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

Методы сжатия данных

Раздел 1

Содержание книги:

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

Дата создания файла: 05.01.2003

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

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

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 словарными 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 сот-

ни 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 требу-

ется 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 мат-

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

рице 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.

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

Упражнение: Проделайте 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	а
1	а
2	а
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.

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

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

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

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 собственно 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	а.........р	9	
1	а.........д	7	
2	а.........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

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

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 “рдакраааабб”. Это символ ‘а’.

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

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. Соче-

тание 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

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

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	а< 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	а< 0>бракадабра	аа<10>бракадабр	аабракадабр
1	а< 3>кадабраабр	аб< 0>ракадабра	абраабракад
2	а< 5>дабраабрак	аб< 7>раабракад	абракадабра
3	а< 7>браабракад	ад< 5>абраабрак	адабраабрак
4	а<10>абракадабр	ак< 3>адабраабр	акадабраабр
5	б< 1>ракадабраа	бр< 1>акадабраа	браабракада
6	б< 8>раабракада	бр< 8>аабракада	бракадабраа

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

Номер строки	ST(1)	ST(2)	BWT
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 Барроуза-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 проделывает 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 верхушке 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.

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

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.

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

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

```
bbbbcccccdddddaaaaab   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*1 + 3*2 + 1*3 + 1*3 = 27 бит.

Упражнение: Произведите преобразование методом стопки книг последовательности "bbbbbccccdddcceaaaaa" 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

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

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 “aaaaabbbccccdd” в результате кодирования будет выглядеть как “aaa2bbb0ccc1dd”. 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) возрастать. 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 szip [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 кодов, что ухудшает нам статистику. Способ преодоления такой неприятности довольно прост. Следует отложить продвижение символа на верхушку списка в том случае, если этот символ не находится в позиции, соответствующей коду 1. Вот как будет

выглядеть преобразование последовательности “рдакраааабб” в этом случае:

Символ	Список	Выход
р	абдкр	4
д	арбдк	3
а	адрбк	0
к	адрбк	4
р	акдрб	3
а	аркдб	0
а	аркдб	0
а	аркдб	0
а	аркдб	0
б	аркдб	4
б	абркд	1

Рис. 5.13. Модифицированное MTF-преобразование строки “рдакраааабб”

Преимущество такой модификации видно на примере MTF-преобразования типичной для английских текстов последовательности “ttttTtttwtt”, полученной из части последнего столбца матрицы перестановок:

he_ ... t
he_ ... t
he_ ... t
he_ ... t
he_ ... T
he_ ... t
he_ ... t
hen_ ... t
hen_ ... w
hen_ ... t
hen_ ... t

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

Предположив, что упорядоченный список содержит символы в порядке {‘t’, ‘w’, ‘T’ ...}, в результате применения метода перемещения стопки книг получаем следующие результаты:

ttttTtttwtt	
00002100210	обычный MTF
00002000200	модифицированный MTF

Как видно из примера, на типах данных, обладающих описанными свойствами, усовершенствованный метод перемещения стопки книг дает распределение кодов MTF, которое лучше поддается сжатию за счет большего количества нулевых кодов.

Но в случае появления двойных редких символов, этот метод дает результаты хуже классического MTF. Например, если нам попалась последовательность “ttttTTtttwwtt”:

ttttTTtttwwtt	
0000201002010	обычный MTF
0000211002110	модифицированный MTF

Но, как уже было замечено, вероятность появления двух идущих подряд редких символов меньше, чем вероятность появления одного.

Картину также может испортить ситуация, когда мы имеем дело с символами, соответствующими не устойчивым контекстам, а участку смены одного контекста другим. В этом случае задержка в перемещении символа к вершине списка может сослужить плохую службу при отслеживании появления нового устойчивого контекста. Справедливости ради стоит отметить, что и в этом случае модифицированное MTF-преобразование все же обычно реагирует достаточно быстро.

Резюмируя, можно сказать, что описанный вариант метода перемещения стопки книг на текстовых данных оказывается лучше классического, а однозначно поручиться за сохранение

такого преимущества при обработке данных другого типа нельзя. Возможно, более правильным выбором будет предварительный анализ данных или подсчет числа отложенных перемещений на вершину списка для принятия решения, какой из вариантов использовать.

Упражнение: Какой из методов MTF-преобразования будет эффективнее для последовательности символов “aaacbaaa”? Предположим, что начальный упорядоченный список символов выглядит как {‘a’, ‘b’, ‘c’}.

Преимущество модифицированного алгоритма для текстовых данных можно оценить на примере файла book1 из набора файлов CalgCC, часто используемого для оценки архиваторов. Ниже приводятся частоты рангов для обоих методов перемещения стопки книг.

Ранг	MTF обычный (%)	MTF модифицированный (%)
0	49.77	51.43
1	15.36	14.93
2	7.91	7.45
3	5.28	4.96
4	3.79	3.67
5	2.91	2.81
6	2.35	2.29
7	1.98	1.94
8	1.68	1.63
9	1.45	1.44
10	1.26	1.23
11	1.06	1.05
12	0.91	0.90

Ранг	MTF обычный (%)	MTF модифицированный (%)
13-255	4.33	4.31

Рис. 5.14. Статистика рангов MTF-преобразования для файла book1

1-2 КОДИРОВАНИЕ

При кодировании длин повторов символов применительно к преобразованию Барроуза-Уилера стоит 2 задачи:
1) обеспечить кодирование чисел любой величины,
2) адаптироваться к изменению величины избыточности данных.

Данные задачи успешно решает алгоритм, нашедший применение в ряде архиваторов (bzip2, IMP, BWC) и названный 1-2 кодирование. Суть его заключается в том, что число, соответствующее количеству повторов, кодируется посредством двухсимвольного алфавита.

При использовании 1-2 кодирования в связке с MTF, мы отводим под нулевой ранг не один, а два символа (назовем их z_1 и z_2), увеличивая таким образом алфавит до 257 символов. Символ, отличный от z_1 и z_2, является признаком окончания записи числа повторов.

Число повторов кодируется следующим образом:

Число повторов	Код
1	z_1
2	z_2
3	z_1z_1
4	z_1z_2
5	z_1z_1
6	z_2z_2
7	$z_1z_1z_1$
8	$z_1z_1z_2$
9	$z_1z_2z_1$

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

Число повторов	Код
10	$z_1z_2z_2$
11	$z_2z_1z_1$
12	$z_2z_1z_2$
13	$z_2z_2z_1$
14	$z_2z_2z_2$
15	$z_1z_1z_1z_1$
...	

Рис. 5.15. 1-2 Кодирование чисел

Упражнение: Закодируйте посредством описанного алгоритма число 30. Какому числу соответствует последовательность $z_2z_2z_1z_1z_2$?

Как можно видеть, данный способ обеспечивает кодирование чисел в любом диапазоне. Этот метод соответствует и второму из предъявляемых требований. Увеличение избыточности данных, приводящее к возрастанию концентрации нулевых рангов MTF, приводит к увеличению частот символов z_1 и z_2. Причем, если преобладают короткие последовательности MTF-0, то частота символа z_1 превосходит частоту символа z_2.

Предвидя возможное замечание о том, что, например, число 7 приводит к большему возрастанию частоты символа z_1, чем число 1, отметим следующее. Как правило, распределение частот чисел повторов в среднем убывает с ростом числа, а частоты появлений близких по значению чисел близки. Причем обычно с ростом чисел различие частот уменьшается. Поэтому:

1) частота числа 7 больше частоты числа 1 только в редких, скорее вырожденных случаях,
2) увеличение частоты числа 7 за счет преобладания над частотами чисел 8, 9 и 10 приводит к более интенсивному использованию символа z_1, т.к. число символов z_1 в коде, соответст-

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

вующем числу 7, максимально среди всех трехсимвольных кодов.

В целом, преобладание частоты символа z_1 над частотой символа z_2 определяется преобладанием частоты одной группы символов над другой:

группа, приводящая к росту доли z_1	группа, приводящая к росту доли z_2
1	2
3	4
5	6
7	8
...	…
3,4	5,6
7,8	9,10
11,12	13,14
...	…
7-10	11-14
15-18	19-22
...	…

Рис. 5.16. Свойства 1-2 кодирования

После двух преобразований строка “абракадабра” выглядела как {4,3,2,4,3,2,0,0,0,4,0}. Подвергнем ее 1-2 кодированию. Воспользовавшись вышеприведенным рисунком, получим {4,3,2,4,3,2,z_1,z_1,4,z_1}.

Упражнение: Как будет выглядеть исходная строка в результате указанных двух преобразований и 1-2 кодирования, если вместо обычного применить модифицированный MTF? Для справки: после

BWT и модифицированного MTF была получена последовательность {4,3,0,4,2,0,0,0,0,4,1}.

Реализация 1-2 кодирования

```
// Функция 1-2 кодирования.
// Выводит последовательность символов z1 и
// z2, соответствующую числу count.

void z1z2( int count ) {

  // длина последовательности
  int len=0;

  // число 0 не кодируется
  if( !count ) return;

  // находим длину последовательности
  { int t = count+1;
    do { len++;
         t >>= 1;
    } while( t > 1 );
  }

  // кодирование последовательности
  do { len--;
    putc((count & (1<<len))? z2:z1, output );
  } while( len );
}

// кодирование
// использует функцию z1z2()
void encode( void ) {
  unsigned char c;
  int count = 0;          // число повторов
  while( !feof( input ) ) {
    c = getc( input );
    if( c == 0 ) count++; // считаем MTF-0
    else {
      // вводим число MTF-0
      z1z2( count );
      count = 0;
      // выводим символ, отличный от MTF_0
      putc( c, output );
```


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

```
    }
  }
  z1z2( count );
}

// декодирование
void decode( void ) {
  unsigned char c;
  int count = 0;           // число повторов
  while( !feof( input ) ) {
    c = getc( input );

    // чтение последовательности z1z2
    if       ( c == z1 ) {
      count += count + 1;
    } else if( c == z2 ) {
      count += count + 2;

    } else {
      // вывод MTF-0, заданные числом count
      while( count-- ) putc( 0, output );
      putc( in[i], output );
    }
    i++;
  }
  while( count-- ) putc( 0, output );
}
```

КОДИРОВАНИЕ РАССТОЯНИЙ

В течение нескольких лет метод перемещения стопки книг был непременным атрибутом архиватора, построенного на основе преобразования Барроуза-Уилера. В силу того, что этот алгоритм недостаточно точно учитывал соотношение частот символов, соответствующих разным рангам, разработчики постоянно стремились найти достойную замену.

Одним из наиболее распространенных был подход, при котором отдельно строилась модель для наиболее часто используемых символов. Для таких символов вероятности просчитывались отдельно. Впервые этот подход был описан Петером Фенвиком [18], но автору не удалось превзойти результаты модели, использующей традиционный подход. Более удачным было

применение кэширования наиболее частых символов в архиваторе szip, разработанном Шиндлером [33].

Самым поздним из методов, пришедших на смену MTF, является метод кодирования расстояний (Distance Coding). Первый же из архиваторов, использующих этот метод, сумел превзойти своих конкурентов по степени сжатия большинства типовых данных. Теперь, помимо архиватора DC, кодирование расстояний используют YBS и SBC.

Были публикации, посвященные родственному методу, названному Inversed Frequencies авторами одной из работ [5,1]. Помимо метода кодирования расстояний, ниже мы рассмотрим и его.

Возьмем для примера ту же строку “рдакраааабб”, полученную в результате преобразования Барроуза-Уилера.

В качестве первого, подготовительного этапа, нам следует определить первые вхождения символов в данную строку. Для этого в начало строки припишем все символы алфавита в произвольном, например, в лексикографическом, порядке. Поскольку при декодировании мы можем проделать то же самое, данная операция является обратимой.

Для удобства добавим в конец строки символ, означающий конец блока. При ссылке на этот символ мы можем распознать окончание цепочки символов, от которых эта ссылка сделана. Обозначим символ конца блока как ‘$’.

Итак, после этих приготовлений наша строка выглядит как “абдкррдакраааабб$”. Причем при декодировании нам на этом подготовительном этапе известны первые символы строки, “абдкр”, и последний ‘$’.

строка:	абдкррдакраааабб$
известные символы:	абдкр...........$

Теперь займемся собственно кодированием расстояний. Для этого берем первый символ ‘а’ и ищем ближайший такой же.

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

Расстояние до него равно 6 — это число иных символов между ‘а’. Но из этих шести символов нам уже известны четыре и при декодировании мы заранее знаем, что очередной символ ‘а’ никак не может попасть в эти позиции. Наша задача — закодировать номер той вакантной позиции, на которую выпадает этот символ. Это номер 6 – 4 = 2.

строка:	абдкррдакраааабб$
известные символы:	**а**бдкр..**а**........$
расстояние:	**2**

Аналогично кодируем еще несколько символов по очереди, подсчитывая число точек, символизирующих незанятые позиции, в строке известных символов.

строка:	абдкррдакраааабб$
известные символы:	а**б**дкр..а......**б**.$
расстояние:	2**8**
известные символы:	аб**д**кр.**д**а......б.$
расстояние:	28**1**
известные символы:	абд**к**р.да**к**.....б.$
расстояние:	281**1**
известные символы:	абдк**рр**дак.....б.$
расстояние:	2811**-**

При кодировании первого из символов ‘р’ вместо ссылки на следующий символ поставлен прочерк, потому что сразу после символа ‘р’ находится вакантная позиция и это означает, что никакой другой символ не сможет на эту позицию сослаться. Значит, нам нет необходимости выполнять кодирование этой ссылки. В данном случае мы наблюдаем специфический эффект, присущий методу кодирования расстояний, который позволяет избежать применения RLE.

строка:	абдкррдакраааабб$
известные символы:	абдкр**р**дак**р**....б.$
расстояние:	2811-**0**

Поскольку очередной символ 'р' занимает ближайшую вакантную позицию, мы кодируем его числом 0. Благодаря такому свойству метода кодирования расстояний, в нем достаточно легко решается проблема случайной смены контекста, ради которой требовалось специально совершенствовать метод перемещения стопки книг.

строка:	абдкррдакраааабб$
известные символы:	абдкрр**д**акр....б.**$**
расстояние:	2811-0**5**

Кодируя символ 'д', мы сделали ссылку на конец строки. При декодировании такая ссылка позволит понять, что символы 'д' закончились.

строка:	абдкррдакраааабб$
известные символы:	абдкррд**а**кр**а**...б.$
расстояние:	2811-05**0**
известные символы:	абдкррда**к**ра...б.**$**
расстояние:	2811-050**4**
известные символы:	абдкррдак**р**а...б.**$**
расстояние:	2811-0504**4**
известные символы:	абдкррдакр**аааа**б.$
расстояние:	2811-05044**---**
известные символы:	Абдкррдакрааа**а**б.**$**
расстояние:	2811-05044---**1**
известные символы:	Абдкррдакраааа**б**б$
расстояние:	2811-05044---1**-**

Рис. 5.17. Кодирование расстояний

В итоге получаем последовательность { 2,8,1,1,0,5,0,4,4,1 }.

Упражнение: Вычислите расстояния для строки "брраааакаакдрр".

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 "абдкр".

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

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

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.

Символ	первая позиция	Число символов	строка	Расстояния
а	2	5	рдакраааабб	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 “брраааакаакдрр”.

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.

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

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

```
eteksehendeynkrtdserttnregensknsgsgsedeneyswmessrne
xgynystslgyegsgstssrhmsstetehselxtptneessthndesddy
htksthwtpfdttegedmmhysyresprssneenselgetdemsetse,t
reehsetrtteseeeeeeesssdeedmnlendeedgtdgttdsgtteeysy
tddentnrxsltshtghnteeernsdpwlttensedehsteeswekheee
teneeeeseslteenestrngsthsdgeyeyrteetklrdtetteyodth
eegeercesyttesedenrtresnyssgttsslsawssygsssrewmsht
gt,etssgehnehessssehneesesdnrnekhtrsslthdsseestste
nbgeeessesdesyndtrdhpeesehesetsrerhyesdnwtlrrhoses
hsetdrptttsdhaynenetyntpgstesknhysftsgssdftgteeedu
```

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,,ttyogdodgdedndygnmotedgwkgodoowtdoddtotet
tndmeggkrdsgtohtdegteddrstttegtddewdddoootdgdntet
ststttstt!uetttdtte-eIttetny,hettItgrlttyIttnttyrt
rttttttettttttdtttstotttttttttntttttttttn,ddtkgnde;
ed,d,stefoefssfrsnsstyslw,fnoeadere,rteeynsfofhynn
nyoytted,yfnedhddtoldtnyhnnhyrtyryttmeryesfoyedney
oymd,sedesgnrrsysnssmydsspdyt'-dssehs,gynsdydgee'o
defddeynt,,tdnd,os;sttyysofy-nnetognfetdnyldlewhe-
odsttemshsdtsyteny,ngdefs,-offsnntettseyesgleay:es
dtsdgllredksyryes,rldosts;dtdeefgghsfrergkdgngkenw
```

Рис. 5.20. Изменяющийся фрагмент

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

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

```
edeeeeeIeydeyeeet'eeeIeeeeeeeeeeeeeeeeeeeeeeleeeee
eeeeeee!eeeeleeeeeeeeeeeeeeeeeeeeeeeeyedeereeeleee
eeeeeeeeeeslyadeeeeeeeeeyeeeeeeeeeeeeeedyereseeeee
eIueeeeeeeyeeeeeIsseseeIl'eeIetenhyalehcesyyseessn
s'dlesffao,dffeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeTeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeteeeeeeeeeeeeeeee
eeeeeeeeeeaerre,see,eywesldsysdhedeeetaeeesaeeyedo
```

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 деревья 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.

