МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНА МЕТАЛУРГІЙНА АКАДЕМІЯ УКРАЇНИ

РОБОЧА ПРОГРАМА,

методичні вказівки та індивідуальні завдання

до вивчення дисципліни «Основи теорії інформації»

для студентів спеціальності 122-комп'ютерні науки

Дніпро НМетАУ 2019

УДК 681.3.06+519.68

Методичні вказівки з дисципліні «Основи теорії інформації». Для студентів напряму 0501 01 – «Комп'ютерні науки» / Укл.: О.І. Деревянко, Т.М. Фененко – Дніпро: НМетАУ, 2019. – 34 с.

Методичні вказівки є практичною частиною комплексу навчально-методичних матеріалів з дисципліні «Основи теорії інформації», в якій розглянуті загальні принципи і математичні моделі перетворення сигналів при цифровій обробці, базові алгоритми та методи програмної реалізації і моделюванняданих.

Методичні вказівки призначені для студентів напряму підготовки 122 – «Комп'ютерні науки», а також для слухачів курсів підвищення кваліфікації, студентів і аспірантів інших спеціальностей.

Укладачі: О.І. Деревянко, кандидат технічних наук, професор,

Т.М. Фененко, ст. викладач

Відповідальний за випуск: О.І. Михальов, д-р техн. наук, проф.

Рецензент: В.І. Корсун, доктор технічних наук, професор

Друкується за авторською редакцією.

Затверджено на засіданні кафедри інформаційних технологій і систем, протокол № 9 від 06.03.2019.

Підписано до друку 10.05.2019. Формат 60х84 1/16. Папір типогр.

Друк різограф. Облік.-вид. арк 4,75. Умов. друк. арк. 3,40.

Тираж 100 пр. Замовл. № 19/12.

ДНВП «Системні технології»

Возникновение теории информации обычно связывают с появлением фундаментальной работы К. Шеннона «Математическая теория связи» (1948). Однако в теорию информации органически вошли и результаты, полученные, например Р. Хартли, впервые предложившим количественную меру информации (1928), акад. В. А. Котельниковым, сформулировавшим важнейшую теорему о возможности представления непрерывной функции совокупностью ее значений в отдельных точках отсчета (1933) и разработавшим оптимальные методы приема сигналов на фоне помех (1946), акад. А. Н. Колмогоровым, внесшим огромный вклад в статистическую теорию колебаний, являющуюся математической основой теории информации (1941).

В последующие годы теория информации получила дальнейшее развитие в трудах А. Н. Колмогорова, А. Я. Хинчина, В. И. Сифорова, Р. Л. Добрушина, М. С. Пинскера, А. Н. Железнова, Л. М. Финка, В. Макмиллана, А. Файнстейна, Д. Габора, Р. М. Фано, Ф. М. Вудворта, С. Гольдмана, Л. Бриллюэна и др.

К теории информации в ее классической постановке относят результаты решения ряда фундаментальных теоретических вопросов, касающихся повышения эффективности функционирования систем связи. В первую очередь:

анализ сигналов как средства передачи сообщений, включающий вопросы оценки переносимого «количества информации»;

анализ информационных характеристик источников сообщений и каналов связи и обоснование принципиальной возможности кодирования и декодирования сообщений, обеспечивающих предельно допустимую скорость передачи сообщений по каналу связи, как при отсутствии, так и при наличии помех.

К компетенции теории информации относят все проблемы и задачи, в формулировку которых входит понятие информации. Ее предметом считают изучение процессов, связанных с получением, передачей, хранением, обработкой и использованием информации.

Попытки широкого использования идей теории информации в различных областях науки связаны с тем, что в основе своей эта теория математическая. Основные ее понятия (энтропия, количество информации, пропускная способность) определяются только через вероятности событий.

Под информацией, содержащейся в сообщении, будем понимать долю устраненной неопределенности о результате физического процесса. Поэтому, мерой информации, содержащейся в сообщении является мера, связанная с затратами на передачу сообщения.

Условимся, что сообщения отображают результат случайных событий. Таким образом, в качестве источника сообщений рассмотрим произвольное дискретное множество X и каждому x<□X поставим в соответствие вероятность p(x).

Собственной информацией $I(x)$ сообщения x, выбираемого из дискретного ансамбля $X = \{x, p(x)\}$, называется величина

$$I(x) = -\log p(x), \qquad (1)$$

В (1) не указано основание логарифма. Далее, всякий раз подразумевается, что логарифмы вычисляются по основанию 2, если не оговорено другое. Это соответствует измерению информации в **битах**. Если логарифм вычисляется по натуральному основанию, единицей измерения информацией будет **нат.** Если основание десятичное - информация измеряется в **хартли.** Предпочтение в использовании битов связано с использованием двоичной системы счисления в компьютерных системах. В теоретико-информационных исследованиях часто предпочитают наты, что связано с операцией дифференцирования.

Из определения собственной информации и свойств логарифма непосредственно вытекают следующие свойства собственной информации.

1. Неотрицательность: $I(x) \geq 0, x < X$.

2. Монотонность: если $x_1, x_2 < X$, $p(x_1) \geq p(x_2)$, то $I(x_1) \leq I(x_2)$

3. Аддитивность. Для независимых сообщений $x_1,...,x_n$ имеет место равенство

$$I(x_1,...,x_n) = \sum_{i=1}^{n} I(x_i).$$

Энтропия

Определенная в (1) мера информации является случайной величиной, т.е. собственная информация сообщения x дискретного ансамбля $X = \{x, p(x)\}$ характеризует "степень неожиданности" конкретного сообщения. Математическое ожидание $I(x)$ по ансамблю $X = \{x, p(x)\}$ будет неслучайной величиной, характеризующей количество информации всего ансамбля *X*.

Если информация в сообщении полностью устраняет неопределенность всех возможных исходов ансамбля событий *X*, то

$$I(X) = H(X),$$

где H(X), - энтропия или мера неопределенности исходов ансамбля событий X.

Таким образом, энтропией дискретного ансамбля $X = \{x, p(x)\}$ называется величина

$$H(X) = \mathbf{M}[-\log p(x)] = -\sum p(x) \log p(x).$$

Можно интерпретировать энтропию как количественную меру априорной неосведомленности о том, какое из сообщений о исходе физического процесса будет порождено источником. Рассмотрим некоторые свойства энтропии.

1. $H(X) \geq 0$.

2. $H(X) \leq \log X$. Равенство имеет место в том случае, когда элементы ансамбля X равновероятны.

3. Если для двух ансамблей X и Y распределения вероятностей представляют собой одинаковые наборы чисел (отличаются только порядком следования элементов), то $H(X) = H(Y)$.

4. Если ансамбли X и Y независимы, то $H(XY) = H(X) + H(Y)$.

5. Энтропия – выпуклая функция распределения вероятностей на элементах ансамбля *X*.

6. Пусть $X = \{x, p(x)\}$ и $A \subset X$. Введем ансамбль $X' = \{x, p2(x)\}$, задав распределение вероятностей $p2(x)$ следующим образом

$$p2(x) = \begin{cases} \dfrac{P(A)}{|A|}, & x \in A, \\ p(x), & x \in A. \end{cases}$$

Тогда $H(X') \geq H(X)$. Иными словами, «выравнивание» вероятностей элементов ансамбля приводит к увеличению энтропии.

Сформулированные выше свойства имеют простую интерпретацию, если вспомнить, что энтропия определяет средние затраты на представление информации в двоичной форме. Рассмотрим источник сообщений с объемом алфавита равным $256=2^8$. При любом распределении вероятностей появления букв в сообщениях источника, одного байта на букву достаточно для хранения сообщений. Свойство

Так как, когда все буквы источника равновероятны, энтропия источника равна 8 бит. Предположим теперь, что речь идет о текстовом файле. Если использовать в качестве вероятностей букв их частоты появления в тексте большого объема, то, получим величину энтропии, близкую к 4,5. Это означает, что буква текста может быть представлена в среднем 4,5 битами и для ее хранения не следует тратить целый байт. Из свойства 4 следует: что затраты на хранение последовательности букв можно

подсчитать как сумму затрат на каждую букву в том случае, когда буквы независимы. Это условие заведомо не выполняется для осмысленного текста.

7. Пусть задан ансамбль X и на множестве его элементов определена функция g(x). Введем ансамбль Y □{y □□g(x)}. Тогда H(Y) ≤□H(X).

Равенство имеет место тогда, когда функция $g(x)$ обратима. Смысл последнего утверждения состоит в том, что *обработка сообщения не приводит к увеличению информации (закон сохранения информации)*.

Рассмотрим двоичный ансамбль $X = \{0,1\}$. Пусть $p(1)=$□p. Тогда $p(0)=1-p=q$. Энтропия этого ансамбля равна

$$H(X)= -p\log p -q\log q= h(p),$$

где $h(p)$ энтропия двоичного ансамбля. Построим график (рис.1) зависимости $h(p)$ от p при p□[0,1]. Начнем с крайних точек $p=0$ и $p=1$. В обоих случаях имеет место неопределенность. Используя правило Лопиталя вычислим значения $h(0) = h(1) = 0$. Кроме того, функция $h(p)$ симметрична относительно точки максимума $p=1/2$.

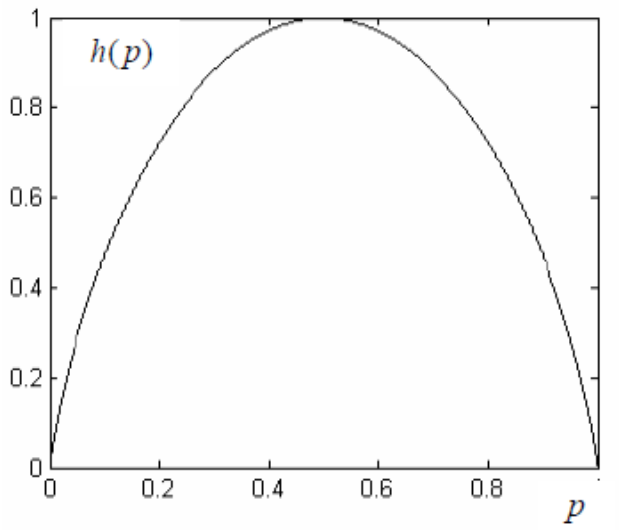


Рис. 1

Программисты и инженеры, обрабатывая информацию, называют битом двоичный разряд в формате двоичного числа или значение двоичного сигнала. В теории информации один бит – максимальное среднее количество информации, передаваемое двоичным сигналом. Однако, если вероятность появления единицы равна 0,1, то энтропия двоичного ансамбля равна 0,469. т.е. меньше 1 бита.

Условная энтропия

Как уже отмечалось, для эффективного кодирования информации необходимо учитывать статистическую зависимость сообщений. Наша ближайшая цель – научиться подсчитывать информационные характеристики последовательностей зависимых сообщений. Начнем с двух сообщений.

