Вадим Юкин. Преобразование Барроуза-Уилера или
алгоритм сжатия без потерь при помощи сортировки блока данных.

1. Введение.

Алгоритм сжатия данных на основе преобразования Барроуза-Уилера (Burrows-Wheeler Transform, далее BWT) впервые был описан сравнительно недавно - в 1994 году. Он был опубликован 10 мая в статье "A Block-sorting Lossless Data Compression Algorithm" [1]. Хотя утверждается, что один из его авторов, Майкл Уилер, придумал его гораздо раньше, в 1983 году, но тогда не придал ему надлежащего значения.

Сейчас этот метод стремительно обретает популярность среди исследователей в области сжатия данных, появляется все больше и больше научных статей, ему посвященных. Не обделяют его вниманием и программисты, разрабатывающие новые архиваторы. Этот метод привлекателен своей простотой и элегантностью, которые, я надеюсь, оценит также и читатель.

Алгоритм, описанный в оригинальной статье, представлял собой совокупность трех методов:
 - метод сортировки блока данных (собственно который и называется преобразованием Барроуза-Уилера),
 - MoveToFront-преобразование (известное также, как метод перемещения стопки книг),
 - простой статистический кодер для сжатия преобразованных на первых двух этапах данных.

Эти методы, а также возможные варианты алгоритмов, их заменяющих, и рассматриваются в данной статье.

2. Описание алгоритма.

2.1. Преобразование Барроуза-Уилера.

Это преобразование, конечно, является основой описываемого алгоритма сжатия. Вкратце, его можно описать как способ перестановки символов в блоке данных, позволяющий осуществить эффективное сжатие. Процедуру преобразования можно условно разделить на четыре этапа:

1) выделяется блок из входного потока,
2) формируется матрица всех перестановок, полученных в результате

   циклического сдвига блока,
3) все перестановки сортируются в соответствии с лексикографическим
   порядком символов каждой перестановки,
4) на выход подается последний столбец матрицы и номер строки,
   соответствующей оригинальному блоку.

На всякий случай следует принять специальные меры, способные предотвратить возможное зацикливание при попытке сравнить две одинаковые строки. Например, при переходе через границу блока прерывать сравнение и принимать однозначное решение в пользу одной из строк.

Итак, получим матрицу перестановок.

```
      абракадабра
      бракадабраа
      ракадабрааб
      акадабраабр
      кадабраабра
      адабраабрак
      дабраабрака
      абраабракад
      браабракада
      раабракадаб
      аабракадабр
```

Затем отсортируем полученные строки, предварительно пометив исходную.

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

Таким образом, в результате преобразования, взяв последний столбец, мы получили строку "рдакрааааабб" и номер исходной строки в отсортированной матрице, равный двум.

Следует отметить основополагающее свойство преобразования. Поскольку осуществлялись именно циклические сдвиги, символы последнего столбца предшествуют начальным символам строк, которые в наибольшей степени участвовали в сортировке. Таким образом, если в исходном файле есть две похожие строки, то символы, предшествующие им обоим, будут находиться поблизости друг от друга в блоке, полученном в результате преобразования. И чем больше эти строки похожи, тем больше вероятность того, что эти символы будут находиться рядом.
Здесь и далее строки, следующие во входном блоке за символами из блока выходного, будут называться контекстами.

2.2. Использование BWT в сжатии данных.

Теперь можно приступить к рассмотрению вопроса, почему описываемое преобразование может помочь нам в сжатии. Начну, как и в предыдущем пункте, с общей краткой формулировки: потому что BWT преобразует

поток данных, обладающий сложными статистическими свойствами высокого порядка, в поток со статистикой гораздо меньшего порядка. Что очень удобно для последующего сжатия.

И действительно, если у нас в тексте часто встречается какое-то слово, например, как слово "архиватор" в данном, то в матрице перестановок нам будут часто встречаться строки, начинающиеся с "рхиватор". Легко догадаться, что практически во всех этих строках в конце будет находиться символ "а". И разве что допущенные мною орфографические ошибки могут этому помешать (конечно, я постараюсь этого не допустить и, соответственно сделать так, чтобы статья сжалась получше). Таким образом, чем больше в файле похожих строк, тем больше похожих символов мы получим рядом в результате преобразования и тем легче нам будет при сжатии.

