Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и среды

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«Сжатие данных без потерь»

Студент Ермолович Дмитрий Сергеевич
Гр. 053504
Ассистент кафедры информатики
Руководитель Давыдчик А.В.

Минск 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сжатие данных - это процесс, при котором исходные данные сжимаются для уменьшения их объема, тем самым упрощая их хранение и передачу. Существует два основных типа сжатия данных: сжатие с потерями и сжатие без потерь. В данном контексте рассмотрим сжатие данных без потерь. В моей курсовой работе я буду рассматривать сжатие данных без потерь.

Сейчас нам доступны носители информации большого объема, и высокоскоростные каналы передачи данных. Однако, одновременно с этим растут и объемы передаваемой информации. Если несколько лет назад можно было смотреть 700-мегабайтные фильмы, умещающиеся на одну флешку, то сегодня фильмы в HD-качестве могут занимать десятки гигабайт.

Конечно, пользы от сжатия всего, и вся не так много. Но все же существуют ситуации, в которых сжатие крайне полезно, если не необходимо.

1 Пересылка документов по электронной почте (особенно больших объемов документов с использованием мобильных устройств)

2 При публикации документов на сайтах, потребность в экономии трафика

3 Экономия дискового пространства в тех случаях, когда замена или добавление средств хранения затруднительно.

Цель моей работы исследовать разные алгоритмы сжатия данных без потерь, сравнить их, выделить их плюсы и минусы.

1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Виды сжатия данных

Все методы сжатия можно разделить на две большие группы: сжатие с потерями и сжатие без потерь. Сжатие без потерь применяется в тех случаях, когда информацию нужно восстановить с точностью до бита. Такой подход является единственно возможным при сжатии, например, текстовых данных.

В некоторых случаях, однако, не требуется точного восстановления информации и допускается использовать алгоритмы, реализующие сжатие с потерями, которое, в отличие от сжатия без потерь, обычно проще реализуется и обеспечивает более высокую степень архивации.

Виды сжатий:

1 Сжатие с потерями:

При таком сжатии мы теряем часть информации. Но смысл алгоритмов сжатия в том, чтобы мы этого не замечали: сжатие должно происходить так, чтобы всё важное передалось, а неважное — нет. Лучшие степени сжатия, при сохранении «достаточно хорошего» качества данных. Применяются в основном для сжатия аналоговых данных — звука, изображений. В таких случаях распакованный файл может очень сильно отличаться от оригинала на уровне сравнения «бит в бит», но практически неотличим для человеческого уха или глаза в большинстве практических применений.

2 Сжатие без потерь:

Данные восстанавливаются с точностью до бита, что не приводит к каким-либо потерям информации. Однако, сжатие без потерь показывает обычно худшие степени сжатия.

1.2 Универсальные методы сжатия без потерь

В общем случае можно выделить три базовых варианта, на которых строятся алгоритмы сжатия.

Первая группа методов – преобразование потока. Это предполагает описание новых поступающих несжатых данных через уже обработанные. При этом не вычисляется никаких вероятностей, кодирование символов осуществляется только на основе тех данных, которые уже были обработаны, как например в LZ – методах (названных по имени Абрахама Лемпеля и Якоба Зива). В этом случае, второе и дальнейшие вхождения некой подстроки, уже известной кодировщику, заменяются ссылками на ее первое вхождение.

Вторая группа методов – это статистические методы сжатия. В свою очередь, эти методы делятся на адаптивные (или поточные), и блочные.

В первом (адаптивном) варианте, вычисление вероятностей для новых данных происходит по данным, уже обработанным при кодировании. К этим методам относятся адаптивные варианты алгоритмов Хаффмана и Шеннона-Фано.

Во втором (блочном) случае, статистика каждого блока данных высчитывается отдельно, и добавляется к самому сжатому блоку. Сюда можно отнести статические варианты методов Хаффмана, Шеннона-Фано, и арифметического кодирования.

