Сжатие текстов

Дискретный анализ 2012/13

Андрей Калинин, Татьяна Романова

27 апреля 2013 г.

Литература

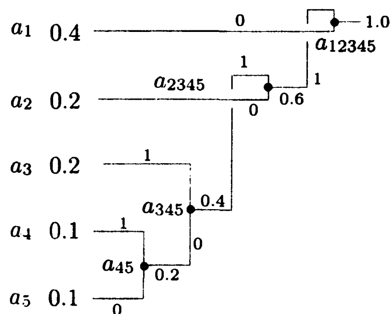
- Witten, Moffat, Bell, Managing gigabytes: compressing and indexing documents and images.
- M. J. Attalah, Algorithms and Theory of Computation Handbook, 12.3 Dynamic Huffman Coding.
- Ватолин Д., Ратушняк А., Смирнов М., Юкин В., Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео.
- Сэлмон Д., Сжатие изображений и звука, М.:Техносфера, 2006.

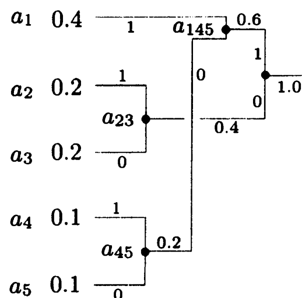
Раздел

Коды Хаффмана
Адаптивное кодирование
Канонические коды Хаффмана

Неоднозначность алгоритма Хаффмана

- Правое и левое поддерево может кодироваться 1-0 или 0-1.
- В случае, если есть несколько символов с одинаковыми частотами, нет правила по их однозначному выбору.





Ограниченность кодирования Хаффмана

- Данные с равновероятными символвами не сжимаются.
- Однако, не все строки, в которых частоты символов равновероятны — случайны.
- Напрмер, последовательность

$$a_1 a_1 \ldots a_1 a_2 a_2 \ldots a_2 a_3 a_3 \ldots$$

  где каждый символ встречается длинными сериями одинаковой длины — может быть сжата методом RLE (run-length encoding), но не методом Хаффмана.
- Для двухсимвольного алфавита код Хаффмана будет $\langle 0, 1 \rangle$ вне зависимости от частот символов.
- Декодер должен получить от кодера статистику использования символов (что увеличивает объём) или кодер с декодером должны использовать статический набор частот (что ухудшает компрессию).

Раздел

Коды Хаффмана
Адаптивное кодирование
Канонические коды Хаффмана

Основная идея

- Кодер и декодер начинают с пустого дерева, которое модифицируется по ходу работы.
- Первый входной символ записываеся в выходной файл в несжатой форме и помещается в дерево с частотой 1, ему присваивается код.
- В следующий раз он будет закодирован с использованием кода, соответствующего его расположению в дереве, а соответствующий счётчик увеличен на 1.
- Аналогичная операция проделывается со следующий символом.
- При добавлении символа в дерево или изменении частот может потребоваться перестроить дерево.

Специальный символ ESC

- Нужно отличать закодированные символы от их первого появления в прямом коде.
- Для этого вводится дополнительный символ, ESC (Escape), который предваряет первое появление символа.
- Этот символ всегда есть в дереве и имеет частоту 1, при появлении новых символов код escape-символа удлинняется.

Модификация дерева Хаффмана

Любое построенное стандартным алгоритмом дерево Хаффмана с n листьями можно представить в виде последовательности $x_0, x_1, ..., x_{2n-2}$ такой, что

1. Веса (частоты), записанные в узлах составят невозрастающую последовательность:

$$weight(x_0) \geq weight(x_1) \geq ... \geq weight(x_{2n-2})$$

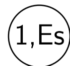
2. Для любого элемента i $(0 \leq i \leq n-2)$ элементы на позициях $2i+1$ и $2i+2$ являются его сыновьями.

Кодирование. Описание.

