

Homework 1: The String-to-String Correction Problem 閱讀心得

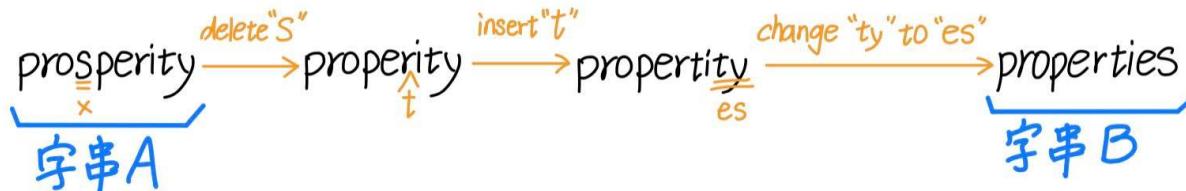
R10942152 游家權

一、問題定義

本論文想解決的問題是如何使用最少的操作把字串A變成字串B。其中能使用的字串操作包含三種：插入(Insert)、刪除(Delete)、置換(Change)。三個操作的代價(Cost)分別由Cost function來定義。目的是找到一連串的字串操作使得加總的代價最低，這個最低的代價就稱為Edit Distance。總結來說，這個演算法就是希望找到字串A,B之間的Edit Distance，還有把字串A變成字串B的操作序列。下圖是例題說明。

例如：

字串A: prosperity 字串B: properties

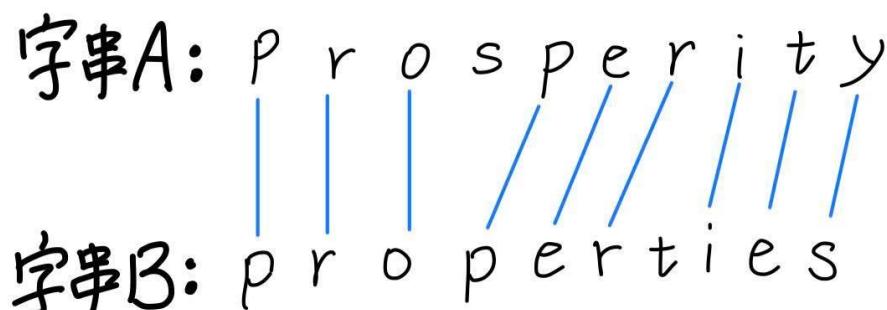


Total Used Operation: $delete \times 1 + Insert \times 1 + change \times 2$

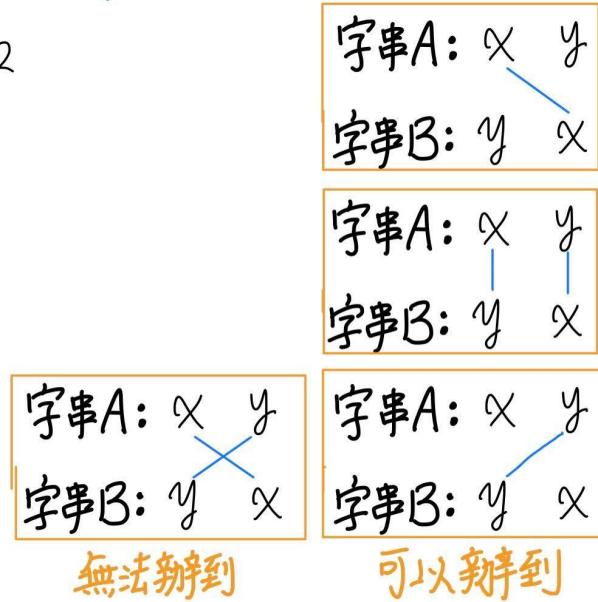
在上圖範例中，字串A透過刪除S、插入T、T置換成E、Y置換成S四個操作之後就可以變成字串B。如果我們假設刪除代價為1、插入代價為1、置換代價為2。那根據這個操作序列，我們可以知道字串A,B的Edit distance就是 $1 \times 1 + 1 \times 1 + 2 \times 2 = 6$ 。

二、觀察問題

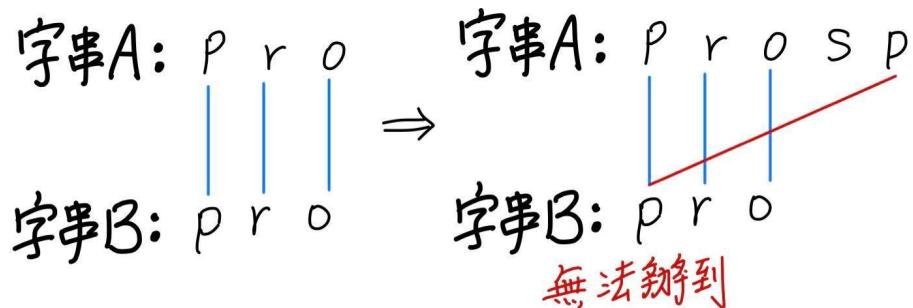
首先，可以先觀察到這個問題是否符合Principle of optimality，也就是小問題最佳解是否會成為大問題的最佳解的一部份。可以將這個問題想成下圖連連看的遊戲：有被線碰到的字母會被置換成字串B的字母，字串A中沒有被線連到的會被刪除，字串B沒有被線連到的會被插入。如此一來，一個連連看遊戲就可以完整表達字串A是如何用字串操作成為字串B的，這在這篇論文裡稱為Traces。



如果認真去看這個連連看的遊戲，我們可以發現藍色的線段之間永遠無法相交，參考下圖簡單的例子。XY要變成YX只有右手邊三種操作可以達到，左邊是無效的操作。



由此我們可以論證連連看的線段之間無法相交，而此性質等同於確保線段之間的有序性，也就是不會發生如果多考慮一個字母，結果發現前面做的Optimal Solution全部被丟掉的情形。由此我們可以確認這個問題符合Principle of optimality，故可以用DP來解。



