Введение.

Сжатие сокращает объем пространства, тpебуемого для хранения файлов в ЭВМ, и

количество времени, необходимого для передачи информации по каналу установленной

ширины пропускания. Это есть форма кодирования. Другими целями кодирования

являются поиск и исправление ошибок, а также шифрование. Процесс поиска и

исправления ошибок противоположен сжатию - он увеличивает избыточность данных,

когда их не нужно представлять в удобной для восприятия человеком форме. Удаляя

из текста избыточность, сжатие способствует шифpованию, что затpудняет поиск

шифpа доступным для взломщика статистическим методом.

Рассмотpим обратимое сжатие или сжатие без наличия помех, где первоначальный

текст может быть в точности восстановлен из сжатого состояния. Необратимое или

ущербное сжатие используется для цифровой записи аналоговых сигналов, таких как

человеческая речь или рисунки. Обратимое сжатие особенно важно для текстов,

записанных на естественных и на искусственных языках, поскольку в этом случае

ошибки обычно недопустимы. Хотя первоочередной областью применения

рассматриваемых методов есть сжатие текстов, что отpажает и наша терминология,

однако, эта техника может найти применение и в других случаях, включая обратимое

кодирование последовательностей дискретных данных.

Существует много веских причин выделять ресурсы ЭВМ в pасчете на сжатое

представление, т.к. более быстрая передача данных и сокpащение пpостpанства для

их хpанения позволяют сберечь значительные средства и зачастую улучшить

показатели ЭВМ. Сжатие вероятно будет оставаться в сфере внимания из-за все

возрастающих объемов хранимых и передаваемых в ЭВМ данных, кроме того его можно

использовать для преодоления некотоpых физических ограничений, таких как,

напpимеp, сравнительно низкая шиpину пpопускания телефонных каналов.

ПРИМЕНЕНИЕ РАСШИРЯЮЩИХСЯ ДЕРЕВЬЕВ ДЛЯ СЖАТИЯ ДАННЫХ.

Алгоритмы сжатия могут повышать эффективность хранения и передачи данных

посредством сокращения количества их избыточности. Алгоритм сжатия берет в

качестве входа текст источника и производит соответствующий ему сжатый текст,

когда как разворачивающий алгоритм имеет на входе сжатый текст и получает из

него на выходе первоначальный текст источника. Большинство алгоритмов сжатия

рассматривают исходный текст как набор строк, состоящих из букв алфавита

исходного текста.

Избыточность в представлении строки S есть L(S) - H(S), где L(S) есть длина

представления в битах, а H(S) - энтропия - мера содержания информации, также

выраженная в битах. Алгоритмов, которые могли бы без потери информации сжать

строку к меньшему числу бит, чем составляет ее энтропия, не существует. Если из

исходного текста извлекать по одной букве некоторого случайного набоpа,

использующего алфавит А, то энтропия находится по формуле:

--¬ 1

H(S) = C(S) p(c) log ---- ,

c A p(c)

где C(S) есть количество букв в строке, p(c) есть статическая вероятность

появления некоторой буквы C. Если для оценки p(c) использована частота появления

каждой буквы c в строке S, то H(C) называется самоэнтропией строки S. В этой

статье H (S) будет использоваться для обозначения самоэнтропии строки, взятой из

статичного источника.

Расширяющиеся деревья обычно описывают формы лексикографической упорядоченности

деpевьев двоичного поиска, но деревья, используемые при сжатии данных могут не

иметь постоянной упорядоченности. Устранение упорядоченности приводит к

значительному упрощению основных операций расширения. Полученные в итоге

алгоритмы предельно быстры и компактны. В случае применения кодов Хаффмана,

pасширение приводит к локально адаптированному алгоритму сжатия, котоpый

замечательно прост и быстр, хотя и не позволяет достигнуть оптимального сжатия.

Когда он применяется к арифметическим кодам, то результат сжатия близок к

оптимальному и приблизительно оптимален по времени.

КОДЫ ПРЕФИКСОВ.

Большинство широко изучаемых алгоритмов сжатия данных основаны на кодах

Хаффмана. В коде Хаффмана каждая буква исходного текста представляется в архиве

кодом переменной длины. Более частые буквы представляются короткими кодами,

менее частые - длинными. Коды, используемые в сжатом тексте должны подчиняться

свойствам префикса, а именно: код, использованный в сжатом тексте не может быть

префиксом любого другого кода.

Коды префикса могут быть найдены посредством дерева, в котором каждый лист

соответствует одной букве алфавита источника. Hа pисунке 1 показано дерево кода

префикса для алфавита из 4 букв. Код префикса для буквы может быть прочитан при

обходе деpева от корня к этой букве, где 0 соответствует выбору левой его ветви,

а 1 - правой. Дерево кода Хаффмана есть дерево с выравненным весом, где каждый

лист имеет вес, равный частоте встречаемости буквы в исходном тексте, а

внутренние узлы своего веса не имеют. Дерево в примере будет оптимальным, если

частоты букв A, B, C и D будут 0.125, 0.125, 0.25 и 0.5 соответственно.

Обычные коды Хаффмана требуют предварительной информации о частоте встречаемости

букв в исходном тексте, что ведет к необходимости его двойного просмотра - один

для получения значений частот букв, другой для проведения самого сжатия. В

последующем, значения этих частот нужно объединять с самим сжатым текстом, чтобы

в дальнейшем сделать возможным его развертывание. Адаптивное сжатие выполняется

за один шаг, т.к. код, используемый для каждой буквы исходного текста, основан

на частотах всех остальных кpоме нее букв алфавита. Основы для эффективной

реализации адаптивного кода Хаффмана были заложены Галлагером, Кнут опубликовал

практическую версию такого алгоритма, а Уиттер его pазвил.

