Курсовой проект по курсу Дискретный Анализ. Arifm + LZW.

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

Условие

Необходимо реализовать два известных метода сжатия данных для сжатия одного файла.

- LZW
- Арифметическое кодирование

Формат запуска должен быть аналогичен формату запуска программы gzip. Должны быть поддержи- ваться следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Описание алгоритма

Алгоритм LZW

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.

Оптимизации:

Для того, чтобы алгоритм действительно начал сжимать данные, необходимо в файл записывать коды не в виде 32-битных чисел, а в виде битовых цепочек переменной длины, не обязательно кратной 8. Изначально в словаре находится 256 ASCII-символов, которые полностью заполняют собой коды длины 8 и меньше. С кода 256 пойдут цепочки длины 9 бит, с кода 512 - коды длины 10 бит, и т.д. А в словаре для удобства коды можно также хранить в виде чисел.

Кроме того, в какой-то момент может произойти переполнение числа типа unsigned int, являющегося кодом внутри словаря, а также переполнение оперативной памяти из-за словаря. Поэтому в определенный момент можно сбросить словарь и начать его заполнение заново. Это снизит эффективность сжатия, зато позволит избежать проблем с памятью. Перед сбросом словаря в файл посылается код очистки, чтобы декодировщик тоже сбросил словарь в нужный момент.

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 - $O(n \cdot logn)$.

Арифметическое кодирование

Алгоритм арифметического кодирования, разработанный в начале 1970-х годов, является одним из методов сжатия данных, обеспечивающих высокую степень сжатия за счет использования вероятностных моделей для представления данных. Этот алгоритм особенно эффективен при сжатии данных с высокой степенью избыточности и может достигать энтропийного предела, то есть теоретически минимально возможного размера данных без потерь, основываясь на их статистических характеристиках.

Классическое арифметическое кодирование

- В кодировщике символы входного потока преобразуются в длинные числовые коды. Это достигается путем деления числового интервала [0, 1) на подинтервалы, пропорциональные вероятностям встречаемости каждого символа.

- В процессе кодирования интервал сужается, отражая последовательность входных символов. Результатом является одно длинное число, представляющее всю последовательность.

- Декодировщик выполняет обратный процесс, интерпретируя числовой код и восстанавливая исходную последовательность символов.

Оптимизации

1. Адаптивное Арифметическое Кодирование. В адаптивной версии алгоритм динамически обновляет вероятности символов по мере их появления, что позволяет эффективно справляться с изменяющимися распределениями данных.

2. Для упрощения вычислений и представления чисел алгоритм реализуется с использованием целочисленной арифметики, что уменьшает требования к вычислительным ресурсам.

3. Нормализация и Масштабирование. Нормализация используется для поддержания числового интервала в управляемых пределах, предотвращая переполнение и потерю точности. Масштабирование применяется для расширения интервала, когда он становится слишком мал, что помогает поддерживать высокую точность кодирования.

Сложность кодирования этим алгоритмом $O(n)$, где n - длина текста, так как пересчет таблицы частот происходит за константное время. Сложность декодирования составляет $O(nlogn)$, из-за бинарного поиска.

Создание композитного алгоритма

При выборе порядка использования алгоритмов важно понимать, что алгоритм LZW эффективен в устранении повторяющихся последовательностей и сокращении общего количества данных без потерь, в то время как арифметическое кодирование может эффективно работать с более однородным и сокращенным набором данных. Поэтому LZW следует применить первым. Он уменьшит количество повторений в данных, а арифметическое кодирование будет полезно для дополнительного сжатия, так как после LZW может остаться много избыточных данных, например после сброса словаря.

Описание программы

Программа реализует функционал утилиты gzip с флагами, указанными в условии. Вот перечисление работы флагов с описанием их реализации:

1. -c - вместо файла вывести сжатые (разжатые) данные в консоль. Присваиваем переменной consoleOutput значение true, и затем выведем результат в std::cout вместо output.

2. -d - активировать декодировщик. В реализации каждого из алгоритмов есть методы сжатия (Compress) и разжатия (Uncompress). Если есть такой флаг, вызываем метод разжатия.

3. -h - получить окно с помощью. Реализовано, используя cout.

4. -k - не удалять исходный файл. Переменной keep присваиваем значение true. Если это так, не удаляем исходный файл после сжатия.

5. -l - статистика по сжатому файлу. Разжимаем файл и подсчитываем его длину, а также длину сжатого файла. Подсчитываем коэффициент сжатия и выводим. Разжатый файл удаляем.

