Assignment_4_Report

109503517_通訊三_蔡雅各

1. 編譯結果

```
jacob@jacob-VirtualBox:~/桌面/hw4/arithmetic_coding$ gcc -o arcd arcd_stream.c arcd.c arcd.h adaptive_model.c adaptive_model.h
arcd.h:1:9: warning: #pragma once in main file
    1 | #pragma once
      |         ^~~~
adaptive_model.h:1:9: warning: #pragma once in main file
    1 | #pragma once
      |         ^~~~
jacob@jacob-VirtualBox:~/桌面/hw4/arithmetic_coding$ cd ..
jacob@jacob-VirtualBox:~/桌面/hw4$ cd huffman_coding
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ gcc -o huffman huffcode.c huffman.c huffman.h
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$
```

Huffman 及 arithmetic 皆須搜尋到指定目錄，並以 gcc 指令做 compile

2. 執行結果

測試文件檔為一文章含 1619 個字元，皆儲存在同一行，如下圖所示。

```
1 In today's world, information is one of the most valuable assets, but there is a major threat to
  it by the evolving sophisticated malware (malicious software). The first malware was created for
  fun, but now its a profit-driven industry [1] and in the last couple of years advanced malware
  uses complex obfuscation methods viz. control/data flow permutation, compression, heap spray,
  etc. [2] to evade the detection techniques. Also, there is an exponential increase in the number
  of malware released every year. To detect these malware, various techniques are proposed in the
  past. These methods range from the early day signature-based detection to Machine Learning
  techniques [3]. However, it appears that proposed approaches are not sufficient to detect the
  unknown sophisticated malware. Thus researchers are continuously searching for better techniques
  to detect the advanced malware and have started applying machine learning techniques for malware
  detection. These algorithms can detect obfuscated malware and have potential to handle both
  complexity and scale problem like data pre-processing, feature extraction/selection, model
  creation and testing. In this Santos et. al claims to obtain an accuracy of ~96% using opcode
  frequency as features [4]. Later on, researchers also applied file-size based segmentation on
  opcode frequency and improved the average accuracy using different Machine Learning models and
  claimed that Random Forest can provide the accuracy up to ~98% with a False Positive Rate (FPR)
  of ~1.07% [5]. For the analysis they used the benchmark malware samples from the Malicia project
  [6].
2
```

Arithmetic coding 測試結果如下圖。在基本的執行及編碼命令後加上 bash 指令來輸入及輸出檔案，編碼內容及程式運行時間皆會記錄於檔案中及 terminal 印出。

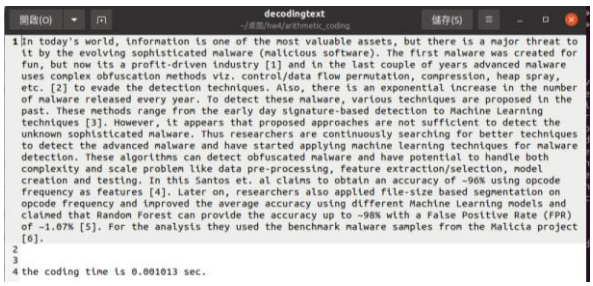
```
jacob@jacob-VirtualBox:~/桌面/hw4/arithmetic_coding$ ./arcd -e <testarticle | tee encodingtext
I$^Ъ½Ue"2孵e}eeeeeie<e:ejeoe
                          I     ze"eQee"me66vB8e$eheRm^?eee-e5eeJeeeeee#eeee(eg
"eeVeQeeVejeXeeeveeie6feNe8ee>2eeexeLeYq'&eee]esa差^@eexie(See;eO6eAeex\Pe
7TF!ceedeeep`eU0z*(e1"ge=4eMQxn3eee4I~
                                  eXp{ewcS|S◆eSeee\eexe^eeGeeRix0eI
e'eeVeeceetW@eS|eeeRey/eEe        eetGee:eeXeZe0eDe+eCeeeleeeeeeYpeO8jeee9e괻Ce@e8!e*e3XeOeeOeeqeee0e*eeee%e
Seeeh?e⊐eV'eeeTebemex!'ee         aegee4eeee=eie0fre      ^eeeeleeeXM$yeΠeee|;wBere,e];=vJe1O
                                                                                    qe!1(
                                                                                        e9ee+eeem7eS뾢 eeee8eweyM
eeeee~i>         Gee~
|heepe}Ne/Teeee,eeee/eeue!OB◆◆
                          eee$]See;\eOeeee1qeS+eee` `ee<eMz=eeeeBeeqIeUeZh^e,ReeeeeWeCee3e/eQeehNe(eeeee採Deee8eee>e<eeTeSee8*0eD[7eeexe·Eie`e9ee$
ee3TCxdeMeOeeSeTme1ef4)eeeeeGeJCe        eee7XeVepe=eRH[e#deHeCe"xa▯eeΔ~eej!eGeeeeeeMntYCgeeeXee▯WT◆P"ee``JW0ep_eeG&eee}reeeVe!eIeeeV"
eeeee?ebee]3eb9.
        aKxe=eeeeee9eeeenE|e@eeNeey`= eee:e8ee21eD{We|>
the coding time is 0.000690 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/arithmetic_coding$ ./arcd -d <encodingtext | tee decodingtext
In today's world, information is one of the most valuable assets, but there is a major threat to it by the evolving sophisticated malware (malicious s
oftware). The first malware was created for fun, but now its a profit-driven industry [1] and in the last couple of years advanced malware uses comple
x obfuscation methods viz. control/data flow permutation, compression, heap spray, etc. [2] to evade the detection techniques. Also, there is an expon
ential increase in the number of malware released every year. To detect these malware, various techniques are proposed in the past. These methods rang
e from the early day signature-based detection to Machine Learning techniques [3]. However, it appears that proposed approaches are not sufficient to
detect the unknown sophisticated malware. Thus researchers are continuously searching for better techniques to detect the advanced malware and have st
arted applying machine learning techniques for malware detection. These algorithms can detect obfuscated malware and have potential to handle both com
plexity and scale problem like data pre-processing, feature extraction/selection, model creation and testing. In this Santos et. al claims to obtain a
n accuracy of ~96% using opcode frequency as features [4]. Later on, researchers also applied file-size based segmentation on opcode frequency and imp
roved the average accuracy using different Machine Learning models and claimed that Random Forest can provide the accuracy up to ~98% with a False Pos
itive Rate (FPR) of ~1.07% [5]. For the analysis they used the benchmark malware samples from the Malicia project [6].


the coding time is 0.001013 sec.
```

