Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Тульский государственный университет

КАФЕДРА ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ

СЛОВАРНЫЕ МЕТОДЫ СЖАТИЯ ДАННЫХ. АЛГОРИТМ LZW

Лабораторная работа № 5
по курсу «Кодирование и сжатие данных»

Вариант №4

Выполнил: студент группы 220601 _____ Белым А.А.
(подпись)
Проверил: к. т. н., доцент каф. ИБ _____ Гетманец В.М.
(подпись)

Тула 2014

Цель работы

Целью работы является ознакомление со словарными алгоритмами кодирования данных, изучение алгоритма LZW и его реализации.

Задание

Изучить исходные коды реализации алгоритма LZW и исследовать его эффективность на файлах различных типов.

Описание метода

Алгори́тм Ле́мпеля — Зи́ва — Ве́лча (Lempel-Ziv-Welch, LZW) — это универсальный алгоритм сжатия данных без потерь, созданный Абрахамом Лемпелем (англ. Abraham Lempel), Якобом Зивом (англ. Jacob Ziv) и Терри Велчем (англ. Terry Welch). Он был опубликован Велчем в 1984 году, в качестве улучшенной реализации алгоритма LZ78, опубликованного Лемпелем и Зивом в 1978 году. Алгоритм разработан так, чтобы его можно было быстро реализовать, но он не обязательно оптимален, поскольку он не проводит никакого анализа входных данных.

Данный алгоритм при сжатии (кодировании) динамически создаёт таблицу преобразования строк: определённым последовательностям символов (словам) ставятся в соответствие группы бит фиксированной длины (обычно 12-битные). Таблица инициализируется всеми 1-символьными строками (в случае 8-битных символов — это 256 записей). По мере кодирования, алгоритм просматривает текст символ за символом, и сохраняет каждую новую, уникальную 2-символьную строку в таблицу в виде пары код/символ, где код ссылается на соответствующий первый символ. После того как новая 2-символьная строка сохранена в таблице, на выход передаётся код первого символа. Когда на входе читается очередной символ, для него по таблице находится уже встречавшаяся строка максимальной длины, после чего в таблице сохраняется код этой строки со следующим символом на входе; на выход выдаётся код этой строки, а следующий символ используется в качестве начала следующей строки.

Алгоритму декодирования на входе требуется только закодированный текст, поскольку он может воссоздать соответствующую таблицу преобразования непосредственно по закодированному тексту.

Текст программы

Далее представлен текст программы, выполняющей кодирование и декодирование по алгоритму LZW.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#define BITS 14                   /* Установка длины кода равной 12, 13   */
#define HASHING_SHIFT (BITS-8)      /* или 14 битам.                        */
#define MAX_VALUE ((1 << BITS) - 1)/* Отметим, что на MS-DOS-машине при    */
#define MAX_CODE (MAX_VALUE - 1)    /* длине кода 14 бит необходимо компи-  */
                                  /* лировать, используя large-модель.    */
#if BITS == 14
  #define TABLE_SIZE 18041        /* Размер таблицы строк должен быть     */
#endif                            /* простым числом, несколько большим,   */
#if BITS == 13                    /* чем 2**BITS.                         */
  #define TABLE_SIZE 9029
#endif
#if BITS <= 12
  #define TABLE_SIZE 5021
#endif

/* Это массив для значений кодов            */
int *code_value;
/* Этот массив содержит префиксы кодов      */
unsigned int *prefix_code;
/* Этот массив содержит добавочные символы  */
unsigned char *append_character;
/* Этот массив содержит декодируемые строки */
unsigned char decode_stack[4000];

/*****************************************************************
** Эта программа получает имя файла из командной строки.
** Она упаковывает
** файл, посылая выходной поток в файл test.lzw. Затем распаковывает
** test.lzw в test.out. Test.out должен быть точной копией исходного
** файла.
*****************************************************************/
void compress(FILE *input,FILE *output);
int find_match(int hash_prefix,unsigned int hash_character);
void expand(FILE *input,FILE *output);
unsigned char *decode_string(unsigned char *buffer,unsigned int code);
unsigned input_code(FILE *input);
void output_code(FILE *output,unsigned int code);
int main(int argc, char *argv[])
{
FILE *input_file;
FILE *output_file;
FILE *lzw_file;
char input_file_name[81];
/*
**  Эти три буфера необходимы на стадии упаковки.
*/
 code_value=(int *)malloc(TABLE_SIZE*sizeof(unsigned int));
 prefix_code=(unsigned *)malloc(TABLE_SIZE*sizeof(unsigned int));
```