6. -r - рекурсивно сжать все файлы в директории. С помощью библиотеки filesystem, используя recursive directory iterator и isdirectory можно рекурсивно пройти по всем файлам в директории и сжать их теми же методами, если они не являются директориями.

7. -t - проверить целостность сжатого файла. С первого по четвертый бит сжатого файла занимает контрольная сумма CRC-32, взятая от несжатого файла. Контрольную сумму берем с помощью zlib. После этого разжимаем файл, берем от разжатого файла контрольную сумму и сравниваем. Если суммы совпали, то проверка пройдена.

8. -1 и -9 определяют степень сжатия. В реализации LZW словарь сбрасывается, когда начинаются 16-битные коды, с флагом -1 - 11-значные, а с флагом -9 коды длины 31. Таким образом флаг -9 позволяет добиться лучшего коэффициента сжатия, в то время как флаг -1 позволяет тратить меньше оперативной памяти и времени.

Программа также может работать с данными из консоли (stdin), если в качестве имя файла указан - или файла нет. Также архиватор учитывает возможные ошибки входных данных, аналогично утилите gzip.

Вспомогательные утилиты

1. BitStream - надстройка над istream и ostream, которая позволяет писать и читать данные из потока по одному биту. Реализовано с помощью хранения текущего байта в памяти, и количество оставшихся бит. Если leftbits = 0, то читается следующий байт из потока и возвращается его бит, leftbits становится 8. Иначе, выдаем последний бит из текущего байта и текущий байт сдвигается на 1. leftBits при этом уменьшается. Также есть метод fillZeros, который заполняет последний байт нулями, так как число бит не кратное 8 хранить в памяти невозможно.

2. BinPow, CalculateCRC32, GenerateRandomString - вспомогательные функции быстрого возведения в степень, подсчета контрольной суммы и генерации рандомной строки.

Реализация арифметического кодирования

Класс Frequency реализует таблицу частот символов в тексте. Конструктор класса инициализирует объект Frequency, задавая начальные значения для всех символов (по умолчанию 1 для каждого символа). Размер data и cumulative устанавливается равным 258 (257 символов + 1), что соответствует расширенному ASCII. Вызывает init(), чтобы заполнить cumulative начальными кумулятивными суммами.

Метод init() вычисляет кумулятивную сумму частот для каждого символа. Это необходимо для определения диапазонов в арифметическом кодировании.

Метод get возвращает частоту для заданного символа, set устанавливает новую частоту для символа и очищает cumulative, так как кумулятивные суммы необходимо пересчитать, а inc увеличивает частоту заданного символа на 1 и также очищает cumulative. Метод right возвращает правую границу диапазона для символа, что соответствует кумулятивной частоте следующего символа. Метод left возвращает левую границу диапазона для символа, что соответствует его собственной кумулятивной частоте. Метод getnum возвращает общую сумму всех частот (num), которая используется в арифметическом кодировании для определения масштаба частот.

Кумулятивные суммы используются для определения начальной и конечной границы диапазона каждого символа. В арифметическом кодировании, символу соответствует определенный диапазон значений в зависимости от его частоты. left(symbol) возвращает нижнюю границу диапазона, а right(symbol) - верхнюю. Когда частота символа изменяется (через метод set() или inc()), кумулятивные суммы нужно пересчитать, так как изменение частоты одного символа влияет на диапазоны всех последующих символов. Очищение cumulative и последующий его перерасчет в init() обеспечивают актуализацию этих диапазонов.

Инициализация в Конструкторе Arifm. sumRange устанавливается в 32, что определяет диапазон значений для кодирования. max - максимальное значение, которое может быть представлено (2^{32}). half и quarter - половина и четверть от max соответственно, используются для нормализации интервалов. min - минимальный порог для нормализации. leftBits - счетчик отложенных битов. mask - маска для ограничения размера интервалов. left и right - начальные значения границ интервала. code - кодовое значение для декодирования.

При кодировании каждого символа вычисляется текущий диапазон (range = right - left + 1). На основе частот символов (freqs) и текущего символа определяются новые левая (newLeft) и правая (newRight) границы интервала. left и right обновляются до этих новых значений. Если старшие биты left и right равны, происходит вывод этого бита. Это означает, что определенная часть диапазона уже закодирована. Если старшие биты различаются, но left и right пересекаются на уровне quarter, активируется механизм под названием "scaling"для предотвращения "схлопывания"интервала. Далее проверяем, равны ли старшие биты left и right. left ̂right вычисляет XOR между left и right, а & half проверяет, равен ли старший бит нулю. Если условие истинно, значит старшие биты left и right одинаковы, и можно осуществить нормализацию. Затем извлекается старший бит из left, который будет записан в выходной поток. left сдвигается вправо