Затем можно воспользоваться еще одним преобразованием, MoveToFront (здесь и далее MTF), которое заключается в том, что все символы алфавита записаны в упорядоченный список и при обработке очередного символа на выход пишется номер этого символа в списке. После чего список модифицируется - затребованный символ перемещается в начало списка, сдвигая остальные в конец (ниже этот алгоритм и его свойства будут рассмотрены подробнее). В результате работы MTF вместо длинных последовательностей одинаковых символов мы получим длинные последовательности нулей, означающих выпадение символа, находящегося в начале списка.

По результатам тестов на стандартном тестовом наборе, используемом для сравнения свойств алгоритмов сжатия, Calgary Corpus, количество нулей на paper1 (статья на английском языке) составило 58.4%, на progp (программа) - 74%, geo (двоичный файл) - 35.8%.

2.3. Обратное преобразование.

После того, как прояснилась польза преобразования Берроуза-Уилера, самое время показать, что это не очередной фокус, а вполне работающий алгоритм, способный не только безвозвратно испортить накопленные непосильным трудом данные, но и при случае восстановить их в исходном виде.

Сначала продемонстрируем в наглядном виде восстановление матрицы перестановок. Для этого отсортируем единственное, что у нас есть - символы последнего столбца. И благодаря этому получим первый столбец. Ведь строки матрицы были отсортированы именно начиная с первого столбца - значит, упорядоченные символы - это и есть первый столбец.

```
  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
```

Можно заметить, что символы последнего столбца с символами первого образуют пары - ведь строки получены в результате циклического сдвига.

И отсортировав эти пары мы получим уже два известных столбца в левой части матрицы. И так далее, пока не восстановим всю матрицу целиком.

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

И, зная номер исходной строки, без труда находим ее в матрице перестановок.

После того, как удалось наглядно показать принципиальную возможность обратного преобразования, пришло время признаться, что на самом деле нет необходимости воспроизводить посимвольно все строки матрицы по одному символу. Обратите внимание, что при каждом проявлении доселе неизвестного столбца выполнялись одни и те же действия. А именно, из строки, начинающейся с некоторого символа последнего столбца получалась строка, в которой этот символ находится на первой позиции. Из строки 0 получается строка 9, из 1 - 7 и т.п.:

```
     0   а.........р   9
     1   а.........д   7
 ==> 2   а.........а   0
     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
```

Для получения вектора обратного преобразования, определим порядок получения символов первого столбца из символов последнего:

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

Полученные значения { 2, 5, 6, 7, 8, 9, 10, 1, 3, 0, 4 }
и есть искомый вектор, содержащий номера позиций символов, упорядоченных в соответствии с положением в алфавите, в строке, которую нам надо декодировать.

Для получения исходной строки надо всего-навсего выписать символы из

строки преобразованной ("рдакрааааабб") в порядке, соответствующему данному вектору, начиная с номера, переданного нам вместе со строкой.

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

Проделаем тоже самое при помощи простейшей программы.
Пусть, N - количество символов в блоке из входного потока
       n - количество символов в алфавите
       x - номер исходной строки в матрице перестановок
       in[N]   - входной поток
       count[n] - частоты каждого символа алфавита во входном потоке
       T[N]   - вектор обратного преобразования.

На первом шаге считаем частоты символов.

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

Затем упорядочиваем символы, чтобы получить первый столбец исходной матрицы.

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

Теперь count[i] указывает на первую позицию символа i в первом столбце. Следующий шаг - создание вектора обратного преобразования.

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

И, наконец, восстановим исходный текст.

```
  for( i=0,j=x; i<N; i++) {
    putc( in[j], output );
    j = T[j];
  }
```

2.4. Преобразование Шиндлера.