Рассмотрим ансамбли X = {x} и Y = {y} и их произведение XY = {(x, y), p(x, y)}. Для любого фиксированного y∈ Y можно построить условное распределение вероятностей p(x | y) на множестве X и для каждого x∈ X подсчитать собственную информацию

$$I(x \mid y) = -\log p(x \mid y),$$

которую называют условной собственной информацией сообщения x при фиксированном y.

Ранее мы назвали энтропией ансамбля X среднюю информацию сообщений x∈ X . Аналогично, усреднив условную информацию I (x | y) по x∈ X , получим величину Σ∈ = − x X

$$H(X \mid y)\ p(x \mid y) \log p(x \mid y), \quad (1.5.1)$$

называемую условной энтропией X при фиксированном y∈ Y. Заметим, что в определении (1.5.1) имеет место неопределенность в случае, когда p(x | y) = 0. Ранее отмечалось, что выражение вида z log z стремится к нулю при z →0 и на этом основании мы считали слагаемые энтропии, соответствующие буквам x с вероятностью p(x) = 0, равными нулю. Точно также в (1.5.1) мы считаем равными нулю слагаемые, для которых p(x | y) = 0 .

Вновь введенная энтропия H(X | y) – случайная величина, поскольку она зависит от случайной переменной y. Чтобы получить неслучайную информационную характеристику пары вероятностных ансамблей, нужно выполнить усреднение в (1.5.1) по всем значениям y. Величина

$$\Sigma\ \Sigma\in\ \in = -x\ X\ y\ YH(X \mid Y)\ p(x, y) \log p(x \mid y)$$

называется условной энтропией ансамбля X при фиксированном ансамбле Y . Отметим ряд свойств условной энтропии.

Свойство 1. $H(X \mid Y) \geq 0$.

Свойство 2. $H(X \mid Y) \leq H(X)$, причем равенство имеет место в том и только в том случае, когда ансамбли X и Y независимы.

Свойство 3. $H(XY) = H(X) + H(Y \mid X) = H(Y) + H(X \mid Y)$.

Свойство 4. (...) ( ) (|) (|) .... (| ,...,) 1 1 2 1 3 1 2 1 −1 = + + + + n n n H X X H X H X X H X X X H X X X .

Свойство 5. $H(X \mid YZ) \leq H(X \mid Y)$ причем равенство имеет место в том и только в том случае, когда ансамбли X и Y условно независимы при всех z∈ Z.

Дискретные случайные последовательности

Задача равномерного кодирования дискретного источника

Теперь, когда известна теоретическая характеристика информационной производительности источника, можно перейти к решению практической задачи сжатия информации. Однако в данном параграфе мы рассмотрим метод равномерного кодирования, который не имеет почти никакого практического значения. Более того, он, по сути, не является сжатием информации без потерь. Тем не менее, изучение равномерного кодирования представляется нужным по двум причинам.

Во-первых, станет еще понятнее, почему энтропия совпадает с предельно достижимой скоростью передачи (хранения) информации (эта скорость называется скоростью создания информации источником).

Во-вторых, на этом примере познакомимся с традиционным для теории информации способом формулировки теоретических результатов – доказательством прямой и обратной теорем кодирования .

При равномерном кодировании последовательность порождаемых источником сообщений , ,... 1 2 x x , x X i ∈ , i =1,2,... разбивается на блоки одинаковой длины N.

Каждый такой блок кодируется независимо от других блоков.

Для кодирования используется некоторый алфавит A ={a}, называемым кодовым, элементы алфавита называют кодовыми символами. Ограничимся рассмотрением двоичных кодов, то есть кодов над алфавитом A ={0,1}.

Кодом длины n называется любое подмножество C множества An , то есть любое подмножество множества последовательностей длины n . Элементы кода C называют кодовыми словами. Мощность кода |C | – это количество кодовых слов в коде C.

Скоростью равномерного кода называется величина log | | (бит/символ источника).

$$NR = C$$

Смысл такого определения скорости кодирования станет понятнее, если предположим, что кодом служит все множество последовательностей длины n, то есть C = An . При этом скорость кода равна (бит/символ источника).

$$NR = n$$

Если работа кодера состоит в том, что каждый блок из N символов источника заменяется на n двоичных символов, записываемых затем в запоминающее устройствоили передаваемых по каналу связи, то смысл определения скорости прозрачен. Скорость кода – это удельные затраты на передачу символа источника. Если же $C \subset An$, то величина log |C | представляет собой количество бит необходимое для указания номера кодового слова, а скорость кода – количество бит, затрачиваемых на передачу одной буквы источника.

Работа кодера (алгоритм кодирования) описывается отображением множества X N на множество слов кода C . Декодирование задается отображением C на X N . Если оба отображения взаимно-однозначные, то на выходе декодера можно будет получить точную копию передаваемой последовательности. Взаимно-однозначное кодирование возможно только тогда, когда

$$| X |N \leq | C | \quad (1.9.2)$$

или

$$R \geq \log | X | \geq H(X) .$$

Следовательно, если буквы источника не равновероятны (H(X) < log | X |), то скорость кода окажется заведомо больше энтропии источника.

Рассмотрим теперь ситуацию, когда условие (2) не выполнено. Тогда кодовых слов недостаточно для того, чтобы сопоставить каждой последовательности источника свое кодовое слово. Для некоторых последовательностей сообщений не найдется кодового слова, по которому декодер смог бы однозначно восстановить переданную информацию.

В общем виде процесс кодирования и декодирования можно описать следующим образом.

Выделим в X N подмножество T , такое, что |T |=|C |, и каждой последовательности из множества T сопоставим индивидуальное кодовое слово. Множество T мы будем называть множеством однозначно кодируемых последовательностей. Остальным последовательностям из X N сопоставим произвольные кодовые слова (например, всем последовательностям из T с сопоставим одно и то же кодовое слово). Декодер при получении некоторого кодового слова $c \in C$ будет выдавать получателю соответствующую этому слову последовательность из множества T.

Из описания следует, что каждый раз, когда источник породит последовательность из дополнения к множеству T , выход декодера не будет совпадать со входом кодера. Это событие называют ошибкой кодирования и вероятность

$$P P(T) e = x \notin$$

называется вероятностью ошибки кодирования.

Пример. Рассмотрим троичный постоянный источник X ={a,b, c} с распределением вероятностей p(a) =1/ 2 , p(b) =1/ 3 , p(c) =1/ 6 . Положим N = 2 . В этом случае множество X N состоит из 9 пар. Распределение вероятностей и пример дволичного кода длины n = 3 приведены в Таблице 1. Кодом служит множество C ={0,1}

(множество всех последовательностей длины 3). Скорость кода равна R = 3 / 2 бита на букву источника. Из таблицы ясно, что множество T имеет вид T ={aa, ab, ac, ba,bb,bc, ca, cb}. Вероятность ошибки P = p(cc) =1/ 36. При появлении на выходе источника последовательности cc декодер будет выдавать получателю последовательность cb .

Таблица 1

Пример равномерного кода

Последовательность источника	Вероятность	Кодовое слово
aa	¼	000
ab	1/6	001
ac	1/12	010
ba	1/6	011
bb	1/9	100
bc	1/18	101
ca	1/12	110
cb	1/18	111
cc	1/36	111

Отметим, что в данном примере кодирование могло быть и другим, но при любом другом выборе множества T вероятность ошибки была не меньше 1/36. В то же время, выбор кодового слова для последовательности из дополнения к T не влияет на вероятность ошибки.

Итак, задача построения равномерного кода со скоростью R для последовательностей длины n эквивалентна задаче выбора 2nR однозначно кодируемых сообщений. Хотелось бы построить такой код, который обеспечивал бы одновременно малую скорость и малую вероятность ошибки. Поскольку скорость может бать уменьшена только за счет уменьшения множества T , с уменьшением скорости неизбежно увеличивается вероятность ошибки. Представляет интерес ответ на вопрос: с какой скоростью возможно кодирование источника при пренебрежимо малой вероятности ошибки?

Для заданного стационарного источника число H называется скоростью создания информации, если для любого R > H существует равномерный код со скоростью R, обеспечивающий сколь угодно малую вероятность ошибки, и, в то же время, при любойскорости кода R < H вероятность ошибки не может быть сделана меньше некоторой положительной величины ε .

Для того, чтобы утверждать, что константа H является скоростью создания информации для данной модели источника, нужно доказать два утверждения. Первое устанавливает, что при R > H достижима сколь угодно малая вероятность ошибки. Это утверждение называют прямой теоремой кодирования. Второе утверждение состоит в том, что при R < H вероятность ошибки не может быть сделана произвольно малой. Это утверждение называется обратной теоремой кодирования.

Мы покажем, что для стационарных источников скорость создания информации совпадает с энтропией на сообщение источника, т. е. что имеют место равенства H H(X | X) H (X) ∞= ∞ = .

В определении понятия «скорость создания информации источником» неявно участвует ограничение на множество способов кодирования, а именно, предполагается, что блоки из фиксированного числа сообщений n преобразуются в кодовые слова фиксированной

длины N . Для обозначения кодов такого типа используют аббревиатуру FF (fixed-to-fixed). Помимо FF-кодов можно рассматривать VF-коды (variable-to-fixed), преобразующие блоки переменной длины в кодовые слова фиксированной длины, FV-коды (fixed-to-variable), когда для кодирования блоков фиксированной длины используются коды с переменной длиной кодовых слов, и наконец самый широкий класс кодов – VV-коды (variable-to-variable), допускающие разбиение сообщений на блоки переменной длины и кодирование блоков неравномерными кодами. Строго говоря, чтобы утверждать, что число H – скорость создания информации источником, при доказательстве обратной теоремы кодирования нужно допустить выбор способов кодирования из любого из четырех перечисленных множеств.

МАРКОВСКИЕ ИСТОЧНИКИ СООБЩЕНИЙ

Модель дискретного источника сообщений имеет сравнительно узкую область применения, поскольку реальные источники вырабатывают слова при наличии статистической зависимости между буквами. В реальных источниках вероятность выбора какой-либо очередной буквы зависит от всех предшествующих букв. Многие реальные источники достаточно хорошо описываются марковскими моделями источника сообщений. Согласно указанной модели условная вероятность выбора источником очередной x_{i_k}, буквы зависит только от ν предшествующих. Математической моделью сообщений, вырабатываемых таким источником, являются цепи Маркова ν-ãî порядка. В рамках указанной модели условная вероятность выбора ik -й буквы

$$p(x_{i_k} | x_{i_{k-\nu}} \ldots, x_{i_{k-\nu-1}}, \ldots, x_{i_1}) = p(x_{i_k} | x_{i_{k-1}}, \ldots, x_{i_{k-\nu}}) .$$

Если последнее равенство не зависит от времени, то есть справедливо при любом значении k, источник называется однородным. Однородный марковский источник называется стационарным, если безусловная вероятность выбора очередной буквы не зависит от k $(p(x_{i_k}) = p(x_i))$. В дальнейшем будем иметь дело только со стационарными источниками. Вычислим производительность источника для простой цепи Маркова (ν=l). В этом случае вероятность

$$p(x_{i_1}, \ldots, x_{i_n}) = p(x_{i_1}) p(x_{i_2} | x_{i_1}) \ldots p(x_{i_n} | x_{i_{n-1}}) .$$

Прологарифмировав последнее равенство, получим

$$-\log p(x_{i_1} \ldots x_{i_n}) = -\log p(x_{i_1}) - \log p(x_{i_2} | x_{i_1}) - \ldots - \log(x_{i_n} | x_{i_{n-1}}) .$$

Это равенство показывает, что индивидуальное количество информации, которое несет слово, равно количеству информации, которое несет первая буква, плюс количество информации, которое несет вторая буква при условии, что первая буква уже принята, и т. д.

