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

Комп'ютерний практикум №4
з курсу алгоритми кодування двійкових даних

РЕАЛІЗАЦІЯ АЛГОРИТМУ СТИСКАННЯ ДАНИХ LZW

Виконав студент
групи ФІ-42мн
Бєш Радомир Андрійович

Київ — 2026

Зміст

1 Мета **3**

2 Постановка задачі **3**

3 Хід роботи **3**
3.1 Особливості реалізації алгоритму LZW 3
3.2 Порівняльний аналіз 6

4 Висновки **8**

1 Мета

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

2 Постановка задачі

У даній роботі необхідно реалізувати алгоритм стискання даних Зіва-Лемпеля-Велч (LZW). Реалізація повинна бути у вигляді автономної утиліти, яка дозволяє стискати файли у архіви та розпаковувати їх назад. Для полегшення вважаємо, що реалізація працює із файлами, розмір яких не перевищує 2^{32} байтів (приблизно 4 Гб).
Результати дослідження:
`https://github.com/Radomir21/Encoding-Algorithms/tree/main/lab_4`.

3 Хід роботи

3.1 Особливості реалізації алгоритму LZW

У моїй лабораторній роботі використовується модуль BitStream побітового читання і запису даних з файлу, написаний в ЛБ №2.

1. **Організація словника алгоритму LZW:**
   Алгоритм використовує словник, який реалізовано у вигляді звичайного словника Python (хеш-таблиці). Початковий словник містить усі можливі одиничні байти (0–255), яким відповідають початкові коди. Пошук фраз у словнику виконується шляхом прямої перевірки наявності ключа у словнику. Якщо така фраза вже є у словнику, вона розширюється. Якщо ж ні, то код поточної фрази записується у бітовий потік, а нова фраза додається до словника з наступним вільним кодом.

```python
def lzw_compress(input_path, output_path=None, max_bits=16, overflow_mode="reset"):
    max_dict_size = 2^max_bits

    if output_path is None:
        output_path = input_path + ".lzw"

    with open(input_path, "rb") as fin, open(output_path, "wb") as fout:
        write_header(fout, max_bits, overflow_mode)
        bs = BitStream(fout, "w")

        dictionary = {bytes([i]): i for i in range(256)}
        next_code = 256

        w = b""

        while True:
            chunk = fin.read(4096)
            if not chunk:
                break

            for byte_val in chunk:
                k = bytes([byte_val])
                wk = w + k

                if wk in dictionary:
                    w = wk
                else:
                    if w:
                        write_bits_fixed(bs, dictionary[w], max_bits)

                    if next_code < max_dict_size:
                        dictionary[wk] = next_code
                        next_code += 1
                    else:
                        if overflow_mode == "reset":
                            dictionary = {bytes([i]): i for i in range(256)}
                            next_code = 256

                    w = k

        if w:
            write_bits_fixed(bs, dictionary[w], max_bits)

        bs.close()

    return output_path
```

2. **Алгоритм стиснення LZW:**
   Стиснення виконує функція `lzw_compress()`.Алгоритм побайтово зчитує вхідний файл та на кожному кроці намагається знайти у словнику найдовшу фразу, що вже там присутня. Для цього формується нова послідовність w + k, де w — поточна фраза, а k — наступний байт.

```python
            for byte_val in chunk:
                k = bytes([byte_val])
                wk = w + k

                if wk in dictionary:
                    w = wk
                else:
                    if w:
                        write_bits_fixed(bs, dictionary[w], max_bits)
```

3. **Обробка переповнення словника:**
   У реалізації передбачено два режими обробки переповнення словника: reset — словник очищується та повертається до початкового стану, freeze — словник перестає розширюватися.Вибір режиму здійснюється за допомогою параметра та зберігається у хедері стисненого файлу для коректного декодування.

```
                if next_code < max_dict_size:
                    dictionary[wk] = next_code
                    next_code += 1
                else:
                    if overflow_mode == "reset":
                        dictionary = {bytes([i]): i for i in range(256)}
                        next_code = 256
```

4. **Заголовок стисненого файлу:**
   Для коректного декодування у стисненому файлі зберігається заголовок, який містить сигнатуру формату (MAGIC) - це просто фіксована послідовність байтів, записана на початку файлу, максимальну довжину кодового слова та режим переповнення словника. Заголовок записується перед стисненими даними та зчитується під час декодування. Заголовок — це кілька байтів на початку файлу, які описують формат файлу та передають параметри кодування декодеру.

```
def write_header(out_f, max_bits, overflow_mode):
    out_f.write(MAGIC)
    write_uint8(out_f, max_bits)

    mode_byte = 0 if overflow_mode == "freeze" else 1
    write_uint8(out_f, mode_byte)


def read_header(in_f):
    magic = in_f.read(4)
    if magic != MAGIC:
        raise ValueError("Це не LZW файл (невірний MAGIC)")

    max_bits = read_uint8(in_f)
    mode_byte = read_uint8(in_f)

    if max_bits < 9 or max_bits > 16:
        raise ValueError(f"Некоректне max_bits={max_bits} (очікується 9..16)")

    overflow_mode = "freeze" if mode_byte == 0 else "reset"
    return max_bits, overflow_mode
```