Указанные свойства послужили причиной того, что практически все реализации сжатия на основе преобразования Барроуза-Уилера уделяют повышенное внимание наиболее частым в текущем фрагменте символам. Для обработки редких символов важна лишь приблизительная оценка возможности их появления.

Рассмотрим две модели, обладающие описанными качествами — структурную и иерархическую модели [18].

Структурная модель. Кодирование символа осуществляется в два этапа. Сначала кодируется номер группы, к которой этот символ относится. А затем — номер символа внутри группы.

Размеры групп различны. Наиболее частые символы помещаются в группы, состоящие из небольшого числа символов. Предложенная Петером Фенвиком структурная модель предполагает самостоятельное существование нулевого и первого ранга, т.е. каждый из них представляет собой отдельную группу. В следующую (третью по счету) группу помещаются второй и третий ранги, затем — с четвертого по седьмой и т.д.

группа	кодируемые ранги			
0	0			
1	1			
2	2	3		
3	4	5	6	7
. . .			. . .	
8	128	129	...	255

Рис. 5.22. Структурная модель

Ниже приводятся частоты групп для файла book1 при использовании обычного и модифицированного MTF:

Номер группы	MTF обычный (%)	MTF модифицированный (%)
0	49.77	51.44
1	15.36	14.93
2	13.19	12.41
3	11.05	10.72
4	8.28	8.16
5	2.14	2.13
6	0.20	0.20
7	0.01	0.01
8	0.00	0.00

Рис. 5.23. Статистика распределения рангов в группах структурной модели

Иерархическая модель. Эта модель также делит символы на группы. И по такому же принципу. Отличие заключается в том, что в каждую группу добавляется символ, означающий уход к группе, находящейся на более низкой ступени иерархии.

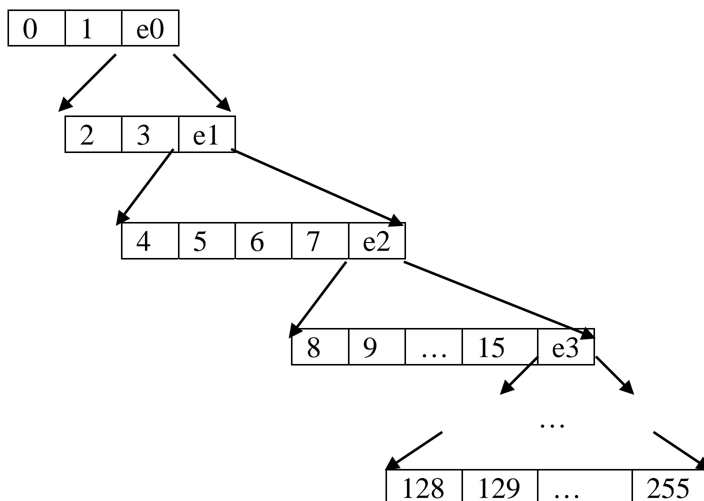

Рис. 5.24. Иерархическая модель

Эксперимент [16] показал, что структурная модель позволяет достичь немного лучшего сжатия и, в силу того, что для каждого символа в среднем требуется меньше операций арифметического кодирования, чуть быстрее.

Размер блока

Один из важных вопросов, который встает перед разработчиком архиватора на основе преобразования Барроуза-Уилера, заключается в выборе размера блока.

BWT — алгоритм, дающий наилучшую результативность при сжатии однородных данных. А однородные данные предполагают наличие устойчивых сочетаний символов. А раз сочетания не меняются, то увеличение размера блока приведут только к увеличению числа одинаковых символов находящихся рядом в результате преобразования. А увеличение количества находящихся рядом символов вдвое в идеале требует всего один дополнительный бит при кодировании. Таким образом, для однородных данных можно сделать однозначный вывод: чем блок больше, тем сжатие будет сильнее. Ниже приведены экспериментальные данные сжатия файла book1 из набора CalgCC. Тестирование производилось на компьютере следующей конфигурации:

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

Размер блока	bzip2 1.01 размер сжатого файла	bzip2 1.01 время сжатия, с	YBS 0.03e размер сжатого файла	YBS 0.03e время сжатия, с
100kb	270,508	0.61	255,428	0.65
200kb	256,002	0.66	239,392	0.66
300kb	249,793	0.71	232,681	0.71
400kb	243,402	0.72	225,659	0.73
500kb	242,662	0.73	224,782	0.74
600kb	241,169	0.75	222,782	0.76
700kb	237,993	0.76	218,826	0.77
800kb	**232,598**	0.77	**213,722**	0.77

Что касается данных неоднородных, то здесь картина иная. Размер блока надо выбирать таким, чтобы его край приходился на то место, где данные резко меняются. Причем, для BWT страшна не сама неоднородность, как таковая, а изменение статистики символов, предсказываемых устойчивыми контекстами. Например, нас не должен пугать тот факт, что в начале файла часто встречаются строки “abcd”, а в конце их место занимают “efgh”. Гораздо неприятнее, когда вместо строк “abcd” начинают появляться строки “abgh”. Это не означает, что файл, в котором наблюдается вторая ситуация, сожмется при помощи BWT-компрессора очень плохо. Но разница в сжатии по сравнению с архиваторами, использующими, например, словарные методы, на таких данных будет не в пользу BWT.

Для иллюстрации зависимости эффективности сжатия неоднородных данных от размера блока сожмем исполнимый файл из дистрибутива компилятора Watcom 10.0, wcc386.exe (536,624 байта).

Размер блока	bzip2 1.01 размер сжатого файла	bzip2 1.01 время сжатия, с	YBS 0.03e размер сжатого файла	YBS 0.03e время сжатия, с
100kb	309,716	0.66	279,596	0.60
200kb	308,668	0.66	277,312	0.61
300kb	308,812	0.66	277,285	0.61
400kb	309,163	0.66	277,374	0.60
500kb	**307,131**	0.66	**274,351**	0.60
600kb	308,624	0.66	276,026	0.60

Переупорядочивание символов

В отличие от многих других методов, сжатие, основанное на преобразовании Барроуза-Уилера, заметно зависит от лексикографического порядка следования символов. Замечено, что символы, имеющие сходное использование в словообразовании, лучше располагать поблизости.

Возьмем такие часто встречающиеся окончания слов русского языка, как “ый” и “ий”. Легко заметить, что им в большинстве случаев предшествуют одни и те же символы, например, буква “н”. Если эти окончания в результате сортировки каким-то образом окажутся в начале соседних строк матрицы перестановок, мы получим в последнем столбце рядом стоящие одинаковые символы.

Этот эффект особенно заметен на текстовых файлах. Для разных языков нюансы выбора лучшего переупорядочивания символов могут отличаться, но общее правило таково — все символы надо поделить на 3 лексикографически отдельных группы: гласные, согласные и знаки препинания.

Файл	Размер архива, байт	
	bzip2 1.01	YBS 0.03e
book1	232,598	213,722
book1, после переупорядочивания символов	231,884	212,975

Направление сортировки

Можно сортировать строки слева направо и в качестве результата преобразования использовать последний столбец матрицы отсортированных строк. А можно — справа налево и использовать первый столбец. Как делать лучше с точки зрения степени сжатия?

Первая стратегия подразумевает предсказание символа исходя из того, какие символы за ним следуют, а вторая — исходя из того, какие были раньше. В литературе для обозначения этих двух направлений используются словосочетания “following contexts” и “preceding contexts” соответственно (правосторонние и левосторонние контексты).

Практика показывает, что большинство архиваторов, использующих традиционные BWT и MTF, достигают лучшего сжатия на текстовых данных при использовании правосторонних контекстов. Для данных, имеющих не «лингвистическую» природу, лучше использовать левосторонние контексты. Например, это справедливо для исполнимых файлов.

Для компрессоров, которые не используют MTF, а проблему адаптации кодера к потоку преобразованных данных решают как-то иначе, выбор направления сортировки может быть иным. Например, DC и YBS многие исполнимые файлы, как и текстовые, сжимают лучше при сортировке слева направо.

Проделать самостоятельное сравнение очень просто. Возьмите ваш любимый архиватор и сожмите с его помощью ваш любимый файл. Затем переверните данные этого файла наоборот, сожмите полученное и сравните результаты.

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 “абабабаб”.

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 сим-

волами, 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;

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

V[i] — номер группы, к которой относится суффикс X[i]; сортировка выполняется до тех пор, пока все значения в V не станут разными;
S[i] — число суффиксов, относящихся к группе i;
k — порядок сортировки.

Как всегда, будем производить преобразование Барроуза-Уилера строки “абракадабра$” (добавим к строке символ ‘$’, означающий конец строки).

Шаг 1

Упорядочим все символы строки (для определенности предположим, что символ конца строки имеет наименьшее значение).

Затем заполним массив I значениями, равными позициям этих символов в исходной строке.

В массив V запишем по порядку номера групп, оставляя место для еще не упорядоченных элементов. Так, для символа ‘б’ укажем номер группы, равный 6, чтобы оставить возможность для упорядочивания всех пяти строк, начинающихся с символа ‘а’.

И, наконец, в массив S запишем размеры полученных групп. Небольшая хитрость, придуманная автором описываемого алгоритма, заключается в том, чтобы для групп, в которых осталась только одна строка, записывать отрицательное значение, равное количеству упорядоченных суффиксов до ближайшей не отсортированной группы. Это существенно ускоряет работу в случаях, когда у нас становится много отсортированных групп (на текстах такой момент, как правило, наступает уже на третьем-четвертом шаге сортировки).

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
упорядоченные символы:	$	а	а	а	а	а	б	б	д	к	р	р
I[i]	11	0	3	5	7	10	1	8	6	4	2	9
V[I[i]]	0	1	1	1	1	1	6	6	8	9	10	10
S[i]	-1	5					2		-2		2	

Шаг 2

Таким образом, завершена обработка суффиксов порядка k=0, т.е. одиночных символов. Теперь отсортируем пары (k=1). Для сортировки суффиксов каждой группы нам потребуются только значения номеров групп суффиксов (обозначаемых далее как Xk), находящихся на k символов правее сортируемых суффиксов. Например, для сортировки группы 1, соответствующей символу ‘а’, нам потребовались группы 6, 9, 8, 6 и 0 от символов ‘б’, ‘к’, ‘д’, ‘б’ и ‘$’.

Позиции:	0	1	2	3	4	5	6	7	8	9	10	11
исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
упорядоченные символы:	$	а	а	а	а	а	б	б	д	к	р	р
суффикс Xk	а	б	к	д	б	$	р	р	а	а	а	а
V[I[i]+1]	1	6	9	8	6	0	10	10	1	1	1	1
пары:	$а	аб	ак	ад	аб	а$	бр	бр	да	ка	ра	ра

После сортировки номеров групп:

V[I[i]+1]	1	0	6	6	8	9	10	10	1	1	1	1
упорядоченные пары:	$а	а$	аб	аб	ад	ак	бр	бр	да	ка	ра	ра
I[i]	11	10	0	7	5	3	1	8	6	4	2	9
V[I[i]]	0	1	2	2	4	5	6	6	8	9	10	10
S[i]	-2		2		-2		2		-2		2	

Шаг 3

Поскольку все пары теперь отсортированы, можно упорядочить четверки. Для этого каждую группу (которые представляют собой суффиксы, начинающиеся с одинаковых пар) отсорти-

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

руем в соответствии с парой символов, следующих после этих одинаковых пар. Прочерками отмечены уже отсортированные суффиксы, которые дальше обрабатывать нет необходимости.

позиции:	0	1	2	3	4	5	6	7	8	9	10	11
исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
упорядоченные пары:	$а	а$	аб	аб	ад	ак	бр	бр	да	ка	ра	ра
суффикс Xk	-	—	ра	ра	—	—	ак	а$	—	—	ка	$а
V[I[i]+2]			10	10			5	1			9	0

После сортировки номеров групп:

суффикс Xk	-	—	ра	ра	—	—	а$	ак	—	—	$а	ка
V[I[i]+2]			10	10			1	5			9	0
I[i]	11	10	0	7	5	3	8	1	6	4	9	2
V[I[i]]	0	1	2	2	4	5	6	7	8	9	10	11
S[i]	-2		2		-8							

Шаг 4

Как можно заметить, нам осталось отсортировать всего одну группу, состоящую из двух элементов.

позиции:	0	1	2	3	4	5	6	7	8	9	10	11
исходная строка:	а	б	р	а	к	а	д	а	б	р	а	$
упорядоченные четверки:	$абр	—	абра	абра	—	—	—	—	—	када	—	-
суффикс Xk	-	—	када	$абр	—	—	—	—	—	—	—	-
V[I[i]+4]			9	0								

После сортировки номеров групп:

V[I[i]+4]			0	9			1	5			9	0
I[i]	11	10	7	0	5	3	8	1	6	4	9	2
V[I[i]]	0	1	2	3	4	5	6	7	8	9	10	11
S[i]	-12											

Таким образом мы получили упорядоченные суффиксы, индексы которых записаны в массиве I:

i	I[i]	суффикс
0	11	$
1	10	а$
2	7	абра$
3	0	абракадабра$
4	5	адабра$
5	3	акадабра$
6	8	бра$
7	1	бракадабра$
8	6	дабра$
9	4	кадабра$
10	9	ра$
11	2	ракадабра$

Рис. 5.25. Результат сортировки суффиксов

Упражнение: Выполните сортировку строки “tobeornottobe”, используя описанный алгоритм.

Сравнение алгоритмов сортировки

Поскольку скорость сортировки во многом определяет быстродействие компрессоров, осуществляющих сжатие при помощи преобразования Барроуза-Уилера, над совершенствованием алгоритмов сортировки постоянно ведется работа. Можно отметить, что исследователи, тяготеющие к практическому применению, уделяют особое внимание сжатию типичных файлов, в то время как большинство публикаций в научных изданиях посвящено методам, позволяющим реализовать сортировку, устойчивую к вырожденным данным.

Для сравнения методов сортировки полезно ввести понятие средней длины совпадений (Average Match Length, AML), вычисляемой по следующей формуле:

$$AML = \frac{1}{n-1} \sum_{i=1}^{n-1} D(X[I[i]], X[I[i+1]])$$

где

N — размер блока данных, подвергаемого преобразованию;

X — массив суффиксов X[*i*], каждый из которых представляет собой строку, начинающуюся с *i*-ой позиции в блоке;

I — массив индексов лексикографически упорядоченных суффиксов;

D(*x*,*y*) — число совпадающих символов в одинаковых позициях строк *x* и *y*, начиная от первого символа строки и заканчивая первым несовпадением.

Исследования показали, что время, затрачиваемое суффиксной сортировкой, пропорционально логарифму средней длины совпадений, в то время как алгоритмы, основанные на быстрой сортировке или сортировке слиянием, демонстрируют, как правило, линейно пропорциональную AML зависимость. Впрочем, справедливости ради стоит отметить, что усовершенствования алгоритма сортировки Бентли-Седжвика, описанные выше, существенно сокращают время сортировки и поэтому этот алгоритм зачастую не уступает суффиксной сортировке даже на ряде вырожденных данных.

Цена такой устойчивости — повышенные накладные расходы на ее обеспечение.

Для сравнения ниже приведены экспериментальные данные, полученные на компьютере с процессором Intel Pentium 233 МГц и оперативной памятью 64 Мбайт. Были выбраны одни из наиболее оптимизированных представителей BWT-компрессоров для выявления слабых и сильных сторон различ-

ных методов. Требования к памяти участвующих в эксперименте программ довольно близки (от 6 до 8 размеров блока).

1) DC 0.99.015b (автор — Эдгар Биндер). В данном архиваторе использованы поразрядная сортировка и сортировка слиянием.
2) BA 1.01br5 (Микаель Лундквист), YBS 0.03e (Вадим Юкин), ARC (Ян Саттон). Во всех трех программах нашла свое применение сортировка Бентли-Седжвика вместе с поразрядной. Еще стоит упомянуть SBC 0.910 (Сами Мякинен), которая не участвовала в эксперименте по причине невозможности выделить сортировку отдельно от всех остальных процедур.
3) QSuf (Ларссон и Садакане). В этой программе реализована только суффиксная сортировка, немного улучшенная по сравнению с описанной выше.

Для эксперимента использовались следующие файлы:

1) book1 из тестового набора “Calgary Corpus”, как файл, обладающий основными свойствами типичных текстов. Размер файла — 768771 байтов.
2) file2 (1,000,000 байтов). Этот файл был составлен из нескольких больших одинаковых частей, позаимствованных из файла book1. Был сконструирован для проверки умения алгоритмов сортировки упорядочивать длинные одинаковые строки.
3) wat.c (1,890,501 байтов). Файл исходных текстов, полученный путем слияния исходных текстов, поставляемых с дистрибутивом компилятора Watcom C 10.0. Как уже отмечалась выше, средняя длина устойчивого контекста у таких файлов немного выше, чем у типичных текстов.
4) kennedy.xls (1,029,744 байтов). Данный файл использовался для анализа способности сортировщиков обрабатывать большое количество одинаковых строк небольшой длины.

	book1	file2	wat_c	kennedy
QSuf *	3.30	9.23	10.00	4.23
YBS	3.13	4.77	6.76	4.15
DC	2.36	4:23.59	6.92	2.97
ARC	4.17	4.34	6.43	5.00
BA	4.45	5.82	7.36	4.73
bzip2 **	3.03	4.23	6.81	4.07

Примечания:

* В программе QSuf реализована только сортировка, без сжатия и записи информации в файл. Это необходимо учитывать при сравнении быстродействия. Например, архиватор YBS потратил на сжатие преобразованного файла book1 примерно одну секунду.

