Тема 6. Некоторые алгоритмы

Лекция 21. Сжатие данных, продолжение

Адаптивнпе кодирование Хаффмена.

Очевидным недостатком метода Хаффмена является необходимость передавать декодировщику таблицу частот символов вместе с кодирующей последовательностью, что уменшает суммарный коэффициент сжатия. Существует адаптивный вариант алгоритма Хаффмена, который не требует передачи дерева (или частотной таблицы) вместе с закодированными данными. Идея этого алгоритма заключается в адаптивном перестроении дерева с учетом поступивших символов при кодировании и соответствующем преобразовании дерева при декодировании.

Пусть мы имеем некоторое дерево Хаффмена. Пронумеруем вершины этого дерева "слева-направо, снизу-вверх", т.е. нумерация начинается с элементов самого нижнего уровня дерева, потом переходит на предыдущий уровень, и т.д. Корень дерева получает максимальный номер.

Будем называть дерево упорядоченным, если веса вершин не убывают в порядке построенной нумерации.

Адаптивное перестроение дерева.

Построим начальное дерево Хаффмена, считая, что все символы имеют вес 1. Теперь при появлении нового символа мы должны перестроить дерево так, чтобы сохранить свойство упорядоченности. Во-первых, надо увеличить вес соответствующей концевой вершины и всех родителей по ветви, ведущей к корню. Если после этого дерево остается упорядоченным, то ничего больше делать не надо. Если же вес какой-либо вершины A становится больше веса правого соседа (по нумерации), то среди правых соседей (по нумерации) ищется последний, вес которого меньше веса A, и эта вершина (вместе со своими потомками) обменивается с A. Затем корректируются веса по ветвям, ведущим к корню от переставленных вершин. Указанный процесс продолжается, пока дерево не станет упорядоченным.

Кодирование и декодирование по адаптивному варианту алгоритма происходит единообразно:

1. Дерево инициализируется равномерным распределение символов (все символы имеют количество повторений 1).

2. Получить очередной символ (или код) и закодировать (раскодировать) его в соответствии с текущим состоянием дерева Хаффмена.

3. Добавить единицу к весу вершины, соответствующей только что полученному символу, и перестроить дерево на его основе для сохранения свойства упорядоченности.

4. Если есть еще символы (коды) во входном потоке, то перейти к пункту 2, иначе закончить работу.

Естественно, адаптивный алгоритм не дает оптимальной кодовой последовательности. Однако помимо отказа от передачи таблицы частот от он обладает свойством настройки на текущее распределение частот. Рассмотрим следующий искуственный, но показательный пример. Пусть входная последовательность содержит достаточно длинные цепочки одинаковых байт, при этом в целом распределение частот является равномерным. Для такой последовательности обычный алгоритм Хаффмена не предложит ничего лучшего чем обычные восьмибитные коды байтов. Однако в адаптивном

варианте он настроится на локальные частотные характеристики и назначит повторяющимся символам более короткие коды, что может дать некоторый выигрыш для файла в целом.

Арифметическое кодирование

В основе алгоритма арифметического кодирования лежит очень простая и красивая идея. Входной поток байтов рассматривается как запись некоторого числа, в которой входные символы играют роль "цифр". Алгоритм переводит эту запись в двоичное представление следующим образом.

Пусть мы имеем входную последовательность длиной n байт. Обозначим частоту появления байта со значением k в этой входной последовательности через α_k, $k = 0, \ldots, 255$. Здесь под частотой мы понимаем отношение количества появлений данного фиксированного байта к количеству всех байт. Таким образом, $0 \leq \alpha_k \leq 1$, $\sum_{k=0}^{255} \alpha_k = 1$, и количество появлений данного байта есть $p_k = \alpha_k n$.

Разобъем отрезок $s_0 = [0, 1]$ на подотрезки, равные по длине α_k и сопоставим эти отрезки соответствующим значениям байтов входной последовательности.

