МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное учреждение высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент математического и компьютерного моделирования

О Т Ч Е Т

по практическому заданию по дисциплине
«Алгоритмы и структуры данных»
об исследовании алгоритма Лемпеля-Зива-Велча

направление подготовки 09.03.03 «Прикладная информатика»
профиль «Прикладная информатика в компьютерном дизайне»

Выполнил студент
гр. № Б9121-09.03.03пикд
_____ Кирилов Д. Ю.
(подпись)

Отчет защищен

с оценкой: _____

Руководитель
_____
(должность, ученое звание)

_____
(подпись) (ФИО)

Регистрационный номер _____

«____» _____2022г.

«_____» _____ 2022г.

г. Владивосток
2022 г.

Оглавление

1. Введение

Алгоритм **Лемпеля—Зива—Велча** (**Lempel-Ziv-Welch**, **LZW**) — это универсальный алгоритм сжатия данных без потерь, названный по именам авторов: Авраама Лемпеля (англ. Abraham Lempel), Яакова Зива (англ. Jacob Ziv) и Терри Уэлча (англ. Terry Welch). Оригинальная версия метода была создана Лемпелем и Зивом в 1978 году (LZ78) и доработана Уэлчем в 1984 году.

Если взглянуть почти на любой файл данных в компьютере, просматривая символ за символом, то можно заметить множество повторяющихся символов/цепочек символов. LZW — это метод сжатия данных, который воспользовался этим повторением.

LZW — алгоритм сжатия на основе "словаря", т. е. LZW кодирует данные, обращаясь к словарю. Таким образом, чтобы закодировать подстроку, в выходной файл нужно записать только одно кодовое число, соответствующее индексу этой подстроки в словаре. Словарь постоянно обновляется во время кодирования и декодирования данных. Это позволяет закодировать повторяющиеся последовательности байтов в данных используя меньше бит для их представления.

Алгоритм LZW работает следующим образом:

1. Создается словарь, который содержит все отдельные символы исходного текста и их коды(индексы).
2. Проходится по исходному тексту, и каждая последовательность символов заменяется на соответствующий ей код из словаря.
3. В словарь добавляются новые сочетания символов, если они встречаются в исходном тексте.

1.1. Преимущества алгоритма LZW

- Алгоритм является однопроходным.

- Для декомпрессии не надо сохранять таблицу строк в файл для распаковки: Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.
- Высокая степень сжатия: LZW способен обеспечить высокую степень сжатия, особенно для файлов, содержащих повторяющиеся цепочки данных.
- Простота реализации: Алгоритм относительно прост в реализации и не требует больших вычислительных мощностей.
- Достаточно высокая скорость сжатия и распаковки.

1.2. Недостатки алгоритма LZW

- Ограничение размером словаря: сжатие LZW может быть ограничено размером словаря, что может быть проблемой для больших файлов или файлов с большим количеством уникальных цепочек.

2. Применение

На момент своего появления алгоритм LZW давал лучший коэффициент сжатия для большинства приложений, чем любой другой хорошо известный метод того времени. Он стал первым широко используемым на компьютерах методом сжатия данных.

Алгоритм (а точнее его модификация, LZC, см. ниже) был реализован в программе compress, которая стала более или менее стандартной утилитой Unix-систем приблизительно в 1986 году. Несколько других популярных утилит-архиваторов также используют этот метод или близкие к нему.

В 1987 году алгоритм стал частью стандарта на формат изображений GIF. Он также может (опционально) использоваться в формате TIFF. Помимо этого, алгоритм применяется в протоколе модемной связи v.42bis и стандарте PDF (хотя по умолчанию большая часть текстовых и визуальных данных в PDF сжимается с помощью алгоритма Deflate).

LZW алгоритм также может применяется для:

- Сжатия изображений и видео.
- Сжатия текста.
- Сжатия исполняемых программ.
- Сжатия потоков данных.
- Сжатия аудиосигналов.
- Сжатия медицинских изображений.
- Сжатия спутниковых изображений.
- Сжатия веб-графики.
- Сжатия файлов автоматизированного проектирования.

3. Идея алгоритма сжатия

LZW-алгоритм начинает работу с исходного словаря символов и использует его в качестве "стандартного" набора. В начале словарь представляет собой односимвольные цепочки, которые могут встретиться в процессе работы алгоритма. Процесс кодирования состоит в том, что алгоритм считывает из входного потока символ и проверяет, есть ли этот символ в словаре. Встретив символ в словаре, алгоритм кодирует их в виде числа, которое представляет собой индекс в этом самом словаре. Всякий раз, встречая новую подстроку (например, "ab"), он добавляет ее в словарь; каждый раз, когда ему попадается подстрока, которая ранее уже встречалась, он просто считывает новый символ и выполняет его конкатенацию с текущей строкой, чтобы получить новую подстроку.

