In *Proceedings of 2001 National Computer Symposium (NCS 2001)*, Vol. A, pp. A115-A122. Taipei, Taiwan. Dec. 20-21, 2001. 中華民國九十年全國計算機會議(NCS 2001)

以估測相似度為基礎的近似字串比對

A Stochastic Similarity Approach for Approximate String Matching

陳貽中
I-Chune Chen

元智大學資訊工程學系
桃園縣中壢市遠東路 135 號
s874127@mail.yzu.edu.tw

楊正仁
Cheng-Zen Yang

元智大學資訊工程學系
桃園縣中壢市遠東路 135 號
czyang@syslab.cse.yzu.edu.tw

摘要

在本篇論文中，我們提出一個以估測相似度為基礎的近似字串比對來改進傳統的近似字串比對演算法。藉由估測相似度矩陣，所提出的演算法在經過與其他近似字串比對演算法比較之後，確實能在個別字元比對上更接近實際的情形，進一步達成更精確的近似字串比對的目標。

關鍵字：近似字串比對、估測相似度、近似字元、相似度矩陣

一、簡介

近似字串比對（approximate string matching）[4,5,7] 是字串比對（string matching）[1,4,7] 以及樣式比對（pattern matching）[4,7] 領域中的一個重要問題。這個問題在早期也被稱之為「容有誤差的字串比對」的問題[9]。這主要是因為在實際的比對過程中，因著使用的環境複雜而往往令使用者除了對完全一樣的字串之外，也對一些稍有「誤差」的字串具有興趣。在某些情況中，由於使用者本身可能對於需要尋找的字串還不太確定，這時更需要將相似的字串結果一併比對出來，供使用者進一步判斷何者才是真正所要找尋的結果。甚至，在某些領域應用中如生物資訊領域，近似字串比對要比精確字串比對（exact string matching）更有用處，因為相同字串出現的比例非常低[6,7]。從 1970 年代起，便有不少研究工作討論近似字串比對，在 [4,5,7] 中便列出超過 150 篇的相關研究文獻。

近似字串比對問題就是在研究，給定特定字串 P（稱作 *pattern*），找出在字串 T（稱作 *text*）中與 P 近似的子字串(substring)的方法。目前常見的近似字串比對演算法可以分成是循序性字串比對（sequential string matching）[7]，以及多重樣式近似比對（multi-pattern approximate matching）[7]。然而目前絕大多數的研究仍偏向循序性字串比對。主要是因為循序性字串比對已經有相當深厚的研究基礎，同時許多其他不同形式的近似比對演算法也都由循序性字串比對演算法擴展演變而來。

在本篇論文中，所提出的近似字串比對演算法，也是屬於循序性近似字串比對演算法中的一種。其主要目的，在於改善傳統循序性近似字串比對演算法中，只單純考慮字元之間不同的比較，卻忽略了每種比對不符

合時的情況，發生的機率未必相等的缺點[6,7]。這是因為在傳統的循序性近似字串比對演算法中，都是以兩個字串的差異性（difference）用不同公式來計算出編輯距離（edit distance，又稱之為 Levenstein distance）。在計算過程中，基本上以刪除、插入或代換單一字元的方式來計算兩個字串之間的差異性。然而這種所謂的 k-differences 的問題，只單純的計算次數，卻並未考慮到這些運算之間發生的機率。但是，實際上任兩個字元之間發生誤差的情況並不一定完全相同。

一個常見的例子就是按鍵輸入錯誤的情況。當使用者者鍵入某個英文字進行查詢時，很可能在輸入某個字元時，誤按到該字元附近的按鍵而未發覺。在這個例子中，誤按到相隔較遠的按鍵的機率，其實是比較低的。這時若以傳統的方式來計算，真正近似的字串反而有可能被認為近似程度不高，因此造成使用者需要多花額外的時間來更正。

在本篇論文中所提的近似字串演算法則加以考慮字元之間的估測相似度（stochastic similarity）。依照所事先設定的估測相似度，將可以依其較精確地找出近似的字串。如此將可避免找到許多其實並不相關的資訊。在演算法的應用中，並可設定一個精確度的 threshold，使用者可依其需要決定比對出的結果應該在多少精確度之上。因此可過濾其實不夠近似的字串結果，也加速整個比對的速度。

我們並設計搜尋字串的實驗來驗證所提出的近似字串比對演算法。以錯誤的字串為輸入，來看看是否能找出原本應該輸入的正確字串。經過與一些傳統的近似字串比對演算法比較，也確實發現所提出的加入估測相似度的近似字串比對演算法能找出正確字串。

