Лекция 13. Сжатие данных

Лекция 13. Сжатие данных....1
1 Понятие информации....1
2 Память компьютера, байт....2
3 Экономное кодирование информации....2
4 Префиксный код....3
5 Алгоритм Шеннона-Фено....5
6 Алгоритм Хаффмена....7
7 Словарные алгоритмы....9
8 Алгоритм Лемпеля-Зива (LZ)....9
9 Как хранить словарь....11
10 Алгоритм LZ77....12
11 Источники....14

1 Понятие информации

В понедельник двое друзей, A и B, договорились, что на следующий день A пришлет B письмо, в котором будет единственное слово – «привет». Во вторник B получил письмо от A, и в нем было единственное слово – «привет». Сколько информации содержалось в этом письме?

Правильный ответ – нисколько. Дело в том, что B ничего не узнал из этого письма: его содержание было известно заранее.

Итак, количество информации – это не длина письма в буквах или других единицах: можно послать очень длинное письмо, весь текст которого заранее известен – и не сообщить таким образом ничего. Количество информации в таком письме равно нулю.

Огорченные таким исходом дела, A и B снова договорились, что A пошлет B письмо, но в нем будет либо слово «привет», либо слово «здравствуй». И это письмо B получил, в письме было слово «привет». Сколько информации получил B?

Здесь нужно договориться о том, в каких единицах измерять информацию. Принято измерять информацию в *битах*. Один бит, как вы знаете, обозначает один двоичный разряд (0 или 1) – или одно значение «истина» или «ложь».

Итак, сколько же информации получил B? Он мог получить одно из двух слов. То есть они с A могли договориться, что A пришлет в письме единицу или ноль, а B будет знать, что, например, единица означает «привет», а ноль – «здравствуй». Значит, B получил один бит информации.

Количество информации зависит не от длины письма, а от количества возможных вариантов содержания письма. Если есть всего два варианта – это один бит.

Если вариантов всего четыре – это два бита. Пусть есть четыре возможных слова {A, B, C, D}. Разобьем их на две пары: {A, B} и {C, D}. Первый бит – это номер пары, а второй – номер слова в паре.

А если бы возможных слов было три? А если пять? А если пятьдесят?

Сколько информации получил B, если A прислал ему письмо с одним из N возможных слов?

Можно, по аналогии со случаем двух слов, дать каждому слову номер, начиная с нуля. Тогда можно считать, что в письме пришло число из диапазона от 0 до N-1. Количество информации соответствует количеству двоичных разрядов в этом числе (включая ведущие нули: даже если не писать их в письме, они все равно имеются в виду и несут информацию).

Количество двоичных разрядов в числе из диапазона [0; N-1] равно двоичному логарифму N, округленному вверх.

$$I(M) = \log_2(N)$$ ***(формула Шеннона[1])***

I(M) – количество информации в сообщении M, N – количество различных сообщений, которые вообще возможны.

2 Память компьютера, байт

Однако, если записать в файл слово «привет» и сохранить его на диске, такой файл займет шесть байт, то есть 48 бит информации. Как же так: одно слово, а столько занимает?!

С помощью 48 бит можно закодировать одно из 2^{48} слов. 2^{48} = 281’474’976’710’656 ≈ $2{,}8*10^{14}$. Никакого обмана нет. В файле из шести байт может быть любое слово из шести букв (а кроме букв могут быть цифры, знаки препинания и непечатаемые символы). Если вам показать файл (не открывать его), и сообщить его размер – 6 байт, вы сможете сказать, что в нем записано? Нет, и даже чтобы угадать, Вам понадобится очень много времени!

Итак, дело не в том, сколько слов записано в сообщении, а в том, сколько всего возможно вариантов различных сообщений.

3 Экономное кодирование информации

Все вы слышали о существовании архиваторов – программ, которые преобразуют файлы так, что те занимают меньше места на диске. Принципы работы архиваторов тесно связаны с идеями, изложенными в предыдущих разделах.

Общая идея алгоритма архивации такова: придумать некоторый способ кодирования информации, который поможет сэкономить память.

Приведем для примера простейший способ архивации данных: если в некоторой последовательности символов несколько одинаковых символов идут подряд, такая цепочка заменяется парой {количество повторений, повторяющийся символ}. Этот алгоритм сжатия называется RLE, Run-Length Encoding (групповое кодирование).

Например, строка

aaaaabbccccaaad

Кодируется как

5a2b4c3a1d

В исходной строке каждый символ занимал 1 байт: всего было 15 символов, то есть 120 бит. Сколько бит в «сжатой» строке? На поверхности лежит такой ответ: 10 символов, значит 10 байт, то есть 80 бит: мы сэкономили 40 бит.

А если бы в исходной строке было 15 букв «а»? Тогда алгоритм RLE дал бы такой код:

15a

И сколько теперь бит? 24 (три байта) или 16 (два)? Можно, конечно, три (можно и пять ☺), но можно и два, поскольку число 15 поместится в 1 байт. Вообще-то число 15 и в полбайта поместится… Но в этом случае, если буква встретится 16 раз, придется уже строить такой код:

15a1a

Дело в том, что каждая пара должна занимать одинаковое количество бит, иначе мы не сможем понять, где кончается одна пара и начинается другая.