Возьмем первый байт входного потока и рассмотрим соответствующий ему отрезок (обозначим его s_1). Разобъем отрезок s_1 пропорционально частотам α_k (аналогично тому, как был разбит отрезок s_0). Теперь возьмем следующий входной байт и рассмотрим соответствующий ему отрезок $s_2 \subset s_1$. Далее процесс продолжается аналогично (т.е. разбиваем текущий выбранный отрезок пропорционально частотам и выбираем в нем следующий отрезок по новому поступившему байту входной последовательности). В результате мы получим последовательность вложенных друг в друга отрезков $s_0 \supset s_1 \supset s_2 \supset \ldots \supset s_n$. Результатом кодирования является любое число, принадлежащее последнему выбранному отрезку s_n. Декодирование производится достаточно просто. Для этого надо взять кодирующее число и проверить какому из отрезков разбиения оно принадлежит. Эта проверка даст первый байт исходного набора данных. Затем можно определить какому второму отрезку принадлежит число и т.д. Для устранения неоднозначности кодирования и декодирования следует исключить из проверки одну из границ отрезков (например, правую). Очевидно также, что для возможности декодирования следует знать таблицу частот и общее количество байтов в исходной последовательности (иначе неясно в какой момент прекращать процесс декодирования).

Опишем алгоритм построения кодирующей дроби в виде псевдокода. Пусть элементы массива `a[i]` задают точки границ отрезков частотного разбиения для байта, т.е. $\alpha_i =$ `a[i+1]` $-$ `a[i]`, а функция `NextByte()` читает и возвращает очередной байт входной последовательности, либо значение `EOI` при окончании набора данных (EOI — End Of Information). Описанный выше алгоритм выглядит так:

```
left = 0;
right = 1;
while ( (k=NextByte()) != EOI ) {
   d = right - left;
   right = left + d*a[k+1];
   left  = left + d*a[k];
}
```

По окончании этого цикла в качестве кодирующей дроби можно взять любое число `x`, удовлетворяющее условию `left` $\le$ `x` $<$ `right`.

Декодирование производится обращением описанного выше процесса.

```
x = <кодирующая дробь>
n = <общее число символов в исходном файле>
while (n--) {
  k = Interval (x);     // найти номер интервала, содержащего x
  Output(Symbol(k));    // вывести k-й символ
  d = a[k+1]-a[k]       // длина интервала для k-го символа
  x = (x - a[k])/d;     // преобразуем дробь для следующего символа
}
```

Алгоритм арифметического кодирования может привести к поразительным результатам. Например, если при кодировании окажется, что окончательный отрезок содержит точку $0.5_{10} = 0.1_2$, то мы получим выходной код длиной в 1 бит! (не считая, конечно, накладных расходов в виде таблицы частот и количества входных байтов). Однако, в общем случае на это нельзя полагаться, и возникает резонный вопрос: а почему этот алгоритм в принципе способен сжимать информационно избыточные наборы данных?

Теорема. *Для заданной входной последовательности длины N символов метод арифметического кодирования дает код, длина которого не больше $L \approx NE$, где E — информационная энтропия таблицы частот символов входной последовательности.*

(информационная энтропия — количество информации по Шеннону $= \sum \alpha_i \log \alpha_i^{-1}$).

Доказательство. Пусть α_k — длины отрезков исходного разбиения и $p_k = \alpha_k N$ — количество появлений байта со значением k. Нетрудно увидеть, что длина последнего кодирущего отрезка равна

$$l = \prod_{k:\alpha_k \neq 0} \alpha_k^{p_k}.$$

Для построения кода нам нужно найти число, принадлежащее такому отрезку и имеющее по возможности наименьшее количество бит в своем двоичном представлении. Для того, чтобы в произвольном отрезке длины l нашлась двоичная дробь, содержащая L значащих разрядов, достаточно выполнения неравенства $2^{-L} \le l$. Отсюда получаем $L \ge -\log_2 l$. Так как нас интересует дробь с наименьшим количеством значащих разрядов, то принимаем $L \approx -\log_2 l$ за длину выходного кода алгоритма, т.е.

$$L \approx -\log_2 \prod_{k:\alpha_k \neq 0} \alpha_k^{p_k} = \sum_{k:\alpha_k \neq 0} N\alpha_k \log_2 \alpha_k = NE.$$

Следствие. *Пусть входная последовательность, состоящая из N байт (или $8N$ бит), имеет неравномерное распределение частот значений отдельных байтов. Тогда кодирующая дробь метода арифметического кодирования может быть представлена менее чем $8n$ битами.*

Лемма. *Пусть функция $f(x)$ является выпуклой вниз на некотором интервале (a, b). Тогда для любого набора $x_1, \ldots, x_m \in (a, b)$ выполнено соотношение*

$$\frac{1}{m}\sum_{k=1}^{m} f(x_k) \ge f\Big(\frac{1}{m}\sum_{k=1}^{m} x_k\Big),$$

причем равенство достигается только при $x_1 = x_2 = \ldots = x_m$.

