文章编号:1005－3751(2002)05－0001－06

# 利用优化哈夫曼编码进行数据压缩的探索

## Research of Data Compression with the Optimized Huffman Coding Algorithm

朱怀宏[1],吴　楠[1],夏黎春[2](1.南京大学 计算机科学与技术系,江苏 南京 210093;2.南京工业职业技术学院,江苏 南京 210093)

ZHU Huai-hong[1], WU Nan[1], XIA Li-chun[2] (1. Computer Sci. and Tech. Dept. of Nanjing Univ., Nanjing JS 210093, China; 2. Nanjing Ind. Prof. Tech. Coll., Nanjing JS 210093, China)

**摘　要**:数据压缩是当今计算机科学领域中十分活跃的论题。哈夫曼编码作为一种最常用的不等长无损压缩编码方法,在数据压缩程序中具有非常重要的应用。文章通过对传统静态哈夫曼编码的讨论以及与动态哈夫曼编码的对比,研究一种改进的数据压缩算法,并用程序实现之。

**关键词**:哈夫曼编码;优化;数据压缩;算法;源程序

**Abstract**: Data compression is a very active theme in the current computer science field. Huffman coding as one of the most in common use unequal－length and non－losing compressing coding algorithm has many important application to the current data compression field. Following the comparison of traditional Huffman coding algorithm and the dynamic optimized Huffman coding algorithm, this disquisition gives a discussion and research to this improved data compression algorithm, We realized the program with the C language.

**Key words**: Huffman coding; optimize; data compression; algorithm; source code

**中图分类号**:TP301.6　　**文献标识码**:A

## 1　引　言

我们正处在一个信息飞速增长的时代。随着科技与经济的迅速发展,海量的数据进入了我们的生活。二十年前,以兆字节为单位的存储要求也是异想天开的事情。可是现在,随着大容量存储设备的迅速发展,即使对于个人用户来说,存储上千兆字节的数据也很平常,并且离太字节(TB,$10^{12}$ Bytes)的存储量也不再遥远。如何正确而迅速的处理和保存这些数据就成为计算机科学中亟待解决的一大问题。因此,数据压缩技术已经成为计算机科学的主要分支。

所谓数据压缩,实际上就是对需要压缩的数据对象进行某种可逆性编码,使编码的总长度小于原数据的总长度,从而达到减小数据总长度的目的。

数据压缩的关键在于编码.编码就是基于由模型提供的概率分布来确定符号的输出表达方式。对于以数据压缩为目的的编码,一般的想法是对于常见的符号,编码器输出较短的码字;而对于少见的符号则用较长的码字表示。

下表列出了符号七个英文字母的编码,一个短语可以通过使用这个表中所给出的代码字去替换其中的符号来进行编码。例如,短语 FEED 的编码为 110101001。解码从左到右依次进行。代码输入从 110 开始,并且以此开头的惟一的代码字就是 F ,即第一个符号。接着执行其余符号的解码。

表 1　代码字和概率值

| 符　号 | 对应码字 | 出现概率 |
|---|---|---|
| A | 0000 | 0.05 |
| B | 0001 | 0.05 |
| C | 001 | 0.10 |
| D | 01 | 0.20 |
| E | 10 | 0.30 |
| F | 110 | 0.20 |
| G | 111 | 0.10 |

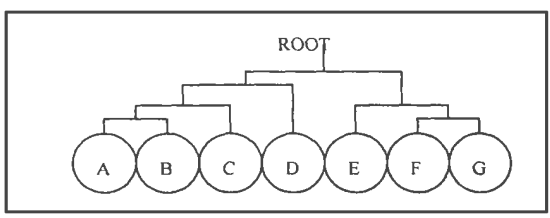


图 1　用于解码的逻辑树

图 1 给出了一个用于解码的逻辑树。树的遍历是通过从根部出发然后再到各分枝来实现的,各分支相当于文本代码中的下一个位。从根到各个符号(叶子)的路径相当于上表中的码字。这种类型的编码叫做前缀编码——或者更准确地说,无前缀编码——因为没