Усредняя равенство по всем словам, получим количество информации, которое в среднем несет каждое слово:

$$H(X_1,\ldots,X_n) = H(X_1) + H(X_2|X_1)+\ldots+H(X_n|X_{n-1}).$$

Поскольку источник стационарный, то энтропия не зависит от k и равна

$$H(X_1,\ldots,X_n) = H(X_1) + (n-1)H(X_k|X_{k-1}) \le nH(X).$$

Подставляя полученный результат в (6) и учитывая, что всегда $H(X) \le \log m_X$, имеем

$$H_И = \lim_{n\to\infty}\left(\frac{H(X)}{n} + \frac{n-1}{n}H(X_k|X_{k-1})\right) = H(X_k|X_{k-1}).$$

В случае марковской цепи ν-ãî порядка Ни вычисляется аналогично и равна НИ=Н(X ν +1|X ν,..., X1).

Таким образом, производительность марковского источника равна неопределенности выбора очередной буквы при условии, что известны v предшествующих.

Для производительности марковского источника всегда справедливо неравенство

$$H_И \le H(X) \le \log m_X.$$

Максимального значения, равного logmX, производительность источника достигает, когда отсутствует статистическая зависимость между буквами в слове и когда все буквы алфавита вырабатываются с равными вероятностями. Очевидно, максимальная производительность источника полностью определяется размером алфавита тХ .

Для того чтобы характеризовать, насколько полно использует источник возможности алфавита, вводится параметр

$$r = \frac{H_{\max}(X) - H_И}{H_{\max}(X)}$$

называемый избыточностью.

Для передачи заданного количества информации, равного I, требуется п=I/НИ букв, если производительность источника равна НИ. В случае, когда производительность источника достигает своего максимального значения, равного Hmax (X)= =logmX, для передачи того же количества информации I требуется минимальное количество букв, равное п0=I/Hmax(X).

Отсюда $\frac{H_И}{H_{\max}(X)} = \frac{n_0}{n}$ I=пНИ=п0Hmax или . Учитывая последнее равенство, выражение для. избыточности можно записать в виде

$$r = 1 - \frac{H_И}{H_{\max}(X)} = \frac{n-n_0}{n}.$$

Таким образом, избыточность показывает, какая часть букв в слове не загружена информацией.

Пример 1. Определить избыточность источника, если он вырабатывает статистически независимую последовательность из единиц и нулей соответственно с вероятностями, равными p=0,3 и q=0,7.

Решение. Поскольку символы в последовательности статистически независимы, то производительность источника

$$H_{И} = H(X) = -p\log p - q\log q \approx 0{,}88 \text{ бит}.$$

Максимально возможная производительность источника $H_{\max}(X) = \log m_X = 1$, поскольку mX=2. При этом символы 1 и 0 должны вырабатываться с равными вероятностями (p=q= =0,5). Отсюда

$$r = 1 - \frac{0{,}88}{1} = 0{,}12.$$

Пример 2. Определить избыточность стационарного марковского источника, алфавит которого состоит из двух символов: 0 и 1. Вырабатываемая источником последовательность представляет собой простую цепь Маркова. Заданы следующие значения условных вероятностей

$$p(x_{i_{k+1}} | x_{i_k}) \qquad (i_{k+1} = \overline{1,2}, \quad i_k = \overline{1,2}):$$

Решение. Безусловную вероятность того, что (k+1)-м символом последовательности будет нуль, по формуле полной вероятности можно представить в виде

$$p_{k+1}(0) = p_k(0)p(0|0) + [1 - p_k(0)]p(0|1).$$

В правую часть неравенства входит вероятность pk(0) того, что k-ì символом последовательности будет нуль. В силу стационарности источника $p_{k+1}(0) = p_k(0) = p(0)$. Подставив в равенство значения p(0|0) и p(0|1), получим p(0)=0,125 , p(1)=1— p(0)=0,875.

Производительность источника

$$H_{И} = p(0)H(X_{k+1}|0) + p(1)H(X_{k+1}|1) = -0{,}125 \times$$
$$\times (0{,}3\log 0{,}3 + 0{,}7\log 0{,}7) - 0{,}875(0{,}1\log 0{,}1 +$$
$$+ 0{,}9\log 0{,}9) \approx 0{,}51,$$

а избыточность источника

$$r = 1 - \frac{H_{И}}{H_{\max}(X)} = 1 - \frac{0{,}51}{1} = 0{,}49,$$

где Hmax(X)=1.

Когда отношение v/n стремится к нулю, при неограниченном возрастании п марковский источник вырабатывает типичные последовательности, количество которых $Q \approx 2^{(n-\nu)H_{И}}$ или приближенно $Q \approx 2^{nH_{И}}$.

СПОСОБЫ СЖАТИЯ ИНФОРМАЦИИ

Сжатие данных можно разделить на два основных типа:

1) сжатие без потерь (или полностью обратимое);

2) сжатие с потерями (когда несущественая часть данных утрачивается и полное восстановление невозможно).

Первый тип сжатия применяют, когда данные важно восстановить после сжатия в неискаженном виде, это важно для текстов, числовых данных и т.п. Полностью обратимое сжатие, по-определению, ничего не удаляет из исходных данных. Сжатие достигается только за счет иного, более экономичного, представления данных.

Второй тип сжатия применяют, в основном, для видеоизображений и звука. За счет потерь может быть достигнута более высокая степень сжатия. В этом случае потери при сжатии означают несущественное искажение изображения (звука) которые не препятствуют нормальному восприятию, но при сличении оригинала и восстановленной после сжатия копии могут быть замечены.

Как уже было сказано алгоритмы с потерями прменяются в основном для сжатия звука и изображений. Дело в том, что при сжатии такого рода информации, теряя какую-то часть данных при сжатии, мало теряем собственно полезных данных, а объем уменьшается существенно. В результате получаются какие-то искажения исходных данных, зависящие от того, какую долю данных мы теряем. Чем больше потери, тем выше степень сжатия и тем сильнее искажения.

Но умеренных искажений, как правило, человек не замечает. Это связано с особенностями его восприятия звука и изображений. Алгоритмы построены так, чтобы при разумной степени компрессии искажения были практически незаметны. Но, разумеется, профессиональный фотограф, к примеру, или человек близкой к нему профессии сразу опознает искажения фотографии, музыкант или человек с хорошим музыкальным слухом отличит предварительно сжатый звук от оригинала.

Существует много форматов графических файлов. Наиболее известны BMP, PCX, GIF, JPEG. Некоторые из них предусматривают сжатие.

Алгоритм GIF является примером тривиального алгоритма с потерями и
позволяет достичь хорошей степени сжатия. Сжатие достигается за счет урезания количества цветов до 256, поэтому при сохранении фотографии резко ухудшается цветопередача.

Более сложным является алгоритм с потерями JPEG. Достоинством стандарта является возможность изменения качества изображения в зависимости от требований к качеству и необходимого объема изображения.

Каждое изображение на входе делится на неперекрываемые блоки пикселей размером 8×8, затем каждый блок подвергается дискретно-косинусному преобразованию (ДКП), аналогичному преобразованию Фурье. Так как это преобразование двухкоординатное, то базисом ДКП являются функции с увеличивающимися частотами в горизонтальной и вертикальной плоскостях. Если после преобразования на приемную сторону были переданы все коэффициенты, то восстановленное изображение не уступает по качеству исходному. Когда абсолютно подлинного качества не требуется, то передаются не все коэффициенты, а лишь то их количество, которое обеспечит нужное качество изображения. В этом случае после ДКП каждый из 64 полученных коэффициентов по однообразной схеме квантируется с определенным квантирующим значением, выражающим весовое значение каждого коэффициента, субъективно влияющего на качество восприятия изображения человеческим глазом. Поскольку значения коэффициентов у ближайших блоков сильно коррелированны, то кодируются не сами коэффициенты, а разница в значениях коэффициентов предыдущего и текущего блоков. Ненулевые квантированные значения оставшихся коэффициентов и их местоположение сканируются зигзаговым проходом блока, а затем кодируются кодовыми словами переменной длины.

Декодер на приемной стороне осуществляет обратное преобразование.
Сначала для каждого блока пикселов декодируется битовый поток с целью определения значения и местоположения ненулевых значений коэффициентов ДКП. После восстановления всех ДКП – коэффициентов производится обратное преобразование, в результате чего получаем блок пикселов.
Чтобы предоставить пользователю возможность управлять характеристиками сжатия, в стандарте JPEG реализована концепция 64-элементной таблицы квантирования. Процесс квантирования заключается в делении каждого коэффициента ДКП на соответствующее ему значение из таблицы и последующего округления результата до ближайшего целого значения. Видно, что степень сжатия может быть задана квантируемым значением. Сильное увеличение степени сжатия приведет к появлению искажений. Поэтому необходимо достигать баланса между степенью сжатия и качеством изображения.
Рассмотрим теперь сжатие видео. Для начала примерно оценим объем одной секунды видеоданных. Пусть кадр имеет размер 320×240 и глубину цвета 16 бит. При таком разрешении и цветности качество изображения довольно низкое. Размер одного кадра 320×240×2 = 150 Кбайт. В секунду необходимо неменее 30 кадров для плавной их смены. Таким образом, 1 секунда видео такого формата занимает 150×30=4.5 Мбайт. При этом мы не учитываем звук, а также то, что для более высокого качества необходимо больший размер кадра.
Видно, что размер обычного фильма получается огромным. Но, к счастью, разработаны методы сжатия, которые позволяют сжать видеопоток в десятки раз. Например, это формат MPEG.
В основе семейства MPEG лежит тот факт, что видеопоток содержит избыточность в двух направлениях – пространственном и временном. Поэтому используется внутрикадровое и межкадровое кодирование —корреляция отдельных элементов изображения и отдельных кадров.
Кодирование по MPEG начинается с интерполяции видеоинформации, убирающей субъективную избыточность, содержащуюся в видео. После этого используются различные методы сжатия.
Например, энтропийное кодирование. Это тот самый алгоритм Хаффмана, о котором уже говорили, только здесь используются не отдельные символы, а цвета.
Другой метод – кодирование с предсказанием. Уменьшение избыточности осуществляется за счет определения окружения пиксела внутри кадра или между кадрами. Этот метод дает хороший результат, если существует хорошая пространственная корреляция ближайших друг к другу пикселов. Некоторые пиксели аппроксимируются с помощью ближайших к ним, а разница между действительным значением цвета и предсказанным – небольшая величина- подвергается энтропийному кодированию.