本篇論文主要的安排如下：第二節介紹所需的相關背景及演算法。第三節說明以估測相似度為基礎的近似字串比對演算法。第四節說明相關實驗內容並分析比較各演算法的表現。第五節為結論及未來工作展望。

二、相關研究

2.1 背景知識

在近似字串比對演算法中最常以 Levenshtein distance(或稱為 edit distance) [4,5,7]或 Hamming distance [1,4,5,7]來評量二個字串間的差異。一個以 Hamming distance 為依據的問題又稱作 approximate string matching with k mismatches，而一個以 Levenshtein distance 為依據的問題又稱作 approximate string matching with k differences。

二個長度為 m 的字串 A, B，它們之間的 Hamming distance 是由累加 A, B 中同一位置但不相同的字元的個數得來的，正式定義如下：

$$Hamming\ distance = \sum_{i=1}^{m} match(A_i, B_i) \quad (1)$$

$$match(A_i, B_i) = \begin{cases} 0 & if\ A_i = B_i \\ 1 & if\ A_i \neq B_i \end{cases} \quad (2)$$

其中 A_i 為字串 A 的第 i 個字元，B_i 為字串 B 的第 i 個字元。

二個字串 A, B(A, B 的長度不必相同)間的 Levenshtein distance 為字串 A, B 間的最小差異(difference)數。Difference [1]出現在下列幾種情形其中之一：

1. 修改(revise)：
   字串 A 的字元在 B 中有相對應不同的字元。
2. 插入(insert)：
   字串 A 的字元在 B 中沒有相對應的字元。
3. 刪除(delete)：

字串 B 的字元在 A 中沒有相對應的字元。

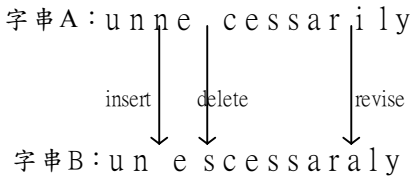

圖 1：Levenshtein distance 的三種計算情形。

圖 1 中描述了這三種情況。在"unnecessarily"與"unescessaraly"兩個單字之間，三種情況分別各出現一次。Levenshtein distance 正式的定義如下：

A_i 表示字串 A 的第 i 個字元, B_i 表示字串 B 的第 i 個字元。a, b 表示二個字元，定義

$$r(a,b)=\begin{cases}0 & if\ a=b\\ 1 & otherwise\end{cases} \qquad (3)$$

假設我們給定二個字串 A 和 B 長度分別為 m, n。我們藉由計算一個 $(m+1)\times(n+1)$ 的整數陣列 d 來求得 Levenshtein distance。d 的定義是遞迴的如下：

對所有的 $d(i, 0), d(0, j)$：

$$d(i,0)=i, \quad i=0,1,\cdots,m \qquad (4)$$

$$d(0,j)=j, \quad j=0,1,\cdots,n \qquad (5)$$

其他的 $d(i,j)$：

$$d(i,j)=\min(\,d(i\text{-}1,j)+1,\ d(i,j\text{-}1)+1,\ d(i\text{-}1,j\text{-}1)+r(A_i,B_i)\,) \qquad (6)$$

則每一個 $d(i, j)$代表字串 A 前 i 個字串和字串 B 前 j 個字串的 Levenshtein distance。

2.2 Robust String Matching

Bret 利用二步驟來達成所謂的 robust string matching [3]。第一步驟，將使用者所輸入的字串與其他所有的字串（可能來自資料庫或字典檔等等）作一個快速但較不精確的近似比對，而得到一個較短的相似字串列表。第二步驟，從前一個步驟得到的列表中再做更精確的評估，這一步驟利用遞迴的方法來比對字串，執行較慢但是判斷的更精確。

在 Bert 的作法中，雖然在最壞情況下，其演算法將需要花 $O(3^{\min(m,n)})$，但是因為經過 K 值的控制，所需要花費的時間將遠小於 $O(3^{\min(m,n)})$。另外，經過兩個步驟的篩選，因此實際上已經大為減少所需要的計算。同時如果有兩個相鄰字元交換（swap）的情況時，也可以算出較小的距離，得到比較精確的效果。