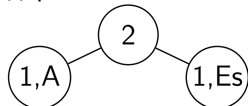
- Дерево Хаффмана хранится в массиве (аналогично куче из пирамидальной сортировки).
- Изначально в дереве находится только ESC-символ. Он всегда будет самым правым элементом массива.
- Если нужно добавить новый узел, на место ESC-символа помещается новый родитель, новый узел становится левым сыном, ESC — правым.
- У узла, соответствующего считанному символу, счетчик частоты увеличивается на 1.
- Также на 1 нужно увеличить счетчики всех предков. При этом частоты узлов в массиве могут перестать быть невозрастающей последовательностью. Тогда нужно "переподвесить"текущий нарушающий порядок узел (и его поддерево) на место узла, хранящегося в самом левом элементе массива с меньшей частотой.

Кодирование. Пример.

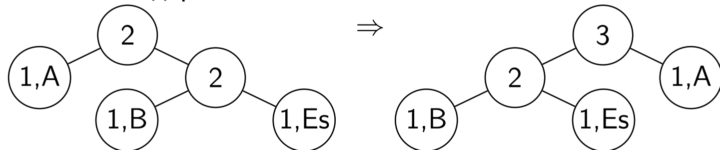
1. Кодируем строку ABBCD. Начальное дерево:



2. Считываем A, выводим 0100 0001 (код 'A'), обновляем дерево:

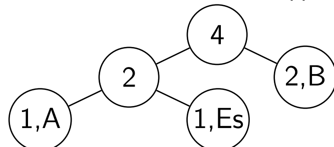


3. Считываем B, выводим 1 (Esc) и 0100 0010 (код 'B'), обновляем дерево:

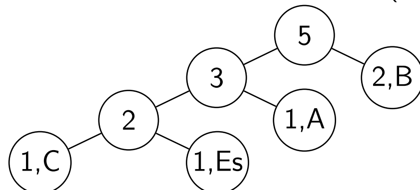


Кодирование. Пример.

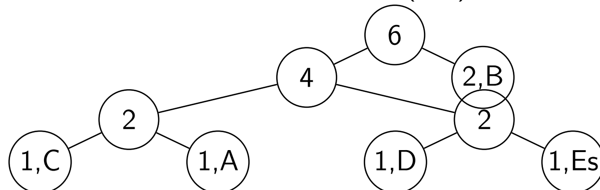
4. Считываем B, выводим 00, обновляем дерево:



5. Считываем C, выводим 01 (Esc) и 0100 0011 (код 'C'):



6. Считываем D, выводим 001 (Esc) и 0100 0100 (код 'D'):



Кодирование. Реализация.

DH-Init

1 $root \leftarrow 0$
2 $child(root) \leftarrow UNDEFINED$
3 $parend(root) \leftarrow UNDEFINED$
4 $weight(root) \leftarrow 1$
5 **for** each letter a in $Alphabet$
6 $\quad leaf[a] \leftarrow UNDEFINED$
7 $leaf[ESC] \leftarrow root$

Кодирование. Реализация.

```
DH-Encoding
1  DH-Init
2  while not eof(fin) and next symbol is a
3      DH-Encode-Symbol(a, fout)
4      DH-Update(a)
```

Кодирование. Реализация.

При поиске кода мы поднимаемся по дереву, поэтому нужен стек. Заметим, что все левые сыновья имеют нечетные номера, правые - четные.

DH-Encode-Symbol($a, fout$)

1. $S \leftarrow$ empty stack
2. $n \leftarrow leaf[a]$
3. **if** $n = UNDEFINED$
4. $n \leftarrow leaf[ESC]$
5. **while** $n! = root$
6.   **if** n нечетное
7.   Push($S, 1$)
8.   **else** Push($S, 0$)
9. $n \leftarrow parent(n)$
10. Send($S, fout$)
11. **if** $leaf[a] = UNDEFINED$
12.   пишем в $fout$ 8-битный код
13.   DH-Add-Node(a)

Кодирование. Реализация.

DH-ADD-NODE(a)

1 $leaf[ESC]$ становится внутренним узлом с весом 1,
2 его левый сын $leaf[a]$ с весом 0,
3 правый - $leaf[ESC]$ с весом 1.

