演算法 Term Project - 算術編碼

一 介紹:

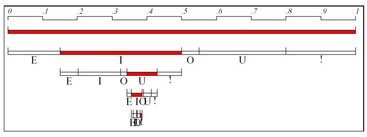
資料壓縮一直是一個很重要的研究，特別在這一個網路的時代，要讓更多的資料利用有限的頻寬來傳輸，他的價值更能顯現出來。

算數碼法（Arithmetic Coding）是Rissanen 於1979年所提出的一種壓縮方法，這方法最大的特點在於以一實數來表示壓縮字串，將一段message利用一個0到1間的區間來表示，當這個message越長，用來表示這個message的區間就越小，那要表示這個message的bit數就變多。 相同的文字在message出現越多，區間變小的速度比較慢所以可以達到資料壓縮的效果。隨著符號序列長度之增加，其算數編碼長度之entropy趨近於無雜訊編碼理論之極限(效率愈高)。

除了算術編碼外，常見的演算法還有霍夫曼法(Huffman)，原理是將欲壓縮之字串，先讀一遍，將字串中的每一相異單字元（Single Character）的出現頻率，做成統計，依此建構霍夫曼樹（Huffman's Tree）。每一相異單字元，用0與1予以編碼，出現次數過多者，給予較少的位元編碼。霍夫曼編碼法的特點在於所編碼出來的檔案具有唯一碼性質的即時碼。也就是各個相異字元所編碼出所位元串並不相同，解碼時能立即解出。

二 原理解說:

假設一個字串是由集合 S = { E, I, O, U, ! } 所組成，其各字母出現的機率依序如下 { .15, .35, .05, .25, .20 }，其機率分佈為 { .0, .15, .5, .55, .8 }。將字母出現的機率分佈標於(0, 1)的區間內，如圖所示。



以上圖為例，實際來編一個字串 IUI 作為範例。首先編第一個字元 I，那我們只要將區間(.15, .5)中的任一實數便能代表 I 這個字元，接下來編第二個字元 U，U 發生的機率為 .25，發生 I 後發生 U 的機率為 .35*.25，標示於圖上區間以機率分佈來表示，則為 [.15+.55*.35, .15+.8*.35) = [.3425, .43)，再來編第三個字元 I，代表字串 IUI 的區間則為

[.3425 + .15*(.43-.3425), .3425 + .5*(.43-.3425)) = [.3425+ .15*.0875, .3425+ .5*.0875) = **[.355625, .38625)**

依此類推，在整個字串都經過演算編碼後，最後再編入我們所定的一個 End_of_sequence marker，則編碼完成。

三 演算法:

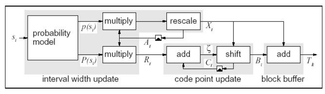
P 代表 P.D.F.
p 代表 p.d.f.
ch 代表 字串內的單一字元

- Input:　一個長度為L(bytes)的字串，其內各字元出現的機率為p(ch)，機率分佈函數為P(ch)
- Ouput:　一個用長度為L'(bytes)的資料量，來表示的實數,其中 L >= L'

step 1.　建立字元出現頻率表
Step 2.　依序取入一個字元 ch 進行編碼
Step 3.　首先計算目前區間長度(range = high - low)，目前區間是以 high 為 upper bound 與 low 為 lower bound 包圍。
Step 4.　依字元出現的頻率以及累計次數來計算一個在目前區間的範圍內的一個新區間，即：

high <- low + range * P(ch+1)
low <- high + range * P(ch)

Step 5.　以二進制計算一個實數，其值於此新區間內，來代表編至目前的字串。
Step 6.　若為 Adaptive 編碼，針對 ch 將其出現次數加一。
Step 7.　重覆 Step2. 直至所有的字元皆編碼完畢。
Step 8.　最後編入一個自定的 End_of_sequence 編碼完成。



Expanded View of Coder Functional Units.

四 程式設計:

為了減少浮點數的運用，所以我們將要表示的區間自 [0, 1) 的區間內比例對應至 [0, 65535) 的區間內。如此亦可改善當區間越來越小，浮點數能表示的精位可能不足的問題。將〔0, 65535) 標上四個部份 [0, 1/4), [1/4, 1/2), [1/2, 3/4), [3/4, 1)， 而字串所編成的 code 區間為 {low, high) 包含在 [0, 65535) 間，在輸出更精確的編碼位元後，重新將最新的 [low, high) 依比例擴展至 [0, 65535) 內。 運算sudo code如下:

```
Half = 1/2 * 65536;
while(1)
{
        if high  <  Half then
                output a bit 0;
                low = 2*low;
                high = 2*high+1;
        else if low  > =  Half
                output a bit 1;
                low = 2*(low-Half);
                high = 2*(high-Half)+1;
        else break
}
```

