Оглавление

[**Глава** **1.** **Алгоритмы** **сжатия** **данных** 3](#_Toc105156724)

[1.1 Алгоритм RLE 3](#_Toc105156725)

[1.2 Арифметический метод 4](#_Toc105156726)

[1.3 Алгоритм LZW 6](#_Toc105156727)

[1.4 Сравнение алгоритмов 8](#_Toc105156728)

[**Глава** **2. Практическая реализация алгоритмов** 9](#_Toc105156729)

[2.1 Базовый класс 9](#_Toc105156730)

[2.2 Классы-наследники 10](#_Toc105156731)

[2.3 Алгоритм кодирования RLE 10](#_Toc105156732)

[2.4 Алгоритм декодирования RLE 11](#_Toc105156733)

[2.5 Алгоритм арифметического кодирования 11](#_Toc105156734)

[2.6 Алгоритм арифметического декодирования 12](#_Toc105156735)

[2.7 Алгоритм кодирования LZW 12](#_Toc105156736)

[2.8 Алгоритм декодирования LZW 13](#_Toc105156737)

[2.9 Файл статистики “stats.txt” 13](#_Toc105156738)

[2.10 Функция showStats() 14](#_Toc105156739)

[2.11 Основное меню 14](#_Toc105156740)

[2.12 Пробные запуски 16](#_Toc105156741)

[**Глава** **3. Безопасность кода** 19](#_Toc105156742)

[3.1. Распределение и освобождение памяти 19](#_Toc105156743)

[3.2. Защита от дурака 19](#_Toc105156744)

[3.3. Контроль доступа к файлам 20](#_Toc105156745)

[3.4 Защита от переполнения буфера 20](#_Toc105156746)

[**Заключение** 21](#_Toc105156747)

[**Список** **литературы** 22](#_Toc105156748)

[**Приложения** 23](#_Toc105156749)

Глава 1. Алгоритмы сжатия данных

Сжатие данных – преобразование данных, производимое с целью уменьшения занимаемого ими объёма. В настоящее время существует большое количество алгоритмов, позволяющих производить успешное сжатие данных для более рационального использования устройств хранения информации. Существует два типа сжатия данных: сжатие без потерь и с потерями. При сжатии данных без потерь алгоритмы сохраняют и преобразуют абсолютно всю информацию, которая хранилась в исходном файле. В то же время при сжатии данных с потерями алгоритмы допускают удаление ненужных данных, что приводит к потере части информации при декодировании файла. Среди самых известных алгоритмов сжатия без потерь часто выделяют RLE, Арифметический метод и алгоритмы LZ.

1.1 Алгоритм RLE

Алгоритм кодирования длин серий (Run-Length Encoding) – алгоритм сжатия данных, который основан на принципе замены последовательностей повторяющихся символов на один символ и число его повторов.

Алгоритм считает количество подряд идущих символов и записывает его и один символ вместо серии одинаковых символов. Например, в строке вида “AAABBBBBCC” в процессе преобразования будет подсчитано 3 символа ‘A’, 5 символов ‘B’ и 2 символа ‘C’, в результате чего получится строка “3A5B2C”, а объём файла уменьшится с 10 байт до 6 байт.

Данный способ является достаточно эффективным для сжатия растровых графических изображений (BMP, PCX, TIFF), так как они содержат длинные серии повторяющихся последовательностей данных. Также очевидным достоинством алгоритма является сжатие данных без потерь и простота его декодирования: для того, чтобы декодировать файл, не нужны дополнительные данные; все данные для успешного декодирования хранятся в самом закодированном файле.

Однако алгоритм показывает худшие результаты, когда сталкивается со строками, в которых много неповторяющихся символов. Например, кодируя строку “ABCD”, алгоритм выведет строку “1A1B1C1D”, а объём файла не уменьшится, а лишь увеличится с 4 байт до 8 байт. В лучшем случае при кодировании строк с достаточно большим количеством неповторяющихся символов длина выходной строки не изменится, поэтому для подобных строк алгоритм мало эффективен и не будет выполнять функцию сжатия данных, поэтому алгоритм не может считаться универсальным.

1.2 Арифметический метод

Арифметический метод кодирования – метод сжатия данных без потерь, основанный на частотности использования символов.

Для того, чтобы закодировать данные, этот алгоритм создает алфавит, состоящий из символов входного файла. Также он вычисляет частотность использования в файле каждого символа. Далее берётся отрезок на координатной прямой, где последовательно от 0 до 1 откладываются частотности каждого символа. В результате выходит последовательность отрезков, каждый из которых соответствует одному символу из образованного алфавита. Теперь необходимо поочерёдно брать из потока символы, находить соответствующие им отрезки и делить их в том же отношении, что и исходный отрезок от 0 до 1. В итоге будет получен определенный отрезок, любое значение в котором является результатом выполнения данного алгоритма. Например, кодируя строку “LEVEL”, алгоритм создаст алфавит из символов ‘L’, ‘E’, ‘V’ с соответствующими частотностями.

Таблица 1. Алфавит и частотность символов, составленные из строки “LEVEL”

|  |  |  |
| --- | --- | --- |
| Символ | Частотность | Отрезок |
| L | 0.4 | 0 – 0.4 |
| E | 0.4 | 0.4 – 0.8 |
| V | 0.2 | 0.8 – 1 |