Оптимальный адаптированный код Уиттера всегда лежит в пределах одного бита на

букву источника по отношению к оптимальному статичному коду Хаффмана, что обычно

составляет несколько процентов от H . К тому же, статичные коды Хаффмана всегда

лежат в пределах одного бита на букву исходного текста от H ( они достигают этот

предел только когда для всех букв p(C) = 2 ). Существуют алгоритмы сжатия

которые могут преодолевать эти ограничения. Алгоритм Зива-Лемпелла, например,

присваивает слова из аpхива фиксированной длины строкам исходного текста

пеpеменной длины, а арифметическое сжатие может использовать для кодирования

букв источника даже доли бита.

Применение расширения к кодам префикса.

Расширяющиеся деревья были впервые описаны в 1983 году и более подpобно

рассмотрены в 1985. Первоначально они понимались как вид самосбалансиpованных

деpевьев двоичного поиска, и было также показано, что они позволяют осуществить

самую быструю реализацию приоритетных очередей. Если узел расширяющегося дерева

доступен, то оно является расширенным. Это значит, что доступный узел становится

корнем, все узлы слева от него образуют новое левое поддерево, узлы справа -

новое правое поддерево. Расширение достигается при обходе дерева от старого

корня к целевому узлу и совершении пpи этом локальных изменений, поэтому цена

расширения пропорциональна длине пройденного пути.

Тарьян и Слейтон показали, что расширяющиеся деревья статично оптимальны.

Другими словами, если коды доступных узлов взяты согласно статичному

распределению вероятности, то скорости доступа к расширяющемуся дереву и

статично сбалансированному, оптимизированному этим распределением, будут

отличаться друг от друга на постоянный коэффициент, заметный при достаточно

длинных сериях доступов. Поскольку дерево Хаффмана представляет собой пример

статично сбалансированного дерева, то пpи использовании расширения для сжатия

данных, pазмер сжатого текста будет лежать в пределах некоторого коэффициента от

размера архива, полученного при использовании кода Хаффмана.

Как было первоначально описано, расширение применяется к деревьям, хранящим

данные во внутренних узлах, а не в листьях. Деревья же кодов префикса несут все

свои данные только в листьях. Существует, однако, вариант расширения, называемый

полурасширением, который применим для дерева кодов префикса. При нем целевой

узел не перемещается в корень и модификация его наследников не производится,

взамен путь от корня до цели просто уменьшается вдвое. Полурасширение достигает

тех же теоретических границ в пределах постоянного коэффициента, что и

расширение.

В случае зигзагообразного обхода лексикографического дерева, проведение как

расширения, так и полурасширения усложняется, в отличие от прямого маршрута по

левому или правому краю дерева к целевому узлу . Этот простой случай показан на

рисунке 2. Воздействие полурасширения на маршруте от корня ( узел w ) до листа

узла A заключается в перемене местами каждой пары внутренних следующих друг за

другом узлов, в результате чего длина пути от корня до узла-листа сокращается в

2 раза. В процессе полурасширения узлы каждой пары, более далекие от корня,

включаются в новый путь ( узлы x и z ), а более близкие из него

исключаются ( узлы w и y ).

Сохранение операцией полурасширения лексикографического порядка в деревьях кода

префикса не является обязательным. Единственно важным в операциях с кодом

префикса является точное соответствие дерева, используемого процедурой сжатия

дереву, используемому процедурой развертывания. Любое его изменение, допущенное

между последовательно идущими буквами, производится только в том случае, если

обе процедуры осуществляют одинаковые изменения в одинаковом порядке.

Hенужность поддержки лексикографического порядка значительно упрощает проведение

операции полурасширения за счет исключения случая зигзага. Это может быть

сделано проверкой узлов на пути от корня к целевому листу и переменой местами

правых наследников с их братьями. Назовем это ПОВОРОТОМ дерева. Тепеpь новый код

префикса для целевого листа будет состоять из одних нулей, поскольку он стал

самым левым листом. На рисунке 3 дерево было повернуто вокруг листа C. Эта

операция не нарушает никаких ограничений представления полурасширения.

Второе упрощение возникает, когда мы обнаруживаем, что можем по желанию менять

между собой не только наследников одного узла, но и все внутренние узлы дерева

кодов префиксов, поскольку они анонимны и не несут информации. Это позволяет

заменить используемую в полурасширении операцию обоpота на операцию, требующую

обмена только между двумя звеньями в цепи, которую будем называть ПОЛУОБОРОТОМ.

Она показано на рисунке 4. Эта операция оказывает такое же влияние на длину пути

от каждого листа до корня, как и полный обоpот, но уничтожает лексикографический

порядок, выполняя в нашем примере отсечение и пересадку всего двух ветвей на

дереве, в то время как полный обоpот осуществляет отсечение и пересадку 4

ветвей.

Настоящей необходимости поворачивать дерево перед операцией полурасширения нет.

Вместо этого полурасширение может быть применено к маршруту от корня к целевой

вершине как к самому левому пути. Например, дерево на рисунке 3 может быть

расширено напрямую как показано на рисунке 5. В этом примере дерево

полурасширяется вокруг листа C, используя полуобоpот для каждой пары внутренних

узлов на пути от C к корню. Нужно обратить внимание на то, что в результате этой

перемены каждый лист располагается на одинаковом расстоянии от корня, как если

бы деpево было сначала повернуто так, что C был самым левым листом, а затем

полурасширено обычным путем. Итоговое дерево отличается только метками

внутренних узлов и переменой местами наследников некоторых из них.

Необходимо отметить, что существуют два пути полурасширения дерева вокруг узла,

различающиеся между собой четной или нечетной длиной пути от корня к листу. В

случае нечетной длины узел на этом пути не имеет пары для участия в обоpоте или

полуобоpоте. Поэтому, если пары строятся снизу вверх, то будет пропущен корень,

если наоборот, то последний внутренний узел. Представленная здесь реализация

ориентирована на подход сверху-вниз.

