散列高阶字典编码及其实现*

谭兆信

(中山大学计算机科学系 广州 510275)

摘要 该文引入了散列高阶字典的概念,提出了一个使用散列高阶字典实现数据压缩的模型.实验数据表明,该模型比传统的数据压缩字典方法具有更高的数据压缩能力.

关键词 数据压缩,图象编码,字典方法,图象处理,散列高阶字典.

中图法分类号 TP391

近年来,多媒体计算机技术和信息高速公路的研究迅速发展,探求更有效的数据压缩理论、方法和技术成为许多学者研究的目标.在无损的(可逆的)数据压缩技术中,Huffman 编码仍然被采用.此外,Written[1]等提出的算术编码方法使统计方法得到提高.在 JPEG(joint photographic coding experts group)[2]的扩展系统中,建议使用算术编码代替 Huffman 编码.为了得到更高的压缩比,人们提出高阶统计编码方法.高阶统计编码这一新颖的思想,无疑是原有的(0 阶的)统计编码的一大进步.正如一些学者指出[3],高阶的统计编码为数据压缩这一领域做了开创性的工作.

除统计方法外,数据压缩的字典方法也获得了显著的成功.现代字典压缩方法起源于 Ziv 和 Lempel 的论文[4],其中的算法称为 LZ77.经过较长时间的改进,LZ77 后来发展成 LZSS 算法.字典压缩的另一著名的算法是 LZ78[5],随后,它发展成 LZW 算法.近年来还提出了许多成功的字典压缩算法,例如,字典森林方法、Fisher 最近提出的 FI 方法和功能字典方法[6,7]等.

本文首先讨论了高阶字典的概念.按照这一概念,原有的 LZSS,LZW 等字典方法可以统称为 0 阶字典方法;而使用 1 阶、2 阶... 等字典的数据压缩方法,则称为高阶字典方法.使用 1 阶字典,可以提高数据文件的压缩比.然而,要使用 2 阶或更高阶字典,则遇到了一个困难,这是因为 2 阶或更高阶字典要占用庞大的存储空间.

在高阶字典的基础上,本文进一步引入了散列字典的概念.散列字典的概念解决了二阶或更高阶字典占用庞大的存储空间的问题.它使得使用 2 阶、3 阶等更高阶的字典进行数据压缩成为可能.

以散列字典的概念为基础,本文介绍了使用 1 阶、2 阶散列字典进行数据压缩的一个具体方法.实验数据表明,这一方法有效地提高了传统的数据压缩的字典方法的压缩能力.

1 高阶字典

本文把要压缩的数据文件称为源文件,用 *sfile* 表示.为叙述方便,有时把 *sfile* 表示为 $s[m](m=1,2,\ldots)$.用 $s(i,j)$ 表示串 $s[i],s[i+1],\ldots,s[j]$.s 的每个元素是 8 bits 的数据,称为字符或象素.压缩算法输出文件称为编码文件,用 *codefile* 表示.输出的每个码都表示源文件中的一个串.

定义 1. 对源文件 *sfile* 的一个串 $s(i,j)$,若其编码 *code* 与串 $s(i-k,i-1)(k\geqslant 1)$ 有关,则称 *code* 为串 $s(i,j)$ 的以 $s(i-k,i-1)$ 为上下文的 k 阶编码;否则,称 *code* 为串 $s(i,j)$ 的 0 阶编码.k 阶编码($k\geqslant 1$)统称为高阶编码,也称为上下文有关编码;0 阶编码也称为低阶编码或上下文无关编码.

定义 2. 包括串的高阶编码信息的字典称为高阶字典;使用高阶字典进行数据压缩的方法称为高阶字典压缩方法.

为了叙述方便,本文只讨论 1 阶和 2 阶的高阶字典压缩,并且,所讨论的压缩编码是动态的无损编码.

在高阶字典方法中,压缩程序对源文件的某些串输出其高阶码;对另一些串可能输出其低阶码.我们用一个称为 *table* 的表存放串的 0 阶、1 阶、2 阶码,*table* 也称为字典.*table* 的结构为

---

* 本文研究得到广东省自然科学基金资助.作者谭兆信,1941 年生,副教授,主要研究领域为模式识别,图象处理,人工智能.

