Wordfilter 修改與新增

本次 Wordfilter 按照要求使用 dictionary tree 之結構修改其程式碼，使其時間複雜度不因 filter.txt 的大小影響，進而大幅縮小其時間複雜度。

- Wordfilter 修改

在原本程式碼中我所使用之 STL 為 vector，在存放在使很好使用的陣列，但其缺點是在要抓取其值時，無法指定查詢與抓取，只能按序尋找；而這次透過提示給的 dictionary tree 我發現 STL 中有個 map 的指令，其內建為一個 Red–black tree 並且能夠進行指定查詢，透過一番測試後得到其較好的結果。

```
for (uPos = 0; uPos < wsInput.length();uPos++){
    for (i = 0; i<16 ;i++){
        ws.assign(wsInput,uPos,i);
        iter = mapfoo.find(ws);
        if ((*iter).second == L"*"){
            wsInput.replace(uPos,ws.size(),(*iter).second);
        }
        m++;
    }
}
```

```
//逐字搜尋完整之輸入句子
//從1位元比對到最高15位元比對
//要與dictionary tree中key比對的字
//找出比對字與key相符的位置
//如果相符則取value之值驗證
//將相符字視為欲過濾字進行取代

//過濾執行時的時間複雜度
```

圖 1 關鍵字尋找與句子取代之程式碼

在程式碼上的修改，將 vector 改成 map 做存取，由於其性質同樣為 STL，因此在修改上並無困難的完成，主要問題點在於搜索字的方式，在原先的 Wordfilter 中是透過取 filter.txt 中每個關鍵字去跟輸入句子逐字比較，而現在是需要拿句子中文字去尋找關鍵字，在搜尋資料的時候大部分的教學普遍都是使用 strtok 去進行切割，但問題點在於 strtok 是需要條件才能切割，例如空白、換行…等，在英文上可能會沒問題，但在中文中沒有人會將主詞、動詞、名詞用空格拆開，應故回想起 uPos 的位置使用。

在原本 Wordfilter 中使用 uPos 來使其關鍵字能夠在輸入句子裡逐字尋找，本次使用方法也是，使用逐字方法與關鍵字進行比較，由於關鍵字長度不一，故而再加入位元增加之迴圈，也就是說會按照順序由 1 位元長度進行比較，比較至最大關鍵字長度，由圖 1 中所以其關鍵字長度為 15 位元，也就是說英文為 15 字，中文為 7 字(中文 1 字為 2 位元)之長度，由此來解決句子比較關鍵字之問題。

- Wordfilter 執行與結果

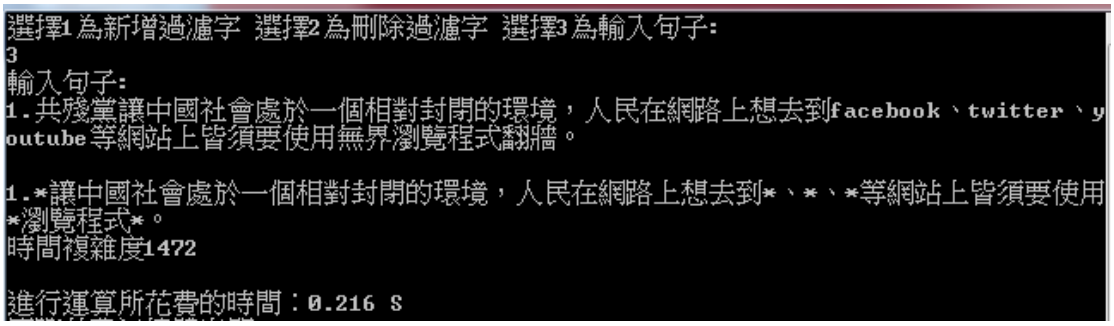

圖 2 執行 1 次之結果

由圖 2 中可以看到其輸入與輸出與原本 Wordfilter 是一樣的，但是其時間複雜度大大的縮小，但是為了能夠準確地將關鍵字一一比較出來，使得時間複雜度 O(N) N = Size of input string 因逐字累積的關係而成為 O(N*15) ，並且也因使用雙重迴圈讓運算時間比原本 Wordfilter 還高。

圖 3 連續執行之結果

同樣採用原本的 10 個句子來觀察其效能以及與原本 Wordfilter 進行比較，並以表 1 來顯示

表 1 ver3 與 ver4 效能差異比較

	Ver3 時間複雜度	Ver3 運算花費時間	Ver4 時間複雜度	Ver4 運算花費時間
第一次	141391	0.073	1472	0.216
第二次	141387	0.071	1904	0.216
第三次	141389	0.038	1232	0.216
第四次	141392	0.052	1360	0.227
第五次	141402	0.028	1392	0.226
第六次	141389	0.061	1024	0.209
第七次	141389	0.056	1680	0.243
第八次	141394	0.057	1824	0.222
第九次	141388	0.070	2176	0.212
第十次	141388	0.029	2144	0.230

由表 1 可以看到時間複雜度明顯下降，但是運算時間因雙重迴圈影響下比原本花費較長之時間。

- 新增項目

在前述中提到為了尋找關鍵字而進行 15 位元以內的比對尋找，以此相對的關鍵字的長度也有所限制，英文長度為 15 字，中文為 7 字之限制，如果超過其限制則不予以輸入，如圖 4 所示。

