

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

Методы сжатия данных

Раздел 1

Содержание книги:

Введение

Раздел 1. Универсальные методы сжатия

...

Глава 4. Методы контекстного моделирования

Глава 5. Преобразование Барроуза-Уилера

...

Раздел 2. Алгоритмы сжатия изображений

Раздел 3. Алгоритмы сжатия видео

Приложение 1

Приложение 2

ISBN 5-86404-170-X

2002

РАЗДЕЛ 1. МЕТОДЫ СЖАТИЯ БЕЗ ПОТЕРЬ	2
Глава 1. Кодирование источников данных без памяти	5
Глава 2. Кодирование источников данных типа «аналоговый сигнал»	5
Глава 3. Словарные методы сжатия данных	5
Глава 4. Методы контекстного моделирования	5
<i>Классификация стратегий моделирования</i>	9
<i>Контекстное моделирование</i>	11
<i>Алгоритмы RPM</i>	20
<i>Оценка вероятности ухода</i>	35
<i>Обновление счетчиков символов</i>	49
<i>Повышение точности оценок в контекстных моделях высоких порядков</i>	51
<i>Различные способы повышения точности предсказания</i>	57
<i>RPM и RPM*</i>	64
<i>Достоинства и недостатки RPM</i>	65
<i>Компрессоры и архиваторы, использующие контекстное моделирование</i>	68
<i>Обзор классических алгоритмов контекстного моделирования</i>	77
<i>Сравнение алгоритмов контекстного моделирования</i>	82
<i>Другие методы контекстного моделирования</i>	83
<i>Вопросы для самоконтроля</i>	84
<i>Литература</i>	85
<i>Список архиваторов и компрессоров</i>	87
Глава 5. Преобразование Барроуза-Уилера	88
<i>Введение</i>	88
<i>Преобразование Барроуза-Уилера</i>	89
<i>Методы, используемые совместно с BWT</i>	102
<i>Способы сжатия преобразованных с помощью BWT данных</i>	120
<i>Сортировка, используемая в BWT</i>	130
<i>Архиваторы, использующие BWT и ST</i>	141
<i>Заключение</i>	149
<i>Литература</i>	150
Глава 6. Обобщенные методы сортирующих преобразований	153
Глава 7. Предварительная обработка данных	153
<i>Выбор метода сжатия</i>	153
УКАЗАТЕЛЬ ТЕРМИНОВ	154

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

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

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

$$H = -\sum_k P_k \cdot H_k = -\sum_{k,i} P_k \cdot p_k(s_i) \log_2 p_k(s_i), \quad (16)$$

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 $n = 0, 1, 2, \dots$. 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

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

Глава 2. Кодирование источников данных типа «аналоговый сигнал»

Глава 3. Словарные методы сжатия данных

Глава 4. Методы контекстного моделирования

Применение методов контекстного моделирования для сжатия данных опирается на парадигму сжатия с помощью «универсальных 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**».

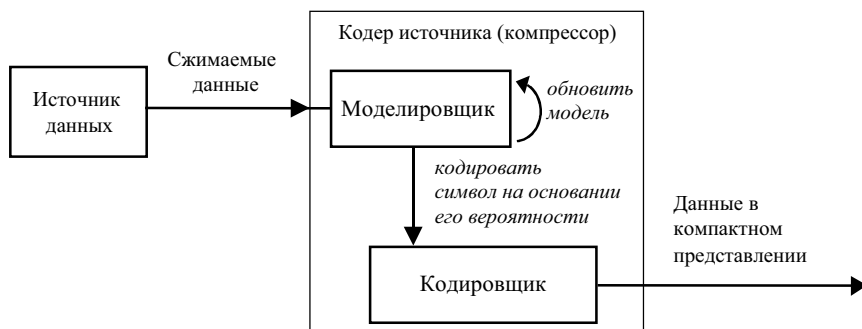


Рис. 4.1. Схема процесса сжатия данных в соответствии с концепцией универсальных моделирования и кодирования

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 сомнительной 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 после-

довательности 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: $p(‘0’) = 0.4$, $p(‘1’) = 0.6$. Пусть наша модель дает такие оценки вероятностей: $q(‘0’) = 0.35$, $q(‘1’) = 0.65$. Энтропия H источника равна

$$\begin{aligned} & -p(‘0’) \log_2 p(‘0’) - p(‘1’) \log_2 p(‘1’) = \\ & = -0.4 \log_2 0.4 - 0.6 \log_2 0.6 \approx 0.971 \text{ бита.} \end{aligned}$$

Если подходить формально, то «энтропия» модели получается равной

$$\begin{aligned} & -q(‘0’) \log_2 q(‘0’) - q(‘1’) \log_2 q(‘1’) = \\ & = -0.35 \log_2 0.35 - 0.65 \log_2 0.65 \approx 0.934 \text{ бита.} \end{aligned}$$

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, решающих 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.

Ожидание разрешения издательства

Пункт
«Классификация стратегий
моделирования»

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, большая 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 (пикселей, сэмплов, отсчетов, чисел) отмечается и при обработке дан-

ных 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 , $o \leq N$ » 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 моделиро-

ванием 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 “абсавббас”, 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 $o \leq N$, 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 “абсавббabc” иллюстрирует сразу две проблемы контекстного моделирования:

- как выбрать 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

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

где $w(o)$ — вес оценки КМ(o).

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

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

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

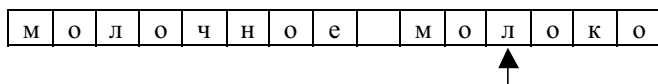
номером $j(o)$ », поскольку контекстных моделей порядка o может быть огромное количество. Но при сжатии каждого текущего символа мы рассматриваем только *одну* КМ для каждого порядка, т.к. контекст определяется *непосредственно* примыкающей слева к символу строкой определенной длины. Иначе говоря, для 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	Частоты	3	5	2	2	2	2	2	1
порядка 0 (контекст “”)	Накоп- ленные частоты	3	8	10	12	14	16	18	19
КМ	Частоты	-	-	1	1	-	1	-	-
порядка 1 (контекст “о”)	Накоп- ленные частоты	-	-	1	2	-	3	-	-
КМ	Частоты	-	-	1	-	-	-	-	-
порядка 2 (“мо”)	Накоп- ленные 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 (частоты 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 Уиттенем (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 RPM 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 «RPM» часто используется для обозначения контекстных методов вообще.

Ниже будет описан некий обобщенный алгоритм RPM, 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, пикселей, 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.

RPM 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 RPM используется неявное взвешивание оценок. Попытка оценки символа начинается с КМ(N), где N является параметром алгоритма и называется порядком RPM-модели. В случае нулевой частоты символа в КМ текущего порядка осуществляется переход к КМ меньшего порядка за счет использования меха-

низма 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.

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 $KM(N)$. 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 $KM(N-1)$ производится очередная попытка оценить ве-

роятность 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, имеющих 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 RPM

Рассмотрим подробнее работу алгоритма RPM с помощью примера.

Пример 2

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

а	б	в	а	в	а	б	в	в	б	б	б	в	?
---	---	---	---	---	---	---	---	---	---	---	---	---	---



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 "абвавабввбббв" 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.

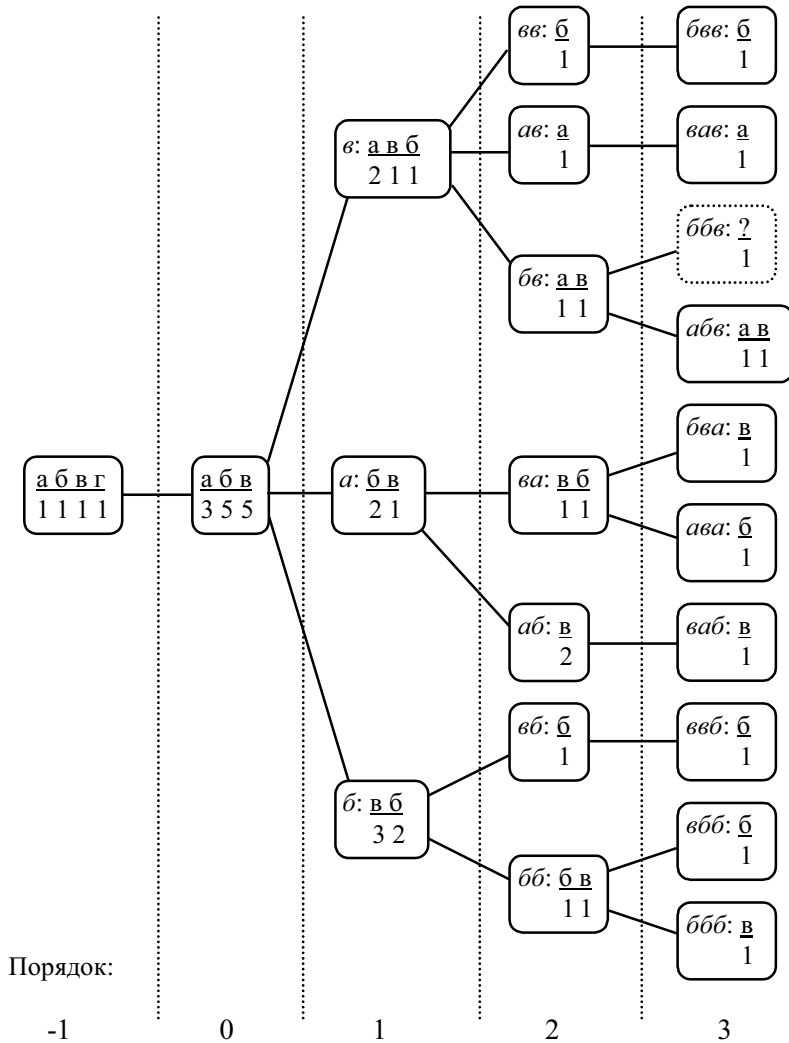


Рис. 4.2. Состояние модели после обработки
 последовательности "абвавабввбббв"

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

Символ s	Последовательность оценок для КМ каждого порядка от 3 до -1					Общая оцен- ка вероятно- сти q(s)	Представ- ление тре- бует 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 RPM-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 0
≤ N
*/
context = "";
while ( ! DataFile.EOF() ){
    с = 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 с в этой КМ, в
        CumFreq получим его накопленную частоту (или
        накопленную частоту символа ухода), в counter –
```

```
    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 RPPM-КОМПРЕССОРА

Рассмотрим основные моменты реализации компрессора RPPM 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 sp 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;
    }
}
```

```
    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, запоним 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 KM  
    init_model ();  
    AC.StartEncode (); // проинициализируем арифм. кодер  
    while (( c = DataFile.ReadSymbol() ) != EOF) {  
        // попробуем закодировать в KM(1)  
        success = encode_sym (&cm[context[0]], c);  
        if (!success)  
            /*уходим на KM(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 KM(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. $CumFreqUnder \leq x < CumFreqUnder+CM->count[i]$. 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 KM; 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 А и метод В. Использующие их алгоритмы PPM были названы PPMA и PPMВ 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	$\begin{cases} \frac{S^{(1)}}{C}, & \text{при } 0 < S^{(1)} < C \\ E^{(C)}, & \text{в противном случае} \end{cases}$

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 .

