Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

**МЕХАНІКО-ТЕХНОЛОГІЧНИЙ ФАКУЛЬТЕТ**

Кафедра програмування та захисту інформації

Звіт

з виконаної лабораторної роботи № 4

дисципліни “ Технології розробки алгоритмів ”

на тему

“ Алгоритми пошуку. Хешування ”

Виконав

студент академічної

групи КІ-15

Аннаев А

Перевірив

Викл.

Гермак В.С.

Кропивницкий- 2017

Лабораторная работа №4

Тема: Алгоритмы поиска. хеширования

Цель: Рассмотреть алгоритмы поиска, исследовать целесообразность применения различных алгоритмов в конкретных случаях исходя из их эффективности. Рассмотреть понятие хэш-функции, суть и цели процесса хеширования.

Ход работы

Вариант 8

Задание: Реализовать алгоритм Боуера-Мура поиска слова в тексте.

static void Main(string[] args)

{

try

{

string[] readText = File.ReadAllLines(@"E:\file.txt");

string text = String.Join(" ", readText);

string findWord = Console.ReadLine();

BoyerMoore bm = new BoyerMoore(findWord);

foreach (int index in bm.BoyerMooreMatch(text))

Console.WriteLine("Matched at index {0}", index);

}

catch { }

Console.ReadKey();

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Labarator\_4\_1

{public class BoyerMoore

{

private int[] m\_badCharacterShift;

private int[] m\_goodSuffixShift;

private int[] m\_suffixes;

private string m\_pattern;

/// <summary>

/// Constructor

/// </summary>

/// <param name="pattern">Pattern for search</param>

public BoyerMoore(string pattern)

{

/\* Preprocessing \*/

m\_pattern = pattern;

m\_badCharacterShift = BuildBadCharacterShift(pattern);

m\_suffixes = FindSuffixes(pattern);

m\_goodSuffixShift = BuildGoodSuffixShift(pattern, m\_suffixes);

}

/// <summary>

/// Build the bad character shift array.

/// </summary>

/// <param name="pattern">Pattern for search</param>

/// <returns>bad character shift array</returns>

private int[] BuildBadCharacterShift(string pattern)

{

int[] badCharacterShift = new int[256];

for (int c = 0; c < badCharacterShift.Length; ++c)

badCharacterShift[c] = pattern.Length;

for (int i = 0; i < pattern.Length - 1; ++i)

badCharacterShift[pattern[i]] = pattern.Length - i - 1;

return badCharacterShift;

}

/// <summary>

/// Find suffixes in the pattern

/// </summary>

/// <param name="pattern">Pattern for search</param>

/// <returns>Suffix array</returns>

private int[] FindSuffixes(string pattern)

{

int f = 0, g;

int patternLength = pattern.Length;

int[] suffixes = new int[pattern.Length + 1];

suffixes[patternLength - 1] = patternLength;

g = patternLength - 1;

for (int i = patternLength - 2; i >= 0; --i)

{

if (i > g && suffixes[i + patternLength - 1 - f] < i - g)

suffixes[i] = suffixes[i + patternLength - 1 - f];

else

{

if (i < g)

g = i;

f = i;

while (g >= 0 && (pattern[g] == pattern[g + patternLength - 1 - f]))

--g;

suffixes[i] = f - g;

}

}

return suffixes;

}

/// <summary>

/// Build the good suffix array.

/// </summary>

/// <param name="pattern">Pattern for search</param>

/// <returns>Good suffix shift array</returns>

private int[] BuildGoodSuffixShift(string pattern, int[] suff)

{

int patternLength = pattern.Length;

int[] goodSuffixShift = new int[pattern.Length + 1];

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

goodSuffixShift[i] = patternLength;

int j = 0;

for (int i = patternLength - 1; i >= -1; --i)

if (i == -1 || suff[i] == i + 1)

for (; j < patternLength - 1 - i; ++j)

if (goodSuffixShift[j] == patternLength)

goodSuffixShift[j] = patternLength - 1 - i;

for (int i = 0; i <= patternLength - 2; ++i)

goodSuffixShift[patternLength - 1 - suff[i]] = patternLength - 1 - i;

return goodSuffixShift;

}

/// <summary>

/// Return all matched of the pattern in the specified text using the .NET String.indexOf API

/// </summary>

/// <param name="text">text to be searched</param>

/// <param name="startingIndex">Index at which search begins</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> BCLMatch(string text, int startingIndex)

{

int patternLength = m\_pattern.Length;

int index = startingIndex;

do

{

index = text.IndexOf(m\_pattern, index, StringComparison.InvariantCultureIgnoreCase);

if (index < 0)

yield break;

yield return index;

index += patternLength;

} while (true);

}

/// <summary>

/// Return all matched of the pattern in the specified text using the .NET String.indexOf API

/// </summary>

/// <param name="text">text to be searched</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> BCLMatch(string text)

{

return BCLMatch(text, 0);

}

/// <summary>

/// Return all matches of the pattern in specified text using the Horspool algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <param name="startingIndex">Index at which search begins</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> HorspoolMatch(string text, int startingIndex)

{

int patternLength = m\_pattern.Length;

int textLength = text.Length;

/\* Searching \*/

int index = startingIndex;

while (index <= textLength - patternLength)

{

int unmatched;

for (

unmatched = patternLength - 1;

unmatched >= 0 && m\_pattern[unmatched] == text[unmatched + index];

