БЕЛАРУСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет радиофизики и электроники

Кафедра системного анализа

ИССЛЕДОВАНИЕ АЛГОРИТМОВ СЖАТИЯ ДАННЫХ

Курсовая работа студента III курса

Поплетеева А.М.

Руководитель: ст. преп. Лутковский В.М.

“Допустить к защите”

Заведующий кафедрой системного анализа

профессор В.В. Апанасович

“___” _____ 2002 г.

МИНСК 2002

Содержание

Введение

В последние годы объем обрабатываемых компьютерами данных необычайно возрос. Это связано, в первую очередь, с достижениями в сфере мультимедиа-технологий. Несмотря на значительное падение цен на устройства памяти, проблема эффективного хранения, передачи и поиска среди большого количества данных остается острой.

Первые методы сжатия данных появились еще до создания вычислительной техники. Первоначально они использовались в криптографии для шифрования информации. На данный момент существует множество алгоритмов сжатия, позволяющих успешно решать задачу хранения и быстрой передачи большого количества данных. Сравнение этих алгоритмов обычно производится по следующим основным критериям:

- Степень сжатия, т.е. отношение размера входного набора символов к размеру выходного.
- Скорость упаковки/распаковки, т.е. время, затрачиваемое на сжатие/восстановление некоторого объема информации.

Тем не менее, остается актуальной проблема эффективного поиска в сжатых данных, ведь большая часть информации хранится именно в упакованном виде. Известно, что если данных очень много и они находятся на удаленном сетевом диске, то большая часть времени поиска затрачивается на передачу данных по сети.

Существует два подхода к решению этой задачи. Первый, самый очевидный – «распаковать-затем-искать», когда сжатый файл сначала распаковывается, а затем применяется какой-либо традиционный алгоритм поиска. Этот метод прост, но неэффективен. Прежде всего, должен быть распакован весь файл, а это довольно длительный процесс, особенно когда дело касается файлов, размер которых исчисляется мегабайтами. К тому же, распакованный файл должен где-то храниться, чтобы можно было произвести поиск.

Другая альтернатива – поиск в сжатом файле без распаковки или, в крайнем случае, только с частичной его распаковкой. Этот подход обладает рядом привлекательных преимуществ. Файлы со сжатыми данными имеют меньший размер, так что алгоритм поиска потребует меньше времени для обработки всего файла. К тому же отпадает необходимость в работе по полной распаковке данных.

Главная сложность поиска в сжатых данных – это то, что упаковка может нарушить структуру файла. Чем больше степень сжатия, тем менее структурированным становится упакованный файл. Поэтому

необходимо соблюдать тонкий баланс между хорошей степенью сжатия и достаточным количеством “подсказок” для алгоритма поиска. Может показаться, что эти две цели являются взаимоисключающими, но на самом деле сжатие и поиск очень близки в том, что многие методы сжатия используют разновидность поиска для нахождения повторяющихся участков во входных данных, что может быть использовано для достижения лучшего сжатия. Результатом этого является то, что такие повторяющиеся участки кодируются особым образом, что, будучи представлено соответствующим образом, может существенно помочь при поиске.

Таким образом, при сравнении алгоритмов сжатия данных необходимо учитывать и такой критерий, как количество полезной информации, которую можно извлечь из сжатого файла без распаковки. Очевидно, что зачастую ради возможности быстрого поиска/распознавания в упакованных данных можно поступиться степенью сжатия.

Для поиска в сжатых данных существует множество методов. Один из популярных – упаковка искомой подстроки и последующий поиск ее сжатого образа в упакованном файле – не будет работать в случаях, когда эта подстрока может иметь несколько различных, зависящих от контекста, сжатых представлений. Это происходит, например, когда границы упакованного текста не соответствуют границам оригинального текста, что происходит при арифметическом кодировании или когда применяются методы адаптивного сжатия.

В данной работе проводится обзор основных алгоритмов сжатия информации, причем отдельное внимание уделяется возможности распознавания и поиска непосредственно в сжатых данных.

Классификация алгоритмов сжатия

Все алгоритмы сжатия делятся на два класса:

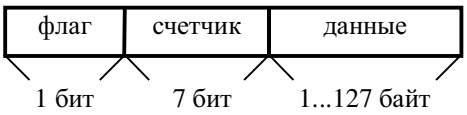
- Алгоритмы сжатия без потерь
- Алгоритмы сжатия с потерями

Алгоритмы сжатия без потерь появились первыми. Они обладают сравнительно невысоким коэффициентом сжатия мультимедиа-данных, но зато обладают свойством обратимости, т.е. позволяют полностью и точно восстановить упакованные данные. Алгоритмы этого класса применяются, когда важно обеспечить идентичность распакованной информации исходной, например, при сжатии текста или исполняемых файлов.

Алгоритм RLE

Групповое кодирование Run Length Encoding (RLE) – один из самых старых и самых простых алгоритмов компрессии. Сжатие происходит за счет того, что в исходном файле встречаются цепочки одинаковых байт. Эти последовательности заменяются парами «счетчик, значение», что уменьшает избыточность данных.

Существует множество реализаций этого алгоритма, отличающихся степенью сжатия. В одном из вариантов пары «счетчик, значение» имеют такой вид:



Если флаг установлен (соответствующий бит равен 1), то счетчик указывает, сколько раз следует повторить следующий за ним байт. В противном случае, счетчик определяет количество записанных за ним неповторяющихся (неупакованных) байт.

Несложно подсчитать, что в лучшем случае файл сжимается в 64 раза, а в худшем – увеличивается на 1/128. В среднем файлы растровых изображений сжимаются приблизительно в три раза. К преимуществам алгоритма можно отнести высокое быстродействие и то, что он не требует дополнительной памяти при архивации и разархивации.

Алгоритм RLE применяется для сжатия деловой и научной графики (графики, чертежи), т.е. изображений с небольшим количеством цветов и большими областями повторяющегося цвета. Используется в изображениях формата PCX, TIFF, TGA, BMP.

Поиск в данных, сжатых алгоритмом RLE является сравнительно простой задачей. Существуют публикации, посвященные ей, например [1]. Нужно просто упаковать образец поиска и применить какой-либо алгоритм поиска к сжатому тексту и образцу. Единственная возникающая сложность – это необходимость в отдельной дополнительной обработке первого и последнего элементов образца.

Большое внимание уделялось также алгоритмам двумерного поиска в данных, упакованных алгоритмом двумерного RLE-сжатия. Двумерное RLE-сжатие (2D RLE) – это конкатенация группового сжатия для всех строк (или столбцов) матрицы.

Задача двумерного поиска в 2D RLE-упакованных данных определяется следующим образом [2]. На входе имеется текстовый массив T и массив-образец поиска P размером m x m, в сжатом по алгоритму 2D RLE виде. На выходе требуется получить местоположения всех встреченных в T совпадений с P, т.е. набор пар (i, j), таких, что $T[i+k, j+l] = P[k+1, l+1]$, $k, l=0..m-1$ и $T[i+k, j] \neq T[i+k, j-1]$, $T[i+k, j+m-1] \neq T[i+k, j+m]$, $k=0..m-1$. Было предложено несколько способов решения этой задачи [1-6].

RLE-сжатие удобно применять для упаковки некоторых видов временных диаграмм или спектров с целью обеспечить возможность их дальнейшего распознавания без распаковки.