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 А и В от других методов.

Упражнение: Выполните действия, описанные в примере 2, используя ОВУ по методу С. 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 .

Ожидание разрешения издательства

Подпункт
«Адаптивные методы»

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 $0, 1, \dots, N$, что и предлагается, в частности, делать в алгоритмах РРМА и РРМВ. Такой подход называется полным обновлением (full updates). Но в случае классического, не использующего наследование информации РРМ, лучшие результаты достигаются когда счетчики оцененного символа увеличиваются только в КМ порядков $o, o+1, \dots, N$, 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 РРМ-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 РРМ* (описание РРМ* приведено ниже), польза от исключения при обновлении не столь очевидна.

Роль контекстов-предков сравнительно небольших порядков значительно возрастает при использовании техники наследова-

ния 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 $KM(i)$ понимаются предки той $KM(o)$, в которой символ был оценен. Например, в компрессоре PPMd делается модификация счетчика с весом $1/2$ только в родительской KM и только в определенных случаях. При этом основное условие выполнения такой модификации требует, чтобы счетчик оцененного символа в $KM(o)$ 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 KM , а если же их нет, то только в недетерминированной KM с самым большим порядком. Под детерминированной понимается такая KM , 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 KM 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 виде улучшения сжатия при кодировании данных с достаточно быстро меняющейся статистикой контекстных моделей. Чем нестабильнее KM , 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 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 дости-

гается при $N = 4 \dots 5$, после чего наблюдается плавное падение степени сжатия.

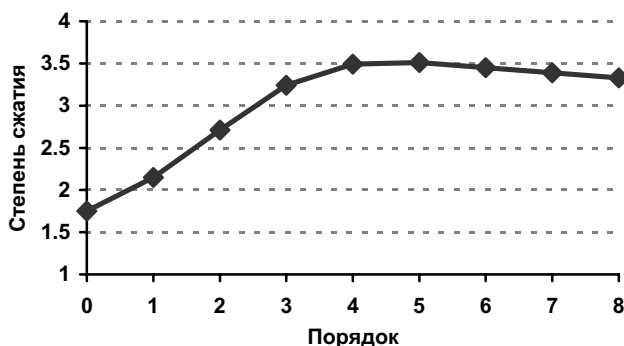


Рис. 4.3. Зависимость степени сжатия от порядка модели для классического RPM-алгоритма

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:

- 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 RPM.

Выбор той или иной функции доверия зависит от особенностей конкретной реализации RPM, 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:

$$a \cdot Q \log_2 Q + b \cdot E (\log_2 E - c) + d \cdot (1 - Q) (\log_2 E - c), \quad (4.1)$$

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 RPPM-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 $0.5 \cdot (2-1) + 0.5 \cdot (0.4-1) = 0.2$ 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 RPM.

НАСЛЕДОВАНИЕ ИНФОРМАЦИИ

Метод наследования информации позволяет существенно улучшить точность оценок. На конец 2001 года имеется по крайней мере одна очень эффективная реализация RPM, 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 больших 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 s_i был успешно оценен. Тогда начальное значение счетчика символа в КМ(o) разумно вычислять исходя из равенства

$$\frac{f^0(s_i | o)}{f(o)} = \frac{f(s_i | o - k)}{f(o - k) + F_{o-k,o}}, \quad (4.2)$$

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 $o-k+1 \dots o$.

$$F_{o-k,o} = \sum_{m=o-k+1}^o \left(f(m) - f(esc | m) - \sum_{j=1}^{S(m)} f^0(s_j | m) \right),$$

где $f(esc|m)$ — частота для символа ухода в $KM(m)$;
 $S(m)$ — количество символов в $KM(m)$, 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 $KM(o-k)$, 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 $o-1 \dots o-k+1$ являются «молодыми», и для них еще не накоплено полезной статистики. Поэтому на практике целесообразно использовать приближенную формулу

$$f^0(i | o) = \frac{f(o) \cdot f(s_i | o - k)}{f(o) - s(o) + f(o - k) - f(s_i | o - k)}.$$

При самой простой реализации величина $f^0(s_i | o)$ вычисляется и присваивается сразу при создании нового счетчика в $KM(o)$. 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 большим объемом статистики, что должно дать более точную оценку $f^0(s_i | o)$ 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 по-

Ожидание разрешения издательства

Подпункты

**«Масштабирование счетчика послед-
него встреченного символа»,
«Масштабирование в
детерминированных контекстах»**

**пункта «Различные способы повы-
шения точности предсказания»**

министическое масштабирование». Нетрудно заметить, что этот механизм есть частный случай rescency 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 $f'(s_i | o - 1)$:

$$f_{\text{корр}}(s_i | o) = \frac{f(o) \cdot f(s_i | o) + f(o-1) \cdot f'(s_i | o-1)}{f(o) + f(o-1)},$$

где $f'(s_i | o-1) = f(s_i | o-1) \cdot \frac{f(o) - f(s_i | o)}{f(o-1) - f(s_i | o-1)}.$

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 Н 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;

Ожидание разрешения издательства

**Подпункт
«Общий случай применения
вторичной оценки символа»
и пункт «PPM и PPM*»**

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 PPMС 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 PPMС 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, RPM-подобный алгоритм будет наилучшим выбором в качестве основы специализированного компрессора.

Если выйти за рамки частной проблемы сжатия данных, то несомненным достоинством RPM 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 RPM 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 RPM 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 контекстов); с точки зрения такой адаптации обычные алгорит-

мы 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.

Симметричность: близка к 1; обычно декодер не-много медленнее кодера.

Характерные особенности: медленная обработка мало избыточных данных.

Компрессоры и архиваторы, использующие контекстное моделирование

НА

Программа НА явилась, пожалуй, первым публично доступным архиватором, использующим контекстное моделирование. Не исключено, что НА стал бы очень популярным архиватором, если бы его автор, Гарри Хирвола (Hirvola), не прекратил работать над проектом.

В НА реализованы алгоритм семейства LZ77 и алгоритм типа PPM.

Алгоритм PPM представляет собой хорошо продуманную модификацию классического PPMС. Метод ОВУ является априорным и основывает оценку ухода из КМ на количестве имеющихся в ней символов с небольшой частотой. LOE не производится, последовательность спуска с КМ высоких порядков является обычной. Максимальный порядок КМ равен 4, минимальный — минус единице. Для организации поиска КМ применяются хеш-цепочки. Хеширование осуществляется по символам контекста и его длине.

Результаты тестирования на CalgCC, представленные в табл. 4.8, получены для версии 0.999с. 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 можно

существенно улучшить за счет увеличения объема доступной памяти.

СМ

СМ 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, СМ 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 кон-

Ожидание разрешения издательства

Подпункты
«RK и RKUC», «PPMN»,
«PPMd и PPMonstr», «PPMY»

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 $T_{\text{код}}$ и декодирования $T_{\text{дек}}$. Средняя степень вычислялась как средняя не взвешенная по размеру файлов степень сжатия. Для $T_{\text{код}}$ и $T_{\text{дек}}$ 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

	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 Стик Валентини (Valentini);

а также компрессор PPMZ, автор Чарльз Блум (Bloom).

Обзор классических алгоритмов контекстного моделирования

ЛОЕМА

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 сторон-

ние отчеты, по которым LOEMA обеспечивает компрессию примерно на уровне РРМС (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 А, и символ кодируется в КМ(0).

Ожидание разрешения издательства

Подпункты
«ADSM», «DHPC», «Алгоритмы РРМА
и РРМВ», «WORD», «Сравнение
алгоритмов контекстного
моделирования»

Алгоритм сРРМII 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* o-5	PPMD o-5	сРРМII 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, изоэтранные 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 СТW при оценке вероятности символа используется с некоторым весом статистика контекстных моделей-предков, т.е. производится явное взвешивание. Описанные алгоритмы СТW работают с бинарным алфавитом.

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 самоконтроля³

1. Объясните связь между точностью предсказания значений данных и степенью сжатия.
2. Что собой представляет модель источника данных в случае использования для моделирования РРМ-алгоритма?
3. Почему технику уходов можно охарактеризовать как способ неявного взвешивания статистики контекстных моделей?
4. В каких случаях оценка вероятности ухода может равняться нулю?

³ Ответы к вопросам и упражнениям и исходные тексты программ вы можете найти на <http://compression.graphicon.ru/>

5. Приведите пример блока данных, которые выгоднее сжимать, предсказывая вероятность символов на базе их безусловных частот, а не с помощью РРМ-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 РРМ-алгоритмами выше?
12. Укажите задачи, при решении которых может использоваться РРМ-моделирование.

Литература

1. Шкарин Д. Повышение эффективности алгоритма РРМ // 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/uaharc04.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) PPM DH — fast PPM compressor for textual data. <ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdh.rar>
9. Smirnov M. (2001) PPMN — not so fast PPM compressor.
<http://msmirn.newmail.ru/ppmnb1.zip>
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>

Глава 5. Преобразование Барроуза-Уилера

Введение

Преобразование Барроуза-Уилера применяется в алгоритмах сжатия качественных данных. Для эффективного использования преобразования необходимо, чтобы характеристики данных соответствовали модели источника с памятью.

Как и многие другие применяемые в алгоритмах сжатия преобразования, преобразование Барроуза-Уилера предназначено для того, чтобы сделать сжатие данных входного блока более эффективным. Посредством перестановки элементов данное преобразование превращает входной блок данных со сложными зависимостями в блок, структуру которого моделировать гораздо легче, причем отображение происходит без потерь информации.

Этот метод появился сравнительно недавно. Впервые он был опубликован 10 мая 1994 года в статье “A Block-sorting Lossless Data Compression Algorithm” [13]. Авторами метода являются Дэвид Уилер и Майк Барроуз. Причем, первый из них придумал этот метод еще в 1983 году, когда работал в компании AT&T Bell Laboratories. Но, видимо, либо тогда не придал значения своему открытию, либо посчитал чрезмерными требования к вычислительным ресурсам.

По имени авторов, алгоритм был назван «Преобразованием Барроуза-Уилера» (“Burrows-Wheeler Transform”, далее BWT). Метод был объявлен компромиссным между быстрыми словар-

ными алгоритмами, с одной стороны, и статистическими алгоритмами, дающими более сильное сжатие, но малоприменимыми в то время на практике, с другой стороны.

Благодаря таким свойствам описываемое преобразование стало довольно популярным и среди разработчиков архиваторов, и среди научных работников. Уже опубликовано более сотни статей на разных языках, посвященных этому методу, и написано столько же программ, его реализующих.

Преобразование Барроуза-Уилера

Как уже указывалось, преобразование Барроуза-Уилера предназначено для того, чтобы преобразовать входной блок в более удобный для сжатия вид. Причем, как показывает практика, полученный в результате преобразования блок обычные методы сжимают не так эффективно, как методы, специально для этого разработанные.

