Д. Ватолин, А. Ратушняк,
М. Смирнов, В. Юкин

Методы сжатия данных

Раздел 1

Содержание книги:

Введение
Раздел 1. Универсальные методы сжатия
Раздел 2. Алгоритмы сжатия изображений
Раздел 3. Алгоритмы сжатия видео
Приложение 1
Приложение 2

Дата создания файла: 05.01.2003

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

РАЗДЕЛ 1. МЕТОДЫ СЖАТИЯ БЕЗ ПОТЕРЬ 25
Глава 1. Кодирование источников данных без памяти 27
Глава 2. Кодирование источников данных типа «аналоговый сигнал» 28
Линейно-предсказывающее кодирование 28
Субполосное кодирование 44
Глава 3. Словарные методы сжатия данных 57
Идея словарных методов 57
Классические алгоритмы Зива-Лемпела 60
Другие алгоритмы LZ 76
Формат Deflate 82
Пути улучшения сжатия для методов LZ 97
Архиваторы и компрессоры, использующие алгоритмы LZ 110
Вопросы для самоконтроля 111
Литература 113
Список архиваторов и компрессоров 114
Глава 4. Методы контекстного моделирования 114
Классификация стратегий моделирования 118
Контекстное моделирование 120
Алгоритмы PPM 130
Оценка вероятности ухода 144
Обновление счетчиков символов 158
Повышение точности оценок в контекстных моделях высоких порядков 160
Различные способы повышения точности предсказания 166
PPM и PPM 173*
Достоинства и недостатки PPM 174
Компрессоры и архиваторы, использующие контекстное моделирование 177
Обзор классических алгоритмов контекстного моделирования 186
Сравнение алгоритмов контекстного моделирования 191
Другие методы контекстного моделирования 192
Вопросы для самоконтроля 193
Литература 194
Список архиваторов и компрессоров 196
Глава 5. Преобразование Барроуза-Уилера 198
Введение 198
Преобразование Барроуза-Уилера 199
Методы, используемые совместно с BWT 212
Способы сжатия преобразованных с помощью BWT данных 230
Сортировка, используемая в BWT 240
Архиваторы, использующие BWT и ST 251
Заключение 260
Литература 260
Глава 6. Обобщенные методы сортирующих преобразований 263

Сортировка параллельных блоков *263*
Фрагментирование *276*
Глава 7. Предварительная обработка данных 287
Препроцессинг текстов *288*
Препроцессинг нетекстовых данных *307*
Вопросы для самоконтроля *315*
Литература *315*
Выбор метода сжатия 316
УКАЗАТЕЛЬ ТЕРМИНОВ 321

Раздел 1. Методы сжатия без потерь

В основе всех методов сжатия лежит простая идея: если представлять часто используемые элементы короткими кодами, а редко используемые — длинными кодами, то для хранения блока данных требуется меньший объем памяти, чем если бы все элементы представлялись кодами одинаковой длины. Данный факт известен давно: вспомним, например, азбуку Морзе, в которой часто используемым символам поставлены в соответствие короткие последовательности точек и тире, а редко встречающимся — длинные.

Точная связь между вероятностями и кодами установлена в теореме Шеннона о кодировании источника, которая гласит, что элемент s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами. Если при кодировании размер кодов всегда в точности получается равным $-\log_2 p(s_i)$ битам, то в этом случае длина закодированной последовательности будет минимальной для всех возможных способов кодирования. Если распределение вероятностей $F = \{p(s_i)\}$ неизменно, и вероятности появления элементов независимы, то мы можем найти среднюю длину кодов как среднее взвешенное

$$H = -\sum_i p(s_i) \cdot \log_2 p(s_i). \qquad (1a)$$

Это значение также называется энтропией распределения вероятностей F или энтропией источника в заданный момент времени.

Обычно вероятность появления элемента является условной, т.е. зависит от какого-то события. В этом случае при кодировании очередного элемента s_i распределение вероятностей F принимает одно из возможных значений F_k, то есть $F = F_k$, и, соответственно, $H = H_k$. Можно сказать, что источник находится в состоянии k, которому соответствует набор вероятностей $p_k(s_i)$ генерации всех возможных элементов s_i. Поэтому среднюю длину кодов можно вычислить по формуле

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

$$H = -\sum_k P_k \cdot H_k = -\sum_{k,i} P_k \cdot p_k(s_i)\log_2 p_k(s_i), \qquad (1б)$$

где P_k — вероятность того, что F примет k-ое значение, или, иначе, вероятность нахождения источника в состоянии k.

Итак, если нам известно распределение вероятностей элементов, генерируемых источником, то мы можем представить данные наиболее компактным образом, при этом средняя длина кодов может быть вычислена по формуле (1).

Но в подавляющем большинстве случаев истинная структура источника нам не известна, поэтому необходимо строить *модель* источника, которая позволила бы нам в каждой позиции входной последовательности оценить вероятность $p(s_i)$ появления каждого элемента s_i алфавита входной последовательности. В этом случае мы оперируем оценкой $q(s_i)$ вероятности элемента s_i.

Методы сжатия могут строить модель источника адаптивно по мере обработки потока данных или использовать фиксированную модель, созданную на основе априорных представлений о природе типовых данных, требующих сжатия.

Процесс моделирования может быть либо явным, либо скрытым. Вероятности элементов могут использоваться в методе как явным, так и неявным образом. Но всегда сжатие достигается за счет устранения **статистической избыточности** в представлении информации.

Ни один компрессор не может сжать **любой** файл. После обработки **любым** компрессором размер части файлов уменьшится, а оставшейся части — увеличится или останется неизменным. Данный факт можно доказать исходя из неравномерности кодирования, т.е. разной длины используемых кодов, но наиболее прост для понимания следующий комбинаторный аргумент.

Существует 2^n различных файлов длины n битов, где n = 0, 1, 2, … Если размер **каждого** такого файла в результате обработки уменьшается хотя бы на 1 бит, то 2^n исходным файлам будет соответствовать самое большее 2^n-1 различающихся архивных файлов. Тогда по крайней мере одному архивному файлу будет

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

соответствовать несколько различающихся исходных, и, следовательно, его декодирование без потерь информации **невозможно в принципе**.

Вышесказанное предполагает, что файл отображается в один файл, и объем данных указывается в самих данных. Если это не так, то следует учитывать не только суммарный размер архивных файлов, но и объем информации, необходимой для описания нескольких взаимосвязанных архивных файлов и/или размера исходного файла. Общность доказательства при этом сохраняется.

Поэтому невозможен «вечный» архиватор, который способен бесконечное число раз сжимать свои же архивы. «Наилучшим» архиватором является программа копирования, поскольку в этом случае мы можем быть всегда уверены в том, что объем данных в результате обработки не увеличится.

Регулярно появляющиеся заявления о создании алгоритмов сжатия, «обеспечивающих сжатие в десятки раз лучшее, чем у обычных архиваторов», являются либо ложными слухами, порожденными невежеством и погоней за сенсациями, либо рекламой аферистов. В области сжатия без потерь, т.е. собственно сжатия, такие революции невозможны. Безусловно, степень сжатия компрессорами типичных данных будет неуклонно расти, но улучшения составят в среднем десятки или даже единицы процентов, при этом каждый последующий этап эволюции будет обходиться значительно дороже предыдущего. С другой стороны, в сфере сжатия с потерями, в первую очередь компрессии видеоданных, все еще возможно многократное улучшение сжатия при сохранении субъективной полноты получаемой информации.

Глава 1. Кодирование источников данных без памяти

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Глава 2. Кодирование источников данных типа «аналоговый сигнал»

Линейно-предсказывающее кодирование

Английское название метода — Linear Prediction Coding (LPC).

Цель — сжатие потока *R*-битных элементов, в предположении, что значение каждого из них является линейной комбинацией значений *h* предыдущих элементов:

$$S_i = \sum_{j=i-1}^{j=i-h} K_j S_j ,$$

где S_i — *i*-ый R-битный элемент;

K_j — некоторые коэффициенты, в общем случае непостоянные.

Основная идея состоит в том, чтобы в формируемый поток записывать ошибки предсказаний: разности между реальными S_i и предсказанными значениями:

$$D_i = S_i - \sum_{j=i-1}^{j=i-h} K_j S_j .$$

Размер данных в результате применения LPC не изменяется. Более того, размер элементов *R* может даже увеличиваться на 1: *R*`=*R*+1 за счет добавления бита для сохранения знака разности.

Чем точнее предсказание, тем больше в преобразованной последовательности элементов с близкими к нулю значениями, тем лучше можно сжать такую последовательность.

Для сжатия результата работы метода может быть применена любая комбинация методов — RLE, MTF, DC, PBS, HUFF, ARIC, ENUC, SEM...

Методы этой группы являются **трансформирующими** и **поточными** (т.е. могут применяться даже в том случае, когда длина блока с данными не задана). Скорость выполнения прямого преобразования равна скорости обратного преобразования и

зависит только от размера данных, но не от их содержания, т.е. Скорость=O(Размер). Памяти требуется порядка *h* байтов.

Из краткого описания общей идеи видно, что

1) можно один раз, при инициализации, задать как *h*, так и K_j,
2) но с точки зрения сложности вычислений и качества сжатия, полезно ограничить *h*, но периодически определять оптимальные K_j;
3) теоретически, одна и та же зависимость может быть задана несколькими формулами, например

   $S_i=(S_{i-1}+S_{i-2})/2$ и $S`_i= S_{i-1}/2 + S_{i-2}/2$,

   но реально, из-за использования целочисленной арифметики, они не эквиваленты; в данном случае при вычислении $S`_i$ используется два округления вместо одного, поэтому мы получим различие.

Прямое преобразование

В базовом простейшем случае иллюстрируется одной строкой:

```
/* предполагаем, что S[i]≈S[i-1], и, следовательно,
   D[i]≈0
*/
D[i]=S[i]-S[i-1];
```

S[*N*] — исходный массив (Source, источник)
D[*N*] — выходной массив преобразованных данных (Destination, приемник, с отклонениями, т.е. разностями, дельтами).

Такой вариант обычно называется дельта-кодирование (Delta Coding), поскольку записываем приращения элементов относительно значений предыдущих элементов. Если в значениях элементов имеется ярко выраженная линейная тенденция, то в результате получаем ряд чисел с близкими значениями.

Три строки, если требуется писать в тот же массив:

```
current=S[i];      //возьмем очередной элемент
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
S[i]=current-last; //вычислим его разность с прошлым
last=current;      //теперь очередной стал прошлым
```

Например, из блока (12, 14, 15, 17, 16, 15, 13, 11, 9), применив этот простейший вариант преобразования, получим: (2, 1, 2, –1, –1, –2, –2, –2).

Более сложный вариант: h=4, предполагаем, что K_i=1/4: S[i]≈(S[i–1] + S[i–2] + S[i–3] + S[i–4])/4.

Тогда в алгоритме:

```
D[i]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Здесь мы предполагаем, что явно выраженная линейная тенденция отсутствует, и значение очередного элемента примерно равно значению предыдущего, причем D[i] приобретает отрицательные и положительные значения с равной вероятностью. Чтобы компенсировать влияние случайных возмущений, значение следующего элемента принимаем равным среднему арифметическому нескольких последних элементов. С другой стороны, чем больше элементов усредняем, тем консервативнее наша модель, тем с большим запаздыванием она будет адаптироваться к существенным изменениям (необязательно линейного характера) во входных данных.

Если результат преобразования требуется записывать в тот же входной массив S, то это можно реализовать следующим образом. Введем вспомогательный массив Last, в котором будем хранить ис-ходные значения последних четырех преобразованных элементов массива S.

```
int Last[4]={ 0,0,0,0 };
int last_pos=0;
```

И далее в цикле преобразования для каждого элемента S[i] выполняем:

```
current=S[i]; // возьмем очередной элемент и вычислим
              // его разность с предсказываемым значением:
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
S[i]=current-(Last[0]+Last[1]+Last[2]+Last[3])/4;
Last[last_pos]=current;  // внесем current в массив Last
last_pos=(last_pos+1)&3;  // новое значение позиции в
Last
```

> **Упражнение:** Как будет выглядеть алгоритм для модели S[*i*]=(S[*i*–1] + S[*i*–2] + S[*i*–3])/3 ? Будет ли он выполняться быстрее или медленнее рассмотренного варианта с h=4 и K_j=1/4 ?

ОБРАТНОЕ ПРЕОБРАЗОВАНИЕ

Алгоритм очень похож на алгоритм прямого преобразования. Единственное отличие — теперь вычисляем не разность, а сумму предсказываемого значения и отклонения:

```
S[i]=D[i]+S[i-1]; // по предположенному, S[i]=S[i-1]
```

Если для D и S используем один и тот же массив (S):

```
S[i]=S[i]+S[i-1]; // сумма дельты с предыдущим элементом
```

В случае h=4 , K_j=1/4 :

```
S[i]=D[i] + (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Если пишем в тот же массив, то и в этом случае, в отличие от прямого преобразования, не требуется каких-то ухищрений:

```
S[i]=S[i] + (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Действительно, к моменту обратного преобразования очередного элемента S[*i*] в ячейках S[*i*–1], ..., S[*i*–4] уже находятся восстановленные значения, а не отклонения, что нам как раз и нужно. Таким образом, использовать при разжатии два массива совершенно необязательно.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Пути увеличения скорости сжатия

Если можно записывать в тот же массив, то можно обойтись без излишнего массива Last[h] и связанных с ним операций, выполняемых на каждом шаге цикла преобразования.

Рассмотрим все тот же пример с h=4 , K_j=1/4. Сделаем так, чтобы внутри цикла было присваивание:

```
S[i-4]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Тогда первые четыре элемента нам придется записывать отдельно:

```
First[0]=S[0];
First[1]=S[1] - S[0];
First[2]=S[2] - (S[1]+S[0])/2;
First[3]=S[3] - (S[2]+S[1]+S[0])/3;
```

Далее с помощью цикла преобразуем цепочку из последних (N–4) элементов и запишем результаты в первые (N–4) ячейки массива S:

```
for (i=4; i<N; i++)
   S[i-4]=S[i] - (S[i-1]+S[i-2]+S[i-3]+S[i-4])/4;
```

Или даже так:

```
sum=S[3]+S[2]+S[1]+S[0];
for (i=4; i<N; i++)
   j=S[i-4], S[i-4]=S[i] - sum/4, sum=sum-j+S[i];
```

Теперь мы можем переписать первые четыре преобразованных элемента из вспомогательного массива First в S:

```
for (i=0; i<4; i++)  S[N-4+i]=First[i];
```

Таким образом, дополнительной памяти вне функции преобразования не требуется, внутри — только h элементов, скорость преобразования в общем случае та же, какая была бы при записи во второй массив.

Единственное осложнение — метод, сжимающий данные, полученные в результате преобразования LPC, должен сначала обработать *h* элементов из конца массива S, а лишь затем (*N*–*h*) из начала.

Если работаем не с потоком, а с блоком, и длина массива S известна заранее, можно обрабатывать из конца в начало: в цикле по *i* от *N*–1 до 0 делаем: S[*i*+1] – =S[*i*].

Увеличение скорости разжатия

Если памяти достаточно и входных данных много, может оказаться выгодным сделать табличную реализацию каких-то арифметических операций (функций). С целью ускорения работы основного цикла можно, например, за счет этого избежать операции деления, выполняемой обычно гораздо дольше, чем операции сложения/вычитания и записи/чтения в/из памяти.

Например, вместо

```
S[i-3]=S[i] - (S[i-1]+S[i-2]+S[i-3])/3;
```

намного быстрее на большинстве процессоров и для большинства компиляторов будет

```
S[i-3]=S[i]- Value[ S[i-1]+S[i-2]+S[i-3] ];
```

или даже

```
              // R - размер элементов массива S в битах
S[i-3]=Value2[ (S[i-1]+S[i-2]+S[i-3])<<R + S[i] ];
```

> **Упражнение:** Напишите циклы, заполняющие вспомогательные массивы Value[] и Value2[].

Естественно, такой подход применим и для увеличения скорости сжатия.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Пути улучшения степени сжатия

Как мы уже отмечали, целесообразно повторно вычислять коэффициенты K_j через каждые P шагов. На основании последних обработанных данных находим значения коэффициентов, оптимальные с точки зрения точности предсказания значений элементов, встреченных в блоке последних обработанных данных. Если P=1, такой вариант обычно называется адаптивным. При большом значении P вариант называется статическим, если значения K_j записываются в сжатый поток, и "полуадаптивным", "блочно-адаптивным" или "квазистатическим", если в явном виде значения K_j не записываются.

При сжатии с потерями можем сохранять не точные значения каждой разности $D_i = S_i - \Sigma(K_j \cdot S_j)$, а только первые B битов или, например, номер первого ненулевого бита. Но тогда при сжатии нужно использовать на следующих шагах то же значение S_i, которое получится при разжатии, а именно $S_i` = \Sigma(K_j \cdot S_j) + D_i`$, то есть результат предсказания плюс сохраненная часть разности-дельты.

Метод LPC применяется обычно для сжатия аналоговых сигналов. И как правило, каждый элемент сигнала отклоняется от своего предсказываемого значения не только из-за «сильных» обусловленных изменений — эволюции, но и из-за «слабых» фоновых колебаний, то есть шума. Поэтому возможно **два противоположных типа моделей**:

- вклад шума невелик по сравнению с вкладом эволюции;
- вклад эволюции невелик по сравнению с вкладом шума.

В первом случае мы будем предсказывать значение S[i] на основании сложившейся линейной тенденции, во втором — как равное среднему арифметическому h предыдущих элементов.

Тенденция может быть и нелинейной, например S[i]=S[i–1]+2*(S[i–1]–S[i–2]), но такие модели на практике в большинстве случаев менее выгодны, и мы их рассматривать не будем.

Если ресурсов достаточно, можно вычислять предсказываемые значения, даваемые несколькими моделями, и затем либо синтезировать эти несколько предсказаний, либо выбирать (через каждые *P* шагов) ту модель, которая оптимальна на заданном числе предыдущих элементов.

Требуется минимизировать ошибки предсказаний, поэтому рациональнее (выгоднее) та модель, которая дает наименьшую сумму абсолютных значений этих ошибок:

$$\min(\sum_i |D_i|)$$

Эта задача имеет элементарное аналитическое решение: min|D| = min (|Y-K*X|) → K = Y*X^{-1}, где Y – вектор из S[i] (реально наблюдаемых), K – матрица коэффициентов и X – матрица S[i-x] (x = 1,...,h). Но (1) сложность вычисления обратной матрицы не менее O(h^2) и перевычислять ее невыгодно; (2) высокая точность прогноза обычно не нужна, т.к. на практике стационарные последовательности встречаются редко.

Рассмотрим подробнее на примерах.

Общий случай

Если *h*=1, то учитывается только один последний элемент: S[*i*]=*K*·S[*i*–1]. Об эволюции не известно ничего, но можно предполагать, что *K*=1.

> **Упражнение:** Изобразите сигнал, оптимально сжимаемый с помощью модели с *h*=1 и *K*= –1.

Все это, однако, не помешает нам действовать следующим образом: выберем начальное значение K_0, шаг *StepK* изменения *K*, размер окна *F*, а также *P*. Будем следить, с какой из трех моделей — $K=K_0$, $K_p=K_0+StepK$ и $K_m=K_0-StepK$, сжатие последних *F* элементов лучше (меньше суммарная ошибка предсказания), и через каждые *P* элементов выбирать лучшую (в данный момент) из трех моделей, с помощью которой и будут сжиматься следующие *P* элементов.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Если это не текущая, а одна из двух с ±*StepK*, то изменяются и K_p с K_m, если, конечно, их значения не совпадают с границами: –1 или 1.

Например, можно использовать K_0=0, *StepK*=1/4, *F*=10 и *P*=2. Имеет смысл задать еще один параметр *M* — минимальный размер выигрыша оптимальной модели у текущей. Если выигрыш меньше *M*, то отказываемся от смены модели.

Следующий по сложности вариант использует переменный шаг для *K*. Также мы можем похожим образом корректировать *F* (например, изменять с шагом *StepF*, через каждые *Q* обработанных элементов).

Если *h*=2.

Выражение $S_i = \sum_{j=i-1}^{j=i-h} K_j S_j$ можно всегда переписать в таком виде:

$$S_i = K`_1 S_{i-1} + \sum_{j=1}^{j=h-1} K`_{j+1}(S_{i-j} - S_{i-j-1})$$

Поэтому модель при *h*=2 можно представить как

$$S[i]=K_1 \cdot S[i-1] + K_2 \cdot (S[i-1] - S[i-2]).$$

Найдем значения коэффициентов «**эволюционной**» модели. Если шум стремится к нулю, то изменения элементов объясняются только линейной тенденцией (эволюцией). Иначе говоря:

$$S[i]-S[i-1]=S[i-1]-S[i-2].$$

Отсюда получаем: K_1=1, K_2=1.

Для «**шумовой**» модели $S[i]=(S[i-1]+S[i-2])/2$, и коэффициенты K_j получим из системы:

$K_1+K_2=1/2$ (коэффициент при S[*i*-1])

$-K_2=1/2$ (коэффициент при S[*i*-2])

Находим: K_1=1, K_2= –1/2. Таким образом, имеет смысл корректировать K_2 в диапазоне не от 0 до 1, как может показаться на первый взгляд, а от –1/2 до 1.

Если h=3:

$$S[i]=K_1\cdot S[i-1] + K_2\cdot(S[i-1]-S[i-2]) + K_3\cdot(S[i-2]-S[i-3])$$

(6.1)

У эволюционных моделей уже нет однозначного ответа о K_1, K_2 и K_3: K_1=1, K_2+K_3=1 (т.к. в общем случае через три точки нельзя провести прямую). Например, можно взять K_2=3/4 и K_3=1/4, т.е. последнее приращение имеет втрое больший вес, чем предпоследнее. Или же K_2=2, K_3= –1 (вторая производная постоянна).

Шумовая модель в одномерном случае одна при любых h, и при h=3:

$$S[i]=(S[i-1] + S[i-2] + S[i-3])/3 \qquad (6.2)$$

Попробуем совместить обе модели — шумовую и эволюционную.

При каких K_1, K_2 и K_3 эволюционная (6.1) становится шумовой (6.2)?

Из системы

K_1+K_2=1/3 (коэффициент при S[i-1])
–K_2+K_3=1/3 (коэффициент при S[i-2])
–K_3=1/3 (коэффициент при S[i-3])

находим: K_1=1, K_2= –2/3, K_3= –1/3.

Имеет смысл пытаться корректировать K_3 от –1 до 1, при этом: (K_2+K_3) от –1 (шумовая модель) до 1 (эволюционная), то есть K_2 от (–1–K_3) до (1–K_3):

K_2 \ K_3	–1	…	…	1
–1–K_3		Ш		
…				
1–K_3	Э	Э	Э	Э

Ш — такая пара значений (K_2= –2/3, K_3= –1/3) соответствует шумовой модели,

Э — такие пары K_2+K_3=1 соответствуют эволюционной (а остальные — «средней»).

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

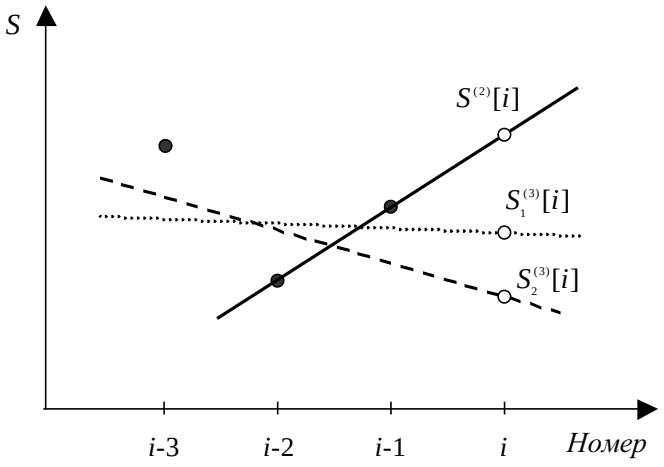


Рис. 6.1. Эволюционная модель при h=2 и h=3

При h = 2 существует только одна эволюционная модель (через две точки всегда можно провести только одну прямую), по которой предсказываемое значение $S[i] = S^{(2)}[i]$;

При h = 3 возможно использование множества моделей: например, если считать изменение $S[i]$-$S[i$-1] более важным, чем $S[i$-2]-$S[i$-3], то такая модель может дать, например, предсказание $S_1^{(3)}[i]$, или, если считать $S[i$-2]-$S[i$-3] и $S[i]$-$S[i$-1] равноправными, то такая модель дает прогнозное значение $S_2^{(3)}[i]$.

Кроме того, при h=3 возникает возможность **уменьшать шум**. Метод заключается в следующем. После того как найдено среднее значение (S[i–1]+S[i–2]+S[i–3])/3 , определим, какой из трех элементов больше отклоняется от него. Будем считать, что именно этот элемент содержит больше шума, чем остальные два, поэтому именно его учитывать не будем (или, что то же самое, положим его равным полусумме двух остальных). В качестве предсказания, даваемого шумовой моделью, возьмем полусумму двух остальных элементов.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Аналогично при ***h*=4** есть возможность взять три приращения («дельты») и оставить те два, которые ближе к среднему всех трех.

Остается добавить, что ряд методов для сжатия речи с потерями (CELP, CS-ACELP, MELP) использует LPC для вычисления нескольких характеристик фрагментов сигнала (процесс называется LPC analysis). После обратного процесса (LPC synthesis) получается сигнал с теми же характеристиками, но совершенно другими данными, то есть значениями элементов.

Двумерный случай

Рассмотрим только самый простой обход плоскости[1] — строками.

В каждой точке плоскости уже обработанные, известные элементы — выше данной точки, а также на том же уровне, но левее ее.

S[*i*-*L*-1]	S[*i*-*L*]	S[*i*-*L*+1]
S[*i*-1]	S[*i*]	…

Здесь L — число элементов в строке. Ближайших к S[i] элементов — четыре. Обозначим для краткости так:

A	B	C
D	E	…

У «шумовой» модели есть 4 варианта с h=1, 6 вариантов с h=2, четыре с h=3 и один с h=4:

$$E=K_1\cdot D + K_2\cdot C + K_3\cdot B + K_4\cdot A \qquad (6.3)$$

[1]Задача обхода плоскости возникает при обработке двумерных данных. Цель обхода — создание одномерного массива D из двумерного S. Подробнее смотри в разделе «Алгоритмы сжатия изображений».

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Можно изначально задать $K_1=K_2=K_3=K_4=1/4$ и корректировать K_i вышеизложенным методом. Границы будут заданы условиями: $K_i>0$, $K_1+K_2+K_3+K_4=1$, и, например, $K_1=K_3$, $K_2=K_4$ (и таким образом $K_1+K_2=1/2$, достаточно корректировать K_1).

У вариантов с $h>2$ есть возможность уменьшать шум, тем же методом, что и в одномерном случае.

Таким образом, если использовать только четыре или меньше ближайших элементов, шумовая модель дает 20 вариантов: 4 с $h=1$, 6 с $h=2$, 8 с $h=3$ и 2 с $h=4$.

«Эволюционная» модель делает то же предположение, что и в одномерном случае: сохраняется линейная тенденция, приращение на текущем шаге примерно равно приращению на предыдущем. Но теперь, в двумерном случае, необходимо как минимум три элемента, а не два:

$E=K_1\cdot B + K_2\cdot(D–A)$,

$E=K_1\cdot D + K_2\cdot(B–A)$, (6.4)

$E=K_1\cdot D + K_2\cdot(B–A) + K_3\cdot(C–B)$. (6.5)

Первые два варианта, по предположению эволюционной модели, сводятся к одному: $K_1=K_2=1$. В третьем, при коррекции в рамках эволюционной модели, границы таковы: $K_1=1$, $K_2+K_3=1$.

Попробуем построить модель с $h=4$, объединяющую два противоположных полюса — «шумовой» и «эволюционный».

При каких K_1, K_2 и K_3 эволюционная модель (6.5) становится шумовой: (6.3) с $K_i=1/4$?

$K_1=1/4$ (коэффициент при D)

$K_3=1/4$ (при C)

$K_2–K_3=1/4$ (B)

$–K_2= 1/4$ (A)

Решений нет и правильный ответ: ни при каких. Потому что в (6.3) коэффициенты при C, B и A должны быть $K_i>0$. Применяя это условие к (6.5), получаем:

(C): $K_3>0$

(A): $K_2<0$

(B): $K_2–K_3>0$ (это несовместимо с результатами по C и A).

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Тем не менее, синтез «шумовой» и «эволюционной» при h=4 возможен. Будем брать предсказание того из 20-и вариантов «шумовой», который ближе всего к предсказанию «эволюционной», то есть разность между ними минимальна.

Как раз такой путь реализован в популярном алгоритме **PNG**, а именно в самом сложном, самом интеллектуальном его фильтре "Paeth": три варианта шумовой (с использованием элементов A, B и D) и эволюционная (6.4). По сути это тот же метод, что и описанный в пункте «Общий случай», только вместо среднего по трем точкам берется «двумерное эволюционное» значение, и исключаются два элемента, а не один.

Другой синтез возможен, если эволюционную модель модифицировать:

$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot A$ (5a)

$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot B$ (5b)

$E=K_1\cdot D+K_2\cdot(B-A)+K_3\cdot(C-B)+K_4\cdot C$ (5c)

Чтобы формулы описывали эволюционную модель, должно быть K_4=0, $K_1=K_2+K_3$=1. В шумовой модели K_1=1/4, а K_2, K_3 и K_4 найдем из условий: коэффициенты при A, B и C должны быть равны 1/4:

в случае

(5a) C: K_3=1/4, B: K_2=1/2, A: K_4= 3/4

(5b) C: K_3=1/4, A: K_2= –1/4, B: K_4= 3/4

(5c) A: K_2= –1/4, B: K_3= –1/2, C: K_4= 3/4

Рассмотрим, например, (5b). Имеет смысл корректировать K_1 от 1/4 (шумовая модель) до 1 (эволюционная); K_4 найдется из условия K_1+K_4=1; K_3=1/4; K_2+K_3 от 0 (шумовая) до 1 (эволюционная), то есть K_2 от –1/4 до 3/4.

Таким образом, из (5b) получается эволюционная модель при K_1=1, K_2=3/4, и шумовая при K_1=1/4, K_2= –1/4.

Упражнение: Каким будет синтез эволюционной и шумовой моделей в случаях (5a) и (5c) ?

LPC в алгоритме PNG

Для сжатия изображения можно выбрать одну из следующих LPC-моделей (называемых также «фильтрами»):

1) E=0 (нет фильтра)
2) E=D (элемент слева)
3) E=B (элемент сверху)
4) E=(B+D)/2
5) вышеописанный алгоритм Paeth

Все пять — варианты шумовой модели, и только последняя модель является комбинированной, учитывающей эволюцию.

LPC в алгоритме Lossless Jpeg

Может быть задан один из восьми предсказателей-фильтров:

1) E=0 (нет фильтра)
2) E=B (элемент сверху)
3) E=D (элемент слева)
4) E=A (выше и левее)
5) E=B+D–A
6) E=B+(D–A)/2
7) E=D+(B–A)/2
8) E=(B+D)/2

Пятый, шестой и седьмой — варианты эволюционной модели, остальные — шумовой.

Выбор фильтра

Итак, имеем множество фильтров $Sp_n[X,Y]=S_n(K_{i,j}\cdot S[X–i,Y–j])$ дающих оценки-предсказания Sp_n для значения элемента в позиции (X,Y) как функцию S_n от значений элементов контекста, то есть элементов, соседних с текущим $S[X,Y]$.

Два уже рассмотренных (в пункте «общий случай») пути дальнейших действий:

1. выбрать одно значение Sp_n , даваемое той функцией S_n , которая точнее на заданном числе предыдущих элементов,
2. выбрать значение, вычисляемое как сумма всех Sp_n , взятых с разными весами: $S_p = \Sigma(W_n\cdot Sp_n)$, $0\le W_n\le 1$.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Кроме довольно очевидного способа — корректировки W_n, весов, с которыми берутся предсказания, даваемые разными фильтрами, есть и еще варианты:

3. брать те K значений Sp_n, которые дают S_n, лучшие на K (задаваемом числе) ближайших элементов контекста (некоторые из этих S_n могут совпадать, и их останется меньше, чем K).
4. все функции S_n сортируются по значению точности предсказания на ближайших элементах контекста и выбирается несколько функций, дающих лучшую точность.

Четвертый вариант является простым расширением первого. Третий же предполагает сопоставление каждому элементу одной S_n, дающей для него самое точное предсказание. В обоих случаях, если требуется выбрать одну S_n из нескольких S_i с одинаковым значением точности, можно посмотреть значения точности этих S_i на большем числе элементов.

Во 2-м, 3-м и 4-м часто полезно уменьшить шум вышеописанным способом. И в конечном итоге сложить оставшиеся $S_p = \Sigma(W_n \cdot Sp_n)$. В простейшем случае $W_n=1/H$, где H — количество оставшихся S_n.

И еще три простых, но важных замечания:

Во-первых, сжатие обычно лучше, если значения оценок-предсказаний Sp_n не выходят за **границы диапазона** допустимых значений элементов входного потока $S[i]$. Если $A<S[i]<B$, то и $Sp_n[i]$ должны быть $A<Sp_n[i]<B$.

Во-вторых, если известно, что **доля шума постоянна** во всем блоке данных, очень полезно делать SEM до LPC, если нет другого метода для отделения шума.

И, в-третьих, в каждой точке (x,y) веса $K_{a,b}$ значений $S_{a,b}$ из (x–a, y–b) должны зависеть от **расстояния** до (x,y), вычисляемого как $R=(a^2+b^2)^{1/2}$. Эти веса должны убывать с увеличением расстояния R. Из четырех ближайших элементов контекста, рассмотренных в пункте «Двумерный случай», D и B находятся на расстоянии 1, а A и C на расстоянии $2^{1/2}$. Поэтому, если как в

формуле (6.3), шумовая модель складывает значения четырех ближайших элементов контекста с разными весами K_i :

$$E=K_1 \cdot D + K_2 \cdot C + K_3 \cdot B + K_4 \cdot A$$

имеет смысл задавать $K_1=K_3$, $K_2=K_4$, и еще из-за учета расстояний $K_2/K_1=2^{1/2}$ (и конечно $K_1+K_2+K_3+K_4=1$).

Точно так же при вычислении точностей фильтров на элементах контекста, ошибки их предсказаний $V_{a,b}$ (а точнее, абсолютные значения этих ошибок) должны учитываться с весами $K_{a,b}$, зависящими от расстояний: $K_{a,b}=K((a^2+b^2)^{1/2})$.

У эволюционной модели, использующей четыре элемента контекста (A, B, C, D), приращения (B–A), (D–A) и (C–B) находятся на равном расстоянии и имеют одинаковый вес. Но если используется больше четырех элементов — тоже необходимо вычислять расстояния.