本文通讯联系人:谭兆信,广州 510275,中山大学计算机科学系

本文 1997-01-03 收到原稿,1997-07-18 收到修改稿

```
struct node
    {unsigned nch;      /* 结点表示的字符 */
     int  child;        /* 结点的第1个儿子的下标 */
     int  brother;      /* 结点的下一个兄弟的下标 */
     int  code0;        /* 结点的0阶码 */
     int  code1;        /* 结点的1阶码 */
     int  code2;        /* 结点的2阶码 */
     int  np;           /* 串的首字符在sfile的下标 */
    } table[tablelen];
```

table 的元素也称为结点，*table* 的结点构成256棵树. 除了单结点的树外，树的每个叶结点的深度均为 *l*max（*l*max 为给定常数，例如12）. 每个结点的 *nch* 字段表示一个字符，树中的一条路径表示了一个串. 深度为 *k* 的结点 *node* 表示从根到 *node* 的路径对应的长为 *k* 的串，我们以 $code_0$ 表示该串的0阶码；而 *node* 的 $code_1$ 则表示以根的 *nch* 字段为上下文的长为 $k-1$ 的串的1阶码，$code_2$ 的含义在下节叙述. 在同一棵树中，祖先结点与后代结点的 $code_0$ 可能相同，不同树的结点，其 $code_0$ 不能相同；而不同树的结点，其 $code_1$ 可以相同，也即是同一个1阶码 $code_1$，由于其上下文不同，而可能代表256种不同的串. 因而，一个1阶码所包含的信息远远大于一个0阶码包含的信息，这就是高阶字典编码具有较强效能的原因.

2　散列高阶字典

在编码和解码算法中，我们把源文件中长度为 *l*max 的串存入 *table* 中. 因而，*table* 中存储了长度不超过 *l*max 的串的0阶码（用结点的 $code_0$ 表示），也存储了长度不超过 *l*max－1 的串的1阶码（用结点的 $code_1$ 字段表示）.

但是，当我们考虑2阶、3阶或更高阶的码时，就会遇到困难. 由于2阶码的上下文有两个字符，因此，要在 *table* 中存储源文件中长度不超过 *l*max－1 的串的2阶码，就需要256×256棵树；存储长度不超过 *l*max－1 的串的3阶码就要 256^3 棵树. 考虑到存储空间的限制，这几乎是不可能的. 为此，我们引入散列高阶字典的概念.

首先，我们定义 *m* 阶散列函数.

定义 3. *m* 阶散列函数 $hf_m(c_1,c_2,\ldots,c_m)$ 是将 *m* 个字符 $c_1,c_2,\ldots,c_m$ 映射为1个字符的函数（其中 $m=1,2,\ldots$）.

在选择计算 *m* 阶散列函数 hf_m 的具体计算公式时，我们的选择标准与软件理论常用的散列函数的选择标准相同，都是计算量要少，而函数值要尽可能均匀分布于它所能取的值域. 而对于 $m=1$，为简单起见，我们令 $hf_1(c_1)=c_1$.

定义 4. 设 $c_{-m+2}c_{-m+3}\ldots c_0c_1c_2\ldots c_k$ 是字符串，令 $c=hf_m(c_{-m+2},c_{-m+3},\ldots,c_0,c_1)$，我们称 c 为串 $c_2c_3\ldots c_k$ 的 *m* 阶散列上下文.

对于 $m=2$，若 $c_0c_1c_2\ldots c_k$ 为字符串，则 $hf_2(c_0,c_1)$ 是串 $c_2c_3\ldots c_k$ 的2阶散列上下文.

定义 5. 对于源文件 *sfile* 的一个串 $s(i,j)$，若其编码 *code* 与该串的 *m* 阶散列上下文有关，则 *code* 为串 $s(i,j)$ 的散列 *m* 阶编码.

定义 6. 包括串的散列高阶编码信息的字典称为散列高阶字典.

由于串的散列上下文只有256种可能的值，因此，256棵深度为 *l*max 的树就可以存储长度不超过 *l*max－1 的有各种散列上下文值的串的散列高阶编码.

