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 Фано

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

Основные сведения [править | править код]

Кодирование Шеннона — Фано (англ. *Shannon–Fano coding*) — алгоритм префиксного неоднородного кодирования. Относится к вероятностным методам сжатия (точнее, методам контекстного моделирования нулевого порядка). Подобно алгоритму Хаффмана, алгоритм Шеннона — Фано использует избыточность сообщения, заключённую в неоднородном распределении частот символов его (первичного) алфавита, то есть заменяет коды более частых символов короткими двоичными последовательностями, а коды более редких символов — более длинными двоичными последовательностями.

Алгоритм был независимо друг от друга разработан Шенноном (публикация «Математическая теория связи», 1948 год) и, позже, Фано (опубликовано как технический отчёт).

Основные этапы [править | править код]

1. Символы первичного алфавита m_1 выписывают по убыванию вероятностей.
2. Символы полученного алфавита делят на две части, суммарные вероятности символов которых максимально близки друг другу.
3. В префиксном коде для первой части алфавита присваивается двоичная цифра «0», второй части — «1».
4. Полученные части рекурсивно делятся, и их частям назначаются соответствующие двоичные цифры в префиксном коде.

a_i	$p(a_i)$	1	2	3	4	Итого
a_1	0.36	0	00			00
a_2	0.18		01			01
a_3	0.18	1	10			10
a_4	0.12		11	110		110
a_5	0.09			111	1110	1110
a_6	0.07				1111	1111

Пример построения кодовой схемы для шести символов a_1 - a_6 и вероятностей p_i

Когда размер подалфавита становится равен нулю или единице, то дальнейшего удлинения префиксного кода для соответствующих ему символов первичного алфавита не происходит, таким образом, алгоритм присваивает различным символам префиксные коды разной длины. На шаге деления алфавита существует неоднозначность, так как разность суммарных вероятностей $p_0 - p_1$ может быть одинакова для двух вариантов разделения (учитывая, что все символы первичного алфавита имеют вероятность больше нуля).

Алгоритм вычисления кодов Шеннона — Фано [править | править код]

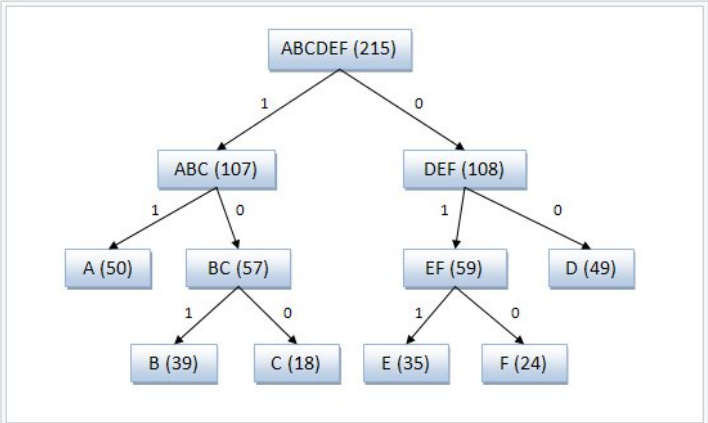
Код Шеннона — Фано строится с помощью дерева. Построение этого дерева начинается от корня. Всё множество кодируемых элементов соответствует корню дерева (вершине первого уровня). Оно разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Эти подмножества соответствуют двум вершинам второго уровня, которые соединяются с корнем. Далее каждое из этих подмножеств разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Им соответствуют вершины третьего уровня. Если подмножество содержит единственный элемент, то ему соответствует концевая вершина кодового дерева; такое подмножество разбиению не подлежит. Подобным образом поступаем до тех пор, пока не получим все концевые вершины. Ветви кодового дерева размечаем символами 1 и 0, как в случае кода Хаффмана.

При построении кода Шеннона — Фано разбиение множества элементов может быть произведено, вообще говоря, несколькими способами. Выбор разбиения на уровне n может ухудшить варианты разбиения на следующем уровне (n + 1) и привести к неоптимальности кода в целом. Другими словами, оптимальное поведение на каждом шаге пути ещё не гарантирует оптимальности всей совокупности действий. Поэтому код Шеннона — Фано не является оптимальным в общем смысле, хотя и дает оптимальные результаты при некоторых распределениях вероятностей. Для одного и того же распределения вероятностей можно построить, вообще говоря, несколько кодов Шеннона — Фано, и все они могут дать различные результаты. Если построить все возможные коды Шеннона — Фано для данного распределения вероятностей, то среди них будут находиться и все коды Хаффмана, то есть оптимальные коды.

Пример кодового дерева [править | править код]

Исходные символы:

- A (частота встречаемости 50)
- B (частота встречаемости 39)
- C (частота встречаемости 18)
- D (частота встречаемости 49)
- E (частота встречаемости 35)
- F (частота встречаемости 24)



Кодовое дерево

Полученный код: A — 11, B — 101, C — 100, D — 00, E — 011, F — 010.

Кодирование Шеннона — Фано является достаточно старым методом сжатия, и на сегодняшний день оно не представляет особого практического интереса. В большинстве случаев длина последовательности, сжатой по данному методу, равна длине сжатой последовательности с использованием кодирования Хаффмана. Но на некоторых последовательностях могут сформироваться неоптимальные коды Шеннона — Фано, поэтому более эффективным считается сжатие методом Хаффмана.