АЛГОРИТМЫ СЖАТИЯ БЕЗ ПОТЕРЬ

В настоящее время существует очень значительное количество алгоритмов сжатия без потерь, частично это открытые алгоритмы, частично коммерческие Открытые алгоритмы обычно рассматривают только саму идею, не останавливаясь на проблемах ее реализации в виде программы.
Коммерческие алгоритмы не публикуются и познакомится с ними невозможно, за исключением ознакомления с результатами работы программ на базе этих алгоритмов. Соответствующие программы (ZIP, ARJ, RAR и др.) достаточно известны и с ними можно познакомиться самостоятельно.

АЛГОРИТМ RUNNING (RLE)

Один из самых простых алгоритмов сжатия без потерь — так называемый алгоритм Running. Допустим, имеем цепочку одинаковых символов. Налицо явная избыточность информации. Сжатие осуществляется следующим образом:

1) подсчитываем количество одинаковых символов в цепочке;
2) вместо цепочки записываем, символ и сколько раз повторяется этот символ.
Например, строка из 40 пробелов. Записываем байт-код, показывающий что идет повторяющийся символ, затем число 40 (сколько раз повторяется), и, наконец, пробел (повторяющийся символ). Строка длиной 40 сжимается до 3 байт.
Данный алгоритм прост и в понимании, и в реализации, но он малоэффективен при применении его в одиночку. Существуют файлы, которые содержат повторяющиеся символы, но не могут быть сжаты описанным выше алгоритмом. Это происходит, когда повторяется не символы, а последовательность символов. Например: «абабабабабаб». Повторяющихся друг за другом символов нет, и сжать предыдущим алгоритмом нельзя. Приходится пользоваться расширенным алгоритмом. Он заключается в следующем:
1) Ищем не повторяющиеся символы, а повторяющиеся последовательности символов (в данном случае это «аб»).
2) Затем подсчитываем, сколько раз повторяется эта последовательность.
3) Записываем по следующему правилу: байт-код сигнала начала последовательности, количество повторяющихся строк (в нашем случае 6), и сама
строка (в нашем случае «аб»).
Но и это не предел. Дело в том, что могут существовать одинаковые фрагменты текста, располагающиеся в разных местах файла. Основная проблема состоит в том, чтобы найти эти фрагменты. Далее, его необходимо записать в определенную кодовую таблицу и приписать ему определенное значение. После этого можем заменять все фрагменты этим кодом-ссылкой.
Реализация такого поиска — весма сложная и трудоемкая задача. Хотя существуют особые ситуации, когда подобный поиск повторяющихся фрагментов может быть ускорен. Например, при передаче видеоизображения данные представляют собой последовательность кадров — картинок, состоящих из массива точек. Эти картинки имеют одинаковый размер и, что гораздо более важно часто предыдущая картинка слабо отличаются от последующей (простейший пример, снимается неподвижный объект). В этом случае реализация Running- подобного алгоритма достаточно проста. Достаточно записывать для каждого следующего кадра только его отличия от предыдущего и можно добыться существенного сжатия данных для медленно меняющихся изображений.
Вышеперечисленные алгоритмы работают, только если в файле имеются повторяющиеся фрагменты и символы.

АЛГОРИТМ ХАФФМАНА

Идея, лежащая в основе кода Хаффмана, достаточно проста. Вместо того чтобы кодировать все символы одинаковым числом бит (как это сделано, например, в ASCII кодировке, где на каждый символ отводится ровно по 8 бит), будем кодировать символы, которые встречаются чаще, меньшим числом бит, чем те, которые встречаются реже. Более того, потребуем, чтобы код был оптимален или, другими словами, минимально-избыточен.

Первым такой алгоритм опубликовал Дэвид Хаффман в 1952 году. Алгоритм Хаффмана двухпроходный. На первом проходе строится частотный словарь и генерируются коды. На втором проходе происходит непосредственно кодирование.

Стоит отметить, что за 50 лет со дня опубликования, код Хаффмана ничуть не потерял своей актуальности и значимости. Так с уверенностью можно сказать, что мы сталкиваемся с ним, в той или иной форме (дело в том, что код Хаффмана редко используется отдельно, чаще работая в связке с другими алгоритмами), практически

каждый раз, когда архивируем файлы, смотрим фотографии, фильмы, посылаем факс или слушаем музыку.

Определение 1: Пусть $A=\{a_1,a_2,...,a_n\}$ - алфавит из n различных символов, $W=\{w_1,w_2,...,w_n\}$ - соответствующий ему набор положительных целых весов. Тогда набор бинарных кодов $C=\{c_1,c_2,...,c_n\}$, такой что:

(1) c_i не является префиксом для c_j, при i!=j

(2) $\sum_{i=1}^{n} w_i|c_i|$ минимальна ($|c_i|$ длина кода c_i)

называется *минимально-избыточным префиксным кодом* или иначе *кодом Хаффмана*.

Замечания:

1. Свойство (1) называется *свойством префиксности*. Оно позволяет однозначно декодировать коды переменной длины.
2. Сумму в свойстве (2) можно трактовать как размер закодированных данных в битах. На практике это очень удобно, т.к. позволяет оценить степень сжатия не прибегая непосредственно к кодированию.
3. В дальнейшем, чтобы избежать недоразумений, под кодом будем понимать битовую строку определенной длины, а под минимально-избыточным кодом или кодом Хаффмана - множество кодов (битовых строк), соответствующих определенным символам и обладающих определенными свойствами.

Известно, что любому бинарному префиксному коду соответствует определенное бинарное дерево.

Определение 2: Бинарное дерево, соответствующее коду Хаффмана, будем называть *деревом Хаффмана*.

Задача построения кода Хаффмана равносильна задаче построения соответствующего ему дерева. Приведем общую схему построения дерева Хаффмана:

1. Составим список кодируемых символов (при этом будем рассматривать каждый символ как одноэлементное бинарное дерево, вес которого равен весу символа).
2. Из списка выберем 2 узла с наименьшим весом.
3. Сформируем новый узел и присоединим к нему, в качестве дочерних, два узла выбранных из списка. При этом вес сформированного узла положим равным сумме весов дочерних узлов.
4. Добавим сформированный узел к списку.
5. Если в списке больше одного узла, то повторить 2-5.

Приведем пример: построим дерево Хаффмана для сообщения S="A H F B H C E H E H C E A H D C E E H H H C H H H D E G H G G E H C H H".

Для начала введем несколько обозначений:

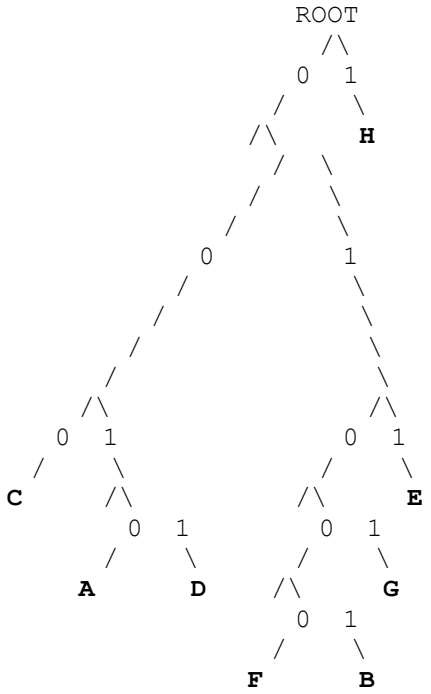
1. Символы кодируемого алфавита будем выделять жирным шрифтом: **A**, **B**, **C**.
2. Веса узлов будем обозначать нижними индексами: $\mathbf{A}_5$, $\mathbf{B}_3$, $\mathbf{C}_7$.

3. Составные узлы будем заключать в скобки: $((\mathbf{A}_5+\mathbf{B}_3)_8+\mathbf{C}_7)_{15}$.

Итак, в нашем случае A={**A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**}, W={2, 1, 5, 2, 7, 1, 3, 15}.

1. $\mathbf{A}_2\ \mathbf{B}_1\ \mathbf{C}_5\ \mathbf{D}_2\ \mathbf{E}_7\ \mathbf{F}_1\ \mathbf{G}_3\ \mathbf{H}_{15}$
2. $\mathbf{A}_2\ \mathbf{C}_5\ \mathbf{D}_2\ \mathbf{E}_7\ \mathbf{G}_3\ \mathbf{H}_{15}\ (\mathbf{F}_1+\mathbf{B}_1)_2$
3. $\mathbf{C}_5\ \mathbf{E}_7\ \mathbf{G}_3\ \mathbf{H}_{15}\ (\mathbf{F}_1+\mathbf{B}_1)_2\ (\mathbf{A}_2+\mathbf{D}_2)_4$
4. $\mathbf{C}_5\ \mathbf{E}_7\ \mathbf{H}_{15}\ (\mathbf{A}_2+\mathbf{D}_2)_4\ ((\mathbf{F}_1+\mathbf{B}_1)_2+\mathbf{G}_3)_5$
5. $\mathbf{E}_7\ \mathbf{H}_{15}\ ((\mathbf{F}_1+\mathbf{B}_1)_2+\mathbf{G}_3)_5\ (\mathbf{C}_5+(\mathbf{A}_2+\mathbf{D}_2)_4)_9$
6. $\mathbf{H}_{15}\ (\mathbf{C}_5+(\mathbf{A}_2+\mathbf{D}_2)_4)_9\ (((\mathbf{F}_1+\mathbf{B}_1)_2+\mathbf{G}_3)_5+\mathbf{E}_7)_{12}$
7. $\mathbf{H}_{15}\ ((\mathbf{C}_5+(\mathbf{A}_2+\mathbf{D}_2)_4)_9+(((\mathbf{F}_1+\mathbf{B}_1)_2+\mathbf{G}_3)_5+\mathbf{E}_7)_{12})_{21}$
8. $(((\mathbf{C}_5+(\mathbf{A}_2+\mathbf{D}_2)_4)_9+(((\mathbf{F}_1+\mathbf{B}_1)_2+\mathbf{G}_3)_5+\mathbf{E}_7)_{12})_{21}+\mathbf{H}_{15})_{36}$

В списке, как и требовалось, остался всего один узел. Дерево Хаффмана построено. Теперь запишем его в более привычном для нас виде.



Листовые узлы дерева Хаффмана соответствуют символам кодируемого алфавита. Глубина листовых узлов равна длине кода соответствующих символов.

