САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Дискретная математика

Отчет

по лабораторной работе №1

«РЕАЛИЗАЦИЯ АЛГОРИТМА ХАФФМАНА»

Выполнили:

Иванов Александр Константинович (368220) гр. R3138

Нечаева Анна Анатольевна (312298) гр. R3138

Велюго Кирилл Олегович (367971) гр. R3138

Пивень Даниил Евгеньевич (368648) гр. R3137

Санкт-Петербург

2022

Цель работы:

Реализовать алгоритм Хаффмана.

Инструментарий и требования к работе:

Кодирование и декодирование должно быть реализовано в одной программе. В качестве пользовательского ввода использовать аргументы командной строки, пример ввода:

./my_script --encode input_file.txt output_file.txt (сжатие)

./my_script --decode input_file.txt output_file.txt (распаковка)

В сжатый файл сначала вписывается размер словаря, затем сам словарь и после – закодированный текст.

Выбранный язык реализации – Phyton.

Задачи:

1. Изучить алгоритм Хаффмана.
2. Разбить работу на «подзадачи»: создать словарь, реализовать функцию обхода графа – поиск в глубину (dfs), функции для кодирования и декодирования.
3. Протестировать полученную программу для нескольких примеров, убедиться, что результаты корректны.
4. Сформулировать вывод.

Ход работы:

Теоретическая часть.

Алгоритм Хаффмана – это алгоритм оптимального префиксного кодирования алфавита. Широко применяется в программах для сжатия данных, например, для сжатия изображений (JPEG, MPEG), в таких архиваторах, как PKZIP, LZH, в протоколах передачи данных HTTP и т.д.

Пусть $A = \{a_1, a_2, \ldots, a_n\}$ — алфавит из n различных символов, $W = \{w_1, w_2, \ldots, w_n\}$ — соответствующий ему набор положительных целых весов. Тогда набор бинарных кодов $C = \{c_1, c_2, \ldots, c_n\}$, где c_i является кодом для символа a_i, такой, что:

- c_i не является префиксом для c_j, при $i \neq j$,
- сумма $\sum_{i \in [1,n]} w_i \cdot |c_i|$ минимальна ($|c_i|$ — длина кода c_i),

называется **кодом Хаффмана**.

Рисунок 1 – Определение кода Хаффмана

Построение кода Хаффмана сводится к построению соответствующего бинарного дерева по следующему алгоритму:

1. Составляется список символов, которые необходимо закодировать, рассматривая каждый символ, как дерево, состоящее из одного элемента с весом, равным количеству вхождений символа в сжимаемое сообщение.
2. Выбираются 2 узла с наименьшими весами.
3. Формируется «родитель» этих двух узлов с весом, равным сумме весов «детей».
4. Новый узел добавляется к списку узлов, его «дети» удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.
6. Пока в списке больше одного узла, повторяются пункты 2 – 6.

Время работы алгоритма:

Если сортировать элементы после каждого суммирования или использовать приоритетную очередь, то алгоритм будет работать за время $O(N \log N)$. Такую асимптотику можно улучшить до $O(N)$, используя обычные массивы.

Пример: закодируем слово «рододендрон»

Шаг 1:

узел	д	о	н	р	е
вес	3	3	2	2	1

Шаг 2:

узел	д	о	ре	н
вес	3	3	3	2

Шаг 3:

узел	рен	д	о
вес	5	3	3

Шаг 4:

узел	до	рен
вес	6	5

Шаг 5:

узел	дорен
вес	11

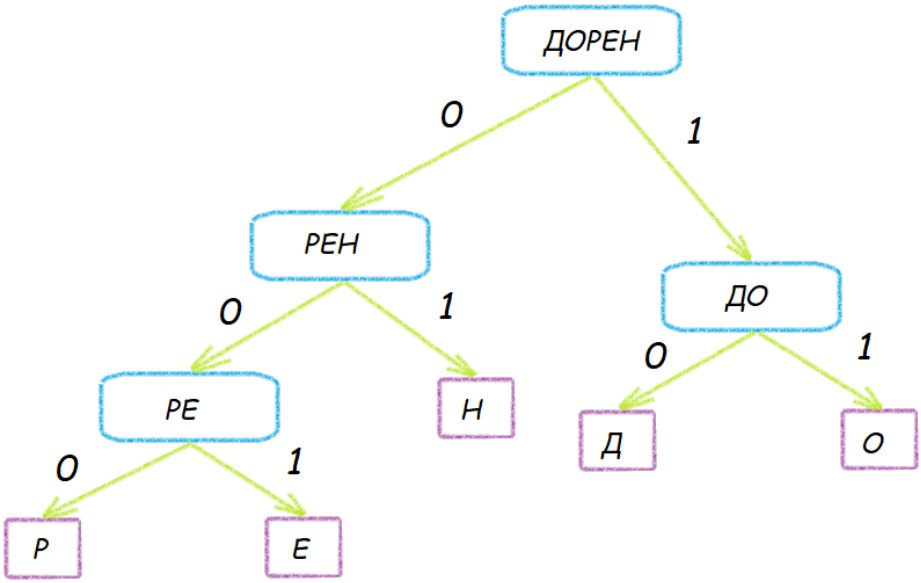

Рисунок 2 – бинарное дерево для кодирования «рододендрон»

Получаем код, соответствующий слову «рододендрон»:

000_11_10_11_10_001_01_10_000_11_01

Практическая часть.

Результаты:

Вывод:

В ходе выполнения лабораторной работы была изучена теория, необходимая для реализации программы для «кодирования / декодирования» на основе алгоритма Хаффмана. Для написания кода был выбран Python, так как он является наиболее оптимальным высокоуровневым языком для подобной программы. Программа была протестирована вручную на корректность работы.

Приложение:

Исходный код программы

```
from collections import Counter
import sys
import pickle



class Node(object):


    def __init__(self, char=None, value=0, right=None,
left=None,):
        self.value = value  # вес ноды ака количество таких
символов в тексте
        self.right = right  # правый ребенок
        self.left = left  # левый ребенок
        self.char = char  # символ, для которого работаем


    def __lt__(self, other):  # less than - для сортировкм нод
        return self.value < other.value


    def __repr__(self): # represent - по большей части для
дебага, возвращает описание ноды
        return "\"" + str(self.char) + "\"-" + str(self.value) +
" (" + str(self.right) + " " + str(self.left) + ")"



def create_dict(text):
    dict = {}
    nodes = []


    def dfs(node, way):
        if node.right or node.left:  # если есть ребенок
            dfs(node.right, way + "1") # запускаем для правого,
записываем путь
            dfs(node.left, way + "0")  # для левого
        else:
            dict[node.char] = way  # дошли до конца - записываем
в словарь код символа (путь)
            return
```

```
            c = Counter(text)  # считаем количество символов
            for e in c:
                nodes.append(Node(e, c[e])) # создаем ноду для каждого
символа


            while len(nodes) > 1:
                nodes.sort()  # сортируем и берем 2 самых маленьких ноды
(те, символов которых меньше всего в тексте)
                m1 = nodes.pop(0)
                m2 = nodes.pop(0)
                nodes.append(Node(m1.char + m2.char, m1.value +
m2.value, m1, m2))  # содаем ноду-родителя, символы конкатенируем, веса
складываем


            dfs(nodes[0], "")  # запускаем дфс
            return dict




        def encode(text, dict):  # просто бежим по символам и записываем
их код
            encoded = ""
            for el in text:
                encoded += dict[el]
            return encoded




        def decode(text, dict):
            letters = {value: key for key, value in dict.items()}  #
транспонируем словарь
            decoded = ""
            buff = ""


            for i in text:
                buff += i  # добавляем в буффер новый бит
                if buff in letters:  # если встретился символ с таким
кодом, записываем. Коды соотв. правилу фано - однозначное трактование
                    decoded += letters[buff]
                    buff = ""
            return decoded




        if __name__ == '__main__':
            if len(sys.argv) < 4:
                raise "Not enough arguments"
            method = sys.argv[1]
            in_path = sys.argv[2]
            out_path = sys.argv[3]
```

```
if method == '--encode':
    inf = open(in_path, 'r')
    ouf = open(out_path, 'wb')


    text = inf.read()
    print("Input text = " + text)


    dict = create_dict(text)  # создаем словарь с кодами
    pickle.dump(len(dict), ouf)


    for el in dict:
        pickle.dump(el, ouf)
        pickle.dump(dict[el], ouf)


    encoded = encode(text, dict)
    print("Encoded text = " + encoded)
    pickle.dump(encoded, ouf)


    inf.close()
    ouf.close()


if method == '--decode':
    inf = open(in_path, 'rb')
    ouf = open(out_path, 'w')


    l = pickle.load(inf)
    dict = {}


    for i in range(l):
        char = pickle.load(inf)
        code = pickle.load(inf)
        dict[char] = code


    text = pickle.load(inf)
    decoded = decode(text, dict)
    print("Decoded text = " + decoded)
    ouf.write(decoded)


    inf.close()
    ouf.close()
```