```
 append_character=(unsigned char*)malloc(TABLE_SIZE*sizeof(unsigned char));
 if (code_value==NULL || prefix_code==NULL || append_character==NULL)
 {
     printf("Fatal error allocating table space!\n");
     exit(0);
 }

/*
** Получить имя файла, открыть его и открыть выходной lzw-файл.
*/
    if (argc>1)
        strcpy(input_file_name,argv[1]);
    else
    {
        printf("Input file name? ");
        scanf("%s",input_file_name);
    }
    input_file=fopen(input_file_name,"rb");
    lzw_file=fopen("test.lzw","wb");
    if (input_file==NULL || lzw_file==NULL)
    {
        printf("Fatal error opening files.\n");
        exit(0);
    };
/*
** Сжатие файла.
*/
    compress(input_file,lzw_file);
    fclose(input_file);
    fclose(lzw_file);
    free(code_value);
/*
** Сейчас открыть файлы для распаковки.
*/
    lzw_file=fopen("test.lzw","rb");
    output_file=fopen("test.out","wb");
    if (lzw_file==NULL || output_file==NULL)
    {
        printf("Fatal error opening files.\n");
        exit(0);
    };
/*
** Распаковка файла.
*/
    expand(lzw_file,output_file);
    fclose(lzw_file);
    fclose(output_file);
    free(prefix_code);
    free(append_character);
    return 0;
}

/*
** Процедура сжатия.
*/
void compress(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int character;
unsigned int string_code;
unsigned int index;
int i;
    next_code=256;    /* Next_code - следующий доступный код строки */
    for (i=0;i<TABLE_SIZE;i++)/*Очистка таблицы строк перед стартом */
        code_value[i]=-1;
    i=0;
```

```
    printf("Compressing...\n");
    string_code=getc(input);    /* Get the first code*/

/*
** Основной цикл. Он выполняется до тех пор, пока возможно чтение
** входного потока.  Отметим, что он прекращает заполнение таблицы
** строк после того, как все возможные коды были использованы.
*/
    while ((character=getc(input)) != (unsigned)EOF)
    {
        if (++i==1000)              /* Печатает * через каждые 1000  */
        {                           /* чтений  входных символов (для */
            i=0;                    /* умиротворения зрителя).       */
            printf("*");
        }
        /* Смотрит, есть ли строка */
        index=find_match(string_code,character);
        if (code_value[index] != -1)        /* в таблице.  Если есть,*/
            string_code=code_value[index];/* получает значение кода*/
        else                                /* Если нет, добавляет ее*/
        {                                   /* в таблицу.            */
            if (next_code <= MAX_CODE)
                        {
                code_value[index]=next_code++;
                prefix_code[index]=string_code;
                append_character[index]=character;
            }
            output_code(output,string_code);/*Когда обнаруживается, что*/
            string_code=character;           /*строки нет в таблице,   */
        }                                   /*выводится последняя строка*/
    }                                       /*перед добавлением новой   */
/*
** End of the main loop.
*/
    output_code(output,string_code);  /* Вывод последнего кода       */
    output_code(output,MAX_VALUE);    /* Вывод признака конца потока */
    output_code(output,0);            /* Очистка буфера вывода       */
    printf("\n");
}
/*
** Процедура хэширования.  Она пытается найти сопоставление для строки
** префикс+символ в таблице строк. Если найдено, возвращается индекс.
** Если нет, то возвращается первый доступный индекс.
*/
int find_match(int hash_prefix,unsigned int hash_character)
{
int index;
int offset;

    index = (hash_character << HASHING_SHIFT) ^ hash_prefix;
    if (index == 0)
        offset = 1;
    else
        offset = TABLE_SIZE - index;
    while (1)
    {
if (code_value[index] == -1)
      return(index);
if (prefix_code[index]==hash_prefix
&&append_character[index]==hash_character)
      return(index);
  index -= offset;
  if (index < 0)
      index += TABLE_SIZE;
    }
}
```