Следует отметить еще одно преобразование, основанное на сортировке блока данных - преобразование Шиндлера (ST) [4]. Отличие от BWT заключается в том, что строки матрицы перестановок упорядочиваются не по всей длине, а только по указанному количеству первых символов. Число таких символов называется порядком преобразования Шиндлера. В случае, если эти символы одинаковы у двух или более строк, они упорядочиваются в соответствии с номером позиции, в которой эти строки встречаются в исходной последовательности.

Справедливости ради надо отметить, что ST является не разновидностью преобразования Барроуза-Уилера, а скорее его обобщением. Можно сказать, что BWT - это преобразование Шиндлера порядка, равного размеру блока.

3. Алгоритмы, используемые совместно с BWT.

3.1. Move To Front.

Как было сказано выше, это тоже преобразование. Его алгоритм легко понять, если представить стопку книг, каждая из которых соответствует определенному символу. По мере востребования из стопки вытаскивается нужная книга и кладется сверху. Через некоторое время те книги, которые используются часто, оказываются ближе к верхушке стопки.

Вернемся к нашему примеру, а именно, к полученной в результате работы BWT строке "рдакрааааабб". Символы этой строки принадлежат алфавиту, содержащему пять элементов. Предположим для простоты, что других символов мы не используем, а начальный список MTF содержит эти символы в следующем порядке: { 'а', 'б', 'д', 'к', 'р' }.

Приступим к преобразованию. Символ 'р' является пятым элементом списка, поэтому первым выходным кодом становится код 2. После перемещения символа 'р' в начало списка, тот принимает вид {'р', 'а', 'б', 'д', 'к'}. И т.п.

Символ	Список	Выход
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

Назначение этого метода - упростить задачу статистическому кодеру. Поскольку сочетания символов в данных, которые нам приходится сжимать, разные, то и символы на выходе BWT, соответствующие контекстам этих сочетаний, тоже отличаются. Поэтому зачастую нам приходится обрабатывать последовательности вида "bbbbbccccccddddddaaaaa", в которых преобладание одного символа непредсказуемо сменяется преобладанием другого.

Попробуем сжать эту последовательность при помощи, например, метода Хаффмана. 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*2 = 40 бит.

Теперь проделаем тоже самое со строкой, подвергнутой MTF-преобразованию (предположим, что начальный список выглядит как 'a', 'b', 'c', 'd').

bbbbbccccccddddddaaaaa - исходная строка
10000200003000030000 - строка после MTF

Символ	Частота	Вероятность	Код Хаффмана
0	16	4/5	0
1	2	1/10	10
2	1	1/20	110
3	1	1/20	111

В результате сжатия получаем последовательность длиной 16*1 + 2*2 + 3*2 = 26 бит.

3.2. Distance coding (кодирование расстояний).

Стоит упоминания метод, который все чаще используется как альтернатива MTF. И, надо сказать, с успехом. По крайней мере, на большинстве файлов архиваторы, его использующие, имеют преимущество по сравнению с работающими в традиционной манере. Пока описание этого метода нигде официально не опубликовано и используют его всего три архиватора - DC, YBS и SBC.

Как всегда, рассмотрим кодирование строки "рдакрааааабб". Первый шаг заключается в том, чтобы определить первое вхождение каждого символа. Для этого к началу строки мы по очереди приписываем все символы алфавита (при декодировании мы можем проделать тоже самое). А также воспользуемся символом конца строки, означающим окончание кодирования определенного символа. Обозначим его как '%'. Получается "абдкррдакрааааабб%".

Находим число - расстояние от первого символа этой последовательности, 'а' до следующего такого же. Оно равно 6 - это число других символов между ними. Но нам заранее известно, что после 'а' идут символы 'б','д','к' и 'р'. А поскольку наша задача выяснить номер позиции, в которой потенциально может оказаться символ 'а' при декодировании, эти символы мы можем не считать. Таким образом, получаем число 2.

```
строка:             абдкррдакрааааабб%
известные символы:  абдкр..а........%
расстояние:         2
```

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

```
строка:             абдкррдакрааааабб%
известные символы:  абдкр..а......б.%
расстояние:         28
известные символы:  абдкр.да......б.%
расстояние:         281
известные символы:  абдкр.дак.....б.%
расстояние:         2811
известные символы:  абдкррдак.....б.%
расстояние:         2811-
```