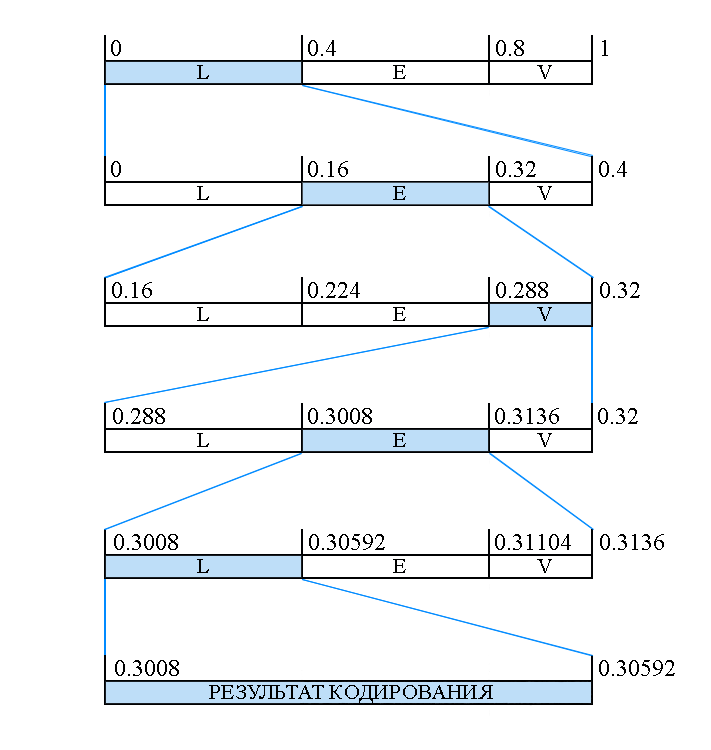
Далее строка “LEVEL” считывается посимвольно. Для первого символа ‘L’ берется соответствующий отрезок [0; 0.4] и делится в том же отношении, что и исходный отрезок. После этого берется следующий символ ‘E’ и соответствующий ему отрезок [0.16; 0.32] и делится в том же исходном отношении.

Рисунок 1. Процесс кодирования арифметическим методом строки “LEVEL”

Данные операции продолжаются до конца входной строки. После проделанной работы будет получен отрезок [0,3008; 0,30592], любое число из которого является результатом сжатия файла. Для декодирования используются исходные отрезки частотностей, с помощью которых определяются закодированные символы. Находится отрезок, которому принадлежит данное число (закодированное сообщение), выводится соответствующий символ, а затем это число преобразуется с учетом параметров данного отрезка для нахождения следующего символа.

Основная идея данного метода заключается в том, чтобы присваивать коды не отдельным символам, а их последовательностям. Метод показывает высокую эффективность для дробных неравномерных интервалов распределения вероятностей кодируемых символов.

Но данный алгоритм имеет определенные недостатки и сложности в реализации. Во-первых, для декодирования необходимо знать данные о символах и об отрезках частотностей, а также длину исходной строки, чтобы декодирование завершилось в нужном месте. Во-вторых, процесс декодирования осложнён несовершенством представления чисел с плавающей точкой в памяти компьютера; поскольку некоторые дробные числа не могут быть точно представлены в двоичном коде, то границы символов будут округлены, что может повлечь за собой неверную работу алгоритма при больших объёмах данных.

1.3 Алгоритм LZW

Алгоритмы семейства LZ – это алгоритмы сжатия данных, основанные на создании и работе со словарями, состоящими из символов и строк. Существует большое количество модификаций оригинальных алгоритмов LZ77 и LZ78 (LZW, LZO, LZS, LZMA и другие).

Алгоритм Лемпеля-Зивы-Уэлча (Lempel-Ziv-Welch) – словарный алгоритм сжатия данных без потерь, созданный Авраамом Лемпелем, Яаковом Зивом и модифицированный Терри Велчем.

Данный алгоритм создает словарь из определенных подстрок и присуждает каждой порядковый номер. Сначала словарь заполняется алфавитом символов из файла, после чего начинают считываться поочерёдно символы из потока. Если текущий символ уже есть в словаре, то к нему прибавляется следующий. То есть, если данная подстрока уже существует в словаре, то к этой подстроке добавляется следующий символ. Если подстрока не существует в словаре, то она добавляется в него, к выходной строке добавляется индекс предыдущей найденной подстроки, а в текущей подстроке остается лишь последний считанный символ. Работу алгоритма можно рассмотреть на примере кодирования строки “ANANAS”. После заполнения словаря символами ‘A’, ‘N’ и ‘S’ алгоритм начинает свою работу с первого символа ‘A’. Так как ‘A’ уже существует в словаре, то берется следующий символ. Новое сочетание “AN” отсутствует в словаре, поэтому оно добавляется в словарь под новым индексом, на выход подается индекс символа ‘A’ и в текущей строке остается символ ‘N’. Далее аналогичная ситуация происходит с подстрокой “NA”: она добавляется в словарь, на выход подается индекс символа ‘N’, в текущей строке остается символ ‘A’. После этого считывается подстрока “AN”, которая уже существует в словаре, поэтому в словарь добавляется уже подстрока “ANA”, на выход подается индекс подстроки “AN”, а в текущей строке остается символ ‘A’. Таким же образом кодируются все оставшиеся данные в файле и в результате выполнения алгоритма будет получена строка “01302”. Так как результатом является строка, состоящая из индексов подстрок в словаре, то процесс декодирования заключается в обращении к словарю по индексам и выводе соответствующих подстрок.

Таблица 2. Данные словаря, составленного для кодирования строки “ANANAS”

|  |  |
| --- | --- |
| Индекс | Подстрока |
| 0 | A |
| 1 | N |
| 2 | S |
| 3 | AN |
| 4 | NA |
| 5 | ANA |
| 6 | AS |

Этот метод позволяет достичь одну из наилучших степеней сжатия среди других методов сжатия данных при полном отсутствии потерь или искажений в исходных файлах. В настоящее время он используется для сжатия данных в графических файлах (TIFF, PDF, GIF, PostScript и других), а также во многих программах сжатия файлов (ZIP, ARJ, LHA).

Среди недостатков данного метода отмечается необходимость передавать все данные о созданном при кодировании словаре во избежание потери информации при декодировании файла.

1.4 Сравнение алгоритмов

При работе с данными алгоритмами важно понимать, какой алгоритм целесообразнее использовать в различных ситуациях. Почти у каждого из них есть весомые недостатки, описанные выше и из-за которых их повсеместное использование недопустимо.

Алгоритм RLE подразумевает в первую очередь работу с большими файлами, содержащими достаточно большое количество подряд идущих повторяющихся символов, поэтому при кодировании данным методом обычного текста с большим количеством одинарных символов алгоритм будет лишь увеличивать вес файла.