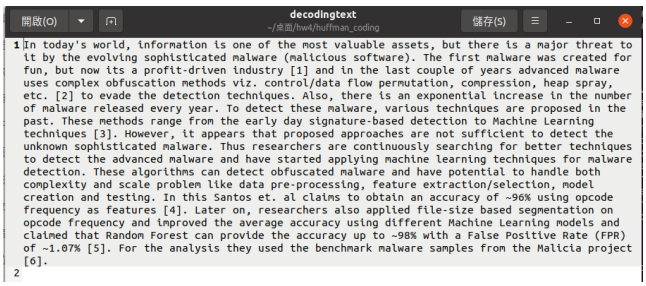
Huffman coding 測試結果如下圖。按照程式要求的 argument 來輸入及輸出檔案並決定編碼命令，檔案只記錄編碼內容，terminal 只印出程式運行時間。

```
jacob@jacob-VirtualBox:~$ cd /home/jacob/桌面/hw4
jacob@jacob-VirtualBox:~/桌面/hw4$ cd huffman_coding
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i testarticle -o encodingtext -c
the coding time is 0.000266 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i encodingtext -o decodingtext -d
the coding time is 0.000101 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i testarticle -o encodingtext -c
the coding time is 0.000173 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i encodingtext -o decodingtext -d
the coding time is 0.000081 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i testarticle -o encodingtext -c
the coding time is 0.000174 sec.
jacob@jacob-VirtualBox:~/桌面/hw4/huffman_coding$ ./huffman -i encodingtext -o decodingtext -d
the coding time is 0.000084 sec.
```

詳細指令會列在 readme，下頁附圖為兩種編碼 decoding 之後的文本內容。



arithmetic coding decoding text



huffman coding decoding text

3. 分析

在分析的部分，我會使用不同字元數的文件來做編碼。從一個字元(1_byte)到一串字串(testtext，46 個字元)，接著以一篇範文(testessay，約 960 個字元)到最後的一篇文章(testarticle，約 16000 個字元)來去觀察兩種編碼方式的運行時間。經過測算之後結果如右表所示。可以觀察到在 100 個字元內，兩者執行時間的差值均小於 1ms，但到了 1000 個字元以上的大量字串，很明顯的 huffman coding 的執行效率遠比 arithmetic coding 來的好，甚至在 article decoding 的部分，時間相差了 20 倍之多。另外，我發現 huffman coding 在 decoding 的效率略比 encoding 還好一些；相反的，arithmetic coding 在 decoding 的效率卻比 encoding 還差一些。

	huffman(sec)	arithmetic(sec)
1_byte encoding	0.000027	0.000013
1_byte decoding	0.000012	0.000013
text encoding	0.000041	0.000041
text decoding	0.000025	0.000047
essay encoding	0.000106	0.000445
essay decoding	0.000051	0.000635
article encoding	0.000810	0.007072
article decoding	0.000579	0.010665

4. 問題探討

(1) Arithmetic coding 如何輸入檔案

我使用的是在 eeclass 附上的開源碼，而這個問題困擾了我蠻長的一段時間，因為在 huffman coding 只要按照他 argument 的指令輸入他就會吃檔案。然而在處理 arcd 的時候並沒有那麼順利。後來慢慢閱讀程式碼，並跟同學互相討論過後，發現他需要透過 bash 指令來 input file，這才解決了執行 arcd 的問題。

(2) Arithmetic coding 運行時間的顯示

運行時間我是使用 time.h 的 clock 函數來做運算，且排除讀寫檔時間，這個部份不是甚麼大問題，但在 arcd 因為它 input 跟 output 皆須透過 bash 指令，而原本我是使用>來 output，結果導致沒有辦法從 terminal 端看到運行時間。因此我改用| tee 這個指令，可以從 file 裡面跟 terminal 上同時看到 coding 結果及運行時間。