** Архиватор bzip2 приведен только для справки, т.к. в нем реализовано сжатие на основе метода Хаффмана, а в остальных — используется арифметическое, которое заметно медленнее, но дает существенно лучшее сжатие (на 5-10 процентов). Кроме того, максимальный размер блока, который позволяет использовать bzip2 — 900 кбайт, что не позволяет достичь должного сжатия всех файлов, кроме book1. Хотя и дает небольшой прирост в скорости (2-3% на файле wat.c). Увеличение размера блока до размера файла могло бы замедлить скорость сжатия файла wat.c на 2-3%.

По результатам эксперимента можно сделать наблюдения:

1) Представитель алгоритмов, реализующих сортировку суффиксов, вел себя довольно ровно, хотя на типичных файлах оказался на последнем-предпоследнем месте. Как уже отмечалось, суффиксная сортировка хороша, но уж больно велики накладные расходы.
2) DC провалился именно там, где ожидалось — на длинных совпадениях. Архиваторы, использующие быструю сортировку, потенциально могут отстать от конкурентов на данных с большим количеством лексикографически упорядоченных совпадений. В принципе, можно успешно бороться и с теми и другими слабостями. Но можно предположить, что на типичных файлах, скорее всего, сортировка слиянием останется лучше на коротких контекстах, а сортировка Бентли-Седжвика — на длинных (что видно на примере файла wat.c).

3) Можно было рассмотреть частичное сортирующее преобразование в качестве альтернативного преобразования, не требующего таких хитростей при сортировке. Но здесь уже появляются другие требования при распаковке — б*о*льшие затраты 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

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

Компрессор и версия	Дата	Автор	Адрес
BWC 0.99	01.1999	Willem Monsuwe	mailto:willem@stack.nl ftp://ftp.stack.nl/pub/users/willem
IMP -2 1.12 WinImp 1.21	01.2000 09.2000	Conor McCarthy	mailto:imp@technelysium.com.au 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 0.99.298b	08.2000	Edgar Binder	mailto:EdgarBinder@t-online.de ftp://ftp.elf.stuba.sk/pub/pc/pack
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.sbbs.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	Alexan-der Ra-tushnyak	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 2.30b12	01.2002	Igor Pavlov	Mailto: support@7-zip.org Http://www.7-zip.org

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.

BA использует аналогичную сортировку. Но повышение устойчивости реализовано в BA другим способом. Деление строк по ключу прекращается в том случае, когда оказывается, что этим строкам предшествуют одинаковые символы. Еще одно новшество, реализованное в BA — 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 переупорядочивание 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 криптосистемы. 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 ос-

нове сортировки Бентли-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 многоконтекстовую 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

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

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с
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
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) b20	508,737	2.62	1.71	
BA 1.01b5 -24	512,696	2.87	1.53	+
Zzip 0.36 -mx -b8	515,672	2.84	2.08	+

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с	Использование фильтров
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 дистрибутива 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 параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с	Фильтры	Уменьшенный размер блока	Автоматическое определение 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			

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

Архиватор, версия и параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с	Фильтры	Уменьшенный размер блока	Автоматическое определение размера блока
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			
szip 1.12 b21 o0	298,668	0.76	0.31			
szip 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 параметры	Размер сжатого файла	Время сжатия, с	Время разжатия, с	Фильтры	Уменьшенный размер блока	Автоматическое определение 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		+	
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., Gartnerm 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-Wheeker

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

Сортировка параллельных блоков

Английское название метода — Parallel Blocks Sorting (PBS).

Два блока A и B называются параллельными, если каждому элементу A[*i*] первого блока поставлен в соответствие один элемент B[i] второго блока, и наоборот. Длины блоков L_A и L_B равны: $L_A = L_B = L$. Размеры элементов блоков R_A и R_B могут быть разными.

Основная идея метода PBS состоит в сортировке элементов In[*i*] входного блока In и их раскладывании в несколько выходных блоков Out_j на основании атрибутов A[*i*] этих элементов. Атрибут A[*i*] есть значение функции A, определяемой значениями предшествующих элементов In[*j*] и/или элементов P[*k*] из параллельного блока P:

```
A[i]=A(i, In[j], P[k]), i=0...L-1; j=0,...,i-1;
k=0,...,L-1
```

При декодировании осуществляется обратное преобразование: элементы из нескольких блоков Out_j собираются в один результирующий, соответствующий несжатому блоку In.

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

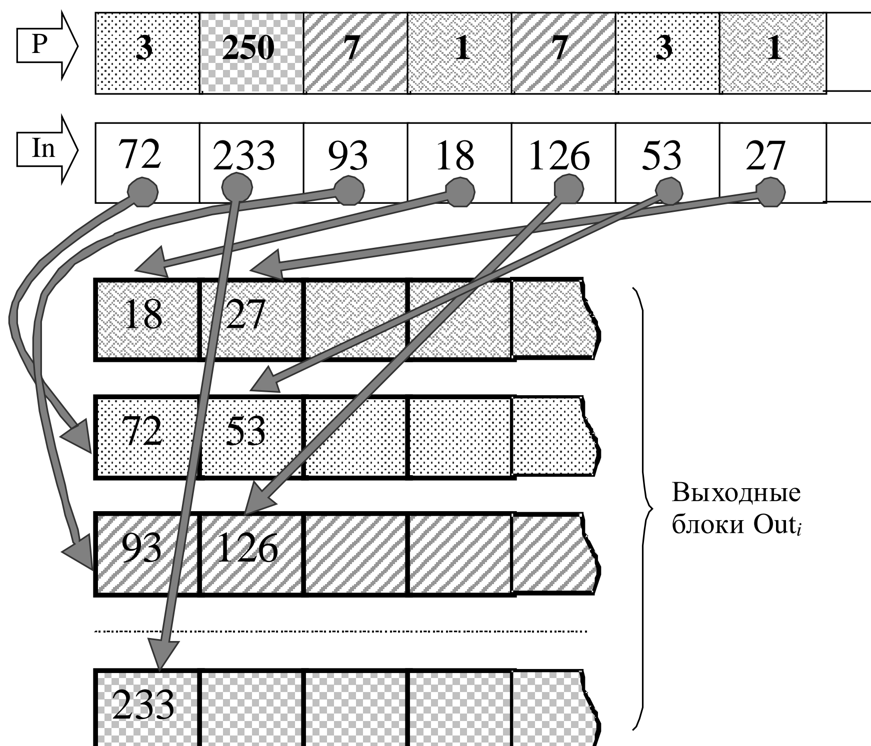

Рис. 6.1. Иллюстрация PBS: атрибут A[*i*] определяется значением элемента P[*i*]; значение A[*i*] показано штриховкой

Чем лучше значения In[i] предсказуемы по значениям A[i], тем эффективнее последующее сжатие блоков Out_j с помощью простых универсальных методов.

Методы этой группы **трансформирующие** и **блочные**, т.е. могут применяться только в том случае, когда известна длина блока с данными.

Размер данных в результате применения PBS, как и при ST/BWT, не изменяется. Для сжатия результата работы метода

может быть применена любая комбинация методов — RLE, LPC, MTF, DC, HUFF, ARIC, ENUC, SEM...

В общем случае скорость V_C работы компрессора (реализующего прямое, «сжимающее» преобразование) равна скорости V_D декомпрессора (реализующего обратное, «разжимающее» преобразование) и зависит только от размера данных, но не от их содержания: $V_C = V_D = O(L)$. Памяти требуется $2 \cdot L + C$. Константа C определяется особенностями реализации и может быть довольно большой. Операций умножения и деления нет.

Из краткого описания общей идеи видно, что

1) для распаковки нужно иметь не только содержимое сортированных блоков Out_j, но и их размеры;
2) параллельных блоков может быть и два, и больше;
3) если функция A не использует те P[k], которые находятся после текущей позиции i, т.е. $k=i+1,...,L$, то можно создавать параллельный блок P одновременно с текущим сортируемым In;
4) если A задана так, что принимает мало значений: от 2 до, допустим, 16, метод вполне может быть применим и к потокам данных;
5) если параллельного блока P нет, получаем ST/BWT как частный случай PBS.

Основной алгоритм преобразования

В простейшем случае сортирующего преобразования ST(1) значение атрибута вычисляется так: A[i]=In[i-1] ; при ST(2): A[i]=In[i-1]$\cdot 2^R$ + In[i-2] , и так далее.

В простейшем случае PBS A[i]=P[i], то есть значение атрибута равно значению элемента из параллельного блока. Выходных блоков столько, сколько значений принимают P[i]. Делая проход по P, считаем частоты значений атрибута, и в результате получаем размеры сортированных блоков (далее «контейнеров»), в которые будем раскладывать элементы из входного блока In:

```
for( a=0; a<n; a++) Attr[a]=0;     //инициализация    (1)
for( i=0; i<L; i++) Attr[P[i]]++; //подсчет частот   (2)
```

In — входной сортируемый блок длины *L*;
P — параллельный блок длины *L*;
L — количество элементов во входном блоке;
$n=2^R$ — число всех возможных значений атрибута;
Attr[*n*] — частоты значений атрибута.

Например, в P содержатся старшие 8 битов массива 16-битных чисел, а в In — младшие 8. Заметим, что этот процесс «дробления» можно продолжать и дальше, создавая вплоть до 16-и параллельных блоков: содержащий первые биты, вторые, и так далее. Кроме того, при сжатии мультимедийных данных часто уже имеются параллельные блоки: например, левый и правый каналы стереозвука, красная, зеленая и синяя компоненты изображения, и т.п.

Если функция A задана иначе, то и при подсчете частот формула будет иной:

```
// A[i]=P[i]&252 :
for( i=0; i<L; i++) Attr[ P[i]&252 ]++; // (2a)

// A[i]=255-P[i] :
for( i=0; i<L; i++) Attr[ 255-P[i] ]++; // (2b)

// A[i]=(P[i]+P[i-1])/2 :
Attr[ P[0] ]++;          // (2c)
for( i=1; i<L; i++) Attr[ (P[i]+P[i-1])/2 ]++;
```

Итак, после двух циклов — инициализации и подсчета частот — мы получили в массиве Attr длины контейнеров (сортированных блоков). Теперь можно вычислить, где какой контейнер будет начинаться, если их последовательно сцепить в один блок в порядке возрастания значения атрибута:

```
for( a=0, pos=0; a<n; a++) {    // (3)
  tmp=Attr[a]; // длина текущего a-го контейнера
// теперь там адрес-указатель на его начало:
  Attr[a]=pos;
         // (указатель на начало свободного места в нем)
// начало следующего — дальше на длину текущего:
  pos+=tmp;
}
```

В том же массиве Attr[n] теперь адреса-указатели на начала контейнеров. Остается только пройти по входному блоку, раскладывая элементы из него по контейнерам:

```
for(i=0; i<L; i++) Out[Attr[P[i]]++]=In[i];    //(4c)
```

Out — выходной отсортированный блок (sorted, transformed), содержащий все контейнеры. Подробнее:

```
for( i=0; i<L; i++) { // На каждом шаге:
// берем следующий элемент s из входного блока In:
 s=In[i];
// берем его атрибут a из параллельного блока P:
  a=P[i];
// и адрес свободного места в контейнере,
// задаваемом этим a;
  pos=Attr[a];
// кладем элемент s в выходной блок Out по этому адресу,
  Out[pos]=s;
//теперь в этом контейнере на один элемент больше:
  pos++;
  Attr[a]=pos; //а свободное место — на один элемент
               // дальше.
}
```

Функция A — та же, что и во втором цикле. То есть, для выше рассмотренных примеров:

```
(2a) a=P[i]&252;
(2b) a=255-P[i];
(2c) a=(P[i]+P[i-1])/2;
       //причем i=1...L-1, а когда i=0, a=P[0].
```

ОБРАТНОЕ ПРЕОБРАЗОВАНИЕ

Совершенно идентичны первые три цикла: инициализируем, в результате прохода по параллельному блоку находим длины контейнеров, затем определяем адреса начал контейнеров во входном отсортированном блоке In. Именно эти адреса — цель выполнения первых трех циклов. И только в четвертом цикле — наоборот: берем из отсортированного блока с контейнерами In, кладем в единый выходной блок Out:

```
for(i=0;i<L; i++) Out[i]=In[Attr[P[i]]++];     //(4d)
```

In — входной отсортированный блок со всеми контейнерами
Out — создаваемый выходной блок

Подробнее:

```
for( i=0; i<L; i++) {
  a=P[i];
  pos=Attr[a];
//так было при прямом (сжимающем) преобразовании:
//Out[pos]=In[i];
        //(в текущих обозначениях это записывается так:
        //In[pos]=Out[i];)
//а так делаем сейчас, при разжимающем преобразовании:
  Out[i]=In[pos];
  pos++;
  Attr[a]=pos;
}
```

ПУТИ УВЕЛИЧЕНИЯ СКОРОСТИ СЖАТИЯ

Итак, в обоих случаях выполняем два цикла от 0 до $n=2^R$, и два цикла от 0 до L. Какой из них займет больше времени? Чаще всего $R=8$, $n=256$, а L — от нескольких килобайт до десятков мегабайт. И четвертый, главный цикл — самый долгий, второй выполняется быстрее, а третий и особенно первый — совсем быстро.

Но возможна и обратная ситуация: если, например, $R=16$, $n=2^{16}=65536$, а $L=512$, т.е. требуется сжать поток блоков из 512- и 16-битных элементов (поток «фреймов»). Тогда наоборот, са-

мым долгим будет третий цикл, затем первый, четвертый и второй.

Два цикла при больших R

Если памяти хватает, поступим так: будем наполнять контейнеры не одновременно, проходя по In, P и записывая элементы In[i] в вычисляемые адреса блока Out, а последовательно: проходя по Out и вычисляя адреса в In, P, из которых нужно брать значения атрибута и элементы, помещаемые затем в Out. Останутся два цикла от 0 до L, а циклы от 0 до n=2^R — исчезнут.

При первом проходе заполняем вспомогательный массив Anext, содержащий цепочки указателей на позиции с одинаковыми атрибутами (A-цепочки). То есть в Anext[i] записываем указатель на следующую позицию (i+k) с таким же значением атрибута A[i], если таковая (i+k) в оставшейся части блока есть. В противоположном случае, если в оставшейся части блока атрибут не принимает значение A[i], в Anext[i] записываем i.

```
// при инициализации: вспомогательные массивы
int Anext[L];
int Beg[n];  // начала,
int End[n];  // концы и
int Flg[n];  // флаги наличия A-цепочек в текущем фрейме

// в цикле для каждого фрейма:
FrameNumber++;            // следующий L-элементный фрейм
for( i=0; i<L; i++) {
a=P[i];                   //вычислим значение атрибута a
if(Flg[a]==FrameNumber)
//если цепочка, определяемая этим a,
  {       // в текущем фрейме уже есть,
  e=End[a];// то конец ее – в массиве концов цепочек
  Anext[e]=i;// теперь прошлый конец указывает на новый
  }
else
  {  // иначе
  Flg[a]=FrameNumber;// запишем, что такая цепочка есть
  Beg[a]=i;          // и сохраним адрес ее начала
  }
Anext[i]=i;  // текущий элемент A-цепочки – ее конец
End[a]=i;    // укажем конец в массиве концов цепочек
}
```

Массив Flg нужен только затем, чтобы инициализировать не Beg и End перед *каждым* фреймом, а только Flg перед *всеми* фреймами. Если создать еще два массива Fprev и Fnext, соединяющие все элементы Flg с одним значением в одну F-цепочку (причем Fnext указывает на следующий элемент F-цепочки, Fprev — на предыдущий), то не придется при втором проходе линейно искать в Flg появлявшиеся в текущем фрейме A-цепочки. И цикла от 0 до n удастся избежать. Соответствующие изменения читатель легко внесет в алгоритм сам.

При втором проходе выполняем сортировку:

```
pos=f=0;
while (Fnext[f]!=f) {   // (линейный поиск следующей
  f=Fnext[f];   // A-цепочки в Flg, если без Fnext)
  i=Beg[f];        // первый элемент текущей A-цепочки,
  do {                  // и далее для всех ее элементов:
    Out[pos]=In[i];     // берем из In, кладем в Out
    pos++;              // последовательно,
    i=Anext[i];    //и так далее, проходя всю A-цепочку
  } while(i!=Anext[i])  //до конца
}
```

На этот раз во втором цикле уже нет обращений к P, поэтому можно совместить A и P. Памяти же требуется не

$2{\cdot}L + n{\cdot}\mathrm{sizeof}(*\mathrm{char})$, а

$2{\cdot}L + 5{\cdot}n{\cdot}\mathrm{sizeof}(*\mathrm{char})$,

так как Beg, End, Flg, Fprv, Fnxt занимают по $n{\cdot}\mathrm{sizeof}(*\mathrm{char})$ байт.