Теперь обратите внимание: отводя на количество повторений 4 бита, мы получили для строки из 16 байт код из трех байт: по четыре на число и по восемь на символ. Если бы мы отводили 8 (или хотя бы 5) бит на число повторений, код получился бы короче. Значит, лучше 8 бит, чем четыре.

С другой стороны, мы ведь рассматривали строку из всех одинаковых символов. Если придется архивировать другую строку, дело обернется иначе. Посмотрите на первый пример: при восьми

[1] Клод Шеннон (1916-2001), американский математик, создатель теории информации, один из крупнейших ученых в области Computer Science, наряду с Аланом Тьюрингом и Джоном Фон-Нейманом.

битах на количество повторений код имеет длину 80 бит, а при четырех – 60. Значит, четыре лучше, чем восемь… ☺

Придется привлечь здравый смысл: какие строки чаще встречаются: из всех одинаковых символов, или из разных? Видимо, все-таки из разных. Значит, стоит ориентироваться на случай с разными символами.

Сделанный нами на основе весьма поверхностных рассуждений вывод имеет фундаментальное значение. Все известные алгоритмы архивации основаны на знаниях о характере кодируемой информации: мы используем «опыт поколений», чтобы узнать, какие строки чаще встречаются – и придумываем алгоритмы, которые позволяют сжимать такие строки.

Эта же идея обосновывает саму возможность сжатия данных. В предыдущих разделах мы познакомились с оценкой количества информации. Казалось бы, теперь мы можем сказать, сколько на самом деле информации в том или ином слове. Однако некоторые слова (и некоторые буквы) встречаются чаще других, **поэтому несут меньше информации**.

Если в поисковой системе в Интернете (например, в Google или Яndex) ввести в строку поиска слово «университет», поисковая система выдаст миллионы страниц, содержащих это слово – вряд ли вы найдете среди них нужную. Дело в том, что слово «университет» встречается очень часто, то есть вы сообщили поисковой системе слишком мало информации о странице, которую ищете.

Слово «университет» довольно длинное, занимает 11 байт, но это не важно в масштабе Интернета – оно встречается слишком часто. Более короткое слово «кряж» дает гораздо меньший диапазон страниц (Google: «университет» - 41 000 000, «кряж» - 280 000, а еще более короткое слово «жом» - 149 000). Если единственное, что Вы запомнили о лучшем сайте в Интернете – это что на нем было слово «кряж», у Вас гораздо больше шансов его найти, чем в предыдущем случае. Слово «кряж» сообщает о сайте больше информации, чем слово «университет».

Повторения фрагментов текста (любой последовательности символов) уменьшают количество информации в этой последовательности, поэтому возможна архивация данных.

Сказанное касается не только повторений, но и любых других закономерностей, наблюдаемых в сжимаемых данных. Если известно, что каждый следующий символ сообщения на 1 больше предыдущего, достаточно знать первый символ и общее их количество для того, чтобы восстановить все сообщение – такое сообщение содержит мало информации (обратите внимание: разных сообщений, подчиняющихся такому правилу гораздо меньше чем вообще всех сообщений такой длины – поэтому-то и меньше информации).

На самом деле приведенная выше формула для количества информации является упрощением формулы Шеннона для случая, когда все возможные сообщения равновероятны. Если вероятности всех сообщений различны (общий случай), формула выглядит так:

$$I(M) = \log_2\left(1 / p(M)\right)$$

Здесь p(M) – вероятность появления сообщения M. Если все $p(M_i)$ равны (равны 1/N, где N – количество возможных сообщений), получается упрощенная формула, приводившаяся ранее.

Пусть слово «жуть» встречается с вероятностью 1/8, слово «мама» - с вероятностью ½, а слово «привет» - с вероятностью ¼ (еще 1/8 приходится на все остальные слова). Тогда слово «мама» несет меньше всех информации $\log_2(2) = 1$ бит, слово «привет» несет $\log_2(4) = 2$ бита, а слово «жуть» $-\log_2(8) = 3$ бита. Остальные (еще более редкие) слова – несут еще больше информации.

4 Префиксный код

Теперь посмотрим на проблему под другим углом. Преподаватель сказал студентам, что с вероятностью 50% они получат на экзамене «3», с вероятностью 30% - «4», с вероятностью 10% - «2», и с вероятностью 10% - «5». Как хранить список оценок наиболее экономно?

Вот пример списка оценок:

4 3 3 5 3 2 4 3 4 3

Если мы посмотрим на оценку из списка, скорее всего это будет «тройка», значит, на эту оценку приходится минимум информации. I(«3») = $\log_2(1/0.5)$ = 1 бит. На «4» приходится больше информации, а на «2» и «5» - еще больше.

