g***Министерство образования Республики Беларусь***

***Учреждение образования***

***«Брестский государственный технический университет»***

***Кафедра ИИТ***

**Лабораторная работа №1**

**По дисциплине КМЗИ за IV семестр**

**Тема: «Простейшие алгоритмы криптографии»**

**Выполнил:**

Студент группы ИИ-15 (1)

2-го курса

Волк И. А.

**Проверил:**

Хацкевич М. В.

Брест 2019

Цель работы: изучить простейшие алгоритмы криптографии

Ход работы

Реализуем гомофонический алгоритм шифрования и сжатие методом Шеннона-Фано.

Листинг:

#include "pch.h"

#include <cmath>

#include <vector>

#include <algorithm>

#include <iostream>

#include <time.h>

#include <iostream>

#include <string>

#include <boost/random.hpp>

#include <fstream>

#include <sstream>

#include <algorithm>

#include <map>

/\*\* Definitions \*\*/

#define VALUE unsigned int

#define COUNT int

#define CODE std::string

typedef std::vector<std::tuple<VALUE, COUNT, CODE>>::iterator table\_row\_iterator;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Classes and structures \*\*/

template<int cols>

class **CTableCipher**;

struct **SCompressedData**;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Functions \*\*/

std::string **ReadFromFile**(const std::string& sFilePath);

void **WriteVectorToFile**(const std::vector<VALUE>& vec, const std::string& sFilePath);

// Format of tuple: value, count, code.

// Here we fill out codes for each row in table

void **FillCodes**(table\_row\_iterator itBegin, table\_row\_iterator itEnd, int nTotalNumber);

SCompressedData **ShannonFanoCompress**(const std::vector<VALUE>& vecInit);

std::vector<VALUE> **ShannonFanoDecompress**(const SCompressedData& data);

void **CompressedDataToFile**(const SCompressedData& data, const std::string& sFilePath);

SCompressedData **CompressedDataFromFile**(const std::string& sFilePath);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Classes and structures \*\*/

// encrypts one character into int using table

template<int cols> // Number of columns in table

class **CTableCipher**

{

private:

VALUE aTable[256][cols];

public:

**CTableCipher**()

{

boost::random::mt19937 gen;

for (int i = 0; i < 256; i++)

{

for (int j = 0; j < cols; j++)

{

unsigned int newValue;

bool bFound = false;

while (!bFound)

{

newValue = gen() % (256 \* cols);

if (IsInTable(newValue)) continue;

bFound = true;

}

aTable[i][j] = newValue;

}

}

}

VALUE **Encrypt**(char cPlainBlock) const

{

int nPlainBlock = static\_cast<int>(cPlainBlock);

boost::random::mt19937 gen;

int col = gen() % cols;

return aTable[nPlainBlock][col];

}

char **Decrypt**(int nEncryptedBlock) const

{

VALUE nPlainBlock = -1;

bool bFound = false;

for (nPlainBlock = 0; nPlainBlock < 256; nPlainBlock++)

{

for (int col = 0; col < cols && !bFound; col++)

if (aTable[nPlainBlock][col] == nEncryptedBlock)

bFound = true;

if (bFound)

break;

}

return static\_cast<char>(nPlainBlock);

}

std::vector<VALUE> **EncryptString**(const std::string& sPlainText) const

{

std::vector<VALUE> vecResult;

for (char ch : sPlainText)

vecResult.push\_back(Encrypt(ch));

return vecResult;

}

std::string **DecryptVector**(const std::vector<VALUE>& vecCipherText) const

{

std::string sResult;

for (int n : vecCipherText)

sResult.push\_back(Decrypt(n));

return sResult;

}

private:

bool **IsInTable**(int nVal)

{

bool bResult = false;

for (int i = 0; i < 256; i++)

{

for (int j = 0; j < cols; j++)

{

if (aTable[i][j] == nVal)

{

bResult = true;

break;

}

}

if (bResult)

break;

}

return bResult;

}

};

struct **SCompressedData**

{

std::map<VALUE, CODE> mapTable;

std::string sCompressedData;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Functions \*\*/

std::string **ReadFromFile**(const std::string& sFilePath)