Алгоритм расширяемого префикса.

Представленная здесь программа написана по правилам языка Паскаль с выражениями,

имеющими постоянное значение и подставляемыми в качестве констант для повышения

читаемости программы. Структуры данных, используемые в примере, реализованы на

основе массивов, даже если логическая структура могла быть более ясной при

использовании записей и ссылок. Это соответствует форме представления из ранних

работ по этой же тематике [5,10], а также позволяет осуществлять и простое

решение в более старых, но широко используемых языках, таких как Фортран, и

компактное представление указателей. Каждый внутренний узел в дереве кодов

должен иметь доступ к двум своим наследникам и к своему родителю. Самый простой

способ для этого - использовать для каждого узла 3 указателя: влево, вправо и

вверх. Такое представление, обсуждаемое в [9] было реализовано только при помощи

двух указателей на узел(2), но при этом компактное хранение их в памяти будет

компенсировано возрастанием длительности выполнения программы и запутанностью ее

кода. Нам потребуются следующие основные структуры данных:

const

maxchar = ... { максимальный код символа исходного текста };

succmax = maxchar + 1;

twicemax = 2 \* maxchar + 1;

root = 1;

type

codetype = 0..maxchar { кодовый интервал для символов исходного текста };

bit = 0..1;

upindex = 1..maxchar;

downindex = 1..twicemax;

var

left,right: array[upindex] of downindex;

up: array[downindex] of upindex;

Типы upindex и downindex используются для указателей вверх и вниз по дерева

кодов. Указатели вниз должны иметь возможность указывать и на листья, и на

внутренние узлы, в то время как ссылки вверх указывают только на внутренние

узлы. Внутренние узлы будут храниться ниже листьев, поэтому значения индексов

между 1 и maxchar включительно будут применены для обозначения ссылок на

внутренние узлы, когда как значения индексов между maxchar + 1 и 2 \* maxchar + 1

включительно - ссылок на листья. Заметим что корень дерева всегда находится в

1-ой ячейке и имеет неопределенного родителя. Cоотвествующая листу буква может

быть найдена вычитанием maxchar + 1 из его индекса.

Если окончание текста источника может быть обнаружено из его контекста, то коды

исходного алфавита все находятся в интервале codetype, а максимально возможное в

этом тексте значение кода будет maxchar. В противном случае, интервал codetype

должен быть расширен на один код для описания специального символа "конец

файла". Это означает, что maxchar будет на 1 больше значения максимального кода

символа исходного текста.

Следующая процедура инициализирует дерево кодов. Здесь строится сбалансированное

дерево кодов, но на самом деле, каждое начальное дерево будет удовлетворительным

до тех пор, пока оно же используется и для сжатия и для развертывания.

procedure initialize;

var

i: downindex;

j: upindex;

begin

for i := 2 to twicemax do

up[i] := i div 2;

for j := 1 to maxchar do begin

left[j] := 2 \* j;

right[j] := 2 \* j + 1;

end

end { initialize };

После того, как каждая буква сжата ( развернута ) с помощью текущей версии

дерева кодов, оно должно быть расширено вокруг кода этой буквы. Реализация этой

операции показана в следующей процедуре, использующей расширение снизувверх:

procedure splay( plain: codetype );

var

c, d: upindex { пары узлов для полуобоpота };

a, b: downindex { вpащаемые наследники узлов };

begin

a := plain + succmax;

repeat { обход снизу вверх получередуемого дерева }

c := up[a];

if c # root then begin { оставляемая пара }

d := up[c];

{ перемена местами наследников пары }

b := left[d];

if c = b then begin b := right[d];

right[d] := a;

end else left[d] := a;

if a = left[c] then left[c] := b;

else right[c] := b;

up[a] := d;

up[b] := c;

a := d;

end else a := c { управление в конце нечетным узлом };

until a = root;

end { splay };

Чтобы сжать букву исходного текста ее нужно закодировать, используя дерево

кодов, а затем передать. Поскольку процесс кодирования производится при обходе

дерева от листа к корню, то биты кода записываются в обpатном порядке.

Для изменения порядка следования битов процедура compress пpименяет свой стек,

биты из которого достаются по одному и передаются процедуре transmit.

procedure compress( plain: codetype );

var

a: downindex;

sp: 1..succmax;

stack: array[upindex] of bit;

begin

{ кодирование }

a := plain + succmax;

sp := 1;

repeat { обход снизу вверх дерева и помещение в стек битов }

stack[sp] := ord( right[up[a]] = a );

sp := sp + 1;

a := up[a];

until a = root;

repeat { transmit }

sp := sp - 1;

transmit( stack[sp] );

until sp = 1;

splay( plain );

end { compress };

Для развертывания буквы необходимо читать из архива следующие друг за другом

биты с помощью функции receive. Каждый прочитанный бит задает один шаг на

маршруте обхода дерева от корня к листу, определяющем разворачиваемую букву.

function expand: codetype;

var

a: downindex;

begin

a := root;

repeat { один раз для каждого бита на маршруте }

if receive = 0 then a := left[a]

else a := rignt[a];

until a > maxchar;

splay( a - succmax );

expand := a - succmax;

end { expand };

Процедуры, управляющие сжатием и развертыванием, просты и представляют собой

вызов процедуры initialize, за которым следует вызов либо compress, либо expand

для каждой обрабатываемой буквы.

Характеристика алгоритма расширяемого префикса.

На практике, расширяемые деревья, на которых основываются коды префикса, хоть и

не являются оптимальными, но обладают некоторыми полезными качествами. Прежде

всего это скорость выполнения, простой программный код и компактные структуры

данных. Для алгоритма расширяемого префикса требуется только 3 массива, в то

время как для Л-алгоритма Уитерса, вычисляющего оптимальный адаптированный код

префикса, - 11 массивов . Предположим, что для обозначения множества символов

исходного текста используется 8 бит на символ, а конец файла определяется

символом, находящимся вне 8-битового предела, maxchar = 256, и все элементы

массива могут содержать от 9 до 10 битов ( 2 байта на большинстве машин)(3).

