**Autor**: Krzysztof Dąbrowski 293101 25.11.2018

SPRAWOZDANIE  
Kompresja LZW

# Cel zadania

Implementacja algorytmu kompresji LZW oraz obliczenie współczynnika kompresji.

# Repozytorium z kodem

<https://github.com/SiwyKrzysiek/Kompresja-LZW>

# Główne funkcje

* Kompresja
* Dekompresja
* Obliczanie współczynnika kompresji

# Użycie

|  |  |
| --- | --- |
| **Parametr** | **Opis** |
| **-h** | Wyświetla pomoc |
| **-c** <*plik\_wejściowy*> <*plik\_wyjściowy*> | Kompresuje plik wejściowy i zapisuje rezultat w pliku wynikowym |
| **-d** <*plik\_wejściowy*> <*plik\_wyjściowy*> | Dekompresuje plik wejściowy i zapisuje rezultat w pliku wynikowym |
| **-demo** | Uruchamia demonstrację działania programu |

# Dodatkowe informacje

Algorytm korzysta z kodowania znaków UTF-16 przez co każdy kod ma wielkość 32 bitów.

Pozwala to na uzyskanie takich samych wyników niezależnie od języka w jakim został napisany oryginalny tekst. Takie podejście jest jednak mniej efektywne dla tekstów składających się jedynie ze znaków ASCII. Dla takich danych lepiej byłoby zastosować 8-bitowy kod znaku i 16-bitowy kod ciągu.

# Przykład działania

.\LZW\_Compersion.exe -demo

DEMO

--------------------------------------

W ramach demonstracji zostanie przetworzona Iliada Homera i pierwsza część Harego Potera

ILIADA:

Dane z pliku iliada.txt zostały skompresowane i zapisane w pliku iliada.lzw

Wielkość pliku wejściowego: 913 KB

Wielkość pliku wyjściowego: 678 KB

Współczynnik kompresji wynosi: 134,57%

Dane z pliku iliada.lzw zostały zdekompresowane i zapisane w pliku ZDEKOMPRESOWANE\_iliada.txt

HARRY POTTER:

Dane z pliku Harry\_Potter.txt zostały skompresowane i zapisane w pliku Harry\_Potter.lzw

Wielkość pliku wejściowego: 448 KB

Wielkość pliku wyjściowego: 357 KB

Współczynnik kompresji wynosi: 125,63%

Dane z pliku Harry\_Potter.lzw zostały zdekompresowane i zapisane w pliku ZDEKOMPRESOWANE\_Harry\_Potter

# Pliki źródłowe

## Program.cs

namespace LZW\_Compersion

{

class MainClass

{

public static void Main(string[] args)

{

ConsoleInterface.HandleCommandlineArguments(args);

}

}

}

## LZW.cs

using System;

using System.Collections.Generic;

using System.Text;

namespace LZW\_Compersion

{

using DataBlock = UInt32; //Ustalenie jak będą zapisywane kody

public static class LZW

{

/// <summary>

/// Kompresuje ciąg znaków algorytmem LZW

/// </summary>

/// <param name="data">Dane do kompresji</param>

/// <returns>Tablica bloków kodu skompresowanych danych</returns>

public static DataBlock[] Compress(string data)

{

if (DataBlock.MaxValue <= char.MaxValue) //Upewnienie się, że typ danych na kody jest dbrze wybrany

throw new Exception("To small DataBlock set in code");

if (data.Length == 0) return new DataBlock[0]; //Edge case dla pustych danych

List<DataBlock> result = new List<DataBlock>(); //Wynik kodowania

Dictionary<string, DataBlock> dictionary = CreateDictionary(); //Utworzenie początkowego słownika

string currentSequence = data[0].ToString(); //Wczytanie pierwszego znaku do kolejki aktualnie kodowanych znaków

for (int i = 1; i < data.Length; i++) //Od 2 elementu, bo piewszy już został wczytany

{

string sequenceWithNewSymbol = currentSequence + data[i]; //Ciąg z aktualnie przetwarzanym znakiem

if (dictionary.ContainsKey(sequenceWithNewSymbol)) //Sprawdzenie czy przedłużony ciąg jest w słowniku

{

currentSequence = sequenceWithNewSymbol; //Jeśli aktualny ciąg jest już w słowniku to dodajemy aktualny znak do aktualnego ciągu