--unmatched

)

; // empty

if (unmatched < 0)

yield return index;

index += m\_badCharacterShift[text[unmatched + patternLength - 1]];

}

}

/// <summary>

/// Return all matches of the pattern in specified text using the Horspool algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> HorspoolMatch(string text)

{

return HorspoolMatch(text, 0);

}

/// <summary>

/// Return all matches of the pattern in specified text using the Boyer-Moore algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <param name="startingIndex">Index at which search begins</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> BoyerMooreMatch(string text, int startingIndex)

{

int patternLength = m\_pattern.Length;

int textLength = text.Length;

/\* Searching \*/

int index = startingIndex;

while (index <= textLength - patternLength)

{

int unmatched;

for (unmatched = patternLength - 1;

unmatched >= 0 && (m\_pattern[unmatched] == text[unmatched + index]);

--unmatched

)

; // empty

if (unmatched < 0)

{

yield return index;

index += m\_goodSuffixShift[0];

}

else

index += Math.Max(m\_goodSuffixShift[unmatched],

m\_badCharacterShift[text[unmatched + index]] - patternLength + 1 + unmatched);

}

}

/// <summary>

/// Return all matches of the pattern in specified text using the Boyer-Moore algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> BoyerMooreMatch(string text)

{

return BoyerMooreMatch(text, 0);

}

/// <summary>

/// Return all matches of the pattern in specified text using the Turbo Boyer-Moore algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <param name="startingIndex">Index at which search begins</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> TurboBoyerMooreMatch(string text, int startingIndex)

{

int patternLength = m\_pattern.Length;

int textLength = text.Length;

/\* Searching \*/

int index = startingIndex;

int overlap = 0;

int shift = patternLength;

while (index <= textLength - patternLength)

{

int unmatched = patternLength - 1;

while (unmatched >= 0 && (m\_pattern[unmatched] == text[unmatched + index]))

{

--unmatched;

if (overlap != 0 && unmatched == patternLength - 1 - shift)

unmatched -= overlap;

}

if (unmatched < 0)

{

yield return index;

shift = m\_goodSuffixShift[0];

overlap = patternLength - shift;

}

else

{

int matched = patternLength - 1 - unmatched;

int turboShift = overlap - matched;

int bcShift = m\_badCharacterShift[text[unmatched + index]] - patternLength + 1 + unmatched;

shift = Math.Max(turboShift, bcShift);

shift = Math.Max(shift, m\_goodSuffixShift[unmatched]);

if (shift == m\_goodSuffixShift[unmatched])

overlap = Math.Min(patternLength - shift, matched);

else

{

if (turboShift < bcShift)

shift = Math.Max(shift, overlap + 1);

overlap = 0;

}

}

index += shift;

}

}

/// <summary>

/// Return all matches of the pattern in specified text using the Turbo Boyer-Moore algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> TurboBoyerMooreMatch(string text)

{

return TurboBoyerMooreMatch(text, 0);

}

/// <summary>

/// Return all matches of the pattern in specified text using the Apostolico-GiancarloMatch algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <param name="startingIndex">Index at which search begins</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> ApostolicoGiancarloMatch(string text, int startingIndex)

{

int patternLength = m\_pattern.Length;

int textLength = text.Length;

int[] skip = new int[patternLength];

int shift;

/\* Searching \*/

int index = startingIndex;

while (index <= textLength - patternLength)

{

int unmatched = patternLength - 1;

while (unmatched >= 0)

{

int skipLength = skip[unmatched];

int suffixLength = m\_suffixes[unmatched];

if (skipLength > 0)

if (skipLength > suffixLength)

{

if (unmatched + 1 == suffixLength)

unmatched = (-1);

else

unmatched -= suffixLength;

break;

}

else

{

unmatched -= skipLength;

if (skipLength < suffixLength)

break;

}

else

{

if (m\_pattern[unmatched] == text[unmatched + index])

--unmatched;

else

break;

}

}

if (unmatched < 0)

{

yield return index;

skip[patternLength - 1] = patternLength;

shift = m\_goodSuffixShift[0];

}

else

{

skip[patternLength - 1] = patternLength - 1 - unmatched;

shift = Math.Max(m\_goodSuffixShift[unmatched],

m\_badCharacterShift[text[unmatched + index]] - patternLength + 1 + unmatched

);

}

index += shift;

for (int copy = 0; copy < patternLength - shift; ++copy)

skip[copy] = skip[copy + shift];

for (int clear = 0; clear < shift; ++clear)

skip[patternLength - shift + clear] = 0;

}

}

/// <summary>

/// Return all matches of the pattern in specified text using the Apostolico-GiancarloMatch algorithm

/// </summary>

/// <param name="text">text to be searched</param>

/// <returns>IEnumerable which returns the indexes of pattern matches</returns>

public IEnumerable<int> ApostolicoGiancarloMatch(string text)

{

return ApostolicoGiancarloMatch(text, 0);

}

}

}

Выводы:

В ходе выполнения лабораторной работы № 4 я рассмотрел алгоритмы поиска, исследовал, целесообразность применения различных алгоритмов в конкретных случаях исходя из их эффективности. Рассмотрел понятие хэш-функции, суть и цели процесса хеширования, реализовал алгоритм Боуера-Мура поиска слова в тексте.