Два цикла при малых R

Если памяти доступно больше, чем необходимо для хранения $2{\cdot}L{+}n{\cdot}C$ элементов, и C достаточно велико — 256 или, например, 512, можно сразу «расформатировать» память под начала контейнеров:

```
#define K 256;
for( i=0; i<n; i++) Attr[i]=i*K;        // (1sm)
// под каждый из n контейнеров отведено K элементов
```

```
//  (1 «сектор»)
```

И затем, при проходе по In, P, если свободное место в каком-то контейнере кончается, будем записывать в конце «сектора» указатель на следующий сектор, выделяемый под продолжение этого контейнера:

```
FreeSector=n;
for( i=0; i<L; i++) {           // (2sm)
  a=P[i];  // берем атрибут a из параллельного блока P,
  pos=Attr[a];// и адрес свободного места в контейнере,
             // задаваемом этим a;
  Out[pos]=In[i];
  pos++;
if(pos%K+sizeof(*char)==K)//если свободное место кончи-
лось,
  {
  Out[pos]=FreeSector;//пишем  указатель  на  следующий
сектор
   // если FreeSector типа int32, а Out[i] типа int8, то:
      // Out[pos] =FreeSector%256;
      // Out[pos+1]=(FreeSector>>8)%256;
      // Out[pos+2]=(FreeSector>>16)%256;
      // Out[pos+3]=(FreeSector>>24)%256;
//свободное место= начало нового сектора:
  pos=FreeSector*K;
// свободный сектор на один сектор дальше:
  FreeSector++;
    }
  Attr[a]=pos;
}
```

Упражнение: Сравните этот цикл с (4c) и оцените, насколько он будет медленней.

Это очень выгодно, если $n \leq 256$, и, кроме того, после PBS выполняется еще один проход по всем элементам блока Out. Как правило, это так. Таким образом, весь алгоритм сводится к одному тривиальному циклу (1sm), и одному проходу по данным (2sm).

Увеличение скорости разжатия

Ускорение разжатия возможно, если длины контейнеров сжимать и передавать отдельно. Если L существенно больше n, описание длин занимает незначительную часть сжатого блока, так что потери в степени сжатия будут невелики. Тогда в массив Attr[n] при разжатии можно сразу записывать не длины, а начала контейнеров. Заметим, что и в случае ST/BWT это даст заметное увеличение скорости: подсчет частот элементов требует при разжатии дополнительного прохода по данным.

Пути улучшения сжатия

Общий случай

Если содержимое каждого контейнера (или некоторых из них) подвергается преобразованию, зависящему от его атрибута, то длины контейнеров надо обязательно передавать декомпрессору, иначе он неправильно построит таблицу частот по содержимому контейнеров.

Если параллельный блок уже существует и не изменяется при прямом и обратном преобразовании, функцию A можно строить без всяких ограничений, используя любую комбинацию арифметических, логических и других операций. Например,

```
A[i]=(int)P[i]/2; // деление целочисленное
```

Если сравнивать со случаем A[i] = P[i], то можно сказать, что каждая пара контейнеров объединяется в один «двойной» контейнер.

```
A[i]=P[i]^(n-1); // или, что то же самое, A[i]=n-1-P[i].
```

То есть контейнеры будут лежать в Out в обратном порядке по сравнению с A[i]=P[i]. Это полезно для второго отсортированного блока Out2, записываемого за первым Out1, особенно если данные — мультимедийные, поскольку в этом случае значения In[i] и In[i+1], как правило, близки.

Еще два варианта вычисления атрибута:

1.
```
A[i]=(P[i] & 16 == 0)? P[i] : (P[i]^15);
```
Младшие четыре бита изменяются только на +1 или –1: abs(P[i]&15 – P[i–1]&15)=1.

2.
```
A[i]=(P[i]<0)? (-P[i]*2+1 ) : P[i]*2;
```
Расстояние до нуля, а знак — в младшем бите.

Упражнение: Напишите к последним двум формулам формулы обратного преобразования.

Функция A может быть и адаптивной, т.е. производится периодическая корректировка каких-то ее коэффициентов, чтобы выходной блок больше соответствовал заданному критерию. Например, известно, что последние 20-30 контейнеров из 256-и полезно объединять в четверки, а остальные — в пары:

```
// последние 256-232=24 контейнера объединяем в четверки
K=232;
A[i]=(P[i]>K)? P[i]/4 : P[i]/2;
                // деление – целочисленное
```

При этом критерий используется такой: сумма первой производной выходного блока Out должна стремиться к нулю:

$$Sum = \sum_{i=1}^{L-1} (Out[i] - Out[i-1]) = 0,$$

где Out[*i*]–Out[*i*–1] — первая производная блока.

Тогда можно вычислять

```
SumCurrent+=Out[i]-Out[i-1], при текущем значении K,
SumMinus+=Outm[i]-Outm[i-1] при K=K-4,
SumPlus+=Outp[i]-Outp[i-1] при K=K+4,
```

и через каждые *StepK* шагов выбирать новое значение *K* по результату сравнения *SumCurrent*, *SumMinus* и *SumPlus*:

```
if (SumMinus<SumCurrent && SumMinus<SumPlus) K=K-4;
```

```
else if (SumPlus<SumCurrent) K=K+4;
```

Может оказаться полезным сохранять блок А вместо Р, особенно если параллельный блок строится одновременно с сортируемым — например, P[*i*] вычисляется по P[*i*-1] и In[*i*-1]. Тогда Р должен быть восстановим по А, и функция А должна быть биективной: по значению A[*i*]=A(P[*i*]) однозначно находится P[*i*]. Реально такая функция А сводится к перестановке: имеем контейнеры с атрибутами A[*i*]=P[*i*], затем перетасовываем их, меняя местами внутри единого выходного блока Out, но не изменяя их содержимого. Но формально А может выглядеть очень по-разному, содержать прибавление константы:

```
A[i]=P[i]+123; // результат берется по модулю n
```

логический XOR:

```
A[i]=P[i]^157;
```

сравнения и перестановки битов, в том числе циклическим вращением:

```
A[i]=(P[i]&(n/2))? ( (P[i]-(n/2))*2+1 ) : P[i]*2;
```

Упражнение: Как будет выглядеть А, собирающая вместе элементы In[*i*], соответствующие тем P[*i*], остаток которых от деления на заданное *K* одинаков? Начните со случаев *K*=4 и *K*=8.

Если функция А не адаптивна или зависит только от элементов параллельного блока, сами контейнеры, поскольку их длины известны, можно сортировать внутри единого выходного блока по их длинам. Это может улучшить сжатие даже при ST/BWT, особенно в случае качественных данных, а не количественных. Часто оказывается выгодным группировать короткие контейнеры в одном конце выходного блока, а длинные — в другом. И

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

при сжатии, и при разжатии процедура сортировки контейнеров по длинам добавится между циклами (2) и (3).

Рассмотрим последний вариант: весь блок A сохраняется и передается (компрессором декомпрессору), поэтому и строится он именно с учетом такой особенности. Если есть параллельный блок P, можно строить A так, чтобы и P преобразовывать на основе A, и чтобы суммарно блоки A, P, In сжимались лучше, чем P и In. Если нет P, то получается, что In распадается на две компоненты: «идеальную» — моделируемые, предсказываемые A[*i*], и «реальную» — отклонения In[*i*] от предсказанных значений A[*i*].

Двумерный случай

Параллельным блоком может быть предыдущая, уже обработанная строка таблицы. Либо предсказываемая (текущая), формируемая на основе одной или нескольких предыдущих строк. В обоих случаях целесообразно вычислять атрибут как среднее арифметическое соседних элементов, особенно для мультимедийных данных:

```
//если известны те, что выше и левее, то соседних - че-
тыре:
//     P[i-2]  P[i-1]  P[i] P[i+1]
//    In[i-2] In[i-1] In[i]
A[i]=(P[i-1]+  P[i]+P[i+1])/3;
A[i]=(P[i-1]+2*P[i]+P[i+1])/4;
A[i]=(P[i-1]+  P[i]+In[i-1])/3;
A[i]=(P[i-1]+2*P[i]+In[i-1])/4;
A[i]=(P[i-1]+  P[i]+2*In[i-1])/4;
A[i]=(P[i-1]+  P[i]+P[i+1]+In[i-1])/4;
```

Тогда, опять же, длины контейнеров (частоты значений атрибутов) надо обязательно передавать декомпрессору. Перед делением полезно делать округление до ближайшего числа, кратного делителю. Еще сложнее будут формулы, если использовать более одной предыдущей строки.

Заметим важное полезное свойство метода: увеличение сложности вычислений не влечет увеличения объема необходимой памяти.

Если рассматривать множество строк как единый блок, то в простейшем случае A[*i*]=P[*i*]=In[*i*–*W*] (*W* — число элементов в строке), то есть в качестве атрибута используется значение «верхнего» элемента. Тогда таблицу длин сохранять не нужно, но первые *W* элементов в неизменном виде — обязательно.

Упражнение: Опишите подробно алгоритм сортировки в случае A[i]=In[i-W]. Начните с W=1 и W=2.

Характеристики методов семейства PBS:

Степень сжатия: увеличивается в 1.0 ... 2.0 раза.

Типы данных: многомерные или многоуровневые данные.

Симметричность по скорости: в общем случае 1:1.

Характерные особенности: необходимо наличие как минимум двух параллельных блоков данных.

Фрагментирование

Цель: разбиение исходного потока или блока на фрагменты с разными статистическими свойствами. Такое разбиение должно увеличить степень последующего сжатия. В простейшем случае битового потока необходимо находить границы участков с преобладанием нулей, участков с преобладанием единиц, и участков с равномерным распределением нулей и единиц. Если

поток символьный, может оказаться выгодным разбить его на фрагменты, отличающиеся распределением вероятностей элементов: например, на фрагменты с русским текстом и фрагменты с английским.

Основная идея состоит в том, чтобы для каждой точки потока *X* (лежащей между парой соседних элементов) находить значение функции отличия FO(*X*) предыдущей части потока от последующей. В базовом простейшем варианте используется «скользящее окно» **фиксированной длины**: *Z* элементов до точки *X*, и *Z* элементов после нее.

Иллюстрация (*Z*=7):
…8536429349586436542 | 9865332 | *X* | 6564387 | 58676780674389…

Цифрами обозначены элементы потока, вертикальными чертами «|» границы левого и правого окон. *X* — не элемент потока, а точка между парой элементов, в данном случае между ”2“ и ”6“.

При фиксированной длине окон *Z* и **одинаковой значимости**, или **весе,** всех элементов внутри окон (независимо от их удаленности от точки *X*), значение функции отличия может быть легко вычислено на основании разности частот элементов в левом и правом окнах. Это сумма по всем 2^R возможным значениям V_i элементов. Суммируются абсолютные значения разностей: число элементов с текущим значением *V* в **левом** окне длиной *Z* **минус** число элементов с данным значением *V* в **правом** окне длиной *Z*. Суммируя 2^R модулей разности, получаем значение FO(*X*) в данной точке *X*:

$$FO(X) = \sum_{V=0}^{2^R-1} \left| CountLeft[V] - CountRight[V] \right|,$$

где CountLeft[V] — число элементов со значением V в **левом** окне;

CountRight[V] — число элементов со значением V в **правом** окне.

Видно, что если в обоих окнах элементы одинаковые, то сумма будет стремиться к нулю, если же элементы совершенно разные — к $2 \cdot Z$.

Для приведенного выше примера:

V =	2	3	4	5	6	7	8	9
FO(*X*) =	+|1-0|	+|2-1|	+|0-1|	+|1-1|	+|1-2|	+|0-1|	+|1-1|	+|1-0|

=**6**

После того как все значения FO(X) найдены, остается отсортировать их по возрастанию и выбрать границы по заданному критерию (заданное число границ и/или пока FO(X)>F_{min}).

Размер данных в результате фрагментирования увеличивается: либо появляется второй поток с длинами фрагментов, либо флаги границ в одном результирующем потоке.

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

Методы этой группы **трансформирующие** и **поточные**, т.е. могут применяться даже в том случае, когда длина блока с данными не задана.

Обратного преобразования не требуется.

В общем случае скорость работы компрессора зависит только от размера данных, но не от их содержания: Скорость=O(Размер). Памяти требуется порядка 2^{R+1}. Для левого окна O(2^R), и столько же для правого. Операций умножения и деления нет.

С точки зрения сегментации данных, метод отличается от RLE лишь тем, что выделяет из потока (или блока) не только цепочки одинаковых элементов, но и вообще отделяет друг от друга фрагменты с разными распределениями вероятностей значений элементов.

Из краткого описания общей идеи видно, что

1) порождается либо второй поток с длинами фрагментов, либо в исходный поток добавляются флаги границ (флаг может отвечать условию не только на значение одного элемента, но и условию на значение функции нескольких элементов);
2) задаваемыми параметрами могут быть (максимальная) длина окна Z, F_{min} — минимальное значение FO(X), минимальное расстояние между границами R_{min} (а в случае фрагментирования блока заданной длины — еще и число границ NG, минимальное и/или максимальное число границ: N_{min}, N_{max});
3) вычисление функции отличия при нескольких длинах окон — Z_1 , Z_2 , … , Z_n — в общем случае улучшит качество фрагментирования;
4) уменьшение веса элементов окна с удалением от точки X также в общем случае улучшит качество фрагментирования.

Основной алгоритм преобразования

Каков бы ни был размер окон Z, при проходе по входным данным In[N] в каждой точке X при переходе к следующей точке (X+1) достаточно следить за тремя элементами: вышедшим из левого окна (из-за сдвига точки X вправо), вошедшим в правое окно, и перешедшим из правого окна в левое. Но сначала разберем самый понятный алгоритм: с проходом по всем элементам окон для каждой точки X.

```
#define Z 32
for( i=0; i<n; i++)
  CountLeft[i]=CountRight[i]=0;    //инициализация    (1)

for( x=0; x<Z; x++) {   //цикл по длине окон:         (2)
```

```
  i=In[x]; //возьмем очередной элемент левого окна In[x]
  CountLeft[i]++;  //в левом окне элементов на 1 больше
  i=In[x+Z];          //и аналогично с правым окном
  CountRight[i]++;
  }
```

In[N] — входной фрагментируемый блок;
N — количество байтов во входном блоке;
n=2^R — каждый байт может иметь 2^R значений;
FO[N–2·Z] — значения функции отличия;
CountLeft[n] — частоты элементов в левом окне;
CountRight[n] — частоты элементов в правом окне.

После двух циклов, инициализирующих оба окна, начинаем проход по входному блоку. Изначально левая граница левого окна совпадает с началом данных.

```
for(x=Z; x<N-Z-1; x++) {
//точка X скользит с позиции Z до N-Z-1                   (3)
  f=0;                //текущее значение функции отличия
  for( i=0; i<n; i++)   //вычислим по формуле как сумму
    f+=abs(CountLeft[i]-CountRight[i]);
        /*т.е., если без abs()
              if (CountLeft[i]>CountRight[i])
                 f+=CountLeft[i]-CountRight[i];
              else f+=CountRight[i]-CountLeft[i];
        */
  FO[x]=f;       //и запишем в массив
  for(i=0;i<n;i++)  CountLeft[i]=CountRight[i]=0;//как  в
(1)
 for(j=x+1-Z;j<x+1;j++){ //цикл по длине окон: как в (2)
 CountLeft[In[j]]++;//подсчет  частот  элементов  в  левом
окне
 CountRight[In[j+Z]]++; //и аналогично в правом
 }
//но теперь левая граница левого окна в позиции (x+1-Z)
}
```

Наконец, последний цикл — это либо сортировка FO[N] с целью нахождения заданного числа максимальных значений (заданного числа самых «четких» границ), либо просто нахож-

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

дение таких FO[k], значение которых больше заданного. Последнее можно делать в основном цикле.

Упражнение: Внесите необходимые изменения в **(3)**, если задано минимальное значение отличия $F_{\min}$.

Пути улучшения скорости

Если последние два цикла внутри (3) перенести и выполнять их до первого, вычисляющего FO, то (1) и (2) не нужны. Но поскольку внутри цикла-прохода (3) по In достаточно следить за тремя элементами, в нем вообще не будет внутренних циклов. После выполнения (1) и (2) поступим так:

```
f=0;                    //исходное значение функции отличия
for( i=0; i<n; i++)  //вычислим по формуле как сумму (2a)
   f+= abs(CountLeft[i]-CountRight[i]);
FO[0]=f;       //и запишем в массив
```

А внутри основного цикла, проходящего по входному блоку In[N]:

```
for(x=Z; x<N-Z-1;x++) {
//точка X скользит с позиции Z до N-Z-1                 (3a)
//элемент, вышедший из левого скользящего окна:
  i=In[x-Z];
  f-=abs(CountLeft[i]-CountRight[i]);
//сумма без 1 слагаемого
  CountLeft[i]--;//теперь в левом окне таких на 1 меньше
  f+=abs(CountLeft[i]-CountRight[i]);
//обратно к полной сумме

 i=In[x+Z];  //элемент, вошедший в правое окно
//совершенно аналогично обновлению для левого окна
 f-=abs(CountLeft[i]-CountRight[i]);
//теперь в правом окне таких на 1 больше:
 CountRight[i]++;
 f+=abs(CountLeft[i]-CountRight[i]);
```

```
  i=In[x]; //элемент, перешедший из правого окна в левое
  f-=abs(CountLeft[i]-CountRight[i]);
//теперь в левом окне таких на 1 больше:
  CountLeft[i]++;
  CountRight[i]--;  //а в правом - на 1 меньше
  f+=abs(CountLeft[i]-CountRight[i]);

  FO[x]=f; //запишем значение функции отличия в массив
}
```

Упражнение: Как изменится значение f, если все три элемента — одинаковы, причем в левом окне таких было 7, а в правом 8?

