

ЛЕКЦИЯ 24

Кодирование

1. Равномерное и неравномерное кодирование

Есть два глобальных подхода в кодировании текста: равномерное кодирование и неравномерное кодирование.

Обозначим алфавит допустимых символов \mathbb{A} .

В неравномерном кодировании код символов разной длины (например, UNICODE UTF 8 — одна из самых популярных кодировок).

Рассмотрим четырехбуквенное кодирование. Закодируем буквы «А», «Б», «В», «Г» таким образом:

$$\begin{aligned} \text{А} &= 0 \\ \text{Б} &= 1 \\ \text{В} &= 10 \\ \text{Г} &= 111 \end{aligned}$$

Тогда запись «ГАГА» можно закодировать так:

$$\text{ГАГА} = 11101110 = \text{БББГА},$$

т.е. декодирование неоднозначно, такое кодирование плохое.

Условие Фано: для того, чтобы сообщение, записанное с помощью неравномерного по длине кода, однозначно раскодировалось, достаточно, чтобы никакой код не был началом другого (более длинного) кода. **Обратное условие Фано** (ни один код не является концом (суффиксом) другого) также является достаточным условием однозначного декодирования неравномерного кода.

Тогда пусть

$$\begin{aligned} \text{А} &= 0 \\ \text{Б} &= 110 \\ \text{В} &= 10 \\ \text{Г} &= 111 \end{aligned}$$

Возьмем (для удобства рядом записан столбец в зеркальном отражении):

$$\begin{array}{l|l} \text{А} = & 1 & 1 \\ \text{Б} = & 10 & 01 \\ \text{В} = & 100 & 001 \\ \text{Г} = & 000 & 000 \end{array}$$

Тогда

$$\text{БАГАВА} = 10100011001$$

— 1100 нет, т.е. в конце ВА. 1000 тоже нет, т.е. по середине ГА. 101 нет, в начале БА. Однозначность есть, хотя и декодировать очень сложно.

Составим суффиксное дерево. Оно не нужно при декодировании, а при доработке дерева (дополнении алфавита) является полезным инструментом. Способ составления: отзеркаливаем код и строим дерево, в котором каждое ребро — цифра в коде.

Кодировка в равномерном кодировании UTF-16. Можно закодировать 2^n различных символов (мощность алфавита).

2. Поиск подстроки в строке

2.1. Наивный поиск подстроки в строке

Программа №2.1. Примитивный поиск подстроки в строке

```
1 s = "abbbbabbbaabbababababb"
2 subs = "bbbaba"
3 def find(s, sub):
4     for pos in range(0, len(s)-len(sub)+1):
5         for i in range(len(sub)):
6             if sub[i] != s[pos+i]:
7                 break
8         else:
9             return pos
10    return -1
```

Построчный комментарий кода:

4) Имеет смысл проходить по основной строке, пока входящая строка влезает в рассматриваемый участок.

5)–6) Пробегаемся по элементам входящей строки и смотрим, совпадают ли они с элементами основной.

7) Если нет, уже можно переходить к следующему элементу основной строки.

9) Если прошли по всем элементам строки вхождения, можно выдать ту позицию, начиная с которой есть вхождение.

10) Если вхождения нет, выдается '-1'.

Сложность алгоритма $O(N \cdot M)$. В итоге алгоритм получается неэффективным.

2.2. Конечный автомат поиска «abcd»

Смотрим на каждый символ только по одному разу! Методика хранения автомата: оргграф. Если конечный автомат уже построен, то время поиска $O(N)$, N — длина строки.

Конечный автомат поиска является частным случаем машины Тьюринга, Подход таков:

1. Изначально система в фазе ноль.
2. Сравниваем букву в основной строке с буквой во входящей строке. Если они совпали, то код продвигается на фазу вперед.
3. Сравниваем следующие буквы. Если они совпали, переходим в фазу два и т.д.
4. В случае несовпадения фаза становится нулевой.

Программа №2.2. Конечный автомат для поиска подстроки «abcd»

```
1 state = 0
2 for c in s:
3     if state == 0:
4         if c == "a":
5             state = 1
6     elif state == 1:
7         if c == 'b':
```

```

8         state = 2
9     elif c == 'a':
10         state = 1
11     else:
12         state = 0
13     elif state == 2:
14         if c == 'c':
15             state = 3
16         elif c == 'a':
17             state = 1
18         else:
19             state = 0
20     elif state == 3:
21         if c == 'a':
22             state = 1
23         elif c == 'd':
24             state = 4
25     else:
26         state = 0

```

3. Расстояние Левенштейна

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Есть 2 строки Мама и Мим. Мы можем превратить их друг в друга путем вставки символа, удаления символа. Минимальный путь в данном случае — удаление последнего и замена, т.е. длина пути 2. Так и определяется расстояние Левенштейна.

$a[:i]$, $b[:j]$ — срезы до i -го и j -го символа. $F_{ij} = L(a[:i], b[:j])$ — расстояние Левенштейна. Тогда

$$F_{ij} = \begin{cases} \text{Последние буквы совпадают, то } F_{(i-1)(j-1)} \\ 1 + \min(F_{(i-1)(j-1)}, F_{(i-1)j}, F_{i(j-1)}) \end{cases}$$

Г. С. Демьянов, [VK](#)
С. С. Клявинек, [VK](#)