Курсовой проект по курсу Дискретный Анализ. Алгоритм LZW.

Выполнил студент группы М8О-307Б-21 МАИ *Павлов Иван Дмитриевич.*

Условие

Реализовать алгоритм LZW. Начальный словарь выглядит следующим образом: $a \to 0; b \to 1; c \to 2; \dots x \to 23; y \to 24; z \to 25; EOF \to 26$

Описание алгоритма

Алгоритм LZW является одним из алгоритмов сжатия текстов типа LZ. Разработан Лемпелем, Зивом и Велчем. Может давать хорошие коэффициенты сжатия и часто применяется в сжатии файлов как составная часть композитных алгоритмов.

Данный алгоритм обнаруживает в тексте повторяющиеся цепочки и составляет таблицу, в соответствие каждой цепочке устанавливая уникальный код. Поэтому в случае частых повторений алгоритм будет эффективным. Важной особенностью алгоритма является то, что кодировщик и декодировщик могут строить таблицу в процессе кодирования / декодирования, что позволяет не хранить ее постоянно в памяти. При этом работа декодировщика фактически зеркальна работе кодировщика. Единственное ограничение - алфавит должен быть изначально известен.

Изначально словарь заполняется всеми возможными односимвольными цепочками из исходного алфавита. При кодировании ключем являются цепочки, при декодировании - коды со значениями согласно условию.

Алгоритм кодирования (сжатия):

Для каждого символа входной цепочки проверяем, есть ли он в словаре. Если символ найден, конкатенируем к нему следующий символ цепочки, пока получающиеся на каждом шаге подцепочки еще входят в словарь. Как только подцепочки в словаре нет, выдаем код самого длинного собственного префикса этой подцепочки, а саму подцепочку записываем в словарь, после чего присваиваем ей текущий символ. Коды изначально равны длине алфавита и увеличиваются на 1 каждый раз при добавлении в словарь.

Алгоритм декодирования (разжатия):

Первый код всегда является односимвольной цепочкой, входящей в алфавит. Проход по входной цепочке кодов начинаем со второго символа, при этом храня предыдущую цепочку (изначально первый символ). На каждом шаге алгоритма проверяем, входит ли код в словарь. Если входит, помещаем в словарь по ключу следующего кода (увеличенного на 1) значение, равное конкатенации предыдущей цепочки и первого символа текущей цепочки (взятой из словаря по текущему коду). После этого переприсваиваем предыдущую цепочку и выдаем текущую. Случай, когда текущий код в словаре не найден может быть получен при кодировании только тогда, когда текущая цепочка

совпала с предыдущей. Поэтому действуем аналогично, только первый символ берем из предыдущей цепочки.

Сложность работы алгоритма $O(n \cdot k)$, где n - длина цепочки, k - сложность чтения и вставки словаря. В случае реализации на хеш таблице сложность будет $O(n)$, если на сбалансированном дереве - $O(n \cdot logn)$.

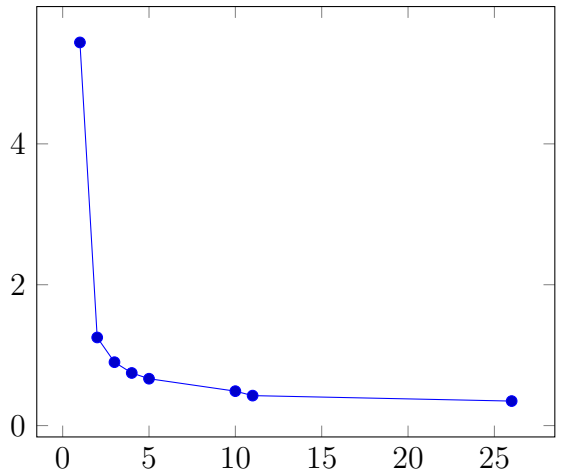
Описание программы

Программа написана на языке C++ и реализует описанный выше алгоритм. Реализация представлена классом ArchiverLZW, содержащем внутри себя 2 словаря, для сжатия и разжатия, поле, определяющее что данный архиватор будет делать (сжимать или разжимать), метод инициализации словаря, методы кодирования и декодирования.

На оценку удовлетворительно было достаточно хранить коды в виде целых чисел размером 32 бита. Для оптимизации лучше было использовать $unordered\_map$, но тесты чекера были пройдены и с помощью обычного $std :: map$. Эвристика с обнулением словаря не была применена из-за условия. Для экономии памяти я не храню результат кодирования, а вывожу сразу по ходу алгоритма.

Тест производительности

В качестве метрики возьму коэффициент сжатия. Он показывает, во сколько раз сжатые данные меньше исходных. Например, если исходный файл имеет размер 100 МБ, и после сжатия его размер становится 50 МБ, коэффициент сжатия будет равен 2. В реализации на оценку удовлетворительно коды хранились в 32-битных целых числах, при замере будем использовать их же. В качестве входных данных будет взята рандомная строка из $n = 1, 2, \ldots, 26$ уникальных символов латинского алфавита. Ниже представлен график зависимости коэффициента сжатия от количества уникальных символов в тексте:



Следует понимать, что такой результат получился только потому, что на хранение

кодов тратится неоправданно много памяти. При кодировании непрерывной битовой строки алгоритм даст куда лучший результат.

Выводы

Алгоритм LZW, разработанный Лемпелем, Зивом и Велчем, подтвердил свою эффективность при сжатии текстовых данных. Основываясь на полученных результатах, можно утверждать, что коэффициент сжатия зависит от уникальности символов в тексте, что демонстрирует график зависимости. Однако следует отметить, что текущая реализация, основанная на 32-битных целых числах для хранения кодов, может быть не самой оптимальной в плане экономии памяти. Тем не менее, даже при такой реализации алгоритм показал свою работоспособность.