Поэтому нельзя рассматривать описываемый алгоритм отдельно от соответствующих специфических методов кодирования данных.

В оригинальной статье была предложена, как одна из возможных реализаций сжатия на основе BWT, совокупность из трех алгоритмов:

- преобразование Барроуза-Уилера,
- преобразование Move-To-Front (известное в русскоязычной литературе как перемещение стопки книг),
- статистический кодер для сжатия данных, полученных в результате последовательного применения двух вышеупомянутых преобразований.

Дальнейшие исследования показали, что второе из перечисленных преобразований не является необходимым и может быть заменено альтернативным. Или даже исключено вовсе за счет усложнения кодирующей фазы. Впрочем, еще сами авторы BWT упоминали о такой возможности.

Итак, начнем с описания главной составляющей части процесса сжатия данных при помощи методов на основе BWT — с самого преобразования.

Прежде всего, следует отметить одну из его особенностей. Он оперирует сразу целым блоком данных. То есть, ему заранее известны сразу все элементы входного потока или, по крайней мере, достаточно большого блока. Это делает затруднительным использование алгоритма в тех областях применения, где требуется сжатие данных «на лету», символ за символом. В этом отношении BWT даже более требователен, чем методы семейства LZ77, использующие для сжатия скользящее окно.

Следует отметить, что возможна реализация сжатия данных на основе BWT, обрабатывающая данные последовательно по символам, а не по блокам. Но скоростные характеристики программ, использующих такую реализацию, будут очень далеки от совершенства.

Таким образом, мы пришли к первой и самой легкой из фаз преобразования — к выделению из непрерывного потока блока данных.

Де-факто описывать BWT стало принято с помощью примера преобразования строки символов “абракадабра”.

Далее, нужно из полученного блока данных создать матрицу всех возможных его циклических перестановок. Первой строкой матрицы будет исходная последовательность, второй строкой — она же, сдвинутая на один символ влево и т.д. Таким образом, получим следующую матрицу:

абракадабра
бракадабраа
ракадабрааб
акадабраабр
кадабраабра
адабраабрак
дабраабрака
абраабракад
браабракада
раабракадаб
аабракадабр

**Рис. 5.1. Множество циклических перестановок строки
“абракадабра”**

Пометим в этой матрице исходную строку и отсортируем все строки в соответствии с лексикографическим порядком символов. Будем считать, что одна строка должна находиться в матрице выше другой в том случае, если в самой левой из позиций, начиная с которой строки отличаются, в этой строке находится символ лексикографически меньший, чем у другой строки. Другими словами, следует отсортировать символы сначала по первому символу, затем строки, у которых первые символы равны, — по второму и т.д.

0	аабракадабр	
1	абраабракад	
2	абракадабра	— исходная строка
3	адабраабрак	
4	акадабраабр	
5	браабракада	
6	бракадабраа	
7	дабраабрака	
8	кадабраабра	
9	раабракадаб	
10	ракадабрааб	

**Рис. 5.2. Матрица циклических перестановок строки
“абракадабра”, отсортированная слева направо в
соответствии с лексикографическим порядком символов ее
строк**

Теперь остался последний шаг — выписать символы последнего столбца и запомнить номер исходной строки среди отсортированных. Итак, “рдакраааабб”,² — это результат, полученный в результате преобразования Барроуза-Уилера.

Теперь нам осталось сделать «всего» три вещи:

- 1) доказать, что преобразование обратимо,
- 2) показать, что оно не требует огромного количества ресурсов,
- 3) что оно полезно для последующего сжатия.

Упражнение: Прodelайте преобразование Барроуза-Уилера строки “карабас”.

ДОКАЗАТЕЛЬСТВО ОБРАТИМОСТИ ПРЕОБРАЗОВАНИЯ БАРРОУЗА-УИЛЕРА

Возможно, Дэвид Уилер не опубликовал описание алгоритма в 1983 году потому, что ему самому показалось странным, что можно восстановить начальную строку из столь сильно перемешанных между собой символов. Но, так или иначе, это не фокус и обратное преобразование действительно возможно.

Пусть нам известны только результат преобразования, т.е. последний столбец матрицы, и номер исходной строки. Рассмотрим процесс восстановления исходной матрицы. Для этого отсортируем все символы последнего столбца.

0	a
1	a
2	a

3	а
4	а
5	б
6	б
7	д
8	к
9	р
10	р

Рис. 5.3. Отсортированные символы исходной строки

Нам известно, что строки матрицы были отсортированы по порядку, начиная с первого символа. Поэтому, очевидно, в результате такой сортировки мы получили первый столбец исходной матрицы.

Поскольку последний столбец по условию задачи нам известен, добавим его в полученную матрицу.

0	а.....р
1	а.....д
2	а.....а
3	а.....к
4	а.....р
5	б.....а
6	б.....а
7	д.....а
8	к.....а
9	р.....б
10	р.....б

Рис. 5.4. Первый и последний столбцы матрицы циклических перестановок

Теперь самое время вспомнить, что строки матрицы были получены в результате циклического сдвига исходной строки. То есть, символы последнего и первого столбцов образуют друг с другом пары. И нам ничто не может помешать отсортировать

эти пары, поскольку обязательно существуют такие строки в матрице, которые начинаются с этих пар. Заодно допишем в матрицу и известный нам последний столбец.

0	аа.....р
1	аб.....д
2	аб.....а
3	ад.....к
4	ак.....р
5	бр.....а
6	бр.....а
7	да.....а
8	ка.....а
9	ра.....б
10	ра.....б

Рис. 5.5. Первый, второй и последний столбцы матрицы

Таким образом, два столбца нам уже известны. Легко заметить, что отсортированные пары вместе с символами последнего столбца составляют тройки. Аналогично восстанавливается вся матрица. А на основании записанного заранее номера исходной строки в матрице — и сама исходная строка.

0	ааб.....р	аабр.....р		аабракада.р	аабракадабр
1	абр.....д	абра.....д		абраабрак.д	абраабракад
2	абр.....а	абра.....а		абракадаб.а	абракадабра
3	ада.....к	адаб.....к		адабраабр.к	адабраабрак
4	ака.....р	акад.....р		акадабраа.р	акадабраабр
5	бра.....а	браа.....а	...	браабрака.а	браабракада
6	бра.....а	брак.....а		бракадабр.а	бракадабраа
7	даб.....а	дабр.....а		дабраабра.а	дабраабрака
8	кад.....а	када.....а		кадабрааб.а	кадабраабра
9	раа.....б	рааб.....б		раабракад.б	раабракадаб
10	рак.....б	рака.....б		ракадабра.б	ракадабрааб

Рис. 5.6. Процесс определения всех столбцов матрицы

Упражнение: В результате преобразования Барроуза-Уилера получена строка “тпрооппо”. Номер исходной строки в матрице преобразований — 5 (считая с нуля). Восстановите исходную строку.

ВЕКТОР ОБРАТНОГО ПРЕОБРАЗОВАНИЯ

После того, как мы доказали принципиальную возможность обратного преобразования, можно показать, что для его осуществления нет необходимости посимвольно выписывать все строки матрицы перестановок. Если бы мы хранили в памяти всю матрицу, требуемую для мегабайтного блока данных, нам потребовалось бы... В общем, видно, что затея была бы бесполезной.

Для обратного преобразования нам дополнительно к собственным данным нужен только вектор обратного преобразования, представляющий собой массив чисел, размер которого равен числу символов в блоке. Поэтому затраты памяти при выполнении обратного преобразования линейно зависят от размера блока.

Обратите внимание, что в процессе выявления очередного столбца матрицы мы совершали одни и те же действия. А именно, получали новую строку, сливая символ из последнего столбца старой строки с известными символами первых столбцов этой же строки. И новая строка после сортировки перемещалась в другую позицию в матрице.

Так из строки 0 мы получали строку 9, из первой — седьмую и т.п.:

номер строки		номер новой строки
0	a.....p	9
1	a.....д	7
2	a.....а	0
		— исходная строка

3	а.....к	8
4	а.....р	10
5	б.....а	1
6	б.....а	2
7	д.....а	3
8	к.....а	4
9	р.....б	5
10	р.....б	6

Рис. 5.7. Способ получения первого столбца матрицы из последнего

Важно, что при добавлении любого столбца перемещения строк на новую позицию были одинаковы. Нулевая строка перемещалась в девятую позицию, первая — в седьмую, и т.д.

Чтобы получить вектор обратного преобразования, следует определить порядок получения символов первого столбца из символов последнего. То есть, отсортировать матрицу по номерам новых строк.

номер строки		номер новой строки	переносим последний столбец в начало
2	а.....а	0	аа.....р
5	б.....а	1	аб.....д
6	б.....а	2	аб.....а
7	д.....а	3	ад.....к
8	к.....а	4	ак.....р
9	р.....б	5	бр.....а
10	р.....б	6	бр.....а
1	а.....д	7	да.....а
3	а.....к	8	ка.....а
0	а.....р	9	ра.....б
4	а.....р	10	ра.....б

Рис. 5.8. Получение вектора обратного преобразования

Полученные значения номеров строк $T = \{ 2, 5, 6, 7, 8, 9, 10, 1, 3, 0, 4 \}$ и есть искомый вектор, содержащий номера позиций символов в строке, которую нам надо декодировать (символы упорядочены в соответствии с положением в алфавите).

Теперь получить исходную строку совсем просто. Первым делом возьмем элемент вектора обратного преобразования, соответствующий номеру исходной строки в матрице циклических перестановок, $T[2] = 6$. Иначе говоря, в качестве первого символа в исходной строке следует взять шестой символ из строки “рдakraaaaбб”. Это символ ‘а’.

Затем нужно определить, какой символ заставил опуститься найденный символ ‘а’ на вторую позицию среди равных. Искомый символ находится в последнем столбце шестой строки матрицы, изображенной на рис. 5.8. А поскольку $T[6] = 10$, в преобразованной строке он находится в десятой позиции. Это символ ‘б’. И т.д.

6	10	4	8	3	7	1	5	9	0	2
а	б	р	а	к	а	д	а	б	р	а

Рис. 5.9. Декодирование исходной строки при помощи вектора обратного преобразования

Упражнение: В результате преобразования Барроуза-Уилера получена строка “тпрооппо”. Номер исходной строки в матрице преобразований — 5 (считая с нуля). Постройте вектор обратного преобразования.

РЕАЛИЗАЦИЯ ОБРАТНОГО ПРЕОБРАЗОВАНИЯ

Получение исходной строки из преобразованной можно проиллюстрировать при помощи небольшой программы.

Введем следующие обозначения:

n — количество символов в блоке входного потока;

N — количество символов в алфавите;
 pos — номер исходной строки в матрице перестановок;
 in — входной блок;
 $count$ — массив частот каждого символа алфавита во входном блоке;
 T — вектор обратного преобразования, размер вектора равен n .