}

else //Nie ma kodu dla rozszeżonego ciągu

{

result.Add(dictionary[currentSequence]); //Wypisanie kodu dla ciągu, który do tej pory był w słowniki

dictionary.Add(sequenceWithNewSymbol, (DataBlock)dictionary.Count); //Dodanie nowego ciągu do słownika

currentSequence = data[i].ToString(); //Zresetowanie ciągu kodowanych znaków

}

}

result.Add(dictionary[currentSequence]); //Dopisanie pozosatałego ciągu znaków

return result.ToArray();

}

/// <summary>

/// Dekompresuje dane skompresowane algorytmeme LZW

/// </summary>

/// <param name="compressedData">Tablica kodów znaków</param>

/// <returns>Rozkodowany ciąg</returns>

public static string Decompress(DataBlock[] compressedData)

{

if (DataBlock.MaxValue <= char.MaxValue) //Upewnienie się, że typ danych na kody jest dbrze wybrany

throw new Exception("To small DataBlock set in code");

if (compressedData.Length == 0) //Obsłużenie pustych danych

return "";

StringBuilder result = new StringBuilder(); //Zmienna na budowanie wyniku dekompresji

Dictionary<DataBlock, string> dictionary = CreateDecmpressionDictionary(); //Utworzenie słownika do dekompresjii

DataBlock previousCode = compressedData[0]; //Na początek wczytuję pierwszy kod

result.Append(dictionary[previousCode]); //Wypisanie znaku zakodowanego pierwszym kodem

for (int i = 1; i < compressedData.Length; i++) //Dopóki są jeszcze kody

{

DataBlock newCode = compressedData[i]; //Zapisanie aktualnie przetwarzanego kodu

string previousSymbol = dictionary[previousCode]; //Zobaczenie jaki ciąg znaków kodował poprzedni kod

if (dictionary.ContainsKey(newCode)) //Jeżeli słownik zawiera aktualnie wczytany kod

{

dictionary.Add((DataBlock)dictionary.Count, previousSymbol + dictionary[newCode][0]); //Dodanie do słownika kodu popszedniego znaku wraz z pierszym znakiem zdekodowanego ciągu

result.Append(dictionary[newCode]); //Wypisanie znaku kodowanego przez rozpoznany kod

}

else //Jeżeli słwonik nie zawiera aktualnie wczytanego kodu

{

dictionary.Add((DataBlock)dictionary.Count, previousSymbol + previousSymbol[0]); //Musiała wystąpić sytuacja, gdzie ciąg został rozszeżony o jeden znak

//Dopisujemy więc taki ciąg do słownika

result.Append(previousSymbol + previousSymbol[0]); //Wypisanie odkodowanego ciągu

}

previousCode = newCode; //Poprzedni kod to będzie aktualnie przetważany kod

}

return result.ToString(); //Finalne zbudowanie stringa

}

/// <summary>

/// //Wypełana słownik pojedyńczymi znakami

/// </summary>

/// <returns>Słownik do kompresji</returns>

private static Dictionary<string, DataBlock> CreateDictionary()

{

Dictionary<string, DataBlock> dictionary = new Dictionary<string, DataBlock>(); //Utworzenie pustego słownika

for (char i = char.MinValue; i < char.MaxValue; i++) //Wpisanie wszystkich możliwych pojednyńczych znaków do słownika

dictionary.Add(i.ToString(), (DataBlock)dictionary.Count); //Generowanie kodów polega na przypisaniu aktualnej wielkości słownika

//Ponieważ słownik nigdy nie maleje kody są unikalne

return dictionary;

}

/// <summary>

/// Wypełana słownik kodami pojedyńczych znaków

/// </summary>

/// <returns>Słownik do dekompresjii</returns>

private static Dictionary<DataBlock, string> CreateDecmpressionDictionary()

{

Dictionary<DataBlock, string> dictionary = new Dictionary<DataBlock, string>(); //Utworzenie pustego słownika

for (char i = char.MinValue; i < char.MaxValue; i++) //Wpisanie do słownika kodów wszystkich pojedyńczych znaków

dictionary.Add((DataBlock)dictionary.Count, i.ToString()); //Generowanie kodów polega na przypisaniu aktualnej wielkości słownika

//Ponieważ słownik nigdy nie maleje kody są unikalne

return dictionary;

}

}

}

## ConsoleInterface.cs

using System;

using System.IO;

using System.Collections.Generic;

namespace LZW\_Compersion

{

using DataBlock = UInt32;

public static class ConsoleInterface

{

public static void HandleCommandlineArguments(string[] args)

{

switch (args[0])

{

case "-c":

TryToCompress(args[1], args[2]);

break;

case "-d":

TryToDecompress(args[1], args[2]);

break;

case "-demo":

