國立臺南大學資訊工程學系

資工三「演算法」課程

第四次作業

題目: Edit Distance

|  |  |  |
| --- | --- | --- |
| 班級 | ： | 資工三 |
| 姓名 | ： | 呂益銓 |
| 學號 | ： | S10659013 |

老師：陳宗禧

中華民國 108年12月24日

# 目錄

1. 簡介及問題描述……….……………..…………………………………………3
   1. 簡介…………….…………………………………………………………………………3
   2. 問題………………………….……………………………………………………………3
2. 理論分析….………………………..………………………………………4
3. 演算法則….…………………………..……………………………………5
4. 程式設計環境架構.………………………..…………………………………9
5. 程式.…………………………………………..………………………………10
6. 執行結果、討論與心得.………………………..……………………………16

參考文獻………………………………………………………….…………………19

1. **簡介及問題描述**

**1.簡介與問題**

[Goal]: 將一篇文章作一編輯後更正

Input: 一個字典(說明格式)及一篇文章

Output: i. BK-tree 字典 (Dic1, Dic2, Dic3)

ii. 根據 BK-tree 字典更正後的文章

iii. 分析

i. 根據自行收集的英文單字建立字典(至少 200 英文單字) 如 ： 動 物 名 稱 (https://tw.blog.voicetube.com/archives/44177) 、 國 家 城 市 名 稱 (https://tw.blog.voicetube.com/archives/39547)，可自行設定主題。撰寫 建構 BK-tree 作 search

ii. Edit distance：實作三種 Edit Distance a. 傳統 Edit Distance: Substitution 設為 1 b. Levenshtein Edit Distance: Substitution 設為 2 c. Damerau–Levenshtein Distance (<https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_> distance)

iii.比較三種編輯距離的差異

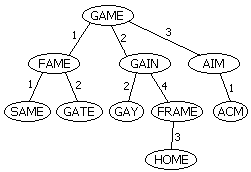
iv. 比較三種編輯距離所建構的 BK-tree (要說明 BK-tee 存在檔案的結構)以及 搜尋 字串的比較次數(錯誤容忍值) v. 輸出更正錯誤的字數(Word)與英文字 (Character)的數量，以及整篇文章字數 (Words)和字元數(Characters)的錯 誤率

**(二) 理論分析**

1. **BKtree**

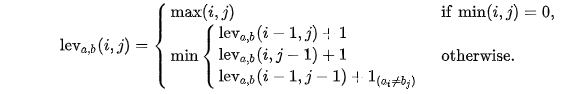
每一個node都有一個word(string)和多個child每一個child為一個struct存有distnace 和一個node的指針，利用這種方法建立出一個節點有多個child的tree，多個child利用紅黑樹來存以便搜尋時可以快一點。

建立完成後先設定root的word，之後再依序將剩下來的字insert進去，如果有一樣的distance就往child node往下搜尋直到找到external node，如果沒有一樣的distance就insert的child的distance是利用edit distance下去計算出來的，node的指針則新增一個nod裡面的child為空word為剛剛insert的字，



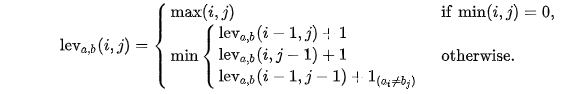
1. **傳統** **Edit Distance**

傳統的edit distance是利用insertions, deletions和substitution，每個insertions都增加1，每個deletions都增加1，每個substitution都增加1，再利用dynamic programming 的方法來實作，如果是用傳統 recursive的方法時間複雜度會到指數的等級，明顯不切實際利用填表格的方式來實作，時間複雜度則到O(mn)快很多。



1. **Levenshtein Distance**

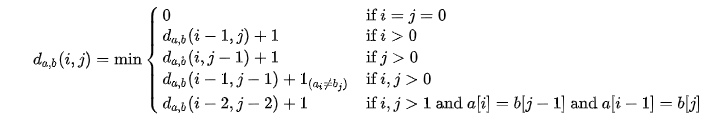
Levenshtein Distance和剛剛的distance基本上一樣只差在substitution的operation是+2而不是+1。



+2

1. **Damerau–Levenshtein distance**

Damerau–Levenshtein distance和傳統edit distance也是一樣，只差別在多出一個operation為transposition，當有兩個字母一樣時就可以交換，如:ac 和ca，這兩個的distance為1，因為只要ac和ca交換就一樣了。



**(三) 演算法則**

1. **Edit Distance**

**Algorithm**

input: strings a[1..length(a)], b[1..length(b)]

output: distance, integer

let d[0..length(a), 0..length(b)] be a 2-d array of integers, dimensions length(a)+1, length(b)+1

// note that d is zero-indexed, while a and b are one-indexed.

for i := 0 to length(a) inclusive do

d[i, 0] := i

for j := 0 to length(b) inclusive do

d[0, j] := j

for i := 1 to length(a) inclusive do

for j := 1 to length(b) inclusive do

if a[i] = b[j] then

cost := 0

else

cost := 1

d[i, j] := minimum(d[i-1, j] + 1, // deletion

d[i, j-1] + 1, // insertion

d[i-1, j-1] + cost) // substitution

return d[length(a), length(b)]