Арифметический метод может работать как с файлами с повторяющимися элементами, так и с неповторяющимися. Его эффективность в программной реализации может быть достаточно низка, потому что сам алгоритм основан на работе с числами с плавающей точкой, но их хранение в памяти компьютера несовершенно, поэтому всегда велик риск ошибки работы такого алгоритма.

Алгоритм LZW показывает лучшие результаты относительно двух других алгоритмов. Он способен эффективно работать и с повторяющимися элементами, и с неповторяющимися. Также он основан на работе со словарями, что исключает возможность потери данных при декодировании, именно поэтому данный алгоритм выполняет свою задачу эффективнее, чем два других алгоритма.

Глава 2. Практическая реализация алгоритмов

2.1 Базовый класс

В базовом абстрактном классе BaseCompress представлены все необходимые конструкторы чисто виртуальные функции, которые переопределяются в производных классах RLE, Arithmetic и LZW. В списке членов класса BaseCompress находятся целочисленные переменные len и capacity, а также символьный указатель ptr. Переменная ptr является указателем на память, которая выделяется под динамический массив символов, с которым будут работать все последующие алгоритмы сжатия; переменная capacity отвечает за размер динамического массива ptr, а len – за длину. В методе public представлены основные конструкторы, деструктор и чисто виртуальные функции.

* Конструктор по умолчанию имеет заданный параметр типа int Capacity, равный 256. Конструктор выделяет память под динамический массив с размером Capacity и задаёт длину массива, равную 0.
* Конструктор, принимающий в качестве параметра переменную типа string, позволяет пользователю инициализировать переменные из строк и работать с ними.
* Деструктор освобождает память, выделенную под динамический массив, а также обнуляет переменную len.
* Чисто виртуальная функция compress типа string переопределяется во всех производных классов для реализации соответствующих алгоритмов кодирования.
* Чисто виртуальная функция decompress типа string переопределяется во всех производных классов для реализации соответствующих алгоритмов декодирования.
* Чисто виртуальная функция showStats типа void переопределяется во всех производных классов для вывода соответствующих данных о кодировании.

2.2 Классы-наследники

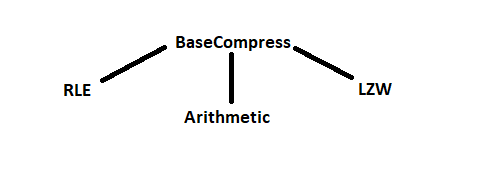
В работе представлено 4 класса: BaseCompress, RLE, Arithmetic и LZW. Класс BaseCompress является базовым классом, а классы RLE, Arithmetic и LZW являются его наследниками. В данных классах наследуются все конструкторы базового класса BaseCompress, а также переопределяются виртуальные функции. В каждом из классов-наследников в соответствии с алгоритмом переопределены виртуальные функции базового класса: compress, decompress и showStats.

Рисунок 2. Схема наследования

2.3 Алгоритм кодирования RLE

В классе RLE представлена функция compress, кодирующая данные с помощью алгоритма кодирования RLE.

Функция перебирает массив и, если текущий символ равен следующему, то увеличивает значение переменной-счётчика. Когда цепочка повторов символа прерывается, в массив строк, созданный в данной функции, записывается количество повторов символа и сам символ, после чего счётчик сбрасывается и функция продолжает считать следующие элементы. Так как происходит сравнение со следующим элементом, то сравнение последних элементов происходит отдельно после перебора.

После всех вычислений все данные кодировки сохраняются в отдельный файл (“stats.txt”), а также возвращаются в виде строки.

2.4 Алгоритм декодирования RLE

В классе RLE представлена функция decompress, декодирующая данные с помощью алгоритма декодирования RLE.

Алгоритм посимвольно считывает файл и проверяет каждый символ на то, является ли он цифрой. Цифры сохраняются в переменную-счётчик типа int. Если идут несколько цифр подряд, то переменная-счётчик увеличивает количество разрядов и прибавляются цифры. Далее считывается следующий символ, не являющийся числом, и заносится в массив символов такое количество раз, которое указано в переменной-счётчике. После этого переменная-счётчик сбрасывается и алгоритм повторяется.

Результат вычислений возвращается в виде строки.

2.5 Алгоритм арифметического кодирования

В классе Arithmetic представлена функция compress, кодирующая данные с помощью арифметического кодирования.

В массив строк заносится алфавит, содержащийся во входящей строке. После этого считается количество повторов каждого символа во всей входящей строке и в массив типа double заносятся частотности каждого символа – количество повторов каждого символа делится на общее количество элементов входящей строки (len). Частотности преобразуются в отрезки суммарной длиной от 0 до 1: для этого каждый элемент массива частотностей складывается с предыдущим (кроме первого). Для сохранения данных в прежнем виде создается копия массива отрезков, в которой будут храниться модифицированные отрезки. Далее с помощью полученных данных кодируются символы по следующему алгоритму: берется символ из входящей строки и находится соответствующий ему отрезок из массива отрезков; вычисляется длина этого отрезка и с таким коэффициентом модифицируется копия-массив; считывается следующий символ и весь алгоритм повторяется до конца входящей строки. Получив в результате две границы отрезка, в котором содержатся все возможные результаты кодирования, берётся среднее арифметическое. Из-за того, что тип данных double хранится и выводится по-разному, то результат и все необходимые данные перед сохранением преобразуются в целочисленный тип long long. Но алгоритм всё равно не сможет работать корректно при большом объёме данных, поэтому перед кодированием программа предупредит пользователя о том, что кодирование строки, имеющей более 10 символов, может быть произведено некорректно.

После всех вычислений все данные кодировки сохраняются в отдельный файл (“stats.txt”), а также возвращаются в виде строки.

2.6 Алгоритм арифметического декодирования

В классе Arithmetic представлена функция decompress, декодирующая данные с помощью арифметического декодирования.

