Санкт-Петербургский Государственный
Технический Университет

Фомин А.А.

Основы сжатия информации

Санкт-Петербург 1998

1. Введение.

Сжатие информации является одним из самых интересных и динамически развивающихся направлений современной науки - теории информации. Вопросы сжатия данных достаточно остро стоят в различных областях науки и техники, везде, где требуется хранение и передача информации. Во-первых, это связано со все еще достаточно высокой стоимостью носителей информации (магнитные и оптические диски, ПЗУ, ОЗУ и т.д.), во-вторых, с необходимостью передачи больших потоков информации по перегруженным линиям связи (радио- и оптическая связь, телефония, сети). Кроме того, сжатие данных неразрывно связано с криптографией и защитой информации от случайного и преднамеренного воздействия. Естественно, универсального алгоритма сжатия данных не существует и для каждой конкретной задачи имеется свой наиболее эффективный метод. Более того, применение нерационально выбранного алгоритма может привести к противоположному результату увеличения потока информации. Иными словами, всегда можно найти такой набор данных, для которого выбранный способ сжатия окажется неэффективным.

Первое знакомство автора с программами сжатия информации (**архиваторами**, **компрессорами**) вызвало в нем одновременно удивление и любопытство: как можно сократить то, что и без того достаточно беспорядочно и нелишне? Еще более поразительно, что текст, уменьшенный в размере в четыре!!! раза может быть восстановлен обратно с точностью до единой буквы! А, как известно, любопытство и удивление-основные причины познания, поэтому вполне понятно стремление узнать: ”А почему ЭТО работает и как ЭТО работает?”

С другой стороны-написание самостоятельной эффективной программы-архиватора может рассматриваться как серьезный тест на звание опытного программиста, поскольку создание подобного продукта невозможно без виртуозного владения языком программирования (в том числе ассемблером), знания архитектуры компьютера, обоснованного выбора структур данных и базовых алгоритмов (сортировка, поиск и т.д.).

К сожалению, в отечественной литературе архивации уделено чрезвычайно мало внимания. Автору известно лишь об одной серьезной книге, посвященной этому вопросу [38], да еще о нескольких переполненных формулами статьях в солидных научных журналах и неплохой подборке в журнале “Монитор” [32, 34, 40]. С зарубежными изданиями дело обстоит несколько лучше, но они *зарубежные*. Плюс еще имеется огромный поток разрозненной информации в “электронном виде”. Поэтому...

 Данная книга призвана помочь сделать первый шаг в изучении методов сжатия информации. Она содержит изложение принципов сокращения избыточности и описание основных известных к настоящему времени алгоритмов архивации с их достоинствами, недостатками и путями оптимизации. Книга не требует от читателя специальной математической или программистской подготовки-достаточно обладать начальными знаниями теории вероятности, навыками программирования на языке C и некоторыми понятиями о структурах данных. Для этих целей вполне подойдут прекрасные издания [39, 44].

Изложение ведется на двух уровнях: первый-общедоступный; второй-для углубленного изучения-набран более мелким шрифтом и может быть пропущен без ущерба для дальнейшего понимания. Текст сопровождается большим количеством несложных примеров и минимально необходимым набором формул и теорем. В конце каждого раздела помещены задачи разной сложности-от простых, помеченных номером (1), до сложных-(5), к некоторым из них даны решения. В конце книги приведены листинги нескольких алгоритмов, библиография список обозначений и используемых терминов. По ходу текста дается обзор существующих зарубежных программ архивации файлов.

 В то же время, читатель не найдет в этой книге строгих математических доказательств приводимых утверждений и подробных листингов и алгоритмов современных высокоэффективных архиваторов (небольшой объем издания и закон об авторских правах не позволяют этого сделать). Однако, автор

уверен, что изложенного здесь материала достаточно для того, чтобы потом самому уверенно продвигаться к оптимальным результатам.

На протяжении всей книги читателя будут сопровождать следующие пиктограммы:

 -Вопросы и задачи;  -Ответы и решения;

 - Преимущества, достоинства;  - Недостатки;

 - Выводы;  - Пример;

 - Материал для углубленного изучения;

 - Основные понятия и определения;

 - Пути оптимизации, варианты алгоритма.

Книга основана на материалах читаемого на протяжении нескольких лет курса лекций “Защита информации при хранении” в рамках специальности “Основы защиты информации” на факультете технической кибернетики СПбГТУ и является его дальнейшим развитием. Она также может быть полезна изучающим курсы “Теория и технология программирования”, “Структуры данных”, “Программное обеспечение персональных компьютеров” и т.д. Автор благодарен сотрудникам кафедры “Измерительных информационных технологий” ФТК, заместителю декана ФТК Евдокимову В.Е., другим работникам университета и его фундаментальной библиотеки, оказавшим поддержку в написании этой книги и помогавшим по крупицам собирать предлагаемый здесь материал.

2. Основные понятия и определения.

2.1 Используемая терминология.

 Предварительно необходимо определить некоторые термины, широко используемые в дальнейшем изложении.

В настоящей книге широко используется понятие **оптимизация**. Смысл этого термина вполне очевиден, однако при более внимательном рассмотрении он приобретает более глубокий и даже философский оттенок. В архиваторах стремятся улучшить два основных параметра-*степень сжатия* и *скорость работы*. (Вообще говоря, скорость работы распадается на два конкурирующих параметра-скорость архивации и скорость разархивации. Часто бывает неважно, как долго мы сжимаем данные, но критичным является время обратного преобразования или наоборот). Иногда немаловажным оказывается также *объем используемой памяти*. Остальные свойства менее значительны и обычно не рассматриваются (к ним можно отнести универсальность, сложность алгоритма, количество проходов, возможность аппаратной реализации, устойчивость к ошибкам и т.д.).

Для трех главных параметров обычно редко удается улучшить что-нибудь одно, не ухудшив при этом другого, поэтому говорить об оптимизации вообще не имеет смысла-то, что оптимально для одних приложений или типов данных, может быть совершенно не пригодно для других. И, тем не менее, если такое происходит, можно говорить об истинно оптимальном алгоритме, о научном прорыве или открытии.

Будем называть входную последовательность данных **текстом**. Текст состоит из **символов**, полный набор которых назовем входным **алфавитом**. Процесс сжатия заключается обычно в том, что каждому символу из входного алфавита ставится в соответствие символ (код) из выходного алфавита таким образом, что выходной текст оказывается короче входного. Процесс восстановления (“разжатия”) выполняет обратную процедуру с целью восстановления исходного текста. Следовательно алгоритмы сжатия (архивации) и

разархивации должны быть взаимно обратимы. Входной алфавит состоит обычно из байтов (8 бит) или 256 различных символов ASCII (если это не будет оговорено особо).

□В некоторых случаях не требуется точного восстановления информации и допускается использовать алгоритмы, реализующие **сжатие с потерями**, которое, в отличие от **сжатия без потерь** обычно проще реализуется и обеспечивает более высокую степень архивации. В выборе того или иного способа следует руководствоваться здравым смыслом.

 Например, при измерении температуры воздуха в комнате мы можем пренебречь всеми значащими цифрами, кроме первых двух: не 22.758, а просто 23 градуса. Если же мы осуществляем банковскую операцию, то потеря третьего-четвертого разряда в рублевом счете может обернуться миллионными (а, может быть, вскоре и более) убытками.

Обычно, сжатие с потерями применяется при обработке файлов, содержащих числовую, звуковую информацию или изображения. Сжатие программных файлов требует безусловного восстановления без потерь.

□Некоторые алгоритмы архивации имеют неприятное свойство **распространения ошибки**, которое проявляется в том, что однажды возникшая в сжатом тексте ошибка портит всю дальнейшую информацию. Тексты, сжатые такими алгоритмами, должны быть особо защищены от сбоев при передаче и хранении. К сожалению, большинство современных программ-архиваторов страдает от этого недостатка, т.к. использует в работе коды переменной длины.

□Будем называть **префиксом** некоторый служебный код, предписывающий алгоритму разархивации обрабатывать следующие несколько кодов из выходного потока особым образом. Иными словами, префикс как бы поясняет, что представляют из себя следующие коды и как с ними поступать. С другой стороны, **префиксом** некоторого символа называют любой код, совпадающий с началом этого символа. Смысл термина “префикс” обычно понятен из контекста, поэтому мы не будем пояснять, какое из двух значений используется.

Алгоритм архивации может работать в **реальном масштабе времени и в потоке**, если в нем не происходит накопления информации и он формирует выходной текст по мере поступления входного сообщения (а не *после*). Алгоритм называется

-однопроходным, если для получения сжатого текста ему необходимо только один раз “просмотреть” исходный текст;

-блочным, если отдельный выходной код соответствует более, чем одному входному символу.

Для работы в реальном масштабе времени очевидно требуется, чтобы алгоритм был однопроходным.

В [25] дана следующая классификация методов архивации:

-статистический (statistical), если предполагает соответствие входного потока определенной модели сигнала (см. раздел 2.5) и осуществляет сжатие на основе собранной о тексте статистической информации;

-инкрементальный (incremental), осуществляющий сжатие путем кодирования отличий в последовательных записях;

-макро или текстовой подстановки (macro or textual substitution), выполняющий сжатие путем поиска совпадающих строк и замены их на более короткие коды.

2.2 Что такое информация - по-научному.

Информация в теории информации определяется только вероятностными (статистическими) свойствами сообщений. Все другие их свойства, например, полезность, интонации или принадлежность автору, игнорируются.

 **Пусть {X, p(x)} ансамбль сообщений, $X=\{x_1,x_2,...,x_L\}$, $p(x_i)$-вероятность i-го сообщения. Количеством собственной информации или собственной информацией [47] в сообщении x_i называется число $I(x_i)$, определяемое как:**

$$I(x_i) = -\log(p(x_i)),\ i = 1, 2, ..., L$$

Если основание логарифма-2, то количество информации измеряется в **битах**, если 10, то в **натах**. В дальнейшем, если не будет оговорено особо, подразумевается основание 2.

Свойства информации:

1. Информация всегда неотрицательна: $I \geq 0$.

2. Для независимых сообщений $\mathbf{p(x_i,y_j) = p(x_i)\cdot p(y_j)}$, следовательно: (свойство аддитивности)

$$\mathbf{I(x_i,y_j) = I(x_i) + I(y_j)}$$

Математическое ожидание H(X) случайной величины I(x), определенной на ансамбле {X, p(x)}, называется энтропией этого ансамбля:

$$\mathbf{H(X) = M\, I(x) = \sum_{i=1}^{L} I(x_i) \bullet p(x_i) = -\sum_{i=1}^{L} p(x_i) \bullet \log(p(x_i))}$$

(M-оператор математического ожидания).

Иными словами-это среднее количество собственной информации в сообщениях ансамбля **X**. Энтропия рассматривается также как мера беспорядка в системе. Как известно, энтропия имеет тенденцию постоянно возрастать (что мы и наблюдаем в наше время). В Листинге 1 приложения приведена простая программа расчета энтропии файлов, содержащих произвольную информацию.

Свойства энтропии:

1. Энтропия неотрицательна: $\mathbf{H(X) \geq 0}$.

2. Энтропия не может превышать логарифма количества сообщений: $\mathbf{H(X) < \log L}$.

3. Для статистически независимых сообщений: (свойство аддитивности)

$$\mathbf{H(XY) = H(X) + H(Y)}$$

Поставим в соответствие каждому сообщению $\mathbf{x_i}$ в соответствие код $\mathbf{a_i}$ из некоторого **алфавита** $\mathbf{A=\{a_1,a_2,...,a_L\}}$. Тогда, **стоимостью кодирования** будет называться величина

$$\mathbf{C(X) = \sum_{i=1}^{L} |a_i| \bullet p(x_i)}$$

где **|a|**-длина кода a в битах. В случае, когда длины всех кодов одинаковы и равны **logL**, мы получаем: **C(X)=logL**. Иначе говоря, под стоимостью кодирования понимается средняя длина кодового слова в битах (то же самое, что и $\mathbf{H_{max}}$).

Условная собственная информация сообщения x при известном сообщении y:

$$\mathbf{I(x|y) = -\log(p(x|y))}$$

Условная энтропия ансамбля X относительно сообщения y:

$$\mathbf{H(X|y) = \sum_{i=1}^{L} p(x_i|y) \bullet I(x_i|y) = -\sum_{i=1}^{L} p(x_i|y) \bullet \log(p(x_i|y))}$$

Условная энтропия ансамбля X относительно ансамбля Y (средняя условная энтропия):

$$\mathbf{H(X|Y) = M\ H(X|y_j) = \sum_{j=1}^{N} p(y_j) \bullet H(X|y_j) = -\sum_{i=1}^{L}\sum_{j=1}^{N} p(x_i, y_j) \bullet \log(p(x_i|y_j))}$$

Свойства:

1. **$H(X|Y) \le H(X)$,** и равно, если ансамбли независимы.
2. **$H(XY) = H(X) + H(Y|X) = H(X|Y) + H(Y) \le H(X) + H(Y)$**

Относительная энтропия ***H*** **- отношение энтропии источника к максимальному значению, которое могло бы быть достигнуто при тех же символах: $H = H/H_{max} = H/C$.**

Избыточность кодирования R равна разности между стоимостью кодирования C и энтропией H: R = C - H.

Величина **R** для декодируемых сообщений всегда больше нуля и в пределе, для идеального архиватора должна стремиться к нулю. Относительная избыточность определяется по формуле:

$$\boldsymbol{R = (1 - H) \bullet 100\% = \frac{C - H}{C} \bullet 100\%}$$

Если средняя длина кодового слова входного текста равна **T**, то в случае выполнения неравенства **T > C** мы можем говорить о том, что имеет место сжатие

информации. С помощью идеального архиватора, для которого **R**=0, можно определить избыточность входного текста $\mathbf{R_t}$:

$$\mathbf{R_t = T - H = T - C\ (бит)}$$

1. Докажите формально свойства и уравнения для энтропии и условной энтропии, приведенные в разделе. (4)

Ветер, ветер! Ты могуч, ты гоняешь стаи туч,
Ты волнуешь сине море, всюду веешь на просторе,...
Аль откажешь мне в ответе? Не видал ли где на свете
Ты царевны молодой? Я жених ее...
(А.С.Пушкин)

2.3 Что такое информация - просто о сложном.

 Попробуем теперь пояснить на простых примерах что же такое энтропия и избыточность-эти основополагающие понятия теории сжатия информации.

Рассмотрим эпиграф к разделу. Если отвлечься от льстивого восхваления свойств и достоинств ветра, а также от родственных связей царевича, то суть этого четверостишия сведется к следующему: ”Не видал ли ветер царевны?” На что хотелось бы получить такой же краткий ответ “Да” или “Нет”. Здесь мы имеем дело с избыточностью, которую автор называет **смысловой** (а в народе-”лить воду”). Такого рода избыточность присуща только человеческому общению и требует для своего анализа аппарата искусственного интеллекта, поэтому в архиваторах она пока не учитывается. После устранения смысловой избыточности остается **физическая** избыточность, связанная с видом или формой представления информации (см. рис. 2.а). В любом языке имеется значительная физическая избыточность. Мы можем, н.пр.м.р в.к.н.ть вс. гл.сн.. б.кв., ил. вы.ин.ть ка.ду. тр.ть. бу.ву, или отброс... окончан.. кажд... слов., и все равно текст будет понятен.

После того, как мы закодируем информацию с помощью какого-нибудь выбранного нами кода (может быть, неоптимального), останется **статистическая** избыточность. Алгоритмы архивации имеют дело в основном со статистической и в меньшей мере физической избыточностью.

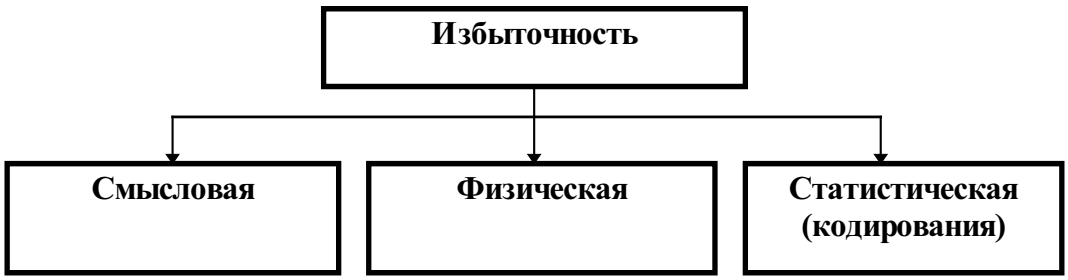


Рис. 2.A Виды избыточности.

Приведем еще несколько примеров избыточности (рис. 2.b): букву *A* можно представить многими разными способами-нарисовать жирным карандашом, нарисовать тонкой линией по линейке, закодировать в матрице знакогенератора размером 8x8 бит или закодировать в ASCII коде.

