НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Комп'ютерний практикум №3
з курсу алгоритми кодування двійкових даних

РЕАЛІЗАЦІЯ АЛГОРИТМУ СТИСКАННЯ ДАНИХ ХАФФМАНА

Виконав студент
групи ФІ-42мн
Бєш Радомир Андрійович

Київ — 2026

Зміст

1 Мета

Опанувати методи роботи із двійковими потоками даних та лагоритми кодування даних зі змінними та/або нефіксованими довжинами кодових слів.

2 Постановка задачі

Реалізувати класичний статичний алгоритм Хаффмана. Реалізація повинна бути у вигляді автономної утиліти, роботи з іменами файлів точно такі само, як і для Base64 із завдання №1. Для полегшення роботи можна вважати, що ваша програма працює із файлами, розмір яких не перевищує 2^{32} байтів (приблизно 4 Гб).
Результати дослідження:
`https://github.com/Radomir21/Encoding-Algorithms/tree/main/lab_3`.

3 Хід роботи

3.1 Особливості реалізації алгоритму Хаффмана

У моїй лабораторній роботі використовується статичний алгоритм Хаффмана для байтового алфавіту (0...255). Робота виконується з бінарними файлами.
Порядок стиснення:

1. Вхідний файл читається як послідовність байтів.
2. Обчислюється таблиця кількостей/частот $freq[256]$, де $freq[i]$ - це скільки разів байт i зустрічається у файлі.

```
16  def build_frequency_table(data):
17      freq = [0] * 256
18      for b in data:
19          freq[b] += 1
20      return freq
```

```
86  def encode_file(input_path, output_path=None):
87      if output_path is None:
88          output_path = input_path + ".huff"
89
90      with open(input_path, "rb") as f:
91          data = f.read()
92
93      freq = build_frequency_table(data)
94      root = build_huffman_tree(freq)
95      codes = build_code_table(root)
96
97      with open(output_path, "wb") as fout:
98          # 1) таблица частот
99          write_freq_table(fout, freq)
100
```

Для того, щоб записати таблицю частот на початку архіва була написана функція: `write_freq_table()` , в ній був застосован модуль `struct`, а саме метод `.pack`, який перетворює звичайне число в послідовніст байтів. Формат запису: «$<I$», тобто порядок байтів little-endian та uint (4 байта).

```
74  def write_freq_table(fout, freq):
75      for f in freq:
76          fout.write(struct.pack("<I", f))
77
```

3. За таблицею частот будується дерево Хаффмана.

```
23  def build_huffman_tree(freq):
24      heap = []
25      uid = 0
26
27      # листя
28      for sym in range(256):
29          f = freq[sym]
30          if f > 0:
31              heap.append((f, uid, make_node(f, sym)))
32              uid += 1
33
34      heapq.heapify(heap)
35
36      if len(heap) == 0:
37          return None
38
39      if len(heap) == 1:
40          return heap[0][2]
41
42      while len(heap) > 1:
43          f1, _, n1 = heapq.heappop(heap)
44          f2, _, n2 = heapq.heappop(heap)
45          parent = make_node(f1 + f2, None, n1, n2)
46          heapq.heappush(heap, (f1 + f2, uid, parent))
47          uid += 1
48
49      return heap[0][2]
```

У `build_huffman_tree(freq)` ми обираємо тільки ті символи, де `freq[sym] > 0`.

4. З дерева формується таблиця кодів: кожному байту відповідає двійковий код змінної довжини (ліворуч = 0, праворуч = 1).

```
52  def build_code_table(root):
53      codes = {}
54      if root is None:
55          return codes
56
57      if is_leaf(root):
58          codes[root[1]] = "0"
59          return codes
60
61      def dfs(node, path):
62          if is_leaf(node):
63              codes[node[1]] = path
64              return
65          if node[2] is not None:
66              dfs(node[2], path + "0")
67          if node[3] is not None:
68              dfs(node[3], path + "1")
69
70      dfs(root, "")
71      return codes
```

Тут є внутрішня функція DFS - (Depth First Search), тобто обхід дерева в глибину. `dfs(node, path)`, де `node` - поточний вузол дерева, `path` - шлях від кореня до вузла. Коли доходимо до листа `node[1]` - це байт (0 -255), `path` - його код Хаффмана. Далі два переходи: Ліворуч - 0, праворуч - 1. `dfs(root, "")` - початок з кореня.