Характеристики методов семейства LPC:

Степень сжатия: увеличивается примерно в 1.1 ... 1.9 раза.

Типы данных: методы предназначены для сжатия количественных данных.

Симметричность по скорости: в общем случае 1:1.

Характерные особенности: чем меньше доля "шума" в данных, тем выгоднее применение методов LPC.

Субполосное кодирование

Английское название метода — «Subband Coding» (SC). Дословный перевод — «кодирование поддиапазонов».

Цель метода — сжатие потока R-битных элементов, в предположении, что значение каждого из них отличается от значений соседних элементов незначительно: $S_i \approx S_{i-1}$.

Основная идея состоит в том, чтобы формировать два потока: для каждой пары S_{2i}, S_{2i+1} сохранять полусумму $(S_{2i} + S_{2i+1})/2$ и разность $(S_{2i} – S_{2i+1})$. Далее эти потоки следует обрабатывать раздельно, поскольку их характеристики существенно различны.

В случае модели «аналоговый сигнал» физический смысл потока с полусуммами — низкие частоты, а с разностями — высокие. Методы SC предназначены для сжатия элементов с «количественными» данными, а не «качественными». Самый распространенный вид количественных данных — мультимедийные. Но не единственный: например, потоки смещений и длин, формируемые методом семейства LZ77, тоже содержат количественные данные.

Размер данных в результате применения SC не изменяется. Более того, в потоке с разностями размер элементов R может даже увеличиваться на 1 бит: R`=R+1, поскольку для сохранения разностей R-битных элементов требуется (R+1) битов.

Для сжатия результата работы метода может быть применена любая комбинация методов — RLE, MTF, DC, PBS, HUFF, ARIC, ENUC, SEM...

Методы этой группы являются **трансформирующими** и **поточными** (т.е. могут применяться даже в том случае, когда длина блока с данными не задана). В общем случае скорость работы компрессора равна скорости декомпрессора и зависит только от размера данных, но не от их содержания: Скорость = O(Размер). Памяти требуется всего лишь несколько байтов.

Из краткого описания общей идеи видно, что

1) к обоим получающимся потокам можно повторно применять метод этого же семейства SC;
2) метод можно применять и к параллельным потокам (например, левый и правый каналы стереозвука);

3) при сжатии аналогового сигнала с потерями, степень сжатия обратно пропорциональна ширине сохраняемого диапазона частот;
4) можно сохранять не полусуммы и разности, а суммы и полуразности (поскольку сумма и разность двух чисел — либо обе четны, либо обе нечетны, одну из них можно делить на 2 без всяких осложнений).

ПРЯМОЕ ПРЕОБРАЗОВАНИЕ

В базовом простейшем случае иллюстрируется тремя строками:

```
for (i=0; i<N/2; i++) { // цикл по длине исходного масси-
ва
  D[2*i]=(S[2*i]+S[2*i+1])/2;// четные - с полусуммами
  D[2*i+1]=S[2*i]-S[2*i+1];  // нечетные - с разностями
}                            // (1)
```

S[*N*] — исходный массив (Source, источник)
D[*N*] — выходной массив преобразованных данных (Destination, приемник, с Дельтами, т.е. разностями, и полусуммами).

Если результат пишем в тот же массив S:

```
for (i=0; i<N/2; i++) {     // (2)
  d=S[2*i]-S[2*i+1];       // разность, и дальше так же:
// четные будут с полусуммами:
  S[2*i]=(S[2*i]+S[2*i+1])/2;
  S[2*i+1]=d;             // нечетные элементы массива -
}                         // с разностями
```

Если же формируем два выходных массива:

```
for (i=0; i<N/2; i++) {      // (3)
  A[i]=(S[2*i]+S[2*i+1])/2;  // полусумма (Average)
  D[i]=S[2*i]-S[2*i+1];      // разность (Delta)
}
```

То каждый из них — вдвое короче, чем исходный S.

Если его длина *N* нечетна, добавим к концу S элемент S[*N*], равный последнему S[*N*–1]; в результате он добавится к массиву A, а к D добавится ноль:

```
A[N/2]=S[N-1];   // последняя полусумма
D[N/2]=0;        // последняя разность
```

Если известно, что в результате разности может возникнуть **переполнение**, то есть невозможно будет записать полученное значение, используя исходное число битов *R* (*R* — размер элементов исходного массива S):

```
if ( (S[2*i]-S[2*i+1])!=(S[2*i]-S[2*i+1])mod(n) )
{} // n=(2 в степени R)
```

то можем поступить одним из следующих **четырех** способов:

1. Изначально отвести под разности массив элементов большего размера:

```
char A[N];   // если под полусуммы - 8 битов,
short D[N];
// то под разности- 9 битов, а реально даже 16
```

2. Формировать третий (битовый) массив с флагами переполнений:

```
D[i]= S[2*i]-S[2*i+1];     //сохраним разность;
if(D[i]!=S[2*i]-S[2*i+1])
//если не хватило битов для записи,
    Overflow[i]=1;         //установим флаг в 1
else Overflow[i]=0;        //иначе его значение - ноль
```

3. Использовать текущее значение разности для вычисления полусуммы:

(4)

```
D[i]= S[2*i]-S[2*i+1];
            // реально D[i]=(S[2*i]-S[2*i+1])mod(n);
A[i]= S[2*i]-D[i]/2;
            //это полусумма, если не было переполнения
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

4. Наоборот — использовать текущее значение суммы для вычисления полуразности:

(5)

```
A[i]= S[2*i]+S[2*i+1];
         // реально A[i]=(S[2*i]+S[2*i+1])mod(n);
D[i]= S[2*i]-A[i]/2;
         // это полуразность, если не было переполнения
```

> **Упражнение:** Можно ли использовать полуразности для вычисления сумм? А полусуммы для вычисления разностей?

Обратное преобразование

Обратное преобразование ничуть не сложнее прямого:

```
for (i=0; i<N/2; i++) {           // (3`)
  S[2*i]=  ( 2*A[i]+D[i] ) /2 ;
  S[2*i+1]=( 2*A[i]-D[i] ) /2 ;
}
```

Если использовалась защита от переполнения, и брались разности для вычисления полусумм:

```
S[2*i]= A[i]+D[i]/2;              // (4`)
S[2*i+1]= S[2*i]-D[i];
               // реально D[i]=(S[2*i]-S[2*i+1])mod(n)
```

Пути улучшения степени сжатия

Общий случай

К получаемым потокам можно и дальше рекурсивно применять разложение на полусуммы (Average) и разности (Delta). При сжатии аналоговых сигналов, как правило, полезно дальнейшее разложение полусумм.

Есть два пути создавать три выходных потока или блока:

1. Original↓
   Average1+Delta1↓
   DA+DD

(**DA** — Average2, получаемая при разложении разностей Delta1, **DD** — Delta2, получаемая при разложении Delta1. Аналогично с остальными обозначениями).

2. Original↓
   Average1↓+**Delta1**
   AA+AD

Пять вариантов создания четырех:

1. Original↓
   Average1+Delta1↓
   DA+DD↓
   DDD+DDA

2. Original↓
   Average1+Delta1↓
   DA↓+**DD**
   DAD+DAA

3. Original1↓
   Average1↓ + Delta1↓
   AA+AD **DA+DD**

4. Original↓
   Average↓+**Delta**
   AA↓+**AD**
   AAA+AAD

5. Original↓
   Average↓+**Delta**
   AA+AD↓

ADA+ADD

14 вариантов создания пяти, и так далее.

> **Упражнение:** Изобразите эти 14 вариантов.

При принятии решения о целесообразности разложения некоторого потока или блока S на A и D, оказывается полезен следующий прием. Будем заглядывать на несколько шагов вперед: если непосредственные результаты разложения A и D сжимаемы суммарно хуже, чем исходный S, может оказаться, что если разложить дальше A и/или D, то только тогда результат — три или четыре потока, по которым восстановим S, — сжимаем лучше, чем S.

Если же и эти AA, AD, DA, DD дают в сумме худшее сжатие, заглянем еще на шаг вперед, то есть попытаемся разложить эти вторичные результаты, и так далее, пока глубина такого просмотра не достигнет заданного предела, или же дальнейшее разложение окажется невозможным из-за уменьшения длин в два раза на каждом шаге.

Еще одно усложнение — поиск границ фрагментов потока, разбивающих его на такие блоки, чтобы их дальнейшее субполосное кодирование давало лучшее сжатие.

При сжатии параллельных потоков

Если имеем два параллельных потока X и Y, каждая пара (X_i, Y_i) описывает один объект. Например, X_i — адрес, а Y_i — длина, или же X — угол, Y — расстояние. В случае «аналоговый сигнал» пара (X_i,Y_i) относится к одному моменту времени t_i.

Число потоков и их длина N остаются неизменными. Можно оставлять один из двух исходных потоков вместо потока с полусуммами (а при сжатии с потерями и вместо разностей): X+D или Y+D, вместо A+D.

Теперь можно искать границы таких блоков, внутри которых один из этих четырех вариантов X+Y, X+D, Y+D, A+D существенно выгоднее остальных трех.

Заметим, что применение LPC, в том числе Дельта-кодирования, к потоку полусумм A выгоднее, чем к X и/или Y: первая производная потока с полусуммами A`[i] не превосходит (по абсолютному значению) максимума первых производных X`[i] и Y`[i]:

$$A`_i = A_{i+1} - A_i = (X_{i+1} + Y_{i+1} - \alpha_{i+1})/2 - (X_i + Y_i - \alpha_i)/2 , \qquad (6.6)$$

«издержки округления» α_k равны единице, если сумма соответствующих X_k+Y_k нечетна, а арифметика используется целочисленная; иначе $\alpha_k=0$.

$$A`_i = (X_{i+1} - X_i + Y_{i+1} - Y_i + \alpha_i - \alpha_{i+1})/2 = (X`_i + Y`_i + \alpha_i - \alpha_{i+1})/2. \qquad (6.7)$$

Таким образом, значение $A`_i$ лежит внутри интервала $[X`_i, Y`_i]$.

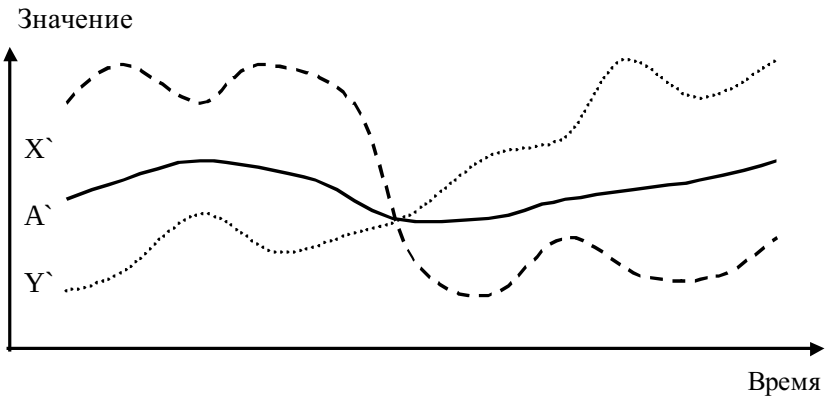


Рис. 2.2. значение $A`_i$ лежит внутри интервала $[X`_i, Y`_i]$

Заметим также, что (при использовании целочисленной арифметики из-за округлений) важен порядок действий: дельта-кодирование потока полусумм A даст другой результат, чем по-

лусумма результатов дельта-кодирования X и Y: в первом случае (6.6) и (6.7), во втором $A``_i = (X`_i+Y`_i) /2$.

Если сумма $(X`_i+Y`_i)$ нечетна, а $\alpha_i-\alpha_{i+1}=1$, то $A`_i \neq A``_i$, $A`_i=A``_i+1$.

Например, если $X_i=0$, $X_{i+1}=Y_i=Y_{i+1}=1$, то $A`_i =1$ (по формуле (6.6)), а $A``_i=(1+0)/2=0$.

Если потоков более двух, например, пять (сжатие именно пятиканального звука становится все актуальнее), возникает задача нахождения оптимальных пар для применения к ним субполосного кодирования. Здесь тоже возможны оба вышеописанных приема: и «просмотр на несколько шагов вперед», еще более усложненный, и «поиск границ интервалов времени», оптимальных для выделяемых затем пар. Например, может выясниться, что сжатие существенно лучше, если использовать знание того, что на интервале между 32765-ым и 53867-ым элементами очень близки первая разность (при первом разложении) внутри полусуммы (1,4), то есть полусуммы 1-го потока с 4-ым, и первая разность внутри полусуммы (3,7).

Двумерный случай

Раньше на каждом этапе можно было принять одно из двух решений: применить SC-преобразование к блоку, или не применять. Теперь же — одно из пяти:

1. применить SC к строкам блока;
2. применить SC к столбцам блока;
3. применить SC к строкам блока, а затем к столбцам;
4. применить SC к столбцам блока, а затем к строкам;
5. не применять SC к блоку вообще.

3 и 4 не совсем эквивалентны при использовании целочисленной арифметики из-за округлений, но математическое ожидание расхождения их результатов равно нулю.

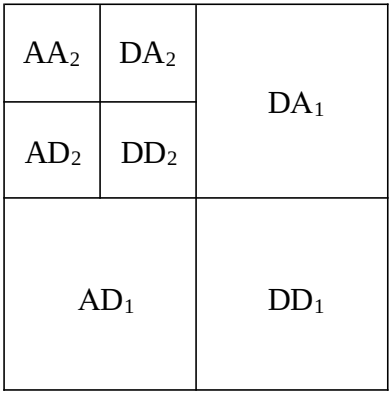


Рис. 2.3. Вариант применения SC в двумерном случае

При сжатии аналоговых сигналов

Появляются дополнительные возможности:

- сжимая с потерями, округлять с заданной точностью: на каждом шаге SC и/или после всех этапов SC;
- сохранять только заданную субполосу, многократно применяя SC-разложение и оставляя только верхнюю половину частот (разности) либо нижнюю (полусуммы);
- использовать дискретное вэйвлетное преобразование для нахождения низкочастотной и высокочастотной компонент.

Дискретное вэйвлетное преобразование

ДВП отличается от SC лишь тем, что в нем для вычисления низко- и высокочастотной компонент используется не два соседних элемента сигнала, а произвольное задаваемое число элементов D>2. Причем, в отличие от других контекстных методов, в частности LPC, контекст элемента S[x] состоит из элементов по обе стороны от него: S[x+i] и S[x–i], i=1,2,3,…,D/2.

Основная же идея — та же, что в SC: сформировать два потока — с низкими H[x] и высокими частотами L[x] — так, чтобы по половине значений H[x] и половине значений L[x] можно

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

было восстановить исходный поток S[x]. Оставляются либо четные H[2·x] и нечетные L[2·x+1], либо наоборот.

При разжатии сначала по H[2·x] и L[2·x+1] восстанавливаем четные S[2·x], затем по S[2·x] и L[2·x+1] находим нечетные S[2·x+1].

Если D=3, L[x]=S[x]–(S[x–1]+S[x+1])/2,

H[x]=S[x]+hi(*x*).

Функция hi(S[i]) должна быть выбрана так, чтобы она была выражаема через L[x+j], j=2k+1. Например,

$hi_1(x)$=(L[x–1]+L[x+1])/2,

$hi_2(x)$=(L[x–1]+L[x+1])/4,

$hi_3(x)$= –(L[x–1]+L[x+1])/2,

$hi_4(x)$= –(L[x–1]+L[x+1])/4,

$hi_5(x)$=(L[x–3]+L[x–1]+L[x+1]+L[x+3])/4, и так далее.

Возьмем вторую функцию hi_2 (именно этот вариант используется в алгоритме JPEG 2000) и посмотрим, каким образом H выражается через S.

H[x]=S[x]+(L[x–1]+L[x+1])/4 =

=S[x]+ (S[x–1] – (S[x–2]+S[x])/2 + S[x+1] – (S[x]+S[x+2])/2) /4

=S[x]+ (2·S[x–1] – S[x–2] – S[x] + 2·S[x+1] – S[x] – S[x+2]) /8

=(–S[x–2] + 2·S[x–1] + 6·S[x] + 2·S[x+1] – S[x+2]) /8.

ДВП обычно задается в виде набора коэффициентов, в данном случае (–1/8, 2/8, 6/8, 2/8, –1/8).

Графически это выглядит так:

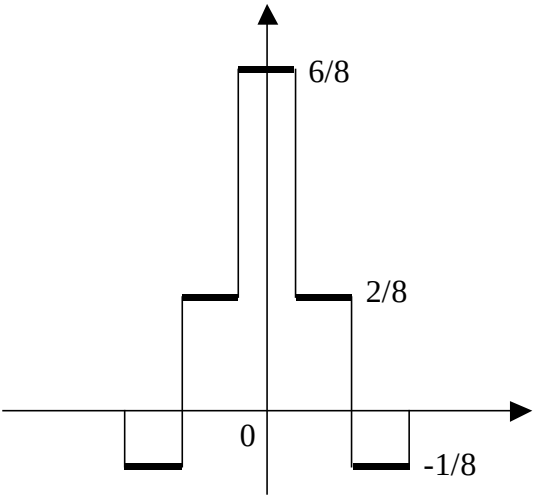


Рис. 2.4. ДВП для набора коэффициентов (–1/8, 2/8, 6/8, 2/8, –1/8)

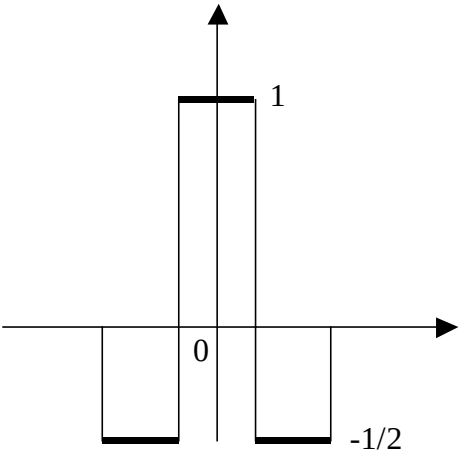


Рис. 2.5. ДВП для набора коэффициентов (–1/2, 1, –1/2)

Упражнение: Каким будет набор коэффициентов, если используем $hi_5(L[i])$?

Итак, при таком построении вэйвлет-фильтров, формируется два выходных потока:

$L[2\cdot x+1]=S[2\cdot x+1]+lo(S[2\cdot x+j])$, j —четные: $j= \pm 2k$, (6.8)

$H[2\cdot x]=S[2\cdot x]+hi(L[2\cdot x+i])$, i —нечетные: $i= \pm(2m+1)$.

Если сжимаем с потерями, функция lo может использовать любые S[x], а не только четные.

Обратное преобразование: сначала найдем все четные элементы:

$S[2\cdot x]=H[2\cdot x] – hi(L[2\cdot x+i])$, i —нечетные;

затем, на основании найденных четных, восстановим все нечетные элементы:

$S[2\cdot x+1]=L[2\cdot x+1] – lo(S[2\cdot x+j])$, j —четные.

Если, наоборот, берем четные L и нечетные H — это ничего по сути не меняет.

Заметим, что при сжатии с потерями существует и альтернативный путь. В прямом преобразовании сначала вычисляется $H[2\cdot x+1]=C\cdot S[2\cdot x+1]+hi(S[2\cdot x+j])$, j — четные: $j= \pm 2k$, $0<C<1$; затем $L[2\cdot x]=S[2\cdot x]+lo(H[2\cdot x+i])$, i — нечетные: $i= \pm(2m+1)$. Сравните с (6.8).

Например, если D=3, $H[x]=(S[x]+(S[x-1]+S[x+1])/2) /2$.

И еще заметим, что для улучшения сжатия применимо все то, что написано выше для SC. Но, кроме того, здесь, как и в ЛПК, можно через каждые K элементов выбирать фильтр, оптимальный для последних M обработанных элементов (K, M — задаваемые параметры). Причем метод может даже не записывать номер выбранного фильтра в поток в явном виде, поскольку аналогичный анализ может выполняться на этапе обратного преобразования.

Характеристики методов семейства SC:

Степень сжатия: увеличивается обычно в 1.1 ... 1.9 раза.

Типы данных: методы предназначены для сжатия количественных данных.

Симметричность по скорости: в общем случае 1:1.

Характерные особенности: может быть целесообразно многократное применение.

Глава 3. Словарные методы сжатия данных

Идея словарных методов

Входную последовательность символов можно рассматривать как последовательность строк, содержащих произвольное количество символов. Идея словарных методов состоит в замене строк символов на такие коды, что их можно трактовать как индексы строк некоторого словаря. Образующие словарь строки будем далее называть *фразами*. При декодировании осуществляется обратная замена индекса на соответствующую ему фразу словаря.

Можно сказать, что мы пытаемся преобразовать исходную последовательность путем ее представления в таком алфавите, что его «буквы» являются фразами словаря, состоящими, в общем случае, из произвольного количества символов входной последовательности.

Словарь — это набор таких фраз, которые, как мы полагаем, будут встречаться в обрабатываемой последовательности. Индексы фраз должны быть построены таким образом, чтобы в

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

среднем их представление занимало меньше места, чем требуют замещаемые строки. За счет этого и происходит сжатие.

Уменьшение размера возможно в первую очередь за счет того, что обычно в сжимаемых данных встречается лишь малая толика всех возможных строк длины *n*, поэтому для представления индекса фразы требуется, как правило, меньшее число битов, чем для представления исходной строки. Например, рассмотрим количество взаимно различных строк длины от 1 до 5 в тексте на русском языке (роман Ф.М. Достоевского «Бесы», обычный неформатированный текст, размер около 1.3 Мбайт):

Длина строки	Количество различных строк	Использовано комбинаций, % от всех возможных
5	196969	0.0004
4	72882	0.0213
3	17481	0.6949
2	2536	13.7111
1	136	100.0000

Иначе говоря, размер (мощность) алфавита равен 136 символам, но реально используется только 2536/(136·136)·100% ≈ 13.7 % от всех возможных двухсимвольных строк, и т.д.

Далее, если у нас есть заслуживающие доверия гипотезы о частоте использования тех или иных фраз, либо проводился какой-то частотный анализ обрабатываемых данных, то мы можем назначить более вероятным фразам коды меньшей длины. Например, для той же электронной версии романа «Бесы» статистика встречаемости строк длины 5 имеет вид:

N	Количество строк длины 5, встретившихся ровно *N* раз	Количество относительно общего числа всех различных строк длины 5, %
1	91227	46.3%
2	30650	15.6%
3	16483	8.4%

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

4	10391	5.3%
5	7224	3.7%
≥6	40994	20.7%
Всего	196969	100.0%

Заметим, что из всех 197 тысяч различных строк длины 5 почти половина встретилась лишь один раз, поэтому они вообще не будут использованы как фразы при словарном кодировании в том случае, если словарь строится только из строк обработанной части потока. Наблюдаемые частоты оставшейся части строк быстро уменьшаются с увеличением N, что указывает на выгодность применения статистического кодирования, когда часто используемым фразам ставятся в соответствие коды меньшей длины.

Обычно же просто предполагается, что короткие фразы используются чаще длинных. Поэтому в большинстве случаев индексы строятся таким образом, чтобы длина индекса короткой фразы была меньше длины индекса длинной фразы. Такой прием обычно способствует улучшению сжатия.

Очевидно, что процессы моделирования и кодирования, рассматриваемые в главе «Методы контекстного моделирования», для словарных методов сливаются. Моделирование в явном виде может выполняться уже только для индексов. Заметим, что апологеты идеи универсальных моделирования и кодирования последовательно изучают любой метод, не вписывающийся явно в их модель, и обычно достаточно убедительно доказывают, что для него можно построить аналог в виде статистического метода. Так, например, доказано, что несколько рассматриваемых ниже словарных алгоритмов семейства Зива-Лемпела могут быть воспроизведены в рамках контекстного моделирования ограниченного порядка, либо с помощью моделей состояний [6, 9].

Ниже будут рассмотрены алгоритмы словарного сжатия, относимые к классу методов Зива-Лемпела. В качестве примера словарного алгоритма иного класса укажем [7].

Методы Зива-Лемпела ориентированы на сжатие качественных данных, причем эффективность применения достигается в

том случае, когда статистические характеристики обрабатываемых данных соответствуют модели источника с памятью.

Классические алгоритмы Зива-Лемпела

Алгоритмы словарного сжатия Зива-Лемпела появились во второй половине 1970-х годов. Это были так называемые алгоритмы LZ77 и LZ78, разработанные совместно Зивом (Ziv) и Лемпелом (Lempel). В дальнейшем первоначальные схемы подвергались множественным изменениям, в результате чего мы сегодня имеем десятки достаточно самостоятельных алгоритмов и бессчетное количество модификаций.

LZ77 и LZ78 являются универсальными алгоритмами сжатия, в которых словарь формируется на основании уже обработанной части входного потока, т.е. адаптивно. Принципиальным отличием является лишь способ формирования фраз. В модификациях первоначальных алгоритмов эта свойство сохраняется. Поэтому словарные алгоритмы Зива-Лемпела разделяют на два семейства — алгоритмы типа LZ77 и алгоритмы типа LZ78. Иногда также говорят о словарных методах LZ1 и LZ2.

Публикации Зива и Лемпела носили чисто теоретический характер, так как эти исследователи на самом деле занимались проблемой измерения «сложности» строки, и применение выработанных алгоритмов к сжатию данных явилось, скорее, лишь частным результатом. Потребовалось некоторое время, чтобы идея организации словаря, часто в переложении уже других людей, достигла разработчиков программного и аппаратного обеспечения. Поэтому практическое использование алгоритмов началось спустя пару лет.

С тех пор методы данного семейства неизменно являются самыми популярными среди всех методов сжатия данных, хотя в последнее время ситуация начала меняться в пользу BWT и PPM, как обеспечивающих лучшее сжатие. Кроме того, практически все реально используемые словарные алгоритмы относятся к семейству Зива-Лемпела.

Необходимо сказать несколько слов о наименованиях алгоритмов и методов. При обозначении семейства общепринятой является аббревиатура «LZ», но расшифровываться она должна как «Ziv-Lempel», поэтому и алгоритмы «Зива-Лемпела», а не «Лемпела-Зива». Согласно общепринятому объяснению этого курьеза, Якоб Зив внес больший вклад в открытие соответствующих словарных схем и исследование их свойств и таким образом заслужил, чтобы первым стояла его фамилия, что мы и видим в заголовках статей [12, 13]. Но случайно была допущена ошибка, и прикрепилось сокращение «LZ» (буквы упорядочены в алфавитном порядке). Иногда, кстати, встречается и обозначение «ZL» (порядок букв соответствует порядку фамилий авторов в публикациях [12, 13]). В дальнейшем, если некий исследователь существенно изменял какой-то алгоритм, относимый к семейству LZ, то в названии полученной модификации к строчке «LZ» обычно дописывалась первая буква его фамилии, например: алгоритм LZB, автор Белл (Bell).

Подчеркнем также наличие большой путаницы с классификацией алгоритмов. Обычно она проявляется в нежелании признавать существование двух самостоятельных семейств LZ, а также в неправильном отнесении алгоритмов к конкретному семейству. Беспорядку часто способствуют сами разработчики: многим невыгодно раскрывать, на основе какого алгоритма создана данная модификация из-за коммерческих, патентных или иных меркантильных соображений. Например, в случае коммерческого программного обеспечения общепринятой является практика классификации используемого алгоритма сжатия как «модификации LZ77». И в этом нет ничего удивительного, ведь алгоритм LZ77 не запатентован.

Алгоритм LZ77

Этот словарный алгоритм сжатия является самым старым среди методов LZ. Описание было опубликовано в 1977 году [12], но сам алгоритм разработан не позднее 1975 года.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Алгоритм LZ77 является «родоначальником» целого семейства словарных схем — так называемых алгоритмов со скользящим словарем, или скользящим окном. Действительно, в LZ77 в качестве словаря используется блок уже закодированной последовательности. Как правило, по мере выполнения обработки положение этого блока относительно начала последовательности постоянно меняется, словарь «скользит» по входному потоку данных.

Скользящее окно имеет длину N, т.е. в него помещается N символов, и состоит из 2 частей:

- последовательности длины $W = N–n$ уже закодированных символов, которая и является словарем;
- упреждающего буфера, или буфера предварительного просмотра (lookahead), длины n; обычно n на порядки меньше W.

Пусть к текущему моменту времени мы уже закодировали t символов s_1, s_2, …, s_t. Тогда словарем будут являться W предшествующих символов $s_{t-(W-1)}$, $s_{t-(W-1)+1}$, …, s_t. Соответственно, в буфере находятся ожидающие кодирования символы s_{t+1}, s_{t+2}, …, s_{t+n}. Очевидно, что если $W \geq t$, то словарем будет являться вся уже обработанная часть входной последовательности.

Идея алгоритма заключается в поиске самого длинного совпадения между строкой буфера, начинающейся с символа s_{t+1}, и всеми фразами словаря. Эти фразы могут начинаться с любого символа $s_{t-(W-1)}$, $s_{t-(W-1)+1}$, …, s_t и выходить за пределы словаря, вторгаясь в область буфера, но должны лежать в окне. Следовательно, фразы не могут начинаться с s_{t+1}, поэтому буфер не может сравниваться сам с собой. Длина совпадения не должна превышать размер буфера. Полученная в результате поиска фраза $s_{t-(i-1)}$, $s_{t-(i-1)+1}$, …, $s_{t-(i-1)+(j-1)}$ кодируется с помощью двух чисел:

1) смещения (offset) от начала буфера, i;
2) длины соответствия, или совпадения (match length), j.

Смещение и длина соответствия играют роль указателя (ссылки), однозначно определяющего фразу. Дополнительно в вы-

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

ходной поток записывается символ *s*, непосредственно следующий за совпавшей строкой буфера.

Таким образом, на каждом шаге кодер выдает описание трех объектов: смещения и длины соответствия, образующих код фразы, равной обработанной строке буфера, и одного символа *s* (литерала). Затем *окно* смещается на *j*+1 символов вправо и осуществляется переход к новому циклу кодирования. Величина сдвига объясняется тем, что мы реально закодировали именно *j*+1 символов: *j* с помощью указателя на фразу в словаре, и 1 с помощью тривиального копирования. Передача одного символа в явном виде позволяет разрешить проблему обработки еще ни разу не виденных символов, но существенно увеличивает размер сжатого блока.

Пример

Попробуем сжать строку «кот_ломом_колол_слона» длиной 21 символ. Пусть длина буфера равна 7 символам, а размер словаря больше длины сжимаемой строки. Условимся также, что:

- нулевое смещение зарезервировали для обозначения конца кодирования;
- символ s_t соответствует единичному смещению относительно символа s_{t+1}, с которого начинается буфер;
- если имеется несколько фраз с одинаковой длиной совпадения, то выбираем ближайшую к буферу;
- в неопределенных ситуациях — когда длина совпадения нулевая — смещению присваиваем единичное значение.

Таблица 3.1

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные		
	Словарь	Буфер		i	j	s
1	-	кот_лом	-	1	0	‘к’
2	к	от_ломо	-	1	0	‘о’
3	ко	т_ломом	-	1	0	‘т’
4	кот	_ломом_	-	1	0	‘_’

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные		
	Словарь	Буфер		i	j	s
5	кот_	ломом_к	-	1	0	‘л’
6	кот_л	омом_ко	о	4	1	‘м’
7	кот_лом	ом_коло	ом	2	2	‘_’
8	кот_ломом_	колол_с	ко	10	2	‘л’
9	кот_ломом_кол	ол_слон	ол	2	2	‘_’
10	..._ломом_колол_	слона	-	1	0	‘с’
11	...ломом_колол_с	лона	ло	5	2	‘н’
12	...ом_колол_слон	а	-	1	0	‘а’

Для кодирования *i* нам достаточно 5 битов, для *j* нужно 3 бита, и пусть символы требуют 1 байта для своего представления. Тогда всего мы потратим 12·(5+3+8) = 192 бита. Исходно строка занимала 21·8 = 168 битов, т.е. LZ77 кодирует нашу строку еще более расточительным образом. Не следует также забывать, что мы опустили шаг кодирования конца последовательности, который потребовал бы еще как минимум 5 битов (размер поля *i* = 5 битам).

Процесс кодирования можно описать следующим образом.

```
while ( ! DataFile.EOF() ){
   /*найдем максимальное совпадение; в match_pos получим
      смещение i, в match_len – длину j, в
      unmatched_sym – первый несовпавший символ s_{t+1+j};
      считаем также, что в функции find_match учитывается
      ограничение на длину совпадения
   */
   find_match (&match_pos, &match_len, &unmatched_sym);
   /*запишем в файл сжатых данных описание найденной
      фразы, при этом длина битового представления i
      задается константой OFFS_LN, длина представления
      j – константой LEN_LN, размер символа s принимаем
      равным 8 битам
   */
   CompressedFile.WriteBits (match_pos, OFFS_LN);
   CompressedFile.WriteBits (match_len, LEN_LN);
```

```
   CompressedFile.WriteBits (unmatched_sym, 8);
   for (i = 0; i <= match_len; i++){
      // прочтем очередной символ
      c = DataFile.ReadSymbol();
      //удалим из словаря одну самую старую фразу
      DeletePhrase ();
      /*добавим в словарь одну фразу, начинающуюся с
        первого символа буфера
      */
      AddPhrase ();
      /*сдвинем окно на 1 позицию, добавим в конец буфера
        символ c
      */
      MoveWindow(c);
   }
}
CompressedFile.WriteBits (0, OFFS_LN);
```

Пример подтвердил, что способ формирования кодов в LZ77 неэффективен и позволяет сжимать только сравнительно длинные последовательности. До некоторой степени сжатие небольших файлов можно улучшить, используя коды переменной длины для смещения *i*. Действительно, даже если мы используем словарь в 32 кбайт, но закодировали еще только 3 кбайт, то смещение реально требует не 15, а 12 битов. Кроме того, происходит существенный проигрыш из-за использования кодов одинаковой длины при указании длин совпадения *j*. Например, для уже упоминавшейся электронной версии романа «Бесы» были получены следующие частоты использования длин совпадения:

j	Количество раз, когда максимальная длина совпадения была равна *j*
0	136
1	1593
2	4675
3	11165
4	20047
5	26939
6	28653
7	24725

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

8	19702
9	14767
10	10820
≥11	27903

Из таблицы следует, что в целях минимизации закодированного представления для $j = 6$ следует использовать код наименьшей длины, так как эта длина совпадения встречается чаще всего.

Хотя авторы алгоритма и доказали, что LZ77 может сжать данные не хуже, чем любой специально на них настроенный полуадаптивный словарный метод, из-за указанных недостатков это выполняется только для последовательностей достаточно большого размера.

Что касается декодирования сжатых данных, то оно осуществляется путем простой замены кода на блок символов, состоящий из фразы словаря и явно передаваемого символа. Естественно, декодер должен выполнять те же действия по изменению окна, что и кодер. Фраза словаря элементарно определяется по смещению и длине, поэтому важным свойством LZ77 и прочих алгоритмов со скользящим окном является очень быстрая работа декодера.

Алгоритм декодирования может иметь следующий вид.