{

std::stringstream ss;

std::fstream fin(sFilePath);

std::string sTmp;

while (std::getline(fin, sTmp))

ss << sTmp;

return ss.str();

}

void **WriteVectorToFile**(const std::vector<VALUE>& vec, const std::string& sFilePath)

{

std::ofstream fout{ sFilePath, std::ios::binary | std::ios::out };

int size = vec.size();

fout.write((char\*)&size, sizeof(size));

for (int n : vec)

fout.write((char\*)&n, sizeof(n));

}

void **FillCodes**( table\_row\_iterator itBegin,

table\_row\_iterator itEnd,

int nTotalNumber)

{

// for debug

auto x = itEnd - itBegin;

// margin condition

if (itBegin+1 == itEnd)

return;

int nDivider = nTotalNumber / 2;

table\_row\_iterator iSecondBegin;

int nFirstTotalNumber = 0;

int nProgress = 0;

bool fSecondPart = false;

for (auto it = itBegin; it != itEnd; it++)

{

nProgress += std::get<1>(\*it);

// there must be at least one elements in each group

// Thus, we need to be sure if this condition is met

if (fSecondPart)

{

if (!nFirstTotalNumber)

{

nFirstTotalNumber = nProgress - std::get<1>(\*it);

iSecondBegin = it;

/\*std::get<2>(\*it).push\_back('1');

continue;\*/

}

std::get<2>(\*it).push\_back('0');

}

else

{

std::get<2>(\*it).push\_back('1');

}

if (nProgress >= nDivider && !fSecondPart)

fSecondPart = true;

}

FillCodes(itBegin, iSecondBegin, nFirstTotalNumber);

FillCodes(iSecondBegin, itEnd, nTotalNumber - nFirstTotalNumber);

}

SCompressedData **ShannonFanoCompress**(const std::vector<VALUE>& vecInit)

{

std::map<VALUE, std::pair<COUNT, CODE>> mapTable; // var => count, code

std::vector<unsigned int> vecValues;

for (VALUE v : vecInit)

{

if (mapTable.find(v) == mapTable.end())

{

mapTable[v] = { 0, "" };

vecValues.push\_back(v);

}

mapTable[v].first++;

}

// I already wrote algorithm for std::vector<std::tuple<int, int, std::string>>

// It's for compatibility

std::vector<std::tuple<VALUE, COUNT, CODE>> vecTable;

for (auto i = vecValues.begin(); i != vecValues.end(); i++)

vecTable.push\_back({ \*i, mapTable[\*i].first , mapTable[\*i].second });

// sort by count of occurrences

std::sort(vecTable.begin(), vecTable.end(), [](auto left, auto right) {

if (std::get<1>(left) > std::get<1>(right))

return true;

return false;

});

FillCodes(vecTable.begin(), vecTable.end(), vecInit.size());

// fill map again

for (int i = 0; i < vecTable.size(); i++)

{

int nKey = std::get<0>(vecTable[i]);

std::string sCode = std::get<2>(vecTable[i]);

mapTable[nKey].second = sCode;

}

// Get flatted string

std::string sFlattedString;

for (int n : vecInit)

{

sFlattedString += mapTable[n].second;

}

// Fill result

SCompressedData result;

for (auto it : mapTable)

result.mapTable[it.first] = it.second.second; // New map: value -> code, i.e. get rid of the count

result.sCompressedData = sFlattedString;

return result;

}

std::vector<VALUE> **ShannonFanoDecompress**(const SCompressedData& data)

{

std::vector<VALUE> vecResult;

std::string sCurrentWindow;

for (int i = 0; i < data.sCompressedData.length(); i++)

{

sCurrentWindow += data.sCompressedData[i];

int nValue = -1;

for (auto it = data.mapTable.begin(); it != data.mapTable.end(); it++)

{

if (it->second == sCurrentWindow) // if code of item is one we have

{

nValue = it->first; // value of string in window

break;

}

}

if (nValue != -1)

{

vecResult.push\_back(nValue);

sCurrentWindow.clear();

}

}

return vecResult;

}

void **CompressedDataToFile**(const SCompressedData& data, const std::string& sFilePath)

{

std::ofstream fout(sFilePath, std::ios::binary | std::ios::out, std::ios::trunc);

