

优化哈夫曼编码数据压缩技术及程序实现*

张全伙 于洪斌 林 榆

(华侨大学计算机科学系, 泉州 362011)

摘要 讨论优化哈夫曼编码的数据压缩技术及C语言程序实现, 并与静态哈夫曼编码方法进行比较, 给出部分压缩率实例.

关键词 哈夫曼树, 编码, 数据压缩, 权值, 压缩率, C语言

分类号 TP 311.13

在计算机系统中, 字符的表示是控制信息和文字信息表示的基础. 现在国际上广泛采用的字符表示形式是ASCII码, 它是一种定长编码, 计算机对各个字符的处理是等同对待的, 也就是字符具有相同的优先级. 但是, 我们注意到在实际应用中, 经常使用的字符往往比较集中, 各个字符出现的频率相差甚大, 如果采用不定长编码, 改变各个字符的优先级, 对出现频率较高的字符采用较少的编码位数, 而对出现频率较低的字符采用较长的编码位数, 这样, 在数据的存储和信息交换时就可以大大减少系统开销. 数据压缩正是寻求减少用于存储和传输信息的位数, 又能对字符进行一一对应译码的一种技术. 哈夫曼编码是数据压缩中一种较为常用的编码技术. 本文介绍的是作者设计的一个经优化的动态哈夫曼编码数据压缩算法, 该算法在AST386/SX机器上, 在DOS V 6.2软件环境下, 用Turbo C 2.0编写的程序加以实现. 该程序可在DOS 5.0版本以上, 在IBM PC及兼容机上运行.

1 哈夫曼树及哈夫曼编码

哈夫曼树又称哈夫曼最优树^[1], 它是一类带权路径最短的树. 树的带权路径长度为树中所有带权叶结点的路径长度之和, 通常记作

$$WPL = \sum_{k=1}^n w_k l_k,$$

其中 w_k 为叶子的权, l_k 为树根到叶子的路径长度. 对于具有 n 个权值 $\{w_1, w_2, \dots, w_n\}$, 很容易构造一棵具有 n 个结点的二叉树, 每个叶结点带权 w_i , 则其中带权路径长度 WPL 最小的二叉树称为哈夫曼树. 哈夫曼树是一类严格的二叉树, 树中不存在度为1的结点. 这样, 一棵有 n 个叶结点的哈夫曼树, 共有 $2n-1$ 个结点, 可以用一个大小为 $2n-1$ 的位串表示.

哈夫曼编码有静态和动态两类. 静态哈夫曼编码是以每个字符出现的频率为权构造哈夫曼树. 字符存于叶子上, 规定左子树为0, 右子树为1, 这样每个字符都有唯一的二进制数序列

* 本文1995-02-20收到

表示. 压缩时, 对应每个 ASCII 码, 只要压入相应的哈夫曼编码; 解压时, 根据取出的哈夫曼编码, 从根结点出发, 编码为 0 时走左子树, 为 1 时走右子树, 直至叶结点. 这样就可以得到哈夫曼编码对应的 ASCII 码, 完成了译码. 由此可知, 为求编码需从叶结点出发走一条从叶子到根的路径, 而为译码需从根出发走一条从根到叶子的路径. 动态哈夫曼编码又称自适应哈夫曼编码, 它对数据压缩的依据是动态变化的哈夫曼编码树, 具体地说, 对第 $i+1$ 个字符的编码是由原始数据中前 i 个字符所建立的哈夫曼树为依据进行的.

2 优化哈夫曼编码算法及程序实现

算法的总控流程可简单描述如图 1.