下面讨论一个利用散列二阶字典实现数据压缩的具体方法. 我们采用的散列字典的结点的结构，如第1节所述. 设 *node* 是 *table* 中的一个结点，从根到 *node* 的路径的各个结点的 *nch* 字段的字符组成的串为 $c_1c_2\ldots c_k$，则结点 *node* 的 $code_2$ 字段是串 $c_2c_3\ldots c_k$ 的以 c_1 为2阶散列上下文的2阶散列编码.

3　散列高阶字典编码算法

由于篇幅所限，本文叙述的算法是概括性的. 为叙述方便，算法使用C语言的语句和自然语言来描述，而并不给出完整的、规范的C语言程度序.

如前所述，我们用 $s(i,j)$ 表示源文件 *sfile* 中由第 $i\sim j$ 个字符组成的串；而用 $s[i]$ 表示 *sfile* 的第 i 个字符.

设串 $s(1,sp-1)$ 已经编码，现在要对 $s[sp]\ldots$ 编码. 我们用 $bufstr(1,lmax+2)$ 存放 $s(sp-2,sp+lmax-1)$. 压缩算法使用过程 *match*(*hc*,*i*,*node*,*k*). 设 *rootindex*=*hc*，*root*=*table*[*rootindex*]；该过程在以 *root* 为根的树中往下搜索，求出与串 $bufstr(i,lmax+i-2)$ 匹配的最长路径，设最深匹配结点为 *node*，而 *node* 的深度为 *k*. 压缩算法分别以 $hc_0=bufstr[3],i=4;hc_1=bufstr[2],i=3;hc_2=hf_2(bufstr[1],bufstr[2]),i=3$ 调用 *match*；设匹配结点的深度分别为 k_0、

k_1,k_2;最深匹配结点对应的0阶、1阶和2阶码分别为 $code_0,code_1,code_2$,令 $bt_0=k_0,bt_1=k_1-1,bt_2=k_2-1$. 设 bt_0,bt_1,bt_2 的最大值为 $match\_l$(若 $bt_2=bt_1$,则优先选取 bt_2;若 $bt_1=bt_0$,则优先选取 bt_1),对应的码为 $code$,压缩程序输出码 $code.code$ 就代表了长度为 $match\_l$ 的串 $bufstr(3,match\_l+2)$,也即是源文件中的串 $s(sp,sp+match\_l-1)$. 压缩程序将串 $s(sp+i,sp+i+lmax-1)(i=0,1,\ldots,match_l-1)$插入相应的树中,也即是使树中有一条长度为 $lmax$ 的路径表示串 $s(sp+i,sp+i+lmax-1)(i=0,1,\ldots,match\_l-1)$. 令 $sp=sp+match\_l$. 然后又调用 match,如此循环,直至源文件结束.

现在,让我们来讨论输出码 $code$ 的各种可能的取值. 用 L_0 表示输出码的长度(例如 $L_0=12$),给定已知常数 h,$lmax$(例如 $h=2,lmax=12$);令 $max\_code=(1\ll L_0)-1$;输出的码 $code$ 满足 $:0\leqslant code\leqslant max\_code$. 整数区间[0,$max\_code$]称为编码区间. 把编码区间分为5个分段 $seg_i(i=0,1,2,3,4)$. 各分段的上下限分别为 $lb_i,ub_i(i=0,1,2,3,4)$. 例如,$lb_0=0,ub_0=255,lb_1=256,ub_1=1\ 023,lb_2=1\ 024,ub_2=2\ 047,lb_3=2\ 048,ub_3=2\ 047+(lmax-1)\times(lmax-h+1),lb_4=ub_3-1,ub_4=max\_code$(设 $L_0=12$,从而 $max\_code=4\ 095$). 对某些分段的编码,还输出附加的信息. 各编码分段的附加信息及其意义如下.

编码分段	附加信息	意义
seg_0	无	单字符码
seg_1	无	1阶码,代表长为 l 的串($h\leqslant l<lmax$)
seg_2	无	2阶码,代表长为 l 的串($h\leqslant l<lmax$)
seg_3	无	不确定串编码
seg_4	4 bits	0阶码,代表长为 l 的串($h\leqslant l<lmax$)

对于0阶码,由于同一路径上可能有多个结点的 $code_0$ 相同,因而用4 bits的附加信息表示匹配路径的长度 l.

