МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра теоретических основ компьютерной безопасности и криптографии

ЛАБОРАТОРНАЯ РАБОТА

Алгоритм Гилберта-Мура

студента 4 курса 431 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Серебрякова Алексея Владимировича

Научный руководитель

доцент, к. п. н. _____ А. С. Гераськин

подпись, дата

Саратов 2022

Код Элиаса (Шеннона-Фано-Элиаса)

Так же известен как код Гильберта-Мура[1]

Каждому символу ставится в соответствие число

$$F_i = \frac{p_i}{2} + \sum_{j=1}^{i-1} p_j.$$

Кодовым словом i-го символа выбираются первые $l_i = \lceil -\log_2 p_i \rceil + 1$ дробных разрядов двоичного представления числа F_i.

Код Элиаса даёт оценку средней длины $L(C, X) \leq H(X) + 2$.

2.6 Код Гилберта-Мура

При построении кода Шеннона мы требовали упорядоченности сообщений по убыванию вероятностей. В алгоритме построения кода Шеннона сортировка букв входного алфавита, пожалуй, наиболее трудоемкая часть. Мы заметно упростим построение кода, если модифицируем кодирование таким образом, чтобы упорядоченность не требовалась. Графическая интерпретация кода Шеннона, показанная на рис. 2.7, подсказывает почти очевидный путь к решению этой задачи.

Предположим, что вероятности не упорядочены и тогда при кодировании сообщения с номером m нужно учитывать не только вероятность (длину отрезка) p_m, но и длину предшествующего отрезка p_{m-1}, которая может быть очень маленькой, почти нулевой, и тогда длина слова будет большой даже если вероятность p_m велика. Как же избавиться от влияния p_{m-1}?

Очень просто. Нужно соответствующую сообщению точку переме-

стить из начала отрезка (точка q_m) в его середину (точка $q_m + p_m/2$), а длину кодового слова l_m выбрать так, чтобы к концу передачи кодового слова длина интервала неопределенности была не больше $p_m/2$. Эти l_m бит и будут кодовым словом кода Гилберта-Мура!

Определим код Гилберта-Мура формально.

Рассмотрим источник, выбирающий буквы из алфавита $X = \{1, ..., M\}$ с вероятностями $\{p_1, ..., p_M\}$. Сопоставим каждой букве $m = 1, ..., M$ кумулятивную вероятность $q_m = \sum_{i=1}^{m-1} p_i$ и вычислим для каждой буквы величину σ_m по формуле

$$\sigma_m = q_m + \frac{p_m}{2}.$$

Кодовым словом кода Гилберта-Мура для x_m является двоичная последовательность, представляющая собой первые $l_m = \lceil -\log(p_m/2) \rceil$ разрядов после запятой в двоичной записи числа σ_m.

Пример 2.6.1 Рассмотрим источник с распределением вероятностей $p_1 = 0,1$, $p_2 = 0,6$, $p_3 = 0,3$. Вычисления, связанные с построением кода Гилберта-Мура для этого источника, приведены в таблице 2.2. Запись $[a]$ обозначает представление числа a в двоичной форме. В последнем столбце таблице показано, как выглядели бы кодовые слова $\tilde{\boldsymbol{c}}_m$ соответствующего кода Шеннона, если бы мы забыли упорядочить буквы по вероятностям. Видно, что код получился бы непрефиксным в отличие от кода Гилберта-Мура.

Таблица 2.2: Пример кода Гилберта-Мура

x_m	p_m	q_m	σ_m	l_m	$\boldsymbol{c}_m$ [a]	$\tilde{\boldsymbol{c}}_m$ [b]
1	0,1	0,0=[0,00000...]	0,05=[0,00001...]	5	00001	0000
2	0,6	0,1=[0,00011...]	0,40=[0,01100...]	2	01	0
3	0,3	0,7=[0,10110...]	0,85=[0,11011...]	3	110	10

[a] Кодовые слова кода Гилберта-Мура

[b] Кодовые слова кода Шеннона без упорядочения вероятностей букв

Докажем однозначную декодируемость кода Гилберта-Мура в общем случае. Для этого выберем сообщения с номерами i и j, $i < j$. Понятно, что $\sigma_j > \sigma_i$. Нужно доказать, что соответствующие слова $\boldsymbol{c}_i$ и $\boldsymbol{c}_j$ отличаются хотя бы в одном из первых $\min\{l_i, l_j\}$ кодовых символов.

Рассмотрим разность

$$\begin{aligned}
\sigma_j - \sigma_i &= \sum_{h=1}^{j-1} p_h + \frac{p_j}{2} - \sum_{h=1}^{i-1} p_h - \frac{p_i}{2} = \\
&= \sum_{h=i}^{j-1} p_h + \frac{p_j - p_i}{2} \geq \\
&\geq p_i + \frac{p_j - p_i}{2} = \\
&= \frac{p_j + p_i}{2} \geq \frac{\max\{p_i, p_j\}}{2}.
\end{aligned}$$

Вспомним, что длина слова и его вероятность связаны соотношением

$$l_m = \left\lceil -\log \frac{p_m}{2} \right\rceil \geq -\log \frac{p_m}{2}.$$

Отсюда следует

$$\sigma_j - \sigma_i \geq \frac{\max\{p_i, p_j\}}{2} \geq 2^{-\min\{l_i, l_j\}},$$

а это означает, что слова $\boldsymbol{c}_i$ и $\boldsymbol{c}_j$ отличаются в одном из первых $\min\{l_i, l_j\}$ двоичных символов и поэтому ни одно из двух слов не может быть началом другого.