Первым делом следует посчитать частоты символов и пронумеровать все исходные символы в порядке их появления в алфавите. По сути, это эквивалентно построению первого столбца матрицы циклических перестановок.

```
for( i=0; i<N; i++) count[i]=0;
for( i=0; i<n; i++) count[in[i]]++;
sum = 0;
for( i=0; i<n; i++) {
    sum += count[i];
    count[i] = sum - count[i];
}
```

После выполнения этих действий $count[i]$ указывает на первую позицию символа с кодом i в первом столбце матрицы. Следующий шаг — создание вектора обратного преобразования.

```
for( i=0; i<n; i++) T[count[in[i]]++] = i;
```

Далее, при помощи полученного вектора восстановим исходный текст.

```
for( i=0; i<n; i++) {
    putc( in[pos], output );
    pos = T[pos];
}
```

ИСПОЛЬЗОВАНИЕ BWT В СЖАТИИ ДАННЫХ

Теперь, после того как выяснилось, что наши действия вполне обратимы, и данные мы не исказим, можно вернуться к вопросу рассмотрения полезности преобразования. Как уже отмечалось выше, главная задача преобразования Барроуза-Уилера

заключается в том, чтобы ловко переставить символы. Переставить так, чтобы их можно было легко сжать, не ломая голову над их взаимосвязями. Потому что преобразование как раз тем и занимается: «вытаскивает» все взаимосвязи наружу. Точнее, очень многие.

Для понимания этого процесса достаточно представить поток данных состоящим из набора некоторых стабильных сочетаний символов. Например, как этот текст, состоящим из слов. Сочетание символов, позволяющее предсказать некоторый неизвестный доселе символ, называется контекстом. Например, если нам известна последовательность символов “реобразование”, то скорее всего ей предшествует символ “п”. Назовем устойчивым (стабильным) такой контекст, для которого распределение частот символов, непосредственно примыкающих к нему слева или справа, меняется незначительно в пределах блока.

Если нам потребуется подвергнуть преобразованию данную главу, можно с уверенностью сказать, что строки, начинающиеся с символов “реобразование”, будут располагаться рядом в отсортированной матрице. И в соответствующих этим строкам позициях последнего столбца матрицы будет находиться символ “п”.

Таким образом, главное свойство преобразования в том, что оно собирает вместе символы, соответствующие похожим контекстам. Чем больше стабильных контекстов в блоке данных, тем лучше будет сжиматься полученный в результате преобразования блок. Практика показывает, что в результате преобразования обычных текстов более половины из всех символов следует за такими же.

Упражнение: Какие еще символы помимо “п” могут оказаться в конце строк, начинающихся с последовательности символов “реобразование”, в результате преобразования данной главы?

ЧАСТИЧНОЕ СОРТИРУЮЩЕЕ ПРЕОБРАЗОВАНИЕ

Некоторое время спустя после появления первых архиваторов, использующих преобразование Барроуза-Уилера, было опубликовано описание еще одного алгоритма, также основанного на сортировке блока данных [33, 1]. Отличие от BWT заключается в том, что сортировка строк матрицы осуществляется не по всей длине строк, а только по некоторому фиксированному количеству символов. В том случае, если у нескольких строк эти символы одинаковы, выше в списке помещается строка, первый символ которой встретился во входном блоке раньше первых символов остальных рассматриваемых строк.

Можно сказать, что позиция первого символа строки во входном блоке участвует в сортировке. Число символов строки, участвующих в преобразовании, называется порядком сортирующего преобразования. Легко заметить, что в результате преобразования нулевого порядка, $ST(0)$, получается исходная строка.

В качестве примера выполним преобразования первого и второго порядка строки “абракадабра”.

Номер строки	ST(1)	ST(2)	BWT
Первый шаг. Построение матрицы преобразования:			
0	a< 0>бракадабра	аб< 0>ракадабра	абракадабра
1	б< 1>ракадабраа	бр< 1>акадабраа	бракадабраа
2	р< 2>акадабрааб	ра< 2>кадабрааб	ракадабрааб
3	а< 3>кадабраабр	ак< 3>адабраабр	акадабраабр
4	к< 4>адабраабра	ка< 4>дабраабра	кадабраабра
5	а< 5>дабраабрак	ад< 5>абраабрак	адабраабрак
6	д< 6>абраабрака	да< 6>браабрака	дабраабрака
7	а< 7>браабракад	аб< 7>раабракад	абраабракад
8	б< 8>раабракада	бр< 8>аабракада	браабракада
9	р< 9>аабракадаб	ра< 9>абракадаб	раабракадаб
10	а<10>абракадабр	аа<10>бракадабр	аабракадабр

Второй шаг. Сортировка:

0	a< 0>бракадабра	aa<10>бракадабр	аабракадабр
1	a< 3>кадабраабр	аб< 0>ракадабра	абраабракад
2	a< 5>дабраабрак	аб< 7>раабракад	абракадабра
3	a< 7>браабракад	ад< 5>абраабрак	адабраабрак
4	a<10>абракадабр	ак< 3>адабраабр	акадабраабр
5	б< 1>ракадабраа	бр< 1>акадабраа	браабракада
6	б< 8>раабракада	бр< 8>аабракада	бракадабраа
7	д< 6>абраабрака	да< 6>браабрака	дабраабрака
8	к< 4>адабраабра	ка< 4>дабраабра	кадабраабра
9	р< 2>акадабрааб	ра< 2>кадабрааб	раабракадаб
10	р< 9>аабракадаб	ра< 9>абракадаб	ракадабрааб

Результат:

аркдраааабб,5 радкраааабб,5 рдакраааабб,2

Рис. 5.10. Частичные сортирующие преобразования первого и второго порядков

Так же, как и в BWT, результатом преобразования являются последний столбец матрицы и номер строки, последний символ которой является начальным символом исходной строки.

Упражнение: Выполните преобразование ST(3) строки “абракадабра”.

Легко заметить, что различие между частичным сортирующим преобразованием и преобразованием Барроуза-Уилера можно увидеть только при сортировке устойчивых контекстов длиннее порядка преобразования ST. В методе BWT в этом случае продолжается процесс сравнения символов, а в ST — сравнение символов прекращается и выше располагается та строка, первый символ которой встретился во входном блоке раньше. Следовательно, те данные, в которых встречаются длинные повторы, более эффективно сжимаются преобразованием Барро-

уза-Уилера. К примеру, типичные текстовые файлы на английском языке теряют в сжатии около 5% при выполнении сортировки по 4 символам.

С другой стороны, частичное сортирующее преобразование не требует полной сортировки всех строк матрицы и свободно от тех проблем, которые возникают при преобразовании очень избыточных данных с использованием полной сортировки. Как правило, данное преобразование выполняется быстрее, чем BWT.

Но при выполнении обратного преобразования наблюдается иная картина. Если в случае BWT оно выполняется легко и просто, то для восстановления исходных данных после частичного сортирующего преобразования необходимо проделывать дополнительные усилия. А именно, вести учет количества одинаковых контекстов. И чем порядок преобразования больше, тем требуется больше времени на подсчет.

Методы, используемые совместно с BWT

Как уже было сказано, само по себе преобразование Барроуза-Уилера не сжимает. Эту работу проделывают другие методы, призванные толково распорядиться теми свойствами, которыми обладают преобразованные данные.

Среди таких методов можно отметить следующие:

- кодирование длин повторов (RLE),
- метод перемещения стопки книг (MTF),
- кодирование расстояний (DC),
- метод Хаффмана,
- арифметическое кодирование.

Последовательность применения методов, используемых совместно с BWT:

Шаг	Используемый алгоритм
1	Кодирование длин повторов (необязательно)
2	Преобразование Барроуза-Уилера Частичное сортирующее преобразование
3	Перемещение стопки книг Кодирование расстояний
4	Кодирование длин повторов (необязательно)
5	Метод Хаффмана Арифметическое кодирование

ПЕРЕМЕЩЕНИЕ СТОПКИ КНИГ

Метод также известен под названием MTF (Move To Front). Суть его легко понять, если представить процесс перемещения книг в стопке, из которой время от времени достают нужную книгу и кладут сверху. Таким образом, через некоторое время наиболее часто используемые книги оказываются ближе к вершухе стопки.

Введем следующие обозначения:

N — число символов в алфавите;

M — упорядоченный список символов размером N ; $M[0]$ соответствует верхней книге стопки, $M[N-1]$ — нижней;

x — очередной символ.

```
int tmp1, tmp2, i=0;
tmp1 = M[i];
M[i] = x;
while( tmp1 != x ) {
    i++;
    tmp2 = tmp1;
    tmp1 = M[i];
    M[i] = tmp2;
}
```

Для примера, произведем преобразование над последовательностью “рдакраааабб”, полученной нами после BWT. Предположим, что мы имеем дело с алфавитом, содержащим только эти пять символов, и в упорядоченном списке символов они расположены в следующем порядке: $M = \{ 'а', 'б', 'д', 'к', 'р' \}$.

Первый из символов последовательности, ‘р’, находится в списке под номером 4. Это число мы и записываем в выходной блок. Затем мы изменяем список, перенося этот символ в вершину списка, при этом сдвигая все остальные элементы, находившиеся до этого выше.

Следующий символ, ‘д’, после этого сдвига оказывается в списке под номером 3. И так далее.

Символ	Список	Номер
р	абдкр	4
д	рабдк	3
а	драбк	2
к	адрбк	4
р	кадрб	3
а	ркадб	2
а	аркдб	0
а	аркдб	0
а	аркдб	0
б	аркдб	4
б	баркд	0

Рис. 5.11. MTF-преобразование строки “рдакраааабб”

Таким образом, в результате преобразования по методу перемещения стопки книг мы получили последовательность “43243200040”.

Вспомним, что результат преобразования Барроуза-Уилера представляет собой последовательность символов, среди которых часто попадаются идущие подряд одинаковые символы. Поэтому, чтобы эффективно сжать такую последовательность, статистическому кодеру необходимо вовремя отслеживать смену одного частого символа другим. MTF предназначен для того, чтобы облегчить задачу статистическому кодеру.

Рассмотрим последовательность “bbbbcccccdddddaaaaab”, обладающую такими свойствами. Попробуем обойтись без MTF и закодировать ее по методу Хаффмана. Для упрощения будем

исходить из предположения, что затраты на хранение дерева, требуемого для обеспечения декодирования, будут равны и с использованием MTF, и без него.

Вероятности всех четырех символов в данном примере равны $1/4$, т.е. для кодирования каждого из символов нам потребуется 2 бита. Легко посчитать, что в результате кодирования мы получим последовательность длиной $20 \cdot 2 = 40$ бит.

Теперь сделаем то же самое со строкой, подвергнутой MTF-преобразованию. Предположим, что начальный список символов выглядит как {'a', 'b', 'c', 'd'}.

bbbbccccccdddddaaaaaab	исходная строка
10002000030000300003	строка после MTF

Символ	Частота	Вероятность	Код Хаффмана
0	15	3/4	0
3	3	3/20	10
1	1	1/20	110
2	1	1/20	111

Рис. 5.12. Кодирование методом Хаффмана строки после MTF-преобразования

В результате кодирования получаем последовательность длиной $15 \cdot 1 + 3 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 27$ бит.

