Автор: Witten I., Neal R., Cleary J. Перевод статьи из: Communications of the ACM, vol. 30, no. 6, pp.520-540, June 1987.

ИДЕЯ АРИФМЕТИЧЕСКОГО КОДИРОВАНИЯ.

При арифметическом кодировании текст представляется вещественными числами в интервале от 0 до 1. По мере кодирования текста, отображающий его интервал уменьшается, а количество битов для его представления возрастает. Очередные символы текста сокращают величину интервала исходя из значений их вероятностей, определяемых моделью. Более вероятные символы делают это в меньшей степени, чем менее вероятные, и, следовательно, добавляют меньше битов к результату.

Перед началом работы соответствующий тексту интервал есть [0; 1). При обработке очередного символа его ширина сужается за счет выделения этому символу части интервала. Например, применим к тексту "eaii!" алфавита { a,e,i,o,u,! } модель с постоянными вероятностями, заданными в Таблице I.

Таблица I. Пример постоянной модели для алфавита { a,e,i,o,u,! }.

Символ	Вероятность	Интервал
a	.2	[0.0; 0.2)
e	.3	[0.2; 0.5)
i	.1	[0.5; 0.6)
o	.2	[0.6; 0.8)
u	.1	[0.8; 0.9)
!	.1	[0.9; 1.0)

И кодировщику, и декодировщику известно, что в самом начале интервал есть [0; 1). После просмотра первого символа "e", кодировщик сужает интервал до [0.2; 0.5), который модель выделяет этому символу. Второй символ "a" сузит этот новый интервал до первой его пятой части, поскольку для "a" выделен фиксированный интервал [0.0; 0.2). В результате получим рабочий интервал [0.2; 0.26), т.к. предыдущий интервал имел ширину в 0.3 единицы и одна пятая от него есть 0.06. Следующему символу "i" соответствует фиксированный интервал [0.5; 0.6), что применительно к рабочему интервалу [0.2; 0.26) суживает его до интервала [0.23, 0.236). Продолжая в том же духе, имеем:

В начале		[0.0;	1.0)
После просмотра	"e"	[0.2;	0.5)
-"-"-"-	"a"	[0.2;	0.26)
-"-"-"-	"i"	[0.23;	0.236)
-"-"-"-	"i"	[0.233;	0.2336)
-"-"-"-	"!"	[0.23354;	0.2336)

Предположим, что все что декодировщик знает о тексте, это конечный интервал [0.23354; 0.2336). Он сразу же понимает, что первый закодированный символ есть "e", т.к. итоговый интервал целиком лежит в интервале, выделенном моделью этому символу согласно Таблице I. Теперь повторим действия кодировщика:

Сначала		[0.0; 1.0)
После просмотра	"e"	[0.2; 0.5)

Отсюда ясно, что второй символ - это "a", поскольку это приведет к интервалу [0.2; 0.26), который полностью вмещает итоговый интервал [0.23354; 0.2336). Продолжая работать таким же образом, декодировщик извлечет весь текст.

Декодировщику нет необходимости знать значения обеих границ итогового интервала, полученного от кодировщика. Даже единственного значения, лежащего внутри него, например 0.23355, уже достаточно. (Другие числа - 0.23354, 0.23357 или даже 0.23354321 - вполне годятся). Однако, чтобы завершить процесс, декодировщику нужно вовремя распознать конец текста. Кроме того, одно и то же число 0.0 можно представить и как "a", и как "aa", "aaa" и

т.д. Для устранения неясности мы должны обозначить завершение каждого текста специальным символом EOF, известным и кодировщику, и декодировщику. Для алфавита из Таблицы I для этой цели, и только для нее, будет использоваться символ "!". Когда декодировщик встречает этот символ, он прекращает свой процесс.

Для фиксированной модели, задаваемой моделью Таблицы I, энтропия 5-символьного текста "eaii!" будет:

```
- log 0.3 - log 0.2 - log 0.1 - log 0.1 - log 0.1 =
                  = - log 0.00006 ~ 4.22.
```

(Здесь применяем логарифм по основанию 10, т.к. вышерассмотренное кодирование выполнялось для десятичных чисел). Это объясняет, почему требуется 5 десятичных цифр для кодирования этого текста. По сути, ширина итогового интервала есть 0.2336 - 0.23354 = 0.00006, а энтропия - отрицательный десятичный логарифм этого числа. Конечно, обычно мы работаем с двоичной арифметикой, передаем двоичные числа и измеряем энтропию в битах.

Пяти десятичных цифр кажется слишком много для кодирования текста из 4-х гласных! Может быть не совсем удачно было заканчивать пример развертыванием, а не сжатием. Однако, ясно, что разные модели дают разную энтропию. Лучшая модель, постоенная на анализе отдельных символов текста "eaii!", есть следующее множество частот символов:

{ "e"(0.2), "a"(0.2), "i"(0.4), "!"(0.2) }.

Она дает энтропию, равную 2.89 в десятичной системе счисления, т.е. кодирует исходный текст числом из 3-х цифр. Однако, более сложные модели, как отмечалось ранее, дают в общем случае гораздо лучший результат.

ПРОГРАММА ДЛЯ АРИФМЕТИЧЕСКОГО КОДИРОВАНИЯ.

На Рисунке 1 показан фрагмент псевдокода, объединяющего процедуры кодирования и декодирования, излагаемые в предыдущем разделе. Символы в нем нумеруются как 1,2,3... Частотный интервал для i-го символа задается от cum_freq[i] до cum_freq[i-1]. При убывании i cum_freq[i] возрастает так, что cum_freq[0] = 1. (Причина такого "обратного" соглашения состоит в том, что cum_freq[0] будет потом содержать нормализующий множитель, который удобно хранить в начале массива). Текущий рабочий интервал задается в [low; high] и будет в самом начале равен [0; 1) и для кодировщика, и для раскодировщика.

К сожалению этот псевдокод очень упрощен, когда как на практике существует несколько факторов, осложняющих и кодирование, и декодирование.

```
/*              АЛГОРИТМ АРИФМЕТИЧЕСКОГО КОДИРОВАНИЯ               */

/* С каждым символом текста обращаться к процедуре encode_symbol() */
/*    Проверить, что "завершающий" символ закодирован последним    */
/*        Вывести полученное значение интервала [low; high)        */

   encode_symbol(symbol,cum_freq)
     range = high - low
     high  = low + range*cum_freq[symbol-1]
     low   = low + range*cum_freq[symbol]

/*            АЛГОРИТМ АРИФМЕТИЧЕСКОГО ДЕКОДИРОВАНИЯ               */

/*             Value - это поступившее на вход число               */
/*   Обращение к процедуре decode_symbol() пока она не возвратит   */
/*                    "завершающий" символ                         */

   decode_symbol(cum_freq)
     поиск такого символа, что
     cum_freq[symbol] <= (value - low)/(high - low) < cum_freq[symbol-1]
/*    Это обеспечивает размещение value внутри нового интервала    */
/*      [low; high), что отражено в оставшейся части программы     */
```

```
    range = high - low
    high  = low + range*cum_freq[symbol-1]
    low   = low + range*cum_freq[symbol]
  return symbol
```

Рисунок 1. Псевдокод арифметического кодирования и декодирования.