Путь от корня дерева к листовому узлу можно представить в виде битовой строки, в которой "0" соответствует выбору левого поддерева, а "1" - правого. Используя этот механизм, мы без труда можем присвоить коды всем символам кодируемого алфавита. Выпишем, к примеру, коды для всех символов в нашем примере:

A=0010_{bin}	**C**=000_{bin}	**E**=011_{bin}	**G**=0101_{bin}
B=01001_{bin}	**D**=0011_{bin}	**F**=01000_{bin}	**H**=1_{bin}

Теперь все необходимое для того чтобы закодировать сообщение S. Достаточно просто заменить каждый символ соответствующим ему кодом:

S'="0010 1 01000 01001 1 000 011 1 011 1 000 011 0010 1 0011 000 011 011 1 1 1 000 1 1 1 0011 011 0101 1 0101 0101 011 1 000 1 1".

Оценим теперь степень сжатия. В исходном сообщении S было 36 символов, на каждый из которых отводилось по $[\log_2|A|]=3$ бита (здесь и далее будем понимать квадратные скобки [] как целую часть, округленную в положительную сторону, т.е. [3,018]=4). Таким образом, размер S равен 36*3=108 бит

Размер закодированного сообщения S' можно получить воспользовавшись замечанием 2 к определению 1, или непосредственно, подсчитав количество бит в S'. И в том и другом случае мы получим 89 бит.

Итак, нам удалось сжать 108 в 89 бит.

Теперь декодируем сообщение S'. Начиная с корня дерева будем двигаться вниз, выбирая левое поддерево, если очередной бит в потоке равен "0", и правое - если "1". Дойдя до листового узла мы декодируем соответствующий ему символ.

Ясно, что следуя этому алгоритму мы в точности получим исходное сообщение S.

Канонический код Хаффмана

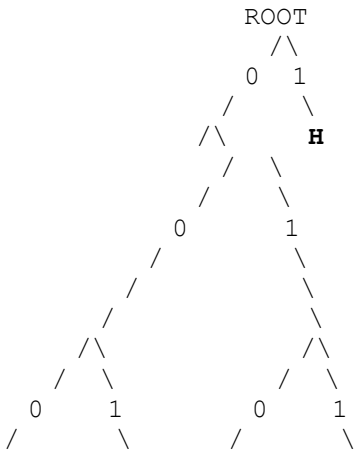
Как можно было заметить из предыдущего раздела, код Хаффмана не единственен. Мы можем подвергать его любым трансформациям без ущерба для эффективности при соблюдении всего двух условий: коды должны остаться префиксными и их длины не должны измениться.

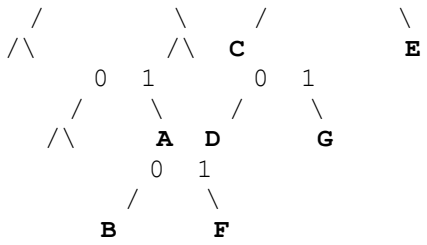
Определение 3: Код Хаффмана $D=\{d_1,d_2,\ldots,d_n\}$ называется *каноническим* [2], если:
(1) Короткие коды (если их дополнить нулями справа) численно больше длинных,
(2) Коды одинаковой длины численно возрастают вместе с алфавитом.

Далее, для краткости, будем называть канонический код Хаффмана просто каноническим кодом.

Определение 4: Бинарное дерево, соответствующее каноническому коду Хаффмана, будем называть *каноническим деревом Хаффмана*.

В качестве примера приведем каноническое дерево Хаффмана для сообщения S, и сравним его с обычным деревом Хаффмана.





Выпишем теперь канонические коды для всех символов нашего алфавита в двоичной и десятичной форме. При этом сгруппируем символы по длине кода.

B=$00000_{bin}=0_{dec}$ **A**=$0001_{bin}=1_{dec}$ **C**=$010_{bin}=2_{dec}$ **H**=$1_{bin}=1_{dec}$

F=$00001_{bin}=1_{dec}$ **D**=$0010_{bin}=2_{dec}$ **E**=$011_{bin}=3_{dec}$

G=$0011_{bin}=3_{dec}$

Убедимся в том, что свойства (1) и (2) из определения 3 выполняются:

Рассмотрим, к примеру, два символа: **E** и **G**. Дополним код символа **E**=$011_{bin}=3_{dec}$ (максимальная_длина_кода-длина_кода)=5-3=2 нулями справа: $\mathbf{E}'$=011 $00_{bin}=12_{dec}$, аналогично получим $\mathbf{G}'$=0011 $0_{bin}=6_{dec}$. Видно что $\mathbf{E}'>\mathbf{G}'$.

Рассмотрим теперь три символа: **A**, **D**, **G**. Все они имеют код одной длины. Лексикографически **A**<**D**<**G**. В таком же отношении находятся и их коды: 1<2<3.

Далее заметим, что порядковый номер любого листового узла, на занимаемом им уровне, численно равен коду соответствующего ему символа. Это свойство канонических кодов называют *числовым (Numerical property)*.

Поясним вышесказанное на примере. Рассмотрим символ **C**. Он находится на 3^{M} уровне (имеет длину кода 3). Его порядковый номер на этом уровне равен 2 (учитывая два нелистовых узла слева), т.е. численно равен коду символа **C**. Теперь запишем этот номер в двоичной форме и дополним его нулевым битом слева (т.к. 2 представляется двумя битами, а код символа **C** тремя): $2_{dec}=10_{bin}$=>0 10_{bin}, т.е. получили в точности код символа **C**.

Таким образом, можно сделать очень важный вывод: канонические коды вполне определяются своими длинами. Это свойство канонических кодов широко используется на практике.

Теперь вновь закодируем сообщение S, но уже при помощи канонических кодов:

Z'="0001 1 00001 00000 1 010 011 1 011 1 010 011 0001 1 0010 010 011 011 1 1 1 010 1 1 1 0010 011 0011 1 0011 0011 011 1 010 1 1"

Т.к. длины кодов не изменились, то размер закодированного сообщения не изменился: $|S'|=|Z'|$=89 бит.

Теперь декодируем сообщение Z', используя свойства канонических кодов.

Построим три массива: base[], symb[], offs[]. Где base[i] - количество нелистовых узлов на уровне i; symb[] - перестановка символов алфавита, отсортированная по двум ключам: первичный - длина кода, вторичный - лексикографическое значение; offs[i] - индекс массива symb[], такой что symb[offs[i]] первый листовой узел (символ) на уровне i.

В нашем случае: base[0..5]={?, 1, 2, 2, 1, 0}, symb[0..7]={B, F, A, D, G, C, E, H}, offs[0..5]={?, 7, ?, 5, 2, 0}.

Приведем несколько пояснений. base[0]=? и offs[0]=? не используются, т.к. нет кодов длины 0, а offs[2]=? - т.к. на втором уровне нет листовых узлов.

Теперь приведем алгоритм декодирования (CANONICAL_DECODE):

1. code = следующий бит из потока, length = 1
2. Пока code < base[length]
   code = code << 1
   code = code + следующий бит из потока
   length = length + 1
3. symbol = symb[offs[length] + code - base[length]]

Другими словами, будем вдвигать слева в переменную code бит за битом из входного потока, до тех пор, пока code < base[length]. При этом на каждой итерации будем увеличивать переменную length на 1 (т.е. продвигаться вниз по дереву). Цикл в (2) остановится когда мы определим длину кода (уровень в дереве, на котором находится искомый символ). Остается лишь определить какой именно символ на этом уровне нужен.

Предположим, что цикл в (2), после нескольких итераций, остановился. В этом случае выражение (code - base[length]) суть порядковый номер искомого узла (символа) на уровне length. Первый узел (символ), находящийся на уровне length в дереве, расположен в массиве symb[] по индексу offs[length]. Но нас интересует не первый символ, а символ под номером (code - base[length]). Поэтому индекс искомого символа в массиве symb[] равен (offs[length] + (code - base[length])). Иначе говоря, искомый символ symbol=symb[offs[length] + code - base[length]].

Приведем конкретный пример. Пользуясь изложенным алгоритмом декодируем сообщение $Z^{/}$.

$Z^{/}$="0001 1 00001 00000 1 010 011 1 011 1 010 011 0001 1 0010 010 011 011 1 1 1 010 1 1 1 0010 011 0011 1 0011 0011 011 1 010 1 1"

1. code = 0, length = 1
2. code = 0 < base[length] = 1
   code = 0 << 1 = 0
   code = 0 + 0 = 0
   length = 1 + 1 = 2
   code = 0 < base[length] = 2
   code = 0 << 1 = 0
   code = 0 + 0 = 0
   length = 2 + 1 = 3
   code = 0 < base[length] = 2
   code = 0 << 1 = 0
   code = 0 + 1 = 1

length = 3 + 1 = 4
code = 1 = base[length] = 1

3. symbol = symb[offs[length] = 2 + code = 1 - base[length] = 1] = symb[2] = **A**

1. code = 1, length = 1
2. code = 1 = base[length] = 1
3. symbol = symb[offs[length] = 7 + code = 1 - base[length] = 1] = symb[7] = **H**

1. code = 0, length = 1
2. code = 0 < base[length] = 1
   code = 0 << 1 = 0
   code = 0 + 0 = 0
   length = 1 + 1 = 2
   code = 0 < base[length] = 2
   code = 0 << 1 = 0
   code = 0 + 0 = 0
   length = 2 + 1 = 3
   code = 0 < base[length] = 2
   code = 0 << 1 = 0
   code = 0 + 0 = 0
   length = 3 + 1 = 4
   code = 0 < base[length] = 1
   code = 0 << 1 = 0
   code = 0 + 1 = 1
   length = 4 + 1 = 5
   code = 1 > base[length] = 0
3. symbol = symb[offs[length] = 0 + code = 1 - base[length] = 0] = symb[1] = **F**

Итак, декодировали 3 первых символа: **A**, **H**, **F**. Ясно, что следуя этому алгоритму получим в точности сообщение S.

Это, пожалуй, самый простой алгоритм для декодирования канонических кодов. К нему можно придумать массу усовершенствований. Подробнее о них можно прочитать в [5] и [9].

Вычисление длин кодов

Для того чтобы закодировать сообщение нам необходимо знать коды символов и их длины. Как уже было отмечено в предыдущем разделе, канонические коды вполне определяются своими длинами. Таким образом, наша главная задача заключается в вычислении длин кодов.

Оказывается, что эта задача, в подавляющем большинстве случаев, не требует построения дерева Хаффмана в явном виде. Более того, алгоритмы использующие внутреннее (не явное) представление дерева Хаффмана оказываются гораздо эффективнее в отношении скорости работы и затрат памяти.

На сегодняшний день существует множество эффективных алгоритмов вычисления длин кодов. Ограничимся рассмотрением лишь одного из них. Этот алгоритм достаточно прост, но несмотря на это очень популярен. Он используется в таких программах как zip, gzip, pkzip, bzip2 и многих других.