На рис.1,а показан самый простой случай. Сжатый алгоритмом RLE двухуровневый сигнал может быть представлен просто набором чисел, отражающих количество повторений логических нулей и единиц. Этот набор можно подавать на входы нейронной сети, которая, будучи соответствующим образом обучена, сможет произвести распознавание. При этом необходимо хранить какой-либо дополнительный признак, позволяющий не спутать сигнал с его инверсией.

Рис. 1,б отображает более сложный, но и более близкий к реальности сигнал, а именно идеализированный атомно-эмиссионный спектр бария. RLE-сжатие в этом случае не будет столь эффективным, как, например, кодирование Хаффмана, но тем не менее, позволяет просто определить ширину спектральной линии и ее положение относительно начала путем прямого считывания двух соответствующих слов из сжатого образа спектра. Кроме того, зная эти величины, возможно приблизительно определить положение характерного изгиба на спектре (60-й отсчет), что довольно важно для идентификации элемента по его спектру.

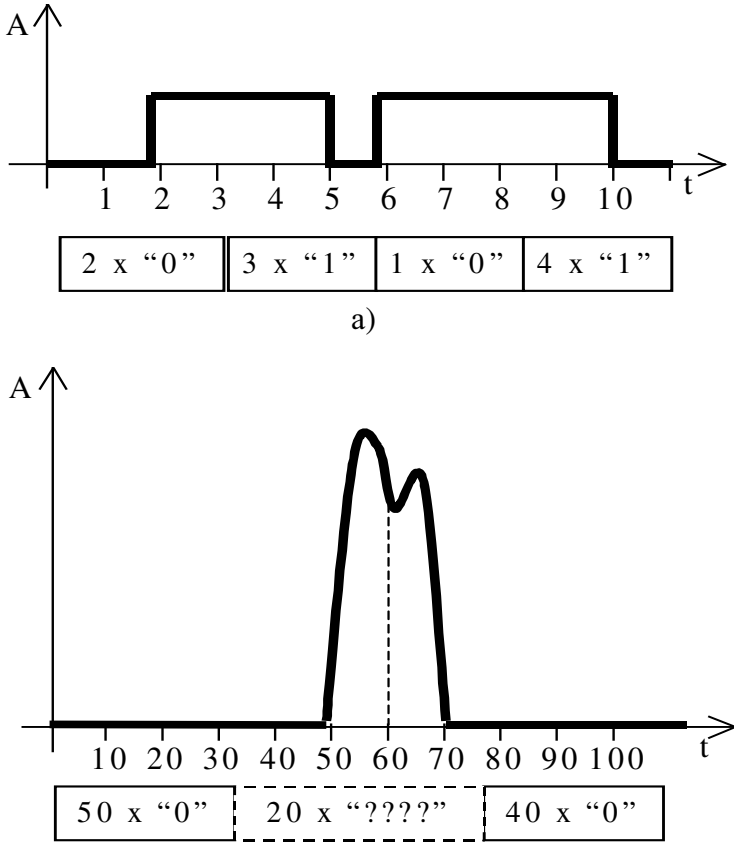


Рис. 1. Применение RLE-сжатия к временной диаграмме:
двухуровневого сигнала (а) и атомно-эмиссионному спектру бария (б)

Алгоритм LZW

Исходный Lempel/Ziv поход к сжатию данных впервые был опубликован Дж. Зивом (J. Ziv) и А. Лемпелем (A. Lempel) в 1977 году в [7]. Впоследствии он был доработан Терри А. Велчем (Terry A. Welch) и алгоритм получил название LZW (Lempel-Ziv-Welch).

Сжатие в алгоритме LZW, в отличие от RLE, осуществляется уже за счет замены одинаковых цепочек байт (строк символов) некоторыми кодами, причем никакого анализа входной информации не производится. При добавлении каждой новой строки символов просматривается таблица строк. Сжатие происходит, когда код заменяет строку символов. Коды, генерируемые LZW-алгоритмом, могут быть любой длины, но они должны содержать больше бит, чем единичный символ. Первые 256 кодов (при использовании 8-битных символов) по умолчанию соответствуют стандартному набору символов. Остальные коды соответствуют обрабатываемым алгоритмом строкам.

Рассмотрим работу простейшего LZW алгоритма сжатия (листинг 1) более детально. Каждый раз, когда генерируется новый код, новая строка добавляется в таблицу строк. Алгоритм постоянно проверяет, встречалась ли строка раньше, т.е. есть ли она в таблице строк, и, если так, выводит существующий код без генерации нового.

```
СТРОКА := очередной символ из входного потока;
WHILE входной поток не пуст DO
    СИМВОЛ := очередной символ из входного потока;
    IF СТРОКА+СИМВОЛ в таблице строк THEN
        СТРОКА := СТРОКА+СИМВОЛ;
    ELSE
        вывести в выходной поток код для СТРОКА;
        добавить в таблицу строк СТРОКА+СИМВОЛ;
        СТРОКА := СИМВОЛ;
    END;
END;
вывести в выходной поток код для СТРОКА;
```

Листинг 1. Простейший LZW алгоритм сжатия

Пример сжатия алгоритмом LZW строки с большой избыточностью, выходной поток и таблица строк показаны в табл. 1.

Таблица 1

Пример работы LZW алгоритма сжатия входной строки «/WED/WE/WEE/WEB»

Вход (символы)	Выход (коды)	Новые коды и соответствующие строки
/W	/	256=/W
E	W	257=WE
D	E	258=ED
/	D	259=D/
WE	256	260=/WE
/	E	261=E/
WEE	260	262=/WEE
/W	261	263=E/W
EB	257	264=WEB
/	B	265=B/
WET	260	266=/WET
<EOF>	T	

Легко заметить, что таблица строк быстро заполняется, т.к. новая строка добавляется в туда каждый раз, когда генерируется код. В этом

явно вырожденном примере было выведено пять закодированных подстрок и семь символов. Если использовать 9-битные коды для вывода, то 19-символьная входная строка будет упакована в 13.5-символьную выходную строку. В реальных случаях сжатие начинается только после построения достаточно большой таблицы строк, обычно после прочтения нескольких сотен входных байт.

Алгоритм распаковки получает на входе поток кодов и использует их для точного восстановления данных. Эффективность LZW-алгоритма обусловлена тем, что он не нуждается в хранении таблицы строк, полученной при сжатии. Эта таблица может быть в точности восстановлена в ходе распаковки на основании только входного потока кодов. Это возможно потому, что алгоритм сжатия выводит строковую и символьную компоненты кода, прежде чем он поместит этот код в выходной поток. Таким образом, нет необходимости сопровождать сжатые данные большой таблицей перевода. Как и алгоритм сжатия, декомпрессор добавляет новые строки в таблицу строк каждый раз, когда он считывает новый код. Все, что остается сделать – это найти строку, соответствующую входному коду, и вывести ее.

Алгоритм распаковки LZW представлен в листинге 2, а процесс восстановления исходной строки из сжатых данных, полученных в предыдущем примере – в табл. 2.

