Implementation of the Huffman Coding Algorithm in Windows 10 IoT Core

Alexandru-Catalin Petrini, Valeriu-Manuel Ionescu
Faculty of Electronics, Communications and Computer Science
University of Pitesti
Pitesti, Romania
valeriu.ionescu@upit.ro

Abstract – Raspberry Pi is a platform that is used in many educational projects because of its cost and connectivity options and uses mainly Linux as an operating system. Windows 10 IoT is the operating system proposed by Microsoft for this platform that proposes new frameworks for application development. This paper tests and compares the performance of .NET Framework and .Net Core Framework on a Raspberry Pi running Windows 10 IoT Core operating system. The tests are based on the C# implementation of Huffman Algorithm and offer an example for undergraduate students to test, observe different software solution performance and learn to draw conclusions in a real-world scenario.

Keywords – Huffman coding; algorithm; performance; Windows IoT; Net Core; frameworks; WinRT; C#; Raspberry Pi

I. INTRODUCTION

Nowadays, there are many online tutorials about implementing interesting, smart projects with cheap electronic components. It is a perfect environment for people of all ages to learn, have fun, or actually solve important annoyances in their life. It is a perfect medium especially for students to get in touch with technology and become creative in a pleasant, but very useful way. The market is flooded by microcontrollers and microcomputers ranging from very affordable prices to expensive ones that offer the means to start prototyping projects by themselves, such as household or greenhouse automation, drones, robots and other machines. The internet challenges them to integrate with the web and enrich the universe of the internet of things. ARM processors are the leading part of a cheap class of microcomputers that can animate their systems, and Raspberry Pi together with Arduino microcontroller boards make learning fun, easy and affordable. Teachers must take advantage of this perfect situation and prepare their students for the technicalities they will meet at their future working places.

Arduino and Raspberry Pi started a great boom on the market as open-source projects, in such a way that Microsoft has decided to join the other Linux operating systems with its own freely distributed, closed source, Windows 10 IoT Core and open up, slowly, .NET Framework to the public. The popular Raspberry Pi was certainly a target for them, among MinnowBoard MAX and DragonBoard 410c [1].

As a teacher or student, it is important to determine which the best software tools to use are. This is also a question that professionals in the industry have to decide upon. Therefore it is useful to exemplify a performance test to initiate students in testing, drawing conclusions and making professional choices.

In this paper we test the performance of Huffman Algorithm on Windows 10 IoT Core – installed on a Raspberry Pi 2 microcomputer. This Microsoft ecosystem presents little choices to make, however we will make our point in analyzing the performance of Huffman Algorithm implemented in C# for the .NET Framework and the newly .Net Core. The latter is an open source project based on already open-sourced parts of present .NET Framework. It is also a community project in early development stage [2], but destined to replace the original framework. This paper is structured as follows: Section II describes Huffman compression algorithm. Section III presents aspects related to Microsoft Windows 10 IoT operating system. Section IV presents the modifications of the code in order to make it compatible with Windows 10 IoT Core C# supported solutions. Section V presents and discusses test results. Section VI formulates conclusions and future applications of this example.

II. Huffman Algorithm

Huffman Coding is an optimal compression algorithm that uses codes (words or bit sequences) of variable length to replace individual symbols (like characters). The length of Huffman codes is related to the frequency of the characters in the text to be compressed. Symbols that have a high appearance rate are assigned smaller words than those that those with a more rare appearance. Huffman algorithm is based on the prefix property; it designates words so that no two messages will have an identical arrangement of the coding digits and no supplementary information is needed to determine where the message coding begins and ends after the message sequence starting in determined [3]. This is how Huffman Coding ensures that there is no uncertainty when decoding the generated bit stream. Huffman algorithm takes place in the Greedy algorithm class defined by choosing the best solution at any step; because of the non ambiguous code it creates, it is considered a lossless algorithm.
Figure 1. Document compression and decompression. Lossless compression requires $D = D'$. 

<table>
<thead>
<tr>
<th>Initial Document $D$</th>
<th>Compress</th>
<th>Compressed Document $C$</th>
<th>Decompress</th>
<th>Reconstituted Document $D'$</th>
</tr>
</thead>
</table>

TABLE I. ALPHABET OF UNIQUE CHARACTERS WITH THEIR FREQUENCIES

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>45%</td>
<td>10%</td>
<td>13%</td>
<td>12%</td>
<td>9%</td>
<td>5.00%</td>
</tr>
</tbody>
</table>

TABLE II. ALPHABET OF UNIQUE CHARACTERS WITH THEIR FREQUENCIES

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

This means that a compressed file will be restored, through decompression, at exactly the same initial state (Fig. 1).