Кодирование. Реализация.

```
DH-Update(a)
1   n ← leaf[a]
2   while n! = root
3       weight(n) = weight(n) + 1
4       Поиск самого левого соседа, с которым можно поменяться
5       Иллюстрация: 6-й шаг примера
6       m ← n
7       while weight(m − 1) < weight(n)
8           m ← m − 1
9       DH-Swap-Nodes(m, n)
10      n ← parent(m)
11  weight(root) = weight(root) + 1
```

Раздел

Коды Хаффмана
Адаптивное кодирование
Канонические коды Хаффмана

Неоднозначность кодов Хаффмана

Буква	Частота	Код 1	Код 2	Код 3
a	10	000	111	000
b	11	001	110	001
c	12	100	011	010
d	13	101	010	011
e	22	01	10	10
f	23	11	00	11

Пример канонического кода

100	17	00000000000000000
101	17	00000000000000001
102	17	00000000000000010
...	...	...
zepyhyr	17	00001101010101000
zigzag	17	00001101010101001
11th	16	0000110101010101
120	16	0000110101010110
...	...	...
you	7	1100001
I	6	110001
in	6	110010
was	6	110011
a	5	11010
and	5	11011
of	5	11100
to	5	11101

Канонический код Хаффмана

- Длины кодов для символов (слов) рассчитаны классическим способом.
- Коды назначются по порядку.
- Все слова в группе кодов одной длины сортируются в лексикографическом порядке.
- Первый код в группе равен коду последнего слова в предыдущей группе без последнего бита и увеличенный на 1.
- Имеет смысл для алфавитов больших размеров (слов)
- Позволяет не хранить дерево в явном виде.

Алгоритм построения канонического кода

```
1   for l ← 1 to maxlength
2       numl[l] ← 0
3   for i ← 1 to n
4       numl[l_i] ← numl[l_i] + 1
5   firstcode[maxlength] ← 0
6   for l ← maxlength − 1 downto 1
7       firstcode[l] ← (firstcode[l + 1] + numl[l + 1])/2
8   for l ← 1 to maxlength
9       nextcode[l] ← firstcode[l]
10  for i ← 1 to n
11      codeword[i] ← nextcode[l_i]
12      symbol[l_i, nextcode[l_i] − firstcode[l_i]] ← i
13      nextcode[l_i] ← nextcode[l_i] + 1
```

Построение канонического кода

i	l_i	$codeword[i]$	биты	1	2	3	4	5
1	2	1	01	0	1			
2	5	0	00000	0	0	0	0	0
3	5	1	00001	0	0	0	0	1
4	3	1	001	0	0	1		
5	2	2	10	1	2			
6	5	2	00010	0	0	0	1	2
7	5	3	00011	0	0	0	1	3
8	2	3	11	1	3			
			$numl[l]$	0	3	1	0	4
			$firstcode[l]$	2	1	1	2	0

Деокдирование канонического кода Хаффмана

```
1  v ← Next-Input-Bit()
2  l ← 1
3  while v < firstcode[l]
4        v ← 2v + Next-Input-Bit()
5        l ← l + 1
6  return symbol[l, v − firstcode[l]]
```