Третья группа методов – это так называемые методы преобразования блока. Входящие данные разбиваются на блоки, которые затем трансформируются целиком. При этом некоторые методы, особенно основанные на перестановке блоков, могут не приводить к существенному (или вообще какому-либо) уменьшению объема данных. Однако после подобной обработки, структура данных значительно улучшается, и последующее сжатие другими алгоритмами проходит более успешно и быстро.

1.3 Общие принципы, на которых основано сжатие данных

Все методы сжатия данных основаны на простом логическом принципе. Если представить, что наиболее часто встречающиеся элементы закодированы более короткими кодами, а реже встречающиеся – более длинными, то для хранения всех данных потребуется меньше места, чем если бы все элементы представлялись кодами одинаковой длины.

Точная взаимосвязь между частотами появления элементов, и оптимальными длинами кодов описана в так называемой теореме Шеннона о источнике шифрования (Shannon's source coding theorem), которая определяет предел максимального сжатия без потерь и энтропию Шеннона.

1.4 Энтропия

Если вероятность появления элемента s_i равна $p(s_i)$, то наиболее выгодно будет представить этот элемент — $log_2 p(s_i)$ битами. Если при кодировании удается добиться того, что длина всех элементов будет приведена к $log_2 p(s_i)$ битам, то и длина всей кодируемой последовательности будет минимальной для всех возможных методов кодирования. При этом, если распределение вероятностей всех элементов $F = \{ p(s_i)\}$ неизменно, и вероятности элементов взаимно независимы, то средняя длина кодов может быть рассчитана как показано на формуле 1:

$$H = -\sum_i p(s_i) * log_2 p(s_i), \quad (1)$$

Это значение называют энтропией распределения вероятностей F, или энтропией источника в заданный момент времени.

Однако обычно вероятность появления элемента не может быть независимой, напротив, она находится в зависимости от каких-то факторов. В этом случае, для каждого нового кодируемого элемента s_i распределение вероятностей F примет некоторое значение F_k, то есть для каждого элемента $F= F_k$ и $H= H_k$.

Иными словами, можно сказать, что источник находится в состоянии k, которому соответствует некий набор вероятностей $p_k(s_i)$ для всех элементов s_i

Поэтому, учитывая эту поправку, можно выразить среднюю длину кодов как показано на формуле 2:

$$H = -\sum_k P_k * H_k = -\sum_{k,i} P_k * p_k(s_i) log_2 p_k(s_i), \quad (2)$$

где P_k — вероятность нахождения источника в состоянии k.

2 КОДИРОВАНИЕ БЕЗ ПАМЯТИ

2.1 Основные понятия

Коды без памяти являются простейшими кодами, на основе которых может быть осуществлено сжатие данных. В коде без памяти каждый символ в кодируемом векторе данных заменяется кодовым словом из префиксного множества двоичных последовательностей или слов.

Пусть задан некоторый алфавит $\Psi = \{a_1, a_2, \dots, a_r\}$ состоящий из некоторого (конечного) числа букв. Назовем каждую конечную последовательность символов из этого алфавита $A = a_{1,}a_{1,} \dots, a_{n,}$ словом, а число n — длиной этого слова.

Пусть задан также другой алфавит $\Omega = \{b_1, b_2, \dots, b_q\}$ Аналогично, обозначим слово в этом алфавите как B.

Введем еще два обозначения для множества всех непустых слов в алфавите. Пусть $S(\Psi)$ — количество непустых слов в первом алфавите, а $S(\Omega)$ — во втором.

Пусть также задано отображение F, которое ставит в соответствие каждому слову A из первого алфавита некоторое слово B=F(A) из второго. Тогда слово B будет называться кодом слова A, а переход от исходного слова к его коду будет называться кодированием.