К сожалению, этот прекрасный алгоритм не лишен некоторых недостатков. Во-первых, таблица строк в процессе упаковки может достаточно быстро стать очень большой. Даже если длина строк в среднем ограничивается тремя-четырьмя символами, верхний предел длин строк может легко превысить 7 или 8 байт на код. К тому же количе-

```
читать СТАРЫЙ_КОД;
вывести СТАРЫЙ_КОД;
WHILE входной поток не пуст DO
    читать НОВЫЙ_КОД;
    СТРОКА := перевести НОВЫЙ_КОД;
    вывести СТРОКУ;
    СИМВОЛ := первый символ СТРОКИ;
    добавить в таблицу перевода СТАРЫЙ_КОД+СИМВОЛ;
    СТАРЫЙ_КОД := НОВЫЙ_КОД;
END;
```

Листинг 2. Алгоритм распаковки данных, сжатых LZW-методом

ство памяти, необходимой для хранения строк, заранее не известно, так как оно зависит от общей длины строк.

Таблица 2

Процесс распаковки LZW-кодов

Вход Новый код	Старый код	Строка Выход	Символ	Новое значение кода и соответствующая строка
/	/	/		
W	/	W	W	256=/W
E	W	E	E	257=WE
D	E	D	D	258=ED
256	D	/W	/	259=D/
E	256	E	E	260=/WE
260	E	/WE	/	261=E/
261	260	E/	E	262=/WEE
257	261	WE	W	263=E/W
B	257	B	B	264=WEB
260	B	/WE	/	265=B/
T	260	T	T	266=/WET

Вторая проблема заключается в организации поиска строк. Каждый раз, когда читается новый символ, необходимо организовать поиск для новой строки вида СТРОКА+СИМВОЛ. Это означает поддержку отсортированного списка строк. В этом случае поиск для каждой строки включает число сравнений порядка log2 от общего числа строк.

Первую проблему можно решить, организовав хранение строк в виде комбинаций код/символ. Так как каждая строка в действительности является представлением комбинации уже существующего кода и добавочного символа, можно хранить ее как этот код плюс символ. Например в представленном выше случае строка "/WEE" хранится как код 260 и символ "E". Это позволяет использовать для хранения только 3 байта вместо 5 (включающих дополнительный байт для конца строки). Идя назад, можно определить, что код 260 хранится как код 256 плюс добавочный символ "E". Наконец, код 256 хранится как "/" плюс "W".

Быстрый поиск строк является более трудной задачей, поэтому существуют несколько вариаций алгоритма Лемпеля-Зива, различающихся в основном организацией работы с таблицей строк. LZW является самым известным.

Описанный способ хранения увеличивает время, необходимое для сравнения строк, но он не влияет на число сравнений. Проблема быст-

рого поиска решается использованием хеширования для хранения строк. Это означает, что код 256 не хранится в каком-либо массиве по адресу 256, а хранится в массиве по адресу, сформированному на основе самой строки. При определении места хранения данной строки можно использовать тестовую строку для генерации хэш-адреса и затем найти целевую строку однократным сравнением. Так как код для любой данной строки нельзя узнать в дальнейшем иначе как по его позиции в массиве, необходимо хранить код для данной строки совместно с данными строки.

Еще одна проблема возникает при сжатии больших файлов: по мере роста числа прочитанных байтов степень сжатия может начать ухудшаться. Причина этого проста: так как размер таблицы строк фиксирован, может возникнуть ситуация, когда новую строку уже просто некуда добавить. Обычно эта проблема решается введением контроля за степенью сжатия. После того, как таблица строк заполнена, упаковщик следит за поведением коэффициента сжатия. При определенном его уменьшении таблица строк очищается и начинает строиться заново. Процедура распаковки определяет этот момент благодаря тому, что упаковщик записывает в свой выходной поток специальный код. Альтернативным способом является применение адаптивного подхода, т.е. определение наиболее часто встречающихся строк и чистка остальных. Однако реализация такого решения довольно сложна.

LZW-сжатие лучше всего работает с данными, содержащими повторяющиеся участки любой структуры. По этой причине метод весьма эффективно работает с текстами на естественном языке и исходными текстами программ. В этом случае исходный файл может быть сжат в два и более раза. Сжатие изображений деловой и научной графики показывает еще лучшие результаты.

Очевидным преимуществом алгоритма LZW является то, что нет необходимости включать таблицу кодировки в файл с упакованными данными. Алгоритм отличается высокой скоростью упаковки и распаковки, довольно скромными требованиями к памяти и простотой аппаратной реализации. Благодаря своей универсальности, алгоритм нашел применение практически во всех популярных программах-архиваторах (arj, gz, lha, pak, zip, zoo) и в некоторых форматах графических файлов (gif, tiff).

Поиск в данных, сжатых LZW-алгоритмом, является нетривиальной задачей. В отличие от RLE, здесь уже нельзя просто упаковать

образец поиска и производить поиск его в сжатом тексте, учитывая некоторые нюансы. Это обусловлено тем, что LZW-сжатие является адаптивным, т.е. текст, представляемый каждым кодом в потоке упакованных данных, динамически определяется по уже считанным данным. Как следствие, одна и та же строка будет кодироваться поразному в зависимости от ее расположения в тексте. Так что сжатие образца поиска бесполезно, т.к. он не встретится в сжатом тексте в упакованном виде.

Пионерами в области поиска в LZW-сжатых данных являются Амир, Бенсон и Фарач (Amir, Benson, Farach).

Рассмотрим их алгоритм поиска в LZW-сжатых данных [8].

Прежде всего, введем несколько обозначений.

Пусть $\sigma = s_1 \ldots s_u$ – строка из алфавита $\Sigma = \{a_1, \ldots a_q\}$. *Сжатой строкой σ.Z* строки σ является строка $\sigma.Z[1] \ldots \sigma.Z[n]$, где $1 \le \sigma.Z[i] \le n+q+1$, для $i=1,\ldots n$. Сжатие происходит с использованием *словарного дерева* T_σ. Это дерево с (n+q+1) узлами, которые помечены символами из Σ и пронумерованы от 0 до n+q. *Строка узла p* в дереве T_σ – это совокупность меток узлов по пути от корневого узла к p.

Каждый элемент сжатой строки представляет собой *цепочку*. Цепочка, представленная элементом σ.Z[i] (σ.Z[i]-тая цепочка) –это строка узла, на который указывает σ.Z[i].

Следует отметить, что словарное дерево является просто одним из вариантов организации таблицы строк, описанной ранее. Номер узла p – это код, который соответствует строке этого узла. Построение словарного дерева подобно построению таблицы строк, поэтому описывать этот процесс не будем.

Пример построенного словарного дерева и сжатия с его помощью изображен на рис. 2.

Как было показано выше, имея строку σ, можно построить T_σ и σ.Z, и наоборот: имея строку σ.Z, можно восстановить T_σ и σ.

Амиром и др. были доказаны следующие два утверждения.

1. Присвоение значения *l*-тому символу строки σ.Z, $1 \le l \le n-1$, вызывает определение и создания в словарном дереве узла номер *l*+q.
2. Пусть $\sigma.Z[i]=l$. Первый символ σ.Z[i]-той цепочки является первым символом σ.Z[*l*-q]-той цепочки. Последний символ σ.Z[i]-той цепочки является первым символом σ.Z[*l*-q+1]-той цепочки.

