

Алгоритми перетворення інформації.
Завдання 3, звіт

Михайло Голуб

2 березня 2025 р.

Реалізація класу `BitArray`:

На відміну від аналогічного класу в завданні 2, цей клас використовує вбудований клас `bytearray` що є мутабельним. Це дозволяє пришвидшити роботу шляхом зміни байтів, не перестворюючи клас `bytes` кожного разу. Також було додано більше методів, які необхідні для роботи інших класів та функцій.

Перелік методів класу `BitArray`:

- `__init__(in_bytes, bit_pointer)` – ініціалізує клас, на вхід приймає список байтів та вказівник на біт, що не використовується в останньому байті (якщо вказівник рівний 8, значить останній байт використовується повністю);
- `__len__()` – повертає довжину файлу;
- `__str__()` – повертає текстову репрезентацію збереженого бітового рядка;
- `__rshift__(other)` – дозволяє робити `BitArray >> число`, знищує перші `*число*` біт в рядку;
- `__lshift__(other)` – дозволяє робити `BitArray << число`, додає `*число*` нулів в початок рядка;
- `__repr__()` – викликає `str(BitArray)`. Існує для відображення бітового рядка як бітового рядка, а не класа з адресою, в функціях що виводять помилки чи текст в консоль;
- `concat(right)` – повертає бітовий рядок, що є результатом конкатинації поточного рядка з іншим;
- `self_concat(right)` – конкатинує до поточного бітового рядка інший рядок. Пришвидшує роботу конкатинації, якщо не потрібно зберігати рядок до конкатинації;
- `__eq__(other)` – дозволяє робити `BitArray1 == BitArray2`;
- `__hash__()` – визначає геш представника класу, як геш його байтів. Метод необхідний для запису представника класу як індекса таблиць кодування та декодування;
- `__getitem__(key)` – дозволяє отримати біт, або підрядок бітів, на певній позиції шляхом виконання `BitArray[i]` та `BitArray[i:j]`;
- `get_bit(key)` – повертає біт на позиції `key`, дещо швидший за `BitArray[key]`;
- `append_bit(bit)` – дописує один біт в кінець бітового рядка;
- `copy()` – повертає представник класу ідентичний поточному представнику.

Реалізація класу `BitSequenceFile`:

Реалізація класу `BitSequenceFile` аналогічна реалізації в завданні 2, з виправленням деяких помилок.

Реалізація класу `ByteCounter`:

Мета класу – порахувати байти. Отримує на вхід шлях файлу, рахує байти в ньому та повертає словник виду `{byte: n_encountered}`. За потреби, результат роботи лічильника бути представлений в консолі у вигляді таблиці

Реалізація класу `HuffmanTree`:

Клас на вхід приймає `source`. Якщо це представник класу `ByteCounter` – будується дерево Хаффмана на основі лічильника. Інакше – клас створюється пустим та не готовим до роботи (необхідно для побудови дерева декодером).

Перелік методів класу `HuffmanTree`:

- `__init__(source)` – ініціалізує клас, якщо отримано лічильник – будує дерево;
- `build(recount = True)` – будує дерево, якщо `recount` – викликає перерахунок лічильника. Дерево будується наступним чином:
Створюється два словники виду `key: {n_encountered, left_child_key, right_child_key, parent_key}`, один словник містить усі вершини дерева, другий містить вершини що ще не були з'єднанні. `key` відповідає значенню байта для листків, або лежить в діапазоні 256-511 для вершин рошггалужень.
Допоки існують необ'єднанні вершини – обрати дві необ'єднанні вершини з найменшою сумарною вагою та об'єднати. `key` останньої вершини запам'ятати як `self.tree_key`.
Викликати `create_encoding_lookup` та `create_decoding_lookup`;
- `create_encoding_lookup()` – для кожного байта в списку вершин будує маршрут до кореня, шляхом переходу до батьківських вершин. Запам'ятовує бітовий рядок маршрута в словнику `encoding_lookup` виду `{byte: BitArray}`. Байти що не зустрічались жодного разу у словнику видаляються;
- `encode(object)` – побайтово знаходить відповідний байту бітовий рядок в `encoding_lookup`. Повертає бітовий рядок що відповідає коду Хаффмана для усіх вхідних байтів;
- `create_decoding_lookup()` – розвертає `encoding_lookup` для утворення `decoding_lookup` вигляду `{BitArray: byte}`;

- `decode(bit_array, prog = False)` – декодує бітовий рядок. Якщо `prog` – виводить прогресбар. Декодер побітово записує вхідний рядок в буфер (що є представником класу `BitArray`), якщо буфер є ключем в `decoding_lookup` – записує відповідний байт у вихідний `bytearray`. Після обробки усіх бітів – повертає масив байтів;
- `store()` – зберігає дерево у бітовий рядок наступного виду: перший байт – кількість розгалужень у дереві, наступні $4 \cdot \text{кількість розгалужень}$ байтів містять `key` лівої дитини (в двох перших байтах) та правої дитини (в третьому та четвертому байтах). Теоретично можна вмістити ключі дітей в 18 біт, але усі спроби це реалізувати були невдалими.