Мы говорили о том, что при кодировании приходится отводить одинаковое количество бит на каждый символ, поскольку это необходимо, чтобы определять границы между кодовыми словами (кодами отдельных символов).

Однако можно придумать способ кодирования, при котором на разные символы будет приходится разное число бит. Для этого нужно разработать правило определения границ кодового слова. Такие коды называются кодами переменной длины.

Наиболее известный код переменной длины – префиксный код.

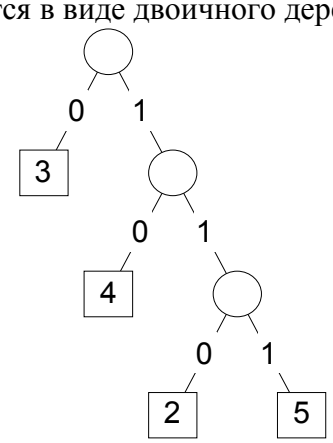
Идея состоит в том, чтобы закодировать каждый символ некоторой последовательностью битов (кодовым словом), причем так, чтобы никакое кодовое слово не являлось началом другого кодового слова.

Вот пример префиксного кода для оценок:

2	110
3	0
4	10
5	111

В этом коде никакое кодовое слово не является началом другого кодового слова: если есть кодовое слово «0», значит не может быть слова «01», а если есть слово «10», не может быть слова «101». Если бы оценка «2» кодировалась как «11», то оценку «5» уже нельзя было бы закодировать (никак), потому что все возможные начала кодов уже были бы заняты.

Префиксный код хорошо изображается в виде двоичного дерева:



Кодируемые символы находятся в листьях, а кодовое слово – это путь в двоичном представлении.

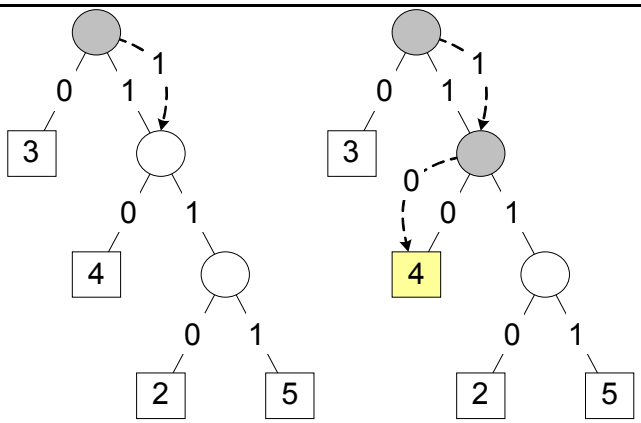
С использованием этого кода приведенный пример записывается так:

Оценка	4	3	3	5	3	2	4	3	4	3
Код	10	0	0	111	0	110	10	0	10	0

В итоге получится такой код (из 17 бит – довольно экономично!): 10001110110100100. Как восстановить исходную последовательность, зная таблицу кодов?

Для этого достаточно перемещаться по дереву кодов, просматривая последовательность слева направо.

Начинаем с корня дерева. Первый бит последовательности – «1», значит от корня мы идем направо. Далее идет «0», значит теперь идем налево. Мы пришли в лист – «4», значит была закодирована «четверка».



После того, как мы пришли в лист, возвращаемся в корень и продолжаем читать последовательность. Далее идет «0» - шаг налево – приходим в лист «3», значит закодирована «тройка», возвращаемся в корень. Снова «0» - снова налево – «тройка». И так далее.

Если построить хороший префиксный код, он поможет сжать информацию – длина кодовой последовательности будет меньше, чем длина исходных данных. Однако и здесь возникают сложности: для раскодирования требуется хранить информацию о коде вместе с заархивированными данными. Сделать это несложно, но нужно экономно расходовать память на таблицу кодов.

5 Алгоритм Шеннона-Фено

Используя префиксные коды, можно кодировать символы, встречающиеся часто (несущие мало информации) более короткими последовательностями, а более редкие символы – более длинными. Код, приведенный выше, построен именно так, поэтому он позволяет довольно хорошо сжимать данные.

Для построения такого кода существует несколько алгоритмов. Один из них предложен независимо К. Шенноном и Р. Фено.

Запишем таблицу встречаемости символов в тексте, для каждого символа записано количество его вхождений в текст, таблица сортируется по возрастанию встречаемости символов:

3	5
4	3
5	1
2	1

Разделим таблицу на две части так, чтобы суммарное количество вхождений символов из разных частей таблицы было равно (или как можно более близко):

3	5
4	3
5	1
2	1

В первую часть войдет символ «3» (суммарная встречаемость – 5), а во вторую – «4», «2» и «5» (суммарная встречаемость – 5). Первый символ кода для первой группы – «0», для второй – «1».

сим-вол	к-во	код
3	5	0
4	3	1
5	1	1
2	1	1

Далее каждая группа (если в ней больше одного символа) снова делится пополам по тому же принципу: вторая группа разделится на «4» (суммарная встречаемость – 3) и «2» и «5» (суммарная встречаемость – 2). Это наиболее близкие по значению суммарной встречаемости группы.