Вернемся к алгоритму построения дерева Хаффмана. На каждой итерации производен линейный поиск двух узлов с наименьшим весом. Ясно, что для этой цели больше подходит очередь приоритетов, такая как пирамида (минимальная). Узел с наименьшим весом при этом будет иметь наивысший приоритет и находиться на вершине пирамиды. Приведем этот алгоритм.

1. Включим все кодируемые символы в пирамиду.
2. Последовательно извлечем из пирамиды 2 узла (это будут два узла с наименьшим весом).
3. Сформируем новый узел и присоединим к нему, в качестве дочерних, два узла взятых из пирамиды. При этом вес сформированного узла положим равным сумме весов дочерних узлов.
4. Включим сформированный узел в пирамиду.
5. Если в пирамиде больше одного узла, то повторить 2-5.

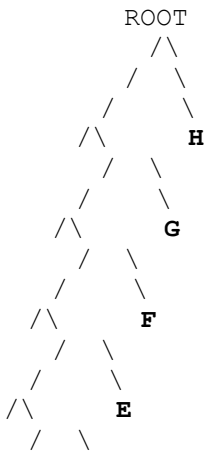
Будем считать, что для каждого узла сохранен указатель на его родителя. У корня дерева этот указатель положим равным NULL. Выберем теперь листовой узел (символ) и следуя сохраненным указателям будем подниматься вверх по дереву до тех пор, пока очередной указатель не станет равен NULL. Последнее условие означает, что мы добрались до корня дерева. Ясно, что число переходов с уровня на уровень равно глубине листового узла (символа), а следовательно и длине его кода. Обойдя таким образом все узлы (символы), ы получим длины их кодов.

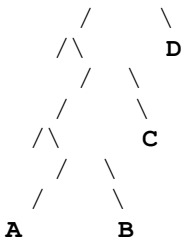
Максимальная длина кода

Как правило, при кодировании используется так называемая *кодовая книга (CodeBook)*, простая структура данных, по сути два массива: один с длинами, другой с кодами. Другими словами, код (как битовая строка) хранится в ячейке памяти или регистре фиксированного размера (чаще 16, 32 или 64). Для того чтобы не произошло переполнение, мы должны быть уверены в том, что код поместится в регистр.

Оказывается, что на N-символьном алфавите максимальный размер кода может достигать (N-1) бит в длину. Иначе говоря, при N=256 (распространенный вариант) мы можем получить код в 255 бит длиной (правда для этого файл должен быть очень велик: 2.292654130570773*10^53~=2^177.259)! Ясно, что такой код в регистр не поместится и с ним нужно что-то делать.

Для начала выясним при каких условиях возникает переполнение. Пусть частота i-го символа равна i-му числу Фибоначчи. Например: **A**-1, **B**-1, **C**-2, **D**-3, **E**-5, **F**-8, **G**-13, **H**-21. Построим соответствующее дерево Хаффмана.





Такое дерево называется *вырожденным*. Для того чтобы его получить частоты символов должны расти как минимум как числа Фибоначчи или еще быстрее. Хотя на практике, на реальных данных, такое дерево получить практически невозможно, его очень легко сгенерировать искусственно. В любом случае эту опасность нужно учитывать.

Эту проблему можно решить двумя приемлемыми способами. Первый из них опирается на одно из свойств канонических кодов. Дело в том, что в каноническом коде (битовой строке) не более $[\log_2 N]$ младших бит могут быть ненулями. Другими словами, все остальные биты можно вообще не сохранять, т.к. они всегда равны нулю. В случае N=256 нам достаточно от каждого кода сохранять лишь младшие 8 битов, подразумевая все остальные биты равными нулю. Это решает проблему, но лишь отчасти. Это значительно усложнит и замедлит как кодирование, так и декодирование. Поэтому этот способ редко применяется на практике.

Второй способ заключается в искусственном ограничении длин кодов (либо во время построения, либо после). Этот способ является общепринятым, поэтому остановимся на нем более подробно.

Существует два типа алгоритмов ограничивающих длины кодов. Эвристические (приблизительные) и оптимальные. Алгоритмы второго типа достаточно сложны в реализации и как правило требуют больших затрат времени и памяти, чем первые. Эффективность эвристически-ограниченного кода определяется его отклонением от оптимально-ограниченного. Чем меньше эта разница, тем лучше. Стоит отметить, что для некоторых эвристических алгоритмов эта разница очень мала ([6], [7], [8]), к тому же они очень часто генерируют оптимальный код (хотя и не гарантируют, что так будет всегда). Более того, т.к. на практике переполнение случается крайне редко (если только не поставлено очень жесткое ограничение на максимальную длину кода), при небольшом размере алфавита целесообразнее применять простые и быстрые эвристические методы.

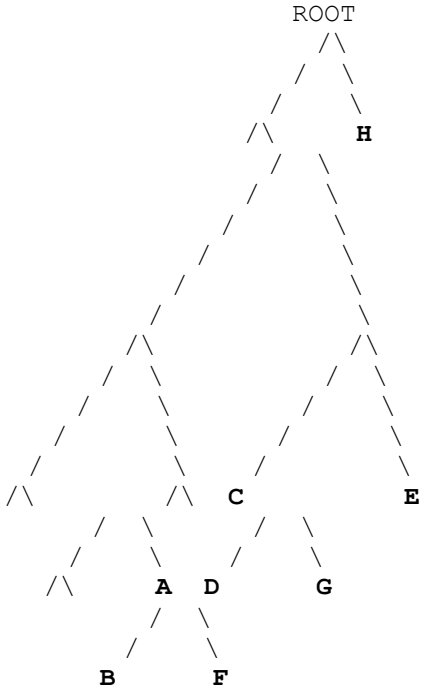
Рассмотрим один достаточно простой и очень популярный эвристический алгоритм. Он нашел свое применение в таких программах как zip, gzip, pkzip, bzip2 и многих других.

Задача ограничения максимальной длины кода эквивалентна задаче ограничения высоты дерева Хаффмана. Заметим, что по построению любой нелистовой узел дерева Хаффмана имеет ровно два потомка. На каждой итерации нашего алгоритма будем уменьшать высоту дерева на 1. Итак, пусть L - максимальная длина кода (высота дерева) и требуется ограничить ее до $L' < L$. Пусть далее RN_i самый правый листовой узел на уровне i, а LN_i - самый левый.

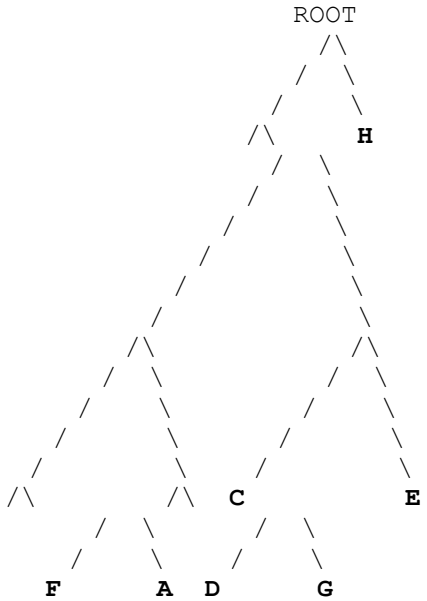
Начнем работу с уровня L. Переместим узел RN_L на место своего родителя. Т.к. узлы идут парами нам необходимо найти место и для соседнего с RN_L узла. Для этого найдем ближайший к L уровень j, содержащий листовые узлы, такой, что $j < (L-1)$. На месте LN_j сформируем нелистовой узел и присоединим к нему в качестве дочерних узел LN_j и оставшийся без пары узел с уровня L. Ко всем оставшимся парам узлов на уровне L применим такую же операцию. Ясно, что перераспределив таким образом узлы, мы

уменьшили высоту нашего дерева на 1. Теперь она равна (L-1). Если теперь $L^{/} < (L-1)$, то проделаем то же самое с уровнем (L-1) и т.д. до тех пор, пока требуемое ограничение не будет достигнуто.

Вернемся к нашему примеру, где L=5. Ограничим максимальную длину кода до $L^{/}=4$.

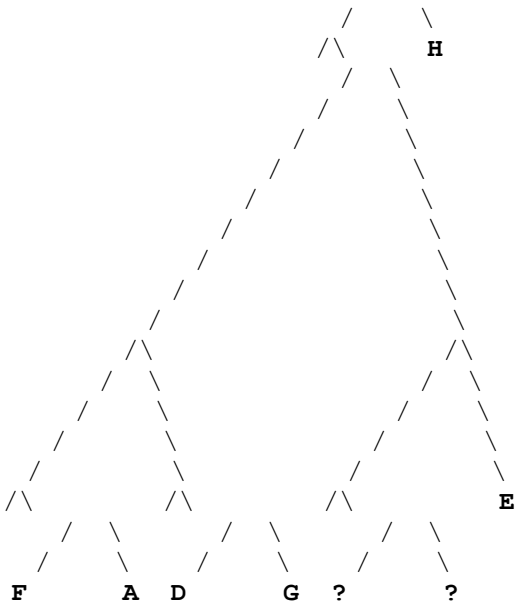


Видно, что в нашем случае RN_L=**F**, j=3, LN_j=**C**. Сначала переместим узел RN_L=**F** на место своего родителя.



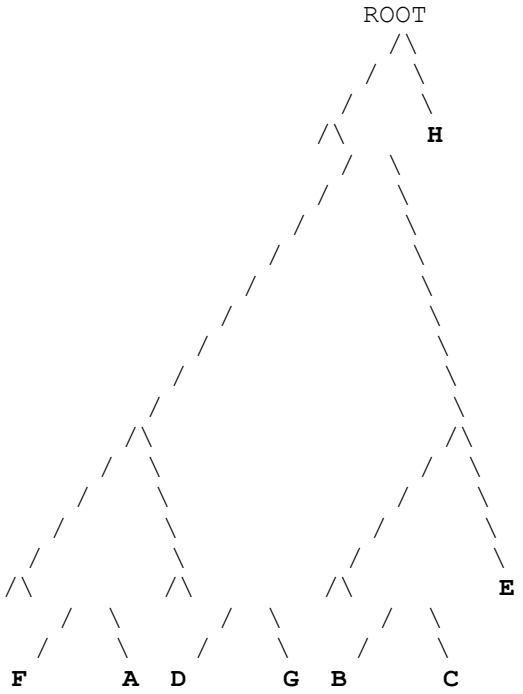
B (непарный узел)

Теперь на месте LN_j=**C** сформируем нелистовой узел.



B (непарный узел)
C (непарный узел)

Присоединим к сформированному узлу два непарных: **B** и **C**.



Таким образом, мы ограничили максимальную длину кода до 4. Ясно, что изменив длины кодов, мы немного потеряли в эффективности. Так сообщение S, закодированное при помощи такого кода, будет иметь размер 92 бита, т.е. на 3 бита больше по сравнению с минимально-избыточным кодом.

Ясно, что чем сильнее мы ограничим максимальную длину кода, тем менее эффективен будет код. Выясним насколько можно ограничивать максимальную длину кода. Очевидно что не короче $[\log_2 N]$ бит.