Рис. 2.B Примеры избыточности.

Первый случай демонстрирует значительную смысловую и физическую избыточность, во втором случае с помощью алгоритма прореживания [43] смысловая избыточность сведена до минимума (для изображения второй буквы *A* достаточно нескольких команд “черепашьей графики”), в третьем варианте еще

больше снижена физическая избыточность и, наконец, четвертая буква имеет, может быть, лишь избыточность кодирования.

Не следует, однако, забывать, что то, что в одном месте является избыточным, в другом месте может оказаться важной и необходимой информацией. Например, для художника или психолога очень важен именно образ нарисованной буквы, который следует правильно передать. Поэтому, для выбора алгоритма архивации важно прежде всего правильно установить или выделить информативные параметры, допускающие или не допускающие сжатие с потерями.

Рассмотрим следующий пример. Пусть нам требуется передавать партнеру одно из трех сообщений “да”, “нет” или “не знаю” (X={“да”,”нет”,”не знаю”}). Очевидно, эти сообщения страдают от большой смысловой избыточности и мы лишь ускорим обмен информацией, если положим X={“д”,”н”,”?”}. Можно не останавливаться на достигнутом и передавать не символы, а двоичные коды т.е. поставим в соответствие сообщениям алфавит A={00, 01, 10}. Нетрудно заметить, что и теперь осталась небольшая физическая избыточность: если партнер принял бит 1, он уже понял, что ответ “не знаю” и ему нет необходимости принимать (а нам нет необходимости загружать линию связи) вторым, бесполезным битом. Поэтому, теперь A={00, 01, 1}. Здесь мы произвольно присвоили одному из сообщений более короткий код, но теория информации позволяет более строго назначать коды и получать еще менее избыточные сообщения уже с учетом их вероятностей $p(x_i)$.

И, наконец, последний пример из области человеческого языка [47]. Избыточность английского печатного текста составляет 50-70%. При избыточности, равной 0 любое двумерное построение букв представляет кроссворд, большие кроссворды становятся возможными при избыточности менее 50%, при избыточности менее 33% возможны трехмерные кроссворды.

Если определить n-граммную энтропию Hn как энтропию с учетом статистических связей не длиннее n символов, то для английского языка она будет распределяться как показано на рис. 2.a (Hn для 26 букв примерно равна 4.5/5.5=0.818 Hn для 26 букв плюс пробел).

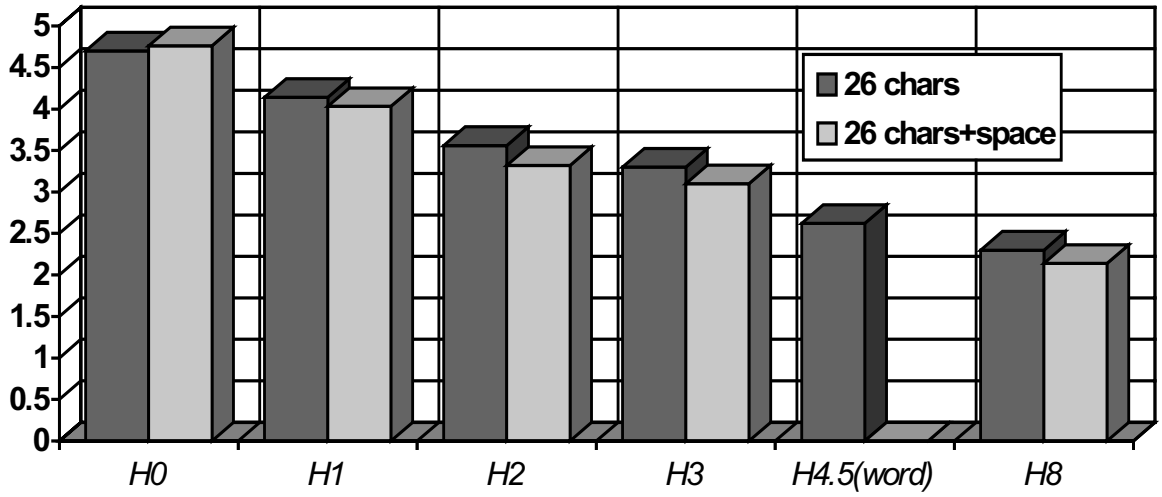


Рис. 2.С n-граммная энтропия для английского текста.

Оценить энтропию H_0 можно, посчитав в каком-нибудь литературном тексте вероятности появления различных букв алфавита $\mathbf{p(a_i)}$ и воспользовавшись формулами из раздела 2.2 (здесь алфавит-все коды ASCII, $\mathbf{L}$=256, $|\mathbf{a_i}|$=8).

1. Постройте таблицу частоты появления букв русского алфавита. Вариант: не делать различия между строчными и заглавными буквами. (1)

2. Какой текст имеет большую избыточность-написанный только строчными буквами или с использованием еще и заглавных букв? (2)

3. Оцените H_0 для русского языка и сопоставьте с цифрами, представленными для английского языка. (2)

4. Оцените H_1 для русского языка и сопоставьте с цифрами, представленными для английского языка. (3)

5. Найдите энтропию и избыточность разных способов кодирования сообщений “да”, ”нет”, ”не знаю” из рассмотренного выше примера, полагая, что вероятности всех сообщений одинаковы. (3)

2.4 Для чего нужны архиваторы?

Вопросы сжатия информации (и не только информации) нам приходится решать в обыденной жизни постоянно и мы даже не задумываемся об этом. А

между тем, именно из реальной жизни наука об архиваторах черпала свои основные методы и алгоритмы. Основная задача архиватора-ускорить передачу сообщения или сократить занимаемое им место на физическом носителе. Поэтому, когда мы...

звоним в дверь условным звонком;

посылаем телеграмму;

работаем на компьютере;

переносим в записную книжку адреса и телефоны с дюжины клочков бумаги;

пытаемся засунуть в чемодан как можно больше вещей;

конспектируем лекции;

работаем “морзянкой”

и даже разговариваем-

мы делаем нечто очень похожее на сжатие информации. Но в некоторых областях науки и техники архиваторы являются незаменимыми помощниками.

2.4.1 Применение в криптографии.

Сжатие информации является одной из составных частей криптографии, позволяющей значительно повысить стойкость криптосистем. Действительно, понижая избыточность входного текста, мы даем криптоаналитику, пытающемуся дешифрировать сообщение, меньше необходимой для успешной работы информации. Об этом, видимо, не знали отрицательные герои рассказа А. Конан-Дойля “Пляшущие человечки”, что и привело к краху солидной тайной организации.

Еще Шеннон [47] сделал естественный вывод о том, что сжатие данных-полезное средство в криптографии. Совершенный алгоритм сжатия данных преобразует источник сообщений в полностью случайный (безизбыточный). Однако, даже несовершенный метод сжатия позволяет значительно увеличить *расстояние единственности* (наименьшее количество символов текста, по которым можно восстановить секретный ключ). Еще в те времена, когда не было теории информации, а сообщения обрабатывались вручную, криптографы часто

удаляли из открытого текста те буквы и пробелы, которые могли быть восстановлены законным получателем [41].

ВОТПРСТОЙПРМЕРСЖАТИЯДННЫХ.

Существуют еще два аспекта, связывающих криптологию и сжатие информации [42,46]:

- сжатый текст труднее поддается перехвату при передаче по линии связи и его труднее обнаружить при хранении (но не следует забывать что при этом возрастает опасность распространения ошибки);
- сжатие информации с варьируемыми в зависимости от ключа параметрами может рассматриваться как один из нетрадиционных программных криптоалгоритмов.

2.4.2 Применение в системах связи.

Родоначальником архиваторов в связи смело можно назвать азбуку Морзе, в которой заложена очевидная и до гениальности простая идея-чем чаще используется буква алфавита, тем короче для повышения скорости связи должен быть ее код.

В дальнейшем, алгоритмы сжатия информации стали широко использоваться в факс-модемной- и радиосвязи, позволяя в обоих случаях повысить скорость передачи информации или, что то же самое, увеличить пропускную способность канала связи во столько раз, во сколько раз удалось уменьшить объем передаваемых данных.

 Для телекоммуникационной передачи долгое время фактическим стандартом системы сжатия данных был протокол *MNP5* фирмы Microcom, предлагавший, в среднем, двукратное сжатие отправляемых сообщений. В конце 1989 г. не смену ему пришел протокол CCITT *V42.bis*, достигающий на практике 2-3-кратного повышения пропускной способности [45]. В [38] описан пример использования сжатия информации (с потерями) для передачи в 1973 г. изображений с искусственного спутника Луны. Канал связи обеспечивал скорость 625 бит/с, в то время как телекамера выдавала информацию со скоростью 20000

бит/с. Оригинальный алгоритм, разработанный Линчем и Миллером на основе прореживания и **RLE**-кодирования, позволил получить сжатие данных в 32! раза.

Вообще, сжатию изображений среди различных направлений архивации уделяется особенно большое и пристальное внимание, для этих целей выпускаются специализированные процессоры [28] и разрабатываются специальные алгоритмы [35].

Например, микросхема, описанная в [28] благодаря комбинации одно- и двумерных инкрементальных алгоритмов совместно с алгоритмами **Хаффмена** и **RLE**, способна при сжатии черно-белой графической информации (факсов) достигать 5-50-кратного уменьшения объема передаваемых данных без потерь.

2.4.3 Применение в системах хранения информации.

Лавинообразное нарастание интереса к архиваторам началось в последние десять лет с появлением IBM-совместимых персональных компьютеров (ПК). До этого тоже существовали программы сжатия информации (например *Compress* в UNIX, *SQUEEZE* в CP/M), но они были известны только специалистам и обеспечивали несравнимо худшие показатели сжатия информации. Сейчас это постоянная тема многих журналов по вычислительной технике, форумов и семинаров в телекоммуникационных сетях и просто обсуждений в среде программистов и любителей. Действительно, вряд ли найдется пользователь ПК, у которого на диске не хранилось бы полдюжины программ архивации файлов, о некоторых из которых он даже не подозревает. Это:

- универсальные программы сжатия информации, среди которых всемирно известные зарубежные программы *PKZIP* [21], *ARJ* [3], *LHA, HA*, и не менее эффективные отечественные-*RAR, AIN, ACB* (которые должны обеспечивать высокую степень сжатия, пусть даже ценой низкого быстродействия);
- самораспаковывающие архиваторы (см. предыдущий абзац) и программы сжатия исполняемых файлов: *PKLITE, LZEXE, DIET, EXEPACK* (к ним предъявляется требование быстрой распаковки при условии высокой степени сжатия);

системные утилиты сжатия дискового пространства на уровне отдельных секторов *Stacker, DoubleSpace, SuperStor* (требуется высокая скорость компрессии/декомпресии при менее жестких требованиях по избыточности);

встроенные в Windows API функции *LZxxx*;

различные программы преобразования и показа форматов видеоизображений, типа *PCXVIEW, GIFVIEW, BITMAP, JPGVIEW, QuickTime*.

Без перечисленных средств жесткий диск размером в 1 Гигабайт уже давно не казался бы достаточным для хранения программ и данных, а фирменное программное обеспечение можно было бы унести разве что в портфеле.

1. Как изменится пропускная способность канала связи со сжатием информации, если по нему передавать файл, обработанный каким-нибудь архиватором? (1)

2. Как изменится пропускная способность канала связи со сжатием информации, если по нему передавать файл, обработанный каким-нибудь архиватором? (2) (Это не опечатка, просто вторая задача предполагает более глубокое понимание проблемы. Кстати, хорошо ли хранить архивы на виртуальных дисках, созданных *Stacker* или *DoubleSpace*?)

2.5 Моделирование источника информации.

Для создания оптимального архиватора недостаточно быть хорошим программистом и разобрать дюжину известных алгоритмов. Необходимо правильно оценить модель источника сигнала, которой подчиняются входные сообщения и выбрать тот алгоритм, который разрабатывался на основе этой модели [23]. Неправильно выбранная модель сигнала может привести к тому, что “хороший” архиватор откажется сжимать входную информацию, хотя она будет достаточно избыточной. Несмотря на разнообразие различных сигналов в основу их анализа положено всего две основные **вероятностные** модели: модель источника сигнала ***Бернулли*** и ***Маркова*** [38]. Понятно, что реальные сигналы лишь более или менее соответствуют выбранной модели, а поэтому:

архиваторы, основанные на разных моделях, могут давать схожий результат;

выбор лучшей модели-в значительной мере сложный и творческий процесс;

существуют универсальные архиваторы, приспособленные для работы с различными моделями сигнала.

2.5.1 Источники Бернулли.

Сигнал порождается источником Бернулли, если очередное сообщение или код *не зависит* от всех предыдущих, иными словами все сообщения независимы и характеризуются только своими вероятностями. Примером процессов, порождаемых источником Бернулли являются, например, результаты бросания монеты или игрального кубика. Результаты бросания кубика-сигнал с равными вероятностями всех событий. Источник такого сигнала называют еще **комбинаторным**. Если же на кубик нанести числа 1, 2, 2, 3, 3, 3, то мы получим прекрасный источник разновероятных сообщений Бернулли. Если в ансамбле сообщений соблюдается одно из двух правил:

$\mathbf{p(x_i) \geq p(x_j)}$ для любого $\mathbf{i > j}$ или

$\mathbf{p(x_i) \leq p(x_j)}$ для любого $\mathbf{i > j}$,

то такой источник называется **монотонным**.

2.5.2 Источники Маркова.

Источник Маркова-более сложная модель. В ней предполагается, что вероятность появления очередного сигнала зависит от n предыдущих сигналов. Соответственно получается источник сигнала Маркова n-го порядка. Чем больше n, тем сложнее учет и оценка зависимостей, поэтому на практике ограничиваются значением n=1..2. Очевидно, что модель Бернулли-это модель Маркова при n=0.

Если для источника Бернулли необходимо знать только вероятности отдельных сообщений, то для источника Маркова необходимо получить условные вероятности в виде n+1-мерных матриц. Для марковского источника первого порядка это будет двумерная матрица размера LxL, где каждый i-й столбец показывает вероятность перехода от сообщения x_i к сообщению x_j, например:

	x_1	…	x_{i-1}	x_i	x_{i+1}	…	x_L
x_1	$p_{1,1}$	…	…	$p_{1,i}$	…	…	$p_{1,L}$
…	…	…	…	…	…	…	…
x_{j-1}	$p_{j-1,1}$	…	…	$p_{j-1,i}$	…	…	$p_{j-1,L}$

x_j	$p_{j,1}$	…	…	$p_{j,i}$	…	…	$p_{j,L}$
x_{j+1}	$p_{j+1,1}$	…	…	$p_{j+1,i}$	…	…	$p_{j+1,L}$
…	…	…	…	…	…	…	…
x_L	$p_{L,1}$	…	…	$p_{L,i}$	…	…	$p_{L,L}$

(Сумма вероятностей любой строки или столбца должна равняться единице.)

Примером источника Маркова являются буквы русского (или, например, английского алфавита), поскольку появление в слове очередной буквы очень сильно зависит от предыдущих букв или даже слов.

Никакой архиватор, работающий на основе модели Маркова n-го порядка не способен сжать информацию до такой степени, что ее стоимость кодирования станет меньше исходной энтропии того же порядка. Таким образом, числа, представленные на рис. 2.c, дают нам нижнюю границу степени сжатия текста практически любым из рассматриваемых архиваторов. Из этого рисунка также следует, что чем ближе эта граница, тем труднее изобрести алгоритм, дающий существенное сжатие и оправдывающий усилия разработчиков.

В зависимости от того, какой модели источника сигнала подчиняется информация, ее удобно разбить (достаточно условно) на несколько групп или видов, например, как показано на рис. 2.d.

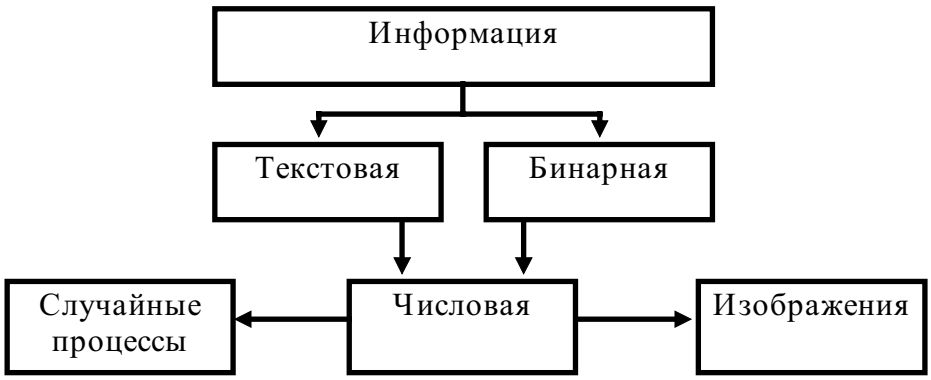


Рис. 2.D Виды информации.

