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 Хаффмана

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

Классический алгоритм Хаффмана [править | править код]

Идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частотностей символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево).[2]

1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Допустим, у нас есть следующая таблица абсолютных частотностей:

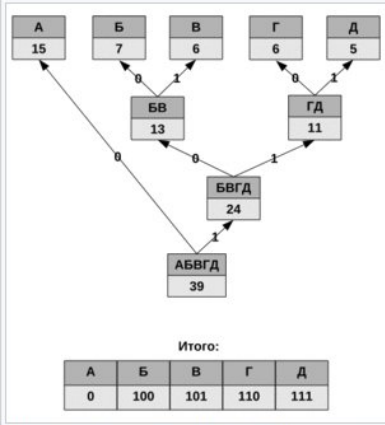
Символ	А	Б	В	Г	Д
Абсолютная частотность	15	7	6	6	5

Этот процесс можно представить как построение дерева, корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n_0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом.

Символ	А	Б	В	Г	Д
Код	0	100	101	110	111



Построение дерева для данного примера

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

При этом общая длина сообщения, состоящего из приведённых в таблице символов, составит 87 бит (в среднем 2,2308 бита на символ). При использовании равномерного кодирования общая длина сообщения составила бы 117 бит (ровно 3 бита на символ). Заметим, что энтропия источника, независимым образом порождающего символы с указанными частотностями, составляет ~2,1858 бита на символ, то есть избыточность построенного для такого источника кода Хаффмана, понимаемая как отличие среднего числа бит на символ от энтропии, составляет менее 0,05 бита на символ.

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Во-первых, для восстановления содержимого сжатого сообщения декодер должен знать таблицу частотностей, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частотностей, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частотностей и Н-дерева), другого - для собственно кодирования. Во-вторых, избыточность кодирования обращается в ноль лишь в тех случаях, когда вероятности кодируемых символов являются обратными степенями числа 2. В-третьих, для источника с энтропией, не превышающей 1 (например, для двоичного источника), непосредственное применение кода Хаффмана бессмысленно.

Код программы

```
import heapq
from collections import namedtuple
import ast
import os



class Node(namedtuple('Node', ['left', 'right'])):
    def walk(self, code, acc):
        self.left.walk(code, acc + '0')
        self.right.walk(code, acc + '1')


class Leaf(namedtuple('Leaf', ['char'])):
    def walk(self, code, acc):
        code[self.char] = acc or '0'


def read_text_from_txt_file(file_name):
    with open(file_name, 'r') as f:
        text = f.read()

    return text


def get_alphabet(text):
    alphabet = list(set(text))
    alphabet.sort()

    return alphabet


def get_symbol_frequencies(alphabet, text):
    frequencies = dict.fromkeys(alphabet, 0)
    for symbol in text:
        frequencies[symbol] += 1

    return frequencies


def huffman_encode(alphabet, text):
    frequencies = get_symbol_frequencies(alphabet, text)
    h = []
    for ch, freq in frequencies.items():
        h.append((freq, len(h), Leaf(ch)))

    heapq.heapify(h)

    count = len(h)
    while len(h) >= 2:
        freq1, _count1, left = heapq.heappop(h)
        freq2, _count2, right = heapq.heappop(h)
        heapq.heappush(h, (freq1 + freq2, count, Node(left, right)))
        count += 1

    code = {}
    if h:
        [(_freq, _count, root)] = h
```

```
        root.walk(code, "")
    return code


def huffman_decode(en, code):
    pointer = 0
    encoded_str = ''
    while pointer < len(en):
        for ch in code.keys():
            if en.startswith(code[ch], pointer):
                encoded_str += ch
                pointer += len(code[ch])
    return encoded_str


def main():

    opt = int(input())

    if opt == 0:
        path = 'studies\\tkisi\\Тест_8.txt'
        text = read_text_from_txt_file(path)
        alphabet = get_alphabet(text)
        code = huffman_encode(alphabet, text)
        encoded = "".join(code[ch] for ch in text)
        original_size = os.path.getsize(path)

        with open('studies\\tkisi\\res.bin', 'wb') as f:
            f.write((str(code) + '\n').encode())
            extra_zero = 0 if len(encoded) % 8 == 0 else 8 - len(encoded) % 8
            f.write((str(extra_zero) + '\n').encode())
            bts = '0' * extra_zero + encoded
            to_write = bytearray()
            for i in range(0, len(bts), 8):
                to_write.append(int(bts[i: i+8], 2))
            f.write(to_write)
            encoded_size = os.path.getsize('studies\\tkisi\\res.bin')
            coeff = original_size / encoded_size
            print(f'Сжали, коэффициент сжатия: {coeff}')

    else:
        with open('studies\\tkisi\\res.bin', 'rb') as f:
            tree = ast.literal_eval(f.readline().decode())
            trim = int(f.readline().decode())
            t = f.read()
            bitstr = ''
            for b in t:
                bits = bin(b)[2:].rjust(8, '0')
                bitstr += bits
            trimmed_bitstr = bitstr[trim:]
            result = huffman_decode(trimmed_bitstr, tree)

        with open('studies\\tkisi\\restored.txt', 'w') as f:
            f.write(result)
            print('Разжали')


if __name__ == '__main__':
    main()
```