RunDemo();

break;

case "-h":

default:

DisplayHelp();

break;

}

}

private static void DisplayHelp()

{

string helpMessage =

"POMOC\n\n" +

"| Parametr | Opis |\n" +

"|--------------------------------------|-------------------------------------------------------------------|\n" +

"| -h | Wyświetla pomoc |\n" +

"| -c <plik\_wejściowy> <plik\_wyjściowy> | Kompresuje plik wejściowy i zapisuje rezultat w pliku wynikowym |\n" +

"| -d <plik\_wejściowy> <plik\_wyjściowy> | Dekompresuje plik wejściowy i zapisuje rezultat w pliku wynikowym |\n" +

"| -demo | Uruchamia demonstrację działania programu |\n";

Console.WriteLine(helpMessage);

}

private static void RunDemo()

{

Console.WriteLine("DEMO\n--------------------------------------\n");

string iliada = "iliada.txt", compressed1 = "iliada.lzw", decompressed1 = "ZDEKOMPRESOWANE\_iliada.txt";

string harryPotter = "Harry\_Potter.txt", compressed2 = "Harry\_Potter.lzw", decompressed2 = "ZDEKOMPRESOWANE\_Harry\_Potter";

if (!File.Exists(iliada))

{

Console.WriteLine($"Demo wymaka pliku {iliada} w katalogu z programem");

}

Console.WriteLine("W ramach demonstracji zostanie przetworzona Iliada Homera i pierwsza część Harego Potera\n");

Console.WriteLine("ILIADA: \n");

TryToCompress(iliada, compressed1);

TryToDecompress(compressed1, decompressed1);

Console.WriteLine("\nHARRY POTTER: \n");

TryToCompress(harryPotter, compressed2);

TryToDecompress(compressed2, decompressed2);

}

private static void TryToCompress(string inputFile, string outputFile)

{

string data;

try

{

data = File.ReadAllText(inputFile);

if (data.Length <= 0) throw new IOException();

}

catch (FileNotFoundException)

{

Console.WriteLine("Nie udało się odnaleźć wskazanego pliku");

return;

}

catch (Exception)

{

Console.WriteLine("Wystąpił błąd podczas czytana pliku z danymi");

return;

}

DataBlock[] compressed = LZW.Compress(data);

try

{

using (BinaryWriter writer = new BinaryWriter(File.Open(outputFile, FileMode.Create)))

{

foreach (DataBlock block in compressed)

{

writer.Write(block);

}

}

}

catch (Exception e)

{

Console.WriteLine("Wystąpił błąd podczas zapisu do pliku");

return;

}

DisplayCompressionInfo(inputFile, outputFile);

}

private static void TryToDecompress(string inputFile, string outputFile)

{

List<DataBlock> data = new List<DataBlock>();

try

{

using (BinaryReader reader = new BinaryReader(File.Open(inputFile, FileMode.Open)))

{

long position = 0;

long length = reader.BaseStream.Length;

while (position < length)

{

data.Add(reader.ReadUInt32());

position += sizeof(DataBlock);

}

}

}

catch (FileNotFoundException)

{

Console.WriteLine("Nie udało się odnaleźć wskazanego pliku");

return;

}

catch (Exception)

{

Console.WriteLine("Wystąpił błąd podczas czytana pliku z danymi");

return;

}

string decompressed = LZW.Decompress(data.ToArray());

try

{

File.WriteAllText(outputFile, decompressed);

}

catch (Exception)

{

Console.WriteLine("Wystąpił błąd podczas zapisu do pliku");

return;

}

DisplayDecompressionInfo(inputFile, outputFile);

}

private static void DisplayDecompressionInfo(string inputFile, string outputFile)

{

Console.WriteLine($"Dane z pliku {inputFile} zostały zdekompresowane i zapisane w pliku {outputFile}\n");

}

private static void DisplayCompressionInfo(string inputFile, string outputFile)

{

long inputFileSize = new FileInfo(inputFile).Length;

long outputFileSize = new FileInfo(outputFile).Length;

double compressionDegree = (double)inputFileSize / outputFileSize \* 100;

Console.WriteLine($"Dane z pliku {inputFile} zostały skompresowane i zapisane w pliku {outputFile}");

Console.WriteLine($"Wielkość pliku wejściowego: {inputFileSize / 1000} KB");

Console.WriteLine($"Wielkość pliku wyjściowego: {outputFileSize / 1000} KB");

Console.WriteLine($"Współczynnik kompresji wynosi: {compressionDegree:0.00}%\n");

}

}

}