///////////////////////////////////

// Figuring out redundant bytes //

///////////////////////////////////

// redundant meand the number of "0" we need to add at the end of text to fit

int cbRedundant = (8 - data.sCompressedData.length() % 8) % 8;

int cb = 0;

if (!cbRedundant)

cb = data.sCompressedData.length() / 8;

else

{

cb = data.sCompressedData.length() / 8 + 1; // 1 more to not lose data

}

fout.write((char\*)&cbRedundant, sizeof(cbRedundant));

// adding this redundant bytes

std::string sActualDataToWrite = data.sCompressedData;

for (int i = 0; i < cbRedundant; i++)

sActualDataToWrite += '0';

/////////////////

// Map writing //

/////////////////

// at first, number of elements

size\_t size = data.mapTable.size();

fout.write((char\*)&size, sizeof(size));

// then elements themself

for (auto el : data.mapTable)

{

VALUE uValue = el.first;

int nLenght = el.second.length();

char szCode[100] = "";

strcpy\_s(szCode, 100, el.second.c\_str());

szCode[nLenght + 1] = '\0';

fout.write((char\*)&uValue, sizeof(uValue));

fout.write((char\*)&szCode, nLenght + 1); // + 1 for \0

}

//////////////////

// Writing data //

//////////////////

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

{

std::string sByte{ sActualDataToWrite.begin() + i \* 8, sActualDataToWrite.begin() + i \* 8 + 8 };

char nByte = 0;

for (int i = 0; i < 8; i++)

{

nByte <<= 1;

nByte |= (sByte[i] == '1') ? 0b1 : 0b0;

}

fout.write((char\*)&nByte, sizeof(nByte));

}

}

SCompressedData **CompressedDataFromFile**(const std::string& sFilePath)

{

SCompressedData cd;

////////////////////////////////////

// Read number of redundant bytes //

////////////////////////////////////

std::ifstream fin(sFilePath, std::ios::binary | std::ios::in);

int cbRedundant = 0;

fin.read((char\*)&cbRedundant, sizeof(cbRedundant));

/////////////////

// Map reading //

/////////////////

size\_t size;

fin.read((char\*)&size, sizeof(size));

for (int i = 0; i < size; i++)

{

VALUE nValue;

fin.read((char\*)&nValue, sizeof(nValue));

CODE sCode;

char ch;

while (true)

{

fin.read(&ch, 1);

if (ch == '\0')

break;

sCode += ch;

}

cd.mapTable[nValue] = sCode;

}

//////////////////

// Reading data //

//////////////////

while (!fin.eof())

{

char nByte;

fin.read(&nByte, 1);

std::string sByte;

sByte.resize(8, '0');

for (int i = 0; i < 8; i++)

{

char cBit = (nByte & 0b1) ? '1' : '0';

sByte[7 - i] = cBit;

nByte >>= 1;

}

cd.sCompressedData += sByte;

}

cd.sCompressedData.erase(cd.sCompressedData.end() - cbRedundant, cd.sCompressedData.end());

return cd;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()

{

// initial string

std::string sPlainText = ReadFromFile("pt.txt");

std::cout << "Initial plain text: " << sPlainText << std::endl;

// encrypting

CTableCipher<3> ct;

std::vector<VALUE> vecCipherText = ct.EncryptString(sPlainText);

WriteVectorToFile(vecCipherText, "ciphertext.txt");

// compression

SCompressedData data = ShannonFanoCompress(vecCipherText);

CompressedDataToFile(data, "compressed.txt");

data = CompressedDataFromFile("compressed.txt");

// decompression

std::vector<VALUE> vecCipherTextAgain = ShannonFanoDecompress(data);

// decryption

std::string sPlainTextAgain = ct.DecryptVector(vecCipherTextAgain);

std::cout << "Plain text after processing: " << sPlainTextAgain << std::endl;

}

Результаты:

1. Для демонстрации корректности возьмем небольшую строку



1. Для демонстрации эффективности будем шифровать и сжимать следующий файл, заполненный случайными буквами.



Изначальный раземер: 215 КБ.

После операции шифрования и сжатия получаем:



Вывод: по ходу данной лабораторной работы научился производить гомофоническое шифрования, а также сжатие методом Шеннона-Фано.