```
for (;;) {
   // читаем смещение
   match_pos = CompressedFile.ReadBits (OFFS_LN);
   if (!match_pos)
      // обнаружен признак конца файла, выходим из цикла
      break;
   // читаем длину совпадения
   match_len = CompressedFile.ReadBits (LEN_LN);
   for (i = 0; i < match_len; i++) {
      /*находим в словаре очередной символ совпавшей
        фразы
      */
      c = Dict (match_pos + i);
      /*сдвигаем словарь на 1 позицию, добавляем в его
        начало с
      */
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
      MoveDict (c)
      /*записываем очередной раскодированный символ в
        выходной файл
      */
      DataFile.WriteSymbol (c);
   }
   /*читаем несовпавший символ, добавляем его в словарь и
     записываем в выходной файл
   */
   c = CompressedFile.ReadBits (8);
   MoveDict (c)
   DataFile.WriteSymbol (c);
}
```

Алгоритмы со скользящим окном характеризуются сильной несимметричностью по времени — кодирование значительно медленнее декодирования, поскольку при сжатии много времени тратится на поиск фраз.

> **Упражнение:** Предложите несколько более эффективных способов кодирования результатов работы LZ77, чем использование простых кодов фиксированной длины.

Алгоритм LZSS

Алгоритм LZSS позволяет достаточно гибко сочетать в выходной последовательности символы и указатели (коды фраз), что до некоторой степени устраняет присущую LZ77 расточительность, проявляющуюся в регулярной передаче одного символа в прямом виде. Эта модификация LZ77 была предложена в 1982 году Сторером (Storer) и Жимански (Szymanski) [10].

Идея алгоритма заключается в добавление к каждому указателю и символу однобитового префикса *f*, позволяющего различать эти объекты. Иначе говоря, однобитовый флаг *f* указывает тип и, соответственно, длину непосредственно следующих за ним данных. Такая техника позволяет:

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

- записывать символы в явном виде, когда соответствующий им код имеет бо́льшую длину, и, следовательно, словарное кодирование только вредит;
- обрабатывать ни разу не встреченные до текущего момента символы.

Пример

Закодируем строку «кот_ломом_колол_слона» из предыдущего примера и сравним коэффициент сжатия для LZ77 и LZSS.

Пусть мы переписываем символ в явном виде, если текущая длина максимального совпадения буфера и какой-то фразы словаря меньше или равна 1. Если мы записываем символ, то перед ним выдаем флаг со значением 0, если указатель — то со значением 1. Если имеется несколько совпадающих фраз одинаковой длины, то выбираем ближайшую к буферу.

Таблица 3.2

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные			
	Словарь	Буфер		f	i	j	s
1	-	кот_лом	-	0	-	-	‘к’
2	к	от_ломо	-	0	-	-	‘о’
3	ко	т_ломом	-	0	-	-	‘т’
4	кот	_ломом_	-	0	-	-	‘_’
5	кот_	ломом_к	-	0	-	-	‘л’
6	кот_л	омом_ко	о	0	-	-	‘о’
7	кот_ло	мом_кол	-	0	-	-	‘м’
8	кот_лом	ом_коло	ом	1	2	2	-
9	кот_ломом	_колол_	_	0	-	-	‘_’
10	кот_ломом_	колол_с	ко	1	10	2	-
11	кот_ломом_ко	лол_сло	ло	1	8	2	-
12	...от_ломом_коло	л_слона	л	0	-	-	‘л’
13	...т_ломом_колол	_слона	_	0	-	-	‘_’
14	..._ломом_колол_	слона	-	0	-	-	‘с’
15	...ломом_колол_с	лона	ло	1	5	2	-

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные			
	Словарь	Буфер		f	i	j	s
16	...мом_колол_сло	на	-	0	-	-	‘н’
17	...ом_колол_слон	а	-	0	-	-	‘а’

Таким образом, для кодирования строки по алгоритму LZSS нам потребовалось 17 шагов: 13 раз символы были переданы в явном виде, и 4 раза мы применили указатели. Заметим, что при работе по алгоритму LZ77 нам потребовалось всего лишь 12 шагов. С другой стороны, если задаться теми же длинами для *i* и *j*, то размер закодированных по LZSS данных равен 13·(1+8) + 4·(1+5+3) = 153 битам. Это означает, что строка действительно была сжата, так как ее исходный размер 168 битов.

Рассмотрим алгоритм сжатия подробнее.

```
const int
   // порог для включения словарного кодирования
   THRESHOLD = 2,
   // размер представления смещения, в битах
   OFFS_LN = 14,
   // размер представления длины совпадения, в битах
   LEN_LN = 4;
const int
   WIN_SIZE = (1 << OFFS_LN), // размер окна
   BUF_SIZE = (1 << LEN_LN) – 1; // размер буфера
//функция вычисления реального положения символа в окне
inline int MOD (int i) { return i & (WIN_SIZE-1); };

...
//собственно алгоритм сжатия
int buf_sz = BUF_SIZE;
/* инициализация: заполнение буфера, поиск совпадения
    для первого шага
*/
...
while ( buf_sz ) {
   if ( match_len > buf_sz) match_len = buf_sz;
   if ( match_len < THRESHOLD ) {
      /*если длина совпадения меньше порога (2 в
```

```
      примере), то запишем в файл сжатых данных флаг и
      символ;
      pos определяет позицию начала буфера
   */
   CompressedFile.WriteBit (0);
   CompressedFile.WriteBits (window [pos], 8);
   // это понадобится при обновлении словаря
   match_len = 1;
}else{
   /*иначе запишем флаг и указатель, состоящий из
      смещения и длины совпадения
   */
   CompressedFile.WriteBit (1);
   CompressedFile.WriteBits (match_offs, OFFS_LN);
   CompressedFile.WriteBits (match_len, LEN_LN);
}
for (int i = 0; i < match_len; i++) {
   /*удалим из словаря фразу, начинающуюся в позиции
      MOD (pos+buf_sz)
   */
   DeletePhrase ( MOD (pos+buf_sz) );
   if ( (c = DataFile.ReadSymbol ()) == EOF)
      // мы в конце файла, надо сократить буфер
      buf_sz--;
   else
      /*иначе надо добавить в конец буфера новый
         символ
      */
      window [MOD (pos+buf_sz)] = c;
   pos = MOD (pos+1); // сдвиг окна на 1 символ
   if (buf_sz)
      /*если буфер не пуст, то добавим в словарь новую
         фразу, начинающуюся в позиции pos;
         считаем, что в функции AddPhrase одновременно
         выполняется поиск максимального совпадения
         между буфером и фразами словаря
      */
      AddPhrase (pos, &match_offs, &match_len)
   ;
}
}
CompressedFile.WriteBit (1);
CompressedFile.WriteBits (0, OFFS_LN); // знак конца
файла
```

Скользящее окно можно реализовывать с помощью «циклического» массива, что и было проделано в вышеприведенном

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

учебном фрагменте программы сжатия. Использованный подход не является лучшим, но сравнительно прост для понимания.

Алгоритм декодирования может быть реализован следующим образом.

```
for (;;) {
   if ( !CompressedFile.ReadBit () ){
      /*это символ, просто выведем его в файл и запишем в
        конец словаря (символ будет соответствовать сме-
        щению i = 1)
      */
      c = CompressedFile.ReadBits (8);
      DataFile.WriteSymbol (c);
      window [pos] = c;
      pos = MOD (pos+1);
   }else{
      // это указатель, прочитаем его
      match_pos = CompressedFile.ReadBits (OFFS_LN);
      if (!match_pos)
         break; // конец файла
      match_pos = MOD (pos - match_pos);
      match_len = CompressedFile.ReadBits (LEN_LN);
      // цикл копирования совпавшей фразы словаря в файл
      for (int i = 0; i < match_len; i++) {
         //выдаем очередной совпавший символ c
         c = window [MOD (match_pos+i)];
         DataFile.WriteSymbol (c);
         window [pos] = c;
         pos = MOD (pos+1);
      }
   }
}
```

Упражнение: Из-за наличия порога THRESHOLD часть допустимых значений длины реально не используется, поэтому размер буфера BUF_SIZE может быть увеличен при неизменном LEN_LN. Проделайте соответствующие модификации фрагментов программ кодирования и декодирования.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Алгоритм LZ78

Алгоритм LZ78 был опубликован в 1978 году [13], и впоследствии стал «отцом» семейства словарных методов LZ78.

Алгоритмы этой группы не используют скользящего окна и в словарь помещают не все встречаемые при кодировании строки, а лишь «перспективные» с точки зрения вероятности последующего использования. На каждом шаге в словарь вставляется новая фраза, которая представляет собой сцепление (конкатенацию) одной из фраз *S* словаря, имеющей самое длинное совпадение со строкой буфера, и символа *s*. Символ *s* является символом, следующим за строкой буфера, для которой найдена совпадающая фраза *S*. В отличие от семейства LZ77, в словаре не может быть одинаковых фраз.

Кодер порождает только последовательность кодов фраз. Каждый код состоит из номера (индекса) *n* «родительской» фразы *S*, или префикса, и символа *s*.

В начале обработки словарь пуст. Далее, теоретически, словарь может расти бесконечно, т.е. на его рост сам алгоритм не налагает ограничений. На практике при достижении определенного объема занимаемой памяти словарь должен очищаться полностью или частично.

Пример

И еще раз закодируем строку «кот_ломом_колол_слона» длиной 21 символ. Для LZ78 буфер, в принципе, не требуется, поскольку достаточно легко так реализовать поиск совпадающей фразы максимальной длины, что последовательность незакодированных символов будет просматриваться только один раз. Поэтому буфер показан только с целью большей доходчивости примера. Фразу с номером 0 зарезервируем для обозначения конца сжатой строки, номером 1 будем задавать пустую фразу словаря.

Таблица 3.3

Шаг	Добавляемая в словарь фраза		Буфер	Совпадающая фраза S	Закодированные данные	
	сама фраза	ее номер			n	s
1	к	2	кот_лом	-	1	‘к’
2	о	3	от_ломо	-	1	‘о’
3	т	4	т_ломом	-	1	‘т’
4	_	5	_ломом_	-	1	‘_’
5	л	6	ломом_к	-	1	‘л’
6	ом	7	омом_ко	о	3	‘м’
7	ом_	8	ом_коло	ом	7	‘_’
8	ко	9	колол_с	к	2	‘о’
9	ло	10	лол_сло	л	6	‘о’
10	л_	11	л_слона	л	6	‘_’
11	с	12	слона	-	1	‘с’
12	лон	13	лона	ло	10	‘н’
13	а	14	а	-	1	‘а’

Строку удалось закодировать за 13 шагов. Так как на каждом шаге выдавался один код, сжатая последовательность состоит из 13 кодов. Возможно использование 15 номеров фраз (от 0 до 14), поэтому для представления *n* посредством кодов фиксированной длины нам потребуется 4 бита. Тогда размер сжатой строки равен 13·(4+8) = 156 битам.

Ниже приведен пример реализации алгоритма сжатия LZ78.

```
n = 1;
while ( ! DataFile.EOF() ){
   s = DataFile.ReadSymbol; // читаем очередной символ
   /*пытаемся найти в словаре фразу, представляющую
      собой конкатенацию родительской фразы с номером n и
      символа s; функция возвращает номер искомой фразы
      в phrase_num; если же фразы нет, то phrase_num
      принимает значение 1, т.е. указывает на пустую
      фразу
```

```
   */
   FindPhrase (&phrase_num, n, s);
   if (phrase_num != 1)
      /*такая фраза имеется в словаре, продолжим поиск
        совпадающей фразы максимальной длины
      */
      n = phrase_num;
   else {
      /*такой фразы нет, запишем в выходной файл код;
        INDEX_LN – это константа, определяющая длину
        битового представления номера n
      */
      CompressedFile.WriteBits (n, INDEX_LN);
      CompressedFile.WriteBits (s, 8);
      AddPhrase (n, s); // добавим фразу в словарь
      n = 1; // подготовимся к следующему шагу
   }
}
// признак конца файла
CompressedFile.WriteBits (0, INDEX_LN);
```

При декодировании необходимо обеспечивать такой же порядок обновления словаря, что и при сжатии. Реализуем алгоритм следующим образом.

```
for (;;){
   // читаем индекс родительской фразы
   n = CompressedFile.ReadBits (INDEX_LN);
   if (!n)
      break; // конец файла
   // читаем несовпавший символ s
   s = CompressedFile.ReadBits (8);
   /*находим в словаре позицию начала фразы с индексом n
      и ее длину
   */
   GetPhrase (&pos, &len, n)
   /*записываем фразу с индексом n в файл
      раскодированных данных
   */
   for (i = 0; i < len; i++)
      DataFile.WriteSymbol (Dict[pos+i]);
   // записываем в файл символ s
   DataFile.WriteSymbol (s);
   AddPhrase (n, s); // добавляем новую фразу в словарь
}
```

Очевидно, что скорость раскодирования для алгоритмов семейства LZ78 потенциально всегда меньше скорости для алгоритмов со скользящим окном, т.к. в случае последних затраты по поддержанию словаря в правильном состоянии минимальны. С другой стороны, для LZ78 и его потомков, например LZW, существуют эффективные реализации процедур поиска и добавления фраз в словарь, что обеспечивает значительное преимущество над алгоритмами семейства LZ77 в скорости сжатия.

Несмотря на относительную быстроту кодирования LZ78, при грамотной реализации алгоритма оно все же медленнее декодирования, соотношение скоростей равно обычно 3:2.

Интересное свойство LZ78 заключается в том, что если исходные данные порождены источником с определенными характеристиками (он должен быть стационарным[2] и эргодическим[3]), то коэффициент сжатия приближается по мере кодирования к минимальному достижимому [13]. Иначе говоря, количество битов, затрачиваемых на кодирование каждого символа, в среднем равно так называемой энтропии источника. Но, к сожалению, сходимость медленная, и на данных реальной длины алгоритм ведет себя не лучшим образом. Так, например, коэффициент сжатия текстов в зависимости от их размера обычно колеблется от 3.5 до 5 битов/символ. Кроме того, нередки ситуации, когда обрабатываемые данные порождены источником с ярко выраженной нестационарностью. Поэтому при оценке реального поведения алгоритма следует относиться с большой

---

[2] Многомерные распределения вероятностей генерации последовательностей (слов) из n символов не меняются во времени, причем n — любое конечное число

[3] Среднее по времени равно среднему по числу реализаций; иначе говоря, для оценки свойств источника достаточно только одной длинной сгенерированной последовательности

осторожностью к теоретическим выкладкам, обращая внимание на выполнение соответствующих условий.

Доказано, что аналогичным свойством сходимости обладает и классический алгоритм LZ77, но скорость приближения к энтропии источника меньше, чем у алгоритма LZ78 [12].

Другие алгоритмы LZ

В табл. 3.4 приведены несколько достаточно характерных алгоритмов LZ. Указанный список далеко не полон, реально существует в несколько раз больше алгоритмов, основанных либо на LZ77, либо на LZ78. Известны и гибридные схемы, сочетающие оба подхода к построению словаря.

Таблица 3.4

№	Название	Авторы, год	Тип алгоритма
1	LZMV	Миллер (Miller) и Уэгнам (Wegman), 1984	Алгоритм семейства LZ78
2	LZW	Уэлч (Welch), 1984	Модификация LZ78
3	LZB	Белл (Bell), 1987	Модификация LZSS
4	LZH	Брент (Brent), 1987	Модификация LZSS
5	LZFG	Файэлэ (Fiala) и Грини (Greene), 1989	Модификация LZ77
6	LZBW	Бендер (Bender) и Вулф (Wolf), 1991	Способ модификация алгоритмов семейства LZ77
7	LZRW1	Уилльямс (Williams), 1991	Модификация LZSS
8	LZCB	Блум (Bloom), 1995	Модификация LZ77
9	LZP	Блум (Bloom), 1995	Основан на LZ77

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

№	Название	Авторы, год	Тип алгоритма
10	LZ77-PM, LZFG-PM, LZW-PM	Хоанг (Hoang), Лонг (Long) и Виттер (Vitter), 1995	Модификации алгоритмов LZ

1) LZMV

Алгоритм семейства LZ78. Интересен способом построения словаря: новая фраза создается с помощью конкатенации двух последних использованных фраз, а не конкатенации фразы и символа, т.е. словарь наполняется «агрессивнее». Практические реализации LZMV неизвестны. Причина, по-видимому, состоит в том, что усложнение алгоритма не приводит к адекватному улучшению сжатия по сравнению с исходным LZW. С другой стороны, выгода от быстрого заполнения и обновления словаря проявляется главным образом при обработке неоднородных данных. Но для сжатия данных такого типа заведомо лучше подходит метод скользящего словаря (семейство LZ77).

2) LZW

Модификация LZ78. За счет предварительного занесения в словарь всех символов алфавита входной последовательности результат работы LZW состоит только из последовательности индексов фраз словаря. Из-за устранения необходимости регулярной передачи одного символа в явном виде LZW обеспечивает лучшее сжатие, чем LZ78. Подробнее об LZW см. в разделе «Алгоритмы сжатия изображений», а также в [11].

3) LZB

Модификация LZSS. Изменения затрагивают кодирование указателей. Смещение задается переменным количеством битов в зависимости от реального размера словаря (в начале сжатия он мал). Длина совпадения записывается γ-кодами Элиаса. И первый, и второй механизмы часто применяются

при разработке простых и обеспечивающих высокую скорость алгоритмов семейства LZ77.

4) LZH

Модификация LZSS. Аналогична LZB, но для сжатия смещений и длин соответствия используются коды Хаффмана. Заметим, что большинство современных архиваторов также применяют кодирование по методу Хаффмана в этих целях.

5) LZFG

Модификация LZ77. На самом деле представляет собой несколько алгоритмов. Идея самого сложного (C2 в обозначении авторов LZFG) заключается в кодировании фразы не парой <длина, смещение>, а индексом соответствующего фразе узла дерева цифрового поиска. Одинаковые фразы словаря имеют один и тот же индекс, что и обеспечивает более экономное кодирование строк. Алгоритмы LZFG не получили распространения на практике. В значительной степени этому способствовали патентные ограничения.

6) LZBW

Способ модификации алгоритмов семейства LZ77. За счет учета имеющихся в словаре одинаковых фраз позволяет уменьшить количество битов, требуемых для кодирования длины совпадения. Подробнее см. в пункте «Пути улучшения сжатия алгоритмов LZ».

7) LZRW1

Алгоритм является модификацией LZSS, точнее, алгоритма A1 группы LZFG, и разработан с целью обеспечения максимальной скорости сжатия и разжатия. Коды фраз состоят из 16 битов: 12 битов указывают смещение *i* и 4 бита задают длину совпадения *j*, а символы *s* передаются как байты (требуют 8 битов). Флаги *f* задаются сразу для последовательности из 16 кодов и литералов, т.е. сначала выдается 2-байтовое слово значений флагов, затем группа из 16 кодов и/или литералов. Для поиска в словаре при сжатии используется хеш-

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

таблица со смешиванием по 3 символам. Хеш-цепочки как таковые отсутствуют, т.е. каждая новая фраза заменяет в таблице старую с таким же значением хеш-функции. За счет устранения побитового ввода-вывода и использования словаря малого размера достигается высокая скорость кодирования и декодирования. Степень сжатия LZRW1 равна примерно 1.5…2.

8) LZCB

По сути, целая группа алгоритмов, являющихся той или иной модификацией LZ77. Основная идея заключается во введении достаточно сильного ограничения на минимальную длину совпадения — от 4 символов и более. Если фраза удовлетворяющей длины не найдена, то кодируется один символ (литерал). Характер типичных данных таков, что литералы и успешно закодированные с помощью словаря строки имеют тенденцию объединяться в группы с себе подобными, т.е., например, на выходе LZCB могут появиться 10 литералов подряд, затем 5 закодированных строк, затем опять несколько литералов и т.д. Эта особенность позволяет реализовать достаточно эффективное статистическое кодирование потоков литералов и указателей фраз. Тем не менее, растеряв преимущество в скорости, LZCB уступает современным алгоритмам PPM по коэффициенту сжатия.

9) LZ77-PM, LZFG-PM, LZW-PM

Модификации алгоритмов LZ. Используется несколько контекстно-зависимых словарей, а не один словарь. В контекстно-зависимый словарь порядка L с номером i попадают только строки, встречаемые после контекста[4] C_i длины L. Кодирование строки буфера S производится с помощью одного

[4] Контекст — это в данном случае конечная последовательность символов. См. также главу «Сжатие данных с помощью контекстных методов моделирования»

или нескольких словарей, номера которых определяются последними закодированными перед *S* символами. Учет контекста позволяет существенно улучшить сжатие исходных словарных схем. Подробнее см. в пункте «Пути улучшения сжатия алгоритмов LZ».

10) LZP

Основан на LZ77. Для каждого входного символа строка из предшествующих *L* символов рассматривается как контекст *C* длины *N*. С помощью хеш-функции в словаре находится одна из совпадающих с контекстом *C* фраз, назовем эту фразу *C'*: *C'* = *C*. Строка *S* буфера сравнивается с фразой, непосредственно следующей за *C'*. Если длина совпадения $L > 0$, то выдается флаг успеха $f = 1$ и *S* кодируется через длину *L*. Так как *C'* находится детерминированным образом, то смещение кодировать не надо. Если $L = 0$, или *C'* не была найдена, то выдается $f = 0$ и первый символ *S* кодируется непосредственно. Декодер должен использовать такой же механизм для поиска *C'*. Изложенный алгоритм справедлив для алгоритма LZP1, достоинством которого является высокая скорость. В более сложных модификациях процесс поиска *C'* повторяется для контекста длины *N*–1 и т.д. В LZP1 литералы просто копируются, а для кодирования флагов и длин используются коды переменной длины. В LZP3 применяется достаточно сложная схема кодирования потоков флагов, длин и литералов на основе алгоритма Хаффмана. В сочетании с PPM техника LZP обеспечивает высокую степень сжатия при неплохих скоростных характеристиках.

В табл. 3.5 приведены результаты сравнения нескольких алгоритмов по степени сжатия на наборе CalgCC.

Таблица 3.5

	LZP1	LZSS	LZW	LZB	LZW-PM	LZFG	LZFG-PM	LZP3
Bib	1.98	2.81	2.48	2.52	3.07	2.76	3.28	3.32
Book1	1.42	2.33	2.52	2.07	2.92	2.21	2.44	2.50

Книга "Методы сжатия данных". ISBN 5-86404-170-X
http://www.compression.ru/

	LZP1	LZSS	LZW	LZB	LZW-PM	LZFG	LZFG-PM	LZP3
Book2	1.76	2.68	2.61	2.44	3.14	2.62	2.88	3.01
Geo	1.19	1.19	1.38	1.30	1.23	1.40	1.42	1.51
News	1.76	2.28	2.21	2.25	2.52	2.33	2.46	2.78
Obj1	1.55	1.57	1.58	1.88	1.56	1.99	1.85	1.83
Obj2	2.21	2.28	1.96	2.55	2.34	2.70	2.63	2.79
Paper1	1.72	2.47	2.21	2.48	2.60	2.64	2.92	2.73
Paper2	1.62	2.50	2.37	2.33	2.72	2.53	2.86	2.72
Pic	6.30	3.98	8.51	7.92	8.16	9.20	8.60	9.30
Progc	1.86	2.44	2.15	2.60	2.52	2.77	2.92	2.73
Progl	2.66	3.28	2.71	3.79	3.38	4.06	4.44	4.19
Progp	2.82	3.31	2.67	3.85	3.33	4.21	4.42	3.98
Trans	3.27	3.46	2.53	3.77	3.46	4.55	4.79	4.79
Итого	**2.29**	**2.61**	**2.71**	**2.98**	**3.07**	**3.28**	**3.42**	**3.44**

Видно, что за счет оптимизации структуры словаря достигается значительное улучшение сжатия. Но, с другой стороны, сложные алгоритмы построения словаря обычно существенно замедляют работу декодера, что сводит на нет достоинства LZ.

Приведенные ниже характеристики степени сжатия и скорости алгоритмов семейств LZ77 и LZ78 относятся к типичным представителям семейств — LZH и LZW соответственно.

Характеристики алгоритмов семейства LZ77:

Степени сжатия: определяются данными, обычно 2…4.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия разнородных данных, например, файлов ресурсов.

Симметричность по скорости: примерно 10:1; если алгоритм обеспечивает хорошее сжатие, то декодер обычно гораздо быстрее кодера.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Характерные особенности: обычно медленно сжимают высоко избыточные данные; из-за высокой скорости разжатия идеально подходят для создания дистрибутивов программного обеспечения.

Характеристики алгоритмов семейства LZ78:

Степени сжатия: определяются данными, обычно 2…3.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия текстов и тому подобных однородных данных, например, рисованных картинок; плохо сжимают разнородные данные.

Симметричность по скорости: примерно 3:2, декодер обычно в полтора раза быстрее кодера.

Характерные особенности: из-за относительно небольшой степени сжатия и невысокой скорости декодирования уступают по распространенности алгоритмам семейства LZ77.

Формат Deflate

Формат словарного сжатия Deflate, предложенный Кацем (Katz), используется популярным архиватором PKZIP [3]. Сжатие осуществляется с помощью алгоритма типа LZH, иначе говоря, указатели и литералы кодируются по методу Хаффмана. Формат специфицирует только работу декодера, т.е. определяет алгоритм декодирования, и не налагает серьезных ограничений на реализацию кодера. В принципе, в качестве алгоритма сжатия может применяться любой работающий со скользящим ок-

ном, лишь бы он исходил из стандартной процедуры обновления словаря для алгоритмов семейства LZ77 и использовал задаваемые форматом типы кодов Хаффмана.

Особенности формата:

- является универсальным, т.е. не ориентирован на конкретный тип данных;
- прост в реализации;
- де-факто является одним из промышленных стандартов на сжатие данных.

Существует множество патентов, покрывающих весь формат полностью или какие-то детали его реализации. С другой стороны, Deflate используется в огромном количестве программных и аппаратных приложений, и отработаны методы защиты в суде в случае предъявления соответствующего иска от компании, обладающей патентом на формат или его часть.

Поучительная ремарка. В 1994 году корпорация Unisys «вспомнила» о своем патенте в США на алгоритм LZW, зарегистрированном в 1985 году, и объявила о незаконности использования LZW без соответствующей лицензии. В частности, Unisys заявила о нарушении своих прав как патентовладельца в случае нелицензированного использования алгоритма LZW в формате GIF. Было объявлено, что производители, программы или аппаратное обеспечение которых читают или записывают файлы в формате GIF, должны покупать лицензию на использование, а также выплачивать определенный процент с прибыли при коммерческом применении. Далее Unisys в лучших традициях тоталитарной пропаганды последовательно продолжала переписывать историю задним числом, неоднократно меняя свои требования к лицензируемым продуктам и условия оплаты. В частности, было оговорено, что плата взимается только в случае коммерческих продуктов, но в любом случае требуется получить официальное разрешение от Unisys. Но, с другой стороны, Unisys требует у всех владельцев Internet (intranet) сайтов, использующих рисунки в формате GIF, приобрести лицензию стоимостью порядка $5000 в том случае, если ПО, с помощью которого были созданы эти файлы GIF, не имеет соответствующей лицензии Unisys. С точки зрения некоторых независимых экспертов по патентному праву, чтение (декодирование) GIF файлов не нарушает права Unisys, но, судя по всему, сама корпорация придерживается другой точки зрения. Также достаточно странно поведение корпорации CompuServ, разработавшей формат GIF и опубликовавшей его в 1987 году, т.е. уже после регистрации патента на LZW, как открытый и свободный от оплаты. По состоянию на 2001 год, LZW запатентован Unisys по меньшей мере в следующих странах: США,

Канада, Великобритания, Германия, Франция, Япония. Текущее состояние дел можно выяснить на сайте компании www.unisys.com. Срок действия основного патента в США истекает не ранее 19 июня 2003 года.

Общее описание

Закодированные в соответствии с форматом Deflate данные представляют собой набор блоков, порядок которых совпадает с последовательностью соответствующих блоков исходных данных. Используется три типа блоков закодированных данных:

1) состоящие из несжатых данных;
2) использующие фиксированные коды Хаффмана;
3) использующие динамические коды Хаффмана.

Длина блоков первого типа не может превышать 64 кбайт, относительно других ограничений по размеру нет. Каждый блок типа 2 и 3 состоит из двух частей:

- описания двух таблиц кодов Хаффмана, использованных для кодирования данных блока;
- собственно закодированных данных.

Коды Хаффмана каждого блока не зависят от использованных в предыдущих блоках.

Само описание динамически создаваемых кодов Хаффмана является, в свою очередь, также сжатым с помощью фиксированных кодов Хаффмана, таблица которых задается форматом.

Алгоритм словарного сжатия может использовать в качестве словаря часть предыдущего блока (блоков), но величина смещения не может быть больше 32 кбайт. Данные в компактном представлении состоят из кодов элементов двух типов:

- литералов (одиночных символов);
- указателей имеющихся в словаре фраз; указатели состоят из пары <длина совпадения, смещение>.

Длина совпавшей строки не может превышать 258 байтов, а смещение фразы — 32 кбайт. Литералы и длины совпадения кодируются с помощью одной таблицы кодов Хаффмана, а для смещений используется другая таблица; иначе говоря, литералы

и длины совпадения образуют один алфавит. Именно эти таблицы кодов и передаются в начале блока третьего типа.

Алгоритм декодирования

Сжатые данные декодируются по следующему алгоритму.