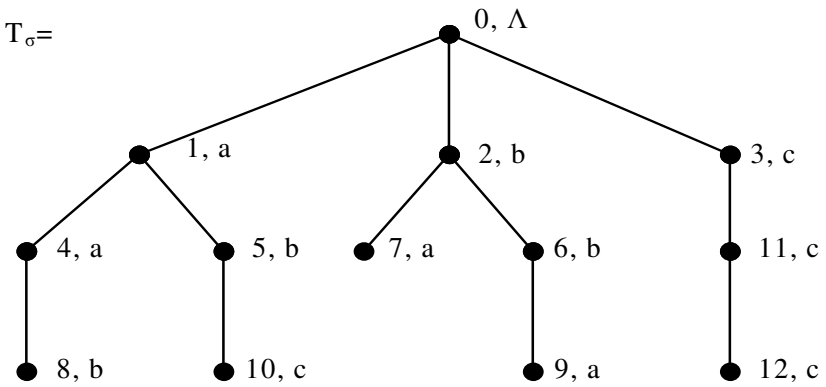


Рис. 2. Словарное дерево и пример строки, упакованной с его помощью

Еще несколько определений. Пусть $P = p_1 \ldots p_n$ – строка-образец поиска. *Внутренняя подстрока* $p_i \ldots p_j$ образца – такая подстрока, где $i>1$ и $j \le m$. Цепочка, являющаяся внутренней подстрокой, называется *внутренней цепочкой*. Цепочка называется *префиксной цепочкой*, если она заканчивается непустым префиксом образца. Префикс-представитель префиксной цепочки – это самый длинный префикс образца, которым она заканчивается. Цепочка называется *суффиксной цепочкой*, если ее начало является непустым суффиксом образца. Ее суффикс-представитель – это самый длинный суффикс образца, с которого она начинается. Рис. 3 поясняет сказанное.

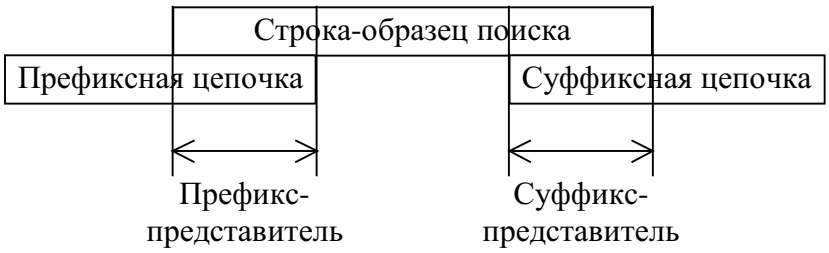


Рис. 3. К определению префиксной и суффиксной цепочек

Теперь перейдем к сути метода [8]. Как и в большинстве задач поиска, алгоритм состоит из двух частей: предобработка строки-образца поиска и просмотр текста (в нашем случае текстом является сжатая строка σ.Z). Примем обозначение S_1S_2 для конкатенации (объединения) строк S_1 и S_2, и <i, j> – для подстроки $p_i \ldots p_j$ образца.

Предобработка образца поиска нужна для того, чтобы можно было ответить на следующие запросы:

1 Пусть S_1 – префикс образца, S_2 – внутренняя подстрока. $Q_1(S_1,S_2)$ = длине самого длинного префикса образца, который является суффиксом S_1S_2.

2 Пусть S_1 – префикс образца, а S_2 – его суффикс.

$$Q_2(S_1,S_2)=\begin{cases} i, & i-\textit{наименьший индекс } S_1S_2\textit{, где есть вхождение образца} \\ 0, & \textit{образец не входит в } S_1S_2 \end{cases}$$

3 Пусть S_1–внутренняя подстрока, $a \in \Sigma$.

$$Q_3(S_1,a)=\begin{cases} < i,j >, & S_1a-\textit{внутренняя подстрока строки } p_i...p_j, \\ & j=m\textit{, если возможно} \\ < 0,0 >, & S_1a-\textit{не внутренняя подстрока} \end{cases}$$

Процедура просмотра текста состоит из двух основных частей: создание и обновление дерева-словаря и собственно поиск образца.

При создании словаря также вводятся и соответствующим образом помечаются *флаг префикса*, *флаг суффикса* и *флаг внутренней строки*. Плюс ко всему, каждый узел хранит первый символ своей строки. Флаг префикса хранит длину префикса-представителя, флаг суффикса – длину суффикса-представителя, а флаг внутренней строки – индексы внутренней подстроки, которую представляет этот узел.

Процедура поиска образца отслеживает самый длинный префикс образца, который заканчивается с предыдущей цепочкой, а затем использует запросы, чтобы определить, расширяет ли текущая цепочка префикс или нет.

Алгоритм обработки LZW-сжатого текста представлен в листинге 3.

Приведенный алгоритм не является оптимальным. Было предпринято несколько более или менее успешных попыток улучшить его, предложены альтернативные методы для поиска в данных, сжатых различными вариантами алгоритма Лемпеля-Зива, например [9-12].

Инициализация: Prefix←Λ
for l=1 to n do *//Нижеследующее выполняется для всех элементов σ.Z*
1. Создание словаря и установка флагов
   (a) Добавить узел номер l+q в T_σ. Этот узел является дочерним для узла номер σ.Z[l]. Его метка является первым символом узла σ.Z[l+1]. (Если σ.Z[l+1]=l+q, то метка является первым символом узла σ.Z[l].)
   (b) Первый символ узла номер l+q – это первый символ его родителя.
   (c) Если σ.Z[l] – суффиксный узел, то l+q – суффиксный узел. У

него такой же суффикс-представитель, как и у его родителя.

(d) Если σ.Z[l] – внутренний узел $p_i \ldots p_k$ и метка (l+q)-того узла равна a, тогда записать во флаг внутренней строки (l+q)-того узла значение $Q_3(p_i \ldots p_k, a) = \langle h,j \rangle$. Если узел l+q является суффиксным (т.е. j = m), то записать в его флаг суффикса значение, соответствующее его суффиксу-представителю. (Это может изменить значение, установленное на шаге (c).)

(e) Если σ.Z[l] – префиксный узел с префиксом-представителем $p_1 \ldots p_i$ и метка узла l+q равна a, то записать во флаг префикса (l+q)-того узла значение $Q_1(p_1 \ldots p_i, a)$.

2. Поиск образца

(a) Если Prefix = Λ и σ.Z[l] – префиксный узел с префиксом-представителем $p_1 \ldots p_i$, то Prefix← $p_1 \ldots p_i$.

(b) Если Prefix ≠ Λ и σ.Z[l] – суффиксный узел с суффиксом-представителем $p_i \ldots p_m$, то если $Q_2(\text{Prefix}, p_i \ldots p_m) \neq 0$, найдено вхождение образца

(c) Если Prefix ≠ Λ и σ.Z[l] – внутренний узел $p_i \ldots p_j$, то Prefix←$Q_1(\text{Prefix}, p_i \ldots p_j)$, иначе σ.Z[l] – не внутренний узел и Prefix←префикс-представитель σ.Z[l].

Листинг 3. Обработка сжатого LZW-алгоритмом текста

Алгоритм Хаффмана.

Алгоритм Хаффмана – один из классических алгоритмов сжатия, известных с 60-годов XX века. В его основе лежит довольно простая идея: символы входного потока заменяются кодовыми последовательностями различной длины. Чем чаще встречается символ, тем короче соответствующая ему кодовая последовательность. Поэтому алгоритм Хаффмана иногда называется также кодированием символами переменной длины (Variable-Length Coding). Код переменной длины позволяет записывать наиболее часто встречающиеся символы короткими кодовыми последовательностями, а редко встречающиеся – более длинными.