2.1 首次出现字符的处理

在动态哈夫曼编码中, 第 i 个字符的编码是通过前 $i-1$ 个字符所构成的哈夫曼树来求得的, 哪些字符会出现事先无法知道. 因此, 当某个字符首次出现时, 因它不在哈夫曼树中, 无法对它进行编码. 解决此问题的一个方法是, 简单地将哈夫曼树初始化为权值为 1 的 256 个所有可能出现的 ASCII 码字符. 这样, 开始时每个字符的编码都是 8 位长, 随着字符出现次数的增加, 其编码位数将随之减少. 但是, 这在多数情况下会造成存储空间的浪费, 降低数据压缩效果. 为此, 我们引入了一个虚设的特殊字符 'KEY', 用它来代替第一次出现的字符. 当某个字符首次出现时, 就用 'KEY' 所在的结点位置对它进行编码, 并把编码写进目标文件. 接着, 把首次出现的字符的 ASCII 码值也写进目标文件. 这样, 就完成了首次出现字符的压缩处理. 随后, 要改变哈夫曼树的结构, 让 'KEY' 代替下一个首次出现的字符. 假定原先哈夫曼树的结构如图 2(a) 所示, 当有一个字符首次出现时, 就增加两个叶结点, 它们分别作为 'KEY' 所在结点的左、右孩子. 左孩子存放首次出现的字符, 并令其权值为 0, 右孩子存放 'KEY', 其权值仍为 1, 改变后的哈夫曼树如图 2(b) 所示. 同样地, 在解压过程中, 当还原出来的字符为 'KEY' 时, 说明当时压缩的这个字符是首次出现的, 接着往下连续读出 8 位 (一个字符长), 该字符就是所压的首次出现的字符. 随后, 使用与上述压缩过程中改变哈夫曼树结构的同样算法来改变哈夫曼树结构, 其流程如图 3 和图 4 所示.