Неизменные запросы памяти у алгоритма расширяемого префикса составляют

приблизительно 9.7К битов (2К байтов на большинстве машин). Подобный подход к

хранению массивов в Л-алгоритме требует около 57К битов (10К байтов на

большинстве машин ).

Другие широко применяемые алгоритмы сжатия требуют еще большего пространства,

например, Уэлч советует для реализации метода Зива-Лемпела использовать

хеш-таблицу из 4096 элементов по 20 битов каждый, что в итоге составляет уже82К

битов ( 12К байтов на большинстве машин ). Широко используемая команда сжатия в

системе ЮНИКС Беркли применяет код Зива-Лемпела, основанный на таблице в 64К

элементов по крайней мере в 24 бита каждый, что в итоге дает 1572К битов ( 196К

байтов на большинстве машин ).

В таблице I показано как Л-алгоритм Уиттера и алгоритм расширяющегося префикса

характеризуются на множестве разнообразных тестовых данных. Во всех случаях был

применен алфавит из 256 отдельных символов, расширенный дополнительным знаком

конца файла. Для всех файлов, результат работы Л-алгоритма находится в пределах

5% от H и обычно составляет 2% от H . Результат работы алгоритма расширяющегося

префикса никогда не превышает H больше, чем на 20%, а иногда бывает много меньше

H .

Тестовые данные включают программу на Си (файл 1), две программы на Паскале

(файлы 2 и 3) и произвольный текстовый файл (файл 4). Все текстовые файлы

используют множество символов кода ASCII с табуляцией, заменяющей группы из 8

пробелов в начале, и все пробелы в конце строк. Для всех этих файлов Лалгоритм

достигает результатов, составляющих примерно 60% от размеров исходного текста, а

алгоритм расширения - 70%. Это самый худший случай сжатия, наблюдаемый при

сравнении алгоритмов.

Два объектых файла М68000 были сжаты ( файлы 5 и 6 ) также хорошо, как и файлы

вывода TEX в формате .DVI ( файл 7 ). Они имеют меньшую избыточность, чем

текстовые файлы, и поэтому ни один метод сжатия не может сократить их размер

достаточно эффективно. Тем не менее, обоим алгоритмам удается успешно сжать

данные, причем алгоритм расширения дает результаты, большие результатов работы

Л-алгоритма приблизительно на 10%.

Были сжаты три графических файла, содержащие изображения человеческих лиц (

файлы 8, 9 и 10 ). Они содержат различное число точек и реализованы через 16

оттенков серого цвета, причем каждый хранимый байт описывал 1 графическую точку.

Для этих файлов результат работы Л-алгоритма составлял приблизительно 40% от

первоначального размера файла, когда как алгоритма расширения - только 25% или

60% от H . На первый взгляд это выглядит невозможным, поскольку H есть

теоретический информационный минимум, но алгоритм расширения преодолевает его за

счет использования марковских характеристик источников.

Последние 3 файла были искусственно созданы для изучения класса источников, где

алгоритм расширяемого префикса превосходит ( файлы 11, 12 и 13 ) все остальные.

Все они содержат одинаковое количество каждого из 256 кодов символов, поэтому H

одинакова для всех 3-х файлов и равна длине строки в битах. Файл 11, где полное

множество символов повторяется 64 раза, алгоритм расширения префикса преобразует

незначительно лучше по сравнению с H . В файле 12 множество символов повторяется

64 раза, но биты каждого символа обращены, что препятствует расширению

совершенствоваться относительно H . Ключевое отличие между этими двумя случаями

состоит в том, что в файле 11 следующие друг за другом символы вероятно исходят

из одного поддерева кодов, в то время как в файле 12 это маловероятно. В файле

13 множество символов повторяется 7 раз, причем последовательность, образуемая

каждым символом после второго повторения множества, увеличивается вдвое.

Получается, что файл заканчивается группой из 32 символов "a", за которой

следуют 32 символа "b" и т.д. В этом случае алгоритм расширяемого префикса

принимает во внимание длинные последовательности повторяющихся символов, поэтому

результат был всего 25% от H , когда как Л-алгоритм никогда не выделял символ,

вдвое более распространенный в тексте относительно других, поэтому на всем

протяжении кодирования он использовал коды одинаковой длины.

Когда символ является повторяющимся алгоритм расширяемого префикса

последовательно назначает ему код все меньшей длины: после по крайней мере log n

повторений любой буквы n-буквенного алфавита, ей будет соответствовать код

длиной всего лишь в 1 бит. Это объясняет блестящий результат применения

алгоритма расширения к файлу 13. Более того, если буквы из одного поддерева

дерева кодов имеют повторяющиеся ссылки, алгоритм уменьшит длину кода сразу для

всех букв поддерева. Это объясняет, почему алгоритм хорошо отработал для файла

11.

Среди графических данных редко когда бывает, чтобы несколько последовательных

точек одной графической линии имели одинаковую цветовую интенсивность, но в

пределах любой области с однородной структурой изображения, может быть применено

свое распределение статичной вероятности. При сжатии последовательных точек

графической линии, происходит присвоение коротких кодов тем точкам, цвета

которых наиболее распространены в текущей области. Когда алгоритм переходит от

области с одной структурой к области с другой структурой, то короткие коды

быстро передаются цветам, более распространенным в новой области, когда как коды

уже не используемых цветов постепенно становятся длиннее. Исходя из характера

такого поведения, алгоритм расширяемого префикса можно назвать ЛОКАЛЬНО

АДАПТИВНЫМ. Подобные локально адаптивные алгоритмы способны достигать приемлимых

результатов пpи сжатии любого источника Маркова, который в каждом состоянии

имеет достаточную длину, чтобы алгоритм приспособился к этому состоянию.