如此 [low, high) 區間永遠會座落在 [0, 65535) 區間內，可避免 Underflow 的情況。

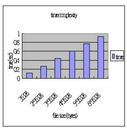
五 實驗結果:

測試環境:

- 作業系統: Windows ME
- 電腦設備: COMPAQ notebook, CPU PIII 800, RAM 128MB
- 編譯器: Microsoft Visual C++ 6.0
- 測試數據:

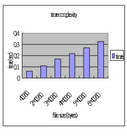
壓縮 pattern : Text File

file size (bytes)	35328	2 * 35328	3 * 35328	4 * 35328	5 * 35328	6 * 35328
time / output size	0.11 / 22919	0.27 / 47308	0.44 / 71709	0.61 / 96073	0.77 / 120686	0.93 / 144845
壓縮比	0.649	0.670	0.677	0.680	0.683	0.684



壓縮 pattern: Binary File

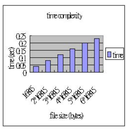
file size (bytes)	40960	2*40960	3*40960	4*40960	5*40960	6*40960
time / output size	0.06 / 30060	0.11 / 61428	0.17 / 92930	0.22 / 124284	0.27 / 155779	0.33 / 187144
壓縮比	0.734	0.75	0.756	0.759	0.761	0.761



- 作業系統: Linux CLE 0.9
- 電腦設備: PIII 600, RAM 128MB
- 編譯器: egcs-2.91.66
- 測試數據:

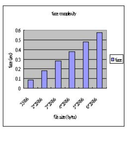
測試pattern: Binary File

file size (16845 bytes/per unit)	1	2	3	4	5	6
time / output size	0.04 / 11713	0.08 / 23229	0.12 / 34776	0.16 / 46331	0.20 / 57882	0.23 / 69429
壓縮比	0.695	0.689	0.688	0.688	0.688	0.687



壓縮 pattern: Text File

file size (bytes)	35966	2*35966	3*35966	4*35966	5*35966	6*35966
time / output size	0.09 / 21602	0.18 / 43051	0.28 / 64504	0.38 / 85954	0.48 / 107406	0.57 / 128857
壓縮比	0.601	0.598	0.598	0.597	0.597	0.597



- 時間複雜度分析:
由於這次實作的算術編碼，並沒有考慮前後文的關係，除了Adaptive的調整每個字元出現的機率，因此每一個字元的編碼都是做一樣多的運算，因此時間複雜度為 O(n)，在實驗結果的圖表亦可看出這樣的趨勢。

六 結論與心得:

這是一個簡單的arithmetic coding的實作，在理論上算術編碼可以很逼近entropy，由於沒有設計出一個良好的機率model以及缺乏一個好的預測strategy，所以此次結果並不亮眼。

這次在兩個平台下測試 Windows 和 Linux，Linux下壓縮在測試期間的數據(壓縮時間)，都比windows下穩定很多，推測是Linux下的排程較穩定公平的原故。由於Adaptive的機制，讓 Text File 壓縮的效果較 Binary File 來的好，因為 Text File常用字元範圍較Binary File來的小。在測試pattern方面，為了造出特定比例大小的檔案，所以利用同一個檔案累加出不同比例的pattern，目地是達到了，不過也使得pattern測試的效果產生了些爭議，這也是為什麼壓縮比很相近的原因。

為了能夠比較此次算術編碼和我們一般使用壓縮軟體的效能為何，我以zip來壓縮部份的測試pattern。在Linux下16845 bytes的binary file以zip的輸出為6872 bytes。而35966 bytes的test file以zip的輸出為 701 bytes，結果可謂是一面倒。除了zip以字典編碼的方式，再配上此次測試pattern產生的方式，搭配起來產生了高的效能，當然了zip長久來的發展，也是功不可沒的，讓人更深深體會技術的需要累積與精益求精。以結果蓋棺論定，可說是一面倒，這次實作的效能還是有很大的發展空間。

許多的影像壓縮技術，都開始採用Arithmetic Coding來做為後端的entropy coding的工具，搭配上有效的機率預測的Model，可以對於輸出的效能更為提升。這次基本的實作是第一步，只要依此根據不同的功能需求，做適當的調整，相信算數編碼會勝任在他壓縮長才的。

七 參考文獻:

- Harry Printz, "Tutorial on Arithmetic Coding," Draft - not for distribution. 1994.2.23
- 戴顯權, "資料壓縮", 1996.3.
- http://163.25.10.166/lab/project/new%E7%AE%97%E6%95%B8%E7%B7%A8%E7%A2%BC%E6%B3%95.htm
- http://noya.ame.ntu.edu.tw/~rtlin/Course01/lecture_notes/c1lecture_note10.htm

八 附錄:

程式碼 arth.zip download

資工所 91級 8934609 鄭耀傑