The algorithm is comprised of two parts:

1) **Build the Huffman Tree.** The Huffman Tree is a binary tree build in bottom-up manner, rather that the traditional top-down.

2) **Traverse the Huffman Tree** in pre-order fashion to obtain codes and designate them to symbols.

Building the Huffman Tree requires the following assumption and steps:

Let the input be an array of characters. After obtaining the unique characters and their frequency of occurrences – as illustrated in Table I – one must obtain the Huffman Tree.

1) Organize the alphabet containing unique characters in a min heap formed of leaf nodes, where a leaf node corresponds to a character and its frequency. The min heap acts as a priority queue in which the frequency field value is used for comparing every two nodes. The least frequent character is root for the min heap.

2) Find and extract two nodes with the minimum frequency in the heap.

3) Form a new node where the frequency field value is the sum of the two nodes frequencies and the left child is first extracted node, respectively the right child the second node. Insert this node into the min heap.

4) Repeat 2 and 3 until the heap is formed of a single node. This node is the root (100% node) of the complete Huffman Tree (Fig. 2).

Fig. 2 presents visually the algorithmic evolution of the Huffman Tree, and Fig. 3 shows how Huffman Codes are formed.

Although very common in the academic environment as a basic algorithm, Huffman coding is present in all mainstream compression formats of today, from GZIP, PKZIP, WinZip and BZIP2, etc. to multimedia formats like JPEG, PNG, MPEG and MP3.

III. MICROSOFT WINDOWS 10 IOT CORE

A. Windows 10 IoT Family

Windows 10 IoT is a family of Windows 10 editions that continues the embedded tradition from Microsoft [5], [6]. It targets intelligent devices like industrial gateways, point of sales terminals or ATMs.
There are three Windows 10 IoT editions in this family:

- **Windows 10 IoT Enterprise** – which is an x86 version of Windows optimized to run unattended inside a non PC device, like POS terminal, kiosk or outdoor display. Supports both Universal Windows (Metro) applications and Classic Windows (desktop) applications. Windows 10 IoT Enterprise is a direct descendant of Windows Embedded OS family.

- **Windows 10 IoT Mobile Enterprise** – relates directly to Windows 10 Mobile OS which powers Windows Phones. It tracks back to Windows 8 Phone and Windows CE. Windows 10 Iot Mobile Enterprise is designed for enterprise mobile. It provides native support for bar-code scanning (such as handheld barcode scanners) and other peripherals having high security features implemented.

- **Windows 10 IoT Core** – represents a completely new strategy from Microsoft as it is targeted towards a hobbyist audience that deals with simple, cheap, low powered kit computers like Raspberry Pi. Windows IoT supports only Windows Universal applications for maximum portability and, because of hardware performance limitations, ships with a reduced WinRT stack (Windows Runtime [5], [7]).

**B. Windows 10 IoT Core – Advantages and Limitations**

Windows IoT was introduced by Microsoft in 2015 to promote and help develop its IoT trend. It has been released as free to use and offers a complementary development solution to Linux distribution variants designed for the following hardware platforms [1]:

- Raspberry Pi 2 and 3
- MinnowBoard MAX
- DragonBoard 410c

Of all the above mentioned, Raspberry Pi is the most popular today, due to its performance over price performance ratio. It is also a highly marketed platform together with the Arduino and their compatible accessories (robotic kits; sensors for measuring humidity, temperature, gyroscopes; communication modules: Bluetooth, XBee (ZigBee protocol), traditional wireless and Ethernet, etc; relays, motors, motor drivers and many others).

Linux distributions usually support well such hardware due to the open-source nature of the projects and the dedication of community members, not to mention the release of model A back in February 2012. From then, many libraries and projects have been showcased on the internet, especially for the official distribution Raspbian, but not restricted to by the free nature of Linux.

Microsoft Windows 10 IoT comes with a big handicap for now, as a fair recent appearance, and as a closed source proprietary OS. It is limited by the reduced in features WinRT (Sec. III-A) and the yet very little documented functions and libraries (not even a commercial or free book ever met in our searches). It must be mentioned though, that Microsoft Developer Network Forum is getting filled with questions and requests on Windows IoT [8] and hopefully more answers or solutions will be available, such as [9] in which a FTDI unsupported driver issue is getting solved.

The great advantage of Windows 10 IoT Core resides in its tight present and future integration with Windows 10 technologies: the actual .NET Framework, the newly .Net Core and, most importantly, the passionate Windows developer community that grows around it. .NET Framework has opened, over time, perspectives for integrating and/or porting open technologies; also, new features are being added in Insider Previews, like the Remote Display Experience (Fig. 3) [10].