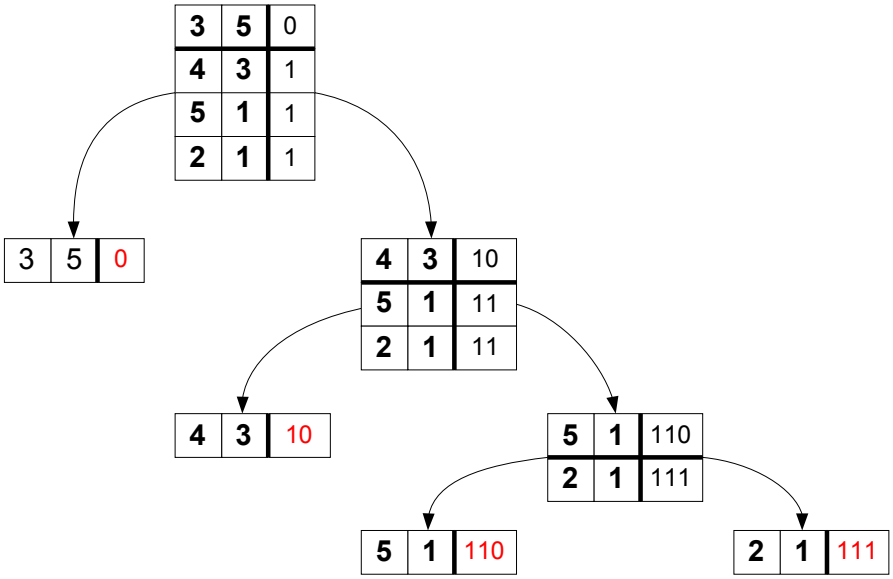
Для первой группы очередной символ кода – ноль, для второй – единица.

И так далее.

символ	к-во	код
3	5	0
4	3	10
5	1	11
2	1	11

символ	к-во	код
3	5	0
4	3	10
5	1	110
2	1	111

Как видно, алгоритм работает рекурсивно. Вот дерево рекурсивных вызовов:



Какова трудоемкость алгоритма Шеннона-Фено?

Подсчет встречаемости символов можно произвести за один проход по тексту. Этот этап мы не будем учитывать при оценке трудоемкости.

Сортировка таблицы по убыванию встречаемости происходит за $O(N*\log_2 N)$, где N – количество различных символов в тексте.

Глубина дерева рекурсии в худшем случае составляет N (если окажется, что каждый раз нам пришлось делить на части размера 1 и N-1). На каждом шаге рекурсии мы дописываем один символ к каждому коду, то есть на каждом уровне дерева обрабатывается вся таблица. Это происходит за O(N). Итого N уровней по N операций – общая трудоемкость $O(N^2)$.

Попробуйте сами построить код для такого примера:

3 3 3 2 5 2 4 4 3 4 5 2

Проверьте себя:

символ	к-во	код
3	4	01
4	3	00
2	3	10
5	2	11

6 Алгоритм Хаффмена

Алгоритм Шеннона-Фено строит префиксный код, согласованный с частотами встречаемости символов в тексте. Однако этот код не всегда оптимален. Иногда можно построить код лучше, чем построенный алгоритмом Шеннона-Фено (дающий более короткий код для всего текста).

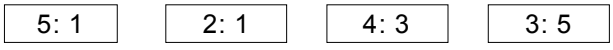
Для построения *оптимального* префиксного кода используется алгоритм Хаффмена.

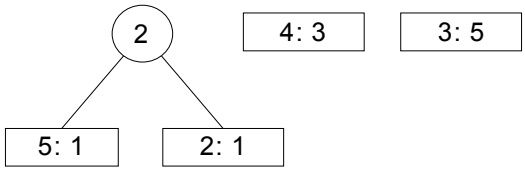
В отличие от алгоритма Шеннона-Фено, алгоритм Хаффмена строит дерево префиксного кода непосредственно. Результатом работы алгоритма является именно дерево, а не таблица кодов.

Идея алгоритма Хаффмена состоит в следующем.

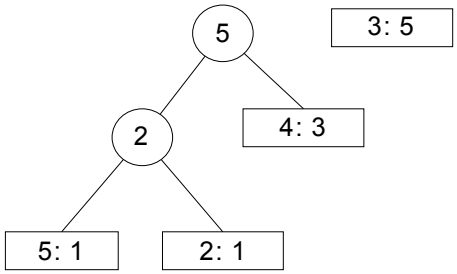
Из всех символов выбираются два с наименьшими значениями встречаемости. Создается вершина дерева и эти два символа становятся ее детьми. В вершину записывается суммарная встречаемость ее детей.

На рисунке в прямоугольных вершинах записан символ, а затем его встречаемость через двоеточие.

