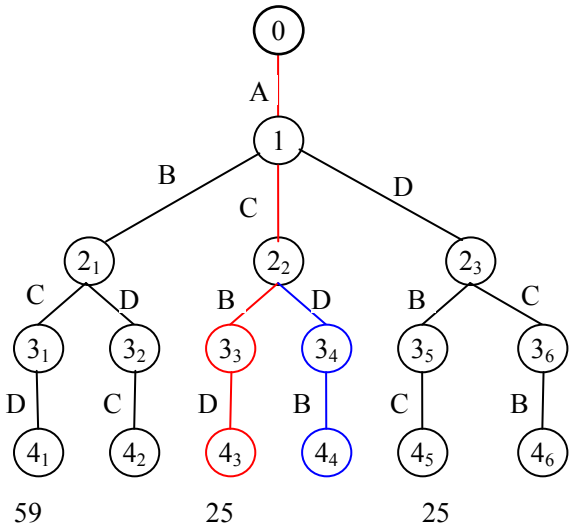


图　G



旅行商解空间树上的搜索情况

§4 图的着色问题

已知一个无向图 G 和 m 种颜色，在只准使用这 m 种颜色对图 G 的顶点进行着色的情况下，是否有一种着色方法，使图中任何两个相邻的顶点都具有不同的颜色？如果存在这样的着色方法，则说图 G 是 m-可着色的，这样的着色称为图 G 的一种 m-着色。使得 G 是 m-可着色的最小数 m 称为图 G 的色数。这一节所讨论的图的着色问题是：

给定无向图 G 和 m 种颜色，求出 G 的所有 m-着色

用 G 的邻接矩阵 W 表示图 G，W[i,j]=1 表示顶点 i 与 j 相邻（有边相连），否则

W[i,j]=0，(i,j = 1,2,...，n)。 m 种颜色分别用 1，2，…,m 表示。图 G 的每一种 m-着色都可以用一个 n-维数组 X[1..n]表示。X[i]=k 表示顶点 i 被着上颜色 k。因为每一个顶点至多有 m 种着色可选，简单分析可知，该问题的解空间是一个完全的 m-叉树，树的高度是 n。X 是可行解，当且仅当

$$W[i,j]=1 \Rightarrow X[i]\neq X[j] \qquad (6.4.1)$$

此即是约束条件。假如前 j-1 个顶点已经着好颜色，即 X[1],…,X[j-1]已经确定，则确定第 j 个顶点是否有可着的颜色 X[j]，根据约束条件，应该验证关系

$$W[i,j]=1 \Rightarrow X[i]\neq X[j],\ 1\leq i<j \qquad (6.4.2)$$

是否成立。这可由下面程序完成。

程序 6.4.1 下一步选色算法

```
NextColor ( j ) //进入此过程前，X[1],…,X[j-1]已经确定且满足约束
  //条件。本过程给 X[j]确定一个整数 k，0≤k≤m：如果还有一种颜色 k
  //可以分配给顶点 j（满足约束条件），则令 X[j]＝k；否则，令 X[j]＝0。
  global integer m, n, X[1..n], W[1..n, 1..n];
  integer j, k;
  loop
    X[j]:=X[j]+1 mod (m+1);
    if X[j]=0 then return; end{if}
    for i to j-1 do //验证约束条件
      if W[i,j]=1 and X[i]=X[j] then break; end{if}
    end{for}
    if i=j then return; end{if} //找到一种颜色
  end{loop}
end{NextColor}
```

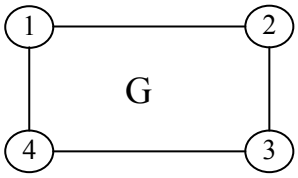
在程序 NextColor 开始执行之前，诸 X[k]均已经有值。这可在着色问题的回溯算法初次调用该函数时赋予初值：X[i]=0,i=1,2,…,n。

程序 6-4-2 图的着色问题回溯算法

```
proc GraphColor（k）// 采用递归。W[1..n, 1..n]是图 G 的邻接矩阵，
 // 1, 2,… m 代表 m 种颜色。k 是下一个要着色的顶点。在调用
 //GraphColor（1）之前，数组 X 的每个分量已经赋值 0。
 global integer m, n, X[1..n], W[1..n, 1..n] ;
```

```
        if k=0 then exit; end{if}
        NextColor(k); // 确定 X[k]的取值
        if X[k]=0 then k:=k-1; //说明没有颜色可以分配给第 k 个点
        else
          if k=n then   //已经找到一种着色方法
            print(X);
          else
            k:=k+1; GraphColor( k );  //递归
          end{if}
        end{if}
    end{GraphColor}
```

例 6.4.1 n＝4，m＝3，
无向图 G 如右图



正好用 3 种颜色
的解只有 12 个

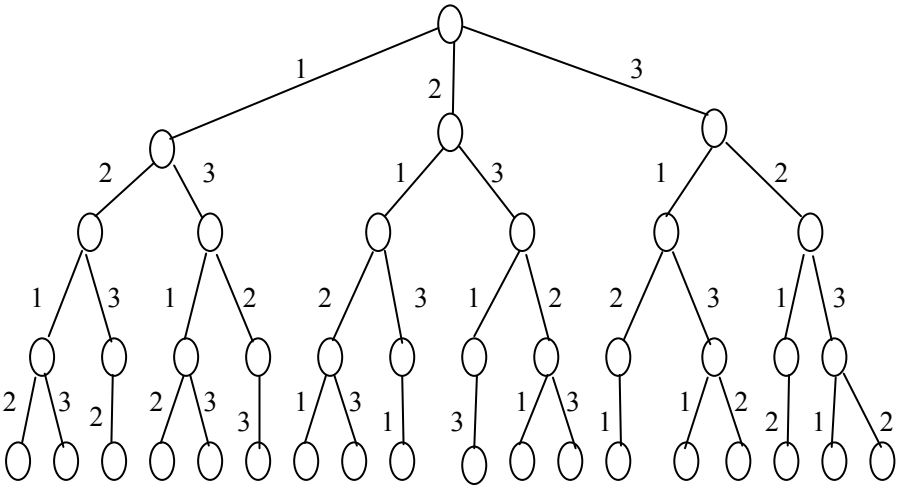


图 6-4-1 一个 4 顶点的图和所有可能的 3 着色

§5. 回溯法的效率分析

一. 回溯法的一般型式

每个节点确定所求解问题的一个问题状态，由根节点到其它节点的所有路径确定了这个问题状态空间。解状态 S 是指由根到节点 S 那条路径能够形成解空间中的一个元组。在图 6-5-2 的解空间树中，所有的节点都是解状态，而在图 6-5-1

的解空间树中，只有叶节点才是解状态。答案状态是指这样的解状态 S，由根到节点 S 的路径确定了这个问题的一个可行解。解空间的树形结构也称为状态空间树。以下是子集树的两个不同表示形式：图 6-5-1 是静态的，而图 6-5-2 是动态的，因为它每个解状态的元组大小是变化的。

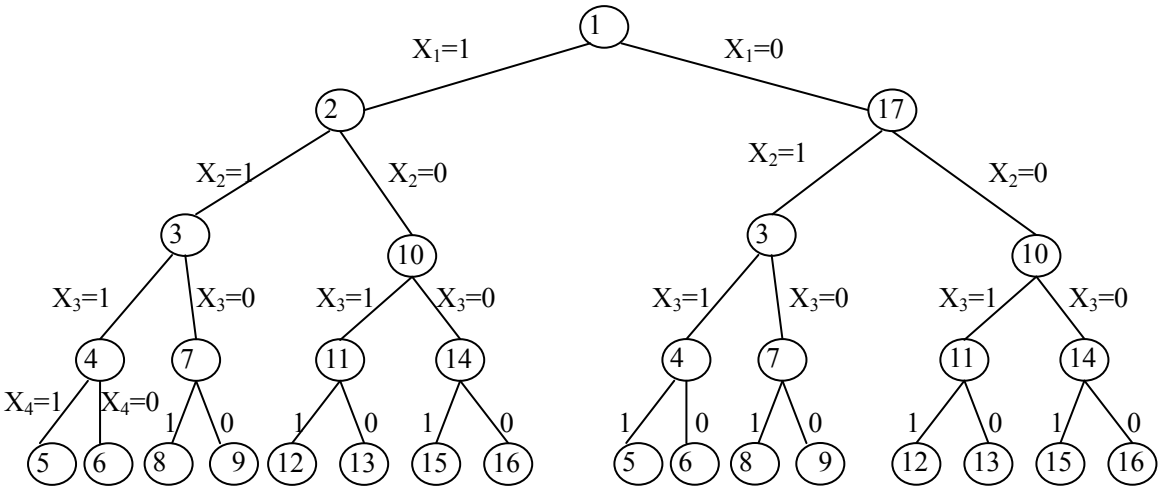


图 6-5-1 元组大小固定的解空间树表示

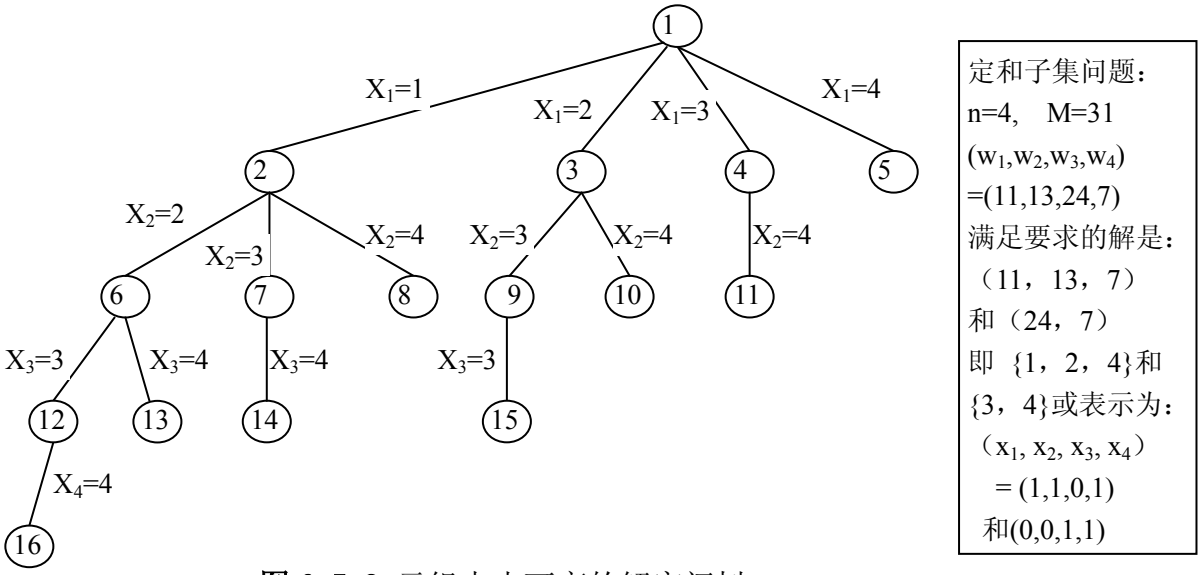


图 6-5-2 元组大小可变的解空间树

对于任何一个问题，一旦设想出一种状态空间树，那么就可以先系统地生成问题状态，接着确定这些问题状态中的哪些状态是解状态，最后确定那些解状态是答案状态，从而将问题解出。回溯法的形式描述如下：

假定要找出所有问题的答案节点。设（x_1，x_2，…，x_{k-1}）是状态空间树中由根到一个节点（即，问题状态）的路径，而 T（x_1，x_2，…，x_{k-1}）是下述所有 x_k 的集合，它使得（x_1，x_2，…，x_{k-1}, x_k）是一条由根到问题状态（节点 X）的路径。还假定存在一些约束条件函数 B_k，如果路径（x_1，x_2，…，x_{k-1}, x_k）不可能延伸到一个答案节点，则 B_k（x_1，x_2，…，x_{k-1}, x_k）取假值，否则，取真值。于是，解向量 X=（x_1，x_2，…，x_{n-1}, x_n）中的第 k 个分量就是选自集合 T（x_1，x_2，…，x_{k-1}）、且使得 B_k 为真的 x_k。

程序 6-5-1 回溯算法可以抽象地描述

```
BACKTRACK(n)
  //每个解都在 X(1..n)中生成，一个解一经确定就立即打印。在
  //X(1), … , X(k-1)已经被选定的情况下，T(X(1), … , X(k-1))
  //给出 X(k)的所有可能的取值。约束条件函数 Bk（x1, x2, …, xk-1, xk）
  //给出哪些元素 X(k)满足隐式约束条件。
  integer k, n;
  local X(1..n);
  k:=1;
  while k > 0 do
    if 还剩有没有被检验过的 X(k)使得
       X(k)∈ T(X(1), … , X(k-1)) and
       Bk（X(1), … , X(k-1),X(k)）=true
     then
       if （X(1), …, X(k-1),X(k)）是一条已到达答案节点的路径
       then  print（X(1), … , X(k-1),X(k)）;
       end{if};
       k:=k+1; //考虑下一个集合
     else k:=k-1;//回溯先前的集合
     end{if};
   end{while}
end{BACKTRACK}
```

二. 回溯法的效率分析

回溯法是以深度优先的方式系统地搜索问题的解空间。解空间是由所有可能的决策序列 $(x_1,x_2,\cdots,x_n)$ 构成。在搜索过程中，采用剪枝函数（约束函数和限界函数）尽量避免无意义的搜索。通过前面的具体实例的讨论容易看出，一个回

溯算法的效率在很大程度上依赖于以下几个因素：

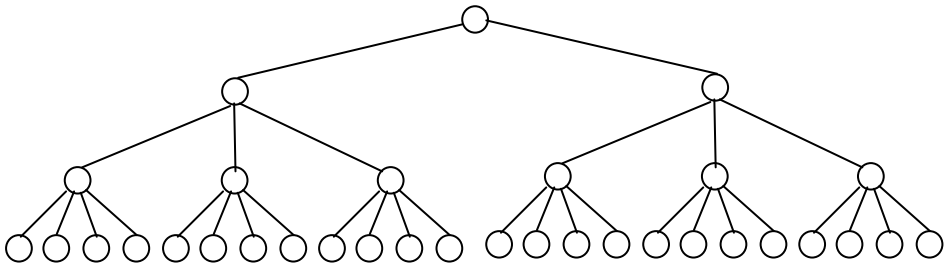
1)．产生 x_k 的时间；

2)． x_k 的取值范围；

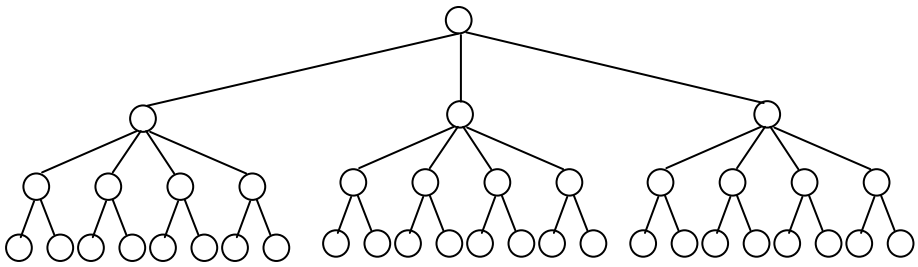
3)．计算约束函数的时间和限界函数的时间；

4)．满足约条件及限界函数条件的 x_k 的个数。

一般地，一个好的约束函数能够显著地减少所生成的节点的数目。但是，这样的约束函数往往计算量较大。因此在选择约束函数时通常存在着节点数与约束函数计算量之间的折中。我们希望总的计算时间较少。为了提高效率，通常采用所谓的“重排原理”。对于许多问题而言，在进行搜索试探时，选取 x_k 值的顺序是任意的。这提示我们，在其它条件相同的前提下，如果让***可能取值个数最少的*** x_k ***优先考虑***，则将会使算法更有效。图 6-5-3 给出了同一个问题的不同的解空间树。在图(a)中，若从第 1 层剪去一棵子树，则从所有应当考虑的三元组中一次消去 12 个三元组；在图(b)中，虽然同样从第 1 层剪去一棵子树，却只从应当考虑的三元组中一次消去 8 个三元组。前者的效果显然比后者好，从中可以体会到这种策略的效力（但这不是绝对的，只是从信息论的角度看，平均来说更为有效，n 皇后问题就是一个反例）。



（a）



（b）

图 6-5-3 解空间树的比较

解空间结构一经选定，影响回溯法效率的前三个因素就可以确定，只剩下生成节点的数目是可变的，它将随问题的具体内容以及拟定的不同生成方式而变化。即使是对于同一问题的不同实例，回溯法所产生的节点的数目也会有很大变

化。对于一个实例，回溯法可能只产生$O(n)$个节点,而对于另一个相近的实例，回溯法可能产生解空间中的所有节点。如果解空间的节点数是2^n或$n!$，则在最坏情况下,回溯法的时间耗费一般为$O(p(n)2^n)$或$O(q(n)n!)$,其中，$p(n)$和$q(n)$均为n的多项式。对于一个具体问题来说，回溯法的有效性往往就体现在当问题实例的规模n较大时，它能够用很少的时间求出问题的解。而对于一个问题的具体实例，我们又很难预测回溯法的行为。特别地，很难估计在解这一具体实例时所产生的节点数。这是在分析回溯法效率时遇到的主要困难。下面所介绍的概率方法也许对解决这个问题有所帮助。

当用回溯法解某一个具体问题实例之前，可用蒙特卡罗方法估算一下即将采用的回溯算法可能要产生的节点数目。该方法的基本思想是在解空间树上产生一条随机路径，然后沿此路径来估算解空间中满足约束条件的节点数 m。设 x 是所产生的随机路径上的一个节点，且位于解空间树的第 i 级上(即深度为 i)。对于 x 的所有儿子节点，用约束函数检测出满足约束条件的节点数 m_i。路径上的下一个节点是从 x 的 m_i 个满足约束条件的儿子节点中随机选取的。这条路径一直延伸到一个叶节点或一个所有儿子节点都不满足约束条件的节点为止。通过这些 m_i 的值，就可估计出解空间树中满足约束条件的节点的总数 m。在用回溯法求问题的所有可行解时，这个估计值特别有用。因为在这种情况下，解空间所有满足约束条件的节点都必须生成，设这样节点的总数是 m。若只要求用回溯法找出问题的一个解，则所生成的节点一般只是 m 个满足约束条件的节点中的一部分。此时用 m 来估计回溯法生成的节点数就过于保守了。为了从 m_i 的值求出 m 的值，还需要对约束函数做一些假定。在估计 m 时，假定所有约束函数是静态的，即在回溯法执行过程中，约束函数并不随随算法所获得信息的多少而动态地改变。进一步还假定对解空间树中***同一级的节点所用的约束函数是相同的。***对于大多数回溯法，这些假定都太强了。实际上，在大多数回溯法中，约束函数是随着搜索过程的深入而逐渐加强的。在这种情形下，按照前面所做假定来估计 m 就显得保守。如果将约束函数变化的因素也加进来考虑，所得出的满足约束条件的节点总数将会少于 m，而且会更精确些。

在静态约束函数假设下，第 0 级共有 m_0 个满足约束条件的节点。若解空间的同一级的节点具有相同的出度，则在第 0 级上的每个节点将会有 m_1 个儿子节点满足约束条件。因此，第 1 级将会有 m_0m_1 个满足约束条件的节点。同理，第 2 级上满足约束条件的节点个数是 $m_0m_1m_2$。依此类推，第 i 级上满足约束条件的节点的数目是 $m_0m_1m_2\cdots m_i$. 因此，对于给定的实例，如果产生解空间树上的一条途径，并求出 m_0，m_1，m_2，…，m_i，…，则可以估计出回溯法要生成的满足约束条件的节点数目为 $m = m_0+ m_0m_1+ m_0m_1m_2+\cdots$。假定回溯法要找出所有的可行解，设

$(x_1,x_2,\cdots,x_{i-1})$是解空间树中由根到一个节点的路径，而$T(x_1,x_2,\cdots,x_{i-1})$表示所有这样的$x_i$，使得$(x_1,x_2,\cdots,x_{i-1},x_i)$能够成为解空间树中的一条路径。$B_k$表示第 k 级上约束函数：$B_k$(X[1], …, X[k])=true 表示这条路径到目前为止未违背约束条件。否则B_k(X[1], …, X[k])=false。

程序 6-5-2 回溯法的递归流程

```
RecurBackTrack（k）//进入算法时，解向量 X[1..n]的前 k-1 个分
   //量 X[1],…, X[k-1]已经赋值
   global n, X[1..n];
   for 每个满足
     "X[k]∈ T(X[1], …, X[k-1]) and Bk(X[1], …, X[k])=true"
     的 X[k]  do
     if (X[1], …, X[k]) 是一条抵达某一答案节点路径 then
        print (X[1], …, X[k]);
     end{if}
     RecurBackTrack（k+1）
   end{for}
 end{RecurBackTrack}
```

根据前面的分析，我们不能给出回溯法递归流程生成的解空间树中节点数目 m。但是，可以采用下面程序给出解空间树中节点个数的估计。

程序 6-5-3 回溯法效率估计

```
Estimate //程序沿着解空间树中一条随机路径估计回溯法生成的顶点
  //总数 m
  m:=1; r:=1; k:=1;
  loop
    T:={X[k]| X[k]∈ T(X[1], …, X[k-1]) and
    Bk(X[1], …, X[k])=true};
    if T ={} then exit; end{if}
    r:=r*size(Tk); m:=m+r;
    X[k]:=Choose(T);  k:=k+1;
  end{loop}
  return(m);
```

```
end{Estimate}
```

其中 Choose(T)是从 T 中随机地挑选一个元素。

当用回溯法求解某个具体问题时，可用算法 Estimate 估算回溯法生成的节点数。若要估计得精确些，可选取若干条不同的随机路径（通常不超过 20 条），分别对各随机路径估计节点数，然后以平均值作为 m。例如 8 皇后问题，回溯算法 nQueens 可以用 Estimate 来估计。参本章附页的“效率分析例图”给出了算法 Estimate 产生的 5 条随机路径所相应的 8×8 棋盘状态。当需要在棋盘上某行放入一个皇后时，所放的列是随机选取的。它与已在棋盘上的其它皇后互不攻击。在图中棋盘下面列出了每层可能生成的满足约束条件的节点数，即

$$m_0,\ m_1,\ m_2,\ \cdots,\ m_i,\ \cdots,$$

以及由此随机路径算出的节点总数 m 的值。这 5 个总点数的均值为 1812.2。

注意到 8 皇后解空间树的节点总数是：

$$1+\sum_{j=0}^{7}\prod_{i=0}^{j}(8-i)=109601$$

1812.2/109601≈1.65%，可见，回溯法效率远远高于穷举法。

习题六

1. 网络设计问题。石油传输网络通常可表示为一个非循环带权有向图 G。G 中有一个称为源的顶点 s。石油从该顶点输送到 G 的其它顶点。图 G 中每条边的权表示该边连接的两个顶点间的距离。网络中的油压随距离增大而减小。为了保证整个输油网络正常工作，需要维持网络中的最低油压 Pmin。为此需要在网络某些或全部顶点处设置增压器。在设置增压器的顶点处油压可升至最大值 Pmax。油压从 Pmax 减至 Pmin 可使石油传输的距离至少为 d。试设计一个算法，计算出网络中增压器的最优放置方案，使得用最少的增压器保证石油运输畅通。

2. 工作分配问题。设有 n 件工作需要分配给 n 个人去完成。将工作 i 分配给第 j 个人完成所需要的费用为 c_{ij}。试设计一个算法，为每一个人分配一件不同的工作，并使总费用达到最小。

3. 最佳调度问题。假设有 n 个任务要由 k 个可并行工作的机器来完成。完成任务 i 需要的时间为 t_i。试设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的结束时间最早。

4. 世界名画陈列馆问题。世界名画陈列馆由 m×n 个陈列室组成。为了防止名画被盗，需要在陈列室中设置机器人哨位。每个警卫机器人除了监视它所在的陈列室外，还可以监视与它所在的陈列室相邻的上、下、左、右 4 个陈列室。试

设计一个安排警卫机器人哨位的算法，使得名画陈列馆中的每一个陈列室都在警卫机器人的监视之下，且所用的警卫机器人最少。

5. 最小重量机器设计问题。设某一机器由 n 个部件组成，每一种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。试设计一个算法，给出总价格不超过 c 的最小重量机器设计。

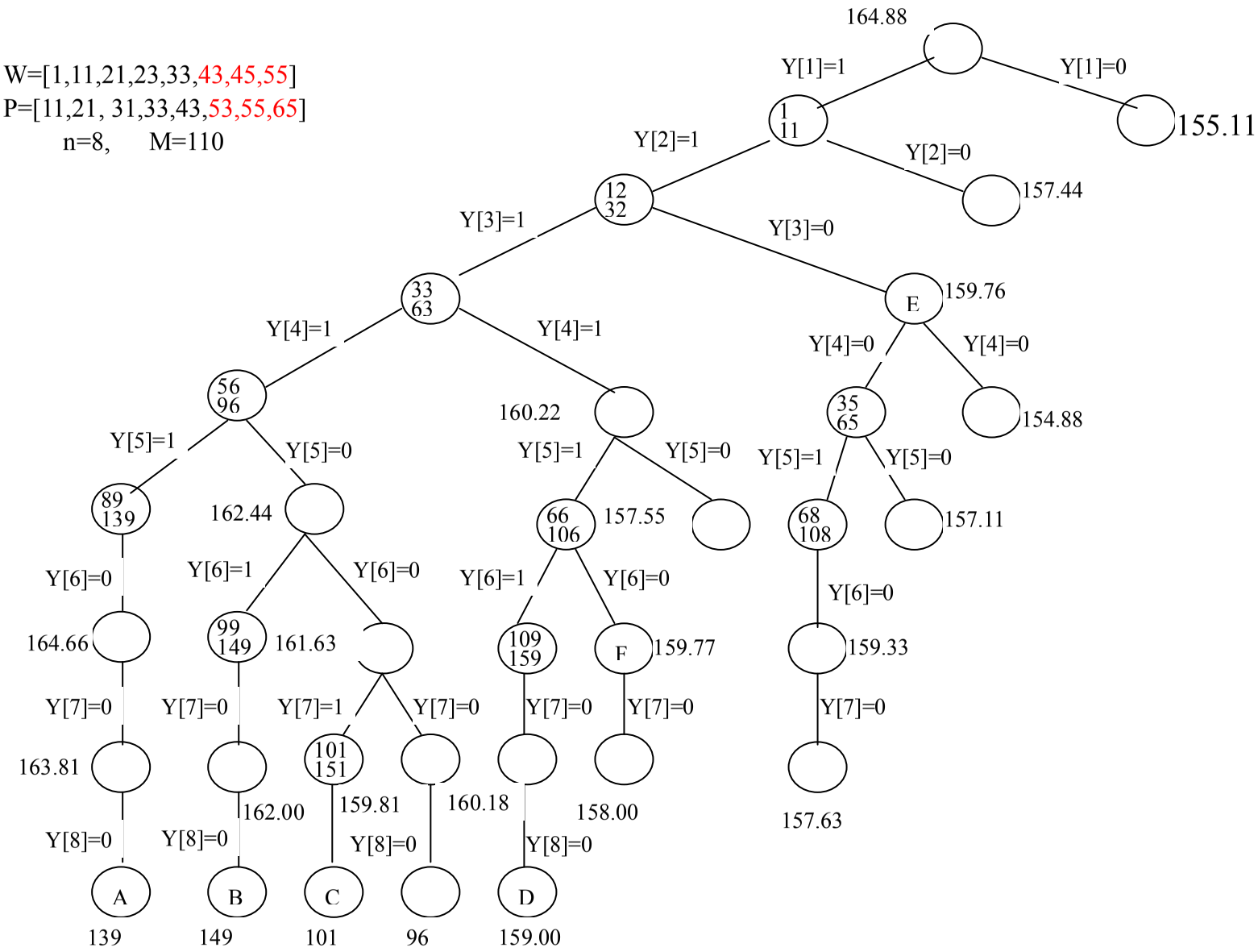
0/1 背包

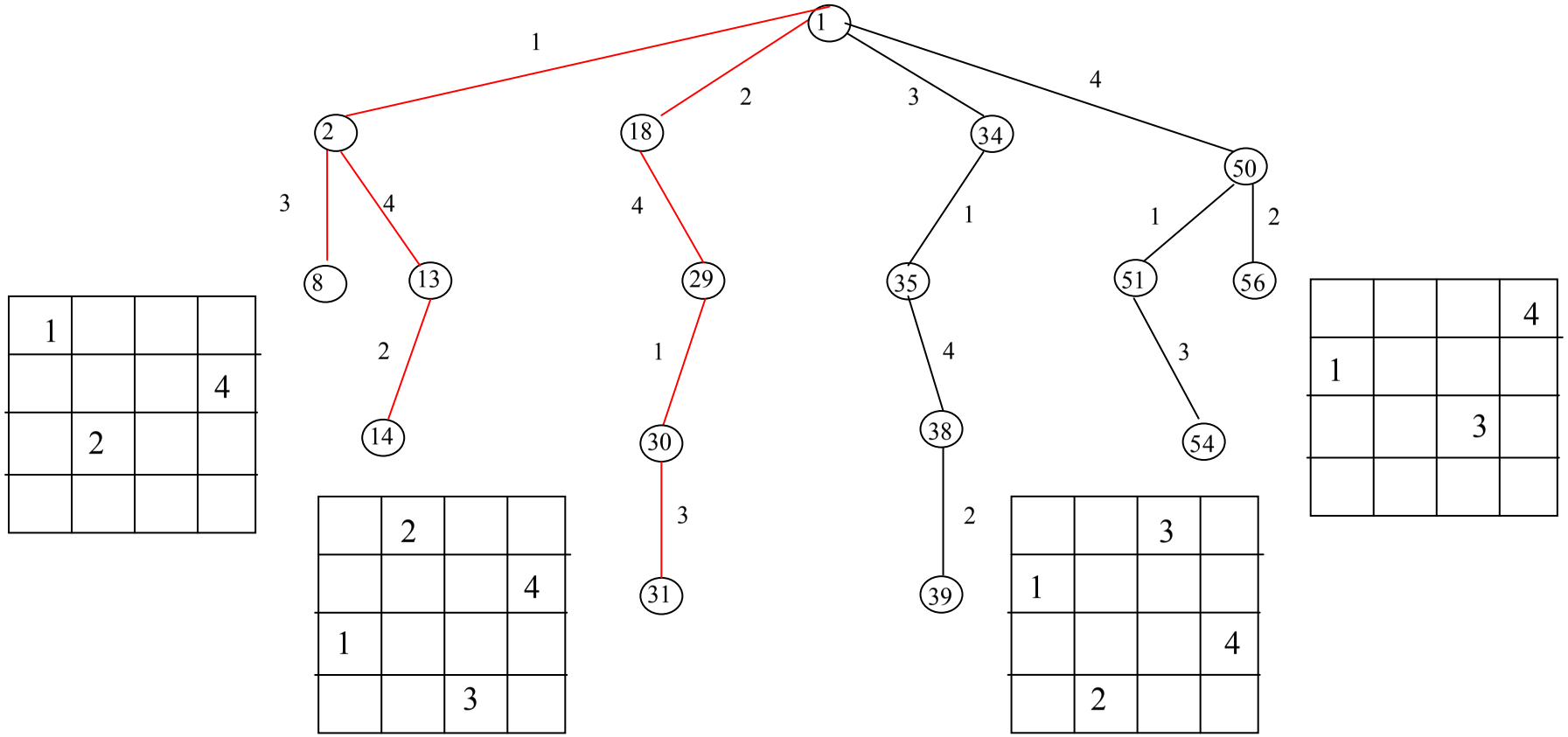
问题

W=[1,11,21,23,33,43,45,55]
P=[11,21, 31,33,43,53,55,65]
n=8,　M=110

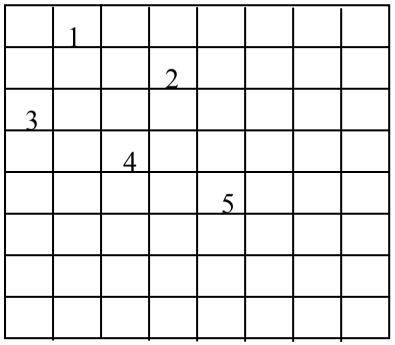
解空间

搜索情况

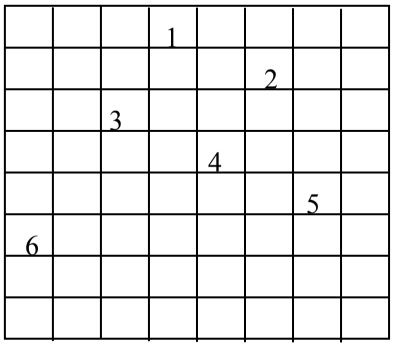




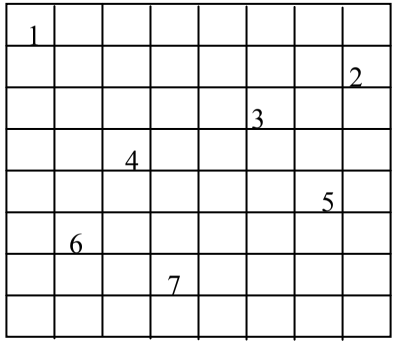
4 皇后解空间树搜索情况



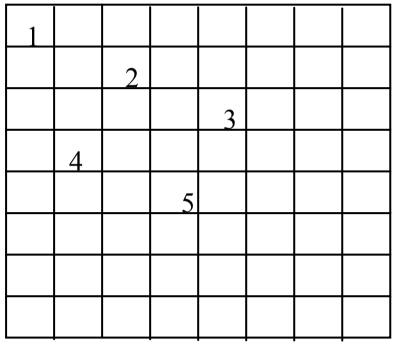
(8,5,4,3,2)=1649



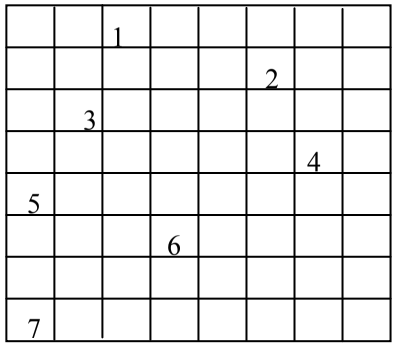
(8,5,3,1,2,1)=769



(8,6,4,2,1,1,1)=1785



(8,6,3,2)=1977



(8,5,3,2,2,1,1,1)=2329

8 皇后解空间树的顶点

总数是：

$$1+\sum_{j=0}^{7}\prod_{i=0}^{j}(8-i)=\mathbf{109601}$$

相当于

1702/109601≈1.55%

第七章　分枝－限界法

§1　算法基本思想

分枝限界法同回溯法类似，它也是在解空间中搜索问题的可行解或最优解，但搜索的方式不同。回溯法采用深度优先的方式，朝纵深方向搜索，直至达到问题的一个可行解，或经判断沿此路径不会达到问题的可行解或最优解时，停止向前搜索，并沿原路返回到该路径上最后一个还可扩展的节点。然后，从该节点出发朝新的方向纵深搜索。分枝限界法则采用宽度优先的方式搜索解空间树，它将活节点存放在一个特殊的表中。其策略是：***在扩展节点处，首先生成其所有的儿子节点，将那些导致不可行解或导致非最优解的儿子节点舍弃，其余儿子节点加入活节点表中。然后，从活节点表中取出一个节点作为当前扩展节点，重复上述节点扩展过程。***