```
do{
   Block.ReadHeader (); // читаем заголовок блока
   /*определяем необходимые действия по разжатию в
      соответствии с типом блока
   */
   switch (Block.Type) {
   case NO_COMP:// данные не сжаты, просто копируем их
      /*заголовок блока не выровнен на границу байта,
         сделаем это
      */
      Block.SeekNextByte ();
      Block.ReadLen (); // читаем длину блока
      /*копируем данные блока из входного файла
         сжатых данных в результирующий DataFile
      */
      PutRawData (Window, Block, DataFile);
      break;
   case DYN_HUF:
      /*блок данных сжат с помощью динамически
         построенных кодов Хаффмана, прочитаем их
      */
      Block.ReadHuffmanCodes ();
   case FIXED_HUF:
      for (;;) {
         /*прочтем один символ алфавита литералов и
            длин совпадения
         */
         value = Block.DecodeSymbol ();
         if ( value < 256)
            // это литерал, запишем его в выходной файл
            DataFile.WriteSymbol (value);
         else if ( value == 256)
            // знак конца блока
            break;
         else {
            // это закодированный указатель
            match_len = Block.DecodeLen ();
            match_pos = Block.DecodePos ();
            /*скопируем соответствующую фразу из словаря
               в выходной файл
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
            */
            CopyPhrase (Window, match_len, match_pos,
                        DataFile);
         }
      };
      break;
   default:
      // Ошибка в блоке данных
      throw BadData (Block);
}while ( !IsLastBlock );
```

Предполагается, что используемые алгоритмы поддерживают правильное обновление скользящего окна.

Кодирование длин и смещений

Как уже указывалось, литералы и длины совпадения объединены в единый алфавит символов со значениями {0, 1, …, 285}, так что 0…255 отведены под литералы, 256 указывает на конец текущего блока, а 257…285 определяют длины совпадения. Код длины совпадения состоит из кода числа, лежащего в диапазоне 257…285 (базы длины совпадения), и, возможно, дополнительно читаемых битов, непосредственно следующих за кодом этого числа. База определяет квантованную длину совпадения, поэтому одному значению базы может соответствовать несколько длин. Значение поля дополнительно читаемых битов используется для доопределения длины совпадения. Таблица отображения значений кодов в длины фраз приведена ниже.

Таблица 3.6

Значение базы	Число доп. битов	Длина совпадения	Значение базы	Число доп. битов	Длина совпадения
257	0	3	272	2	31…34
258	0	4	273	3	35…42
259	0	5	274	3	43…50
260	0	6	275	3	51…58
261	0	7	276	3	59…66
262	0	8	277	4	67…82

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Значение базы	Число доп. битов	Длина совпадения	Значение базы	Число доп. битов	Длина совпадения
263	0	9	278	4	83…98
264	0	10	279	4	99…114
265	1	11,12	280	4	115…130
266	1	13,14	281	5	131…162
267	1	15,16	282	5	163…194
268	1	17,18	283	5	195…226
269	2	19…22	284	5	227…257
270	2	23…26	285	0	258
271	2	27…30			

Поле дополнительных битов представляет собой целое число заданной длины, в котором первым записан самый старший бит. Длина совпадения 258 кодируется с помощью небольшого числа битов, поскольку это максимально допустимая в Deflate длина совпадения, и такой прием позволяет увеличить степень сжатия высокоизбыточных файлов.

Таким образом, функция `Block.DecodeSymbol`, упомянутая в предыдущем подпункте, читает символ, который может быть либо литералом, либо знаком конца блока, либо базой совпадения. В последнем случае, т.е. когда значение символа > 256, дополнительные биты читаются с помощью функции `Block.DecodeLen`.

Представление смещений также состоит из двух полей: базы и поля дополнительных битов (табл. 3.7). Объединение смещений в группы позволяет использовать достаточно эффективные коды Хаффмана небольшой длины, канонический алгоритм построения которых обеспечивает быстрое декодирование. Объединение целесообразно также потому, что распределение величин смещений в отдельной группе обычно носит случайный характер.

Таблица 3.7

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Значение базы	Число доп. битов	Значение смещения	Значение базы	Число доп. битов	Значение смещения
0	0	1	15	6	193…256
1	0	2	16	7	257…384
2	0	3	17	7	385…512
3	0	4	18	8	513…768
4	1	5,6	19	8	769…1024
5	1	7,8	20	9	1025…1536
6	2	9…12	21	9	1537…2048
7	2	13…16	22	10	2049…3072
8	3	17…24	23	10	3073…4096
9	3	25…32	24	11	4097…6144
10	4	33…48	25	11	6145…8192
11	4	49…64	26	12	8193…12288
12	5	65…96	27	12	12289…16384
13	5	97…128	28	13	16385…24576
14	6	129…192	29	13	24577…32768

Функция `Block.DecodePos` должна выполнить два действия: прочитать базу смещения и, на основании значения базы, прочитать необходимое количество дополнительных битов. Как и в случае литералов/длин совпадения, дополнительные биты соответствуют целым числам заданной длины, в которых первым записан самый старший бит.

Кодирование блоков фиксированными кодами Хаффмана

В этом случае для сжатия символов алфавита литералов и длин совпадения используются заранее построенные коды Хаффмана, известные кодеру и декодеру, т.е. нам не нужно передавать их описания. Длины кодов определяются значением символов (см. табл. 3.8).

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Таблица 3.8

Значение символа	Длина кода в битах	Значение кода (в двоичной системе счисления)
0–143	8	00110000 … 10111111
144–255	9	110010000 … 111111111
256–279	7	0000000 … 0010111
280–287	8	11000000 … 11000111

Символы со значениями 286 и 287 не должны появляться в сжатых данных, хотя для них и отведено кодовое пространство.

Базы смещений кодируются 5-битовыми числами так, что 00000 соответствует 0, 11111 — 31. При этом запрещается использовать базы со значениями 30 и 31, так как их смысл не определен.

Пример

Покажем, как выглядит в закодированном виде следующая последовательность замещенных строк и литерала:

Элемент	Смещение i	Длина совпадения j	Литерал
1	260	5	-
2	45	20	-
3	-	-	3

Длина совпадения раскладывается на 2 поля, поэтому для j = 5 получаем (см. табл. 3.6):

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Длина совпадения	Соответствующий символ в алфавите литералов/длин	База	Число дополнительных битов
5	259	259	0

Длина совпадения кодируется как 0000011 (см. табл. 3.8).

В общем случае смещение состоит также из 2 полей, и для i = 260 получаем:

Смещение	База	Число дополнительных битов	Значение дополнительных битов
260	16	7	3

Смещение 260 записывается как последовательность битов 10000_0000011 (подчеркиванием показана граница чисел), где первое число — база, второе — поле дополнительных битов, равное 260 – 257 = 3.

Действуя аналогичным образом, на основании табл. 3.6, 3.7, 3.8 находим отображение всей последовательности:

Элемент	Последовательность битов	Комментарии
1	0000011 10000_0000011	• база длины совпадения = 259 • поле дополнительных битов совпадения отсутствует • база смещения = 16 • поле дополнительных битов смещения = 3, длина поля = 7
2	0001101_01 01010_1100	• база длины совпадения = 269 • поле дополнительных битов совпадения = 1, длина поля = 2 • база смещения = 10 • поле дополнительных битов смещения = 12, длина поля = 4
3	00110011	литерал = 3

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Кодирование блоков динамически создаваемыми кодами Хаффмана

В этом случае перед собственно сжатыми данными передается описание кодов литералов/длин и описание кодов смещений. В методе Deflate используется канонический алгоритм Хаффмана, поэтому для указания кода символа достаточно задать только длину этого кода. Неявным параметром является положение символа в упорядоченном списке символов. Таким образом, динамические коды Хаффмана описываются цепочкой длин кодов, упорядоченных по значению соответствующего кодам числа (литерала/длины совпадения, в одном случае, и смещения, в другом). При этом алфавит *CWL* (codeword lengths) длин кодов имеет вид, описанный в табл. 3.9.

Таблица 3.9

Значение символа алфавита CWL	Что определяет
0…15	Соответствует длинам кодов от 0 до 15
16	Копировать предыдущую длину кода x раз, где x определяется значением двух битов, читаемых после кода символа ‘16’; можно указать необходимость повторить предыдущую длину 3…6 раз
17	Позволяет задать для x кодов, начиная с текущего, длину 0; x может принимать значения от 3 до 10 и определяется таким же образом, как и для 16
18	Аналогично 17, но x может быть от 11 до 138

Например, если в результате декодирования описания кодов была получена цепочка длин ‘2’, ‘3’, ‘16’ (x = 4), ‘17’ (x=3), ‘4’,…, то длина кодов символов будет равна:

Значение символа	0	1	2	3	4	5	6	7	8	9	…
Длина кода символа	2	3	3	3	3	3	0	0	0	4	?

Обратите внимание, что символы упорядочены по возрастанию их значений.

С целью увеличения сжатия сами эти цепочки длин кодируются с помощью кодов Хаффмана. Если длина кода символа (т.е. длина кода длин кодов) равна нулю, то это значит, что соответствующий символ в данных не встречается, и код ему не отводится.

Формат блока с динамическими кодами Хаффмана описан в табл. 3.10.

Таблица 3.10

Поле	Описание	Размер
HLIT	Хранит количество кодов литералов/длин минус 257	5 битов
HDIST	Хранит количество кодов смещений минус 1	5 битов
HCLEN	Хранит количество кодов длин кодов минус 4	4 бита
Таблица описания кодов длин кодов (коды 2)	Содержит описание (последовательность длин кодов) длин кодов в следующем порядке: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. Иначе говоря, это порядок символов в алфавите CWL. Длина кода любого символа CWL задается с помощью 3 битов; таким образом, длина кода длины кода может быть от 0 (соответствующий символ из CWL не используется) до $2^3 - 1 = 7$ битов. Длины однозначно задают совокупность кодов	(HCLEN+4)·3 бита
Таблица описания кодов литералов/длин (коды 1)	Содержит описание HLIT+257 кодов литералов/длин совпадения, закодировано кодами длин кодов	Переменный

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Поле	Описание	Размер
Таблица описания кодов смещений	Содержит описание HDIST+1 кодов смещений, закодировано кодами длин кодов	Переменный
Сжатые данные	Содержит данные, сжатые с помощью двух заданных выше совокупностей кодов	Переменный
Знак конца блока	Число 256, сжатое с помощью кодов литералов/длин	Переменный

На основании вышесказанного можно дать такое описание алгоритма кодирования литералов/длин:

- литералы/длины совпадения кодируются с помощью кодов Хаффмана для литералов/длин, назовем их кодами 1;
- описание кодов 1 передается в виде цепочки длин этих кодов;
- при указании длин кодов могут использоваться специальные символы (см. алфавит *CWL*);
- длины кодов, т.е. символы алфавита *CWL*, кодируются с помощью кодов Хаффмана для длин кодов, назовем их кодами 2;
- коды 2 также описываются через последовательность их длин;
- длины кодов 2 задаются с помощью 3-битовых чисел.

Для кодирования смещений используется такой же подход, при этом длины кодов смещений также сжимаются с помощью кодов 2.

Упражнение: Объясните, почему размеры полей, хранящих величины HLIT, HDIST и HCLEN, именно таковы, как это указано в табл. 3.10.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Алгоритм словарного сжатия для Deflate

Как уже указывалось, формат Deflate не имеет четкой спецификации алгоритма словарного сжатия. Разработчики могут использовать какие-то свои алгоритмы, подходящие для решения специфических задач.

Рассмотрим свободный от патентов алгоритм сжатия для Deflate, используемый в разрабатываемой Info-ZIP group утилите Zip.

Для поиска фраз используется метод хеш-цепочек. Хеш-функция вычисляется на основании трех байтов данных (напомним, что формат Deflate не позволяет кодировать строки длиной менее трех байтов). Функция может принимать значения от 0 до заданного числа HASH_MASK – 1 и имеет вид выражения, последовательно вычисляемого для каждого очередного символа:

```
int UPDATE_HASH (int h, char c) {
   return ( (h<<H_SHIFT) ^ c ) & HASH_MASK;
}
```

где h — текущее значение хеш-функции;
c — очередной символ;
H_SHIFT — параметр сдвига значения функции.

H_SHIFT назначается таким образом, чтобы после очередного сдвига значение самого старого байта не влияло на значение хеш-функции.

На каждом шаге компрессор читает очередную 3-байтовую строку, располагающуюся в начале буфера. После соответствующего обновления хеш-функции производится обращение к первому элементу хеш-цепочки, адрес которого определяется значением функции. Если цепочка пуста, то кодируется литерал, производится сдвиг окна на 1 символ и осуществляется переход к следующему шагу. Иначе хеш-цепочка анализируется с целью найти самое длинное совпадение между буфером и фразами, на которые ссылаются элементы (узлы) хеш-цепочки. Обновление хеш-цепочек организовано так, что поиск начинается с самых «новых» узлов, что позволяет сместить распределение частот смещений кодируемых фраз в пользу коротких смещений и,

следовательно, улучшить сжатие, т.к. небольшие смещения имеют коды малой длины.

Для ускорения кодирования в случае обработки избыточных данных очень длинные хеш-цепочки обрубаются до определенной длины, задаваемой параметрами алгоритма. Усечение производится в зависимости от длины уже найденного совпадения: чем оно длиннее, тем больше отрезаем.

Сжатие может быть улучшено за счет механизма «ленивого» сравнения (lazy matching, или lazy evaluation). Этот подход позволяет отойти от прямолинейного, «жадного» разбора входной последовательности и повысить эффективность сжатия путем более аккуратного выбора фраз словаря. После того, как определяется совпадающая фраза *match* (t+1) длины *match_len* (t+1) = L для строки s_{t+1}, s_{t+2}, s_{t+3}, …, находящейся в начале буфера, выполняется поиск совпадения *match* (t+2) для строки s_{t+2}, s_{t+3}, s_{t+4}… Если *match_len* (t+2) > *match_len* (t+1) = L, то отказываемся от замещения строки s_{t+1}, s_{t+2}, s_{t+3}… Решение о том, следует кодировать *match* (t+2) или нет, принимается на шаге t+3 по результатам аналогичной проверки. Иначе кодирование протекает обычным образом, но с «запаздыванием» на один шаг. Подробнее:

```
// минимальная длина совпадения
const int THRESHOLD = 3;
// смещение и длина совпадения для match(t+1)
int  prev_pos,
     prev_len;
// смещение и длина совпадения для match(t+2)
int  match_pos,
     match_len = THRESHOLD - 1;
// признак отложенного кодирования фразы match(t+1)
int  match_available = 0;
...
prev_pos = match_pos;     prev_len = match_len;
/*найдем максимальное (или достаточно длинное)
   совпадение для позиции t+2
*/
find_phrase (&match_pos, &match_len);
if ( prev_len >= THRESHOLD &&
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
      match_len <= prev_len ) {
   /* считаем, что выгоднее закодировать фразу
      match(t+1)
   */
   encode_phrase (prev_pos, prev_len);
   match_available = 0;
   match_len = THRESHOLD - 1;
   // сдвинем окно на match_len(t+1)-1 символов
   move_window (prev_len-1);
   t += prev_len-1;
} else {
   // отложим решение о кодировании на один шаг
   if (match_available) {
      /*кодирование литерала s_{t+1} или фразы match(t+1)
         было отложено; кодируем литерал s_{t+1}
      */
      encode_literal (window[t+1]);
   }else{
      match_available = 1;
   }
   move_window (1);
   t++;
}
```

Можно сказать, что это одна из возможных реализаций схемы ленивого сравнения с просмотром на один символ вперед. В зависимости от параметров алгоритма, для обеспечения желаемого соотношения скорости и коэффициента сжатия механизм ленивого сравнения может запускаться при различных значениях L.

Недостатком рассмотренной реализации ленивого сравнения является порождение длинной цепочки литералов, если на каждом последующем шаге длина совпадения больше, чем на предыдущем.

> **Упражнение:** Объясните, почему использование ленивого сравнения при сжатии не требует внесения изменений в алгоритм декодера.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Кодер обрывает текущий блок данных, если определяет, что изменение кодов Хаффмана может улучшить сжатие, или когда происходит переполнение буфера хранения блока.

Пути улучшения сжатия для методов LZ

Улучшать сжатие алгоритмов семейства Зива-Лемпела можно двумя путями:

1) уменьшением количества указателей при неизменной или большей общей длине закодированных фраз за счет более эффективного разбиения входной последовательности на фразы словаря;
2) увеличением эффективности кодирования индексов фраз словаря и литералов, т.е. уменьшением количества битов, в среднем требуемых для кодирования индекса или литерала.

Идея приемов, относящихся к первому пути, была продемонстрирована на примере ленивого сравнения при описании Deflate. Действительно, для одного и того же словаря мы имеем огромное количество вариантов построения набора фраз для замещения им сжимаемой последовательности. Естественным является так называемый «жадный» разбор (greedy parsing), при котором на каждом шаге кодер выбирает самую длинную фразу. Заметим, что такой способ разбиения данных используют все рассмотренные нами классические алгоритмы LZ. Если поиск ведется по всему словарю, то жадный разбор обеспечивает наибольшую скорость, но и практически всегда наихудшее сжатие. Стратегии оптимального разбора позволяют значительно улучшить сжатие, до 10% и более, но серьезным образом замедляют работу компрессора. Алгоритмы оптимального разбора для алгоритмов семейства LZ77 рассмотрены, например, в [1, 10], а для семейства LZ78 — в [8].

Добиваться большей компактности представления индексов словаря можно за счет:

- применения более сложных алгоритмов сжатия, например, на базе контекстных методов моделирования;

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

- минимизации объема словаря путем удаления излишних совпадающих фраз.

В качестве способа увеличения эффективности кодирования литералов может выступать явное статистическое моделирования вероятностей появления литералов в сочетании с арифметическим кодированием, которое собственно и обеспечивает сжатие. При этом, как показывают эксперименты, для улучшения сжатия целесообразно использовать контекстное моделирование 1 или 2 порядков.

Идеи нескольких способов увеличения степени сжатия для методов Зива-Лемпела достаточно подробно описаны ниже.

Стратегия разбора LFF

Как возможную технику улучшения качества разбора входной последовательности на фразы словаря LZ77 укажем метод кодирования самой длинной строки первой — Longest Fragment First, или LFF. Суть LFF заключается в том, что рассматривается несколько вариантов разбиения буфера на фразы словаря и первой замещается строка, с которой совпала фраза максимальной длины, причем строка может начинаться в любой позиции буфера.

Пример

Допустим, у нас в буфере находится строка «абракадабра». Пусть в словаре для каждой позиции буфера можно найти следующие совпадающие фразы максимальной длины:

№ позиции	Совпадающая фраза максимальной длины	Длина фразы
1	аб	2
2	брак	4
3	рак	3
4	ака	3
5	кадаб	5
6	адаб	4
7	даб	3

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

№ позиции	Совпадающая фраза максимальной длины	Длина фразы
…		

Поиск прерван на 7 позиции, поскольку уже никакое совпадение не может быть длиннее максимального встреченного 5. Так как способ кодирования самой длинной строки «кадаб» определен, то теперь для оставшейся подстроки «абра» снова ищется самая длинная совпадающая фраза. Это будет «бра». Оставшаяся слева строка «а» может быть закодирована как литерал. Тогда разбор буфера будет иметь вид:

«абракадаб…» → <а> <бра> <кадаб>.

Эта последовательность из литерала и двух фраз кодируется; окно смещается на 9 символов, а буфер приобретает вид «ра…».

Заметим, что в случае жадного разбора буфер был бы разбит таким образом:

«абракадаб…» → <аб> <рак> <адаб>.

LFF позволяет улучшить сжатие на 0.5…1% по сравнению с жадным разбором.

Оптимальный разбор для методов LZ77

Эвристические техники повышения эффективности разбора входной последовательности — например, рассмотренные ленивое сравнение и метод LFF, не решают проблемы получения очень хорошего разбиения в общем случае. Очевидно, что хотелось бы иметь оптимальную стратегию разбора, *во всех случаях* обеспечивающую минимизацию длины закодированной последовательности, порождаемой компрессорами с алгоритмом словарного сжатия типа LZ77 или LZ78.

Для алгоритмов семейства LZ77 эта задача была рассмотрена в [10] и признана *NP*-полной, т.е. требующей полного перебора всех вариантов для нахождения оптимального решения. В диссертации [1] был изложен однопроходной алгоритм, который, как утверждается, позволяет получить оптимальное разбиение при затратах времени не больших, чем вносимых ленивым сравнением.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Мы рассмотрим алгоритм, обеспечивающий получение почти оптимального решения для методов LZ77. Похожая схема используется, например, в компрессоре CABARC.

В общем случае, для каждой позиции *t* находим все совпадающие фразы длины от 2 до максимальной `MAX_LEN`. На основании информации об используемых для сжатия фраз и литералов кодах мы можем оценить, сколько примерно битов потребуется для кодирования каждой фразы (литерала), или какова цена `price` ее кодирования. Тогда мы можем найти близкое к оптимальному решение за один проход следующим образом.

В массив `offsets` будем записывать ссылки на фразы, подходящие для замещения строки буфера, длины от 2 до `max_len`, где `max_len` является длиной максимального совпадения для текущей позиции *t*. В поле `offs[len]` сохраняем смещение фразы с длиной `len`. Если имеется несколько вариантов фраз с одной и той же `len`, то выбираем фразу с наименьшей ценой `price`: `offsets [t].offs[len] = offset_min_price (t, len)`. Размер обрабатываемого блока равен `MAX_T`. Для обеспечения эффективности разбора `MAX_T` должно быть достаточно большим — несколько сотен байтов и более.

```
struct {
   int max_len;
   int offs[MAX_LEN+1];
} offsets[MAX_T+1];
```

В ячейках `path[t]` массива `path` будем хранить информацию о том, как добраться до позиции *t*, «заплатив» минимальную цену.

```
struct Node {
   /*цена самого «дешевого» пути до t (из пока известных
      путей)
   */
   int price;
   //с какой позиции мы попадаем в t
   int prev_t;
   /*если мы попадем в t, кодируя фразу, то здесь
```

 Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
      хранится смещение этой фразы в словаре
   */
   int offs;
} path[MAX_T+1];
```

Основной цикл разбора:

```
path[0].price = 0;
for (t = 1; t < MAX_T; t++){
   //установим недостижимо большую цену для всех t
   path[t].price = INFINITY;
}
for (t = 0; t < MAX_T; t++){
   /*найдем все совпадающие фразы для строки,
      начинающейся в позиции t
   */
   find_matches (offsets[t]);
   for (len = 1; len <= offsets[t].max_len; len++){
      /*определяем цену рассматриваемого перехода на len
         символов вперед
      */
      if (len == 1)
         // найдем цену кодирования литерала
         price = get_literal_price (t);
      else
         // найдем цену кодирования фразы длины len
         price = get_match_price (len,
                     offsets[t].offs[len]);
      // вычислим цену пути до t + len
      new_price = path[t].price + price;
      if (new_price < path[t+len].price){
         /*рассматриваемый путь до t + len выгоднее
            хранящегося в path[t+len]
         */
         path[t+len].price = new_price;
         path[t+len].prev_t = t;
         if (len > 1)
            path[t+len].offs = offsets[t].offs[len];
      }
   }
}
```

В результате работы алгоритма получаем почти оптимальное решение, записанное в виде односвязного списка. Если предположить, что длина `max_len` ограничивается в функции `find_matches` так, чтобы мы не «перепрыгнули» позицию `MAX_T`,

то головой списка является элемент `path[MAX_T]`. Путь записан в обратном порядке, т.е. фраза со смещением `path[MAX_T].offs` и длиной `MAX_T - path[MAX_T].prev_t` (или литерал в позиции `MAX_T-1`) должна кодироваться самой последней.

На практике применяется несколько эвристических правил, ускоряющих поиск за счет уменьшения числа рассматриваемых вариантов ветвления. Описанный алгоритм позволяет улучшить сжатие для алгоритмов семейства LZ77 на несколько процентов.

> **Упражнение:** Придумайте алгоритм кодирования найденной последовательности фраз и литералов.

Алгоритм Бендера-Вулфа

Очевидно, что классические алгоритмы семейства LZ77 обладают большой избыточностью словаря. Так, например, в словаре может быть несколько одинаковых фраз длины от *match_len* и менее, совпадающих со строкой в начале буфере. Но классический LZ77 никак не использует такую информацию и, фактически, отводит каждой фразе одинаковый объем кодового пространства, считает их равновероятными. В 1991 Бендер (Bender) и Вулф (Wolf) описали прием, позволяющий до некоторой степени компенсировать данную «врожденную» избыточность алгоритмов LZ со скользящим окном [2]. В этом алгоритме (LZBW) после нахождения фразы *S*, имеющей самое длинное совпадение с буфером, производится поиск самой длинной совпадающей фразы *S'* среди добавленных в словарь позже *S* и полностью находящихся в словаре (не вторгающихся в область буфера). Длина *S* передается декодеру как разница между длиной *S* и длиной *S'*. Например, если *S* = “абсд” и *S'* = “аб”, то длина *S* передается с помощью разностной длины 2.

Пример

Модифицируем LZSS с помощью техники Бендера-Вулфа. Рассмотрим процесс кодирования строки

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

“колол_кот_ломом_колесо”, начиная с первого появления символа ‘м’. В отличие от рассмотренного выше примера LZSS, словарное кодирование будем применять при длине совпадения 1 и более.

Таблица 3.11

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные			
	Словарь	Буфер		f	i	j	s
k	колол_кот_ло	мом_кол	-	0	-	-	‘м’
k+1	колол_кот_лом	ом_коле	ом	1	2	2	-
k+2	колол_кот_ломом	_колесо	_	1	6	1	-
k+3	колол_кот_ломом_	колесо	кол	1	16	1	-
k+4	…л_кот_ломом_ко л	есо	-	0	-	-	‘е’

На шаге *k* мы встретили символ ‘м’, отсутствующий в словаре, поэтому передаем его в явном виде. Сдвигаем окно на 1 позицию.

На шаге *k*+1 совпадающая фраза равна “ом”, и в словаре нет никаких других фраз, добавленных позже “ом”. Поэтому положим длину *S*’ равной 0, тогда передаваемая разность *j* есть 2. Сдвигаем окно на 2 символа.

На шаге *k*+2 совпадающая фраза состоит из одного символа ‘_’, последний раз встреченного 6 символов назад. Здесь, как и на шаге *k*+1, разностная длина совпадающей фразы равна ее длине, т.е. единице. Сдвигаем окно на 1 символ.

На шаге *k*+3 совпадающая фраза максимальной длины есть “кол”, но среди фраз, добавленных в словарь после “кол”, имеется фраза “ко”, поэтому длина “кол” кодируется как разница 3 и 2. Если бы в части словаря, ограниченной фразой “кол” слева и началом буфера справа, не нашлось бы фразы “ко” с длиной совпадения 2, а была бы обнаружена, например, только фраза “к” с длиной совпадения 1, то длина “кол” была бы представлена как 3 – 1 = 2.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

При декодировании необходимо поддерживать словарь в таком же виде, что и при кодировании. После получения смещения *match_pos* декодер производит сравнение фразы, начинающейся с найденной позиции *t*–(*match_pos*–1) (*t*+1 — позиция начала буфера), со всеми более «новыми» фразами словаря, т.е. лежащими в области *t*–(match_pos–1), …, *t*. Длина фразы восстанавливается как сумма максимального совпадения и полученной от кодера разностной длины *j*. Так, например, процедура декодирования для шага *k*+3 будет выглядеть следующим образом. Декодер читает смещение *i* = 16, разностную длину *j* = 1 и начинает сравнивать фразу “колол_…”, начало которой определяется данным смещением *i*, со всеми фразами словаря, начало которых имеет смещение от 15 до 1. Максимальное совпадение имеет фраза “ко”, расположенная по смещению 10. Поэтому длина закодированной фразы равна 2 + *j* = 2 + 1 = 3, т.е. кодер передал указатель на фразу “кол”.

Использование техники Бендера-Вулфа позволяет улучшить сжатие для алгоритмов типа LZH примерно на 1%. При этом декодирование сильно замедляется (в разы!), поскольку мы вынуждены выполнять такой же дополнительный поиск для определения длины, что и при сжатии.

Алгоритм Файэлэ-Грини

Еще один способ борьбы с избыточностью словаря LZ77 предложен Файэлэ (Fiala) и Грини (Greene) [4]. В разработанном ими алгоритме LZFG для просмотра словаря используется дерево цифрового поиска, и любая фраза кодируется не парой <длина, смещение>, а индексом узла, соответствующего этой фразе. Иначе говоря, всем одинаковым фразам соответствует один и тот же индекс. Устранение повторяющихся фраз из словаря позволяет уменьшить среднюю длину кодов фраз. Обычно коэффициент сжатия LZFG лучше коэффициента сжатия LZSS примерно на 10%.

КОНТЕКСТНО-ЗАВИСИМЫЕ СЛОВАРИ

Объем словаря и, соответственно, средняя длина закодированного индекса могут быть уменьшены за счет использования приемов контекстного моделирования. Было установлено, что применение контекстно-зависимых словарей улучшает сжатие для алгоритмов семейства LZ77 и, в особенности, семейства LZ78 [5]. Идея подхода состоит в следующем. Пусть мы только что закодировали последовательность символов *S* небольшой длины *L*, тогда на текущем шаге в качестве словаря используются только строки, встреченные в уже обработанной части потока непосредственно после строк, равных *S*. Эти строки образуют контекстно-зависимый словарь порядка *L* для *S*. Если в словаре порядка *L* совпадение обнаружить не удалось, то происходит уход к словарю порядка *L*–1. В конечном итоге, если не было найдено совпадающих фраз ни в одном из доступных словарей, то текущий символ передается в явном виде.

Например, при использовании техники контекстно-зависимых словарей порядка в сочетании с LZ77, после обработки последовательности “абракадабра” получаем следующий состав словарей порядков 2, 1 и 0 для текущего контекста:

Порядок *L*	Состав контекстно-зависимого словаря порядка *L* (фразы словаря могут начинаться только в первой позиции указанных последовательностей)
2 (контекст “ра”)	кадабра
1 (контекст “а”)	бракадабра кадабра дабра бра

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Порядок L	Состав контекстно-зависимого словаря порядка L (фразы словаря могут начинаться только в первой позиции указанных последовательностей)
0 (пустой контекст)	абракадабра бракадабра ракадабра акадабра кадабра адабра дабра абра бра ра а (обычный словарь LZ77)

Эксперименты показали, что наилучшие результаты достигаются при $L = 1$ или 2, т.е. в качестве контекста достаточно использовать один или два предыдущих символа. Применение контекстно-зависимых словарей позволяет улучшить сжатие LZSS на 1–2%, LZFG — на 5%, LZW — примерно на 10%. Потери в скорости в случае модификации LZFG составляют порядка 20–30%. Соответствующие версии алгоритмов известны как LZ77-PM, LZFG-PM, LZW-PM [5].

Недостатком техники является необходимость поддерживать достаточно сложную структуру контекстно-зависимых словарей не только при кодировании, но и при декодировании.

Буферизация смещений

Если была закодирована фраза со смещением i, то увеличивается вероятность того, что вскоре нам может потребоваться закодировать фразы с приблизительно таким же смещением $i \pm \delta$, где δ — небольшое число. Это особенно часто проявляется при обработке двоичных данных (исполнимых файлов, файлов ресурсов), поскольку для них характерно наличие сравнительно

длинных последовательностей, отличающихся лишь в нескольких позициях.

Смещение обычно представляется посредством двух (иногда более) полей: базы (сегмента) и поля дополнительных битов, уточняющего значение смещения относительно базы. Поэтому в потоке закодированных данных, порождаемом алгоритмами семейства LZ77, коды фраз с одним и тем же значением базы смещения часто располагаются недалеко друг от друга. Это свойство можно использовать для улучшения сжатия, применив технику буферизации баз смещений.

В буфере запоминается m последних использованных баз смещений, различающихся между собой. Буфер обновляется по принципу списка LRU, т.е. самая последняя использованная база имеет индекс 0, самая «старая» — индекс m–1. Если база B смещения текущей фразы совпадает с одной из содержащихся в буфере баз B_i, то вместо B кодируется индекс i буфера. Затем B_i перемещается в начало списка LRU, т.е. получает индекс 0, а все B_0, B_1, …, B_{i-1} сдвигаются на одну позицию к концу списка. Иначе кодируется собственно база B, после чего она добавляется в начало буфер как B_0, а все буферизованные базы смещаются на одну позицию к концу списка LRU, при этом B_{m-1} удаляется из буфера.

Пример

Примем m равным 2. Если база не совпадает ни с одной содержащейся в буфере, то кодируемое значение базы равно абсолютному значению плюс m. Пусть также содержимое буфера равно {0, 1}, тогда пошаговое преобразование последовательности баз смещений 15, 14, 14, 2, 3, 2,… будет выглядеть следующим образом.

Номер шага	Абсолютное значение базы	Содержимое буфера в начале шага		Кодируемое значение базы
		B_0	B_1	
1	15	0	1	17
2	14	15	0	16

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

		Содержимое буфера в начале шага		
3	14	14	15	0
4	2	14	15	4
5	3	2	14	5
6	2	3	2	1
7	?	2	3	?

Обратите внимание на состояние буфера после шага 3. Оно не изменилось, поскольку все элементы буфера должны быть различны. Иначе мы ухудшим сжатие из-за внесения избыточности в описание баз смещений, поскольку в этом случае одна и та же база может задаваться несколькими числами.

Как показывают эксперименты, оптимальное значение *m* лежит в пределах 4…8.

Стоимость кодирования индекса буфера обычно ниже стоимости кодирования базы смещения непосредственно. Поэтому имеет смысл принудительно увеличивать частоту использования буферизованных смещений за счет подбора фраз с «нужным» расположением в словаре.

Применение рассмотренной техники заметно улучшает сжатие двоичных файлов — до нескольких процентов, но слабо влияет в случае обработки текстов.

Буферизация смещений используется практически во всех современных архиваторах, реализующих алгоритмы семейства LZ77, например: 7-Zip, CABARC, WinRAR.

Совместное кодирование длин и смещений

Между величиной смещения и длиной совпадения имеется незначительная корреляция, величина которой возрастает в случае применения буферизации смещений. Это свойство можно использовать, объединив в один метасимвол длину совпадения `match_len` и базу смещения `offset_base`, и, таким образом, кодировать метасимвол на основании статистики совместного появления определенных длины и смещения. Как и в случае сме-

щения, в метасимвол лучше включать не полностью длину, а ее квантованное значение.

Так, например, в формате LZX (используется в компрессоре CABARC) длины совпадения от 2 до 8 входят в состав метасимвола непосредственно, а все длины `match_len` > 8 отображаются в одно значение. В последнем случае длина совпадения доопределяется путем отдельной передачи величины `match_len` – 9. Метасимволы длина/смещение и литералы входят в один алфавит, поэтому в упрощенном виде алгоритм кодирования таков:

```
if (match_len >= 2) {
   // закодируем фразу
   if ( match_len <= 8 )
      metasymbol = (offset_base<<3) || (match_len-2);
   else
      metasymbol = (offset_base<<3) || 7;
   /*закодируем метасимвол, указав, что это не литерал,
      для правильного отображения значения метасимвола в
      алфавит длин/смещений и литералов
   */
   encode_symbol (metasymbol, NON_LITERAL);
   if (match_len > 8)
      // доопределим длину совпадения
      encode_length_footer (match_len - 9);
   // закодируем младшие биты смещения
   encode_offset_footer (...);
   ...
}else{
   // закодируем литерал в текущей позиции t+1
   encode_symbol (window[t+1], LITERAL)
   ...
}
```

> **Упражнение:** Напишите упрощенный алгоритм декодирования.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Архиваторы и компрессоры, использующие алгоритмы LZ

Среди огромного количества LZ-архиваторов отметим следующие:

1) 7-Zip, автор Игорь Павлов (Pavlov);
2) ACE, автор Маркел Лемке (Lemke);
3) ARJ, автор Роберт Джанг (Jung);
4) ARJZ, автор Булат Зиганшин (Ziganshin);
5) CABARC, корпорация Microsoft;
6) Imp, фирма Technelysium Pty Ltd;
7) JAR, автор Роберт Джанг (Jung);
8) PKZIP, фирма PKWARE Inc.;
9) RAR, автор Евгений Рошал (Roshal);
10) WinZip, фирма Nico Mak Computing;
11) Zip, Info-ZIP group.

Эти архиваторы являются или одними из самых эффективных в классе применяющих методы Зива-Лемпела, или пользуются популярностью, или оказали существенное влияние на развитие словарных алгоритмов, или интересны с точки зрения нескольких указанных критериев. За исключением 7-Zip, словарные алгоритмы всех указанных архиваторов можно рассматривать как модификации LZH. В алгоритме LZMA, реализованном в 7-Zip, совместно со словарными заменами используется контекстное моделирование и арифметическое кодирование.

В табл. 3.12 представлены результаты сравнения некоторых архиваторов по степени сжатия файлов набора CalgCC.

Таблица 3.12

	ARJ	PKZIP	ACE	RAR	CABARC	7-Zip
Bib	3.08	3.16	3.38	3.39	3.45	3.62
Book1	2.41	2.46	2.78	2.80	2.91	2.94
Book2	2.90	2.95	3.36	3.39	3.51	3.59
Geo	1.48	1.49	1.56	1.53	1.70	1.89
News	2.56	2.61	3.00	3.00	3.07	3.16
Obj1	2.06	2.07	2.19	2.18	2.20	2.26
Obj2	3.01	3.04	3.39	3.38	3.54	3.96

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

	ARJ	PKZIP	ACE	RAR	CABARC	7-Zip
Paper1	2.84	2.85	2.91	2.93	2.99	3.07
Paper2	2.74	2.77	2.86	2.88	2.95	3.01
Pic	9.30	9.76	10.53	10.39	10.67	11.76
Progc	2.93	2.94	3.00	3.01	3.04	3.15
Progl	4.35	4.42	4.49	4.55	4.62	4.76
Progp	4.32	4.37	4.55	4.57	4.62	4.73
Trans	4.65	4.79	5.19	5.23	5.30	5.56
Итого	**3.47**	**3.55**	**3.80**	**3.80**	**3.90**	**4.10**

Использованные версии архиваторов: ARJ 2.50a, PKZIP 2.04g, WinRAR 2.71, ACE 2.04, 7-Zip 2.30 beta 7. Во всех случаях применялся тот алгоритм LZ, который обеспечивал наилучшее сжатие. Заметим, что 7-Zip использует специальные методы препроцессинга нетекстовых данных, «отключить» которые не удалось, что до некоторой степени исказило картину. Тем не менее, преимущество этого архиватора на данном тестовом наборе несомненно. В случае WinRAR и ACE режим мультимедийной компрессии намеренно не включался.

При сравнении программ 7-Zip версии 2.3 и RAR версии 3 с другими LZ-архиваторами необходимо следить, чтобы 7-Zip и RAR использовали алгоритм типа LZ, поскольку они имеют в своем арсенале алгоритм PPMII, обеспечивающий высокую степень сжатия текстов.

При сравнении следует учитывать, что скорость сжатия ARJ и PKZIP была примерно в 4.5 раза выше, чем у RAR и ACE, которые, в свою очередь, были быстрее CABARC и 7-Zip приблизительно на 30% . Размер словаря в ARJ и PKZIP в десятки раз меньше, чем в остальных программах.

Вопросы для самоконтроля[5]

1. Какие свойства данных определяют принципиальную возможность их сжатия с помощью LZ-методов?