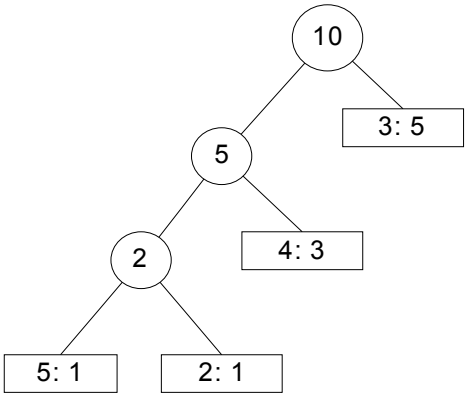




Символы «5» и «2» стали детьми новой вершины со значением 2. Значение в этой вершине – это встречаемость нового «псевдо-символа» «5 или 2». Этот символ добавляется в набор вместо своих детей. На следующем шаге будет выбираться минимум из символов «3» (встречаемость 5), «4» (встречаемость 3) и «5 или 2» (встречаемость 2). Будут выбраны два минимальных: «*5 или 2*» и «4».

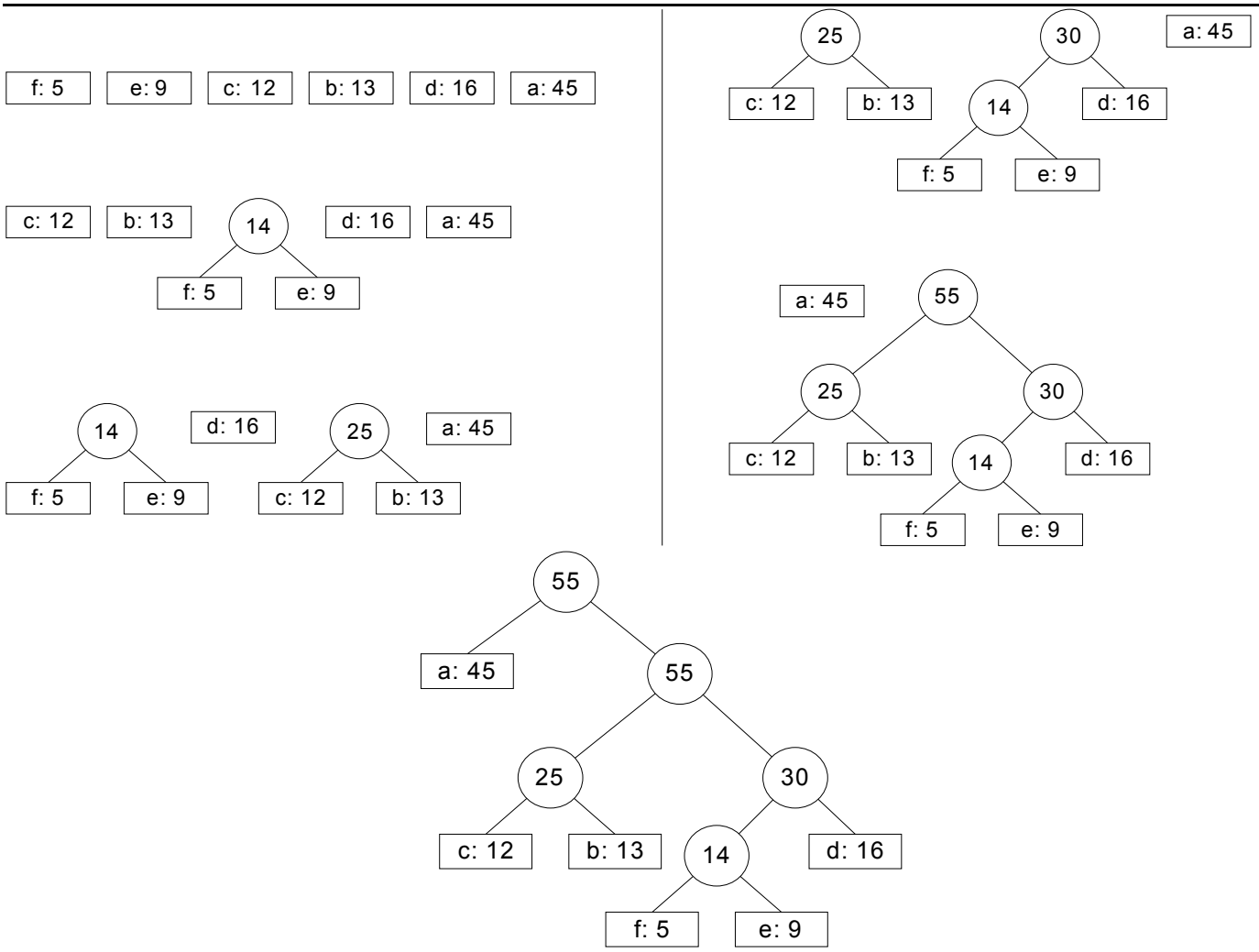


Снова создается вершина, выбранные символы становятся ее детьми. Теперь осталось два символа: «3» и ««*5 или 2*» *или 4*» (встречаемость 5). Это и есть два символа с минимальными значениями встречаемости.



Последний шаг заканчивает построение дерева, создавая корневую вершину.

Вот пример работы алгоритма Хаффмена на более содержательном наборе данных.



