Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
(СибГУТИ)
Институт информатики и вычислительной техники
Кафедра прикладной математики и кибернетики

Практическая работа №3
по дисциплине «Теория информации»
на тему «Блочное кодирование»

Выполнили:
студенты гр.ИП-014
Обухов А.И.

Проверила:
Старший преподаватель каф. ПМиК
Дементьева Кристина Игоревна

Новосибирск 2024 г.

Цель работы: Экспериментальное изучение свойств блочного кодирования.

Язык программирования: C, C++, C#, Python

Результат: программа, тестовые примеры, отчет.

Задание:

1. Для выполнения работы необходим сгенерированный файл с неравномерным распределением из практической работы 1.

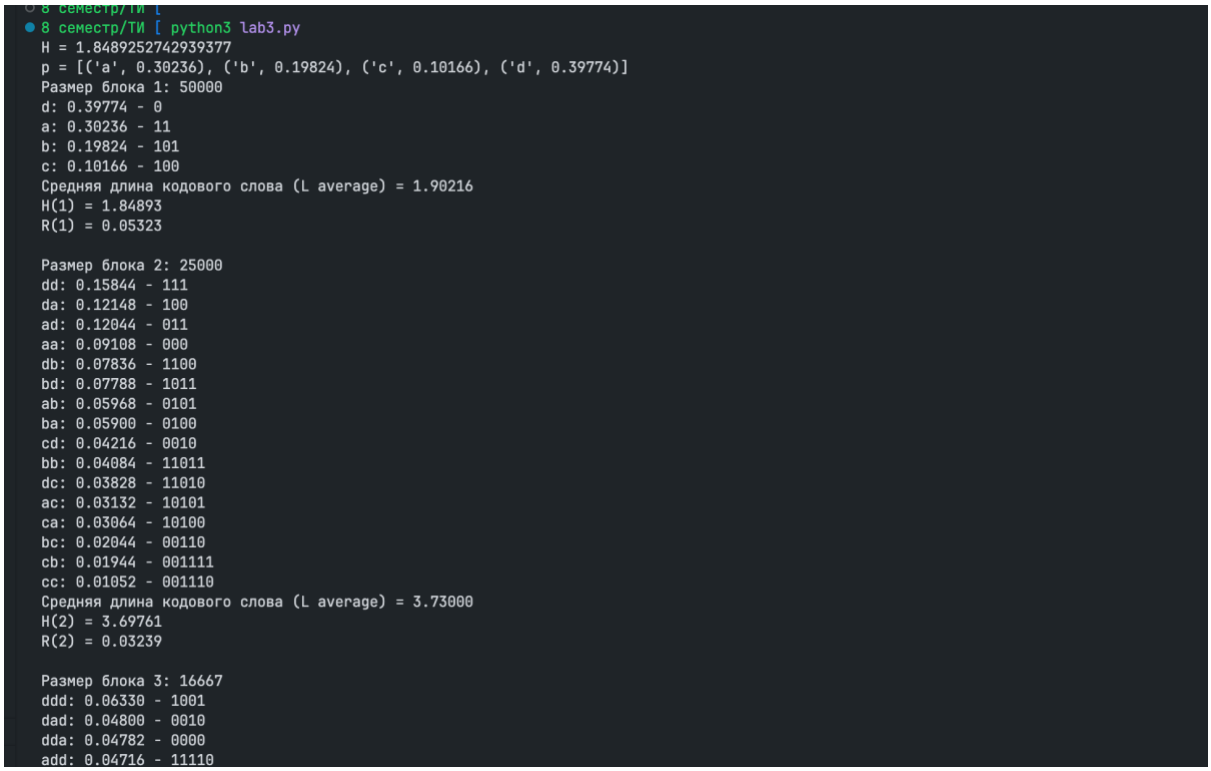
При блочном кодировании входная последовательность разбивается на блоки равной длины, которые кодируются целиком. Поскольку вероятностное распределение символов в файле известно, то и вероятности блоков могут быть вычислены и использованы для построения кода.

2. Закодировать файл блочным методом кодирования (можно использовать любой метод кодирования), размер блока $n = 1,2,3,4$. Вычислить избыточность кодирования на символ входной последовательности для каждого размера блока.

3. После тестирования программы необходимо заполнить таблицу и проанализировать полученные результаты, сравнить с теоретическими оценками.