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枝限界法。最常见的有以下两种方式：

1)．队列式

这种方式是将活节点表组织成一个队列或者一个栈，并按队列的先进先出（FIFO）或后入先出（LIFO）原则选取下一个节点作为当前扩展节点。

2)．优先队列式

这种方式是将活节点表组织成一个优先队列，并按优先队列给节点规定的优先级选取优先级最高的下一个节点作为当前扩展节点。

这两种方式各有优缺点，第一种方式组织活节点表简单省时，但是搜索解的路线比较单一，可能会绕道找到解；第二种方式在搜索方向上有所选择，可能会取近道找到解，但是组织节点表较复杂，有时会占用不少的时间。

队列式分枝限界法搜索解空间树的方式类似于解空间树的宽度优先搜索，不同的是队列式分枝限界法不搜索不可行节点（已经被判定不可能导致可行解或不可能导致最优解的节点）为根的子树。为达此目的，算法不把这样的节点列入活节点表。在选择当前扩展节点上是按照某种固定顺序，可称为静态选择法。

优先队列式分枝限界法的搜索方式是根据活节点表中节点的优先级确定下一个扩展节点。节点的优先级常用一个与该节点有关的数值 p 来表示。最大优先队列规定 p 值较大的节点的优先级较高。在算法实现时通常用一个最大堆来实现最大优先队列，用最大堆的 Deletemax 运算抽取堆的根节点作为当前扩展节点，体现最大效益优先的原则。类似地，最小优先队列规定 p 值较小的节点的优先级较高。在算法实现时，常用一个最小堆来实现，用最小堆的 Deletemin 运算抽取堆的根节点作为当前扩展节点，体现最小优先的原则。采用优先队列式分枝限界算法解决具体问题时，应根据问题的特点选用最大优先或最小优先队列，确定各

个节点的 p 值。

例7.1.1 旅行商问题，n=4，其解空间树是一棵排列树。对于如下的赋权图 G 表示的旅行商问题，我们分别说明队列式分支限界算法和优先队列分枝限界算法的执行过程。

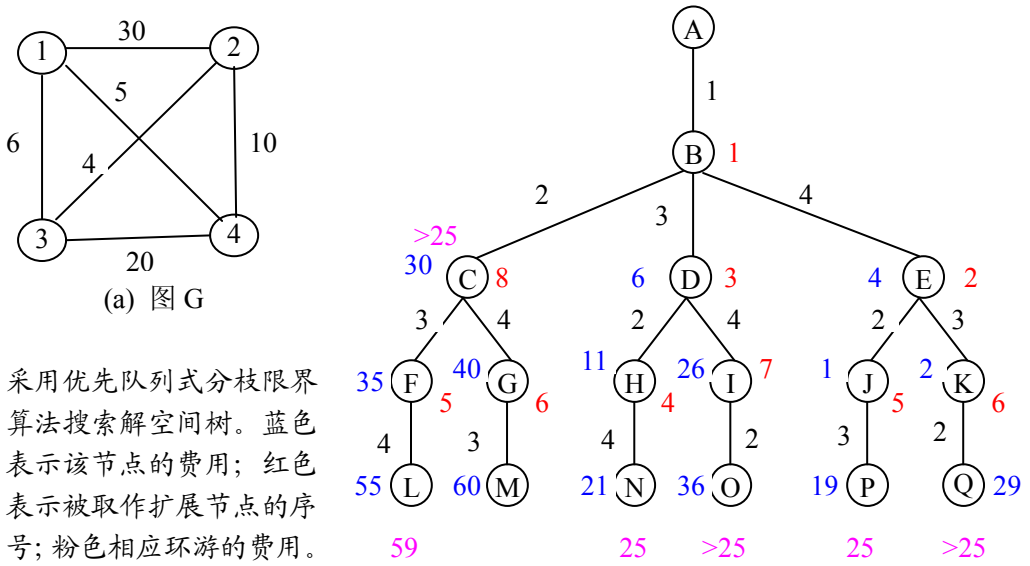


图 7-1-1 表示旅行商问题的赋权图 G 及解空间树的搜索情况

采用**队列式分枝限界法**以排列树中的节点 B 作为初始扩展节点，此时，活节点队列为空。由于从图 G 的顶点 1 到顶点 2、3 和 4 均有边相连，B 的儿子 C、D 和 E 都是可行节点，它们被依次加入到活节点队列中。当前活节点队列中的队首节点 C 成为下一个扩展节点。由于图 G 的顶点 2 到顶点 3 和 4 有边相连，故节点 C 的二个儿子 F 和 G 均为可行节点，可以加入活节点队列。接下来，节点 D 和节点 E 相继成为扩展节点。此时活节点队列中的节点依次为 F、G、H、I、J、K。节点 F 成为下一个扩展节点，但其儿子 L 是解空间树的叶节点，我们找到了一条 Hamilton 圈（1，2，3，4，1），其费用为 59。此时记录这个目标函数值 f=59。下一个扩展节点 G 的儿子 M 也是叶节点，得到另一条 Hamilton 圈（1，2，4，3，1），其费用为 66。节点 H 成为当前扩展节点，其儿子 N 也是叶节点，得到第三条 Hamilton 圈，其费用为 25，因为它比记录中的目标函数值还小，所以修改目标函数值记录：f=25。下一个扩展节点是 I，由于从根节点到节点 I 的费用 26 已经超过目标函数的当前值，故没有必要扩展 I，以 I 为根的子树被剪掉。最后 J 和 K 被依次扩展，活节点队列成为空集，算法终止。算法搜索到的最优值是 25，相应的最优解是从根节点到节点 N 的路径

（1，3，2，4，1）

采用**优先队列式分枝限界法**，用一个最小堆存储活节点表，优先级函数值是

节点的当前费用。算法还是从排列树的节点 B 和空队列开始。节点 B 被扩展后，它的 3 个儿子 C、D 和 E 被依次插入最小堆中。此时，由于 E 是堆中具有最小当前费用（为 4）的节点，所以处于堆顶的位置，它自然成为下一个扩展节点。节点 E 被扩展后，其儿子 J 和 K 被插入当前堆中，它们的费用分别为 14 和 24。此时，堆顶元素是 D，它成为下一个扩展节点。它的 2 个儿子 H 和 I 被插入堆中。此时，堆中元素是节点 C、H、I、J、K。在这些节点中，H 具有最小费用，成为下一个扩展节点。扩展节点 H 后得到一条 Hamilton 圈（1，3，2，4，1），相应的费用为 25。接下来，节点 J 成为扩展节点，并由此得到一条 Hamilton 圈（1，4,2,3,1),费用仍为 25。此后的两个扩展节点是 K 和 I。由节点 K 得到的 Hamilton 圈的费用高于当前所知最小费用，节点 I 当前的费用已经高于当前所知最小费用，因而，它们都不能得到最优解。最后，优先队列为空，算法终止。

对于优化问题，要记录一个到目前已经取得的最优可行解及对应的目标函数值，这个记录要根据最优的原则更新。无论采用队列式还是优先队列式搜索，常常用目标函数的一个动态界（函数）来剪掉不必要搜索的分枝。

对于最大值优化问题，常引用一个可能获得的目标函数值的一个上界 CUB（经此节点可能达到的最大“效益值”)。如果当前扩展节点的儿子节点处的动态上界 CUB 小于目前所取得的目标函数值 prev，则该儿子节点不被放入活节点表。实际上相当于剪掉了解空间树中以该节点为根的子树。

对于最小值优化问题，常引用一个可能出现的目标函数值的一个动态下界 CLB（经此节点可能出现的最小“成本”)，如果当前扩展节点的儿子节点处的动态下界 CLB 大于目前所取得的目标函数值 prev，则该节点不被放入活节点队列。

对于只是找可行解的问题，我们可以考虑如何降低搜索成本。常引用一个可能需要的成本的动态下界 CLB，如果当前扩展节点的儿子节点处的动态下界 CLB 大于目前所知道的到达最小成本答案节点所需要的成本 prev，则该节点不放入活节点表。

上述动态界称为剪枝函数，采用剪枝函数可以减少活节点数，降低搜索过程的复杂度。另一方面，对于采用优先队列式分支限界算法，选择当前扩展节点应该朝着效率高的方向努力。对于只是找可行解的问题，给每个进入活节点表的节点一个成本最大下界估值$\hat{c}(X)$，当前扩展节点就选活节点表中$\hat{c}(X)$最小的。对于最小化问题也同样处理，这里的成本可以采用目标值。对于最大化问题，给每个进入活节点表的节点一个“效益”最小上界估值$\hat{c}(X)$，当前扩展节点就选活节点表中$\hat{c}(X)$最大的。基于这种考虑，第四节将提出统一的 LC－分枝限界算法。

§2 0/1 背包问题的分枝－限界法

用优先队列式分枝限界法解决 0/1 背包问题(作为最大优化问题)，需要确定以下四个问题：

i. 解空间树中节点的结构；

ii. 如何生成一个给定节点的儿子节点；

iii. 如何组织活节点表；

iv. 如何识别答案节点。

采用完整的二叉树作为解空间树，放在活节点表中的每个节点具有 6 个信息段：

Parent、Level、Tag、CC、CV、CUB

其中 Parent 是节点 X 的父亲节点连接指针；Level 标出节点 X 在解空间树中的深度，通过置 $X_{Level(X)+1}=1$ 表示生成 X 的左儿子，置 $X_{Level(X)+1}=0$ 表示生成 X 的右儿子；信息段 Tag 用来输出最优解各个分量 x_i 的值；信息段 CC 记录背包在节点 X 处的可用空间(即剩余空间)，在确定 X 左儿子的可行性时用；CV 记录在节点 X 处背包中已装物品的价值(或效益值)，等于 $\sum_{1\le i\le Level(X)} p_i x_i$；信息段 CUB 用来存放节点 X 的 Pvu 值。这里，Pvu 表示在节点 X 所表示的状态下，可行解所能达到的可能价值的一个上界。也即是说，当 $x_1,\cdots,x_{l-1},x_l$ 的值确定后，可行解 $x_1,\cdots,x_l,x_{l+1},\cdots,x_n$ 所能达到的效益值的上界。类似地，当 $x_1,\cdots,x_{l-1},x_l$ 的值确定后，可行解 $x_1,\cdots,x_l,x_{l+1},\cdots,x_n$ 所能达到的最大效益值记做 Pvl。prev 是到目前为止能够探测到解的目标值的一个最大下界。如果节点 X 满足 Pvu<prev，则应该杀死节点 X（不放入节点表）。但是，当 Pvu = prev 时要谨慎处理。因为 prev 可能是某个答案节点的目标值，此时，由 Pvu=prev 可知沿此线路从节点 X 继续搜索下去不会得到更好的解，可以杀死节点 X。但是，prev 也可能只是一个纯粹的最大下界，此时 Pvu(X)=prev=Pvl(Y)，其中 Y 是已经放在节点表中的某个节点。因而，如果我们在一般节点 Z 处更新 prev 值时用 prev:=max(prev,Pvl(Z)) 这种做法，则 prev 的值可能是还没有产生具有这个目标值的解之前就知道了。此时如果在 Pvu(X)=prev 时贸然杀死节点 X 可能导致具有 prev 目标值的解夭折。为了区分上述两种情况，我们可以引进一个充分小的数 e，在一般节点 Z 处更新 prev 值时采取 prev:=max(prev,Pvl(Z)-e)这样的做法，那么，当出现 Pvu=prev 时，就知道 prev 是某个答案节点的目标值，可以杀死该节点。但是，为了不影响活节点的优先级，e 必须满足 Pvl(Z)<Pvl(Y)⇒Pvl(Z)<Pvl(Y)-e。采用上述处理后，我们可以使用规则：

当 Pvu ≤ prev 时杀死节点 X

可以杀死更多不必要搜索的节点。Pvu(X)可以作为优先级函数，在活节点表中，选择 Pvu 值大的节点作为当前扩展节点。关于 Pvl(X)和 Pvu(X)的计算将由一个子程序给出。

作为求最大值优化问题处理的优先队列式分枝限界法解 0/1 背包问题的程序 LFKNAP 采用了六个子程序：LUBound、NewNode、Finish、Init、GetNode 和 Largest。子程序 LUBound 计算 Pvl 和 Pvu 之用；NewNode 生成一个具有六个信息段的节点，给各个信息段置入适当的值，并将此节点加入节点表；Finish 打印出最优解的值和此最优解中 $x_i=1$ 的物品；Init 对可用节点表和活节点表置初值；GetNode 在程序开始取一个可用节点；Largest 在活节点表中取一个具有最大 Pvu 值的节点作为当前扩展节点。

程序 7-2-1　0/1 背包问题的优先队列式分枝限界算法

```
proc LFKNAP(P,W,M,N,e)//假定物品的排列顺序遵循
    //P[i]/W[i]≥P[i+1]/W[i+1];
    real P[1..N],W[1..N],M,CL,Pvl,Pvu,cap,cv,prev;
    integer ANS，X，N;
1.   Init; //初始化可用节点表及活节点表
2.   GetNode(E); //生成根节点
3.   Parent(E):=0; Level(E):=0; CC(E):=M; CV(E)=0;
4.   LUBound(P,W,M,0,N,1,Pvl,Pvu);
5.   prev:=Pvl-e; CLB(E):=Pvl; CUB(E):=Pvu; Tag(E):=0;
6.   Loop
7.     i:=Level(E)+1, cap:=CC(E), cv:=CV(E);
8.     case:
9.     i=N+1: //解节点
10.      if cv>prev then
11.        prev:=cv; ANS:=E;
12.      end{if}
13.    else: //E 是内部节点，有两个儿子
14.      if cap≥W[i] then //左儿子可行
15.        NewNode(E,i,1,cap-W[i],cv+P[i],CLB(E));
17.      end{if}
18.      LUBound(P,W,cap,cv,N,i+1,Pvl,Pvu);
19.      if Pvu>prev then //右儿子会活
```

```
20.            NewNode(E,i,0,cap,cv,Pvl);
21.            prev:=max(prev,Pvl-e);
22.        end{if}
23.      end{case}
24.      if 不再有活节点 then exit; end{if}
25.      Largest(E); //取一个活节点作为当前扩展节点
26.    until CUB(E)≤prev
27.    Finish(cv,ANS,N);
28.  end{LFKNAP}
```

算法中有两点值得注意：

1). 第 6～26 行的循环依次检查所生成的每个节点。此循环在以下两种情况下终止：或者活节点队列为空，或者为了扩展而选择的节点 E（扩展节点）满足 CUB(E)≤prev.在后一种情况下，由扩展节点的选法可知，对所有的扩展节点 X 均有 CUB(X)≤CUB(E)≤ prev，因而它们都不能导致其值比 prev 更大的解。

2). 在左儿子 X 可行的情况下，由 LUBound 算出它的上界、下界与 E 的相同，因而可不调用 LUBound 再计算一次。因为 CUB(E)>prev，所以将 X 加入活节点表。但是右儿子节点 Y 则不同，需要调用函数 LUBound 来获取 CUB(Y)= Pvu.如果 Pvu≤prev,则杀死节点 Y(即，不放在节点表中)。否则，将节点 Y 加入活节点表，并修改 prev 的值（第 21 行）。以下附上前面提到的几个子程序。

程序 7-2-2 计算节点状态下的可能取得最大效益值的上、下界

```
LUBound(P,W,cap,cv,N,k,Pvl,Pvu)// k 为当前节点的级（level），cap
  //是背包当前的剩余容量，cv 是当前背包中物品的总价值（已取得的效
  //益值），还有物品 k,…,N 要考虑
 Real rw; //随时记录本函数执行过程中是背包的剩余容量
 Pvl:=cv; rw:=cap;
 for i from k+1 to N do
   if rw<W[i] then Pvu:=Pvl+rw*P[i]/W[i];
     // 从第 k+1 件到第 N 件至少有一件物品不能装进背包的情形
     for j from i+1 to N do
       if rw≥W[j] then
           rw:=rw-W[j]; Pvl:=Pvl+P[j];
       end{if}
     end{for}
```

```
        return //此时 Pvl < Pvu
      end{if}
      rw:=rw-W[i]; Pvl:=Pvl+P[i];
    end{for}
    Pvu:=Pvl; // 从第 k+1 件物品到第 N 件物品都能装进背包的情形出现,
end{LUBound}
```

程序 7-2-3　程序生成新节点算法

```
NewNode(par,lev,t,cap,cv,ub) //生成一个新节点 J，并把它加到活节点表
  GetNode(J);
  Parent(J):=par; Level(J):=lev; Tag(J):=t;
  CC(J):=cap; CV(J):=cv; CUB(J):=ub;
  Add(J);
end{NewNode}
```

程序 7-2-4　打印答案程序

```
Finish(CV,ANS,N)//输出解
 real CV; global Tag,Parent;
 print( 'OBJECTS IN KNAPSACK ARE' )
    for j from N by -1 to 1 do
      if Tag(ANS)=1 then
        print(j);
      end{if}
      ANS:=Parent(ANS);
    end{for}
 end{Finish}
```

例子　已知 n=4, P=(10,10,12,18), W=(2,4,6,9), M=15. 试绘出算法 LFKNAP 求最优解的检索过程。

解：如图 7-2-1 所示，用红色数字标识解空间树中的各个节点。首先计算出节点 1 的上下界估值：Pvu(1)=38，Pvl(1)=32，并令 prev=32-e；因为 32>prev，此时，该节点的两个儿子节点 2，3 进入活节点表。节点 2 的上界估值最大，选为当前扩展节点。节点 2 的左儿子节点 4 的上下界估值与其父亲一致，右儿子节点 5 的上下界估值为：Pvu(3)=36，Pvl(3)=22。因为 36>prev，节点 4 进入活节

点表。在活节点表中，Pvu(4)=38 最大，节点 4 选作当前扩展节点。左儿子节点 6 的上下界估值与父亲节点相同；右儿子 7 的上下界估值为：Pvu(7)=38，Pvl(7)=38，因而 prev:=max{prev,38-e}=38-e。这两个节点都进入活节点表。选择节点 6 作为当前扩展节点。因为它的左儿子节点生成会破坏约束条件，所以不能放进活节点表；右儿子节点是答案节点，但获得的目标值 32 低于 prev 值，也不能进入活节点表。再选择节点 7 作为当前扩展节点，它的左儿子结点是答案结点，获得目标值 38，更新 Prev:=38。它的右儿子结点的上下界估值都是 20，因而不能进入活节点表。注意活节点表中的其它节点的上界估值 Pvu 都小于 Prev，所以不必再扩展了。程序结束，得到最优解(1,1,0,1)，最优目标值为 38。

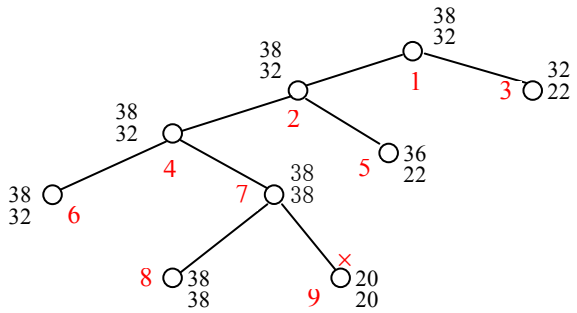
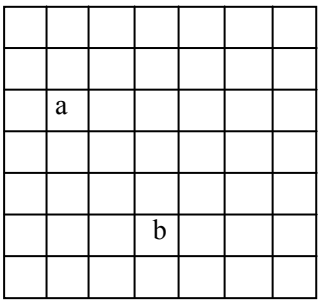


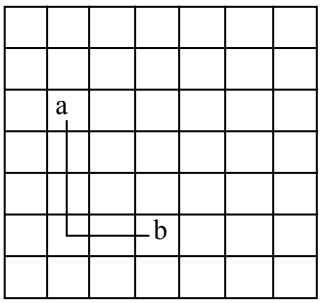
图 7-2-1 LCKNAP 的搜索过程

§3 电路板布线问题

印刷电路板将布线区域分成 n×m 个方格(阵列)，如下图(a).精确的电路布线问题要求确定连接方格 a 的中点到方格 b 的中点的最短布线方案。在布线时，电路只能沿直线或直角布线，如下图(b)所示。为了避免线路相交，已布了线的方格做了封锁标记，其它线路不允许穿过被封锁的方格。



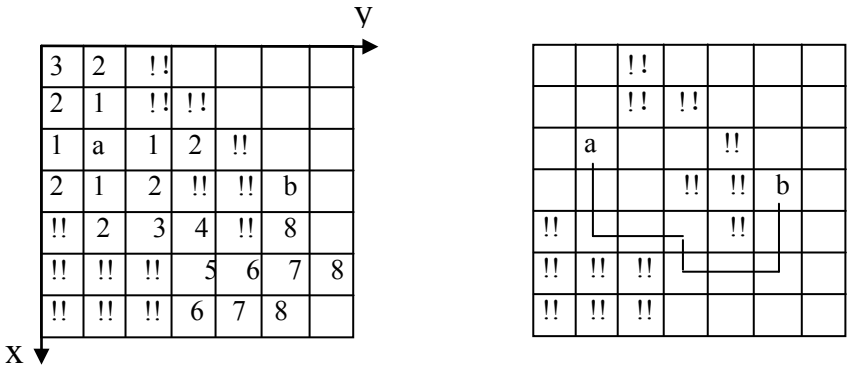
图（a）



图（b）

采用队列式(FIFO)分枝限界法解此问题，它的解空间树是一个多叉树。首先结点 a 作为第一个扩展结点。与该扩展结点相邻并且可达的方格成为可行结点被加入到活结点队列，这些结点被加入队列的顺序是：右、下、左、上。将这些

方格标记为 1，它表示从起始方格 a 到这些方格的距离为 1。接着从活结点队列中取出队首结点作为下一个扩展结点，并将与扩展结点相邻且未标记过的方格标记为 2，并以右、下、左、上的顺序将这些结点存放到活结点队列中。这个过程一直持续到算法搜索到目标方格 b 或者活结点队列为空时停止。注意到，搜索过程在遇到标记封锁的方格时，自动放弃此方格。下图是在 7×7 方格阵列中布线的例子。其中，起始点的位置是 a=(3,2)，目标位置是 b=(4，6).有!! 号的方格表示被封锁。搜索过程如下图(c)所示。



本例中，a 到 b 的最短距离是 9。要构造出与最短距离相应的最短路径，需从目标方格开始向起始方格回溯，逐步构造出最优解。每次向比当前方格标号小 1 的相邻方格移动，直至到达起始方格为止。图(d)给出了该例子的最短路径，它是从目标方格 b 移动到(5,6)，然后移至(6,6)，…,最终移至起始方格 a。

用 C++语言实现算法时，首先定义一个表示电路板上方格位置的类 Position，它有两个私有成员 row 和 col，分别表示方格所在的行和列。在电路板的任一方格处，布线可沿右、下、左、上四个方向进行，分别记为移动 0，1，2，3。offset[0].row=0 and offset[0].col=1 表示向右前进一步；offset[3].row=-1 and offset[3].col=0 表示向上前进一步。类似地讨论向下和向左前进的情况。一般用一个二维数组 grid 表示所给的方格阵列。初始时，grid[i][j]=0,表示该方格允许布线，而 grid[i][j]=1 表示该方格不允许布线（有封锁标记）。为了便于处理边界方格的情况，算法对所给的方格阵列的四周设置一道“围墙”，即增设标记为 1 的附加方格。

算法开始时测试初始方格与目标方格是否相同。若相同，则不必计算，直接返回最短距离 0。否则，算法设置方格阵列的围墙，初始化位移矩阵 offset。算法将起始位置的距离记为 2，这是因为数字 0 和 1 用于表示方格的开放或封闭状态，因而将所有的距离值都加 2。实际距离应为标记距离减 2。算法从起始位置 start 开始，首先标记所有标记距离为 3 的方格，并存入活结点队列，然后依次标记所有标记距离为 4，5，… 的方格，直至到达目标方格 finish 或活结点队

列为空时为止。

程序 7-3-1 布线问题的队列式分枝限界算法

```
bool FindPath(Position start,Position finish,int& PathLen,
            Position * &path)
{//计算从起点位置 start 到目标位置 finish 的最短布线路径，
 //找到最短布线路径则返回 true，否则返回 false
  if((start.row==finish.row) && (start.col==finish.col)
    {PathLen=0; return true;} //start=finish
 //设置方格阵列“围墙”
 for(int i=0; i<= m+1; i++)
   grid[0][i]=grid[n+1][i]=1; //顶部和底部
 for(int i=0; i<= n+1; i++)
   grid[i][0]=grid[i][m+1]=1; //左翼和右翼
//初始化相对位移
Position offset[4];
offset[0].row=0; offset[0].col=1;//右
offset[1].row=1; offset[1].col=0;//下
offset[2].row=0; offset[2].col=-1;//左
offset[3].row=-1; offset[3].col=0;//上
int NumOfNbrs=4;//相邻方格数
Position here,nbr;
here.row=start.row;
here.col=start.col;
grid[start.row][start.col]=2;
//标记可达方格位置
LinkedQueue<Position> Q;
do {//标记相邻可达方格
  for(int i=0; i<NumOfNbrs; i++){
     nbr.row=here.row + offset[i].row;
     nbr.col=here.col+offset[i].col;
    if(grid[nbr.row][nbr.col]==0){
     //该方格未被标记
        grid[nbr.row][nbr.col]=grid[here.row][here.col]+1;
        if((nbr.row==finish.row) &&
```

```
            (nbr.col==finish.col)) break; //完成布线
          Q.Add(nbr);}
        }
      //是否到达目标位置 finish?
      if((nbr.row==finish.row)
         (nbr.col==finish.col)) break;//完成布线
      //活结点队列是否非空?
      if(Q.IsEmpty()) return false;//无解
      Q.Delete(here);//取下一个扩展结点
    }while(true);
    //构造最短布线路径
    PathLen=grid[finish.row][finish.col]-2;
    path=new Position[PathLen];
    //从目标位置 finish 开始向起始位置回溯
    here=finish;
    for(int j=PathLen-1; j>=0; j--){
      path[j]=here;
      //找前驱位置
      for(int i=0; i<NumOfNbrs; i++){
         nbr.row=here.row+offset[i].row;
         nbr.col=here.col+offset[i];
         if(grid[nbr.row][nbr.col]==j+2) break;
       }
       here=nbr;//向前移动
     }
     return true;
  }
```

由于每个方格成为活结点进入活结点队列最多 1 次，活结点队列中最多只处理 $O(mn)$ 个活结点。扩展每个活结点需要 $O(1)$ 时间，因此共耗时 $O(mn)$。构造相应的最短距离需要 $O(L)$ 时间，其中，L 是最短路径的长度。

§4　优先级的确定与 LC-检索

对于优先队列式分枝限界法，节点优先级的确定直接影响着算法性能。我们当然希望具有下列特征的活节点 X 成为当前扩展节点：

1).以 X 为根的子树中含有问题的答案节点；

2).在所有满足条件 1)的活节点中，X 距离答案节点“最近”。

但是，要给可能导致答案节点的活节点附以优先级，必须要附加若干计算工作，即要付出代价。对于任一节点，与之相关的搜索成本可以使用两种标准来度量：

(i).在生成一个答案节点之前，子树 X 需要生成的节点数；

(ii).以 X 为根的子树中，离 X 最近的那个答案节点到 X 的路径长度。

容易看出，如果使用度量(i)，则对于每一种分枝限界算法，总是本着生成最小数目的节点去做；如果采用度量(ii),则要成为扩展节点的节点只是那些由根到最近的那个答案节点路径上的那些节点。用 c(•)表示最小搜索成本函数，它可以递归地定义如下：

a).如果 X 是答案节点，则 c(X)是解空间树中由根节点到 X 的搜索成本(即所用的代价，如深度、其它计算复杂度等)；

b).如果 X 不是答案节点，而且以 X 为根的子树中不含答案节点，则 c(X)应该定义为一个充分大的数，不妨以 ∞ 表示之；

c).如果 X 不是答案节点，但是以 X 为根的子树中含答案节点，则 c(X)应该是具有最小成本的答案节点的成本。

如果以最小成本函数作为优先级函数，则优先队列式分枝限界法将以最少的搜索成本找到问题的解。然而，这样的最小搜索成本函数往往是不易得到的，有时可能需要不亚于求解原问题的计算工作量。实际问题中都是采用一个成本估计函数 ，它由两部分决定：解空间树的根节点到 X 的搜索成本 f(X)，以及由 X 到答案节点的搜索成本 g(X)。

$$(X)= f(X)+g(X),$$

称为成本估计函数。

根据成本估计函数，选择下一个扩展节点的策略总是选取 (•)值最小的活节点作为当前扩展节点。这种搜索策略称为最小成本检索(Least Cost Search)，简称 LC-检索。如果取 g=0 且 f(X)等于 X 在解空间树中的深度，则 LC-检索即是宽度优先搜索(BFS)；如果 f=0，而且 g 满足：

$$Y \text{ 是 } X \text{ 的儿子} \Rightarrow g(Y) \le g(X) \qquad (7.4.1)$$

则 LC-检索即是深度优先搜索(DFS)。在以后所提到的成本函数 中，函数 g 都满足(7.4.1)式。

例子 7.4.1 15 迷问题

在一个 4×4 的棋盘上排列 15 块号牌，如图 7.4.1(a)。其中会出现一个空格。棋盘上号牌的一次合法移动是指将与空格相邻的号牌的一块号牌移入空格。15 迷问题要求通过一系列合法移动，将号牌的初始排列转换成自然排列，如图 7.4.1(b).