Если *Z* существенно меньше *n*, вместо **(2a)** лучше делать другой цикл, до 2*Z. Внутри него будет прибавление модуля разности для текущего элемента к сумме, если этот модуль еще не вошел в сумму, и процедура, запоминающая какие элементы уже вошли в сумму:

```
f=0;            // исходное значение функции отличия
stacksize=0;    // и размера стека
for (x=0; x<2*Z; x++)                              (2z)
{
   i=In[x];     // текущий элемент
   s=0;         // просмотрим все обработанные:
   while(s<stacksize) if (Stack[s]==i) goto NEXT
   // если элемент еще не вошел
   f+= abs(CountLeft[i]-CountRight[i]);
   Stack[stacksize++]=i; // то вносим сейчас
NEXT:
}
FO[0]=f;      // запишем в массив
```

Точно так же можно модифицировать первый цикл внутри **(3)**, вычисляющий FO: вместо него будет **(2z)**, т.е. цикл не до n, а до $2 \cdot Z$.

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

Чтобы избавиться от второго цикла до n внутри **(3)**, заведем два массива с флагами FlagLeft и FlagRight, параллельные CountLeft и CountRight. В них будем записывать, в какой позиции *x* делалась запись в соответствующую позицию массива Count. При чтении же из Count будем читать и из Flag. Если значение в Flag меньше текущего *x*, то в соответствующей ячейке массива Count должен быть ноль.

> **Упражнение:** Перепишите **(3)** так, чтобы не было циклов до *n*.

Пути улучшения сжатия

Общий случай

Самый выгодный путь — использование **окна с убыванием веса элементов** при удалении от точки *X*. Линейное убывание, например: вес двух элементов, между которыми лежит точка *X*, равен *Z*, вес двух следующих, на расстоянии 1 от *X*, равен *Z*-1, и так далее, (*Z*-*d*) у элементов на расстоянии *d*, ноль у элемента, только что вышедшего из левого окна, и у элемента, который войдет в правое окно на следующем шаге.

Внутри основного цикла будет цикл либо до *Z* — по длине окна, либо до *n* — по всем возможным значениям элементов. В зависимости от значений *Z* и *n*, один из этих двух вариантов может оказаться существенно эффективнее другого.

В первом случае — *Z* существенно меньше *n*, выгоднее цикл до *Z* — вместо инкрементирования будем добавлять веса, то есть расстояния от *X* до дальней границы (самой левой в случае левого окна, самой правой в случае правого):

```
for( x=0; x<Z; x++) {   //цикл по длине окон:          (2`)
//  CountLeft[In[x]]++; //так было в цикле (2)
```


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

```
//  CountRight[In[x+Z]]++;//а теперь: прибавляем
// расстояние от левой границы до x:
CountLeft[In[x]]+=x+1;
CountRight[In[x+Z]]+=Z-x;// от x+Z до правой границы:
                        // 2*Z-(x+Z)
}
```

И аналогичные модификации в цикле **(3)**.

Во втором случае — *n* существенно меньше *Z*, выгоднее цикл до *n* — реализация гораздо сложнее: в дополнение к массиву со счетчиками появляется второй массив с весами. Зато сохраняется возможность действовать так, как в **(3а)**, не делая цикла по длине окна *Z* внутри основного цикла:

```
for(x=Z;x<N-Z-1;x++) {//точка X- с позиции Z до N-Z-1
(3a`)
 f=0;
 for(i=0;i<n; i++)
          // как изменятся веса элементов со значением
  {       // i при переходе к позиции x+1?
  WeightLeft[i]-=CountLeft[i]; // всего их CountLeft[i],
                               // каждый изменится на -1
// а каждый правый - на +1
  WeightRight[i]+=CountRight[i]);
// считаем значение отличия
  f+=abs(WeightLeft[i]-WeightRight[i]);
  }
// теперь сдвинем x на 1 вправо, внося и вынося элементы

//элемент, вышедший из левого скользящего окна:
   i=In[x-Z];
// теперь в левом окне таких на 1 меньше
   CountLeft[i]--;
      //вес его уже 0, поэтому WeightLeft[i] не изменен

   i=In[x+Z];    //элемент, вошедший в правое окно
   f-=abs(WeightLeft[i]-WeightRight[i]);
// сумма без 1 слагаемого
//теперь в правом окне таких на 1 больше:
  CountRight[i]++;
  WeightRight[i]++;//и вес их на 1 больше
// обратно к полной сумме
  f+=abs(WeightLeft[i]-WeightRight[i]);
```

```
// элемент, перешедший из правого окна в левое:
  i=In[x];
f-=abs(WeightLeft[i]-WeightRight[i]);
// сумма без 1 слагаемого

// теперь в левом окне таких на 1 больше:
  CountLeft[i]++;
  WeightLeft[i]+=Z;      // вес их на Z больше
  CountRight[i]--;     // а в правом - на 1 меньше
  WeightRight[i]-=Z;     // вес их на Z меньше
// обратно к полной сумме
  f+=abs(WeightLeft[i]-WeightRight[i]);

  FO[x]=f; // запишем значение функции отличия в массив
}
```

Упражнение: Как будут выглядеть циклы **(1)** и **(2)** ?

Второй путь улучшения качества фрагментирования — находить максимальное значение функции отличия при **нескольких длинах окна**: Z_1, Z_2, … , Z_q . Тогда эти вычисляемые значения FO_i придется делить на длину соответствующего окна Z_i, при котором это FO_i вычислено. И предварительно умножать на 2^K, чтобы получались величины не от 0 до 2, а от 0 до 2^{K+1}.

Третий путь — периодическое (через каждые *Y* точек) нахождение оптимальной длины окна *Z0*.

Четвертый — анализ массива FO, после того как он полностью заполнен.

Пятый — использование и других критериев при вычислении FO(*X*): например,
не $\Sigma|$ CountLeft[V] – CountRight[V] | ,
а Σ(CountLeft[V] – CountRight[V]$)^2$.

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

D-мерный случай

В двумерном случае появляется возможность следить за изменением характеристик элементов при перемещении через точку *X* по *K* разным прямым. В простейшем варианте — по двум перпендикулярным: горизонтальной и вертикальной.

Алгоритм может выглядеть, например, так: сначала проходим по строкам и заполняем массив FO значениями функции отличия при горизонтальном проходе: $FO[X]=FO_{ГОР}(X)$. Затем идем по столбцам, вычисляя отличия при вертикальном проходе $FO_{ВЕР}(X)$ и записывая в массив максимальное из этих двух значений — $FO_{ГОР}(X)$ и $FO_{ВЕР}(X)$. Дальше можно таким же образом добавить FO при двух диагональных проходах (см. рис. 6.2), причем не для каждой точки, а в окрестностях точек с максимумами FO[*X*].

3		2		4
	3	2	4	
1	1	*X*	1	1
	4	2	3	
4		2		3

Рис. 6.2. Иллюстрация двумерного случая с *K*=4

Кроме того, если хватает ресурсов, можно рассматривать отличие не отрезков длиной *Z*, отмеченных на *K* прямых, а секторов круга радиуса *Z*, разбитого на *K* секторов. Это могут быть, например, две половины круга, или же четыре четвертинки круга.

Теперь можно варьировать не только *Z*, пытаясь найти оптимальное значение, но также и *K*.

Таким образом, будут найдены «контуры» — границы, отделяющие области с разными характеристиками совокупности содержащихся в этих областях элементов.

В трехмерном случае: либо используем $K_П$ прямых, либо $K_К$ секторов круга, либо $K_С$ секторов сферы. Причем оптимальные

(с точки зрения вычисления) значения $K_П$ — не 2, 4, 8…, как в двумерном, а 3, 7, 13…

> **Упражнение:** Нарисуйте эти варианты с тремя прямыми, семью и тринадцатью. Как продолжить процесс дальше?

Совершенно аналогично и в D-мерном случае.

> **Характеристики метода фрагментирования:**
>
> **Степень сжатия:** увеличивается в 1.0 ... 1.2 раза.
>
> **Типы данных:** неоднородные данные, особенно с точными границами фрагментов.
>
> **Симметричность по скорости:** более чем 10:1.
>
> **Характерные особенности:** может применяться не только для сжатия данных, но и для анализа их структуры.

Глава 7. Предварительная обработка данных

Предварительная обработка данных выполняется до их сжатия как такового и призвана улучшить коэффициент сжатия. Схема кодирования в этом случае приобретает вид:

исходные данные → препроцессор → кодер → сжатые данные,

а схема декодирования:

сжатые данные → декодер → постпроцессор → восстановленные данные.

Препроцессор должен так видоизменить входной поток, чтобы коэффициент сжатия преобразованных данных был в среднем лучше коэффициента сжатия исходных, «сырых» данных. Система препроцессор-постпроцессор работает автономно; кодер «не знает», что он сжимает уже преобразованные данные. Постпроцессор восстанавливает исходные данные без потерь информации, поэтому результаты работы и само существование системы препроцессор-постпроцессор могут быть не заметны для внешнего наблюдателя.

В общем случае, преобразования, выполняемые препроцессором, могут быть реализованы в кодере и при этом быть такими же эффективными с точки зрения улучшения сжатия. Но предварительная обработка позволяет существенно упростить алгоритм работы кодера и декодера, дает возможность создания достаточно гибкой специализированной системы сжатия данных определенного типа на основе универсальных алгоритмов. Модули «препроцессор-постпроцессор» обычно можно применять в сочетании с различными кодерами и архиваторами, повышая их степень сжатия и, возможно, скорость работы.

В этой главе рассмотрено несколько способов предварительной обработки типичных данных. Предполагается, что алгоритм сжатия предназначен для кодирования источников с памятью. Описываемые методы препроцессинга достаточно хорошо известны в кругу разработчиков программ сжатия, но не получили широкого освещения в литературе.

Препроцессинг текстов

В последние несколько лет приобрели популярность и были существенно развиты методы предварительной обработки текстовой информации. В настоящее время специализированный препроцессинг текстов, позволяющий заметно улучшить сжатие, используется в таких архиваторах как ARHANGEL, JAR, RK, SBC, UHARC, в компрессорах DC, PPMN.

Использование словарей

Идея преобразования данных с помощью словаря заключается в замене каких-то строк текста на коды фраз словаря, соответствующих этим строкам. Часто указывается просто номер фразы в словаре. Пожалуй, метод словарной замены является самым старым и известным среди техник предварительного преобразования текстов, да и любых данных вообще. Сама словарная замена может приводить как сжатию представления информации, так и к расширению. Главное, чтобы при этом достигалась цель преобразования — изменение структуры данных, позволяющее повысить эффективность последующего сжатия.

Можно выделить несколько стратегий построения словаря. По способу построения словари бывают:

- статическими, т.е. заранее построенными и полностью известными как препроцессору, так и постпроцессору;
- полуадаптивными, когда словарь выбирается из нескольких заранее сконструированных и известных препроцессору и постпроцессору *или* достраивается, при этом один из имеющихся словарей берется за основу;
- адаптивными, т.е. целиком создаваемыми специально для сжимаемого файла (блока) данных на основании его анализа.

В качестве фраз обычно используются [4]:

- целые слова;
- последовательности из двух символов (биграфы);
- пары букв, фонетически эквивалентных одному звуку;
- пары букв согласная-гласная или гласная-согласная;
- последовательности из n символов (n-графы).

Как правило, существует огромное количество последовательностей, которые в принципе могут стать фразами, поэтому необходимо применять какие-то критерии отбора. Обычно в словарь добавляются:

- последовательности, чаще всего встречающиеся в сжимаемом тексте или в текстах определенного класса;

- одни из самых часто используемых последовательностей, удовлетворяющие некоторым ограничениям;
- слова, без которых едва ли сможет получиться связный текст: предлоги, местоимения, союзы, артикли и т.п.

Словарь n-графов

Судя по всему, наибольшее распространение в современных архиваторах и компрессорах получила стратегия статического словаря, состоящего из последовательностей букв длины от 2 до небольшого числа *n* (обычно 4…5). В большинстве случаев размер словаря равен примерно 100 таким фразам. К достоинствам данного типа словаря можно отнести:

- малый размер;
- отсутствие жесткой привязки к определенному языку;
- обеспечение существенного прироста степени сжатия;
- простота реализации.

Небольшой размер словаря обусловлен двумя причинами:

- это упрощает кодирование фраз словаря;
- дальнейшее увеличение размера словаря улучшает сжатие лишь незначительно (справедливо для BWT, и, в меньшей степени, LZ), либо даже вредит в большинстве случаев (справедливо для PPM).

Обычно тексты представлены в формате «plain text», когда один байт соответствует одному символу. Так как размер словаря мал, то в качестве индекса фраз выступают неиспользуемые, или редко используемые значения байтов. Например, если обрабатывается текст на английском языке, то появление не-ASCII байтов со значением 0x80 и больше маловероятно. Поэтому мы можем заместить все биграфы “th” на число 0x80. Если байт 0x80 все же встретится в обрабатываемых данных, то он может быть передан как пара (<флаг исключения>, <0x80>), где флаг исключения может быть равен, например, 0xFF. Это обеспечивает однозначность восстановления текста постпроцессором.

Упражнение: Каким образом будет передан байт, равный самому флагу исключения?

Недостатком данного типа словаря является отсутствие формализованного алгоритма построения. Как показали эксперименты, добавление в словарь самых часто используемых последовательностей букв небольшой длины действительно улучшает сжатие, но такой подход не приводит к получению оптимального состава словаря. Поэтому построение словаря делается в полуавтоматическом режиме, когда для заданного кодера и заданного тестового набора текстов эмпирически определяется целесообразность внесения или удаления из словаря той или иной фразы исходя из изменения коэффициента сжатия тестового набора.

Например, для английского языка хорошо работает словарь, представленный в табл. 7.1. В него входят 45 последовательностей длины 2 (биграфов), 25 последовательностей длины 3 (триграфов) и 16 последовательностей длины 4 (тетраграфов). Список отсортирован в примерном порядке убывания полезности фраз (внутри каждой группы *n*-графов — слева направо и сверху вниз).

Таблица 7.1

Биграфы	Триграфы	Тетраграфы
th er in ou an en	the ing and	ight self
ea or ll is on ar	for ess ver	ward this
st gh ed ee om oo	was igh ous	have been
ow ss ur ld at sh	our ell een	able nder
id sa ic tr al il	had ich ugh	ttle with
as ir ec ul ly et	her out his	ound reat
ai ch ot it av im	ead ard ome	that what
ol to qu	est ght rom	from ther
	ith	

Пример

Строка

he took his vorpal sword in hand

будет преобразована в последовательность:

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

he <to>ok <his> v<or>p<al> sw<or>d <in> h<and>

В угловых скобках показаны *n*-графы, заменяемые на их индекс в словаре.

Сначала отображаются тетраграфы (в разбираемой строке их нет), затем триграфы, и, в самую последнюю очередь, биграфы.

Рассмотрим пример реализации простейшей словарной замены для английских текстов. Пусть словарь состоит только из 10 биграфов, в качестве кодов биграфов используются не-ASCII байты со значениями 128…137, исключительные ситуации не отслеживаются, т.е. предполагается, что во входном файле отсутствуют не-ASCII символы.

```
const int BIGRAPH_NUM = 10;
const char blist [BIGRAPH_NUM][3] = {
   "th", "er", "in", "ou", "an",
   "en", "ea", "or", "ll", "is"
};
/*предполагаем, что bnum — глобальная переменная и
   инициализируется нулями; в этом массиве будем хранить
   коды биграфов
*/
unsigned char bnum [256][256];

int code = 128;
for (int i = 0; i < BIGRAPH_NUM; i++){ //заполним bnum
   bnum [ blist[i][0] ] [ blist[i][1] ] = code++;
}
int c1, c2;
c1 = DataFile.ReadSymbol();
while ( c1 != EOF ) {
   c2 = DataFile.ReadSymbol();
   if ( c2 == EOF ) {
      PreprocFile.WriteSymbol (c1);
      break;
   }
   if (bnum[c1][c2]){
   //такой биграф имеется в словаре, произведем замену
      PreprocFile.WriteSymbol (bnum[c1][c2]);
      c1 = DataFile.ReadSymbol();
   }else{
      PreprocFile.WriteSymbol (c1);
```

```
        c1 = c2;
    }
}
```

Для русского языка неплохие результаты показывает словарь, описанный в табл. 7.2. Заметим, что словарь для русских текстов имеет меньший размер, чем для английских.

Таблица 7.2

Биграфы	Триграфы	Тетраграфы
`то ст ов ен по аз ак ер ол ор`	`ств был при`	`лько врем`
`он ел ет ам от ом ас ан ин ск`	`про ере ого`	`тобы огда`
`на за ар ик пр ев ив ит ил ед`	`ост ись енн`	`азал ольш`
`ем ть ал ат ав ся ес об од ос`	`вет`	`еред отор`
`ис ог им ег ич сь`		

За счет описанной словарной замены достигается значительное улучшение сжатия текстов (см. табл. 7.3). В качестве тестового файла был использован роман на английском языке «Three men in a boat (to say nothing of the dog)» («Трое в лодке, не считая собаки»), электронный вариант которого занимал около 360 кбайт в исходном виде. Отказ от сравнения с помощью больших текстовых файлов Book1 и Book2, входящих в состав стандартного набора CalgCC, был обусловлен тем, что они являются не вполне типичными текстами. Первый файл содержит значительное количество опечаток, а второй — большой объем служебной информации о форматировании текста.