Приращаемые передача и получение информации. Описанный алгоритм кодирования ничего не передает до полного завершения кодирования всего текста, также и декодировщик не начинает процесс, пока не получит сжатый текст полностью. Для большинства случаев необходим постепенный режим выполнения.

Желательное использование целочисленной арифметики. Требуемая для представления интервала [low; high) точность возрастает вместе с длиной текста. Постепенное выполнение помогает преодолеть эту проблему, требуя при этом внимательного учета возможностей переполнения и отрицательного переполнения.

Эффективная реализация модели. Реализация модели должна минимизировать время определения следующего символа алгоритмом декодирования. Кроме того, адаптивные модели должны также минимизировать время, требуемое для поддержания накапливаемых частот.

Программа 1 содержит рабочий код процедур арифметического кодирования и декодирования. Он значительно более детальный чем псевдокод на Рисунке 1. Реализация двух различных моделей дана в Программе 2, при этом Программа 1 может использовать любую из них.

В оставшейся части раздела более подробно рассматривается Программа 1 и приводится доказательство правильности раскодирования в целочисленном исполнении, а также делается обзор ограничений на длину слов в программе.

arithmetic_coding.h

```
 1  /*        ОБЪЯВЛЕНИЯ, НЕОБХОДИМЫЕ ДЛЯ АРИФМЕТИЧЕСКОГО       */
 2  /*               КОДИРОВАНИЯ И ДЕКОДИРОВАНИЯ                */
 3
 4  /*          ИНТЕРВАЛ ЗНАЧЕНИЙ АРИФМЕТИЧЕСКОГО КОДА          */
 5
 6  #define Code_value_bits 16     /* Количество битов для кода */
 7  typedef long code_value;       /* Тип  арифметического кода */
 8
 9  #define Top_value (((long) 1 << Code_value_bits) - 1)
10  /* Максимальное значение кода */
11
12  /* УКАЗАТЕЛИ НА СЕРЕДИНУ И ЧЕТВЕРТИ ИНТЕРВАЛА ЗНАЧЕНИЙ КОДА */
13
14  #define First_qtr (Top_value/4+1)  /* Конец первой черверти */
15  #define Half      (2*First_qtr)    /* Конец первой половины */
16  #define Third_qtr (3*First_qtr)   /* Конец третьей четверти */
```

model.h

```
17  /*                   ИНТЕРФЕЙС С МОДЕЛЬЮ                    */
18
19
20  /*              МНОЖЕСТВО КОДИРУЕМЫХ СИМВОЛОВ               */
21
22  #define No_of_chars 256     /* Количество исходных символов */
23  #define EOF_symbol (No_of_chars+1)    /* Индекс конца файла */
24
25  #define No_of_symbols (No_of_chars+1) /*   Всего символов   */
```

```
26
27
28  /*     Таблицы перекодировки исходных и рабочих символов    */
29
30  int char_to_index[No_of_chars];   /* Из исходного в рабочий */
31  unsigned char index_to_char[No_of_symbols+1];   /* Наоборот */
32
33
34  /*               ТАБЛИЦА НАКОПЛЕННЫХ ЧАСТОТ                 */
35
36  #define Max_frequency 16383        /* Максимальное значение */
37                                     /* частоты = 2^14 - 1    */
38  int cum_freq[No_of_symbols+1]; /* Массив накопленных частот */
```

encode.c

```
39  /*              ГОЛОВНАЯ ПРОЦЕДУРА КОДИРОВАНИЯ              */
40
41  #include
42  #include "model.h"
43
44  main()
45  {   start_model();
46      start_outputing_bits();
47      start_encoding();
48      for (;;) {                  /* Цикл обработки символов  */
49          int ch; int symbol;
50          ch = getc(stdin);       /* Чтение исходного символа */
51          if (ch==EOF) break;     /* Выход по концу файла     */
52          symbol = char_to_index[ch]; /* Найти рабочий символ */
53          encode_symbol(symbol,cum_freq); /* Закодировать его */
54          update_model(symbol);           /* Обновить модель  */
55      }
56      encode_symbol(EOF_symbol,cum_freq); /* Кодирование EOF  */
57      done_encoding();       /* Добавление еще нескольких бит */
58      done_outputing_bits();
59      exit(0);
60  }
```

arithmetic_encode.c

```
61  /*           АЛГОРИТМ АРИФМЕТИЧЕСКОГО КОДИРОВАНИЯ           */
62
63  #include "arithmetic_coding.h"
64
65  static void bit_plus_follow();
66
67
68  /*              ТЕКУЩЕЕ СОСТОЯНИЕ КОДИРОВАНИЯ               */
69
70  static code_value low, high;  /* Края текущей области кодов */
71  static long bits_to_follow;   /* Количество  битов, выводи- */
72  /*       мых после следующего бита с обратным ему значением */
73
74
75  /*           НАЧАЛО КОДИРОВАНИЯ ПОТОКА СИМВОЛОВ             */
76
77  start_encoding()
78  {   low = 0;                /* Полный кодовый интервал     */
79      high = Top_value;
80      bits_to_follow = 0;     /* Добавлять биты пока не надо */
81  }
```

```
 82
 83
 84  /*                    КОДИРОВАНИЕ СИМВОЛА                        */
 85
 86  encode_symbol(symbol,cum_freq)
 87      int symbol;                     /* Кодируемый символ     */
 88      int cum_freq[];                 /* Накапливаемые частоты */
 89  {   long range;                     /* Ширина текущего       */
 90      range = (long)(high-low)+1;     /* кодового интервала    */
 91      high = low +                    /* Сужение интервала ко- */
 92        (range*cum_freq[symbol-1])/cum_freq[0]-1; /*  дов  до */
 93      low = low +                     /* выделенного для symbol*/
 94        (range*cum_freq[symbol])/cum_freq[0];
 95      for (;;) {                      /* Цикл по выводу битов  */
 96          if (high=Half) {      /* Если в верхней, то    */
100              bit_plus_follow(1);     /* вывод 1, а затем      */
101              low -= Half;            /* убрать известную у    */
102              high -= Half;           /* границ общую часть    */
103          }
104          else if (low>=First_qtr     /* Если текущий интервал */
105                && highFirst_qtr) bit_plus_follow(0); /* лежащую в     */
122      else bit_plus_follow(1);        /* текущем интервале     */
123  }
124
125
126  /*   ВЫВОД БИТА ВМЕСТЕ СО СЛЕДУЮЩИМИ ЗА НИМ ОБРАТНЫМИ ЕМУ   */
127
128  static void bit_plus_follow(bit)
129      int bit;
130  {   output_bit(bit);
131      while (bits_to_follow>0) {
132          output_bit(!bit);
133          bits_to_follow -= 1;
134      }
135  }
```