Сначала считываются и заносятся в соответствующие массивы данные из файла “stats.txt” (символы, числа, длина исходной строки). Так как числовые данные были сохранены в файле в целочисленном виде, то необходимо при их занесении в массив отрезков преобразовать их обратно в числа с плавающей точкой. После этого берётся результат кодирования и проверяется, в каком отрезке он находится. Далее из этого числа вычитается левая граница отрезка и все это делится на длину этого отрезка. Затем такой алгоритм повторяется нужное количество раз (длина исходной строки).

Результат вычислений возвращается в виде строки.

2.7 Алгоритм кодирования LZW

В классе LZW представлена функция compress, кодирующая данные с помощью алгоритма кодирования LZW.

В массив строк заносится алфавит, содержащийся во входящей строке. Далее массив заполняется по следующему алгоритму: массив строк проверяется на наличие текущей подстроки; если такая строка присутствует, то прибавляется следующий символ и алгоритм повторяется; если такая строка отсутствует, то она добавляется в массив строк, а в текущей строке остается лишь последний считанный символ. После заполнения словаря в целочисленный массив необходимо добавить индексы нужных для успешного кодирования подстрок. Для этого все элементы словаря сравниваются друг с другом на предмет отличия на один последний символ. Если такое условие соблюдается, то массив чисел принимает индекс и продолжает работать по алгоритму.

После всех вычислений все данные кодировки сохраняются в отдельный файл (“stats.txt”), а также возвращаются в виде строки.

2.8 Алгоритм декодирования LZW

В классе Decompress представлена функция декодирования для алгоритма LZW.

Сначала считываются и заносятся в соответствующие массивы данные из файла “stats.txt” (словарь, индексы). Далее по массиву индексов в выходную строку добавляются соответствующие элементы словаря.

Результат вычислений возвращается в виде строки.

2.9 Файл статистики “stats.txt”

В файле “stats.txt” хранятся все данные для успешного декодирования файлов (для арифметического метода и алгоритма LZW), а также для вывода статистики по кодированию. Файл “stats.txt” почти всегда имеет одну и ту же структуру (отличается лишь только при кодировании арифметическим методом): сначала в первой строке указывается код алгоритма кодирования; на следующей строке через пробел указываются все символьные данные кодирования; на последней строке указываются через пробел все числовые данные. При кодировании арифметическим методом вторая строка в файле статистики занята длиной исходного файла (чтобы декодирование прошло успешно и могло завершиться в нужный момент). Код алгоритма кодирования – это число от 1 до 3, которое позволяет функции вывода статистики showStats() корректно вывести данные о кодировании, так как разные методы кодирования по-разному сохраняют данные. Также арифметический метод использует в своей основе числа с плавающей точкой типа double, но такие числа хранятся и выводятся в консоль или файл совершенно по-разному. Так как необходимая часть дробного числа не будет выводиться в консоль или файл, то для более корректной работы такие числовые данные преобразуются в целые числа и таким образом хранятся в файле “stats.txt”.

2.10 Функция showStats()

Функция showStats() типа void считывает данные из файла “stats.txt” и выводит их в консоль. Сначала алгоритм считывает первый символ – код алгоритма кодирования. После этого считываются все данные и сохраняются в соответствующие массивы и переменные. Все данные выводятся в виде таблиц кодов: при алгоритме RLE выводятся столбцы с символами и их повторами; при арифметическом кодировании выводятся столбцы с символами, частотностью и отрезками; при алгоритме LWZ выводятся столбцы с необходимыми индексами, новыми занесёнными подстроками в словарь и текущими подстроками. Если первый символ не является числом от 1 до 3, то функция не будет работать корректно и выдаст предупреждение об ошибке.

2.11 Основное меню

Предварительно пользователю необходимо создать в папке проекта текстовый файл и ввести в него данные, которые необходимо закодировать. При старте работы программы пользователя просят ввести имя текстового файла, хранящегося в папке проекта и содержащего текст для сжатия. В случае, если название указано неверно или такого файла не существует, то программа выведет соответствующую ошибку и завершит работу. Программа выводит и сохраняет содержимое файла, после чего просит пользователя выбрать метод, которым данная строка будет закодирована. Пользователь должен ввести число от 1 до 3: 1 – кодирование с помощью RLE; 2 – арифметическое кодирование; 3 – кодирование с помощью LZW. Если пользователь вводит другие данные, то программа снова попросит его выбрать один из предлагаемых вариантов. При арифметическом кодировании строки, количество символов которой превышает 10, программа предупредит пользователя о возможной потере данных. После выбора алгоритма программа закодирует его, продемонстрирует результат в консоли и сохранит в файл “compressed.txt”. Далее программа предложит пользователю ввести число от 1 до 4: 1 – показать статистику, собранную во время кодирования; 2 – декодировать результат программы и сохранить его в файл “decompressed.txt”; 3 – вернуться к кодированию; 4 – выйти из программы. Программа будет выполнять указанные пользователем задачи и снова предлагать выбор до тех пор, пока пользователь не решит выйти из программы. Если пользователь будет вводить посторонние данные, то программа будет всегда просить его снова ввести их корректно.

2.12 Пробные запуски

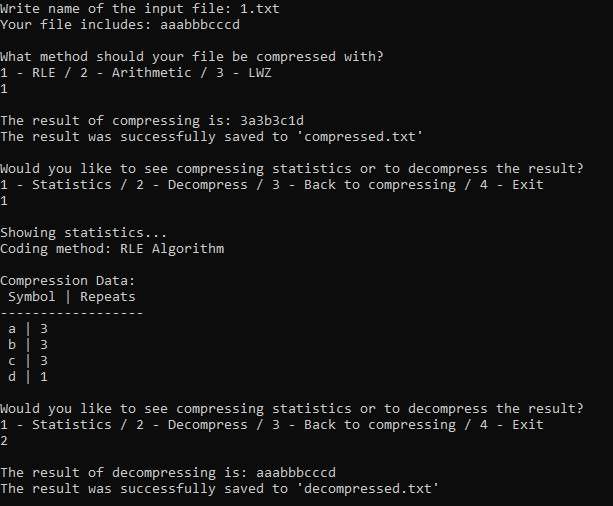
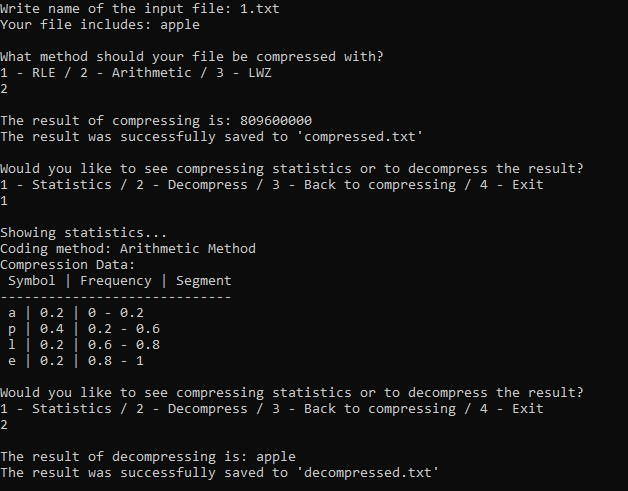
Ниже представлены примеры запуска программы с разными входными данными и возможными выборами пользователя.