Вычисление канонических кодов

Как мы уже неоднократно отмечали, длин кодов достаточно для того чтобы сгенерировать сами коды. Покажем как это можно сделать. Предположим, что уже вычислили длины кодов и подсчитали сколько кодов каждой длины у нас есть. Пусть L - максимальная длина кода, а T_i - количество кодов длины i.

Вычислим S_i - начальное значение кода длины i, для всех i из [1..L]

$S_L = 0$ (всегда)
$S_{L-1} = (S_L + T_L) >> 1$
$S_{L-2} = (S_{L-1} + T_{L-1}) >> 1$
...
$S_1 = 1$ (всегда)

Для нашего примера L = 5, $T_{1\,..\,5} = \{1, 0, 2, 3, 2\}$.

$S_5 = 00000_{bin} = 0_{dec}$
$S_4 = (S_5{=}0 + T_5{=}2) >> 1 = (00010_{bin} >> 1) = 0001_{bin} = 1_{dec}$
$S_3 = (S_4{=}1 + T_4{=}3) >> 1 = (0100_{bin} >> 1) = 010_{bin} = 2_{dec}$
$S_2 = (S_3{=}2 + T_3{=}2) >> 1 = (100_{bin} >> 1) = 10_{bin} = 2_{dec}$
$S_1 = (S_2{=}2 + T_2{=}0) >> 1 = (10_{bin} >> 1) = 1_{bin} = 1_{dec}$

Видно, что S_5, S_4, S_3, S_1 - в точности коды символов **B**, **A**, **C**, **H**. Эти символы объединяет то, что все они стоят на первом месте, каждый на своем уровне. Другими словами, мы нашли начальное значение кода для каждой длины (или уровня).

Теперь присвоим коды остальным символам. Код первого символа на уровне i равен S_i, второго $S_i + 1$, третьего $S_i + 2$ и т.д.

Выпишем оставшиеся коды для нашего примера:

B = $S_5 = 00000_{bin}$ | **A** = $S_4 = 0001_{bin}$ | **C** = $S_3 = 010_{bin}$ | **H** = $S_1 = 1_{bin}$

F = $S_5 + 1 = 00001_{bin}$ | **D** = $S_4 + 1 = 0010_{bin}$ | **E** = $S_3 + 1 = 011_{bin}$

G = $S_4 + 2 = 0011_{bin}$

Видно, что мы получили точно такие же коды, как если бы явно построили каноническое дерево Хаффмана.

Передача кодового дерева

Для того чтобы закодированное сообщение удалось декодировать, декодеру необходимо иметь такое же кодовое дерево (в той или иной форме), какое использовалось при кодировании. Поэтому вместе с закодированными данными мы вынуждены сохранять соответствующее кодовое дерево. Ясно, что чем компактнее оно будет, тем лучше.

Решить эту задачу можно несколькими способами. Самое очевидное решение - сохранить дерево в явном виде (т.е. как упорядоченное множество узлов и указателей того или иного

вида). Это пожалуй самый расточительный и неэффективный способ. На практике он не используется.

Можно сохранить список частот символов (т.е. частотный словарь). С его помощью декодер без труда сможет реконструировать кодовое дерево. Хотя этот способ и менее расточителен чем предыдущий, он не является наилучшим.

Наконец, можно использовать одно из свойств канонических кодов. Как уже было отмечено ранее, канонические коды вполне определяются своими длинами. Другими словами, все что необходимо декодеру - это список длин кодов символов. Учитывая, что в среднем длину одного кода для N-символьного алфавита можно закодировать $[(\log_2(\log_2 N))]$ битами, получим очень эффективный алгоритм. На нем остановимся подробнее.

Предположим, что размер алфавита N=256, и мы сжимаем обыкновенный текстовый файл (ASCII). Скорее всего мы не встретим все N символов нашего алфавита в таком файле. Положим тогда длину кода отсутвующих символов равной нулю. В этом случае сохраняемый список длин кодов будет содержать достаточно большое число нулей (длин кодов отсутствующих символов) сгруппированных вместе. Каждую такую группу можно сжать при помощи так называемого группового кодирования - RLE (Run - Length - Encoding). Этот алгоритм чрезвычайно прост. Вместо последовательности из M одинаковых элементов идущих подряд, будем сохранять первый элемент этой последовательности и число его повторений, т.е. (M-1). Пример: RLE("AAAABBBCDDDDDDD")=A3 B2 C0 D6.

Более того, этот метод можно несколько расширить. Можно применить алгоритм RLE не только к группам нулевых длин, но и ко всем остальным. Такой способ передачи кодового дерева является общепринятым и применяется в большинстве современных реализаций.

Арифметическое кодирование

Метод Хаффмана является простым, но эффективным только в том случае, когда вероятности появления символов равны числам $1/2^n$, где n - любое целое положительное число. Это связано с тем, что код Хаффмана присваивает каждому символу алфавита код с целым числом бит. Вместе с тем в теории информации известно, что, например, при вероятности появления символа равной 0,4, ему в идеале следует поставить код длиной $-\log_2 0,4 \approx 1,32$ бит. Понятно, что при построении кодов Хаффмана нельзя задать длину кода в 1,32 бита, а только лишь в 1 или 2 бита, что приведет в результате к ухудшению сжатия данных. Арифметическое кодирование решает эту проблему путем присвоения кода всему, обычно, большому передаваемому файлу вместо кодирования отдельных символов.

Идею арифметического кодирования лучше всего рассмотреть на простом примере. Предположим, что необходимо закодировать три символа входного потока, для определенности – это строка SWISS_MISS с заданными частотами появления символов: S – 0,5, W – 0,1, I – 0,2, M – 0,1 и _ - 0,1. В арифметическом кодере каждый символ представляется интервалом в диапазоне чисел [0, 1) в соответствии с частотой его появления. В данном примере, для символов нашего алфавита получим следующие наборы интервалов:

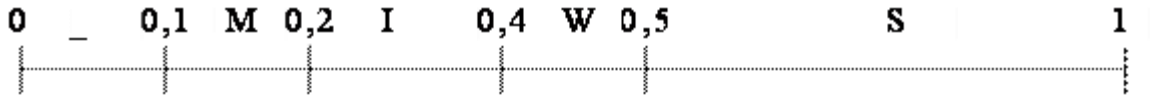


Рис. 2. Распределение интервалов представление символов

Процесс кодирования начинается со считывания первого символа входного потока и присвоения ему интервала из начального диапазона [0, 1). В данном случае для первого символа S получаем диапазон [0,5, 1). Затем, считывается второй символ – W, которому соответствует диапазон [0,4, 0,5). Но исходный диапазон [0, 1) уже сократился до [0,5, 1), поэтому символ W необходимо представить в этом новом диапазоне. Для этого достаточно вычислить новые нижнюю и верхнюю границы. Значение 0,4 будет соответствовать значению 0,7, а значение 0,5 – значению 0,75 (рис. 3).

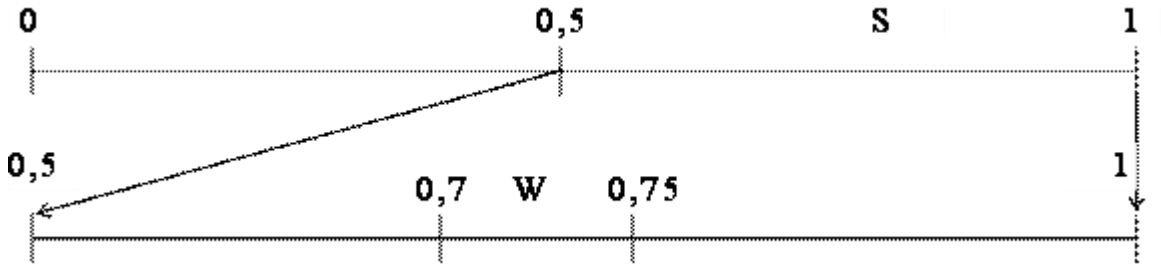


Рис. 3. Схема представления новых границ символа W

Данные границы можно вычислить по формулам:

NewHigh = OldLow + (OldHigh-OldLow)*HighRange(X),

NewLow = OldLow + (OldHigh-OldLow)*LowRange(X),

где OldLow – нижняя граница интервала, в котором представляется текущий символ; OldHigh – верхняя граница интервала; HighRange(X) – исходная верхняя граница кодируемого символа; LowRange(X) – исходная нижняя граница кодируемого символа. Применяя данные формулы к вычислению границ символа W, получаем:

OldLow = 0,5, OldHigh = 1,

HighRange(W) = 0,5, LowRange(W) = 0,4,

NewHigh = 0,5 + (1-0,5)*0,5 = 0,75,

NewLow = 0,5 + (1-0,5)*0,4 = 0,7.

Аналогичным образом выполняется кодирование символа I, для которого новые интервалы также можно вычислить по приведенной формуле:

OldLow = 0,7, OldHigh = 0,75,

HighRange(I) = 0,4, LowRange(I) = 0,2,

NewHigh = 0,7 + (0,75-0,7)*0,4 = 0,72,

NewLow = 0,7 + (0,75-0,7)*0,2 = 0,71.

Ниже, в табл. 1 представлены значения границ при кодировании строки SWISS_MISS.

Символ		Границы
S	L	0.0+(1.0-0.0)*0.5 = 0.5
	H	0.0+(1.0-0.0)*1.0 = 1.0
W	L	0.5+(1.0-0.5)*0.4=0.70
	H	0.5+(1.0-0.5)*0.5=0.75
I	L	0.7+(0.75-0.7)*0.2=0.71
	H	0.7+(0.75-0.7)*0.4=0.72
S	L	0.71+(0.72-0.71)*0.5=0.715
	H	0.71+(0.72-0.71)*1.0=0.72
S	L	0.715+(0.72-0.715)*0.5=0.7175
	H	0.715+(0.72-0.715)*1.0=0.72
_	L	0.7175+(0.72-0.7175)*0.0=0.7175
	H	0.7175+(0.72-0.7175)*0.1=0.71775
M	L	0.7175+(0.71775-0.7175)*0.1=0.717525
	H	0.7175+(0.71775-0.7175)*0.2=0.717550
I	L	0.717525+(0.717550-0.717525)*0.2=0.717530
	H	0.717525+(0.717550-0.717525)*0.4=0.717535
S	L	0.717530+(0.717535-0.717530)*0.5=0.7175325
	H	0.717530+(0.717535-0.717530)*1.0=0.717535
S	L	0.7175325+(0.717535-0.7175325)*0.5=0.71753375
	H	0.7175325+(0.717535-0.7175325)*1.0=0.717535

Конечный выходной код – это последнее значение переменной Low, равное 0.71753375, из которого следует взять лишь восемь цифр 71753375 для записи в файл.