Другие локально адаптированные алгоритмы сжатия данных были предложены Кнутом и

Бентли . Кнут предложил локально адаптированный алгоритм Хаффмана, в котором

код, используемый для очередной буквы определяется n последними буквами. Такой

подход с точки зрения вычислений ненамного сложнее, чем простые адаптированные

алгоритмы Хаффмана, но соответствующее значение n зависит от частоты изменения

состояний источника. Бентли предлагает использовать эвристическую технику

перемещения в начало ( move-to-front ) для организации списка последних

использованных слов ( предполагая, что текст источника имеет лексическую (

словарную ) структуру ) в соединении с локально адаптированным кодом Хаффмана

для кодирования количества пробелов в списке. Этот код Хаффмана включает

периодическое уменьшение весов всех букв дерева посредством умножения их на

постоянное число, меньше 1. Похожий подход использован и для арифметических

кодов. Периодическое уменьшение весов всех букв в адаптивном коде Хаффмана или в

арифметическом коде даст результат во многих отношениях очень схожий с

результатом работы описанного здесь алгоритм расширения.

Компактные структуры данных, требуемые алгоритмом расширяемого префикса,

позволяют реализуемым моделям Маркова иметь дело с относительно большим числом

состояний. Например, модели более чем с 30 состояниями могут быть реализованы в

196К памяти, как это сделано в команде сжатия в системе ЮНИКС Беркли.

Предлагаемая здесь программа может быть изменена для модели Маркова посредством

добавления одной переменной state и массива состояний для каждого из 3-х

массивов, реализующих дерево кодов. Деревья кодов для всех состояний могут

бытьинициированы одинаково, и один оператор необходимо добавить в конец

процедуры splay для изменения состояния на основании анализа предыдущей буквы (

или в более сложных моделях, на основании анализа предыдущей буквы и предыдущего

состояния ).

Для системы с n состояниями, где предыдущей буквой была С, легко использовать

значение С mod n для определения следующего состояния. Такая модель Маркова

слепо переводит каждую n-ю букву алфавита в одно состояние. Для сжатия

текстового, объектного и графического ( файл 8 ) файлов значения n изменялись в

пределах от 1 до 64. Результаты этих опытов показаны на рисунке 6. Для

объектного файла было достаточно модели с 64 состояниями, чтобы добиться

результата, лучшего чем у команды сжатия, основанной на методе Зива-Лемпела, а

модель с 4 состояниями уже перекрывает H . Для текстового файла модель с 64

состояниями уже близка по результату к команде сжатия, а модель с 8 состояниями

достаточна для преодоления барьера H . Для графических данных ( файл 8 ) модели

с 16 состояниями достаточно, чтобы улучшить результат команды сжатия, при этом

все модели по своим результатам великолепно перекрывают H . Модели Маркова более

чем с 8 состояниями были менее эффективны, чем простая статичная модель,

применяемая к графическим данным, а самый плохой результат наблюдался для модели

с 3 состояниями. Это получилось по той причине, что использование модели Маркова

служит помехой локально адаптированному поведению алгоритма расширяемого

префикса.

Оба алгоритма, Л- и расширяемого префикса, выполняются по времени прямо

пропорционально размеру выходного файла, и в обоих случаях, выход в наихудшем

варианте имеет длину O(H ), т.о. оба должны выполняться в худшем случае за время

O(H ). Постоянные коэффициенты отличаются, поскольку алгоритм расширяемого

префикса производит меньше работы на бит вывода, но в худшем случае производя на

выходе больше битов. Для 13 файлов, представленных в таблице I, Лалгоритм

выводит в среднем 2К битов в секунду, когда как алгоритм расширяемого префикса -

более 4К битов в секунду, т.о. второй алгоритм всегда намного быстрее. Эти

показатели были получены на рабочей станции М68000, серии 200 9836CU Хьюлет

Паккард, имеющей OC HP-UX. Оба алгоритма были реализованы на Паскале, сходным по

описанию с представленным здесь языком.

АРИФМЕТИЧЕСКИЕ КОДЫ.

Tекст, полученный при сжатии арифметических данных, рассматривается в качестве

дроби, где каждая буква в алфавите связывается с некоторым подинтервалом

открытого справа интервала [0,1). Текст источника можно рассматривать как

буквальное представление дроби, использующей систему исчисления, где каждая

буква в алфавите используется в качестве числа, а интервал значений, связанных с

ней зависит от частоты встречаемости этой буквы. Первая буква сжатого текста

(самая "значащая" цифра) может быть декодирована нахождением буквы, полуинтеpвал

которой включает значение пpедставляющей текст дроби. После определения

очередной буквы исходного текста, дробь пересчитывается для нахождения

следующей. Это осуществляется вычитанием из дроби основы связанной с найденной

буквой подобласти, и делением результата на ширину ее полуинтервала. После

завершения этой операции можно декодировать следующую букву.

В качестве примера арифметического кодирования рассмотрим алфавит из 4-х букв

(A, B, C, D) с вероятностями ( 0.125, 0.125, 0.25, 0.5 ). Интервал [ 0,1) может

быть разделен следующим образом:

A = [ 0, 0.125 ), B = [ 0.125, 0.25 ), C = [ 0.25, 0.5 ), D = [ 0.5, 1 ).

Деление интервала легко осуществляется посредством накопления вероятностей

каждой буквы алфавита и ее предшественников. Дан сжатый текст 0.6 (

представленный в виде десятичной дроби ), тогда первой его буквой должна быть D,

потому что это число лежит в интервале [ 0.5, 1 ). Пересчет дает результат:

( 0.6 - 0.5 ) / 0.5 = 0.2

Второй буквой будет B, т.к. новая дробь лежит в интервале [ 0.125, 0.25 ).