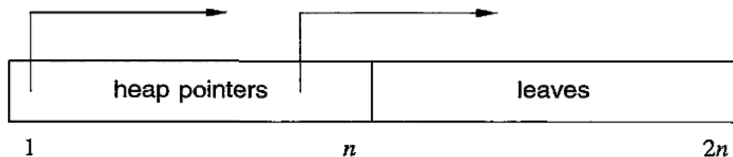
Вычисление длин кодов Хаффмана

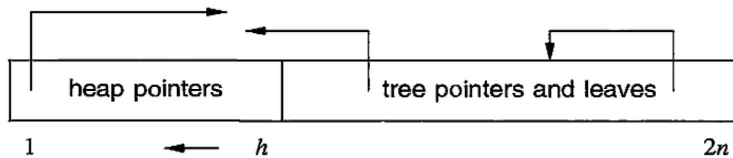
1. Создать массив A из $2n$ элементов, прочитать в нижнюю половину частоты, в верхнюю половину поместить «указатели» на эти частоты.
2. Построить из верхней половины массива A «пирамиду» (по значениям, соответствующим указателям).
3. Пройти по пирамиде, выбирая последовательно два минимальных элемента и помещая в пирамиду элемент с суммой значений выбранных. Заменять значения в удалённых элементах на номер родителя.
4. Восстановить по пирамиде из «указателей» на родителя длины кодов для каждого символа.

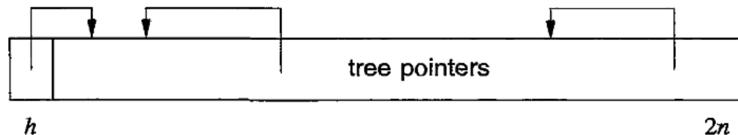
Алгоритм вычисления длин кодов

```
1   // Создать A из 2n элементов.
2   for i ← 1 to n
3         A[n + i] ← c_i, A[i] ← n + i
4   h ← n
    // Построить «пирамиду» из A[1 . . . h],
    // В A[1] хранится m_1 = arg min{A[n + 1] . . . 2n}
5   while h > 1
6         m_1 ← A[1], A[1] ← A[h], h ← h − 1
7         // Просеять «пирамиду» A[1 . . . h] от A[1]
8         m_2 ← A[1]
9         A[h + 1] ← A[m_1] + A[m_2], A[1] ← h + 1
10        A[m_1] ← A[m_2] ← h + 1
11        // Просеять пирамиду A[1 . . . h] от A[1]
12  A[2] ← 0
13  for i ← 3 to 2n
14        A[i] ← A[A[i]] + 1
```

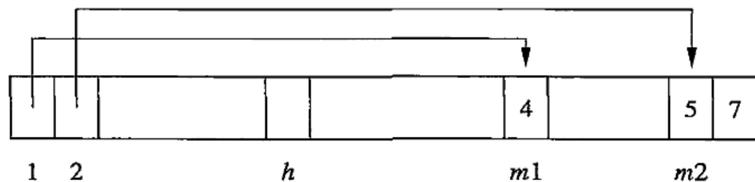
Этапы работы алгоритма

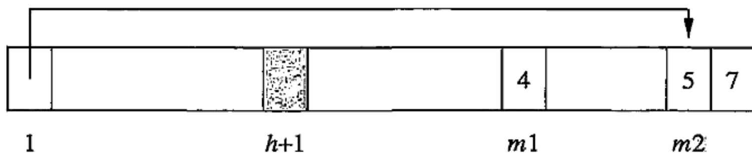




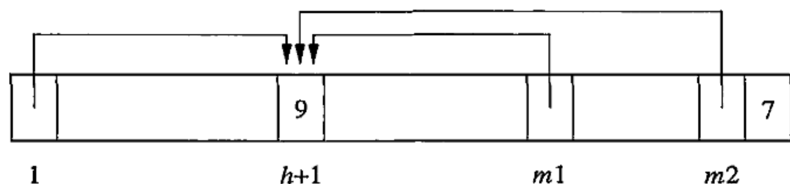


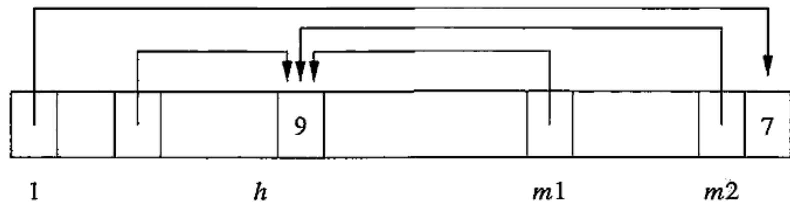
Извлечение минимальных частот





Сохранение нового узла дерева





Вычисление длин кодов