Код программы

```
import time

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

if ans == 0:

    a = open('studies\\tkisi\\Тест_8.txt')
    start_time = time.perf_counter()
    text = a.read()
    input_text = text
    d = []
    alph = []
    # Пробежимся по строке и будем добавлять в список пары (частота, символ) а затем удалять все вхождения символа из строки
    l_t = len(text)
    while len(text) > 0:
        x = text[0]
        alph.append(x)
        d.append((text.count(x)/l_t, x))
        text = text.replace(x, '')

    # Сортировка списка будет моделировать очередь с приоритетом
    d.sort()
    d = d[::-1]
    # Создадим дерево таким образом:
    #   Будем добавлять в дерево вершины в виде - имя_вершины : (имя_родителя, метка пути к родителю)
    d_1 = {}
    node_1 = ''
    for x in d:
        d_1[x[1]] = x[0]
        node_1 += x[1]

    tree = {}

    tree[node_1] = (1, -1, -1)
    q = [node_1]

    while len(q) > 0:
        parent = q[0]
        if len(parent) == 1:
            q = q[1:]
            continue
        c1 = ''
        c2 = ''
        c1p = 0
        c2p = 0
        for x in parent:
            if c1p < 0.5:
                c1p += d_1[x] / tree[parent][0]
                d_1[x] = d_1[x] / tree[parent][0]
                c1 += x
            else:
                c2p += d_1[x] / tree[parent][0]
                d_1[x] = d_1[x] / tree[parent][0]
                c2 += x
        tree[c1] = (c1p, 0, parent)
        if c2 != '':
            tree[c2] = (c2p, 1, parent)
```

```
    q = q[1:]
    q.append(c1)
    if c2 != '':
        q.append(c2)

#if len(d) == 1: tree[d[0][1]] =  (d[0][1]+d[0][1], 1)
print(tree)

dictionary = {}
l = list(tree.keys())
for x in alph:
    s = ''
    z = x
    while True:
        if tree[z][2] != -1:
            s = str(tree[z][1]) + s
            z = tree[z][2]
        else:
            break
    reversed(s)
    dictionary[x] = s

print('Сопоставим исходным символам их код: ', dictionary)

bin_file = open('res2.bin', 'wb+')
l = list(dictionary.keys())
bin_file.write(b' ')
for x in l:
    s2 = int(dictionary[x], 2)
    bin_file.write(x.encode())
    bin_file.write(dictionary[x].encode())
    bin_file.write(b' ')

if (l.count('\n') == 0):
    bin_file.write(b'\n')
s = ''
bin_file.write(b'\n')
for x in input_text:
    s += dictionary[x]
#print('\n', s, len(s))

if len(s) % 8 != 0:
    bin_file.write(str((8*(len(s)//8) + 8 - len(s))).encode())
    s = '0' * (8*(len(s)//8) + 8 - len(s)) + s
else:
    bin_file.write((str(0)).encode())
bin_file.write(b'\n')

#print('\n', s, len(s))

while s != '':
    sub = s[0:8]
    s = s[8:]
    s1 = int(sub, 2)
    s2 = s1.to_bytes((s1.bit_length() + 7) // 8, 'big')
    bin_file.write(s1.to_bytes((s1.bit_length() + 7) // 8, 'big'))
bin_file.close()

print("--- %s seconds ---" % (time.perf_counter() - start_time))
```

```
else:

    bf = open('res2.bin', 'rb')

    start_time = time.perf_counter()

    decode_text = bf.readline().decode('utf-8')
    decode_text += bf.readline().decode('utf-8')
    decode_text += bf.readline().decode('utf-8')
    # print(decode_text)

    output_text = bf.read()
    a = ''
    for x in output_text:
        a += '0' * (8 - len(bin(x)[2:])) + bin(x)[2:]

    if len(a) % 8 != 0:
        a = '0' * (8*(len(a)//8) + 8 - len(a)) + a

    output_text = decode_text + a

    # print(output_text)

    dec_dict = {}

    t = 0
    a = ''
    t2 = 0
    for x in output_text:
        if x == '\n' and t == 1:
            break
        if x == '\n':
            t += 1
        if x != ' ':
            a += x
            if t2 != 0:
                t2 = 0
        if x == ' ' and t2 == 1:
            t2 = -1
            a += x
        if x == ' ' and t2 != -1 and a != '':
            t2 = 1
            dec_dict[a[1:]] = a[0]
            a = ''

    output_text = output_text[output_text.index('\n')+1:]
    output_text = output_text[output_text.index('\n')+1:]

    zero = int(output_text[0])
    output_text = output_text[zero+2:]
    #print(dec_dict, output_text)

    out_f = open('restore2.txt', 'w')

    final_text = ''
    a = ''
    l = list(dec_dict.keys())
    for x in output_text:
        a += x
        if l.count(a) != 0:
            out_f.write(dec_dict[a])
```

```
        #final_text += dec_dict[a]
        a = ''

print("--- %s seconds ---" % (time.perf_counter() - start_time))
```