Рисунок 3. Демонстрация работы алгоритма RLE

Рисунок 4. Демонстрация работы арифметического кодирования

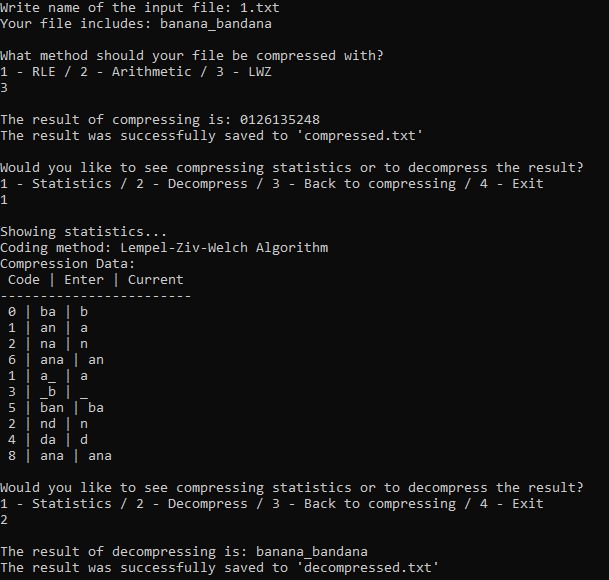


Рисунок 5. Демонстрация работы алгоритма LZW

Глава 3. Безопасность кода

Безопасность кода – один из важных аспектов любой программы. Безопасный код позволяет программе работать без ошибок и появления непреднамеренных результатов. В данном случае безопасность кода была достигнута путём правильного распределения и освобождения памяти, защитой от дурака, контролем доступа к необходимым текстовым файлам и защитой от переполнения буфера.

3.1. Распределение и освобождение памяти

Для работы с большим количеством символов и числовых данных в методах классов используются динамические массивы. На каждый из них выделяется память в необходимом количестве. Во избежание возможных ошибок и проблем с памятью в конце каждого метода освобождается память всех ранее использованных в данном методе динамических массивов, а также всем указателям присуждается значение NULL. Аналогично в деструкторе базового класса освобождается память, выделенная под массив символов ptr.

3.2. Защита от дурака

Так как пользователь может ввести неверные данные, программа всё равно должна продолжать работать верно и не должна аварийно завершать свою работу. Поэтому, чтобы программа продолжала верно работать и выполнять свои функции, пользователю предлагается ограниченный выбор возможностей. Если пользователь введет неверные данные или случайно укажет неверную команду, то программа предупредит пользователя об отсутствии такой возможности и снова попросит ввести команду корректно, выводя при этом все доступные пользователю команды. Исключение – ввод названия файла. Если указано неверное название файла или он отсутствует, то программа выведет ошибку и завершит свою работу.

3.3. Контроль доступа к файлам

Во время работы программы файлы “stats.txt”, “compressed.txt” и “decompressed.txt” могут быть намеренно или случайно повреждены пользователем. Чтобы программа не завершалась аварийно при невозможности их прочтения, доступ к чтению данных файлов проверяется при каждом обращении к ним. В случае, если программа не может получить к ним доступ, то она предупредит пользователя о невозможности прочтения файла и об их возможном повреждении, выводя соответствующую ошибку, и завершит свою работу.

3.4 Защита от переполнения буфера

Выделенной памяти под массивы для чтения файлов может не хватать, поэтому все данные файла, которые при его чтении вышли за пределы выделенной памяти, могут стать причиной переполнения буфера, что делает программу уязвимой. Чтобы такого не произошло, необходимо всегда проверять границы массивов и возможность сохранения в них еще одного символа. В программе реализована проверка возможности добавления нового символа, а также реализовано чтение с помощью строк.

Заключение

В процессе работы были изучены три алгоритма сжатия данных: RLE, арифметический метод, LZW, а также семейство алгоритмов LZ. Были рассмотрены преимущества и недостатки каждого алгоритма, использованного в данной работе. После сравнения работы всех алгоритмов алгоритм LZW был отмечен как самый эффективный среди представленных. Были реализованы оригинальные алгоритмы, поэтому существует возможность их модификации и усовершенствования. В работе присутствуют как алгоритмы кодирования, так и алгоритмы их декодирования. Также была введена возможность просмотра пользователем статистики кодирования разными методами. Реализована возможность хранить все полученные данные в разных текстовых файлах. Была сделана попытка улучшения работы одного из алгоритмов для более корректной работы. Присутствует понятный пользователю интерфейс, позволяющий проводить все доступные операции с имеющимися данными. Для более комфортного пользования программой были реализованы разные методы обеспечения безопасности кода.