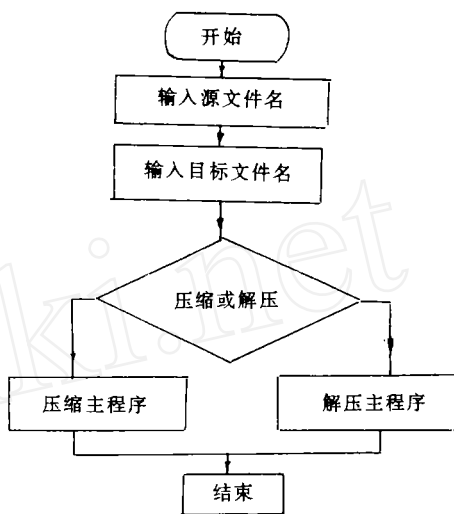


图 1 总控流程图

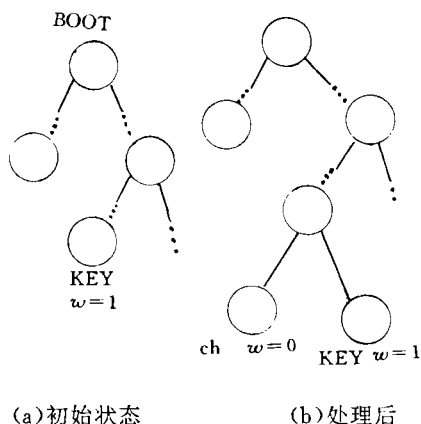


图 2 首次出现的字符处理示意图

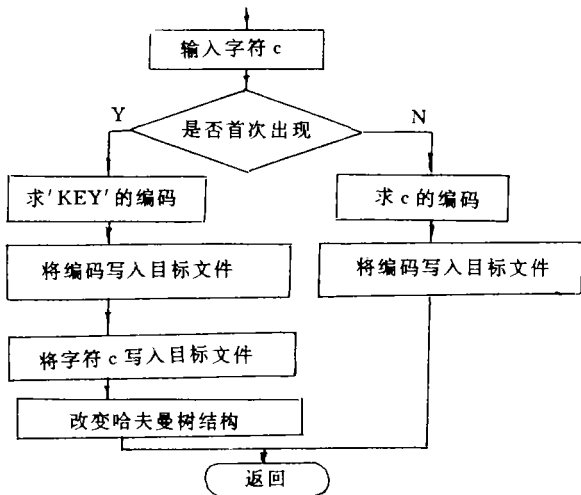


图3 压缩流程

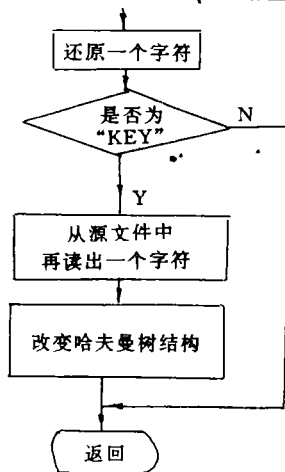


图4 解压流程

2.2 哈夫曼编码的更新

在动态哈夫曼编码中,第 $i+1$ 个字符的压缩是由第 i 个字符压缩后所构造的哈夫曼树确定的.因此,每当压缩完一个字符,要随之更新哈夫曼树.哈夫曼树的更新流程如图5所示.它主要由权值的更改和调整两个步骤实现.

欲更改某个字符的权值,只要把该字符所对应的叶结点的权值增1,接着向上移至其父结点,把其权值也增1,同样的处理一直向上直至根结点.

哈夫曼树有一个重要的特性:树中任一个结点(根结点除外)的权值都小于或等于它上面任一结点的权值,并与它的兄弟相邻.当树中某个结点权值改变时,可能破坏这一特性.此时,就必须进行调整,使之仍为一棵真正的哈夫曼树.但是,权值改变未必破坏这一特性,因此这一步骤并不总是每次要执行.当某个结点的权值改变后,只要把当前结点的权值与它前一个结点的权值进行比较,若前者大于后者,则需要调整;否则不必调整.为了减少调整的工作量,我们向上查找,找到某权值小于当前结点的最远的一个结点,它就是所求的被交换结点,把当前结点与所求结点连同它们的左右子树(如果有的话)进行交换,就得到一棵正确有序的哈夫曼树.

2.3 哈夫曼树的初始化

哈夫曼树初始化为两个值^[2]: 'KEY' 和 'ENDF', 并指定 'KEY' = 257, 'ENDF' = 256, 权值均为1. 其中 'KEY' 用来代替首次出现的字符, 'ENDF' 为文件结束标志. 根结点权值为2, 其左孩子指向 'ENDF', 右孩子指向 'KEY'. 结构如图6所示. 随着树的生长, 'KEY' 和 'ENDF' 将降到树的最远的枝叉上, 具有是长的代码.

2.4 压缩主程序

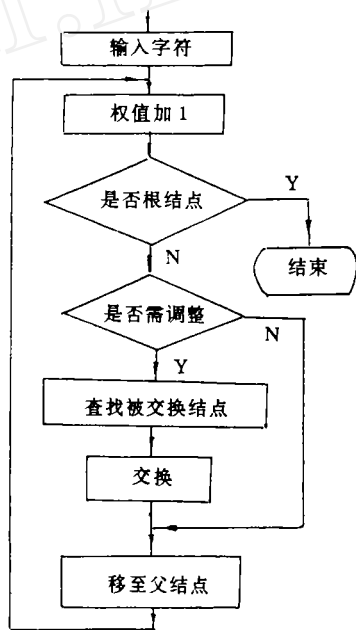


图5 哈夫曼编码树更新流程

压缩主程序描述如下

```
Inittree(tree);
while ((c=getc(infile))!=EOF){
    compres(outfile,tree,c);
    updatetree(tree,c);
}
compres(outfile, tree,ENDF);
```

树经初始化后,程序处于等待编码字符和更新模型的循环中。

当没有更多的编码字符输入时,它就编码流结束符。

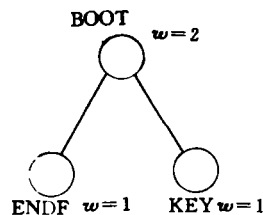


图6 初始化形态

2.5 解压主程序

树一旦初始化,它通过 update 例行程序读入符号,之后将它们写到输出文件中. 每个符号译码后,写到输出文件,并更新模型. 解压主程序如下

```
Inittree(tree);
while ((c=expant(infile,tree))!=ENDF){
    putc(outfile,c);
    update(tree,c);
}
```