**收稿日期**:2002－01－03

有哪个码字会是另一个码字的前缀。不过,在图 1 中情况并非如此,解码树上会有符号出现在交叉结点上,导致解码出现非单值性。因此,这种编码方法是不可取的。

## 2 哈夫曼编码的原理

表 1 中的代码是通过一种叫做哈夫曼编码的技术得到的。是一种变长度编码方法。只要给定符号的概率分布,哈夫曼编码算法就能够计算出给定字符的代码。对于给定概率分布的无前缀代码,产生的码字可实现很好的压缩效果。哈夫曼编码通过从底向顶构造解码树来解码。例如前面所举的关于字符集 A～G 的示例,哈夫曼编码算法为每一个符号创造一个包含了这个符号和概率值的叶结点(如图 2 所示),然后把具有最小概率值的两个叶结点排列在同一个父结点下,成为兄弟结点,父结点的概率值等于两个子结点的概率值之和(见图 3)。

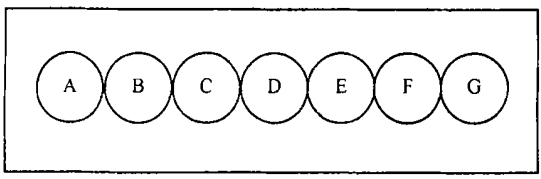


图 2 叶结点概率值

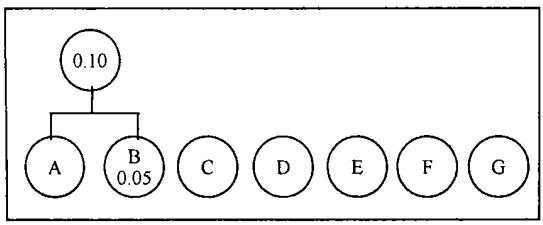


图 3 父结点概率值

忽略已连接的子结点,选择两个具有最小概率值的子结点重复进行连接操作。例如,A 与 B 已连接生成新的结点,下一步就是要把这个新的结点与结点 C 连接起来,产生一个概率为 P =0.20 的结点。将这个过程重复下去,直到只有一个结点没有父结点为止,这个结点即成为解码树的根(见图 4)。那么,两个没有叶结点的分支就被标记为 0 和 1(顺序并不重要)从而形成树。

生成哈夫曼树的具体步骤如下:

(1) 根据给定的 $n$ 个权值 $\cdots W_1, W_2, \cdots, W_j$ 构成 $n$ 棵二叉树的集合 $F = \{T_1, T_2, \cdots, T_n\}$,其中每棵二叉树 $T_i$ 中只有一个带权为 $W_i$ 的根结点,其左右子树均空。

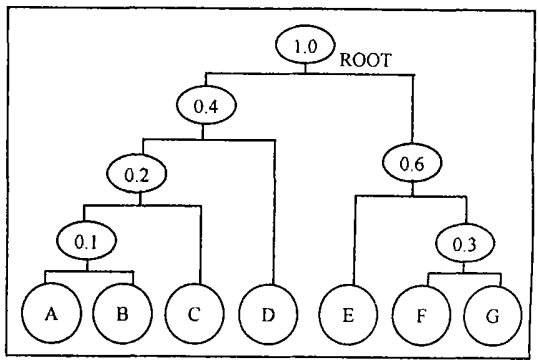


图 4 解码树

(2) 在结点集中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树,且置新的二叉树的结点的权值为其左、右子树上结点的权值之和。

(3) 在结点集中删除这两棵树,同时将新得到的二叉树加入结点集中。

(4) 重复(2)和(3),直到结点集中只含一棵树为止。这棵树便是哈夫曼树。

生成哈夫曼树之后,在树的左右分枝上赋值 1 和 0 即可得到哈夫曼编码。

由于哈夫曼树中没有度为 1 的结点,一棵有 $n$ 个叶子结点的哈夫曼树共有 $2n-1$ 个结点,可以以一个大小为 $2n-1$ 的向量表示。由于在构成哈夫曼树之后,为求得编码,需从叶子结点出发走一条从叶到根的路径;而为译码需从根出发走一条从根到叶子的路径。则对每个结点而言,既需知道其双亲的信息,又需知道其子结点的信息。

译码的过程是,分解原文中字符串,从根结点出发,按字符'0'或'1'确定访问子结点的路径,直至叶结点,便求得该子串相对应的字符。

上述的哈夫曼编码是利用压缩对象出现频率的不等性进行编码的一种常用的无损数据压缩方法。不论从算法的复杂度还是在实现的难度以及对数据压缩的效果来看,哈夫曼编码都不失为一种较好的数据压缩算法。