Поскольку слово может состоять и из одной буквы, то мы можем выявить соответствие букв первого алфавита и соответствующих им слов из второго:

a_1 <->B_1

a_2 <->B_2

…

a_n <-> B_n

Это соответствие называют схемой, и обозначают $\sum$.

В этом случае слова B_1, B_2,…, B_n называют элементарными кодами, а вид кодирования с их помощью — алфавитным кодированием. Конечно, большинство из нас сталкивались с таким видом кодирования, пусть даже и не зная всего того, что я описал выше.

Итак, мы определились с понятиями алфавит, слово, код, и кодирование. Теперь введем понятие префикс.

Пусть слово B имеет вид B=B'B''. Тогда B' называют началом, или префиксом слова B, а B'' — его концом. Это довольно простое определение, но нужно отметить, что для любого слова B, и некое пустое слово ʌ («пробел»), и само слово B, могут считаться и началами и концами.

Итак, мы подошли вплотную к пониманию определения кодов без памяти. Последнее определение, которое нам осталось понять — это префиксное множество. Схема $\sum$ обладает свойством префикса, если для любых $1 \le i, j \le r, i \ne j$, слово B_i не является префиксом слова B_j. Проще говоря, префиксное множество – это такое конечное множество, в котором ни один элемент не является префиксом (или началом) любого другого элемента. Простым примером такого множества является, например, обычный алфавит.

Итак, мы разобрались с основными определениями. Так как же происходит само кодирование без памяти?

Оно происходит в три этапа:

1 Составляется алфавит Ψ символов исходного сообщения, причем символы алфавита сортируются по убыванию их вероятности появления в сообщении.

2 Каждому символу a_i из алфавита Ψ ставится в соответствие некое слово B_i из префиксного множества Ω.

3 Осуществляется кодирование каждого символа, с последующим объединением кодов в один поток данных, который будет являться результатам сжатия.

Одним из канонических алгоритмов, которые иллюстрируют данный метод, является алгоритм Хаффмана.

2.2 Алгоритм Хаффмана

Каждый символ представляет собой последовательность 0's а также 1's и хранится с использованием 8-бит. Это известно, как “кодирование с фиксированной длиной”, так как каждый символ использует одинаковое количество фиксированных битов памяти.

Как уменьшить количество места, необходимое для хранения символа?

Идея состоит в том, чтобы использовать “кодирование переменной длины”. Мы можем использовать тот факт, что одни символы встречаются в тексте чаще, чем другие для разработки алгоритма, который может представлять тот же фрагмент текста, используя меньшее количество битов. При кодировании с переменной длиной мы присваиваем символам переменное количество битов в зависимости от их частоты в данном тексте. Таким образом, некоторые символы могут в конечном итоге занимать один бит, а некоторые — два бита, некоторые могут быть закодированы с использованием трех битов и так далее. Проблема с кодированием переменной длины заключается в его декодировании.

Рассмотрим строку `aabacdab`. Оно имеет `8` символов в нем и использует 64-битное хранилище (с использованием кодирования фиксированной длины).

Если принять во внимание, что частота символов a, b, c, а также d находятся 4, 2, 1, 1, соответственно. Попробуем представить aabacdab используя меньшее количество битов, используя тот факт, что a встречается чаще, чем b, а также b встречается чаще, чем c а также d. Начнем со случайного присвоения однобитового кода 0 к a, 2-битный код 11 к b, и 3-битный код 100 а также 011 к персонажам c а также d, соответственно.

На рисунке 1 приведен пример кодирования символов.

```
a 0
b 11
c 100
d 011
```

Рисунок 1 – Кодирования символов

Итак, строка aabacdab будет закодирован в 00110100011011 (0|0|11|0|100|011|0|11) используя приведенные выше коды. Но настоящая проблема заключается в расшифровке. Если мы попытаемся декодировать строку 00110100011011, это приведет к неоднозначности, так как его можно декодировать.

