Адаптивное арифметическое кодирование

Для арифметиеского кодирования существует адаптивный алгоритм. Его идея заключется в том, что частоты символов, использующиеся при кодировании входной последовательности, изменяются по мере получения каждого нового символа. То есть происходит «адаптация» алгоритма к входным данным. При декодировании происходит аналогичный процесс.

Алгоритм кодирования

На вход алгоритму передаётся последовательность символов и алфавит. Каждому символу алфавита $\alpha \in \sum$ сопоставляется его вес w_α. В начале работы алгоритма все веса символов равны 1. Символы располагаются в естественном порядке, например, по возрастанию. Вероятность каждого символа α — $p(\alpha)$ устанавливется равной его весу, делённому на суммарный вес всех символов: $p(\alpha) = \frac{w_\alpha}{\sum_{i=1}^{n} w_i}$. После получения очередного символа и построения нужного интервала, вес символа увеличивается на 1. Когда все символы последовательности будут обработаны, необходимо либо записать маркер конца последовательности, либо запомнить её длину, чтобы позже передать декодировщику. После получения нужных границ интервала $[l, r]$, в котором будет лежать код, необходмо выбрать число x, описывающее кодирование: $x \in [l, r]$. Выбор числа x производится также, как и в неадаптивном алгоритме. Выбирается число вида $\frac{x}{2^p} : x, p \in \mathbb{N}$.

Псевдокод алгоритма

- `in` — строка, подаваемая на вход;
- `Segment` — структура, задающая подотрезок отрезка $[0, 1)$, соответствующая конкретному символу. Имеет следующие поля:
  - `left` — левая граница подотрезка
  - `right` — правая граница подотрезка
- `m` — мощность алфавита;

- `weight` — веса символов алфавита;
- `segments` — набор подотрезков, соответствующих символам алфавита;
- `left, right` - границы отрезка, содержащие возможный результат арифметического кодирования;

Подотрезок

```
struct Segment:
    double left;
    double right;
```

Определение начальных границ подотрезков

```
Map<char, Segment> defineSegments(set<char> alphabet):
    double p = 1 / alphabet.size()
    Segments[m] segments
    double curLeft = 0
    double curRight = p
    for i = 0 to m - 1:
        segments[i].left = curLeft
        segments[i].right = curRight
        curLeft = curRight
        curRight = curRight + p
    return segments
```

Перестройка подотрезков

```
Map<Char, Segment> resizeSegments(alphabet : char[m], weight :
Map<char, int>, segments : Map<char, Segment>):
    double l = 0
    int sum = 0
    for i = 0 to m - 1:
        sum = sum + weight[i]
    for i = 0 to m - 1:
        char c = alphabet[i]
        segments[c].left = l
```

```
10          segments[c].right = l + (weight[c] / sum)
11          l = segments[c].right;
12      return segments
```

Построение алфавита по входной строке

```
1   set<char> getAlphabet(in : String):
2       Set<char> alphabet
3       for i = 0 to in.length() - 1:
4           alphabet.add(in[i])
5       return alphabet
```

Определение начальных весов символов алфавита

```
1   Map<char, int> defineWeights(alphabet : Set<char>):
2       Map<char, int> weight
3       for i = 0 to m - 1:
4           char c = alphabet[i]
5           weight[c] = 1
6       return weight
```

Кодирование строки

```
1   double adaptiveCoding(in : String, alphabet : Set<char>):
2       int[m] weight = defineWeights(alphabet)
3       int[m] segments = defineSegments(alphabet)
4       double left = 0
5       double right = 1
6       for i = 0 to n - 1:
7           char c = alphabet[i]
8           weight[c]++
9           double newRight = left + (right - left) * segments[c].right
10          double newLeft = left + (right - left) * segments[c].left
11          left = newLeft
12          right = newRight
13          resizeSegments(alphabet, weight, segments)
14      return (left + right) / 2;
```

Алгоритм декодирования

При декодировании последовательности символов также используется множество весов w символов алфавита $\sum$. В начале работы алгоритма все веса символов равны 1. На каждом шаге определяется интевал, содержащий данный код, по интервалу находится символ, который потом записывается в выходную последовательность. Вес полученного символа α увеличивается на 1. Отрезки, соответствующие символам алфавита, перестраиваются в зависимости от изменённых весов символов и размера текущего подотрезка. При получении символа конца последовательности или обработки нужного числа символов алгоритм завершает работу.

Псевдокод алгоритма

При декодировании строки будут использоваться функции `defineWeights()` и `defineSegments()` из алгоритма кодирования.