Бинарная информация (исполняемые файлы, программные библиотеки и т.д.) наиболее близко описываются моделью Бернулли и Маркова малых порядков, в то время, как текстовая информация ближе к моделям Маркова высоких

порядков. Частным случаем текстовой или бинарной информации являются числовые данные (в зависимости от того, в каком виде они представлены-ASCII или двоичном). Особое положение среди числовых данных занимают изображения (двумерные числовые массивы, описываемые моделями Маркова невысоких порядков) и аналогичные одномерные массивы, представляющие собой временную последовательность значений какого-нибудь процесса (измерительного сигнала, звукового сопровождения и т.д.). Изображения выделены в особую группу еще и потому, что они любят занимать особенно много памяти, требуют при этом высокой скорости обработки и, к счастью, часто допускают сжатие с потерями.

1. Часто при описании новых разработок программ архивации можно встретить следующий довод: ”Моя программа хорошая, потому что файл, сжатый *моей* программой не смог сжать ни один *признанный* архиватор.” В чем здесь ошибка, иными словами, почему результаты сжатия сначала плохим, а потом хорошим архиватором много хуже, чем, если сделать наоборот? (4)

3. Традиционные методы сжатия информации.

В этой главе описываются методы сжатия информации, условно названные автором *традиционными*. Это не значит что они являются чем то особенным и превосходят или уступают алгоритмам из четвертой главы. Скорее всего, даже наоборот, они могут взаимодополнять друг друга или даже иметь много общего. Просто, рассматриваемые здесь приемы архивации данных или были известны задолго до появления такой науки как “информатика”, или были заимствованы теорией сжатия информации из других точных наук.

3.1 Сжатие текстовой информации.

Специфичный способ, основанный на том, что в текстовом файле используется только ограниченный набор 8-битовых кодов ASCII. Например, для задания всех букв английского алфавита и знаков препинания может хватить всего 64 символов выходного 6-битного алфавита (как сделано в коде EBCDIC). При

этом размер выходного файла составит 75% от входного. Подобный алгоритм применяется при сжатии текстовых файлов в архиваторе *ARJ*. Если допустить сжатие с потерями, можно сделать следующий шаг и заменить пробелы символами табуляции, заменить пары символов <CR> <LF> только на <CR>, выкинуть пустые строки, представлять текст только заглавными буквами, исключить буквы *Й* и первую букву в слове “елка” (на моей клавиатуре она *уже* исключена). Именно с использованием таких приемов работала самая первая известная автору программа-архиватор, исходные коды которой он видел. И она действительно сжимала информацию, хотя в настоящее время перечисленные приемы следует использовать как предварительный проход перед запуском более серьезной программы сжатия.

В качестве разрядки можно предложить интересный метод сжатия информации до 1/3 первоначального объема, предложенный в [44]. Он заключается в том, что из исходного текста выкидываются каждая вторая и третья из трех последовательных букв. Например, после сжатия сообщения ”**Д**аж**е** Э**д**ди на**с** о**п**ере**д**и**л** с де**т**ск**и**м **х**оро**м**.” мы получим строку вида “Дед спел тихо”. Как видно, фраза после такой “обработки” не утратила некоторого смысла.

3.2 Перекодировка.

Под **перекодировкой** мы будем понимать любое удачно выбранное представление исходной информации при котором ее

- легко можно восстановить обратно и
- которое занимает меньший физический объем на носителе.

Любой читатель при желании сможет сочинить множество различных способов перекодировки, поэтому мы приведем лишь некоторые конкретные примеры, дающие общее представление о теме разговора и не лишенные в то же время практического смысла:

общепринятые условные обозначения, такие как π (вместо 3.1415...); 1/3 (вместо 0.3333....); оператор математического ожидания M; обозначения для единичных, диагональных, нулевых и прочих матриц и т.д.

общепринятые сокращения: MS-DOS, FAT, модем и т.д и т.п.

стенография, нотная запись,

запись программы в исходных текстах тоже может рассматриваться как способ архивации: если программа написана на языке высокого уровня, то ее текст обычно меньше размера исполняемого файла и поэтому несколько килобайт кода могут быть представлены одной строкой вида *printf("%d",Num);*. Компилятор в этом случае выполняет роль декомпрессора, а вот написать компрессор-декомпилятор - задача неимоверно более трудная. Следует также помнить, что программа, написанная на языке ассемблера обычно значительно длиннее своего скомпилированного варианта.

знаменательный пример, относящийся к сжатию комбинаторных источников информации: представим, что наш источник с равной вероятностью вырабатывает 6 различных сообщений. Для их кодирования приходит на ум очевидное решение использовать двоичные коды чисел от 0 до 5-000, 001, 010, 011, 100, 101. Так же очевидно, что стоимость кодирования получилась 3 бита, а энтропия несколько меньше ($2 < H_0=\log(6) < 3$). Причину такого явления понять не трудно-у нас остались неиспользованными 2 кода: 110 и 111. Значит надо сделать так, чтобы лишних кодов не было и использовать для этого **коды переменной длины**-

0 - 00 1 - 01 2 - 100 3 - 101 4 - 110 5 - 111.

Стоимость кодирования такого кода равна 2.667, что значительно ближе к энтропии.

3.3 Упаковка.

Упаковка (не путать с упаковкой из раздела 4.1)-процесс, напоминающий укладку чемодана. Если путем изменения местоположения и взаимного расположения носителей информации удается сократить ее объем, то мы и будем считать такой процесс упаковкой. Может быть, у кого-то знак равенства между

упаковкой и архивацией вызовет улыбку, и, тем не менее, это-один из самых старых и по сей день популярных способов сжатия информации, о котором помнят, однако, только специалисты. Он отличается исключительной простотой и высокой скоростью работы. Вот лишь краткий перечень задач, элегантно решаемых с помощью упаковки:

многие знают, что файлы на дисках хранятся по секторам. Размер сектора обычно равен или кратен 512 байтам, размеры же файлов самые разные. Поскольку кластер-минимальная, неделимая единица объема диска, состоит из нескольких секторов , то файлы длиной 2 и 508 байт займут один кластер, а файлы размером в 513 и 1001 байт-уже, может быть, два. В любом случае мы можем потерять до 511 неиспользованных байт.

(еще хуже дело обстоит на современных жестких дисках объемом в сотни МБ с файловой системой FAT под управлением MS-DOS. Так как на таком диске может быть не более 64000 кластеров (округленно), то для 600МБ диска размер кластера составляет 16 кБ!!! Такими же могут быть и потери в расчете на один файл)

Одна из первых программ-архиваторов *MatchBox* объединяла несколько маленьких файлов в один непрерывный и могла при необходимости извлекать их обратно, экономя в среднем по полкластера на файл.

Такую же процедуру выполняет архиватор *RAR* при создании так называемых Solid-архивов, выигрывая на этом несколько процентов в степени сжатия.

Различные менеджеры expanded-памяти (*QEMM и Optimize, EMM386 и Memmaker*). Эти утилиты призваны поместить в области High и Upper Memory MS-DOS как можно больше файлов и для этого они пытаются так расположить (упаковать) программы, чтобы свести к минимуму потери памяти.

3.4 Сжатие числовой информации.

Одним из широко известных способов сжатия последовательности цифровых данных (например результатов измерения процесса $F(t_i)$ в дискретные промежутки времени t_i, $i=1..N$) является **аппроксимация**. Действительно, вместо передачи длинного набора чисел можно передать лишь набор коэффициентов

аппроксимирующего полинома P_j, $j=0..K$. Степень сжатия в этом случае равна **(K+1)/N**.

В качестве аппроксимирующего полинома можно взять один из широко известных в математике (полученных по методу наименьших квадратов-МНК, преобразование Фурье, Хартли) или подобрать свой собственный, наиболее соответствующий задаче. Интерполяционные полиномы (Лагранжа, Ньютона), очевидно, использовать нельзя, т.к. для них количество коэффициентов равно количеству отсчетов процесса (узлов интерполяции). От соотношения параметров **K** и **N** зависит компромисс между степенью сжатия и точностью восстановления, поскольку извлечение информации при таком методе возможно лишь с некоторой погрешностью (потерями), которые тем больше, чем выше степень сжатия.

А как быть, если требуется передать данные без потерь? И здесь предложенный способ может помочь, стоит его лишь несколько расширить [33]:

 Предположим, нам надо передать массив из N чисел, каждое из которых не превышает величины n. При непосредственном представлении для этого потребуется $\mathbf{N}\cdot\lceil\mathbf{\log(n)}\rceil$ бит. Пусть нам удалось подобрать такой аппроксимирующий полином порядка K, что все погрешности (ошибки отклонения истинного значения от его аппроксимированного варианта) укладываются в диапазон значений $k<n$. Тогда, как нетрудно подсчитать, после сжатия длина массива составит $\mathbf{N}\lceil\mathbf{\log(k)}\rceil + \mathbf{(K+1)}\lceil\mathbf{\log(n)}\rceil$**.** При достаточно больших значениях N (N >> K) степень сжатия (в %) будет примерно равна

$$\lceil\mathbf{\log(k)}\rceil / \lceil\mathbf{\log(n)}\rceil.$$

Рассмотренный способ особенно эффективно работает для данных, описываемых моделью Маркова 1 порядка, у которых большая часть условных вероятностей сосредоточена вблизи главной диагонали, т.е. $\mathbf{p_{i,j}}$ тем больше, чем меньше разность между i и j. К таким процессам относятся, например, годовые или суточные колебания температуры. Маловероятно, что, если сегодня температура воздуха -20°C, то завтра будет +10. Скорее ожидать -19 или -21.

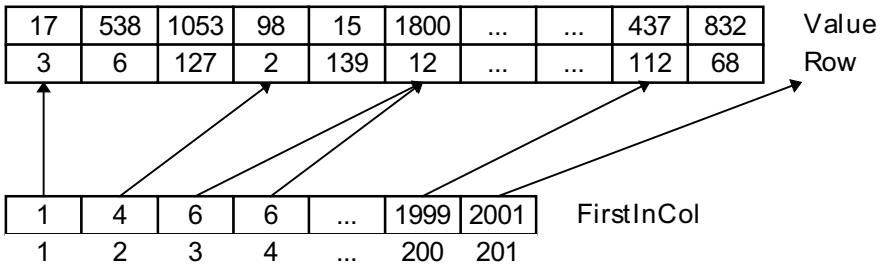
Очень простым способом сжатия с потерями является отбрасывание незначащих разрядов или округление (о чем уже говорилось) или прореживание, т.е. простое отбрасывание некоторого количества результатов (обычно-через один, через два и т.д.). И то и другое, в конечном итоге, является методом снижения высокочастотного шума и сводится к более общему понятию **низкочастотной фильтрации**.

Рассмотрим особо очень часто возникающую задачу сжатия разряженных двумерных массивов (матриц). Один из возможных и изящных подходов изложен в книге [33]:

 Пусть имеется массив размера XxY, верхняя левая часть которого имеет вид:

	Столб.1							Столб.8
Стр. 1								
		98						**162**
	17							
						1121		
				965				
Стр.6	**538**							

Если каждое число представлять двумя байтами, то такая матрица займет в памяти 2XY байт. Нельзя ли сократить этот объем? Вместо массива, оказывается, можно хранить только три одномерных вектора. Назовем их *FirstInCol* (ПервыйВКолонке), *Row* (Ряд) и *Value* (Значение)



Длина вектора *FirstInCol* равна количеству столбцов матрицы, а длины *Row* и *Value*-количеству ее непустых элементов (в данном примере - 2000). **i**-ое число первого вектора показывает, с какой позиции в векторе *Row* начинаются значения **i**-го столбца. Числа в *Row*-номера строк, где находятся соответствующие непустые значения *Value*. Так, второй столбец содержит два числа - 98 во второй

строке и 15 в сто тридцать девятой. При определенных условиях (см. Задачи) с помощью приведенного метода можно получить существенное снижение объема числовой и прочей информации.

1. Какой степени сжатия можно достичь перекодировкой ASCII кодов, если писать только русскими заглавными буквами без знаков припинания? (1)

2. Постройте оптимальный код для комбинаторного источника с десятью состояниями; найдите его энтропию и стоимость кодирования. Каким из сообщений следует присваивать более короткие коды, а каким-более длинные? (2)

3. При какой “плотности заполнения” двумерного массива числами (отношении пустых ячеек к заполненным) еще возможно его сжатие. Решите задачу в общем виде-размер массива *XxY*, максимальное значение чисел в ячейках-*k.* (3)

4. Сжатие информации в системах передачи и хранения.

4.1 RLE-кодирование.

Алгоритм **RLE** (Run Length Encoding, упаковка, кодирование длин серий), предложенный в [14] является самым быстрым, простым и понятным алгоритмом сжатия данных и при этом иногда оказывается весьма эффективным. Именно подобный алгоритм используется для сжатия изображений в файлах *PCX*. Он заключается в следующем: любой последовательности повторяющихся входных символов ставится в соответствие набор из трех выходных символов: первый-байт префикса, говорящий о том, что встретилась входная повторяющаяся последовательность, второй-байт, определяющий длину входной последовательности, третий-сам входной символ-<*prefix,length,symbol*>. Лучше всего работу алгоритма пояснить на конкретном примере.


Например: пусть имеется (шестнадцатиричный) текст вида

05 05 05 05 05 05 01 01 03 03 03 03 03 03 05 03 FF FF FF FF

из 20 байт. Выберем в качестве префикса байт FF. Тогда на выходе архиватора мы получим последовательность

FF 06 05 FF 02 01 FF 06 03 FF 01 05 FF 01 03 FF 04 FF

Ее длина-18 байт, то есть достигнуто некоторое сжатие. Однако, нетрудно заметить, что при кодировании некоторых символов размер выходного кода возрастает (например, вместо 01 01 - FF 02 01). Очевидно, одиночные или дважды (трижды) повторяющиеся символы кодировать не имеет смысла-их надо записывать в явном виде. Получим новую последовательность:

FF 06 05 01 01 FF 06 03 05 03 FF 04 FF

длиной 13 байт. Достигнутая степень сжатия: 13/20 = 65%.

Нетрудно заметить, что префикс может совпасть с одним из входных символов. В этом случае входной символ может быть заменен своим “префиксным” представлением, например:

FF то же самое, что и FF 01 FF (три байта вместо одного).

Поэтому, от правильного выбора префикса зависит качество самого алгоритма сжатия, так как, если бы в нашем исходном тексте часто встречались одиночные символы FF, размер выходного текста мог бы даже превысить входной. В общем, случае в качестве префикса следует выбирать самый редкий символ входного алфавита.

 Можно сделать следующий шаг, повышающий степень сжатия, если объединить префикс и длину в одном байте. Пусть префикс-число F0...FF, где вторая цифра определяет длину *length* от 0 до 15. В этом случае выходной код будет двухбайтным, но мы сужаем диапазон представления длины с 255 до 15 символов и еще более ограничиваем себя в выборе префикса. Выходной же текст для нашего примера будет иметь вид:

F6 05 F2 01 F6 03 05 03 F4 FF

Длина-10 байт, степень сжатия-50%.

Далее, так как последовательность длиной от 0 до 3 мы условились не кодировать, код *length* удобно использовать со смещением на три, то есть 00=3, 0F=18, FF=258, упаковывая за один раз более длинные цепочки.

Если одиночные символы встречаются достаточно редко, хорошей может оказаться модификация алгоритма RLE вообще без префикса, только <*length,symbol*>. В этом случае одиночные символы также *обязательно* должны кодироваться в префиксном виде, чтобы декодировщик мог различить их в выходном потоке:

06 05 02 01 06 03 01 05 01 03 04 FF

Длина-12 байт, степень сжатия-60%.

Возможен вариант алгоритма, когда вместо длины *length* кодируется позиция относительно начала текста *distance* первого символа, отличающегося от предыдущего. Для нашего примера это будет выходная строка вида:

01 05 07 01 09 03 0F 05 10 03 11 FF

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
RLE	2-3	10	10	Да	1	Нет	Возм.

Здесь и далее приняты следующие обозначения: РО-Распространение ошибки, ВИ-Возрастание избыточности. Степень сжатия, скорость и используемая оперативная память оцениваются по десятибалльной системе с точки зрения *автора.* Чем больше величина, тем лучше указанный параметр (выше скорость работы, выше степень сжатия и меньшая потребляемая память).

Работа: в реальном масштабе времени и в потоке.

Основное применение: *PCX*, сжатие изображений.

Термин (в терминологии PkZip): Packing.

Модель: Маркова 1 порядка.