Кодовые слова восстанавливаются по дереву как двоичные записи путей до листьев: влево – ноль, вправо – единица.

В процессе работы алгоритма требуется многократно искать минимум. Для этих целей имеет смысл организовать из вершин дерева очередь с приоритетами (ее по-прежнему можно реализовать с помощью двоичной кучи).

Перед началом алгоритма для каждого символа создается вершина дерева и все они помещаются в очередь с приоритетами (невозрастающую кучу, где ключом является встречаемость, записанная в вершине).

Далее алгоритм выглядит так:

- Пока в очереди больше одной вершины:
  1. Извлечь из очереди два минимальных элемента: A и B
  2. Создать новую вершину C
  3. Сделать A левым сыном C, а B – правым
  4. Записать в C значение встречаемости, равное сумме встречаемостей A и B
  5. Добавить C в очередь с приоритетами

Алгоритм остановится, когда в очереди останется единственная вершина – корень дерева кода.

Какова трудоемкость алгоритма Хаффмена?

Начальное построение очереди с приоритетами происходит за O(N), где N – количество символов.

На каждом шаге алгоритма нам требуется дважды извлечь минимум из двоичной кучи[2]: $O(\log_2 N)$, создание новой вершины и установка значений полей происходит за O(1), а добавление в кучу – снова за $O(\log_2 N)$. Размер кучи изменятся от N в начале алгоритма до 1 в конце, причем на каждом шаге размер кучи уменьшается на единицу (извлекаем две вершины, помещаем одну), следовательно общее количество шагов алгоритма равно N-1, то есть O(N).

Общая трудоемкость составляет, таким образом, $O(N*\log_2 N)$.

Алгоритм Хаффмена сложнее алгоритма Шеннона-Фено, но работает за асимптотически меньшее время. Кроме того, код, который он строит является *оптимальным*, а не почти оптимальным, как код, построенный алгоритмом Шеннона-Фено. Иначе говоря, используя алгоритм Хаффмена, мы получаем наименьшую из всех возможных длин кода для исходной строки.

Это обеспечивается тем, что данный алгоритм сопоставляет каждому символу кодовое слово максимально близкое по длине к количеству информации, которое этот символ несет в данном тексте.

7 Словарные алгоритмы

Казалось бы, раз алгоритм Хаффемна позволяет получить оптимальный код – задача сжатия данных решена, ведь лучше не получится. Однако это неверно. Алгоритм Хаффмена (как и алгоритм Шеннона-Фено) учитывает лишь частоту встречаемости символов.

Однако, как было сказано выше, повторяются не только символы, но и целые слова. Поэтому все современные архиваторы используют алгоритмы *словарной* группы – алгоритмы, учитывающие повторение не отдельных символов, а подстрок сжимаемого текста.

Здесь нужно отметить, что такие подстроки не обязательно будут словами в общепринятом смысле: цепочками символов, ограниченными пробелами. На самом деле гораздо чаще повторяются фрагменты слов и цепочки, содержащие пробелы и знаки препинания (например, сочетание запятая-пробел встречается очень часто). Кроме того, архивироваться может не только текстовая информация, но и изображения, звук и так далее.

В любом массиве данных могут обнаружиться повторяющиеся фрагменты. Словарные алгоритмы базируются на том, что учитываются такие повторения.

8 Алгоритм Лемпеля-Зива (LZ)

Первый словарный алгоритм был предложен в 1977 году А. Лемпелем и Я. Зивом. Этот алгоритм называется LZ77, после его опубликования появилось множество модификаций, все они образую группу алгоритмов LZ.

Идея алгоритма такова: имеется *словарь* – набор подстрок кодируемого текста, каждой из которых присвоен номер, номера начинаются с единицы. Считается, что в словаре под нулевым номером лежит пустая строка. Код представляет из себя последовательность записей вида (<Номер слова>, <Символ>,) здесь <Номер слова> - это индекс слова из словаря, а символ – просто некоторый символ входного текста, оставленный без изменений.

Вот пример словаря:

1 – A

2 – B

3 – BC

Вот пример кода: 0A0B2C3D3F3D

Это код для такого текста: ABBCBCDBCFBCD

[2] Число вершин в очереди только уменьшается (две заменяются одной), поэтому общее количество вершин в куче не превосходит N.

Вот подробная расшифровка:

Код	0, **A**	0, **B**	2, **C**	3, **D**	3, **F**	3, **D**
Текст	**A**	**B**	B**C**	BC**D**	BC**F**	BC**D**

С помощью того же словаря этот текст можно закодировать иначе: 1B2C3D3F3D – это будет короче, потому что мы использовали словарь в первой же кодовой паре. Однако при таком способе кодирования нужно вместе с кодом хранить словарь, а он занимает место.

Для расшифровки предыдущего кода словарь не нужен, если знать об одной договоренности: первое слово словаря – это расшифровка первой кодовой пары, второе – расшифровка второй и т.д.

Рассмотрим процесс расшифровки.