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

图（a）

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

图（b）

	#		#
#		#	
	#		#
#		#	

图（c）

如果将棋盘的各个位置按照号牌的自然排列发给序号，右下角发给 16 号。则号牌的每一种排列都可以看作是 1，…，15，16 这 16 个数的排列，其中，16 的位置代表空格。如图 7.4.1(a)对应的排列是

(1,3,4,15,2,16,5,12,7,6,11,14,8,9, 10,13)

事实上，不是号牌的每一种排列都能够经过一系列合法移动转换成自然排列的。用 Less(i)记排列 P 中位于 i 后面且号码比 i 小的号牌个数，则排列 P 可以经一系列合法移动转换成自然排列的充要条件是

$$\sum_{1\le i\le 16} Less(i)+\tau_0 \quad \text{是偶数} \qquad (7.4.2)$$

其中，当空格在图 7.4.1(c)的某个＃位置时，τ_0=1；否则τ_0=0. 以后记

$$\tau(P)=\sum_{1\le i\le 16} Less(i)$$

称为排列 P 的逆序数。例如，图 7.4.1(a)中排列的逆序数为 37，τ_0=0,所以图 7.4.1(a)中排列不能经一系列合法移动转换成自然排列。

在处理实际问题中，一般是根据具体问题的特性，确定成本估价函数

(X)= f(X)+g(X)

中的函数 f(X)和 g(X).在本例中，我们令 f(x)是由根到节点 X 的路径的长度，g(X)是以 X 为根的子树中，由 X 到目标状态(自然排列)的一条最短路径长度的估计值。为此，g(X)应是把状态 X 转换成目标状态所需的最少移动数目。对它的一种可能的选择是

g(X)= 排列 X 的不在自然位置的号牌的数目　　(7.4.3)

图 7.4.1(a)的排列中，不在自然位置的牌号分别是：3，4，15，2，5，12，7，6，14，8，9，10，13，共 13 个。g(X)=13。将图 7.4.1(a)中的排列转换成自然排列至少需要 13 次合法的移动。可见，成本估价函数　(X)是函数 c(X)的下界。以这样成本估价函数为优先级，则 15 迷问题解空间树的搜索过程可以从本章附页“15 迷问题解空间树”的图(a)中看出。

首先以节点 1 作为扩展节点，生成 4 个儿子：2，3，4，5。这 4 个儿子的成本估价分别为：　(2)=1+4；　(3)=1+4；　(4)=1+2；　(5)=1+4.因而节点 4 成为当前扩展节点，生成 4 个儿子，但有一个儿子同其祖父相同被舍弃。剩下的 3 个儿

子的成本估值分别为 (10)=2+1； (11)=2+3； (12)=2+3，此时，活节点表中共有 6 个节点：2，3，5，10，11，12，其中节点 10 的成本估值最小。故以节点 10 作为新的扩展节点。它能生成 3 个儿子，其中有一个因同其祖父相同而被舍弃。剩下的 2 个儿子是节点 22 和 23。但节点 23 所表示的排列是自然的，至此得到可行解。

如果采用队列式分枝限界法，即采用宽度优先搜索，势必将图 7.4.2(a)中的所有状态全部搜索。如果采用深度优先搜索，则图 7.4.2(b)中也只给出了一部分搜索步骤，而且节点 11 表示的号牌排列与自然排列还相距甚远。由此看出选择合适的优先级函数对于分枝限界法效率的影响甚大。

值得注意的是按照成本估价函数 (X)确定的优先级进行搜索，所得到的答案节点未必是最小成本答案节点。例如，在后面的图 7-4-3 所示的解空间树中，

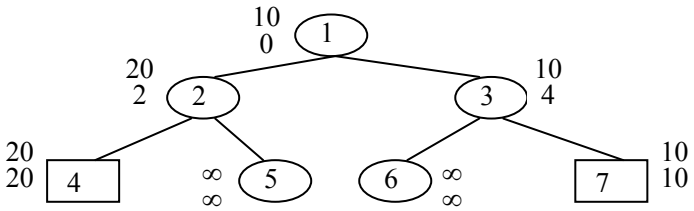


图 7-4-3 一棵假想的解空间树

每个节点 X 旁有两个数：上面的是最小成本 c(X),下面的是最小成本估价 (X)。整个解空间树中有两个答案节点。如果按照估价函数进行搜索，根节点 1 首先成为扩展节点，然后是节点 2，在扩展节点 2 得到一个答案节点 4，它是节点 2 的一个儿子。这个答案节点的成本是 20，比另一个答案节点 7 的成本大。

定理 7.4.1 在有限的解空间树中，如果对每对节点 X 和 Y 都有

“c(X) < c(Y)” => “ (X) < (Y)”

则按照最小成本估计函数搜索能够达到最小成本答案节点。

一般情况下，不易找到定理 7.4.1 中要求的成本估计函数。但是，对于成本估计函数一般有一个基本要求：

(X)≤ c(X)，对任意节点 X； (7.4.4)

(X)＝c(X)，当 X 是答案节点时。 (7.4.5)

在以下给出的算法中，要求搜索一直继续到一个答案节点变成扩展节点为止。

程序 7-4-1 搜索最小成本答案节点的 LC－检索

```
LCSearch(T,  ) //T 是问题的解空间树
```

```
E:=T; //第一个扩展节点
置活节点表为空;
loop
  if E 是答案节点 then
    输出从 E 到 T 的路径; return;
  end{if}
  for E 的每个儿子 X  do
    Add(X); Parent(X):=E;
  end{for}
  if 不再有活节点  then
    print( 'no answer node' ); stop;
  end{if}
  Least(E);
endloop
end{LeastCost}
```

其中，Least(X)是从活节点表中找一个具有最小　值的活节点，并从活节点表中删除该节点，再将该节点赋给变量 X；Add(X)是将新的活节点加到活节点表中。Parent(X)将活节点 X 与它的父亲相连接。在算法 LCSearch 中，由于答案节点 E 是扩展节点，所以，对于活节点表中的每个节点 P 均有　(E)≤　(P)。再由假设　(E)=c(E)，　(P) ≤ c(P)，得 c(E)≤ c(P)，即 E 是一个最小成本答案节点。

综上所述，采用优先队列式分枝限界法，其搜索解空间树的算法主要决定于三个因素：约束函数、限界函数和优先级函数。前两项主要是限制被搜索的节点数量，而优先级函数则是用来确定搜索的方向。但是，在处理实际问题中，未必把这三种函数全部列出来，特别是优先级函数往往同限界函数合为一体。以下给出处理最小值最优问题的 LC-分枝限界法的一般流程，这里“成本”就采用问题解的目标值。

程序 7-4-2　最小化问题的 LC-分枝限界算法

```
proc LCBB(T,  ,u,ε,cost) //假定解空间树 T 包含一个解节点 X 且
  // (X) ≤ c(X)≤ u(X)。c(X)是最小成本函数，一般都是未知的;  (X)是
  //最小成本估价函数，  u(X)是成本上界估计函数; cost(X)是 X 所对应的
  //解的成本。ε 是一个充分小的正数。
  E:=T; Parent(E):=0;
  if T 是解节点 then U:=min(cost(T),u(T)+ ε); ans:=T;
```

```
3      else U:=u(T)+ε; ans:=0; // 将活节点表初始化为空集;
4    end{if}
5    loop
6      for E 的每个儿子节点 X do
7        if  (X)<U && X 是一个可行节点 then
8           Add(X); Parent(X):=E;  //把 X 加入活节点 表
9           case:
10           X 是解节点 && cost(X)< U:
11             U:=min(cost(X),u(X)); ans:=X;
12           u(X) < U:
13             U:=u(X);
14          end{case}
15       end{if}
16     end{for}
17     if 不再有活节点 or 下一个扩展节点满足  ≥ U then
18       print( 'least cost=' ,U);
19       while ans≠0 do
20         print(ans); ans:=Parent(ans);
21       end{while}
22       return;
23     end{if}
24     Least(E);
25   end{loop}
27 end{LCBB}
```

这里我们将选择扩展节点的任务交给函数 Least(E)。所谓的“成本”是指目标函数的值。所以，当 X 是可行解的答案节点时，c(X)表示该可行解的目标函数值 cost(X)；当 X 是不可行节点时，c(X)=∞；当 X 是可行节点但不是答案节点时，c(X)表示以 X 为根的子树中**最小成本答案节点**的成本。此时，成本估价函数 (X)即变成对于节点 X 的目标函数值的一个估计。我们要求 (X) ≤ c(X)。算法中的 U 是一个上界值，开始时，U 可以取为一个充分大的数。**例如**，假定一个可行解需要确定 n 个数 $x_1,x_2,\cdots,x_n$，而每个取值 x_i 会使得成本开销增加 c_ix_i，于是，一个可行解 X=$(x_1,x_2,\cdots,x_n)$的总开销为 $c(X)=\sum_{1\le i\le n}c_ix_i$ 。

若当前只确定了前 k 个数 $x_1,x_2,\cdots,x_k$，用节点 X 表示当前之状态，则以这 k

个数为部分解的可行解的开销一定不小于$\hat{c}(X)=\sum_{1\le i\le k}c_i x_i$，不大于

$$u(X)=\sum_{k+1\le j\le n}c_j\bar{x}_j+\sum_{1\le i\le k}c_i x_i$$

其中$\bar{x}_j$是x_j可取的最大值。这样选取的函数 (X)和u(X)就会满足上面我们提出的要求：

$$(X)\ \le\ c(X)\le u(X)$$

从上面的例子可以看出：如果X是成本值为cost(X)的解节点，且cost(X)< U,则可用min{cost(X),u(X)}更新U的值；或者X不是解节点，但u(X)< U，可用u(X)更新U的值。如果U是某个可行解（或答案节点）的成本值（开销），则当 (X)≥U时，可以杀死节点X。但是，当U只是一个简单的上界，不是某个可行解的成本值时，应该谨慎考虑 (X)=U的情况。因为，如果X不是答案节点，则X的某个子孙Y可能是答案节点，以X为根的子树中含有答案节点。如果前面盲目地杀死节点X，则可能丢失一些最小成本答案节点。所以，在这种情况下，不能杀死节点X。为此，在用u(X)更新U值时，常常多加一个微量ε（ε的选取应当满足："u(X)<u(Y)⇒ u(X)+ ε < u(Y)"，即用 u(X)+ ε 更新U。于是，U的更新可以描述为：设E是当前扩展节点，X是E的一个儿子节点:

(1).当X是一个解节点，而且cost(X)< U时，可用min(cost(X),u(X)+ ε)来更新U;

(2).当X不是解节点，而且u(X)+ ε <U时，直接用u(X)+ ε 更新U。

§5　旅行商问题

本节采用最小成本分枝限界算法求解旅行商问题。设G=(V,E)是旅行商问题实例中定义的有向图，边(i,j)的成本为c_{ij}，若(i,j)∉E，则$c_{ij}=\infty$，图G的顶点数为|V|=n。不失一般性，假设每次环游都是从顶点1开始和结束：$(1,i_1,\cdots,i_{n-1},1)$。其状态空间树中每条从根节点到叶节点的一条路径唯一代表一条Hamilton回路，其成本是该回路上所有有向边的代价之和，这样的环游有(n-1)!个，如第六章第一节所指出的，它的状态空间树是一棵排列树。使用最小成本分枝限界算法，需要定义状态空间树中各节点的成本函数 c(X)，和上下界函数u(X)和ĉ(X)。成本函数可以如下定义：

$$C(X)=\begin{cases}\text{从根到X的路径所定义的回路成本，当X是叶节点时；}\\ \text{子树X中一个最小成本叶节点的成本，当X不是叶节点时。}\end{cases}$$

定义ĉ(X)为根节点到节点X的路径的成本，显然有C(X)≤ĉ(X)，但是，当X是叶节点时，ĉ(X)≠C(X)。实际上，我们可以用简约成本矩阵获得更好的ĉ(X)，它满

足ĉ(X)≤C(X)，而且在 X 是叶节点时，ĉ(X)=C(X)。矩阵 A 称为简约的，如果它的每个元素都是非负的，而且它的每一行（列）都至少有一个元素是 0。我们可以将旅行商问题的成本矩阵(即表示它的有向赋权图的邻接矩阵)约化成简约矩阵：先将矩阵的每一行各元素减去该行的最小元素，再将所得矩阵的每一列各元素减去该列的最小元素。因为有向图的一条有向 hamilton 回路含有每个顶点 i 的恰好一条出边(i,j)和一条入边(k,i)，从成本矩阵的一列或者一行的每一个项减去一个常数 t，就会使得每条回路成本正好减少 t。这不改变最低成本回路，即最小成本回路仍然是最小成本回路。如果 i 行使用的减数是 r_i，j 列使用的减数是 c_j，则旅行商问题的最小回路成本不会低于这些数的和：

$$\hat{c}=\sum_{i=1}^{n} r_i+\sum_{j=1}^{n} c_j$$

因此，$\hat{c}$ 可以作为状态空间树中根节点的下界估值，上述简约矩阵称为根节点的简约矩阵。对于其它的节点 S，设 R 是它的父亲节点，状态空间树中边(R,S)对应 Hamilton 回路中包含的边(i,j)。如果 S 不是叶节点，S 的简约矩阵 A_S 可以通过修简 R 的简约矩阵 A_R 而得：(1)将 A_R 中 i 行和 j 列的所有项都改为∞，防止任何其它离开顶点 i 的边，进入顶点 j 的边的使用。(2)将 A_R 的(j,1)元素置为∞，防止使用边(j,1)。(3)约简经过(1)、(2)两步操作后得到的矩阵。上面三步操作后得到的矩阵就作为节点 S 的简约矩阵，并且给出节点 S 的下界估值如下：

$$\hat{c}(S)=\hat{c}(R)+A_R(i,j)+\tilde{c}$$

其中，$A_R(i,j)$ 是矩阵 A_R 的 (i,j) 元素，$\tilde{c}$ 是约简步骤(3)施行时减掉的总数。如果 S 是叶节点，则直接计算 $\hat{c}(S)=c(S)$ 即可，不必计算伴随矩阵。

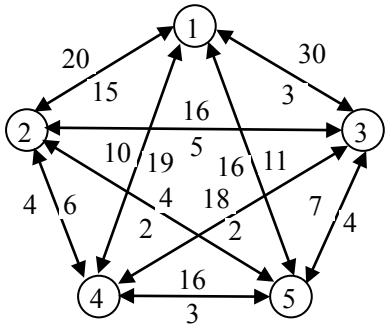
每个上界估值可用 u(x)=∞。

考察下面成本矩阵 A 描述的旅行商问题，对应有向赋权图如图 7-5-1 所示，它的简约矩阵为 A(1)。采用最小成本分枝限界算法，搜索最优解的过程如下，产生的状态空间树由图 7-5-2 表示。

$$A=\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix} \qquad A(1)=\begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$$

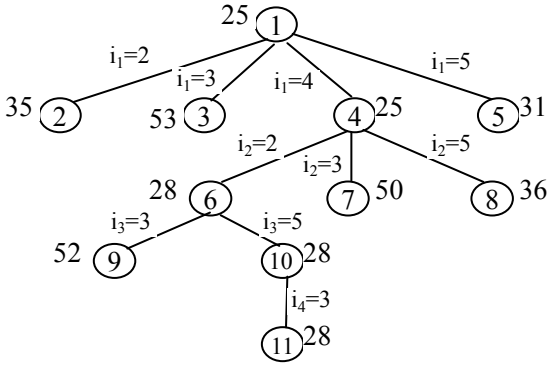
令 Prev 表示目前所知的最小回路成本，初始时 Prev=∞。根节点 1 的简约矩阵就是成本矩阵的简约矩阵 A(1)，而ĉ(1)=25。从根节点作为可扩展节点开始，

依次序产生节点 2,3,4,5。对应这些节点的简约矩阵分别是 A(i),i=2,3,4,5。例如,简约矩阵 A(3)对应的路径是有向图的顶点 1 到顶点 3,该矩阵是这样获得的:将矩阵 A(1)的第 1 行、第 3 列所有项及(3,1)项都置为∞，这样得到的矩阵不是简约的，但只有第 1 列没有零，将第 1 列的每项都减去 11，得到简约矩阵 A(3)。此时，节点 3 的成本下界估值为ĉ(3)=25+17+11=53。



由小到大权值标于外侧,由大到小标于内侧

图 7-5-1 有五个顶点的赋权有向图



节点外面的数字是ĉ 值

图 7-5-2 用 LC-搜索产生的状态空间树

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$$

A(2),路径 1,2

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{pmatrix}$$

A(3),路径 1,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

A(4),路径 1,4

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix}$$

A(5),路径 1,5

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

A(6),路径 1,4,2

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{pmatrix}$$

A(7),路径 1,4,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

A(8),路径 1,4,5

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{pmatrix}$$

A(9),路径 1,4,2,3

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

A(10),路径 1,4,2,5

其它三个节点的成本下界估值分别为：ĉ(2)=35，ĉ(4)=25，ĉ(5)=31。节点 4 选为当前扩展节点，产生子节点 6,7,8，它们的成本下界估值分别为：ĉ(6)=28,

ĉ(7)=50，ĉ(8)=36。现在活节点表中有节点 2,3,5,6,7,8，节点 6 被选为当前扩展节点，生成其子节点 9,10，它们的成本下界估值分别是ĉ(9)=52，ĉ(10)=28，活节点表中的节点 10 称为当前扩展节点。它只有一个子节点 11，而且是解节点，其成本下界估值即是其成本值ĉ(11)=c(11)=28。更新 Prev 的值：Prev:=28。注意到目前活节点表中所有节点的成本下界估值都大于 Prev，所以，不需要继续扩展，程序执行至此结束。得到该旅行商问题的最低成本环游：(1,4,2,5,3,1)，成本为 28。

习题 七

1. 假设对称旅行商问题的邻接矩阵如图 1 所示，试用优先队列式分枝限界算法给出最短环游。画出解空间树的搜索图，并说明搜索过程。

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ & \infty & 16 & 4 & 2 \\ & & \infty & 6 & 7 \\ & & & \infty & 12 \\ & & & & \infty \end{pmatrix}$$

图 1 邻接矩阵

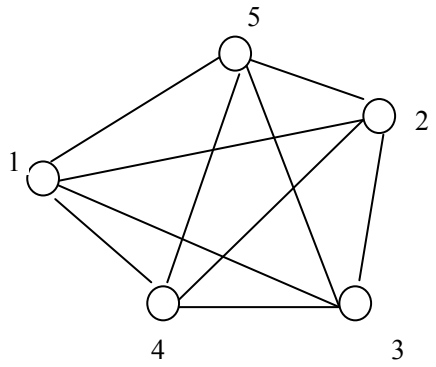
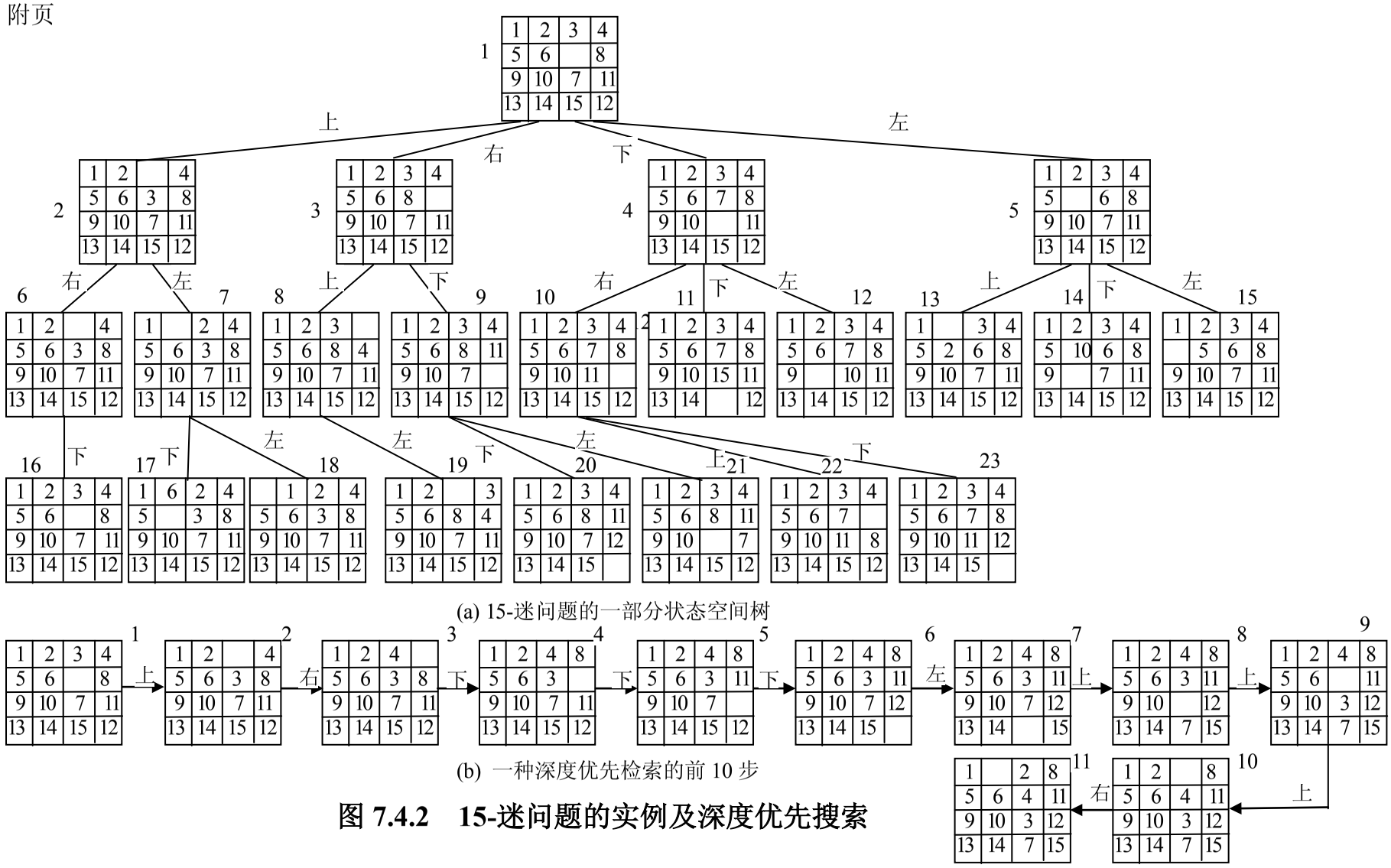


图 2 旅行商问题

2. 试写出 0/1 背包问题的队列式分枝限界算法程序，并找一个物品个数是 16 的例子检验程序的运行情况。
3. 最佳调度问题:假设有 n 个任务要由 k 个可并行工作的机器来完成，完成任务需要的时间为 t_i。试设计一个分枝限界算法，找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最短。

附页



(a) 15-迷问题的一部分状态空间树

(b) 一种深度优先检索的前 10 步

图 7.4.2　15-迷问题的实例及深度优先搜索

第八章　NP-完全问题

§1　关于问题及算法的描述

为了应用算法复杂性理论，首先要对问题、问题的一般描述、计算模型、算法、算法的复杂性给出严格的定义。但在给出精确定义之前，我们先回顾一下有关概念的粗略解释。

所谓一个**问题**(problem)是指一个有待回答、通常含有几个取值还未确定的自由变量的一个一般性提问(question)。它由两部分构成：一是对其关于参数的一般性描述；二是对该问题的答案所应满足的某些特性的说明。而一个问题的某个**实例**则可通过指定问题中所有参数的具体取值来得到。以下用Π表示某个问题，用I表示其实例。

旅行商问题的参数是由所需访问城市的一个有限集合$C=\{C_1,C_1,\cdots,C_m\}$和C中每对城市C_i,C_j之间的距离$d(C_i,C_j)$所组成。它的一个解是对所给城市的一个排序

$$C_{\pi(1)},C_{\pi(2)},\cdots,C_{\pi(m)}$$

使得该排序极小化下面表达式（目标函数）的值

$$\sum_{i=1}^{m-1}d(C_{\pi(i)},C_{\pi(i+1)})+d(C_{\pi(m)},C_{\pi(1)})$$

旅行商问题的一个**实例**是通过指定城市的数目，并指定每两个城市之间的具体距离而得到的。例如：

$$C=\{C_1,C_2,C_3,C_4\},\quad \begin{array}{l} d(C_1,C_2)=10,d(C_1,C_3)=5,d(C_1,C_4)=9,\\ d(C_2,C_3)=6,d(C_2,C_4)=9,d(C_3,C_4)=3 \end{array}$$

就是旅行商问题的一个实例，这个实例的一个最优解是排序C_1,C_3,C_4,C_2，因为四个城市的这个排序所对应旅行路线是所有可能环游路线中长度最小的，其长度为27。

目前广泛采用的描述问题的方法主要有两种：一是将任一问题转化为所谓的可行性检验问题(feasibility problem)；二是把问题转化为判定问题(decision problem)。实际中几乎所有问题都可直接或间接地转述为判定问题。

判定问题是答案只有“是”与“非”两种可能的问题。一个判定问题Π可简单地由其所有例子的集合D_Π与答案为“是”的例子所构成的子集$Y_\Pi\subset D_\Pi$来刻画。不过，为了反映实际问题所具有的特性，通常所采用的描述方法由两部分组

成。第一部分用诸如集合、图、函数等各种描述分量来刻画判定问题的一般性例子，以下用“例”表示这一部分；第二部分则陈述基于一般性例子所提出的一个“是非”提问，以下简称“问”。因此，一个具体例子属于D_Π当且仅当它可通过用具有特定性质的某些对象替代一般性例子的所有一般性描述分量而得到，而一个例子属于Y_Π当且仅当限定于该例子时，它对所述提问的回答为“是”。

例 待访问城市的有限集合$C=\{C_1,C_2,\cdots,C_m\}$、每对城市$C_i,C_j\in C$之间的距离$d(C_i,C_j)\in Z^+$以及一个界$B\in Z^+$。

问 在C中存在一个总长不超过B的、通过所有城市的旅行路线吗？也就是说，存在C的一个排序$C_{\pi(1)},C_{\pi(2)},\cdots,C_{\pi(m)}$，使得

$$\sum_{i=1}^{m-1} d(C_{\pi(i)},C_{\pi(i+1)})+d(C_{\pi(m)},C_{\pi(1)})\le B$$

这是一个将优化问题转化成判定问题的例子。一般地，一个优化问题可以看作是其所有实例的集合，而每一个实例为一个元素对(F,c)，其中F是可行解集，c是目标函数。一个最优化问题可以提出下述三种模式：

- ✓ 最优化模式：求最优解；
- ✓ 求值模式：求出最优值；
- ✓ 判定模式：给定一个实例(F,c)和一个整数L，问是否存在一个可行解$f\in F$，使得$c(f)\le L$？

显然，在求解最优值不太困难的假设下，上述三种模式的每一种都不比前一种困难。一般而且比较现实的假设是：最优值是一个整数，且这个整数或其绝对值的对数被输入长度的一个多项式所界定。在这种情况下，用二分搜索(或叫折半搜索)技术证明，只要判定模式存在多项式时间算法，求值模式也存在多项式时间算法。

所谓**算法**(algorithm)是指用来求解某一问题的、带有一般性的一步一步的过程。它是用来描述可在许多计算机上实现任一计算流程的抽象形式，其一般性可以超越任何具体实现时的细节。

注意，复杂性理论中对算法的定义与我们通常理解的具体算法(即用某种计算机语言编写的、可在某一特定计算机上实现的计算机程序)有所不同。不过，将算法想象为某个具体的计算机程序，在许多情况下可以帮助我们理解有关概念和理论。

附：一个算法的严格定义直到1936年才出现，丘奇(Alonzo Church)和图灵(Alan Turing)分别在他们的文章中给出。丘奇使用称为λ－演算的记号系统来定义算法，图灵用机器(图灵机)来定义算法，后来证明两者是等价的。此前，希尔伯特的第10问题就是要设计一个算法来测试多元多项式是否有整数根。不过他不用“算法”这个词，而是用一句短语：“通过有限次运算就可以决定的过程”。我们这里采用图灵的定义,即借用图灵机计算模型来给出算法的精确定义。

到目前为止，关于算法的描述大致有三种不同的程次：一是形式描述，即详尽的写出图灵机的状态、转移函数等等，这是最底层也最详细的描述；二是实现描述，使用日常语言来描述图灵机的运行，如怎样移动读写头，怎样在带上存储数据等，但是不给出状态和转移函数的细节；三是高层的描述，它也使用日常语言来描述算法，但是忽略实现的模型，这种描述不需要提及机器是怎样管理它的带子或读写头的。

称一个算法求解一个问题Π，是指该算法可以应用到Π的任一实例I，并保证能找到该实例的一个解。一个算法的有效性可以用执行该算法所需要的各种计算资源的量来度量。最典型也是最主要的两个资源就是所需要的时间和内存空间。但时间需求常常是决定某一特定算法在实际中是否真正有用和有效的决定性因素。

应该注意到，由于计算机速度和指定系统的不同，同一算法所需时间的多少随着计算机的不同可能有很大差别。为使算法分析具有一般性和在实际中有用，所给时间的度量不应该依赖于具体计算机，即是说，如果它们用不同的编程语言来描述，或在不同的计算机上运行，好的算法仍然保持为好的。另一点需要注意的是，即使同一算法和同一计算机，当用它来求解某一问题的不同例子时，由于有关参数取值的变化，使得所运行时间也有很大差别。对于前一个问题的解决，人们提出了一些简单但又能反映绝大多数计算机的基本运作原理的计算模型，如各种形式的图灵(Turing)机、随机存储机(RAM)等，然后基于这些计算模型来研究算法。对于第二个问题的解决，人们引进了问题例子大小(size)的概念。所谓一个问题例子的大小是指***为描述或表示它而需要的信息量***。只要表示的合适，可望例子的大小的值应该与求解的难易程度成正比。并称相应的表示法为编码策略(encoding scheme)。

事实上，作为输入提供给计算机的对任一问题例子的描述可以看作是从某一有限字母表中选取所需的字符而构成的有限长字符串。通常称该字母表中的字符为码，而由其中之字符如何组成描述问题例子的字符串的方法则称为编码策略。合理的编码策略应该具有可解码性(decodability)和简洁性(conciseness)。一种典型的方法就是利用所谓的结构化字符串，通过递归、复合的方式来给出所考虑问题的合理编码策略。

给定一个问题Π，假设已经找到描述问题一般性例子的一个合理编码策略e，则对Π的任一实例I，称其依编码策略 e 所得的描述相应问题实例的字符串中所含有字符的个数为其输入长度，并将该输入长度的具体值作为例子I的大小的正式度量。例如，若用字母表

$$\{C,[,],/,0,1,2,3,4,5,6,7,8,9\}$$

中的字符表示旅行商问题的例子，则前面所给该问题的具体例子可以用字符串

$$"C[1]C[2]C[3]C[4]//10/5/9//6/9//3"$$

来描述，其相应的输入长度为 32。我们说旅行商问题的这个例子的大小为 32。

虽然所有的判定问题均可用统一的术语来描述，但从数学上讲还是不够严密，不宜于给出算法的严格定义，也不利于算法复杂性研究。为了克服这些不足，我们引进“语言”这一术语，它与判定问题有着很自然的联系。

设Σ是一个字符集，用Σ^*表示由Σ中的字符组成的所有有限长字符串的集合。Σ^*的任何非空子集L都称为Σ上的一个语言(language)。例如$\{01,001,111,1010110\}$就是字符集$\{0,1\}$上的一个语言。判定问题与语言之间的对应关系是通过编码策略来实现的。一旦选定了某个字符集Σ，则对于任一个判定问题Π及其编码策略 e，Π和 e 将会把Σ^*中的所有字符串划分为三部分：那些不是Π的例子的编码、那些对Π的回答为“非”的例子的编码和那些对Π的答案为“是”的例子的编码。后一类编码正是与Π通过 e 来联系的语言。定义：

$$L[\Pi,e]=\left\{x\in\Sigma^* \mid \Sigma\text{ 为 e 所使用的字符集，}\ x\text{ 为某个例子 }\mathrm{I}\in Y_\Pi\text{ 在 e 下的编码}\right\}$$

如果一个结论对语言$L[\Pi,e]$成立，则说它在编码策略 e 下对问题Π成立（计算复杂性理论所直接考虑的是对语言或字符串集的分析）。

对于任何两个合理的编码策略 e 和 e’，问题Π的某个性质要么对$L[\Pi,e]$和$L[\Pi,e']$均成立，要么对二者均不成立。因此，可以直接说某个性质对Π成立或不成立，也常常将$L[\Pi,e]$简记为$L[\Pi]$。但是，这样的省略却失去了对输入长度的具体确定办法，而我们还需要象这样的一个参变量以便能确切表述复杂性的概念。为此，以后总是隐含假定：

对每个判定问题Π，均有一个不依赖于编码方式的函数

$$\text{Length：}\ D_\Pi \to Z^+$$

该函数的值将与从任一合理的编码策略所得的关于Π的任一例子的输入长度多项式相关。这里，多项式相关是指：对于Π的任一合理编码策略 e，都存在两个多项式p和q，使得如果$\mathrm{I} \in D_{\Pi}$且x为I在 e 下的编码，则有

$$\mathrm{Length[I]} \le p(|x|) \text{ 且 } |x| \le q(\mathrm{Length[I]})$$

这里，$|x|$表示字符串x的长度。易知，Π的任何两个合理的编码策略将导出多项式相关的输入长度。例如，对于旅行商问题，对应的判定问题的任一例子I，可以定义

$$\mathrm{Length[I]} = m + \lceil \mathrm{lb}B \rceil + \max\left\{ \lceil \mathrm{lb}d(C_i, C_j) \rceil \mid C_i, C_j \in C \right\}$$