5. Дані файлу кодуються у бітовий потік цими кодами й записуються у вихідний файл. Тут використовував бітовий потік з классу з ЛБ №2.

```python
def encode_file(input_path, output_path=None):
    if output_path is None:
        output_path = input_path + ".huff"

    with open(input_path, "rb") as f:
        data = f.read()

    freq = build_frequency_table(data)
    root = build_huffman_tree(freq)
    codes = build_code_table(root)

    with open(output_path, "wb") as fout:
        # 1) таблиця частот
        write_freq_table(fout, freq)

        # 2) бітовий потік
        bs = BitStream(fout, "w")
        try:
            for b in data:
                code = codes[b]
                for ch in code:
                    bs.write_bit(1 if ch == "1" else 0)
        finally:
            bs.close()

    return output_path
```

6. Формат стиснутого файлу: Перші 1024 байти — це таблиця частот (256 чисел по 4 байти, uint32). Далі йде закодований бітовий потік. Тобто кодове дерево прямо не зберігається: воно відновлюється декодером на основі збереженої таблиці частот. Особливість запису бітів, такий саме як в ЛБ №2

7. Декдоування виконується шляхом послідовного зчитування бітів та руху по дереву Хаффмана. При читанні біта 0 виконується перехід у ліву гілку дерева, при читанні біта 1 — у праву. При досягненні листового вузла відновлюється відповідний байт, після цього декодер повертається в корінь дерева. Процес повторюється, поки не буде відновлено `sum(freq)` байтів.

3.2 Порівняльний аналіз

Для аналізу було використано такі типи файлів по 10 файлів кожного типу:

- **TXT**
- **CSV**
- **BMP**
- **PDF**
- **EXE**

Коефіцієнт стискання розраховується, як:

$$K = \frac{\text{SIZE}_{\text{архів}}}{\text{SIZE}_{\text{оригінал}}} \tag{1}$$

Коефіцієнт стискання без метаданих розраховується, як:

$$K = \frac{\text{SIZE}_{\text{архів}} - 1024}{\text{SIZE}_{\text{оригінал}}} \tag{2}$$

Таблиці результатів:

Табл. 1: Коефіцієнт стиснення для різних типів файлів з метаданими

Тип файлів	**Кількість файлів**	$K_{\textbf{середнє}}$	$K_{\textbf{мін}}$	$K_{\textbf{макс}}$	**Коментар**
TXT	10	0.5917	0.5692	0.6273	добре стискаються
CSV	10	0.5965	0.4348	0.7689	добре стискаються
BMP	10	0.6817	0.2315	1.0985	добре стискаються
PDF	10	0.9955	0.9897	1.0005	погано стискаються
EXE	10	0.6839	0.3872	0.8388	добре стискаються

Табл. 2: Коефіцієнт стиснення для різних типів файлів без метаданих

Тип файлів	$K_{\textbf{середнє}}$	$K_{\textbf{мін}}$	$K_{\textbf{макс}}$
TXT	0.5886	0.5669	0.6226
CSV	0.5627	0.4246	0.7193
BMP	0.6469	0.2315	0.9274
PDF	0.9948	0.9895	0.9994
EXE	0.6669	0.3622	0.8348

4 Висновки

У даній лабораторній роботі було реалізовано статичний алгоритм стискання даних Хаффмана для роботи з бінарними файлами різних форматів. Реалізація використовує кодування даних у вигляді бітового потоку з подальшим вирівнюванням до межі байта. Для коректного декодування у стиснутому файлі зберігається таблиця частот символів розміром 1024 байти, за якою декодер відновлює дерево Хаффмана. Тестування проводилося на п'яти типах файлів (BMP, CSV, TXT, PDF, EXE), по десять файлів кожного типу. Для кожного типу було обчислено коефіцієнт стискання як з урахуванням метаданих, так і без них. Текстові файли (TXT, CSV) показали найкращі результати стискання. Середні значення коефіцієнта стискання становлять приблизно 0.59–0.60, що свідчить про значне зменшення розміру файлів. Файли формату BMP у середньому також добре стискаються (K приблизно 0.68), оскільки цей формат не використовує внутрішнє стиснення. Виконувані файли (EXE) у середньому також добре стискаються, коефіцієнт стискання близько 0.68. Файли формату PDF практично не зазнали стискання (K майже 1). Це очікуваний результат, оскільки PDF-файли зазвичай уже містять внутрішні алгоритми стискання (наприклад, для тексту та зображень), і додаткове застосування Хаффмана дає мінімальний ефект. Порівняння коефіцієнтів стискання з урахуванням і без урахування метаданих показало, що таблиця частот (1024 байти) істотно впливає на результат для невеликих файлів, зменшуючи загальну ефективність стискання. При цьому коефіцієнт без метаданих краще відображає якість самого алгоритму Хаффмана.