Упражнение: Произведите преобразование методом стопки книг последовательности “bbbbbbccccccdddddccseaaaaa” и определите, будет ли использование MTF давать преимущество при кодировании методом Хаффмана. Начальный упорядоченный список символов установить {'a', 'b', 'c', 'd', 'e'}. Исходите из предположения, что алфавит состоит только из указанных пяти символов.

КОДИРОВАНИЕ ДЛИН ПОВТОРОВ

Этот метод также называется Run Length Encoding (RLE). Это один из наиболее старых методов сжатия. Суть этого метода заключается в замене идущих подряд одинаковых символов числом, характеризующим их количество. Конечно, также мы должны и указать признак «включения» механизма кодирования длин повторов, который можем распознать при декодировании.

Один из возможных вариантов — включать кодирование, когда число повторяющихся символов превысит некоторый порог. Например, если мы условимся, что порог равняется трем символам, то последовательность “aaaaabbccccdd” в результате кодирования будет выглядеть как “aaa2bbb0cccc1dd”. Если мы выберем в качестве порога 4 символа, то получим “aaaa1bbbcccc0dd”.

Упражнение: Что получится, если закодировать повторы в данной строке, используя порог, равный двум символам?

Главное назначение кодирования длин повторов в связке с BWT -увеличить скорость сжатия и разжатия.

RLE можно применить дважды — до преобразования и после. До преобразования данный метод может пригодиться, если мы имеем дело с потоком, содержащим много повторов одинаковых символов. Сортировка строк матрицы перестановок — наиболее длительная из процедур, необходимых для сжатия при помощи BWT. В случае высокоизбыточных данных время выполнения этой процедуры может существенно (в разы) возрасти. Сейчас разработаны методы сортировки, устойчивые к такого рода избыточности данных, но ранее метод кодирования длин повторов широко использовался на этом этапе ценой небольшого ухудшения сжатия. RLE следует применять, если указанных повторов уж слишком много.

Не является обязательным и другое применение RLE — кодирование длин повторов после преобразования Барроуза-Уилера. Оно довольно эффективно реализовано в `gzip` [33] и `BA`, но известны архиваторы, в которых RLE не требуется, например, которые используют кодирование расстояний (`DC`, `YBS`). Ряд архиваторов использует некую разновидность кодирования длин повторов — 1-2 кодирование, описанное ниже. В любом случае, если не воспользоваться каким-нибудь из перечисленных методов сокращения количества выходных символов, скорость работы будет оставлять желать лучшего, особенно в случае архиваторов, в которых используется арифметическое кодирование.

УСОВЕРШЕНСТВОВАНИЕ МЕТОДА ПЕРЕМЕЩЕНИЯ СТОПКИ КНИГ

MTF является вполне самостоятельным преобразованием и, помимо использования вкупе с `BWT`, применяется и в других областях. Но сейчас мы рассмотрим модификации метода перемещения стопки книг, которые помогают улучшить сжатие данных, полученных именно после преобразования Барроуза-Уилера.

Когда мы выписываем символы последнего столбца матрицы перестановок, относящиеся к весьма близким контекстам, мы можем с достаточно большой долей уверенности утверждать, что для многих типов данных эти символы будут одинаковы. В частности, к таким типам данных относятся файлы, содержащие текст на естественном языке.

Однако возможны небольшие нарушения такой закономерности — за счет ошибок, за счет наличия больших букв в начале предложения, переносов слов и т. д. Эти нарушения на выходе `BWT` часто выглядят как небольшие вкрапления посторонних символов среди длинной цепочки одинаковых. Очевидно, вкрапления из одного редкого символа будут встречаться чаще двойных, тройных и более длинных.

Можно заметить, что при MTF-преобразовании такие одиночные символы приводят к двойному появлению единичных

Ожидание разрешения издательства

Подпункты
«Усовершенствование метода
перемещения стопки книг»,
«1-2 кодирование»,
«Реализация 1-2 кодирования»,
«Кодирование расстояний»

Упражнение: Вычислите расстояния для строки “бrrаааакаакдрр”.

ОБРАТНЫЕ ЧАСТОТЫ

Есть еще один метод, похожий на описанный выше. В нем также кодируются расстояния между одинаковыми символами. Отличие только в том, что символы, для которых определяются расстояния, берутся не в порядке поступления, а исходя из некоторого фиксированного порядка. Например, по алфавиту.

Авторы данного алгоритма, названного Inversed Frequencies (IF), исходили из того, что расстояние между одинаковыми символами характеризует частоту использования этих символов на данном отрезке символьной последовательности. Чем расстояние меньше, тем выше частота. Поясним работу алгоритма на примере.

Предположим, нам нужно определить IF для строки “рдакраааабб”, а расчет расстояний будем проводить в соответствии с положением символов в алфавите “абджк”.

Сначала запишем для символа 'а' положение первого из таких символов в исходной строке и их количество.

символ	первая позиция	число символов
а	2	5

Следующим шагом – для каждого из символов ‘а’ в качестве расстояния запишем число иных символов до следующего ‘а’.

исходная строка:	рдакраааабб
расстояния:	2 000

После того, как все позиции символа ‘а’ определены, мы можем удалить их из исходной строки и продолжить обработку строки, как будто их и не было.

Сим- вол	первая позиция	Число символов	строка	Расстоя- ния
а	2	5	рдakraaaaбб	2, 0, 0, 0
б	4	2	рдкрбб	0
д	1	1	рдкр	
к	1	1	ркр	

Рис. 5.18. Вычисление обратных частот

В кодировании символа ‘р’ нет необходимости, так как заведомо известно, что он занимает оставшиеся позиции в строке. Таким образом, мы получили следующую последовательность: { 2,5,2,0,0,0 }, { 4,2,0 }, { 1,1 }, { 1,1 } }.

Легко заметить, что способ и эффективность кодирования зависит от того порядка, в котором мы будем обрабатывать символы. Еще одно важное отличие данного метода от кодирования расстояний заключается в том, что он не освобождает от необходимости кодирования длинных последовательностей нулевых значений обратных частот.

Упражнение: Определите обратные частоты для строки “бrrаааакаакдрp”.

Способы сжатия преобразованных с помощью BWT данных

Перейдем к рассмотрению способов сжатия данных, полученных в результате описанных выше преобразований. Для начала надо разобраться с данными, которые нам необходимо сжать.

Рассмотрим фрагмент данных, типичный для текста, преобразованного посредством BWT. Для примера взят файл book1 из набора “Calgary Corpus”.

```
eteksehendeynkrttdserttnregenskngsgsedeneyswmessrne  
xgynystslgyegsgstssrhmsstetehselxtptneessthnndesddy  
htksthtwtpfdtтеgedmmhysyresprssneelselgetdemsetse,t  
reehsetrttseeeeeesssdeedmnlendeedgtdgtdtdsgtteeysy  
tddentnrxsltshtghnteeernsdpwlттensedehsteeswekheeee  
teneeeeeseslteenestrngsthsgdeyeyrteetklrtdtetteyodth  
eegeercesyttesedenrtresnyssgttsslsawssygsssrewmsht  
gt,etssgehehehssssehneesesdnrnekhtrsslthdsseestste  
nbgeeessesdesyndtrdhpeesehesetsrerhyesdnwtlrrhoses  
hsetdrptттtsdhaynenetyntpgstesknhysftsgssdftgteeeedu
```

Рис. 5.19. Однородный фрагмент

Можно заметить, что распределение символов на этом фрагменте меняется незначительно. Совсем другую картину мы можем наблюдать на фрагменте, приведенном ниже, когда преобладание одних символов сменяется преобладанием других.

```
ygeldsd,,ttyogdodgdedndygnmotedgwkgodooowtdoddtotet  
tndmeggkrdsqtohtdegтеddrсттттеgtddewdddootdgdntet  
ststттstt!uetттdtte-eIttetny,hettItgrlIttyItnttтыrt  
rtттттттттттттттттstотттттттттттntттттттtn,ddtkgnde;  
ed,d,stefoefssfrsnsstyslw,fnoeadere,rteeynsfofhynn  
nyoytted,yfnedhddtoldtnyhnnhyrтыryttmeryesfoyedney  
oymd,sedesgnrrsysnssmydsspdyt'-dssehs,gynsydygee'o  
defddeynt,,tdnd,os;sttyysofy-nnetognfetdnyldlewhе-  
odsttemshsdtsyteny,ngdefs,-offsnntettseyesgleay:es  
dtsdgllredksyryes,rldosts;dtdeefgghsfrergkdngkenw
```

Рис. 5.20. Изменяющийся фрагмент

Также имеют место быть случаи, когда на протяжении всего фрагмента явно превалирует один определенный символ.

```
edeeeeeeIeydeyeeet'eeeIeeeeeeeeeeeeeeeeeeeeeeleeeee  
eeeeeeee!eeeeleeeeeeeeeeeeeeeeeeeeeeyedeeereeeleee  
eeeeeeeeeeeslyadeeeeeeeeeeyeeeeeeeeeeeeedyereseeeee
```

eIueeeeeeyeeeeeeIsseseeIl'eeIetenhyalehcesyyseessn
s'dlesffao,dfefeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeTeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeteeeeeeeeeeeeeeee
eeeeeeeeeeaaerre, see, eywesldsystdheedeetaeeesaeyedo

Рис. 5.21. Фрагмент с преобладанием одного символа

Итак, выделим три вида данных:

- 1) на протяжении всего фрагмента несколько символов имеют постоянную частоту.
- 2) промежуток в фрагменте, когда преобладание одних символов сменяется преобладанием других.
- 3) один из символов встречается намного чаще других.

Одна из задач выбираемой модели заключается в том, чтобы позволить оперативно настроиться на текущий вид данных.

СЖАТИЕ ПРИ ПОМОЩИ КОДИРОВАНИЯ ПО АЛГОРИТМУ ХАФФМАНА

Указанные свойства преобразованных данных использует алгоритм, реализованный в архиваторе bzip2 и позднее позаимствованный также разработчиками архиватора IMP.

После преобразований BWT и MTF блок данных делится на равные 50-символьные куски. Полученные куски объединяются в группы по степени близости распределений MTF-рангов. Количество групп зависит от размера файла. Например, для мегабайтного файла таких групп будет шесть.

Группирование кусков выполняется итеративно. Изначально куски приписываются группам в порядке следования в блоке так, чтобы каждой группе соответствовало примерно равное количество кусков. Для каждой группы строится отдельное дерево Хаффмана. В архив записываются деревья Хаффмана, номер группы для каждого из 50-символьного кусков, а затем все куски сжимаются по алгоритму Хаффмана в соответствии с номером той группы, к которой они относятся.

Выбор группы для куска делается на основе подсчета длины закодированного куска, который получится при использовании каждого из построенных деревьев Хаффмана. Выбирается та группа, при выборе которой код получается короче. Поскольку

после этого статистика использования символов в группах меняется, по завершении обработки блока дерева для каждой группы строится заново. Практика показывает, что вполне достаточно четырех итераций для получения приемлемого сжатия. С каждой новой итерации прирост эффективности резко уменьшается.