Таким образом, для сжатия нужно знать частоту встречаемости каждого символа во входном файле. Для этого необходимо предварительно просмотреть весь файл, посчитать сколько раз там встречается тот или иной символ, и построить таблицу частот. В этом заключается один из недостатков метода Хаффмана – необходимость в двух просмотрах файла (первый – для построения таблицы частот, второй – в

процессе сжатия), что не может не сказаться на быстродействии и вызывает определенные трудности при сжатии потоковых данных.

Одна из сложностей использования кодов переменной длины заключается в том, чтобы узнать, достигнут ли в процессе чтения отдельных бит конец символа. Эта проблема может быть решена, если использовать такой код, что ни один полный код любого символа не является началом кода другого символа. Такие коды называются префиксными.

Код Хаффмана может быть представлен двоичным деревом, листья которого представляют закодированные символы. Каждому листу дерева присвоен определенный вес (частота встречаемости соответствующего символа). Каждый узел дерева имеет вес, равный сумме всех весов листьев, расположенных ниже него.

Кодовое дерево строится на основе таблицы частот, для чего она, прежде всего, сортируется по убыванию. Элементы этой таблицы становятся листьями кодового дерева. Построение дерева происходит от листьев к корню.

Для этого выбираются два узла (листа) верхнего уровня с наименьшими весами, и над ними создается новый узел с весом, равным сумме этих весов. Это повторяется до тех пор, пока дерево не сформировано, т.е. пока все не сведется к одному, корневому узлу.

Пример построения кодового дерева изображен на рис. 4.

Когда дерево построено, можно переходить непосредственно к кодированию входного файла. Кодирование символа начинается из корня. Далее алгоритм должен отслеживать все повороты ветвей дерева на пути к кодируемому символу. При переходе на левую ветвь в выходной поток записывается, скажем, бит «0», а при переходе на правую – бит «1».

Для рассматриваемого примера получается таблица кодировки символов (табл. 3). В исходном файле каждый символ занимает 8 бит, а в упакованном – от 2-х до 4-х бит. В результате исходный файл сжимается на 70%.

Это очень неплохой результат, однако есть один неучтенный нюанс: для восстановления сжатых методом Хаффмана данных необходимо иметь кодовое дерево, использовавшееся при упаковке. Т.е. в сжатом файле необходимо дополнительно хранить либо кодовое дерево, либо таблицу частот, по которой его можно построить.

Таблица частот:

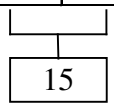
Символ	A	B	C	D	E	F
Число вхождений	10	20	30	5	25	10

Таблица частот, отсортированная по убыванию

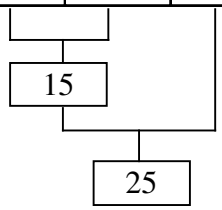
Символ	C	E	B	F	A	D
Число вхождений	30	25	20	10	10	5

Процесс построения кодового дерева:

Символ	C	A	D	F	B	E
Число вхождений	30	10	5	10	20	25



Символ	C	A	D	F	B	E
Число вхождений	30	10	5	10	20	25



Построенное кодовое дерево:

Символ	C	A	D	F	B	E
Число вхождений	30	10	5	10	20	25

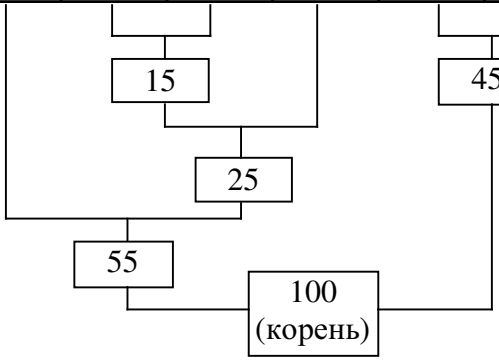


Рис. 4. Пример построения кодового дерева

Второй вариант является более предпочтительным по степени сжатия, но несколько ухудшает быстродействие при распаковке, т.к. на построение кодового дерева теряется дополнительное время.

Однако этот недостаток метода Хаффмана – необходимость хранения с упакованными данными таблицу частот символов – на самом деле является и его достоинством.

Таблица 3

Таблица кодировки

Символ	Код	Длина кода, бит
C	00	2
A	0100	4
D	0101	4
F	011	3
B	10	2
E	11	2

При упаковке методом Хаффмана атомно-эмиссионных спектров различных химических элементов было замечено, что таблицы частот для спектров одного элемента похожи, и при этом сильно отличаются от таблиц частот, полученных при сжатии спектров других элементов [13]. На рис. 5 показаны несколько спектров и соответствующие им гистограммы частот.

При решении задачи распознавания спектров таблица частот играет двойную роль: спектры хранятся в базе данных в сжатом виде, и таблица частот не только позволяет при необходимости восстановить упакованный спектр, но и является своеобразным его образом, позволяющим осуществлять распознавание без распаковки.

Для распознавания применяется искусственная нейронная сеть (ИНС)[14], предварительно обученная на примерах из базы эталонных спектров. При идентификации неизвестного спектра по нему строится таблица частот, которая подается на входы ИНС. На одном из выходов ИНС появляется высокий уровень, свидетельствующий о принадлежности данного спектра определенному элементу.

Если распознавание по этому методу дало неоднозначный результат (высокий уровень на нескольких выходах ИНС), спектр можно частично или полностью распаковать и применить какой-либо другой способ распознавания. Частичная распаковка, позволяющая восстановить спектр в огрубленном виде, возможна благодаря особому порядку записи сжатых данных, суть которого состоит в том, что в упакованный образ спектра сначала записываются сжатые значения амплитуд двух крайних отсчетов, а затем в цикле – сжатые значения амплитуд отсчетов, находящихся посередине между двумя соседними уже