decode.c

```
136  /* ГОЛОВНАЯ ПРОЦЕДУРА ДЛЯ ДЕКОДИРОВАНИЯ */
137
138  #include
139  #include "model.h"
140
141  main()
142  {   start_model();
143      start_inputing_bits();
144      start_decoding();
145      for (;;) {
145          int ch; int symbol;
147          symbol = decode_symbol(cum_freq);
148          if (symbol == EOF_symbol) break;
149          ch = index_to_char(symbol);
150          putc(ch,stdout);
151          update_model(symbol);
152      }
153      exit(0);
154  }
```

arithmetic_decode.c

```
155  /*          АЛГОРИТМ АРИФМЕТИЧЕСКОГО ДЕКОДИРОВАНИЯ          */
156
```

```
#include "arithmetic_coding.h"


/*             ТЕКУЩЕЕ СОСТОЯНИЕ ДЕКОДИРОВАНИЯ              */

static code_value value;           /* Текущее значение кода */
static code_value low, high;       /* Границы текущего      */
                                   /* кодового интервала    */

/*           НАЧАЛО ДЕКОДИРОВАНИЯ ПОТОКА СИМВОЛОВ           */

start_decoding();
{   int i;
    value = 0;                     /* Ввод битов для запол- */
    for (i = 1; i<=Code_value_bits; i++) { /* нения значе-  */
        value = 2*value+input_bit();       /* ния кода      */
    }
    low = 0;                       /* В самом начале теку-  */
    high = Top_value;              /* щий рабочий интервал  */
}                                  /* равен исходному       */


/*             ДЕКОДИРОВАНИЕ СЛЕДУЮЩЕГО СИМВОЛА             */

int decode_symbol(cum_freq)
    int cum_freq[];                /* Накопленные частоты   */
{   long range;                    /* Ширина интервала      */
    int cum;                       /* Накопленная частота   */
    int symbol;                    /* Декодируемый символ   */
    range = (long)(high-low)+1;
    cum =    /* Нахождение значения накопленной частоты для */
      (((long)(value-low)+1)*cum_freq[0]-1)/range; /* value */
    for (symbol = 1; cum_freq[symbol]>cum; symbol++);
    high = low +                   /* После нахождения сим- */
      (range*cum_freq[symbol-1])/cum_freq[0]-1;     /* вола */
    low = low +
      (range*cum_freq[symbol])/cum_freq[0];
    for (;;) {                     /*Цикл отбрасывания битов*/
        if (high=Half) {      /* Расширение верхней    */
            value -= Half;         /* половины после вычи-  */
            low -= Half;           /* тание смещения Half   */
            high -= Half;
        }
        else if (low>=First_qtr    /* Расширение средней    */
              && highstdio.h>
#include "arithmetic_coding.h"


/*                      БИТОВЫЙ БУФЕР                       */

static int buffer;                 /* Сам буфер             */
static int bits_to_go;             /* Сколько битов в буфере*/
static int garbage_bits;           /* Количество битов      */
                                   /* после конца файла     */

/*              ИНИЦИАЛИЗАЦИЯ ПОБИТНОГО ВВОДА               */

start_inputing_bits()
{   bits_to_go = 0;                /* Вначале буфер пуст    */
    garbage_bits = 0;
}


/* ВВОД БИТА */
```

```
238
239  int input_bit()
240  {   int t;
241      if (bits_to_go==0) {           /* Чтение байта, если    */
242          buffer = getc(stdin);      /* буфер пуст            */
243          if (buffer==EOF) {
244              garbage_bits += 1;     /* Помещение любых битов */
245              if (garbage_bits>Code_value_bits-2) {  /* после */
246                  fprintf(stderr,"Bad input file\n"); /* кон- */
247                  exit(-1);          /* ца файла  с проверкой */
248              }                      /* на слишком большое их */
249          }                          /* количество            */
250          bits_to_go = 8;
251      }
252      t = buffer&1;                  /* Выдача очередного     */
253      buffer >>= 1;                  /* бита с правого конца  */
254      bits_to_go -= 1;               /* (дна) буфера          */
255      return t;
256  }
```

bit_output.c

```
257  /*                  ПРОЦЕДУРЫ ВЫВОДА БИТОВ                   */
258
259  #include
260
261
262  /*                      БИТОВЫЙ БУФЕР                        */
263
264  static int buffer;                 /* Биты для вывода       */
265  static int bits_to_go;             /* Количество свободных  */
266                                     /* битов в буфере        */
267
268  /*              ИНИЦИАЛИЗАЦИЯ БИТОВОГО ВЫВОДА                */
269
270  start_outputing_bits()
271  {   buffer = 0;                    /* Вначале буфер пуст    */
272      bits_to_go = 8;
273  }
274
275
276  /*                        ВЫВОД БИТА                         */
277
278  output_bit(bit)
279      int bit;
280  {   buffer >>= 1;                  /* Бит - в начало буфера */
281      if (bit) buffer |= 0x80;
282      bits_to_go -= 1;
283      if (bits_to_go==0) {
284          putc(buffer,stdout);       /* Вывод полного буфера  */
285          bits_to_go = 8;
286      }
287  }
288
289
290  /*                ВЫМЫВАНИЕ ПОСЛЕДНИХ БИТОВ                  */
291
292  done_outputing_bits()
293  {   putc(buffer>>bits_to_go,stdout);
294  }
```