時間複雜度(time complexity)

O(mn)

空間複雜度(space complexity)

O(mn)

1. **Levenshtein Distance**

**Algorithm**

input: strings a[1..length(a)], b[1..length(b)]

output: distance, integer

let d[0..length(a), 0..length(b)] be a 2-d array of integers, dimensions length(a)+1, length(b)+1

// note that d is zero-indexed, while a and b are one-indexed.

for i := 0 to length(a) inclusive do

d[i, 0] := i

for j := 0 to length(b) inclusive do

d[0, j] := j

for i := 1 to length(a) inclusive do

for j := 1 to length(b) inclusive do

if a[i] = b[j] then

cost := 0

else

cost := 2

d[i, j] := minimum(d[i-1, j] + 1, // deletion

d[i, j-1] + 1, // insertion

d[i-1, j-1] + cost) // substitution

return d[length(a), length(b)]

時間複雜度(time complexity)

O(mn)

空間複雜度(space complexity)

O(mn)

1. **Damerau–Levenshtein distance**

input: strings a[1..length(a)], b[1..length(b)]

output: distance, integer

da := new array of |Σ| integers

for i := 1 to |Σ| inclusive do

da[i] := 0

let d[−1..length(a), −1..length(b)] be a 2-d array of integers, dimensions length(a)+2, length(b)+2

// note that d has indices starting at −1, while a, b and da are one-indexed.

maxdist := length(a) + length(b)

d[−1, −1] := maxdist

for i := 0 to length(a) inclusive do

d[i, −1] := maxdist

d[i, 0] := i

for j := 0 to length(b) inclusive do

d[−1, j] := maxdist

d[0, j] := j

for i := 1 to length(a) inclusive do

db := 0

for j := 1 to length(b) inclusive do

k := da[b[j]]

ℓ := db

if a[i] = b[j] then

cost := 0

db := j

else

cost := 1

d[i, j] := minimum(d[i−1, j−1] + cost, //substitution

d[i, j−1] + 1, //insertion

d[i−1, j ] + 1, //deletion

d[k−1, ℓ−1] + (i−k−1) + 1 + (j-ℓ−1)) //transposition

da[a[i]] := i

return d[length(a), length(b)]

時間複雜度(time complexity)

O(mn)

空間複雜度(space complexity)

O(mn)

**(四) 程式設計環境架構**

程式設計語言、工具、環境與電腦硬體等規格說明…

1. 程式語言

C in MS Windows

2. 程式開發工具

Visual Studio Code + MinGw64

3. 電腦硬體

CPU: Intel i5-9400f

Main Memory: 16GB

Disk:1T HDD

500G SSD

250G SSD

OS: W10 x64

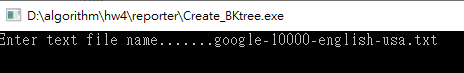
**(五) 程式 (含source code, input code, and output code)**