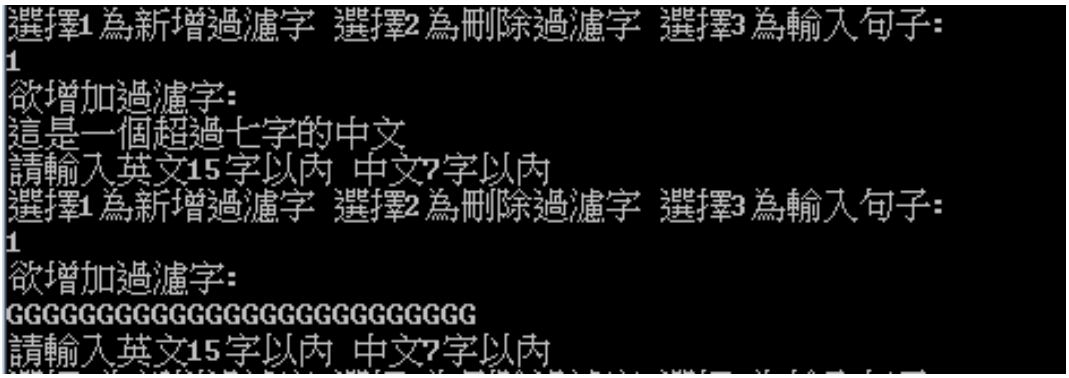

圖 4 新增字過長之結果

- 記憶體花費討論

記憶體使用花費的討論中原本使用的 vector 是能夠先行定義其容量大小，讓設計者能夠更有效率的使用記憶體空間，但 map 無法如此使用它會自行設置記憶體空間，使得記憶體空間花費非常的差，如圖 4 所示。

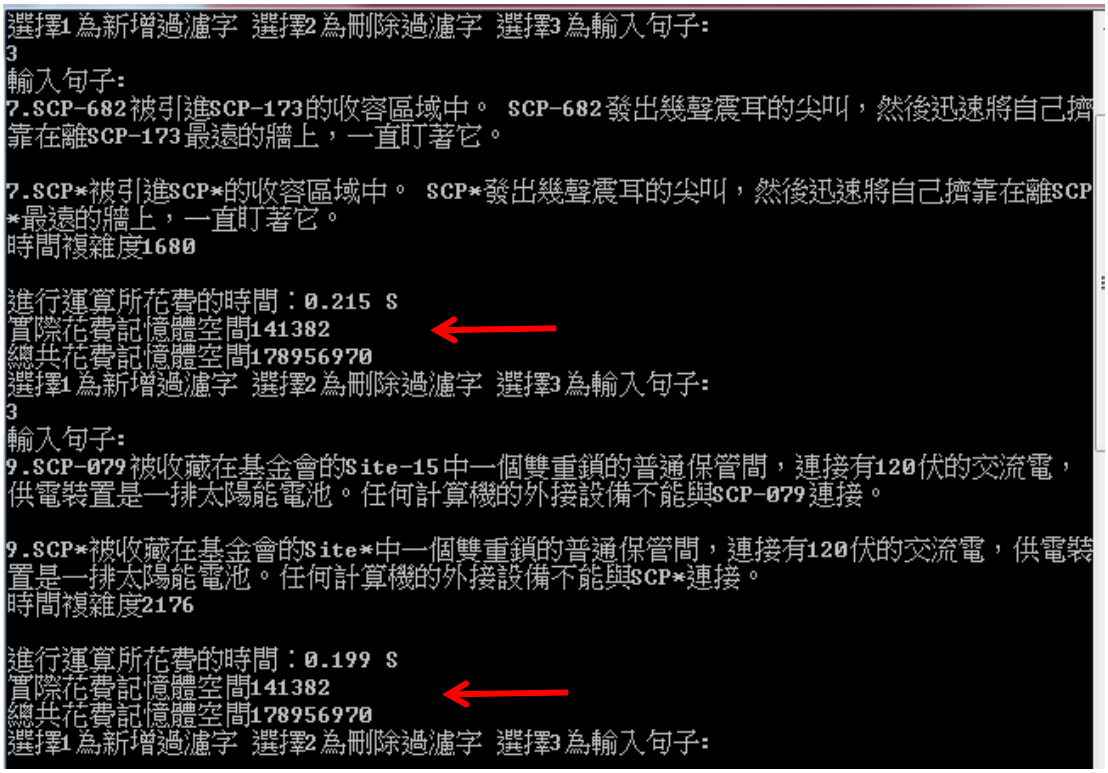

圖 5 記憶體花費示意圖

因此像 map 這種 dictionary tree 在使用上，雖然能夠有效降低時間複雜度，但是在記憶體的花費上是非常差的，所以必須針對記憶體的花費進行優化。

在尋找 dictionary tree 的相關文章時發現了一種 tree 能夠改善其問題，其為 Trie-tree，Trie-tree 與 dictionary tree 不同在於 Trie-tree 是將詞語結構化組織，使其形成具有公共前綴字，使其能夠利用前綴字來搜尋相關文字，進而減少記憶體花費，例如:向搜尋引擎輸入 C++之後顯示與 C++相關字，而非無相關字。

- 總結

透過 map 來將程式執行之時間複雜度與 filter.txt 關聯分開，使其下降，並使用雙重迴圈來逐字與關鍵字比較並過濾，再將其執行結果與之前版本進行比較並以表格顯示，再來以過濾字長度為基礎設計新增字的長度限制，最後針對使用的 map 進行記憶體分析與可利用 Trie-tree 優化之方向。