[5] Ответы к вопросам и упражнениям и исходные тексты программ вы можете найти на http://www.compression.ru/

2. В чем основная разница между алгоритмами семейства LZ77 и семейства LZ78?
3. Какие особенности строения словаря LZ77 позволяют создавать для одного и того же входного файла несколько различных архивных, которые затем можно разжать без потерь информации с помощью одного и того же декодера LZ77? Возможно ли это в случае алгоритма LZ78?
4. Почему в алгоритмах семейства LZ77 короткие строки часто выгоднее сжимать не с помощью словарной замены, а через кодирование как последовательности литералов? Каким образом это связано с величиной смещения фразы, совпадающей со строкой?
5. Приведите пример блока данных, которые в общем случае выгоднее сжимать алгоритмом семейства LZ77, нежели семейства LZ78, а также обратный пример. На основании каких критериев можно сделать предварительный выбор между алгоритмами семейства LZ77 или семейства LZ78, если задаваться только целью максимизации степени сжатия?
6. Почему дистрибутивы программного обеспечения целесообразно архивировать с помощью алгоритмов семейства LZ77?
7. В каких случаях имеет смысл использовать методы кодирования целых чисел — коды Элиаса, Голомба и т.п. — для сжатия потоков смещений и длин совпадения?
8. Применим ли алгоритм «почти оптимального» разбора для методов LZ77, рассмотренный в пункте «Пути улучшения сжатия для методов LZ», для обработки потоков? Можно ли однозначно сказать, что любая стратегия получения оптимального разбора требует поблочной обработки данных?
9. Почему применение контекстно-зависимых словарей улучшает степень сжатия для алгоритмов семейства LZ78 значительно больше, чем для алгоритмов семейства LZ77?

Литература

1. Кадач А.В. Эффективные алгоритмы неискажающего сжатия текстовой информации. — Диссертация на соискание ученой степени к.ф-м.н. — Институт систем информатики им. А.П.Ершова, 1997.
2. Bender P.E., Wolf J.K. New asymptotic bounds and improvements on the Lempel-Ziv data compression algorithm // IEEE Transactions on Information Theory. Vol. 37(3), pp. 721-727. May 1991.
3. Deutsch L.P. (1996) DEFLATE Compressed Data Format Specification v.1.3 (RFC1951) http://www.compression.ru/download/articles/lz/rfc1951_pdf.rar
4. Fiala E.R., Greene D.H. Data compression with finite windows. Commun // ACM Vol. 32(4), pp.490-505. Apr. 1989.
5. Hoang D.T., Long P.M., Vitter J.S. Multiple-dictionary compression using partial matching // Proceedings of Data Compression Conference, pp.272-281, Snowbird, Utah, March 1995.
6. Langdon G.G. A note on the Ziv-Lempel model for compressing individual sequences // IEEE Transactions on Information Theory, Vol. 29(2), pp.284-287. March 1983.
7. Larsson J., Moffat A. Offline dictionary-based compression // Proceedings IEEE, Vol. 88(11), pp.1722-1732, Nov. 2000.
8. Matias Y., Rajpoot N., Sahinalp S.C. Implementation and experimental evaluation of flexible parsing for dynamic dictionary based data compression // Proceedings WAE'98, 1998.
9. Rissanen J.J., Langdon G.G. Universal modeling and coding // IEEE Transactions on Information Theory, Vol. 27(1), pp.12-23, Jan. 1981.
10. Storer J.A., Szymanski T.G. Data compression via textual substitution // Journal of ACM, Vol. 29(4), pp.928-951, Oct. 1982.

11. Welch T.A. A technique for high-performance data compression // IEEE Computer, Vol. 17(6), pp.8-19, June 1984.
12. Ziv J., Lempel A. A universal algorithm for sequential data compression // IEEE Transactions on Information Theory, Vol. 23(3), pp.337-343, May 1977.
13. Ziv J. and Lempel A. Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory, Vol. 24(5), pp.530-536, Sept. 1978.

Список архиваторов и компрессоров

1. Info-ZIP group. Info-ZIP's portable Zip — C sources. http://www.infozip.org
2. Jung R. ARJ archiver. http://www.arjsoftware.com
3. Jung R. JAR archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/jar102x.exe
4. Lemke M. ACE archiver. http://www.winace.com
5. Microlog Cabinet Manager 2001 for Win9x/NT — Compression tool for .cab files. ftp://ftp.elf.stuba.sk/pub/pc/pack/cab2001.zip
6. Microsoft Corporation. Cabinet Software Development Tool. http://msdn.microsoft.com/library/en-us/dnsamples/cab-sdk.exe
7. Nico Mak Computing. WinZip archiver. http://www.winzip.com
8. Pavlov I. 7-Zip archiver. http://www.7-zip.org
9. PKWARE Inc. PKZIP archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/pk250dos.exe
10. Roshal E. RAR for Windows. http://www.rarsoft.com
11. Technelysium Pty Ltd. IMP archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/imp112.exe
12. Ziganshin B. ARJZ archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/arjz015.zip

Глава 4. Методы контекстного моделирования

Применение методов контекстного моделирования для сжатия данных опирается на парадигму сжатия с помощью «уни-

версальных моделирования и кодирования» (universal modelling and coding), предложенную Риссаненом (Rissanen) и Лэнгдоном (Langdon) в 1981 году [12]. В соответствии с данной идеей процесс сжатия состоит из двух самостоятельных частей:

- **моделирование**;
- **кодирование**.

Под моделированием понимается построение модели информационного источника, породившего сжимаемые данные, а под кодированием — отображение обрабатываемых данных в сжатую форму представления на основании результатов моделирования (рис. 4.1). «**Кодировщик**» создает выходной поток, являющийся компактной формой представления обрабатываемой последовательности, на основании информации, поставляемой ему «**моделировщиком**».

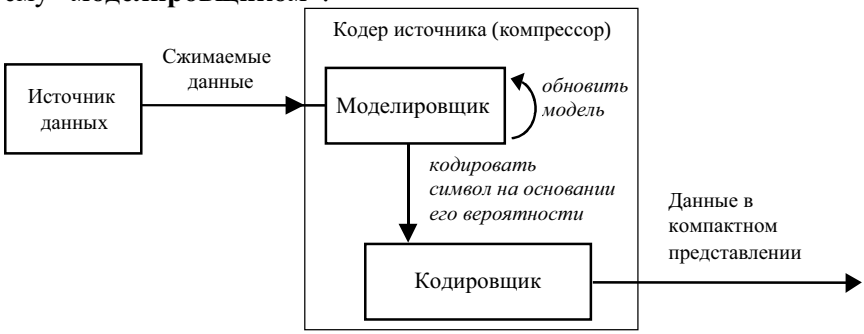


Рис. 4.1. Схема процесса сжатия данных в соответствии с концепцией универсальных моделирования и кодирования

Следует заметить, что понятие «кодирование» часто используют в широком смысле для обозначения всего процесса сжатия, т.е. включая моделирование в данном нами определении. Таким образом, необходимо различать понятия кодирования в широком смысле (весь процесс) и в узком (генерация потока кодов на основании информации модели). Понятие «статистическое кодирование» также используется, зачастую с сомнитель-

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

ной корректностью, для обозначения того или иного уровня кодирования. Во избежание путаницы ряд авторов применяет термин «энтропийное кодирование» для кодирования в узком смысле. Это наименование далеко от совершенства и встречает вполне обоснованную критику. Далее в этой главе процесс кодирования в широком смысле будем именовать «кодированием», а в узком смысле — «статистическим кодированием», или «собственно кодированием».

Из теоремы Шеннона о кодировании источника [13] известно, что символ s_i, вероятность появления которого равняется $p(s_i)$, выгоднее всего представлять $-\log_2 p(s_i)$ битами, при этом средняя длина кодов может быть вычислена по приводившейся ранее формуле (1). Практически всегда истинная структура источника скрыта, поэтому необходимо строить **модель источника**, которая позволила бы нам в каждой позиции входной последовательности найти оценку $q(s_i)$ вероятности появления каждого символа s_i алфавита входной последовательности.

Оценка вероятностей символов при моделировании производится на основании известной статистики и, возможно, априорных предположений, поэтому часто говорят о задаче статистического моделирования. Можно сказать, что моделировщик **предсказывает** вероятность появления каждого символа в каждой позиции входной строки, отсюда еще одно наименование этого компонента — «предсказатель», или «предиктор» (от “predictor”). На этапе статистического кодирования выполняется замещение символа s_i с оценкой вероятности появления $q(s_i)$ кодом длиной $-\log_2 q(s_i)$ битов.

Рассмотрим пример. Предположим, что мы сжимаем последовательность символов алфавита {‘0’,’1’}, порожденную источником без памяти, и вероятности генерации символов следующие: $p(\text{‘0’}) = 0.4$, $p(\text{‘1’}) = 0.6$. Пусть наша модель дает такие оценки вероятностей: $q(\text{‘0’}) = 0.35$, $q(\text{‘1’}) = 0.65$. Энтропия H источника равна

$$- p('0')\log_2 p('0') - p('1')\log_2 p('1') =$$
$$= -0.4\log_2 0.4 - 0.6\log_2 0.6 \approx 0.971 \textit{ бита}.$$

Если подходить формально, то «энтропия» модели получается равной

$$- q('0')\log_2 q('0') - q('1')\log_2 q('1') =$$
$$= -0.35\log_2 0.35 - 0.65\log_2 0.65 \approx 0.934 \textit{ бита}.$$

Казалось бы, что модель обеспечивает лучшее сжатие, чем это позволяет формула Шеннона. Но истинные вероятности появления символов не изменились! Если исходить из вероятностей p, то ‘0’ следует кодировать $-\log_2 0.4 \approx 1.322$ бита, а для ‘1’ нужно отводить $-\log_2 0.6 \approx 0.737$ бита. Для оценок вероятностей q мы имеем $-\log_2 0.35 \approx 1.515$ бита и $-\log_2 0.65 \approx 0.621$ бита соответственно. При каждом кодировании на основании информации модели в случае ‘0’ мы будем терять $1.515 - 1.322 = 0.193$ бита, а в случае ‘1’ выигрывать $0.737 - 0.621 = 0.116$ бита. С учетом вероятностей появления символов средний проигрыш при каждом кодировании составит $0.4 \cdot 0.193 - 0.6 \cdot 0.116 = 0.008$ бита.

Вывод: ***Чем точнее оценка*** вероятностей появления символов, тем больше коды соответствуют оптимальным, ***тем лучше сжатие***.

Правильность декодирования обеспечивается использованием точно такой же модели, что была применена при кодировании. Следовательно, при моделировании для сжатия данных нельзя пользоваться информацией, которая неизвестна декодеру.

Осознание двойственной природы процесса сжатия позволяет осуществлять декомпозицию задач компрессии данных со сложной структурой и нетривиальными взаимозависимостями, обеспечивать определенную самостоятельность процедур, ре-

шающих частные проблемы, сосредотачивать больше внимания на деталях реализации конкретного элемента.

Задача статистического кодирования была в целом успешно решена к началу 1980-х годов. Арифметический кодер позволяет сгенерировать сжатую последовательность, длина которой обычно всего лишь на десятые доли процента превышает теоретическую длину, рассчитанную с помощью формулы (1) (см. пункт «Арифметическое сжатие» главы 1). Более того, применение современной модификации арифметического кодера — интервального кодера — позволяет осуществлять собственно кодирование очень быстро. Скорость статистического кодирования составляет миллионы символов в секунду на современных ПК.

В свете вышесказанного, повышение точности моделей является, фактически, единственным способом существенного улучшения сжатия.

Классификация стратегий моделирования

Перед рассмотрением контекстных методов моделирования следует сказать о классификации стратегий моделирования источника данных по способу построения и обновления модели. Выделяют четыре варианта моделирования:

- статическое;
- полуадаптивное;
- адаптивное (динамическое);
- блочно-адаптивное.

При статическом моделировании для любых обрабатываемых данных используется одна и та же модель. Иначе говоря, не производится адаптация модели к особенностям сжимаемых данных. Описание заранее построенной модели хранится в структурах данных кодера и декодера; таким образом достигается однозначность кодирования, с одной стороны, и отсутствие необходимости в явной передаче модели, с другой. Недостаток подхода также очевиден: мы можем получать плохое сжатие и даже увеличивать размер представления, если обрабатываемые

данные не соответствуют выбранной модели. Поэтому такая стратегия используется только в специализированных приложениях, когда тип сжимаемых данных неизменен и заранее известен.

Полуадаптивное сжатие является развитием стратегии статического моделирования. В этом случае для сжатия заданной последовательности выбирается *или* строится модель на основании анализа именно обрабатываемых данных. Понятно, что кодер должен передавать декодеру не только закодированные данные, но и описание использованной модели. Если модель выбирается из заранее созданных и известных как кодеру, так и декодеру, то это просто порядковый номер модели. Иначе, если модель была настроена или построена при кодировании, то необходимо передавать либо значения параметров настройки, либо модель полностью. В общем случае полуадаптивный подход дает лучшее сжатие, чем статический, т.к. обеспечивает приспособление к природе обрабатываемых данных, уменьшая вероятность значительной разницы между предсказаниями модели и реальным поведением потока данных.

Адаптивное моделирование является естественной противоположностью статической стратегии. По мере кодирования модель изменяется по заданному алгоритму после сжатия каждого символа. Однозначность декодирования достигается тем, что, во-первых, изначально кодер и декодер имеют идентичную и обычно очень простую модель и, во-вторых, модификация модели при сжатии и разжатии осуществляется одинаковым образом. Опыт использования моделей различных типов показывает, что адаптивное моделирование является не только элегантной техникой, но и обеспечивает, по крайней мере, не худшее сжатие, чем полуадаптивное моделирование. Понятно, что если стоит задача создания «универсального» компрессора для сжатия данных несходных типов, то адаптивный подход является естественным выбором разработчика.

Блочно-адаптивное моделирование можно рассматривать как частный случай адаптивной стратегии (или наоборот, что сути

дела не меняет). В зависимости от конкретного алгоритма обновления модели, оценки вероятностей символов, метода статистического кодирования и самих данных изменение модели после обработки каждого символа может быть сопряжено со следующими неприятностями:

- потеря устойчивости (робастности) оценок, если данные «зашумлены», или имеются значительные локальные изменения статистических взаимосвязей между символами обрабатываемого потока; иначе говоря, чересчур быстрая, «агрессивная» адаптация модели может приводить к ухудшению точности оценок;
- большие вычислительные расходы на обновление модели (как пример — в случае адаптивного кодирования по алгоритму Хаффмана);
- большие расходы памяти для хранения структур данных, обеспечивающих быструю модификацию модели.

Поэтому обновление модели может выполняться после обработки целого блока символов, в общем случае переменной длины. Для обеспечения правильности разжатия декодер должен выполнять такую же последовательность действий по обновлению модели, что и кодер, либо кодеру необходимо передавать вместе со сжатыми данными инструкции по модификации модели. Последний вариант достаточно часто используется при блочно-адаптивном моделировании для ускорения процесса декодирования в ущерб коэффициенту сжатия.

Понятно, что приведенная классификация является до некоторой степени абстрактной, и на практике часто используют гибридные схемы.

Контекстное моделирование

Итак, нам необходимо решить задачу оценки вероятностей появления символов в каждой позиции обрабатываемой последовательности. Для того чтобы разжатие произошло без потерь, мы можем пользоваться только той информацией, которая в полной мере известна как кодеру, так и декодеру. Обычно это

означает, что оценка вероятности очередного символа должна зависеть только от свойств уже обработанного блока данных.

Пожалуй, наиболее простой способ оценки реализуется с помощью полуадаптивного моделирования и заключается в предварительном подсчете безусловной частоты появления символов в сжимаемом блоке. Полученное распределение вероятностей используется для статистического кодирования всех символов блока. Если, например, такую модель применить для сжатия текста на русском языке, то в среднем на кодирование каждого символа будет потрачено примерно 4.5 бита. Это значение является средней длиной кодов для модели, базирующейся на использовании безусловного распределения вероятностей букв в тексте. Заметим, что уже в этом простом случае достигается степень сжатия 1.5 по отношению к тривиальному кодированию, когда всем символам назначаются коды одинаковой длины. Действительно, размер алфавита русского текста превышает 64, но меньше 128 знаков (строчные и заглавные буквы, знаки препинания, пробел), что требует 7-битовых кодов.

Анализ распространенных типов данных — например, тех же текстов на естественных языках, — выявляет сильную зависимость вероятности появления символов от непосредственно им предшествующих. Иначе говоря, б*о*льшая часть данных, с которыми мы сталкиваемся, порождается источниками с памятью. Допустим, нам известно, что сжимаемый блок является текстом на русском языке. Если, например, строка из трех только что обработанных символов равна “_цы” (подчеркиванием здесь и далее обозначается пробел), то текущий символ скорее всего входит в следующую группу: ‘г’ («цыган»), ‘к’ («цыкать»), ‘п’ («цыпочки»), ‘ц’ («цыц»). Или, в случае анализа сразу нескольких слов, если предыдущая строка равна “Вставай,_проклятьем_заклейменный,”, то продолжением явно будет “весь_мир_”. Следовательно, учет зависимости частоты появления символа (в общем случае — блока символов) от предыдущих должен давать более точные оценки и, в конечном счете, лучшее сжатие. Действительно, в случае посимвольного

кодирования при использовании информации об одном непосредственно предшествующем символе достигается средняя длина кодов в 3.6 бита для русских текстов, при учете двух последних — уже порядка 3.2 бита. В первом случае моделируются условные распределения вероятностей символов, зависящие от значения строки из одного непосредственно предшествующего символа, во втором — зависящие от строки из двух предшествующих символов.

Любопытно, что модели, оперирующие безусловными частотами и частотами в зависимости от одного предшествующего символа, дают примерно одинаковые результаты для всех европейских языков (за исключением, быть может, самых экзотических) — 4.5 и 3.6 бита соответственно.

Улучшение сжатия при учете предыдущих элементов (пикселов, сэмплов, отсчетов, чисел) отмечается и при обработке данных других распространенных типов: объектных файлов, изображений, аудиозаписей, таблиц чисел.

Терминология

Под контекстным моделированием будем понимать оценку вероятности появления символа (элемента, пиксела, сэмпла, отсчета и даже набора качественно разных объектов) в зависимости от непосредственно ему предшествующих, или контекста.

Заметим, что в быту понятие «контекст» обычно используется в глобальном значении — как совокупность символов (элементов), окружающих текущий обрабатываемый. Это контекст в широком смысле. Выделяют также «левосторонние» и «правосторонние» контексты, т.е. последовательности символов, непосредственно примыкающие к текущему символу слева и справа соответственно. Здесь и далее под контекстом будем понимать именно классический левосторонний: так, например, для последнего символа ‘о’ последовательности “…молоко…” контекстом является “…молок”.

Если длина контекста ограничена, то такой подход будем называть контекстным моделированием ограниченного порядка

(finite-context modeling), при этом под порядком понимается максимальная длина используемых контекстов *N*. Например, при моделировании порядка 3 для последнего символа ‘о’ в последовательности “…молоко…” контекстом максимальной длины 3 является строка “лок”. При сжатии этого символа под «текущими контекстами» могут пониматься “лок”, “ок”, “к”, а также пустая строка “”. Все эти контексты длины от *N* до 0 назовем активными контекстами в том смысле, что при оценке символа может быть использована накопленная для них статистика.

Далее вместо «контекст длины *o*, $o \le N$» мы будем обычно говорить «контекст порядка *o*».

В силу объективных причин — ограниченность вычислительных ресурсов — техника контекстного моделирования именно ограниченного порядка получила наибольшее развитие и распространение, поэтому далее под контекстным моделированием будем понимать именно ее. Дальнейшее изложение также учитывает специфику того, что контекстное моделирование практически всегда применяется как адаптивное.

Оценки вероятностей при контекстном моделировании строятся на основании обычных счетчиков частот, связанных с текущим контекстом. Если мы обработали строку “абсабвбабс”, то для контекста “аб” счетчик символа ‘с’ равен двум (говорят, что символ ‘с’ *появился в контексте* “аб” два раза), символа ‘в’ — единице. На основании этой статистики можно утверждать, что вероятность появления ‘с’ после “аб” равна 2/3, а вероятность появления ‘в’ — 1/3, т.е. оценки формируются на основе уже просмотренной части потока.

В общем случае для каждого контекста конечной длины $o \le N$, встречаемого в обрабатываемой последовательности, создается контекстная модель КМ. Любая КМ включает в себя счетчики всех символов, встреченных в соответствующем ей контексте, т.е. сразу после строки контекста. После каждого появления какого-то символа *s* в рассматриваемом контексте производится увеличение значения счетчика символа *s* в соответствующей контексту КМ. Обычно счетчики инициализируются

нулями. На практике счетчики обычно создаются по мере появления в заданном контексте новых символов, т.е. счетчиков ни разу не виденных в заданном контексте символов просто не существует.

Под порядком КМ будем понимать длину соответствующего ей контекста. Если порядок КМ равен *o*, то будем обозначать такую КМ как «КМ(*o*)».

Кроме обычных КМ, часто используют контекстную модель минус первого порядка КМ(–1), присваивающую одинаковую вероятность всем символам алфавита сжимаемого потока.

Понятно, что для нулевого и минус первого порядка контекстная модель одна, а КМ большего порядка может быть несколько, вплоть до q^N, где q — размер алфавита обрабатываемой последовательности. КМ(0) и КМ(–1) всегда активны.

Заметим, что часто не делается различий между понятием «контекст» и «контекстная модель». Авторы этой книги такое соглашение не поддерживают.

Часто говорят о «родительских» и «дочерних» контекстах. Для контекста “к” дочерними являются “ок” и “лк”, поскольку они образованы сцеплением (конкатенацией) одного символа и контекста “к”. Аналогично, для контекста “лок” родительским является контекст “ок”, а контекстами-предками — “ок”, “к”, “”. Очевидно, что «пустой» контекст “” является предком для всех. Аналогичные термины применяются для КМ, соответствующих контекстам.

Совокупность КМ образует модель источника данных. Под порядком модели понимается максимальный порядок используемых КМ.

Виды контекстного моделирования

Пример обработки строки “абсабвбабс” иллюстрирует сразу две проблемы контекстного моделирования:

- как выбирать подходящий контекст (или контексты) среди активных с целью получения более точной оценки, ведь текущий символ может лучше предсказываться не

контекстом второго порядка “аб”, а контекстом первого порядка “б”;

- как оценивать вероятность символов, имеющих нулевую частоту (например, ‘г’).

Выше были приведены цифры, в соответствии с которыми при увеличении длины используемого контекста сжатие данных улучшается. К сожалению, при кодировании блоков типичной длины — единицы мегабайтов и меньше — это справедливо только для небольших порядков модели, т.к. статистика для длинных контекстов медленно накапливается. При этом также следует учитывать, что большинство реальных данных характеризуется неоднородностью, нестабильностью силы и вида статистических взаимосвязей, поэтому «старая» статистика контекстно-зависимых частот появления символов малополезна или даже вредна. Поэтому модели, строящие оценку только на основании информации КМ максимального порядка N, обеспечивают сравнительно низкую точность предсказания. Кроме того, хранение модели большого порядка требует много памяти.

Если в модели используются для оценки только КМ(N), то иногда такой подход называют «чистым» (pure) контекстным моделированием порядка N. Из-за вышеуказанного недостатка «чистые» модели представляют обычно только научный интерес.

Действительно, реально используемые файлы обычно имеют сравнительно небольшой размер, поэтому для улучшения их сжатия необходимо учитывать оценки вероятностей, получаемые на основании статистики контекстов разных длин. Техника объединения оценок вероятностей, соответствующих отдельным активным контекстам, в одну оценку называется смешиванием (blending). Известно несколько способов выполнения смешивания.

Рассмотрим модель произвольного порядка N. Если $q(s_i|o)$ есть вероятность, присваиваемая в активной КМ(o) символу s_i алфавита сжимаемого потока, то смешанная вероятность $q(s_i)$ вычисляется в общем случае как

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

$$q(s_i) = \sum_{o=-1}^{N} w(o) q(s_i \mid o),$$

где $w(o)$ — вес оценки КМ(o) .

Оценка $q(s_i|o)$ обычно определяется через частоту символа s_i по тривиальной формуле

$$q(s_i \mid o) = \frac{f(s_i \mid o)}{f(o)},$$

где $f(s_i|o)$ — частота появления символа s_i в соответствующем контексте порядка o;

$f(o)$ — общая частота появления соответствующего контекста порядка o в обработанной последовательности.

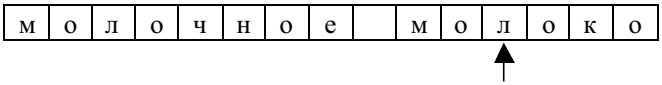
Заметим, что правильнее было бы писать не, скажем, $f(s_i|o)$, а $f(s_i|C_{j(o)})$, т.е. «частота появления символа s_i в КМ порядка o с номером $j(o)$», поскольку контекстных моделей порядка o может быть огромное количество. Но при сжатии каждого текущего символа мы рассматриваем только *одну* КМ для каждого порядка, т.к. контекст определяется *непосредственно* примыкающей слева к символу строкой определенной длины. Иначе говоря, для каждого символа мы имеем набор из N+1 активных контекстов длины от N до 0, каждому из которых однозначно соответствует только одна КМ, если она вообще есть. Поэтому здесь и далее используется сокращенная запись.

Если вес $w(-1) > 0$, то это гарантирует успешность кодирования любого символа входного потока, т.к. наличие КМ(–1) позволяет всегда получать ненулевую оценку вероятности и, соответственно, код конечной длины.

Различают модели с полным смешиванием (fully blended), когда предсказание определяется статистикой КМ всех используемых порядков, и с частичным смешиванием (partially blended) — в противном случае.

Пример 1

Рассмотрим процесс оценки отмеченного на рисунке стрелкой символа ‘л’, встретившегося в блоке “молочное_молоко”. Считаем, что модель работает на уровне символов.



Пусть мы используем контекстное моделирование порядка 2 и делаем полное смешивание оценок распределений вероятностей в КМ второго, первого и нулевого порядков с весами 0.6, 0.3 и 0.1. Считаем, что в начале кодирования в КМ(0) создаются счетчики для всех символов алфавита {‘м’, ‘о’, ‘л’, ‘ч’, ‘н’, ‘е’, ‘_’, ‘к’} и инициализируются единицей; счетчик символа после его обработки увеличивается на 1.

Для текущего символа ‘л’ имеются контексты “мо”, “о” и пустой (нулевого порядка). К данному моменту для них накоплена статистика, показанная в табл. 4.1.

Таблица 4.1

Символы		‘м’	‘о’	‘л’	‘ч’	‘н’	‘е’	‘_’	‘к’
КМ порядка 0 (контекст “”)	Частоты	3	5	2	2	2	2	2	1
	Накопленные частоты	3	8	10	12	14	16	18	19
КМ порядка 1 (контекст “о”)	Частоты	-	-	1	1	-	1	-	-
	Накопленные частоты	-	-	1	2	-	3	-	-
КМ порядка 2 (“мо”)	Частоты	-	-	1	-	-	-	-	-
	Накопленные частоты	-	-	1	-	-	-	-	-

Тогда оценка вероятности для символа ‘л’ будет равна

$$q('л') = 0.1 \cdot \frac{2}{19} + 0.3 \cdot \frac{1}{3} + 0.6 \cdot \frac{1}{1} = 0{,}71.$$

В общем случае, для однозначного кодирования символа ‘л’ такую оценку необходимо проделать для всех символов алфавита. Действительно, с одной стороны, декодер не знает, чему равен текущий символ, с другой стороны, оценка вероятности не гарантирует уникальность кода, а лишь задает его длину. Поэтому статистическое кодирование выполняется на основании накопленной частоты (см. подробности в примере 2 и в пункте «Арифметическое сжатие» главы 1). Например, если кодировать на основании статистики только нулевого порядка, то существует взаимно однозначное соответствие между накопленными частотами из диапазона (8,10] и символом ‘л’, что не имеет места в случае просто частоты (частоту 2 имеют еще 4 символа). Понятно, что аналогичные свойства остаются в силе и в случае оценок, получаемых частичным смешиванием.

Упражнение: Предложите способы увеличения средней скорости вычисления оценок для методов контекстного моделирования со смешиванием, как полным, так и частичным.

Очевидно, что успех применения смешивания зависит от способа выбора весов $w(o)$. Простой путь состоит в использовании заданного набора фиксированных весов КМ разных порядков при каждой оценке; этот способ был применен в примере 2. Естественно, альтернативой является адаптация весов по мере кодирования. Приспособление может заключаться в придании все большей значимости КМ все больших порядков или, скажем, попытке выбрать наилучшие веса на основании определенных статистических характеристик последнего обработанного блока данных. Но так исторически сложилось, что реальное

развитие получили методы неявного взвешивания. Это объясняется в первую очередь их меньшей вычислительной сложностью.

Техника неявного взвешивания связана с введением вспомогательного символа ухода (escape). Символ ухода является квазисимволом и не должен принадлежать алфавиту сжимаемой последовательности. Фактически, он используется для передачи декодеру указаний кодера. Идея заключается в том, что если используемая КМ не позволяет оценить текущий символ (его счетчик равен 0 в этой КМ), то на выход посылается закодированный символ ухода и производится попытка оценить текущий символ в другой КМ, которой соответствует контекст иной длины. Обычно попытка оценки начинается с КМ наибольшего порядка N, затем в определенной последовательности осуществляется переход к контекстным моделям меньших порядков.

Естественно, статистическое кодирование символа ухода выполняется на основании его вероятности, так называемой вероятности ухода. Очевидно, что символы ухода порождаются не источником данных, а моделью. Следовательно, их вероятность может зависеть от характеристик сжимаемых данных, свойств КМ, с которой производится уход, свойств КМ, на которую происходит уход, и т.д. Как можно оценить эту вероятность, имея в виду, что конечный критерий качества — улучшение сжатия? Вероятность ухода — это вероятность появления в контексте нового символа. Тогда, фактически, необходимо оценить правдоподобность наступления ни разу не происходившего события. Теоретического фундамента для решения этой проблемы, видимо, не существует, но за время развития техник контекстного моделирования было предложено несколько подходов, хорошо работающих в большинстве реальных ситуаций. Кроме того, эксперименты показывают, что модели с неявным взвешиванием устойчивы относительно используемого метода оценки вероятности ухода, т.е. выбор какого-то способа вычисления этой величины не влияет на коэффициент сжатия кардинальным образом.

Алгоритмы PPM

Техника контекстного моделирования Prediction by Partial Matching (предсказание по частичному совпадению), предложенная в 1984 году Клири (Cleary) и Уиттеном (Witten) [5], является одним из самых известных подходов к сжатию качественных данных и уж точно самым популярным среди контекстных методов. Значимость подхода обусловлена и тем фактом, что алгоритмы, причисляемые к PPM, неизменно обеспечивают в среднем наилучшее сжатие при кодировании данных различных типов и служат стандартом, «точкой отсчета» при сравнении универсальных алгоритмов сжатия.

Перед собственно рассмотрением алгоритмов PPM необходимо сделать замечание о корректности используемой терминологии. На протяжении примерно 10 лет — с середины 1980-х годов до середины 1990-х — под PPM понималась группа методов с вполне определенными характеристиками. В последние годы, вероятно из-за резкого увеличения числа всевозможных гибридных схем и активного практического использования статистических моделей для сжатия, произошло смешение понятий, и термин «PPM» часто используется для обозначения контекстных методов вообще.

Ниже будет описан некий обобщенный алгоритм PPM, а затем особенности конкретных распространенных схем.

Как и в случае многих других контекстных методов, для каждого контекста, встречаемого в обрабатываемой последовательности, создается своя контекстная модель КМ. При этом под контекстом понимается последовательность элементов одного типа — символов, пикселов, чисел, но не набор разнородных объектов. Далее вместо слова «элемент» мы будем использовать «символ». Каждая КМ включает в себя счетчики всех символов, встреченных в соответствующем контексте.

PPM относится к адаптивным методам моделирования. Исходно кодеру и декодеру поставлена в соответствие начальная модель источника данных. Будем считать, что она состоит из

КМ(–1), присваивающей одинаковую вероятность всем символам алфавита входной последовательности. После обработки текущего символа кодер и декодер изменяют свои модели одинаковым образом, в частности, наращивая величину оценки вероятности рассматриваемого символа. Следующий символ кодируется (декодируется) на основании новой, измененной модели, после чего модель снова модифицируется и т.д. На каждом шаге обеспечивается идентичность модели кодера и декодера за счет применения одинакового механизма ее обновления.

В PPM используется неявное взвешивание оценок. Попытка оценки символа начинается с КМ(N), где N является параметром алгоритма и называется порядком PPM-модели. В случае нулевой частоты символа в КМ текущего порядка осуществляется переход к КМ меньшего порядка за счет использования механизма уходов (escape strategy), рассмотренного в предыдущем пункте.

Фактически, вероятность ухода — это суммарная вероятность всех символов алфавита входного потока, еще ни разу не появлявшихся в контексте. Любая КМ должна давать отличную от нуля оценку вероятности ухода. Исключения из этого правила возможны только в тех случаях, когда значения всех счетчиков КМ для всех символов алфавита отличны от нуля, то есть любой символ может быть оценен в рассматриваемом контексте. Оценка вероятности ухода традиционно является одной из основных проблем алгоритмов с неявным взвешиванием, и она будет специально рассмотрена ниже в пункте «Оценка вероятности ухода».

Вообще говоря, способ моделирования источника с помощью классических алгоритмов PPM опирается на следующие предположения о природе источника:

1. источник является марковским с порядком N, т.е. вероятность генерации символа зависит от N предыдущих символов и только от них;
2. источник имеет такую дополнительную особенность, что чем ближе располагается один из символов контекста к текущему символу, тем больше корреляция между ними.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Таким образом, механизм уходов первоначально рассматривался лишь как вспомогательный прием, позволяющий решить проблему кодирования символов, ни разу не встречавшихся в контексте порядка *N*. В идеале, достигаемом после обработки достаточно длинного блока, никакого обращения к КМ порядка меньше *N* происходить не должно. Иначе говоря, причисление классических алгоритмов PPM к методам, производящим взвешивание, пусть и неявным образом, является не вполне корректным.

При сжатии очередного символа выполняются следующие действия.

Если символ *s* обрабатывается с использованием PPM-модели порядка *N*, то, как мы уже отмечали, в первую очередь рассматривается КМ(*N*). Если она оценивает вероятность *s* числом, не равным нулю, то сама и используется для кодирования *s*. Иначе выдается сигнал в виде символа ухода, и на основе меньшей по порядку КМ(*N*–1) производится очередная попытка оценить вероятность *s*. Кодирование происходит через уход к КМ меньших порядков до тех пор, пока *s* не будет оценен. КМ(–1) гарантирует, что это в конце концов произойдет. Таким образом, каждый символ кодируется серией кодов символа ухода, за которой следует код самого символа. Из этого следует, что вероятность ухода также можно рассматривать как вероятность перехода к контекстной модели меньшего порядка.

