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):

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

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 = "abbbbabbaaabbababaabb"
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: орграф. 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»

```
state = 0
for c in s:
    if state == 0:
        if c == "a":
            state = 1
    elif state == 1:
        if c == 'b':
```

```python
                state = 2
            elif c == 'a':
                state = 1
            else:
                state = 0
        elif state == 2:
            if c == 'c':
                state = 3
            elif c == 'a':
                state = 1
            else:
                state = 0
        elif state == 3:
            if c == 'a':
                state = 1
            elif c == 'd':
                state = 4
            else:
                state = 0
```

3. Расстояние Левенштейна

3.1. Определение

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

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

3.2. Реализация алгоритма поиска расстояния Левенштейна

Программа №3.1. Рекуррентная реализация поиска расстояния Левенштейна

```python
def lev(a, b):
    if not a:
        return len(b)
    if not b:
        return len(a)
    return min(lev(a[1:], b[1:])+(a[0] != b[0]), lev(a[1:], b)+1, lev(a, b[1:])+1)
```

Данный алгоритм записывается компактно, но асимптотика этого алгоритма ужасна.

Программа №3.2. Реализация поиска расстояния Левенштейна

```python
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)

    if len(s2) == 0:
        return len(s1)

    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row

    return previous_row[-1]
```

Г. С. Демьянов, VK
С. С. Клявинек, VK