Декодирование

- `code` — вещественное число, подаваемое на вход;
- `len` — длина декодируемой строки;
- `m` — мощность алфавита;
- `weight` — веса символов алфавита;
- `segments` — набор подотрезков, соответствующих символам алфавита;

```
1   String decode(code : double, alphabet : Set<char>, int len):
2       Map<char, int> weight = defineWeights(alphabet)
3       Map<char, Segment> segments = defineSegments(alphabet)
4       String ans = ""
5       for i = 0 to len - 1:
6           for j = 0 to m - 1:
7               char c  = alphabet[j]
8               if code >= segments[c].left and code < segments[c].right:
9                   ans = ans + c
10                  weight[c]++
11                  code = (code - segments[c].left) /
12  segments[c].right - segments[c].left)
13                  resizeSegments(alphabet, weight, segments)
14                  break;
15      return ans
```

Оценка минимальной и максимальной длины кода

Теорема 1 *При адаптивном арифметическом кодировании строки длины l, символы которой принадлежат алфавиту мощности n, полученный код будет лежать в диапазоне* $[\frac{(n-1)!}{(n+l-1)!}, \frac{l!(n-1)!}{(n+l-1)!}]$

Доказательство. Во время кодирования строки алгоритм выбирает необходимый подотрезок, увеличивает вес символа и перестраивает подотрезки. Пусть L — результат кодирования строки длины l, использующей алфавит длины n. Код L формируется следующим образом: на каждом из шагов $k = 1, 2, \ldots, l$ он изменяется в $\frac{w_{\alpha_k}}{n+k-1}$ раз. В общем виде его можно представить так:

$$L = \prod_{i=1}^{l} \frac{w_{\alpha_i}}{n+i-1}$$

Знаменатель каждого следующего члена произведения будет увеличиваться на 1, так как на каждом шаге увеличивается вес одного из символов алфавита. Соответственно, чтобы минимизировать произведение, необходимо минимизировать числители его членов. Этого можно достичь, если передать на вход алгоритму строку, состоящую из неповторяющихся символов. В таком случае вес каждого из полученных символов

будет равен 1, а значение кода на каждом из шагов $k = 1, 2, \ldots, l$ будет изменяться в $\frac{1}{n+k-1}$ раз. Соответственно, формула примет вид:

$$L_{min} = \prod_{i=1}^{l} \frac{1}{n+i-1} = \frac{1}{\frac{(n+l-1)!}{(n-1)!}} = \frac{(n-1)!}{(n+l-1)!}$$

Можно записать, используя формулы комбинаторики:

$$L_{min} = \frac{1}{\binom{l}{n+l-1} l!} = \frac{1}{C^l_{n+l-1} l!}$$

Докажем верхнюю границу:

Для того, чтобы максимизировать произведение, необходимо увеличить числитель каждого члена произведения. Этого можно достичь, если передать на вход алгоритму строку, состоящую из одинаковых символов. В таком случае, на каждом из шагов $k = 1, 2, \ldots, l$ вес символа будет равен k, а значение кода будет изменяться в $\frac{k}{n+k-1}$ раз.

Соответственно, формула будет иметь следующий вид:

$$L_{max} = \prod_{i=1}^{l} \frac{i}{n+i-1} = \frac{l!(n-1)!}{(n+l-1)!}$$

Можно записать, используя формулы комбинаторики:

$$L_{max} = \frac{1}{\binom{n+l-1}{l}} = \frac{1}{C^l_{n+l-1}}$$

■

Следствие 1 *При адаптивном арифметическом кодировании строки длины l, символы которой принадлежат алфавиту мощности n, количество бит, необходимых для кодирования сообщения будет лежать в диапазоне* $[-\sum_{i=1}^{l} log_2 \frac{1}{n+i-1}, -\sum_{i=0}^{l-1} log_2 \frac{i+1}{n+i}]$

Доказательство. Произведём оценку количества бит, необходимых для записи кода L в общем случае:

$$log_2 L = -\sum_{i=1}^{l} log_2 \frac{w_{\alpha_i}}{n+i-1}$$

Все коды лежат в диапазоне $[0, 1)$.

Таким образом:

Максимальное количество бит будет затрачено при записи кода, являющегося минимальной дробью:

$$log_2 L_{min} = -\sum_{i=1}^{l} log_2 \frac{1}{n+i-1}$$

Минимальное число бит будет затрачено в случае записи кода максимального размера:

$$log_2 L_{max} = -\sum_{i=0}^{l-1} log_2 \frac{i+1}{n+i}$$

■

Необходимость применения адаптивного алгоритма возникает в том случае, если вероятностные оценки символов сообщения не известны до начала работы алгоритма. Преимущество такого подхода кодирования заключается в том, что декодировщику не нужно передавать вероятностные оценки для символов, он будет строить их по мере декодирования сообщения, что может сильно сократить вес такого сообщения.