При более внимательном изучении нетрудно заметить, что наилучшего сжатия можно достичь, если данные описываются матрицей условных вероятностей (см. 2.5.2), в которой $p_{i,i} >> p_{i,j}$, т.е. с максимальными значениями на главной диагонали.

Классификация: блочный, инкрементальный или макро.

1. Оцените степень сжатия алгоритма RLE, если все условные вероятности $p_{i,i}$=0.9. (3)

4.2 Унарное кодирование.

Один из очевидных способов сжатия информации, в особенности текстовой, основывается на том, что вероятность использования в тексте разных символов сильно различна. В частности, в русском языке чаще всего используются буквы “о”, “а”, редко-“ъ”,”щ” и т.д. Поэтому, предлагается использовать для частых символов более короткий код, для редких-более длинный. Трудность заключается в том, что коды получаются переменной длины и в общем потоке выходных данных трудно обнаружить конец одного кода и начало другого.

Однако, существуют коды [38], для которых задача определения длины легко решается (так называемые **неравномерные коды со свойством однозначного декодирования**). Например, **унарный** код (в двоичной системе), где:

“о”-1

“а”-01

“и”-001

...

“щ”-00....1

т.е. для самой частой буквы длина кода составляет 1 бит, для самой редкой-33 бита (32 нуля и единица). Этот код очень прост, но далеко не эффективен. Однако, он еще может нам пригодиться для построения **кодов целых переменной длины (Variable Length Integers (VLI) codes).** Именно эти коды с успехом применяются в архиваторах *ARJ* и *LHA*. VLI-коды состоят из двух частей [12]: сначала следует унарный код, определяющий группу чисел определенной длины, а затем само число указанной длины. Таким образом, унарный код выполняет роль префикса, указывающего-какой длины будет следующее число. Для записи VLI-кодов

используют три числа: (*start,step,stop*), где *start* определяет начальную длину числа, *step*-приращение длины для следующей группы, *stop*-конечную длину.

 Например: код (3,2,9) задает четыре группы чисел-

0xxx - числа от 0 до 7, группа 0,

10xxxxx - числа от 8 до 39, группа 1,

110xxxxxxx - числа от 40 до 167, группа 2,

111xxxxxxxxx - числа от 168 до 679, группа 3.

Унарный код для n-ой группы-n единиц, затем ноль; для последней группы-только n единиц, так как это не вносит неопределенности. Размер числа из n-ой группы равен *start*+n·*step*. VLI-коды удобно применять для кодирования монотонных источников, для которых вероятность появления меньших чисел выше, чем больших. При отсутствии кодирования любое число представлялось бы десятью битами.

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
VLI	2-3	9	10	Да	1	Да	Возм.

Работа: в реальном масштабе времени и в потоке.

Модель: Источник Бернулли (монотонный).

Классификация: неблочный, статистический.

1. Как будут выглядеть (1,1,5) VLI-коды? (1)

2. “Изобретите” формулу, по которой можно найти максимальное число в n-ой группе, если известны *start* и *step*. (3)

4.3 Коды Хаффмена.

 Доказано, что наиболее эффективным кодом переменной длины, в котором ни одно слово не совпадает с началом другого (т.е. **префиксный код**) является **код *Хаффмена*** [15].

 Пусть $l_1,...,l_k$-положительные целые числа (k>0). Для того, чтобы существовал префиксный код, длины слов которого равны $l_1,...,l_k$, необходимо и достаточно выполнение **неравенства *Крафта*** [38]**:**

$$\sum_{i=1}^{k} 2^{-li} \le 1$$

Все префиксные коды являются кодами со свойством однозначного декодирования, но не наоборот (например, однозначно декодируемый код 0, 01, 011, 0111, ... не является префиксным). Избыточность дешифрируемого кодирования неотрицательна. Для кода Хаффмена (Шеннона) избыточность не превышает 1, т.е. $\mathbf{0 \le R \le 1}$. Длина кода Шеннона равна

$$|a_i| = \lceil -\log(p(a_i)) \rceil,$$

а длина кода Хаффмена не превышает величины $|a_i|$.

Отсюда, в частности следует вывод, что чем больше длина **T** символов входного алфавита (для которого строится код Хаффмена), тем меньше избыточность выходного текста и тем выше степень сжатия. Однако, как будет видно в дальнейшем, при этом значительно возрастают требования к памяти данных и к быстродействию программы, поскольку количество кодов равно количеству символов исходных букв. Избыточность кода Хаффмена в значительной мере зависит, как следует из приведенной формулы, от того, на сколько вероятности появления символов близки к отрицательным степеням числа 2. Для двухсимвольного алфавита, например, код Хаффмена никогда не сможет дать сокращения избыточности, пусть даже вероятности различаются на несколько порядков.

 Рассмотрим построение кода Хаффмена на простом примере:

пусть есть текст ABACCADAA. При кодировке двоичным кодом постоянной длины вида:

A - 00, B - 01, C - 10, D - 11

мы получим сообщение 000100101000110000 длиной 18 бит. Теперь построим код Хаффмена, используя **дерево Хаффмена**.

Для этого первоначально требуется просмотреть весь исходный текст и получить вероятность (или, что проще-частоту) каждого входящего в него символа. Далее, возьмем два самых редких кода и объединим их в единый узел, частота появления которого равна сумме частот появления входящих в него символов. Рассматривая этот узел, как новый символ, объединяющий два предыдущих, будем повторять операцию слияния символов до тех пор, пока не останется ни одной буквы из входного алфавита. Получим дерево вида рис. 4.а, где листьями являются символы входного алфавита, а узлами-соединение символов из листов-потомков данного узла. Рассматривая каждую левую ветвь как 0, а правую как 1 и спускаясь по дереву вниз до листьев, получим код любого символа:

A - 0, B - 110, C - 10, D - 111

Исходное сообщение после кодирования станет 011001010011100 (15 бит). Степень сжатия: 15/18 = 83%. Легко убедиться, что имея коды Хаффмена, можно однозначно восстановить первоначальный текст (см. Листинг 2 приложения).

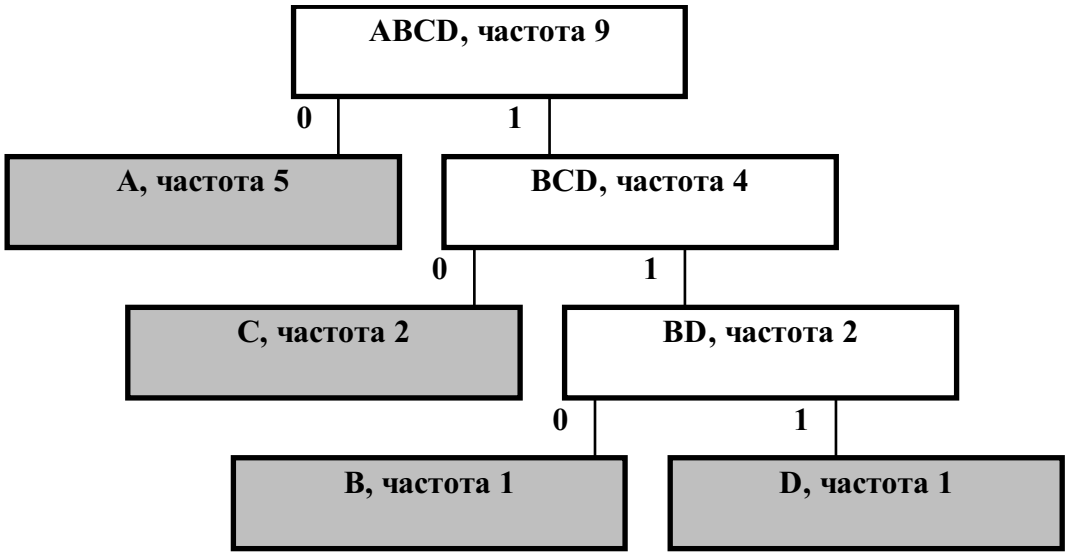


Рис. 4.А Построение дерева Хаффмена.

Другой алгоритм построения оптимальных префиксных кодов, дающий аналогичный результат, был предложен ***Шенноном и Фано*** [36, 47].

Недостатком такого кодирования является тот факт, что вместе с закодированным сообщением необходимо передавать также и построенную таблицу кодов (дерево), что снижает величину сжатия. Однако существует **динамический** алгоритм построения дерева Хаффмена [13], при котором кодовая таблица обновляется самим кодировщиком и (синхронно и аналогично) декодировщиком в процессе работы после получения каждого очередного символа. Получаемые при этом коды оказываются квазиоптимальными, поэтому текст сжимается несколько хуже. Более того, возрастают сложность алгоритма и время работы программы (хотя она и становится однопроходной), поэтому, в настоящее время динамический алгоритм почти полностью уступил место статическому.

Следующим вариантом рассматриваемого алгоритма является **фиксированный** алгоритм Хаффмена, сочетающий в себе достоинства двух предыдущих-высокую скорость работы, простоту и отсутствие дополнительного словаря, создающего излишнюю избыточность. Идея его в том, чтобы пользоваться некоторым усредненным по многим текстам деревом, одинаковым для кодировщика и декодировщика. Такое дерево не надо строить и передавать вместе с текстом, а значит-отпадает необходимость первого прохода. Но, с другой стороны, усредненное фиксированное дерево оказывается еще более неоптимальным, чем динамическое. Поэтому, иногда может быть удобно иметь несколько фиксированных деревьев для разных видов информации.

Вариантами дерева Хаффмена следует также считать деревья, полученные для монотонных источников [38]. Эти источники имеют два важных для наших целей свойства:

- нет необходимости вычислять частоты появления символов входного текста-как следствие, однопроходный алгоритм;
- дерево Хаффмена для таких источников может быть представлено в виде вектора из нескольких байтов, каждый из которых обозначает количество кодов дерева определенной длины (см. Задачи), например: запись 1,1,0,2,4 говорит о том, что в дереве имеется по одному коду длин 1 и 2 бита, 0 кодов длиной 3 бита, 2 кода длиной 4 бита и 4 кода длиной 5 бит. Сумма всех чисел в записи даст количество символов в алфавите.

На рис. 4.b представлены деревья Хаффмена для монотонных источников и для количества символов входного алфавита от 4 до 8.

Сравнительно недавно появилась еще одна разновидность адаптивного алгоритма Хаффмена, описанная ***Рябко***, а затем ***Бентли*** и названная, соответственно, **алгоритмом стопки книг** или **“двигай вверх” (“MTF-Move To Front”)** [5, 38]. Каждому символу (букве) присваивается код в зависимости от его положения в алфавите-чем ближе символ к началу алфавита-тем короче код (в качестве кода можно использовать, например, код дерева для монотонных источников). После кодирования очередного символа мы помещаем его в начало алфавита, сдвигая все остальные буквы на одну позицию вглубь. Через некоторое время наиболее часто встречаемые символы сгруппируются в начале, что и требуется для успешного кодирования. Работа алгоритма, действительно, напоминает перекладывание наиболее нужных книг из глубин библиотеки ближе к верху, вследствие чего потом их будет легче найти (снова аналогия с обыденной жизнью).

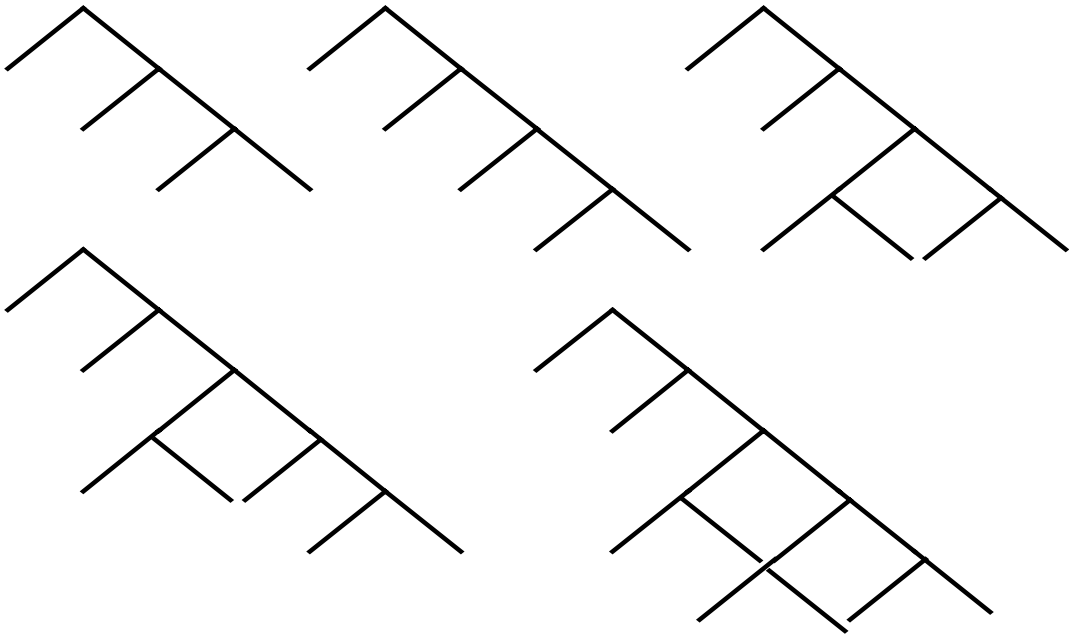


Рис. 4.В Кодовые деревья для монотонных источников с алфавитом от 4 до 8 символов.

Выводы

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
Хаффмен статич.	5-6	8	8	Да	2	Да	Редко
Хаффмен динамич.	4-5	6	7	Да	1	Да	Редко
Хаффмен фиксир.	3-6	9	9	Да	1	Да	Возм.
Стопка книг	3-5	7	8	Да	1	Да	Возм.
Монотон.	3-4	8	9	Да	1	Да	Возм.

Работа: в реальном масштабе времени и в потоке (кроме статического).

Основное применение: универсальный, сжатие текстов и бинарной информации. Динамический алгоритм построения кода Хаффмена используется в архиваторе *ICE*, статический-в *LHA, ARJ*. Статический алгоритм Шеннона-Фано используется архиватором *PKZIP*. Применяется для сжатия изображений (формат *JPEG*).

Термин: Squashing.

Модель: Бернулли.

Классификация: неблочный, статистический.

1. Докажите, что для любого входного текста существует несколько различных кодов Хаффмена. (1)

2. Сформулируйте правила, по которым можно декодировать сообщение, представленное в кодах Хаффмена. (2)

3. Какое из деревьев может являться реальным деревом Хаффмена-1,0,2,2,4; 1,0,2,1,5? (2)

4. Приведите пример текста, который сжимается алгоритмом RLE и не сжимается с помощью кодов Хаффмена (и наоборот). (2)

5. Закодируйте слово ABRACADABRA. (2)

6. Как будет выглядеть дерево Хаффмена для комбинаторных источников? Какова максимальная и минимальная возможная высота дерева из N кодов? Сколько операций слияния кодов надо произвести для получения законченного дерева? (3)

7. Докажите, что используя приведенное выше представление дерева Хаффмена для монотонных источников в виде вектора, можно однозначно восстановить коды Хаффмена. (3)

8. Предложите наиболее компактный способ представления дерева Хаффмена (без использования алгоритмов архивации), если известно, что длина кода не превышает 16 бит. (4)

4.4 Арифметическое кодирование

Арифметическое кодирование [2, 24, 37] является методом, позволяющим упаковывать символы входного алфавита без потерь при условии, что известно распределение частот этих символов. Арифметическое кодирование является оптимальным, достигая теоретической границы степени сжатия $\mathbf{H_0}$**.** Однако, коды Хаффмена в предыдущем разделе мы тоже называли оптимальными. Как объяснить этот парадокс? Ответ заключается в следующем: коды Хаффмена являются неблочными, т.е каждой *букве* входного алфавита ставится в соответствие определенный выходной код. Арифметическое кодирование-блочное и выходной код является уникальным для каждого из возможных входных *сообщений*; его нельзя разбить на коды отдельных символов.

Текст, сжатый арифметическим кодером, рассматривается как некоторая двоичная дробь из интервала [0, 1). Результат сжатия можно представить как последовательность двоичных цифр из записи этой дроби. Каждый символ исходного текста представляется отрезком на числовой оси с длиной, равной вероятности его появления и началом, совпадающим с концом отрезка символа,

предшествующего ему в алфавите. Сумма всех отрезков, очевидно должна равняться единице. Если рассматривать на каждом шаге текущий интервал как целое, то каждый вновь поступивший входной символ “вырезает” из него подинтервал пропорционально своей длине и положению (см рис. 4.c).

Поясним работу метода на примере:

 Пусть алфавит состоит из двух символов: “a” и “b” с вероятностями соответственно 3/4 и 1/4.