Если в процессе оценки обнаруживается, что текущий рассматриваемый контекст встречается в первый раз, то для него создается КМ.

При оценке вероятности символа в КМ порядка $o < N$ можно исключить из рассмотрения все символы, которые содержатся в КМ(o+1), поскольку ни один из них точно не является символом *s*. Для этого в текущей КМ(o) нужно замаскировать, т.е. временно установить в ноль, значения счетчиков всех символов, имеющихся в КМ(o+1). Такая техника называется методом исключения (exclusion).

После собственно кодирования символа обычно осуществляется обновление статистики всех КМ, использованных при оценке его вероятности, за исключением статической КМ(–1).

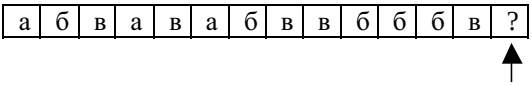
Такой подход называется методом исключения при обновлении. Простейшим способом модификации является инкремент счетчиков символа в этих КМ. Подробнее о стратегиях обновления будет сказано в пункте «Обновление счетчиков символов».

Пример работы алгоритма PPM

Рассмотрим подробнее работу алгоритма PPM с помощью примера.

Пример 2

Имеется последовательность символов "абвавабвббб" алфавита {‘а’, ‘б’, ‘в’, ‘г’}, которая уже была закодирована.



Пусть счетчик символа ухода равен 1 для всех КМ, при обновлении модели счетчики символов увеличиваются на 1 во всех активных КМ, применяется метод исключения, и максимальная длина контекста равна 3, т.е. $N = 3$.

Первоначально модель состоит из КМ(–1), в которой счетчики всех четырех символов алфавита имеют значение 1. Состояние модели обработки последовательности "абвавабвббб" представлено на рис. 4.2, где прямоугольниками обозначены контекстные модели, при этом для каждой КМ указан курсивом контекст, а также встречавшиеся в контексте символы и их частоты.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

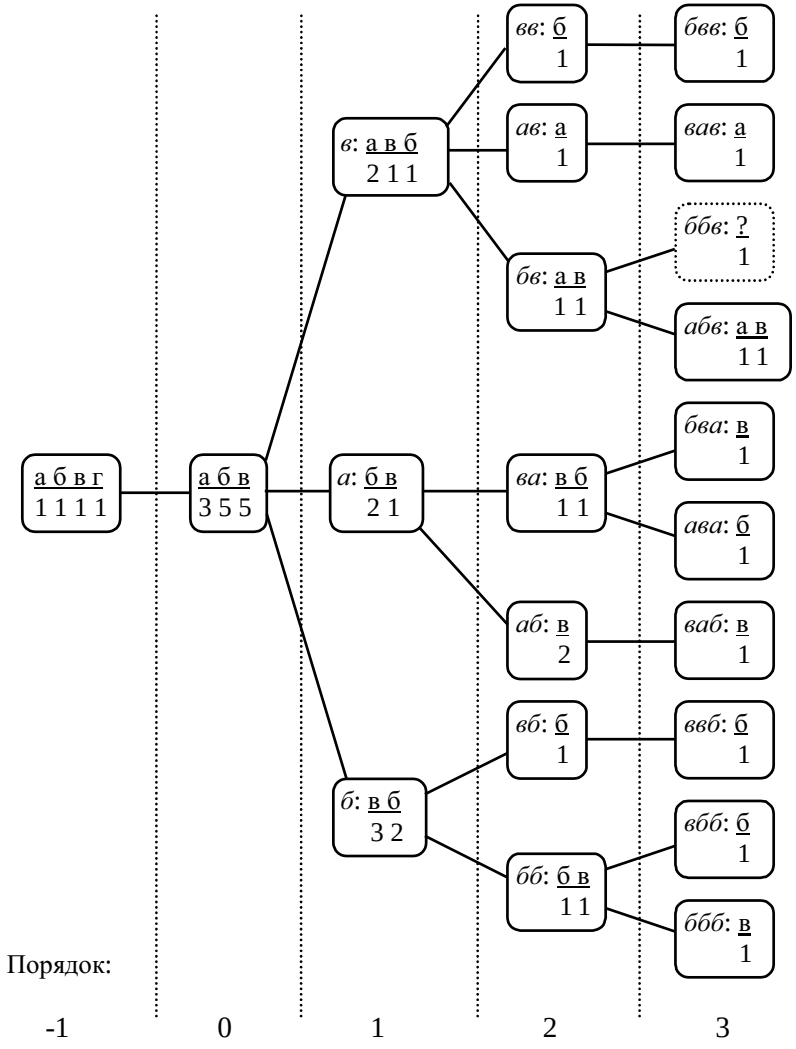


Рис. 4.2. Состояние модели после обработки последовательности "абвавабвббббв"

Пусть текущий символ равен ‘г’, т.е. ‘?’ = ‘г’, тогда процесс его кодирования будет выглядеть следующим образом.

Сначала рассматривается контекст 3-го порядка “ббв”. Ранее он не встречался, поэтому кодер, ничего не послав на выход, переходит к анализу статистики для контекста 2-го порядка. В этом контексте (“бв”) встречались символ ‘а’ и символ ‘в’, счетчики которых в соответствующей КМ равны 1 каждый, поэтому символ ухода кодируется с вероятностью 1/(2+1), где в знаменателе число 2 — это наблюдавшаяся частота появления контекста “бв”, 1 — это значение счетчика символа ухода. В контексте 1-го порядка “в” дважды встречался символ ‘а’, который исключается (маскируется), один раз также исключаемый ‘в’ и один раз ‘б’, поэтому оценка вероятности ухода будет равна 1/(1+1). В КМ(0) символ ‘г’ также оценить нельзя, причем все имеющиеся в этой КМ символы ‘а’, ’б’, ‘в’ исключаются, так как уже встречались нам в КМ более высокого порядка. Поэтому вероятность ухода получается равной 1. Цикл оценивания завершается на уровне КМ(–1), где ‘г’ к этому времени остается единственным до сих пор не попадавшимся символом, поэтому он получает вероятность 1 и кодируется посредством 0 битов. Таким образом, при использовании хорошего статистического кодировщика для представления ‘г’ потребуется в целом примерно 2.6 бита.

Перед обработкой следующего символа создается КМ для строки “ббв” и производится модификация счетчиков символа ‘г’ в созданной и во всех просмотренных КМ. В данном случае требуется изменение КМ всех порядков от 0 до *N*.

Табл. 4.2 демонстрирует оценки вероятностей, которые должны были быть использованы при кодировании символов алфавита {‘а’, ‘б’, ‘в’, ‘г’} в текущей позиции.

Таблица 4.2

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Символ s	Последовательность оценок для КМ каждого порядка от 3 до –1					Общая оценка вероятности q(s)	Представление требует битов
	3	2	1	0	–1		
	“ббв”	“бв”	“в”	“”			
‘а’	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
‘б’	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	-	-	$\frac{1}{6}$	2.6
‘в’	-	$\frac{1}{2+1}$	-	-	-	$\frac{1}{3}$	1.6
‘г’	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	1	1	$\frac{1}{6}$	2.6

Алгоритм декодирования абсолютно симметричен алгоритму кодирования. После декодирования символа в текущей КМ проверяется, не является ли он символом ухода, если это так, то выполняется переход к КМ порядком ниже. Иначе считается, что исходный символ восстановлен, он записывается в декодированный поток и осуществляется переход к следующему шагу. Содержание процедур обновления счетчиков, создания новых контекстных моделей, прочих вспомогательных действий и последовательность их применения должны быть строго одинаковыми при кодировании и декодировании. Иначе возможна рассинхронизация копий модели кодера и декодера, что рано или поздно приведет к ошибочному декодированию какого-то символа. Начиная с этой позиции, вся оставшаяся часть сжатой последовательности будет разжата неправильно.

Разница между кодами символов, оценки вероятности которых одинаковы, достигается за счет того, что PPM-предсказатель передает кодировщику так называемые накопленные частоты (или накопленные вероятности) оцениваемого символа и его соседей или кодовые пространства символов. Так,

например, для контекста “бв” из примера 2 можно составить табл. 4.3.

Таблица 4.3

Символ	Частота	Оценка вероятности	Накопленная вероятность (оценка)	Кодовое пространство
‘а’	1	$\frac{1}{3}$	$\frac{1}{3}$	[0 … 0.33)
‘б’	0	-	-	-
‘в’	1	$\frac{1}{3}$	$\frac{2}{3}$	[0.33 … 0.66)
‘г’	0	-	-	-
Уход	1	$\frac{1}{3}$	1	[0.66 … 1)

Хороший кодировщик должен отобразить символ s с оценкой вероятности $q(s)$ в код длины $\log_2 q(s)$, что и обеспечит сжатие всей обрабатываемой последовательности в целом.

В обобщенном виде алгоритм кодирования можно записать так.

```
/*инициализация контекста длины N (в смысле строки предыдущих
   символов), эта строка должна содержать N предыдущих
   символов, определяя набор активных контекстов длины o
   ≤ N
*/
context = "";
while ( ! DataFile.EOF() ){
   c = DataFile.ReadSymbol(); // текущий символ
   order = N; // текущий порядок КМ
   success = 0; // успешность оценки в текущей КМ
   do{
      // найдем КМ для контекста текущей длины
      CM = ContextModel.FindModel (context, order);

      /*попробуем найти текущий символ c в этой КМ, в
         CumFreq получим его накопленную частоту (или
         накопленную частоту символа ухода), в counter –
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
            ссылку на счетчик символа; флаг success
            указывает на отсутствие ухода
         */
         success = CM.EvaluateSymbol (c, &CumFreq, counter);
         /*запомним в стеке КМ и указатель на счетчик для
            последующего обновления модели
         */
         Stack.Push (CM, counter);
         // закодируем c или символ ухода
         StatCoder.Encode (CM, CumFreq, counter);
         order--;
      }while ( ! success );
      /*обновим модель: добавим КМ в случае необходимости,
         изменим значения счетчиков и т.д.
      */
      UpdateModel (Stack);
      // обновим контекст: сдвинем влево, справа добавим c
      MoveContext (c);
   }
```

Пример реализации PPM-компрессора

Рассмотрим основные моменты реализации компрессора PPM для простейшего случая с порядком модели $N = 1$ без исключения символов. Будем также исходить из того, что статистическое кодирование выполняется арифметическим кодером.

При контекстном моделировании 1-го порядка нам не требуются сложные структуры данных, обеспечивающие эффективное хранение и доступ к информации отдельных КМ. Можно просто хранить описания КМ в одномерном массиве, размер которого равен количеству символов в алфавите входной последовательности, и находить нужную КМ, используя символ ее контекста как индекс. Мы используем байт-ориентированное моделирование, поэтому размер массива для контекстных моделей порядка 1 будет равен 256. Чтобы не плодить лишних сущностей, мы, во-первых, откажемся от КМ(–1) за счет соответствующей инициализации КМ(0), и, во-вторых, будем хранить КМ(0) в том же массиве, что и КМ(1). Считаем, что КМ(0) соответствует индекс 256.

В структуру контекстной модели `ContextModel` включим массив счетчиков `count` для всех возможных 256 символов. Для

символа ухода введем в структуру КМ специальный счетчик esc, а также добавим поле TotFr, в котором будет содержаться сумма значений счетчиков всех обычных символов. Использование поля TotFr не обязательно, но позволит ускорить обработку данных.

С учетом сказанного структуры данных компрессора будут такими.

```
struct ContextModel{
   int  esc,
        TotFr;
   int  count[256];
};
ContextModel cm[257];
```

Если размер типа int равен 4 байтам, то нам потребуется не менее 257 кбайт памяти для хранения модели.

Опишем стек, в котором будут храниться указатели на требующие модификации КМ, а также указатель стека SP и контекст context.

```
ContextModel *stack[2];
int          SP,
             context [1];
                //контекст вырождается в 1 символ
```

Больше никаких глобальных переменных и структур данных нам не нужно.

Инициализацию модели будем выполнять в общей для кодера и декодера функции init_model.

```
void init_model (void){
   /*Так как cm является глобальной переменной, то
      значения всех полей равны 0. Нам требуется только
      распределить кодовое пространство в КМ(0) так,
      чтобы все символы, включая символ ухода, всегда бы
      имели ненулевые оценки.
      Пусть также символы будут равновероятными.
   */
   for ( int j = 0; j < 256; j++ )
```

```
      cm[256].count[j] = 1 ;
   cm[256].TotFr = 256;
   /*Явно запишем, что в начале моделирования мы считаем
      контекст равным 0. Число не имеет значения, лишь бы
      кодер и декодер точно следовали принятым
      соглашениям. Обратите на это внимание.
   */
   context [0] = 0;
   SP = 0;
}
```

Функции обновления модели также будут общими для кодера и декодера. В `update_model` производится инкремент счетчиков просмотренных КМ, а в `rescale` осуществляется масштабирование счетчиков. Необходимость масштабирования обусловлена особенностями типичных реализаций арифметического кодирования и заключается в делении значений счетчиков пополам при достижении суммы значений всех счетчиков `TotFr+esc` некоторого порога. Подробнее об этом рассказано в пункте «Обновление счетчиков символов».

```
const int MAX_TotFr = 0x3fff;
void rescale (ContextModel *CM){
   CM->TotFr = 0;
   for (int i = 0; i < 256; i++){
      /*обеспечим отличие от нуля значения
         счетчика после масштабирования
      */
      CM->count[i] -= CM->count[i] >> 1;
      CM->TotFr += CM->count[i];
   }
}

void update_model (int c){
   while (SP) {
      SP--;
      if ((stack[SP]->TotFr + stack[SP]->esc) >=
            MAX_TotFr)
         rescale (stack[SP]);
      if (!stack[SP]->count[c])
         /*в этом контексте это новый символ, увеличим
            счетчик уходов
         */
         stack[SP]->esc += 1;
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
      stack[SP]->count[c] += 1;
      stack[SP]->TotFr += 1;
   }
}
```

Собственно кодер реализуем функцией encode. Эта функция управляет последовательностью действий при сжатии данных, вызывая вспомогательные процедуры в требуемом порядке, а также находит нужную КМ. Оценка текущего символа производится в функции encode_sym, которая передает результаты своей работы арифметическому кодеру.

```
int encode_sym (ContextModel *CM, int c){
   // КМ потребует инкремента счетчиков, запомним ее
   stack [SP++] = CM;

   if (CM->count[c]){
      /*счетчик сжимаемого символа не равен нулю, тогда
        его можно оценить в текущей КМ; найдем
        накопленную частоту предыдущего в массиве count
        символа
      */
      int CumFreqUnder = 0;
      for (int i = 0; i < c; i++)
         CumFreqUnder += CM->count[i];
      /*передадим описание кодового пространства,
        занимаемого символом c, арифметическому кодеру
      */
      AC.encode (CumFreqUnder, CM->count[c],
                 CM->TotFr + CM->esc);
      return 1;  // возвращаемся в encode с победой
   }else{
      /*нужно уходить на КМ(0);
        если текущий контекст 1-го порядка встретился
        первый раз, то заранее известно, что его КМ
        пуста (все счетчики равны нулю), и кодировать
        уход не только не имеет смысла, но и нельзя,
        т.к. TotFr+esc = 0
      */
      if (CM->esc)
         AC.encode (CM->TotFr, CM->esc,
                    CM->TotFr + CM->esc)
      ;
      return 0; // закодировать символ не удалось
```

```
   }
}

void encode (void){
   int   c, // текущий символ
         success;  // успешность кодирования символа в КМ
   init_model ();
   AC.StartEncode ();   // проинициализируем арифм. кодер
   while (( c = DataFile.ReadSymbol() ) != EOF) {
      // попробуем закодировать в КМ(1)
      success = encode_sym (&cm[context[0]], c);
      if (!success)
         /*уходим на КМ(0), где любой символ получит
            ненулевую оценку и будет закодирован
         */
         encode_sym (&cm[256], c);
      update_model (c);
      context [0] = c; // сдвинем контекст
   }
   // закодируем знак конца файла символом ухода с КМ(0)

   AC.encode (cm[context[0]].TotFr, cm[context[0]].esc,
      cm[context[0]].TotFr + cm[context[0]].esc);
   AC.encode (cm[256].TotFr, cm[256].esc,
      cm[256].TotFr + cm[256].esc);
   // завершим работу арифметического кодера
   AC.FinishEncode();
}
```

Реализация декодера выглядит аналогично. Внимания заслуживает разве что только процедура поиска символа по описанию его кодового пространства. Метод `get_freq` арифметического кодера возвращает число x, лежащее в диапазоне [`CumFreqUnder`, `CumFreqUnder+CM->count[i]`), т.е. `CumFreqUnder` $\le x <$ `CumFreqUnder+CM->count[i]`. Поэтому искомым символом является i, для которого выполнится это условие.

```
int decode_sym (ContextModel *CM, int *c){
   stack [SP++] = CM;
   if (!CM->esc) return 0;

   int cum_freq = AC.get_freq (CM->TotFr + CM->esc);
   if (cum_freq < CM->TotFr){
      /*символ был закодирован в этой КМ; найдем символ и
```

```
      его точное кодовое пространство
   */
   int CumFreqUnder = 0;
   int i = 0;
   for (;;){
      if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
         CumFreqUnder += CM->count[i];
      else break;
      i++;
   }
   /*обновим  состояние арифметического кодера на
      основании точной накопленной частоты символа
   */
   AC.decode_update (CumFreqUnder, CM->count[i],
                     CM->TotFr + CM->esc);
   *c = i;
   return 1;
}else{
   /*обновим  состояние арифметического кодера на
      основании точной накопленной частоты символа,
      оказавшегося символом ухода
   */
   AC.decode_update (CM->TotFr, CM->esc,
                     CM->TotFr + CM->esc);
   return 0;
}
}


void decode (void){
   int  c,
        success;
   init_model ();
   AC.StartDecode ();
   for (;;){
      success = decode_sym (&cm[context[0]], &c);
      if (!success){
         success = decode_sym (&cm[256], &c);
         if (!success) break; //признак конца файла
      }
      update_model (c);
      context [0] = c;
      DataFile.WriteSymbol (c);
   }
}
```

Характеристики созданного компрессора, названного Dummy, приведены в пункте «Производительность на тестовом наборе Calgary Compression Corpus». Полный текст реализации Dummy оформлен в виде приложения 1.

Оценка вероятности ухода

На долю символов ухода обычно приходится порядка 30% и более от всех оценок, вычисляемых моделировщиком PPM. Это определило пристальное внимание к проблеме оценки вероятности символов с нулевой частотой. Львиная доля публикаций, посвященных PPM, прямо касаются оценки вероятности ухода (ОВУ).

Можно выделить два подхода к решению проблемы ОВУ: априорные методы, основанные на предположениях о природе сжимаемых данных, и адаптивные методы, которые пытаются приспособить оценку к данным. Понятно, что первые призваны обеспечить хороший коэффициент сжатия при обработке типичных данных в сочетании с высокой скоростью вычислений, а вторые ориентированы на обеспечение максимально возможной степени сжатия.

Априорные методы

Введем обозначения:

C — общее число просмотров контекста, т.е. сколько раз он встретился в обработанном блоке данных;

S — количество разных символов в контексте;

$S^{(i)}$ — количество таких разных символов, что они встречались в контексте ровно i раз;

$E^{(x)}$ — значение ОВУ по методу x.

Изобретатели алгоритма PPM предложили два метода ОВУ: так называемые метод A и метод B. Использующие их алгоритмы PPM были названы PPMA и PPMB соответственно.

В дальнейшем было описано еще 5 априорных подходов к ОВУ: методы C, D, P, X и XC [8, 10, 17]. По аналогии с PPMA и

PPMB, алгоритмы PPM, применяющие методы C и D, получили названия PPMC и PPMD соответственно.

Идея методов и их сравнение представлены в табл. 4.4 и табл. 4.5.

Таблица 4.4

Метод	$E^{(x)} =$
A	$\frac{1}{C+1}$
B	$\frac{S-S^{(1)}}{C}$
C	$\frac{S}{C+S}$
D	$\frac{S}{2C}$
P	$\frac{S^{(1)}}{C}-\frac{S^{(2)}}{C^2}+\frac{S^{(3)}}{C^3}-\dots$
X	$\frac{S^{(1)}}{C}$
XC	$\begin{cases}\frac{S^{(1)}}{C}, & при \quad 0<S^{(1)}<C \\ E^{(C)}, & в\ противном\ случае\end{cases}$

Кстати, в примере 2 был использован метод A, а в компрессоре Dummy — метод C.

При реализации метода B воздерживаются от оценки символов до тех пор, пока они не появятся в текущем контексте более одного раза. Это достигается за счет вычитания единицы из счетчиков. Методы P, X, XC базируются на предположении о том, что вероятность появления в обрабатываемых данных символа s_i подчиняется закону Пуассона с параметром λ_i.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Таблица 4.5

Тип файлов	Точность предсказания						
	Лучше			→			хуже
Тексты	XC	D	P	X	C	B	A
Двоичные файлы	C	X	P	XC	D	B	A

Места в табл. 4.5 очень условны. Так, например, при сжатии текстов методы XC, D, P, X показывают весьма близкие результаты, и многое зависит от порядка модели и используемых для сравнения файлов. В большинстве случаев существенным является только отставание точности ОВУ по способам A и B от других методов.

Упражнение: Выполните действия, описанные в примере 2, используя ОВУ по методу C. Если текущий символ ‘б’, то точность его предсказания улучшится, останется неизменной или ухудшится?

АДАПТИВНЫЕ МЕТОДЫ

Чтобы улучшить оценку вероятности ухода, необходимо иметь такую модель оценки, которая бы адаптировалась к обрабатываемым данным. Подобный адаптивный механизм получил название Secondary Escape Estimation (SEE), т.е. «дополнительной оценки ухода», или «вторичной оценки ухода». Метод заключается в тривиальном вычислении вероятности ухода из текущей КМ через частоту появления новых символов (или, что то же, символов ухода) в контекстных моделях со схожими характеристиками:

$$E^{(SEE)}(i) = \frac{f_i(esc)}{n_i},$$

где $f_i(esc)$ — число наблюдавшихся уходов из контекстных моделей типа i;
n_i — число просмотров контекстных моделей типа i.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Вразумительные обоснования выбора этих характеристик и критериев «схожести» при отсутствии априорных знаний о характере сжимаемой последовательности дать сложно, поэтому известные алгоритмы адаптивной оценки базируются на эмпирическом анализе типовых данных.

Метод Z

Одна из самых ранних попыток реализации SEE известна как метод Z, а использующая его разновидность алгоритма PPM — PPMZ [3]. Для точности описания этой техники SEE объект «контекст» ниже будет также именоваться «PPM-контекстом».

Для нахождения ОВУ строятся так называемые контексты ухода (escape contexts) КУ, формируемые из четырех полей. В полях КУ содержится информация о значениях следующих величин: последние четыре символа PPM-контекста, порядок PPM-контекста, количество уходов и количество успешных оценок в соответствующей КМ. Нескольким КМ может соответствовать один КУ.

Информация о фактическом количестве уходов и успешных кодирований во всех контекстных моделях, имеющих общий КУ, запоминается в счетчиках контекстной модели уходов КМУ, построенной для данного КУ. Эта информация определяет ОВУ для текущей КМ. ОВУ находится путем взвешивания оценок, которые дают три КМУ (КМУ порядка 2, 1 и 0), отвечающие характеристикам текущей КМ.

КУ порядка 2 наиболее точно соответствует текущей КМ, контексты ухода порядком ниже формируются главным образом путем выбрасывания части информации из полей КУ порядка 2. Компоненты КУ порядка 2 определяются в соответствии с табл. 4.6 [3].

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Таблица 4.6

Номер поля	Размер в битах	Способ формирования значения поля	
		Параметр и его значения	Значение поля
1	2	Порядок КМ	порядок КМ / 2 (с округлением до младшего);
2	2	Количество уходов из КМ	
		1	0
		2	1
		3	2
		> 3	3
3	3	Количество успешных оценок в КМ	
		0	0
		1	1
		2	2
		3, 4	3
		5, 6	4
		7, 8, 9	5
		10, 11, 12	6
		> 12	7
4	9	X_1 = семь младших битов последнего (только что обработанного) символа PPM-контекста; X_2 = шестой и пятый биты предпоследнего символа; т.е., если расписать байт как совокупность 8 битов xXXxxxxx, то это биты XX	((X_2&0x60) << 2) | X_1

В состав КУ всех порядков входят поля 1, 2, 3. Для КУ порядка 1 поле 4 состоит из 8 битов и строится из шестых и пятых битов последних четырех обработанных символов. У КУ порядка 0 четвертое поле отсутствует. Очевидно, что алгоритм по-

строения поля 4 для КУ порядков 1 и 2 призван улучшить предсказание ухода для текстов на английском языке в кодировке ASCII. Аналогичный прием, хотя и в не столь явном виде, используется в адаптивных методах ОВУ SEE-d1 и SEE-d2, рассмотренных ниже.

При взвешивании статистики КМУ(n) используются следующие веса w_n

$$\frac{1}{w_n} = e \cdot \log_2 \left(\frac{1}{e}\right) + (1-e) \cdot \log_2 \left(\frac{1}{1-e}\right),$$

где e — ОВУ, которую дает данная взвешиваемая КМУ(n); формируется из фактического количества уходов и успешных кодирований в контекстных моделях, соответствующих этой КМУ, или, иначе, определяется наблюдавшейся частотой ухода из таких КМ.

Окончательная оценка:

$$E^{(Z)} = \frac{\sum_{n=0}^{2} e_n w_n}{\sum_{n=0}^{2} w_n}.$$

После ОВУ выполняется поиск текущего символа среди имеющихся в КМ. По результатам поиска (символ найден или нет) обновляются счетчики соответствующих трех КМУ порядка 0, 1 и 2.

Методы SEE-d1 и SEE-d2

Современным адаптивным методом является подход, предложенный Шкариным и успешно применяемый в компрессорах PPMd и PPMonstr [1]. При каждой оценке используется КУ только какого-то одного типа (в методе Z их три), но, в зависимости от требований, предъявляемых к компрессору, автор предлагает применять один из двух методов. Назовем первый метод SEE-d1, а второй — SEE-d2.

Метод SEE-d1 используется в компрессоре PPMd и ориентирован на хорошую точность оценки при небольших вычислительных затратах. Рассмотрим его подробно.

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Все КМ разобьем на три типа:

- детерминированные (или бинарные), содержащие только один символ; назовем их контекстными моделями типа d;
- с незамаскированными символами, т.е. ни один из символов, имеющихся в данной КМ, не встречался в КМ б*о*льших порядков; обычно это КМ максимального порядка или КМ, с которой началась оценка; назовем их контекстными моделями типа nm;
- с замаскированными символами; назовем их контекстными моделями типа m.

Как показали эксперименты, вероятность ухода из КМ разных типов коррелирует с разными характеристиками.

Случай КМ типа d является наиболее простым, так как у такой модели малое число степеней свободы. Естественно, наибольшее влияние на вероятность ухода оказывает счетчик частоты единственного символа.

Ниже изложен способ формирования КУ, при этом описание полей отсортировано в порядке убывания степени их влияния на точность оценки ухода.

1) Счетчик частоты символа квантуется до 128 значений.
2) Существует сильная взаимосвязь между родительскими и дочерними КМ, поэтому число символов в родительской КМ квантуется до 4 значений;
3) При сжатии реальных данных характерно чередование блоков хорошо предсказуемых символов с плохо предсказуемыми. Включим в КУ оценку вероятности предыдущего символа чтобы отслеживать переходы между такими блоками. Величина квантуется до 2 значений.
4) Существует сильная статистическая взаимосвязь между текущим и предыдущим символами. Введем однобитовый флаг, принимающий значение 0, если два старших бита предыдущего символа нулевые, и значение 1 в прочих случаях.
5) Кроме чередования блоков хорошо и плохо предсказуемых символов, часто встречаются длинные блоки очень хорошо

предсказуемых данных. Это обычно бывает в случае множественных повторов длинных строк в сжимаемом потоке. Часто PPM-модели небольших порядков плохо работают в таких ситуациях. Введем однобитовый флаг, свидетельствующий о нахождении в таком блоке. Флаг принимает значение 1, если при обработке предыдущих символов ни разу не происходил уход, и оценки вероятности превышали 0.5 для L или большего количества этих символов. L обычно равно порядку PPM-модели.

6) Возможность ухода зависит от единственного символа КМ типа d. Пусть соответствующий флаг равен 0, если два старших бита символа нулевые, и 1 в остальных случаях.

Таким образом, всего возможно 128·4·2·2·2·2 = 8192контекстов ухода для КМ типа d.

В случае КМ типа nm адаптивная оценка затруднена из-за небольшой частоты их использования, что приводит к отсутствию представительной статистики в большинстве случаев. Поэтому применим полуадаптивный метод, доказавший на практике свою эффективность. Допустим, распределение символов геометрическое:

$$p(s_i^k \mid o) = p^k(1-p),$$

где $p(s_i^k \mid o)$ — вероятность появления символа s_i в заданной КМ(o) после серии из k иных символов.

Иначе говоря, $p(s_i^k \mid o)$ — вероятность успеха в первый раз после ровно k испытаний по схеме Бернулли при вероятности успеха $(1 - p)$.

Параметр p геометрического распределения может быть найден через оценку для соответствующей КМ типа d. Получив p, можно оценить частоту счетчика числа уходов. Это делается только для КМ, содержащих 1 символ, при добавлении нового символа, т.е. когда КМ перестает быть детерминированной. Далее значение счетчика символа ухода изменяется только при добавлении в КМ новых символов. Величина инкремента δ счетчика равна

$$\delta = \begin{cases} 1/2, \text{ при } 4S(o) < S(o-k) \\ 1/4, \text{ при } 2S(o) < S(o-k) \\ 0, \text{ для всех прочих случаев} \end{cases},$$

где $S(o)$ — число символов в модифицируемой КМ(o);
$S(o–k)$ — число символов в КМ($o–k$), в которой реально был оценен текущий символ.

Если новому символу соответствует небольшая вероятность, то δ увеличивается на $1– f^0(s_i \mid o)$, где $f^0(s_i \mid o)$ есть наследуемая частота нового символа s_i (см. подпункт «Наследование информации»).

Вероятность ухода из КМ типа m больше всего зависит от суммы значений всех счетчиков. Но эта сумма должна представляться достаточно точно, что приведет к большому количеству используемых КМУ и, как следствие, к недостатку статистики в каждой КМУ. Поэтому будем моделировать не вероятность ухода, а величину счетчика символа ухода, которая слабо зависит от суммы частот. Поля КУ формируются следующим образом.

1) Имеется сильная взаимосвязь между частотой уходов и числом незамаскированных символов; произведем квантование этого числа до 25 значений.
2) Результат сравнения числа незамаскированных символов $S(o) – S(o+1)$ в КМ(o) с числом символов $S(o+1)$ в дочерней КМ($o+1$) запишем в виде однобитового флага.
3) Аналогично полю 2 введем флаг результата сравнения числа незамаскированных символов $S(o) – S(o+1)$ с числом символов $S(o–1) – S(o)$, которые останутся незамаскированными в родительской КМ($o–1$), если текущая КМ(o) не сможет оценить обрабатываемый символ.
4) Существует сильная статистическая взаимосвязь между текущим и предыдущим символами. Введем однобитовый флаг, принимающий значение 0, если два старших бита

предыдущего символа нулевые, и значение 1 в прочих случаях.

5) Существует статистическая взаимосвязь между частотой уходов и средней частотой символов $\frac{f(o)}{S(o)+1}$ в КМ, включая символ ухода, где $f(o)$ есть сумма значений всех счетчиков текущей КМ(o).

Всего может использоваться до $25 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 400$ контекстов ухода для КМ типа m.

Метод SEE-d2, используемый в компрессоре PPMonstr, отличается от SEE-d1 следующим образом:

1) добавлено 4 поля в КУ для КМ типа d;
2) добавлено 2 поля в КУ для КМ типа m;
3) для КМ типа nm используется адаптивная оценка.

Данная модификация позволяет улучшить сжатие на 0,5–1%, но вычисление значений дополнительных полей существенно замедляет работу компрессора.

Как SEE-d2, так и SEE-d1 обычно эффективнее SEE по методу Z с точки зрения точности оценок и вычислительной сложности.

Пример реализации адаптивной ОВУ

Заменим в нашем компрессоре Dummy априорный метод ОВУ на адаптивный. С целью максимального упрощения контекст ухода будем формировать на основании только частоты появления контекста `TotFr`.

Рассмотрим как изменится программа. Для подсчета числа успешных кодирований и числа уходов создадим структуры данных:

```
struct SEE_item { //счетчики для контекста ухода
   int e, s;
};
int  TF2Index [MAX_TotFr+1]; //таблица квантования TotFr
SEE_item  *SEE;
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

Алгоритм квантования TotFr описывается следующим образом:

Значение TotFr	Номер КУ
1	1
2…3	2
4…7	3
8…15	4
…	…

Иначе говоря, по мере роста TotFr диапазон возможных значений TotFr группы КМ, относимых к одному и тому же КУ, в два раза больше предыдущего. Если MAX_TotFr = 0x3fff, то всего может использоваться до 14 КУ.

Изменим соответствующим образом функцию init_model.

```
void init_model (void){
   ... // ранее описанные действия по инициализации
   int  ind = 0,   //номер КУ
        i = 1,     //значение TotFr
        size = 1;  //размер диапазона
   do{
      int j = 0;
      do{
         TF2Index [i+j] = ind;
      }while (++j < size);
      i += j;
      size <<= 1;
      ind++;
   }while ((i + size) <= MAX_TotFr);
   for (; i <= MAX_TotFr; i++)
      TF2Index [i] = ind;
   /*на всякий случай отнесем КМ с TotFr = 0 к КУ
      с номером 0
   */
   TF2Index [0] = 0;
   SEE = (SEE_item*) new SEE_item[ind+1];
   //проинициализируем счетчики КМУ
   for (i = 0; i <= ind; i++) {
      SEE[i].e = 0;
      /* это предотвратит деление на 0, хотя и сместит
         оценку
      */
      SEE[i].s = 1;
```

```
   }
}
```

Функция encode_sym примет такой вид (изменения выделены жирным шрифтом):

```
int encode_sym (ContextModel *CM, int c){
   int  esc;
   stack [SP++] = CM;

   SEE_item  *E;
   E = calc_SEE (CM, &esc); //находим адаптивную ОВУ
   if (CM->count[c]){
      int CumFreqUnder = 0;
      for (int i = 0; i < c; i++)
         CumFreqUnder += CM->count[i];
      AC.encode (CumFreqUnder, CM->count[c],
                 CM->TotFr + esc);
      /* увеличиваем счетчик успешных кодирований для
         текущего КУ
      */
      E->s++;
      return 1;
   }else{
      if (CM->esc){
         AC.encode (CM->TotFr, esc, CM->TotFr + esc);
         //увеличиваем счетчик уходов для текущего КУ
         E->e++;
      }
      return 0;
   }
}
```

В функции декодирования символа decode_sym произведем аналогичные изменения.