fixed_model.c

```
/*              МОДЕЛЬ С ФИКСИРОВАННЫМ ИСТОЧНИКОМ             */

include "model.h"

int freq[No_of_symbols+1] = {
   0,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,124,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,

/*     !   "   #   $   %   &   '   (   )   *   +   ,   -   .   / */
1236,  1, 21,  9,  3,  1, 25, 15,  2,  2,  2,  1, 79, 19, 60,  1,

/* 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ? */
  15, 15,  8,  5,  4,  7,  5,  4,  6,  3,  2,  1,  1,  1,  1,  1,

/* @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O */
   1, 24, 15, 22, 12, 15, 10,  9, 16, 16,  8,  6, 12, 23, 13, 1,

/* P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _ */
  14,  1, 14, 28, 29,  6,  3, 11,  1,  3,  1,  1,  1,  1,  1,  3,

/* '   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o */
   1,491, 85,173,232,744,127,110,293,418,  6, 39,250,139,429,446,

/* p   q   r   s   t   u   v   w   x   y   z   {   |   }         */
 111,  5,388,375,531,152, 57, 97, 12,101,  5,  2,  1,  2,  3,  1,

   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
   1
};


/*                     ИНИЦИАЛИЗАЦИЯ МОДЕЛИ                   */

start_model()
{   int i;
    for (i = 0; i0; i--) {        /* Установка  */
        cum_freq[i-1] = cum_freq[i] + freq[i]; /* счетчиков  */
    }                               /*   накопленных частот  */
    if (cum_freq[0] > Max_frequency) abort();  /* Проверка   */
}                                   /* счетчиков по границам */


/*          ОБНОВИТЬ МОДЕЛЬ В СВЯЗИ С НОВЫМ СИМВОЛОМ          */

update_model(symbol)
    int symbol;
{                                   /* Ничего не делается    */
}
```

adaptive_model.c

```
/*              МОДЕЛЬ С НАСТРАИВАЕМЫМ ИСТОЧНИКОМ             */

include "model.h"
```

```
 4
 5  int freq[No_of_symbols+1]           /* Частоты символов      */
 6
 7
 8  /* ИНИЦИАЛИЗАЦИЯ МОДЕЛИ */
 9
10  start_model()
11  {   int i;
12      for (i = 0; iNo_of_symbols; i++) { /* Установка значений*/
17          freq[i] = 1;                /* счетчиков частот в 1  */
18          cum_freq[i] = No_of_symbol-i; /* для  всех символов  */
19      }
20      freq[0] = 0;                    /* freq[0] должен отли-  */
21  }                                   /* чаться   от  freq[1]  */
22
23
24  /*     ОБНОВЛЕНИЕ МОДЕЛИ В СООТВЕСТВИИ С НОВЫМ СИМВОЛОМ      */
25
26  update_model(symbol)
27      int symbol;                     /* Индекс нового символа */
28  {   int i;                          /* Новый индекс          */
29      if (cum_freq[0]==Max_frequency) {  /* Если счетчики час- */
30          int cum;                    /* тот  достигли  своего */
31          cum = 0;                    /* максимума             */
32          for (i = No_of_symbols; i>=0; i--) { /* Тогда  делим */
33              freq[i] = (freq[i]+1)/2;   /* их  всех  пополам, */
34              cum_freq[i] = cum;      /* не приводя к нулю     */
35              cum += freq[i];
36          }
37      }
38      for (i = symbol; freq[i]==freq[i-1]; i--);
39      if (i0) {                   /* счетчика частоты для  */
50          i -= 1;                     /* символа  и  обновить  */
51          cum_freq[i] += 1;           /* накопленные  частоты  */
52      }
53  }
```

Реализация модели.

Сама реализация обсуждается в следующем разделе, а здесь мы коснемся только интерфейса с моделью (строки 20-38). В языке Си байт представляет собой целое число от 0 до 255 (тип char). Здесь же мы представляем байт как целое число от 1 до 257 включительно (тип index), где EOF трактуется как 257-ой символ. Предпочтительно отсортировать модель в порядке убывания частот для минимизации количества выполнения цикла декодирования (строка 189). Перевод из типа char в index, и наоборот, реализуется с помощью двух таблиц - index_to_char[] и char_to_index[]. В одной из наших моделей эти таблицы формируют index простым добавлением 1 к char, но в другой выполняется более сложное перекодирование, присваивающее часто используемым символам маленькие индексы.

Вероятности представляются в модели как целочисленные счетчики частот, а накапливаемые частоты хранятся в массиве cum_freq[]. Как и в предыдущем случае, этот массив - "обратный", и счетчик общей частоты, применяемый для нормализации всех частот, размещается в cum_freq[0]. Накапливаемые частоты не должны превышать установленный в Max_frequency максимум, а реализация модели должна предотвращать переполнение соответствующим масштабированием. Необходимо также хотя бы на 1 обеспечить различие между двумя соседними значениями cum_freq[], иначе рассматриваемый символ не сможет быть передан.

Приращаемая передача и получение.

В отличие от псеводокода на рисунке 1, программа 1 представляет low и high целыми числами. Для них, и для других полезных констант, опре-

делен специальный тип данных code_value. Это - Top_value, определяющий максимально возможный code_value, First_qtr и Third_qtr, представляющие части интервала (строки 6-16). В псевдокоде текущий интервал представлен через [low; high), а в программе 1 это [low; high] - интервал, включающий в себя значение high. На самом деле более правильно, хотя и более непонятно, утверждать, что в программе 1 представляемый интервал есть [low; high + 0.1111...) по той причине, что при масштабитовании границ для увеличения точности, нули смещаются к младшим битам low, а единицы смещаются в high. Хотя можно писать программу на основе разных договоренностей, данная имеет некоторые преимущества в упрощении кода программы.

По мере сужения кодового интервала, старшие биты low и high становятся одинаковыми, и поэтому могут быть переданы немедленно, т.к. на них будущие сужения интервала все равно уже не будут влиять. Поскольку мы знаем, что low<=high, это воплотится в следующую программу:

```
for (;;) {
  if (high < Half) {
    output_bit(0);
    low = 2 * low;
    high = 2 * high + 1;
  }
  else if (low >= Half) {
     output_bit(1);
     low = 2 * (low - Half);
     high = 2 * (high - Half) + 1;
  }
  else break;
}
```

гарантирующую, что после ее завершения будет спреведливо неравенство:

```
low Half) {
       value = 2 * (value - Half) + input_bit();
       low   = 2 * (low - Half);
       high  = 2 * (high - Half) + 1;
     }
     else break;
   }
```

Доказательство правильности декодирования

Проверим верность определения процедурой decode_symbol() следующего символа. Из псевдокода на рисунке 1 видно, что decode_symbol() должна использовать value для поиска символа, сократившего при кодировании рабочий интервал так, что он продолжает включать в себя value. Строки 186-188 в decode_symbol() определяют такой символ, для которого

$$\text{cum\_freq[symbol]} \le \left\lfloor \frac{(\text{value}-\text{low}+1)*\text{cum\_freq[0]}-1}{\text{high}-\text{low}+1} \right\rfloor < \text{cum\_freq[symbol-1]},$$

где "L -" обозначает операцию взятия целой части - деление с отбрасыванием дробной части. В приложении показано, что это предполагает:

$$\text{low} + \left\lfloor \frac{(\text{high}-\text{low}+1)*\text{cum\_freq[symbol]}}{\text{cum\_freq[0]}} \right\rfloor \le \text{value} \le \text{low} + \left\lfloor \frac{(\text{high}-\text{low}+1)*\text{cum\_freq[symbol-1]}}{\text{cum\_freq[0]}} \right\rfloor,$$

таким образом, что value лежит внутри нового интервала, вычисляемого процедурой decode_symbol() в строках 190-193. Это определенно гарантирует корректность определения каждого символа операцией декодирования.