Говоря простым языком, LZW-алгоритм считывает очередной непрочитанный символ и пытается найти его в словаре. Если он в словаре есть, то алгоритм пытается его удлинить, путём добавления следующего, и ищет в словаре уже удлиненную цепочку. Как только алгоритм находит цепочку, которая в коротком варианте уже записана в словаре, а в удлинённом нет, короткий вариант кодируется индексом, который уже есть в словаре, а удлиненный вариант заносится в словарь.

3.1. Пример

Предположим, что наш поток, который мы хотим сжать, это «abdabccabcdab», при этом используется только начальный словарь (табл. 1).

ИНДЕКС	ВХОД
0	a
1	b
2	c
3	d

Табл. 1 – Начальный словарь

Этапы кодирования будут выполняться следующим образом (табл. 2).

ВХОД	ТЕКУЩАЯ СТРОКА	ВИДЕЛИ ЭТО РАНЬШЕ?	КОДИРОВАННЫЙ ВЫХОД	НОВАЯ ЗАПИСЬ В СЛОВАРЕ/ ИНДЕКС
a	*a*	да	ничего	никакой
a*b*	a*b*	нет	0	ab / 4
ab*d*	b*d*	нет	0, 1	bd / 5
abd*a*	d*a*	нет	0, 1, 3	da / 6
abda*b*	a*b*	да	без изменений	никакой
abdab*c*	ab*c*	нет	0, 1, 3, 4	abc / 7
abdabc*c*	c*c*	нет	0, 1, 3, 4, 2	cc / 8
abdabcc*a*	c*a*	нет	0, 1, 3, 4, 2, 2	ca / 9
abdabcca*b*	a*b*	да	без изменений	никакой
abdabccab*c*	ab*c*	да	без изменений	никакой
abdabccabc*d*	abc*d*	нет	0, 1, 3, 4, 2, 2, 7	abcd / 10
abdabccabcd*a*	d*a*	да	без изменений	никакой
abdabccabcda*b*	da*b*	нет	0, 1, 3, 4, 2, 2, 7, 6	dab / 11
abdabccabcdab	b	да	0, 1, 3, 4, 2, 2, 7, 6, 1	никакой

Табл. 2 – Процесс кодирования при входной строке «abdabccabcdab»

После считывания последнего символа, ‘b’, должен быть выведен индекс последней подстроки, ‘b’.

3.2. Псевдокод алгоритма сжатия

```
string s;
char ch;
...

s = empty string; //пустая цепочка
while (there is still data to be read) //пока есть символы во входном потоке
{
    ch = read a character; //взять очередной символ
    if (dictionary contains s+ch) //цепочка s+ch уже есть в таблице
    {
      s = s+ch; //удлиняем цепочку прочитанным символом
    }
```

```
    else //цепочки s+ch нет в таблице
    {
      encode s to output file; //вывод кода, соответствующего цепочке s
      add s+ch to dictionary; //добавляем новую цепочку s+ch в таблицу
      s = ch; //готовимся к «обнаружению» следующей цепочки
    }
}
encode s to output file; //вывод кода для оставшейся цепочки s
```

Алгоритм кодирования каждый раз пытается найти в таблице наиболее длинную цепочку, соответствующую читаемой последовательности символов. Если это в какой-то момент не удается, то накопленная к этому времени цепочка заносится в таблицу. В какой-то момент может наступить переполнение таблицы. В этом случае кодировщик выводит в выходной поток специальный код очистки, и таблица цепочек инициализируется заново. Обычно в реальных алгоритмах применяется таблица с 4096 входами для цепочек, при этом число 256 является кодом очистки (CLC), а число 257 – кодом конца информации (EOI), эти строки таблицы не используются для цепочек.

Заметим, что в этом случае для каждого выходного кода достаточно 12 бит. Поэтому при выводе целесообразно упаковывать каждые два кода в три байта.

4. Идея алгоритма распаковки

Формирование словаря независимо от того, находимся мы на стороне кодировщика или декодировщика. И кодировщик и декодировщик работают по одной и той же схеме, и поэтому словарь они заполняют как бы синхронно. Поэтому словари у них получаются идентичные и никакой дополнительной информации о частотных или каких-либо ещё характеристиках набора символов им передавать не нужно. В процессе работы словарь, идентичный тому, который был создан во время сжатия, восстанавливается. Программы кодирования и декодирования должны начинаться с одного и того же начального словаря.