这里，$\mathrm{lb}x$表示$\log_2 x$。

在后面的陈述中，会交替地使用(判定)问题、语言这两个术语而不加区分。

§2　图灵机与确定性算法

为了算法复杂性分析具有普适性，即一个算法的效能与具体的计算机无关，我们需要选定一种可用于描述计算的计算模型。第一个给出这种模型的是英国数学家图灵，后人称他所提出的模型为图灵机。**图灵机本质上是**一个具有存储载体（通常用一个有无限多个方格线性带表示）的、按照具体指令可完成向左或右移动、放置标记、抹去标记以及在计算终止时停机等四种基本操作的、**用于描述算法的语言**。在原理上，与我们用于和计算机交流的更为复杂的各种程序语言同样有力。由于其简单性，图灵机及其各种变形已被广泛用于计算复杂性的理论研究。

首先选择确定性单带图灵机(deterministic one-tape Turing machine)，简称确定图灵机，记为 DTM。一个 DTM 由一个有限状态控制器、一个读写头和一条双向的具有无限多个格的线性带所组成。

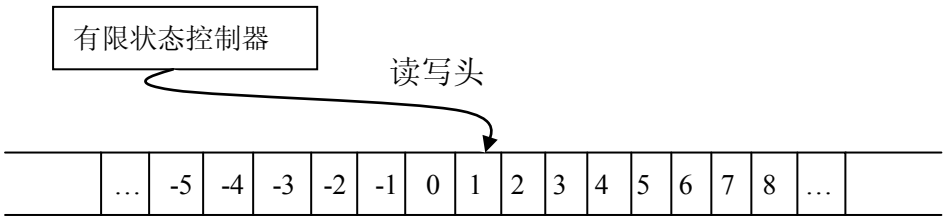


图 8-2-1 单带确定性图灵机示意

一个 DTM 程序(program)应详细规定下列信息：

(a). 带中所用字符的一个有限集合Γ，它应包含输入字符表子$\Sigma \subset \Gamma$和一个特别的空白符号 $b \in \Gamma - \Sigma$；

(b). 一个有限状态集Q，它包括初始状态q_0和两个特有的停机状态q_Y和q_N。

(c). 一个转移函数$\delta:(Q-\{q_Y,q_N\})\times\Gamma \to Q\times\Gamma\times\{l,r\}$

$$(q,s)\mapsto(q',s',\Delta)$$

表示了状态变化、字符变化及读写头移动方向，而且是唯一的。

一个 DTM 程序运行很简单。假设对 DTM 的输入为字符串$x \in \Sigma^*$，则该字符串首先被一格一个字符地存放在带格 1 到带格$|x|$中。所有其它的带格均存放空白字符b。该程序从初始状态q_0开始它的运算，并且，读写头先位于带格 1，然后按下述规则一步一步地进行：

若当前状态q为q_Y或q_N，则计算终止，且如果$q=q_Y$，就回答“是”，否则回答“非”。

若当前状态为$q \in Q-\{q_Y,q_N\}$，且$s \in \Gamma$为读写头当前扫描带格中的字符，而转移函数此时对应的取值为$\delta(q,s)=(q',s',\Delta)$，那么，该程序将执行这样的几个操作：读写头抹去当前带格中的s，并写上字符s'；同时，如果$\Delta=l$，则读写头左移一格；如果$\Delta=r$，则读写头右移一格。最后，有限状态控制器将从状态q变到状态q'。这样就完成了程序的一步计算，并为下一步计算做好准备，除非已处于停机状态。

可见，当前状态、带格的内容以及读写头所在的位置是图灵机的要素，这三者的整体称为一个**格局**，图灵机的运行就是根据转移函数发出的指令从一个格局转移到另一个格局。

有了 DTM 这个计算模型以及定义于它上面的程序概念，就可以给出算法及其复杂性函数的严格定义。设 M 是一个 DTM 程序，输入字符表为Σ。我们说程序 M 接受字符串$x \in \Sigma^*$，如果它作用于x时停机于状态q_Y，并称

$$L_M=\{x\in\Sigma^* \mid M \text{ 接受 } x\}$$

为程序 M 所识别的语言。因此，若 $x \in \Sigma^* - L_M$，则 M 的计算要么停机于状态 q_N，要么永不停机（不是该问题的实例）。只有当一个 DTM 程序对定义于其输入字符表上的所有可能字符串均（在有限步内）停机时，才称其为一个**算法**（实际是判定器）。相应地，称一个 DTM 程序 M 在编码策略 e 之下求解判定问题 Π，如果 M 对定义于其输入字符表上的所有输入字符串均停机，且有

$$L_M = L[\Pi, e]。$$

一个DTM程序M对于输入 x 的计算时间定义为该程序从开始直至进入停机状态为止所运行的步数。由此可以给出时间复杂性函数的定义：对于一个对所有输入 $x \in \Sigma^*$ 均停机的 DTM 程序 M，其时间复杂性函数 $T_M : Z^+ \to Z^+$ 定义为：

$$T_M(n) = \max\{m \mid \text{存在一个 } x \in \Sigma^*, |x| = n, \text{使得 M 对输入 } x \text{ 的计算需要时间为} m\}$$

若存在一个多项式 p，使得对所有的 $n \in Z^+$，有 $T_M(n) \le p(n)$，则称程序 M 为一个多项式时间 DTM 程序。我们称

$$P = \{L \mid \text{存在一个多项式时间 } DTM \text{ 程序 } M\text{，使得 } L = L_M\}$$

为 P 语言类。如果存在一个多项式时间 DTM 程序，它在编码策略 e 之下求解 Π，即 $L[\Pi, e] \in P$，则称该判定问题 Π 属于 P。

§3　NP 类问题

不难看出，上面定义的 P 类语言只能用来描述那些存在有效算法（多项式时间）的问题。然而，在实际中存在许多别的重要问题，对于它们，至今尚未找到有效的求解算法。其中有一大类这样的问题，虽然不知道求解它们的有效算法，但是，一旦通过某种办法给出了其答案的一个猜测或估计，就能设计出一个多项式时间算法来验证其真实性(称为多项式时间可验证性)。这类问题的分析和描述需要借助另一类图灵机作为计算模型。

非确定性单带图灵机(non-deterministic one-tape Turing machine)，简记为 NDTM，是一种假想的机器。通常有两种方式描述它：**多值模型**和**猜想模块模型**。

多值模型认为它和确定性图灵机的共同之处是也包括：

(a). 带中字符集 Γ，使得 $\Sigma \subset \Gamma$，且 $b \in \Gamma - \Sigma$；

(b). 有限状态集 $Q \supseteq \{q_Y, q_N, q_0\}$；

不同之处在于

(c). 多值转移函数 $\bar{\delta}:(Q\setminus\{q_Y,q_N\})\times\Gamma\to 2^{Q\times\Gamma\times\{l,r\}}$,

$$(q,s)\mapsto S\subseteq Q\times\Gamma\times\{l,r\}$$

确定性图灵机在任一状态下只能做一种运算，而非确定性图灵机可以被想象为在同一时刻能够独立、并行地完成多种运算（表现在转移函数的多值性），这显然不现实。

通过允许作不受限制的并行计算可以对不确定性算法做出明确的解释。每当要作某种选择时，算法就好像给自己复制了若干副本，每种可能的选择有一个副本，于是，许多副本同时被执行。第一个获得成功的副本将引起对其它副本计算的结束。如果一个副本获得不成功的完成则只该副本终止。

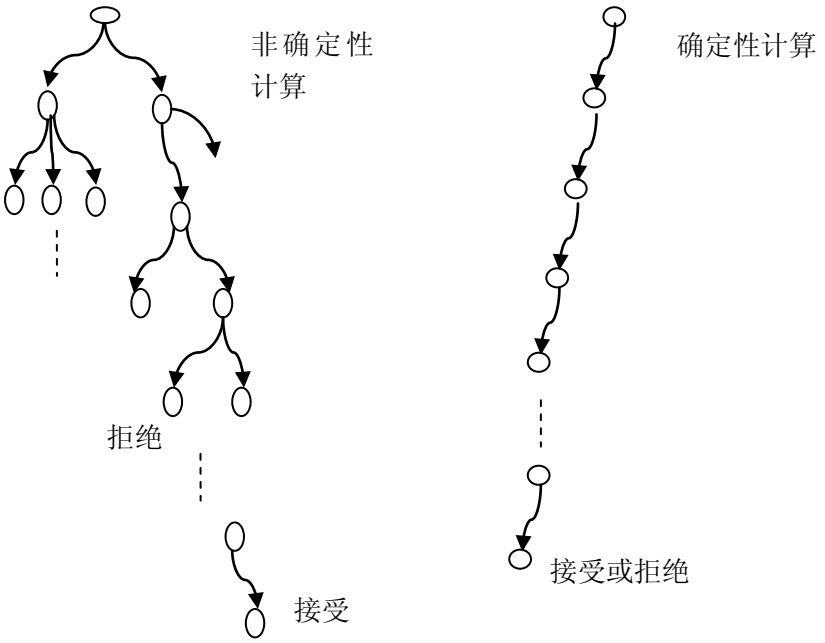


图 8-3-1 与确定性算法的比较

不确定算法举例：

程序 8-3-1 不确定性排序算法

```
pro NSort(A,n)
  //对 n 个正整数排序
  integer A(n),B(n),n,i,j;
  B:=0 //对 B 进行初始化
  for i to n do
     j:=choice(1..n);
     if B[j]≠0 then failure;  end{if};
     B[j]:=A[i];
  end{for}
  for i to n-1 do //验证 B 的次序
```

```
      if B[i]>B[i+1] then failure;  end{if};
    end{for}
    print(B);
    success;
  end{NSort}
```

程序 8-3-2　0/1 背包判定问题的不确定算法

```
proc NPack(P,W,n,M,R,X)
  integer P(n),W(n),R,X(n),n,M,i;
  X:=0 //对 X 进行初始化
  for i to n do
     X[i]:=choice(0,1);
  end{for};
  if ∑_{1≤i≤n} w(i)X(i) > M or ∑_{1≤i≤n} P(i)X(i) < R then failure;
  else success;
  end{if};
end{NPack}
```

“**猜想模块模型**”是另一种更形象、直观的解释方法。可将 NDTM 描述成：除多了一个猜想模块（guessing module）外，NDTM 与 DTM 有着完全相同的结构，而这个猜想模块带有自己的仅可对带进行写操作的猜想头，它提供写下猜想的方法，仅此而已。

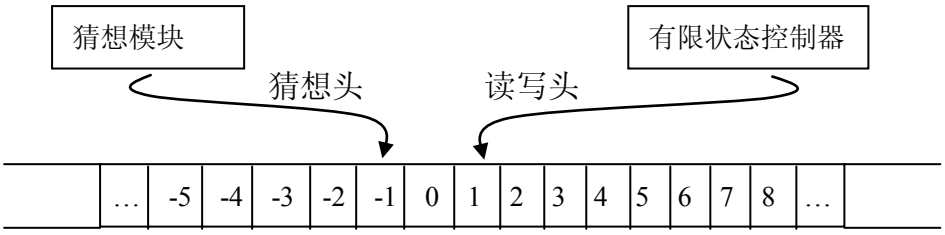


图 8-3-2　NDTM 猜想模块模型示意图

基于这一模型，一个 NDTM 程序可以类同于一个 DTM 程序的方式来进行定义，并用相同的记号（包括带中字符集Γ，输入字符表Σ，空白符号b，状态集Q，初始状态q_0，两个停机状态q_Y和q_N，以及状态转移函数

$$\delta:(Q\setminus\{q_Y,q_N\})\times\Gamma\to Q\times\Gamma\times\{l,r\}$$

但对于一个输入$x\in\Sigma^*$，NDTM 程序的计算过程却与 DTM 程序的计算过程不同，它把计算过程分为两个阶段，即**猜想阶段**、**检验阶段**。

第一阶段为猜想阶段。开始时，输入字符串x被写在由格 1 到格$|x|$的带上，其余带格为空白字符。读写头将扫描带格 1，而猜想头在扫描带格 -1，有限状态控制器处于不起作用状态，猜想模块处于起作用状态，并一次一步地指示猜想头：要么在被扫描的带格中写下Γ中的某一个字符并左移一格；要么停止。若为停止，猜想模块即变为不起作用的，而有限状态控制器变为起作用的并处于状态q_0。猜想模块是否保持起作用，以及起作用时该从Γ中选择哪个字符写在带格中，均由猜想模块以某种完全任意的方式决定。

当有限状态控制器处于状态q_0时，检验阶段就开始了。从此时起，计算机就在该 NDTM 程序的指示下，按照与 DTM 程序完全相同的规则进行。而猜想模块及其猜想头在完成了将所猜字符串写到带上的任务后将不再参与到程序的执行中去。当然，在检验阶段，前面所猜字符串可能会被，而且通常将被考察。当有限状态控制器进入到两个停机状态之一时，计算就会停止。若停机为q_Y，则说是一个可接受的计算，否则（包括永不停机），说是一个未接受的计算。

一般说来，对于一个给定的输入字符串x，NDTM 程序 M 将会做无限多个可能的计算，对每一个Γ^*中的可能猜想串都有一个相应的计算。如果这些计算中至少有一个为可接受的计算，则称 NDTM 程序 M 接受x，相应地 M 所识别的语言为

$$L_M = \left\{ x \in \Sigma^* \mid M \text{ 接受 } x \right\}$$

一个 NDTM 程序 M 接受$x \in L_M$的时间定义为：在 M 对于x的所有可接受计算中，程序从一开始直到停机状态q_Y为止在猜想和检验阶段所进行的步数的最小值。而 M 的时间复杂性函数$T_M : Z^+ \to Z^+$则定义为：

$$T_M(n) = \max\left[\{1\} \cup \left\{ m: \begin{array}{c} \text{存在一个长度为 } n \text{ 的 } x \in L_M, \text{使得} \\ \text{M 要接受它所需的时间为 } m \end{array} \right\} \right]$$

若存在多项式p，使得对所有的$n \geq 1$有$T_M(n) \leq p(n)$，则称 NDTM 程序 M 为一个多项式时间 NDTM 程序。并称

$$NP = \left\{ L \mid \text{存在一个多项式时间 } N\text{DTM 程序 M，使得 } L_M = L \right\}$$

为 NP 语言类。称判定问题Π在编码策略 e 下属于 NP，若$L[\Pi, e] \in NP$。与 P 类语言时的讨论一样，只要编码策略是合理的，就可以简单地称问题Π属于 NP。

因为任何现有的计算模型都可以通过加上一个类似于 NDTM 中的带有只写头

的猜想模块来扩充，然后相对于所得的模型重新描述上述的正式定义，而且所有如此得到的计算模型在多项式时间内可相互转换的意义下将是等价的。因此，也没有必要特别提及 NDTM 模型，我们将简单地说“**多项式时间不确定性算法**”，并将 NP 类语言与所有可用多项式时间不确定性算法求解的判定问题等同看待。

例子：考虑无向图的团问题：

例：给定一个有 n 个顶点的无向图 $G=(V,E)$ 和一个整数 k 。

问：G 是否包含一个具有 k 个顶点的完全子图（团）？

如果用邻接矩阵表示图 G，用二进制串表示整数 k，则团问题的一个实例可用长度为 $m=n^2+\log k+1$ 的二进制串表示。团问题可表示为语言

$$CLIQUE=\{w\#v \mid w,v\in\{0,1\}^*,\ \text{以}w\text{为邻接矩阵的图}G\text{有一个}k\text{顶点的团，}\ v\text{是}k\text{的二进制表示}\}$$

一个接受语言 *CLIQUE* 的非确定性算法可以设计如下：

第一阶段将输入串 $w\#v$ 进行分解，并计算出 $n=\sqrt{|w|}$，以及用 v 表示的整数 k。若输入不具有形式 $w\#v$ 或 $|w|$ 不是一个平方数，则拒绝输入。这阶段可在 $O(n)$ 内完成。

第二阶段，非确定性地选择 V 的一个 k 元子集 $V'\subseteq V$，并用一个位向量 A[1..n]表示：

A[i]=1 当且仅当 $i\in V'$，因而 A 中恰有 k 个 1.

程序 8-3-3 非确定性选择算法

```
integer j,m;
j:=0;
for i to n do
   m:=choice(0,1);
   case:
     m=0: A[i]:=0;
     m=1; A[i]:=1; j:=j+1;
   end{case}
end{for}
if j≠k then failure; end{if}
```

第三阶段，确定性地检查 V' 的团性质。若 V' 是一个团则接受，否则拒绝。

这可以在 $O(k^2)$ 时间内完成。因此整个算法的时间复杂性为 $O(n+k^2)=O(n^2)=O(m)$。

如果图 $G=(V,E)$ 不包含一个 k 团，则算法的第二阶段产生的任何 k 元子集 V' 不具有团性质。因此算法没有导致接受状态的计算路径。反之，若图 G 含有一个 k 团 V'，则算法的第二阶段中有一个计算路径产生 V'，使得在算法的第三阶段导致接受状态。

注意到，算法的第二阶段是非确定性的且耗时为 $O(n)$。整个算法的计算时间复杂性主要取决于第三阶段的验证算法，即给定图 G 的一个 k 团猜测 V'，验证它是否是一个团。若验证部分可在多项式时间内完成，则整个非确定性算法具有多项式时间复杂性。这一特征具有一般性，为此我们定义验证算法：

验证算法定义为具有两个自变量的算法 A，其中一个自变量是通常的输入串 X，另一个自变量是一个称为“证书”的二进制串 Y。如果对任意串 X∈L，存在一个证书 Y 并且 A 可以用 Y 来证明 X∈L，则算法 A 就验证了语言 L. 称

$$VP=\left\{\begin{array}{l} L \mid L\in\Sigma^*,\Sigma\text{为一个有限字符集，存在一个多项式 } p\text{和} \\ \quad\text{多项式时间验证算法}A(X,Y)\text{，使得对于任意}X\in\Sigma^*, \\ \quad X\in L\text{当且仅当存在}Y\in\Sigma^*,|Y|\le p(|X|)\text{且}A(X,Y)=1 \end{array}\right\}$$

为多项式时间可验证语言类。

定理 1：$VP=NP$。

证明：先证明 $VP\subseteq NP$。对于任意 $L\in VP$，设 p 是一个多项式且 A 是一个多项式时间验证算法，则下面的非确定性算法接受语言 L：

1） 对于输入 X，非确定性地产生一个字符串 $Y\in\Sigma^*$；

2） 当 A(X,Y)=1 时接受 X.

该算法的第一步与团问题的第二阶段非确定性算法一样，至多在 $O(|X|)$ 时间内完成。第二步的计算时间是|X|和|Y|的多项式，而 $|Y|\le p(|X|)$。因此它也是|X|的多项式。整个算法可多项式时间内完成。至此 $L\in NP$。

反之，设 $L\in NP$，$L\in\Sigma^*$，且非确定性图灵机 M 在多项式时间 p 内接受语言 L。设 M 在任何情况下只有不超过 d 个的下一动作选择，则对于输入串 X，M

的任一动作序列可用$\{0,1,\cdots,d-1\}$的长度不超过$p(|X|)$的字符串来编码。不失一般性，设$|\Sigma|\geq d$。验证算法A(X,Y)用于验证“Y是M上关于输入X的一条接受计算路径的编码”。即当Y是这样一个编码时A(X,Y)=1. A(X,Y)显然可在多项式时间内确定性地进行验证，且

$$L=\{X\mid 存在Y使得|Y|\leq p(|X|)且A(X,Y)=1\}$$

因此$L\in VP$。　　证毕

定理2 若$\Pi\in NP$，则存在一个多项式p，使得Π可以用一个时间复杂性为$O(2^{p(n)})$的确定性算法来求解。

证明：设A为求解Π的一个多项式时间不确定性算法，其时间复杂性由一个多项式$q(n)$来界定。不失一般性，设q可在多项式时间内被估值。因此，对于长度为n的每个被接受的输入，必然存在字符集Γ上长度至多为$q(n)$的某个猜想字符串，使算法A的检验阶段在不多于$q(n)$步内回答“是”。若$|\Gamma|=k$，则所需要考虑的可能猜测的数目最多为$k^{q(n)}$。对于一个长度为n的给定输入，通过应用算法A的确定性检验阶段到相应的$k^{q(n)}$多个可能猜测字符串中的每一个，直到停止或运行$q(n)$步，我们可以肯定地发现A对于该输入是否有一个可接受计算。如果A在该时间界内遇到一个导致可接受计算的猜测串，则该模拟过程回答“是”；否则回答“非”。这显然形成了一个求解Π的确定性算法，而且该算法的时间复杂度将以$q(n)k^{q(n)}$为上界，其等价于$O(2^{p(n)})$。　　证毕

定理2的证明指出，在非确定性图灵机上时间复杂性为$q(n)$的判定问题与在确定性图灵机上时间复杂性为$q(n)k^{q(n)}$的问题相当，因此，直观上我们有理由认为多项式时间不确定性算法要比多项式时间确定性算法的速度快得多，能够在多项式时间内求解后者所不能够求解的许多其它问题。由此及许多其它理由，在目前已知知识背景下，人们普遍认为P是NP的真子集。关于这方面的研究基本上有两条线路：

1）　证明NP类中的某些问题是难解的，从而得到$P\neq NP$。但是这同原问题的难度几乎相当，也许只有建立一套全新的数学论证方法才有希望解决。

2）　考虑NP类中问题之间的关系，从中找到一些具有特定性质的、与P

中问题有显著不同的问题。沿此路线人们已经证明了在 NP 类中存在一个称为 NP－完全的子类，并由此发展出一套著名的 NP－完全理论。而证明一个问题是 NP－完全的通常被认为一个告诉我们应该放弃寻找、设计求解该问题的有效算法（多项式时间算法）的强有力证明。

§4 NP 完全问题

研究 P=NP 的问题有两条基本思路：

1．证明 NP 类中的某些问题是难解的，从而得到 NP≠P。但是要证明这一点几乎同证明 P=NP 一样困难。

2．考察 NP 类中问题之间的关系，从中找到一些具有特殊性质的问题。沿着这一路线人们已经证明了在 NP 类中存在被称为 NP-完全的子类，简称 NPC 问题，并由此发展了一套著名的 NP 完全理论。

本节简要介绍 NP-完全性理论。为此，首先引入各语言之间的多项式变换的概念。

定义 1 所谓从一个语言 $L_1 \subseteq \Sigma_1^*$ 到另一个语言 $L_2 \subseteq \Sigma_2^*$ 的多项式变换是指满足下面两个条件的函数 $f:\Sigma_1^* \to \Sigma_2^*$，

(1) 存在计算 f 的一个多项式时间 DTM 程序；

(2) 对于所有的 $x \in \Sigma_1^*$ 有：$x \in L_1$ 当且仅当 $f(x) \in L_2$。

用 $L_1 \propto L_2$ 表示存在一个从语言 L_1 到语言 L_2 的多项式变换。相应地，对于判定问题 Π_1, Π_2，设 e_1 和 e_2 是相应的编码策略。若 $L[\Pi_1, e_1] \propto L[\Pi_2, e_2]$，则记为 $\Pi_1 \propto \Pi_2$。

也可以从问题的角度来叙述：由判定问题 Π_1 到判定问题 Π_2 的多项式变换是满足下列条件的函数 $f: D_{\Pi_1} \to D_{\Pi_2}$，

(1) f 可由一个多项式时间的确定性算法来计算；

(2) 对于所有的 $\mathrm{I} \in D_{\Pi_1}$ 有：$\mathrm{I} \in Y_{\Pi_1}$ 当且仅当 $f(\mathrm{I}) \in Y_{\Pi_2}$。

定义 2 称一个语言 L（判定问题 Π）为 NP-完全的（NPC），如果

$$L \in NP\ (\Pi \in NP)\ ,$$

且对于所有别的语言$L' \in NP$（判定问题$\Pi' \in NP$）均有

$$L' \propto L\ (\Pi' \propto \Pi)\ 。$$

按照定义 2，要证明问题Π是 NP 完全的，需要证明所有的 NP 问题均能够经多项式变换变成Π。这几乎是很难做到的。如果 NP-完全问题比较多，我们也不能对每一个这样的问题都这样验证。为此我们讨论一些 NPC 问题的有用的性质。

性质 1 如果$L' \propto L$，则$L \in P$意味着$L' \in P$。

性质 2 如果$L' \propto L$而且$L \propto L''$，则$L' \propto L''$。

由性质 1，2，不能推出下列结论，

定理 2 设Π是 NP 完全的，如果$P \neq NP$，则$\Pi \in NP \setminus P$。

定理 3 如果$L', L \in NP, L' \propto L$，则$L' \in NPC \Rightarrow L \in NPC$.

定理 3 是证明 NP 完全问题的基础。但这需要一个 NPC 问题作为源问题。Cook 首先给出了这样一个问题--可满足性问题。可满足性问题是数理逻辑中一个重要问题，它定义在布尔变量之上。

给定布尔变量集$U = \{u_1, u_2, \cdots, u_m\}$，$U$上的一个真值分配是指一个映射$t: U \to \{T, F\}$。$U$上的一个子句 c 就是由一些布尔变量（或它们的“非”）通过逻辑“或”连接起来的布尔表达式

$$c = z_1 \vee z_2 \vee \cdots \vee z_k, z_i \in U \cup \bar{U}, i = 1, 2, \cdots, k \qquad (8.4.1)$$

其中，$\bar{U} = \{\bar{u}_1, \bar{u}_2, \cdots, \bar{u}_n\}$。若**存在**对于布尔变量集$U$的一个真值分配，使得该子句取值为真，则说该子句被满足。子句的集合说是可满足的，如果存在U的一个真值分配，使得集合中的每个子句的取值均为真。可满足性问题可陈述如下：

例 给定布尔变量之集U以及U上子句的一个集合 C。

问 是否存在U的一个真值分配，使得 C 是可满足的。

Cook 定理 可满足性问题是 NP-完全问题。

从 Cook 的开创性工作至今，人们已经发现并证明了数千个 NPC 问题（如，0/1 背包问题和 Hamilton 回路问题），总结出证明 NP-完全性的几种方法，并建立了如何分析、进而近似求解 NP-完全问题的方法。以下列出几个典型的 NPC 问题：

✧ **三维匹配问题** 3DM(3 Dimensional Matching)

例： 给定三个互不相交的、均含有 q 个元素的集合 W,X,Y，取 $M \subseteq W \times X \times Y$。

问： M 包含一个匹配吗？即是说，是否存在一个子集 $M' \subseteq M$，使得 $|M'| = q$，且 M' 中任意两个三元组都没有相同的分量。

✧ **三元精确覆盖**问题 X3C(Exact Cover by 3-sets)

例：给定有限集合 $X, |X| = 3q$，以及 X 的三元子集族 C。

问： C 含有 X 的一个精确覆盖吗？即是说，是否存在一个子族 $C' \subseteq C$，使得 X 的每个元素恰好只出现在 C' 的一个三元子集中。

注意到，如果令 $U = W \cup X \cup Y$，$C = \{\{w,x,y\} | w \in W, x \in X, y \in Y\}$，则三元匹配问题就转化为三元精确覆盖问题（若已知道三元匹配问题是 NP-完全问题，那么，三元精确覆盖问题也必是 NP-问题)。

✧ **顶点覆盖**问题 VC（Vertex Cover）

例：给定一个图 G(V,E)和一个正整数 K≤|V|.

问： 是否存在 G 的一个顶点数不超过 K 的覆盖？即是否存在一个顶点子集 $V' \subseteq V, |V'| \le K$，使得对于每一条边{u,v}∈E，u 与 v 中至少有一个属于 V'.

✧ **Hamilton 回路**问题 HC（Hamiltonian Circuit）

例：已知一个图 G(V,E)。

问：G 含有一个 Hamilton 回路吗？G 的 Hamilton 回路是指包含图 G 的所有顶点的简单回路(圈)，即是 G 的顶点的一个排序：$[v_1, v_2, \ldots, v_n]$，其中 n=|V|，使得对所有的 i: $1 \le i < n$, $\{v_i, v_{i+1}\} \in E$，$\{v_n, v_1\} \in E$.