Реалізація кодування файлів:

На вхід приймається шлях файлу, бажаний шлях закодованого файлу (якщо такий відсутній, до поточного шляху файла дописується `.huff`) та розмір кроку з яким читати файл. Рахується кількість байтів у файлі, створюється дерево, дерево зберігається через `store()` в перших байтах файла, після чого файл читається та кодується кроками (за замовчуванням 1024 байта), результат кодування кроку одразу ж записується в файл (якщо файл не кратний довжині кроку – останній крок буде коротшим, щоб це врахувати). Після кодування додається ще один байт, в якому записано вказівник на перший невикористаний біт в останньому байті, оскільки довжина закодованої бітової послідовності може бути не кратна восьми.

Реалізація декодування файлів:

На вхід приймається шлях файлу, бажаний шлях декодованого файлу (якщо такий відсутній, з поточного шляху файла видаляється `.huff` (звісно ж, якщо це закінчення там присутнє)). Декодер читає кількість розгалужень в першому байті, читає інформацію про розгалуження в наступних $4 \cdot \text{кількість розгалужень}$ байтах та записує її в список вершин, ходить по кожному розгалуженню і прописує його дітям `key` батька, якщо дитина це байт – створює його у списку вершин, шукає корінь, записує список вершин та корінь в дерево та запускає `create_encoding_lookup()` і `create_decoding_lookup()`. Після цього читається вказівник у останньому байті, читається бітовий рядок між метаданими на початку файлу та останнім байтом, цей бітовий рядок декодується та результат записується в файл.

Аналіз метаданих:

В завданні наведено приклад метаданих що завжди займає 1024 байти. Імплементований варіант в найгіршому випадку використовує 2 байти + 4 байти на кожне розгалуження (для алфавіту довжиною 256 кількість розгалужень 255), тобто 1020 байтів. Для текстових файлів цей показник ближчий до 400 байт.

Якщо ж використовувати на кожне розгалуження не 4 байти, а 18 біт запропонованого варіанту, то в найгіршому випадку буде використовуватись 575 байт 6 біт.

Аналіз стиснення:

Для аналізу стиснення обрано наступні типи файлів: `.txt`, `.stl`, `.jpg`, `.mp3`, `.pdf`, `.mp4`. Обрано по 10 файлів для кожного типу. Оскільки обрані файли мають розміри значно більші за 1КБ (максимальний розмір метаданих), то коефіцієнт стиснення завжди буде рахуватись без відкидання метаданих з довжини файлу. Вказані відсоткові значення – відношення закодованого розміру файлу до некоедованого

Стиснення `.jpg` в середньому відсутнє. В найкращому випадку – 85.4% (чорний текст на білому фоні), в найгіршому – 118.4% (відносно малий розмір файлу, кавун на білому фоні).

Стиснення `.mp3` в середньому 99.09%. В найкращому випадку – 97.7%. В найгіршому – 99.8%.

Стиснення `.mp4` в середньому відсутнє. В найкращому випадку – 99.8%. В найгіршому – 100.1%.

Стиснення `.pdf` в середньому 99.34%. В найкращому випадку – 95.9% (текстовий `.pdf`). В найгіршому – 99.9%.

Стиснення `.stl` в середньому 74.27%. В найкращому випадку – 52.3%. В найгіршому – 91.3%

Стиснення `.txt` в середньому 59.83%. В найкращому випадку – 55.4% (Володар перстнів, перша книга). В найгіршому – 66.3% (конфігураційний файл гри Minecraft).

Ознайомитись з пофайловими результатами аналізу стиснення можна у файлі `files/test_log.txt`

Висновки з результатів аналізу стиснення:

- Найкраще стискаються текстові файли з природнім текстом, оскільки вони не використовують значну частину кодів символів, а наявні символи мають значно різні частоти.
- Найгірше стиснений текстовий файл, конфігураційний файл гри Minecraft містить багато символів (в основному цифр) з однаковою частотою, через це його стиснення було менш ефективним.

- Найгірше стискаються, а точніше не стискаються, типи файлів що вже були стиснуті. У їх числі MPEG, JPEG та PDF файли які використовують дуже ефективне стиснення самі по собі. Серед них найкраще стиснулось зображення чорного тексту на білому фоні.
- Створене програмне забезпечення не ефективно застосовувати на файлах розміром менше 10КБ, оскільки якщо файл має високий рівень випадковості байтів, то він буде стиснутий недостатньо, щоб вихідний файл був менший ніж початковий.

Посилання на код на GitHub:

https://github.com/MINIAProgramStudio/algorithms_of_data_transformation/tree/main/task3