Декодер LZW сначала считывает индекс (целое число), ищет этот индекс в словаре и выводит подстроку, связанную с этим индексом. Первый символ этой подстроки конкатенируется с текущей рабочей строкой. Эта новая конкатенация добавляется в словарь. Затем декодированная строка становится текущей рабочей строкой (текущий индекс, т. е. подстрока, запоминается), и процесс повторяется.

4.1. Пример

Этапы декодирования будут выполняться следующим образом (табл. 3) (при использовании всё того же начального словаря из табл. 1.).

КОДИРОВАННЫЙ ВХОД	**ПРЕОБРАЗОВАНИЕ СЛОВАРЯ**	**ДЕКОДИРОВАННЫЙ ВЫХОД**	**ТЕКУЩАЯ СТРОКА**	**НОВАЯ СЛОВАРНАЯ ЗАПИСЬ/ ИНДЕКС**
0	0 = *a*	*a*	никакая	никакая
0, *1*	1 = *b*	a*b*	a	ab / 4
0, 1, *3*	3 = *d*	ab*d*	b	bd / 5
0, 1, 3, *4*	4 = *ab*	abd*ab*	d	da / 6
0, 1, 3, 4, *2*	2 = *c*	abdab*c*	ab	abc / 7
0, 1, 3, 4, 2, *2*	2 = *c*	abdabc*c*	c	cc / 8
0, 1, 3, 4, 2, 2, 7	7 = *abc*	abdabcc*abc*	c	ca / 9
0, 1, 3, 4, 2, 2, 7, 6	6 = *da*	abdabccabc*da*	abc	abcd / 10

0, 1, 3, 4, 2, 2, 7, 6, 1	1 = *b*	abdabccabcda*b*	da	dab / 11

Табл. 3 – Процесс декодирования при кодированном входе «0, 1, 3, 4, 2, 2, 7, 6, 1»

4.2. Возможные проблемы при декодировании

Есть исключение, когда алгоритм не работает; это происходит, когда код вызывает индекс, который еще не был введен в словарь (или когда только что закодированная строка встречается на входе). Например, предположим, у нас есть строка «abababa» и начальный словарь, состоящий только из a и b с индексами 0 и 1 соответственно. Тогда процесс кодирования будет состоять из следующих этапов (табл. 4).

ВХОД	**ТЕКУЩАЯ СТРОКА**	**ВИДЕЛИ ЭТО РАНЬШЕ?**	**КОДИРОВАННЫЙ ВЫХОД**	**НОВАЯ ЗАПИСЬ В СЛОВАРЕ/ ИНДЕКС**
a	*a*	да	ничего	никакой
a*b*	a*b*	нет	0	ab / 2
ab*a*	b*a*	нет	0, 1	ba / 3
aba*b*	a*b*	да	без изменений	никакой
abab*a*	ab*a*	нет	0, 1, 2	aba / 4
ababa*b*	a*b*	да	без изменений	никакой
abababa	ab*a*	да	0, 1, 2, 4	никакой

Табл. 4 – Процесс кодирования при входной строке «abababa»

Тогда при декодировании возникает следующая проблема (табл. 5).

КОДИРОВАННЫЙ ВХОД	**ПРЕОБРАЗОВАНИЕ СЛОВАРЯ**	**ДЕКОДИРОВАННЫЙ ВЫХОД**	**ТЕКУЩАЯ СТРОКА**	**НОВАЯ СЛОВАРНАЯ ЗАПИСЬ/ ИНДЕКС**
0	0 = *a*	*a*	никакая	никакая
0, *1*	1 = *b*	a*b*	a	ab / 2
0, 1, *2*	2 = *ab*	ab*ab*	b	ba / 3
0, 1, 2, *4*	4 = *???*	abab*???*	ab	???

Табл. 5 – Процесс декодирования при кодированном входе «0, 1, 2, 4»

Таким образом, чтобы алгоритм справился с этим исключением, нужно взять подстроку, которую уже получили, ‘ab’, и конкатенировать ее первый символ с самим собой, ‘ab’ + ‘a’ = “aba” вместо того, чтобы следовать процедуре как обычно.