На рисунке 2 приведен пример декодирования строк.

```
0|011|0|100|011|0|11    adacdab
0|0|11|0|100|0|11|011    aabacabd
0|011|0|100|0|11|0|11    adacabab
…
and so on
```

Рисунок 2 – Декодирования строк

Чтобы предотвратить двусмысленность при декодировании, мы обеспечим соответствие нашего кодирования “правилу префикса”, что приведет к “уникально декодируемым кодам”. Правило префикса гласит, что ни один код не является префиксом другого кода. Под кодом мы подразумеваем биты, используемые для определенного символа. В приведенном выше примере 0 является префиксом 011, что нарушает правило префикса. Если

наши коды удовлетворяют префиксному правилу, декодирование будет однозначным (и наоборот).

Давайте снова рассмотрим приведенный выше пример. На этот раз мы присваиваем символам коды, удовлетворяющие правилу префикса. 'a', 'b', 'c', а также 'd'.

На рисунке 3 приведен пример кодирования символов в соответствии с префиксами.

```
a   0
b   10
c   110
d   111
```

Рисунок 3 – Кодирования символов в соответствии с префиксами

Используя приведенные выше коды, строка aabacdab будет закодирован в 00100110111010 (0|0|10|0|110|111|0|10). Теперь мы можем однозначно декодировать 00100110111010 вернуться к нашей исходной строке aabacdab.

Теперь, когда мы разобрались с кодированием переменной длины и правилом префиксов, давайте поговорим о кодировании Хаффмана.

Кодирование Хаффмана

Техника работает, создавая бинарное дерево узлов. Узел может быть листовым узлом или внутренним узлом. Изначально все узлы являются листовыми узлами, которые содержат сам персонаж, вес (частоту появления) персонажа. Внутренние узлы содержат вес символов и ссылки на два дочерних узла. По общему соглашению, бит 0 представляет следующий левый дочерний элемент, и немного 1 представляет следующий правильный ребенок. Готовое дерево имеет n листовые узлы и n-1 внутренние узлы. Рекомендуется, чтобы дерево Хаффмана отбрасывало неиспользуемые символы в тексте, чтобы получить наиболее оптимальную длину кода.

Мы будем использовать приоритетная очередь для построения дерева Хаффмана, где узел с наименьшей частотой имеет наивысший приоритет. Ниже приведены полные шаги:

1. Создайте конечный узел для каждого символа и добавьте их в очередь приоритетов.

2. Пока в queue больше одного узла:

2.1 Удалите из queue два узла с наивысшим приоритетом (самой низкой частотой).

2.2 Создайте новый внутренний узел с этими двумя узлами в качестве дочерних элементов и частотой, равной сумме частот обоих узлов.

2.3 Добавьте новый узел в очередь приоритетов.

3. Оставшийся узел является корневым узлом, и дерево завершено.

Рассмотрим некоторый текст, состоящий только из 'A', 'B', 'C', 'D', а также 'E' символов, а их частота 15, 7, 6, 6, 5, соответственно. Рисунке 4,5 иллюстрируют шаги, за которыми следует алгоритм.