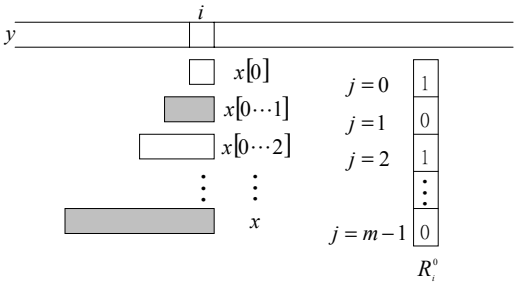

圖 2：R^0 的含義

2.3 Wu-Manber Algorithm

Wu 和 Manber 在 1992 年提出一個近似字串比對演算法，我們在本篇論文中稱作 Wu-Manber algorithm [9]。他們的演算法用來解決字串 x (長度為 m)與字串 y (長度為 n)的最多 k differences 近似字串比對問題 (approximate string matching with at most k differences)。這個演算法定義了 $k+1$ 個大小為 m 的位元陣列(bit array) $R^0, R^1, \cdots, R^k$，且對所有 a 屬於字母表(alphabet)，定義 S_a 如下：

$$for\ 0\le j\le m\text{-}1,\quad S_a[j]=0\ iff\ x[j]=a \qquad (7)$$

R_i^0 表示近似比對進行到 $y[i]$ 時陣列 R^0 的值(如圖 2 所示)，它的值含有字串 x 與字串 y 在位置 i 時的比對資訊：

$$R_i^0[j]=\begin{cases}0 & if\ x[0\cdots j]=y[i-j\cdots j]\\ 1 & otherwise\end{cases} \qquad (8)$$

R_i^0 也可以由下列的遞迴關係來得到：

$$R_i^0[j]=\begin{cases}0 & if\ R_{i-1}^0[j-1]=0\ and\ x[j]=y[i]\\ 1 & otherwise\end{cases} \quad (9)$$

$$R_i^0[0]=\begin{cases}0 & if\ x[0]=y[0]\\ 1 & otherwise\end{cases} \quad (10)$$

而其他的陣列 R^j $(1\le j\le k)$ 可由下列公式求得：

$$\begin{aligned}R_i^j=&\left(SHIFT\left(R_{i-1}^j\right)OR\ S_{y[i]}\right)\\ &AND\ SHIFT\left(R_i^{j-1}\right)\\ &AND\ SHIFT\left(R_{i-1}^{j-1}\right)\\ &AND\ R_{i-1}^{j-1}\end{aligned} \quad (11)$$

簡化之後得到：

$$\begin{aligned}R_i^j=&\left(SHIFT\left(R_{i-1}^j\right)OR\ S_{y[i]}\right)\\ &AND\ SHIFT\left(R_i^{j-1}\ AND\ R_{i-1}^{j-1}\right)\\ &AND\ R_{i-1}^{j-1}\end{aligned} \quad (12)$$

若 $R_i^j[r]$ 的值等於 0，表示 $x[0\cdots r]$ 和 $y[i-r\cdots i]$ 最多有 j 個 differences。

在 Wu-Manber algorithm 中，需要 $O(m|\Sigma|)$ 來 preprocessing，當比對時，通常需要花 $O(nk)$ 的時間。但是 Wu-Manber algorithm 只針對單純的 unit distance 來改善演算法的效能，並未考慮事實上各種情況可能發生的機率並不相同。

2.4 Frequency Distribution

Phillips 提出一個以頻率分佈(frequency distribution)為基礎的近似字串比對演算法[8]，一個單字的頻率分佈表示單字中每個字母出現的次數。例如，“PEOPLE”的頻率分佈為：P-2, E-2, O-1, L-1。我們可以藉由二個字串的頻率分佈來比較它們的相似程度。

假設我們利用此演算法來對字串 *A, B* 作近似比對。首先，計算字串 *A, B* 的頻率分佈並相減，將結果存於陣列 freq_count[]中。接下來，對陣列 freq_count[]取絕對值。此時 freq_count[]中每個元素累加起來的值在此稱作 *divergence*，表示字串 *A, B* 間近似的程度，divergence 越小表示二個字串越接近。字串“approximate”和“appointment”的頻率分佈差如下所示：

表 1：freq_count[ ]的值

freq_count[ ]	‘a’	‘p’	‘r’	‘o’	‘x’
值	1	0	1	0	1
freq_count[ ]	‘m’	‘t’	‘e’	‘n’	others
值	0	1	0	1	0