Таблица 7.3

Архиватор	Тип метода сжатия	Размер архива исходного файла	Размер архива обработанного файла	Улучшение сжатия
Bzip2, вер. 1.00	BWT	109736	107904	1.7%
WinRAR, вер. 2.71	LZ77	130174	126026	3.2%
HA a2, вер. 0.999c	PPM	108443	106831	1.5%

Заметим, что в случае Bzip2 и HA сжатие могло быть улучшено. С одной стороны, коэффициент сжатия алгоритма BWT зависит от номеров символов в алфавите, а нами не делалось никаких попыток переупорядочить более подходящим образом ASCII-символы и номера добавленных нами фраз. С другой стороны, HA использует небольшой объем памяти, и часть накап-

ливаемой в PPM-модели статистики периодически выбрасывается, что ухудшает предсказание и, соответственно, сжатие.

Эффективность использования n-графового словаря с другим составом фраз совместно с BWT архиваторами оценена в [2].

Обычно применение n-графового словаря улучшает сжатие компрессоров, использующих PPM или BWT, на 2%.

Степень сжатия компрессоров на базе методов Зива-Лемпела может быть заметно улучшена за счет увеличения размера словаря препроцессора и использования фраз большей длины. В принципе, фразы могут включать не только буквы, но и пробелы, что, кстати, приводит к ухудшению сжатия в случае BWT или PPM.

Словарь LIPT

В качестве примера словаря, в котором фразами являются слова, рассмотрим схему Length Index Preserving Transformation (LIPT) — «преобразование с сохранением индекса длины» [1]. В словарь включаются самые часто используемые слова, определяемые исходя из анализа большого количества текстов на заданном языке и, возможно, определенной тематики. Под словом здесь понимается последовательность букв, ограниченная с двух сторон символами, не являющимися буквами («не-букв»). Весь словарь делится на части (подсловари). В зависимости от своей длины L_i слово попадает в часть словаря с номером i. В пределах подсловаря фразы отсортированы в порядке убывания частоты, т.е. самое часто используемое слово имеет минимальный индекс 0. Каждое слово исходной последовательности, которому соответствует какая-та фраза словаря, кодируется следующим образом:

флаг	длина слова (номер подсловаря)	индекс в подсловаре

В качестве алфавита для записи длины слова и индекса авторами алгоритма предлагается использовать алфавит языка. Например, если мы работаем с английским языком, то “a” соответствует 1, “b” — 2, …, “z” — 26, “A” — 27, …, “Z” — 52, и, далее, “aa” — 53, “ab” — 54… Нулевой индекс явным образом

не передается. Если, допустим, слово “mere” имеет индекс 29 в своем подсловаре 4, то оно будем преобразовано так (для большей доходчивости различные части кода фразы выделены подчеркиванием):

“mere” → “<флаг>_d_C”.

Если индекс равен 56, то отображение будет таким:

“mere” → “<флаг>_d_ad”.

Индекс указан как “ad”, поскольку он записывается в позиционной системе счисления, и первая буква соответствует старшему порядку. Конец последовательности, передающей индекс, нет нужды указывать явно, поскольку, если мы рассматриваем “mere” как слово, то оно должно ограничиваться какой-то «небуквой», которая и станет маркером конца записи индекса.

Если слово отсутствует в словаре, то оно без изменений копируется в файл преобразованных данных.

Эксперименты показывают, что для различных алгоритмов сжатия выгоднее использовать разные алфавиты длины слова и индекса. Также иногда имеет смысл использовать иной принцип разбиения словаря. Рассмотрим результаты сжатия текста «Three men in a boat (to say nothing of the dog)» для следующих трех алгоритмов LIPT:

1. алгоритм 1 — практически соответствует авторскому, но алфавит ограничен только строчными английскими буквами;
2. алгоритм 2 — алфавиты длины слова и индекса отличаются от алфавита букв и не пересекаются между собой;
3. алгоритм 3 — словарь разбивается на подсловари не только по критерию длины фраз, но и по соответствию слова определенной части речи[9]; используются такие же алфавиты, что и в алгоритме 2.

---

[9] Эта модификация LIPT разработана и реализована М.А.Смирновым летом 2001 года

Словарь LIPT был построен на основании анализа примерно 50 Мбайт английских текстов различного характера. Общий объем словаря составил около 53 тыс. фраз, или 480 кбайт. Для реализации алгоритма 3 примерно 11.6 тысячам слов был присвоен атрибут принадлежности к определенной части речи, например, «the» — артикль и т.д. Классификация была очень груба — использовалось всего лишь 9 категорий. Если слово могло относиться к нескольким частям речи, то выбиралась самая часто употребляемая форма (понятно, здесь был определенный произвол). Слова, не получившие такого лексического атрибута, трактовались как существительные. Схема кодирования для алгоритма 3 имела вид:

флаг	часть речи	длина слова	индекс в подсловаре

Например:

“mere” → “<флаг><прилагательное><длина = 4><индекс>”,

где индекс определяет положение фразы в подсловаре 4-буквенных прилагательных.

Результаты эксперимента приведены в табл. 7.4.

Таблица 7.4

Архиватор	Тип метода сжатия	Алгоритм LIPT 1		Алгоритм LIPT 2		Алгоритм LIPT 3	
		Размер архива	Улучшение сжатия	Размер архива	Улучшение сжатия	Размер архива	Улучшение сжатия
Bzip2, вер. 1.00	BWT	102797	6.3%	100106	8.8%	101397	7.6%
WinRAR, вер. 2.71	LZ77	118647	8.9%	122118	6.2%	125725	3.4%
HA a2, вер. 0.999c	PPM	100342	7.5%	98838	8.9%	98173	9.5%

Таким образом, для LZ77 лучше всего подходит обычный алгоритм LIPT. Очевидно, что LZ77 плохо использует корреляцию между строками, и основной выигрыш достигается за счет уменьшения длин совпадения и величин смещений. Алгоритм 2 можно признать компромиссным вариантом. Для PPM-компрессоров имеет смысл использовать алгоритм 3.

Преобразование заглавных букв

Заглавные буквы существенно увеличивают число встречающихся в тексте последовательностей и, соответственно, приводят к ухудшению сжатия по сравнению с тем случаем, если бы их не было вообще. Способ частичного устранения этого неприятного явления очевиден. Если слово начинается с заглавной буквы, то будем преобразовывать его так, как показано на рис. 7.1 [2].

флаг	первая буква, преобразованная в строчную	оставшаяся часть слова

Рис. 7.1. Преобразованный вид слова, начинавшегося с заглавной буквы

При этом под словом понимается последовательность букв, ограниченная с двух сторон «не-буквами».

Например, если в качестве флага используется байт 0x00, то преобразование может иметь вид:

“_Если_” → “_<0x00>если_”

Для BWT- и PPM-компрессоров отмечается улучшение сжатия, если после флага вставляется пробел, т.е. когда результат трансформации имеет вид типа “_<0x00>_если_”. Невыгодно преобразовывать слова, состоящие только из одной заглавной буквы.

Иногда в текстах встречается много слов, набранных полностью заглавными буквами. Очевидно, что в этом случае описанное преобразование не только не помогает, но и, возможно, даже вредит. Поэтому целесообразно использовать еще одно отображение, переводящее слова, состоящие целиком из заглавных букв, в соответствующие слова из строчных букв (рис 7.2).

флаг 2	последовательность букв слова, преобразованных в строчные

Рис. 7.2. Преобразованный вид слова, целиком состоявшего из заглавных букв

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

Если в роли флага 2 выступает байт 0x01, то справедлив такой пример отображения:

“_АЛГОРИТМЫ_” → “_<0x01>алгоритмы_”.

Как уже указывалось, добавление пробела после флагов улучшает сжатие BWT и PPM компрессоров.

Может показаться, что описанный алгоритм вносит избыточность за счет использования флага даже в тех случаях, когда в нем нет особой необходимости. Если текущее слово является первым встреченным после точки, восклицательного или вопросительного знака, то его начальная буква наверняка является заглавной, и флаг излишен. Естественно, для обеспечения правильности декодирования необходимо либо:

- особо обрабатывать ситуации, когда первая буква все-таки строчная, например, использовать специальный флаг, сигнализирующий об исключении;
- делать компенсирующее преобразование, отображающее все строчные буквы, встречаемые после знаков конца предложения, в заглавные.

Несмотря на кажущуюся эффективность, в случае компрессоров BWT и PPM такая техника работает хуже ранее рассмотренных, поскольку нарушает регулярность в использовании флагов и искажает контекстно-зависимую статистику частот символов. Если же необходимо разработать препроцессор текстовых данных исключительно для LZ-архиваторов, то действительно следует избавиться от ненужных флагов.

В табл. 7.5 описывается эффект от использования преобразования заглавных букв для архиваторов различных типов на примере сжатия уже упоминавшегося электронного варианта книги «Three men in a boat (to say nothing of the dog)». Чтобы продемонстрировать роль вставки пробела после флагов, мы рассмотрели два алгоритма предварительной обработки. Алгоритм 1 преобразует:

- слова, начинающиеся с заглавной буквы, но далее состоящие из одной или более строчных, в соответствии с правилом, изображенном на рис. 7.1;

- слова из 2 и более символов, содержащие только заглавные буквы, в соответствии с правилом, приведенном на рис. 7.2.

Алгоритм 2 использует эти же два отображения, но добавляет после флагов пробел.

Таблица 7.5

Архиватор	Тип метода сжатия	Алгоритм 1		Алгоритм 2	
		Размер архива	Улучшение сжатия	Размер архива	Улучшение сжатия
Bzip2, вер. 1.00	BWT	108883	0.8%	108600	1.0%
WinRAR, вер. 2.71	LZ77	128351	1.4%	128563	1.2%
HA a2, вер. 0.999c	PPM	107285	1.1%	107137	1.2%

Добавление пробела заметно улучшило сжатие для алгоритма BWT, благоприятно повлияло на эффективность PPM, но сказалось отрицательно в случае LZ77.

Модификация разделителей

Текст содержит не только буквы, но и символы-разделители. Разделители бывают:

- естественными — это знаки препинания, пробелы;
- связанными с форматированием текста — символы конца строки (СКС), символы табуляции.

Символы-разделители в большинстве случаев плохо предсказываются на основании контекстно-зависимой статистики. Особенно плохо предсказываются символы конца строки, т.е. пара символов {перевод каретки, перевод строки} CR/LF или символ перевода строки LF.

Коэффициент сжатия BWT и PPM компрессоров может быть улучшен, если преобразовать знаки препинания и СКС, выделив их из потока букв. Наиболее эффективным и простым способом модификации является добавление пробела перед этими разделителями [2]. Например:

“очевидно,” → “очевидно_,”,

при этом:

“очевидно_,” → “очевидно__,”.

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

Преобразование однозначно: когда постпроцессор встречает пробел, он смотрит на следующий символ, и, если это разделитель (знак препинания или СКС), пробел на выход не выдается.

Положительный эффект отображения объясняется тем, что пробел встречается в таких же контекстах, что и преобразуемые знаки. Поэтому выполнение преобразования приводит к следующему:

- уменьшается количество используемых контекстов и, следовательно, увеличивается точность накапливаемой статистики;
- пробел сжимается часто сильнее, чем соответствующий знак препинания или СКС, и при этом предоставляет несколько лучший с точки зрения точности предсказания контекст для последующего разделителя.

Укажем несколько способов повышения производительности схемы:

- в случае СКС пробел в большинстве случае выгодно добавлять и после символа (-ов) конца строки; при этом лучше воздержаться от преобразования, если первый символ новой строки не является ни буквой, ни пробелом;
- сжатие обычно улучшается в среднем, если не делается преобразование знаков препинания, за которыми не следует ни пробел, ни СКС;
- не следует вставлять пробел перед знаком препинания, если предыдущий символ не является ни буквой, ни пробелом.

> **Упражнение:** Почему необходимо делать модификацию в том случае, когда предыдущий символ является пробелом?

В целом, эффект от модификации разделителей менее стабилен, чем от словарной замены или перевода заглавных букв в

строчные. Выигрыш практически всегда достигается главным образом за счет преобразования СКС и составляет 1-2%. В случае BWT преобразование знаков препинания дает неустойчивый эффект, лучше обрабатывать таким образом только СКС.

Результаты сжатия различными архиваторами преобразованного текста книги «Three men in a boat (to say nothing of the dog)» приведены в табл. 7.6.

Таблица 7.6

Архиватор	Тип метода сжатия	Размер архива обработанного файла	Улучшение сжатия
Bzip2, вер. 1.00	BWT	108864	0.8%
WinRAR, вер. 2.71	LZ77	130835	-0.5%
HA a2, вер. 0.999c	PPM	106582	1.7%

Данные табл. 7.6 подтверждают, что использование описанного преобразования в сочетании с методов LZ77 бессмысленно.

Специальное кодирование символов конца строки

Как уже отмечалось, СКС плохо сжимаются сами и ухудшают сжатие окружающих их символов.

Очевидно, что если бы мы заменили СКС на пробелы, то сжатие текстов улучшилось бы существенным образом. Этого можно достигнуть, искусственно разбив исходный файл на два блока: собственно текст, в котором СКС заменены на пробелы, и сведения о расположении СКС в файле, т.е., фактически, информация о длинах строк. Если в расположении СКС имеется достаточно строгая регулярность, то сумма размеров сжатого блока преобразованного текста и сжатого блока длин строк будет меньше размера архива исходного файла [2].

Эффективность специального кодирования СКС напрямую зависит от характера распределения длин строк в тексте. Если текст отформатирован по ширине строки, то сведения о длинах строк могут быть представлены очень компактно. Напротив, если в тексте много коротких строк, максимальная длина строки в символах не выдерживается постоянной, то, скорее всего, спе-

циальное кодирование СКС будет мало полезно или вовсе не выгодно.

Одним из возможным способов задания длины строки является количество пробелов, лежащих между предыдущим и текущим СКС. Например:

Строка	Длина строки в пробелах
Twas_brillig,_and_the_slithy_toves	5
Did_gyre_and_gimble_in_the_wabe;	6
All_mimsy_were_the_borogoves,	4
And_the_mome_raths_outgrabe.	4

Поскольку мы измеряем длину строки не в символах, а, скорее, в словах, то количество наблюдаемых длин строк не будет велико, и с помощью арифметического кодирования можно достаточно компактно представить информацию о расположении СКС в тексте. В простейшем случае достаточно кодировать длины на основании безусловных частот их использования.

Постпроцессор получит два блока данных:

Блок	Данные блока
Текст	Twas_brillig,_and_the_slithy_toves_ Did_gyre_and_gimble_in_the_wabe;_ All_mimsy_…
Длины строк	5, 6, …

Так как постпроцессору известно, что первая строка содержит 5 пробелов, то он заменит 6-ой по счету пробел на СКС:

"…toves_ Did…" → "…toves<СКС>Did…"

Для следующей строки он заменит на СКС 7-ой пробел и т.д. Таким образом текст будет полностью реконструирован.

Укажем несколько способов повышения степени сжатия.

Оценка вероятности длины строки может быть улучшена, если принимать во внимание символы, примыкающие к пробелу слева и справа, и если учитывать текущую длину строки в символах. Это позволит компактнее представить информацию о длинах строк с помощью арифметического кодирования.

Эксперименты показывают, что можно заменять на пробелы не все СКС, а только соответствующие достаточно длинным строкам. Зачастую это способствует повышению общей степени сжатия. При этом, однако, появляется проблема определения при каких же именно длинах строк выгодно «запускать» преобразование. В компрессоре PPMN используется следующий способ нахождения порога, при превышении которого длиной L строки включается преобразование СКС. Величина L принимается за постоянную на протяжении обработки всего файла. Для вычисления L анализируется достаточно большой блок (до 32 кбайт) исходного файла и собирается информация о количестве (частоте) строк с определенной длиной L, измеряемой в символах. В подавляющем большинстве случаев частоты длин строк максимальны в районе наибольшей наблюдаемой длины L_{max} строк, а затем достаточно резко падают с уменьшением L. Поэтому, двигаясь от L_{max} к нулевой длине, находим сначала максимум частоты, а затем, продолжая уменьшать L, ищем пороговое значение. Порог принимается равным L, для которой частота впервые становится меньше средней частоты длин строк. С другой стороны, в качестве порога целесообразно выбирать точку, в которой произошло многократное падение частоты, что характерно для текстов со строгим выравниванием по ширине.

Данная техника позволяет определять порог с очень высокой точностью в случае текстов достаточно простым форматированием.

На рис. 7.3 изображено распределение частот длин строк, полученное для первых 32 кбайт текста «Three men in a boat (to say nothing of the dog)». Видно, что $L_{max} = 73$, максимум частоты находится в точке $L = 72$, а порог равен 65 символам, поскольку при этом частота впервые становится меньше средней частоты, если считать от точки $L = 72$.

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

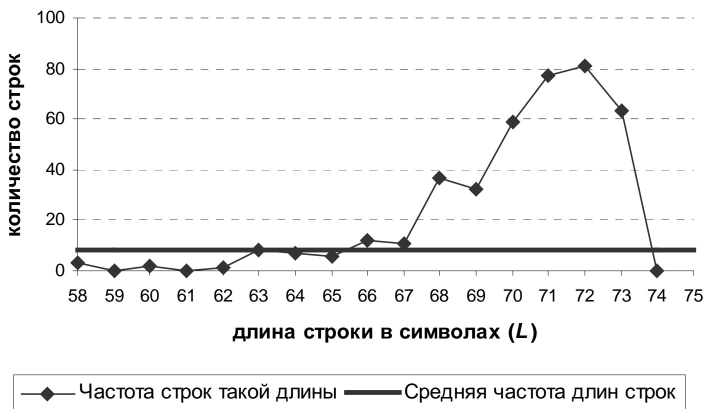

Рис. 7.3. Распределение частот длин строк, полученное для первых 32 кбайт текста «Three men in a boat (to say nothing of the dog)»

В качестве примера укажем в табл. 7.7 результаты сжатия тремя выбранными нами архиваторами преобразованного текста «Three men in a boat (to say nothing of the dog)». Длина строк измерялась в пробелах, преобразовывались только СКС в строках с длиной 65 символов и более. Длины строк сжимались с помощью арифметического кодера на основании их безусловных частот. Размер закодированного описания длин получился равным 793 байтам, и это число добавлялось к размеру архива обработанного файла при сравнении эффективности преобразования относительно разных методов сжатия (BWT, LZ77, PPM).

