B06507002 材料二 林柏勳

壓縮檔包含

a) 一份報告: b06507002_report.pdf
b) 兩份程式檔:

- b06507002.py，包含 Optimal, MTF (move to front), Transpose, BIT, FC (frequency count)的實作
- TimeStamp.py，包含對 TS 的實作

c) 一份輸出檔:output.txt (Bit 的輸出跟報告中使用的輸出不同)

這次的作業包含:五個規定的演算法以及 bonus 的演算法 TimeStamp。

本人使用 python 實作上述的演算法，並且可以用以下方式測試該程式:

command line at windows or linux

python b06507002.py input.txt output.txt

python TimeStamp.py input.txt output_2.txt

以下敘述實作六個演算法的方法:

1. Optimal:

   因為是最佳解的形式，所以直接統計不同輸入長度各個數字出現的次數，然後再將數字依頻率由大排到小，這樣的擺放方式為最佳解。

   令統計 0~9 數字 2*10 array 為 A，A 第一個 row 放的是數字，第二個 row 放的是出現的頻率，用第二個 row 的大小進行排序，最後計算 Cost 如下:

   $\sum_{i=0}^{9}(i+1)\times A[1][i] = 1 * \mathrm{A}[1][0] + 2 * \mathrm{A}[1][1] + \cdots \ldots 10 * \mathrm{A}[1][9]$。

2. MTF:

   Move to front 則是先使用一個 list 儲存 1~9 的數字，根據 input array (r) 的編號依序讀起各個數字，從 list 的頭開始讀數字，讀取之後，紀錄該數字當下在 list 中的位置，計算 cost=array 中的編號+1。然後將該數字移至 list 的第一個數字(編號 0)。

   在 python 中可以寫成: lis=[lis[j]]+lis[:j]+lis[j+1:]

   j 為被搜尋的數字在 array 中的位置。

   Lis[:j]為選取 list 中從編號 0 到編號 j-1 所有數字構成的 array。

   Lis[j+1:]為選取 list 中編號 j+1 到最後一個數字所構成的 array。

3. Transpose:

   Transpose 也是先用一個 list 儲存 0~9 的數字，根據 input array (r)的編號，依序讀起各個數字，從 list 的頭開始讀起，如果讀到該數字的話，則將該數字往前一格，也就是將該數字與前面一個數字交換，紀錄 cost = 該數字在 array 中的編號+1。

   這邊用 python 的寫法為:

   ```
   If j!=0:
       lis[j]=lis[j-1]
       lis[j-1]=check
   ```

先用 Check 存取為要被搜尋的數字，而我們搜尋出 list 中的第 j 個位置為我們要找的數字。若用 j-1 位置的數字取代 j 位置的數字，然後再用 check 取代第 j-1 位置的數字，即可以達成 Transpose 的效果，值得注意的是，當被搜尋的數字位於 list 的頭時，並不會移動該數字。

4. BIT

   BIT，我用 2*n 的 list 儲存 bit 跟數字，第一個 column 儲存數字，第二個 column 儲存該數字的 bit。一樣依 input array (r)的編號，依序讀起各個數字，從 list 的頭開始讀起，如果找到該數字的話，則會將該數字的 bit 會加一然後 mod 2

   ，如果這樣的動作做完，被換 bit 的數字變成一的話，則將該數字移動到 list 最前面的位置。

   在 python 中移動的寫法如下:

   lis=[[check,1]]+lis[:j]+lis[j+1:]

   check 為當下被尋找的數字，它在 list 的第 j 個位置，使用[check,1]的原因為整體是一個 n*2 的 array，所以加入時要順便把第二個 column 的元素加入。

5. FC

   FC 一樣存一個 n*2 的 array，但是它的第二個 column 存的是:到目前為止被搜尋的次數。每次搜尋完一個數字後，都會依照第二個 column 的數字由大排到小，將整個 list 重新排列。

6. TimeStamp (bonus)

   這個演算法的敘述如下:

   給定一個機率 p:

   有 p 的機率會執行 move to front

   有 1-p 的機率會執行以下步驟:

   a) 如果該數字 x 從來都沒有被搜尋過，則不作任何更動
   b) 如果該數字 x 曾經被搜尋過，則朝 x 現在所處的位置往前搜尋，找到離它最近的一個數字 v 符合以下條件:
      被搜尋的數字從上次被搜尋到現在，v 被搜尋的次數只能是一次或零次
      找到後將 x 插入 v 的前方，如果搜尋到頭都沒能找到符合條件的 v，則不做任何動作。

隨機的部分:

使用 random.uniform(0,1)隨機從 0 到 1 取樣一個值，如果該值小於等於 p 則執行 MTF，如果大於 P 則執行後者。

有一定機率執行 MTF 的原因在於 MTF 已被證明最多查找次數不超過最佳解的兩倍，所以可以適度地用來提高速度。

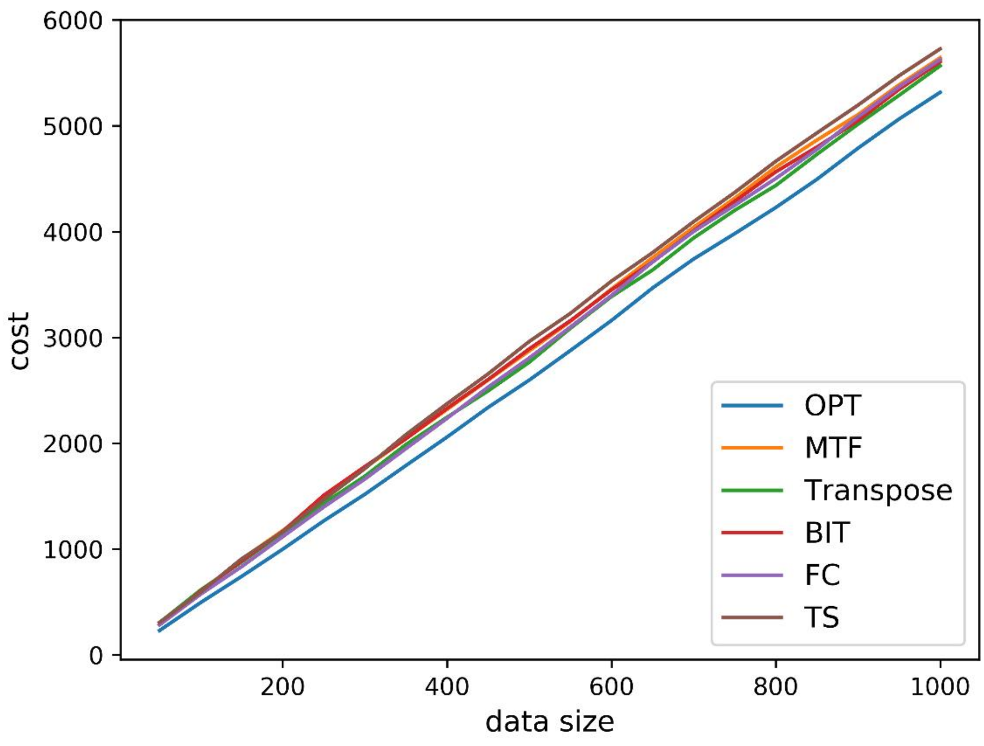
有一定機率執行另外步驟，則是想藉由其他方法去減少整體查找的次數。

之所以限制該數字至少要找過一次才能移動的原因在於:

如果該數字只出現過一次，就將他移動可能會太武斷了些。
而該數字如果曾經被找過，則使用 6-b)這樣的移動條件的原因在於:
1.要確保較少找過的數字能夠被慢慢的洗掉
2.不武斷、大幅度移動數字造成整體的巨大的改變
所以我用一個 10*11 的 list 進行實作這樣的一個演算法。
list 的最後一個 column 存的是該數字是否曾經被找過，
list 的前面 10*10 的部分在於紀錄某數字出現過後，其他數字被查找的次數，所以每次有一個數字被尋找過後，該數字的 column 的每個元素會+1(+1 僅限於該 row 的數字曾經被找尋過)，該數字的 row 的每個元素會被清成 0。找 v 用的判斷也是使用該數字的 row 的值進行判斷。

根據參考資料三，當$p=\frac{3-\sqrt{5}}{2}$ ，算出來的 cost 會最小。

結果

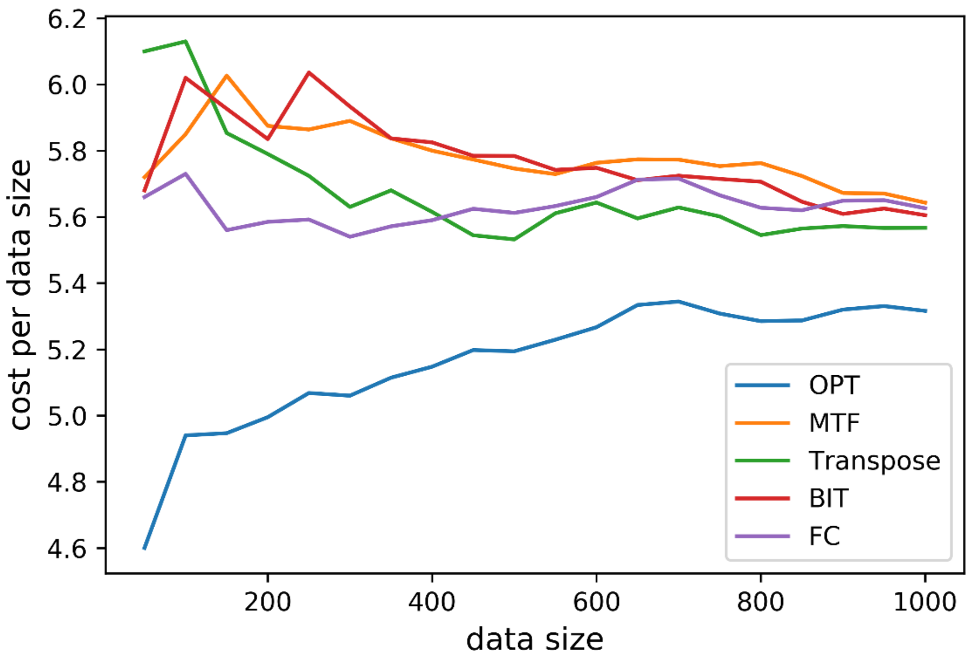


圖一、比較六種 on-line algorithm 的 cost 大小

註: BIT 部分使用的 initial bit 為 ”1000111100”，TS 使用的 p 為 $\frac{3-\sqrt{5}}{2}$，使用範例測資。

由圖一可以看出，OPT 確實為最佳解，扣除 OPT 的話，整體而言 Transpose 的表現不錯。
而如果把 cost 除以資料大小對資料大小作圖，結果如圖二，可以發現所有的方