```
int decode_sym (ContextModel *CM, int *c){
   stack [SP++] = CM;
   if (!CM->esc) return 0;
   int  esc;
   SEE_item  *E = calc_SEE (CM, &esc);

   int cum_freq = AC.get_freq (CM->TotFr + esc);
   if (cum_freq < CM->TotFr){
      int CumFreqUnder = 0;
```

```
      int i = 0;
      for (;;){
         if ( (CumFreqUnder + CM->count[i]) <= cum_freq)
            CumFreqUnder += CM->count[i];
         else break;
         i++;
      }
      AC.decode_update (CumFreqUnder, CM->count[i],
                        CM->TotFr + esc);
      *c = i;
      E->s++;
      return 1;
   }else{
      AC.decode_update (CM->TotFr, esc,
                        CM->TotFr + esc);
      E->e++;
      return 0;
   }
}
```

Зависимость между требуемым значением счетчика уходов, с одной стороны, и количеством уходов и успешных кодирований, с другой стороны, имеет вид

$$\frac{esc}{TotFr + esc} = \frac{e}{e + s},$$

откуда

$$esc = \frac{e}{s} \cdot TotFr\,.$$

Поэтому функцию `calc_SEE`, в которой собственно и осуществляется адаптивная ОВУ, реализуем так:

```
const int SEE_THRESH1 = 10;
const int SEE_THRESH2 = 0x7ff;
SEE_item* calc_SEE (ContextModel* CM, int *esc){
   SEE_item*  E = &SEE[TF2Index[CM->TotFr]];
   if ((E->e + E->s) > SEE_THRESH1){
      *esc = E->e * CM->TotFr / E->s; //адаптивная оценка
      if (!(*esc)) *esc = 1;
      if ((E->e + E->s) > SEE_THRESH2){
         E->e -= E->e >> 1;
         E->s -= E->s >> 1;
      }
```


Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

```
    }
    else *esc = CM->esc; //априорная оценка
    return E;
}
```

Константа `SEE_THRESH1` определяет порог частоты использования КУ, ниже которого предпочтительнее все же применение априорного метода оценки, т.к. не набралось еще значительного объема статистики для текущего КУ. Константа `SEE_THRESH2` налагает ограничение на значение счетчиков успешных кодирований `s` и ухода `e` чтобы, с одной стороны, предотвратить переполнение при умножении `E->e * CM->TotFr` и, с другой, улучшить адаптацию к локальным изменениям характеристик сжимаемого потока.

Предложенная реализация вычисления средней частоты уходов крайне неэкономна. Так, например, можно избежать умножения на `CM->TotFr`, т.к. обычно в пределах группы КМ, относимых к одному КУ, эта величина изменяется не сильно, поэтому имеет смысл заложить неявное умножение на соответствующую константу в сам алгоритм изменения счетчиков `e` и `s`. Практичная реализация адаптивной оценки среднего изложена в [1].

Также необходимо следить за величиной `esc`, поскольку достаточно большая сумма `CM->TotFr + esc` может привести к переполнению в арифметическом кодере. Мы не делали соответствующих проверок лишь с целью упрощения описания.

> **Упражнение:** Добавление адаптивного метода ОВУ требует изменения действий по кодированию знака конца файла. Предложите вариант такой модификации.

Если сравнивать с помощью CalgCC, то применение описанного метода адаптивной ОВУ улучшает сжатие всего лишь приблизительно на 0.3%. Небольшой эффект объясняется не только

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

простым механизмом вычисления ОВУ, но и малым порядком используемой модели.

> **Заключение:** Даже сложные адаптивные методы ОВУ улучшают сжатие обычно лишь на 1–2% по сравнению с априорными методами, но требуют существенных вычислительных затрат.

Обновление счетчиков символов

Модификация счетчиков после обработки очередного символа может быть реализована по-разному. После кодирования каждого символа естественно изменять соответствующие счетчики во всех КМ порядков 0, 1, …, N, что и предлагается, в частности, делать в алгоритмах PPMA и PPMB. Такой подход называется полным обновлением (full updates). Но в случае классического, не использующего наследование информации PPM, лучшие результаты достигаются когда счетчики оцененного символа увеличиваются только в КМ порядков o, o+1, …, N, где o — порядок КМ, в которой символ был закодирован. Иначе говоря, счетчик обработанного символа не увеличивается в какой-то активной КМ, если он был оценен в КМ более высокого порядка. Эта техника имеет самостоятельное название — исключение при обновлении (update exclusion).

Термин «исключение при обновлении» не следует путать с исключением (exclusion), под которым понимают сам механизм уходов с маскированием счетчиков тех символов, которые встречались в контекстах большего порядка.

Применение исключения при обновлении позволяет улучшить сжатие обычного PPM-компрессора примерно на 1–2% по сравнению с тем случаем, когда производится обновление счетчиков символа во всех КМ. Одновременно ускоряется работа компрессора. В случае применения наследования информации, а

также для алгоритма PPM* (описание PPM* приведено ниже), польза от исключения при обновлении не столь очевидна.

Роль контекстов-предков сравнительно небольших порядков значительно возрастает при использовании техники наследования информации, поэтому необходимо более быстрое обновление их статистики. Как показывают эксперименты, полное обновление работает все же плохо и в этом случае. Поэтому обычно следует использовать решение, промежуточное между исключением при обновлении и полным обновлением. Например, помимо увеличения с весом 1 в рамках реализации исключения при обновлении, имеет смысл инкрементировать с весом 1/(*o*–*i*+1) счетчики символа в контекстных моделях меньших порядков *i*. Под КМ(*i*) понимаются предки той КМ(*o*), в которой символ был оценен. Например, в компрессоре PPMd делается модификация счетчика с весом 1/2 только в родительской КМ и только в определенных случаях. При этом основное условие выполнения такой модификации требует, чтобы счетчик оцененного символа в КМ(*o*) был меньше некоторого порога.

В алгоритме PPM* применяется частичное исключение при обновлении (partial update exclusion). В этом случае производится увеличение счетчиков во всех так называемых детерминированных КМ, а если же их нет, то только в недетерминированной КМ с самым большим порядком. Под детерминированной понимается такая КМ, что в соответствующем ей контексте до данного момента встречался только один символ (любое число раз). Аналогично, такой контекст называется детерминированным.

Для собственно сжатия в связке с PPM практически всегда используется арифметическое кодирование. Увеличение значений счетчиков КМ может привести к ошибке переполнения в арифметическом кодере. Во избежание этого обычно производят деление значений пополам при достижении заданного порога. Максимальная величина порога определяется особенностями конкретной реализации арифметического кодера. С другой стороны, масштабирование счетчиков дает побочный эффект в ви-

де улучшения сжатия при кодировании данных с достаточно быстро меняющейся статистикой контекстных моделей. Чем нестабильнее КМ, тем чаще следует масштабировать ее счетчики, отбрасывая таким образом часть информации о поведении данной КМ в прошлом. В свете этого естественной является идея об увеличении счетчиков с неравным шагом, так чтобы недавно встреченные символы получали большие веса, чем «старые» символы. В качестве полумеры можно применять масштабирование счетчика последнего встреченного символа, которое эксплуатирует такую же особенность типичных данных (см. ниже подпункт «Масштабирование счетчика последнего встреченного символа»). Существенному улучшению сжатия в таких случаях также способствует вторичная оценка вероятности символов (см. далее по тексту подпункт «Увеличение точности предсказания наиболее вероятных символов»).

Использование в качестве шага прироста счетчиков величин, больших единицы, необходимо для успешной работы сложных методов обновления, а также способствует лучшей адаптации модели при масштабировании. В качестве добавки веса 1 хорошо работают 4 или 8, при этом все еще отсутствует проблема переполнения даже при использовании для счетчиков 16-битных машинных слов. Например, если шаг прироста = 4, то счетчик символа может принимать значения: 4 (инициализация при первом появлении символа в контексте), 8, 12, 16… В компрессоре Dummy используется единичный шаг прироста.

Повышение точности оценок в контекстных моделях высоких порядков

Наряду с задачей оценки вероятности ухода серьезной проблемой PPM является недостаточный объем статистики в КМ высоких порядков, что приводит к большим погрешностям оценок. Как побочный результат имеется неприятная зависимость порядка обычной PPM-модели, обеспечивающего наилучшее сжатие, от вида данных. Как правило, оптимальный порядок обычной модели колеблется от 0 до 16 (для текстов в районе 4–6), кроме того, часто существуют значительные локальные изме-

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

), кроме того, часто существуют значительные локальные изменения внутри файла. Например, на рис. 4.3. приведен типичный для классического PPM-алгоритма график зависимости степени сжатия текста от порядка модели. Видно, что максимум достигается при $N = 4…5$, после чего наблюдается плавное падение степени сжатия.

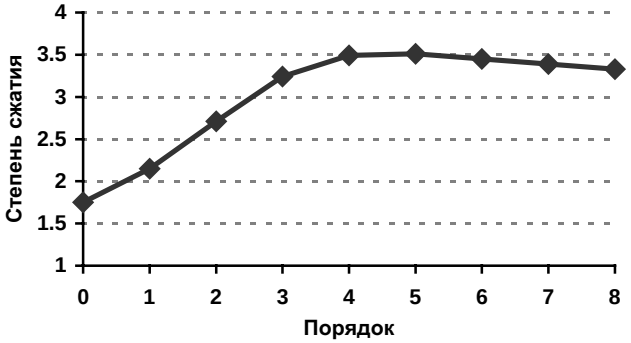


Рис. 4.3. Зависимость степени сжатия от порядка модели для классического PPM-алгоритма

Можно выделить 2 подхода к решению проблемы:

- выбор порядка модели, обеспечивающего наилучшее сжатие, при оценивании каждого символа;
- использование статистики контекстов-предков.

Выбор локального порядка модели

Механизм выбора порядка модели для кодирования каждого символа получил название Local Order Estimation (LOE) — «оценка локального порядка». Схемы LOE носят чисто эвристический характер и заключаются в том, что мы задаем некую функцию «доверия» и пробуем предсказать с ее помощью эффективность кодирования текущего символа в той или иной доступной — соответствующей активному контексту и физически существующей — КМ порядка от 0 до заданного N. Можно выделить три типа схем LOE:

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

1) ищем оптимальный порядок сверху вниз от КМ максимального порядка N к КМ минимального порядка; прекращаем поиск, как только КМ меньшего порядка кажется нам более «подозрительной», чем текущая, которую и используем для оценки вероятности символа;
2) поиск снизу вверх;
3) оценка всех доступных КМ.

Если в выбранной КМ закодировать символ не удалось, то, вообще говоря, процедуру поиска следующей оптимальной по заданному критерию КМ можно повторить. Тем не менее, обычно ищут только начальный порядок, а в случае ухода просто спускаются на уровень ниже, то есть дальше используют обычный алгоритм PPM.

Выбор той или иной функции доверия зависит от особенностей конкретной реализации PPM, характеристик данных, для сжатия которых разрабатывается компрессор, и, как нередко бывает, личных пристрастий разработчика. Как показал опыт, различные «наивные» энтропийные оценки надежности КМ работают плохо. Эти оценки основываются на сравнении средней длины кодов для текущей КМ(o) и родительской КМ(o–1). Неудача объясняется, видимо, тем, что в силу чистой случайности функция распределения в КМ(o) может быть более плоской, чем в следующей рассматриваемой КМ(o–1). Поэтому для получения правдоподобной оценки надо сравнивать среднюю длину кодов для всех дочерних контекстных моделей текущей КМ(o) со средней длиной кодов для всех дочерних контекстных моделей родительской КМ(o–1).

В [3] был предложен эффективный и простой метод, дающий оценку надежности КМ исходя из оценки вероятности Q наиболее вероятного в КМ символа (Most Probable Symbol's Probability, MPS-P) и количества уходов E из КМ. Обобщенно формулу можно записать так:

$$a \cdot Q \log_2 Q + b \cdot E(\log_2 E - c) + d \cdot (1-Q)(\log_2 E - c), \qquad (4.1)$$

где константы a, b, c, d подбираются эмпирическим путем.

Критерием выбора КМ является минимальное значение выражения (4.1) среди всех доступных КМ.

К счастью, оценка только по Q дает хорошие результаты уже в том случае, когда просто выбираем КМ с максимальным Q (соответствует варианту обобщенной формулы при $b = d = 0$).

Можно дать следующий пример, демонстрирующий недостатки «наивного» энтропийного подхода и подхода MPS-P. Пусть мы кодируем с помощью PPM-моделирования порядка 2 последовательность алфавита {‘0’, ‘1’}, порожденную источником с такими вероятностями генерации символов:

$p(0|00) = 0$, $p(1|00) = 1$,

$p(0|10) = p(1|10) = 0.5$.

Пусть появление строк “00” и “10” равновероятно. Если уже обработан большой блок входной последовательности, и контекст текущего символа равен “10”, то, в соответствии с заданным описанием источника, в КМ(2) оценки вероятностей

$q(0|10) = q(1|10) = 0.5$,

а в КМ(1) оценки составляют

$q(0|0) = 0.25$, $q(1|0) = 0.75$.

Средняя длина кодов для КМ(2) равна

$$-q(0|10)\log_2 q(0|10) - q(1|10)\log_2 q(1|10),$$

что составляет 1 бит. В то же время средняя длина для КМ(1) равна

$$-q(0|0)\log_2 q(0|0) - q(1|0)\log_2 q(1|0),$$

что соответствует примерно 0.8 бита. Поэтому, если пользуемся «наивным» энтропийным критерием, то мы должны выбрать для кодирования КМ(1). Критерий MPS-P также указывает на КМ(1). Этот выбор ошибочен. Действительно, если мы кодируем в КМ(2), то символы ‘0’ и ‘1’ требуют по $-\log_2 0.5 = 1$ биту. Если в КМ(1), то для кодирования ‘0’ нужно $-\log_2 0.25 = 2$ бита, а для ‘1’ требуется $-\log_2 0.75 \approx 0.4$ бита. В соответствии с принятым описанием источника вероятности появления ‘0’ и ‘1’ после строки “10” одинаковы, поэтому при каждом оценивании на основании статистики для контекста “0” вместо статистики

для контекста “10” мы будем терять в среднем 0.5·(2–1) + 0.5·(0.4–1) = 0.2 бита.

Добавим, что известные варианты реализации подхода LOE плохо сочетаются в смысле улучшения сжатия с механизмом наследованием информации, т.к. эти техники эксплуатируют примерно одинаковые недостатки классического алгоритма PPM.

НАСЛЕДОВАНИЕ ИНФОРМАЦИИ

Метод наследования информации позволяет существенно улучшить точность оценок. На конец 2001 года имеется по крайней мере одна очень эффективная реализация PPM, обеспечивающая высокую степень сжатия типовых данных, в особенности текстов, в первую очередь из-за применения наследования информации [1].

Метод наследования информации, предложенный Шкариным в [1], борется с неточностью оценок символов в КМ больш*и*х порядков и основан на очень простой идее. Логично предположить, что распределения частот символов в родительских и дочерних КМ похожи. Тогда при появлении в дочерней КМ(o) нового символа s_i целесообразно инициализировать его счетчик $f(s_i|o)$ некоторой величиной $f^0(s_i|o)$, зависящей от информации о данном символе в родительской КМ (или нескольких контекстных моделях-предках). Пусть в результате серии уходов мы спустились с КМ(o) на КМ(o–k), где символ и был успешно оценен. Тогда начальное значение счетчика символа в КМ(o) разумно вычислять исходя из равенства

$$\frac{f^0(s_i|o)}{f(o)} = \frac{f(s_i|o-k)}{f(o-k)+F_{o-k,o}}, \qquad (4.2)$$

где $f^0(s_i|o)$ — *наследуемая частота*;

$f(o)$ — сумма частот всех символов КМ(o), включая символ ухода;

$F_{o-k,o}$ — сумма частот всех символов, реально встреченных в контекстах порядков o–k+1 … o.

$$F_{o-k,o} = \sum_{m=o-k+1}^{o} \left(f(m) - f(esc \mid m) - \sum_{j=1}^{S(m)} f^{0}(s_j \mid m) \right),$$

где $f(esc|m)$ — частота для символа ухода в КМ(m);
$S(m)$ — количество символов в КМ(m), не включая символ ухода.

Упражнение: Объясните смысл слагаемого $F_{o-k,o}$ в выражении (4.2).

Выражение (4.2) обладает большой вычислительной сложностью и требует существенных затрат памяти для организации вычисления. Кроме того, имеет смысл учитывать только статистику КМ(o–k), так как все равно в большинстве случаев k равно 1, т.е. успешное кодирование происходит в родительской КМ, если же это не так, то, скорее всего, контексты порядков o–1 … o–k+1 являются «молодыми», и для них еще не накоплено полезной статистики. Поэтому на практике целесообразно использовать приближенную формулу

$$f^{0}(i \mid o) = \frac{f(o) \cdot f(s_i \mid o-k)}{f(o) - s(o) + f(o-k) - f(s_i \mid o-k)}.$$

При самой простой реализации величина $f^{0}(s_i \mid o)$ вычисляется и присваивается сразу при создании нового счетчика в КМ(o). Но можно отложить наследование частоты до следующего появления символа s_i в этом контексте порядка o. Вероятнее всего, к тому времени родительская КМ будет обладать б*о*льшим объемом статистики, что должно дать более точную оценку $f^{0}(s_i \mid o)$ и, в конечном итоге, улучшить сжатие. Такое «отложенное» наследование требует повторного поиска родительской КМ и символа s_i в ней, что может существенно замедлить обработку.

В зависимости от порядка модели, особенностей реализации и собственно сжимаемых данных наследование информации

позволяет улучшить сжатие примерно на 1…10%. Типичной является цифра порядка 2–4%.

> **Заключение:** Существующие реализации компрессоров, использующих механизм наследования информации, обеспечивают лучшее сжатие, чем компрессоры с LOE.

Различные способы повышения точности предсказания

Масштабирование счетчика последнего встреченного символа

Если в последний раз в текущем контексте был встречен некий символ *s*, то вероятность того, что и текущий символ также *s*, вырастает, причем часто значительно. Этот факт позволяет улучшить предсказание за счет умножения счетчика последнего встреченного символа (ПВС) на некий коэффициент. Судя по всему, впервые такая техника описана в [15], где она называется “recency scaling” — «масштабирование новизны».

Техника была исследована М.А. Смирновым при разработке компрессора PPMN. По состоянию на 2001 год неизвестны другие программы сжатия, использующие recency scaling.

В большинстве случаев хорошо работает множитель 1.1–1.2, то есть при таком значении наблюдается улучшение сжатия большинства файлов, и маловероятно ухудшение сжатия какого-то «привередливого» файла. Но часто оптимальная величина масштабирующего коэффициента колеблется в районе 2–2.5, так что можно улучшить оценку за счет адаптивного изменения множителя. Подходящее значение выбирается на основе анализа сжимаемых данных с помощью эмпирически полученных формул. Исследования показали, что величина наблюдаемой частоты совпадения последнего встреченного и текущего символов в недетерминированных контекстах позволяет подбирать коэффициент с хорошей точностью.

Для повышения точности оценки также следует учитывать расстояние между текущей позицией и моментом последней встречи текущего контекста. Если расстояние больше 5000 символов, то, скорее всего, масштабирование счетчика ПВС только повредит. Но следует понимать, что учет этой характеристики требует существенных затрат памяти.

Рассмотрим реализацию простейшего алгоритма recency scaling на примере нашего компрессора Dummy.

В описание структуры КМ внесем поле `last`, описывающее ПВС для заданного контекста.

```
struct ContextModel{
   int  esc,
        TotFr,
        last;
   int  count[256];
};
```

Значение `last` будем вычислять как сумму номера символа и 1, для того чтобы нулевое значение `last` указывало на отсутствие необходимости выполнять масштабирование. В начале работы программы `last` будет равно 0, т.к. `cm` является глобальной переменной.

В начало функций `encode_sym` и `decode_sym` добавим действия по умножению счетчика ПВС на постоянный коэффициент 1.25.

```
int  addition;
if (CM->last){
   addition = CM->count[CM->last-1]>>2;
   CM->count[CM->last-1] += addition;
   CM->TotFr += addition;
}
```

Кроме того, добавим сразу же после вызова методов `AC.encode` и `AC.decode_update` условные операторы, исправляющие значения счетчиков на первоначальные.

```
if (CM->last){
```

```
   CM->count[CM->last-1] -= addition;
   CM->TotFr -= addition;
}
```

И, наконец, обеспечим правильное обновление `last` в функции `update_model`.