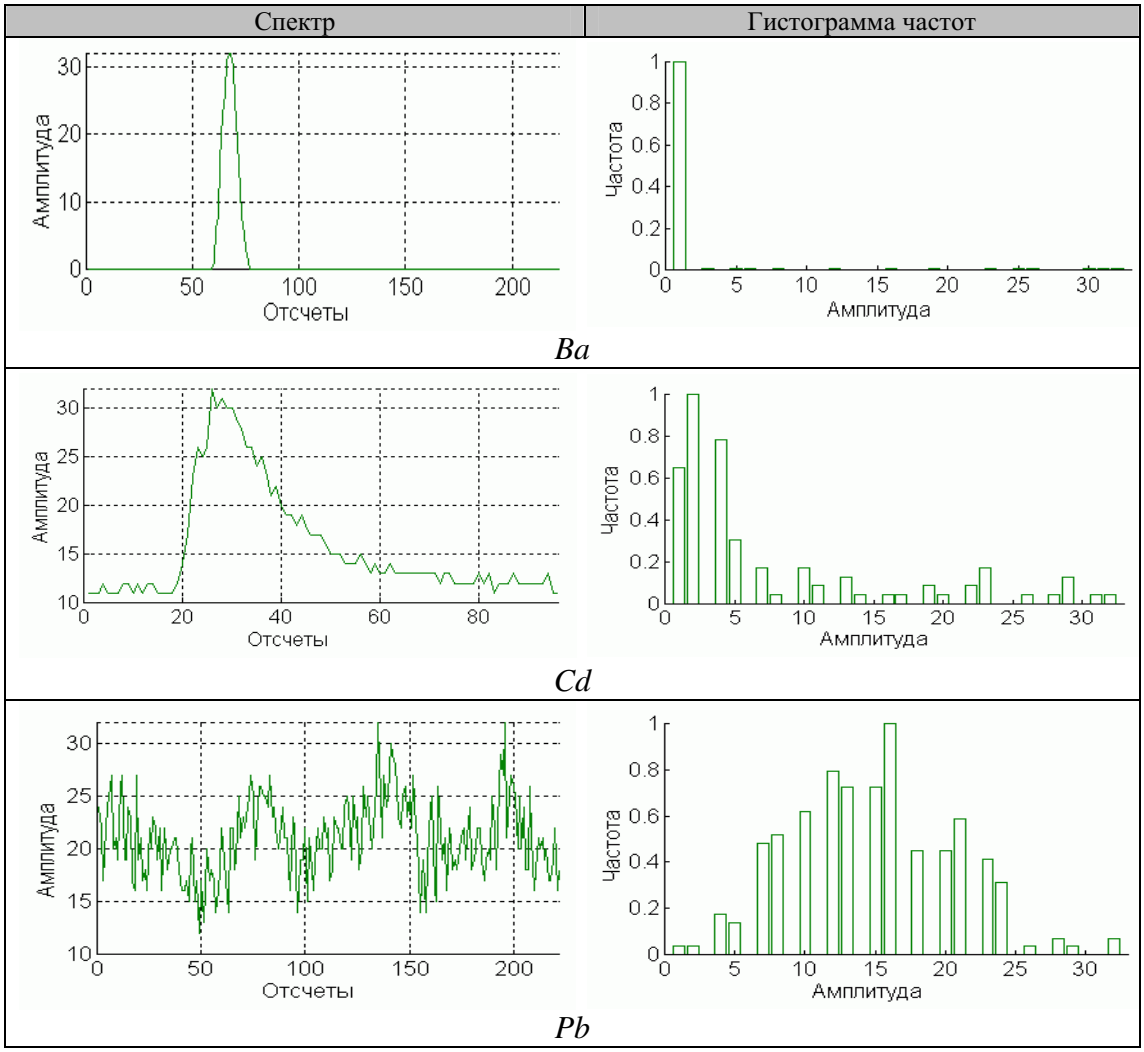


Рис. 5. Примеры эмиссионных спектров и соответствующие частоты встречаемости амплитуд для различных элементов.

записанными отсчетами. Сказанное иллюстрирует рис. 6.

Обычно из-за неточностей механики спектрометра и недостаточной стабильности свечения плазменного факела возникают горизонтальные и вертикальные смещения спектров, которые обычно вызывают определенные затруднения при идентификации элементов. В описанном методе эти недостатки практически не влияют на точность распознавания элементов.

Распаковка Хаффман-кодов является процессом, обратным сжатию: из сжатого файла считывается (или восстанавливается по таблице частот) кодовое дерево, а затем производится побитовое чтение входного файла. Считываемые биты определяют путь обхода дерева.

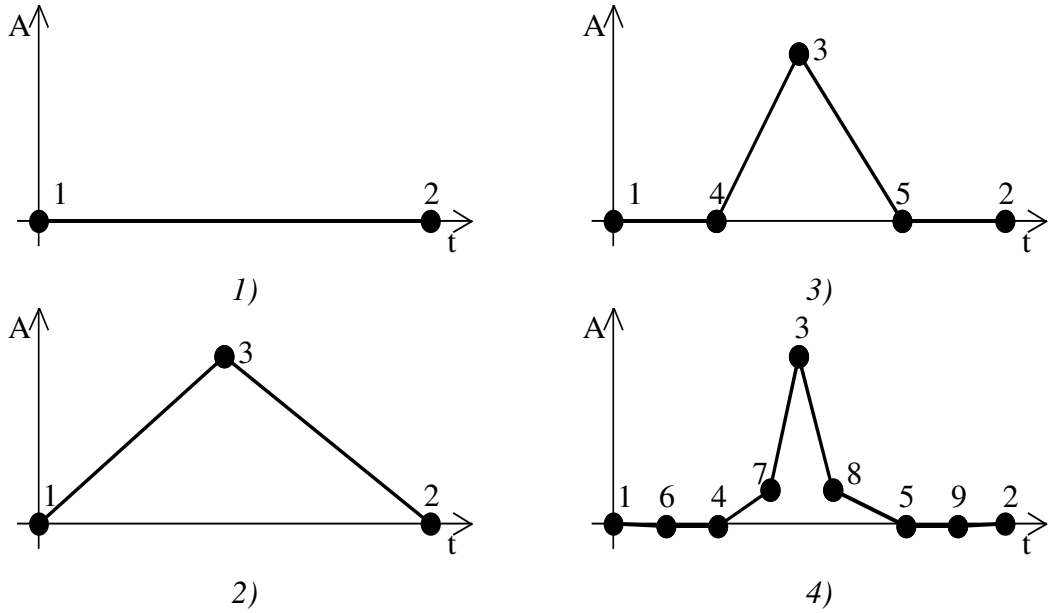


Рис. 6. Процесс постепенной распаковки спектра.
Цифрами указан порядок чтения отсчетов из сжатого спектра.

Обход начинается из корня и продолжается до тех пор, пока не будет достигнут один из листьев дерева. В этом случае в поток восстановленных данных записывается соответствующий данному листу символ, указатель текущего узла возвращается в корень дерева и процесс чтения/обхода возобновляется.

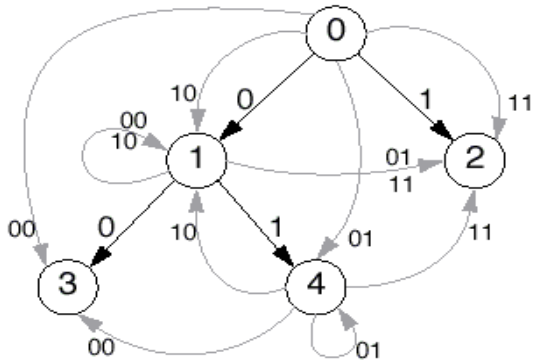
Алгоритм сжатия Хаффмана практически не применяется в чистом виде. Обычно он используется как один из этапов компрессии в сложных многоступенчатых схемах. Наилучшее сжатие достигается, когда значения частот входных символов имеют вид 2^X, где X – целое число. Характерной особенностью этого алгоритма является то, что он, в отличие от описанных выше методов, не увеличивает размер файла в худшем случае (если не учитывать необходимости хранения таблицы частот вместе с файлом).

Проблема поиска в сжатых методом Хаффмана данных и сопутствующие ей задачи рассматривались в [15-17].

Разумеется, побитовая обработка входного потока и необходимость пошагового обхода дерева замедляют процесс распаковки. Поэтому в [17] был предложен метод для быстрой распаковки кодов Хаффмана, а также – что особенно важно при поиске – способ определения границ отдельных символов в потоке сжатых данных.

Чтобы ускорить распаковку, можно читать и обрабатывать данные группами бит, то есть байтами. Для любого начального узла дерева и заданной последовательности бит конечный узел обхода дерева,

получающийся при быстром декодировании Хаффмана, определяется единственным образом. Пусть размер слова фиксирован и равен k битам. Тогда для быстрого обхода дерева может быть использована таблица размером 2k для каждого узла, исключая листья (см. рис. 7). Это позволяет эффективно переходить от одного узла кодового дерева к другому посредством чтения байт, а не отдельных бит.