法在資料量很大的情況之下，都會收斂，不過前提應為資料是隨機的。

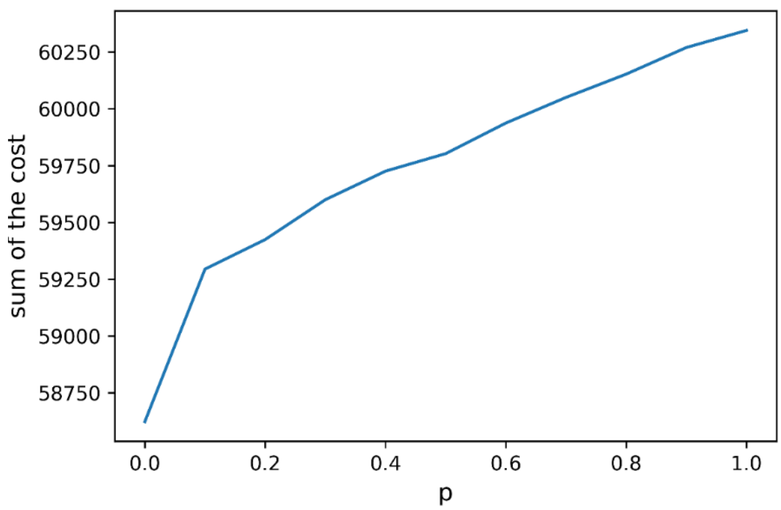


圖二、單位資料大小的 cost 對資料大小作圖，資料為範例測資。

Bonus

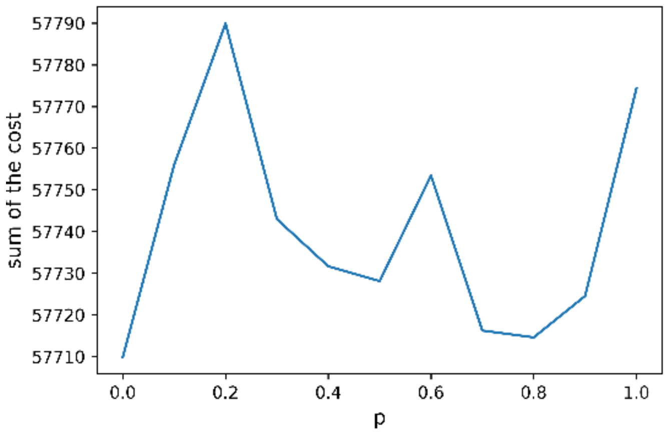
研究在 Time Stamp 演算法 p 值大小是否影響整體而言的 cost，我使用了以下的模擬，模擬的測資為範例測資，依序調整 p=0、0.1、0.2 ......1.0，每個 p 值做一千次取平均，橫軸座標為 p，縱軸座標為 i=50,100,150......1000 的時候得到的 cost 值的加總，得到結果如下:

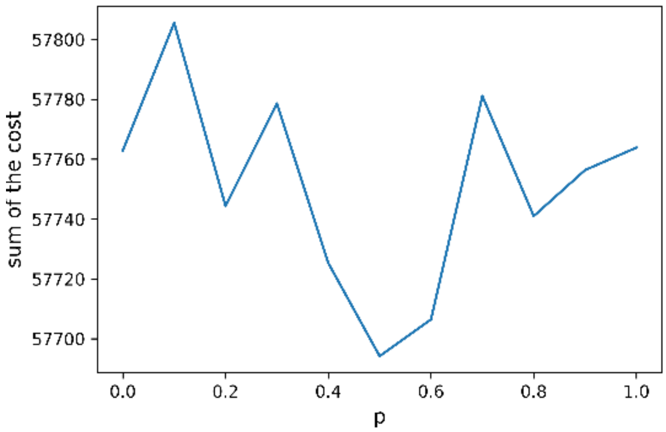
發現在 p=0 的情況之下，得到的整體 cost 會最小，所以對於本次的測資，TS 要採用 6-b)會使一個比較好的測略。



圖三、sum of total cost 對 p 作圖

但是單看範例測資並不能作為一般的情況，所以我又使用長度為 1000 的隨機 input 數列進行 1000 次模擬，並取做平均，結果如下:





圖四、圖五、 total cost 對 p 作圖

發現在圖四、五中，p 在接近 0.4 的部分有局部極小值，這樣的結果可以驗證理論值最佳 $p=\frac{3-\sqrt{5}}{2}\simeq 0.38$。

參考資料:

1. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2016/MSC/MSC-2016-08.pdf
2. http://www.cs.cmu.edu/afs/cs/academic/class/15451-s14/LectureNotes/online.pdf
3. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.3259&rep=rep1&type=pdf