Arithmetic coding for data compression

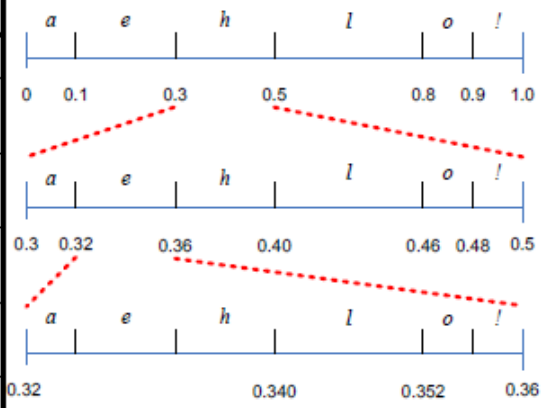
一、 論文之主要目的

一般的算數編碼通常將資料出現的頻率轉化為其對應的機率，並建立一符號 (包含 Stop word => !) 與其對應的機率表格，故機率總合為 [0,1]，如下圖一所示。並透過不斷的壓縮符號對應的機率區間，以達成 Encode 的目的，請參考圖二；相反的 Decode 時只需要不斷的計算當前機率區間其對應的符號為何即可，請參考圖三。

```
codeTableData.push_back(make_pair("a", 0.1));
codeTableData.push_back(make_pair("e", 0.2));
codeTableData.push_back(make_pair("h", 0.2));
codeTableData.push_back(make_pair("l", 0.3));
codeTableData.push_back(make_pair("o", 0.1));
codeTableData.push_back(make_pair("!", 0.1));
```

圖一、符號與機率對應表

Input	low	high	range = high - low
h	0.3	0.5	0.2
e	0.3 + 0.2×0.1 = 0.32	0.3 + 0.2×0.3 = 0.36	0.04
l	0.32 + 0.04×0.5 = 0.34	0.32 + 0.04×0.8 = 0.352	0.012
l	0.34 + 0.012×0.5 = 0.346	0.34 + 0.012×0.8 = 0.3496	0.0036
o	0.346 + 0.0036×0.8 = 0.34888	0.346 + 0.0036×0.9 = 0.34924	0.00036
!	0.34888 + 0.00036×0.9 = 0.349204	0.34888 + 0.00036×1.0 = 0.34924	0.000036



圖二、Encode 之範例流程說明

Example (cont., x = 0.34921)

Serial no.	x	$[p_c, q_c)$	output	interval
1	0.34921	[0.3, 0.5)	*h*	0.2
2	(0.34921-0.3) / 0.2 = 0.24605	[0.1, 0.3)	*e*	0.2
3	(0.24605-0.1) / 0.2 = 0.73025	[0.5, 0.8)	*l*	0.3
4	(0.73025-0.5) / 0.3 = 0.7675	[0.5, 0.8)	*l*	0.3
5	(0.7675-0.5) / 0.3 = 0.8917	[0.8, 0.9)	*o*	0.1
6	(0.8917-0.8) / 0.1 = 0.917	[0.9, 1.0]	*!*	0.1

圖三、Decode 之範例流程說明

由圖二範例可見，壓縮 hello!後的結果落在 0.349204 和 0.34924 之間，差距為 0.000036。此演算法最大的問題即在此處，若區間過小即會造成小數點的 underflow 問題，意即分不出誰是 low 誰是 high，於接下來的壓縮或是解壓縮都有可能造成數值上的錯誤。

本論文最大的貢獻在於，提出一演算法能夠不會有 underflow/overflow 的錯誤，還能夠根據出現過的符號次數，自動的調整演算法，使得符號的壓縮率提升。

二、 論文原理介紹

Encode 原理概述：

由於原始的 Arithmetic Coding 是字頻映射至[0, 1]的空間，透過不斷切細將新字母壓進區間內，因而造成 underflow 等問題。此論文反其道而行，將字頻的區間不斷的放大映射至[0, 65535]的空間，判斷最高有效位元後即輸出，讓剩下的位元與新字元能夠有足夠的空間壓縮至[0, 65535]內。至於如何決定最高有效位元的輸出機制、為什麼不斷放大不會有 underflow 問題等細節將於論文演算法步驟介紹一併說明。

Decode 原理概述：

Encode 完畢後，會獲得一串 2 進制的位元，並不是 fix 固定長度，在論文中他可能為了方便 socket 之類的網路傳輸或是顯示等，將資料 Encode 成一個一個 char 輸出(putc)，但核心精神還是以一串 2 進制的位元為基準。

Decode 的方式其實就是將 2 進制的位元字串轉成 10 進制，反過頭回去查詢此數字落在字頻的累積表上哪一個 index，而此 index 即為原先壓縮的資料。取得資料後，參考 Encode 調整數值區間的方式進行調整，並讀入一新的 bit，重新再迴一次 loop，直到數值區間穩定，若在穩定前以無 bit 可讀，將新 bit 視為 0 即可。(因為一開始 Encode 完的 2 進制位元字串長度會不夠 Decode 最後一個字元，故需補零)

Adaptive source mode 原理概述：

由於原始的 Arithmetic Coding 是根據字頻下去壓縮的，故在 Encode/Decode 階段都需要告知其字元與其頻率的對照表為何，因此在 Encode/Decode 的前置階段需要先行傳送頻率對照表。此方法雖然簡單且容易實作，但需要先行分析頻率和耗費頻寬。但現實的情況很多時候並不知道所有字元的頻率狀態，故壓縮出來的位元可能不會是最佳的。