а)

Слова

Начальные узлы	00	01	10	11
0	3	4	1	2
1	1	2	1	2
2	3	4	1	2
3	3	4	1	2
4	3	4	1	2

б)

Рис. 7. К пояснению метода быстрого декодирования Хаффман-кодов.
а) Кодовое дерево Хаффмана с изображенными межузловыми переходами с использованием слов размером 2 бита для кодов A=00, B=01, C=1. б) Полная таблица межузловых переходов, показывающая конечный узел для любой заданной комбинации начального узла и словом данных. Отметим, что все листья имеют такую же таблицу переходов, что и корневой узел, и могут не сохраняться.

Описанного способа быстрой навигации по двоичному дереву недостаточно для декомпрессии, т.к. для каждого перехода нужно знать соответствующий выходной символ. В примере, показанном на рис. 7, если начать с узла 1 и двухбитового слова 01, конечным узлом окажется узел 2, при этом декодировались символы A и C. Это может быть показано в таблице, похожей на таблицу переходов рис. 7, б. Табл. 4 отображает таблицу выходных символов для того же примера. Заметим, что листья в данном отношении эквивалентны корневому узлу и поэтому соответствующие записи могут быть опущены.

Таблица 4

Выходные символы для каждого перехода между узлами.

Начальный узел	Слово	Символы
0	00	A
0	01	B
0	10	C
0	11	C, C
1	00	A
1	01	A, C

Начальный узел	Слово	Символы
1	10	B
1	11	B, C

Процедура декодирования очень проста. Используя описанные выше структуры данных, распаковщик может просто читать поток сжатых данных по словам и обновлять текущую позицию в двоичном кодовом дереве в соответствии с таблицей переходов. Перед обновлением текущего узла нужно прочесть символы из таблицы выходных символов. Следует заметить, что кодовое дерево больше не используется, т.к. таблица межузловых переходов и выходная таблица полностью определяют процесс декодирования. Алгоритм быстрой распаковки кодов Хаффмана представлен в листинге 4.

```
PROCEDURE Decode (in: InputStream; code: CodeTree);
VAR
    byte: Byte;
    cur: Node;
BEGIN
    cur := code.root;
    WHILE NOT in.EOF() DO
        byte := in.readByte();
        code.outputSymbols(cur, byte);
        cur := code.nextNode(cur, byte);
    END of WHILE;
END;
```

Листинг 4. Псевдокод алгоритма быстрой распаковки кодов Хаффмана
CodeTree – структура, хранящая таблицу переходов и выходную таблицу.

Обработка сжатых данных по словам, а не побитно, важно не только для быстрой распаковки, но и при необходимости просмотреть сжатый файл и остановиться в указанной позиции. В частности, в [18] стояла задача последовательного чтения сжатых данных без непосредственной их распаковки и подсчета количества встреченных символов. Кроме того, при остановке в указанной позиции было нужно знать положение последнего считанного символа в потоке бит.

Рассмотрим способ решения этой задачи, описанный в [17].

Максимальное количество символов, кодируемых k битами, равно k. Поэтому поля из k бит будет достаточно, чтобы обозначать границы символа, представленного в кодах переменной длины, в слове размером k бит. Такое битовое поле нужно для каждого перехода между уз-

лами, чтобы иметь возможность узнать позиции бит в последнем считанном слове, где заканчиваются считанные закодированные символы (см. табл. 5.)

Таблица 5

Начальный узел	Слово	Поле завершений*	#**
0	00	[0, 1]	1
0	01	[0, 1]	1
0	10	[1, 0]	1
0	11	[1, 1]	2
1	00	[1, 0]	1
1	01	[1, 1]	2
1	10	[1, 0]	1
1	11	[1, 1]	2

*Битовое поле, показывающее завершения символов в обработанном слове.
**Количество закодированных символов в последнем считанном слове.

Чтобы удовлетворить второму требованию, т.е. узнать сколько символов было закодировано в определенной позиции потока сжатых данных, достаточно добавить только еще одну запись для каждого перехода между узлами. Значение этой записи фактически равно количеству единиц в описанном выше наборе бит, представляющем количество завершений символа в последнем считанном слове.

Алгоритм, показанный в листинге 5, позволяет просматривать сжатый файл без распаковки каждого символа, при этом ведя подсчет количества закодированных символов. Он также представляет возможность за постоянное время определить был ли и если был, то где, завершен символ в последнем прочитанном слове.

```
PROCEDURE Counting (in: InputStream; code: CodeTree);
VAR
    count, n, i: Integer;
    field: BitField;
    byte: Byte;
    cur: Node;
BEGIN
    cur := code.root;
    count := 0;
    n := 0;
    WHILE NOT условие остановки DO
        byte := in.readByte();
        INC(n);
        count := count + code.numberOfSymbols(cur, byte);
```

```
            field := code.endingsField(cur, byte);
            cur := code.nextNode(cur, byte);
        END;
        out.print(count, " символов закодировано");
        i := 8;
        WHILE NOT field[i] DO
            DEC(i);
        IF i > 0 THEN
            out.print("Последний символ заканчивается на ",
                    (n-1)*8+i, "-том бите");
        ELSE
            out.print("В последнем слове не заканчивается
                    ни один символ");
        END;
END;
```

Листинг 5. Алгоритм быстрого подсчета закодированных методом Хаффмана символов и определения их границ
В структуру CodeTree добавлены битовое поле завершений и запись количества символов для каждого межузлового перехода

Описанные выше алгоритмы могут оказывать существенную помощь при решении задач поиска в сжатых методом Хаффмана данных.

Арифметическое кодирование

Арифметическое кодирование [19] – один из алгоритмов статистического кодирования, похожий на алгоритм кодирования Хаффмана. Но здесь применяется иной подход к использованию частот символов, и это дает лучшие результаты, чем кодирование Хаффмана, где лучшее сжатие достигалось, когда частоты символов имели вид 2^X, где X – целое число.

Рассматриваемый алгоритм свободен от такого недостатка. Представление кодов не ограничивается целым числом бит. Поэтому, когда частоты символов произвольны, арифметическое кодирование имеет лучшую степень сжатия, чем кодирование Хаффмана.

Арифметическое кодирование является методом, позволяющим упаковывать символы входного алфавита без потерь при условии, что известно распределение частот этих символов и является наиболее оптимальным, т.к. достигается теоретическая граница степени сжатия.

Предполагаемая требуемая последовательность символов, при сжатии методом арифметического кодирования рассматривается как некоторая двоичная дробь из интервала [0, 1). Результат сжатия представляется как последовательность двоичных цифр из записи этой дроби.

Идея метода состоит в том, что исходный текст рассматривается как запись этой дроби, где каждый входной символ является "цифрой" с весом, пропорциональным вероятности его появления. Этим объясняется интервал, соответствующий минимальной и максимальной вероятностям появления символа в потоке.

Рассмотрим пример.

Пусть алфавит состоит из двух символов: a и b с вероятностями соответственно 0,75 и 0,25.

Рассмотрим начальный интервал вероятностей [0, 1). Разобьем его на части, длина которых пропорциональна вероятностям символов. В нашем случае это [0; 0,75) и [0,75; 1). Суть алгоритма в следующем: каждому слову во входном алфавите соответствует некоторый подынтервал из интервала [0, 1), а пустому слову – весь интервал [0, 1). После получения каждого следующего символа интервал уменьшается с выбором той его части, которая соответствует новому символу. Кодом цепочки является интервал, выделенный после обработки всех ее символов, точнее, двоичная запись любой точки из этого интервала, а длина полученного интервала пропорциональна вероятности появления кодируемой цепочки.