```
if (!stack[SP]->count[c])
   stack[SP]->esc += 1;
else stack[SP]->last = c+1;
```

Обращаем внимание, что необходимо контролировать величину `addition`, т.к. масштабирование может привести к переполнению в арифметическом кодере. Мы не реализовали эту проверку лишь с целью упрощения описания.

Описанная модификация компрессора позволяет улучшить степень сжатия CalgCC примерно на 0.5%. Но оптимальная величина коэффициента масштабирования существенно меняется от файла к файлу. Так, например, для файла Obj2 максимальное улучшение сжатия — на 6% — было отмечено при коэффициенте 4.

> **Вывод:** Масштабирование счетчика последнего встреченного символа (recency scaling) позволяет улучшить сжатие в среднем лишь на 0.5–1%, но может дать существенный выигрыш при обработке данных, статистические характеристики которых подвержены частым изменениям.

МАСШТАБИРОВАНИЕ В ДЕТЕРМИНИРОВАННЫХ КОНТЕКСТАХ

Известно, что методы ОВУ A, B, C и, в меньшей степени, другие рассмотренные априорные методы назначают завышенную вероятность ухода из детерминированных контекстов [15]. Это можно исправить, умножая счетчик единственного символа на определенный коэффициент. Такой прием был впервые описан в [15] под наименованием “deterministic scaling” — «детер-

министическое масштабирование». Нетрудно заметить, что этот механизм есть частный случай recency scaling.

Эффект от deterministic scaling увеличивается, если при этом используется частичное исключение при обновлении, а не обычное исключение при обновлении [15].

Deterministic scaling имеет смысл применять только в сочетании с априорными методами ОВУ, ведь чем точнее вычисляется вероятность ухода, тем пользы от этого масштабирования меньше.

Увеличение точности предсказания наиболее вероятных символов

Контексты большой длины встречаются редко, и статистика, набираемая в соответствующих КМ, обычно является недостаточной для получения надежной оценки даже в случае использования механизма наследования информации. Наиболее негативно это проявляется при предсказании наиболее вероятных символов (НбВС) и наименее вероятных символов (НмВС), так как нахождение их истинной частоты требует большого числа наблюдений. Очевидно, что при этом основную избыточность кодирования мы вносим из-за неточной оценки не НмВС, а НбВС в силу их более частой встречаемости.

В этих случаях естественным является учет информации родительской КМ, что можно рассматривать как переход от неявного взвешивания к явному.

Под НбВС будем здесь понимать символы с оценками вероятностей $p(s_i|o) \geq p(s_{mps}|o)/2$, где $p(s_{mps}|o)$ соответствует самому вероятному символу (most probable symbol) s_{mps} в текущей КМ. Частоты НбВС будем корректировать, если $f(o) < f(o–1)$. В этих случаях КМ(o) «молода», и частота $f(s_i|o)$ символа s_i еще, как правило, меньше частоты $f(s_i|o–1)$. Уточненное значение $f_{\text{скорр}}(s_i|o)$ счетчика представляет собой среднее взвешенное $f(s_i|o)$ и «приведенной» частоты $f'(s_i|o-1)$:

$$f_{скорр}(s_i \mid o) = \frac{f(o) \cdot f(s_i \mid o) + f(o-1) \cdot f'(s_i \mid o-1)}{f(o) + f(o-1)},$$

где $$f'(s_i \mid o-1) = f(s_i \mid o-1) \cdot \frac{f(o) - f(s_i \mid o)}{f(o-1) - f(s_i \mid o-1)}.$$

В случае использования наследуемых частот аналогичную коррекцию имеет смысл применять при наследовании.

Очевидно, что степень сжатия сильно зависит от точности предсказания НбВС, поэтому после выполнения коррекции частоты целесообразно делать адаптивную оценку вероятности символа по аналогии с SEE. Назовем такую технику Secondary Symbol Estimation (SSE) — «вторичная оценка символа». В компрессоре PPMonstr версии H вторичная оценка выполняется для КМ с замаскированными символами (КМ типа m) и недерминированных КМ с незамаскированными символами (КМ типа nm). Поля контекстов для SSE строятся следующим образом.

Для КМ типа nm:

1) частота $f(s_{mps}|o)$ самого вероятного символа, квантуемая до 68 значений;
2) однобитовый флаг, указывающий, что было произведено масштабирование счетчиков контекстной модели;
3) однобитовый флаг, хранящий результат сравнения порядка o текущей КМ со средним порядком контекстных моделей, в которых были закодированы последние 128 символов;
4) однобитовый флаг, принимающий значение 0, если два старших бита предыдущего обработанного символа нулевые, и значение 1 в прочих случаях (аналог поля 4 контекста ухода для КМ типа d в SEE-d1);
5) однобитовый флаг, равный 0, если два старших бита символа s_{mps} нулевые, и 1 в остальных случаях (аналог поля 6 КУ для КМ типа d в SEE-d1).

Для КМ типа m:

1) оценка вероятности $p(s_{mps}|o)$, квантуемая до 40 значений;

2) однобитовый флаг результата сравнения средней частоты замаскированных и незамаскированных символов;
3) однобитовый флаг, указывающий, что только один символ остался незамаскированным;
4) аналог поля 2 контекста SSE для КМ типа nm (см. предыдущий список);
5) аналог поля 4 в предыдущем списке.

Техника SSE разработана Шкариным и используется для уточнения предсказания НбВС в компрессоре PPMonstr версии H.

> **Заключение:** Применение SSE к НбВС дает выигрыш в районе 0.5–1% и особенно полезно при сжатии неоднородных данных, т.е. либо порожденных источником, быстро меняющим свои характеристики, либо состоящих из фрагментов, сгенерированных существенно отличающимися источниками.

Общий случай применения вторичной оценки символа

Естественной является идея применения SSE ко всем символам контекста. Действительно, этот прием обеспечивает существенное улучшение сжатия неоднородных данных.

Рассмотрим обобщенный алгоритм применения SSE подробнее[6]. Пусть символы КМ хранятся в виде упорядоченного списка, так что символ с наибольшей частотой записан первым, т.е. имеет ранг 1. Процедура оценки очередного символа *s* приобретает вид цепочки последовательных оценок альтернатив «да»/«нет»: «да» — символ с текущим рангом равен *s*, «нет» — символы различаются, необходимо увеличить ранг на единицу и продолжить поиск. Вероятность «да» для ранга *i* находится как

[6] Алгоритм разработан и реализован Д.Шкариным в ноябре 2001 года

$$Q_i(o) = q^{SSE}(s_i \mid o),$$

где $q^{SSE}(s_i \mid o)$ — оценка вероятности символа с рангом *i*, полученная с помощью SSE *после* оценивания символа с рангом *i*–1; вычисляется на основании обычной оценки $q(s_i \mid o)$ и значений $w_i(o)$ полей контекста для SSE.

$$q^{SSE}(s_i \mid o) = SSE(q(s_i \mid o), w_1(o), w_2(o), \ldots w_p(o)),$$

$$q(s_i \mid o) = \frac{f(s_i \mid o)}{f(o) - \sum_{r=1}^{i-1} f(s_r \mid o)}.$$

Например, пусть список символов текущей КМ(*o*) имеет вид ‘в’, ‘б’, ‘а’, ‘г’. Если обрабатывается символ ‘а’, то он будет оценен посредством цепочки ответов «нет», «нет», «да» и получит вероятность $(1 - Q_1(o))\cdot(1 - Q_2(o))\cdot Q_3(o)$.

Дальнейшее улучшение сжатия обеспечивается соединением SSE с техникой recency scaling. В этом случае символом с рангом 1 считается последний встреченный в контексте символ (ПВС), если, конечно, он не замаскирован. Такая «адаптация новизны» позволяет значительно улучшить приспособляемость модели к потоку данных с меняющимися вероятностными характеристиками.

Положительный эффект обеспечивает внесение в контексты для SSE флагов, указывающих на совпадение ПВС текущего контекста и контекстов-предков. Аналогично, присваивать ПВС ранг 1 лучше только в том случае, если он равен ПВС «ближайших» предков.

В табл. 4.7 приведено сравнение характеристик сжатия компрессора PPMonstr версии H и его модификации, использующей обобщенный алгоритм SSE в сочетании с адаптацией новизны. В сравнении были использованы файлы из наборов CalgCC и VYCCT. Все файлы, за исключением текстового Book1, содержат неоднородные бинарные данные или смесь текста и бинарных данных. Использовались модели 64-го порядка.

Таблица 4.7

Название файла	Размер сжатого файла, байтов		Улучшение сжатия, %	Падение скорости сжатия, раз
	для версии H	для модификации версии H		
Book1	205144	203669	0.7%	2.0
Fileware.doc	112111	100454	10.4%	2.5
Obj2	65093	58466	10.2%	2.1
OS2.ini	94218	83405	11.5%	2.2
Samples.xls	70912	64666	8.8%	2.0
Wcc386.exe	278045	255719	8.0%	3.3

Вывод: Применение SSE ко всем символам контекста в сочетании с адаптацией новизны значительно улучшает сжатие неоднородных данных. Это позволяет получать стабильно наилучшую степень сжатия по сравнению с другими универсальными методами (BWT, LZ) при обработке файлов самых разнообразных типов, а не только текстов. Ценой увеличения степени сжатия является падение скорости в 2 и более раз

PPM и PPM*

При фиксировании максимального порядка контекстов в районе
5–6 PPM даже без наследования информации дает отличные результаты на текстах, но не очень хорошо работает на высоко избыточных данных с большим количеством длинных повторяющихся строк. В середине 1990-х годов был предложен метод борьбы с этим недостатком [6]. Предложенный алгоритм, PPM* (произносится как «пи-пи-эм ста»), был основан на использовании контекстов неограниченной длины. Авторы алгоритма предложили следующую стратегию выбора максимального порядка на каждом шаге: максимальный порядок соответствует порядку самого короткого детерминированного контекста. Под

детерминированным понимается контекст, в котором до данного момента встречался только один символ (любое число раз). Если детерминированных контекстов нет, то выбирается самый длинный среди имеющихся. После выбора максимального порядка процедура оценки вероятности символа в алгоритме PPM* ничем не отличается от применяемой в алгоритмах обычного PPM, т.е. PPM-моделирования ограниченного порядка.

Реализация PPM*, описанная в [6], имела не впечатляющие характеристики: сжатие на уровне PPMC порядка 5, скорость кодирования, как утверждается, также сопоставима, но памяти расходуется значительно больше. Судя по всему, авторам очень хотелось доказать превосходство их схемы над другими методами PPM и стандартным PPMC в частности, наличие которого весьма сомнительно. Читатель может самостоятельно сравнить степень сжатия PPM* с другими алгоритмами PPM, пользуясь табл. 4.8 и 4.9.

В принципе, расходы памяти для PPM и PPM* могут быть одинаковы, что показано в [4].

> **Вывод:** Преимущество подхода PPM* над обычным PPM не очевидно

Достоинства и недостатки PPM

Вот уже в течение полутора десятков лет представители семейства PPM остаются наиболее мощными практическими алгоритмами с точки зрения степени сжатия. По-видимому, добиться лучших результатов смогут только более изощренные контекстные (в широком смысле) методы, которые, несомненно, будут появляться, так как производятся все более быстрые процессоры, а объем оперативной памяти ЭВМ становится все больше.

Наилучшие результаты алгоритмы PPM показывают на текстах: отличный коэффициент сжатия при высокой скорости, чему наглядным примером являются компрессоры PPMd и

PPMonstr. Кроме того, если стоит задача максимизации степени сжатия определенных данных, то, скорее всего, PPM-подобный алгоритм будет наилучшим выбором в качестве основы специализированного компрессора.

Если выйти за рамки частной проблемы сжатия данных, то несомненным достоинством PPM является возможность получения хорошей статистической модели обработанной последовательности качественных данных (или сгенерировавшего ее источника). Действительно, модель, позволяющую эффективно предсказывать неизвестные символы сообщения, можно применять не только для сжатия, но и для решения задач коррекции текста в системах OCR, распознавания речи, классификации типа текста, семантического анализа текста, шифрования [18].

Недостатки реализаций подхода PPM заключаются в следующем:

- медленное декодирование (обычно на 5–10% медленнее кодирования);
- несовместимость кодера и декодера в случае изменения алгоритма оценки; в то же время алгоритмы семейства LZ77 допускают серьезную модификацию кодера без необходимости исправления декодера;
- медленная обработка мало избыточных данных (скорость может падать в разы);
- наилучшее сжатие различных файлов достигается при порядках модели PPM в районе 4…12 для моделей, не применяющих технику наследования информации и/или LOE, и при порядках 16…32 в противном случае; поэтому при выборе какого-то фиксированного порядка модели мы можем терять либо в степени сжатия, либо использовать чересчур много ресурсов ЭВМ;
- в общем случае недостаточно хорошее сжатие файлов, статистические характеристики которых подвержены частым изменениям такого типа, что оценки распределений вероятностей в контекстных моделях быстро устаревают (так называемая нестабильность статистик

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

контекстов); с точки зрения такой адаптации обычные алгоритмы PPM уступают алгоритмам типа LZ77, хотя известны способы ослабления или, вообще, устранения этого неприятного эффекта (см. подпункт «Общий случай применения вторичной оценки символа»);

- большие запросы памяти — десятки мегабайтов — в случае использования сложных моделей высокого порядка в сочетании с симметричностью алгоритма препятствуют организации эффективного доступа к сжатым данным;
- заметный проигрыш в эффективности по сравнению с алгоритмами типа LZ77 при сжатии файлов, имеющих длинные повторяющиеся блоки символов.

Практически всегда можно подобрать и настроить такую PPM-модель, или, точнее, контекстную модель с неявным взвешиванием, что она будет давать лучшее сжатие, чем LZ или BWT. Несмотря на это, применение PPM-компрессоров целесообразно главным образом для сжатия текстов на естественных языках и подобных им данных, поскольку при обработке малоизбыточных файлов велики временные затраты. Избыточные файлы с длинными повторяющимися строками (например, тексты программ) имеет смысл сжимать с помощью BWT-компрессоров и даже словарных компрессоров, так как соотношение сжатие-скорость-память обычно лучше. Для сильно избыточных данных предпочтительнее все-таки использовать PPM, так как методы LZ и BWT, особенно не использующие предобработку, работают при этом сравнительно медленно из-за деградации структур данных.

Характеристики алгоритмов семейства PPM:

Степени сжатия: определяются данными, для текстов обычно 3–4, для объектных файлов 2–3.

Типы данных: алгоритмы универсальны, но лучше всего подходят для сжатия текстов.

Книга "Методы сжатия данных". ISBN 5-86404-170-X
http://www.compression.ru/

Симметричность: близка к 1; обычно декодер немного медленнее кодера.

Характерные особенности: медленная обработка мало избыточных данных.

Компрессоры и архиваторы, использующие контекстное моделирование

HA

Программа HA явилась, пожалуй, первым публично доступным архиватором, использующим контекстное моделирование. Не исключено, что HA стал бы очень популярным архиватором, если бы его автор, Гарри Хирвола (Hirvola), не прекратил работать над проектом.

В HA реализованы алгоритм семейства LZ77 и алгоритм типа PPM.

Алгоритм PPM представляет собой хорошо продуманную модификацию классического PPMC. Метод OBY является априорным и основывает оценку ухода из KM на количестве имеющихся в ней символов с небольшой частотой. LOE не производится, последовательность спуска с KM высоких порядков является обычной. Максимальный порядок KM равен 4, минимальный — минус единице. Для организации поиска KM применяются хеш-цепочки. Хеширование осуществляется по символам контекста и его длине.

Результаты тестирования на CalgCC, представленные в табл. 4.8, получены для версии 0.999c. Сжатие файлов осуществлялось с помощью метода 2 программы, который как раз и соответствует PPM.

Архиватор разрабатывался для работы в MS DOS, и размер используемой памяти ограничен примерно 400 кбайт, что, вообще говоря, мало для модели 4 порядка, поэтому сжатие мож-

но существенно улучшить за счет увеличения объема доступной памяти.

CM

CM Булата Зиганшина (Ziganshin) является компрессором, применяющим блочно-адаптивное контекстное моделирование.

Алгоритм работы кодера следующий. Читается блок входных данных, по умолчанию до 1 Мбайт, и на основании его статистики строится модель заданного порядка *N*. Модель сохраняется в компактном виде в выходном файле, после чего с ее использованием кодируется сам считанный блок данных. Затем, если еще не весь входной файл обработан, производятся аналогичные действия для следующего блока и т.д.

Идея построения модели заключается в следующем:

- первоначально строится модель порядка *N*, содержащая статистику для всех встреченных в блоке контекстов длиной от 0 до *N*;
- из модели удаляются контекстные модели и счетчики символов с частотой меньше порога f_{min}, являющегося параметром алгоритма.

Дерево оставшихся КМ записывается в выходной файл, при этом описания символов и их частот в определенных КМ сжимаются на основании информации КМ-предков.

Оценка вероятности ухода из КМ при кодировании самих данных зависит от количества ее дочерних КМ, удаленных при «прочистке» модели. Собственно алгоритм оценки вероятности символов не отличается от классического PPM.

Программа написана достаточно давно и не отвечает современным требованиям на соотношение скорости и степени сжатия. Тем не менее, CM демонстрирует интересный подход, и при соответствующей доработке практическое использование такой техники может быть целесообразно.

Характеристики степени сжатия компрессора, приведенные в табл. 4.8, были получены при запуске программы с параметрами –o4 и –m10000000, т.е. был задан порядок $N = 4$, а максималь-

ный объем памяти для хранения модели был увеличен с 5 Мбайт, используемых по умолчанию, до 10 Мбайт, что обеспечило отсутствие переполнения при обработке всех файлов CalgCC.

RK и RKUC

С точки зрения коэффициента сжатия, разрабатываемый Малькольмом Тейлором (Taylor) архиватор RK является лучшим среди существующих на момент написания этой книги. Но достигается это не столько за счет очень хорошего PPM-компрессора, сколько благодаря большому количеству применяемых техник предварительного преобразования данных, позволяющих значительно улучшить сжатие файлов определенных типов. Именно грамотно реализованный препроцессинг и позволяет показывать RK стабильно хорошие результаты при сжатии таких типовых данных, как объектные файлы, файлы ресурсов, документы MS Word и таблицы MS Excel, тексты на английском языке.

В RK реализовано два алгоритма: статистический типа PPM и словарный типа Зива-Лемпела. В качестве PPM-компрессора в RK применяется облегченный вариант программы RKUC, созданной также Тейлором.

С учетом сказанного мы исключили RK из таблицы сравнения контекстных компрессоров по степени сжатия (табл. 4.8).

RKUC реализует контекстное моделирование с максимальным порядком 16. Порядок КМ может быть равен 16, 12, 8, 5, …, 0 и, вероятно, –1. Иначе говоря в RKUC используется отличающийся от классического механизм выбора порядка следующей КМ в случае ухода. В дополнение к этому в зависимости от параметров вызова программы может выполняться LOE.

Еще одна из опций компрессора разрешает использовать при оценке вероятности статистику, накопленную для разбросанных (sparse) контекстов, или бинарных (binary) в терминологии автора компрессора. Идея заключается в том, что несколько обычных контекстов одинаковой длины могут считаться одним

контекстом, если в определенных позициях их символы одинаковы. Например, если для контекстов длины 4 требуется совпадения первого[7] (последнего обработанного), второго и четвертого символов, то строки «абсд» и «аасд» являются одним и тем же разбросанным контекстом «аXсд», где X — любой символ. Таким образом, техника разбросанных контекстов заключается в объединении информации, собираемой для нескольких классических контекстов. Применение этого механизма часто позволяет заметно улучшить сжатие блоков данных с регулярной структурой.

В RKUC применяется улучшенный вариант адаптивной ОВУ по методу Z.

При тестировании использовался RKUC версии 1.04, который запускался с параметрами -m10 -o16 -x -b, т.е. использовалась модель 16 порядка, ограниченная 10 Мбайт памяти, и были включены механизмы LOE и разбросанных контекстов. Если отказаться от использования разбросанных контекстов, то степень сжатия текстовых файлов улучшается примерно на 1%, а бинарных (Geo, Obj1, Obj2) — ухудшается на несколько процентов.

PPMN

Компрессор PPMN разработан Максимом Смирновым (Smirnov) в экспериментальных целях. Основная цель разработки состояла в создании PPM-компрессора, обеспечивающего степень сжатия на уровне компрессоров, использующих разновидности алгоритма PPMZ, и значительно более высокую скорость кодирования при меньших ограничениях на объем используемой памяти.

В PPMN реализован алгоритм PPM с ограниченным порядком контекстов. Максимальный порядок N модели равен 7, но

[7] Символы контекста обычно нумеруются справа налево, поэтому первый символ контекста соответствует последнему обработанному

также реализованы варианты модели с «псевдопорядками» 8 и 9, при которых каждой КМ(*N*), $N = 8…9$, в действительности может соответствовать несколько контекстов порядка *N*. Иначе говоря, осуществляется смешивание статистики, набираемой для нескольких похожих контекстов.

Для ОВУ используется адаптивный механизм, который можно рассматривать как упрощенный метод Z. Используется только один тип контекста ухода, поля которого определяются:

- порядком КМ;
- количеством просмотров КМ;
- количеством символов в КМ;
- последним обработанным символом;
- однобитовым флагом, указывающем на то, что при кодировании строки последних обработанных символов заданной длины $L = 16$ ни разу не происходил уход.

Если предыдущий символ был закодирован в КМ меньшего порядка, чем у текущей рассматриваемой, или величина оценки вероятности ухода значительно меньше вычисленной для предыдущего символа, то производиться увеличение оценки. Обновление счетчиков контекстной модели уходов имеет следующую специфику: если рассматривается КМ максимального порядка среди имеющихся активных, т.е. не производился уход, то шаг увеличения счетчиков КМУ имеет вес 4, иначе — 1. Как показывают эксперименты, используемая схема превосходит метод Z по точности ОВУ и при этом требует меньших вычислительных затрат.

Как и в компрессоре RKUC, последовательность выбора порядка контекстной модели в случае ухода отличается от классической — часть порядков просто не используется. Например, при $N = 5$ PPM-модель состоит из контекстных моделей 5, 3, 2, 1, 0, –1 порядков, поэтому в случае ухода из КМ(5) рассматривается КМ(3). Такой прием позволяет ускорить обработку, а уменьшение степени сжатия либо незначительно, либо, наоборот, наблюдается ее увеличение.

Как уже упоминалось ранее, в PPMN используется масштабирование счетчика последнего встреченного символа. Улучшению сжатия неоднородных файлов также способствует ограничение на количество КМ порядков 1 и 2, приводящее к интенсивному обновлению этих КМ — статистика не используемых дольше всего контекстов просто удаляется.

В PPMN применяется упрощенный способ наследования информации: при добавлении символа в КМ(o) начальное значение его счетчика определяется частотой символа в той КМ(o–k), $k > 0$, в которой он был закодирован. Механизм инкремента счетчиков соответствует обычному исключению при обновлении.

По аналогии с НА, в качестве структуры данных для доступа к КМ используются хеш-цепочки. Для ускорения поиска контекстных моделей каждый символ КМ(N) имеет внешний указатель на КМ(N), соответствующую следующему символу обрабатываемого потока.

PPMN содержит механизмы предварительной обработки текстов на английском и русском языках и исполнимых файлов для процессоров типа Intel x86 (см. главу «Предварительная обработка данных»). Кроме того, имеются средства кодирования таблиц 8-и, 16-и и 32-х битных элементов, а также кодирования длин повторов (RLE). Параметры SEE, механизмов наследования информации и обновления счетчиков настроены на обеспечение наилучшей степени сжатия в режиме обработки текстов с использованием средств препроцессинга.

В табл. 4.8 указаны результаты тестирования PPMN версии 1.00 beta 1 в режиме сжатия без использования предварительной обработки (опция –da компрессора). Порядок модели равнялся 6 (опция –o6), а ее размер был ограничен 20 Мбайт.

PPMd и PPMonstr

Программы PPMd и PPMonstr Дмитрия Шкарина (Shkarin) реализуют разработанные им же алгоритмы PPMII и cPPMII (complicated PPMII, т.е. «усложненный» PPMII) [1]. Оба ком-

прессора используют механизм наследования информации. В PPMd применяется адаптивный метод ОВУ SEE-d1, а в PPMonstr — SEE-d2. Кроме этого отличия, в PPMonstr реализовано несколько дополнительных приемов улучшения сжатия, основными из которых являются отложенное наследование и улучшение точности предсказания наиболее вероятных символов.

PPMII является одним из алгоритмов, используемых в архиваторе RAR версии 3.

В тестировании были использованы версии H компрессоров. PPMd запускался с опцией –o8, а PPMonstr — с опцией –o1, что соответствует моделям 8 и 64 порядков. Максимальный размер модели был ограничен 20 Мбайт (параметр –m20) в обоих случаях, что предотвратило сброс статистики модели, осуществляемый при полном заполнении выделенного объема памяти.

PPMY

PPMY является экспериментальным компрессором, разрабатываемым Евгением Шелвиным (Shelwien). Данная программа относится к немногочисленному классу компрессоров, использующих полное смешивание.

При обработке каждого символа производится смешивание оценок распределений вероятностей из КМ всех доступных порядков от некоторого N до –1. Величина N ограничивается сверху значением одного из параметров программы. Если позволяет данное ограничение, то в качестве КМ максимального порядка выбирается КМ с самым длинным контекстом, ранее встречавшимся по крайней мере один раз. Компрессор осуществляет моделирование на уровне байтов, поэтому КМ(–1) содержит счетчики для всех 256 возможных символов; значение каждого счетчика равно 1. Кроме счетчиков встречаемых в ее контексте символов, каждая КМ(o) содержит еще две переменные:

- число символов в контексте; будем обозначать здесь как $f(esc|o)$; принятое обозначение характеристики обусловлено тем, что ее значение совпадало бы с количеством

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

уходов из КМ(o), если бы мы имели дело с обычным алгоритмом PPM;

- число случаев, когда вероятность обрабатываемого символа в данной КМ(o) была максимальной по сравнению с прочими контекстными моделями; обозначим как $Opt(o)$.

Оценки вероятности символа, даваемые разными КМ некоторого порядка o, взвешиваются с помощью адаптивно вычисляемых весов w_o. Процедуру нахождения w_o можно описать следующим образом:

$$W_o^{(esc)} = \frac{\min(f(o), ET) - f(esc \mid o)}{f(o)},$$

$$W_o^{(opt)} = \frac{(w^{(opt)} + Opt(o))}{w^{(opt)} + f(o)},$$

$$w_o = w^{(E)} \cdot W^{(opt)} + (1 - w^{(E)}) \cdot W_o^{(esc)} \cdot W_o^{(opt)},$$

где $f(o)$ — общее количество просмотров соответствующей КМ(o);
ET, $w^{(opt)}$, $w^{(E)}$ — некоторые параметры.

Коэффициентам $W_o^{(esc)}$ и $W_o^{(opt)}$ можно дать такую упрощенную интерпретацию:

- $W_o^{(esc)}$ — это вероятность того, что символ присутствует в текущей КМ(o), или, в терминах классического PPM, вероятность того, что ухода не будет;
- $W_o^{(opt)}$ — это вероятность того, что именно данная КМ(o) имеет максимальную оценку вероятности текущего символа.

Алгоритм нахождения значений параметров ET, $w^{(opt)}$, $w^{(E)}$, обеспечивающий хороший коэффициент сжатия, был получен эмпирическим путем, и результаты вычислений сведены в таблицы. Для каждой КМ(o) величины параметров выбираются из соответствующей таблицы по адресу, определяемому значением пары (o, N).

Вообще говоря, в версии PPMY 3b используется до 120 специально подобранных параметров. Возможно, использование других функций построения весов позволит уменьшить количество параметров без ущерба для степени сжатия.

В табл. 4.8 приведены результаты сжатия набора файлов CalgCC для PPMY версии 3b. Кодер запускался с опцией /o64.

Производительность на тестовом наборе Calgary Compression Corpus

Все рассмотренные компрессоры и архиваторы были протестированы на наборе CalgCC, описанном в пункте «Сравнение алгоритмов по степени сжатия». Для сравнения добавлены характеристики тривиального компрессора Dummy, на примере которого мы объясняли идею PPM.

В таблице указаны степени сжатия отдельных файлов набора, средняя степень сжатия, общее время кодирования $T_{код}$ и декодирования $T_{дек}$. Средняя степень вычислялась как средняя не взвешенная по размеру файлов степень сжатия. Для $T_{код}$ и $T_{дек}$ за единицу принято время сжатия всего CalgCC компрессором PPMd. Чтобы внести ясность, заметим, что единица примерно соответствует скорости кодирования 700 кбайт/с для ПК с процессором типа Pentium III 733 МГц.

Таблица 4.8

	Dummy	CM	HA	PPMY	RKUC	PPMN	PPMd	PPMonstr
Bib	2.31	3.01	4.12	4.55	4.55	4.57	4.62	4.76
book1	2.22	2.99	3.27	3.72	3.62	3.64	3.65	3.74
book2	2.12	3.19	3.74	4.35	4.30	4.31	4.35	4.49
Geo	1.68	1.74	1.72	1.74	2.12	1.96	1.84	1.92
News	1.92	2.52	3.05	3.60	3.57	3.58	3.62	3.74
obj1	1.76	1.69	2.19	2.09	2.25	2.32	2.26	2.29
obj2	1.94	2.32	3.07	3.46	3.79	3.65	3.62	3.79
Paper1	2.08	2.37	3.39	3.59	3.51	3.59	3.65	3.74
Paper2	2.20	2.61	3.43	3.67	3.57	3.62	3.67	3.77
Pic	9.41	9.30	10.00	10.26	10.67	11.01	10.53	11.43
Progc	2.06	2.27	3.36	3.51	3.45	3.54	3.60	3.70
Progl	2.40	3.07	4.68	5.30	5.37	5.26	5.44	5.76

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

	Dummy	CM	HA	PPMY	RKUC	PPMN	PPMd	PPMonstr
Progp	2.36	2.99	4.71	5.33	5.30	5.19	5.26	5.76
Trans	2.28	2.96	5.23	6.35	6.40	6.17	6.35	6.84
Итого	2.62	3.07	4.00	4.39	4.46	4.46	4.46	4.70
$T_{код}$	0.5	2.4	3.0	14.6	6.5	1.9	1.0	2.3
$T_{дек}$	0.6	0.8	3.1	15.6	6.7	2.0	1.0	2.4

Другие архиваторы и компрессоры

Существует множество компрессоров, применяющих контекстное моделирование, которые не были охвачены данным обзором. Отметим следующие архиваторы:

- BOA, автор Ян Саттон (Sutton);
- ARHANGEL, автор Юрий Ляпко (Lyapko);
- LGHA, являющийся реализацией архиватора HA на языке Ассемблера, выполненной Юрием Ляпко;
- UHARC, автор Уве Херклоц (Herklotz);
- X1, автор Стиг Валентини (Valentini);

а также компрессор PPMZ, автор Чарльз Блум (Bloom).

Обзор классических алгоритмов контекстного моделирования

LOEMA

Судя по всему, впервые алгоритм контекстного моделирования был реализован в 1982 году Робертсом (Roberts) [2]. Автор назвал свой алгоритм Local Order Estimation Markov Analysis (Марковский анализ посредством оценивания локального порядка). В LOEMA используется полное смешивание оценок КМ различного порядка, при этом веса представляют собой значения уровня доверия к оценке в том смысле, как это понимается в математической статистике. Сравнение степени сжатия LOEMA с другими алгоритмами затруднено, т.к., с одной стороны, программа, реализующая алгоритм, не стала публично доступной, и, с другой стороны, файлы, на которых Робертс доказывал эффективность своего подхода, также не доступны. Имеются сто-

ронние отчеты, по которым LOEMA обеспечивает компрессию примерно на уровне PPMC (см. табл. 4.9) при значительно меньшей скорости, что выглядит вполне правдоподобно. Судя по всему, LOEMA Робертса позволял только оценивать, но не сжимать, т.е. выполнял исключительно статистическое моделирование.

DAFC

Алгоритм Double-Adaptive File Compression (Дважды адаптивное сжатие файлов), разработанный Лэнгдоном (Langdon) и Риссаненом (Rissanen) в 1983 году, сыграл серьезную роль в развитии контекстных методов [2]. Во-первых, в нем впервые, если не считать LOEMA, была реализована идея разделения процесса кодирования на моделирование и статистическое кодирование, а также идея одновременной адаптации структуры модели (т.е. набора КМ) и частот символов. Во-вторых, простота алгоритма обеспечивала возможность его реального применения при относительно скромных возможностях вычислительной техники 1980-х годов. В-третьих, DAFC полюбился научным работникам, охотно использовавшим его результаты для сравнения при написании статей.

В DAFC используются контекстная модель нулевого порядка и n контекстных моделей первого порядка. В начале сжатия используется КМ(0), в которой все символы алфавита обрабатываемой последовательности имеют отличные от нуля счетчики. По мере хода кодирования КМ(1) создаются только для первых n символов, встретившихся в уже обработанном блоке m раз. Из соображений экономии памяти авторы предлагали использовать $n = 31$ и $m = 50$. Далее, если текущий символ встречается в контексте C и для этого контекста существует КМ(1), то производится попытка закодировать символ в ней, иначе выдается символ ухода с вероятностью, оцениваемой по методу A, и символ кодируется в КМ(0).

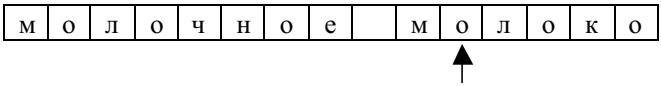
Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

В DAFC также применяется кодирование длин повторов (RLE), которое «запускается» при встрече последовательности из трех одинаковых символов.

> **Упражнение:** Сравните DAFC с алгоритмом работы компрессора Dummy. Если пренебречь RLE, то какие изменения следует внести в Dummy, чтобы получить реализацию DAFC?
>
> Пользуясь приведенными в тексте таблицами, сравните DAFC и Dummy по степени сжатия файлов набора CalgCC (см. табл. 4.8. и 4.9).

ADSM

Алгоритм Adaptive Dependency Source Model (Модель источника с адаптирующейся зависимостью) Абрахамсона (Abrahamson) представляет собой образец интересного подхода к реализации идеи контекстного моделирования [2]. Здесь осуществляется чистое контекстное моделирование 1 порядка, но собственно оценка строится на основании только одного распределения частот, общего для всех КМ. Достигается это следующим образом. В каждой КМ(1) счетчики символов хранятся в виде упорядоченного по величине частот списка. Счетчики ранжируются так, что символ с наибольшей частотой имеет наименьший ранг 1, а с наименьшей частотой — наибольший ранг. При обработке текущего символа находится его ранг (это может быть просто номер символа в упорядоченном списке символов текущей контекстной модели), и оценка определяется частотой использования этого ранга. Частоты рангов изменяются после кодирования каждого символа. Таким образом, статистическое кодирование осуществляется не на базе частоты *символа*, а на основании количества появлений символов с соответствующим *рангом* частоты. Рассмотрим сжатие строки “молочное_молоко”, начиная с отмеченного на рисунке стрелкой символа.



В контексте “м” реально встречался только один символ ‘о’, соответствующий текущему, поэтому мы кодируем ‘о’, опираясь на частоту *fr*(1) использования ранга 1. Затем *fr*(1) обновляется как *fr*(1)++. Следующий символ, ‘л’, кодируется в контексте “о”. Соответствующая КМ(1) содержит 3 реально наблюдавшихся символа — ‘л’, ’ч’, ’е’. Если принять, что в случае одинаковости частот наименьший ранг имеет последний встреченный символ, то ‘л’ кодируется на базе частоты *fr*(3) применения ранга 3. Производится обновление *fr*(3)++ и переход к обработке следующей буквы (‘о’).

Ни разу не встреченные в соответствующей КМ(1) символы, т.е. имеющие нулевую частоту, не трактуются как особый случай, им также присваивается какой-то ранг.

Были исследованы расширения алгоритма ADSM для второго и третьего порядков [9]. Результаты экспериментов показывают, что, в отличие от многих других классических алгоритмов контекстного моделирования, ADSM актуален и по сей день, так как обеспечивает хорошее соотношение скорости работы, объема используемой памяти и коэффициента сжатия. Так, например, техника ADSM использована в утилите безущербного сжатия изображений BMF, являющейся одной из лучших в своем классе.

> **Упражнение:** Сожмите строку с самого начала, найдите действительные значения *fr*(1) и *fr*(3), используемые при кодировании ‘о’ и ‘л’ в примере.

DHPC

Техника Dynamic-History Predictive Compression (Сжатие на основе предсказания по динамической истории), предложенная Уильямсом (Williams), интересна в сравнении с DAFC как ис-

пользующая иной способ ограничения роста модели. В этом алгоритме происходит создание КМ(*o*) только если родительская КМ(*o*+1) достаточно часто использовалась. Если представить контекст дочерней КМ в виде сцепления (конкатенации) *aC* какого-то символа *a* и контекста *C* родительской, в случае классического DAFC являющегося пустой строкой, то можно дать такую сравнительную характеристику. В DAFC образование дочерней КМ возможно в случае превышения частотой появления образующего контекст символа ‘a’ заданного порога, а в DHPC — при превышении порога частотой появления родительского контекста *C*, т.е. только «заслуженные» КМ могут иметь «детей».

При исчерпании доступной памяти рост модели DHPC прекращается, далее возможна адаптация только за счет изменения счетчиков символов.

Благодаря использованию КМ б*о*льших порядков DHPC дает лучшее сжатие, чем DAFC, но уступает по этому показателю классическим представителям семейства PPM (PPMC и PPMD).

Алгоритмы PPMA и PPMB

Алгоритмы PPMA и PPMB являются самыми ранними среди представителей семейства PPM [5]. Они были предложены авторами техники PPM одновременно в одной и той же статье и могут рассматриваться как единственные «чистокровные» PPM.

В PPMA и PPMB применяется ОВУ по методам A и B соответственно. После кодирования символа производится полное обновление счетчиков. Значения счетчиков символов не масштабируются, что требует достаточно больших счетчиков (авторы алгоритмов использовали 32-х битные).

WORD

Алгоритм WORD был создан Моффатом (Moffat) в конце 1980-х годов специально для сжатия текстов, и является ярким представителем особого семейства контекстных методов.

В алгоритме используются КМ не только для символов, но и для последовательностей (строк) конечной длины. Весь алфавит сжимаемого блока делится на «буквы» и «не-буквы». Последовательность букв называется «словом», а не-букв — «не-словом». Для оценивания применяются КМ 1 и 0 порядков, при этом буква предсказывается буквой, а слово — словом, аналогично для не-букв и не-слов. Если обрабатываемое слово ни разу не встречалось в КМ(1) для слов, то производится уход на уровень КМ(0); если же и там оценивание невозможно, т.е. такая строка встретилась впервые, то слово передается как последовательность букв. Для этого сначала кодируется длина слова, а затем составляющие его буквы с использованием КМ первого, нулевого и минус первого порядков. При оценке вероятностей букв используется техника уходов. Обработка не-слов и не-букв осуществляется аналогично. Таким образом, в WORD используется всего 12 типов КМ: первого и нулевого порядка для слов (не-слов), первого, нулевого и минус первого порядка для букв (не-букв), нулевого порядка для длин слов (не-слов).

Длина слова (не-слова) ограничивается 20 символами. Как и в случае DHPC, при достижении моделью заданного размера удаление всей структуры или ее части не производится, просто прекращается рост.

Сравнение алгоритмов контекстного моделирования

В табл. 4.9 представлены сведения о степени сжатия файлов набора CalgCC компрессорами, реализующими соответствующие алгоритмы контекстного моделирования. В первой строке указано название алгоритма, во второй, по необходимости, порядок использованной модели — строка «о-*N*» указывает, что использовалась модель порядка *N*. В строке «Итого» указана средняя не взвешенная по размеру файлов степень сжатия всего CalgCC.

Алгоритм cPPMII реализует механизм наследования информации и использует SEE-d2. Описание прочих алгоритмов было дано выше.

Таблица 4.9

	ADSM	DAFC	WORD	PPMC o-3	PPMC o-5	PPM*	PPMD o-5	cPPMII o-64
Bib	2.07	2.08	3.65	3.79	4.17	4.19	4.26	4.76
Book1	2.11	2.17	2.96	3.23	3.42	3.33	3.48	3.74
Book2	2.03	2.04	3.19	3.54	4.00	3.96	4.06	4.49
Geo	1.46	1.72	1.58	1.67	1.69	1.66	1.70	1.92
News	1.84	1.84	2.60	3.02	3.33	3.31	3.39	3.74
Obj1	1.60	1.55	1.78	2.13	2.14	2.00	2.14	2.29
Obj2	1.81	1.39	1.84	2.97	3.27	3.29	3.31	3.79
Paper1	1.96	1.90	3.10	3.23	3.38	3.38	3.42	3.74
Paper2	2.08	2.08	3.35	3.27	3.39	3.39	3.46	3.77
Pic	7.77	8.89	8.99	7.34	9.76	9.41	9.88	11.43
Progc	1.90	1.81	2.95	3.21	3.32	3.33	3.36	3.70
Progl	2.18	2.22	4.21	4.21	4.62	4.79	4.73	5.76
Progp	2.14	2.08	4.17	4.35	4.57	4.94	4.65	5.76
Trans	2.06	1.95	4.19	4.52	5.19	5.52	5.33	6.84
Итого	**2.36**	**2.41**	**3.47**	**3.61**	**4.02**	**4.04**	**4.08**	**4.70**

Таким образом, изощренные модели большого порядка обеспечивают лучшее сжатие данных, но разница в производительности схем обычно составляет лишь десятки, а то и единицы процентов. Поэтому обоснованный выбор алгоритма моделирования следует делать на базе комплексной оценки, включающей также объем используемой памяти, скорости кодирования и декодирования, и, конечно же, вычисляемой именно для тех данных, которые требуется сжимать.

Другие методы контекстного моделирования

Среди нерассмотренных остался интересный метод универсального сжатия Context Tree Weighting («Взвешивание контекстного дерева»), или CTW, который потенциально обеспечивает лучшую степень сжатия среди всех известных алгоритмов [16].

В CTW при оценке вероятности символа используется с некоторым весом статистика контекстных моделей-предков, т.е. производится явное взвешивание.

Контекстное моделирование ограниченного порядка хорошо работает на практике, обеспечивая высокую степень сжатия при терпимых требованиях к вычислительным ресурсам. Но оно представляет собой всего лишь один из типов контекстного моделирования в широком смысле. Можно отметить другие методы:

- модели состояний; в качестве конкретного алгоритма можно указать «Динамическое марковское сжатие» (Dynamic Markov Compression, или DMC) [4, 7];
- грамматические модели; конкретный алгоритм — SEQUITUR [11];
- модели с использованием искусственных нейронных сетей для построения предсказателя [14].

Рассмотрение алгоритмов моделирования данных видов выходит за рамки этой книги.

Вопросы для самоконтроля[8]

1. Объясните связь между точностью предсказания значений данных и степенью сжатия.
2. Что собой представляет модель источника данных в случае использования для моделирования PPM-алгоритма?
3. Почему технику уходов можно охарактеризовать как способ неявного взвешивания статистики контекстных моделей?
4. В каких случаях оценка вероятности ухода может равняться нулю?
5. Приведите пример блока данных, которые выгоднее сжимать, предсказывая вероятность символов на базе их без-

[8] Ответы к вопросам и упражнениям и исходные тексты программ вы можете найти на http://www.compression.ru/

Книга “Методы сжатия данных”. ISBN 5-86404-170-X
http://www.compression.ru/

условных частот, а не с помощью PPM-моделирования порядка 1.

6. Почему применение метода исключения всегда улучшает степень сжатия?
7. Как вы думаете, целесообразно ли применение метода наследования информации для обновления статистики контекстных моделей ухода?
8. Почему в случае использования наследования информации применение метода исключения при обновлении обычно не позволяет достигать потенциально возможной степени сжатия?
9. Почему метод наследования информации является потенциально более мощным способом компенсации недостатка статистики в контекстных моделях высоких порядков, чем метод выбора локального порядка?
10. В чем идея способов улучшения точности предсказания при обработке неоднородных данных?
11. Почему чем избыточнее со статистической точки зрения данные, тем скорость их обработки PPM-алгоритмами выше?
12. Укажите задачи, при решении которых может использоваться PPM-моделирование.

Литература

1. Шкарин Д. Повышение эффективности алгоритма PPM // Проблемы передачи информации, 34(3), с.44-54, 2001.
2. Bell T.C., Witten I.H., Cleary, J.G. Modeling for text compression // ACM Computer Surveys, 24(4), pp.555-591, 1989.
3. Bloom C. Solving the problems of context modeling // California Institute of Technology, 1996.
4. Bunton S. On-Line Stochastic Processes in Data Compression // PhD thesis, University of Washington, 1996.

5. Cleary J.G., Witten I.H. Data compression using adaptive coding and partial string matching // IEEE Transactions on Communications, Vol. 32(4), pp.396-402, April 1984.
6. Cleary J.G., Teahan W.J. Unbounded length contexts for PPM // The Computer Journal, 40 (2/3), pp. 67-75. 1997.
7. Cormack G.V., Horspool R.N. Data compression using dynamic Markov modelling // The Computer Journal 30(6), pp.541-550. Dec. 1987.
8. Howard P.G. The design and analysis of efficient lossless data compression systems // PhD thesis, Brown University, Providence, Rhode Island. 1993.
9. Lelewer D.A., Hirschberg D.S. Streamlining Context Models for Data Compression. // Proceedings of Data Compression Conference, Snowbird, Utah, pp.313-322, 1991.
10. Moffat A. Implementing the PPM data compression scheme // IEEE Transactions on Communications, Vol. 38(11), pp.1917-1921, Nov. 1990.
11. Nevill-Manning C.G., Witten I.H. Linear-time, incremental hierarchy inference for compression // Proceedings of Data Compression Conference, J.A. Storer and M. Cohn (Eds.), Los Alamitos, CA: IEEE Press, pp.3-11. 1997.
12. Rissanen J.J., Langdon G.G. Universal modeling and coding // IEEE Transactions on Information Theory, Vol. 27(1), pp.12-23, Jan. 1981.
13. Shannon C.E. A mathematical theory of communication // The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, July, October, 1948.
14. Schmidhuber J. Sequential neural text compression // IEEE Transactions on Neural Networks, Vol. 7(1), pp. 142-146. 1996.
15. Teahan W.J. Probability estimation for PPM // Proceedings of the New Zealand Computer Science Research Students Conference, 1995. University of Waikato, Hamilton, New Zealand.

16. Willems F.M.J., Shtarkov Y.M., Tjalkens T.J. The context-tree weighting method: Basic properties // IEEE Transactions on Information Theory, Vol. 41(3): pp. 653-664, May 1995.
17. Witten I.H., Bell T.C. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression // IEEE Transactions on Information Theory, Vol. 37(4), pp. 1085-1094. July 1991.
18. Witten I.H., Bray Z., Mahoui M., Teahan B. Text mining: a new frontier for lossless compression // University of Waikato, NZ, 1999.

Список архиваторов и компрессоров

1. Bloom C. (1996) PPMZ — PPM Based Compressor — Win95/NT version. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmz_ntx.zip
2. Herklotz U. (2001) UHARC — high compression multimedia archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/uharc04.zip
3. Hirvola H. (1995) HA — file archiver utility. ftp://ftp.elf.stuba.sk/pub/pc/pack/ha0999.zip
4. Lyapko G. (2000) ARHANGEL — file archiving utility. ftp://ftp.elf.stuba.sk/pub/pc/pack/arh140.zip
5. Lyapko G. (1997) LGHA — archive processor. ftp://ftp.elf.stuba.sk/pub/pc/pack/lgha11g.zip
6. Shelwien E. (2001) PPMY — context modelling compressor. http://www.pilabs.org.ua/sh/ppmy_3b.zip
7. Shkarin D. (1999) BMF — lossless image compressor. ftp://ftp.elf.stuba.sk/pub/pc/pack/bmf_1_10.zip
8. Shkarin D. (2001) PPMDH — fast PPM compressor for textual data. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdh.rar
9. Smirnov M. (2002) PPMN — not so fast PPM compressor. http://www.compression.ru/ms/ppmnb1+.rar
10. Sutton I. (1998) BOA constrictor archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/boa058.zip
11. Taylor M. (2000) RK — file archiver. ftp://ftp.elf.stuba.sk/pub/pc/pack/rk104a1d.exe

12. Taylor M. (1999) RKUC — universal compressor.
ftp://ftp.elf.stuba.sk/pub/pc/pack/rkuc104.zip
13. Valentini S. (1996) X1 archiver.
ftp://ftp.elf.stuba.sk/pub/pc/pack/x1dos95a.zip
14. Ziganshin B. (1997). CM — static context modeling archiver.
http://sochi.net.ru/~maxime/src/cm.cpp.gz