Список литературы

1. Ватолин Д., Ратушняк А., Смирнов М., Юкин В. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – М. : ДИАЛОГ-МИФИ, 2003. — 384 с.
2. Пантелеев Е. Р. Алгоритмы сжатия данных без потерь: учебное пособие для вузов / Е. Р. Пантелеев, А. Л. Алыкова. – СПб. : Лань, 2021. — 172 с.
3. Аграновский А. В, Хади Р. А. Практическая криптография: алгоритмы и их программирование / А. В. Аграновский, Р. А. Хади. – М. : СОЛОН-Пресс, 2002. — 254 с.
4. Тропченко А. Ю., Тропченко А. А. Методы сжатия изображений, аудиосигналов и видео. Учебное пособие по дисциплине «Теоретическая информатика» / А. Ю. Тропченко, А. А. Тропченко – СПб. : СПбГУ ИТМО, 2009. — 108 с.
5. Шилдт Г. Самоучитель C++ / Г. Шилдт ; пер. с англ. А. Жданов – М. : БХВ-Петербург, 2003. — 688 c.

Приложения

Приложение 1

#include <iostream>

#include <string>

#include <cstdlib>

#include <fstream>

#define LONG 10000000000

**using** **namespace** std;

**class** BaseCompress

{

**protected**:

**char**\* ptr;

**int** len;

**int** capacity;

**public**:

BaseCompress(**int** Capacity = 256)

{

len = 0;

capacity = Capacity;

ptr = **new** **char**[capacity];

}

BaseCompress(string s)

{

len = s.length();

capacity = len >= 256 ? len + 1 : 256;

ptr = **new** **char**[capacity];

**for** (**int** i = 0; i < len; i++)

{

ptr[i] = s[i];

}

ptr[len] = '\0';

}

~BaseCompress()

{

**if** (ptr != NULL)

**delete**[] ptr;

len = 0;

}

**virtual** string compress() = 0;

**virtual** string decompress() = 0;

**virtual** **void** showStats() = 0;

};

**class** RLE : **public** BaseCompress

{

**public**:

RLE(**int** Capacity = 256) : BaseCompress(Capacity) {}

RLE(string s) : BaseCompress(s) {}

~RLE() {}

string compress()

{

string\* dict = **new** string[capacity];

string output;

**int** count = 0, num = 1;

**for** (**int** i = 0; i < len - 1; i++)

{

Приложение 2

**if** (ptr[i] == ptr[i + 1])

{

num++;

}

**else**

{

dict[count] = to\_string(num);

count++;

dict[count] = ptr[i];

count++;

num = 1;

}

}

**if** (ptr[len - 2] == ptr[len - 1])

{

dict[count] = to\_string(count);

}

**else**

{

dict[count] = "1";

}

count++;

dict[count] = ptr[len - 1];

count++;

ofstream file("stats.txt");

file << "1\n";

**for** (**int** i = 0; i < count; i++)

{

**if** (i % 2 != 0)

{

file << dict[i] << " ";

}

output += dict[i];

}

file << '\n';

**for** (**int** i = 0; i < count; i++)

{

**if** (i % 2 == 0)

{

file << " " << dict[i];

}

}

file.close();

**delete**[] dict;

dict = NULL;

**return** output;

}

string decompress()

{

string output;

**int** count = 0;

**char** c;

**for** (**int** i = 0; i < len - 1; i++)

{

**if** (ptr[i] >= '0' && ptr[i] <= '9')

{

Приложение 3

**int** n = ptr[i] - '0';

**for** (**int** j = i + 1; j < len; j++)

{

**if** (ptr[j] >= '0' && ptr[j] <= '9')

{

n = n \* 10 + (ptr[j] - '0');

}

**else**

{

c = ptr[j];

**break**;

}

}

**for** (**int** j = 0; j < n; j++)

{

output += c;

}

count += n;

}

}

**return** output;

}

**void** showStats()

{

ifstream file("stats.txt");

{

**if** (file.is\_open())

{

**char** c;

file.get(c);

**if** (c == '1')

{

string line;

getline(file, line);

**char**\* sym = **new** **char**[capacity];

**int**\* num = **new** **int**[capacity];

**int** count\_int = 0, count\_str = 0;

**bool** flag = **false**;

line = "";

**while** (!file.eof())

{

**if** (count\_str < capacity - 1 && count\_int < capacity - 1)

{

getline(file, line, ' ');

**if** (line != "\n")

{

**if** (flag)

{

num[count\_int] = stoi(line);

count\_int++;

}

**else**

{

sym[count\_str] = line[0];

count\_str++;

}

Приложение 4

}

**else**

{

flag = **true**;

}

}

}

file.close();

cout << "\nCoding method: RLE Algorithm\n";

cout << "\nCompression Data: \n";

cout << " Symbol | Repeats " << endl;

cout << "------------------" << endl;

**for** (**int** i = 0; i < count\_str; i++)

{

cout << " " << sym[i] << " | " << num[i] << endl;

}

**delete**[] sym;

sym = NULL;

**delete**[] num;

num = NULL;

}

}

**else**

{

cout << "\nFile 'stats.txt' was corrupted!" << endl;

exit(1);

}

}

}

};

**class** Arithmetic : **public** BaseCompress

{

**public**:

Arithmetic(**int** Capacity = 256) : BaseCompress(Capacity) {}

Arithmetic(string s) : BaseCompress(s) {}

~Arithmetic() {}

string compress()

{

**double**\* temp = **new** **double**[capacity];

**double**\* period = **new** **double**[capacity];

**char**\* sym = **new** **char**[capacity];

**int** count = 0, len\_res = 0;

**double** left = 0, right = 1, line = 1, save = 0, left\_cpy = 0;

**bool** flag = **true**;

**if** (len > 10)

{

cout << "\nAttention: possible data loss!\n";

}

**for** (**int** i = 0; i < len; i++)

{

**for** (**int** j = 0; j < len\_res; j++)

{

**if** (ptr[i] == sym[j])

Приложение 5

{

flag = **false**;

**break**;

}

}

**if** (flag)

{

len\_res++;

sym[len\_res - 1] = ptr[i];

}

flag = **true**;

}

**for** (**int** i = 0; i < len\_res; i++)

{

**for** (**int** j = 0; j < len; j++)

{

**if** (sym[i] == ptr[j])

{

count++;

}

}

period[i] = ((**double**)count) / len + save;

save = period[i];

temp[i] = period[i];

count = 0;