在算法中,使用变量 $nextc_0$ 和数组 $nextc_1[256],nextc_2[256]$实现对结点指定0阶、1阶和2阶码. 其初值为 $nextc_0=lb_4,nextc_1[i]=lb_1,nextc_2[i]=lb_2(i=0,1,\ldots,255)$.

现在我们来讨论不确定串问题. 如前述,在编码阶段,每当将字符 $s[i]$编码,就将串 $s(i,i+lmax-1)$插入 $table$ 中,但在解码阶段,当将一个字符 $s[i]$解码后,才将串 $s(i-lmax+1,i)$插入 $table$ 中. 因此,对 $s[i]\ldots$进行解码时,比对 $s[i]\ldots$编码时,$table$ 中的路径要少. 为了解决这个问题,我们使用不确定串编码段 seg_3. 设 $ub_3=lb_3-1+(lmax-1)\times(lmax-h+1)$,当解码 $code$ 时,若 $lb_3\leqslant code\leqslant ub_3$,则令 $offset=(code-lb_3)DIV\ (lmax-h+1)+1,len=(code-lb_3)MOD\ (lmax-h+1)+h$;则 $code$ 代表的串是从输出文件当前下标回退 $offset$ 个字符的长为 len 的串.

压缩程序除使用上述的过程 match 外,还使用过程 insert. 令 $rootindex=bufstr[3],root=table[rootindex]$,过程 insert 把串 $bufstr(3,lmax+2)$插入以 $root$ 为根的树中,也即是使树中有表示串 $bufstr(3,lmax+2)$的路径. 该路径可能有一部分结点是树中原有的,另一部分则是新加入树中的,设深度 $j(j<k)$的结点是原有的,其余是新加入的. 则对于深度为 $j(j<k)$的结点只修改其np字段为sp;而对新加入的结点 $node$,则令 $node.brother=-1,node.code=nextc_0,node.code_1=nextc_1[rootindex]+j-k,node.code_2=nextc_2[rootindex]+j-k,node.np=sp$. 这些同一路径上新加入的结点的 $code_0$ 均相同.

算法1.　散列高阶字典压缩算法HHDC(hash high order dictionary coding)

step1. 将 $table$ 初始化.
step2. 将 $h,lmax,L_0,lb_0,\ldots,ub_4$ 等常数输出到 $codefile$.
step3. 从 $sfile$ 输入 $lmax-2$ 个字符到 $bufstr(1,lmax+2)$
{若输入时文件 $sfile$ 结束,则输出 $bufstr$ 中的字符,算法结束}. 令 $sp=1$.
step4. 输出单字符码 $bufstr[1]$和 $bufstr[2]$到 $codefile$,令 $match\_l=2$.
step5. 将串插入 $table$ 中,即

```
for(i=1;i<=match_l;i++)
    {if sfile 结束
      {输出 bufstr(3,match_l+lmax)的各个字符;
       算法结束;
      }
    else
        insert;   /* 将 bufstr(3,lmax+2)插入以 bufstr[3]为根的树中 */
    从 sfile 读入一个字符 ch;
    bufstr(1,lmax+2)=bufstr(2,lmax+2)+ch;sp++;
    }
```

step6. 求匹配串及相应的编码.
step7. 输出 $code$,若 $lb_4<=code<=ub_4$,则输出 $match\_l$;转到 step5.

4 散列高阶字典解码算法

解码算法也使用 *table* 表，*table* 的结点也构成 256 棵树，树的结构与编码时相同，但 child 字段则改为 father 字段. 设已经将串 $s(1,sp)$ 解码输出，则串 $s(i,i+lmax-1)$ $(1\leqslant i\leqslant sp-lmax+1)$ 在树中有对应的路径. 在解码阶段，我们不必将串在树中查找匹配路径. 但是，当将 *code* 解码时，若 *code* 属于 seg_1，seg_2 或 seg_4，则要由 *code* 找出它表示的结点，从而找出由根到该结点的路径和得到它表示的串，下面我们来讨论这个问题.