4.3. Псевдокод алгоритма распаковки

```
while ((k = nextcode)) != EOI) //пока не конец информации
{
    if (k == CLC) //k есть код очистки
    {
     InitTable(); //заново инициализируем таблицу
     k = nextcode; //читаем следующий код
     if (k == EOI)
      break;
     OutString(k); //выводим цепочку для кода k
     old = k; //запоминаем текущий код
    }
    else
    {
     if (InTable(k)) //в таблице есть строка для кода k
     {
      OutString(k); //выводим цепочку для кода k
      AddString(String(old)+Char(k)); //формируем и добавляем новую цепочку
      old = k;
     }
     else // в таблице нет строки для кода k
     {
      s = String(old)+Char(old); //формируем цепочку
      OutString(s); //выводим цепочку
      AddString(s); //и добавляем её в таблицу
      old = k;
     }
    }
}
```

5. Оценка алгоритма

Будем считать, что базовый алфавит закодирован числами от 0 до 2^n-1 записанными n-битными блоками, и кодирование собственно документа начинается с номера 2^n. В таких предположениях, количество бит, необходимых для кодирования L первых слов кодируемого текста равно

$$\sum_{j=n}^{K-1} 2^j(j+1) + (L - \sum_{j=n}^{K-1} 2^j)(K+1) = (K-n+2)2^n - 2^{K+1} + L(K+1)$$

при $K=\lfloor \log^2 L \rfloor$ ($\lfloor$ $\rfloor$ означает целую часть числа).

К примеру текст, содержащий 613 символов и с исходным словарём в 3 символа, может быть закодирован 3*613 = 1839 битами. LZW алгоритм в таком случае даёт 241 цепочку в словарь. Стоимость кодирования, при использовании алгоритма, составит 3*3 + 8*4+ 16*5 + 32*6 + 64*7 + (241 - 3 - 120)*8 = 1705 бит. 1705/1839 ≈ 0.927, из этого следует, что эффективность сжатия составит порядка 10%.

LZW-сжатие выделяется среди прочих, когда встречается с потоком данных, содержащим повторяющиеся строки любой структуры. По этой причине он работает весьма эффективно, когда встречает английский текст. Уровень сжатия может достигать 50% и выше.

Поскольку размер словаря фиксирован и не зависит от длины ввода, сложность LZW обычно равна O(n) (где n - количество символов во входных данных), т. к. каждый байт считывается только один раз, а сложность операции для каждого символа постоянна. Это означает, что время выполнения алгоритма линейно увеличивается с размером входных данных. Пространственная сложность алгоритма LZW обычно равна O(k), где k - количество уникальных строк во входных данных. Это означает, что объем памяти, требуемый алгоритмом, увеличивается с увеличением количества уникальных строк во входных данных.

Другой способ оценить алгоритм LZW - сравнить его степень сжатия и скорость декомпрессии с другими подобными алгоритмами. Алгоритм LZW довольно эффективен с точки зрения степени сжатия и скорости декомпрессии по сравнению с другими алгоритмами, такими как кодирование Хаффмана. Однако он работает не так хорошо, как более поздние алгоритмы, такие как LZ77 и LZ78.

6. Описание реализации

Одной из основных задач итоговой реализации являлась наглядная демонстрация работы алгоритма. В процессе работы стояла цель разработать такой вариант реализации, который бы отражал основные функции алгоритма (считывание данных, кодирование(сжатие)/декодирование(распаковка)). Для этого было необходимо выбрать такой тип входных данных, который наглядно демонстрирует пользователю принцип работы этих функций, и текстовые данные справляются с этой задачей лучше всего.

В результате были выбраны конструкции кода, основанные на описанных ранее принципах работы функций, и которые, как следствие, выполняют свою работу поэтапно (т. е. считывание, сравнение со словарём, заполнение словарей и/или переход к следующей цепочке и т. д.). Поэтапное выполнение легло в основу реализации для удовлетворения описанных ранее целей. От этого время работы кода напрямую зависит от входных данных (чем больше файл, тем выше время выполнения алгоритма соответственно).

Однако, даже с учетом описанных выше замечаний, разработанная реализация удовлетворяет цель, для которой в 1978 году и создавался данный алгоритм. Это можно заметить при больших входных данных, когда в процессе работы словарь успевает заполниться больше, а следовательно цепочки «сравнения/замены» становятся длиннее.

7. Автоматическое тестирование

Разработанная реализация имеет возможность автоматизировать процесс при большом потоке входных файлов. За это отвечает следующая конструкция:

bool auto_tester = true; //запуск автоматических тестов

void test_process(int test_amount, string path) { //test_amount - количество тестов, path - путь к папке с тестами

Для запуска автоматического тестирования измените значение переменной auto_tester с "false" на "true" в строке 76, а также укажите количество созданных заранее тестов и путь к ним в строке 250. Например:

}

else test_process(19, "C:\\LZWtest");

}

8. Тесты

Ниже приведены тесты, используемые для автоматического тестирования, итоговой реализации.