Таблица 7.7

Архиватор	Тип метода сжатия	Размер архива исходного файла	Размер архива обработанного файла плюс размер описания длин строк	Улучшение сжатия
Bzip2, вер. 1.00	BWT	109736	106069	3.3%
WinRAR, вер. 2.71	LZ77	130174	126042	3.2%
HA a2, вер. 0.999c	PPM	108443	105018	3.2%

Видно, что специальное кодирование СКС выгодно использовать в сочетании с любым универсальным методом сжатия.

ОЦЕНКА ОБЩЕГО ЭФФЕКТА ОТ ИСПОЛЬЗОВАНИЯ ПРЕДВАРИТЕЛЬНОЙ ОБРАБОТКИ

До сих мы рассматривали различные методы препроцессинга текстов независимо друг от друга. Естественно, что практический интерес требует их одновременного использования. Получим ли мы при этом увеличение сжатия равным сумме улучшений, обеспечиваемых каждым способом препроцессинга по отдельности? Ответ на этот вопрос дают табл. 7.8 и табл. 7.9, в которых приведены результаты сжатия текста «Three men in a boat (to say nothing of the dog)», преобразованного с помощью последовательного применения 4 техник:

- специального кодирования СКС;
- преобразования заглавных букв;
- модификации разделителей;
- словарной замены *n*-графов (словарь из табл. 7.1).

В табл. 7.9 ожидаемое улучшение сжатия вычислялось как сумма улучшений для всех четырех типов препроцессинга относительно размера архива исходного непреобразованного файла (см. табл. 7.3, 7.5, 7.6, 7.7).

Таблица 7.8

Архиватор	Тип метода сжатия	Размер архива исходного файла	Размер архива обработанного файла плюс размер описания длин строк	Улучшение сжатия	Улучшение сжатия без выполнения модификации разделителей
Bzip2, вер. 1.00	BWT	109736	103047	6.1%	6.0%
WinRAR, вер. 2.71	LZ77	130174	120632	7.3%	7.8%
HA a2, вер. 0.999c	PPM	108443	99788	8.0%	7.3%

Таблица 7.9

Архиватор	Тип метода сжатия	Улучшение сжатия	Ожидаемое улучшение сжатия
Bzip2, вер. 1.00	BWT	6.1%	6.8%
WinRAR, вер. 2.71	LZ77	7.3%	7.1%
HA a2, вер. 0.999c	PPM	8.0%	7.6%

Из таблиц видно, что для HA и, в меньшей степени, WinRAR, проявился даже положительный кумулятивный эффект от применения нескольких алгоритмов предварительной обработки. Это достаточно странный на первый взгляд результат, так как чем совершеннее алгоритм сжатия, тем меньший выигрыш должно давать использование дополнительных механизмов, что, собственно, мы и наблюдаем в случае архиватора BZIP2. До некоторой степени это можно объяснить тем, что после преобразования заглавных букв большее количество *n*-графов может быть заменено на соответствующие им индексы словаря, что уменьшает разнообразие используемых строк и способствует увеличению сжатия. Возможно, сочетание словарной замены со специальным кодированием СКС настолько уменьшает общее количество строк, сжимаемых с помощью словаря LZ77, при одновременном уменьшении их фиктивной длины, что это компенсирует падение общего процента улучшения сжатия. Вставка пробелов или замена СКС на пробелы уменьшает количество контекстов и, соответственно, уменьшает размер PPM-модели, поэтому HA, ограниченный всего лишь примерно 400 кбайт памяти, может использовать для оценки большее количество статистики, что улучшает сжатие. Судя по всему, реализация BWT и сопутствующих методов в BZIP2 и принципиальные особенности алгоритма блочной сортировки не позволили BWT «воспользоваться» ситуацией так же эффективно, как LZ77 и PPM.

Рассмотрим, что произойдет при использовании LIPT вместо словарной замены *n*-графов. В табл. 7.10 представлены результаты сжатия преобразованного текста, полученного с помощью трех упоминавшихся техник препроцессинга с последующим

использованием LIPT, алгоритм 2 (см. подпункт «Использование словарей»).

Таблица 7.10

Архиватор	Тип метода сжатия	Размер архива исходного файла	Размер архива обработанного файла плюс размер описания длин строк	Улучшение сжатия	Ожидаемое улучшение сжатия
Bzip2, вер. 1.00	BWT	109736	93882	14.4%	12.9%
WinRAR, вер. 2.71	LZ77	130174	114566	12.0%	10.1%
HA a2, вер. 0.999c	PPM	108443	94191	13.1%	14.9%

И опять мы видим, что различные способы препроцессинга дополняют друг друга, обеспечивая рост степени сжатия. Хотя теперь ситуация до некоторой степени поменялась: увеличение сжатия больше ожидаемого для BWT и LZ77, а в случае PPM наблюдается эффект «насыщения». Отметим, что использованная схема предварительной обработки далеко не самая лучшая, если ее предполагается использовать совместно с LZ-компрессором. В этом случае за счет упоминавшихся модификаций можно повысить степень сжатия еще на несколько процентов.

Вывод:

Одновременное применение рассмотренных способов предварительной обработки текстов позволяет улучшить сжатие на 5-8% в случае простой словарной схемы препроцессинга и на 12-15% при использовании громоздкого словаря.

Препроцессинг нетекстовых данных

Было замечено, что многие файлы содержат данные, записанные не в самом удобном виде для сжатия традиционными универсальными компрессорами. И, если изменить форму пред-

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

ставления этих данных, то эффективность сжатия файлов заметно увеличится. Многие компрессоры уже оснащены препроцессорами регулярных структур и исполнимых файлов. Среди них можно отметить 7-Zip, ACE, CABARC, DC, IMP, PPMN, SBC, UHARC, YBS.

Преобразование относительных адресов

Как известно, в системе команд процессоров Intel адреса меток в ряде случаев записываются в виде смещения от адреса текущей команды до адреса соответствующей метки. Так записываются команды CALL (код операции 0xE8), JMP (код операции 0xE9). В результате, если ряд команд ссылается на одну и ту же метку, каждый раз адрес этой метки записывается по-разному.

Главная идея преобразования заключается в замене относительных адресов на абсолютные. Обычно подпрограмма вызывается несколько раз из разных участков программы. Тогда несколько относительных адресов будут преобразованы в один абсолютный адрес, вследствие чего количество различных строк в файле уменьшится, а степень сжатия возрастет. Причем, не обязательно получать истинные абсолютные адреса меток. Лишь бы преобразование было обратимо.

Рассмотрим, например, преобразование адресов 32-разрядной команды CALL. Команда записывается в виде последовательности пяти байтов:

0xE8	R_0	R_1	R_2	R_3

Относительное смещение R вычисляется по формуле:

$$R = R_0 + (R_1<<8) + (R_2<<16) + (R_3<<24)$$

Зная смещение команды CALL от начала файла и относительное смещение R, можно вычислить абсолютный адрес. При этом следует учитывать, что не все символы с кодом 0xE8 являются началом команды CALL. Чтобы отличить настоящие

команды, предлагается довольно простой способ, примененный в архиваторе CABARC [3].

Введем следующие обозначения:

C	величина смещения анализируемой команды от начала файла (а именно, смещение байта кода операции – 0xE8 или 0xE9)
N	размер файла
R	смещение метки, на которую указывает операнд команды, относительно самой команды CALL
A	абсолютный адрес метки

Разделим все значения относительных смещений на 4 диапазона.

Первым делом определим диапазон значений смещений, которые имеет смысл подвергать преобразованию. Минимальное значение этого диапазона соответствует ссылке команды на начало файла, максимальное — на конец.

Смещения, принимающие значения меньше вышеуказанных, нецелесообразно подвергать преобразованию, поскольку очевидно, они не принадлежат командам. Это второй диапазон.

Прежде чем определять остальные два диапазона, рассмотрим процесс преобразования относительных значений из первого диапазона в абсолютные:

$$A = R + C$$

В результате преобразования получим величину, которая может принимать значения от нуля до N–1. Таким образом, в результате преобразования мы отображаем значения из отрезка [–C,N–C) на значения отрезка [0,N).

Для обеспечения возможности однозначного декодирования введем третий диапазон [0,N), над которым будем осуществлять компенсирующее преобразование, т.е. [0,N) → [–C, N–C).

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

Оставшиеся значения смещений поместим в четвертый диапазон. Эти значения в результате преобразования будут оставаться неизменными.

Преобразование относительных адресов можно представить в виде рис. 7.4. Преобразование относительных значений [–C,N–C) в абсолютные значения [0,N) показано толстой сплошной стрелкой, компенсирующее преобразование — толстой пунктирной.

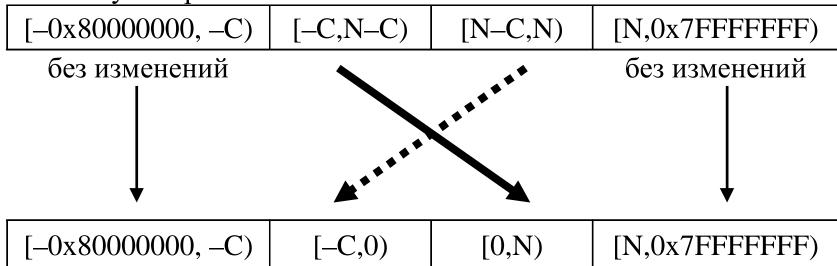

Рис. 7.4. Схема преобразования относительных значений в абсолютные

После преобразования заменим запись команды CALL следующей последовательностью:

0xE8	A_0	A_1	A_2	A_3

где
$A_0 = A \ \& \ 0xFF$
$A_1 = (A>>8) \ \& \ 0xFF$
$A_2 = (A>>16) \ \& \ 0xFF$
$A_3 = (A>>24) \ \& \ 0xFF$

Функция преобразования выглядит следующим образом:

```
// in – указатель на найденную команду
// op – адрес в буфере, где записан операнд
```

```
// cur_pos — номер позиции команды в файле
// file_size — размер файла

op = (long *)&in[1];
if( *op >= -cur_pos &&
    *op <  file_size — cur_pos ) {
    *op += cur_pos;
} else if( *op  >  0 &&
           *op  < file_size ) {
           *op -= file_size;
}
```

Аналогичное преобразование выполняется и для команды безусловного перехода, JMP. Правда, замена относительных значений адресов для этой команды не так эффективна. В частности, из-за того, что, как правило, число команд относительного перехода на один и тот же абсолютный адрес в среднем меньше, чем количество вызовов одной и той же подпрограммы. А также из-за того, что относительные адреса обычно принимают меньшие значения, чем в случае с командой CALL, т.е. могут быть эффективно сжаты и без какого-либо преобразования. Поэтому преобразование может даже ухудшить сжатие. Решение о применении данного преобразования можно принимать на основе оценки статистики его выполнения. Критерием целесообразности выполнения преобразования может выступать следующее условие: доля компенсирующих преобразований адресов незначительна, и в результате преобразования получается большое число совпадающих значений абсолютных адресов.

Сравним влияние описываемого преобразования команд CALL и JMP на сжатие данных компрессорами, представляющими различные методы. В качестве тестового файла был использован исполнимый модуль wcc386.exe из дистрибутива Watcom C 10.0.

Таблица 7.11

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

Архиватор	Тип метода сжатия	Размер архива	Замена операндов команды CALL		Замена операндов команд CALL и JMP	
			Размер архива	Улучшение сжатия	Размер архива	Улучшение сжатия
Bzip2, вер 1.00	BWT	308624	291492	5.55%	292051	5.37%
WinRAR, вер 2.70	LZ77	298959	281584	5.81%	280995	6.01%
HA a2, вер. 0.999c	PPM	296769	280316	5.54%	279959	5.66%

Следует отметить, что данное преобразование, хотя и является достаточно простым и эффективным, может быть усовершенствовано для достижения более сильного сжатия. Например, можно заметить, что код операции 0xE8 соседствует с младшим байтом значения абсолютного адреса A_0, в то время как более логичным было бы расположить рядом более связанные друг с другом 0xE8 и A_3. Т.е., можно записывать значение абсолютного адреса старшими байтами вперед. Другой способ повышения эффективности заключается в предварительном составлении списка всех процедур, на которые делаются ссылки в программе, и указании номеров процедур в списке вместо адресов.

Упражнение: Напишите процедуру обратного преобразования абсолютного значения адреса в относительное.

Преобразование табличных структур

В файлах зачастую можно встретить регулярные структуры, такие как, например, различного рода служебные таблицы, и

т.п. Очевидно, такие структуры требуют особого внимания. Причем, необязательно их выносить в отдельный файл и сжимать при помощи специализированного алгоритма. Иногда достаточно преобразовать их в такой вид, который будет приемлем и для универсального архиватора.

Например, рассмотрим преобразование таблиц, представляющих собой упорядоченный по возрастанию список 32-разрядных значений. Чем нам могут помешать эти структуры в первоначальном виде? Статистические характеристики таких табличных последовательностей обычно сильно отличаются от характеристик файла в целом. Например, в таблицах рядом находятся байты, имеющие разный вес в составе 32-разрядных значений и статистически слабосвязанные между собой. Таким образом, эти таблицы не только сами хранятся в неудобном для сжатия виде, но и ухудшают сжатие окружающих их данных за счет искажения вероятностных характеристик.

Первая задача, которая встает перед нами — распознать такие таблицы среди остальных данных. Учтем, что таблицы могут быть произвольного размера и располагаться в любом месте файла. Будем рассматривать какую-то область данных как таблицу при выполнении следующих условий:

1) Если нам в файле встретились подряд три 32-разрядных числа, идущие в неубывающем порядке, будем считать это началом таблицы. Например (первый байт самый младший):
   0x05 0x00 0x80 0x3f
   0x05 0x00 0x80 0x3f
   0x35 0x00 0x80 0x3f
2) Если эти числа могут быть закодированы при помощи RLE, отказываемся от применения преобразования таблиц.
3) Концом таблицы будем считать число, которое превышает предыдущее более чем на $2^8–2$, или меньше предыдущего.

Выполним преобразование таких таблиц следующим образом:

1) Первые 3 числа таблицы записываем в неизменном виде.

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

2) Для записи остальных чисел нам требуется только величина, на которую каждое число отличается от предыдущего. Для записи этой разницы достаточно одного байта.
3) В качестве признака конца таблицы записываем дополнительно символ с кодом 2^8-1 = 0xFF.

В качестве тестового файла возьмем тот же исполнимый модуль wcc386.exe из дистрибутива Watcom C 10.0.

Таблица 7.12

Архиватор	Тип метода сжатия	Размер архива	Преобразование 32-разрядных таблиц	
			Размер архива	Улучшение сжатия
Bzip2, вер 1.00	BWT	308624	306791	0.59%
WinRAR, вер 2.70	LZ77	298959	298342	0.21%
HA a2, вер. 0.999c	PPM	296769	295240	0.52%

Можно заметить, что для архиватора, использующего LZ77, выигрыш в сжатии оказался существенно меньше, чем для других методов. Это объясняется особенностью алгоритма кодирования указателей, используемого в WinRAR, позволяющего эффективно обрабатывать такого рода структуры.

Описанный алгоритм не является самым эффективным с точки зрения сжатия, но позволяет оценить полезные свойства такого рода преобразований. Способов улучшения довольно много. Например, большие таблицы следует кодировать отдельно от всего файла. Кроме того, помимо 32-разрядных таблиц, в файле могут присутствовать и, например, 16-разрядные; в таблицах могут находиться как возрастающие, так и убывающие последовательности чисел и т.п. И, надо сказать, зачастую суммарный эффект от применения такого рода преобразований оказывается довольно ощутимым.

Упражнение: Придумайте способ преобразования для 16-разрядных таблиц.

Вопросы для самоконтроля[10]

1. Почему невыгодно включение длинных фраз в словарь *n*-графов?
2. Каковы недостатки и преимущества динамического составления словаря *n*-графов?
3. Объясните, почему использование при словарной замене фраз, содержащих пробелы, приводит к уменьшению эффективности предварительной обработки в случае алгоритмов класса PPM и BWT.
4. Почему при организации LIPT для записи индекса фраз удобно использовать только те символы, которые входят в словарь букв?
5. В каких случаях при специальном кодировании символов конца строки выгоднее указывать длину строки не через количество символов, а через количество пробелов?
6. Почему при преобразовании относительного адреса подпрограмм, вызываемых командой CALL, обычно выгодно записывать значение абсолютного адреса старшими байтами вперед?
7. Почему преобразование относительных адресов для команды JUMP, как правило, менее эффективно, чем для CALL?

Литература

1. Awan F., Motgi N.Zh.N., Iqbal R., Mukherjee A. LIPT: A Reversible Lossless Text Transform to Improve Compression Per-

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

formance // Proceedings of Data Compression Conference, Snowbird, Utah, March 2001.
2. Grabowski Sz. Text preprocessing for Burrows-Wheeler block sorting compression // VII Konferencja "Sieci i Systemy Informatyczne" (7th Conference "Networks and IT Systems"), Lodz, Oct. 1999, conf. proc., pp. 229-239. http://www.compression.ru/download/articles/text/grabowski_1999_preproc_rtf.rar
3. Microsoft Corporation. Microsoft LZX Data Compression Format. 1997.
4. Teahan W.J. Modelling English Texts // PhD thesis, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, May 1998.