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.

```
строка:             абдкррдакрааааабб%
известные символы:  абдкррдакр....б.%
расстояние:         2811-0
известные символы:  абдкррдакр....б.%
расстояние:         2811-05
```

Кодируя символ 'д' мы сделали ссылку на конец строки. При декодировании по ней мы можем понять, что символы 'д' закончились.

```
строка:             абдкррдакрааааабб%
известные символы:  абдкррдакра...б.%
```

```
расстояние:          2811-050
известные символы:   абдкррдакра...б.%
расстояние:          2811-0504
известные символы:   абдкррдакра...б.%
расстояние:          2811-05044
известные символы:   абдкррдакрааааб.%
расстояние:          2811-05044---
известные символы:   абдкррдакрааааб.%
расстояние:          2811-05044---1
известные символы:   абдкррдакрааааабб%
расстояние:          2811-05044---1-
```

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

3.3. Run Length Encoding.

Как видно из названия, суть этого кодирования - в замене части последовательности одинаковых символов на число, показывающее количество замененных символов.

Например, мы можем включать RLE в случае, когда количество одинаковых символов превышает 4. Тогда строка "abbbbbbbbccddddd" будет записана в виде "abbbb4ccdddd1".

RLE может пригодиться тогда, когда в данных есть длинные повторы одинаковых символов. Такие повторы могут возникнуть в двух случаях - в исходных данных и после преобразования Барроуза-Уилера.
В первом случае, особенно если повторов особенно много, RLE нам может заметно помочь ускорить сортировку матрицы перестановок ценой небольшого ухудшения сжатия. Аналогично, тот же эффект может иметь применение RLE и во втором случае. Правда, как было показано в предыдущем параграфе, использование DC делает RLE ненужным.

3.4. 1-2 coding.

Это один из широко используемых способов кодирования длинных повторов. Обычно он применяется после MTF. Этот метод нашел свое применение в Bzip2, Imp.

Как известно, после MTF-преобразования мы получаем последовательность, в которой при удачном раскладе присутствует большое количество нулей, соответствующих нулевому рангу MTF (см 3.1).

Если кодировать каждый код MTF-0, то во-первых, это накладно по времени и, во-вторых, может заметно ухудшить сжатие из-за погрешностей статистического кодера.

Элегантный выход был найден в отведении под MTF-0 не одного, а двух символов (назовем их z1 и z2). Таким образом, у нас в MTF-алфавите получилось не 256, а 257 символов. Эти z1 и z2 можно использовать для кодирования числа идущих подряд MTF-0:

```
1 - z1                7 - z1z1z1
2 - z2                8 - z1z1z2
3 - z1z1              9 - z1z2z1
4 - z1z2             10 - z1z2z2
5 - z2z1             11 - z2z1z1
6 - z2z2                 ...
```

Как вы помните, наша строка "абракадабра" после двух преобразований

выглядела как {4,3,2,4,3,2,0,0,0,4,0}. Подвергнем ее еще одному - и получим {4,3,2,4,3,2,z1,z1,4,z1}.

3.5. Структурная модель.

Было замечено, что данные, полученные после преобразования Барроуза-Уилера, состоят из так называемых "хороших фрагментов" (которые соответствуют часто встречаемым сочетаниям символов), разделенных "плохими фрагментами". Одна из главных задач при сжатии заключается в своевременном обнаружении границ фрагментов. То есть, в быстрой адаптации к смене фрагмента одного типа другим.

Еще одно наблюдение связано со статистикой MTF-рангов, полученных на типичных данных. Как уже упоминалось, для большинства текстов количество нулей превышает половину всех значений. Количество остальных рангов в среднем убывает по мере увеличения ранга и, соответственно, увеличивается погрешность при появлении конкретного значения и сглаживается разница между соседними рангами. Кроме того, появление нескольких больших рангов подряд в файле, подвергнутом BWT и MTF, может служить признаком "плохого фрагмента".