✧ **划分问题**

例 已知一个有限集合 A 及对于每个 $a \in A$ 的一个权值 $s(a) \in Z^+$。

问 问是否存在 A 的一个子集 $A' \subseteq A$，使得 $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$ ？

✧ **三元可满足性问题 3SAT**

例 给定布尔变量的一个有限集合 U 及定义于其上的子句集 $C = \{c_1, c_2, \cdots, c_m\}$，其中 $|c_i| = 3, i = 1,2,\cdots,m$

问 是否存在 U 上的一个真赋值，使得 C 中所有的子句均被满足？

§5　证明新问题是 NPC 问题的方法

根据上节的结论，证明一个新问题Π是 NPC 问题的一般步骤是：

(a)　证明 $\Pi \in NP$；

(b)　选取一个已知的 NP 完全问题Π'；

(c)　构造一个从Π'到Π的变换f；

(d)　证明f为一个多项式变换。

这里一个关键的问题是如何选取“参照物”Π'和构造多项式变换 f。在实际证明过程中参照物的选择带有一定的经验性，已知的 NPC 问题实例越多越有利。一般情况下，很难写出多项式变换的明确的解析表达式，但是可以给出问题的转换过程，并说明这个转换过程可以通过多项式时间完成。

例 8.5.1 证明三元可满足性问题 3SAT(3-Satisfiability)是 NPC 问题

例：给定布尔变量的一个有限集合$U=\{u_1,u_2,\cdots,u_n\}$及定义于其上的逻辑语句$C=C_1\wedge C_2\wedge\cdots\wedge C_m$，其中，$|C_i|=3$，$i=1,2,\cdots,m$。

问：是否存在U的一个真赋值，使得C为真？

证明：我们选取可满足性问题 SAT 作为参照物。设$U=\{u_1,u_2,\cdots,u_n\}$及语句$C=C_1\wedge C_2\wedge\cdots\wedge C_m$构成 SAT 的任一实例。我们构造新的布尔变量集$U'$及定义其上的三元语句$C'$，使得$C$可满足当且仅当$C'$可满足。

对每个子句C_j，设 $C_j=z_1\vee z_2\vee\cdots\vee z_k$，其中$z_i\in\overline{U}\cup U$，$1\le i\le k$，$\overline{U}=\{\overline{u}_1,\overline{u}_2,\cdots,\overline{u}_n\}$

情形 1：k=1, 令

$$U_j'=\left\{y_j^1,y_j^2\right\}$$

$$C_j'=\left(z_1\vee y_j^1\vee y_j^2\right)\wedge\left(z_1\vee y_j^1\vee \overline{y}_j^2\right)\wedge\left(z_1\vee \overline{y}_j^1\vee \overline{y}_j^2\right)\wedge\left(z_1\vee \overline{y}_j^1\vee \overline{y}_j^2\right);$$

情形 2：k=2，令

$$U_j'=\{y_j^1\},\quad C_j'=(z_1\vee z_2\vee y_j^1)\wedge(z_1\vee z_2\vee \bar{y}_j^1);$$

情形 3：k=3，令

$$U_j'=\{\ \},\quad C_j'=C_j;$$

情形 4：k>3，令

$$U_j'=\{y_j^i\mid 1\le i\le k-3\}$$

$$C_j'=(z_1\vee z_2\vee y_j^1)\wedge\bigwedge_{1\le i\le k-4}(\bar{y}_j^i\vee z_{i+2}\vee y_j^{i+1})\wedge(\bar{y}_j^{k-3}\vee z_{k-1}\vee z_k)$$

最后令 $U'=U\cup\bigcup_{i=1}^{m}U_i',\quad C'=\bigwedge_{1\le i\le m}C_i'$

要证明上述转换的确是一个变换，只需证明：C'可满足当且仅当C可满足。

设$t:U\to\{T,F\}$为满足C的一个真赋值，以下证明t可扩展成满足C'的一个真赋值：$t':U'\to\{T,F\}$。因为$U'\backslash U$中的变量可分解成不同的集合$U_j',1\le j\le m$，而每个U_j'的变量仅出现在属于C_j'的子句中，我们仅需要说明如何将t扩充到各个U_j'即可，且证明在上述四种情形的每一种情形下，C_j'中的所有子句均被满足。

若U_j'属于情形 $k=1$或情形 $k=2$，则C_j'中的字句已被t所满足，从而可任意地扩展它到U_j'，如对所有的$y\in U_j'$，令$t'(y)=T$。

若U_j'是由情形$k=3$所确定的，那么U_j'为空集，而t的赋值已经使$C_j'=C_j$中的单个子句取真值。

若U_j'是由情形$k>3$所确定的，因为t为C的一种可满足性真赋值，必然存在一个最小的整数，使得文字z_l在t之下被赋予真值。若l为 1 或 2，则可对$i:1\le i\le k-3$，令$t'(y_j^i)=F$；若l为$k-1$或k，则对于$i:1\le i\le k-3$，令$t'(y_j^i)=T$；其余情况，令$t'(y_j^i)=T,1\le i\le l-2;$ 。 $t'(y_j^i)=F,l-1\le i\le k-3$

容易证明，这些选择将保证C_j'中所有的子句均被满足，进而C'中的所有子

句也均被赋值t'所满足。

反之，若t'为C'的一个可满足性真赋值，则不难证明t'对于U中变量的限制必形成对C的一个可满足性真赋值。至此，我们证明了C可满足当且仅当C'可满足。

要证明上述变换可在多项式时间内完成，只需注意到C'中三元子句的数目被$m \cdot n$的一个多项式所界定,因为，任一个子句的长度k不超过$2n$。也就是说，3SAT 例子的大小由 SAT 例子的大小的一个多项式函数所界定。由此，根据上述构造方法，不难证明它是一个多项式变换。

例 8.5.2 团问题（CLIQUE）

例：给定一个无向图$G(V,E)$和一个正整数k。

问：G是否包含一个k团？即是否存在$V' \subseteq V, |V'| = k$，且对任意$u,v \in V'$，有$(u,v) \in E$。

可以证明 CLIQUE 是 NP 问题，下面通过 3-SAT∝CLIQUE 来证明 CLIQUE 是 NP 难的，从而说明 CLIQUE 是 NP 完全的。

设$C = C_1 \wedge C_2 \wedge \cdots \wedge C_k$是一个三元合取范式，其中

$$C_i = c_1^i \vee c_2^i \vee c_3^i, \quad i = 1,2,\cdots,k$$

构造图$G(V,E)$如下：对于每个子句C_i定义三个顶点：v_1^i, v_2^i, v_3^i分别与子句中的成分c_1^i, c_2^i, c_3^i对应，以每个子句对应的三个顶点为一部分，构造一个k部图，只要$i \neq j$而且$c_s^i \neq \overline{c}_t^j$（不是互非的），则顶点$v_s^i$与顶点$v_t^j$之间就连一条边（参看图 8-5-1）。

以下证明C可满足当且仅当G有k团。如果C可满足，则存在逻辑变量的一组真值分配，使得$C = T$，因而每个子句都取真值：$C_i = T, \quad i = 1,2,\cdots,k$。在每个子句中至少有一个成分取真值，这样在 G 的每部分中取一个顶点，这个顶点对应的子句成分是取真值的，我们得到一个子集合$V' \subseteq V$。因为V'中任何两个顶点属于不同部分，而且它们所对应的成分不可能是互为非的（因为都取真值），所以这两个顶点是相连的。因而V'构成一个k团。反之，如果G有一个k团V'，则

V'中的顶点来源于k部图 G 的每一部分恰好一个。对于$v_s^i \in V'$，v_s^i等于逻辑变量u_p时，则给u_p分配真值；如果v_s^i等于逻辑变量u_p的非$\overline{u}_p$，则给u_p分配假值。这样就给部分逻辑变量分配了值，而且这种分配不会出现矛盾（根据图 G 的定义），其余未提到的变量随意取定值即可。对于这种真值分配，C一定是满足的，因为每个子句中至少有一个成分取真值。

考察例子$C=(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

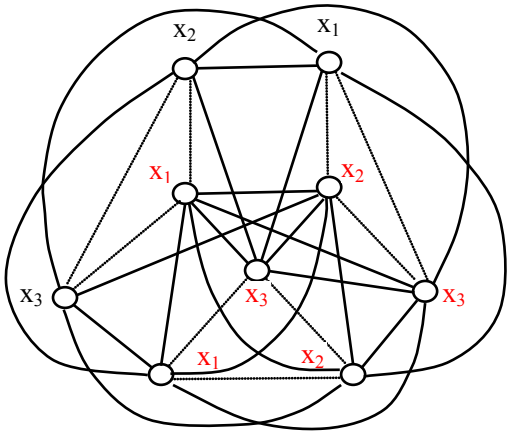


图 8.5.1 一个三元子句对应的图

例 8.5.3 顶点覆盖问题 VERTEX-COVER

例： 给定无向图$G(V,E)$和一个正整数k。

问： G是否有k覆盖，即是否存在$V' \subseteq V, |V'| = k$，使得对任意$(u,v) \in E$，有$u \in V' \; or \; v \in V'$。

顶点覆盖问题是 NP 问题。我们通过 CLIQUE∝VERTEX-COVER 来证明 VERTEX-COVER 是 NP-完全的。

对于 CLIQUE，设 n 个顶点的图 G(V,E)有k团$V' \subseteq V$，则G的补图$\overline{G}$有 n-k 覆盖$V'' = V \setminus V'$。反之亦然。

例 8.5.4 精确覆盖问题 XC

例： 已知有限集合S的子集族C

问： C是否包含一个精确覆盖，即是否存在$C' \subseteq C$，使得C'中元素互不相交，且$\bigcup_{c \in C'} c = S$。

这个问题是 NPC 问题。我们将利用它证明定和子集问题是 NPC 问题。

例 8.5.5　定和子集问题 DSS

例：给定非负整数集合 S，正整数 M

问：是否存在子集 $C \subseteq S$，使得 $\sum_{c \in C} c = M$

给定精确覆盖的一个实例：集合 $F = \{f_1, \cdots, f_n\}$，其子集族 $T = \{T_1, \cdots, T_k\}$，构造定和子集问题的一个实例：$S = \{s_1, \cdots, s_k\}$，其中 $s_i = \sum_{1 \le j \le n} \delta_{ij} (k+1)^{j-1}$，其中，当 $f_j \in T_i$ 时 $\delta_{ij} = 1$；当 $f_j \notin T_i$ 时 $\delta_{ij} = 0$。取 $M = \sum_{1 \le j \le n} (k+1)^{j-1} = \left((k+1)^n - 1\right)/k$。当 T 含有 F 的精确覆盖 T' 时，令 $S' = \{s_i \mid T_i \in T'\}$，则 $\sum_{s_i \in S'} s_i = M$。反之，由 $\sum_{s_i \in S'} s_i = M$ 令 $T' = \{T_i \mid s_i \in S'\}$ 可知 T' 是 F 的精确覆盖。

例 8.5.6 划分问题 PARTS

例：已知一个有限集合 A，及对每个 $a \in A$ 赋予的权值 $s(a) \in Z^+$

问：是否存在一个子集 $A' \subseteq A$，使得 $\sum_{s \in A'} s(a) = \sum_{s \in A \setminus A'} s(a)$

以下证明 DSS $\propto$ PARTS。给定非负整数集合 $S = \{s_1, \cdots, s_k\}$ 和正整数 M，构造集合 $A = \{a_1, \cdots, a_k, a_{k+1}, a_{k+2}\}$，其中

$$a_i = s_i, 1 \le i \le k; \quad a_{k+1} = M + 1, a_{k+2} = 1 - M + \sum_{1 \le i \le k} s_i$$

当且仅当 S 有一个和数为 M 的子集时，A 有一划分。

例 8.5.7　0/1 背包（判定）问题 0/1 KPS

例：给定一个有限集合 U，对于每个 $a \in U$，对应一个值 $w(a) \in Z^+$ 和另一个值 $v(a) \in Z^+$。另外给定一个约束值 $B \in Z^+$ 和目标 $K \in Z^+$

问：是否存在 U 的一个子集 $U' \subseteq U$，使得 $\sum_{a \in U'} w(a) \le B$，而且 $\sum_{a \in U'} v(a) \ge K$

以下证明：PARTS $\propto$ KPS。给定一个划分问题的实例：有限集合 A，及对每个 $a \in A$ 的一个权值 $s(a) \in Z^+$。构造一个 0/1 背包问题实例：$U = A, \forall a \in U, w(a) = v(a) = s(a)$，定义 $w(a) = v(a) = s(a)$。再令 $B = K = \frac{1}{2} \sum_{a \in A} s(a)$。如果 A 有一个划分 A'，则 $\sum_{a \in A'} s(a) = K = B$，因而 $\sum_{a \in A'} w(a) = B$，而且 $\sum_{a \in A'} v(a) = K$，

取$U'=A'$即是 0/1 背包问题上述实例的解。反之，若$U'\subseteq U$是 0/1 背包问题上述实例的解，则$\sum_{a\in U'}w(a)\le B$，而且$\sum_{a\in U'}v(a)\ge K=B$，因而$\sum_{a\in U'}v(a)=B=\frac{1}{2}\sum_{a\in A}v(a)$，于是取$A'=U'$即得划分问题上述实例的解。

例 8.5.8 区间排序问题（Sequencing within intervals）

例：给定任务的有限集合W，对于每个任务$w\in W$，相应有一个开始时间$r(w)$和终止时间$d(w)$以及工作时间$l(w)$，其中$r(w)\in Z^+\cup\{0\}$，$d(w),l(w)\in Z^+$。

问：是否存在任务集的一个可行时间排序表，即是否存在函数$\sigma:W\to Z^+\cup\{0\}$，满足下面两个条件：

a) 对每个$w\in W$，有$\sigma(w)\ge r(w)$, 且$\sigma(w)+l(w)\le d(w)$；

b) 若$w'\in W, w'\ne w$，则$\sigma(w')+l(w')\le\sigma(w)$或$\sigma(w')\ge\sigma(w)+l(w)$。

证明：选取划分问题作为“参照物”：有限集合A以及对每个$a\in A$的权值$s(a)\in Z^+$。现在由划分的一般性实例构造区间排序问题的一般性实例。令$B=\sum_{a\in A}s(a)$。将$a\in A$用一项任务w_a来置换，并令其满足$r(w_a)=0$, $d(w_a)=B+1$, $l(w_a)=s(a)$。再引进一个附加任务$\overline{w}$，其满足$r(\overline{w})=\lceil B/2\rceil$, $d(\overline{w})=\lceil (B+1)/2\rceil$, $l(\overline{w})=1$。这一构造过程显然可以在多项式时间内完成。以下证明：当且仅当上述划分问题的例子回答为是时所构造的区间排序问题例子的回答才为是。

附加任务$\overline{w}$对于可行时间排序表的限制表现在两个方面。首先，它确保B为奇数时不可能构造出一可行排序，因为此时由$r(\overline{w})=d(\overline{w})$，而$l(\overline{w})=1$，故$\overline{w}$不可能被排序。同时，$B$为奇数表明所对应的划分问题例子的回答必为非。故以下不妨设B为偶数，但这时附加任务$\overline{w}$所起的第二个限制就表现出来了。因为B是偶数，所以$r(\overline{w})=B/2, d(\overline{w})=r(\overline{w})+1$，并由可行排序的第一个要求知，可行排序必然使$\sigma(\overline{w})=B/2$, 这样就限制了余下的任务只能安排被任务$\overline{w}$分离的两个

时间段内，且每个时间段的长度为$B/2$，如下图所示

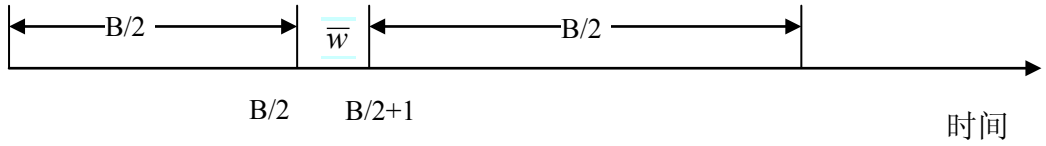


图 8.5.2 区间排序示意图

因此，排序问题就被转化为把余下任务划分为两个子集的问题，其中一个子集中所选的任务都被安排在任务$\overline{w}$之前的时间段内完成，而另一个子集中的任务均被安排在任务$\overline{w}$之后的时间段内完成。由于两个时间段的总的时间长度恰好等于除了$\overline{w}$以外的所有任务总的工作时间，故两个时间段恰好被排满。然而要做到这一点的充要条件是存在一个子集$A' \subseteq A$，使得

$$\sum_{a \in A'} s(a) = B/2 = \sum_{a \in A \setminus A'} s(a) 。$$

因此，对划分问题的回答为是当且仅当对相应的区间排序问题存在一个可行时间排序，即对它的回答也为是。至此，证明了划分问题可多项式变换到区间排序问题。

例 8.5.9 有向哈密顿圈问题 DHC

例：给定有向图$G(V,E)$

问：G 是否有一个有向 Hamilton 圈，即长度为$n=|V|$，而且恰好经过每个顶点一次，然后回到起始顶点。

DHC 是 NP 问题。我们通过 CNF-可满足性 $\propto$ DHC 证明 DHC 是 NP 完全的。设$C = C_1 \wedge C_2 \wedge \cdots \wedge C_k$，$n$个逻辑变量：$x_1, x_2, \cdots, x_{n.}$。画一个有$r$行和$2n$列的数组，第$i$行表示子句$C_i$。对每个$j$作两个相邻的列，前一列代表$x_j$，后一列代表$\overline{x}_j$，$j = 1,2,\cdots,n$。若$x_j$是$C_i$的成分，则在对应$C_i$行与$x_j$列交叉处画一个⊗；若$\overline{x}_j$是$C_i$的成分，则在对应$C_i$行与$\overline{x}_j$列交叉处画一个⊗。在$x_j$和$\overline{x}_j$两列之间引入两个顶点：$u_j$在列的顶端，$v_j$在列的底部。对于每个$j$，从$v_j$向上到$u_j$画两条（由边组成的）链，一条把$x_j$列上的⊗连起来，另一条把$\overline{x}_j$列的⊗连起来。然后再画边

(u_j, v_{j+1})， $j = 1,2,\cdots,n-1$。在每一行 C_i 的右端引一个方框 $\boxed{i}$，并画边(u_r, $\boxed{i}$)和($\boxed{i}$, v_1)，再画边($\boxed{i}$, $\boxed{i+}$)，$i = 1,2,\cdots,r$。如下图

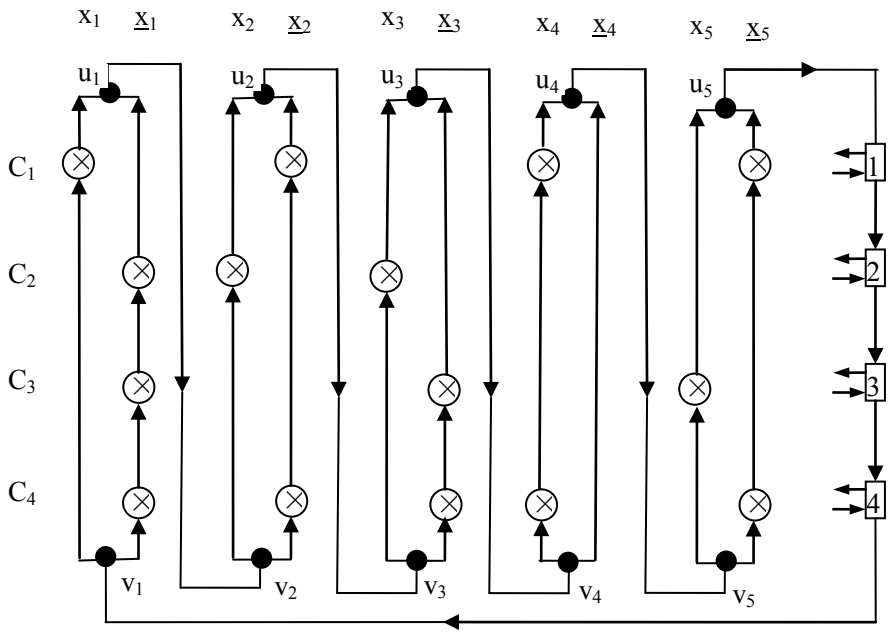


图 8.5.3 Hamilton 问题

此外，还要将图中的每一个⊗用如下左边的子图替代， $\boxed{i}$用如下右边的子图替代(见图 4.3)。这里，A_i 是入口顶点，B_i 是出口顶点，每条边($\boxed{i}$,$\boxed{i+}$)代之以（B_i， A_{i+1}），边(u_r, $\boxed{i}$)和($\boxed{i}$,v_1)分别代之以(u_r, A_1)和(B_r ,v_1). 而 $R_{is}\to\otimes\to R_{i\,s+1}$ 表示从 R_{is} 进入 ⊗ 的 w_1 顶点，$R_{i\,s+1}$ 从 ⊗ 的 w_3 顶点引出。至此，有向图 G 构造完毕(见图 8.5.3)。

若 C 满足，S 是相应的真值分配。则 G 的有向 Hamilton 圈可以从 v_1 开始，行进到 u_1，然后到 v_2，再 u_2，v_3，再 u_3，…，u_r. 在由 v_i 向上行进到 u_i 时，若 x_i 在 S 中为真则采用 x_i 对应那一列；否则沿 $\underline{x}_i$ 对应的列向上行进。这个圈将从 u_r 行进到 A_1，然后经过 R_{11}，R_{12}，…，R_{1p} ，B_1(注意，这里的 $R_{1,1}$ 实际上应该是 $R_{1,j+1}$，而 j 是第一个进到的 ⊗ ，然后诸 $R_{1,k}$ 按照逆时针循环检查)到 A_2，…，最后回到 v_1。在任一子图 的 R_{is} 行进到 $R_{i\,s+1}$ 的过程中，当且仅当某个子图 ⊗ 的顶点还不在 v_1 到 R_{is} 的路径上时，则转移到第 i 行的 ⊗ 子图。注意到，若 C_i 的长度是

i_p，则 至多转移到 i_p-1 个 ⊗ 子图，这是因为在C_i至少已经通过了一个 ⊗ 子图。从而，若C满足，则G有一个有向 Hamilton 圈。

反之，若G有一个有向 Hamilton 圈，则G的有向 Hamilton 圈上的顶点 v_1 开始，由于子图⊗和子图　的结构，这样的有向圈必须向上恰好沿每一对(x_i，$\underline{x}_i$)中一列行进。另外，圈的这部分在每一行必须至少经过一个⊗子图。因此，用于从 v_i 行进到 u_i（i=1,2, … ,n）的那一列定义了一组使C为真的真值分配。

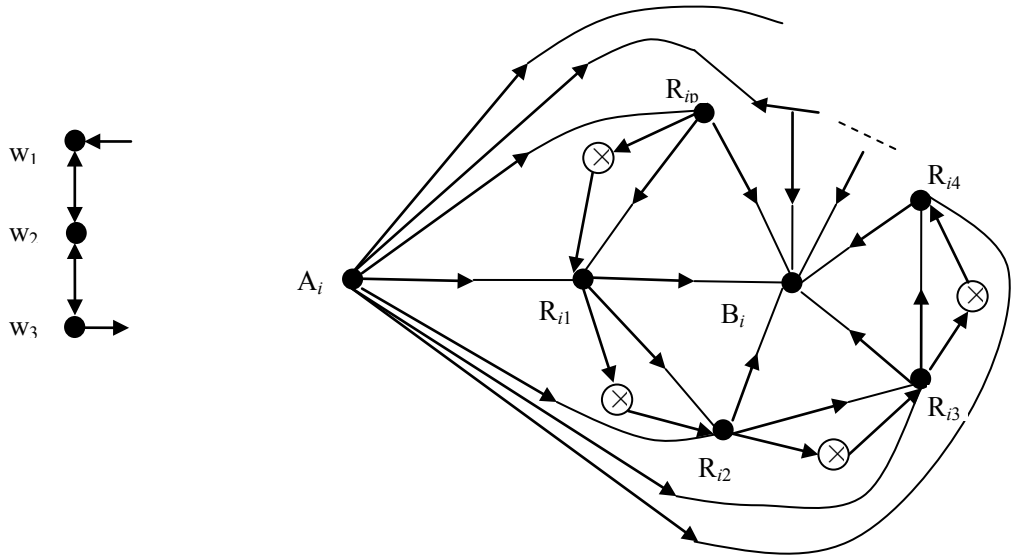


图 8.5.4 Hamilton 圈分解图

例 8.5.10　三元精确覆盖问题 X3C

例：给定有限集合 X, |X|=3q, 以及 X 的三元子集族 C。

问：C 含有 X 的一个精确覆盖吗？即是否存在一个子族$C' \subseteq C$，使得 X 的每个元素恰好只出现C'的一个三元子集中。

这个问题是 NPC 问题。我们将利用它证明 Steiner 树问题是 NPC 问题。该问题可广泛用于描述诸如有关服务设施、厂址的最优设置、某个区域内最廉价交通网、通讯线路的设计等实际问题。

例 8.5.11 Steiner 树问题

例：给定图$G=(V,E)$，对其每条边$e \in E$都有相应的权$w(e) \in Z^+$，另外有 G 的顶点子集$R \subseteq V$，某个界$B \in Z^+$。

问：是否存在 G 的一颗子树 $T=(V_1,E_1)$，使 $R\subseteq V_1\subseteq V, E_1\subseteq E$，而且 $\sum_{e\in E_1} w(e)\le B$？

以下证明 Steiner 树问题是 NPC。选 X3C 问题作为参照物：$X=\{x_1,x_2,\cdots,x_{3q}\}$，三元子集族 $C=\{c_1,c_2,\cdots,c_n\}$，构造 Steiner 树问题的相应例子如下：取 $G=(V,E)$，其中

$$V=\{v_0\}\cup C\cup X,\quad E=\{(v_0,c_i)\mid 1\le i\le n\}\cup\{(c_i,x_j)\mid x_j\in c_i, 1\le i\le n, 1\le j\le 3q\}$$

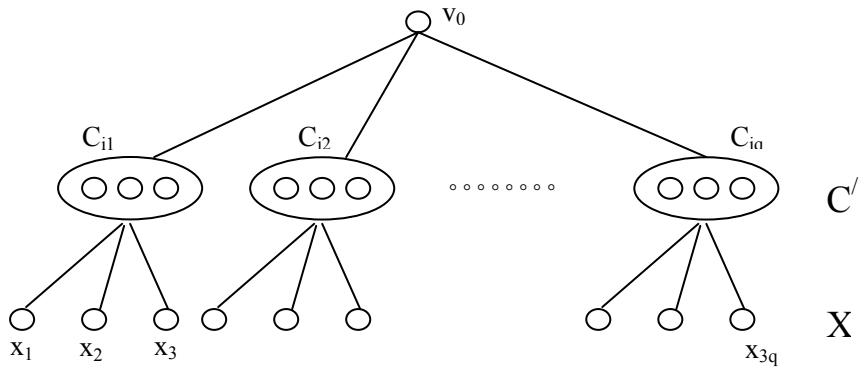


图 8.5.5 Steiner 树问题

令所有边的权值为 1，$R=\{v_0\}\cup X$，$B=4q$。显然这一构造过程可在多项式时间内完成。

如果 $C'\subseteq C$ 为 X 的一个精确覆盖，则取 $T=(V_1,E_1)$，其中

$$V_1=\{v_0\}\cup\{c_i\mid c_i\in C'\}\cup X,$$

$$E=\{(v_0,c_i)\mid c_i\in C'\}\cup\{(c_i,x_j)\mid c_i\in C', x_j\in c_i, 1\le j\le 3q\}$$

由于每个 x_j 恰好出现在 C' 的某个三元子集里，故 T 是连通的无圈图。而且 $|E_1|=4q$，因而是一棵树。注意到 $R\subseteq V_1$，且 $\sum_{e\in E_1} w(e)=|E_1|\le B$，由此知 Steiner 树问题的回答为“是”。

反之，若 $T=(V',E')$ 为 $G=(V,E)$ 的一棵 Steiner 树，$R\subseteq V'$，则每个 $x_j(1\le j\le 3q)$ 都是 T 的顶点。不妨设每个 x_j 都是 T 的叶顶点，因为，否则的话，

顶点x_j的度数大于 1，及存在边$\left(x_j,c_{i_1}\right),\left(x_j,c_{i_2}\right)\in E'\subseteq E$，删去其中一条边，如$\left(x_j,c_{i_1}\right)$。由于$\left(v_0,c_{i_1}\right)$，$\left(v_0,c_{i_2}\right)$不可能都属于$E'$，否则$T$包含有圈，将不属于$E'$的那条边添入$E'$得到另一棵树$T_1$，已知$T_1$是总权值不变的 Steiner 树。再由连通性，每个x_j恰好和某个c_i在T中邻接，令 $C'=\left\{c_i \mid \left(c_i,x_j\right)\in E',1\le j\le 3q\right\}$，则$C'$显然是$X$的一个精确覆盖。至此，证明了 X3C 问题可多项式变换到 Steiner 树问题。

§6　NP 困难问题

设Π_1和Π_2是两个判定问题，我们说Π_1在多项式时间内可图灵归约为Π_2，记做$\Pi_1 \propto_T \Pi_2$，如果存在Π_1的一个算法A_1，它多次调用求解Π_2的算法A_2作为其子程序，而且，若假设每次调用该子程序A_2均需用单位时间，则A_1为一个多项式时间算法。称A_1为从Π_1到Π_2的多项式归约。

图灵归约也有多项式变换类似的两个性质，特别地，如果判定问题Π_1可以归约到Π_2，则Π_2至少和Π_1一样难。图灵归约的定义可不限于判定问题，它可以适用于最优化问题等更广的一类问题。

定义 8.6.1 对于问题Π，如果存在一个 NP 完全问题Π'，使得$\Pi' \propto_T \Pi$，则称问题Π是 NP 困难的（NP-hard）.

由定义 8.6.1， 所有的 NP 类问题都可以多项式归约到任一个 NP 困难问题Π，这有时也作为 NP 困难问题的定义。注意，在上述定义中，并不要求$\Pi \in NP$。类似于对 NP 完全问题的讨论方法，不难推出 NP 困难问题的下述性质：

若Π是 NP 困难的， 则Π不可能在多项式时间内求解，除非 P=NP；

若Π是 NP-完全的和 NP 困难的，则其补问题$\overline{\Pi}$必是 NP 困难的。

一个典型的不属于 NP，但是 NP 困难的问题就是第 k 个最重子集问题：

例：已知整数$c_1,c_2,\cdots,c_n,k,L$；

问：存在 k 个不同的子集$S_1,S_2,\cdots,S_k \subseteq \{1,2,\cdots,n\}$，使得对于$i=1,\cdots,k$有

$\sum_{j \in S_i} c_j \geq L$ 吗？

我们已经知道划分问题是 NP-完全问题，据此可以证明第 k 最重子集问题是 NP-困难问题。事实上，设 $S=\{c_1,c_2,\cdots,c_n\}$ 是划分问题的某个给定的实例，假设已经有个算法 $A[S,k,L]$ 可以用来求解第 k 最重子集问题，则可设计出求解划分问题的算法如下：

首先，若 $\sum_{i=1}^{n} c_i$ 为奇数，则立即推出问题的回答为否；若 $\sum_{i=1}^{n} c_i$ 为偶数，当 $L=\frac{1}{2}\sum_{i=1}^{n} c_i$ 为奇数时，用算法 $A[S,k,L]$ 作为子程序，按下列二分搜索技术确定重量至少是 L 的子集的数目 K^*：

(a) 令 $K_{\min}=0, K_{\max}=2^n$；

(b) 若 $K_{\max}-K_{\min}=1$，则置 $K^*=K_{\min}$，且停机；

(c) 令 $K=(K_{\max}+K_{\min})/2$，并调用算法 $A[S,k,L]$。若回答为“是“，则令 $K_{\min}=K$，转(b)；否则，令 $K_{\max}=K$，转(b) 。

以上过程通过恰好 n 次调用算法 $A[S,k,L]$，即可找到 K^*。至此，只需再调用一次算法即可给出所考虑划分问题例子的答案，即调用 $A[S,K^*,L+1]$。若该次调用的答案为“是”，则所有 S 中重量至少为 L 的子集也必满足其重至少为 $L+1$，因此，S 没有重为 L 的子集，故对划分问题的回答为“否”。相反，若该次调用的回答为“否”，则意味着存在某一个 S 的子集，使得其重为 L，对划分问题的回答为“是”。

由上述迭代过程容易看出，若假设每次调用算法 A 只需要单位时间，则以上即给出了求解划分问题的一个多项式时间算法。也就是说，我们证明了划分问题可多项式时间归约到第 k 最重子集问题。

另一个典型的 NP 困难问题是对称（距离矩阵是对称的）旅行商问题。如果假定已经知道无向图的 Hamilton 问题是 NPC 问题，则可以证明对称旅行商问题

是 NP 困难问题如下：

证明：首先，对称旅行商问题不是 NP 的。因为，对其解的任一猜想，要检验它是否是最优的，需要同所有其它的环游比较，这样的环游会有指数多个，因而不可能在多项式时间内完成。

考虑图的 Hamilton 回路问题，已知无向图 $G=(V,E),|V|=n$，构造其对应的旅行商问题如下：

$$d_{ij}=\begin{cases}1, & \text{if } (v_i,v_j)\in E\\ 2, & \text{otherwise}\end{cases}$$

显然，这一变换可以在多项式时间内完成，而且，图 G 有 Hamilton 回路的充分必要条件是上述构建的旅行商问题有解，且其解对应的路程长度为 n。因为，若 G 不含 Hamilton 回路，则这时的旅行商问题的解对应的路程长度至少为 $n+1$。因为已知图的 Hamilton 回路问题是 NP-完全的，且上述变换为多项式变换，故我们证明了对称旅行商问题是 NP-困难问题。

事实上，如果问题 Π 的判定模式是 NPC，则其最优模式一般是 NP-困难的。

习题 八

1．叙述二元可满足性问题，并证明二元可满足性问题是 P 类问题；

2．相遇集问题

例：给定集合 S 的一个子集族 C 和一个正整数 K；

问：S 是否包含子集 S'，$|S'|\le K$，使得 S' 与 C 中的任何一个子集的交均非空？（S' 称为 C 的相遇子集）试判定相遇集问题是 P 类的还是 NP 完全的，并给出你的证明？

3．0/1 整数规划问题

例：给定一个 $m\times n$ 矩阵 A 和一个 m 元整数向量 b；

问：是否存在一个 n 元 0/1 向量 x，使得 $Ax\le b$？

试证明 0/1 整数规划问题是 NP 完全问题。

4.　如何证明精确覆盖问题和三元精确覆盖问题都是 NPC 问题？

5. 独立集问题：

例：对于给定的无向图 $G=(V,E)$ 和正整数 $k\ (\le|V|)$

问：G 是否包含一个于 k 一独立集 V'，即是否存在一个子集 $V'\subseteq V,|V'|=k$，使得 V' 中的任何两个顶点在图中 G 都不相邻。

证明独立集问题都是 NPC 问题（提示：考虑独立集和团的关系：如果V'是图G的团，则V'是G的补图$\widetilde{G}$的独立集；反之亦然）。

6. 证明无向图的 Hamilton 圈问题是 NPC 问题。

7. 说明 0/1 背包问题是 NP 困难问题。

8. NP-完全问题一定是 NP 困难问题吗？

提示：对于无向图的 Hamilton 圈问题，可将有向图的（有向）Hamilton 圈问题实例变换成无向图的 Hamilton 问题实例：把已知有向图D的每个顶点u换成欲构造的无向图G的三个顶点：u^i, u^m, u^o，并将顶点u^m与顶点u^i, u^o分别相连。如果在有向图D中有从顶点u到顶点v的有向边（弧），则在无向图G中将顶点u^o与顶点v^i连接一条边。对于独立集问题，

其它问题参考答案：

1．二元可满足性问题 2SAT

例：给定布尔变量的一个有限集合$U=\{u_1,u_2,\cdots,u_n\}$及定义其上的子句$C=\{c_1,c_2,\cdots,c_m\}$，其中$|c_k|=2, k=1,2,\cdots,m$。

问：是否存在U的一个真赋值，使得C中所有的子句均被满足？

证明： 2SAT 是 P－类问题。为叙述方便，采用数理逻辑中的“合取式”表达逻辑命题，于是

$$C=c_1\wedge c_2\wedge\cdots\wedge c_m=\prod_{k=1}^{m}c_k=\prod_{k=1}^{m}(x_k+y_k)$$

其中$c_i\cdot c_j$表示逻辑“与”，x_k+y_k表示逻辑“或”，x_k, y_k是某个u_j或$\overline{u}_i$。

考虑表达式$C=\prod_{k=1}^{m}(x_k+y_k)$，如果有某个$x_k+y_k=u_i+\overline{u}_i$，则在乘积式中可以去掉该子句：$C'=C\setminus(u_i+\overline{u}_i)$，可见$C$与$C'$的可满足性是等价的。所以我们可以假定$C$中不含有形如$u_i+\overline{u}_i$的子句。注意到此时$C$中的子句个数不会超过$n(n-1)$。