Отрицательное переполнение.

Как показано в псевдокоде, арифметическое кодирование работает при помощи масштабирования накопленных вероятностей, поставляемых моделью в интервале [low; high] для каждого передаваемого символа. Предположим, что low и high настолько близки друг к другу, что операция масштабирования приводит полученные от модели разные символы к одному целому числу, входящему в [low; high]. В этом случае дальнейшее кодирование продолжать невозможно. Следовательно, кодировщик должен следить за тем, чтобы интервал [low; high] всегда был достаточно широк. Простейшим способом для этого является обеспечение ширины интервала не меньшей Max_frequency - максимального значения суммы всех накапливаемых частот (строка 36).

Как можно сделать это условие менее строгим? Объясненная выше операция битового сдвига гарантирует, что low и high могут только тогда становиться опасно близкими, когда заключают между собой Half. Предположим, они становятся настолько близки, что

First_qtr <= low < Half <= high < Third_qtr. (*)

Тогда следующие два бита вывода будут иметь взаимообратные значения: 01 или 10. Например, если следующий бит будет нулем (т.е. high опускается ниже Half и [0; Half] становится рабочим интервалом), а следующий за ним - единицей, т.к. интервал должен располагаться выше средней точки рабочего интервала. Наоборот, если следующий бит оказался 1, то за ним будет следовать 0. Поэтому теперь интервал можно безопасно расширить вправо, если только мы запомним, что какой бы бит не был следующим, вслед за ним необходимо также передать в выходной поток его обратное значение. Т.о. строки 104-109 преобразуют [First_qtr;Third_qtr] в целый интервал, запоминая в bits_to_follow значение бита, за которым надо посылать обратный ему. Это объясняет, почему весь вывод совершается через процедуру bit_plus_follow() (строки 128-135), а не непосредственно через output_bit().

Но что делать, если после этой операции соотношение (*) остается справедливым? Рисунок 2 показывает такой случай, когда отмеченный жирной линией рабочий интервал [low; high] расширяется 3 раза подряд. Пусть очередной бит, как обозначено стрелкой, расположенной на рисунке 2a ниже средней точки первоначального интервала, оказался нулем. Тогда следующие 3 бита будут единицами, поскольку стрелка находится не просто во второй его четверти, а в верхней четверти, даже в верхней восьмой части нижней половины первоначельного интервала - вот почему расширение можно произвести 3 раза. То же самое показано на рисунке 2b для случая, когда очередной бит оказался единицей, и за ним будут следовать нули. Значит в общем случае необходимо сначала сосчитать количество расширений, а затем вслед за очередным битом послать в выходной поток найденное количество обратных ему битов (строки 106 и 131-134).

Следуя этим рекомендациям, кодировщик гарантирует, что после операций сдвига будет или

low < First_qtr < Half <= high (1a)

или

low < Half < Third_qtr <= high (1b).

Значит, пока целочисленный интервал, охватываемый накопленными частотами, помещается в ее четверть, представленную в code_value, проблема отрицательного переполнения не возникнет. Это соответствует условию:

$$\text{Max\_frequency} \le \frac{\text{Top\_value} + 1}{4} + 1,$$

которое удовлетворяет в программе 1, т.к. Max_frequency = 2^14 - 1 и Top_value = 2^16 - 1 (строки 36, 9). Нельзя без увеличения количества битов, выделяемых для code_values, использовать для представления счетчиков накопленных частот больше 14 битов.

Мы рассмотрели проблему отрицательного переполнения только относительно кодировщика, поскольку при декодировании каждого символа процесс следует за операцией кодирования, и отрицательное переполнение не произойдет, если выполняется такое же масштабирование с теми же усло-

виями.

Переполнение

Теперь рассмотрим возможность переполнения при целочисленном умножении, имеющее место в строках 91-94 и 190-193. Переполнения не произойдет, если произведение range*Max_frequency вмещается в целое слово, т.к. накопленные частоты не могут превышать Max_frequency. Range имеет наибольшее значение в Top_value + 1, поэтому максимально возможное произведение в программе 1 есть 2^16*(2^14 - 1), которое меньше 2^30. Для определения code_value (строка 7) и range (строки 89,183) использован тип long, чтобы обеспечить 32-х битовую точность арифметических вычислений.

Ограниченность реализации

Ограничения, связанные с длиной слова и вызванные возможностью переполнения, можно обобщить полагая, что счетчики частот представляются f битами, а code_values - с битами. Программа будет работать корректно при f <= c - 2 и f + c <= p, где p есть точность арифметики.

В большинстве реализаций на Си, p=31, если используются целые числа типа long, и p=32 - при unsigned long. В программе 1 f=14 и c=16. При соответствующих изменениях в объявлениях на unsigned long можно применять f=15 и c=17. На языке ассемблера c=16 является естественным выбором, поскольку он ускоряет некоторые операции сравнения и манипулирования битами (например для строк 95-113 и 194-213).

Если ограничить p 16 битами, то лучшие из возможных значений c и f есть соответственно 9 и 7, что не позволяет кодировать полный алфавит из 256 символов, поскольку каждый из них будет иметь значение счетчика не меньше единицы. С меньший алфавитом (например из 26 букв или 4-х битовых величин) справится еще можно.

Завершение

При завершении процесса кодирования необходимо послать уникальный завершающий символ (EOF-символ, строка 56), а затем послать вслед достаточное количество битов для гарантии того, что закодированная строка попадет в итоговый рабочий интервал. Т.к. процедура done_encoding() (строки 119-123) может быть уверена, что low и high ограничены либо выражением (1a), либо (1b), ему нужно только передать 01 или 10 соответственно, для удаления оставшейся неопределенности. Удобно это делать с помощью ранее рассмотренной процедуры bit_plus_follow(). Процедура input_bit() на самом деле будет читать немного больше битов, из тех, что вывела output_bit(), потому что ей нужно сохранять заполнение нижнего конца буфера. Неважно, какое значение имеют эти биты, поскольку EOF уникально определяется последними переданными битами.