Поскольку все слова кода Гилберта-Мура ровно на единицу длиннее слов кода Шеннона, получаем следующую оценку средней длины кодовых слов

$$\bar{l} < H + 2.$$

Мы завершим рассмотрение кода Гилберта-Мура графической интерпретацией, представленной на рис. 2.8 для источника из примера 2.6.1. Мы снова предполагаем, что передается буква с номером 2. Ей соответствует точка σ_2. По построению, соседние точки σ удалены от нее на расстояние по меньшей мере $p_2/2$. Кодер передает бит за битом и при этом каждый раз интервал неопределенности сужается вдвое. Передачу можно завершить, когда длина интервала неопределенности будет не больше $p_2/2$. В данном примере достаточно передать 2 бита.

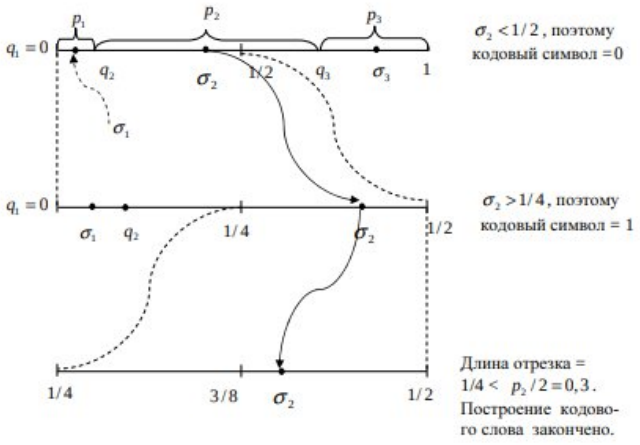

Рис. 2.8: Графическая интерпретация кода Гилберта-Мура

Код программы

```
opt = int(input('Сжать - 0, Разжать - 1\n'))

if opt == 0:
    f = open('studies\\tkisi\\Тест_8.txt')

    d = {}
    tree = {}
    alphabet = {}
    text = ''

    while True:
        string = f.readline()
        if not string:
            break
        else:
            text += string
            for i in string:
                if i == '\n':
                    j = '*'
                elif i == ' ':
                    j = '_'
                else:
                    j = i
                if d.get(j):
                    d.update({j: d.get(j) + 1})
                else:
                    d.update({j: 1})

    f.close()

    for i in list(d.keys()):
        d.update({i: d[i]/len(text)})

    def dict_sort(d):
        sort_d = sorted(d.items(), key=lambda x: x[1])
        return dict(sort_d)

    d = dict_sort(d)

    def create_tree(d):
        keys = list(d.keys())
        r = len(keys)
        tree = {keys[0]: d[keys[0]]/2}
        for i in range(r - 1):
            tree.update({keys[i + 1]: tree[keys[i]] +
                         d[keys[i]]/2 + d[keys[i + 1]]/2})
        return tree

    def create_word(tree, j):
        start = 0.5
        ans = ''
        for i in range(j):
            if (tree > start):
                ans += '1'
                tree -= start
            else:
                ans += '0'
            start /= 2
        return ans
```

```
    def create_alphabet_help(i, tree, d):
        j = -1
        size = d[i]
        iterator = 2
        while iterator > size:
            iterator /= 2
            j += 1
        word = create_word(tree[i], j)
        return word

    def create_alphabet(tree, d):
        alphabet = {}
        for i in list(tree.keys()):
            alphabet.update({i: create_alphabet_help(i, tree, d)})
        return alphabet

    tree = create_tree(d)
    alphabet = create_alphabet(tree, d)

    f = open("res6.bin", "wb")
    keys = ''
    for i in list(alphabet.keys()):
        keys += i + ' ' + alphabet[i] + ' '
    keys += '\n'
    f.write(str(keys).encode())

    code = ''
    for i in text:
        if i == ' ':
            code += alphabet['_']
        elif i == '\n':
            code += alphabet['*']
        else:
            code += alphabet[i]

    extra_zero = 0 if len(code) % 8 == 0 else 8 - len(code) % 8
    f.write((str(extra_zero) + '\n').encode())
    bts = '0' * extra_zero + code
    to_write = bytearray()
    for i in range(0, len(bts), 8):
        to_write.append(int(bts[i: i+8], 2))

    f.write(to_write)
    f.close()

else:
    f = open("res6.bin", "rb")

    d = {}
    tree = {}
    alphabet = {}
    text = ''
    ans = ''

    str_keys = f.readline().decode()[:-2].split()
    b = True
    for i in str_keys:
        if b:
            tmp = i
            b = False
```

```
    else:
        alphabet.update({i: tmp})
        b = True

count_of_zero = int(f.readline().decode())
dump = f.read()
bitstr = ''
for b in dump:
    bits = bin(b)[2:].rjust(8, '0')
    bitstr += bits

text = bitstr[count_of_zero:]
f.close()

def true_word(pos, word, text):
    if (len(word) <= len(text) - pos):
        b = True
        for i in range(len(word)):
            if word[i] == text[pos + i]:
                b = b and True
            else:
                b = b and False
    else:
        b = False
    return b

i = 0
while i < len(text):
    for j in list(alphabet.keys()):
        if true_word(i, j, text):
            ans += alphabet[j]
            i += len(j)
            break

ans = ans.replace('_', ' ')
ans = ans.replace('*', '\n')

f = open("restore6.txt", "w")
f.write(ans)
f.close()
```