Дискретный анализ. Курсовой проект. Алгоритмы LZW и арифметическое кодирование

Выполнил студент группы 08-307 МАИ *Рылов Александр.*

Условие

Архиватор. Реализуйте алгоритм LZW и арифметическое кодирование. Формат запуска должен быть аналогичен формату запуска программы gzip. Должны быть поддерживаться следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Метод решения

Алгоритм LZW – словарный адаптивный алгоритм сжатия данных из семейства LZ. Словарный – так как группа байт может быть закодирована одним кодом. Адаптивный – так как словарь дополняется во время кодирования/декодирования.
Кодирование в LZW работает следующим образом: входной текст считывается побайтово, байт добавляется к уже прочитанному слову. Если слово есть в словаре – добавить символ к концу слова и продолжать, иначе – вывовести код слова и добавить в словарь новый элемент: слово + считанный байт. Таким образом, файл будет преобразован в последовательность кодов.
Декодирование: считываем последовательность кодов, выводим соответсвующие им строки из словаря. На каждом шаге в словарь добавляется новое слово (конкатенация найденного + первый байт рабочей строки) – так восстанавливается словарь, который получился при кодировании. Особый случай – когда считанный код ещё не добавлен в словарь. Это происходит в том случае, если подстрока текста начинается и заканчивается на один и тот же символ – в этом случае конкатенируем рабочую строку с её же первым байт, добавляем это в словарь.

Арифметическое кодирование – символьный адаптивный алгоритм сжатия. Символьный – так как каждый байт (символ) кодируется отдельно. Адаптивный – так как распределение вероятностей изменяется в ходе кодирования/декодирования.
Кодирование работает следующим образом: входной текст считывается побайтово, дополняется модель распределения вероятностей, затем перерасчитываются границы диапазона кодирования. Если левая и правая границы приближаются друг к другу слишком близко, выполняется масштабирование (3 типа, возможно несколько масштабирований подряд, при этом происходит запись одного или нескольких битов в выходной файл). Бинарный файл на выходе представляет собой длинное дробное число, по которому при декодировании будет восстановлена исходная цепочка.
Декодирование: закодированное а архиве число побитово считывается, определяется текущий закодированный байт, обновляется текущее распределения вероятностей, пере-

расчитываются диапазоны кодирования. Для корректной работы значения внутренних переменных декодировщика должны полностью совпадать со значениями переменных при кодировании.

Описание программы

LZW реализован в классе *LZW* в файле *lzw.hpp*, который содержит два метода – *Compress*() и *Decompress*(), возвращающих закодированную/декодированную строку. В качестве словаря используется $std :: unordered_map$. Начальные словари инициализируются при создании экземпляра класса.
Арифметическое кодирование реализовано в классе *BAC* (binary arithmetic coder) в файле *bac.hpp*. Используется адаптивная модель в виде массива на 258 чисел, каждое число означает вероятность появляения какого-либо байта (индекс 256 соответствует EOF, 257 символизирует верхнюю границу всего диапазона).
Для побитовой работы с вводом/выводом оба кодировщика используют объекты класса *BitStream* из файла *bitstream.hpp*. Для измерения времени работы используется заголовочнй файл *timer_guard.hpp*. В нём класс, при инициализации запускающий таймер. При выходе экземпляра класса из области видимости вызывается дектруктор, в нём таймер останавливается, результат выводится в консоль. Логика работы консольной программы описана в файле *console.cpp*.
Флаги консольной программы:

- -c – результат кодирования/декодирования выводится в консоль
- -d – декодировать данный файл (по умолчанию используется LZW)
- -k – не удалать исходный файл
- -l – вывести информацию о размере входного и выходного файла, степень сжатия и время работы (при декодировании только время работы)
- -r – рекурсивно обойти введённую директорию и обработать каждый встреченный файл
- -1 – использовать арифметическое кодирование (по умолчанию используется LZW)
- -9 – использовать комбинацию из LZW + Arifm (комбинация Arifm + LZW из-за неэффективности не используется (бенчмарки ниже))

Дневник отладки

Изначально размер буфера в классе *BitStream* был равен всего 1 байту, я переписал класс, чтобы он поддерживал произвольный размер буфера (установлен на 128 килобайт).
В LZW длина кода должна быть динамической, но в так как декодировщик "отстаёт"на

один шаг от кодировщика, не сразу удалось это имплементировать. Та же проблема возникла со сбросом словаря при переполнении.