МОДЕЛИ ДЛЯ АРИФМЕТИЧЕСКОГО КОДИРОВАНИЯ

Программа 1 должна работать с моделью, которая предоставляет пару перекодировочных таблиц index_to_char[] и char_to_index[], и массив накопленных частот cum_freq[]. Причем к последнему предъявляются следующие требования:

- cum_freq[i-1] >= cum_freq[i];
- никогда не делается попытка кодировать символ i, для которого cum_freq[i-1] = cum_freq[i];
- cum_freq[0] <= Max_frequency.

Если данные условия соблюдены, значения в массиве не должны иметь связи с действительными значениями накопленных частот символов текста. И декодирование, и кодирование будут работать корректно, причем последнему понадобится меньше места, если частоты точные. (Вспомним успешное кодирование "eaii!" в соответствии с моделью из Таблицы I, не отражающей, однако, подлинной частоты в тексте).

Фиксированные модели

Простейшей моделью является та, в которой частоты символов постоянны. Первая модель из программы 2 задает частоты символов, приближенные к общим для английского текста (взятым из части Свода Брауна). Накопленным частотам байтов, не появлявшимся в этом образце, даются значения, равные 1 (поэтому модель будет работать и для двоичных файлов, где есть все 256 байтов). Все частоты были нормализованы в целом до 8000. Процедура инициализации start_model() просто подсчитывает накопленную версию этих частот (строки 48-51), сначала инициализируя таблицы перекодировки (строки 44-47). Скорость выполнения будет ускорена, если эти таблицы переупорядочить так, чтобы наиболее частые символы располагались в начале массива cum_freq[]. Т.к. модель фиксированная, то процедура update_model(), вызываемая из encode.c и decode.c будет просто заглушкой.

Строгой моделью является та, где частоты символов текста в точности соответствуют предписаниям модели. Например, фиксированная модель из программы 2 близка к строгой модели для некоторого фрагмента из Свода Брауна, откуда она была взята. Однако, для того, чтобы быть истинно строгой, ее, не появлявшиеся в этом фрагменте, символы должны иметь счетчики равные нулю, а не 1 (при этом жертвуя возможностями исходных текстов, которые содержат эти символы). Кроме того, счетчики частот не должны масштабироваться к заданной накопленной частоте, как это было в программе 2. Строгая модель может быть вычислена и передана перед пересылкой текста. Клири и Уиттен показали, что при общих условиях это не даст общего лучшего сжатия по сравнению с описываемым ниже адаптивным кодированием.

Адаптивная модель

Она изменяет частоты уже найденных в тексте символов. В начале все счетчики могут быть равны, что отражает отсутствие начальных данных, но по мере просмотра каждого входного символа они изменяются, приближаясь к наблюдаемым частотам. И кодировщик, и декодировщик используют одинаковые начальные значения (например, равные счетчики) и один и тот же алгоритм обновления, что позволит их моделям всегда оставаться на одном уровне. Кодировщик получает очередной символ, кодирует его и изменяет модель. Декодировщик определяет очередной символ на основании своей текущей модели, а затем обновляет ее.

Вторая часть программы 2 демонстрирует такую адаптивную модель, рекомендуемую для использования в программе 1, поскольку на практике она превосходит фиксированную модель по эффективности сжатия. Инициализация проводится также, как для фиксированной модели, за исключением того, что все частоты устанавливаются в 0. Процедура update_model(symbol), вызывается из encode_symbol() и decode_symbol() (программа 1, строки 54 и 151) после обработки каждого символа.

Обновление модели довольно дорого по причине необходимости поддержания накопленных сумм. В программе 2 используемые счетчики частот оптимально размещены в массиве в порядке убывания своих значений, что является эффективным видом самоорганизуемого линейного поиска. Процедура update_model() сначала проверяет новую модель на предмет превышения ею ограничений по величине накопленной частоты, и если оно имеет место, то уменьшает все частоты делением на 2, заботясь при этом, чтобы счетчики не превратились в 0, и перевычисляет накопленные значения (программа 2, строки 29-37). Затем, если необходимо, update_model() переупорядочивает символы для того, чтобы разместить текущий в его правильной категории относительно частотного порядка, чередуя для отражения изменений перекодировочные таблицы. В итоге процедура увеличивает значение соответствующего счетчика частоты и приводит в порядок соответствующие накопленные частоты.

ХАРАКТЕРИСТИКА

Теперь рассмотрим показатели эффективности сжатия и времени выпол-

нения программы 1.

Эффективность сжатия

Вообще, при кодировании текста арифметическим методом, количество битов в закодированной строке равно энтропии этого текста относительно использованной для кодирования модели. Три фактора вызывают ухудшение этой характеристики:

(1) расходы на завершение текста;
(2) использование арифметики небесконечной точности;
(3) такое масштабирование счетчиков, что их сумма не превышает Max_frequency.

Как было показано, ни один из них не значителен. В порядке выделения результатов арифметического кодирования, модель будет рассматриваться как строгая (в определенном выше смысле).

Арифметическое кодирование должно досылать дополнительные биты в конец каждого текста, совершая т.о. дополнительные усилия на завершение текста. Для ликвидации неясности с последним символом процедура done_encoding() (программа 1 строки 119-123) посылает два бита. В случае, когда перед кодированием поток битов должен блокироваться в 8-битовые символы, будет необходимо закругляться к концу блока. Такое комбинирование может дополнительно потребовать 9 битов.

Затраты при использовании арифметики конечной точности проявляются в сокращении остатков при делении. Это видно при сравнении с теоретической энтропией, которая выводит частоты из счетчиков точно также масштабируемых при кодировании. Здесь затраты незначительны - порядка 10^-4 битов/символ.

Дополнительные затраты на масштабирование счетчиков отчасти больше, но все равно очень малы. Для коротких текстов (меньших 2^14 байт) их нет. Но даже с текстами в 10^5 - 10^6 байтов накладные расходы, подсчитанные экспериментально, составляют менее 0.25% от кодируемой строки.

Адаптивная модель в программе 2, при угрозе превышения общей суммой накопленных частот значение Max_frequency, уменьшает все счетчики. Это приводит к тому, что взвешивать последние события тяжелее, чем более ранние. Т.о. показатели имеют тенденцию прослеживать изменения во входной последовательности, которые могут быть очень полезными. (Мы сталкивались со случаями, когда ограничение счетчиков до 6-7 битов давало лучшие результаты, чем повышение точности арифметики). Конечно, это зависит от источника, к которому применяется модель.