```
/*
**  Процедура распаковки.  Она читает файл LZW-формата и распаковывает
**  его в выходной файл.
*/
void expand(FILE *input,FILE *output)
{
unsigned int next_code;
unsigned int new_code;
unsigned int old_code;
int character;
int counter;
unsigned char *string;

    next_code=256;            /* Следующий доступный код.          */
    counter=0;                 /* Используется при выводе на экран.*/
    printf("Expanding...\n");

  old_code=input_code(input);/*Читается первый код, инициализируется*/
  character=old_code;   /* переменная character и посылается первый */
  putc(old_code,output);      /* код в выходной файл.      */
/*
**  Основной цикл распаковки.  Читаются коды из LZW-файла до тех пор,
**  пока не встретится специальный код, указывающий на конец данных.
*/
    while ((new_code=input_code(input)) != (MAX_VALUE))
    {
        if (++counter==1000) { counter=0; printf("*"); }
/*
** Проверка кода для специального случая
** STRING+CHARACTER+STRING+CHARACTER+
** STRING, когда генерируется неопределенный код.
** Это заставляет его декодировать последний код,
** добавив CHARACTER в конец декод. строки.
*/
        if (new_code>=next_code)
        {
            *decode_stack=character;
            string=decode_string(decode_stack+1,old_code);
        }
/*
** Иначе декодируется новый код.
*/
        else
            string=decode_string(decode_stack,new_code);
/*
** Выводится декодируемая строка в обратном порядке.
*/
        character=*string;
        while (string >= decode_stack)
            putc(*string--,output);
/*
** Наконец, если возможно, добавляется новый код в таблицу строк.
*/
        if (next_code <= MAX_CODE)
        {
            prefix_code[next_code]=old_code;
            append_character[next_code]=character;
            next_code++;
        }
        old_code=new_code;
    }
    printf("\n");
}

/*
```

```
** Процедура простого декодирования строки из таблицы строк,
* сохраняющая
** результат в буфер.  Этот буфер потом может быть выведен
** в обратном порядке программой распаковки.
*/
unsigned char *decode_string(unsigned char *buffer,unsigned int code)
{
    int i;
    i=0;
    while (code > 255)
    {
        *buffer++ = append_character[code];
        code=prefix_code[code];
        if (i++>=4094)
        {
            printf("Fatal error during code expansion.\n");
            exit(0);
        }
    }
    *buffer=code;
    return(buffer);
}
/*
** Следующие две процедуры управляют вводом/выводом кодов
** переменной длины.  Они для ясности написаны чрезвычайно
** простыми и не очень эффективны.
*/
unsigned input_code(FILE *input)
{
    unsigned int return_value;
    static int input_bit_count=0;
    static unsigned long input_bit_buffer=0L;
    while (input_bit_count <= 24)
    {
        input_bit_buffer|=(unsigned long)getc(input)<<(24-input_bit_count);
        input_bit_count += 8;
    }
    return_value=input_bit_buffer >> (32-BITS);
    input_bit_buffer <<= BITS;
    input_bit_count -= BITS;
    return(return_value);
}

void output_code(FILE *output,unsigned int code)
{
    static int output_bit_count=0;
    static unsigned long output_bit_buffer=0L;
    output_bit_buffer|=(unsigned long)code<<(32-BITS-output_bit_count);
    output_bit_count += BITS;
    while (output_bit_count >= 8)
    {
        putc(output_bit_buffer >> 24,output);
        output_bit_buffer <<= 8;
        output_bit_count -= 8;
    }
}
```

Тестовый пример

1. Для исполняемого файла исходный размер составлял 115 299 байт. После сжатия длина файла сократилась до 94 242 байт. Итого на 1 символ приходится

6.54 бита. После сжатия ZIP длина файла составила 47 513 байт. Это составляет 3.4 бита на символ. Оценка энтропии составляет 5.68 бита на символ.

2. Для файла с русским текстом исходный размер составлял 1 351 754 байт. После сжатия длина файла сократилась до 452 292 байт. Итого на 1 символ приходится 2.68 бита. После сжатия ZIP длина файла составила 359 415 байт. Это составляет 2.13 бита на символ. Оценка энтропии составляет 4.21 бита на символ.

3. Для картинки в формате BMP исходный размер составлял 5 572 854 байт. После сжатия длина файла сократилась до 1 915 679 байт. Итого на 1 символ приходится 2.75 бита. После сжатия ZIP длина файла составила 1 470 592 байт. Это составляет 2.11 бита на символ. Оценка энтропии составляет 2.86 бита на символ.

Вывод

В данной работе рассмотрен словарный алгоритм LZW. Словарные алгоритмы достигают большой эффективности при наличии больших повторяющихся последовательностей, как текстовые файлы, и часто используются в программах-архиваторах. Алгоритм LZW также применяется в форматах GIF и TIFF.

Была написана программа, реализующая кодирование и декодирование алгоритмом LZW.