如果逻辑变量u_n或它的非$\overline{u}_n$在C的某个子句中出现，我们将C表示成

$$C=G\cdot(u_n+y_1)\cdots(u_n+y_k)(\overline{u}_n+z_1)\cdots(\overline{u}_n+z_h) \qquad (1)$$

其中G是C的一部分子句，而且不出现逻辑变量u_i或它的非$\overline{u}_i$。令

$$C' = G \cdot \prod_{1 \le i \le k, 1 \le j \le h} (y_i + z_j) \qquad (2)$$

(2)式中不再含有变量u_n和它的非$\overline{u}_n$。记$U' = \{u_1, u_2, \cdots, u_{n-1}\}$。如果存在$U$的真赋值，使得$C$满足，在$U'$也一定满足。因为如果$u_n$取真值，则所有的$z_j$必然取真值；$u_n$取价值，所有的$y_i$必然都取真值，不管那中情况，$C'$的乘积部分必然取真值。反之，假设存在$U'$的真赋值，使得$C'$满足。若有某个$y_i$取假值，则所有的$z_j$必然取真值，此时令$u_n$取真值，得到$U$的真赋值，使得$C$满足。若有某个$z_j$取假值，则令$u_n$取假值，得到$U$的真赋值，使得$C$满足。如果所有的$y_i$和$z_j$都取真值，$u_n$取假值得到$U$的真赋值，使得$C$满足。至此我们得到：$C$与$C'$的可满足性是等价的。但是后者涉及的变量数比前者少 1，子句数为$m-(k+h)+kh$。但是，我们可以象前面一样简化掉所有形如$u_i+\overline{u}_i$的子句，因而可以假定C'中子句个数不超过$(n-1)(n-2)$。

上述过程可以一直进行到判定只含有一个逻辑变量的逻辑语句的可满足性问题。这需要一个常数时间即可。注意到我们每一步简化都可以在多项式（关于n的）步骤内完成，总共需要至多$n-1$步简化，因而，在多项式时间内可以完成 2SAT 二满足性问题的判定。即 2SAT 是 P－类问题。　　证毕

2. 相遇集问题是 NP－完全问题。证明采用限制法：（首先说明相遇集问题是 NP－类问题，略）若对所有的$c \in C$，置$|c| = 2$，则该问题变成顶点覆盖问题 VC，故由 VC 的 NP 完全性知相遇集问题也是 NP－完全的。

3．证明也是采用限制法，比照 0/1 背包问题的判定模式是 NPC－问题。（注意说明 0/1 整数规划问题是 NP－类问题）。

附录 1. 关于 C++程序设计

在程序开发过程中通常需要做到如下两点：一是高效地描述数据；二是设计一个好的算法，该算法最终可用程序来实现。要想高效地描述数据，必须具备数据结构领域的专门知识;要想设计一个好的算法,需要算法设计领域的专门知识。这里我们采用 C++语言来描述算法，并假定大家已经掌握了 C++程序设计的基本方法。同时还假设大家已经具备数据结构的必备知识，剩下的主要任务是探讨算法设计领域的专门知识。为了便于今后的学习，我们还是简要地回顾一下 C++程序设计方面的有关知识和数据结构方面的知识。本章主要复习 C++程序设计方面的知识，内容主要有：

模板函数、参数传递方式、类与模板类、类的共享成员－保护成员－私有成员、友元、操作符重载、数组的动态分配与释放等。

§1 模板函数

1.1 函数与模板函数

考察下列函数：

程序 1-1 计算一个整数表达式

```
int Abc(int a, int b, int c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

程序 1-2 计算一个浮点数表达式

```
float Abc(float a, float b, float c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
```

```
}
```

这两个函数中除了形式参数与返回值的类型不同外，再没有别的差别。如果我们能够编写一段通用代码，将参数的数据类型作为一个变量，它的值由编译器来确定，就可以简化程序的编写工作，这一特性在C++语言的模板函数中体现。

程序 1-3 利用模板函数计算一个表达式

```
template <class T>
T Abc(T a, T b, T c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

利用这段通用代码，通过把 T 替换成 int，编译器可以立即构作出程序 1-1；把 T 替换成 float 又可以立即构作初程序 1-2。同样可以把 T 替换成 double 和 long，等等。把函数 Abc 编写成模板函数可以使我们免去了解形式参数的类型。

1.2 传值参数和引用参数

在上面的函数 Abc 中，a，b，c 是形式参数。如果调用函数 Abc 时采用

z=Abc(x,y,2)

x，y，2 分别是对应 a，b，c 的实际参数。a，b，c 都是传值参数。运行时，与传值形式参数对应的实际参数的值将在函数执行之前被复制给形式参数，复制过程是由该形式参数所属数据类型的复制构造函数（copy constructor）完成的。如果实际参数与形式参数的数据类型不同，必须进行类型转换，从实际参数的类型转换为形式参数的类型（这里假定该种类型转换是允许的）。当函数运行结束时，形式参数所属数据类型的析构函数（destructor）负责释放该形式参数。当一个函数返回时，形式参数不会被复制到对应的实际参数中。因此，函数调用不会修改实际参数的值。

值得注意的是，采用传值参数会增加程序的开销。假定 a，b，c 是传值参数，在函数 Abc 被调用时，类型 T 的复制构造函数把相应的实际参数分别复制到形式参数 a,b,c 之中，以供函数 Abc 使用。如果用具有 100 个元素的矩阵 Matrix 作为实际参数来调用函数 Abc，那么复制三个实际参数给 a，b，c，将需要 300 次操作。同理，当函数 Abc 返回时，其析构函数又需要花费额外的 300 次操作来释放 a，b，c。

C++语言通过使用引用参数来减少这笔开销。

程序 1-4 利用引用参数计算一个表达式

```
template<class T>
T Abc(T& a, T& b, T& c)
{
  return a+b+c+b*c+(a+b+c)/(a+b)+4;
}
```

在程序 1-4 中，a，b，c，是引用参数（reference parameter）。如果用语句 Abc（x，y，z）来调用函数 Abc，其中 x，y，z 是相同的数据类型，那么这些实际参数被分别赋予名称 a，b，c，因此，在函数 Abc 执行期间，x，y，z 被用来替换对应的 a，b，c。与传值情况不同，在函数调用时，本程序并没有复制实际参数的值，在返回也没有调用析构函数。如果 x，y，z 分别是 100 个元素的矩阵，由于不需要把 x，y，z 的值复制给对应的形式参数，因此，我们至少可以节省采用传值参数进行复制时所需要的 300 次操作。

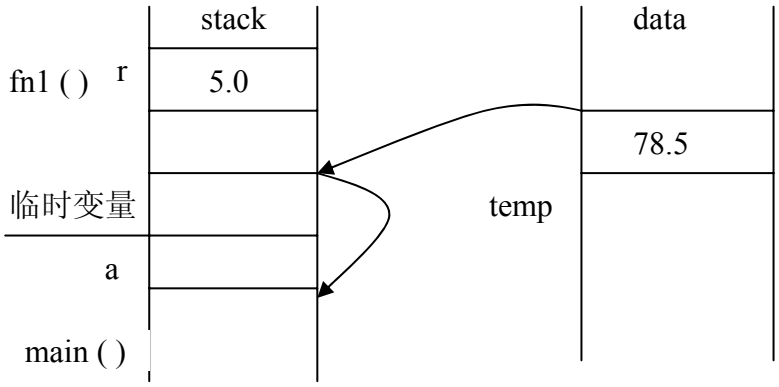
事实上，引用只是个别名，当建立引用时，程序用另一个变量或对象（目标）的名字初始化它。引用不是值，不占用存储空间，声明引用时，目标的存储状态不会改变。引用一旦初始化，它就维系在一定的目标上，再也不分手。

函数的返回值也可以采取引用的方式。

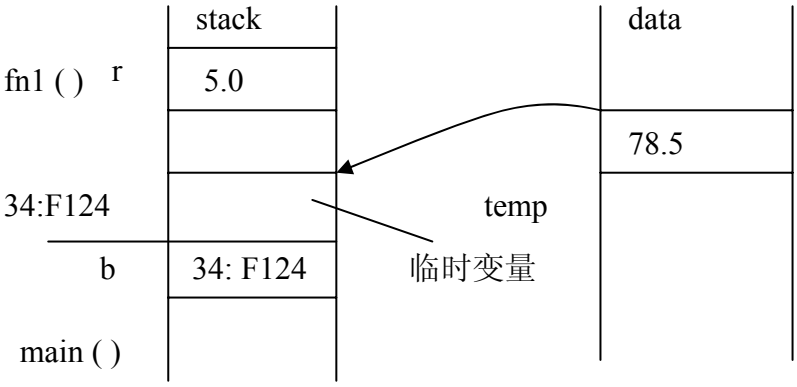
程序 1-6 关于函数的返回值或返回引用的程序

```
# include <iostream.h>
float temp;
float fn1(float r)
{
  temp=r*r*3.14;
  return temp;
}
float& fn2(float r)
{
  temp=r*r*3.14;
  return temp;
}
void main()
  float a=fn1(5.0);  //形式 1
  float& b=fn1(5.0);  //形式 2
  float c=fn2(5.0);  //形式 3
  float& d=fn2(5.0);  //形式 4
   cout <<a <<endl;
   cout <<b <<endl;
   cout <<c <<endl;
   cout <<d <<endl;
}
```

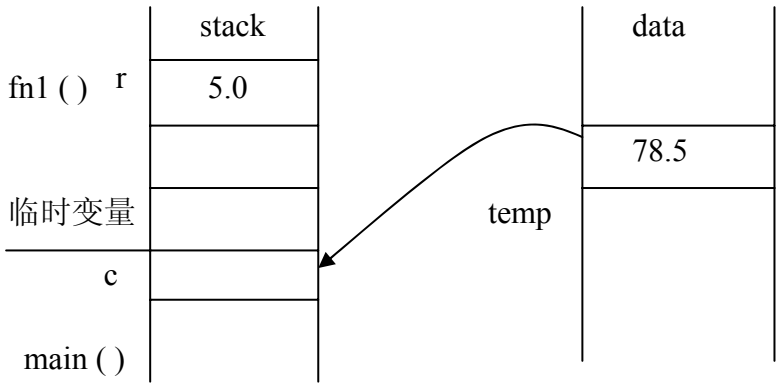
对于主函数的四种返回方式，运行结果虽是一样的，但它们在内存中的活动情况各不相同。其中，变量 temp 是全局数据，驻留在全局数据区 data，函数 main（）、函数 fn1（）及函数 fn2（）驻留在栈区 stack。以下四种返回形式的表示：



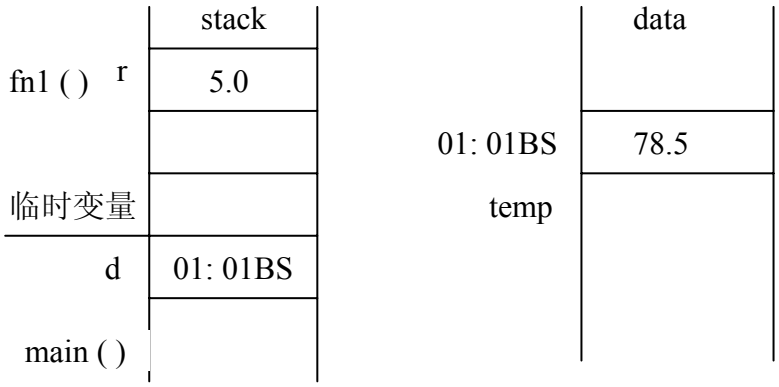
1． 传值参数和返回值方式内存布局



2． 返回值初始引用的内存布局



3． 返回引用方式的内存布局



4. 返回引用方式的值作为引用的初始化

采用引用参数或返回引用应充分考虑到被引用变量的生命期，免得出现数据失效现象。

§1.2 动态存储分配与异常引发

C++操作符 new 可用来进行动态存储分配，该操作符返回一个指向所分配空间的指针。例如在使用数组时，很多情况下数组的大小在编译时可能是未知的。事实上，它们可能是随着函数调用的变化而变化。对于一个大小为 n 的一维浮点数组可以按如下方式来创建：

```
float *x=new float [n];
```

构建二维数组时，可以把它看成是由若干行组合起来的，每一行都是一个一维数组，动态存储时需多次调用 new。

程序 1-7 为二维数组动态分配空间

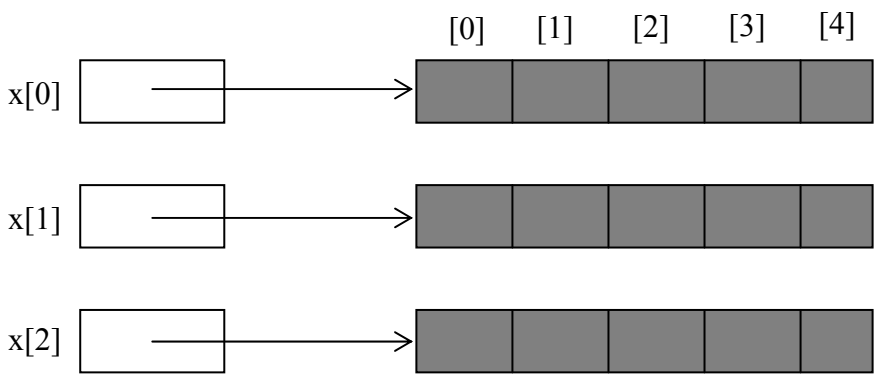
```
template<class T>
bool Make2Darray(T** &x, int rows, int cols)
{//创建一个二维数组
```

```
        try{//创建指针
          x=new T* [rows];
          //为每一行分配空间
          for (int i=0; i<rows; i++)
          x[i]=new int [cols];
         return true;
       }
      catch(xalloc){return false;}
 }
```

这里 rows 是数组的行数，cols 是数组的列数。语句 catch（xalloc）{return false;}是引发异常，当空间分配失败时，返回 false。

程序 1-8 释放由 Make2Darray 所分配的空间

```
template<class T>
void Delete2Darray(T** &x, int rows)
{//删除二维数组 x
 //释放为每一行所分配的空间
  for(int i=0; i<rows; i++)
    delete [] x[i];
  //删除行指针
  delete [] x;
  x=0;
}
```



一个 3×5 数组的存储结构

§1.3　C++的类与类模板

类构成了实现 C++面向对象程序设计的基础。类是 C++封装的基本单元，它把数据和函数封装在一起。当类的成员声明为保护时，外部不能访问；声明为公共时，则在任何地方都可以访问。将类定义和其成员函数定义分开，是目前开发程序的通常做法。我们可以把类定义（头文件）看成是类的外部接口，类的成员函数定义看成是类的内部实现。定义了一个类以后，可以把该类名作为一种数据类型，定义其“变量”（对象）。

类中有两个特殊的成员函数：构造函数与析构函数。构造函数创建类对象，初始化其成员，析构函数撤销类对象。这两个函数成员的设计和应用，直接影响编译程序处理对象的方式，它们的实现使 C++的类机制得以充分的体现。

例子　类 Currency 的建立

类 Currency 的对象主要拥有三个成员：符号（sign）、美元（dollars）、美分（cents）。如$2.35, sign＝＋，dollars＝2，cents＝35；-$6.05, sign＝－，dollars＝6，cents＝5。对于这种类型的对象，主要执行的操作是：

1）、设置成员的值；

2）、确定各成员的值；

3）、增加两种货币的类型；

4）、增加成员的值；

5)、输出。

首先确定用无符号的长整型变量 dollars、无符号整型变量 cents 和 sign 类型的变量 sgn 来描述货币对象，其中 sign 类型的定义为：

```
enum sign{plus, minus};
```

程序 1-9 定义 Currency 类

```
 class Currency{
  public
  // 构造函数
  Currency(sign s=plus, unsigned long d=0, unsigned int c=0);
  // 析构函数
  ～Currency() {}
  bool Set(sign s, unsigned long d=0, unsigned int c=0);
  bool Set(float a);
  sign Sign() const {return sgn;}
  unsigned long Dollars() const {return dollars;}
  unsigned int Cents() const {return cents;}
  Currency Add(const Currency& x) const;
  Currency& Increment(const Currency& x);
  Void Output() const;
 private:
  sign sgn;
  unsigned long dollars;
  unsigned int cents;
}
```

说明：public 部分的第一个函数与 Currency 类同名，即是构造函数，其指明如何创建一个类型的对象，它不可有返回值。此处，构造函数有三个参数。在创建一个类对象时，构造函数被自动唤醒。如

```
Currency f, g(plus,3.45), h(minus,10);
Currency *m=new Currency(plus,8,12);
```

分别创建了“变量”对象和一个指向 Currency 对象的指针（变量）对象。但是，在程序 1-9 中，类的成员函数只有声明，未有描述(或实现)，因而还必须在类定义的外部给出各个成员函数的描述。

程序 1-9.1 Currency 的构造函数

```
Currency::Currency(sign s, unsigned long d, unsigned int c)
{// 创建一个 Currency 对象
  if(c>99)
    {//美分数目过多
     cerr<<”Cents should be < 100” << endl;
     exit(1);}
  sgn=s; dollars=d; cents=c;
}
```

构造函数在初始化当前对象的 sgn、dollars、cents 等数据成员之前需要验证参数的合法性。

程序 1-9.2 设置 private 数据成员

```
bool Currency::Set(sign s, unsigned long d, unsigned int c)
{// 取值
  if(c>99) return false;
  sgn=s; dollars=d; cents=c;
```

```
  return true;
}
bool Currency::Set(float a)
{// 取值
  if(a<0) {sgn=minus; a=-a;}
  else sgn=plus;
  dollars=a; //抽取整数部分
  // 获取两个小数位
  cents=(a+0.005-dollars)*100;
  return true;
}
```

程序 1-9.3 累加两个 Currency

```
Currency Currency::Add(const Currency& x) const
{// 把 x 累加到 *this
  long a1, a2, a3;
  Currency ans;
  // 把当前对象转换成带符号的整数
  a1=dollars * 100 + cents;
  if(sgn==minus) a1=-a1;
  // 把 x 转换成带符号的整数
  a2=x.dollars * 100 + x.cents;
  if(x.sgn==minus) a2=-a2;
  a3=a1 + a2;
```

```
    // 转换成 currency 形式
    if(a3<0) {ans.sgn=mins; a3=-a3;}
   else ans.sgn=plus;
   ans.dollars=a3/100;
   ans.cents=a3-ans.dollars * 100;
   return ans;
}
```

说明：在 C++中，保留关键字 this 用于指向当前对象，*this 代表对象本身。

程序 1-9.4　Increment 与 Output

```
Currency& Currency::Increment(const Currency& x)
{// 增加量 x
   *this=Add(x);
   return *this;
}
void Currency::Output() const
{// 输出 currency 的值
  if(sgn==minus) cout <<' - ';
  cout <<' $' <<dollars <<'.' ;
  if(cents <10) cout << "0" ;
  cout << cents;
}
```

说明：利用成员函数来设置数据成员的值可以确保成员拥有合法的值，如，

构造函数和 Set 函数已经做到这一点。其他成员当然也应该保证数据成员的合法性。因此，在诸如 Add 和 Output 函数的代码中就不必再验证了。

程序 1-10 Currency 类应用示例

```
#include <iostream.h>
#include "curr1.h"
void main(void)
{
    Currency g, h(plus, 3, 50), i, j;
    g.Set(minus, 2, 25);
  i.Set(-6.45);
   j=h.Ad(g);
   j.Output(); cout << endl;
   i.Increment(h);
   i.Output(); cout << endl;
   j=i.Add(g).Add(h);
   j.Output(); cout << endl;
   i.Output(); cout << endl;
}
```

说明：程序 1-10 的第一行定义了四个Currency类变量：g，h，i和j。除h具有初值$3.50 外，构造函数把它们都初始化位$0.00.接下来的两行将g和I分别设置成-$2.25 和-$6.45,之后调用函数Add把g和h加在一起，并把返回的对象（值为$1.25）付给j。为此，需要使用缺省的赋值过程把右侧对象的各数据分别复制到相应的数据成员，复制的结果是使j具有值$1.25,此值在下一行被输出。再下一句使i增加一个量h，然后输出i的新值（-$2.95）。语句j=i.Add(g).Add(h);表示首先将i与g相加，返回一个临时变量（值为-$5.20）,然后把这个临时变量与h相加，并返回另一个临时变量（值为-$1.70）,最后，将新的临时变量复制到

j中。这里，’.’序列的处理顺序是从左至右的。

§1.4 重载与友元

严格来讲，一个函数的返回值、参数及“函数规则”三个因素就确定了该函数。有一个因素不同，就认为是不同的函数，尽管它们有相同的名字。函数的重载正是利用了“参数的变化或还回值的变化”而得到的（属于不同类的函数认为是不同的）。为了使程序简明，C++程序中采用“重载”是常见的。

在 Currency 类中包含了几个与 C++标准操作相类似的成员函数，如，Add 进行 ＋ 操作，Increment 进行 ＋＝ 操作等。直接使用这些标准的操作符比另外定义新的函数（如 Add 和 Increment）要自然得多。可以借助操作符重载的过程来使用＋和＋＝。操作符重载允许扩充 C++操作符的功能，以便包它们直接应用到新的数据类型或类中。

程序 1-11.1 使用操作符重载的 Currency 类定义

```
class Currency {
  public:
  //构造函数
  Currency(sign s=plus, unsigned long d=0, unsigned int c=0);
  //析构函数
  ~Currency() {}
  bool Set(sign s, unsigned long d, unsigned int c);
  bool Set(float a);
  sign Sign() const
    {if (amount < 0) return minus;
     else return plus;}
  unsigned long Dollars() const
   {if (amount < 0) return (-amount)/100;
    else return amount/100;}
```

```
    unsigned int Cents() const
      {if (amount < 0)
        return  - amount - Dollars()*100;
       else return amount  -  Dollars()*100;}
    Currency operator+(const Currency& x) const;
    Currency& operator+=(const Currency& x)
      {amount+=x.amount; return *this;}
    void Output(ostream& out) const;
  private:
    long amount;
};
```

程序 1-11.2 十，output 和 << 代码

```
 Currency Currency:: operator+(const Currency& x) const
{//把 x 累加到*this
  Currency y;
  y.amount=amount+x.amount;
}
void Currency::Output(ostream& out) const
{//将 currency 的值插入到输出流
  long a=amount;
  if (a<0) {out << '-'; a=-a;}
  long d=a/100; //美元
  out << '$' << d << '.';
```

```
  int c=a-d*100; //美分
  if (c<10) out << "0";
  out << c;
}
//重载 <<
ostream& operator << (ostream& out, const Currency& x)
  {x.Output (out); return out;}
```

注：由于在前面的类定义中没有关于<< 的函数声明，所以，在程序 1-11.2 的关于操作符 << 的重载时，借用了类成员函数 Output。这是因为，函数

```
ostream& operator <<(ostream& out, const Currency& x)
```

不能访问 private 成员 amount。

事实上，一个类的 private 成员仅对于类的成员函数时可见，而在有些应用中，必须把对这些 private 成员的访问权授予其他的类和函数，一般的做法是把这些类和函数定义为友元。上面关于 << 的重载可以如下完成：

1. 在类 Currency 的定义中添加语句

```
friend ostream& operator << (ostream&, const Currency&);
```

2. 给出重载友元 << 程序

程序 1-11.3 重载友元 <<

```
  // 重载 <<
  ostream& operator << (ostream& out, const Currency& x)
  {//把 currency 的值插入到输出流
    long a=x.amount;
    if (a<0) {out << '-'; a=-a;}
    long d=a/100; //美元
    out << '$' << d << '.';
```

```
    int c=a-d*100; //美分
    if (c<10) out << “0”;
    out << c;
    return out;
}
```

利用操作符重载，程序 1-10 可以改写如下，看起来更自然、可读。

程序 1-11.3 操作符重载的应用

```
# include <iostream.h>
# include “curr3.h”
void main(void)
{
  currency g, h(plus, 3, 50), i, j;
  g.Set(minus, 2, 25);
   i.Set(-6.45);
   j=h+g;
   cout << j << endl;
   i+=h;
   cout << i <<endl;
   j=i+g+h;
   cout << j << endl;
   j=(i+=g)+h;
   cout << j << endl;
   cout << i <<endl;
}
```

§1.5 检测与调试

检查程序的正确性，主要是数学证明方法和程序测试过程。很多情况下，用数学证明的方法，在后面要提到。但是，多数情况下用数学证明方法证明一个程序的正确性是十分困难的，这里谈一下程序测试过程。所谓程序测试过程是指在目标计算机上利用输入数据，也称为测试数据，来实际运行该程序。把程序的实际行为与所期望的行为进行比较。如果两种行为不同，就可判定程序中有问题存在。值得注意的是只使用个别的数据进行测试而未发现问题，并不能说明程序本身是正确的。所以，用测试过程来说明程序的正确性必须有足够的测试数据。对于大多数实际的程序，可能的测试数据数量很大，不可能进行穷尽测试。为此，我们需要选取具有代表性的部分数据进行测试。实际用来测试的输入数据空间的子集称为测试集。

例 求解二次方程

$$ax^2 + bx + c = 0$$

其中a,b,c都是已知数，$a \neq 0$.我们用如下程序来求解：

程序 1-12　计算并输出一个二次方程的根

```
template < class T >
void OutputRoots( T a, T b, T c )
{//计算并输出一个二次方程的根
  T d = b*b-4*a*c;
  if ( d > 0 ) {// 两个实根
    float sqrtd = sqrt (d );
    cout << “There are two real roots”
    << (-b+sqrtd)/(2*a) << “and”
    << (-b+sqrtd)/(2*a)
    <<endl; }
```

```
    else if (d==0)
        // 两个根相同
        cout << “There is only one distinct root”
           << -b/(2*a)
           << endl;
    else // 复数根
       cout << “The roots are complex”
           << endl
           << “The real part is”
           << -b/(2*a) << endl
           << sqrt(-d)/(2*a) << endl;
}
```

如果采用测试过程来验证该程序的正确性，则所有可能的输入数据就是所有不同的三元组 (a,b,c)，其中 $a \neq 0$。即使 a,b,c 都被限制为整数，而且长度为 16 位，所有不同的三元组的数目将达到 $2^{32}(2^{16}-1)$。如果目标计算机能按每秒 100 万个三元组进行测试，也至少需要三天才能完成。所以，实际测试集仅是测试数据空间的子集。显然，使用测试子集所完成的测试不能保证程序的正确性。测试的目的不是去建立正确性认证，而是为了暴露程序中的错误。因此，必须选取能暴露程序中所存在的错误的测试数据。设计测试数据的技术分为两种：黑盒法和白盒法。

比较流行的黑盒法是 I/0 分类及因果图。在 I/0 分类方法中，输入数据和/或输出数据空间被分成若干类，不同类中的数据会使程序表现出的行为有本质的不同，而相同类中的数据则使程序表现出本质上类似的行为。如二次方程求解的例子，输入和/或输出数据有三种本质不同的行为：产生两个不同的实根、产生唯一的实根、产生复数根。据此，将测试数据空间分为三个类，一个测试集应至少从每个类中抽取一个输入数据。$\{(1,-5,6),(1,-8,16),(1,2,5)\}$ 是符合黑盒法的测试集。

白盒法基于对代码的考察来设计测试数据。对一个测试集最起码的要求是

使程序中的每条语句都至少执行一次，此称为语句覆盖。另有分支覆盖，其要求测试集能使程序中的每一个条件都能分别出现 true 和 false 两种情况。可以加强分支覆盖的要求，要求每个条件中的每个从句（不包含布尔操作符&&,||,!的布尔表达式）既能出现 true 又能出现 false 情况，此称为从句覆盖。如果取定一个测试数据，程序在执行时各个语句被执行可以按先后排出一个顺序，形成一个执行路径。不同的测试数据可能会得到不同的执行路径。在白盒法中，一般要求实现执行路径覆盖。但是，在实践中，一般不可能进行全部执行路径覆盖，因为所有可能的执行路径可能具有很大的数目。在这里，我们要求使用白盒法时，应至少达到语句覆盖。另外，在测试时还应特别注意容易使程序出错的特定情形（一般是边界情形）。

程序 1-13 寻找最大元素

```
template < class T >
int Max(T a[], int n)
{//寻找 a[0:n-1]中的最大元素
  int pos=0;
  for (i=1; i < n; i ++)
  if (a[pos] < a[i])
    pos = I;
  return pos;
}
```

对于程序 1-13，测试数据 a[0:4]=[2,4,6,8,9]能够提供语句覆盖，但不能提供分支覆盖，因为条件 a[pos]<a[i] 不会变成 false。测试数据[4,2,6,8,9]既能提供语句覆盖也能提供分支覆盖。在程序 1-12 中只存在三条执行路径：1～7 行；1、2、8～12 行；1、2、8、13～19 行。在程序 1-13 中，执行路径的条数随着 n 的增加而增加，以至能够达到任意多。

测试能够发现程序中的错误，确定并纠正程序错误的过程称为调试（debug）。几点建议：

1. 可用采用逻辑的方法来确定错误的语句。

2. 如果方法 1 失败，还可以采用程序跟踪的方法，以确定程序在何时出现错误。

3. 如果普遍跟踪不易做到（对于给定的测试数据，程序需要运行的语句或指令太多），则采用专门跟踪的办法，这要求首先将可以的代码分离出来。

4. 在测试和调试一个有错的程序时，从一个与其他函数独立的函数开始。这个函数最好是一个典型的输入或输出函数。然后每次引入一个尚未测试的函数，测试并调试更大一点的程序。此称为增量测试与调试。在使用这种策略时，有理由认为产生错误的语句位于刚刚引入的函数之中。

5. 不要试图通过产生异常来纠正错误。在纠正一个错误时，必须保证不会产生一个新的、以前没有的错误。用原本能使程序正确运行的测试数据来运行纠正过错误的程序，确信对于该数据，程序仍然正确。

附录 2

基本数据结构

改进算法的基本技术之一就是数据结构化，使得由此产生的操作得以高效的执行。本章介绍几种基本的数据结构，包括栈和队列、二叉树以及图的概念。

2.1 栈和队列

在计算机程序中最长用的数据组织形式是有序表和线性表，通常记做 $a=(a_1,a_2,\cdots,a_n)$，这里 a_i 称为元素，来源于某个集合。空表有 $n=0$ 个元素。栈就是一个有序表，其插入和删除操作都在称作栈顶的一端进行。队列同样也是一个有序表，其插入操作在队尾一端进行，而删除在头一端进行。栈也被称作后入先出表（LIFO），表示对其中元素的操作顺序，而队列也被称作先入先出（FIFO）表，表示先进入队列的元素先出队。表示栈最简单的方式是用一维数组，如 $\text{stack}[MaxSize]$，其中 $MaxSize$ 是该栈容纳元素的最大个数。栈中最底元素存储在 $\text{stack}[0]$，第 i 个元素存储在 $\text{stack}[i-1], i=1,2,\cdots$。用一个与数组相关的变量指向栈中最顶层的元素，该变量记做 top。要检验栈是否为空，考察 $\text{if (top} < 0)$ 是否成立，如果不成立，则最顶层元素就在 stack[top] 中；要检验栈是否已满，可以考察 $\text{if (top} >= (MaxSize-1))$ 是否成立。对栈的两个重要操作是插入（Add）和删除（Delete）元素。用 C++语言可以描述如下

程序 2-1-1 栈的类定义

```
include < iotream.h >
template < class Type >
class Stack
{
   private :
         int top, MaxSize ;
         Type    * stack ;
     public:
         Stack (int MSize ) : MaxSize ( MSize )
         { Stack = new Type [ MaxSize ] ; top = -1 ; }
         ~Stack ( )
         {    delete [ ]    stack ; }
```

```
            incline bool    Add ( const Type item )
        // Push an element onto the stack ; return true
          //    if successful    else return false.
       { if ( StackFull ( ) ) {
          cout << "Stack is full" ; return false ;
          }
          else {
               stack [++top] = item ; return true ;
          }
     }
     incline bool Delete ( Type & item )
        // Pop the top element from the stack ; return true
        // if successful    else return false .
     {    if ( StakEmpty ( ) {
            cout << "Stack is empty" ; return false ;
          }
          else {
             item = stack [top--] ; return true ;
          }
     }
     incline bool StackEmpty ( )
        // Is the stack empty ?
     {    if ( top < 0 ) return true ;
        else return false ; }
     incline bool StackFull ( )
        // Is the Stack full ?
       { if ( top >= (MaxSize – 1 )) return true ;
        else return false ; }
  }
```

每个 Add 和 Delete 都花费固定的时间，并且与栈中元素个数无关。StackFull ( ) 和 StackEmpty ( ) 都是时间复杂度为 O(1)的函数。