Первый шаг:

- Словарь пуст, точнее в нем есть только пустая строка с номером ноль.
- Расшифруем первую пару: 0A –> A. Это первый символ текста и первое слово словаря. Номер слова в этой паре обязательно должен быть нулем – иначе мы ее не расшифруем.
- Добавим A в словарь.

Второй шаг:

- В словаре первое слово – A.
- Расшифруем вторую пару: 0B -> B. Здесь появился новый символ, поскольку его не могло быть в словаре, номер слова также должен быть нулем.
- Добавим B в словарь.

Третий шаг:

- В словаре слова A и B.
- Расшифруем 2C -> BC, слово номер 2 уже есть в словаре.
- Поместим BC в словарь.

Четвертый шаг:

- В словаре слова A, B и BC.
- Расшифруем 3D -> BCD.
- Поместим BCD в словарь.

Пятый шаг:

- В словаре слова A, B, BC и BCD.
- Расшифруем 3F -> BCF.
- Поместим BCF в словарь.

Шестой шаг:

- В словаре слова A, B, BC и BCD, BCF.
- Расшифруем 3D -> BCD.
- BCD уже есть в словаре – словарь не изменяется.

Как построить такой код?

Аналогично: первый символ, A, кодируется как 0A, потому что его нет в словаре, и добавляется в словарь. Второй – также, потому что его тоже нет в словаре.

Далее мы выполняем стандартный шаг: считываем по одному символы текста и смотрим: если накопленная строка есть в словаре, считываем еще символ, а если нет (значит строка состоит из двух частей: строки из словаря, возможно, пустой, и последнего символа), выводим пару: код строки из словаря и последний символ, и заносим строку в словарь. На самом деле два первых шага тоже подчиняются этому правилу.

Рассмотрим процесс кодирования примера ABBCBCDBCFBCD.

1. Считываем A
    a. Строки «A» нет в словаре:
        i. записываем кодовую пару 0A (строка состоит из пустой строки из словаря с номером ноль и последнего символа A)
        ii. заносим строку A в словарь (номер 1)
2. Считываем B

   a. Строки «B» нет в словаре:
      i. записываем кодовую пару 0B
      ii. заносим строку B в словарь (номер 2)
3. Считываем B
   a. Строка «B» есть в словаре
4. Считываем C
   a. Строки «BC» нет в словаре:
      i. Выводим кодовую пару 2C (номер строки B и последний символ C)
      ii. Заносим BC в словарь (номер 3)
5. Считываем B
   a. Строка «B» есть в словаре
6. Считываем C
   a. Строка «BC» есть в словаре
7. Считываем D
   a. Строки «BCD» нет в словаре:
      i. Выводим кодовую пару 3D (номер строки BC и последний символ D)
      ii. Заносим BCD в словарь (номер 4)
8. Считываем B
   a. Строка «B» есть в словаре
9. Считываем C
   a. Строка «BC» есть в словаре
10. Считываем F
   a. Строки «BCF» нет в словаре:
      i. Выводим кодовую пару 3F (номер строки BC и последний символ F)
      ii. Заносим BCF в словарь (номер 5)
11. Считываем B
   a. Строка «B» есть в словаре
12. Считываем C
   a. Строка «BC» есть в словаре
13. Считываем B
   a. Строка «BCD» есть в словаре
14. Текст кончился
   a. Выводим код для текущей строки: 3D. Несмотря на то, что строка есть в словаре, мы должны вывести пару (номер, символ), поэтому мы берем не номер строки в словаре, а ее код как пары (строка, последний символ).

Как и в алгоритме RLE, здесь есть насущная проблема: сколько бит отвести под номер слова в словаре. От этого напрямую зависит размер словаря (количество номеров) и длина каждой кодовой пары, а значит и кода в целом. Обычно на практике под номер слова отводят 12 бит, что соответствует 4096 строкам в словаре (считая пустую). Для маленьких текстов это приводит лишь к увеличению размера данных после «архивации», но для данных большого объема – дает хороший результат, поскольку учитывает большое количество информации о совпадениях.

Использование словаря позволяет учитывать повторения фрагментов текста. Словарь строится *адаптивно*, то есть при его построении учитываются свойства конкретного текста.

Кроме этого, преимуществом алгоритмов группы LZ является то, что словарь строится прямо при просмотре текста – не требуется предварительный проход для подсчета встречаемостей символов, как для алгоритмов Хаффмена и Шеннона-Фено.

9 Как хранить словарь

Словарь можно хранить как массив строк, но это дает довольно большую трудоемкость поиска совпадения. Это важно только на этапе кодирования, когда нужно искать, при расшифровке же массив – лучший вариант.

Некоторое улучшение времени поиска даст сбалансированное дерево, только вместе с каждым словом нужно хранить его номер в словаре (то есть номер в порядке добавления, а не в отсортированном наборе), поскольку эти номера участвуют в кодировании..

Еще лучший результат даст хеш-таблица. Это один из наиболее ярких примеров использования хеш-таблиц. В хеш-таблице также нужно хранить номера строк в порядке добавления.