程式含source code, input code, and output code等…

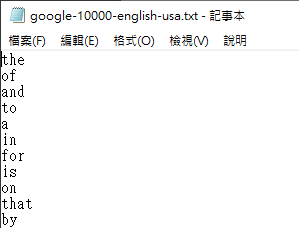
1. **主程式**

Create BKtree

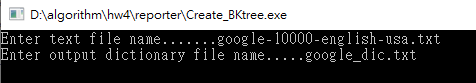
1.先輸入檔名



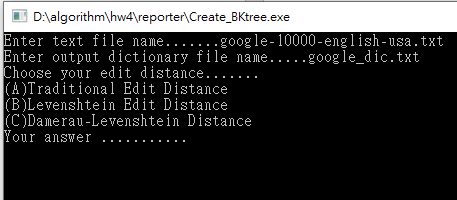
檔案格式



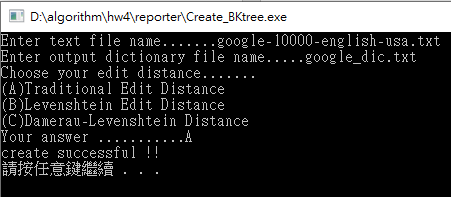
2.再輸入要儲存的字典名稱



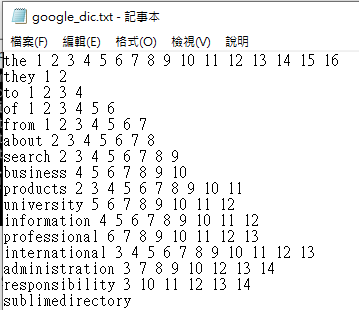
3.在選擇建立樹的時候要使用的distance種類



4.建立成功



輸出後的檔案格式

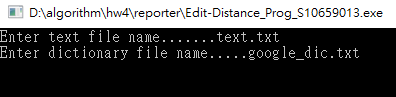


Edit-Distance\_Prog\_S10659013

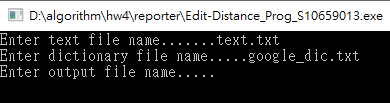
1.先輸入檔名



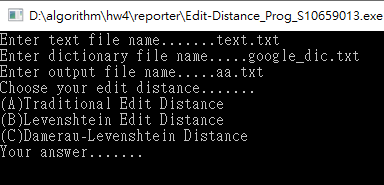
1. 再輸入字典的file name



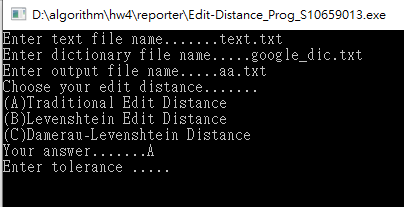
1. 再輸入要輸出更正後的filename



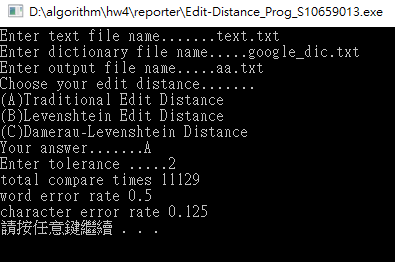
3.要使用的distance種類(要和剛剛建立的字典distance種類一樣)



4.再輸入容忍值如果找到容忍值以下的最小值(不為0)則會將單字修改成找到的字



5.輸出總共比較的次數與word error rate 與 character error rate



**(五) 執行結果、討論與心得**

1. 比較三種編輯距離的差異

同一個字串傳統的distance比Levenshtein Distance算出來的距離還要短因為 Levenshtein Distance的substitution為2而傳統的為1，而Damerau–Levenshtein distance可以會比剛剛傳統distance還要短或是一樣，因為Damerau–Levenshtein會 增加一種operation為transposition(交換)所以出來的距離才會小於或等於傳統的 distance

Example:ca 和 abc Levenshtein Distance為3 ca->a->ab->c

Damerau–Levenshtein為2 ca->ac->abc

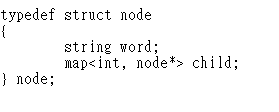
同一字串比較 Damerau–Levenshtein <= 傳統的distance <= Levenshtein Distance

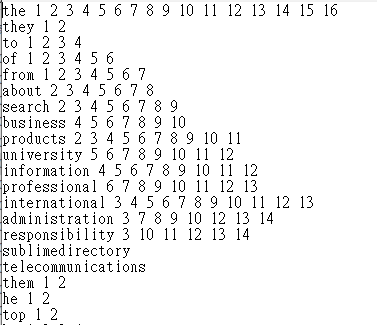
2. 比較三種編輯距離所建構的 BK-tree (要說明 BK-tee 存在檔案的結構)以及搜尋 字串的比較次數(錯誤容忍值)

BK-tee 存在檔案的結構:

由node所組成的multiway search tree ，child 是可變的所以可以一直加下去。

使用BFS來儲存進file





第二個child的word和他所包含child的distance

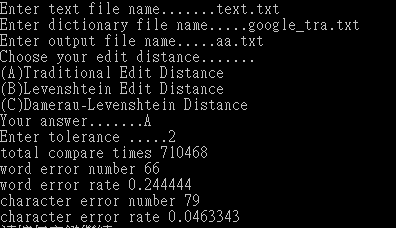
剛剛root第一個child的word和他所包含child的distance

root Node的word

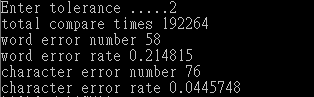
和child的distance

三種BKtree來更正同一文章:

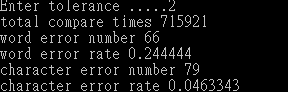
1.traditional:



2. Levenshtein Distance



3. Damerau–Levenshtein



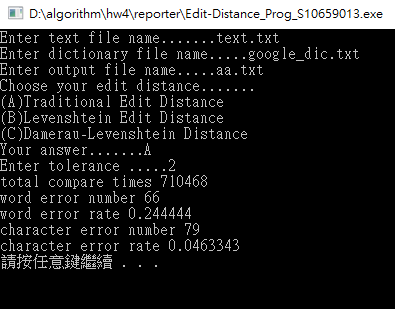
Levenshtein Distance因為距離最大所以找出來的字最少所以比較次數最少，其他的蒐 集出來的字差不多所以比較次數差不多

容忍錯誤值:當容忍錯誤值增加時所搜尋的字數變多所以total compare times 會增加 還有可能因為搜尋的字數增加所以原本搜不到的字也被蒐集出來了所以導致錯誤率增 加。

3. 輸出更正錯誤的字數(Word)與英文字(Character)的數量，以及整篇文章字數 (Words) 和字元數(Characters)的錯誤率

字元錯誤率算法:(total changed word number)/total word number

字數錯誤率算法:(total edit distance)/ total characters



**參考文獻**

[1]

1. 台南大學E-course 演算法課程 Algorithms Report Format