Одна из довольно эффективных моделей, призванных воспользоваться этой информацией, была описана Петером Фенвиком [2]. Все MTF-ранги были поделены на группы:

MTF-ранги	Номер группы
0	0
1	1
2-3	2
4-7	3
8-15	4
16-31	5
32-63	6
64-255	7

При обработке очередного ранга первым делом кодируется номер группы, а затем, в случае необходимости - номер ранга внутри группы.

Описанная модель получила довольно широкое распространение и используется в ряде современных архиваторах.

3.6. Переупорядочивание алфавита.

В отличие от многих других методов, сжатие, основанное на преобразовании, которому посвящена данная статья, заметно зависит от лексикографического порядка следования символов. Замечено, что символы, имеющие сходное использование в словообразованиях, лучше располагать поблизости.

Возьмем такие часто встречающихся 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 рядом, мы получим в последнем столбце рядом стоящие одинаковые символы.

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

препинания.

4. Характеристики метода.

4.1. Сравнение алгоритмов сжатия на базе BWT с другими методами.

Эта глава заинтересует как потенциальных разработчиков алгоритмов сжатия на основе BWT, так и возможных пользователей, раздумывающих, какому архиватору отдать предпочтение для хранения своих данных. Безусловно, следует сравнить архиваторы как по силе сжатия, так и по скорости работы и по расходам памяти.

Сравнение будет производиться с архиваторами, использующими LZ77 и метод Хаффман (сокращенно LZHuf). Эти архиваторы получили наибольшее применение еще со времен, когда компьютеры не имели мощности и объема памяти, достаточных для того, чтобы пользователям компьютеров были привлекательны более изощренные методы. В качестве представителей можно назвать PKZIP, WinZIP, 7ZIP, RAR, ACE, CABARC, ARJ, JAR.

Также стоит сравнить с архиваторами, использующими метод PPM (Prediction by Partial Matching). Среди них - HA, RK (а также RKUC, RKIVE), BOA, PPMD.

Скорость сжатия - на уровне архиваторов, применяющих LZHuf Хаффмана, а на больших словарях (от 1Mb) - и быстрее. У сжимателя, использующего преобразование Шиндлера, SZIP, скорость сжатия для малых порядков еще выше.

Скорость разжатия у архиваторов, использующих BWT, в 3-4 раза быстрее сжатия. В целом, классические LZHuf разжимают, конечно, быстрее.

Степень сжатия сильно зависит от типа сжимаемых данных. Наиболее эффективно использование BWT для текстов и любых потоков данных со стабильными контекстами. В этом случае рассматриваемые компрессоры по своим характеристикам близки к PPM-сжимателям при, как правило, большей скорости.

На неоднородных данных известные BWT-архиваторы заметно уступают по сжатию лучшим современным сжимателям на LZHuf и чуть не дотягивают до результатов, показываемых представителям PPM-сжатия. Впрочем, есть способы сильно увеличить сжатие неоднородных файлов. Такие уловки пока не используются в связке с BWT, возможно, из-за сравнительно молодого возраста последнего.

Расходы памяти при сжатии довольно близки у всех рассматриваемых методов и могут колебаться в больших пределах. Если не рассматривать специфические экономные реализации - LZHuf с маленьким словарем, PPM с малым числом контекстов и BWT с кластерной сортировкой, то основные различия в требованиях к памяти наблюдаются при декодировании. Наиболее скромными в этом отношении являются архиваторы, использующие LZHuf. PPM'ы памяти требуют столько же, сколько же и при сжатии. Промежуточное положение занимают BWT-компрессоры.

4.2. Существующие BWT-архиваторы и их особенности.