Рассмотрим (открытый справа) интервал [0, 1). Разобьем его на части, длина которых пропорциональна вероятностям символов. В нашем случае это [0, 3/4) и [3/4, 1). Суть алгоритма в следующем: каждому слову во входном алфавите соответствует некоторый подинтервал из [0, 1). Пустому слову соответствует весь интервал [0, 1). После получения каждого очередного символа арифметический кодер уменьшает интервал, выбирая ту его часть, которая соответствует вновь поступившему символу. Кодом сообщения является интервал, выделенный после обработки всех его символов, точнее, *число минимальной длины*, входящее в этот интервал. Длина полученного интервала пропорциональна вероятности появления кодируемого текста.

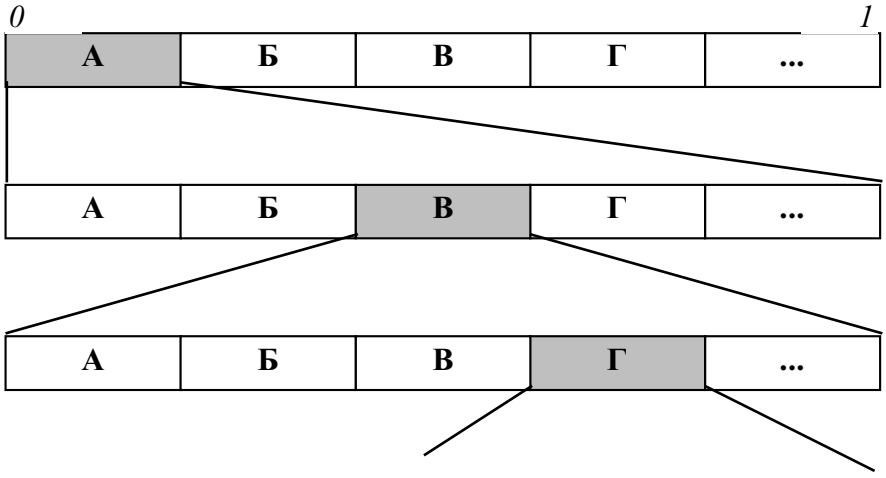


Рис. 4.C Построение интервала для сообщения “АВГ...”.

Выполним алгоритм для цепочки “aaba” (см. таблица 4.a):

Таблица 4.А Построение арифметического кода.

Шаг	Просмотренная цепочка	Интервал
0.	“”	[0, 1) = [0, 1)
1.	“a”	[0, 3/4) = [0, 0.11)
2.	“aa”	[0, 9/16) = [0, 0.1001)
3.	“aab”	[27/64, 36/64) = [0.011011, 0.100100)
4.	“aaba”	[108/256, 135/256) = [0.01101100, 0.10000111)

На первом шаге мы берем первые 3/4 интервала, соответствующие символу “a”, затем оставляем от него еще только 3/4. После третьего шага от предыдущего интервала останется его правая четверть в соответствии с положением и вероятностью символа ”b”. И, наконец, на четвертом шаге мы оставляем лишь первые 3/4 от результата. Это и есть интервал, которому принадлежит исходное сообщение.

В качестве кода можно взять любое число из диапазона, полученного на шаге 4. Внимательный читатель, которому эта книга досталась не в нагрузку к телевизору, возможно спросит: “А где же здесь сжатие? Исходный текст можно было закодировать четырьмя битами, а мы получили восьмибитный интервал.” Дело в том, что в качестве кода мы можем выбрать, например, самый короткий код из интервала, равный 0.1 и получить четырехкратное сокращение объема текста. Для сравнения, код Хаффмена не смог бы сжать подобное сообщение, однако, на практике выигрыш обычно невелик и предпочтение отдается более простому и быстрому алгоритму из предыдущего раздела.

Арифметический декодер работает синхронно с кодером: начав с интервала [0, 1), он последовательно определяет символы входной цепочки. В частности, в нашем случае он вначале разделит (пропорционально частотам символов) интервал [0, 1) на [0, 0.11) и [0.11, 1). Поскольку число 0.1 (переданный кодером код цепочки “aaba”) находится в первом из них, можно получить первый символ: “a”. Затем делим первый подинтервал [0, 0.11) на [0, 0.1001) и [0.1001, 0.1100) (пропорционально частотам символов). Опять выбираем первый, так как 0

< 0.1 < 0.1001. Продолжая этот процесс, мы однозначно декодируем все четыре символа.

 При рассмотрении этого метода возникают две проблемы: во-первых, необходима вещественная арифметика, вообще говоря, неограниченной точности, и во-вторых, результат кодирования становится известен лишь при окончании входного потока. Дальнейшие исследования, однако, показали, что можно практически без потерь обойтись целочисленной арифметикой небольшой точности (16-32 разряда), а также добиться инкрементальной работы алгоритма: цифры кода могут выдаваться последовательно по мере чтения входного потока [24].

 Подобно алгоритму Хаффмена, арифметический кодер также является двупроходным и требует передачи вместе с закодированным текстом еще и таблицы частот символов. Вообще, эти алгоритмы очень похожи и могут легко взаимозаменяться. Следовательно, существует адаптивный алгоритм арифметического кодирования со всеми вытекающими достоинствами и недостатками. Его основное отличие от статического в том, что новые интервалы вероятности перерассчитываются после получения каждого следующего символа из входного потока.

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
Арифмет. статич.	5-6	8	8	Да	2	Да	Редко
Арифмет. динамич.	4-5	6	7	Да	1	Да	Редко

Работа: в реальном масштабе времени и в потоке (кроме статического).

Основное применение: универсальный, сжатие текстов. Используется в архиваторе *LZARI*, для сжатия изображений (*JPEG*).

Модель: Бернулли.

Классификация: блочный, статистический.

1. Получите коды для остальных возможных сообщений из рассмотренного примера: “aaab”, “abaa”, “baaa”. (2)

2. Предложите алгоритм нахождения самого короткого кода, принадлежащего интервалу. (3)

4.5 Вероятностное сжатие.

 Чрезвычайно интересный и самый быстрый из известных (наряду с RLE) методов сжатия информации, дающий, к тому же неплохие результаты. **Вероятностное сжатие** [26] во многом напоминает гадание на кофейной гуще или предсказание погоды, только более эффективно (см. Листинг 3 приложения, программа *FIN*).

Работает алгоритм следующим образом: имеется достаточно большая, динамически обновляющаяся **таблица предсказаний**, в которой для каждой возможной пары последовательных входных символов указывается предсказываемый следующий (третий) символ. Если символ предсказан правильно-генерируется код в виде однобитного префикса, равного 1. Если же мы не угадали-выдается код в виде префикса, равного 0 и неугаданного символа. При этом неугаданный символ замещает в таблице бывший там до этого, обеспечивая постоянное обновление статистической информации.

Внимательный читатель уже, наверное заметил, что алгоритм использует модель сигнала, являющуюся моделью Маркова 2 порядка. Алгоритм является адаптивным и поэтому он однопроходный и не требует хранения таблицы совместно с сжатым текстом.

Декомпрессор работает по аналогичной программе, поддерживая и обновляя таблицу синхронно с компрессором. Если поступил префикс 1, очередная выходная буква извлекается из таблицы по индексу, полученному из двух

предыдущих букв, иначе-просто копируется код из входного потока в выходной. Для небольшого повышения степени сжатия рекомендуется инициализировать таблицу перед началом работы какими-нибудь часто встречающимися буквосочетаниями.

Похожий алгоритм, основанный на модели Маркова 1 порядка, использовался в архиваторе *PKZIP Ver.1.5* под именем Reducing [21]. Он является почти по всем параметрам хуже, чем *Fin* (двупроходный, требует передачи таблицы вместе с текстом), но тем не менее достоин упоминания. Таблица программы Reducing содержит массив символов **V[256][32]**, т.е. для каждой входной буквы на первом проходе находятся не более, чем 32 (но может быть и меньше) наиболее часто встречающихся следующих буквы**.** На втором проходе дело обстоит сходно с *Fin*-если очередной символ *b*, следующий за символом *a* находится среди 32 ожидаемых, генерируется код *<prefix,position>, prefix=1, position*=положению символа *b* в массиве **V[*a*][ ]** из 32 букв. Иначе выдается префикс 0 и сам символ *b*. Алгоритм статический, поэтому таблица не обновляется в процессе работы.

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
Reducing	4-5	8	7	Да	2	Да	Редко.
Вероятн.	5-6	9	6	Да	1	Да	Возм.

Работа: в реальном масштабе времени и в потоке.

Основное применение: универальный.

Термин: Reducing.

Модель: Маркова 1 или 2 порядка.

Наилучшего сжатия можно достичь, если данные описываются матрицей условных вероятностей (см. 2.5.2), для которой в каждом столбце имеется один (или несколько) явно выраженных максимумов, т.е. определенные переходы из состояния в состояние наиболее вероятны.

Классификация: неблочный, статистический.

1. Как зависит объем таблицы предсказаний от модели сигнала (1, 2 или 3 порядка)? (1)

2. Оцените предельную степень сжатия алгоритма *FIN.* (2)

3. Как модифицировать описанный вариант программы Reducing, чтобы повысить ее степень сжатия? Как оценить для нее максимально достижимую степень сжатия? (3)

4.6 Алгоритм LZ77.

 Самый эффективный алгоритм сжатия информации, являющийся одновременно и очень простым для понимания и очень сложным в реализации, был предложен ***Лемпелом*** и ***Зивом*** (***Lempel, Ziv***) в 1977 г. (**LZ77**).

Заключается он в следующем: имеется **"скользящий" (*sliding*) словарь V** объемом **|V|**=2...32 кБ. Если очередная входная **строка s** текста совпадает со строкой из словаря, то она заменяется на **указатель ptr** на эту строку вида **ptr**=<*prefix,distance,length*>, для которой *prefix*=1 [12, 20, 21, 25, 31], после чего текущая позиция в исходном тексте *pos* сдвигается на *length* символов дальше. Понятие словаря и строки несколько отличается от обыденного: под строкой подразумевается любая непрерывная последовательность символов (букв), начинающаяся с определенной позиции в словаре или в тексте, длиной $|\mathbf{s}| \le \mathbf{s}_{max}$. $\mathbf{s}_{max}$ обычно выбирается из диапазона 16...256 байт; в качестве словаря используются **|V|** байт сжимаемого текста, *предшествующие* текущему кодируемому символу в позиции *pos* (отсюда и название "скользящий"-словарь как бы скользит вдоль по тексту от его начала к концу, поддерживая самую свежую информацию о его содержании). Поскольку, перед началом процесса компрессии перед первым символом ничего нет, первоначально словарь пуст (или заполнен каким-нибудь определенным символом) (см. рис. 4.d a).

Словами в словаре выступают любые строки длины не более $\mathbf{s}_{max}$, начинающиеся в любой позиции словаря. Таким образом, в словаре всегда присутствует $|\mathbf{V}| \cdot \mathbf{s}_{max}$ слов. Если же строки **s** в словаре не оказалось, генерируется код **chr** вида **chr**=<*prefix,symbol*>, где *prefix*=0, а *symbol*-текущий символ исходного

текста (в позиции *pos*). Префикс, как видно, нужен для того, чтобы отличить код **ptr** от кода **chr**. Именно префикс вносит в алгоритм кодирования LZ77 некоторую дополнительную информацию, из-за которой возможно возрастание избыточности.

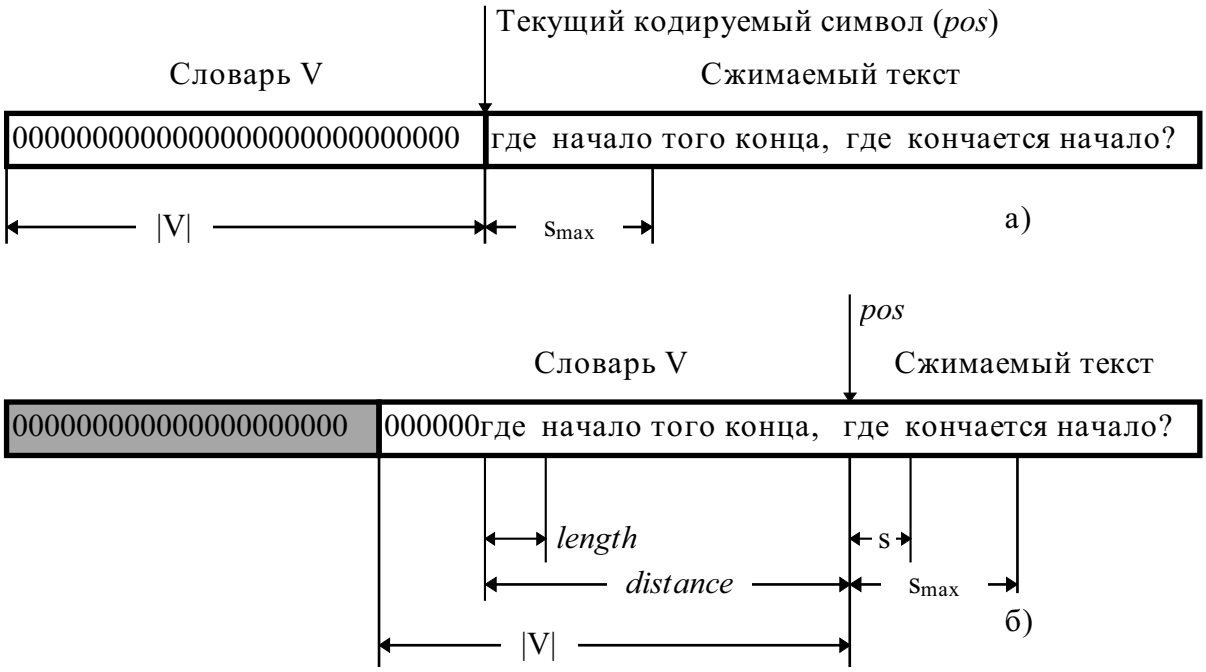


Рис. 4.D Вид словаря и исходного текста для алгоритма LZ77 перед началом работы а) и на 23-ем шаге б).

На рис. 4.d б) показан вид словаря и кодируемого текста на 23-ем символе, когда алгоритм обнаружил совпадение слова “где “ с таким же словом в словаре длины *length*=4 байта, на расстоянии *distance*=23 байта от *pos*. Будет сгенерирован код **ptr**=<1,23,4>. Длина **ptr**-кода

$$|\mathbf{ptr}| = 1 + \lceil \log(|V|) \rceil + \lceil \log(|s_{max}|) \rceil \text{ (бит)},$$

chr-кода

$$|\mathbf{chr}| = 1 + \lceil \log(|symbol|) \rceil \text{ (бит)}.$$

Отсюда следует вывод, что, если длина **ptr**-кода в байтах превышает *length*, то такое кодирование лишь увеличит избыточность, поэтому в алгоритм

вводят **порог (threshold)**, равный обычно 2-3 байтам и, если совпадающая строка короче этого порога, она не кодируется (вернее, кодируется явно, посимвольно, посредством **chr**-кодов).

 Реализация алгоритма построения словаря и поиска в словаре сами по себе уже достаточно сложные и медленные процедуры, поэтому в данной книге не рассматриваются (достаточно заметить, что в простейшем варианте для поиска используется линейный просмотр, требующий в худшем случае до $|\mathbf{V}|\cdot\mathbf{s_{max}}$ операций сравнения на *каждый входной символ!* (см. Листинг 4 приложения)*;* **бинарные деревья** (алгоритм **LZSS**) и **TRIE**-структуры [3, 12, 39] позволяют на несколько порядков ускорить работу архиватора и усложнить написание и отладку программы). Этим объясняется то, что до 1987 г., когда ***Беллом*** [4] был осуществлен алгоритмический прорыв, метод LZ77 не находил практического применения. (Забегая вперед, заметим что подобное “забвение” на шесть лет постигло и другой алгоритм - **LZ78**).

Неплохой альтернативой последовательному поиску может служить использование **множественных двусвязных списков** [39]-несложной структуры данных, которая позволяет ускорить поиск более чем в 10 раз. Она, конечно, существенно уступает в скорости бинарным деревьям, но значительно проще в реализации. Идея метода заключается в том, что создается 256 (или более) двусвязных списков, в каждый из которых включаются слова, начинающиеся на одинаковую букву (или 1.5-2 буквы). Поиск очередного слова осуществляется, очевидно, только в том списке, слова которого начинаются на ту же букву, что и первая буква текущего слова *symbol*.

Обратное восстановление текста осуществляется значительно быстрее и проще. Поскольку к моменту декодирования очередного символа весь предыдущий текст уже известен, декодировщику не требуется передавать словарь вместе с упакованными данными. Словарь строится декомпрессором по тому же алгоритму, что и компрессором, на основе уже полученных символов. Если декодировщик обнаруживает (по префиксу), что поступил **chr**-код, он просто копирует *symbol* в выходной поток и в словарь, передвигая затем словарь на одну позицию вслед за текстом. Если же поступил **ptr**-код, декодировщик копирует *length* символов, начиная с позиции *distance* в словаре. Напомним, что *distance* отсчитывается от конца словаря к началу.