## 3 静态哈夫曼编码压缩算法的程序实现

为了深入理解哈夫曼编码的思想,作者按照以下步骤实现了静态哈夫曼编码的压缩和解压缩算法。

(1) 扫描原文件的全部数据,完成字符频度的统计。

(2) 依据字符频度建立哈夫曼树。

(3) 将哈夫曼树的信息写入输出文件(压缩后文件),以备解压缩时使用。

(4) 进行第二遍扫描,将原文件所有字符转化为哈夫曼编码,并以 ASCII 字符形式保存到输出文件。

(5) 在输出文件上做标记以标识压缩文件,完成对原文件的压缩。

该程序的算法源于静态哈夫曼编码的经典思想。即压缩时,首先扫描原文件的全部数据,生成字符(ASCII 字符)频率表。然后构造哈夫曼树并生成与字符相对应的不等长编码。再将生成的哈夫曼编码转换为字符写入文件中。最后,将文件结尾倒数第五个字符标记为"H",以标识压缩后的文件。解压缩时,将哈夫曼编码做逆变换后写入文件即可。这个程序在 Borland C++5.0 上调试成功。

通过测试,静态哈夫曼编码的压缩结果如下(解压缩后正常):

表 2　压缩结果 1

| 文件名称及类型 | 文件原始长度 | 压缩后文件的长度 | 压缩度 | 压缩时间 |
|---|---|---|---|---|
| TEST1.TXT (文本文件) | 208 930 Bytes | 139 746 Bytes | 66.9% | 0.68 s |
| TEST2.BMP (位图文件) | 153 720 Bytes | 52 968 Bytes | 34.5% | 0.61 s |
| TEST3.EXE (可执行文件) | 238 352 Bytes | 181 321 Bytes | 76.1% | 0.71s |
| TEST4.COM (可执行文件) | 69 886 Bytes | 61 225 Bytes | 87.6% | 0.49s |
| TEST5.DAT (超大文本文件) | 271 129 422 Bytes | 157 607 533 Bytes | 58.1% | 805.71s |

通过测试可以看出,该程序能够以较高的压缩度将文件进行压缩. 但是这种静态的哈夫曼编码压缩方法存在明显不足,即需要对原文件进行两遍扫描:第一遍扫描进行原文件字符频率的统计,利用得到的频率值创建哈夫曼树并将树的有关信息保存起来,便于解压缩时使用;然后通过第二遍扫描根据前面得到的哈夫曼树对原始数据进行编码,并将编码信息储存起来。这种重复扫描的方式明显影响了压缩编码的效率,特别是在网络传输中将引起较大的延迟,破坏网络传输的同步性。另外,对于大文件的压缩,重复扫描引起的额外的磁盘访问将严重降低该算法的执行速度。

## 4　动态哈夫曼编码数据压缩算法

针对静态哈夫曼编码的上述缺点,提出了动态哈夫曼编码的算法。改进算法对数据进行编码的依据是动态变化的哈夫曼树,也就是说对第 $t+1$ 个字符的编码是根据原数据中前 $t$ 个字符得到的哈夫曼树来进行的。压缩和解压子程序具有相同的初始化树,每处理完一个字符,压缩和解压缩使用相同的算法修改哈夫曼树,所以该算法不需要为解压缩而保存哈夫曼树的信息。压缩和解压缩一个字符所需要的时间与该字符的编码长度成正比,故该过程可以实时进行。动态哈夫曼编码算法克服了传统哈夫曼编码必须对文本进行两遍扫描的缺点,压缩效率大幅度提高。尤其对一些庞大的原文件而言,压缩和解压效率可以提高一倍至三倍。

下面对这种高效的算法作详细的分析,为了便于说明,首先进行一些定义:

$n$ :字母表的长度;

$a_j$:字母表中第 $j$ 个字符;

$t$ :已经处理的原始数据中的字符的总个数;

$k$ :已经处理的数据中各不相同字符的个数;

显然,$j \geqslant 1$ , $k \leqslant n$ 。

在压缩开始之前,需要引进一个空叶结点,它的权重始终为 0。在以后的压缩和解压缩过程中,如果 $k < n$,则用它来表示 $n-k$ 个字母表中还未出现的字符。初始化的哈夫曼树中只有一个根结点和一个空叶结点。

压缩算法执行的过程是这样的:读进一个字符后,将该字符加到根结点的右分枝上,而空叶结点仍然保留在左分枝上,然后将该字符以 ASCII 码方式输出。解压子程序对哈夫曼树做同样的调整。