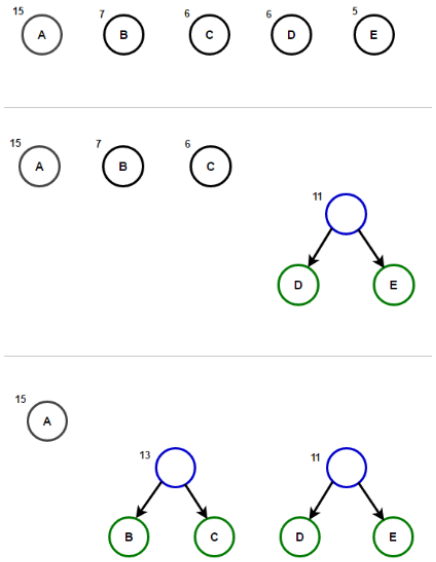

Рисунок 4 – Алгоритм Хаффмана

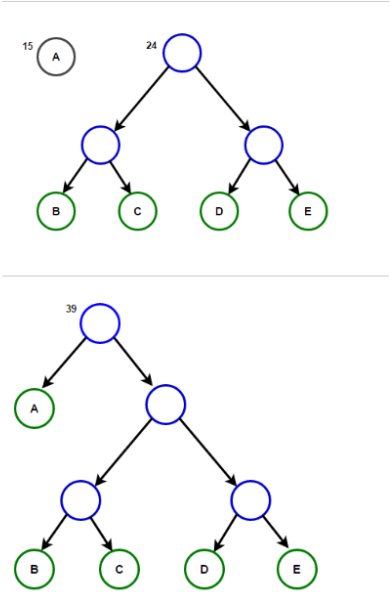

Рисунок 5 – Алгоритм Хаффмана

Путь от корня к любому конечному узлу хранит оптимальный код префикса (также называемый кодом Хаффмана), соответствующий символу, связанному с этим конечным узлом.

На рисунке 6 изображено дерева с помощью которого можно закодировать символ в соответствии с его частотой.

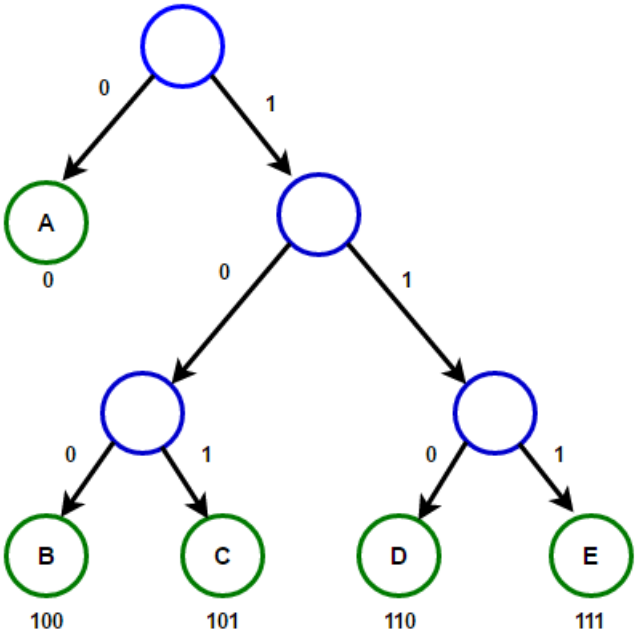

Рисунок 6 – Конечное дерево

Рассмотрим еще один пример пусть на вход у нас была строка из 1000 символов, в которой символ a_1 встречался 500 раз, a_2 — 240, a_3 — 150, и a_4 — 110 раз. После кодирования мы получим $a_1 = 0$, $a_2 = 11$, $a_3 = 100$, $a_4 = 101$.

Если предположить, что изначально для хранения каждого символа использовался один байт, то можно посчитать, насколько нам удалось уменьшить данные.

Изначально данная строка занимала 8000 бит. После кодирования мы получим строку длинной в $\sum p_i l_i = 500 * 1 + 240 * 2 + 150 * 3 + 110 * 3 = 1760$ бит. Итак, нам удалось сжать данные в 4,54 раза, потратив в среднем 1,76 бита на кодирование каждого символа потока.

Согласно Шеннону, средняя длина кодов составляет $H = -\sum_i p(s_i) * log_2 p(s_i)$.

Подставив в это уравнение наши значения вероятностей, мы получим среднюю длину кодов равную 1.75496602732291, что весьма и весьма близко к полученному нами результату.