 Все характеристики “классического” варианта алгоритма почти целиком определяются размером словаря **|V|** и максимальной длиной строк в словаре $\mathbf{s_{max}}$ (см. Задачи).

Высокоэффективным оказывается способ **двухступенчатого кодирования** информации, когда выходной поток кодировщика LZ77 поступает на вход блока арифметического кодера (*LZARI*) или кодера Хаффмена (*LZHUF*).

Такое возможно, т.к. коды *length* и *distance* распределены далеко не случайно и неплохо описываются моделью для монотонных источников. Действительно, более длинные совпадения встречаются явно реже коротких, а вероятность совпадения слов на небольшом расстоянии выше, чем вероятность встретить похожее слово в “глубинах” словаря. Оставшиеся после первого прохода коды *symbol* по-прежнему подчиняются модели Бернулли. Подобное решение, используемое в архиваторах *LHA, ARJ, PKZIP, RAR* делает их одними из самых эффективных и популярных в настоящий момент.

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
LZ77 LZSS	9	3	2	Да	1	Да	Редко
LZHUF LZARI	10	2	2	Да	2	Да	Редко

Работа: в реальном масштабе времени и в потоке (кроме *LZARI, LZHUF*).

Основное применение: универсальный, сжатие текстов (*LHA, ICE*, *ARJ, PKZIP, PAK, LZEXE*).

Термин: Imploding, Deflating.

Модель: Маркова высоких порядков (плюс Бернулли для *LZARI, LZHUF*).

Классификация: блочный, макро (плюс статистический для *LZARI, LZHUF*).

1. Каким символом вы бы порекомендовали первоначально заполнять словарь? (1)

2. Обоснуйте-из каких соображений выбирается величина порога. (2)

3. Если найдено несколько совпадающих строк одинаковой длины-какую из них следует выбрать? (2)

4. Проанализируйте-как зависят различные характеристики архиватора от объема словаря и максимальной длины совпадающей строки. (3)

5. Приведите пример данных, которые бы сжимались алгоритмом Хаффмена, но не сжимались бы программой LZ77. (3)

6. “Изобретите” реализацию алгоритма LZ77, которая является значительно быстрее “классического” метода. Подсказка: посмотрите на тексты, упакованные программой *LZEXE* или *PKLITE*. (5)

7. Модифицируйте листинг программы LZ77, приведенный в приложении, чтобы она использовала двусвязные списки. Оцените, на сколько возросло быстродействие программы. От чего оно в основном теперь зависит? (5)

4.7 Алгоритм LZ78-LZW84.

Алгоритм **LZ78**, предложенный в 1978 г. Лемпелом и Зивом [30], нашел свое практическое применение только после реализации **LZW84** [27], предложенной ***Велчем*** (***Welch***) в 1984 г. По мнению автора-этот способ сжатия является наиболее оптимальным сочетанием втепени сжатия и быстродействия. Вероятно, поэтому ему посвящено значительное количество публикаций, в т.ч. и по аппаратному внедрению [1, 6, 22, 26, 37, 40].

В отличие от предыдущего (LZ77), в данном алгоритме понятия строки и словаря несколько изменены. **Словарь** является **расширяющимся** (expanding). Первоначально в нем содержится только 256 строк длиной в одну букву-все коды ASCII. В процессе работы словарь разрастается до своего максимального объема $|\mathbf{V_{max}}|$ строк (слов). Обычно, объем словаря достигает нескольких десятков тысяч слов. Каждая строка в словаре имеет свою известную длину и этим похожа на привычные нам книжные словари и отличается от строк LZ77, которые допускали использование подстрок. Таким образом, количество слов в словаре точно равно его текущему объему. В процессе работы словарь пополняется по следующему закону:

1. В словаре ищется слово *str*, максимально совпадающее с текущим кодируемым словом в позиции *pos* исходного текста. Так как словарь первоначально не пустой, такое слово всегда найдется;

2. В выходной файл помещается номер найденного слова в словаре *position* и следующий символ из входного текста *B* (на котором обнаружилось различие)-<*position,B*>. Длина кода равна $|\boldsymbol{position}|+|\boldsymbol{B}|=\lceil \mathbf{logVmax} \rceil+\mathbf{8}$ (бит);

3. Если словарь еще не полон, новая строка *strB* добавляется в словарь по адресу *last_position*, размер словаря возрастает на одну позицию;

4. Указатель в исходном тексте *pos* смещается на $|\boldsymbol{strB}|=|\boldsymbol{str}|+\mathbf{1}$ байт дальше-к символу, следующему за *B*.

 В таком варианте алгоритм почти не нашел практического применения и был значительно модернизирован. Изменения коснулись принципов управления словарем (его расширения и обновления) и способа формирования выходного кода:

- так как словарь увеличивается постепенно и одинаково для кодировщика и декодировщика, для кодирования позиции нет необходимости использовать $\lceil \mathbf{logV_{max}} \rceil$ бит, а можно брать лишь $\lceil \mathbf{logV} \rceil$ бит, где **V**-*текущий* объем словаря.

 самая серьезная проблема LZ78-переполнение словаря: если словарь полностью заполнен, прекращается его обновление и процесс сжатия может быть заметно ухудшен (метод **FREEZE**). Отсюда следует вывод-словарь нужно иногда обновлять, но когда и как значительно? Этому вопросу посвящено множество публикаций [22 и обзор литературы]. Самый простой способ-как только словарь заполнился-его полностью обновляют. Недостаток очевиден-кодирование начинается на пустом месте, как бы с начала, и пока словарь не накопится-сжатие будет незначительным, а дальше-замкнутый цикл-опять очистка словаря!.. Поэтому предлагается словарь обновлять не сразу после его заполнения, а только после того, как степень сжатия начала падать (метод **FLUSH**). Более сложные алгоритмы используют два словаря, которые заполняются синхронно, но с задержкой на $|\mathbf{V}|/\mathbf{2}$ слов один относительно другого. После заполнения одного словаря, он очищается,

а работа переключается на другой (метод **SWAP**). Еще более сложными являются эвристические методы обновления словарей в зависимости от частоты использования тех или иных слов (**LRU, TAG**).

выходной код также формируется несколько иначе (сравните с предыдущим описанием):

1. В словаре ищется слово *str*, максимально совпадающее с текущим кодируемым словом в позиции *pos* исходного текста.;

2. В выходной файл помещается номер найденного слова в словаре <*position*>. Длина кода равна $|\boldsymbol{position}|=\lceil \mathbf{logV} \rceil$ (бит);

3. Если словарь еще не полон, новая строка *strB* добавляется в словарь по адресу *last_position*, размер словаря возрастает на одну позицию;

4. Указатель в исходном тексте *pos* смещается на |*str*| байт дальше-к символу *B*.

На псевдокоде это будет выглядеть следующим образом:

Алгоритм LZW на псевдокоде.

```
ROUTINE LZW_COMPRESS
STRING=входной символ
WHILE входной текст еще не закончился DO
   B=следующий входной символ
   IF STRING+B в словаре THEN
      STRING=STRING+B
   ELSE
      выдать в выходной файл код position для STRING
      добавить STRING+B в словарь по адресу last_position
      STRING=B
   END of IF
END of WHILE
выдать в выходной файл код position для STRING

ROUTINE LZW_DECOMPRESS
ввести из входного файла position в OLD_CODE
записать в выходной файл OLD_CODE
WHILE входной текст еще не закончился DO
```

```
   прочитать из входного файла position в NEW_CODE
IF NEW_CODE отсутствует в словаре THEN
/* Внимание! Очень важный момент. */
      STRING=строка в словаре по адресу OLD_CODE
      STRING=STRING+B
   ELSE
      STRING=строка в словаре по адресу NEW_CODE
   END of IF
   записать в файл STRING
/* Внимание! Очень важный момент. */
   B=первый символ в STRING
   добавить OLD_CODE+B в словарь по адресу last_position
   OLD_CODE=NEW_CODE
END of WHILE
```

В таком варианте словарь декодировщика отстает на один “шаг” от словаря кодировщика, т.к. кодировщик сразу добавляет *strB* в словарь, а декодировщик-только после того, как получит следующий код и по нему вычислит символ *B*, являющийся первым символом следующего слова (почему?). Из-за этого отставания иногда может возникать небольшая сложность: предположим, что в словаре уже есть слово *Bstr*, а слова *BstrB* еще нет и нам нужно сжать строку вида *BstrBstrB*. На первом шаге мы найдем код строки *Bstr* и поместим в словарь слово *BstrB*. На втором шаге мы передаем только что полученный новый код слова *BstrB*, но декодировщику этот код *еще не известен!* Поэтому в программу введена проверка на существование нового кода в словаре, и если он отсутствует-строка формируется из предыдущей строки и ее первого символа. В остальном работа декодировщика очевидна и похожа на алгоритм LZ77.

 Для лучшего понимания работы достаточно сложного на первый взгляд архиватора LZ78-LZW84 приведем небольшой пример [18]. Пусть алфавит состоит только из четырех букв A, B, C, D и нам надо сжать текст вида:

A B A B A B A B B B A B A B A A C D A C D A D C A B A A A B A B ...

Получим после кодирования собщение:

A B 6 8 B 10 9 A A C D 14 16 D C 8 ...

Входной текст (*pos*)	*last_position*	*str*	*strB*	Выходной Текст (*position*)
A	0			
B	1			
C	2			
D	3			
Clear	4	Специальный код очистки словаря		
End	5	Специальный код конца текста		
A	6	A	AB	A
B	7	B	BA	B
A	8	AB	ABA	6
B				
A	9	ABA	ABAB	8
B				
A				
B	10	B	BB	B
B	11	BB	BBA	10
B				
A	12	ABAB	ABABA	9
B				
A				
B				
A	13	A	AA	A
A	14	A	AC	A
C	15	C	CD	C
D	16	D	DA	D
A	17	AC	ACD	14
C				
D	18	DA	DAD	16
A				
D	19	D	DC	D
C	20	C	CA	C
A	21	ABA	ABAA	8
B				
A				
A	22	AA	AAB	13

A

B 23 BA BAB 7

A

B

Для тех, кто заинтересуется вопросами организации словаря и поиска в нем отметим два важных момента: для любой строки *strB*, находящейся в словаре, в словаре существует также и строка *str*, “укороченная” справа на один символ, поэтому нет необходимости хранить все слова, а нужен только указатель на строку-предка (*position*) и последняя буква *B*; процедура поиска в словаре осуществляется чаще всего с использованием алгоритмов хеширования [26, 39].

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
LZ78 LZW84	6-8	6-7	5	Да	1	Да	Редко

Работа: в реальном масштабе времени и в потоке.

Основное применение: универсальный, сжатие текстов, изображений (*GIF, PKZIP, Windows API*), микросхемы аппаратных компрессоров.

Термин: Shrinking, Crunching.

Модель: Маркова высоких порядков.

Классификация: блочный, макро.

1. Декодируйте обратно текст из рассмотренного примера. (3)

Напишите программу, реализующую псевдокод, приведенный в этом разделе, добавив в нее метод FLUSH обновления словаря. (5)

4.8 BWT-преобразование и компрессор.

BWT-компрессор (***Burrows-Wheeler*** Transform)-сравнительно новая и революционная техника для сжатия информации (в особенности-текстов), основанная на преобразовании, открытом в 1983 г.!!! и описанная в 1994 г. [8]. BWT является удивительным алгоритмом. Во-первых, необычно само преобразование, открытое в научной области, далекой от архиваторов. Во-вторых,

даже зная BWT, не совсем ясно, как его применить к сжатию информации. В-третьих, BW преобразование чрезвычайно просто. И, наконец, сам BWT компрессор состоит из “магической” последовательности нескольких рассмотренных ранее алгоритмов и требует, поэтому, для своей реализации самых разнообразных программных навыков.

BWT не сжимает данные, но преобразует блок данных в формат, исключительно подходящий для компрессии. Он делает это настолько хорошо, что даже его “демонстрационная” версия конкурирует с программами уровня PkZip [19]. Рассмотрим его работу на упрощенном примере. Пусть имеется словарь **V** из **N** символов. Циклически переставляя символы в словаре влево, можно получить **N** различных строк длиной **N** каждая. В нашем примере словарь-это слово **V**=“БАРАБАН” и **N**=7. Отсортируем эти строки лексикографически и запишем одну под другой:

F	**L**
АБАНБАР	
АНБАРАБ	
АРАБАНБ	
БАНБАРА	
БАРАБАН	
НБАРАБА	
РАБАНБА	

Далее нас будут интересовать только первый столбец **F** и последний столбец **L**. Оба они содержат все те же символы, что и исходная строка (словарь). Причем, в столбце **F** они отсортированы, а каждый символ из **L** является префиксом для соответствующего символа из **F**.

Фактический “выход” преобразования состоит из строки **L**=“РББАНАА” и первичного индекса **I**, показывающего, какой символ из **L** является действительным первым символом словаря **V** (в нашем случае **I**=2). Можно доказать, что зная **L** и **I** можно восстановить строку **V**, пользуясь приведенной ниже функцией.

```
int T[]={3,5,6,1,2,4,0};
```

```
char V[7];
char L[]="РББАНАА";
int primary_index=2;

void decode()
 {
 int index=primary_index;
 for( int i=0; i < 7; i++ )
   { V[i]=L[index]; index=T[index]; }
 }
```

Необходимый для работы функции вектор **T**-массив индексов, показывающий для каждой строки **F[i]** из столбца **F**, где находится строка, полученная сдвигом **F[i]** на один символ влево. Его можно легко найти, зная **F** и **L**. А **F**, в свою очередь, можно получить, отсортировав **L**!!!

F	**L**	**T**
А	Р	3
А	Б	5
А	Б	6
Б	А	1
Б	Н	2
Н	А	4
Р	А	0

На первый взгляд кажется непонятным: чем же строка **L** лучше исходной? Тем не менее, можно доказать, что **L** оказывается более упорядоченной. Рассмотрим в качестве примера лишь небольшую часть отсортированного массива строк, полученного в результате обработки реального английского текста.

LF

t: hat acts like this:

t: hat buffer to the constructor

t: hat corrupted the heap, or wo

W: hat goes up must come down

t: hat happens, it isn't likely

w: hat if you want to dynamicall

t: hat indicates an error.

t: hat it removes arguments from

t: hat looks like this:

t: hat looks something like this

t: hat looks something like this

t: hat once I detect the mangled

Эта часть распечатки содержит группу последовательных отсортированных строк, причем все они начинаются словом "hat". Не удивительно, что предыдущая буква почти для всех этих строк-'t' (пару раз встречается ‘w’- **имеются в виду английские слова ‘that’ и ‘what’**).

Можно взять выход BWT и применить к нему обычный компрессор, но авторы предлагают улучшенный подход. Они рекомендуют применить MTF-алгоритм (см. раздел 4.3), после которого использовать энтропийный кодировщик (например, арифметический).

MTF-кодер работает очень просто. Первоначально он имеет список всех 256 букв алфавита. Каждый раз, когда должен быть записан очередной символ, выводится его индекс в списке, после чего этот символ помещается в начало списка. В предыдущем примере для последовательности "tttWtwttttt" будет сгенерирован выходной файл вида { 116, 0, 0, 88, 1, 119, 1, 0, 0, 0, 0 ,0 }.

В результате, BWT-компрессор реализует следующую последовательность преобразований:

input-file --> RLE --> BWT --> MTF --> RLE --> ARI --> output-file

Очевидно, BWT-декомпрессор должен проследовательно вызывать соответствующие декомпрессоры, но уже в обратном порядке.

 **Выводы**

Алгоритм	Степень сжатия	Ско-рость	Память	Сжатие без потерь	Про-ходы	РО	ВИ
BWT	9-10	1	3	Да	>1	Да	Редко

Работа: НЕ в потоке.

Основное применение: универсальный, сжатие текстов, изображений.

Модель: Маркова высокого порядка порядка плюс Бернулли.

Классификация: блочный плюс статистический.