5. **Алгоритм декодування:**
   Декодування реалізовано у функції `lzw_decompress`. Декодер полягає у послідовному зчитуванні кодів зі стисненого файлу та заміні їх відповідними послідовностями байтів зі словника. Словник формується динамічно під час декодування у тому ж порядку, що і при стисненні. У випадку, коли зчитаний код ще відсутній у словнику, фраза відновлюється як конкатенація попередньої фрази та її першого байта.

```python
def lzw_decompress(input_path, output_path=None):
    if output_path is None:
        if input_path.endswith(".lzw"):
            output_path = input_path[:-4]
        else:
            output_path = input_path + ".out"

    with open(input_path, "rb") as fin, open(output_path, "wb") as fout:
        max_bits, overflow_mode = read_header(fin)
        max_dict_size = 1 << max_bits

        bs = BitStream(fin, "r")

        dictionary = {i: bytes([i]) for i in range(256)}
        next_code = 256

        try:
            prev_code = read_bits_fixed(bs, max_bits)
        except EOFError:
            return output_path

        if prev_code not in dictionary:
            raise ValueError("Пошкоджені дані: перший код не в словнику")

        w = dictionary[prev_code]
        fout.write(w)

        while True:
            try:
                code = read_bits_fixed(bs, max_bits)
            except EOFError:
                break

            if code in dictionary:
                entry = dictionary[code]
            elif code == next_code:
                entry = w + w[:1]
            else:
                raise ValueError(f"Пошкоджені дані: зустріли невірний код {code}")

            fout.write(entry)

            if next_code < max_dict_size:
                dictionary[next_code] = w + entry[:1]
                next_code += 1
            else:
                if overflow_mode == "reset":
                    dictionary = {i: bytes([i]) for i in range(256)}
                    next_code = 256

            w = entry

        bs.close()

    return output_path

```

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:

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

Таблиці результатів:

Табл. 1: Коефіцієнт стиснення для різних типів файлів

Тип файлів	**Кількість файлів**	$K_{\textbf{середнє}}$	$K_{\textbf{мін}}$	$K_{\textbf{макс}}$	**Коментар**
TXT	10	0.4570	0.4051	0.4969	добре стискаються
CSV	10	0.4324	0.1311	0.7861	добре стискаються
BMP	10	0.5120	0.0231	1.2182	добре стискаються
PDF	10	1.3300	1.2533	1.4080	розмір збільшився
EXE	10	0.6381	0.2554	0.9022	добре стискаються

PDF вже стиснутий у середині, а LZW — це ще один словниковий алгоритм, який не може вдруге стиснути вже стиснені дані. Тому результати стиснення коректні. Розмір збільшився через наявність хедера та кодів фіксовоної довжини (16 біт).

Табл. 2: Порівняння середнього коефіцієнту стиснення Хаффмана та LZW

Тип файлів	$K_{\textbf{середнє}}$**Хаффман**	$K_{\textbf{середнє}}$**LZW**
TXT	0.5917	0.4570
CSV	0.5965	0.4324
BMP	0.6817	0.5120
PDF	0.9955	1.3300
EXE	0.6839	0.6381

Висновки за результатами порівняння алгоритмів Хаффмана та LZW:

Порівняльний аналіз показав, що алгоритм LZW забезпечує кращу ефективність стиснення для текстових та табличних файлів порівняно з алгоритмом Хаффмана. Для бінарних і виконуваних файлів обидва алгоритми демонструють подібні результати. Для вже стиснених форматів, таких як PDF, застосування LZW є неефективним і призводить до збільшення розміру файлів.

4 Висновки

У результаті проведеного порівняльного аналізу ефективності стиснення даних алгоритмом LZW було встановлено, що ефективність стиснення суттєво залежить від типу вхідних даних та рівня їх повторюваності. Найкращі результати стиснення було отримано для табличних та текстових файлів. Для файлів формату CSV середній коефіцієнт стиснення становив 0.4324, а для текстових файлів TXT — 0.4570, що свідчить про високу ефективність алгоритму для даних з повторювальною структурою. Файли формату BMP та EXE продемонстрували середню ефективність стиснення. Для зображень BMP середнє значення коефіцієнта становило 0.5120, однак спостерігалася значна варіативність результатів, що пояснюється різною структурою зображень. Для виконуваних файлів EXE середній коефіцієнт стиснення склав 0.6381, що є типовим результатом для бінарних файлів зі складною внутрішньою структурою. Найгірші результати було зафіксовано для файлів формату PDF, де середній коефіцієнт стиснення перевищив одиницю (1.3300). Це означає, що після застосування алгоритму LZW розмір файлів збільшився. Така поведінка пояснюється тим, що формат PDF зазвичай вже містить внутрішні механізми стиснення, а додаткове словникове кодування не дає позитивного ефекту. Отримані результати підтверджують, що алгоритм LZW найбільш ефективний для даних з високим рівнем повторюваності, зокрема текстових і табличних файлів, і малоефективний або неефективний для форматів, які вже містять стиснені дані. Загалом реалізація алгоритму LZW коректно працює для різних типів файлів та демонструє очікувані властивості словникового стиснення.