Although we feel there is much hope for Windows IoT Core, we would like to mention some negative remarks; in our experience we have encountered problems such as:

- We tried to find or compile ARM versions of Java and Mono – therefore, we were unable to make a fair comparison between Linux and Windows IoT Core;

- WindowsRT reduced feature set, security and task oriented programming, made cross-platform programming difficult. It presents:
  - no support for System.IO.Console.* functions although they are supported by Mono ARM for Linux;
  - isolated application security feature [11] made writing and reading to/from SD Card impossible the usual fashion; instead we had to use async read/write functions and Windows.Storage.* namespace classes;
  - very poor command-line monitoring support for data collection. One should write specific application for this;

- capturing physical keyboard in an application and reading files from an external FTP server proved much harder tasks than it should be;

- there is no similar VNC option to access the screen, but starting with Window IoT OS Version 10.0.14295.1000 [10];

- there is no support for WinRT C# Console applications and, in fact, one cannot use System.IO.Console.* functions. An alternative would be to create a C++ WinRT container application to run C# code;

- CoreCLR ASP.NET 5 support from .Net Core can be regarded as beta stage development process;

- limited official hardware support [12] – for ex., many wireless adapters will not work;
– other bugs and issues are also reported massively on the MSDN forum when searching for Windows IoT section [8].

Figure 4. Using Remote Display Experience on Windows 10+ tablet to control a motor and a Raspberry Pi 3 through touch and accelerometer, video [10]

C. .NET and .Net Core Frameworks

.Net Core is the future of .NET, but not the present. Today, Windows 10 IoT Core supports only the .NET Universal Windows applications build on top of WinRT [10].

WinRT is designed to provide hardware access (for ex.: GPIO pins and touch-based scenarios) and is optimized as such.

.Net Core may be used to run console applications on Windows 10 IoT, but this can be done only by using ASP.NET 5, essentially a build for cloud/server use. .Net Core is cross-platform by design, though this feature is implemented with help from Mono in Linux.

It is also early development stage and CoreCLR runtimes is in beta stage.

Bottom line, we are comparing frameworks (parts of frameworks) designed for different purposes, stable vs. unstable, build mostly for Windows vs. cross-platform, specialized vs. universal use.

IV. HUFFMAN SOFTWARE APPLICATION

The code of the application from [13] was modified as follows:

1. We removed elements that could obstruct performance of the algorithm:

   The script model of the application – the original application was composed of two applications itself: the main one – with the algorithm – designed to be used as a console command and the graphical front-end, that would use the before mentioned and further do some estimate calculus on the results. Therefore, we kept the command-line program and hard coded the tests in Main() function.

2. We measured time of execution strictly for algorithm sections.

3. Necessary code adaptations were made for writing to the SD card on Windows IoT, keeping in mind that the execution of the algorithm is sequential.

4. We minimized static memory access – by writing data into memory streams instead of sending them directly on the MicroSD card. Upon algorithm completion, memory streams are flushed onto the card. Access of static memory is made only outside of the algorithm sections.

Fig. 5 and Fig. 6 present the implemented flow diagrams for the encoder and the decoder of the application. The encoding algorithm (Fig. 5) creates a header/body structure for the resulting data.

The information that is stored in the header is composed of: alphabet size; length expressed as multiple of 16 bits; the number of bits used for padding the final 16 bits; alphabet characters and their frequency.

The header data is used by the decoding algorithm in order to properly extract the encoded information.
VI. IMPLEMENTATION RESULTS

In our application we measured execution time for Huffman algorithm over 50 iteration; we ran the tests on two text files (Fig. 7). Fig. 8 and Fig. 9 present the mean and standard deviation results. Standard deviation is a statistical measure that quantifies the amount of variation for a dataset, and a low value indicates that the data values tend to be close to the mean. It is a measure of stability. From these one can remark the following:

- Native WinRT C# obtains the lowest time values. It also makes the most stable runs.
- .Net Core headed and headless (without graphical interface) runs present a large degree of instability, besides much higher value readings.
- Contrary to any expectation, headless .Net Core runs are less stable and less time performant then the headed results (Fig. 8, 9). This is more visible on first test file which is larger in dimensions.
- Stability decreases with test file size.
- The best choice for developing an application that runs on Windows 10 IoT Core is to be based on .NET WinRT, because of stability, time performance, and the tight operating system integration.

VI. CONCLUSIONS

This paper presented an educational manner for testing performance of two provided software solutions on the same operating system.

This example can be further extended by testing different types of algorithmic applications, added with data collection capability code, or specialized testing software, for professional purposes.
Our tests show that the better solution is the one provided implicitly by the operating system provider (C# WinRT), a possible cause being that it is a stable version and the tight integration with the programming tools.

The research and results presented in this paper can be used as bases for undergraduate laboratories or for student’s homework projects.

REFERENCES