1. Декодируйте обратно слово “OBRSDDB” с индексом 5. (2)

2. Докажите обратимость BWT. (5)

3. Опишите процедуру, которую вы бы использовали для сортировки строк. (3)

5. Заключение

Рассмотрением алгоритма LZ78-LZW84 заканчивается эта книга, но не исчерпывается многообразие методов и алгоритмов архивации. Рассмотрены лишь самые простые, самые широко используемые и самые фундаментальные способы сжатия информации. К сожалению, за рамками книги остались очень эффективные и оригинальные методы, такие как:

- для архивации изображений-**фрактальное сжатие**, разностные алгоритмы сжатия (**DCA, DIC**), блочное сжатие с округлением (**BTC**), **JPEG**, **MPEG** (два последних являются по сути дела комбинацией уже рассмотренных алгоритмов-аппроксимации, округления, RLE, Хаффмена и арифметического);
- кодировщики, рассматривающие контекст как модель Маркова высоких порядков (до 5-12-го): **DMC** [11], **PPM** [9, 10, 17], **LZP** [7], ассоциативное кодирование **ACB** [34];
- для сжатия аудиоинформации и речи-GSM **RPE-LTP транскодер, NYB1, ACOMP.**

Досконально освоение современных реализаций сверхоптимизи-рованных алгоритмов (в особенности арифметического, LZ, JPEG) невозможно без профессиональных навыков программирования, знания архитектуры компьютера, глубокого понимания различных структур данных и основательной математической подготовки. Но не следует забывать, что данная книга-лишь *азбука* архивации, которая должна позволить читателю свободно ориентироваться в этой области, грамотно формулировать свои идеи, заложить фундамент из понятий, терминов и определений и дать (при наличии определенного желания)

толчок к самостоятельному образованию и созданию, в конце концов, конкурентноспособных и оригинальных программ сжатия данных. Более того, автор надеется, что это заключение станет отправной точкой более глубокой и подробной книги.

Ваши отзывы, пожелания и рекомендации Вы можете посылать по адресу: fominandy@hotmail.com

6. Ответы и решения

Раздел 2.4.3 2. Пропускная способность канала скорее всего даже снизится, т.к. повторное сжатие (особенно, после применения хорошего архиватора) приводит, обычно, к возрастанию избыточности. Поэтому, многие протоколы связи предусматривают адаптивное отключение сжатия информации, если оно ухудшает характеристики связи.

Раздел 3.4 2. Например, такой код:

0-000, 1-001, 2-010, 3-011, 4-100, 5-101, 6-1100, 7-1101, 8-1110, 9-1111

Раздел 4.1 1. Грубо степень сжатия можно оценить следующим образом: Т.к. вероятность равна 0.9, в среднем в тексте содержатся цепочки повторяющихся символов длиной в 9 байт. На каждую такую цепочку требуется 3 байта кода, следовательно, объем данных уменьшится до 1/3 первоначального.

Раздел 4.3 1. Например, другой вариант кодов Хаффмена, где 1 заменяется на 0 и наоборот.

4. Возможны такие варианты (Алфавит A={a,b,c,d}):

Сжимается алгоритмом RLE-aaaaaaabbbbbbbccccccddddddd;

Сжимается алгоритмом Хаффмена-aabaacaaddaabaacaadd.

8. Достаточно хранить только длины кодов для каждого из 256 символов входного алфавита. На каждую длину требуется 4 бита, поэтому все дерево займет 128 байт.

Раздел 4.5 2. Не более, чем в 8 раз.

Раздел 4.6 1. Самым частым в тексте, например, нулем или пробелом.

2. Порог должен быть не меньше длины **ptr**-кода. Если **ptr**-код переменной длины, порог также желательно делать переменным.

3. Строку, код которой наименьший, чтобы как можно ближе удовлетворять монотонной модели сигнала.

5. Например, такой код: ABACADAEAFAG...

7. Приложения

7.1 Список использованных обозначений

{X, p(x)}, X={x_1,x_2,...,x_L}-ансамбль сообщений и их вероятности.

L-количество сообщений в ансамбле X.

p(x_i)-вероятность сообщения x_i.

|x_i|-длина сообщения x_i в битах.

H, *H*, H_{max}, Hn-энтропия (бит), относительная энтропия (%), максимальная энтропия ансамбля, n-грамная энтропия.

C-стоимость кодирования (бит).

I(x)-количество информации в сообщении x.

R, R_t, *R*-избыточность кода (бит), избыточность входного текста, относительная избыточность (%).

M-оператор математического ожидания.

S-источник сообщений.

A-алфавит (набор кодов) источника**, A={a_1,a_2,...,a_L}.**

|A|-количество символов в алфавите A.

|a|-длина символа (слова) a (бит).

T-средняя длина входного символа (обычно, 8 бит).

⌊x⌋-наибольшее целое число, не большее x.

⌈x⌉-наименьшее целое число, не меньшее x.

V, |V|-словарь, длина словаря (байт) или объем словаря (слов).

s, |s|, s_{max}-строка, длина строки (байт), максимальная длина строки.

ptr, |ptr|-указатель на слово в словаре, длина этого указателя (бит).

chr, |chr|-символьный указатель, его длина (бит).

<...,...,...>-выходной код, состоящий из нескольких кодов, таких как *prefix, length, distance, symbol, position.*

7.2 Тексты программ

Приведенные здесь программы носят демонстрационный характер. Основное внимание уделено простоте и легкости понимания текста, поэтому контроль ошибок и оптимизация сведены к минимуму, реализовывались "классические" варианты алгоритмов, пригодные, тем не менее, для практического использования и дальнейших модернизаций в качестве самостоятельных упражнений.

Листинг 1.

```
/************************************************
 ** (c) == FAStWare ==                  28.08.96 **
 ** Entropy Calculation Program         Ver. 1.1 **
 ** Fomin A.A.                                   **
 ******************************** Borland C ******/
/* Программа  ENTR рассчета энтропии файла */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

static long code[256]; /* Массив частот символов */
long sum=0;
float entr=0, prob, log2;
FILE *sfile;
int i, ch;

/**************/
void main(int argc, char *argv[])
 {
 if( argc < 2 )
   { printf("\n\nSyntax: ENTR SrcFile"); exit(1); }
 if( (sfile=fopen(argv[1],"rb")) == NULL )
   { printf("\n\nUnable to Open SrcFile"); exit(2); }
 /* Подсчитываем частоты символов */
 while( (ch=fgetc(sfile)) != EOF )
   { code[ch]++; sum++; }
 log2=log(2);
 /* Рассчет энтропии */
 for( i=0; i < 256; i++ )
   {
   if( code[i] == 0 ) continue;
```

```
   prob=code[i]/(float)sum; entr-=(prob*log(prob)/log2);
   }
 printf("\n Bytes: %8ld, Entropy= %6.3f, \
        Relative Entropy= %6.1f\%", sum,entr,entr/8*100);
 }
```

Листинг 2.1.

```
/************************************************
 ** (c) == FAStWare ==                  25.08.95 **
 ** Huffman Tree Builder                Ver. 1.1 **
 ** Fomin A.A.                                   **
 ********************************** Borland C ****/
/* Программа HTREE-строит дерево Хаффмена и таблицу длин
по статистике входного файла */

#include <stdio.h>
#include <stdlib.h>

#define NCH   256       /* Количество символов в алфавите */
#define NCH2  NCH*2
#define MAXL  10000000L /* Очень большое число            */

typedef unsigned int uint;
typedef unsigned char uchar;

/* Прототипы */
void maketree( void );

/* Дерево Хаффмена */
uint dad[NCH2], left[NCH2], right[NCH2];
uchar cleng[NCH2];      /* Длина кодов Хаффмена           */
long cnt[NCH2], fleng=0;
FILE *sfile, *tfile;
int err;

/**************/
void main(int argc, char *argv[])
 {
 printf("\nHTREE - Huffman Tree Builder,  Ver. 1.1, 25.08.95");
 printf("\n(c) == FAStWare ==");
 if( argc < 3 )
```

```
   { printf("\n\nSyntax: HTREE SrcFile TreeFile"); exit(1); }
 if( (sfile=fopen(argv[1],"rb")) == NULL )
   { printf("\n\nUnable to Open SrcFile"); exit(2); }
 if( (tfile=fopen(argv[2],"wb")) == NULL )
   { printf("\n\nUnable to Open TreeFile"); exit(3); }
 maketree();
 printf("\nOk.");
 err=fwrite(cleng,1,NCH,tfile);
 if( err != NCH )
   { printf("\n\nUnable to Write TreeFile"); exit(4); }
 fclose(tfile);
 }

/**************/
void maketree()
 {
 int ch, pos=NCH, tch;
 long min=MAXL;
 uint leng;

 printf("\nGetting Distribution...");
 /* Считаем частоты появления символов алфавита в файле */
 for(ch=0; ch < NCH; ch++ ) cnt[ch]=0;
 while( (ch=fgetc(sfile)) != EOF )
   { cnt[ch]++; fleng++; }
 fseek(sfile,0,SEEK_SET);
 /* Находим самый редкий символ в тексте */
 for( ch=0; ch < NCH; ch++ )
   if( cnt[ch] < min && cnt[ch] > 0 ) min=cnt[ch];
 min=min/4+1;
 /* Всем отсутствующим символам присваиваем частоту min */
 for( ch=0; ch < NCH; ch++ )
   if( cnt[ch] == 0 ) cnt[ch]=min;
 printf("\nMaking Tree...");
 /* Строим дерево */
 while( pos < NCH2-1 )
   {
 /* Находим два самых редких символа и об'единяем их
 в один узел с суммарной частотой */
   min=MAXL;
   for( ch=0; ch < pos; ch++ )
```

```
    if( cnt[ch] < min )
      { min=cnt[ch]; tch=ch; }
  cnt[pos]=cnt[tch]; dad[tch]=pos; cnt[tch]=MAXL; left[pos]=tch;
  min=MAXL;
  for( ch=0; ch < pos; ch++ )
    if( cnt[ch] < min )
      { min=cnt[ch]; tch=ch; }
  cnt[pos] += cnt[tch]; dad[tch]=pos; cnt[tch]=MAXL; right[pos]=tch;
  pos++;
  }
printf("\nMaking Table...");
/* Строим таблицу длин кодов символов, проходя по дереву
от каждого листа к корню */
for( ch=0; ch < NCH; ch++ )
  {
  leng=0; tch=ch;
  do
    {
    leng++; tch=dad[tch];
    } while( tch != NCH2-2 );
/* Максимальная длина кода для этой программы - 16 бит */
  if( leng > 16 )
    { printf("\n\nBad Code Table"); exit(5); }
/* Так как длина кода > 0, уменьшаем все длины на 1 */
  cleng[ch]=leng-1;
  }
}
```

Листинг 2.2.

```
/************************************************
 ** (c) == FAStWare ==                  28.08.95 **
 ** Huffman Encoder-Decoder             Ver. 1.1 **
 ** Fomin A.A.                                   **
 ********************************* Borland C *****/
/* Прграмма HAF-кодирует и декодирует файлы с использованием
дерева Хаффмена, полученного программой HTREE */

#include <stdio.h>
#include <ctype.h>
#include <alloc.h>
#include <stdlib.h>
```

```
#define NCH   256       /* Количество символов в алфавите */
#define NCH2  NCH*2
#define DUMMY 77        /* "Сторожевая" константа         */

typedef unsigned int uint;
typedef unsigned char uchar;

/* Прототипы */
void maketable( void );
void encode( void );
void decode( void );

/* Дерево Хаффмена */
uint code[NCH2], dad[NCH2], left[NCH2], right[NCH2];
uchar cleng[NCH2];      /* Длина кодов Хаффмена           */
long srcleng=0, dstleng;
uchar *srcbuf, *dstbuf, *dststart, *srcstart;
FILE *sfile, *dfile, *hfile;
long err;

/**************/
void main(int argc, char *argv[])
 {
 printf("\nHAF - Huffman Encoder-Decoder,  Ver. 1.1, 25.08.95");
 printf("\n(c) == FAStWare ==");
 if( argc < 4 )
    {
    printf("\n\nSyntax: HAF e SrcFile DstFile [TreeFile]");
    printf("\n        HAF d SrcFile DstFile [TreeFile]"); exit(1);
    }
 srcbuf=farmalloc(0xfff0L);
 if( (dstbuf=farmalloc(0xfff0L)) == NULL )
   { printf("\n\nNot Enough Memory"); exit(6); }
 dststart=dstbuf; srcstart=srcbuf;
 if( (sfile=fopen(argv[2],"rb")) == NULL )
   { printf("\n\nUnable to Open SrcFile"); exit(2); }
 if( (dfile=fopen(argv[3],"wb")) == NULL )
   { printf("\n\nUnable to Open DstFile"); exit(3); }
 if( argc < 5 )
   {
```

```
  if( (hfile=fopen("htree.rus","rb")) == NULL )
    { printf("\n\nUnable to Open TreeFile"); exit(4); }
  }
 else
  {
  if( (hfile=fopen(argv[4],"rb")) == NULL )
    { printf("\n\nUnable to Open TreeFile"); exit(4); }
  }
 if( (srcleng=fread(srcbuf,1,0xfff0U,sfile)) == 0xfff0U )
    { printf("\n\nSrcFile Too Long"); exit(2); }
 if( fread(cleng,1,NCH,hfile) != NCH )
  { printf("\n\nUnable to Read TreeFile"); exit(5); }
 maketable();
 if( toupper(argv[1][0]) == 'E' ) encode();
 else decode();
 if( (err=fwrite(dststart,1,dstleng,dfile)) != dstleng )
  { printf("\n\nUnable to Write DstFile"); exit(7); }
 fclose(dfile); farfree(srcstart); farfree(dststart);
 printf("\nOk");
 }

/**************/
void maketable()
 {
 int ch, pos=NCH, tch;
 int min;
 uint cod, leng, bit;

 printf("\nMaking Tree...");
 min=15;
 /* Строим дерево Хаффмена */
 while( pos < NCH2-1 )
  {
 /* Ищем два самых длинных кода и об'единяем их в один узел
 с более коротким кодом */
  for( ch=0; ch < pos; ch++ )
    if( cleng[ch] == min ) break;
  if( ch == pos )
    { min--; continue; }
 /* Получили левого потомка */
  cleng[pos]=min-1; dad[ch]=pos; cleng[ch]=DUMMY; left[pos]=ch;
```

```
    for( ch=0; ch < pos; ch++ )
      if( cleng[ch] == min ) break;
    if( ch == pos )
      { min--; continue; }
  /* Получили правого потомка */
    dad[ch]=pos; cleng[ch]=DUMMY; right[pos]=ch; pos++;
    }
  printf("\nMaking Table...");
  /* Восстанавливаем коды, просматривая дерево от
  каждого листа к корню */
  for( ch=0; ch < NCH; ch++ )
    {
    leng=cod=0; bit=1; tch=ch;
    do
      {
  /* Если правый потомок - бит=1, иначе - 0 */
      if( right[dad[tch]] == tch ) cod += bit;
      leng++; bit <<= 1; tch=dad[tch];
      } while( tch != NCH2-2 );
  /* Код не должен быть более 16 бит */
    if( leng > 16 )
      { printf("\n\nBad Code Table"); exit(5); }
    cleng[ch]=leng-1; code[ch]=cod << (16-leng);
    }
  }

/**************/
void encode()
  {
  long i;
  uint cod, mask;
  int byte=0, cnt=0, leng;

  printf("\nEncoding...");
  /* Записали исходную длину файла */
  *(long *)dstbuf=srcleng; dstbuf+=4; dstleng=4;
  /* Кодируем файл */
  for( i=0;  i < srcleng; i++ )
    {
    cod=code[*srcbuf]; leng=cleng[*srcbuf]; srcbuf++; mask=0x8000U;
  /* Упаковываем коды в байты и посылаем в dstbuf */
```

```
   for( ; leng >= 0; leng-- )
     {
     byte <<= 1;
     if( cod & mask ) byte++;
     cnt++; mask >>= 1;
 /* Если байт собран, записываем его в dstbuf */
     if( cnt == 8 )
       { *dstbuf=byte; dstbuf++; dstleng++; byte=cnt=0; }
     }
   }
 /* Дописываем последний байт */
 while( cnt < 8 )
      { cnt++; byte <<= 1; }
 *dstbuf=byte; dstleng++;
 }

/**************/
void decode()
 {
 long i;
 int cnt=0, pos;

 printf("\nDecoding...");
 /* Прочитали длину исходного файла */
 dstleng=*(long *)srcbuf; srcbuf+=4;
 /* Декодируем данные побитно, проходя по дереву от корня к
 листу, пока не получим символ из алфавита */
 for( i=0; i < dstleng; i++ )
   {
   pos=NCH2-2;  /* Корень */
   while( pos >= NCH )
     {
     if( *srcbuf & 0x80 ) pos=right[pos];
     else pos=left[pos];
     cnt++;
 /* Если все биты обработаны, переходим к следующему байту */
     if( cnt == 8 ) { srcbuf++; cnt=0; }
     else (*srcbuf) <<= 1;
     }
   *dstbuf=pos; dstbuf++;
   }
```

```
 }
```