Теперь рассмотрим возможность восстановления закодированной информации по восьми цифрам 71753375 и известным интервалам символов. Первая из восьми цифр – это 7, т.е. 0,7. Она принадлежит одному из заданных интервалов [0.5, 1), который соответствует символу S. Поэтому первый декодированный символ – это S. Теперь вернемся к рис. 3 и заметим, что второй символ был представлен в интервале символа S, т.е. [0.5, 1). Но для удобства декодирования его лучше представить в исходном интервале [0, 1). Для этого достаточно интервал [0.5, 1) увеличить до начального, т.е. умножить на два и границы сдвинуть на величину 0.5*2=1 (рис. 4).

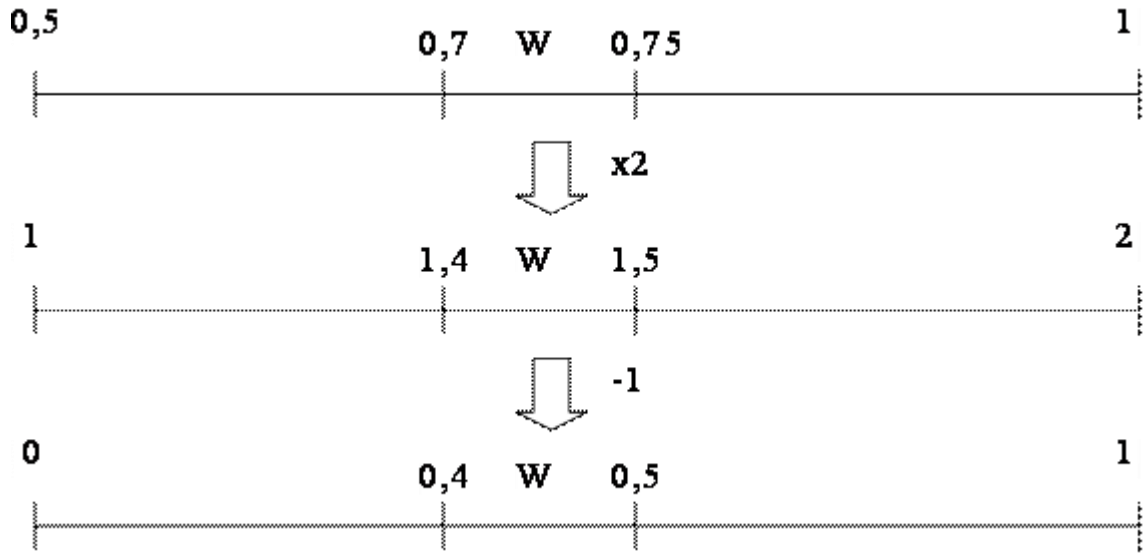


Рис. 4. Схема восстановления исходных интервалов символа W

Применяя данную схему к числу 0.71753375, получаем нижнюю границу следующего закодированного символа как будто он был начальным при кодировании:

$$0.71753375*2-1 = 0.4350675.$$

Полученное значение принадлежит диапазону [0.4, 0.5), который соответствует символу W. Затем, также полученное число 0.4350675 следует нормировать, что в общем случае выполняется по формуле:

$$Code = (Code-LowRange(X))/(HighRange(X)-LowRange(X)),$$

где Code – текущее значение кода. Например, пользуясь этой формулой применительно к коду 0.71753375, получаем значение

$$Code = (0.71753375-0.5)/(1-0.5)= 0.4350675,$$

которое в точности совпадает с предыдущей схемой вычисления. Аналогичным образом выполняется декодирование всех символов строки. В табл. 2 представлены коды, вычисляемые при декодировании символов. Здесь можно заметить, что процесс декодирования в данном случае можно остановить, если значение кода равно нулю.

Таблица 2. Вычисление кодов при декодировании

Символ	Code-Low	Область	
S	0.71753375 – 0.5	= 0.21753375	/ 0.5 = 0.4350675
W	0.4350675 – 0.4	= 0.0350675	/ 0.1 = 0.350675
I	0.350675 – 0.2	= 0.150675	/ 0.2 = 0.753375
S	0.753375 – 0.5	= 0.253375	/ 0.5 = 0.50675
S	0.50675 – 0.5	= 0.00675	/ 0.5 = 0.0135
_	0.0135 – 0	= 0.0135	/ 0.1 = 0.135
M	0.135 – 0.1	= 0.035	/ 0.1 = 0.35
I	0.35 – 0.2	= 0.15	/ 0.2 = 0.75
S	0.75 – 0.5	= 0.25	/ 0.5 = 0.5

S	0.5 – 0.5	= 0	/ 0.5 = 0

Однако это не всегда так. Бывают случаи, когда ноль содержит в себе код очередного символа, а не означает конец процедуры декодирования. Здесь возникает проблема завершения декодирования. Для этого используют специальный символ eof, говорящий о том, что он является последним и декодирование последовательности можно завершить. При этом частота этого символа очевидно должна быть маленькой по сравнению с частотой символов алфавита последовательности, например, [0,999999 1).

Описанный выше процесс кодирования невозможно реализовать на практике, т.к. в нем предполагается, что в переменных Low и High хранятся числа с неограниченной точностью. По существу, результат кодирования – это вещественное число с очень большой точностью. Например, файл объемом 1Мб будет сжиматься, скажем, до 500 КБ, в котором будет записано одно число. Арифметические операции с такими числами реализовать сложно и долго. Поэтому любая практическая реализация арифметического кодера должна основываться на операциях с целыми числами, которые не должны быть слишком длинными. Рассмотрим такую реализацию, в которой переменные Low и High будут целыми числами длиной 16 или 32 бита. Эти переменные будут хранить верхние и нижние концы текущего подинтервала, но мы не будем им позволять неограниченно расти. Анализ табл. 1 показывает, что как только самые левые цифры переменных Low и High становятся одинаковыми, они уже не меняются в дальнейшем. Следовательно, эти цифры можно выдвинуть за скобки и работать с оставшейся дробной частью. После сдвига цифр мы будем справа дописывать 0 в переменную Low, а в переменную High – цифру 9. Для того, чтобы лучше понять весь процесс, можно представлять себе эти переменные как левый конец бесконечно длинного числа. Число Low имеет вид xxxx00… а число High = yyyy99…

Проблема состоит в том, что переменная High в начале должна равняться 1, однако интерпретируем Low и High как десятичные дроби меньшие 1. Решение заключается в присвоении переменной High значения 9999…, которое соответствует бесконечной дроби 0,9999…, равной 1.

Как это все работает? Закодируем этим способом ту же строку SWISS_MISS. Первая буква S определена диапазоном [0,5 1) и формально границы равны

L=0.0+(1.0-0.0)*0.5=0.5

H=0.0+(1.0-0.0)*1.0=1.0

но в нашем случае данные формулы следует преобразовать, чтобы переменные Low и High были целыми. Поэтому запишем их в таком виде:

Low=0+(10000-0)*0.5=5000

High=0+(10000-0)*1.0=10000

но по условию граница High является открытой, т.е. не включает число 10000, поэтому от конечного значения нужно отнять 1:

High=0+(10000-0)*1.0-1=9999

Таким образом, получили формулы для вычисления целочисленных границ Low и High, и процесс кодирования будет выглядеть так (табл. 3).

Табл. 3. Кодирование сообщения сдвигами

1		2			4	5
S	L=	0+(10000-0)*0.5	=	5000		5000
	H=	0+(10000-0)*1.0-1	=	9999		9999
W	L=	5000+(10000-5000)*0.4	=	7000	7	0000
	H=	5000+(10000-5000)*0.5	=	7499		4999
I	L=	0+(5000-0)*0.2	=	1000	1	0000
	H=	0+(5000-0)*0.4-1	=	1999		9999
S	L=	0+(10000-0)*0.5	=	5000		5000
	H=	0+(10000-0)*1.0-1	=	9999		9999
S	L=	5000+(10000-5000)*0.5	=	7500		7500
	H=	5000+(10000-5000)*1.0-1	=	9999		9999
_	L=	7500+(10000-7500)*0.0	=	7500	7	5000
	H=	7500+(10000-7500)*0.1-1	=	7749		7499
M	L=	5000+(7500-5000)*0.1	=	5250	5	2500
	H=	5000+(7500-5000)*0.2	=	5499		4999
I	L=	2500+(5000-2500)*0.2	=	3000	3	0000
	H=	2500+(5000-2500)*0.4-1	=	3499		4999
S	L=	0+(5000-0)*0.5	=	2500		2500
	H=	0+(5000-0)*1.0-1	=	4999		4999
S	L=	2500+(5000-2500)*0.5	=	3750	3750	3750
	H=	2500+(5000-2500)*1.0-1	=	4999		4999

На последнем шаге операции кодирования записываются все 4 цифры и полученная выходная последовательность имеет вид: 717533750.

Декодер работает в обратном порядке. В начале переменным Low и High присваиваются значения 0000 и 9999 соответственно, а переменной Code значение 7175. На основе этой информации требуется определить первый закодированный символ. Для этого число переменной Code нужно корректно представить в интервале от 0 до 1. В самом начале интервал такой и есть, поэтому частота символа, соответствующая значению Code будет равна

index = 7175/10000=0,7175.

Эта частота попадает в диапазон [0,5 1) и соответствует символу S. Теперь границы Low и High пересчитываются, так как это делалось в кодере и принимают значения 5000 и 9999

соответственно. Так как значащие цифры этих переменных отличаются, то переменная Code остается прежней и величина

index = (7175-5000)/(10000-5000)=0,4350.

Это значение попадает в диапазон [0,4 0,5) и соответствует символу W. После этого величины Low и High принимают значения 7000 и 7499 и после отбрасывания значащей цифры переходят в 0000 и 4999 соответственно, а переменная Code преобразуется в 1753. Таким образом раскодируется вся последовательность.

На практике обычно значение index принимает целочисленные значения, которые вычисляются по формуле

index = ((Code-Low)*10-1)/(High-Low+1)

и округляют до ближайшего целого.

Может показаться, что приведенный выше пример не производит никакого сжатия. Для того чтобы выяснить степень сжатия результаты кодирования нужно перевести в двоичную форму. Так как из конечного интервала

[0.71753375 0.717535)

можно выбрать любое число, выберем наименьшее для хранения – это 717534, которому соответствует битовое представление 10101111001011011110 и составляет 20 бит. И строка из 10 символов сжимается в 20 бит. Хорошее ли это сжатие? Для этого нужно найти энтропию кодируемой последовательности и она будет равна

$$H(X) = -0,5\log_2 0,5 - 0,2\log_2 0,2 - 0,1\log_2 0,1 - 0,1\log_2 0,1 - 0,1\log_2 0,1 \approx 1,96$$ бит/сим

и составит величину

$$I = H(X) \cdot 10 \approx 19,6$$ бит,

что в целых значениях равно 20 бит. Следовательно, полученный код достиг минимально возможного значения и является оптимальным. В общем случае можно показать, что при достаточно большой последовательности арифметический кодер всегда приводит к оптимальным результатам сжатия, т.е. является наилучшим среди всех энтропийных кодеров.