Время выполнения

Программа 1 была написана скорее для ясности, чем для скорости. При выполнении ее вместе с адаптивной моделью из программы 2, потребовалось около 420 мкс на байт исходного текста на ЭВМ VAX-11/780 для кодирования и почти столько же для декодирования. Однако, легко устраняемые расходы, такие как вызовы некоторых процедур, создающие многие из них, и некоторая простая оптимизация, увеличивают скорость в 2 раза. В приведенной версии программы на языке Си были сделаны следующие изменения:

(1) процедуры input_bit(), output_bit() и bit_plus_follow() были переведены в макросы, устранившие расходы по вызову процедур;
(2) часто используемые величины были помещены в регистровые переменные;
(3) умножения не два были заменены добавлениями ("+=");
(4) индексный доступ к массиву в циклах строк 189 программы 1 и 49-52 программы 2 адаптивной модели был заменен операциями с указателями.

Это средне оптимизированная реализация на Си имела время выполнения в 214/252 мкс на входной байт, для кодирования/декодирования 100.000 байтов английского текста на VAX-11/780, как показано в Таблице II. Там же даны результаты для той же программы на Apple Macintosh и SUN-3/75. Как можно видеть, кодирование программы на Си одной и той же

длины везде осуществляется несколько дольше, исключая только лишь двоичные объектные файлы. Причина этого обсуждается далее. В тестовый набор были включены два искусственных файла, чтобы позволить читателям повторять результаты. 100000 байтный "алфавит" состоит из повторяемого 26-буквенного алфавита. "Ассиметричные показатели" содержит 10000 копий строки "aaaabaaaac". Эти примеры показывают, что файлы могут быть сжаты плотнее, чем 1 бит/символ (12092-х байтный выход равен 93736 битам). Все приведенные результаты получены с использованием адаптивной модели из программы 2.

Таблица II. Результаты кодирования и декодирования 100000 байтовых файлов.

		VAX-11/780		Macintosh 512K		SUN-3/75	
	Вывод (байты)	Код. (mkc)	Дек. (mkc)	Код. (mkc)	Дек. (mkc)	Код. (mkc)	Дек. (mkc)
Среднеоптимизированная реализация на Си							
Текстовые файлы	57718	214	262	687	881	98	121
Си-программы	62991	230	288	729	950	105	131
Объектные файлы VAX	73501	313	406	950	1334	145	190
Алфавит	59292	223	277	719	942	105	130
Ассиметричные показатели	12092	143	170	507	645	70	85
Аккуратно оптимизированная реализация на ассемблере							
Текстовые файлы	57718	104	135	194	243	46	58
Си-программы	62991	109	151	208	266	51	65
Объектные файлы VAX	73501	158	241	280	402	75	107
Алфавит	59292	105	145	204	264	51	65
Ассиметричные показатели	12092	63	81	126	160	28	36

Дальнейшее снижение в 2 раза временных затрат может быть достигнуто перепрограммированием приведенной программы на язык ассемблера. Тщательно оптимизированная версия программ 1 и 2 (адаптивная модель) была реализована для VAX и для M68000. Регистры использовались полностью, а code_value было взято размером в 16 битов, что позволило ускорить некоторые важные операции сравнения и упростить вычитание Half. Характеристики этих программ также приведены в Таблице II, чтобы дать читателям представление о типичной скорости выполнения.

Временные характеристики ассемблерной реализации на VAX-11/780 даны в Таблице III. Они были получены при использовании возможности профиля UNIXa и точны только в пределах 10%. (Этот механизм создает гистограмму значений программного счетчика прерываний часов реального времени и страдает от статистической вариантности также как и некоторые системные ошибки). "Вычисление границ" относится к начальным частям encode_symbol() и decode_symbol() (программа 1 строки 90-94 и 190-193), которые содержат операции умножения и деления. "Сдвиг битов" - это главный цикл в процедурах кодирования и декодирования (строки 95-113 и 194-213). Требующее умножения и деления вычисление cum в decode_symbol(), а также последующий цикл для определения следующего символа (строки 187-189), есть "декодирование символа". А "обновление модели" относится к адаптивной процедуре update_model() из программы 2 (строки 26-53).

Таблица III. Временные интервалы ассемблерной версии VAX-11/780.

	Время кодирования	Время декодирования

	(мкс)	(мкс)
Текстовые файлы	104	135
Вычисление границ	32	31
Сдвиг битов	39	30
Обновление модели	29	29
Декодирование символа	-	45
Остальное	4	0
Си - программа	109	151
Вычисление границ	30	28
Сдвиг битов	42	35
Обновление модели	33	36
Декодирование символа	-	51
Остальное	4	1
Объектный файл VAX	158	241
Вычисление границ	34	31
Сдвиг битов	46	40
Обновление модели	75	75
Декодирование символа	-	94
Остальное	3	1

Как и предполагалось, вычисление границ и обновление модели требуют одинакового количества времени и для кодирования и для декодирования в пределах ошибки эксперимента. Сдвиг битов осуществляется быстрее для текстовых файлов, чем для Си-программ и объектных файлов из-за лучшего его сжатия. Дополнительное время для декодирования по сравнению с кодированием возникает из-за шага "декодирование символа" - цикла в строке 189, выполняемого чаще (в среднем 9 раз, 13 раз и 35 раз соответственно). Это также влияет на время обновления модели, т.к. связано с количеством накапливающих счетчиков, значения которых необходимо увеличивать в строках 49-52 программы 2. В худшем случае, когда символы распределены одинаково, эти циклы выполняются в среднем 128 раз. Такое положение можно улучшить применяя в качестве СД для частот дерево более сложной организации, но это замедлит операции с текстовыми файлами.

Адаптивное сжатие текстов

Результаты сжатия, достигнутые программами 1 и 2 варьируются от 4.8-5.3 битов/символ для коротких английских текстов (10^3-10^4 байтов) до 4.5-4.7 битов/символ для длинных (10^5-10^6 байтов). Хотя существуют и адаптивные техники Хаффмана, они все же испытывают недостаток концептуальной простоты, свойственной арифметическому кодированию. При сравнении они оказываются более медленными. Например, Таблица IV приводит характеристики среднеоптимизированной реализации арифметического кодирования на Си с той из программ compact UNIXа, что реализует адаптивное кодирование Хаффмана с применением сходной модели. (Для длинных файлов, как те, что используются в Таблице IV, модель compact по-существу такая же, но для коротких файлов по сравнению с приведенной в программе 2 она лучше). Небрежная проверка compact показывает, что внимание к оптимизации для обоих систем сравнимо при том, что арифметическое кодирование выполняется в 2 раза быстрее. Показатели сжатия в некоторой степени лучше у арифметического кодирования для всех тестовых файлов. Различие будет заметным в случае применения более сложных моделей, предсказывающих символы с вероятностями, зависящими от определенных обстоятельств (например, следования за буквой q буквы u).