Компрессор и версия	Автор	Адрес

bredX		D.J.Wheeler	ftp://ftp.cl.cam.ac.uk/users/djw3
bzip	0.21	Julian Seward	<jseward@acm.org>
bzip2	1.01	Julian Seward	<jseward@acm.org> http://sourceware.cygnus.com/bzip2
szip	1.12	Michael Schindler	<michael@compressconsult.com> http://www.compressconsult.com
imp -2	1.10	Conor McCarthy	<imp@technelysium.com.au> http://www.technelysium.com.au
x1 -m7	0.95	Stig Valentini	<x1develop@dk-online.dk> http://www.saunalahti.fi/~x1
bwc	0.99	Willem Monsuwe	<willem@stack.nl> ftp://ftp.stack.nl/pub/users/willem
ba	1.01br5	Mikael Lundqvist	<mikael@2.sbbs.se> http://hem.spray.se/mikael.lundqvist
zzip	0.36b	Damien Debin	<damien.debin@via.ecp.fr> http://www.zzip.f2s.com
dc	0.99.015b	Edgar Binder	<EdgarBinder@t-online.de> ftp://ftp.elf.stuba.sk/pub/pc/pack
eri	4.8fre	Alexander Ratushnyak	<ratush@srsc-gw.sscc.ru> http://artest.i.am
ybs	0.03e	Vadim Yoockin	<yoockinv@mtu-net.ru> <vy@thermosyn.com> http://members.xoom.com/vycct
sbc	0.361b	Sami Mdkinen	<sjm@pp.inet.fi> www.geocities.com/sbcarchiver

Для сжатия данных в этих архиваторах обычно применяется арифметический метод, за исключением BZIP2 и IMP, которые используют метод Хаффмана.

Почти во всех указанных архиваторах в дополнение к BWT находит свое применение MTF-преобразование. Кроме DC, YBS и SBC, в которых отдано предпочтение кодированию расстояний.

Семейство BRED'ов написаны одним из родоначальников BWT, когда узок был круг людей, занимающихся этим методом. Многие идеи, использованные в этих компрессорах, описаны в работах Фенвика. Многие идеи, описанные в [2] нашли свое отражение в BZIP, BZIP2, BWC, IMP, ZZIP, BA и YBS.

BZIP использует сортировку, выросшую из BRED'ов и структурную модель, описанную Петером Фенвиком в одной из его статей. Выход MTF-преобразования сжимается арифметическим кодером с использованием так называемого 1-2 кодинга для сжатия повторяющихся последовательностей нулей. Аналогично устроен и BWC, за тем исключением, что многие алгоритмы в нем оптимизированы.

BZIP2, в отличие от BZIP, выход MTF-преобразования дожимает при помощи метода Хаффмана. Сортировка изменена незначительно - только повышена устойчивость к избыточным данным. Доступность исходных текстов BZIP и BZIP2 привела к появлению большого количество клонов, использующих те же идеи.

IMP использует собственную сортировку, очень быструю на обычных текстах, но буквально зависающую на избыточных данных. Кодирование полностью позаимствовано из BZIP2.

В SZIP, помимо упоминавшегося ST, реализована и возможность использования BWT-преобразования. Реализована, прямо скажем, только для примера, без затей. А вот дожиматель - очень интересный.

Представляет из себя некий гибрид MTF-преобразования и адаптивного кодера, ведущего статистику при помощи короткого окна по выходу BWT-преобразования.

Новые версии ZZIP'а выпускаются очень часто и свойства этого компрессора постепенно улучшаются. В нем применяется все те же испытанные временем структурная модель и сортировка из BZIP.

BA использует сортировку, похожую на BZIP'скую тем, что все строки также делятся на пакеты, которые сортируются отдельно при помощи Multikey QuickSort [5]. За счет динамического определения размера блока заметно улучшено сжатие неоднородных файлов. Для усиления сжатия английских текстов используется переупорядочивание алфавита.

DC - достаточно новый компрессор, в котором реализован целый ряд новаторских идей. Во-первых, конечно, это модель сжатия, отличная от MTF - distance coding. Во-вторых, новый метод сортировки (информация о нем, может быть, будет опубликована позже), очень быстрый на текстах, хотя и дающий слабину на сильно избыточных данных. И, наконец, большой набор текстовых фильтров, позволяющий добиться особенного успеха на английских текстах.

Отличительная особенность SBC - в наличии мощной криптосистемы. Ни в одном из архиваторов, пожалуй, не реализовано столько известных алгоритмов, как в SBC. Для сжатия в нем используется BWT, ориентированный на большие блоки избыточных данных. И хотя SBC не входит в тройку самых быстрых BWT-компрессоров на типичных данных, он ловко сортирует данные с большим количеством длинных повторяющихся строк, достигая при этом очень хорошей скорости.