在這個例子中的 divergence 為 5。頻率分佈的方法在一般狀況下的表現都不錯，但它卻沒辦法分辨由相同的一組字母排列組合後產生的單字。例如，此演算法無法分辨“lisp”和“lips”的不同，因為它們的頻率分佈都是：(l,1), (i,1), (s,1), (p,1)。其中一個可行方法是檢視字串的頭尾字母，如果二個字串的第一個字母或最後一個字母是相同的，則減少 divergence。若是頭尾的字母都不相同，就增加 divergence 的值。這樣就可以有效的解決這個問題，但是必須多花費一些時間在比對頭尾的字母上。

2.5 Weighted Edit Distance

Kurtz 提出兩個以加權 edit distance 的近似字串演算法 rSESA 與 lazySESA 來改善傳統近似字串演算法中並未考慮到不同權重的情況[6]。在這些演算法中，是先針對字串 p 來建立一個 DFA (deterministic finite automaton)。接下來以這個 DFA 進行 $O(n)$ 的近似字串比對。基本上，DFA 的建立在最壞情況下將需要 $O(m\cdot|\Sigma|^{m+k+1})$ 的時間與 $O(m^2+|\Sigma|^{m+k+1})$ 的空間。在實際應用上，這將是很大的代價。因此在 rSESA，中利用一個參數 r 來控制 DFA 的深度，使建立時間為

$O(m\cdot|\Sigma|^{r+1}+qm+n)$ 。 Kurtz 並進而在 lazySESA 中應用 lazy evaluation 的技術來化簡 DFA 的狀態及 transition。不過 lazySESA 最差的情況下仍需要 $O(n(m+|\Sigma|))$ 的時間。然而這兩者所需要的空間仍然不小。同時，對每一個不同的字串 p 必須建立起對應的 DFA，也將造成實際應用上的困擾。

三、以估測相似度為基礎的近似字串比對

如前所述，目前相關的近似字串比對演算法多只單純考慮字元間的比較，卻忽略了每種比對不符合時的情況，發生的機率未必相等。例如，一般人利用鍵盤輸入文字時，拼錯單字的原因常常是因為不小心按到鄰近區域的字母。如圖 3 所示，原本要按'S'，很容易不小心按到'S'鄰近區域的字母'A', 'Q', 'W', 'E', 'D', 'C'或'X'。

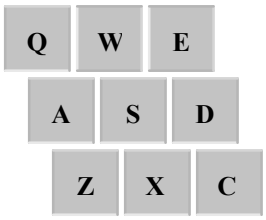

圖 3：'S'鄰近區域的字母

因此，我們以「估測相似度」(stochastic similarity)，標註出二個字元間的相似度的關係。若字元 α、β 間的相似度越高，表示 α 與 β 越容易混淆，越容易不小心將 α 輸入成 β。例如'S'與鄰近區域的字母'A', 'Q', 'W', 'E', 'D', 'C'或'X'的相似度就比非鄰近區域的字母高。

若字母表(alphabet)中含有 n 個元素 $\alpha_1,\alpha_2,\cdots,\alpha_n$，則我們可以建立一個 $n\times n$ 的相似度矩陣 A：

$$(A)_{ij}=\begin{cases} closeness(\alpha_i,\alpha_j) & if\ i\neq j \\ 1 & if\ i=j \end{cases} \qquad (13)$$

$closeness(\alpha_i,\alpha_j)$ 為字元 α_i 與 α_j 間的相似度，$closeness(\alpha_i,\alpha_j)$ 的值越大，表示相似度越高。$closeness(\alpha_i,\alpha_j)$ 的值可利用統計的結果得到，其中一個可行的計算方法為

$$closeness(\alpha_i,\alpha_j)=\Pr(\alpha_i,\alpha_j),\quad 0\le\Pr(\alpha_i,\alpha_j)\le 1,\forall i\neq j \qquad (14)$$

其中 $\Pr(\alpha_i,\alpha_j)$ 為 α_i 與 α_j 可能的相似度。

表 2：相似度矩陣

	A	B	C	D	E	…	S	T	…
A	1	0	0	0.5	0.5	…	…	…	…
B	0	1	…	…	…	…			
C	0	…	…	…	…				
D	0.5	…	…	1					
E	0.5	…	…		1		0.7		
⋮	…	…				…			
S	…				0.7		1		
T	…							1	
⋮	…								1