Процесс кодирования строки aaba отображен в табл. 6.

Таблица 6

Процесс арифметического кодирования

Шаг	Просмотренная цепочка	Интервал
0	нет	[0; 1)
1	a	[0; 0.75)
2	aa	[0; 0.5625)
3	aab	[0.421875; 0.5625)
4	aaba	[0.421875; 0.52734375)

Результатом кодирования является любое число из последнего интервала, например 0,5.

Алгоритм декодирования работает аналогично кодировщику. На вход поступает число 0,43 и производится разбиение интервала. Этот

процесс повторяется до тех пор, пока не будут декодированы все четыре символа. Чтобы можно было определить конец цепочки, можно либо сохранять ее длину отдельно, либо использовать дополнительный уникальный символ «конец цепочки».

Казалось бы, для данного алгоритма кодирования необходимо использовать арифметику с плавающей точкой над числами большой длины. Однако на практике хорошие результаты дает применение целочисленной арифметики к стандартным 16-ти и 32-х битным целым числам.

Несмотря на эффективность арифметического кодирования, оно пока не используется повсеместно. Это можно объяснить тем, что это довольно новый метод и он еще не нашел общего применения. Другая причина представляется более существенной: арифметическое сжатие требует больших вычислительных ресурсов (и процессорного времени, и памяти.)

К сожалению, возможностей поиска в данных, упакованных арифметическим кодированием, без предварительной распаковки, до настоящего времени не найдено. Возможно, это обусловлено тем, что рассмотренный метод позволяет достигать теоретической границы сжатия, при котором в упакованном файле фактически не остается никаких меток для алгоритмов поиска.

Заключение

В данной работе был произведен обзор нескольких распространенных алгоритмов сжатия и соответствующих им методов поиска/распознавания упакованных данных. Поиск непосредственно в сжатых данных без их распаковки зачастую обладает некоторыми преимуществами по сравнению с рядом обычных алгоритмов поиска.

В данное время исследования этой проблемы ведутся по нескольким направлениям:

- Изучение возможности поиска в данных, упакованных классическими алгоритмами. Как видно из данного обзора, это направление уже довольно хорошо разработано, хотя еще вполне возможно обнаружение полезных побочных эффектов у таких алгоритмов.
- Создание различных модификаций классических алгоритмов сжатия данных с целью обеспечить возможность или улучшить эффективность быстрого поиска/распознавания.
- Создание новых методов сжатия специально для решения описанной задачи. В этом направлении уже достигнуты некоторые успехи. В частности, перспективным представляется сжатие с использованием преобразования Бэрроуза-Уилера (Burrows-Wheeler) [20, 21] и использование для сжатия и последующего распознавания изображений искусственных нейронных сетей [22-26].

Обработка сжатых данных является обширной и многообещающей сферой приложения усилий ученых, поэтому исследования в этой области продолжаются.

Литература

1. *T. Eilam-Tsoreff and U. Vishkin*. Matching patterns in strings subject to multi-linear transformations. Theoretical Computer Science, 60(3):231-254, 1998
2. *A. Amir, G.M. Landau, D. Sokol*. Inplace Run-length 2D Compressed Search. Symposium on Discrete Algorithms, pages 817-818, 2000
3. *A. Amir and G. Benson*. Efficient two-dimensional matching. In Proc. Data Compression Conference, page 279, 1992
4. *A. Amir and G. Benson*. Two-dimensional periodicity and its application. In Proc. Of the 3rd Ann. ACM-SIAM Symp. On Discrete Algorithms, pages 440-452, 1992
5. *A. Amir, G.M. Benson and M. Farach*. Optimal two-dimensional compressed matching. Journal of Algorithms, 24(2):354-379, 1997
6. *A. Amir, G.M. Landau and U. Vishkin*. Efficient pattern matching with scaling. Journal of Algorithms, 13(1):2-32, 1992
7. *J. Ziv and A. Lempel*, IEEE Trans. Inform. Theory IT-23 (3): 337, 1977
8. *A. Amir, G. Benson, M. Farach*. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. Journal of Computer and System Sciences 52: 299-307, 1996
9. *M. Farach and M.Thorup*. String-matching in Lempel-Ziv compressed strings. In 27th ACM STOC, pages 703-713, 1995
10. *L. Gąsieniec, M.Karpinski, W. Plandowski and W. Rytter*. Efficient algorithms for Lempel-Ziv encoding. In Proc. 4th Scandinavian Workshop on Algorithm Theory, volume 1097 of Lecture Notes in Computer Science, pages 392-403. Springer-Verlag, 1996
11. *T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa***.** Multiple pattern matching in LZW compressed text. In Proc. Data Compression Conference '98, pages 103-112. IEEE Computer Society, 1998
12. *T. Kida, M. Takeda, A.Shinohara, S. Arikawa*. Shift-And Approach to Pattern matching in LZW Compressed Text. In Proc. of the10th Annual Symposium on Combinatorial Pattern Matching, Springer-Verlag, Berlin, 1999
13. Е.В. Макарова, А.М. Поплетеев, В.М. Лутковский. Алгоритмы автоматической идентификации химических элементов по оптическим спектрам. (Отправлено на Всероссийскую Научную Конференцию «Физика Радиоволн», Томск, Россия, 2002)
14. *Bishop M*. Neural Networks for Pattern Recognition. Oxford: Clarendon Press, 1997
15. *S. Fukamachi, T. Shinohara, and M. Takeda*. String pattern matching for compressed data using variable length codes. In Proc. Symposium on Informatics 1992, pages 95-103, 1992
16. *G. Jacobson*. Random access in Huffman-coded files. In Proc. Data Compression Conference, pages 268-277, IEEE, 1992
17. *R. Pajarola*. Fast Huffman Code Processing. UCI-ICS Technical Report No. 99-43, Department of Information & Computer science, University of California, Irvine,1999
18. *R. Pajarola and P.Widmayer.* Pattern matching in compressed raster images. In Third South American Workshop on String Processing WSP 1996, International Informatics Series 4, pages 228-242. Carleton University Press, 1996.

19. *I. Witten, R. Neal and J. Cleary*. Arithmetic Coding for Data Compression. Communications of the ACM, pages 520-540, 1987
20. *T. Bell, M. Powell, A. Mukherjee, D. Adjeroh*. Searching BWT compressed text with the Boyer-Moore algorithm and binary search. (Submitted to Data Compression Conference '2002)
21. *M. Powell*. Compressed-Domain Pattern Matching with the Burrows-Wheeler Transform. Honours Project Report, 2001
22. *M. Cao and C.-I Chang*. Image compression by neural network International Symposium on Neural Networks, National Chiao Tung University, Hsin Chu, Taiwan, Republic of China, 1995
23. *J. Jiang*. Design of neural networks for lossless data compression. Optical Engineering, Vol 35, No 7, pages 1837-1843, 1996
24. *J. Jiang*. Neural network based lossless image compression. In Proc. of Visual'96: International Conference on Visual Information Systems, Australia, pages 192-200, 1996
25. *J. Jiang*. Image compression with neural networks - A survey. Image Communication, ELSEVIER, vol 14,No 9, pages 737-760, 1999
26. *M. Mahoney*. Fast text compression with neural networks. In Proc. Of FLAIRS, Orlando, 2000