|  |  |
| --- | --- |
| Министерство образования Республики Беларусь  Учреждение образования «Полоцкий государственный университет» | |
|  | Факультет информационных технологий  Кафедра технологий программирования |
| Лабораторная работа №1 по курсу «Теория информации»  «Сжатие без потерь» | |
| Выполнил | Студент гр. 21-ИТ-1  Катушёнок И.В. |
| Проверила | Васильева Д.М. |
| Полоцк, 2023г. | |

**Ход работы**

Метод оптимального побуквенного кодирования был разработан в 1952 г. Д. Хаффманом. Оптимальный код Хаффмана обладает минимальной средней длиной кодового слова среди всех побуквенных кодов для данного источника с алфавитом *А=*{*a*1*,…,an*} и вероятностями *pi =P*(*ai*).

**Задание:**

1. Изучить теоретический материал
2. Реализовать процедуру построения оптимального кода Хаффмана.
3. Построить код Хаффмана для текста на языке, обозначенном преподавателем, использовать файл не менее 1 Кб.
4. Проверить выполнение неравенства Крафта-МакМиллана для полученного кода
5. Вычислить энтропию исходного файла и сравнить со средней длиной кодового слова.
6. Восстановить исходный текст

Листинг 1 - реализация программы:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Please enter the string:");

string input = Console.ReadLine();

HuffmanTree huffmanTree = new HuffmanTree();

// Build the Huffman tree

huffmanTree.Build(input);

// Encode

BitArray encoded = huffmanTree.Encode(input);

Console.Write("Encoded: ");

foreach (bool bit in encoded)

{

Console.Write((bit ? 1 : 0) + "");

}

Console.WriteLine();

// Decode

string decoded = huffmanTree.Decode(encoded);

Console.WriteLine("Decoded: " + decoded);

Console.ReadLine();

}

}

public class HuffmanTree

{

private List<Node> nodes = new List<Node>();

public Node Root { get; set; }

public Dictionary<char, int> Frequencies = new Dictionary<char, int>();

public void Build(string source)

{

for (int i = 0; i < source.Length; i++)

{

if (!Frequencies.ContainsKey(source[i]))

{

Frequencies.Add(source[i], 0);

}

Frequencies[source[i]]++;

}

foreach (KeyValuePair<char, int> symbol in Frequencies)

{

nodes.Add(new Node() { Symbol = symbol.Key, Frequency = symbol.Value });

}

while (nodes.Count > 1)

{

List<Node> orderedNodes = nodes.OrderBy(node => node.Frequency).ToList<Node>();

if (orderedNodes.Count >= 2)

{

// Take first two items

List<Node> taken = orderedNodes.Take(2).ToList<Node>();

// Create a parent node by combining the frequencies

Node parent = new Node()

{

Symbol = '\*',

Frequency = taken[0].Frequency + taken[1].Frequency,

Left = taken[0],

Right = taken[1]

};

nodes.Remove(taken[0]);

nodes.Remove(taken[1]);

nodes.Add(parent);

}

this.Root = nodes.FirstOrDefault();

}

}

public BitArray Encode(string source)

{

List<bool> encodedSource = new List<bool>();

for (int i = 0; i < source.Length; i++)

{

List<bool> encodedSymbol = this.Root.Traverse(source[i], new List<bool>());

encodedSource.AddRange(encodedSymbol);

}

BitArray bits = new BitArray(encodedSource.ToArray());

return bits;

}

public string Decode(BitArray bits)

{

Node current = this.Root;

string decoded = "";

foreach (bool bit in bits)

{

if (bit)

{

if (current.Right != null)

{

current = current.Right;

}

}

else

{

if (current.Left != null)

{

current = current.Left;

}

}

if (IsLeaf(current))

{

decoded += current.Symbol;

current = this.Root;

}

}

return decoded;

}

public bool IsLeaf(Node node)

{

return (node.Left == null && node.Right == null);

}

}

public class Node

{

public char Symbol { get; set; }

public int Frequency { get; set; }

public Node Right { get; set; }

public Node Left { get; set; }

public List<bool> Traverse(char symbol, List<bool> data)

{

// Leaf

if (Right == null && Left == null)

{

if (symbol.Equals(this.Symbol))

{

return data;

}

else

{

return null;

}

}

else

{

List<bool> left = null;

List<bool> right = null;

if (Left != null)

{

List<bool> leftPath = new List<bool>();

leftPath.AddRange(data);

leftPath.Add(false);

left = Left.Traverse(symbol, leftPath);

}

if (Right != null)

{

List<bool> rightPath = new List<bool>();

rightPath.AddRange(data);

rightPath.Add(true);

right = Right.Traverse(symbol, rightPath);

}

if (left != null)

{

return left;

}

else

{

return right;

}

}

}

}

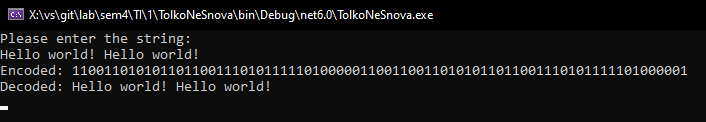


Рисунок 1 – результат работы программы