例如在前面鍵盤輸入的例子中，由於'E'與'S'較相近，我們可以看到表 2 中 $(A)_{'E''S'}$ 的值比矩陣中其他元素的值大。

此相似度矩陣的值在起始時，是系統事先的一個估測值。這些估測值將可能因為輸入裝置的不同或其他因素而改變。例如使用語音輸入法時，因為 c 和 k，s 和 z，m 和 n…的發音相似，容易造成語音辨識軟體的混淆，所以此時 c 和 k 之間就非常相似。另外一個例子是，在使用手寫輸入法時，C 就和 G 非常相似，因為它們的外型相像，手寫辨識軟體不易分辨它們之間的不同；相同的情形還有 5, S 之間， o, 0, D 之間，i, I, l, 1 之間也不易區分。由此可知相似度矩陣的內容，應隨著輸入裝置的不同或其他因素而改變，才能正確地反應出字元間的相似度。

我們選擇 Bert 的演算法來加入「估測相似度」以改良成為主要的演算法架構，主要

原因乃是因為，第一，所需要的資料結構單純，不需要因著不同的比對字串來建立不同的資料結構。第二，演算法的效能可經由 k 值適當控制而有不錯的表現。第三、原先的演算法能很容易加上估測相似度的考慮。

估測相似度為基礎的近似字串比對演算法如下：

Algorithm:

Step1: Establish a similarity-matrix A by stochastic approach.

Step2: Read in two string x of length m, y of length n and maximal difference k.

Step3: Initialize integer r to 0.

Step4: Call a recursive function *string_match*.

圖 4 中說明了我們所使用的近似字串比對核心函式 string_match()。其中 k 在原本的演算法中表示的是距離，但在此處則表示的是一種精確度。所用到的 *r, r1, r2, r3*，表示的是所算出來的累計精確度。

Function string_match()

Input: string x, y and integer r

Output: r: distance between x and y, $r \le k$

Method:

Let $d = 1 - (A)_{x_0 y_0}$.

Initialize *z, r1, r2, r3* to *(r+d)*.

If $r > k$

Return

Else

find i such that $x_i \neq y_i$

If $x_i = nil$

$r = r$ +length of y_i to y_n ;

Return

If $y_i = nil$

$r = r$ + length of x_i to x_m ;

Return

string_match(x_{i+1} , y_{i+1} , *r3*);

string_match(x_i , y_{i+1} , *r1*);

string_match(x_{i+1} , y_i , *r2*);

r = min(*r1, r2, r3*);

Return

圖 4：近似字串比對核心函式 string_match()。

四、分析評估比較

我們以兩個實際的搜尋相似字串的應用來比較不同演算法的表現。在實驗中，我們選用一個隨 WinEdt 軟體所附的字典檔來查詢。如果換用其他的字典檔，應該也有類似的結果出現。

我們將以估測相似度為基礎的近似字串比對演算法與其他相關的演算法作比較。由於 frequency distribution 方法計算字串差異的方法特別不同，所以我們設定 frequency distribution 方法中的 divergence 為 2，至於 robust string matching, Wu-Manber algorithm 和 stochastic approach 中的 distance 都取 1，然後對字典檔作比對找出字典檔中所有的近似字串。有一點值得注意的，在 stochastic approach 中，由於字元比較不同時，得到的 $(A)_{x_i y_j}$ 可能是浮點數，所以最後計算出來的 distance 也將會是浮點數而不一定是正整數，所以我們在 stochastic approach 中所標明的 distance 指的是一個 distance 的上界，所有小於或等於該 distance 之值的字串都會被尋找出來。我們在此處並不與 Kurtz 的演算法比較，因為這個演算法的參數相當多，因此不易得出比較結果。

在實驗中，為了簡化 stochastic approach 的比對情況，我們以鍵盤按鍵輸入錯誤為範例。所有相鄰的兩個字元，其估測相似度設為 0.5。如(h,j,0.5)。不相鄰的則設為 0，如(h,k,0)。因此我們可依此建立起一個相似度矩陣。

- 實驗一：h 誤輸入成 j

假設原本要輸入 “humor”，卻誤輸入成“jumor”，則各個演算法輸出的近似字串如下：

演算法	所搜尋出來的字串
Robust String Matching	humor juror rumor tumor
Frequency Distribution	our rom amour forum humor jumbo juror major mourn rumor tumor
Wu-Manber Algorithm	humor juror rumor tumor jurors rumors tumors rumored conjuror humorous rumoring conjurors humorously
Stochastic Approach	**(distance＝0.5)** humor juror **(distance＝1)** judo jump jumbo rumor tumor jumper junior jurors