Таблица IV. Сравнение адаптивных кодирований Хаффмана и арифметического.

	Арифметическое кодирование			Кодирование Хаффмана		
	Вывод (байты)	Код. (мкс)	Дек. (мкс)	Вывод (байты)	Код. (мкс)	Дек. (мкс)
Текстовые файлы	57718	214	262	57781	550	414
Си-программы	62991	230	288	63731	596	441
Объектные файлы VAX	73501	313	406	76950	822	606
Алфавит	59292	223	277	60127	598	411
Ассиметричные показатели	12092	143	170	16257	215	132

Неадаптированное кодирование

Оно может быть выполнено арифметическим методов с помощью постоянной модели, подобной приведенной в программе 2. При этом сжатие будет лучше, чем при кодировании Хаффмана. В порядке минимизации времени выполнения, сумма частот cum_freq[0] будет выбираться равной степени двойки, чтобы операции деления при вычислении границ (программа 1, строки 91-94 и 190-193) выполнялись через сдвиги. Для реализации на ассемблере VAX-11/780 время кодирования/декодирования составило 60-90 мкс. Аккуратно написанная реализация кодирования Хаффмана с использованием таблиц просмотра для кодирования и декодирования будет выполнятся немного быстрее.

Кодирование черно-белых изображений

Применение для этих целей арифметического кодирования было рассмотрено Лангдоном и Риссаненом, получившим при этом блестящие результаты с помощью модели, использующей оценку вероятности цвета точки относительно окружающего ее некоторого шаблона. Он представляет собой совокупность из 10 точек, лежаших сверху и спереди от текущей, поэтому при сканировании растра они ей предшествуют. Это создает 1024 возможных контекста, относительно которых вероятность черного цвето у данной точки оценивается адаптивно по мере просмотра изображения. После чего каждая полярность точки кодировалась арифметическим методом в соответствии с этой вероятностью. Такой подход улучшил сжатие на 20-30% по сравнению с более ранними методами. Для увеличения скорости кодирования Лангдон и Риссанен применили приблизительный метод арифметического кодирования, который избежал операций умножения путем представления вероятностей в виде целых степеней дроби 1/2. Кодирование Хаффмана для этого случая не может быть использовано прямо, поскольку оно никогда не выполняет сжатия двухсимвольного алфавита. Другую возможность для арифметического кодирования представляет применяемый к такому алфавиту популярный метод кодирования длин тиражей (run-length method). Модель здесь приводит данные к последовательности длин серий одинаковых символов (например, изображение представляются длинами последовательностей черных точек, идущих за белыми, следующих за черными, которым предшествуют белые и т.д.). В итоге должна быть передана последовательность длин. Стандарт факсимильных аппаратов CCITT строит код Хаффмана на основе частот, с которыми черные и белые последовательности разных длин появляются в образцах документов. Фиксированное арифметическое кодирование, которое будет использовать те же частоты, будет иметь лучшие характеристики, а адаптация таких частот для каждого отдельного документа будет работать еще лучше.

Кодирование произвольно распределенных целых чисел.

Оно часто рассматривается на основе применения более сложных моделей текстов, изображений или других данных. Рассмотрим, например, ло-

кально адаптивную схему сжатия Бентли et al, где кодирование и декодирование работает с N последними разными словами. Слово, находящееся в кэш-буфере, определяется по целочисленному индексу буфера. Слово, которое в нем не находится, передаются в кэш-буфер через посылку его маркера, который следует за самими символами этого слова. Это блестящая модель для текста, в котором слова часто используются в течении некоторого короткого времени, а затем уже долго не используются. Их статья обсуждает несколько кодирований переменной длины уже для целочисленных индексов кэш-буфера. В качестве основы для кодов переменной длины арифметический метод позволяет использовать любое распределение вероятностей, включая среди множества других и приведенные здесь. Кроме того, он допускает для индексов кэш-буфера применение адаптивной модели, что желательно в случае, когда распределение доступов к кэш-буферу трудно-предсказуемо. Еще, при арифметическом кодировании ..... , предназначенные для этих индексов, могут пропорционально уменьшаться, чтобы приспособить для маркера нового слова любую желаемую вероятность.

ПРИЛОЖЕНИЕ. Доказательство декодирующего неравенства.

Полагаем:

$$\text{cum\_freq[symbol]} \le \left\lfloor \frac{(\text{value}-\text{low}+1)*\text{cum\_freq}[0]-1}{\text{high}-\text{low}+1} \right\rfloor < \text{cum\_freq}[\text{symbol}-1].$$

Другими словами:

$$\text{cum\_freq[symbol]} \le \frac{(\text{value}-\text{low}+1)*\text{cum\_freq}[0]-1}{\text{range}} + e < \text{cum\_freq}[\text{symbol}-1], \quad (1)$$

где $\text{range} = \text{high} - \text{low} + 1$, $0 \le e \le \frac{\text{range} - 1}{\text{range}}$.

(Последнее неравенство выражения (1) происходит из факта, что cum_freq[symbol-1] должно быть целым). Затем мы хотим показать, что low' <= value <= high', где low' и high' есть обновленные значения для low и high как определено ниже.

$$\text{(a) low'} \cdot \text{low} + \left\lfloor \frac{\text{range}*\text{cum\_freq[symbol]}}{\text{cum\_freq}[0]} \right\rfloor \le$$

$$\le \text{low} + \frac{\text{range}}{\text{cum\_freq}[0]} \left\lfloor \frac{(\text{value}-\text{low}+1)*\text{cum\_freq}[0]-1}{\text{range}} - e \right\rfloor$$

Из выражения (1) имеем: $\le \text{value} + 1 - \frac{1}{\text{cum\_freq}[0]}$,

Поэтому low' <= value, т.к. и value, и low', и cum_freq[0] > 0.

$$\text{(a) high'} \cdot \text{low} + \left\lfloor \frac{\text{range}*\text{cum\_freq[symbol-1]}}{\text{cum\_freq}[0]} \right\rfloor - 1 \ge$$

$$\ge \text{low} + \frac{\text{range}}{\text{cum\_freq}[0]} \left\lfloor \frac{(\text{value}-\text{low}+1)*\text{cum\_freq}[0]-1}{\text{range}} + 1 - e \right\rfloor - 1$$

Из выражения (1) имеем:

$$\ge \text{value} + \frac{\text{range}}{\text{cum\_freq}[0]} \left\lfloor \frac{1}{\text{range}} + 1 - \frac{\text{range}-1}{\text{range}} \right\rfloor = \text{value}.$$