此論文一開始將所有的字頻全部設定為 1，之後根據出現的次數做增加的動作，若是一字元的出現次數偏高，那麼在字頻的累積表上其數值區間就會變大，故 Encode/Decode 只要參考這變動的字頻累積表即可。

三、 論文演算法步驟介紹

Ascii Code 為 0-255，共 256 個符號，因 Arithmetic Coding 需要將 STOP word 一併 Encode，故一共要 Encode 257 個符號。本論文為了實作方便，將符號的 Ascii Code+1 當作該符號於陣列的 Index，故 Array 大小開 258 格，因此 Frequency[‘A’+1] = 1 => Frequency[66] = 1。字頻累積表(Cumulative Frequency)則是由最後一個字元開始累加，若從 index = 0 開始看則為由大到小排序，index = 257 => CumulativeFrequency[257] = 0。無論 Encode/Decode 皆需初始化此表。

Symbol	Index	Frequency	Cumulative Frequency
NULL	0	1	257
.	.	.	.
A	66	1	191
B	67	1	190
C	68	1	189
D	69	1	188
.	.	.	.
STOP word	257	1	0

以欲壓縮 ABC 為例：

- Step 1：設定 low = 0, high = 65535, bitsToFollow = 0，將 Frequency, CumFre 表格初始化。
- Step 2：並提取一新字元 symbol 開始 Encode
  - Step 2.1：range = high - low + 1
  - Step 2.2：high = low + range * CumFre[symbol - 1] / CumFre[0] =1
  - Step 2.3：low = low + range * CumFre[symbol] / CumFre[0]
  - Step 2.4：WriteBits(0) :- high < HALF
    WriteBits(1), low-=HALF, high-=HALF :- low >= HALF
    bitsToFollow++, low-=FIRST_QTR, high-=FIRST_QTR
    :- low >= FIRST_QTR and high < THIRD_QTR
    Jump to Step 3 :- not satisfy above conditions
  - Step 2.5：low = low * 2
  - Step 2.6：high = high * 2 + 1
  - Step 2.7：Jump back to Step 2.4
- Step 3：若是 CumFre[0] >= 16383, Frequency[i] = (Frequency[i] + 1) / 2, i for all element in Frequency, and Recalculate CumFre
- Step 4：Frequency[symbol]++, CumFre[i]++, i for 0 to symbol-1
- Step 5：Jump to Step2, until not symbol to retrieve

WriteBits(bit)：

- Step 1：output bit
- Step 2：while(bitsToFollow>0) output !bit, bitToFollow--

以 Decode 00000010001001111000000110111111101111101 為例：

- Step 1：設定 low = 0, high = 65535，將 Frequency, CumFre 表格初始化。
- Step 2：由右往左讀取 16 位元，value = 1011111101111110 = 49022
- Step 3：開始解碼
  - Step 3.1：range = high - low + 1
  - Step 3.2：cum = ((value - low + 1) * CumFre[0] - 1) / range
  - Step 3.3：find symbol, symbol from 1 to CumFre[symbol]>cum
  - Step 3.4：high = low + (range * CumFre[symbol -1]) / CumFre[0] - 1
  - Step 3.5：low = low + (range * CumFre[symbol]) / CumFre[0]
  - Step 3.6：do nothing :- high < HALF
    Value -= HALF, low-=HALF, high-=HALF :- low >= HALF
    Value-=FIRST_QTR, low-=FIRST_QTR, high-=FIRST_QTR
    :- low >= FIRST_QTR and high < THIRD_QTR
    Jump to Step 3 :- not satisfy above conditions
  - Step 3.7：low = low * 2
  - Step 3.8：high = high * 2 + 1
  - Step 3.9：value <<=1, value + 從 EncodeedString 讀取一新 bit, 若無新 bit，則讀 0
  - Step 3.10：Jump back to Step 3.1

- Step 4：symbol :- STOP word, Done!
- Step 5：origin text = symbol – 1
- Step 6：若是 CumFre[0] >= 16383, Frequency[i] = (Frequency[i] + 1) / 2, i for all element in Frequency, and Recalculate CumFre
- Step 7：Frequency[symbol]++, CumFre[i]++, i for 0 to symbol-1, Jump back to Step 3

Adaptive source mode/underflow problem 總結：

之所以此論文的演算法能夠避免 underflow 的原因在於在 Scale 區間的時候，是 Scale Up 而不是 Scale Down。而且最高有效位元每次都會被 output 故視為以被記錄了。再加上累計符號頻率表不超過 16383，若超過則會被 Scale Down，因此只要保證 f <= c - 2, f + c <= p, f 為最高頻率所用的位元，c 則是數字區間所用的位元，p 則是 programing 所用的容器之位元。以目前演算法來說，c = 16, f = 14, 故 p 要選擇 30 位元以上的資料型態，以防止 overflow。
Adaptive source mode 的話，單就實作面其實非常的簡單，就只是每次 Encode/Decode 完，更新當前字頻表，使得壓縮的區間逐漸變大，而使得字母的 Entropy 變小，壓縮率變高。

四、 論文之演算法實作說明

此論文實作之 Source Code 透過以下連結即可取得，此專案為 Visual Studio2013 所開發，若使用 Visual Studio 2015 則需安裝 Platform Toolset v120 方可編譯。
https://github.com/WindAzure/VideoSinalProcessing/tree/master/reading%20assignment