Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Дисциплина «Защита информации и надежность информационных систем»

**Отчёт по лабораторной работе №10**

Сжатие/распаковка данных методом Лемпеля-Зива

Студент: Жук С.С.

ФИТ 3 курс 2 группа

Преподаватель: Савельева М.Г.

**Содержание**

[1 Теоретические сведения 3](#_Toc196860213)

[2 Прямое и обратное преобразование текста из файла 4](#_Toc196860214)

[3 Скорость и эффективность выполнения сжатия/распаковки 9](#_Toc196860215)

[Вывод 10](#_Toc196860216)

# **1 Теоретические сведения**

Классический алгоритм Лемпеля-Зива – LZ77, названный так по году представления метода, формулируется следующим образом: «если в проанализированном (сжатом) ранее выходном потоке уже встречалась подобная последовательность байт, причем запись о ее длине и смещении от текущей позиции короче, чем сама эта последовательность, то в выходной файл записывается ссылка (смещение, длина), а не сама последовательность».

Известный метод сжатия RLE, который заключается в записи вместо последовательности одинаковых символов одного символа и их количества, является подклассом LZ77.

Суть метода LZ77 в следующем: упаковщик постоянно хранит некоторое количество последних обработанных символов в буфере. По мере обработки входного потока вновь поступившие символы попадают в конец буфера, сдвигая предшествующие символы и вытесняя самые старые. Размеры этого буфера, называемого также скользящим словарем, варьируются в разных реализациях систем сжатия. Скользящее окно имеет длину *n*, т. е. в него помещается *n* символов, и состоит из двух частей:

* последовательности длины *n*1 = *n* – *n*2 уже закодированных символов (словарь);
* упреждающего буфера (буфера предварительного просмотра) длиной *n*2 – буфера кодирования.

Нужно найти самое длинное совпадение между строкой буфера кодирования и всеми фразами словаря.

Эти фразы могут начинаться с любого символа, выходить за пределы словаря, вторгаясь в область буфера, но должны лежать в окне. Буфер не может сравниваться сам с собой. Длина совпадения не должна превышать размера буфера. Полученная в результате поиска фраза кодируется с помощью двух чисел:

1. смещения от начала буфера *p*;
2. длины соответствия, или совпадения *q*.

Ссылки (*p* и *q* − указатели) однозначно определяют фразу. Дополнительно в выходной поток записывается символ *s*, следующий за совпавшей строкой буфера.

Длина кодовой комбинации (триады – *p*, *q*, *s*) на каждом шаге определяется соотношением

(1.1)  
где *N* – мощность алфавита.

После каждого шага окно смещается на *q* + 1 символов вправо и осуществляется переход к новому циклу кодирования. Величина сдвига объясняется тем, что мы реально закодировали именно *q* + 1 символов: *q* – с помощью указателя и 1 − с помощью тривиального копирования.

Передача одного символа в явном виде (*s*) позволяет разрешить проблему обработки еще ни разу не встречавшихся символов, но существенно увеличивает размер сжатого блока.

# **2 Прямое и обратное преобразование текста из файла**

Для начала опишем функцию, которая добавляет символы из скользящего окна в буфер. Она принимает количество символов для добавления, текущий буфер и строку скользящего окна. Функция вычисляет минимальное значение между запрошенным количеством символов и длиной окна, чтобы избежать выхода за границы строки. Затем она добавляет эти символы в конец буфера и обновляет содержимое окна, удаляя добавленные символы. Код представлен в листинге 2.1.

|  |
| --- |
| const addSymbolsToBuffer = (numberOfSymbols, buffer, slidingWindow) => {  const symbolsToAdd = *Math*.min(numberOfSymbols, slidingWindow.length);  if (symbolsToAdd > 0) {  buffer += slidingWindow.slice(0, symbolsToAdd);  slidingWindow = slidingWindow.slice(symbolsToAdd);  }  return { buffer, slidingWindow }; }; |

Листинг 2.1 – Функция добавления символов в буфер

Далее опишем функцию, которая находит максимальное совпадение начала строки скользящего окна с подстрокой внутри буфера. Алгоритм перебирает начальные фрагменты окна и ищет их в буфере, фиксируя наибольшую длину совпадения и позицию первого вхождения. Когда большее совпадение найти не удаётся, возвращаются индекс совпадения (сдвинутый на 1), длина найденной последовательности и символ, следующий за совпадением. Программная реализация функции показана в листинге 2.2.

|  |
| --- |
| const findMatchingSequence = (buffer, slidingWindow) => {  let matchLength = 0;  let nextChar = slidingWindow[0] || ' ';  let matchIndex = 0;   for (let i = 1; i <= slidingWindow.length; i++) {  const sequence = slidingWindow.slice(0, i);  const index = buffer.indexOf(sequence);  if (index !== -1) {  matchLength = i;  matchIndex = index + 1;  nextChar = slidingWindow[i] || ' ';  } else {  break;  }  }   return { matchIndex, matchLength, nextChar }; }; |