另一种表示栈的方法是使用链接（或指针），节点是数据和指针信息的集合。栈可由两个域组成的节点来表示，这两个域分别称为 data 域和 link 域。每个节点的数据域包含了栈中的一项，而相应的指针域指向包含栈中下一项的节点。例如，在下图中，含有项 A,B,C,D,E

的以上述顺序插入。

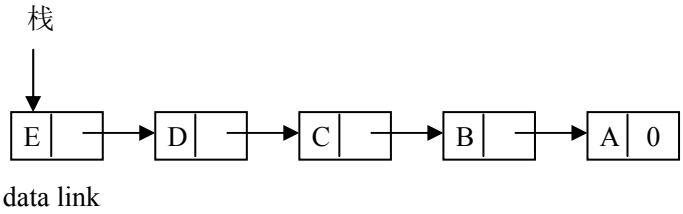


图 2-1-1 含有 5 个元素

程序 2-1-2　　链栈的表示

```
#define NULL    0
# include < iostream.h >
class Stack
{
    private :
        struct node {
            Type data ; struct node    * link ;
        }
    public :
        Stack ( )
        {    top = NULL ; }
        ~Stack ( )
        {    struct node    * temp ;
            while ( top ) {
                temp = top ;    top = top -> link ;
                delete temp ;
            }
        }
      bool Add ( const Type item ) ;
      bool Delete ( Type & item ) ;
      inline bool StackEmpty ( )
          // Is the stack empty ?
     {    if ( top ) return false ;
          else return true ;
     }
};
```

top 变量指向表中的最顶层节点（最后插入的那一项）。通过设置 top = 0 表示空栈。程序 2-1-2 是一个类的定义，对应栈的表示。由于采用了指针，所以很容易完成插入和删除操作。例如，要向栈中插入一项应当描述如下

```
bool Stack :: Add (const Type item )
{   struct node * temp = new node ;
      if ( temp ) {    temp -> data = item ; temp -> link = top ;
                       top = temp ; return true ; }
      else { cout << "Out of space!" << endl ;
           return false ;    }
}
```

语句 * temp = new node ; 将一个可用节点的地址赋值给变量 temp 。如果没有其他节点，返回 0。如果存在一个节点，就将适当的值存入该节点的两个域中。然后更新变量 top ，使其指向链表中新的栈顶元素。最后返回 true 。如果空间不足，将打印出错信息并返回 false 。

注意到程序 2-1-2 的类定义中没有 StackFull ( ) ，Add 程序与在程序 2-1-1 中也稍有不同。在链栈中实现 StackFull ( ) 的方法之一是采用如下语句：struct node * temp = new node ; 。对于删除操作，当然完成： 如果 temp 非空，就删除 temp 并返回 true ，否则返回 false 。但是这样会导致插入操作效率不高。为此如下设计删除程序：

```
bool Stack :: Delete ( Type & item )
{   if ( StackEmpty ( ) ) {
        cout << "Stack is empty" << endl ;
        return false ;
    }
    else {
        struct   node   * temp ;
        item = top -> data ; temp = top ;
        top = top -> link ;
        delete   temp ; return true ;
    }
}
```

如果栈为空，删除操作会产生错误信息 Stack is empty 并返回 false 。否则，栈顶元素的值存储到变量 item 中，保持指向第一个节点的指针，更新栈顶 top 指向下一个节点。返回删除的节点以便以后使用，最后返回 true 。

用链表存储比顺序数组 Stack [MaxSize] 更耗空间，然而链表有更大的灵活性，对于许多结构可以同时使用同一个可用的存储池。这两种表示方式的插入和删除操作的次数都与栈的大小无关。

采用数组 q [MaxSize] 可以有效地表示队列，并可以看成是循环的。除了数组，队列类的定义包含了 3 个整型变量：front , rear , MaxSize ，通过增加变量 rear 到下一个空位来插入元素。当 rear = MaxSize – 1 时，如果 q [0] 处有空位，则新元素就插入到该处。变量 front 总是指向队列中第一个元素沿逆时针方向相邻的位置，当且仅当队列为空时，front = rear 成立。初始设置为 front = rear = 0 。图 2.3 描述一个容量 n > 4 并包含从 J1 到 J4 四个元素的循环队列的两种可能的情况。

程序 2-1-3 队列的类定义

```
# define NULL    0
# include < iostream.h >
class queue
{
    private :
        Type * q ;
        int    front , rear , MaxSize ;
    public :
        Queue ( int MSize ) ; MaxSize (MSize )
        { q = new Type [MaxSize ] ; rear = front =0 ; }
        ~Queue ( )
        { delete [ ]    q ;}
        bool    AddQ (Type item ) ;
        bool    DeleteQ ( Type & item ) ;
        bool    QFull ( ) ;
        bool    QEmpty ( ) ;
}
```

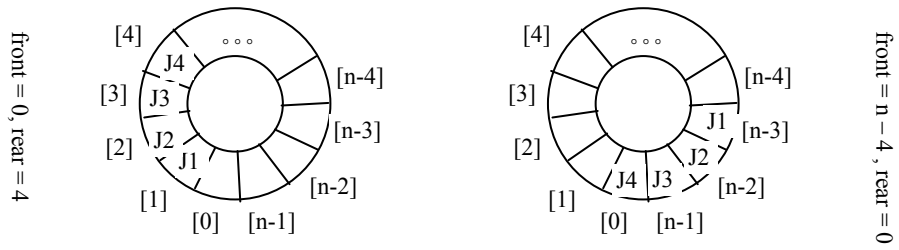


图 2-1-2 含有 4 各元素 J1、J2、J3、J4 的循环队列

要插入一个元素，需要顺时针移动 rear 一个位置。这可以使用下面代码实现：

```
if ( rear = = MaxSize – 1 )    rear = 0 ;
else rear ++ ;
```

但实现该操作的更好的方法是使用内置的计算余数的操作％ 。在插入之前，通过 rear = (++ rear) % MaxSize ; 来递增 rear 指针。类似地，每次删除时，要顺时针 移动 front 一个位置。程序 2-1-4a 和程序 2-1-4b 表明：通过把队列看作循环数组，队列的插入和删除操作能在固定的时间（O(1)）内得以实现。

2.2 树

树 T 是含有一个或多个节点的有限集合，其中有一个是特别指定的节点称为根节点，其余的节点被分割成 $n \geq 0$ 个互不相交的集合 $T_1, \cdots, T_n$，这里的每个集合都是一棵树，这些集合称为树 T 的子树。没有子树的节点称为叶节点，既不是根节点又不是叶节点的节点称为内节点。如果节点 X 不是叶节点，则 X 的子树的根节点称为 X 的子节点（或孩子），X 自然就称为这些子节点的父节点（或双亲）。叶节点的度定义为零，其它节点的度为其子节点的个数。从根节点到节点 X 的路径上的所有节点统称为 X 的祖先。

节点的层（层次）递归地定义为：根节点为第 1 层；如果节点 X 位于第 p 层，则它的子节点的层次为 $p+1$ 层。各个节点层次的最大值定义为该树的高度（或深度）。如下图

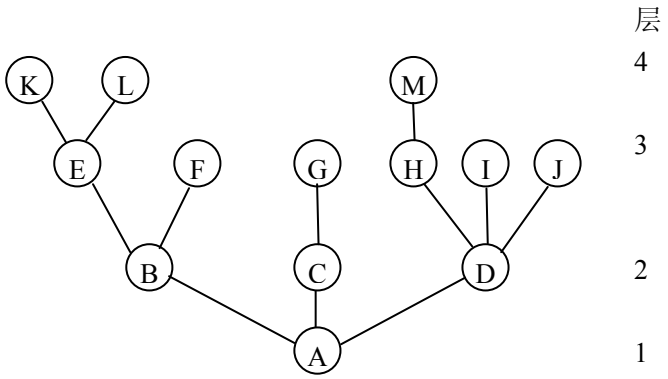
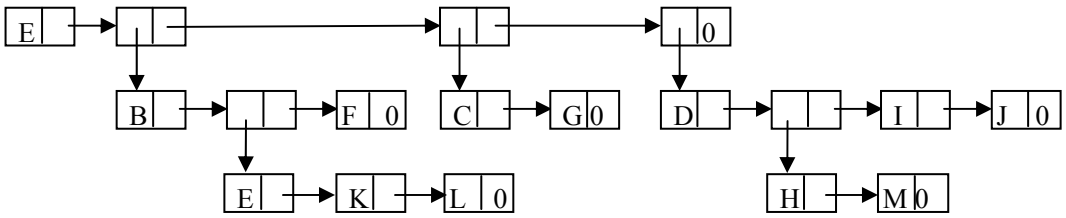


图 2-2-1 树的例

上图很像一棵树，但一般情况下树的示意图都是倒着画的。$n \geq 0$ 个不相交的树的集合称为森林，所以森林可以是空的。将一棵至少有 2 个子树的树的根删掉，则会得到一个森林。

如何在计算机内存中表示一棵树呢？若用链表来表示，其中链表中的每个节点与树的每个节点相对应。由于各个节点的子节点数目不同，每个节点将有不同数目的域。然而在编写算法时，数据中节点的大小固定常常是比较简单的。所以，通常选用节点大小固定的列表结构来表示一棵树。下图给出了图 2-2-1 中树的一种列链 tag = 1 时，data 域不再保存数据项，而是一个指向列表的指针。通过将根存入第一个节点，并将随后的节点指向子列表（每个子列表包含根的一棵子树）来表示一棵树。



节点的 tag 域为 1，表明该节点有向下的指针；否则 tag 域为 0

图 2-2-2 图 2-2-1 中树的列表表示

二叉树是一种特殊的树，每个节点的子节点至多有两个。但允许零个节点情况存在。

定义 2.2.1 二叉树是一个有限的节点集合，该集合要么为空，要么包含一个根节点和两棵不相交的二叉树，这两个二叉树分别称为左子树和右子树。

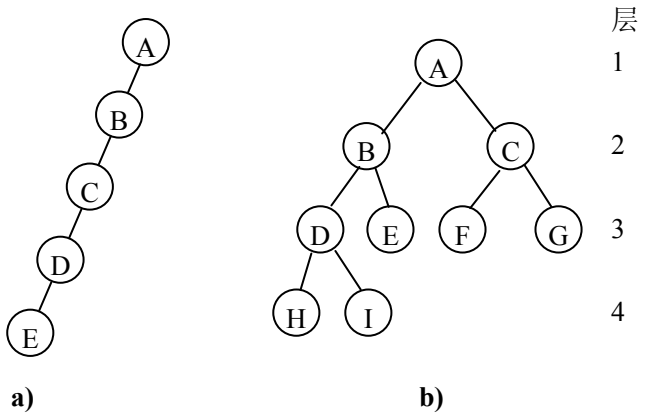


图 2-2-3 二叉树的例

直接计算可知，二叉树的第 i 层节点的数目至多为 2^{i-1}，而高度为 k 的二叉树的节点个数不超过 2^k-1，如果等于 2^k-1，则说这个二叉树为满二叉树。二叉树节点的编号是从第 1 层到第 2 层，等等，每一层都从左子节点开始。根节点编号为 1，然后每数一个加 1，这样最后一个叶节点的编号是 2^k-1。具有 n 个节点高度为 k 的二叉树，当且仅当它的节点编号与高度为 k 的满二叉树的前 n 个节点编号一致时，才说该二叉树是完全二叉树。具有 n 个节点的完全二叉树可以被压缩存储在一维数组 tree[n+1]，其中，编号为 i 的节点存在 tree[i] 中。这种存储可以简单地确定每个节点的父节点、左子节点、右子节点，而不用任何链接信息。

引理 2.2.1 如果按照上述顺序表示具有 n 个节点的完全二叉树，则对于索引为 i 的节点有下列性质：

i. 如果 $i \neq 1$，则 i 的父亲 $\text{parent}(i)$ 位于 $\lfloor i/2 \rfloor$；如果 $i=1$，该节点为根节点，无父亲。

ii. 如果 $2i \le n$，则左子节点 $\text{lchild}(i)$ 位于 $2i$；如果 $2i > n$，则该节点没有左子节点。

iii. 如果 $2i+1 \le n$，则右子节点 $\text{rchild}(i)$ 位于 $2i+1$；如果 $2i+1 > n$，则该节点没有右子节点。

一般二叉树也可以这种表示，但是浪费空间很大，如高度为 k 的右斜二叉树，在最坏情况下，需要 2^k-1 个位置，却只使用了其中的 k 个。此外，这种表示法具有顺序表示法的普遍缺点，当插入和删除节点时，为了反映其余节点层次的变化，需要移动表中许多节点。通过链表表示很容易克服这个缺点。每一个节点有三个域：lchild、data、rchild。尽管这种结构使确定父节点变得困难，但对于大多数应用还是合适的。对于需要频繁确定父节点这一情

况，可以包含第 4 个域：parent。

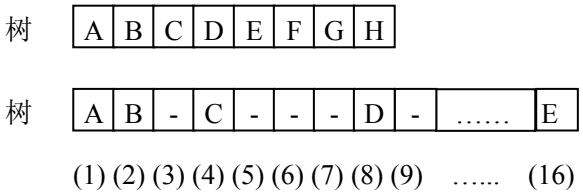


图 2-2-4 树的顺序表示

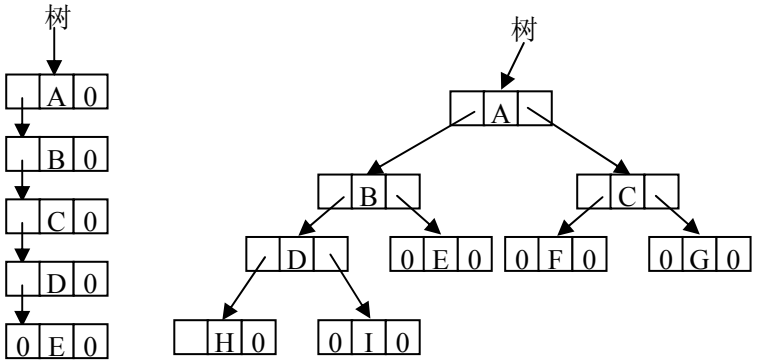


图 2-2-5 图 2-2-4 中二叉树的链表表示

2.3 二叉查找树

定义 2.3.1 二插查找树是一棵二叉树，可以为空二插树，如果不为空，则满足下列要求：

i. 根节点有一个关键字；

ii. 若其左子树不空，则左子树上所有节点的关键字均小于其根节点的关键字；

iii. 若其右子树非空，则右子树上所有节点的关键字均大于根节点的关键字；

iv. 其左右子树也分别为二插查找树。

二插查找树支持查找、插入和删除操作。事实上，可以通过关键字和顺序号来检索数据节点（如检索关键字为 x 的节点，检索第 k 小节点，删除关键字为 x 的节点，删除第 k 小节点，插入一个节点并确定其顺序号，等等）。

程序 2-3-1 二插查找树的类定义

```
class TreeNode{
    friend class BSTree;
    private:
        TreeNode * lchild, * rchild ;
        Type data ;
}
class BSTree
{
```

```
private:
   TreeNode * tree ;
   TreeNode * Search (TreeNode * t , Type x ) ;
Public:
    BSTree ( ) {tree = NULL ; }
    ~BSTree ( ) {delete tree ; }
    TreeNode * Search (Type x ) ;       // recurvely search for x .
    TreeNode * ISearch ( Type x ) ;    // Iteratively search for x .
    void Insert ( Type x ) ;    // Insert x .
    void Delete (Type x ) ;    // Delete x .
};
```

2.3.1 二叉查找树的检索

根据二插查找树定义的递归性质，容易设计一个递归的检索方法。假设要检索一个关键字为 x 的节点。节点通常是含有几个域的任意结构，key 是其中的一个域。为简化起见，假定节点仅由 Type 类型的 key 构成。下面将交换使用元素与关键字这两个术语。从根节点开始，若树的根为 NULL，则待检索的树没有元素，因此检索失败；否则，将 x 同根节点的关键字进行比较，若 x 等于根节点的关键字，则检索成功，终止；若 x 小于根节点的关键字，则在右子树中不会存在关键字为 x 的节点，只需在根节点的左子树中检索 x；若 x 大于根节点的关键字，则按类似的方法在根节点的右子树中检索。程序 2-3-2 描述了这个递归检索算法。采用链表结构表示查找树，每个节点有 3 个域：左指针域 lchild、右指针域 rchild、数据域 data 。也可以用程序 2-3-3 的 while 循环实现。

程序 2-3-2 二插查找树的递归检索

```
# define NULL    0
TreeNode * BSTree :: Search ( Type x )
{     return Search ( tree , x ) ; }
TreeNode * BSTree :: Search ( TreeNode * t , Type x )
{     if ( t = = NULL ) return 0 ;
      else if ( x = = t -> data ) return t ;
      else if ( x < t -> data ) return ( Search (t -> lchild , x )) ;
      else return ( Search ( t -> rchild , x )) ;
}
```

程序 2-3-3 二插查找树的迭代检索

```
# define NULL 0
```

```
TreeNode * BSTree :: ISearch ( Type x )
{   bool found = false ;
    TreeNode * t = tree ;
    while ( ( t ) && ( !found )) {
        if ( x = = t -> data ) found = true ;
        else if ( x < t -> data )    t = t -> lchild ;
        else    t = t -> rchild ;
    }
    if ( found ) return t ;
    else return NULL ;
}
```

如果二插查找树的高度为 h ,则上述检索的时间复杂度为 $O(h)$ 。如果根据顺序号检索，每个节点应该有一个附加的域 leftsize，其值等于该节点左子树中节点的个数加 1。算法 2-3-4 给出检索第 k 节点的过程。

程序 2-3-4 二插查找树的顺序号查找

```
# define NULL    0
class TreeNode {
    friend class BSTree ;
    private :
        TreeNode * lchild , * rchild ;
        Type data ; int leftsize ;
};
TreeNode * BSTree :: Searchk ( int k )
{   bool found = false ;    TreeNode * t = tree ;
    while ( ( t ) && ( !found )) {
        if ( k = = ( t -> leftsize ) ) found = true ;
        else if (t < ( t -> leftsize ) )    t = t -> lchild ;
        else {
            k - = ( t -> leftsize ) ;
            t = t -> rchild ;
        }
    }
    if ( found ) return t ;
```

```
    else return NULL ;
}
```

顺序号插值的时间复杂度也为 $O(h)$ 。

2.3.2 二叉查找树的插入

要插入一个新的元素 x ，首先要验证 x 的关键字不同于那些已存在节点的关键字。实现这一点要进行检索操作。若检索失败，则在检索终止处插入该元素。如在图 2-3-1a 所示的二插查找树中插入关键字为 80 的节点，首先检索关键字为 80 的节点，检索过程失败，最后一个检查的节点的关键字为 40 的节点，插入新节点使其称为该节点的右子节点，结果如图 2-3-1b 所示。图 2-3-1c 描述了在图 2-3-1b 所示的二插查找树中插入关键字为 35 的元素后的结果。

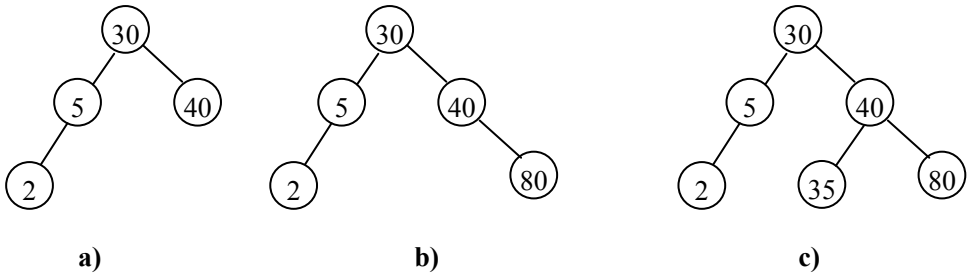


图 2-3-1 二插查找树的插入

程序 2-3-5 给出了上述插入方法的算法描述。若节点含有 leftsize 域，则需要更新该数组。

程序 2-3-5 二插查找树的插入

```
# define NULL    0
void BSTree :: Insert ( Type x )
   //insert x into the binary search tree.
{   bool found = false ;
    TreeNode * p = tree , * q ;
    //Serch for x . q is parent of p .
    while ( ( p ) && ( !found ) ) {
        q = p ; // Save p .
        if ( x = = ( p -> data) )    found = true ;
        else if (x < ( p -> data ) )    p = (p -> lchild ) ;
        else { p = ( p -> rchild ) ;
      }
       // perform insertion .
```

```
    if ( !found ) {
    p = new TreeNode ;
    p -> lchild = NULL ; p -> rchild = NULL ; p -> data = x ;
    if ( tree ) {
       if ( x < ( q -> data ) )    q -> lchild = p ;
       else q -> rchild = p ;
    }
    else    tree = p ;
}
```

插入操作所需要的时间为 $O(h)$ 。

2.3.3 二叉查找树的的删除

在二插查找树中删除一个叶子节点很容易。如从图 2.3.1c)所示的树中删除关键字为 35 的节点，只需要将其父节点的左子域设为 NULL，该节点就被删除了；删除只有一个子节点的非叶子节点也很容易，只需删除包含该元素的节点本身，同时将该节点的子节点上移，取代其位置。如，从图 2.3.1b)所示的树中删除关键字为 5 的节点，只需修改其父节点（关键字为 30 的节点）的指针，使其指向单子节点（关键字为 2 的节点）即可；删除有两个孩子的非叶子节点，可取其左子树中具有最大关键字的节点或取其右子树中具有最小关键字的节点替代要被删除的节点，然后从子树中的相应位置处删除被取节点。如，希望从图 2.3.1c)所示的树中删除关键字为 30 的节点，只需以其左子树中关键字（为 5）最大的节点或以右子树中关键字（为 35）最小的节点取代欲被删除的节点，然后删除被选中的那一个节点。比如，若选择关键字为 35 的节点，则首先将欲被删除的节点的数据修改为 35，然后将关键字为 35 的节点的父节点的左指针设为 NULL 即可。可见，在高度为 h 二插查找树中执行删除操作的时间为 $O(h)$ 。

2.4 优先队列

提供检索最小（或最大）、插入和删除最小（或最大）元素操作的任何数据结构均称为优先队列。表示优先队列最简单的方法是采用无序线性表。假设队列中有 n 个元素，并且该队列支持删除最大元素的操作。用顺序结构表示的线性表在表尾插入一个元素很容易，插入操作所需时间为 $\Theta(1)$ 。如果要删除，需要检索关键字最大的元素，因为在无序队列中查找这样的元素所需时间为 $\Theta(n)$ ，所以删除操作所需时间为 $\Theta(n)$ 。还可以采用有序线性表来表示一个优先队列，假设队列中每个元素按非降序排列，则删除操作所需时间为 $\Theta(1)$ ，而插入操作所需时间为 $\Theta(n)$ 。一种均衡删除和插入操作的有序队列结构是堆，若采用最大堆，

则插入和删除操作所需时间均为 $\Theta(\log n)$ 。

2.4.1 堆的基本操作

定义 2.4.1 最大（最小）堆是完全二叉树，具有这样的性质：树中所有非叶子节点的值均不小于（或不大于）其子节点（如果存在的话）的值。

在最大堆中，最大元素在树的根部。可以用数组表示最大堆。向最大堆中插入一个关键值为 x 的元素，是从堆的底部向根检索插入位置，并使得插入节点后仍为完全二叉树，且具有最大堆的性质（程序 2-4-2）；从最大堆中删除一个节点是删除根节点，但是剩下的两棵子树要整合成新的最大堆，整合的方法是：取下底部节点 x ，然后从原树的根部向底部检索插入的位置，实现的方法是：将数组中的最后一个元素复制到（原堆的）根，然后调用 Adjust(array,1,Nel)，写在程序 2-4-3 中。下面的算法给出了最大堆的定义、插入操作及删除操作。

程序 2-4-1 最大堆的类定义

```
class Heap
{
    private :
      Type * array ;
      Int MaxSize , Nel ;
          // Max , size of the heap , no. of elements
      void Adjust ( Type a [ ] , int i , n ) ;
    public :
      Heap ( int MSize ) : MaxSize ( MSize )
      { array = new Type [ MaxSize + 1 ] ; Nel = 0 ; }
      ~Heap ( ) { delete [ ] array ; }
      bool Insert ( Type item ) ;
          // insert item
      bool DelMax ( Type & item ) ;
          // delete the maximum .
}
```

程序 2-4-2 最大堆的插入

```
bool Heap :: Insert ( Type item )
{   // Insert item
    int i = ++ Nel ;
    if ( i = = MaxSize ) { cout << “heap size exceeded”   << endl ;
          return false ;
```

```
        }
        while ( ( i > 1) && ( array [ i/2 ] < item ) ) {
              array [ i ] = array [ i/2 ] ; i / = 2 ;
         }
         array [ i ] = item ;    return true ;
    }
```

最大堆插入操作的复杂度为 $O(\log n)$

例释：图 2-4-1a) 给出一个具有 5 个元素的最大堆。由于堆是完全二叉树，当加入一个元素形成 6 元素的堆时，其结构形如图 2-4-1b) 所示。如果被插的元素值为 1，则插入后该元素成为 2 的左子节点；若被插元素的值为 5，则该元素不能成为 2 的左子节点，应该把 2 下移为左子节点，如图 2-4-1c) 所示，同时还要决定在最大堆中，5 是否该占据 2 原来的位置。由于父节点 20 大于新插入元素 5，因此 5 可以插在 2 原来所在的位置；又若被插元素的值为 21，则如同图 2-4-1c)一样，把 2 下移为左子节点，由于 21 比原 2 位置的父节点的值还大，所以 21 不能占据原 2 的位置，需要把 20 移到其右子节点，21 插入堆的根节点，如图 2-4-1d) 所示。

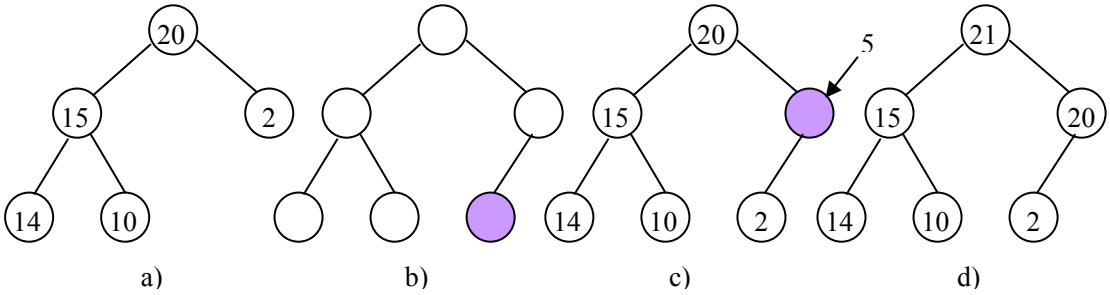


图 2-4-1 最大堆的插入

程序 2-4-3 最大堆的删除

```
void Heap :: Adjust ( Type a [ ] , int i , int n )
   // the complete binary trees with roots 2*i and 2*i + 1 are combined
   // with node i to form a heap rooted at i . No node has an address
   // greater than n or less than 1 .
{
    int j = 2*i ,    item = a [ i ] ;
    while ( j <= n) {
        if ( ( j < n ) && ( a [ j ] < a [ j+1 ] ) )    j ++ ;
           // compare left and right child and let j be the larger child
```

```
            if ( item >= a [j ] ) break ;
                // A position for item is found
            a [ j/2 ] = a [j ] ;    j * = 2 ;
        }
        a [ j/2 ] = item;
    }
    bool Heap :: DelMax ( Type & item )
    {   if ( !Nel ) { cout << "heap is empty" << endl ;
            return false ;
        }
        item = array [1] ; array [1] = array [ Nel -- ] ;
        Adjust ( array , 1 , Nel ) ;    return true ;
    }
```

Adjust 操作的时间复杂度为 $O(\log n)$ 。

例释：从图 2-4-1d)所示的最大堆中删除一个元素，首先将 21 从根部移出，剩下的 5 个元素分布在两个子堆中，需要将这两个子堆重新构造成一个最大堆。为此可以移动位置 6 中的元素，即 2，这样就得到了正确的结构，如图 2-4-2a)所示。但此时根节点为空且元素 2 也不在堆中。如果将 2 直接插入根节点，得到的二叉树不是最大堆，所以，根节点的值应该取 2、根的左子节点的值、根的右子节点的值三数之最大者。这个值为 20，将其移到根节点，此时位置 3 形成一个空位。由于这个位置没有子节点，所以 2 可以直接插入，结果如图 2-4-2b)所示。

如果从图 2-4-2b)所示的最大堆中删除一个元素，则首先将 20 从根部删除，删除之后，堆的二叉树结构应该如图 2-4-2c) 所示。为得到这个结构，10 从位置 5 移出。如果将 10 放在根节点，结果并不是最大堆，此时需将根节点的两个子节点中最大的值，即 15，移到根节点，位置 2 出现空位。如果将 10 直接插入位置 2，则形成的树不是最大堆，因而将 14 上移，将 10 插入位置 4，结果如图 2-4-2d) 。

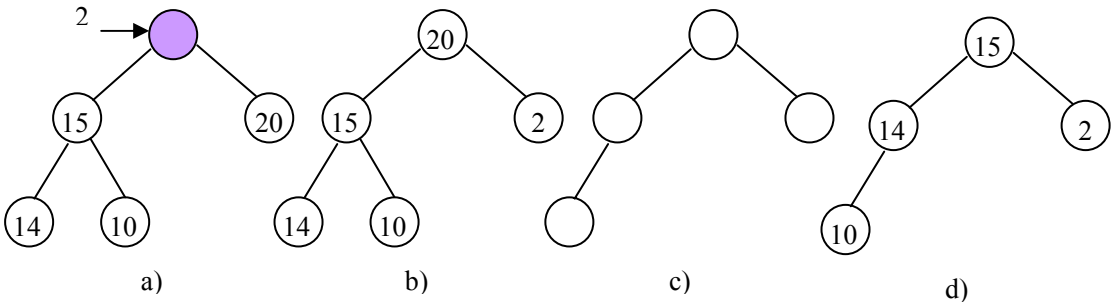


图 2-4-2 最大堆的删除

2.4.2 堆的初始化

给定 n 个元素的数组，可以用 Insert 建立堆，这要从空堆开始，通过逐渐插入元素的办法来完成。实际上，这相当于对已知数组元素进行排列。也可以通过使用类似于 Adjust 的函数 AdjustH 来建立堆，这时我们需要将数组看成一棵完全二叉树，然后在树上从底部到根部逐步进行调整。这两种创建堆的方法分别在程序 2-4-4 和程序 2-4-5 中实现。

程序 2-4-4 逐步插入建立一个最大堆

```
void Sort (Type a [ ] , int n )
  // Sort the elements a [1: n ] .
{   Heap heap ( SIZE ) ;
    for ( int i= 1 ; i <= n ; i+ + )
      heap . Insert ( a [i ] ; Type    x );
    for (i = 1 ; i <= n ; i + + ) {
      heap . DelMax (x ) ;    a [ n- i + 1 ] = x ;
    }
}
```

例释：图 2-4-3 给出了采用 Insert 程序将数据序列(40,80,35,90,45,50,70)建成堆的过程。→左边的树表示调用 Insert 之前数组 heap.array[1:i]的状态，→右边的树表示调用 Insert 之后数组 heap.array[1:i]的状态。整个过程中，数组都被表示成完全二叉树。