Тест 1

Проверка на пустоту файла.

Тест 2

Проверка на отсутствие данных для кодирования.

Тест 3

Проверка на отсутствие данных для декодирования.

Тест 4

Проверка на незначащие символы (пробелы, переносы строк).

Тест 5

Проверка на незначащие символы при декодировании (пробелы, переносы строк).

Тест 6

Проверка на тип входных данных при декодировании.

Тест 7

Проверка на существование записи в словаре при начальном заходе в алгоритм декодирования.

Тест 8

Проверка на запрос encode/decode.

Тест 9

Проверка на подряд идущие пробелы/переносы строк при декодировании.

Тест 10–13

Проверка на корректность работы кодирования.

Тест 14–17

Проверка на корректность работы декодирования.

Список литературы

1. Семенюк В. В. Экономное кодирование дискретной информации. –СПб.: СПбГИТМО (ТУ), 2001. – 115 с. - Режим доступа: http://compression.ru/download/articles/rev_univ/semenyuk_2001_econom_encoding.pdf
2. Mark R. Nelson «LZW Data Compression» [Электронный ресурс]: электрон. статья/ Mark R. Nelson, 1989. – Режим доступа: https://marknelson.us/posts/1989/10/01/lzw-data-compression.html
3. Алгоритм LZW [Электронный ресурс]: электрон. статья/ информационный ресурс ИТМО, 2022. – Режим доступа: http://neerc.ifmo.ru/mediawiki/index.php?title=Алгоритм_LZW&redirect=no
4. Hans Wennborg «Shrink, Reduce, and Implode: The Legacy Zip Compression Methods» [Электронный ресурс]: электрон. статья/ Hans Wennborg, 2021. – Режим доступа: https://www.hanshq.net/zip2.html
5. «Подстановочные или словарно-ориентированные алгоритмы сжатия информации. Методы Лемпела-Зива» [Электронный ресурс]/ ИНТУИТ, 2007. Режим доступа: https://intuit.ru/studies/courses/2256/140/lecture/3904
6. Michael Dipperstein «Lempel-Ziv-Welch (LZW) Encoding Discussion and Implementation» [Электронный ресурс]: электрон. статья/ Michael Dipperstein, 2015. Режим доступа: http://michael.dipperstein.com/lzw/
7. Ziv J., Lempel A. «Compression of Individual Sequences via Variable-Rate Coding. » / IEEE Trans. Inform. Theory, 1978 – 530-536 с
8. Welch T. «A Technique for High-Performance Data Compression. »/ Computer, 1984 – 8-19 с.
9. Сэломон Д. «Сжатие данных, изображений и звука.» / М.Техносфера. 2006.
10. Пантелеев Е. Р., Алыкова А. Л. «Алгоритмы сжатия данных без потерь»: учебное пособие для вузов / Е. Р. Пантелеев, А. Л. Алыкова. — Санкт-Петербург: Лань, 2021. – 172 с
11. Bell, Cleary, and Witten «Text Compression», Prentice Hall, 1990.

12. David Salomon, Springer, "Data Compression: The Complete Reference", 2004.
13. Paul Howard, Jean-Loup Gailly «Data Compression: The Art of Data Compression», M&T Books, 1996.
14. David MacKay, «Information Theory, Inference and Learning Algorithms» / Cambridge University Press, 2003.
15. Khalid Sayood, Morgan Kaufmann, «Introduction to Data Compression», 2000.
16. D. S. Hirschberg, D. J. DeWitt, Morgan Kaufmann, « Data Compression: An Algorithmic Perspective», 1992.
17. David Salomon, Springer, «Handbook of Data Compression», 2015.
18. Storer, J. A. «Data Compression: Methods and Theory» / Computer Science Press, 1988.
19. Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic Coding for Data Compression. Communications of the ACM, 30(6), 520–540.
20. Storer, J. A., & Szymanski, T. G. (1989). The Context Tree Weighting Method: Basic Properties. IEEE Transactions on Information Theory, 35(3), 613–621.
21. Korn, A., & Korn, M. (2007). Data Compression: The Complete Reference. Springer.
22. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.
23. Gersho, A., & Gray, R. M. (1992). Vector quantization and signal compression.
24. Ziv, J., & Lempel, A. (1992). Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory, 28(6), 807-814.
25. Ziv, J., & Lempel, A. (1993). On the complexity of universal lossless data compression. IEEE Transactions on Information Theory, 39(3), 517-521.
26. Cleary, J. G., & Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications
27. Storer, J. A., & Szymanski, T. G. (1988). Data compression via textual substitution. Communications of the ACM, 31(4), 300–312.