Такой способ кодирования называется полуадаптивным алгоритмом Хаффмана. Полуадаптивность заключается в том, что адаптация происходит за счет выбора подходящего дерева Хаффмана для очередного куска данных, а не за счет перестройки текущего дерева.

Сжатие по алгоритму Хаффмана довольно эффективно, хоть и уступает алгоритмам, в которых реализовано арифметическое сжатие, но зато заметно быстрее при декодировании.

Алгоритмы, использующие кодирование по методу Хаффмана, довольно эффективны, хоть и уступают алгоритмам, в которых реализовано арифметическое сжатие

СТРУКТУРНАЯ И ИЕРАРХИЧЕСКАЯ МОДЕЛИ

Отметим основное свойство таких преобразований, как MTF, DC и IF: на большинстве данных, полученных в результате преобразования Барроуза-Уилера, малые значения встречаются гораздо чаще больших и легче поддаются предсказанию. Это свойство очень важно учитывать при построении адаптивных моделей, использующих арифметическое кодирование.

Анализируя перечисленные выше три вида фрагментов, можно отметить некоторые особенности, которыми можно воспользоваться при моделировании:

- 1) большое количество идущих подряд одинаковых символов свидетельствует о том, что вероятно появление такой же длинной их последовательности и в будущем.
- 2) появление символа, довольно давно встречавшегося последний раз, говорит о том, что вероятна смена устойчивого контекста, а следовательно, и наиболее частые до этого символы могут смениться другими.

Ожидание разрешения издательства

Подпункты
«Структурная и иерархическая
модели»,
«Размер блока»,
«Переупорядочивание символов»,
«Направление сортировки»

Файл	Направление сортировки	Размер сжатого файла, байт	
		bzip2 1.01	YBS
			0.03e
book1	following contexts	232,598	213,722
book1	preceding contexts	234,538	214,890
wcc386.exe	following contexts	308,624	276,026
wcc386.exe	preceding contexts	306,020	279,198

Некоторые программы самостоятельно пытаются определить тип данных и выбрать направление сортировки. Иногда, впрочем, ошибаясь. Простейший способ обмануть такой архиватор — дать ему сжать русский текст.

Сортировка, используемая в BWT

Сортировка — это очень важный компонент архиватора, реализующего сжатие на основе преобразования Барроуза-Уилера. Именно от нее зависит скорость сжатия. До недавнего времени именно сортировка была узким местом. В настоящее время моделирование стало достаточно сложным, чтобы конкурировать по времени работы с процедурой сортировки, реализации которой, напротив, совершенствуются в сторону ускорения. Но и теперь возможна ситуация, когда характеристики сжимаемых данных таковы, что могут существенно замедлить сортировку.

Основные требования к сортировке заключаются в том, что она должна обеспечивать быстрое сжатие обычных (преимущественно текстовых) данных и не приводить к существенному замедлению на очень избыточных данных.

Помешать сортировке могут два вида избыточности — когда в сортируемых данных содержатся:

- 1) длинные одинаковые строки,
- 2) короткие одинаковые строки в большом количестве.

Отдельный случай представляют собой большое число идущих подряд одинаковых символов или длинные последовательности перемежающихся символов типа “абабаб”.

Что касается текстовых файлов, наиболее часто встречаемая длина повторяющихся строк — 3-5 символа. Для файлов с исходными текстами программ, как правило, эта длина несколько больше — 8-12 символов.

Рассмотрим алгоритмы сортировки, получившие наибольшую известность.

СОРТИРОВКА БЕНТЛИ-СЕДЖВИКА

Данный алгоритм получил, пожалуй, наибольшее распространение среди всех известных сортировок. Впервые он был применен еще одним из основоположников, Уилером. Затем эта сортировка была реализована в bzip2 и других архиваторах (BWC, BA, YBS).

Сортировка Бентли-Седжвика (Bentley-Sedgewick) представляет собой модификацию быстрой сортировки (quick-sort), ориентированную на сравнение длинных строк, среди которых может оказаться значительное количество похожих.

Главная идея описываемой сортировки заключается в том, что все сравниваемые с эталонной строки делятся не на две, а на три группы. В третью группу входит сама эталонная строка и строки, сравниваемые символы которых равны соответствующим символам эталонной строки.

Выделение третьей группы помогает нам уменьшить число операций сравнения строк, имеющих много совпадающих подстрок. В эту группу попадают строки с одинаковыми начальными символами, что избавляет нас от необходимости сравнивать эти символы еще раз.

Работу алгоритма можно описать при помощи следующего исходного текста на языке Си:

```
void sort(char **s, int n, int d) {
    char **s_less, **s_eq, **s_greater;
    int *n_less, *n_eq, *n_greater;

    // выбор значения, с которым будут
    // сравниваться d-ые символы строк
    char v = choose_value(&s, d);
```

```
// осталась только одна строка
if( n <= 1 ) return;

// деление всех строк на группы
compare(&s, v, d,
// строки, d-ый символ которых меньше v
&s_less, &n_less,
// строки, d-ый символ которых равен v
&s_eq, &n_eq,
// строки, d-ый символ которых больше v
&s_greater, &n_greater);

sort(&s_less, &n_less, d);
sort(&s_eq, &n_eq, d+1);
sort(&s_greater, &n_greater, d);
}
```

Данная рекурсивная функция сортирует последовательность из n строк s , имеющих d одинаковых начальных символов. Самый первый вызов выглядит как `sort(s,n,0)`;

Разумеется, с течением времени придумывалось все больше ухищрений, ускоряющих работу сортировки Бентли-Седжвика применительно к BWT. Перечислим основные из них:

- 1) Поразрядная сортировка. При большом количестве сортируемых строк предварительно осуществляется поразрядная сортировка по нескольким символам. По результату поразрядной сортировки строки делятся на пакеты, каждый из которых обрабатывается при помощи сортировки Бентли-Седжвика.
- 2) Использование результата предыдущих сравнений для последующих. После окончания сортировки некоторого количества строк, можно легко отсортировать строки, начинающиеся на один символ раньше отсортированных. Для этого достаточно сравнить только первые их символы, а дальше — воспользоваться результатами предыдущей сортировки.
- 3) Сравнение не одиночных символов, а одновременно нескольких. Большая разрядность современных компьютеров позволяет выполнять операции сразу над несколькими символами,

обычно представляемыми байтами. Например, если команды процессора позволяют оперировать 32-х разрядными данными, то можно осуществлять одновременное сравнение четырех байтов.

- 4) Неполная сортировка. В результате преобразования Барроуза-Уилера мы должны получить последовательность символов последнего столбца матрицы перестановок. Нам не важно, какая из двух строк будет лексикографически меньше, если им соответствуют одинаковые символы последнего столбца. Поэтому мы можем избежать лишних сравнений при сортировке строк.

СОРТИРОВКА СУФФИКСОВ

Применяя данную сортировку, мы исходим из того, что нам приходится сортировать строки, каждая из которых является частью другой, начинающейся с более ранней позиции в блоке, т.е. является ее суффиксом. Задача сортировки суффиксов неразрывно связана с построением дерева суффиксов, которое помимо сжатия данных может быть также использовано для быстрого поиска строк в блоке.

Главное свойство всех суффиксных сортировок заключается в том, что время сортировки почти не зависит от данных. Опишем одну из получивших большую известность суффиксных сортировок, которая была опубликована в 1998 году Кунихико Садакане.

Рассмотрим алгоритм на примере. Введем обозначения:

- X — массив суффиксов $X[i]$, каждый из которых представляет собой строку, начинающуюся с i -ой позиции в блоке;
I — массив индексов суффиксов; положение индексов в этом массиве должно соответствовать порядку лексикографически отсортированных суффиксов;

Ожидание разрешения издательства

Подпункты
«Сортировка суффиксов»,
«Сравнение алгоритмов сортировки»

траты памяти и времени. И, как правило, немного худшее сжатие.

Хотя сортировка суффиксов и не зависит от избыточности данных, на типичных данных методы, использующие быструю сортировку и сортировку слиянием, оказываются быстрее.

Архиваторы, использующие BWT и ST

Довольно быстро после опубликования статьи Барроуза и Уилера стали появляться первые компрессоры. Это объясняется, во-первых, тем, что этот метод оказался хорошим компромиссом между быстрыми архиваторами, использующими словарное сжатие, и медленными по тому времени статистическими компрессорами. Во-вторых, авторы сразу заявили, что разрешают некоммерческое использование своего изобретения.

С тех пор количество программ, использующих преобразование Барроуза-Уилера, непрерывно растет. Ниже приведена таблица, в которой упомянуты наиболее интересные из них.

Компрессор и версия	Дата	Автор	Адрес
BRed*	06.1997	D.J. Wheeler	ftp://ftp.cl.cam.ac.uk/users/djw3
X1 -m7 0.95	05.1997	Stig Valentini	mailto:x1develop@dk-online.dk http://www.saunalahti.fi/~x1
BWC 0.99	01.1999	Willem Monuwe	mailto:willem@stack.nl ftp://ftp.stack.nl/pub/users/willem
IMP -2 1.12	01.2000	Conor	mailto:imp@technelysium.com.au
WinImp 1.21	09.2000	McCarthy	http://www.technelysium.com.au/ http://www.winimp.com

szip 1.12	03.2000	Michael Schindler	mailto:michael@compressconsult.com http://www.compressconsult.com/
bzip2 1.01 (bzip 0.21)	06.2000	Julian Seward	mailto:jseward@acm.org http://sourceware.cygnus.com/bzip2
DC	08.2000	Edgar Binder	mailto:EdgarBinder@t-online.de ftp://ftp.elf.stuba.sk/pub/pc/pack
0.99.298b			
YBS 0.03e	09.2000	Vadim Yoockin	mailto:yoockinv@mtu-net.ru mailto:vy@thermosyn.com http://ybs.freeservers.com
BA 1.01br5	10.2000	Mikael Lundqvist	mailto:mikael@2.sbs.se http://hem.spray.se/mikael.lundqvist
Zzip 0.36c	06.2001	Damien Debin	mailto:damien.debin@via.ecp.fr http://www.zzip.f2s.com/
SBC 0.910b	11.2001	Sami Makinen	mailto:sjm@pp.inet.fi http://www.geocities.com/sbcarchiver
ERI 5.0re	12.2001	Alexander Ratushnyak	mailto:artest@inbox.ru http://geocities.com/eri32
GCA 0.9k	12.2001	Shin-ichi Tsuruta	mailto:synsyr@pop21.odn.ne.jp http://www1.odn.ne.jp/~synsyr/
7-Zip	01.2002	Igor Pavlov	Mailto: support@7-zip.org Http://www.7-zip.org
2.30b12			

Семейство программ BRed, BRed и BRed3 написано одним из родоначальников BWT, Дэвидом Уилером. Многие идеи, использованные в этих компрессорах, описаны в работах Петера Фенвика [18] и нашли свое применение в ряде других программ, например в X1.