在解码时，我们要使用 3 个数组 A_0、A_1 和 A_2. A_0 是长度为 ub_3-lb_3+1 的整数数组，若 *code* 是 0 阶码，令 $i=A_0[code-lb_4]$，则 *table*[*i*] 就是 $code_0$ 字段为 *code* 的最靠近根的结点，在调用过程 insert 将串插入 *table* 时，我们将新加入树的叶结点的 $code_0$ 字段赋值为 $nextc_0$，而 $A_0[nextc_0]$ 赋值为相应结点的下标值. 另外，A_1 的结构为

```
struct node
        {int code1;
         int llink;
         int rlink;
        }A1[tablelen];
```

A_1 的长度与 *table* 相同，我们使用常用的平衡二叉树(Balanced Binary Tree)的方法，实现对结点的插入和查找.

A_2 的结构与 A_1 相似，当 *code* 是二阶码时，对 *code* 的解码方法与 1 阶码完全类似.

当 *code* 属于 seg_0 或 seg_3 时，解码方法是显然的.

5 实验结果

为了分析 HHDC 的压缩能力，我们选择了字典方法中最有代表性的、使用最普遍的 LZSS 和 LZW 算法与 HHDC 进行比较. LZSS 算法为当今流行的 LH 系列文件压缩软件所采用，而 LZW 算法则为 UNIX COMPRESS 程序所采用. 我们用 5 个图象文件、5 个正文文件和 5 个可执行文件进行测试，下表列出压缩的结果.

表 1　3 种压缩算法的比较

	图象文件	正文文件	可执行文件
HHDC 算法(*ratio*(%))	44.02	68.22	51.52
LZSS 算法(*ratio*(%))	36.88	65.74	44.80
LZW13E 算法(*ratio*(%))	41.08	57.30	39.82
源文件平均长度(byte)	142 697	12 541	26 145

其中 HHDC 的常数的取值如第 3 节所述，源文件长度单位为 byte，而 *ratio* 为

$$ratio=(\text{源文件长度}-\text{编码文件长度})/\text{源文件长度}.$$

HHDC 压缩程序运行时需要较多的存储空间，它使用了约 500k 的常规内存. 另外，其运行速度也比 LZSS 和 LZW 要慢. 它是一种无损的压缩方法，也即是对各种数据文件(包括图象文件)的压缩是无失真的.

6 结束语

上述实验数据表明，散列高阶字典编码是一种有较强压缩能力的方法. HHDC 算法的主要时间花费是对二叉树的查找和插入. 幸好，人们对二叉树的查找和插入已有深入的研究，已有成熟、高效的算法. 如果采用并行处理技术，并进一步采用 3 阶、4 阶或更高阶的码，散列高阶字典编码的优点将会进一步发挥出来. 探讨并行处理与散列高阶字典编码相结合的模型是我们今后的任务.

参考文献

1 Written I H, Neal R M, Cleary J G. Arithmetic coding for data compression. Communications of the ACM, 1987, 30(6): 520～540

2 Wallace G K. The JPEG still picture compression standard. Communications of the ACM, 1991, 34(4): 30～44

3 钱国祥等. 数据压缩技术经典. 北京：学苑出版社，1994
(Qian Guo-liang *et al*. Classics of Data Compression Techniques. Beijing: Xueyuan Publishing House, 1994)

4 Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 1977, IT-23(3): 337～343

5 Ziv J, Lempel A. Compression of individual sequences via variable rate coding. IEEE Transactions on Information Theory, 1978, IT-24(5): 530～536

6 Fisher P S, Cleopas A. Text Compression Using FI Sequences. In: Proceedings of the 9th Computing in Aerospace. Oct. 1993
7 谭兆信. 数据压缩的功能字典方法. 中山大学学报, 1996, 35(4): 39～44
(Tan Zhao-xin. Function dictionary for data compression. Journal of Zhongshan University, 1996, 35(4): 39～44)

Hash High Order Dictionary Coding and Its Implementation

TAN Zhao-xin

(*Department of Computer Science Zhongshan University Guangzhou* 510275)

Abstract In this paper, the author first introduces the concept of hash high order dictionary, and then presents a data compression model using hash high order dictionary. The experiment results show that the model has more power ability to compression data.

Key words Data compression, picture coding, dictionary method, image processing, hash high order dictionary.