三、解法敘述

發現這個問題有符合Principle of optimality之後，我們可以用DP的想法來解這個問題，每一次迴圈我們就多考慮一個字母，最終涵蓋的整個Input字串時演算法停止並生成解答。由於每次考慮問題時都會遞歸的使用到以前算過的小問題，所以顯然我們需要一個矩陣來記錄以前計算的小問題的結果，來避免重複計算。

接下來，我們嘗試來寫出Top-down的遞迴式。每次只考慮A, B字串最右邊的字母，只會發生三種可能：將A字串中的最右字母置換成B字串中的、刪除A字串中的最右字母、插入B字串中的最右字母。而演算法將會選擇這三個可能中，代價最小的方法作為這個問

題的Optimal Solution。下圖示意，這個解法會遞迴的呼叫到size比較小的問題，並且每次遞迴呼叫都會至少讓Problem size減少一個字母。

$$D(i, j) = \min \begin{cases} D(i-1, j-1) + A[i] \text{變成} B[j] \text{的代價} \\ D(i-1, j) + \text{刪除} A[i] \text{的代價} \\ D(i, j-1) + \text{插入} B[j] \text{的代價} \end{cases}$$

上圖中，D是我們要解的問題，i, j 分別為A,B字串的indice。

用遞迴式寫出來的解答雖然很簡潔，但是在這篇論文裡面解法是用Bottom-Up的方法來解，因為實務上如果寫Recurrsve的話有可能在Runtime的時候把記憶體的Stack用光，發生錯誤。使用Bottom-Up迴圈的方式進行，不僅比較有效率，記憶體空間也比較容易控管。所以下面的解法步驟將會採用Bottom-Up填表格的方式來解。

最後，我列出詳細的解法步驟，並搭配Properties、Prosperity這兩個字串來舉例說明，為了方便闡述，定義置換的代價為2、刪除的代價為1、插入的代價為1。

(1) 創建矩陣D用於紀錄Optimal Solution，並且將Base case的結果直接填入矩陣之中。

y	9								
t	8								
i	7								
r	6								
e	5								
p	4								
s	3								
o	2								
r	1								
p	0	1	2	3	4	5	6	7	8
		p	r	o	p	e	r	t	i
									s

(2)填表格的順序是由左往右、由下到上來填，確保我們在寫新問題的時候，他所需要的小問題都已經被算完了。而每次要填入新的值時，值的來源只有三種可能：下方的值加一、左方的值加一，或是左下方的值加零，但走左下方只有當A[i] == B[j]時才能這樣走。完成後如下圖。

y	9	8	7	6	5	4	3	4	5	6
t	8	7	6	5	4	3	2	3	4	5
i	7	6	5	4	3	2	3	2	3	4
r	6	5	4	3	2	1	2	3	4	5
e	5	4	3	2	1	2	3	4	5	6
p	4	3	2	1	2	3	4	5	6	7
s	3	2	1	2	3	4	5	6	7	6
o	2	1	0	1	2	3	4	5	6	7
r	1	0	1	2	3	4	5	6	7	8
p	0	1	2	3	4	5	6	7	8	9

p r o p e r t i e s

(3) 最後，我們想要的正確答案就是矩陣最右上角的數字，因此Edit distance就是6。至於想要得到操作的序列就必須做Back tracing，如下圖所示。實務上會需要另外一個矩陣來去紀錄每次選擇的方向，方便back tracing 時找到方向。

y	9	8	7	6	5	4	3	4	5	6
t	8	7	6	5	4	3	2	3	4	5
i	7	6	5	4	3	2	3	2	3	4
r	6	5	4	3	2	1	2	3	4	5
e	5	4	3	2	1	2	3	4	5	6
p	4	3	2	1	2	3	4	5	6	7
s	3	2	1	2	3	4	5	6	7	6
o	2	1	0	1	2	3	4	5	6	7
r	1	0	1	2	3	4	5	6	7	8
p	0	1	2	3	4	5	6	7	8	9

p r o p e r t i e s

由上圖橘色箭頭back tracing的結果，我們可以知道字串A如何變成字串B的，走斜線代表不需要改變，走左邊代表刪除字串、走下面代表插入字串。

四、讀後心得

雖然之前就有在其他地方看過Edit distance的解法，不過當時一直不懂為何會需要求字串之間的距離，看了這篇論文才發現這個演算法原本是想解決打孔機(Keypunched words)修正字串的問題。以前寫程式都是用打孔機在紙上打孔，如果打錯一個字，要修

正回來想必很麻煩，所以這個演算法才想要去找能改正字串的最少修正次數，能讓業務變得有效率。

除了了解演算法開發的動機之外，在推導解法的過程也讓我印象深刻。尤其是把編輯字串轉換成連連看的問題。把問題圖像化，可以方便我們來了解它。更甚之，連連看裡線段不能相交的性質，同時也在暗示這個問題的Optimility跟它可以被DP解決的事實。

另外，我發現在DP解法裡面，如何去定義”問題”是很重要的事情。像在Edit distance跟LCS裡面都是用A,B字串的indice: i, j 來定義現在這個字串做到哪裡了。我一開始想這個問題的時候，老是會想要用四個indice分別代表A,B字串的兩個Substring，但是這麼做不但問題變得很複雜，也沒有必要。後來發現講義跟論文裡，固定左手邊的indice讓他永遠指向字串的頭才是最簡潔的方式，不僅解法優雅，遞迴式寫出來也很優雅。