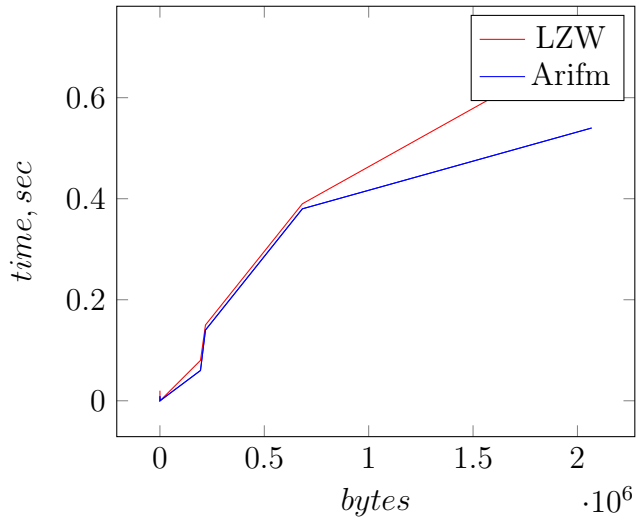
Анализ производительности

Тесты

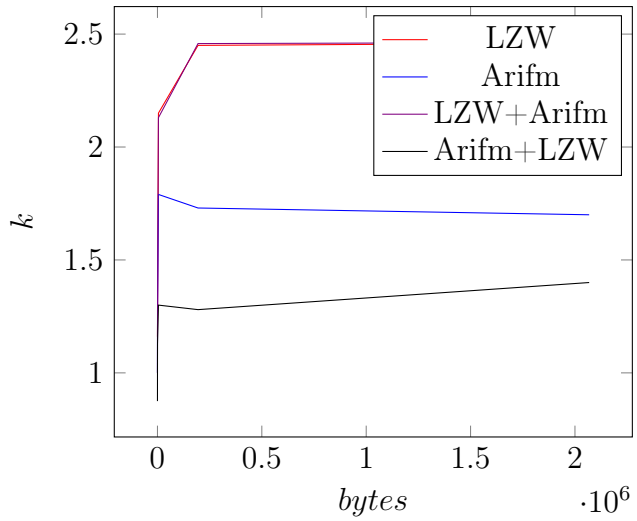
- Крочешный (14 байт) тест с повторениями
- Гамлет (одна тысячная часть)
- Гамлет полностью
- Евгений Онегин (на русском)
- Один из томов Британской Энциклопедии (буква B)

Все декодированные файлы были проверены на целостность и соответствие с оригиналом.

Зависимость скорости сжатия от размера входного текста



Зависимость коэффициента сжатия от размера входного текста



Как видно из графиков, арифметическое кодирование работает быстрее, чем LZW, но сжимает данные хуже. Комбинирование алгоритмов заметных преимуществ не даёт: LZW + Arifm лишь незначительно увеличивает степень сжатия, а Arifm + LZW сжимает даже хуже, чем каждый алгоритм отдельно (Arifm уничтожает повторяющиеся цепочки и LZW нечего кодировать). Также я выяснил, что ни один из алгоритмов и их комбинаций не сжимает изображения или видео (даже на raw файлах). Хотя арифметическое кодирование и используется в кодеках, изображение там проходит несколько этапов преобразований, прежде чем попасть кодировщик – вероятно, со специально преобразованными фото, Arifm справится лучше.

Выводы

Реализовал алгоритм LZW и арифметическое кодирования – оба адаптивные. В LZW поддерживается сброс словаря и кодовые цепочки переменной длины. В арифметическом кодировании отслеживается и предотвращается переполнения типа в модели распределения вероятности. LZW работает лучше арифметического кодирования на реальных текстах с большим количеством повторяющихся слов, но зато Arifm более эффективен по памяти и скорости работы. Комбинация этих алгоритмов не имеет смысла, так как коэффициент сжатия существенно не увеличивается.