	Длина блока n=1	Длина блока n=2	Длина блока n=3	Длина блока n=4
Оценка избыточности кодирования на один символ входной последовательности	0.05323	0.03636	0.03239	0.03009

Скриншоты работы программы:



Анализ результатов работы программы

В ходе работы были получены избыточности при кодировании блоков разной длины:

1. При размере блока n=1 избыточность составляет 0.05323
2. При размере блока n=2 избыточность составляет 0.03636
3. При размере блока n=3 избыточность составляет 0.03239
4. При размере блока n=4 избыточность составляет 0.03009

Таким образом, увеличение размера блоков в кодировании приводит к снижению избыточности на один символ входной последовательности. Это происходит из-за увеличения количества символов, на которые распространяется кодирование. В результате, требуется меньшее количество бит для кодирования символов, что снижает избыточность.

Теоретические оценки подтверждают, что с ростом размера блоков избыточность снижается, как и показали наши эксперименты.

Таким образом, использование больших размеров блоков в кодировании может быть более эффективным, особенно при работе с большими объемами данных.

Листинг программы

```
import heapq
from collections import Counter
import math
from lab1 import *
from lab2 import Node
from typing import Final

FILE_LENGTH: Final[int] = 50_000
PROBABILITIES: Final[dict[str, float]] = {'a': 0.3, 'b': 0.2, 'c': 0.1, 'd':0.4}
BLOCK_SIZES: Final[list[int]] = list(range(1, 5))

def generate_block_partitions(sequence, block_sizes: list[int]) -> list[list[str]]:
    partitions = []
    for size in block_sizes:
        partitions.append([sequence[i:i+size] for i in range(0, len(sequence), size)])

    return partitions


def calculate_entropy_by_block_size(text: str, block_size: int) -> float:
    blocks = [text[i:i+block_size] for i in range(0, len(text), block_size)]
    freqs = Counter(blocks)
    probs = {k: v / len(blocks) for k, v in freqs.items()}
    entropy = -sum(p * math.log2(p) for p in probs.values() if p != 0)
    return entropy


def huffman_encode(line: str, block_size: int) -> None:
    def build_huffman_tree(text):
        char_freq = Counter(text)
        heap = [Node(char, freq) for char, freq in char_freq.items()]
        heapq.heapify(heap)

        while len(heap) > 1:
            left = heapq.heappop(heap)
```

```
            right = heapq.heappop(heap)
            merged = Node(None, left.freq + right.freq)
            merged.left = left # type:ignore
            merged.right = right # type:ignore
            heapq.heappush(heap, merged)

        return heap[0]

    def build_huffman_codes(node, prefix="", codes={}):
        if node:
            if node.char is not None:
                codes[node.char] = prefix
            build_huffman_codes(node.left, prefix + '0', codes)
            build_huffman_codes(node.right, prefix + '1', codes)

    split_line = [line[i: i + block_size] for i in range(0, len(line), block_size)]
    probabilities = {k: v / len(split_line) for k, v in Counter(split_line).items()}
    probabilities = dict(sorted(probabilities.items(), key=lambda item: item[1], reverse=True))

    root = build_huffman_tree(split_line)
    codes = {}
    build_huffman_codes(root, "", codes)
    for i in probabilities.keys():
        print(f"{i}: {probabilities.get(i, .0):.5f} - {codes.get(i, '')}")
    l_average = sum(probabilities[i] * len(codes.get(i, '')) for i in probabilities.keys())
    print(f"Средняя длина кодового слова (L average) = {l_average:.5f}")

    entropy = calculate_entropy_by_block_size(line, block_size)
    print(f'H({block_size}) = {entropy:.5f}')
    r_h = l_average - entropy
    print(f"R({block_size}) = {r_h:.5f}")


def main() -> int:
    generate_file('./input/diff_prob.txt', PROBABILITIES, FILE_LENGTH)
    input_text = preprocess_file('./input/diff_prob.txt', 'en')
    orig_entropy = calc_entropy(input_text, 1)
    print(f"H = {orig_entropy[0]}\np = {orig_entropy[1]}")

    blocks = generate_block_partitions(input_text, BLOCK_SIZES)
    for size, block_list in zip(BLOCK_SIZES, blocks):
```

```
            print(f"Размер блока {size}: {len(block_list)}")
            huffman_encode(''.join(block_list), size)
            print()

    return 0

if __name__ == "__main__":
    exit(main())
```