Доказательство легко проводится индукцией по m с учетом хорошо известного неравенства для выпуклых вниз функций: $\alpha f(x_1) + (1-\alpha) f(x_2) > f(\alpha x_1 + (1-\alpha) x_2)$, $0 < \alpha < 1$.

Доказательство следствия. Для байтовой последовательности таблица частот $\{\alpha_0, \ldots, \alpha_{255}\}$.

$$E = -256N \frac{1}{256} \sum_{k:\alpha_k \neq 0} \alpha_k \log_2 \alpha_k \leq -N \cdot 256 \left(\frac{\alpha_0 + \ldots + \alpha_{255}}{256} \right) \log_2 \left(\frac{\alpha_0 + \ldots + \alpha_{255}}{256} \right) =$$

$$= -N \log_2 \frac{1}{256} = 8N.$$

Так как не все α_k одинаковые, то имеем знак $<$ в неравенстве.

Алгоритм арифметического кодирования имеет и адаптивный вариант, который по принципам построения аналогичен адаптивному методу Хаффмена. Этот алгоритм можно описать следующим образом (на примере байтовой последовательности)

1. Строится таблица частот дла равномерного распределения (т.е. все отрезки разбиения имеют длину $1/256$).

2. Очередной отрезок разбиения выбирается по очередному входному символу на основании текущей таблицы частот.

3. Таблица частот пересчитывается с учетом появления только что обработанного входного символа.

4. Продолжаем обработку по пунктам 2–4, пока есть входные символы.

5. В окончательном отрезке разбиения выбираем кодирующую дробь.

Адаптивное декодирование строится аналогично.

1. Строится таблица частот дла равномерного распределения (т.е. все отрезки разбиения имеют длину $1/256$).

2. Ищется очередной отрезок, содержащий кодирующую дробь, на основании текущей таблицы частот. Выходной символ определяется по найденному отрезку.

3. Таблица частот пересчитывается с учетом появления только что полученного выходного символа.

4. Продолжаем обработку по пунктам 2–4, пока есть не исчерпается общее количество символов.

Лекция 22. Алгоритм LZW

В этом разделе описывается простейший вариант Lempel–Ziv–Welch алгоритма (LZW), который в различных модификациях используется в программах-архиваторах.

Суть алгоритма состоит в обнаружении во входном потоке повторяющихся цепочек байтов, составлении таблицы таких обнаруженных цепочек и выдаче в выходной поток кодов (номеров строк таблицы), соответствующих обнаруженной цепочке. Таким образом, если исходный файл имеет много повторяющихся последовательностей байтов, то каждая такая последовательность будет заменена на ее порядковый номер в таблице. Важной особенностью алгоритма является то, что распаковщик, анализируя сжатые данные в процессе работы, может построить копию исходной таблицы и, следовательно, ее не надо передавать вместе с сжатыми данными.

Опишем схему алгоритма, используя С-подобный псевдокод.

Кодирование. Создается таблица, способная вместить достаточно большое количество строк. Первые 256 строк этой таблицы инициализируются всеми возможными односимвольными цепочками (т.е. соответствуют символам с кодами от 0 до 255). Дальнейший алгоритм выглядит так:

```
s = <пустая цепочка>;
while (есть символы во входном потоке) {
   c = NextSym();           // взять очередной символ
   if ( InTable (s+c) ) {   // цепочка s+c уже есть в таблице
      s = s+c;              // удлиняем цепочку прочитанным символом
   } else {                 // цепочки s+c нет в таблице
      OutCode (s);          // вывод кода, соответствующего цепочке s
      AddString (s+c);      // добавляем новую цепочку s+c в таблицу
      s = c;                // готовимся к обнаружению следующей цепочки
   }
}
OutCode (s);                // вывод кода для оставшейся цепочки s
```

