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

Адаптивный код 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

Адаптивный алгоритм Хаффмана

Адаптивное кодирование Хаффмана (также называемое **динамическое кодирование Хаффмана**) — адаптивный метод, основанный на кодировании Хаффмана. Он позволяет строить кодовую схему в поточном режиме (без предварительного сканирования данных), не имея никаких начальных знаний из исходного распределения, что позволяет за один проход сжать данные. Преимуществом этого способа является возможность кодировать на лету.

Алгоритмы [править | править код]

Существует несколько реализаций этого метода, наиболее примечательными являются **«FGK»** (ФГК: Фоллер, Галлагер и Кнут) и алгоритм Виттера.

ФГК алгоритм [править | править код]

Он позволяет динамически регулировать дерево Хаффмана, не имея начальных частот. В ФГК дереве Хаффмана есть особый внешний узел, называемый *0-узел*, используемый для идентификации входящих символов. То есть, всякий раз, когда встречается новый символ — его путь в дереве начинается с нулевого узла. Самое важное — то, что нужно усекать и балансировать ФГК дерево Хаффмана при необходимости, и обновлять частоту связанных узлов. Как только частота символа увеличивается, частота всех его родителей должна быть тоже увеличена. Это достигается путём последовательной перестановки узлов, поддеревьев или и тех и других.

Важной особенностью ФГК дерева является принцип братства (или соперничества): каждый узел имеет два потомка (узлы без потомков называются листами) и веса идут в порядке убывания. Благодаря этому свойству дерево можно хранить в обычном массиве, что увеличивает производительность.[1][2]

Алгоритм Виттера [править | править код]

Код представляется в виде древовидной структуры, в которой каждый узел имеет соответствующий вес и уникальный номер.

Цифры идут вниз, и справа налево.

Веса должны удовлетворять принципу братства. Таким образом, если A является родительским узлом B и C является потомком B, то $W(A) > W(B) > W(C)$.

Вес — это всего лишь количество символов, встреченных ранее.

Набор узлов с одинаковыми весами представляют собой **блок**.

Чтобы получить код для каждого узла, в случае двоичного дерева мы могли бы просто пройти все пути от корня к узлу, записывая, например, «1» если мы идем направо, и «0» если мы пойдем налево.

Также в этом алгоритме используется специальный лист (узел без потомков), NYT (от англ. not yet transmitted — ещё не переданный символ), из которого «растут» новые, ранее не встречавшиеся, символы.

Кодер и декодер начинают только с корневого узла, который имеет максимальный вес. В начале это и есть наш NYT узел.

Когда мы передаем NYT символ, мы должны передать вначале код самого узла, а затем данные.

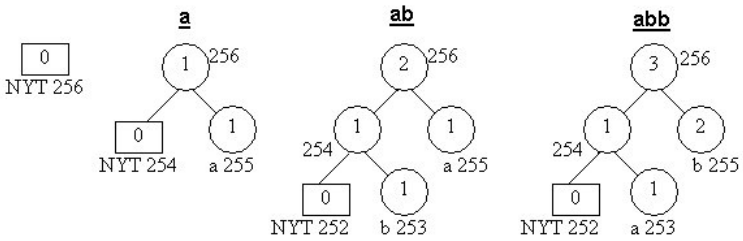
Для каждого символа, который уже находится в дереве, мы должны только передавать код конечных узлов (листов).

Для каждого передающегося символа передатчик и приёмник выполняют процедуру обновления:

1. Если текущий символ является не встречавшимся — добавить к NYT два дочерних узла: один для следующего NYT, другой для символа. Увеличить вес нового листа и старого NYT и переходить к шагу 4. Если текущий символ является не NYT, перейти к листу символа.
2. Если этот узел не имеет наибольший вес в блоке, поменять его с узлом, имеющим наибольшее число, за исключением, если этот узел является родительским элементом[3]
3. Увеличение веса для текущего узла
4. Если это не корневой узел зайти в родительский узел затем перейдите к шагу 2. Если это корень, окончание.

Примечание: замена узлов означает замену весов и соответствующих символов, но не чисел.

Пример [править | править код]



Начинаем с пустого дерева.

Для **«a»** передаём его двоичный код.

NYT порождает два дочерних узла: 254 и 255. Увеличиваем вес корня. Код **«a»**, связанный с узлом 255, становится 1.

Для **«b»** передавать 0 (код NYT узла), затем его двоичный код.

NYT порождает два дочерних узла: 252 для NYT и 253 для **b**. Увеличиваем веса 253, 254 и корня. Код для **«b»** равен 01.

Для следующего «b» передаётся 01.

Идём в лист 253. У нас есть блок весов в 1 и наибольшее число в блоке 255, так что меняем веса и символы узлов 253 и 255, увеличиваем вес, идём в корень и увеличиваем вес корня.

В будущем код «b» — это 1, а для «a» — это теперь 01, который отражает их частоту.