Листинг 2.2 – Функция для поиска максимального совпадения

Следующим шагом будет описана функция, которая проверяет размер буфера и, при необходимости, обрезает его до заданного максимального значения. Если длина буфера превышает допустимый предел, удаляются самые старые символы (оставляются только последние maxSize символов). Такая стратегия позволяет поддерживать фиксированный размер буфера. Функция продемонстрирована в листинге 2.3.

|  |
| --- |
| const checkBufferSize = (maxSize, buffer) => {  return buffer.length > maxSize ? buffer.slice(-maxSize) : buffer; }; |

Листинг 2.3 – Функция для создания циклических сдвигов строки

Далее опишем процедуру получения параметров сжатия от пользователя. Для взаимодействия используется интерфейс командной строки readline. Последовательно запрашиваются размеры словаря (dictionarySize) и скользящего окна (slidingWindowSize), причём ввод продолжается до получения допустимого значения (не равного нулю). После ввода необходимых параметров интерфейс ввода закрывается. Программная реализация функции показана в листинге 2.4.

|  |
| --- |
| const rl = readline.createInterface({  input: *process*.stdin,  output: *process*.stdout });  const promptUser = (query) => new Promise(resolve => rl.question(query, resolve)); let dictionarySize = 0; let slidingWindowSize = 0;  while (dictionarySize === 0) {  dictionarySize = parseInt(await promptUser("\nРазмер словаря (n1): ")); } while (slidingWindowSize === 0) {  slidingWindowSize = parseInt(await promptUser("\nРазмер скользящего окна (n2): ")); }  rl.close(); |

Листинг 2.4 – Процедура получения параметров сжатия

Далее происходит инициализация переменных для работы алгоритма сжатия. Создаются буфер словаря, скользящее окно и оставшийся текст, который пока не попал в окно. Буфер словаря заполняется нулями для симуляции начального состояния. Подготовительный этап представлен в листинге 2.5.

|  |
| --- |
| let dictionaryBuffer = '0'.repeat(dictionarySize); let slidingWindow = inputText.substring(0, slidingWindowSize); let remainingText = inputText.substring(slidingWindowSize); |

Листинг 2.5 – Функция для создания циклических сдвигов строки

Затем реализован основной цикл выполнения сжатия данных. На каждой итерации вызывается функция поиска совпадения между буфером и скользящим окном. После этого символы, соответствующие найденному совпадению, добавляются в буфер словаря и окно, а сами буферы корректируются по размеру с помощью ранее описанных функций. Промежуточные состояния буферов и результат кодирования каждой итерации выводятся в консоль. Код сжатия показан в листинге 2.6.

|  |
| --- |
| while (slidingWindow.length > 0) {  ({ matchIndex: matchStartIndex, matchLength, nextChar: nextSymbol } = findMatchingSequence(dictionaryBuffer, slidingWindow));   ({ buffer: dictionaryBuffer, slidingWindow } = addSymbolsToBuffer(matchLength + 1, dictionaryBuffer, slidingWindow));  dictionaryBuffer = checkBufferSize(dictionarySize, dictionaryBuffer);   ({ buffer: slidingWindow, slidingWindow: remainingText } = addSymbolsToBuffer(matchLength + 1, slidingWindow, remainingText));  slidingWindow = checkBufferSize(slidingWindowSize, slidingWindow);  *console*.log(`${dictionaryBuffer}\t|\t${slidingWindow}\t|\t${matchStartIndex}${matchLength}${nextSymbol}\n`);   compressedOutput += `${matchStartIndex}${matchLength}${nextSymbol}`; } |

Листинг 2.6 – Цикл выполнения сжатия данных

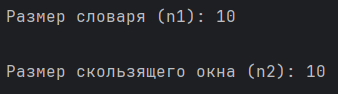
Далее начинается процесс распаковки сжатых данных. Инициализируется новый буфер словаря, и для каждого кода из сжатого сообщения выполняется восстановление исходных символов. Если совпадение отсутствует, просто добавляется следующий символ. При наличии совпадения строится строка на основе буфера словаря. Этап распаковки представлен в листинге 2.7.

|  |
| --- |
| for (let i = 0; i < compressedOutput.length / 3; i++) {  matchStartIndex = parseInt(compressedOutput[3 \* i]);  matchLength = parseInt(compressedOutput[3 \* i + 1]);  nextSymbol = compressedOutput[3 \* i + 2];   if (matchLength === 0 && matchStartIndex === 0) {  decompressedText += nextSymbol;  dictionaryBuffer += nextSymbol;  } else {  tempString = dictionaryBuffer.substring(matchStartIndex - 1, matchStartIndex - 1 + matchLength) + nextSymbol;  decompressedText += tempString;  dictionaryBuffer += tempString;  }   dictionaryBuffer = checkBufferSize(dictionarySize, dictionaryBuffer); } |

Листинг 2.7 – Процесс распаковки сжатых данных

Результат работы программы показан на рисунках 2.1 и 2.2.





Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным. Изображение выглядит как текст, снимок экрана

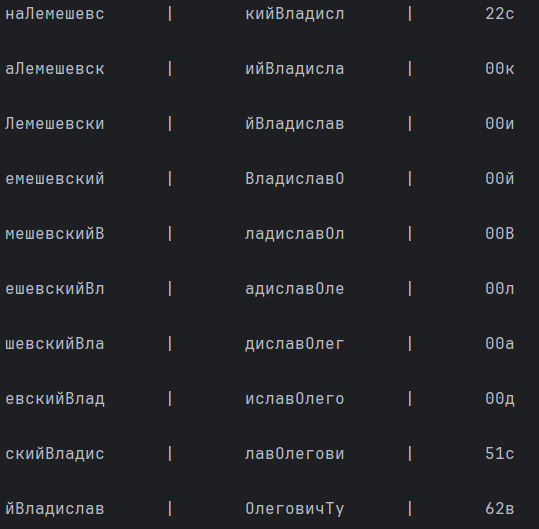
Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным. Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.1 – Вывод результатов

 Изображение выглядит как текст, снимок экрана, меню

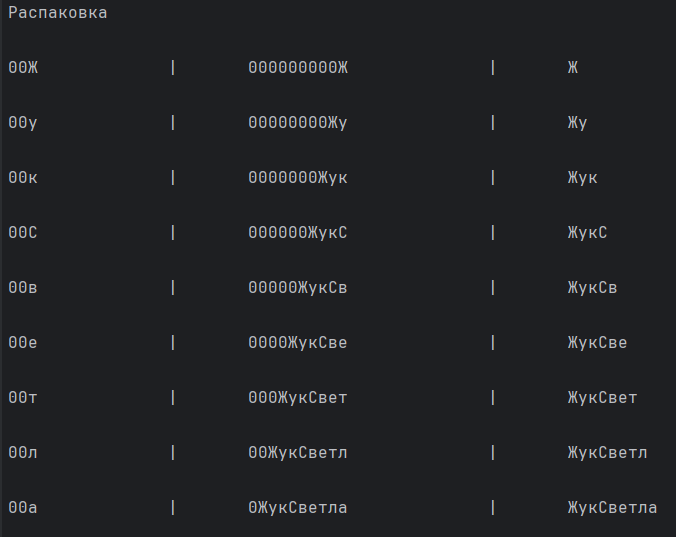
Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, меню

Содержимое, созданное искусственным интеллектом, может быть неверным. Изображение выглядит как текст, снимок экрана, дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.



 Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 2.2 – Вывод результатов

# **3 Скорость и эффективность выполнения сжатия/распаковки**

Для вычисления скорости выполнения сжатия/распаковки будет подключен модуль perf\_hooks, который позволяет измерять время в node.js.

Для проведения эксперимента с оценкой скорости и эффективности сжатия/распаковки при изменении размеров окон (буфера и словаря), начнем с использования размеров *n*1 = 10 и *n*2 = 10. На рисунке 3.1 будет представлен результат при данных значениях параметров.

Изображение выглядит как текст, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 3.1 – Вывод результатов

Следующие размеры будут *n*1 = 5 и *n*2 = 5. На рисунке 3.2 будет представлен результат при данных значениях параметров.

Изображение выглядит как текст, Шрифт, снимок экрана, типография

Содержимое, созданное искусственным интеллектом, может быть неверным.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 3.2 – Вывод результатов

Можно заключить, что увеличение размеров скользящего окна приводит к снижению времени кодирования и декодирования, а также к росту степени сжатия. Напротив, уменьшение размеров окна вызывает увеличение времени выполнения операций и снижение эффективности сжатия.

Данное поведение объясняется тем, что при больших размерах окна алгоритм получает больше возможностей для поиска повторяющихся фрагментов текста. Это позволяет заменять их более короткими представлениями, что, в свою очередь, увеличивает общий коэффициент сжатия и уменьшает объём данных, подлежащих хранению.

# **Вывод**

В ходе данной лабораторной работы исследования показали, что увеличение размеров словаря и буфера в алгоритмах сжатия приводит к росту времени выполнения операций, несмотря на потенциальное улучшение степени сжатия.

Метод Лемпеля-Зива, являющийся одним из наиболее эффективных и распространённых подходов, демонстрирует высокую эффективность за счёт замены повторяющихся фрагментов данных. Благодаря хорошему балансу между степенью сжатия и скоростью работы, он нашёл широкое применение в таких сферах, как обработка изображений, аудио и других типов данных.