Huffman encoding algoritması veri sıkıştırmak için tasarlanmış bir algoritmadır. Verilen bir string’deki her karakterin kullanım sıklığını yani frekansını tutan bir ağaç oluşturulur. Ağacın kökünden dallarına doğru frekans azalacak şekilde karakterler ağaca yerleştirilir. Ağacın kökünden yapraklarına doğru ilerlerken her sol dal için 0, sağ dal için 1 kodu verilir. Oluşan bit haritaları verilerin sıkıştırılmış halleridir. Ortalama olarak huffman ağacı kullanılınca bir verinin sıkışman oranı %80 civarındadır.

Genel olarak. jpeg ve .png gibi resim formatlarında, mp3 gibi ses dosya formatında, winzip gibi sıkıştırılmış dosya formatlarında, telefon numaralarımız da dahil olmak üzere huffman encoding algoritması günümüzde çok sık başvurulan bir yöntemdir.

HUFFMAN ENCODING:

HuffmanTree sınıfında Node tipinde düğümleri olan ve bu düğümleri List<> hazır koleksiyon ile tutan HuffmanTree tipinde Huffman ağacımızı yaratıp verimizi içine yerleştirdik. For döngüsü içinde o anki kökün tüm indekslerindeki harfleri Traverse() methodu ile sıkıştırılmış formdaki binary halini List<bool> encodedSource’un sonuna ekliyoruz. Encode() metodunun sonunda da BitArray tipine dönüştürüp geri main’e döndürüyoruz.

HUFFMAN DECODING:

Kök düğümden başlayarak sağda 1 oldukça sağa doğru gidiyoruz ve 1i buluncaya kadarki ikili sembollerin toplamını sembole dönüştürerek ve bir değişkende tutarak birleştiriyoruz. Geri döndürdüğümüz stringimiz programın başında input olarak alınmış olan string ile birebir aynı.

**KAYNAK KODU:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

namespace HuffmanTest

{

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;

}

}

}

}

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);

}

}

class Program

{

static void Main(string[] args)

{

string input = "data structures";

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();

}

}

}