Вообще говоря, если о сжимаемых данных многое известно заранее, например, мы знаем, что сжимаем только тексты на русском языке, можно заранее составить фиксированный словарь – и больше об этом не думать, но такой словарь никогда не даст такого качества сжатия как словарь построенный специально для конкретного текста.

Что делать, если словарь переполнится?

Такая ситуация вполне возможна: в словаре может быть конечное число строк, потому что конечна длина номера слова, значит при достаточной длине текста мы добавим в словарь слишком много строк.

В этом случае придется перестраивать словарь. Самый простой способ – это очистить его и начать «с чистого листа».

Кроме того, можно удалить из словаря те подстроки, которые встречались слишком редко, но это потребует ухищрений, поскольку словарь уже не будет так легко восстанавливаться при расшифровке.

10 Алгоритм LZ77

В оригинальном алгоритме LZ77 предлагалась очень интересная, но довольно сложная для эффективной реализации идея: словарем считаются последние N символов текста (N по-прежнему зависит от длины номера слова). Вместо кодовой пары используется кодовая тройка: номер слова заменяется смещением первого символа слова и длиной слова. Словарь представляет собой «скользящее окно», которое перемещается за указателем чтения.

```
AB [ ABCBCD ] ABBCXDSF
   словарь    ↑ -- указатель чтения
```

Чтобы закодировать строку AB в данном примере, найдем ее вхождение в «словарь»: это два символа, начиная с первого, то есть в кодовой тройке эта строка будет кодироваться как (1, 2) – сначала смещение (номер первого символа), потом – длина. Тройка же будет выглядеть так: (1, 2, B) – для строки ABB, которой нет в словаре. Добавление нового слова в словарь реализуется очень просто: нужно переместить окно дальше.

```
ABABC [ BCDABB ] CXDSF
      словарь    ↑ -- указатель чтения
```

Теперь в словаре есть строка ABB, а строка ABC, встречавшаяся давно, вытеснена из словаря.

Такая реализация словаря довольно эффективна с точки зрения качества сжатия: рассматривается больше вариантов совпадений, чем в приведенном выше алгоритме, поскольку совпадение может начинаться не только с первого символа некоторой ранее закодированной последовательности, но и с ее середины. Однако необходимость многократного поиска подстроки делает этот алгоритм довольно медленным. Впрочем, использование соответствующих структур данных позволяет решить эту проблему.

Давайте закодируем наш пример с помощью алгоритма LZ77 с длиной словаря в 3 символа.

Пример: ABBCBCDBCFBCD

1. Словарь пуст, считываем A

   a. Строки A нет в словаре
      i. выводим код: 0, 0, A
      ii. сдвигаем окно словаря
2. В словаре символ A, считываем B
   a. Строки A нет в словаре
      i. выводим код: 0, 0, B
      ii. сдвигаем окно словаря
3. В словаре AB, считываем B
   a. Строка B есть в словаре
4. Считываем C
   a. Строки BC нет в словаре
      i. выводим код: 1, 1, C – строка из словаря начинается с первого символа и имеет длину 1, за ней идет символ C
      ii. сдвигаем окно словаря на 2
5. В словаре BBC, считываем B
   a. Строка B есть в словаре
6. Считываем C
   a. Строка BC есть в словаре
7. Считываем D
   a. Строки BCD нет в словаре
      i. выводим код: 2, 2, D – строка из словаря начинается со второго символа и имеет длину 2, за ней идет символ D
      ii. сдвигаем окно словаря на 3
8. В словаре BCD, считываем B
   a. Строка B есть в словаре
9. Считываем C
   a. Строка BC есть в словаре
10. Считываем F
   a. Строки BCF нет в словаре
      i. выводим код: 1, 2, F
      ii. сдвигаем окно словаря на 3
11. В словаре BCF, считываем B
   a. Строка B есть в словаре
12. Считываем C
   a. Строка BC есть в словаре
13. Считываем D
   a. Строки BCD нет в словаре
      i. выводим код: 1, 2, D
      ii. сдвигаем окно словаря на 3
14. Текст закончился

Ясно, что при раскодировании словарь будет строиться из уже раскодированных символов естественным образом. Ясно, что проблема переполнения словаря здесь не возникает.

Обратите внимание, что «память» алгоритма сильно зависит от длины словаря: если при явном хранении словаря второе вхождение BCD уже обнаруживалось в словаре, то здесь алгоритм успел «забыть» о нем, передвигая окно.

В алгоритме с явным хранением словаря нам понадобилось шесть пар. Если считать, что словарь имел размер 16 строк (четыре бита на номер слова), то мы получаем 6 * (4 + 8) = 66 бит.

В LZ77 нам понадобилось шесть троек, по два бита на номер и длину (и то, и другое может быть в диапазоне от 0 до 3) – то есть столько же памяти.

Однако все зависит от размера словаря и длины текста.

11 Источники

- Семенюк В.В., «Экономное кодирование дискретной информации»
- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К., «Алгоритмы: построение и анализ»