·17·08·2000·····

Ответы на незаданные вопросы

.....

1. Что такое арифметическое кодирование
2. Чисто ari'шые эвристики
3. Примеры
4. Библиография

.....

```
· if (sym == Stats->Symbol)
· {
·   rc.Encode(0, Stats->Freq, SummFreq);
·   Stats->Freq += 8;
·   SummFreq += 8;
·   return;
· }
```

.....

– Д. Шкарин

1. Что такое арифметическое кодирование

Частотное кодирование вообще, как я его понимаю, есть попытка перенумеровать все файлы, имеющие данную таблицу частот символов и записать вместо данного файла его номер.

Для бинарного случая, как я показал в ark1.doc, существует вполне прямолинейный способ это сделать. Надо сказать, в не дошедший до ru.compress "полный комплект" этой текстовки входила программка на асме, которая упаковывала блок размером где-то в 768 битов, пользуясь длинной арифметикой. Но это непрактично.

Напомню алгоритм: существует ровно F(a,b)=(a+b)!/a!/b! последовательностей битов, имеющих таблицу частот {0:a;1:b}. Т.е., все такие последовательности можно перенумеровать числами в интервале [0..F(a,b)). Таким же образом известно, что таких последовательностей с нулевым первым битом ровно F(a-1,b), из чего следует, что можно присвоить последовательностям 0... интервал [0..F(a-1,b)), а 1... – интервал [F(a-1,b)..F(a,b)) и это не будет ничему противоречить. Очевидно, что этот процесс можно легко продолжить и получить полный номер последовательности, а не интервал. И обратить этот процесс столь же просто.

Но для прямого воплощения вышеописанного требуется длинная арифметика, применять которую мы себе позволить не можем. Поэтому эвристика1: принципиально важным свойством функции F(a,b) является лишь то, чтобы F(a,b) не было меньше суммы F(a-1,b) и F(a,b-1) – это требуется для однозначного декодирования. Таким образом, мы можем рассчитать аппроксимацию "правильной" функции, имеющую нужную разрядность мантиссы. Скажем, можно просто так и вычислять F(a,b) как сумму F(a-1,b) и F(a,b-1) с округлением вверх. Это уже позволяет применять данный алгоритм практически – точности, обеспечиваемой таблицей F(x,y), помещающейся в 64k вполне достаточно для сжатия, очень близкого к обеспечиваемого традиционной реализацией битового арифметического кодера.

Существует и альтернативный вариант реализации вышеописанного, не требующий таблиц. Дело в том, что из F(a,b) мы можем получить F(a-1,b) путем умножения на (a) и деления на (a+b). Аналогично и F(a,b-1). А чтобы не пересекались интервалы [0..F(a-1,b)) и [F(a-1,b)..F(a,b)) мы можем просто взять в них округления F(a,b)*a/(a+b) в разные стороны.

Вот. Только вот, как ни печально, последнее уже называется не ark (c) Eugene D. Shelwien, а "арифметическое кодирование" (c) IBM и черт знает кто еще :).

Ari, таким образом, – это всего лишь способ эвристической _комбинаторной_ нумерации блоков данных, имеющих одинаковые таблицы частот. Впрочем, есть и маленькое отличие/упрощение – эвристика2: если заменить первоначальный коэффициент F(a,b) на 1, то этого никто не заметит. ;-), разница в размере кода будет не всегда и в худшем случае на несколько бит).

[Btw, по поводу сравнения ari и ark см. mark#1.rk/text/notables.txt]

.....

2. Чисто ari'шые эвристики

Теперь, пожалуй, можно расписать, как приблизительно выглядит арифметический кодер. Только, пожалуй, сразу "нормальный", а не бинарный. Бинарность, вообще-то, потребовалась мне больше для упрощения. Ну, кроме того, таблицу мультиномиальных коэффициентов даже для трехсимвольного алфавита помещать абсолютно некуда :). Тем не менее, F(a1,...,aI-1,...,aN) все так же равно F(a1,...,aN)*aI/Sum(aX,x=1..N), так что:

```
// нижняя граница интервала кодового пространства, соответствующего
// уже закодированному отрезку данных
low = 0
// вместо F(a1,...,aN). Всегда можно успеть на него домножить ;).
rng = 1 ;
T = SymbolNumber
for all c = SourceChar
  L = low(c)
  R = freq(c)
  hai = low + int(rng*(L+R)/T)  // (Я _знаю_, как пишется high :)
  low = low + int(rng*L/T) + 1
  rng = hai - low
  freq(c)--, T--
next c
```

Если при взятии целой части числа дробная просто отбрасывается, то вышеприведенного вполне достаточно для обеспечения возможности однозначно определить всю последовательность символов, по очереди находя, интервалу какого символа соответствует значение low, полученное в результате кодирования всего файла. Для этого требуется, чтобы всегда существовал ровно один такой символ (x), для которого выполнялось быть неравенство

```
int(rng*low(x))+1 <= low-DecodeLow < int(rng*(low(x)+freq(x))
```

Эвристика3: поскольку разрядность границ этого интервала все равно фиксированная, то можно их сравнивать лишь с соответствующим числом старших бит low - остальные ничего не изменят.

Эвристика4: rng никогда не возрастает, только убывает. Кроме того, понятно, что если старшие биты low и hai совпадают, то в low они уже никогда не изменятся. А поскольку во все формулы они входят лишь аддитивно, то их свободно можно хранить отдельно :).