Bzip использует адаптированную к BWT сортировку Бентли-Седжвика, во многом позаимствованную из вышеупомянутого семейства. После BWT выполняется преобразование методом

стопки книг, выходные данные которого сжимаются при помощи интервального кодирования (аналог арифметического сжатия) с использованием 1-2 кодирования и структурной модели Петера Фенвика.

Для того чтобы создать программу, которую можно свободно использовать в некоммерческих целях, в bzip2 интервальное кодирование было заменено на сжатие по методу Хаффмана. Видимо, благодаря этому bzip2 находит все большее распространение в различных областях применения и де-факто уже становится одним из стандартов. Сортировка в bzip2 изменена незначительно по сравнению с bzip. В основном, повышена устойчивость к избыточным данным и оптимизирован ряд процедур.

В BWC используются такие же методы, что и bzip и bzip2. А именно, оптимизированная сортировка, MTF, 1-2 кодирование и интервальное кодирование.

IMP использует собственную сортировку, очень быструю на обычных текстах, но буквально зависающую на данных, в которых встречаются длинные одинаковые последовательности символов. Сжатие полностью позаимствовано из bzip2.

Алгоритм сжатия, используемый в bzip2, также включен и в архиватор 7-Zip.

В szip, помимо упоминавшегося частичного сортирующего преобразования, реализована и возможность использования BWT. Реализована, прямо скажем, только для примера, без затей. А вот для сжатия используются очень интересные решения, представляющие собой некий гибрид MTF-преобразования и адаптивный кодер, берущий статистику из короткого окна преобразованных с помощью BWT данных. С участием автора szip и с использованием описанных решений был также создан архиватор ICT UC.

В Zzip применяется все те же испытанные временем структурная модель, сортировка Бентли-Седжвика и кодирование диапазонов.

ВА использует аналогичную сортировку. Но повышение устойчивости реализовано в ВА другим способом. Деление строк

по ключу прекращается в том случае, когда оказывается, что этим строкам предшествуют одинаковые символы. Еще одно новшество, реализованное в ВА — это выбор структурной модели MTF в отдельном проходе. Также за счет динамического определения размера блока улучшено сжатие неоднородных файлов. Для усиления сжатия английских текстов используется перепорядочивание алфавита.

В DC впервые реализован целый ряд новаторских идей. Во-первых, конечно, это модель сжатия, отличная от MTF — кодирование расстояний. Во-вторых, новый метод сортировки, очень быстрый на текстах, хотя и дающий слабину на сильно избыточных данных. И, наконец, большой набор методов препроцессинга текстовых данных, позволяющий добиться особенного успеха на английских текстах.

Отличительная особенность SBC — наличие мощной крипто-системы. Ни в одном из архиваторов, пожалуй, не реализовано столько алгоритмов шифрования, как в SBC. В SBC используется алгоритм BWT, ориентированный на большие блоки избыточных данных и позволяющий очень быстро сортировать данные с большим количеством длинных похожих строк. Вместо MTF в архиваторе используется кодирование расстояний, хотя пока не так эффективно, как в YBS и DC, но это компенсируется большим количеством фильтров (методов препроцессинга), настроенных на определенные типы данных.

ARC (автор Ian Sutton, которому также принадлежит PPM-архиватор BOA). Как и многие другие, использует BWT на основе сортировки Бенгли-Седжвика и MTF. Как и в SBC, дополнительно отслеживаются очень длинные повторы данных.

Первые версии YBS также использовали перемещение стопки книг, которое затем было заменено на кодирование расстояний. Что дало заметный выигрыш в степени сжатия.

Среди не распространяемых свободно компрессоров, описание которых опубликовано в научных трудах, можно отметить BKS98 и BKS99, которые принадлежат сразу трем авторам [10].

Эти компрессоры используют суффиксную сортировку и много-контекстовую модель MTF по трем последним кодам.

СРАВНЕНИЕ BWT-АРХИВАТОРОВ

Параметры, используемые для указания режима работы архиваторов, выбраны таким образом, чтобы добиться наилучших результатов в сжатии без особого ухудшения скорости.

Тестирование производилось на компьютере со следующей конфигурацией:

Процессор	Intel Pentium III 840 МГц
Частота шины	140 МГц
Оперативная память	256 Мбайт SDRAM
Жесткий диск	Quantum FB 4.3 Гбайт
Операционная система	Windows NT 4.0 Service Pack 3

Начнем со сжатия русских текстов, потому что BWT-архиваторы особенно эффективны именно для сжатия текстов. А русские тексты выбраны для того, чтобы показать эффективность сжатия в чистом виде, без использования текстовых фильтров, которые для русских текстов еще не созданы авторами описываемых программ. Файл имеет длину 1,639,139 байтов.

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с
YBS 0.03e	446,151	1.81	0.93
DC 0.99.298b -a	449,403	1.21	1.00
SBC 0.860	451,240	1.69	0.87
ARC (I.Sutton) b20	459,409	2.08	1.37
Compressia' b2048	462,873	2.92	2.66
BA 1.01b5 -24-m	463,214	2.17	1.26
Zzip 0.36 -mx -b8	467,383	1.96	1.65
szip 1.12 b21 o0	470,894	3.34	0.78
ICT UC 1.0	472,556	2.54	1.27
szip 1.12 b21 o8	472,577	2.32	1.12
GCA 0.90g -v	477,999	2.17	1.17
BWC/PGCC 0.99 m2m	479,162	1.69	0.83

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с
BWC/PGCC 0.99 m900k	503,556	1.56	0.83
szip 1.12 b21 o4	506,348	0.48	0.94
IMP 1.10 -2 u1000	506,524	1.07	0.64
bzip2/PGCC 1.0b7 -9	507,828	1.55	0.66

Как можно заметить, первенство удерживают компрессоры, использующие кодирование расстояний.

На английском тексте (2,042,760 байтов) некоторые архиваторы используют фильтры, тем самым заметно улучшая сжатие. Ниже приведены результаты, принадлежащие тем программам, которые показали наилучшие результаты в первом тесте.

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с	Использование фильтров
DC 0.99.298b	476,215	1.58	1.28	+
SBC 0.860 b3m1	489,612	1.59	0.96	+
YBS 0.03e	496,703	2.32	1.09	
DC 0.99.298b -a	500,421	1.50	1.18	
ARC (I.Sutton)	508,737	2.62	1.71	
b20				
BA 1.01b5 -24	512,696	2.87	1.53	+
Zzip 0.36 -mx -b8	515,672	2.84	2.08	+
Compressia b2048	517,484	3.67	2.12	
BA 1.01b5 -24-x	517,626	2.75	1.42	

При сжатии данных, представляющих собой исходные тексты программ, распределение среди лидеров тестов практически не меняется.

Для иллюстрации поведения BWT-архиваторов на неоднородных данных использован исполнимый модуль из дистрибу-

тива компилятора Watcom 10.0 wcc386.exe (536,624 байта). Для того, чтобы можно было судить об эффективности различных методов и режимов, некоторые строки помечены специальными знаками:

Архиватор, версия и па- раметры	Размер сжатого файла	Вре- мя сжа- тия, с	Вре- мя раз- жа- тия, с	Фильт ры	Умень- шенный размер блока	Авто- матиче- ское опреде- ление размера блока
YBS 0.03e - m512k	275,396	0.66	0.51	+	+	
YBS 0.03e	276,035	0.66	0.57	+		
SBC 0.860 m3a	278,061	0.98	0.69	+		+
ARC b5mm1	278,392	1.33	0.48	+	+	
DC 0.99.298b -b512	279,424	0.67	0.36	+	+	
DC 0.99.298b	279,759	0.66	0.37	+		
ARC mm1	280,052	1.34	0.46	+		
Zzip 0.36 -mx	291,199	0.74	0.66			
ARC (I.Sutton) b5	291,345	0.58	0.48		+	
ARC (I.Sutton)	292,979	0.58	0.48			
BA 1.01b5 - 24-z	293,489	0.82	0.64			+
DC 0.99.298b -a	293,807	0.52	0.39			
IMP 1.10 -2 u1000	294,679	0.38	0.18	+		
Compressia b512	297,647	0.97	1.16			
ICT UC 1.0	298,348	0.75	0.53			
BA 1.01b5	298,617	0.82	0.66			

Архиватор, версия и па- раметры	Размер сжатого файла	Вре- мя сжа- тия, с	Вре- мя раз- жа- тия, с	Фильт ры	Умень- шенный размер блока	Авто- матиче- ское опреде- ление размера блока
gzip 1.12 b21 o0	298,668	0.76	0.31			
gzip 1.12 b21 o4	299,249	0.27	0.39			
BWC/PGCC 0.99 m600k	304,996	0.58	0.37			
bzip2/PGCC 1.0b7 -6	308,624	0.63	0.26			

В качестве файла, содержащего смесь текстовых и бинарных данных, использовался Fileware.doc размером 427,520 байтов из поставки русского MS Office'95. Данный пример показывает, что иногда модель, использующая MTF, оказывается достаточно эффективной.

Архиватор, версия и па- раметры	Размер сжатого файла	Вре- мя сжа- тия, с	Вре- мя раз- жа- тия, с	Филь- тры	Умень- шен- ный размер блока	Авто- матиче- ское опреде- ление размера блока
SBC 0.860 m3a	126,811	0.69	0.42	+		+
DC 0.99.298b	127,377	0.38	0.18	+		
ARC b2	128,685	0.38	0.23	+	+	
YBS 0.03e - m256k	130,356	0.37	0.24		+	
Compressia b256	131,737	0.61	0.40		+	

Архиватор, версия и па- раметры	Размер сжатого файла	Вре- мя сжа- тия, с	Вре- мя раз- жа- тия, с	Филь- тры	Умень- шен- ный размер блока	Авто- матиче- ское опреде- ление размера блока
BA 1.01b5 - 24-r	132,651	0.41	0.30			
Zzip 0.36 -a1	132,711	0.65	0.40			
DC 0.99.298b -a	133,825	0.34	0.23			
YBS 0.03e	133,915	0.37	0.25			
BWC/PGCC 0.99 m600k	134,183	0.33	0.19		+	
bzip2/PGCC 1.0b7 -6	134,932	0.44	0.14		+	
szip 1.12 b21 o0	134,945	0.90	0.15			
IMP 1.10 -2 u1000	135,431	0.30	0.12			
ICT UC 1.0	136,842	0.41	0.29			
BA 1.01b5 - 24-z	137,566	0.49	0.31			+
szip 1.12 b21 o4	141,784	0.17	0.18			

Заключение

Несмотря на то, что преобразование Барроуза-Уилера было опубликовано сравнительно недавно, оно пользуется большим вниманием со стороны разработчиков архиваторов. И, пожалуй, еще впереди новые исследования, которые позволят повысить эффективность сжатия на основе BWT еще в большей степени.

Можно отметить характерные особенности архиваторов, использующих описанное преобразование, по сравнению с другими.

Скорость сжатия — на уровне архиваторов, применяющих словарные методы, например, LZ77. Разжатие, как правило, в 3-4 раза быстрее сжатия. Степень сжатия сильно зависит от типа данных.

