**Arithmetic coding for data compression**

1. **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。

|  |  |
| --- | --- |
| 圖一、符號與機率對應表 | 圖三、Decode之範例流程說明 |
| 圖二、Encode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。**

1. **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。

1. **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  (Ascii Code+1) | 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 0000001000100111100000011011111101111101 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。

1. **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>