以后每读进一个字符 $a_{it}$,压缩子程序都执行以下的步骤:首先检查 $a_{it}$ 是否出现在编码树中,如果是,压缩子程序就以与静态哈夫曼编码中相同的方式对 $a_{it}$ 进行编码;如果 $a_{it}$ 不在编码树中,压缩子程序首先对空叶结点进行编码,然后将 $a_{it}$ 以 ASCII 码方式输出,最后在编码树中增加两个结点:在空叶结点的右分枝上加入一个新的结点,并将 $a_{it}$ 放在里面,然后在其左分枝上加入一个新的空叶结点。

解压子程序对解压缩树也做同样的调整,因为它和压缩子程序具有相同的哈夫曼树。当遍历到空叶结点时,解压子程序就从压缩数据中取出一个 ASCII 字符,然后对解压树做与压缩子程序相同的调整。

每处理一个字符,压缩和解压程序都需要修改各自的哈夫曼树,为了修改的方便,树中每个结点都具有一个序号,它是根据结点的权重由大到小排列而确定的一个递减序列。对于图 5 中的例子,已经处理了 32

个字符，即：$t=32$，$a_{it+1}=$“$b$”，此时并不是简单地将叶结点 $a_{it+1}$和它的父结点的权重加 1，因为这样做之后，该树就不再是哈夫曼树，且各个结点的序号也不再是按结点权重由大到小排列而确定的递减序列，结点 4 和结点 6 的权重为 6，而结点 5 的权重为 5。


图 5 编码示例 1

动态哈夫曼编码技术的关键就是如何将前 $t$ 个字符的哈夫曼树调整成一棵前 $t+1$ 个字符的哈夫曼树，为了解决上述问题，可以分为两步来执行：第一步把前 $t$ 个字符的哈夫曼树转换成它的另一种形式，在该树中只需在第二步中简单地把由根到叶结点 $a_{it+1}$ 路径上的所有结点权重加 1，就可以变成前 $t+1$ 个字符的哈夫曼树。其过程就是以叶结点 $a_{it+1}$ 为初始的当前结点，重复地将当前结点与具有相同权重的序号最大的结点进行交换，并使得后者的父结点成为新的当前结点，直到遇到根结点为止。以图 5 为例，结点 2 无需进行交换，所以结点 4 成为新的当前结点，结点 4 与结点

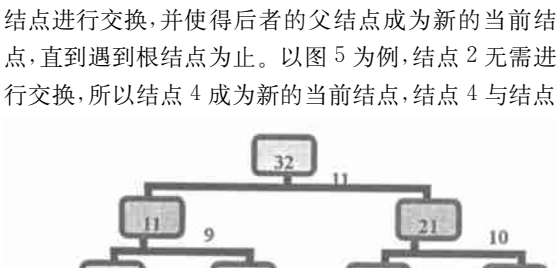


图 6 编码示例 2

5 交换后，结点 8 就成为当前结点，最后结点 8 与结点 9 进行交换，结点 11 成为当前结点，循环结束。到此为止第一步已经完成，其结果如图 6 所示，容易验证它也是前 $t$ 个字符的一种哈夫曼树形式，因为交换只是在相同权重的结点之间进行。第二步通过将由根结点到叶结点 $a_{it+1}$ 路径上的所有结点权重加 1，该树就变成了前 $t+1$ 个字符的哈夫曼树。如图 7 所示。