Наиболее эффективно применение BWT-архиваторов для текстов и любых данных со стабильными контекстами. В этом случае рассматриваемые компрессоры по своим характеристикам близки к программам, использующим PPM. На неоднородных данных существующие архиваторы на основе BWT немного уступают лучшим современным компрессорам, использующим словарные методы или PPM. Впрочем, существуют способы компенсировать этот недостаток.

Расходы памяти в режиме максимального сжатия довольно близки у всех современных архиваторов. Наибольшее отличие наблюдается при декодировании. Наиболее скромными в этом отношении являются архиваторы, использующие алгоритмы семейства LZ77, а наиболее расточительными — PPM-компрессоры, требующие столько же ресурсов, сколько им требуется при сжатии. Архиваторы на основе BWT занимают промежуточное положение.

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. Albers S., v.Stengel B., Werchner R. A combined BIT and TIMESTAMP Algorithm for the List Update Problem. 1995.
3. Ambuhl C., Gartner B., v.Stengel B. A New Lower Bound for the List Update Problem in the Partial Cost Model. 1999.
4. Arimura M., Yamamoto H. Asymptotic Optimality of the Block Sorting Data Compression Algorithm. 1998.
5. Arnavaut Z., Magliveras S.S. Block Sorting and Compression // Proceedings of Data Compression Conference. 1999.

6. Arnavaut Z., Magliveras S.S. Lexical Permutation Sorting Algorithm.
7. Baik H-K., Ha D.S., Yook H-G., Shin S-C., Park M-S. A New Method to Improve the Performance of JPEG Entropy Coding Using Burrows-Wheeler Transformation. 1999.
8. Baik H., Ha D.S., Yook H-G., Shin S-C., Park M-S. Selective Application of Burrows-Wheeler Transformation for Enhancement of JPEG Entropy Coding. 1999.
9. Balkenhol B., Kurtz S. Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice.
10. Balkenhol B., Kurtz S., Shtarkov Y.M. Modifications of the Burrows and Wheeler Data Compression Algorithm // Proceedings of Data Compression Conference, Snowbird, Utah, IEEE Computer Society Press, 1999, pp. 188-197.
11. Balkenhol B., Shtarkov Y.M. One attempt of a compression algorithm using the BWT.
12. Baron D., Bresler Y. Tree Source Identification with the BWT. 2000.
13. Burrows M., Wheeler D.J. A Block-sorting Lossless Data Compression Algorithm // SRC Research Report 124, Digital Systems Research Center, Palo Alto, May 1994.
<http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z>
14. Chapin B. Switching Between Two On-line List Update algorithms for Higher Compression of Burrows-Wheeler Transformed Data // Proceedings of Data Compression Conference. 2000.
15. Chapin B., Tate S. Higher Compression from the Burrows-Wheeler Transform by Modified strings. 2000.
16. Deorowicz S. An analysis of second step algorithms in the Burrows-Wheeler compression algorithm. 2000.
17. Deorowicz S. Improvements to Burrows-Wheeler Compression Algorithm. 2000.

18. Fenwick P.M. Block sorting text compression // Australasian Computer Science Conference, ACSC'96, Melbourne, Australia, Feb 1996. <ftp://ftp.cs.auckland.ac.nz/out/peter-f/ACSC96.ps>
19. Ferragina P., Manzini G. An experimental study of an opportunistic index. 2001.
20. Kruse H., Mukherjee A. Improve Text Compression Ratios with Burrows-Wheeler Transform. 1999.
21. Kurtz S. Reducing the Space Requirement of Suffix Trees.
22. Kurtz S. Space efficient linear time computation of the Burrows and Wheeler Transformation // Proceedings of Data Compression Conference. 2000.
23. Kurtz S., Giegerich R., Stoye J. Efficient Implementation of Lazy Suffix Trees. 1999.
24. Larsson J. Attack of the Mutant Suffix Tree.
25. Larsson J. The Context Trees of Block Sorting Compression.
26. Larsson J., Sadakane K. Faster Suffix Sorting.
27. Manzini G. The Burrows-Wheeler Transform: Theory and Practice. 1999.
28. Nelson P.M. Data Compression with the Burrows Wheeler Transform // Dr. Dobbs Journal, Sept. 1996, pp 46-50. <http://web2.airmail.net/markn/articles/bwt/bwt.htm>
29. Sadakane K. A Fast Algorithm for Making Suffix Arrays and for BWT.
30. Sadakane K. Comparison among Suffix Array Constructions Algorithms.
31. Sadakane K. On Optimality of Variants of Block-Sorting Compression.
32. Sadakane K. Text Compression using Recency Rank with Context and Relation to Context Sorting, Block Sorting and PPM.
33. Schindler M. A Fast Block-sorting Algorithm for lossless Data Compression // Vienna University of Technology. 1997.
34. Schulz F. Two New Families of List Update Algorithms. 1998.

Глава 6. Обобщенные методы сортирующих преобразований

Глава 7. Предварительная обработка данных

Выбор метода сжатия

Если вы обнаружите ошибки или неточности в данном тексте, просьба сообщить о них авторам по адресу:
compression@graphicon.ru

Ответы к вопросам и упражнениям и исходные тексты программ вы можете найти на
<http://compression.graphicon.ru/>

УКАЗАТЕЛЬ ТЕРМИНОВ

A

Abrahamson.....79
 ADMSM79
 ARHANGEL77

B

Bentley-Sedgewick.....131
 Blending16
 Bloom.....77
 BMF80
 Boa77
 Burrows-Wheeler Transform.....89
 BWT89

C

CM69
 Context tree weighting83
 cPPMII73
 CTW83

D

DAFC78, 80
 Deterministic scaling59
 DHPC80
 Distance Coding116
 DMC.....84
 Dynamic Markov compression.....84

E

Escape19, 22
 Exclusion.....23

F

Fenwick Peter.....115
 Finite-context modeling.....13
 Full updates49

H

HA.....68
 Herklotz77
 Hirvola68

I

Inversed Frequencies.....119

L

Langdon5, 78
 LGHA77
 Local Order Estimation..... *См.* LOE
 LOE.....52, 70
 LOEMA77
 Lyapko77

M

Move To Front *См.* MTF
 MTF103

P

PPM20
 пример23
 PPM*50, 64
 PPMA35, 81
 PPMV.....35, 81
 PPMC.....36
 PPMd40, 73
 PPMD36
 PPMII.....73
 PPMN57, 71
 PPMonstr40, 73
 PPMY74
 PPMZ38, 77
 Prediction by Partial Matching*См.*
 PPM

R

Radix sorting	132
Recency scaling	57
Reordering	128
Rissanen	5, 78
RK	70
RKUC	70
RLE	106
Roberts	77

S

Sadakane Kunihiro	133
Schindler Mikael	116
Schindler Transform	100
Secondary Escape Estimation..... <i>См. SEE</i> Symbol Estimation..... <i>См. SSE</i>	
SEE	37
Sepulizing..... <i>См. Сепулирование</i>	
SEQUITUR	84
Shelwien	74
Shkarin	73
Smirnov	71
Sort Transformation	100
SSE	61
Suffix sorting	133
Sutton	77

T

Taylor	70
--------------	----

U

UHARC	77
Universal modelling and coding	5
Update exclusion	49
partial.....	50

V

Valentini	77
-----------------	----

W

Williams	80
----------------	----

WORD	81
------------	----

X

X1	77
----------	----

Z

Ziganshin	69
-----------------	----

A

Абрахамсон	79
Алгоритм Хаффмана	122

B

Барроуз Майк	88
Барроуза-Уилера преобразование <i>См. преобразование</i>	
Блум	77

B

Валентини	77
Вектор обратного преобразования	95
Вероятность ухода	22, 35
Взвешивание	16
неявное	19

D

Динамическое марковское сжатие	84
---	----

З

Зиганшин	69
----------------	----

И

Исключение	23
Исключение при обновлении	49
частичное	50, 60
Источник данных	5
Маркова	22

К

Кодер	5
Кодирование	5
1-2.....	111
длин повторов	См. RLE
расстоянийСм. Distance	
Coding	
статистическое	6
Кодировщик	5
Коды	
средняя длина.....	2
Компрессор	5
Dummy	35
PPM.....	29
Контекст	13
активный.....	13
детерминированный	50, 59
дочерний.....	15
левосторонний	13
правосторонний	13
-предок.....	15
разбросанный	70
родительский.....	15
ухода	38
Контекстная модель	14
детерминированная	41, 50
с замаскированными	
символами	41
с незамаскированными	
символами	41
уходов	38
Контекстное моделирование	11
ограниченного порядка	13
с полным смешиванием	17
с частичным смешиванием	
чистое порядка N	16, 79

Л

Лэнгдон	5, 78
---------------	-------

Ляпко.....	77
------------	----

М

Матрица циклических	
перестановок	92
Механизм уходов	22
Моделирование	5
адаптивное	10
блочно-адаптивное	10, 69
контекстное ограниченного	
порядка.....	13
полуадаптивное.....	9
статическое	9
Моделировщик	5
Модель	
иерархическая	123
источника данных.....	5, 15
структурная	123

Н

Накопленная частота.....	27
Наследование информации	55
отложенное.....	56

О

Обратные частоты	119
ОВУ	35
Оценка вероятности ухода	35
адаптивные методы ..См. SEE	
априорные методы	35
метод А.....	36, 78
метод В	36
метод С	36
метод D.....	36
метод Р	36
метод SEE-d1	40
метод SEE-d2	40
метод Х.....	36
метод ХС	36
метод Z	38, 71

П

Перемещение стопки книг <i>См.</i>	
MTF	
Переупорядочивание символов	128
Полное обновление счетчиков ...	49
Порядок модели RPPM.....	21, 66
Предсказание	
наиболее вероятных	
символов.....	60
по частичному совпадению	
.....	<i>См.</i> RPPM
Преобразование	
Барроуза-Уилера	89
сортирующее частичное....	100
Шиндлера	100

Р

Размер блока в BWT.....	126
Риссанен	5, 78
Робертс	77

С

Садакане Кунихико	133
Саттон.....	77
Сепулирование.....	<i>См.</i> Sepulizing
Символ ухода	19
Смешивание	16
Смирнов	71
Сортировка	
Бентли-Седжвика	131
быстрая	131
используемая в BWT	130
направление.....	129

поразрядная.....	132
суффиксов	133
Сравнение	
алгоритмов контекстного	
моделирования.....	83
архиваторов RPPM.....	76

Т

Тейлор	70
Теорема о кодировании источника	
.....	2, 6

У

Уилер Дэвид	88
Уилльямс.....	80

Ф

Фенвик Петер	115
--------------------	-----

Х

Характеристики	
алгоритмов RPPM	67
Херклоц.....	77
Хирвола.....	68

Ш

Шелвин.....	74
Шеннон	2, 6
Шиндлер Микаэль.....	116
Шкарин	40, 55, 73

Э

Энтропия.....	2
---------------	---