Здесь знак "+" обозначает конкатенацию цепочек. Алгоритм кодирования каждый раз пытается найти в таблице наиболее длинную цепочку, соответствующую читаемой последовательности символов. Если это в какой-то момент не удается, то накопленная к этому времени цепочка заносится в таблицу. В какой-то момент может наступить переполнение таблицы. В этом случае кодировщик выводит в выходной поток специальный код очистки и таблица цепочек инициализируется заново. Обычно в реальных алгоритмах применяется таблица с 4096 входами для цепочек, при этом число 256 является кодом очистки (обозначим его CLC), а число 257 — кодом конца информации (EOI), эти строки таблицы не используется для цепочек. Заметим, что в этом случае для каждого выходного кода достаточно 12 бит. Поэтому при выводе целесообразно упаковывать каждые два кода в три байта. Для упрощения логики декодера обычно первым кодом сжатых данных является CLC, что сразу вызывает инициализацию таблицы цепочек.

Декодирование. Распаковка сжатых данных основывается на построении идентичной таблицы цепочек. Инициализация таблицы выполняется так же как и при кодировании.

```
while ( (k=NextCode()) != EOI ) {  // пока не конец информации
   if ( k == CLC ) {               // k есть код очистки
      InitTable();                 // заново инициализируем таблицу
      k = NextCode();              // читаем следующий код
      if ( k == EOI ) break;
      OutString(k);                // выводим цепочку для кода k
      old = k;                     // запоминаем текущий код
   } else {
      if ( InTable(k) )  {    // в таблице есть строка для кода k
         OutString(k);        // выводим цепочку для кода k
         AddString( String(old)+Char(k) );  // формируем и добавляем новую цепочку
         old = k;
      } else {                // в таблице нет строки для кода k
```

```
            s = String(old)+Char(old);  // формируем цепочку
            OutString(s);               // выводим цепочку
            AddString(s);               // и добавляем ее в таблицу
            old = k;
        }
    }
}
```

В этом алгоритме функция `String(i)` возвращает строку из таблицы, соответствующую коду `i`, а функция `Char(i)` возвращает только первый символ такой строки. Заметим, что функция `NextCode()` должна извлекать 12-битные коды из входного потока байтов.

Степень сжатия этого алгоритма оценить непросто. Но здесь применим все тот же вывод: сжатие будет хорошим, если входной поток байтов обладает свойствами повторяемости отдельных фрагментов. С другой стороны, можно предложить такую последовательность входных байтов, что почти все они будут кодироваться односимвольными цепочками и алгоритм даст даже некоторый проигрыш в размере результата.

Заметим, что кодировщик и декодировщик алгоритма LZW заполяют таблицу цепочек по одинаковым правилам. Если кодировщик обработал m входных байтов, а декодировщик раскодировал m выходных байтов, то они имеют одинаковые состояния таблиц цепочек. Этот факт позволяет еще немного улучшить степень сжатия алгоритма.

Действительно, пока в таблице менее 512 цепочек, кодировщих может выдавать на выход 9-битные коды. Когда число зарегистрированных цепочек превысит 511 но еще будет меньше 1024, кодировщик может перейти на 10-битные коды. В диапазоне от 1024 до 2048 декодировщик выдает 11-битные коды, и только потом — 12-битные. Декодировщик сначала извлекает из входного потока 9-битные коды и аналогично переходит к более длинным кодам по мере разрастания своей таблицы цепочек.

В заключение заметим, что четыре рассмотренные примера алгоритмов сжатия используют различные подходы к определению информационной избыточности и различные методы для ее сокращения. Поэтому с практической точки зрения оправдано последовательное применение нескольких различных алгоритмов сжатия. Так, например, выходная кодовая последовательность алгоритма LZW может содержать много одинаковых кодов для некоторой часто встречающейся цепочки входных символов. В этом случае можно сжать сам код некоторым алгоритмом, ориентированным на анализ частотных характеристик появления отдельных байтов (Хаффмена либо арифметического кодирования). Для больших черно-белых изображений можно сначала использовать алгоритм RLE, а затем какой-либо другой алгорим.