YBS основан на сортировке, аналогичной BZIP2. В перспективе думаю сделать сортировку, более экономно расходующую память и дающую значительное ускорение на коротких контекстах. Сейчас на текстах уступает по скорости IMP'у и DC, но на очень избыточных данных их обгоняет. По скорости сжатия на данный момент - один из самых быстрых среди BWT-компрессоров, использующих арифметическое кодирование. На текущий момент достигнута степень сжатия близкая к DC, за исключением того, что YBS пока не использует фильтры.

Как ведут себя эти сжиматели по сравнению с другими можно посмотреть на http://members.xoom.com/vycct и на русскоязычном сайте http://arctest.cjb.net. Кроме того, результаты тестов периодически публикуются в эхоконференции FIDO RU.COMPRESS.

Большую часть из указанных архиваторов можно также взять на ftp://ftp.elf.stuba.sk/pub/pc/pack

Много полезной информации можно почерпнуть на сайте на http://dogma.net/DataCompression/BWT.shtml

5. Литература.

1. M. Burrows and D.J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", SRC Research Report 124, Digital Systems Research Center, Palo Alto, May 1994 gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z
2. P.M. Fenwick, "Block sorting text compression", Australasian Computer Science Conference, ACSC'96, Melbourne, Australia, Feb 1996. ftp.cs.auckland.ac.nz/out/peter-f/ACSC96.ps
3. M.R. Nelson: Data Compression with the Burrows Wheeler Transform. Dr. Dobbs Journal, Sept. 1996, pp 46-50.

http://web2.airmail.net/markn/articles/bwt/bwt.htm
4. DI Michael Schindler. A Fast Block-sorting Algorithm for lossless Data Compression. IEEE Data Compression Conference, 1997.
5. Jon L. Bentley, Robert Sedgewick. Fast Algorithms for Sorting and Searching Strings. Dr.Dobb's Journal, 1998.
6. Manber, U. and Myers, G. Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22 (1993), 935-948.
7. Bentley, J.L. and McIlroy, M.D. Engineering A Sort Function. Software-Practice and Experience 23, 1 (1993), 1249-1265.
8. B.Balkenhol, S.Kurtz, Y.M.Shtarkov. Modifications of the Burrows and Wheeler Data Compression Algorithm. In Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, IEEE Computer Society Press, 1999, 188-197.
9. B.Balkenhol, S.Kurtz. Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice. 1999
10. M.Arimura, H.Yamamoto. Asymptotic Optimality of the Block Sorting Data Compression Algorithm. 1998
11. Z.Arnavaut, S.S.Magliveras. Block Sorting and Compression. Data Compression Conference, 1999.
12. K.Sadakane. A Fast Algorithm for Making Suffix Arrays and for BWT.
13. N.J.Larsson, K.Sadakane. Faster Suffix Sorting.
14. B.Chapin. Switching Between Two On-line List Update algorithms for Higher Compression of Burrows-Wheeker Transformed Data. Data Compression Conference, 2000.
15. B.Chapin, S.Tate. Higher Compression from the Burrows-Wheeler Transform by Modified strings. 2000.
16. P.Ferragina, G.Manzini. An experimental study of an opportunistic index. 2001.
17. S.Albers, B.v.Stengel, R.Werchner. A combined BIT and TIMESTAMP Algorithm for the List Update Problem. 1995.
18. F.Schulz. Two New Families of List Update Algorithms. 1998.
19. C.Ambuhl, B.Gartnerm B.v.Stengel. A New Lower Bound for the List Update Problem in the Partial Cost Model. 1999.
20. S.Deorowicz. Improvements to Burrows-Wheeler Compression Algorithm. 2000.
21. S.Deorowicz. An analysis of second step algorithms in the Burrows-Wheeler compression algorithm. 2000.
22. H-K.Baik, D.S.Ha, H-G.Yook, S-C.Shin, M-S.Park. A New Method to Improve the Performance of JPEG Entropy Coding Using Burrows-wheeler Transformation. 1999.