Выбор метода сжатия

В заключение раздела скажем несколько слов о процедуре выбора метода сжатия.

Выбор метода — это первая задача, которую должен решить разработчик программных средств сжатия данных. Выбор зависит от типа данных, которые предстоит обрабатывать, аппаратных ресурсов, требований к степени сжатия и ограничений на время работы программы. Дадим ряд рекомендаций, предполагая, что необходимо сжимать качественные данные, между элементами которых имеются сильные и достаточно протяженные статистические связи, т.е. данные порождены источником с памятью.

Прежде всего, следует учитывать, что каждый из рассмотренных в разделе универсальных методов сжатия данных источников с памятью допускает модификации, позволяющие существенно изменить параметры компрессора. Так, увеличение порядка модели PPM приводит к заметному усилению сжатия ценой замедления работы и увеличения расходов памяти. К аналогичному результату приводит увеличение размера словаря

LZ77-методов, но при этом время разжатия остается практически неизменным. Свойства компрессоров на основе BWT варьируются в меньшей степени.

Ограничения на время работы программы возникают в зависимости от условий, в которых предстоит работать архиватору. Например, при сжатии данных для формирования дистрибутива программного обеспечения ограничение на время компрессии практически отсутствует, в то время как требуется максимально уменьшить время восстановления исходных данных. При резервном сохранении данных положение вещей обратно, поскольку ситуация, когда требуется выполнить разжатие, довольно редка. Необходимость оперативной передачи данных по сети обуславливает одинаковые требования к временам сжатия и разжатия; если данные передаются нескольким абонентам, некоторое преимущество имеют архиваторы, обладающие более быстрым алгоритмом разжатия.

Методы семейства LZ77 обладают наибольшей скоростью декомпрессии. Превышение над скоростью сжатия при использовании метода Хаффмана для кодирования результатов работы LZ77-метода — десятикратное. Меньшая разница у методов на основе BWT — в среднем скорость разжатия в 2-4 раза выше скорости сжатия. Декодирование при использовании PPM на 5-10% медленнее кодирования. Компрессоры на базе частичных сортирующих преобразований малого порядка характеризуются еще большим отставанием разжатия — на некоторых файлах оно в несколько раз медленнее сжатия.

Похожая картина наблюдается, если сравнивать использование памяти при декодировании. В случае применения LZ77 расходы памяти минимальны. Архиваторы на основе PPM наиболее требовательны — им необходимо столько же памяти, сколько и при кодировании. Следует отметить, что при сжатии требования к памяти у программ-представителей разных методов примерно близки, хотя и могут изменяться для разных типов сжимаемых данных.

Таким образом, если можно пренебречь степенью сжатия, методы семейства LZ77 наиболее эффективны для создания дистрибутивов, а методы на основе частичных сортирующих преобразований — для резервного копирования.

Типичные данные качественного характера можно условно разделить на 4 типа:

1) однородные данные (например, типичные тексты);

2) однородные данные с большой избыточностью в виде длинных повторяющихся строк (например, набор исходных текстов);

3) неоднородные данные, в которых имеется выраженная нестабильность контекстно-зависимой статистики (например, исполнимые файлы);

4) данные с малой избыточностью (например, файлы, содержащие уже сжатые блоки);

Рассмотрим, как ведут себя различные методы при сжатии данных разных типов. При анализе будем ориентироваться на наилучших представителей методов.

Для сжатия таких однородных данных, как тексты на естественных языках, наиболее подходящими являются PPM-методы и методы на основе BWT. Первые позволяют достичь большей степени сжатия, вторые обладают большей скоростью декодирования. Программы, использующие методы семейства LZ77, сжимают указанные данные заметно хуже и, при увеличении длины словаря, существенно медленнее.

Если в однородных данных есть длинные повторяющиеся строки, у программ на основе LZ77 есть шанс себя реабилитировать. Впрочем, в этом случае наиболее выгодным будет использовать гибрид LZ77 и любого из двух его конкурентов или применять LZ77-препроцессинг.

При сжатии неоднородных данных пальма первенства принадлежит семейству методов LZ77, которые при сохранении своих высоких скоростных качеств настигают по степени сжатия заметно более медленных лучших представителей PPM-компрессоров. Если не использовать фрагментирование, BWT-

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

компрессоры показывают не очень высокую степень сжатия неоднородных данных.

На данных с малой избыточностью все методы выступают не лучшим образом. Некоторое преимущество в степени сжатия имеют архиваторы на основе LZ77 и PPM. Но последние при этом требуют значительного расхода памяти и скорость их работы заметно падает.

В заключение приведем таблицу, в которой описаны свойства методов при сжатии качественных данных различных типов.

Параметры	Метод	Однородные данные (типичный текст)	Однородные данные с большой избыточностью (исходные тексты программ)	Неоднородные данные	Данные с малой избыточностью
Степень сжатия	PPM	высокая	высокая	высокая	невысокая
	BWT	близкая к PPM	близкая к PPM	без фрагментирования — худшая	невысокая
	LZ77	заметно худшая	при большом количестве длинных повторов довольно высокая	близкая к PPM	невысокая
Скорость кодирования	BWT	высокая	средняя	высокая	высокая
	PPM	при большом порядке модели — самая низкая, при небольшом — немного быстрее BWT	если не использовать сложное моделирование — высокая	средняя	низкая
	LZ77	средняя, а при малом словаре — самая высокая	средняя, при малом словаре — высокая	высокая	высокая
Скорость декодирования	LZ77	примерно в 10 раз выше скорости кодирования; разница еще больше на избыточных данных			
	PPM	обычно на 5-10% медленнее кодирования			
	BWT	в 2-4 раза выше скорости кодирования			
Требуемый объем памяти при сжатии	BWT	постоянный при сжатии данных любого типа			

	PPM	варьируется в широких пределах, в зависимости от сложности моделирования и порядка модели; вырастает на малоизбыточных и очень неоднородных данных[11]
	LZ77	пропорционален размеру словаря
Требуемый объем памяти при разжатии	LZ77	минимальный
	PPM	максимальный, если процесс моделирования симметричен, то примерно равен расходу памяти при сжатии
	BWT	средний

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://www.compression.ru/

[11] Утверждение об увеличении расхода памяти в случае малоизбыточных данных спорно. Это зависит от используемых в компрессоре структур данных. В том случае, если в модели не хранится в явном виде информация о так называемых "виртуальных" состояниях, возникших только один раз, то ситуация может быть и обратной, особенно при большом порядке модели. В итоге для кодирования уже сжатого файла (степень сжатия около 1) может требоваться меньший объем памяти, чем для кодирования текстового файла (степень сжатия 3-4) такого же размера.

Указатель терминов

7

7-Zip, 108, 308

A

Abrahamson, 188
ACE, 110, 308
ADSM, 188
ARHANGEL, 186, 288
ARJ, 110
ARJZ, 110

B

Bell, 61, 76
Bender, 76, 102
Bentley-Sedgewick, 241
Blending, 125
Bloom, 76, 186
BMF, 189
Boa, 186
Brent, 76
Burrows-Wheeler Transform, 199
BWT, 199, 264, 265, 272

C

CABARC, 100, 108, 109, 110, 308
CELP, 39
CM, 178
Context tree weighting, 193
cPPMII, 183
CS-ACELP, 39
CTW, 193

D

DAFC, 187, 190
DC, 288, 308
Deflate, 82
Delta Coding, 29
Deterministic scaling, 168
DHPC, 190
Distance Coding, 226
DMC, 193
Dynamic Markov compression, 193

E

Escape, 129, 131
Exclusion, 132

F

Fenwick Peter, 225
Fiala, 76, 104
Finite-context modeling, 123
Full updates, 158

G

Greedy parsing, 97
Greene, 76, 104

H

HA, 177
Herklotz, 186
Hirvola, 177
Hoang, 77

I

Imp, 110
IMP, 308
Info-ZIP, 94, 110
Inversed Frequencies, 229

J

JAR, 110, 288
Jung, 110

K

Katz, 82

L

Langdon, 115, 187
Lazy matching, 95
Lemke, 110
Lempel, 60
Length Index Preserving
  Transformation. *См.* LIPT
LFF, 98
LGHA, 186
Linear Prediction Coding. *См.* LPC
LIPT, 294
Local Order Estimation. *См.* LOE
LOE, 161, 179
LOEMA, 186
Long, 77
Longest Fragment First. *См.* LFF
Lookahead buffer, 62
LPC, 28
  analysis, 39
  synthesis, 39
  оптимальная модель, 35
  синтез моделей, 40
LRU, 107
Lyapko, 186
LZ77, 60, 61
  Пример, 63
LZ77-PM, 77, 106
LZ78, 60, 72
  Пример, 72
LZB, 76
LZBW, 76, 102
  Пример, 102
LZCB, 76
LZFG, 76, 104
LZFG-PM, 77, 106
LZH, 76
LZMV, 76
LZP, 76
LZRW1, 76
LZSS, 67
  Пример, 68
LZW, 76, 83
LZW-PM, 77, 106
LZX, 109

M

Match length, 62
MELP, 39
Microsoft, 110
Miller, 76
Move To Front. *См.* MTF
MTF, 213

N

n-граф, 289

O

Offset, 62

P

Parallel Blocks Sorting. *См.* PBS
Pavlov, 110
PBS, 263

PKWARE, 110
PKZIP, 82, 110
PPM, 80, 130
  пример, 133
PPM*, 159, 173
PPMA, 144, 190
PPMB, 144, 190
PPMC, 145
PPMd, 149, 183
PPMD, 145
PPMII, 183
PPMN, 166, 180, 288, 308
PPMonstr, 149, 183
PPMY, 183
PPMZ, 147, 186
Prediction by Partial Matching. *См.* PPM

R

Radix sorting, 242
RAR, 108, 110
Recency scaling, 166
Reordering, 238
Rissanen, 115, 187
RK, 179, 288
RKUC, 179
RLE, 215
Roberts, 186
Roshal, 110

S

Sadakane Kunihiko, 243
SBC, 288, 308
Schindler Mikael, 226
Schindler Transform, 209
Secondary
  Escape Estimation. *См.* SEE
  Symbol Estimation. *См.* SSE
SEE, 146
SEM, 43
Sepulizing. *См.* Сепулирование
SEQUITUR, 193
Shelwien, 183
Shkarin, 183
Smirnov, 180
Sort Transformation, 209
SSE, 170
ST, 264, 265, 272
Storer, 67
Subband Coding, 44
Suffix sorting, 243
Sutton, 186
Szymanski, 67

T

Taylor, 179
Technelysium, 110

U

UHARC, 186, 288, 308
Unisys, 83
Universal modelling and coding, 115
Update exclusion, 158
  partial, 159

V

Valentini, 186
Vitter, 77

W

Wegman, 76
Welch, 76
Williams, 76, 190
WinRAR. *См.* RAR
WinZip, 110
Wolf, 76, 102
WORD, 191

X

X1, 186

Y

YBS, 308

Z

Ziganshin, 110, 178
Zip, 94, 110
Ziv, 60

А

Абрахамсон, 188
Алгоритм
  Lossless JPEG, 42
  PNG, 41, 42
  Хаффмана, 232
Атрибут, 263

Б

База
  длины совпадения, 86
  смещения, 87
Барроуз Майк, 198
Барроуза-Уилера преобразование. *См.* преобразование
Белл, 61, 76
Бендер, 76, 102
Биграф. *См.* *n*-граф
Блум, 76, 186
Брент, 76
Буфер упреждающий, 62
Буферизация смещений, 106

В

Валентини, 186
Вектор обратного преобразования, 204
Вероятность ухода, 131, 144
Взвешивание, 126
  неявное, 129
Виттер, 77
Вулф, 76, 102
Вэйвлет-фильтр, 56

Г

Границы диапазона допустимых значений, 43
Грини, 76, 104

Д

Дельта-кодирование, 29, 51
  пример, 30
Джанг, 110
Динамическое марковское сжатие, 193
Дискретное вэйвлетное преобразование, 53
Длина совпадения, 62
Длина соответствия. *См.* длина совпадения

Ж

Жадный разбор, 97
Жимански, 67

З

Зив, 60
Зиганшин, 110, 178

И

Исключение, 132
Исключение при обновлении, 158

частичное, 159, 169
Источник
данных, 115
Маркова, 131

К

Кац, 82
Кодер, 115
Кодирование, 115
1-2, 221
длин повторов. *См.* RLE
линейно-предсказывающее. *См.* LPC
расстояний. *См.* Distance Coding
статистическое, 116
субполосное. *См.* Subband Coding
Кодировщик, 115
Коды
средняя длина, 25
Хаффмана, 84
Элиаса, 77
Компрессор, 115
Dummy, 144
PPM, 138
Контейнер, 265
Контекст, 42, 122
активный, 123
детерминированный, 159, 168
дочерний, 124
левосторонний, 122
правосторонний, 122
-предок, 124
разбросанный, 180
родительский, 124
ухода, 147
Контекстная модель, 123
детерминированная, 150, 159
с замаскированными символами, 150
с незамаскированными символами, 150
уходов, 147
Контекстное моделирование, 98, 105, 120
ограниченного порядка, 122
с полным смешиванием, 126
с частичным смешиванием, 126
чистое порядка *N*, 125, 188
Контур, 286

Л

Лемке, 110
Лемпел, 60
Ленивое сравнение, 95
Линейная комбинация, 28
Линейно-предсказывающее кодирование. *См.* LPC
Литерал, 63
Лонг, 77
Лэнгдон, 115, 187
Ляпко, 186

М

Матрица циклических перестановок, 201
Методы Зива-Лемпела, 60
Механизм уходов, 131
Миллер, 76
Моделирование, 115
адаптивное, 119
блочно-адаптивное, 119, 178
контекстное ограниченного порядка, 122
полуадаптивное, 119
статическое, 118
Моделировщик, 115
Модель
"аналоговый сигнал", 45
иерархическая, 233
источника данных, 115, 124

структурная, 233
шумовая, 34
эволюционная, 34

Н

Накопленная частота, 136
Наследование информации, 164
отложенное, 165

О

Обработка данных
предварительная. *См.* предобработка
Обратные частоты, 229
Обход плоскости, 39
ОВУ, 144
Оптимальный разбор, 97, 99
Оценка вероятности ухода, 144
адаптивные методы. *См.* SEE
априорные методы, 144
метод A, 145, 188
метод B, 145
метод C, 145
метод D, 145
метод P, 145
метод SEE-d1, 149
метод SEE-d2, 149
метод X, 145
метод XC, 145
метод Z, 147, 180
Ошибка предсказания, 28
минимизация, 35

П

Павлов, 110
Параллельные блоки, 263
Перемещение стопки книг. *См.* MTF
Перестановка, 274
Переупорядочивание символов, 238
Поиск границ, 50, 276
Полное обновление счетчиков, 158
Порядок модели PPM, 131, 175
Постпроцессор, 287
Предобработка, 287
Предсказание
наиболее вероятных символов, 169
по частичному совпадению. *См.* PPM
Преобразование
Барроуза-Уилера, 199
относительных адресов, 308
сортирующее частичное, 209
табличных структур, 312
Шиндлера, 209
Препроцессинг. *См.* предобработка
Препроцессор, 287
Производная блока, 273

Р

Разложение на полусуммы и разности, 45
Размер блока в BWT, 236
Риссанен, 115, 187
Робертс, 186
Рошал, 110

С

Садакане Кунихико, 243
Саттон, 186
Сепулирование. *См.* Sepulizing
Сжатие параллельных потоков, 50
Символ ухода, 129
Символы конца строки, 299
Скользящее окно, 62, 277
СКС. *См.* символы конца строки
Словарь, 57, 289

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

классификация, 289
контекстно-зависимый, 105
скользящий, 62
Смешивание, 125
Смещение, 62
Смирнов, 180
Сортировка
Бентли-Седжвика, 241
быстрая, 241
используемая в BWT, 240
направление, 239
параллельных блоков. *См.* PBS
поразрядная, 242
суффиксов, 243
Сравнение
алгоритмов LZ, 80
алгоритмов контекстного моделирования, 192
архиваторов LZ, 110
архиваторов PPM, 185
Сторер, 67
Субполосное кодирование. *См.* Subband Coding

Т

Тейлор, 179
Теорема о кодировании источника, 25, 116
Тетраграф. *См.* *n*-граф
Триграф. *См.* *n*-граф

У

Уилер Дэвид, 198
Уилльямс, 76, 190
Указатель, 62
Уменьшение шума, 38
Уэгнам, 76
Уэлч, 76

Ф

Файэлэ, 76, 104
Фенвик Петер, 225
Фильтр
Paeth, 41
выбор, 42, 56
Формат
Deflate, 82
Фрагментирование, 276
Фраза словаря, 57, 289
Фрейм, 268
Функция
адаптивная, 273
отличия, 277

Х

Характеристики
алгоритмов PPM, 176
алгоритмов семейства LZ77, 81
алгоритмов семейства LZ78, 82
Херклоц, 186
Хеш-функция, 94
Хирвола, 177
Хоанг, 77

Ч

Частота элемента, 266, 272, 277

Ш

Шелвин, 183
Шеннон, 25, 116
Шиндлер Микаэль, 226
Шкарин, 149, 164, 183

Э

Энтропия, 25