一个比较完整的动态哈夫曼编码实例如图 8 所示。

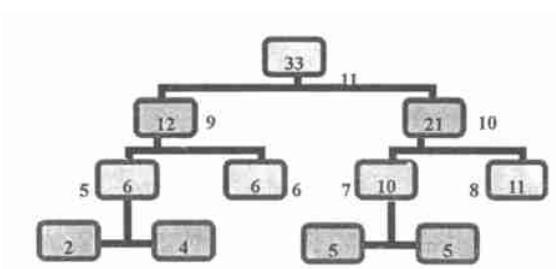


图 7 编码示例 3

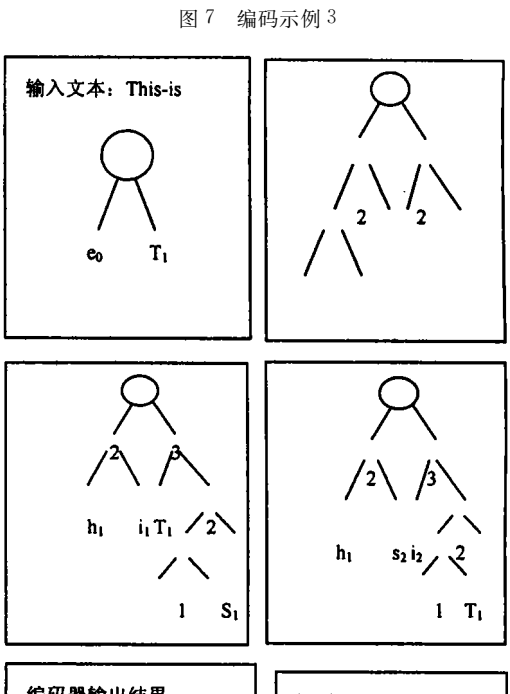


图 8 动态哈夫曼编码实例

## 5 动态哈夫曼编码压缩算法的实现

作者依据上文提出的这种高效的动态哈夫曼编码算法，实现了动态哈夫曼编码的程序设计。

为节省篇幅，本文所示的程序只列出动态哈夫曼编码压缩算法实现中比较关键的部分，一些辅助函数仅在程序前端声明。

```
struct Node          // 字符结点数据结构
{
    unsigned char Letter ;    // 字符值
    int Weight, Order ;       // 结点权重及结点编号
```

```
    Node * Parent, * LeftChild, * RightChild ;    //结点父结点及左右孩子结点
    Node * Front, * After ;    // 前后相同权重结点指针
} * Root ;    // 哈夫曼树的根结点
struct LeafNode    // 叶结点数据结构
{
    LeafNode * Next ;
    Node * CharNode ;
} * Leaf, * Weight ;
int OldHandle, NewHandle ;    // 原文件与生成文件的句柄
long All ;    // 原文件长度
char False=1 ;
void initial()
    // 根结点初始化函数,根结点各信息赋初值,初始化叶结点链,初始化权重链
void AddChar(unsigned char) ;    // 本函数将一个字符加入到哈夫曼树中
unsigned char GetBit() ;    // 本函数从文件中获取一位信息
unsigned char ReadChar() ;    // 本函数从文本中读取一个字符
void OutBit(unsigned char) ;    // 本函数将编码按位进行输出
void InsertWeight(struct Node * ) ;    // 本函数按照权重插入结点
void ProduceCode(struct Node * ) ;    // 产生字符的编码
void compression(void)    // 压缩子程序
{
    LeafNode * p ;
    …    // 创建并打开原文件和输出文件
    initial() ;    // 初始化
    …
}
void UpDate(struct Node * Temp)    // 动态更新结点信息
{
    Node * Tempa, * Tempc, * Pointer ;
    LeafNode * p, * q, * b ;
    unsigned char Letter ;
    …
}
void ProduceCode(struct Node * Pointer)    // 产生字符的编码
{
    char Code[50], Count=0, i ;    // 最大码长为 50 位
    if(Pointer! =Root)
    {
        while(Pointer! =Root {
                …    // 从叶结点到根结点按左 0 右 1 的方式编码
        }
            …    // 将编码按位输出
    }
}
```

对与上一个测试相同的原文件进行压缩测试,得到的结果如下表 3 所示:

表 3　压缩结果 2

| 文件名称及类型 | 文件原始长度 | 压缩后文件的长度 | 压缩度 | 压缩时间 |
|---|---|---|---|---|
| TEST1.TXT (文本文件) | 208 930 Bytes | 124 357 Bytes | 59.5% | 0.22s |
| TEST2.BMP (位图文件) | 153 720 Bytes | 60 335 Bytes | 39.2% | 0.19s |
| TEST3.EXE (可执行文件) | 238 352 Bytes | 177 589 Bytes | 74.5% | 0.27s |
| TEST4.COM (可执行文件) | 69 886 Bytes | 60 993 Bytes | 82.3% | 0.11s |
| TEST5.DAT (超大文本文件) | 271 129 422 Bytes | 152 218 902 Bytes | 56.1% | 242.60s |

可以非常明显地看出,两种算法的压缩度相差不大,但是改进算法的时间消耗远远少于经典的静态哈夫曼算法。

## 6　总　结

数据压缩是计算机科学中非常具有生命力的论题。一个好的数据压缩方法往往能够明显减少数据的存储空间,大大提高存储媒体的访问速度。

哈夫曼编码是数据压缩领域中最著名的编码方式之一。它通过对象频率出现的不等性,构造最优编码,达到减小文件大小的目的。目前广泛应用的许多其他高效的数据压缩算法(如算术编码,可预测编码等)也是在哈夫曼编码的基础上发展起来的。所以,研究哈夫曼编码的思想,对于深入理解数据结构、程序设计等学科中的相关课题是十分有益的。特别是对动态哈夫曼编码算法的探索,尽可能使程序稳定、快速、高效地运行,充分体现了对软件时空需求的进行优化与权衡的思想,这也是现代软件工程中十分重视的核心问题。另外,不断更新的算法设计思想也充分体现了否定之