Код программы

```
import sys
import getopt
sys.path.insert(0, "studies\\tkisi")
from tree import Node
import array
import time

class FGK(object):
    def __init__(self):
        super(FGK, self).__init__()
        self.NYT = Node(symbol="NYT")
        self.root = self.NYT
        self.nodes = []
        self.seen = [None] * 10000

    def get_code(self, s, node, code=''):
        if node.left is None and node.right is None:
            return code if node.symbol == s else ''
        else:
            temp = ''
            if node.left is not None:
                temp = self.get_code(s, node.left, code+'0')
            if not temp and node.right is not None:
                temp = self.get_code(s, node.right, code+'1')
            return temp

    def find_largest_node(self, weight): # просто пробегаемся по списку вершин в обратном порядке
        for n in reversed(self.nodes): # и выбираем вершину с весом равным заданному
            if n.weight == weight:
                return n

    def swap_node(self, n1, n2):
        i1, i2 = self.nodes.index(n1), self.nodes.index(n2) # В общем списке всех элементов найдем
номера данных двух
        self.nodes[i1], self.nodes[i2] = self.nodes[i2], self.nodes[i1] # и в списке поменяем их местами

        tmp_parent = n1.parent # поменяем им их родителей
        n1.parent = n2.parent
        n2.parent = tmp_parent

        if n1.parent.left is n2: # если второй был левым потомком своего родителя
            n1.parent.left = n1 # то первй станет левым
        else:
            n1.parent.right = n1 # иначе правым

        if n2.parent.left is n1: # аналогично
            n2.parent.left = n2
        else:
            n2.parent.right = n2

    def insert(self, s): #Добавление символа в дерево
        node = self.seen[ord(s)] #Проверим присутствует ли наш элемент в списке из всевозможных
256 элементов

        if node is None:  # Если нет, то мы встретили его впервые
            spawn = Node(symbol=s, weight=1)  # создадим новый лист в дереве
            internal = Node(symbol='', weight=1, parent=self.NYT.parent,  # на месте старого NYT
создадим вершину с весом 1 и пустым именем
                left=self.NYT, right=spawn) #  родителем которой станет бывший родитель NYT
```

```
            spawn.parent = internal # Эта вершина станет родителем для нового листа
            self.NYT.parent = internal # И для нового NYT

            if internal.parent is not None:  # Если у нового узла не пустой родитель
                internal.parent.left = internal # то скажем что левым потомком этого родителя становится
наш новый узел
            else:
                self.root = internal # иначе этот узел корень

            self.nodes.insert(0, internal) # В список вершин строящегося дерева добавим новый узел
            self.nodes.insert(0, spawn) # и новый лист

            self.seen[ord(s)] = spawn # В списке всевозможных элементов отметим новый лист как уже
встретившийся
            node = internal.parent

        # Обновим дерево
        while node is not None: # пока не дошли до пустой вершины (несуществующий отец корня
дерева)
            largest = self.find_largest_node(node.weight)

            if (node is not largest and node is not largest.parent and
                largest is not node.parent): # если текущая вершина сама не наибольшая и не ее родитель
и не ее потомок
                self.swap_node(node, largest) # то поменяем их местами

            node.weight = node.weight + 1
            node = node.parent

    def encode(self, text):
        result = ''

        for s in text:
            if self.seen[ord(s)]: # if symbol already seen then return code
                result += self.get_code(s, self.root)
            else:
                result += self.get_code('NYT', self.root)  # else code of NYT followed by the code of the symbol
                result += bin(ord(s))[2:].zfill(8)

            self.insert(s)

        return result

    def get_symbol_by_ascii(self, bin_str):
        return chr(int(bin_str, 2))

    def decode(self, text):
        result = ''

        symbol = self.get_symbol_by_ascii(text[:8])
        result += symbol
        self.insert(symbol)
        node = self.root

        i = 8
        while i < len(text):  # читаем побитово
            node = node.left if text[i] == '0' else node.right # Переходим в левого потомка если 0, если 1 то
в правого
            symbol = node.symbol
```

```
            if symbol:  # если это не лист и не корень то есть имя этой вершины не пустое
                if symbol == 'NYT': # и если случилось так что мы этим путем из 0 и 1 дошли до NYT
                    symbol = self.get_symbol_by_ascii(text[i+1:i+9]) # то дальше будет символ и его нужно
считать
                    i += 8 # его длинна была 8 бит и его нужно перешагнуть

                result += symbol
                self.insert(symbol)
                node = self.root

            i += 1

        return result

def bytes2bits (textbin):
    byte = textbin.read(1)
    bincode = ""
    lastcode = ""
    while byte:
        code = str(int(bin(int.from_bytes(byte, byteorder="little"))[2:]))
        lastcode = code
        if len(code) < 8:
            code = "0" * (8 - len(code)) + code
        bincode += code
        byte = textbin.read(1)
    bincode = bincode[:-7]
    bincode += lastcode
    return bincode



def main(argv = '-e studies\\tkisi'):
    argv = []
    e_d = 'd'
    body = 'Compressed'
    r = 'bin'
    for i in range(11, 12):
        argv.append( f'-{e_d}  studies\\tkisi\\{body}{i}.{r}'.split('  ')[0] )
        argv.append( f'-{e_d}  studies\\tkisi\\{body}{i}.{r}'.split('  ')[1] )

    try:
        opts, args = getopt.getopt(argv, "e:d:")
    except getopt.GetoptError:
        sys.exit(2)

    text = None
    result = None
    i = 11
    for opt, arg in opts:
        if opt == '-e':
            with open(arg) as f:
                text = f.read()
            datac = array.array('B')
            start = int(round(time.time() * 1000))
            result = str(FGK().encode(text))
            copm_text = (open(f"Compressed{i}.txt", mode='w'))
            bintext = (open(f"Compressed{i}.bin", mode="wb"))
            tmp = result
            while len(result) > 0:
                datac.append(int(result[:8], 2))
                result = result[8:]
```

```
            end = int(round(time.time() * 1000))
            datac.tofile(bintext)
            end = int(round(time.time() * 1000))
            print ("The speed is: " + str((end - start)) + " ms")
            copm_text.write(tmp)
            copm_text.close()
            bintext.close()
            i += 1
        elif opt == '-d':
            with open(arg, 'rb') as f:
                bincode = bytes2bits(f)
            result = FGK().decode(bincode)
            decomp = open(f"Original{i}.txt", mode='w')
            decomp.write(result)
            decomp.close()


if __name__ == '__main__':
    main(sys.argv[1:])
```