在本實驗中，使用者按錯了一個字母而輸入了“jumor”，我們可以看到四種演算法都能順利的找到“humor”。但經過我們分析之後，發現 robust string matching 和 Wu-Manber algorithm 中所列出的字串的其實 distance 都為 1，“humor”會列在前面主要是因為字母排列順序所為。Frequency distribution 列出的字串 divergence 都是 2，而且經過字母順序排列之後，更排在後面。Stochastic approach 則找出 distance 分別為 0.5 及 1 的近似字串。列出的字串中，“humor”、 “juror”比其他字串更為近似。這是由於我們的演算法中加入了字元間相似度的考量，因此在比對上更為敏銳。如果不是恰好因我們所找的字具有字母排列順序，stochastic approach 會有更好的表現。

- 實驗二：再將 m 誤按成 n

假設原本要輸入 “humor”，卻誤輸入成“junor”，則各個演算法輸出的近似字串如下：

演算法	所搜尋出來的字串
Robust String Matching	junor juror junior
Frequency Distribution	**(divergence＝1)** junior **(divergence＝2)** jun nor our run urn juror mourn round adjourn conjure journal journey
Wu-Manber Algorithm	juror junior jurors conjuror conjurors
Stochastic Approach	**(distance＝0.5)** juror junior **(distance＝1)** jun june judo junk humor minor junker jurors

在本實驗中，雖然由於錯誤增多，使得找到“humor”排列的順序延後，我們發現只有 stochastic approach 可以正確地找到“humor”。這是由於其他的方法對字元比對不符的情形所發生的機率都一視同仁，因此無法搜尋出真正較為接近的字串。然而實際上，每種比對不符合發生的情形的機率並不相等。因此，建立一個相似度矩陣，藉由參考矩陣的內容，可以有效地反應實際的情形，達成更精確的近似字串比對。

五、結論及未來工作展望

近似字串比對在許多應用領域上是一個重要的研究課題。其應用領域包含生物資訊中的基因序列尋找，Web 上搜尋引擎的核心，資訊擷取等等。然而在傳統的近似字串比對演算法中，偏重於單純的字元之間的相異性，並未充分考慮它們之間相似的可能性。

在本文中我們對「估測相似度」加以考量，在傳統的近似字串演算法基礎上加以改良，計算二個字元間的相似度關係。藉由推測字元間的相似度，建立一個相似度矩陣來計算字串間的 distance，並由實際的例子中證明可有效的提高近似字串比對的準確度。

由於我們以 Bert 的演算法為基礎，加上

我們更詳細考慮估測相似度，因此整個演算法所花費的時間將多過於$O(3^{\min(m,n)})$。但經由 K 值控制，則能大幅減少所花費的執行時間。

然而，目前相似度矩陣的內容是一種靜態的設定，並不會隨環境不同而自動動態調整。這在實際應用上，將會造成很大的限制，因此，動態調整相似度矩陣是我們未來進一步研究的方向之一。

另外，我們所提出的演算法還缺乏更詳細的演算法複雜度分析，這也將是我們需要更進一步探討的地方。

六、參考文獻

[1] S. Baase and A. Gelder, "Computer Algorithms ─ Introduction to Design and Analysis", 3rd Ed., Addison-Wesley, 2000.

[2] A. Bogomolny, "Interactive Mathematics Miscellany and Puzzles", http://www.cut-the-knot.com/do_you_know/Strings.html

[3] C. Bret, "Robust String Matching", Computer Language, Dec. 1991, pp. 71- 83.

[4] M. Crochemore and T. Lecroq, "Pattern matching and text data compression algorithms", in "The Computer Science and Engineering Handbook", Allen B. Tucker Jr., ed., chapter 6, pp. 162-202, CRC Press Inc., Boca Raton, FL, 1996.

[5] P. Hall and G. Dowling, "Approximate String Matching", ACM Computing Survey, Vol. 12, No. 4, pp. 381-402, Dec. 1980.

[6] S. Kurtz, "Approximate String Searching under Weighted Edit Distance", in Proc. of the 3rd South American Workshop on String Processing, pp. 156-170, 1996.

[7] G. Navarro, "A Guided Tour to Approximate String Matching", ACM Computing Survey, Vol. 33, No. 1, pp.31-88, Mar. 2001.

[8] T. Phillips, "Approximate string matching", C/C++ Users J. Vol. 12, No.4, Apr. 1994, Article 5.

[9] S. Wu and U Manber, "Fast Text Searching Allowing Errors", Comm. of the ACM, Vol.35, No. 10, pp. 83-91, Oct. 1992.