Листинг 3.

```
/* Программа FIN вероятностного сжатия   */
/* (c) Puttonen J., Paita T., Teuhola J. */
#include <stdio.h>
#include <string.h>

/* Таблица предсказаний */
char pcTable[32768U];

/* Macro для расчета индекса в таблице предсказаний по
двум предыдущим символам */
#define INDEX(p1,p2) (((unsigned)(unsigned char)p1<<7)^(unsigned
char)p2)

void Compress(FILE *pfIn, FILE *pfOut)
   {
   int c;                /* Символ */
   int i;                /* Счетчик цикла */
   char p1=0, p2=0;      /* Два предыдущих символа */
   char buf[8];
   int ctr=0;            /* Счетчик символов в маске */
   int bctr=0;
   unsigned char mask=0; /* Маска отметок успешного предсказания */

   memset(pcTable, 32, 32768U); /* Пробел (ASCII 32) - самый частый
символ */
   c = fgetc(pfIn);
   while( c != EOF )
     {
     /* Пытаемся предсказать следующий символ */
     if( pcTable[INDEX(p1,p2)] == (char)c )
       {
       /* Верное предсказание, помечаем соответствующий бит */
       mask = mask ^ (1<<ctr);
       }
     else
       {
       /* Ошибочное предсказание, обновляем таблицу */
       pcTable[INDEX(p1,p2)]=(char)c;
```

```
        /* Храним символы во временном буфере */
        buf[bctr++] = (char)c;
        }
       /* Проверяем - не заполнена ли маска */
      if( ++ctr == 8 )
        {
        /* Записываем маску */
        fputc((char)mask, pfOut);
        /* Записываем сохраненные символы */
        for( i=0; i < bctr; i++ )
          fputc(buf[i], pfOut);
        /* Обнуляем переменные */
        ctr=0;
        bctr=0;
        mask=0;
        }
      /* Сдвигаем символы */
      p1 = p2; p2 = (char)c;
      c = fgetc(pfIn);
      }
    /* Если остались несохраненные символы */
    if( ctr )
      {
      /* Записываем маску */
      fputc((char)mask, pfOut);
      /* Записываем оставшиеся символы */
      for( i=0; i < bctr; i++ )
        fputc(buf[i], pfOut);
      }
    }

/**************/
void Decompress(FILE *pfin, FILE *pfout)
    {
    int ci,co;            /* Символы (входной и выходной) */
    char p1=0, p2=0;
    int ctr=8;
    unsigned char mask=0;

    memset(pcTable, 32, 32768U);
    ci = fgetc(pfin);
```

```
   while( ci != EOF )
     {
     /* Получаем маску */
     mask = (unsigned char)(char)ci;
     /* Для каждого бита в маске */
     for( ctr=0; ctr < 8; ctr++ )
       {
       if( mask & (1<<ctr) )
         {
         /* Предсказанный символ */
         co = pcTable[INDEX(p1,p2)];
         }
       else
         {
         /* Неугаданный символ */
         co = fgetc(pfin);
         if( co == EOF ) return;
         pcTable[INDEX(p1,p2)] = (char)co;
         }
       fputc(co, pfout);
       p1 = p2; p2 = co;
       }
     ci = fgetc(pfin);
     }
   }

/* Тест-программа */
void main()
   {
   FILE *a = fopen("in","rb"); FILE *b = fopen("out","wb");
   Compress(a, b); fclose(a); fclose(b);
   a = fopen("out","rb"); b = fopen("rin","wb");
   Decompress(a, b); fclose(a); fclose(b);
   }
```

Листинг 4.

```
/************************************************
 ** (c) == FAStWare ==                  28.08.96 **
 ** LZ77 Encoder-Decoder                Ver. 1.1 **
 ** Fomin A.A.                                   **
 ******************************** Borland C *****/
```

```
/* Кодирование/декодирование файла с использованием
"классического" алгоритма LZ77 */

#include <stdio.h>
#include <alloc.h>
#include <memory.h>
#include <io.h>
#include <ctype.h>
#include <stdlib.h>

#define DICBITS   12             /* Log(DICSIZE) */
#define DICSIZE   (1<<DICBITS)   /* 4kB (Max-32kB) Размер словаря */
#define THRESHOLD 2              /* Порог */
#define STRBITS   6              /* Log(STRMAX-THRESHOLD)  */
#define STRMAX    ((1<<STRBITS)+THRESHOLD)
                                 /* Max. длина строки */
#define BUFSIZE   0xff00U        /* Полный размер буфера */
#define TEXTSIZE  (BUFSIZE-DICSIZE-STRMAX)
                                 /* Размер буфера текста   */
#define YES       1
#define NO        0

typedef unsigned char uchar;

/* Прототипы */
void encode( void );
void decode( void );
void putbits( int data, int nbits );
int  getbits( int nbits );

long srcleng=0, fileleng;
uchar far *srcbuf, *srcstart;
FILE *sfile, *dfile;
long err;

/**************/
void main(int argc, char *argv[])
 {
 printf("\nLZ77 Encoder-Decoder,  Ver. 1.1, 28.08.96");
 printf("\n(c) == FAStWare ==");
 if( argc < 4 )
```

```
   {
   printf("\n\nSyntax: LZ77 e SrcFile DstFile");
   printf("\n        LZ77 d SrcFile DstFile"); exit(1);
   }
 if( (srcbuf=farmalloc((long)BUFSIZE)) == NULL )
   { printf("\n\nNot Enough Memory"); exit(6); }
 /* Инициализация буфера текста и словаря */
 srcstart=srcbuf+DICSIZE; memset(srcbuf,0,DICSIZE);
 if( (sfile=fopen(argv[2],"rb")) == NULL )
   { printf("\n\nUnable to Open SrcFile"); exit(2); }
 if( (dfile=fopen(argv[3],"wb")) == NULL )
   { printf("\n\nUnable to Open DstFile"); exit(3); }
 if( toupper(argv[1][0]) == 'E' ) encode();
 else decode();
 fclose(dfile); farfree(srcbuf);
 }

/**************/
void encode(void)
 {
 uchar far *pos, *ptr;
 int   i, dist, offset=0, last=NO, cnt, maxleng;

 printf("\nEncoding...>");
 /* Записываем размер исходного файла */
 fileleng=filelength(fileno(sfile));
 write(fileno(dfile), &fileleng, sizeof(long));
 /* Читаем файл в буфер по частям размера TEXTSIZE */
 while( (srcleng=fread(srcstart+offset, 1, TEXTSIZE, sfile)) > 0 )
   {
   if( srcleng < TEXTSIZE ) last=YES; /* Последняя часть текста */
   pos=srcstart; ptr=srcbuf; srcleng+=offset;
   while( srcleng > 0 )
     {
     maxleng=0;
 /* Если в буфере текста осталось мало символов, сдвигаем
 словарь и оставшийся текст в начало буфера и дочитываем
 следующую часть из файла */
     if( (last == NO) && (srcleng < STRMAX) )
       {
       putc('>',stdout); memcpy(srcbuf,ptr,DICSIZE+(int)srcleng);
```

```
      offset=(int)srcleng; break;
      }
 /* Ищем самую длинную совпадающую строку в словаре */
    for( i=DICSIZE-1; i >= 0; i-- )
      {
      for( cnt=0; cnt < STRMAX; cnt++ )
        if( *(pos+cnt) != *(ptr+i+cnt) ) break;
 /* Если длина меньше порога, отбрасываем */
      if( cnt <= THRESHOLD ) continue;
 /* Если максимальная строка, дальше не ищем */
      if( cnt == STRMAX )
        { dist=DICSIZE-1-i; maxleng=STRMAX; break; }
 /* Если очередная строка длиннее уже найденных, сохраняем
 ее длину и позицию */
      if( cnt > maxleng )
        { dist=DICSIZE-1-i; maxleng=cnt; }
      }
 /* Проверяем, чтобы не было выхода за границы файла */
    if( (last == YES) && (maxleng > srcleng) ) maxleng=(int)srcleng;
    if( maxleng > THRESHOLD )
      {
 /* Если строка достаточно длинная, формируем ptr-код */
      putbits(1,1); putbits(dist,DICBITS);
      putbits(maxleng-THRESHOLD-1,STRBITS);
      pos+=maxleng; srcleng-=maxleng; ptr+=maxleng;
      }
    else
      {
 /* ...Иначе - chr-код */
      putbits(0,1); putbits(*pos,8);
      pos++; srcleng--; ptr++;
      }
    }
  }
 /* Записать оставшиеся биты */
 putbits(0,8);
 }

/**************/
void decode(void)
 {
```

```
 uchar far *pos;
 int   i, dist, ch, maxleng;

 printf("\nDecoding...>");
 /* Получаем длину исходного файла */
 read(fileno(sfile),&fileleng,sizeof(long));
 pos=srcstart;
 while( fileleng > 0 )
   {
   if( (ch=getbits(1)) == 0 )
 /* Если chr-код, копируем в буфер текста символ */
     { ch=getbits(8); putc(ch,dfile); *pos=ch; pos++; fileleng--; }
   else
     {
 /* ...Иначе - копируем maxleng символов из словаря, начиная
 с позиции dist */
  dist=getbits(DICBITS)+1; maxleng=getbits(STRBITS)+THRESHOLD+1;
     for( i=0; i < maxleng; i++ )
       { *(pos+i)=*(pos+i-dist); putc(*(pos+i-dist),dfile); }
     pos+=maxleng; fileleng-=maxleng;
     }
 /* Если буфер заполнен, записываем его на диск и сдвигаем
 словарь в начало буфера */
   if( pos > srcstart+TEXTSIZE )
     {
     putc('>',stdout); memcpy(srcbuf,pos-DICSIZE,DICSIZE);
     pos=srcstart;
     }
   }
 }

/**************/
/* Записывает в файл код data длиной nbits бит */
void putbits(int data, int nbits)
 {
 static int bitcnt=0;
 static int outdata=0;
 int bit, err;

 data<<=(16-nbits);
 for( ; nbits > 0; nbits-- )
```

```
   {
   if( bitcnt == 8 )
     {
     bitcnt=0; err=putc(outdata,dfile);
     if( err == EOF )
       { printf("\n\nUnable to Write DstFile"); exit(4); }
     }
   outdata<<=1; bit=( data & 0x8000 ) ? 1:0;
   outdata+=bit; bitcnt++; data<<=1;
   }
 }

/**************/
/* Читает nbits бит из файла */
int getbits(int nbits)
 {
 static int bitcnt=8;
 static int indata=0;
 int bit, data=0;

 for( ; nbits > 0; nbits-- )
   {
   if( bitcnt == 8 )
     { bitcnt=0; indata=getc(sfile); }
   if( indata == EOF )
     { printf("\n\nUnable to Read SrcFile"); exit(5); }
   bit=( indata & 0x80 ) ? 1:0;
   data<<=1; data+=bit; bitcnt++; indata<<=1;
   }
 return data;
 }
```

7.3 Предметный указатель

М

Н

О

П

Р

С

Т

У

Ф

Э

7.4 Список литературы

1. A new implementation of the Ziv-Lempel incremental parsing algorithm. IEEE Tr. Inf. Theory IT-37, 5 (1991).

2. Abramson N. Information Theory and Coding. McGraw-Hill, New York, 1963.

3. ARJ. Technical Reference. 1990. (#)

4. Bell T.C. IEEE Trans. COM-34, p. 1176-1182 (1986).

5. Bentley J.L., Sleator D.D., Tarjan R.E., Wei V.K. A locally adaptive data compression scheme. Commun. ACM 29, 4 (Apr. 1986), p. 320-330.

6. Blackstock S. Объяснение LZW и GIF. Софтпанорама 1992, No.4 (28), перевод А. Самотохина. (#)

7. Bloom C. LZP: a new data compression algorithm. 1997. (#)

8. Burrows M., Wheeler D.J. "A Block-sorting Lossless Data Compression Algorithm", Digital Systems Research Center Research Report 124, (1994).

9. Cleary J.G., Witten I.H. Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications, 32(4), (1984), p. 396-402.

10. Cleary J.G., Witten I.H., Teahan W.J. Unbounded length context for PPM. 1995. (#)

11. Cormack G., Horspool N. Data compression using dynamic Markov modelling. Computer Journal 30:6 (Dec. 1987).

12. Fiala E.R., Greene D.H. Data compression with finite windows. CACM-32, 4 (1989), p. 490-505.

13. Gallager R.G. Variations on the theme by Huffman. IEEE Trans. Inf. Theory IT-24, 6 (Nov. 1978), p. 668-674.

14. Golomb S.W. Run-length encoding. IEEE Tr. Inf. Theory IT-12, (1966), p. 399-401.

15. Huffman D.A. A method for the construction of minimum-redundancy codes. Proc. Inst. Electr. Radio Eng. 40, 9 (Sept. 1952), p. 1098-1101.

16. Knuth D.E. Dynamic Huffman coding. J. Algorithms 6, 2 (Feb. 1985), p. 163-180.

17. Moffat A. Implementing the PPM data compression scheme. IEEE Transactions on Communications, 38(11), (1990), p. 1917-1921.

18. Montgomery B. LZW compression used to encode/decode a GIF file. (#)

19. Nelson M. Data Compression with the Burrows-Wheeler Transform. Dr. Dobb's Journal, (Sep. 1996).

20. Okumura H. Data Compression Algorithms of LARC and Lharc. April, 1989. (#)

21. PkZip Version 1.5. Technical Reference. (#)

22. Practical dictionary management for hardware data compression. CACM-35, 1 (Jan. 1992), p. 95.

23. Rissanen J., Langdon G. Universal modelling and coding. IEEE Trans. Inf. Theory IT-27, 1 (Jan 1981), p. 12-23.

24. Rubin F. Arithmetic stream coding using fixed precision registers. IEEE Trans. Inf. Theory IT-25, 6 (Nov. 1979), p. 672-675.

25. Storer J.A., Szymanski T.G. Data compression via textual substitution. Journal of the ACM 29, 4 (Oct. 1982), p. 928-951.

26. Vries N.E. Loseless Data compression Sources-kit. (#)

27. Welch T.A. A technique for high-perfomance data compression. // IEEE Comput. 17, 6 (June 1984), p. 8-19.

28. Wilson N. Single-chip engine for document compression. // Electronics and wireless world 95, 1636 (Feb. 1989), p.116-119.

29. Witten I.H., Neal R.M., and Cleary J.G. Arithmetic coding for data compression. // Commun. ACM 30, 6 (June 1987), p. 520-540.

30. Ziv J., and Lempel, A. Compression of individual sequences via variable-rate coding. // IEEE Trans. Inf. Theory IT-24, 5 (Sept. 1978), p. 530-536.

31. Ziv J., Lempel A. A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory IT-23, 3 (1977), p. 337-343.

32. Арапов Д. Пишем упаковщик. // Монитор.-1993.-№ 1.-С. 16-26.

33. Бентли Д. Жемчужины творчества программистов. М.: Радио и связь, 1990.

34. Буяновский Г. Ассоциативное кодирование. // Монитор.-1994.-№8.-С. 10-22.

35. Глушанков Е.И., Карпов В.С. Анализ качества сжатия видеоинформации при передаче по каналу связи с ограниченной пропускной способностью. // Тезисы докладов VII научно-технической конференции “Оптические, сотовые и спутниковые сети и системы связи”.-Пушкин.-1996.-с. 81-82.

36. Колесник В.Д., Полтырев Г.Ш. Курс теории информации. М.: Наука, 1982.

37. Кохманюк Д.С. Реализация ввода-вывода со сжатием данных. Киев, 1992. (#)

38. Кричевский, Р.Е. Сжатие и поиск информации. М.: Радио и связь, 1989.

39. Лэнгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. М.: Мир, 1989.

40. Мастрюков Д. Алгоритмы сжатия информации : Часть 2-7. // Монитор.-1994.-№1-6.

41. Месси Дж. Л. Введение в современную криптологию. // ТИИЭР.- т. 76.-№ 5.-1988.-с. 24.

42. Молдовян Н.А. Программные шифры с недетерминированным алгоритмом. // Тезисы докладов VII научно-технической конференции “Оптические, сотовые и спутниковые сети и системы связи”.-Пушкин.-1996.-с. 192-193.

43. Павлидис Т. Алгоритмы машинной графики и обработки изображений. М.: 1989.

44. Уэйт М., Прата С., Мартин Д. Язык “Си”. М.: Мир, 1988.-512 с.

45. Феоктистов А., Моргунов М. Что нужно учитывать при выборе модема. // КомпьютерПресс.-1994.-№ 3.-С. 52-56.

46. Фомин А.А. Сжатие информации как один из методов ее защиты. // Тезисы докладов республиканского НТС "Методы и технические средства защиты информации".-СПб.-1993.

47. Шеннон К. Работы по теории информации и кибернетике. М.: ИЛ, 1963.

(#)-Издания, представленные в “электронном” виде.