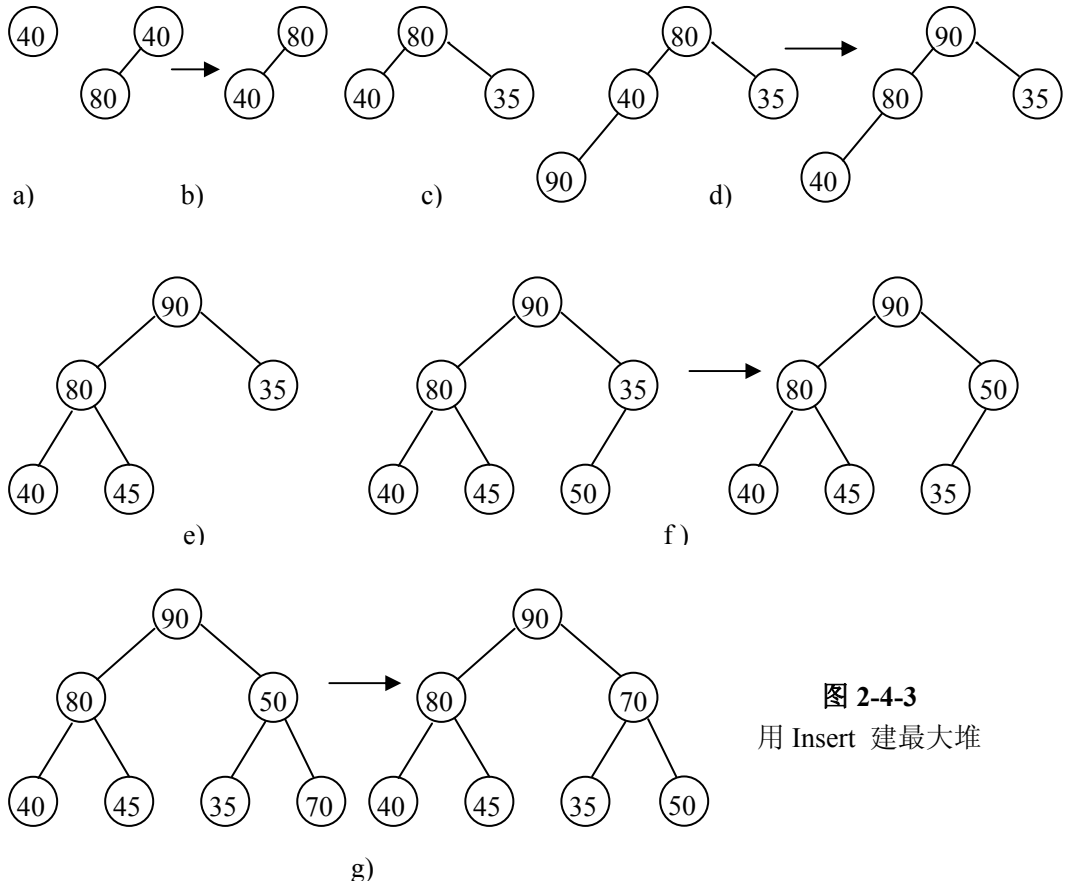


图 2-4-3
用 Insert 建最大堆

在最坏情况下，数据数组的元素按递增排列，每个新元素将会成为新的根节点。在完成二叉树的第 i 层上，至多有 2^{i-1} 个节点，$1 \le i \le \lceil \log_2(n+1) \rceil$。第 i 层上的节点到根的距离为 $i-1$，因此，使用 Insert 创建堆在最坏情况下所需时间为

$$\sum_{1 \le i \le \lceil \log_2(n+1) \rceil} (i-1)2^{i-1} < \log_2(n+1)2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

程序 2-4-5 创建 n 个任意元素的堆

```
void Heapify (Type a [ ] , int n )
   // Readjust the elements in a [1: n ] to form a heap .
{
    for ( int i = n/2 ; 1 ; i- - )    AdjustH (a , i , n ) ;
}
```

例释：图 2-4-4 给出用 AdjustH 构建最大堆的过程，这一过程将图 2-4-4a)所示的具有 $n=10$ 个节点的完全二叉树转化成最大堆。从已知完全二叉树最后一个具有子节点的节点（即值为 10 的节点）开始，这个节点一定是最后一个节点的父节点，其位置是 $i=\lfloor 10/2 \rfloor$。如果以这个元素为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为最大堆。随后，继续检查以 $i-1, i-2$ 等节点为根的子树，直到检查到整个二叉树的根节点（其位置为 1）。

最初 $i=5$，由于 $10>1$,以位置 i 为根的子树已是最大堆。下一步检查根节点在位置 4 的子树。由于 $15<17$，不是最大堆。为将其调整为最大堆，可将 15 与 17 进行交换，得到的树如图 2-4-4b)所示。然后检查以位置 3 为根的子树，为使其变为最大堆，将 80 与 35 进行交换。之后，检查根位于位置 2 的子树，通过重建过程该子树变为最大堆。将该子树重构为最大堆时需确定子节点中值较大的一个，因为 $12<17$，所以 17 成为重构子树的根，下一步将 12 与位置 4 的两个子节点中两个值较大的一个进行比较，由于 $12<15$，15 被移到位置 4，空位 8 没有子节点，将 12 插入位置 8，形成的二叉树如图 2-4-4c)。最后，检查位置 1，这时以位置 2 或位置 3 为根的子树已是最大堆，然而 $20<(\max\{17,80\})$，因此，80 成为最大树的根，当 80 移入根，位置 3 空出。由于 $20<(\max\{35,30\})$，位置 3 被 35 占据，最后 20 占据位置 6。图 2-4-4d)显示了最终形成的最大堆。

令 $2^{k-1} \le n < 2^k$，其中 $k=\lceil \log_2(n+1) \rceil$。对于第 i 层的节点，在最坏情况下，AdjustH 的迭代次数为 $k-i$。函数 Heapify 总共所需要的时间与下面的公式成正比：

$$\sum_{1 \le i \le k} (k-i)2^{i-1} = \sum_{1 \le i \le k-1} i2^{k-i-1} \le n \sum_{1 \le i \le k} i/2^i \le 2n = O(n)$$

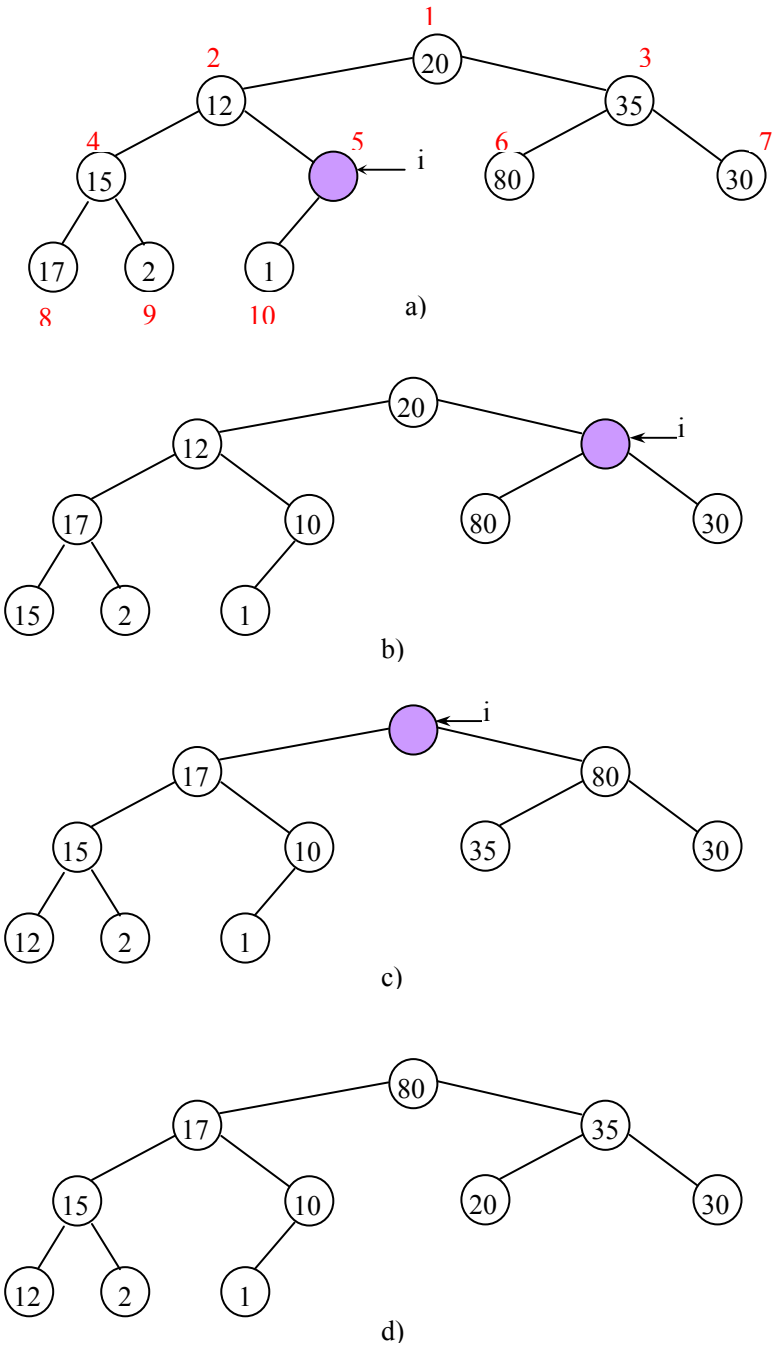


图 2-4-4 用 AdjustH 构建最大堆

2.5 集合的表示和运算

我们打算用树表示一个集合。比如，三个集合 $S_1=\{1,7,8,9\}, S_2=\{2,510\}, S_3=\{3,4,6\}$ 可以用图 2-5-1 中三棵树分别表示出来，当然，表示一个集合的树结构可能不是唯一的，这里的关键是能够判断一个元素是不是在给定的集合中，其中一种方法是将树中非根节点的原来指向其子节点的指针域换成一个指向其父节点的指针域，对于根节点，其父节点可以代表

集合的名，其值规定为 -1。这样，要检查节点 i 是否在已知的树 T 上，只需从节点 i 开始查找它的父节点 j，再查找 j 的父节点 k，直至查到节点 l，而节点 l 的父节点值为 -1 。此时就知道节点 i 是以节点 l 为根的树中的节点。这个过程称为查找，查找的结果记做 $\mathrm{Find}(i)$ 。如果事先已将某数组元素分成互不相交的集合，比如前面提到的三个集合 S_1, S_2, S_3，而每个集合都已经有树表示，则查找过程能够确定被查找元素属于那个子集合。

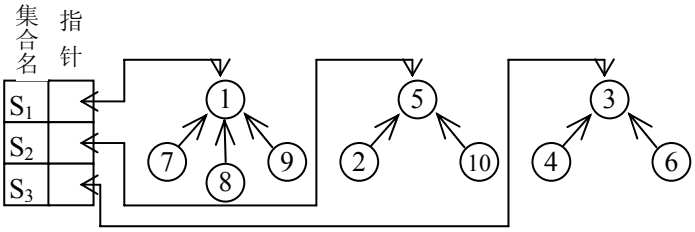


图 2-5-1 S_1、S_2、S_3的数据表示

上述三个子集合的树结构可以用各节点的指针组成的数组表示出来，图 2-5-2 给出了指针与元素的对应关系。

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

图 2-5-2 S_1、S_2、S_3的数组表示

当采用树表示后，合并两个不相交子集就非常简单，只需将代表一棵树的根节点指针修改成指向代表另一棵树的根节点即可。用这种方式合并两个根节点分别为 i, j 的子树形成的一棵更大的树，结果记做 $\mathrm{Union}(i, j)$ 。程序 2-5-1 给出了以上讨论的并和查找操作的算法。

程序 2-5-1 并和查找的简单算法

```
class Sets
{
    private :
      int * p , n ;
    public :
      Sets ( int Size ) : n (Size )
      {   p = new int [ n +1 ] ;
          for ( int i = 0 ; i <= n ; i + + )
                p [i] = -1 ;
       }
```

```
        ~Sets ( ) { delete [ ] p ; }
        void SimpleUnion ( int i , int j ) ;
        int SimpleFind ( int i ) ;
    }
    void Sets :: SimpleUnion ( int i , int j )
    {   p [i] = j ; }
    int Sets :: SimpleFind ( int i )
    { while ( p[i] >= 0 )   i = p [i] ;
      return i ;
    }
```

算法 2-5-1 也定义了类 Sets。虽然描述这两个算法很容易，但它们的性能不理想。例如，从 q 个单点集开始，即初始状态由含 q 个单点树构成的森林，$p[i]=-1, 1\le i\le q$，依次进行下列并和查找操作：

$$\text{Union}(1,2);\text{Find}(1),\text{Union}(2,3);\text{Find}(2),\text{Union}(3,4);\cdots;\text{Find}(q-2),\text{Union}(q-1,q)$$

则所得到的表示原 q 个单点的集合的树是一棵单枝的树。这里，$\text{Union}(i,j)$ 采用了规则：修改节点 i 的指针为指向节点 j。如果将此规则稍做修正，变为“向节点多的子树靠拢”，则不会出现上述情况。为此需在每棵树的根节点标出以其为根的树的节点个数 count，由于所有除根节点之外的节点的 p 域都是正数，因此可以在根的 p 域中以负数形式保留 count 值。

程序 2-5-2 使用带权规则的并算法

```
void Sets :: WeightUnion (int i , int j )
    // Union sets with roots i and j , i != j , using the weighting rule .
    // p[i] = = - count[i] , p[j] = = - count[j] .
{
    int temp = p[i] + p[j] ;
    if ( p[i] > p[j] ) { i has fewer nodes .
        p[i] = j ; p[j] = temp ;
    }
    else { // j has fewer or equal nodes .
        p[j] = i ; p[i] = temp ;
    }
}
```

例子 数组 $A=(1,2,3,5,6,7,8)$，考虑从初始状态 $p[i]=-\text{count}[i]=-1, 1\le i\le 8=n$ 开始，依次执行下列并操作的过程：

$$\text{Union}(1,2),\text{Union}(3,4),\text{Union}(5,6),\text{Union}(7,8);\text{Union}(1,3),\text{Union}(5,7);\text{Union}(1,5)$$

则结果如图 2-5-3 表示。

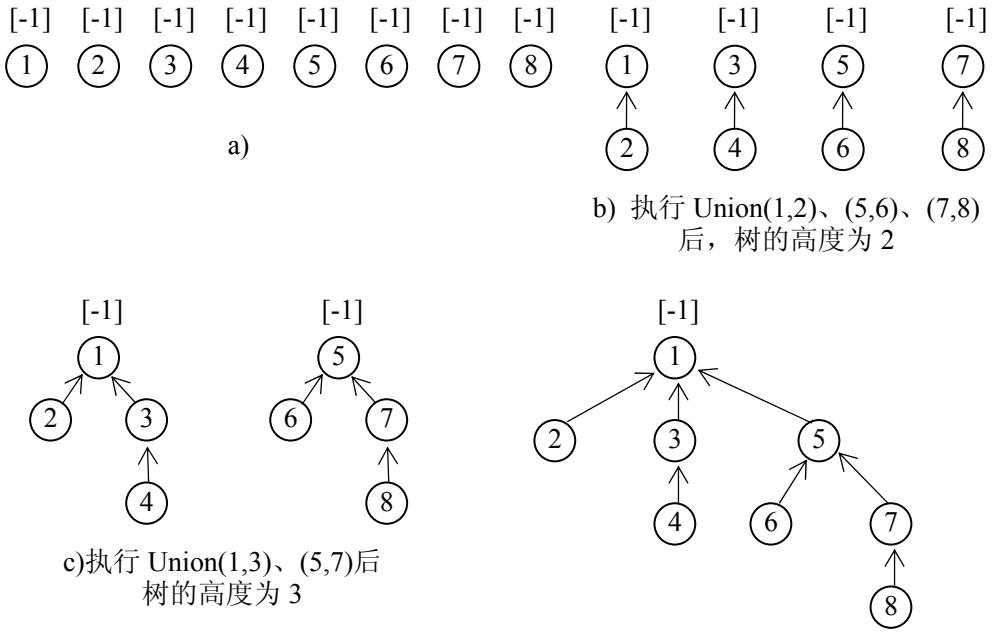


图 2-5-3 采用加权规则处理已知并序列产生的树

命题 2.5.1 假定以森林开始，森林中每一棵树有一个节点。令 T 是由一些并操作生成的具有 m 个节点的树，其中每个并的执行使用 WeightUnion 算法，则 T 的高度不超过 $\lfloor \log_2 m \rfloor+1$。

证明 当 $m=1$ 时，命题成立显然。假定对于有 $i(i\le m-1)$ 个节点的所有上述方法生成的树来说，命题都已成立，以下证明 $i=m$ 时，命题仍成立。令 T 是 WeightUnion 生成的具有 m 个节点的树。考虑最后的并操作 $\text{Union}(k,j)$。令 n 是树 j 中节点的个数，则树 k 中节点的个数为 $m-n$。不失一般性，假定 $1\le n\le m/2$。那么树 T 的高度或者与树 k 的相同，或者比树 j 的高 1。若是前一种情况，则树 T 的高度 $\le \lfloor \log_2(m-a) \rfloor+1 \le \lfloor \log_2 m \rfloor+1$。若是后一种情况，则树 T 的高度 $\le \lfloor \log_2 a \rfloor+2 \le \lfloor \log_2 m/2 \rfloor+2 \le \lfloor \log_2 m \rfloor+1$。 证毕

在图 2-5-3 所示的树中查找节点 8 的根需要移动三次指针，如果同时又要查找节点 7 的根还得需要移动两次指针，假如下一次又要查找节点 8 的根，则又要重新移动三次指针。实

际上，有些操作重复执行了。为避免这种浪费，可以采用压缩规则，使得第一次查找节点 8 的根时，就将由 8 到根节点路上的节点(包括节点 8)的指针修改成根节点标号（即 1）。这种做法实际是改变了原来表示集合的树的结构。实际上，表示集合 $A=\{1,2,3,4,5,6,7,8\}$ 的树最有利于查找的是如下的结构：

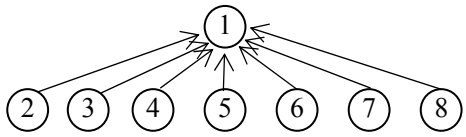


图 2-5-4 采用加权规则生成的最好 8 节点树

程序 2-5-3 采用压缩规则的查找算法

```
int Sets :: CompressFind ( int i )
   // Find the root of the tree containing element i . Use the compressing rule
   // to compress the paths to root for all nodes from i to the root .
{
    int r = i ;
    while ( p[r] > 0 ) r = p[r] ;    // Find root .
    while ( i != r ) { // compress nodes from i to root r .
        int s = p[i] ; p[i] = r ; i = s ;
    }
    return ( r ) ;
}
```

在图 2-5-3d)所示的树中，执行一次 CompreesFind(8)后得到树结构为

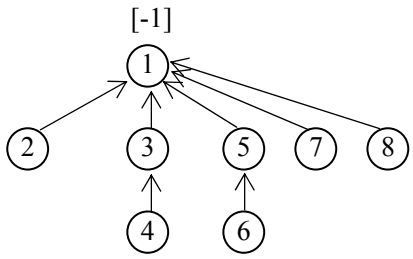


图 2-5-5 执行查找后原树结构发生了变化
更有利于查找操作

命题 2.5.2 假定从 n 个单点树的森林开始，执行了 m 次并和 s 次查找的混合操作序列。记所需的最大时间需求为 $T(s,m)$，则当 $m \ge n/2$ 时，有正常数 k_1, k_2，使得

$$k_1\left(n+s\cdot\alpha(s+n,n)\right) \le T(s,m) \le k_2\left(n+s\cdot\alpha(s+n,n)\right)$$

其中 $\alpha(p,q)=\min\left\{i\ge 1 \mid A(i,\lfloor p/q\rfloor)>\log_2 q\right\}$, $p\ge 1, q\ge 1$，而 $A(i,j)$ 为 Ackermann 数：

$$A(1,j)=2^j, \text{ for } j\ge 1$$
$$A(i,1)=A(i-1,2), \text{ for } \mathrm{i}\ge 2$$
$$A(i,j)=A(i-1,A(i,j-1)), \text{ for } \mathrm{i},j\ge 2$$

2.6 图的表示

本节陈述图的三种数据表示方法：邻接矩阵、邻接链表和邻接多重表。始终用 $G=(V,E)$ 代表一个简单图，其节点数 $n\ge 1$。下面是几个图的例子

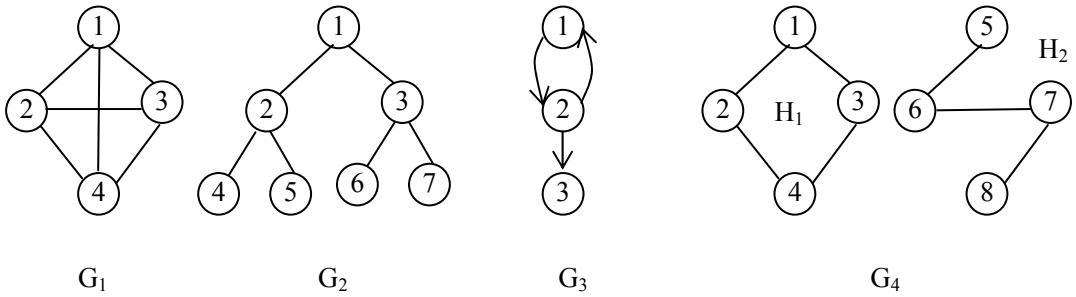


图 2-6-1 几个图例

2.6.1 邻接矩阵

G 的邻接矩阵是 $n\times n$ 的 2 维数组，记为 a：当且仅当 (i,j) 是图 G 的一条边时，元素 $a[i][j]=1$，否则 $a[i][j]=0$。根据邻接矩阵很容易知道是否有一条边链接顶点 i 和 j。对于无向图，任何一个顶点的度等于邻接矩阵中对应行的元素之和：$\sum_{1\le j\le n} a[i][j]$。对于有向图而言，行元素之和等于对应顶点的出度，而列元素的和等于对应顶点的入度。如果回答图有多少条边？是否连通？等问题，采用邻接矩阵表示时，需要的时间为 $O(n^2)$，因为邻接矩阵除对角线元素外，其余 n^2-n 个元素都要检查。

2.6.2 邻接链表

考虑连通问题时，只需检查图中顶点的链接情况。若图的边不太多，则不必查那些与之不邻接的顶点。采用邻接链表能够有效减少这些不必要的检查。在图的这种表示中，n 行的邻接矩阵被 n 个链表代替。每个顶点都有一个链表，链表 i 上的节点代表了与顶点 i 相邻的顶点。每个节点至少有两个域：顶点和指针。注意每个链表中的节点并不要求按顺序排列。每个链表都有一个头节点。所有的头节点按顺序排列，因而可以很容易的随机访问到邻接链表中任何特定的节点。

程序 2-6-1 邻接链表的类定义

```
class Graph
{
    private :
      int n ; // Number of vertices
      struct node {
          int vertex ;
          struct node * link ;
      }
      struct node * headnodes [n+1] ;
  public :
      Graph ( )
      { for ( int i = 1 ; i <= n; i ++ ) headnodes [i] = NULL ; }
}
```

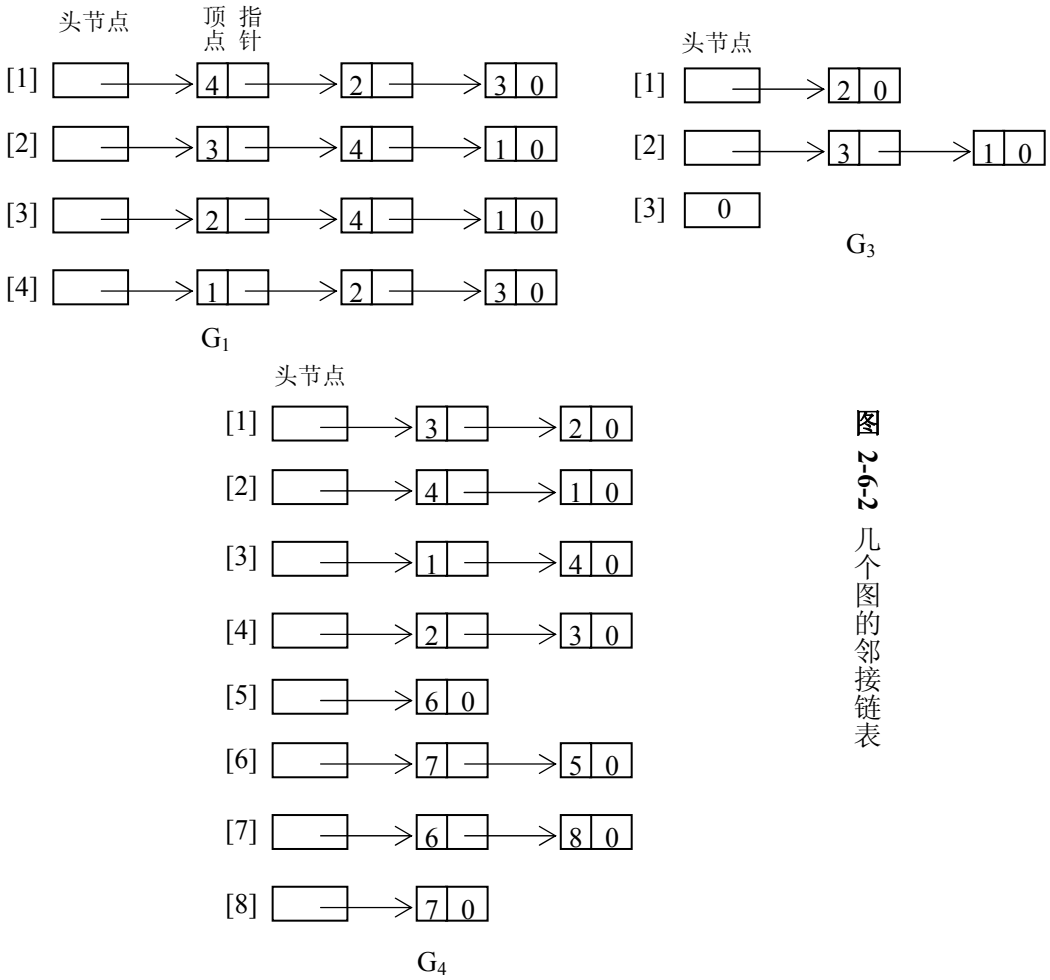


图 2-6-2 几个图的邻接链表

对于有 n 个顶点和 e 条边的无向图，这种表示法需要 n 个头节点和 $2e$ 个表节点。每个表

节点有两个域。由于表示一个值为 m 的数需要占用 $O(\log m)$ 位，所以头节点需要乘以 $\log n$，表节点需要乘以 $\log n + \log e$。通常可以顺序地**压缩存储邻接表**中的节点，因而可以不用指针。这种情况下，可以利用一个数组 $\text{node}[n+2e+2]$。$\text{node}[i]$ 给出了顶点 $i(1 \le i \le n)$ 对应的链表在数组中的起始位置。并且 $\text{node}(n+1)$ 被设置为 $n+2e+2$。顶点 i 的邻接点存储在 $\text{node}[i],\cdots,\text{node}[i+1]-1, 1 \le i \le n$。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10	12	14	16	18	19	21	23	24	3	2	4	1	1	4	2	3	6	7	5	6	8	7

图 2-6-3 图G_4的顺序表示

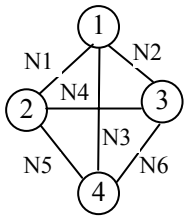
对于有向图而言，其表节点的个数仅仅是 e。任何一个顶点的出度可以由该顶点的邻接表中的表节点个数来确定，因而确定图的有向边的总数需要 $O(n+e)$ 的时间。

2.6.3 邻接多重表

在无向图的邻接表表示中，每条边 (u,v) 由两项来表示，表中的一项来表示 u，而另一项表示 v。在一些应用中需要能够确定某一条指定的边的第二项，并且，当那条边被检查过之后要进行标记。如果将邻接表改造成多重表（即表中的节点可以被多个表共享），那么很容易达到上面提出的要求。多重表仍以顶点作为头节点，而以边作为表节点。表节点由 5 个域组成，结构为

flag	顶点 1	顶点 2	表 1	表 2

其中，flag是占一个位的标志域，可以用来指示这条边是否已经检查过。当我们把所有的边编号以后，所有以 v 为头的边按照编号从小到大的顺序依次链起来，然后挂到顶点 v 上。图 2-6-4b) 给出了图G_1的邻接示意图，其边的编号如图 2-6-4a)，图 2-6-4c)给出了各边对应的



a) G_1

顶点 1：N1→N2→N3
顶点 2：N1→N4→N5
顶点 3：N2→N4→N6
顶点 4：N3→N5→N6

b)

c)

N1		1	2	N2	N4
N2		1	3	N3	N4
N3		1	4	0	N5
N4		2	3	N5	N6
N5		2	4	0	N6
N6		3	4	0	0

边 (1,2)
边 (1,3)
边 (1,4)
边 (2,3)
边 (2,4)
边 (3,4)

图 2-6-4 多重链表

表节点的内容，比如，节点 N1，它代表的边是(1,2)，而 N2、N4 是在链表 b)中链到 N1 的两个表节点。

程序 2-6-4 邻接多重表的定义

```
class Graph
{
    private :
      int n ;    // Number of vertices
      struct edge {
        bool    flag ;
        int vertex1 , vertex 2 ;
        struct edge * path1 , * path2 ;
      }
     struct edge * headnodes [n+1] ;
  public :
     Graph ( )
     { for (int i = 1 ; i <= n ; i++ ) headnodes [i] = NULL ; }
}
```

附录 3. ALGEN 语言

对于算法的描述可以采用一种高级计算机语言，如 C/C++、Java、Fortran、Basic 等，但是这些语言都要强调每一步实现的细节，这对于分析算法的性能不是完全必要的。描述算法最要紧的是表达清楚算法的基本思想和关键过程，选用简明易于理解同时便于人工或采用机器翻译成其它实际使用的计算机高级语言是必要的。为此，我们采用 ALGEN 语言，它与 ALGOL 语言和 PASCAL 语言相似，符合人们通常的表达习惯。

1．变量类型的声明

在 ELGEN 语言中，只作简单的变量类型区别，它们是 integer，real，bool 和 char。这主要考虑到对这些对象进行操作（或运算）的特殊要求。所有操作（或运算）过程都理解为精确的。当需要考虑数值计算的精度时，需在相应的位置加语句 digit:= n；表示保留小数点后 n 位。这个约束的作用从该语句开始直到出现语句 end{digit}时结束。此外，用 evalf(a,n)表示对数值 a 保留 n 位小数；用 floor(a)表示不大于数值 a 的最大整数，ceil(a)表示不小于数值 a 的最小整数。

2．保留字

变量的命名是以字母开头的字母与数字及下划线组成的字, 区分大小写, 但不准采用保留字, 这里的保留字主要包括以下字:

逻辑符: and、or、not、true、false

结束符：exit、return、end、break、continue

语句关键字： if 、elif、 then、 else ；for、from、by、to、do、while、in、loop、until

关系符：<、>、=、≤、≥、≠、……

3．对变量的赋值采用 v:=expr；其中 v 是变量（名），expr 是一个（表达式的）值。

4．for 循环语句

for i **from** s **by** d **to** t **do**

statements;

end{for}

当起始值 s 为 1 时，from s 可以省略；当步长 d 为 1 时，by d 可以省略；

for i **in** S conds **do**

statements;

end{for}

S 可以是一个集合、序列或表，conds 表示约束条件，可根据情况选取或不选取，所以是可选项。

5． while 循环与 loop 循环语句

while conds **do**

statements;

end{while}

loop

statements；

until conds；

end{loop}

6．if 条件语句

if cond0 **then**

statement0；

elif cond1 **then**

statement1；

……..

elif condk **then**

statementk；

else

statements；

end{if}

这里，elif、else 及它们后边的语句都是可选项。条件 cond0、cond1、…、condk 之间都是不相交的.

7．case 条件语句

为使用方便，ALGEN 也使用 case 条件语句，格式如下：

case :

c1: statement1;

…….

ck: statementk;

else statement;

end{case}

其中，else 是选项，而前面的 k 种情况不能为空，且所有情况之间是相互排斥的。

8．进程: proc name(formal parameters)

进程是指独立完成一个任务的程序段。一个进程可以调用其它进程，也可以被其它进程所调用。这里的进程有两种，一是纯过程，它可能会改变输入参数，但没有返回值；另一类有返回值，称为函数。前者的调用方式为 name(actual parameters); 后者的调用方式为 v:=name(actual parameters); 因而，后者必须带有

语句 retrurn(expr); 指出返回的内容。而前者是通过全局变量将信息返回给调用它的进程的。主进程的返回信息被存在内存中。

一个进程的完整格式为：

```
proc name (formal parameters)
    global variables;
    local variables;
    statements;
end{name}
```

9． break 强制结束当前循环体；continue 强制跳过本次循环；exit 强制结束当前进程；return 强制结束当前进程并返回后跟其括号中的内容。end 是任何进程代码结束的标志符。

10． go to label 从当前位置转到带有标号 label 的语句，label 后跟冒号:。这个语句用于跳出 while 循环体以及跳出递归程序体都是比较方便的，但尽量少用，因为它可能会使整个程序的逻辑关系变得晦涩难懂。

11．数组是所有数据结构的基础，这里用 A[1..n]表示一个长为 n 的一维数组，它的第 i 个元素记为 A[i]，而 A[k..m]表示子块，它是该数组的一个连贯部分：从元素 A[k]到元素 A[m]。二维数组定义为 A[1..m,1..n]，它对应于一个 m×n 矩阵。更高维的数组可类似定义，它们的子块也可类似表示。给数组赋值通过给元素赋值完成。

12．有时为叙述方便简洁，ALGEN 允许使用数学表达式和自然语言，但含义必须是清楚明确的。

13．每个执行语句都以 ; 号结尾；注释语句的每一行都以双斜杠 // 开始。