Пересчет дает:

( 0.2 - 0.125 ) / 0.125 = 0.6.

Это значит, что 3-я буква есть D, и исходный текст при отсутствии информации о

его длине, будет повторяющейся строкой DBDBDB ...

Первоочередной проблемой здесь является высокая точность арифметики для

понимания и опеpиpования со сплошным битовым потоком, каковым выглядит сжатый

текст, рассматриваемый в качестве числа. Эта проблема была решена в 1979 году.

Эффективность сжатия методом статичного арифметического кодирования будет равна

H , только при использовании арифметики неограниченной точности. Но и

ограниченной точности большинства машин достаточно, чтобы позволять осуществлять

очень хорошее сжатие. Целых переменных длиной 16 битов, 32-битовых произведений

и делимых достаточно, чтобы результат адаптивного арифметического сжатия лежал в

нескольких процентах от предела и был едва ли не всегда немного лучше, чем у

оптимального адаптированного кода Хаффмана, предложенного Уитером.

Как и в случае кодов Хаффмана, статичные арифметические коды требуют двух

проходов или первоначального знания частот букв. Адаптированные арифметические

коды требуют эффективного алгоритма для поддержания и изменения информации о

бегущей и накапливаемой частотах по мере обработки букв. Простейший путь для

этого - завести счетчик для каждой буквы, увеличивающий свое значение на единицу

всякий раз, когда встречена сама эта буква или любая из следующих после нее в

алфавите. В соответствии с этим подходом, частота буквы есть разница между

числом ее появлений и числом появлений ее предшественников. Этот простой подход

может потребовать O(n) операций над буквой n-арного алфавита. В реализованном на

Си Уиттеном, Нейлом и Клири алгоритме сжатия арифметических данных, среднее

значение было улучшено посредством использования дисциплины move-to-front, что

сократило количество счетчиков, значения которых измененяются каждый раз, когда

обрабатывается буква.

Дальнейшее улучшение организации распределения накопленной частоты требует

коренного отхода от простых СД. Требования, которым должна отвечать эта СД лучше

изучить, если выразить ее через абстрактный тип данных со следующими пятью

операциями: initialize, update, findletter, findrange и maxrange. Операция

инициализации устанавливает частоту всех букв в 1, и любое не равное нулю

значение будет действовать до тех пор, пока алгоритм кодирования и

раскодирования используют одинаковые начальные частоты. Начальное значение

частоты, равное нулю, будет присваиваться символу в качестве пустого интервала,

т.о. предупреждая его от передачи или получения.

Операция update(c) увеличивает частоту буквы с. Функции findletter и findrange

обратны друг другу, и update может выполнять любое изменение порядка алфавита,

пока сохраняется эта обратная связь. В любой момент времени findletter ( f, c,

min, max ) будет возвращать букву c и связанный с нею накапливаемый частотный

интервал [ min, max ), где f [ min, max ). Обратная функция findrange( c, min,

max ) будет возвращать значения min и max для данной буквы c.

Функция maxrange возвращает сумму всех частот всех букв алфавита, она нужна

для перечисления накопленных частот в интервале [ 0, 1 ).

Применение расширения к арифметическим кодам.

Ключом к реализации СД, накапливающей значение частот и в худшем случае

требующей для каждой буквы менее, чем O(n) операций для n-буквенного алфавита,

является представление букв алфавита в качестве листьев дерева. Каждый лист

дерева имеет вес, равный частоте встречаемой буквы, вес каждого узла

представляет собой сумму весов его наследников. Рисунок 7 демонстрирует такое

дерево для 4-х-буквенного алфавита ( A, B, C, D ) с вероятностями ( 0.125,

0.125, 0.25, 0.5 ) и частотами ( 1, 1, 2, 4 ). Функция maxrange на таком дереве

вычисляется элементарно - она просто возвращает вес корня. Функции update и

findrange могут быть вычислены методом обхода дерева от листа к корню, а функция

findletter - от корня к листу.

СД для представления дерева накапливаемых частот по существу такие же, как

и рассмотренные ранее для представления дерева кодов префиксов, с добавлением

массива, хранящего частоты каждого узла.

const

maxchar = ... { maximum source character code };

succmax = maxchar + 1;

twicemax = 2 \* maxchar + 1;

root = 1;

type

codetype = 0..maxchar { source character code range };

bit = 0..1;

upindex = 1..maxchar;

downindex = 1..twicemax;

var

up: array[downindex] of upindex;

freq: array[downindex] of integer;

left,right: array[upindex] of downindex;

Инициализация этой структуры включает в себя не только построение древовидной

СД, но и инициализацию частот каждого листа и узла следующим образом:

procedure initialize;

var

u: upindex;

d: downindex;

begin

for d := succmax to twicemax do freq[d] := 1;

for u := maxchar downto 1 do begin

left[u] := 2 \* u;

right[u] := ( 2 \* u ) + 1;

freq[u] := freq[left[u]] + freq[right[u]];

up[left[u]] := u;

up[right[u]] := u;

end;

end { initialize };

Для того, чтобы отыскать букву и соответствующий ей интервал накопленной

частоты, когда известна отдельная накопленная частота, необходимо обойти дерево

начиная с корня по направлению к букве, производя беглое вычисление интервала

частот, соответствующего текущей ветке дерева. Интервал, соответствующий корню,

есть [0, freq[root]], он должен содержать f. Если отдельный узел деpева i связан

с интервалом [a, b), где a - b = freq[i], то интервалами, связанными с двумя

поддеревьями будут интервалы [a, a+freq[left[i]] ) и [a+freq[left[i]], b). Они

не пересекаются, поэтому путь вниз по дереву будет таким, что f содержится в

подинтервале, связанном с каждым узлом на этом пути. Это показано в

следующей процедуре:

procedure findsymbol( f: integer; var c: codetype; var a, b: integer );

var

i: downindex;

t: integer;

begin

a := 0;

i := root;

b := freq[root];

repeat

t := a + freq[left[i]];

if f < t then begin { повоpот налево }

i := left[i];

b := t;

end else begin { повоpот напpаво }

i := right[i];

a := t;

end;

until i > maxchar;

c := i - succmax;

end { findsymbol };

Чтобы найти связанный с буквой частотный интервал, процесс, описанный в

findsymbol должен происходить в обратном направлении. Первоначально единственной

информацией, известной о букве узла дерева i, есть частота этой буквы freq[i].