}

**for** (**int** i = 0; i < len; i++)

{

**for** (**int** j = 0; j < len\_res; j++)

{

**if** (ptr[i] == sym[j])

{

**if** (j == 0)

{

left = left\_cpy;

right = temp[j];

}

**else**

{

left = temp[j - 1];

right = temp[j];

}

line = right - left;

left\_cpy = left;

**for** (**int** k = 0; k < len\_res; k++)

{

temp[k] = left + period[k] \* line;

}

}

}

}

ofstream file("stats.txt");

file << "2\n" << len << "\n";

**for** (**int** i = 0; i < len\_res; i++)

{

file << sym[i] << " ";

Приложение 6

}

file << '\n';

**for** (**int** i = 0; i < len\_res - 1; i++)

{

file << " " << (**long** **long**)(period[i] \* LONG);

}

file.close();

string output;

output = to\_string((**long** **long**)((left + right) / 2 \* LONG));

**delete**[] temp;

temp = NULL;

**delete**[] period;

period = NULL;

**delete**[] sym;

sym = NULL;

**return** output;

}

string decompress()

{

**char**\* sym = **new** **char**[capacity];

**double**\* period = **new** **double**[capacity];

**double** res = atof(ptr) / LONG;

string output, line;

**int** res\_count = 0, count\_doub = 0, count\_str = 0;

**char** c;

**bool** flag = **false**;

ifstream file("stats.txt");

**if** (file.is\_open())

{

getline(file, line); getline(file, line);

res\_count = stoi(line);

**while** (!file.eof())

{

**if** (count\_str < capacity - 1 && count\_doub < capacity - 1)

{

getline(file, line, ' ');

**if** (line != "\n")

{

**if** (flag)

{

period[count\_doub] = atof(line.c\_str()) / LONG;

count\_doub++;

}

**else**

{

sym[count\_str] = line[0];

count\_str++;

}

}

**else**

{

flag = **true**;

}

}

}

Приложение 7

file.close();

period[count\_doub] = 1;

count\_doub++;

}

**else**

{

cout << "File 'compressed.txt' was corrupted!" << endl;

exit(1);

}

**for** (**int** i = 0; i < res\_count; i++)

{

**for** (**int** j = 0; j < count\_str; j++)

{

**if** (res < period[j])

{

output += sym[j];

**if** (j == 0)

{

res /= period[j];

}

**else**

{

res = (res - period[j - 1]) / (period[j] - period[j - 1]);

}

**break**;

}

}

}

**delete**[] sym;

sym = NULL;

**delete**[] period;

period = NULL;

**return** output;

}

**void** showStats()

{

ifstream file("stats.txt");

**if** (file.is\_open())

{

**char** c;

file.get(c);

**if** (c == '2')

{

string line;

getline(file, line); getline(file, line);

**char**\* sym = **new** **char**[capacity];

**double**\* period = **new** **double**[capacity];

**int** count\_int = 0, count\_str = 0;

line = "";

**bool** flag = **false**;

**while** (!file.eof())

{

**if** (count\_str < capacity - 1 && count\_int < capacity - 1)

{

Приложение 8

getline(file, line, ' ');

**if** (line != "\n")

{

**if** (flag)

{

period[count\_int] = atof(line.c\_str()) / LONG;

count\_int++;

}

**else**

{

sym[count\_str] = line[0];

count\_str++;

}

}

**else**

{

flag = **true**;

}

}

}

period[count\_int] = 1;

count\_int++;

file.close();

cout << "\nCoding method: Arithmetic Method";

cout << "\nCompression Data: \n";

cout << " Symbol | Frequency | Segment" << endl;

cout << "-----------------------------" << endl;

**for** (**int** i = 0; i < count\_str; i++)

{

**if** (i == 0)

{

cout << " " << sym[i] << " | " << period[i] << " | " << "0" << " - " << period[i] << endl;

}

**else**

{

cout << " " << sym[i] << " | " << period[i] - period[i - 1] << " | " << period[i - 1] << " - " << period[i] << endl;

}

}

**delete**[] sym;

sym = NULL;

**delete**[] period;

period = NULL;

}

}

**else**

{

cout << "\nFile 'stats.txt' was corrupted!" << endl;

exit(1);

}

}

};

Приложение 9

**class** LZW : **public** BaseCompress

{

**public**:

LZW(**int** Capacity = 256) : BaseCompress(Capacity) {}

LZW(string s) : BaseCompress(s) {}

~LZW() {}

string compress()

{

**int** len\_res = 0, diff, count = 0;

string line, output;

**bool** flag = **true**;

**int**\* res = **new** **int**[capacity];

string\* dict = **new** string[capacity];

**for** (**int** i = 0; i < len; i++)

{

**for** (**int** j = 0; j < len\_res; j++)

{

**if** (ptr[i] == dict[j][0])

{

flag = **false**;

**break**;

}

}

**if** (flag)

{

len\_res++;

dict[len\_res - 1] = ptr[i];

}

flag = **true**;

}

diff = len\_res;

**for** (**int** i = 0; i < len; i++)

{

string temp;

temp += ptr[i];

**if** (i != len - 1)

{

line = temp + ptr[i + 1];

}

**else**

{

line = temp;

}

**for** (**int** j = 0; j < len\_res; j++)

{

**if** (i != len - 1 && line == dict[j])

{

i++;

**if** (i != len - 1)

{

line += ptr[i + 1];

}

j = 0;

}

}

dict[len\_res] = line;

len\_res++;

}

Приложение 10

**for** (**int** i = diff; i < len\_res; i++)

{

**for** (**int** j = 0; j < len\_res; j++)

{

**if** (i != len\_res - 1 && dict[j] == dict[i].substr(0, dict[i].length() - 1))

{

res[count] = j;

count++;

**break**;

}

**else** **if** (i == len\_res - 1 && dict[j] == dict[i])

{

res[count] = j;

**break**;

}

}

}

ofstream file("stats.txt");

file << "3\n";

**for** (**int** i = 0; i < len\_res; i++)

{

file << dict[i] << " ";