2.7 数据结构

```
struct tree{
    init index[255];
    int next;
    struct node{
        unsigned long weight;
        int parent;
        int tag;
        int child;
    }nodes[515]
}tree;
```

其中 index[255]指明每个字符(0~255)在哈夫曼树中叶子结点的位置,它被初始化为-1; next 指明首次出现字符插入哈夫曼树中的位置;weight 指明每个结点的权值;parent 指明该结点的父结点位置;tag 指明该结点是否叶结点,若是,则置 tag=0,否则置 tag=1;child 指明该结点是叶结点,则叶子上存放字符的值;否则指明该结点左孩子的位置,其右孩子的位置是 child+1.

3 比较与压缩率实例

下面我们对静态哈夫曼编码和动态哈夫曼编码方法稍作比较. 静态哈夫曼编码的缺点在于需对原始数据进行两遍扫描. 第一遍扫描统计字符出现频率并建树,第二遍扫描根据所建哈夫曼树进行编码. 由此,在压缩时,将会降低压缩速度. 同时,为保存哈夫曼树以供解压时用,也将浪费一部分存储空间. 经验证明,由于静态建树,其压缩率也有所下降.

如前所述,动态哈夫曼编码对数据的压缩是依据动态变化的哈夫曼编码树,亦即对第 $i+1$ 个字符的编码是由原始数据中前 i 个字符所建立的哈夫曼树确定的. 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,压缩和解压使用相同的算法更新哈夫曼树,不必为解压而保存哈夫曼树的有关信息,从而提高了数据压缩率. 而且,由于只要一遍扫描就可完成压

缩和解压处理,大大提高了压缩速度.但是,由于解压时采用与压缩时相同方法建树,增加了解压时间,从而降低了还原速度.而静态哈夫曼编码由于对哈夫曼树进行保存,还原时不必重新建树,节省了还原时间.我们应用所设计的压缩程序对多种类型的文件进行压缩并就压缩率加以比较,从而发现此压缩程序对文本文件的压缩率较高,对可执行文件等非文本文件的压缩率相对较低.而且压缩率与文件长度有关,文件较长时其压缩率也随之下降.这正是动态哈夫曼编码数据压缩技术相对于其它压缩技术的一个缺点.下面给出部分文件压缩率见附表.

附表 部分文件压缩率

文件名	源文件大小	目标文件大小	压缩率
AHUF.C	12 906	7 715	41%
Stdio.h	6 890	4 165	40%
Graphics.h	12 289	8 132	34%
AHUF.EXE	22 256	17 603	21%
NDD.EXE	393 410	346 754	12%
TC.EXE	290 249	255 393	13%
DOSKEY.COM	5 861	5 308	10%
ATT.BGI	6 269	5 682	10%
IBM8514.BGI	6 665	4 595	32%
CL.LIB	110 973	88 813	20%
Graphics.LIB	29 247	24 174	18%

参 考 文 献

- 1 严蔚敏,吴伟民.数据结构.北京:清华大学出版社,1987.146~153
- 2 侯 阳.数据压缩技术及C语言实例.北京:学苑出版社,1994.53~90

Optimization of Hoffman Coding Data Compress Technique and Implementation of Its Program

Zhang Quanhua Yu Hongbin Lin Yu

(Dept. of Computer Science, Huaqiao Univ., 362011, Quanzhou)

Abstract A discussion is devoted to the optimization of Hoffman coding data compress technique and implementation of its program in C language. The optimized Hoffman code is compared with static one; and some examples of compressibility are given.

Keywords Hoffman tree, code, data compress, weight, compressibility, C language