Тем не менее, не все так просто, как кажется. Ari присущ неприятный эффект, которого ark избегает за счет более свободного подбора коэффициентов (они должны лишь удовлетворять неравенству; не обязательно должна быть возможность получать их один из другого) - переносы в low. Т.е., вполне реальна такая ситуация, что low не совпадает с hai ни в одном бите, хотя точность rng уже упала ниже допустимых пределов. Для решения этой проблемы применяется

Эвристика5: (стандартная) старшие биты low можно все же отбрасывать, но наличие несовпадения все же учитывать. Фишка в том, что если сдвинуть старшие биты low в выходной поток несмотря на то, что они не совпадают с hai, то весьма вероятно, что через некоторое время возникнет ситуация, когда low+rng>maxint :). Конечно же, возникнет бит переноса, который надо будет куда-то девать. Так вот, стандартное средство для этого случая - кэшировать непрерывную последовательность единичных битов (и один ноль перед ними), выдвинутую из low непосредственно перед этим и инвертировать ее (ноль менять на единицы, а единицы - на нули) при возникновении переноса. В общем, обычный эффект сложения - перенос не идет дальше ближайшего нулевого бита. С одной стороны, запрограммировать такое кэширование не очень сложно. Но с другой, смысла в этом нет, т.к. существует

Эвристика6: (хочу знать, кто придумал; я увидел у Субботина) можно

просто так срезАть rng, чтобы перенос _не возникал_ :). Действительно, делать rng меньше можно когда нам удобно - лишь бы в декодере можно было делать это синхронно; работоспособности алгоритма это не повредит. Таким образом, если мы хотим несколько старших бит low записать в выходной поток (т.к. rng уже слишком уменьшился), то достаточно просто проследить, чтобы в них _не происходил_ перенос, пересчитав при необходимости значение rng соответствующим образом.

Эвристика7: (моя собственная :) можно совместить #5 и #6, т.к. применение #6 самой по себе заметно портит коэффициент сжатия на некоторых данных. Т.е., просто закэшировать несколько старших байт low, уже как бы "закодированных" и выполнять отсечение rng только в случае, если перенос проходит еще дальше. Надо сказать, что при достаточной точности вычислений отсечение практически не требуется. Так, мой кодер CL-A с 32 бит на low и rng и 32 бит "кэша" low таких отсечений требовал лишь около пяти штук даже на pic из Calgary Corpus, весьма этому эффекту способствующем. Последний же кодер CL-F на FPU с 64-битной точностью (64+24 на low и 64 на rng) поймать на выполнении отсечения мне пока не удалось :). А, ну и

Эвристика8: (Shindler's rangecoder) если дожидаться, пока не зафиксируются старшие _восемь_ (а не один, как в традиционном ari) бит low, то с битами кодеру работать не придется и насчет операций битового i/o думать тоже не нужно.

.....

3. Примеры

Мои кодеры написаны на ассемблере. Некоторые люди говорят, что ассемблер трудно читать :). Поэтому пусть читают субботинский же кодер из coder.hpp, содержащегося в исходниках ppmd v.F by Dmitry Shkarin. Впрочем, я его немножко переделал, для понятности :)

```
#define  TOP       (1<<24)
#define  BOT       (1<<16)

void Encode (uint cumFreq, uint freq, uint totFreq)
{
   rng /= totFreq
   low += cumFreq * rng;
   rng *= freq;
   hai  = low + rng
   // проверить, не слишком ли мал rng; если да - workaround переноса
   if (low^hai>=TOP && rng<BOT) rng = (low | (BOT-1)) - low;
   // зафиксировался байт?
   if (low^hai<TOP)
   {
     // выбрасываем старший байт, совпадающий в Low и Hai
     OutByte(low>>24);
     // перемещаем "плавающую точку"
     rng<<=8, low<<=8;
   }
}

void Decode (uint cumFreq, uint freq, uint totFreq)
{
   rng /= totFreq;
   low += cumFreq * rng;
   rng *= freq;
   if (low^hai>=TOP && rng<BOT) rng = (low | (BOT-1)) - low;
   if (low^hai<TOP)
   {
     // передвигаем отрезок кода, используемый для предсказания символов
     code = code<<8 | InByte();
     rng<<=8, low<<=8;
   }
}
```

Впрочем, эвристика9: (моя, вроде бы) при делении теряется информация; во избежание деление рекомендуется всегда выполнять _после_ всех умножений в формуле. Все виденные мною арифметические кодеры же относятся к этому правилу наплевательски, т.к. результат предварительного умножения может не поместиться в разрядную сетку используемого типа данных. Таков практический вред от применения языков программирования высокого уровня - ассемблерная команда MUL _всегда_ вычисляет результат с удвоенной разрядностью операндов. А команда DIV ей в пару выполняет деление двойного слова, содержащегося в двух регистрах, на одинарное. Но на ЯВУ, соблюдая портабельность, этим воспользоваться нельзя, увы :).

Такие дела. :)

.....

4. Библиография

1. ark1.doc
   Публиковался в RU.COMPRESS 29 июня 1998 года.

2. mark#1.rk, демонстрационная упаковка мультисимвольного Ark-кодера.
   Кому надо, у тех есть :).

3. Исходники компрессора PPMD Дмитрия Шкарина
   ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmde.rar (coder.hpp по Шиндлеру)
   ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdf.rar (coder.hpp по Субботину)

4. Исходники order0 и order1 rangecoder'ов Дмитрия Шкарина
   Личная переписка :).

.....