文章编号:1005－3751(2002)05－0006－04

# 决策树学习算法 ID3 的研究

## Research on Decision Tree Learning Algorithm of ID3

杨 明,张载鸿(北京工业大学 计算机学院,北京 100022)

YANG Ming, ZHANG Zai-hong(Computer Science College of Beijing Polytechnic Univ., Beijing 100022, China)

**摘 要**:ID3 是决策树学习的核心算法,为此详细叙述了决策树表示方法和 ID3 决策树学习算法,特别说明了决策属性的选取法则。通过一个学习实例给出该算法第一次选取决策属性的详细过程,并且对该算法进行了讨论。一般情况下,ID3 算法可以找出最优决策树。

**关键词**:决策树学习;ID3 算法;机器学习;熵;信息赢取

**Abstract**: ID3 is the key algorithm of decision tree learning. For this reason, presented the expressing method of decision tree and an optimized decision tree learning algorithm of ID3, especially explained how to select the regulation of decision attribution. Through an instance, the paper showed the procedure of selecting the decision attribution in detail at the first time, as well as discussed the algorithm. In general, the optimum decision tree can be found by ID3 algorithm.

**Key words**: decision tree learning; algorithm of ID3; machine learning; entropy; information gain

**中图分类号**:TP18;TP301.6 **文献标识码**:A

## 1 决策树概念

决策树是定义布尔函数的一种方法,其输入是一组属性描述的对象,输出为 yes/no 决策。决策树代表一个假设,可以写成逻辑公式。决策树的表达能力限于命题逻辑,该对象的任一个属性的任一次测试均是一个命题[1]。在命题逻辑范围内,决策树的表达能力是完全的。一棵决策树可以代表一个决定训练例集分类的决策过程,树的每个结点对应于一个属性名或一个特定的测试,该测试在此结点根据测试的可能结果对训练例集进行划分。划分出的每个部分都对应于相应训练例集子空间的一个分类子问题,该分类子问题可以由一棵决策树来解决。因此,一棵决策树可以看作是一个对目标分类的划分和获取策略[2]。

## 2 ID3 算法

决策树学习是一种归纳学习方法,决策树学习的核心算法——ID3 算法是在所有可能的决策树空间中一种自顶向下、贪婪的搜索方法。ID3 算法的关键是确定属性表 As 中可对训练例集 Es 进行的最佳分类的属性 A,即在树的每一个节点上确定一个候选属性,它的测试对训练例的分类最有利[3]。ID3 搜索的假设空间是可能的决策树的集合,而 ID3 搜索目的是构造与训练数据一致的一棵决策树。ID3 的搜索策略是爬山法,在构造决策树时从简单到复杂,用信息赢取作为指导爬山法的评价函数。

ID3 算法通过对一个训练例集进行学习生成一棵决策树,训练例集中的每个例子都组织成属性－属性值对的形式。为了简化讨论,假设一个例子仅属于两种分类之一:正例,即符合被学习的目标概念的例子;反例,即不符合目标概念的例子。另外,假设例子的所有属性都是离散属性。对于某个训练例集 Es,如果正例的比例为 $p_+$,负例的比例为 $p_- = 1 - p_+$。则熵的公式为[4]:$\text{Entropy}(Es) = -p_+\log_2 p_+ - p_-\log_2 p_-$(这里约定 $0\log_2 0 = 0$)。

**收稿日期**:2002－01－26

**作者简介**:杨 明(1974－),男,山东莱洲人,北京工业大学计算机学院 2000 级硕士生,研究方向为网络安全,算法研究;张载鸿(1943－),男,湖北人,北京工业大学校长助理,教授。

否定的世界观与方法论。

**参考文献**:

[1] 严蔚敏,吴伟民.数据结构(第 2 版)[M].北京:清华大学出版社,1992.

[2] 美 Ian H Witten.张仲颖,曹文斌,曹永革,译.海量数据管理——文档和图像的压缩与索引[M].北京:科学出版社,1996.

[3] William Ford, William Topp. DATA STRUCTURES with C++(影印版)[M].北京:清华大学出版社·PRENTICE HALL,1997.

[4] 鲁沐浴.C 语言最新编程技巧 200 例(修订本)[M].北京:电子工业出版社,1997.