на sumRange - 1 битов (31 бит, если sumRange равно 32), что оставляет только старший бит. Извлеченный бит записывается в выходной поток. Если имеются какие-либо отложенные биты (leftBits), они записываются в выходной поток. Биты, которые записываются, являются дополнением (XOR 1) к записанному ранее биту. left сдвигается на один бит влево, и применяется маска (mask) для ограничения его размера. Аналогично, right сдвигается на один бит влево, применяется маска и затем устанавливается младший бит в 1, чтобы сохранить интервал. Если left и right пересекаются на уровне quarter, то происходит сдвиг обоих границ влево и инкремент leftBits. Это предотвращает слишком сильное сближение границ и обеспечивает достаточное разрешение для последующих символов. Далее в цикле увеличиваем счетчик отложенных битов. left сдвигается на один бит влево, а затем применяется XOR с half, чтобы инвертировать его старший бит. Это помогает расширить интервал кодирования. Аналогично, старший бит right инвертируется, затем значение сдвигается на один бит влево и добавляется half | 1, чтобы гарантировать, что right остается больше left.

При расжатии создается объект freqs класса Frequency, хранящий частоты символов. Читаются первые sumRange бит из входного потока и формируется начальное значение code. Это значение будет использоваться для определения, какой символ был закодирован. Далее на каждой итерации определяется текущий диапазон (range = right - left + 1). Вычисляется shift, который представляет собой разницу между code и left. Вычисляется target, используя shift, который помогает определить, какой символ был закодирован. Применяется бинарный поиск для определения закодированного символа. l и r используются как границы поиска, начиная с 0 до 257. В цикле, пока r - l $>$ 1, находится середина (mid) и сравнивается с target. Это помогает сузить диапазон поиска до тех пор, пока не будет найден правильный символ. Определяются новые границы интервала (newLeft и newRight) на основе найденного символа. Если старшие биты left и right равны, читается следующий бит и происходит обновление code, left, и right. Если границы интервала пересекаются на уровне quarter, то происходит чтение следующего бита, и выполняется "scaling"для code, left, и right. Если найден специальный терминальный символ (256), цикл декодирования завершается. Иначе, полученный символ преобразуется в символ и записывается в выходной поток. Частота этого символа увеличивается в freqs. Этот процесс продолжается до тех пор, пока не будет достигнут терминальный символ, сигнализирующий о конце закодированных данных.

Реализация алгоритма LZW

Конструктор LZW инициализирует объект Compressor с параметрами, такими как имена входного и выходного файлов, режим работы (сжатие или расжатие), флаги ввода и вывода в консоль, а также значение compressValue, которое может использоваться для управления процессом сжатия и вызывает InitDict(), который инициализирует словарь для сжатия или расжатия.

Метод Compress (Сжатие) читает символы из входного потока и строит строку tmpStr, используя словарь compressDict, который отображает строки на коды. Когда строка tmpStr больше не соответствует ни одному ключу в словаре, код для tmpStr записывается в выходной поток, а tmpStr + symbol добавляется в словарь. Если достигнут

предел размера словаря, производится его сброс и повторная инициализация.

Метод Uncompress (Расжатие) читает биты из сжатого файла и преобразует их в коды, используя словарь uncompressDict, который содержит соответствия между кодами и строками. Специальный код 256 может использоваться для сброса словаря и его повторной инициализации. Если достигнут конец файла (код 257), процесс расжатия завершается.