}

file << '\n';

**for** (**int** i = 0; i < len\_res - diff; i++)

{

file << " " << res[i];

output += to\_string(res[i]);

}

file.close();

**delete**[] res;

res = NULL;

**delete**[] dict;

dict = NULL;

**delete**[] dict;

dict = NULL;

**return** output;

}

string decompress()

{

string\* dict = **new** string[capacity];

**int**\* num = **new** **int**[capacity];

string output, line;

**int** count\_int = 0, count\_str = 0;

**char** c;

**bool** flag = **false**;

ifstream file("stats.txt");

**if** (file.is\_open())

{

file.get(c); file.get(c);

**while** (!file.eof())

{

**if** (count\_str < capacity - 1 && count\_int < capacity - 1)

{

getline(file, line, ' ');

Приложение 11

**if** (line != "\n")

{

**if** (flag)

{

num[count\_int] = stoi(line);

count\_int++;

}

**else**

{

dict[count\_str] = line;

count\_str++;

}

}

**else**

{

flag = **true**;

}

}

}

file.close();

}

**else**

{

cout << "File 'stats.txt' was corrupted!" << endl;

exit(1);

}

**for** (**int** i = 0; i < count\_int; i++)

{

output += dict[num[i]];

}

**delete**[] dict;

dict = NULL;

**delete**[] num;

num = NULL;

**return** output;

}

**void** showStats()

{

ifstream file("stats.txt");

**if** (file.is\_open())

{

**char** c;

file.get(c);

**if** (c == '3')

{

string line;

getline(file, line);

string\* dict = **new** string[capacity];

**int**\* num = **new** **int**[capacity];

line = "";

**int** count\_int = 0, count\_str = 0;

**bool** flag = **false**;

**while** (!file.eof())

{

**if** (count\_str < capacity - 1 && count\_int < capacity - 1)

{

Приложение 12

getline(file, line, ' ');

**if** (line != "\n")

{

**if** (flag)

{

num[count\_int] = stoi(line);

count\_int++;

}

**else**

{

dict[count\_str] = line;

count\_str++;

}

}

**else**

{

flag = **true**;

}

}

}

file.close();

cout << "\nCoding method: Lempel-Ziv-Welch Algorithm";

cout << "\nCompression Data: \n";

cout << " Code | Enter | Current " << endl;

cout << "------------------------" << endl;

**int** diff = count\_str - count\_int;

**for** (**int** i = diff; i < count\_str; i++)

{

**if** (i != count\_str - 1)

{

cout << " " << num[i - diff] << " | " << dict[i] << " | " << dict[i].substr(0, dict[i].length() - 1) << endl;

}

**else**

{

cout << " " << num[i - diff] << " | " << dict[i] << " | " << dict[i] << endl;

}

}

**delete**[] dict;

dict = NULL;

**delete**[] num;

num = NULL;

}

}

**else**

{

cout << "\nFile 'stats.txt' was corrupted!" << endl;

exit(1);

}

}

};

Приложение 13

**int** main()

{

string name;

cout << "Write name of the input file: ";

cin >> name;

ifstream file(name);

**if** (file.is\_open())

{

string line1, line2;

getline(file, line1);

file.close();

cout << "Your file includes: " << line1 << endl;

string result\_c, result\_d;

**int** key = 0;

**while** (key != 1 && key != 2 && key != 3)

{

cout << "\nWhat method should your file be compressed with?" << endl << "1 - RLE / 2 - Arithmetic / 3 - LWZ" << endl;

cin >> key;

**switch** (key)

{

**case** 1:

{

RLE comp(line1);

result\_c = comp.compress();

**break**;

}

**case** 2:

{

Arithmetic comp(line1);

result\_c = comp.compress();

**break**;

}

**case** 3:

{

LZW comp(line1);

result\_c = comp.compress();

**break**;

}

**default**:

**break**;

}

**if** (key == 1 || key == 2 || key == 3)

{

cout << endl << "The result of compressing is: " << result\_c << endl;

ofstream file1("compressed.txt");

file1 << result\_c;

file1.close();

cout << "The result was successfully saved to 'compressed.txt'" << endl;

**int** key1 = 0;

**while** (key1 != 3)

{

**bool** back = **false**;

cout << "\nWould you like to see compressing statistics or to decompress the result?" << endl << "1 - Statistics / 2 - Decompress / 3 - Back to compressing / 4 - Exit" << endl;

Приложение 14

cin >> key1;

**switch** (key1)

{

**case** 1:

{

cout << "\nShowing statistics...";

**switch** (key)

{

**case** 1:

{

RLE stat(line1);

stat.showStats();

**break**;

}

**case** 2:

{

Arithmetic stat(line1);

stat.showStats();

**break**;

}

**case** 3:

{

LZW stat(line1);

stat.showStats();

**break**;

}

**default**:

**break**;

}

**break**;

}

**case** 2:

{

ifstream file("compressed.txt");

**if** (file.is\_open())

{

getline(file, line2);

file.close();

**switch** (key)

{

**case** 1:

{

RLE decomp(line2);

result\_d = decomp.decompress();

**break**;

}

**case** 2:

{

Arithmetic decomp(line2);

result\_d = decomp.decompress();

**break**;

}

**case** 3:

{

LZW decomp(line2);

result\_d = decomp.decompress();

Приложение 15

**break**;

}

**default**:

cout << "There is no such variant, try again!" << endl;

**break**;

}

cout << endl << "The result of decompressing is: " << result\_d << endl;

ofstream file1("decompressed.txt");

file1 << result\_d;

file1.close();

cout << "The result was successfully saved to 'decompressed.txt'" << endl;

}

**else**

{

cout << "File 'compressed.txt' was corrupted!" << endl;

**return** 0;

}

**break**;

}

**case** 3:

back = **true**;

**break**;

**case** 4:

**return** 0;

**default**:

cout << "There is no such variant, try again!" << endl;

**break**;

}

**if** (back)

{

key = 0;

**break**;

}

}

}

}

}

**else**

{

cout << endl << "There is no file with name '" << name << "'!" << endl;

}

}