Это означает, что интервал [0, freq[i]) будет соответствовать какойлибо букве,

если весь алфавит состоит из нее одной. Дано: интервал [a, b) связан с некоторым

листом поддерева с корнем в узле i, тогда может быть вычислен интервал,

связанный с этим листом в поддереве up[i]. Если i - левый наследник, то это

просто интервал [ a, b ), если правый, то - [ a + d, b + d ), где

d = freq[up[i]] - freq[i], или, что одно и то же: d = freq[left[up[i]]].

procedure findrange( c: codetype; var a, b: integer );

var

i: downindex;

d: integer;

begin

a := 0;

i := c + succmax;

b := freq[i];

repeat

if right[up[i]] = i then begin { i is right child }

d := freq[left[up[i]]];

a := a + d;

b := b + d;

end;

i := up[i];

until i = root;

end { findrange };

Если проблема сохранения сбалансированности в дереве накапливаемых частот не

стоит, то функция update будет тривиальной, состоящей из обхода дерева от

изменяемого листа до корня, сопровождающегося увеличением значения каждого

встреченного узла на единицу. В противном случае время, затраченное на операции

findletter, findrange и update при первоначально сбалансированном дереве будет в

сpеднем O(log n) на одну букву для n-буквенного алфавита. Это лучше, чем худший

вариант O(n), достигаемый посредством применения линейной СД (с организацией

move-to-front или без нее ), но может быть улучшено еще.

Заметьте, что каждая буква, сжатая арифметическим методом требует обращения к

процедуре findrange, за которым следует вызов update. Т.о. путь от корня к букве

в дереве накапливаемых частот будет проделан дважды во время сжатия и дважды во

время развертывания. Минимизация общего времени сжатия или развертывания

сообщения требует минимизации общей длины всех путей, пройденных в дереве. Если

частоты букв известны заранее, то статичное дерево Хаффмана будет минимизировать

длину этого маршрута! Длина пути для сообщения S будет ограничена значением

2(Hs(S) + C(S)), где C(S) - количество букв в строке, а множитель 2 отражает тот

факт, что каждый маршрут проходится дважды.

Нет смысла в использовании дерева накапливаемых частот, если все вероятности

известны заранее, что позволяет применять простую поисковую таблицу для

нахождения вероятностей. Если они неизвестны, то оптимальный Л-алгоритм Уиттера

может быть легко модифицирован для управления деревом накапливаемых частот,

причем длина пути обхода дерева, имеющая место во время сжатия или развертывания

не будет превышать значение 2( H (S) + C(S) ). Аналогично можно использовать

алгоритм расширяющегося префикса, дающего ограничение O(H (S)) для длины пути,

но при большем постоянном множителе. Ранее пpиведенные опытные результаты

показывают, что эти постоянные множители более чем компенсируются простотой

алгоритма расширяющегося префикса.

В соответствии с этим алгоритмом операции расширения не нужно затрагивать

информации внутренних узлов дерева. Когда расширение выполняется как часть

операции update, каждая операция полувpащения должна предохранять инвариацию

регулирования весов узлов дерева. На рисунке 8 дерево полувpащается вокруг А,

имея результатом то, что вес Х сокращается весом А и наращивается весом С. В то

же время, поскольку это есть часть повторного пути от А к корню, вес А

увеличивается. Итоговый код будет:

procedure update( c: codetype );

var

c, d: upindex { пара полувpащаемых узлов };

a, b: downindex { наследники полувpащемых узлов };

begin

a := c + succmax;

repeat { вверх по дереву, чередуя и наращивая }

c := up[a];

if c # root then begin { оставшаяся пара }

d := up[c];

{ обмен между наследниками пары }

b := left[d];

if c = b then begin b := right[d];

right[d] := a;

end else left[d] := a;

if a = left[c] then left[c] := b

else right[c] := b;

up[a] := d;

up[b] := c;

freq[c] := ( freq[c] - freq[a] ) + freq[b];

freq[a] := freq[a] + 1;

a := d;

end else begin { помещение непарного ( нечетного ) узла в конец пути }

freq[a] := freq[a] + 1;

a := up[a];

end;

until a = root;

freq[root] := freq[root] + 1;

end { update };

Программа игнорирует проблему переполнения счетчиков частот. Арифметическое

сжатие данных постоянно производит вычисление по формуле a \* b / c, и предел

точности результата вычисления определяется размером памяти, выделяемой

промежуточным произведениям и делимым, а не самим целочисленным перемен ным.

Многие 32-битные машины накладывают 32-битовое ограничение на произведения и

делимые, и т.о. на самом деле устанавливают 16-битовый предел на представление

целых чисел a, b и c в вышеуказанном выражении. Когда это ограничение передается

коду самой программе архиватора, то чистый результат имеет ограничение в 16383

для максимального значения, возвращаемого функцией maxrange или значения

freq[root]. Поэтому, если сжатый файл имеет длину более 16383 байтов, необходимо

периодически пересчитывать все частоты в СД, чтобы втиснуть их в этот интервал.

Простой путь для этого - разделить значения всех частот на маленькую константу,

например 2, и округлением вверх предохранить частоты от обнуления.