3 КОДИРОВАНИЕ ШЕННОНА-ФАНО

Кодирование Шеннона-Фано является одним из самых первых алгоритмов сжатия, который впервые сформулировали американские учёные Шеннон (Shannon) и Фано (Fano). Данный метод сжатия имеет большое сходство с кодированием Хаффмана, которое появилось на несколько лет позже. Главная идея этого метода - заменить часто встречающиеся символы более короткими кодами, а редко встречающиеся последовательности более длинными кодами. Таким образом, алгоритм основывается на кодах переменной длины. Для того, чобы декомпрессор впоследствии смог раскодировать сжатую последовательность, коды Шеннона-Фано должны обладать уникальностью, то есть, не смотря на их переменную длину, каждый код уникально определяет один закодированый символ и не является префиксом любого другого кода.

Рассмотрим алгоритм вычисления кодов Шеннона-Фано (для наглядности возьмём в качестве примера последовательность 'aa bbb cccc ddddd'). Для вычисления кодов, необходимо создать таблицу уникальных символов сообщения c(i) и их вероятностей p(c(i)), и отсортировать её в порядке невозрастания вероятности символов.

На рисунке 7 приведена таблица вероятностей символов.

c(i)	p(c(i))
d	5 / 17
c	4 / 17
space	3 / 17
b	3 / 17
a	2 / 17

Рисунок 7 – Таблица вероятностей символов

Далее, таблица символов делится на две группы таким образом, чтобы каждая из групп имела приблизительно одинаковую частоту по сумме символов. Первой группе устанавливается начало кода в '0', второй в '1'. Для вычисления следующих бит кодов символов, данная процедура повторяется рекурсивно для каждой группы, в которой больше одного символа. Таким образом для нашего случая получаем следующие коды символов.

На рисунке 8 приведена таблица кодирования символов.

символ	код
d	00
c	01
space	10
b	110
a	111

Рисунок 8 – Таблица кодирования символов

Длина кода s(i) в полученной таблице равна int(-lg p(c(i))), если символы удалость разделить на группы с одинаковой частотой, в противном случае, длина кода равна int(-lg p(c(i))) + 1. Это можно записать в виде отношения: int(-lg p(c(i))) <= s(i) <= int(-lg p(c(i))) + 1.

Используя полученную таблицу кодов, кодируем входной поток - заменяем каждый символ соответствующим кодом. Естественно для расжатия полученной последовательности, данную таблицу необходимо сохранять вместе со сжатым потоком, что является одним из недостатков данного метода. В сжатом виде, наша последовательность принимает вид: 11111110110110110100101010110000000000. Ее длина в 39 бит. Учитывая, что оригинал имел длину равную 136 бит, получаем коэффициент сжатия ~28% - не так уж и плохо.

Мы не можем, как в случае кодирования, заменять каждые 8 бит входного потока, кодом переменной длины. При расжатии нам необходимо всё сделать наоборот - заменить код переменной длины символом длиной 8 бит. В данном случае, лучше всего будет использовать бинарное дерево, листьями которого будут является символы (аналог дерева Хаффмана).

Кодирование Шеннона-Фано является достаточно старым методом сжатия, и на сегодняшний день оно не представляет особого практического интереса (разве что как упражнение по курсу структур данных). В большинстве случаев, длина сжатой последовательности, по данному методу, равна длине сжатой последовательности с использованием кодирования Хаффмана. Но на некоторых последовательностях всё же формируются не оптимальные коды Шеннона-Фано, поэтому сжатие методом Хаффмана принято считать более эффективным. Для примера, рассмотрим последовательность с таким содержанием символов: 'a' - 14, 'b' - 7, 'c' - 5, 'd' - 5, 'e' - 4. Метод Хаффмана сжимает её до 77 бит, а вот Шеннона-Фано до 79 бит.

На рисунке 9 приведена таблица кодирования символов методами Хаффмана и Шеннона-Фано.

символ	код Хаффмана	код Шеннона-Фано
a	0	00
b	111	01
c	101	10
d	110	110
e	100	111

Рисунок 9 – Таблица кодирования символов разными способами

4 ПРЕОБРАЗОВАНИЕ БАРРОУЗА-УИЛИРА.