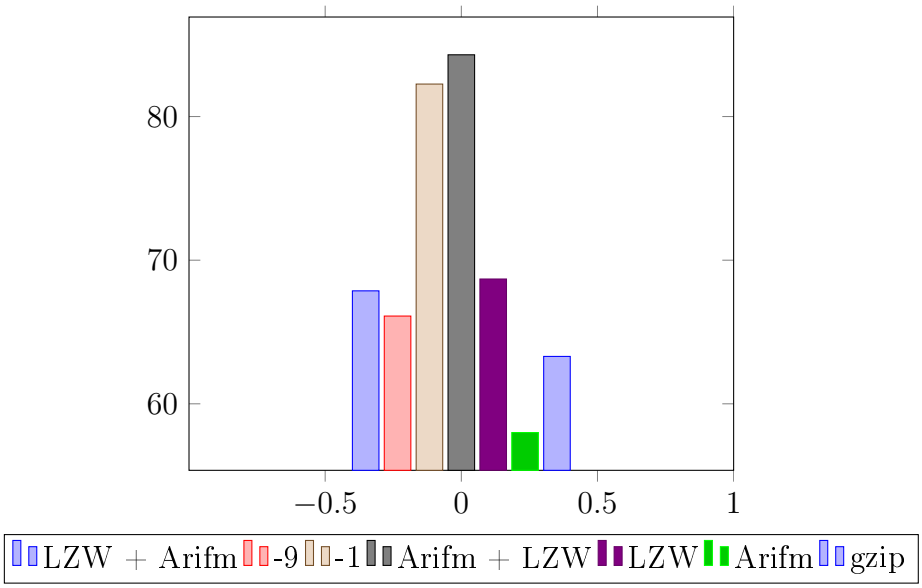
Тесты

Будет проведен ряд тестов на сравнение разных методов сжатия. Будет 2 серии тестов: на спам-тексте и на книге в формате txt. Оценивать будем ratio = (<размер сжатого> / <размер сжатого>) * 100. Чем ментше ratio, тем лучше сжат файл.

Сравниваемые алгоритмы:

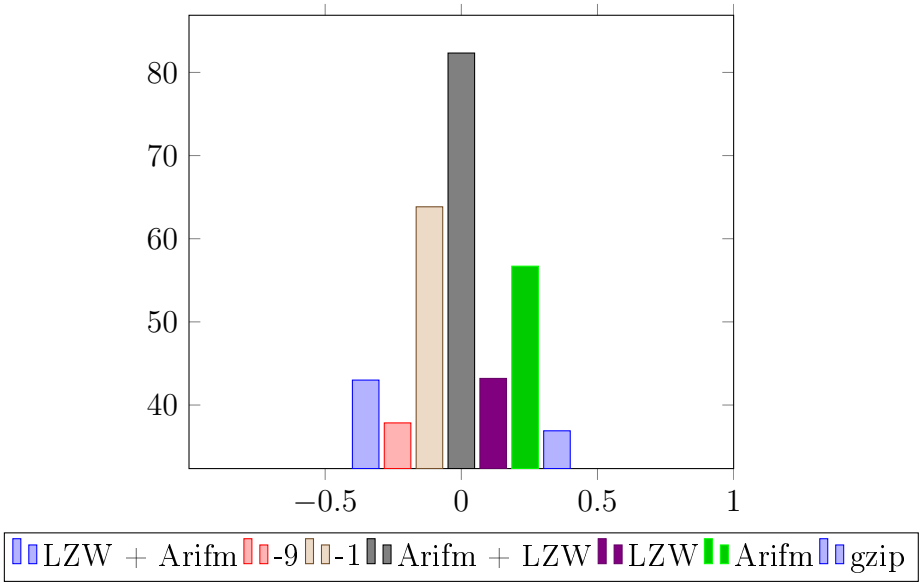
1. LZW + Arifm
2. LZW + Arifm с флагом -1
3. LZW + Arifm с флагом -9
4. Arifm + LZW
5. LZW
6. Arifm
7. gzip

Тест 1. Сжатие спам-текста размером 1 МБ



Результаты первого теста показали, что LZW после арифметического сжатия абсолютно неэффективен, а арифметическое сжатие после LZW хоть и улучшает коэффициент сжатия, но не сильно. Арифметическое сжатие в чистом виде имеет коэффициент лучше чем у утилиты gzip. LZW на спам-тексте тоже хорошо себя показал.

Тест 2. Сжатие книги в формате txt размером 1 КБ



На реальной книге LZW показал себя лучше, чем арифметическое кодирование, скорее всего это связано с повторяющимися словами в книге. Композитный вариант Arifm + LZW снова выдал худший результат, а LZW + Arifm с флагом -9 имеет почти такой же коэффициент, как у утилиты gzip.

Выводы

В ходе выполнения данного курсового проекта я реализовал нечто похожее на утилиту gzip, применив алгоритмы сжатия текстов, а также средства для работы с файлами.

Алгоритмы LZW и арифметическое кодирование хорошо себя показали, сжимая текст практически в 2 раза. LZW лучше применим к реальным текстам, так как в них большое количество повторений, в то время как арифметическое сжатие лучше на рандомных однородных данных.

Композитный алгоритм Arifm + LZW практически неприменим, так как LZW на предварительно обработанных арифметическим кодированием данных лишь расширяет текст, а не сжимает.

Композитный алгоритм LZW + Arifm достаточно эффективен, однако вклад арифметического сжатия в нем незначителен, хотя он и есть.

Таким образом можно сделать вывод о том, что комбинация арифметического сжатия и LZW не является самой эффективной.