Значения листьев в дереве накапливаемых частот легко могут быть пересчитаны

делением на 2, но значения внутренних узлов пересчитать на так легко изза

трудности распространения округляемых результатов вверх по дереву. Простейший

способ перестройки дерева показан в следующей процедуре:

procedure rescale;

var

u: upindex;

d: downindex;

begin

for d := succmax to twicemax do

freq[d] := ( freq[d] + 1 ) div 2;

for u := maxchar downto 1 do begin

left[u] := 2 \* u;

right[u] := ( 2 \* u ) + 1;

freq[u] := freq[left[u]] + freq[right[u]];

up[left[u]] := u;

up[right[u]] := u;

end;

end { rescale };

Характеристика арифметических кодов.

Hа основе алгоpитма Виттена, Нейла и Клири вышепредставленные процедуры были

обьединены в среде языка Паскаль. Как и ожидалось, значительной разницы между

сжатыми текстами, полученными в результате работ первоначального и

модифицированного алгоритмов арифметического сжатия не оказалось. Обычно эти

тексты имеют одинаковую длину.

Рисунок 9 показывает скорость двух этих алгоритмов как функцию от H . Время

представлено в милисекундах на байт исходного текста, а энтропия - в битах на

байт источника. Файлы с 2 битами/байт и 8 битами/байт созданы искусственно, а

остальные представляют собой:

цифровой графический файл, использующий 16 оттенков серого цвета ( 3.49

бит/байт );

текстовой файл ( 4.91 бит/байт исходного текста );

M68000 объектный файл ( 6.02 бит/байт ).

Время измерялось на рабочей станции HP9836 в среде HP-UX.

Как показано на рисунке 9, применение расширения к дереву накапливаемых частот

улучшает алгоритм move-to-front, используемый Виттеном, Нейлом и Клири [12],

только когда сжимаемые данные имеют энтропию больше, чем 6.5 бит/байт. Ниже

этого значения метод move-to-front всегда работает немного лучше расширения.

Т.о. расширение или другие подходы к балансированию дерева накапливаемых частот

вероятно не оправдываются пpи сжатии данных, использующих 256-буквенный алфавит.

Однако, опыты показывают, что для большего алфавита pасширение может быть лучшим

подходом.

Заключение.

Представленный здесь алгоритм расширяемого префикса является вероятно самым

простым и быстрым адаптивным алгоритмом сжатия, основанном на использовании кода

префикса. Его характерные черты - очень небольшое количество требуемой ОП и

локально адаптивное поведение. Когда доступны большие объемы памяти,

использование этого алгоритма вместе с моделью Маркова часто позволяет сжать

данные лучше, чем это делают конкурирующие алгоритмы на этом же объеме памяти.

Преимущества алгоритма расширяющегося префикса нагляднее всего видны при сжатии

графических данных. Локально адаптированный характер алгоритма позволяет сжимать

изображение к меньшему количеству бит, чем самоэнтропия, измеренная у статичного

источника. В итоге, простая модель Маркова, применяемая в алгоритме

расширяющегося префикса, часто позволяет осуществить лучшее сжатие, чем широко

используемый алгоритм Зива-Лемпела на сопоставимом объеме памяти.

Алгоритмы арифметического сжатия данных могут выполняться за время O(H) при

использовании дерева накапливаемых частот, балансируемого эвристическим

расширением для требуемой алгоритмом статистической модели. Это создает новое

ограничение, поэтому простой эвристический метод помещения в начало ( move

-to-front ) является более эффективным для маленьких типовых алфавитов.

И алгоритм расширяющегося префикса, и использование расширения для управления

деревом накапливаемых частот служат полезными иллюстрациями применения

расширения для управления лексикогpафически неупорядоченными деревьями. Идея

поворота, предваряющего расширение дерева, может найти применение и в

нелексикографических деревьях, равно как и понятие полуобоpота для балансировки

таких деревьев. Например, их можно применять для слияния, пpи использовании

двоичного дерева с 2-я путями слияния для построения n-путевого слияния.

Интересно отметить, что по сравнению с другими адаптивными схемами сжатия,

потеря здесь 1 бита из потока сжатых данных является катастрофой! Поэтому

pешение проблемы восстановления этой потери представляет несомненный интерес,

что кроме того предполагает возможность использования таких схем сжатия в

криптографии. Хорошо известно, что сжатие сообщения перед его шифровкой

увеличивает трудность взламывания кода просто потому, что поиск кода основан на

избыточности информации в зашифрованном тексте, а сжатие сокращает это

излишество. Новая возможность, представленная в описанных здесь алгоритмах

сжатия, состоит в использовании начального состояния дерева префикса кодов или

начального состояния дерева накапливаемых частот в качестве ключа для прямого

шифрования в процессе сжатия. Алгоритм арифметического сжатия может кроме того

усложнить работу взломщика кодов тем, что границы букв не обязательно находятся

также и между битами.

Ключевое пространство для такого алгоритма шифрования огромно. Для n букв

алфавита существует n! перестановок на листьях каждого из C деревьев, содержащих

n - 1 внутренних узлов, где C = ( 2i )! / i! ( i+1 )! есть i-ое число Каталана.

Это произведение упрощается к ( 2( n-1 ) )! / ( n-1 )!. Для n = 257 ( 256 букв с

символом end-of-file конца файла ) это будет 512!/256! или что-то меньшее 2 .

Компактное целое представление ключа из этого пространства будет занимать 675

байт, поэтому несомненно такие большие ключи могут поставить в тупик. На

практике одно из решение будет заключаться в начале работы с уже

сбалансированным деревом, как и в рассмотренном здесь алгоритмах сжатия, а затем

расширении этого дерева вокруг каждого символа из ключевой строки,

предоставленной пользователем. Вpяд ли они будет вводить ключи длиной 675 байт,

хотя, чтобы позволить расширению установить дерево во все возможные состояния,

нужны ключи еще длиннее чем этот, но даже короткий ключ может позволить

осуществить шифрование на приемлемом уровне.