Algorithms for Approximate String Matching

Part I
Levenshtein Distance
Hamming Distance
Approximate String Matching with k Differences
Longest Common Subsequences

Part II
"A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequences Problem"

Yoan Pinzón
Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
ypinzon@unal.edu.co
www.pinzon.co.uk

August 2006

Levenshtein Distance is named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965. If you cannot spell or pronounce Levenshtein, the metric is also called *edit distance*.

The edit distance $\delta(p, t)$ between two strings $p$ (pattern) and $t$ (text) ($m = |p|, n = |t|$) is the minimum number of insertions, deletions and replacements to make $p$ equal to $t$.

- **[Insertion]** insert a new letter $a$ into $x$. An insertion operation on the string $x = vw$ consists in adding a letter $a$, converting $x$ into $x' = vaw$.

- **[Deletion]** delete a letter $a$ from $x$. A deletion operation on the string $x = vaw$ consists in removing a letter, converting $x$ into $x' = vw$.

- **[Replacement]** replace a letter $a$ in $x$. A replacement operation on the string $x = vaw$ consists in replacing a letter for another, converting $x$ into $x' = vbw$. 
Example 1:

\[ p = "\text{approximate} \text{ matching}" \]
\[ t = "\text{appropriate} \text{ meaning}" \]

String \( t \): appropriate meaning

\[
\begin{array}{cccccccccccccc}
\hline
\end{array}
\]

String \( p \): approximate matching

\[
\begin{array}{cccccccccccccc}
\hline
\end{array}
\]

\[ \delta(t, p) = 7 \]

Example 2:

\[ p = "\text{surgery}" \]
\[ t = "\text{survey}" \]

String \( t \): survey

\[
\begin{array}{cccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array}
\]

String \( p \): surgery

\[
\begin{array}{cccccccccccccc}
\hline
\end{array}
\]

\[ \delta(t, p) = 2 \]

Solution: Using Dynamic Programming (DP):
We need to compute a matrix \( D[0..m, 0..n] \), where \( D_{i,j} \) represents the minimum number of operations needed to match \( p_{1..i} \) to \( t_{1..j} \).

This is computed as follows:

\[
D[i, 0] = i \\
D[0, j] = j \\
D[i, j] = \min\{D[i-1, j] + 1, D[i, j-1] + 1, D[i, j] + \delta(p_i, t_j)\}
\]

\[ \delta(p, t) = D[m, n] \]

Pseudo-code:


def ED(p, t):
    m = |p|, n = |t|
    for i ← 0 to m do
        D[i, 0] ← i
    for j ← 0 to n do
        D[0, j] ← j
    for i ← 1 to m do
        for j ← 1 to n do
            if \( p_i = t_j \) then
                D[i, j] ← D[i-1, j-1]
            else
                D[i, j] ← \min(D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + \delta(p_i, t_j))
        od
    od
    return D[m, n]

called

Running time: \( O(nm) \)
A Graph Reformulation for the Edit Distance problem

The Dynamic Programming matrix $D$ can be seen as a graph where the nodes are the cells and the edges represent the operations. The cost (weight) of the edges corresponds to the cost of the operations.

$$\delta(t, p) = \text{shortest-path from node } [0,0] \text{ to the node } [n,m].$$

**Running time:** $O(nm \log(nm))$

**Example:** $p=\text{"survey"}$, $t=\text{"surgery"}$

String $t$: surger y

String $p$: surve ey

![Diagram](image-url)
Hamming Distance

The Hamming distance $H$ is defined only for strings of the same length. For two strings $p$ and $t$, $H(p, t)$ is the number of places in which the two strings differ, i.e., have different characters.

- **Examples:**
  
  - $H("pinzon", "pinion") = 1$
  - $H("josh", "jose") = 1$
  - $H("here", "hear") = 2$
  - $H("kelly", "belly") = 1$
  - $H("AAT", "TAA") = 2$
  - $H("AGCAA", "ACATA") = 3$
  - $H("AGCACACA", "ACACACTA") = 6$

- **Pseudo-code:** too easy!!

- **Running time:** $O(n)$

Approximate String Matching with $k$ Differences

- **Problem:** The $k$-differences approximate string matching problem is to find all occurrences of the pattern string $p$ in the text string $t$ with at most $k$ differences (substitution, insertions, deletions).

- **Solution:** Using DP

  
  $D[i, 0] = i$
  $D[0, j] = 0$
  $D[i, j] = \min\{D[i - 1, j] + 1, D[i, j - 1] + 1, D[i, j] + \delta(p_i, t_j)\}$

  if $D[m, j] \leq k$ then we say that $p$ occurs at position $j$ of $t$. 


**Pseudo-code:**

1. procedure KDifferences(p, t, k) \( \{m = |p|, n = |t|\}\)  
2. begin  
3. for i ← 0 to m do \( D[i, 0] \leftarrow i \)  
4. for j ← 0 to n do \( D[0, j] \leftarrow 0 \)  
5. for i ← 1 to m do  
6. for j ← 1 to n do  
7. if \( p_i = t_j \) then \( D[i, j] \leftarrow D[i - 1, j - 1] \)  
8. else \( D[i, j] \leftarrow min(D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]) + 1 \)  
9. end  
10. for j ← 0 to n do  
11. if \( D[m][j] \leq k \) then Output(j)  
12. end  
13. end

**Running time:** \( O(nm) \)

**Example:**

\( p \) = "CDDA",  
\( t \) = "CADDACDDBACBA"  
\( k = 1 \)

\( p \) occurs in \( t \) ending at positions 5, 8 and 12.
For two sequences \( x = x_1 \cdots x_m \) and \( y_1 \cdots y_n \) \((n \geq m)\)

we say that \( x \) is a subsequence of \( y \) and equivalently, \( y \) is a supersequence of \( x \), if for some \( i_1 < \cdots < i_p \), \( x_j = y_{i_j} \).

Given a finite set of sequences, \( S \), a longest common subsequence (LCS) of \( S \) is a longest possible sequence \( s \) such that each sequence in \( S \) is a supersequence of \( s \).

Example: \( y = \)"longest", \( x = \)"large"

<table>
<thead>
<tr>
<th>String y: longest</th>
</tr>
</thead>
<tbody>
<tr>
<td>String x: large</td>
</tr>
</tbody>
</table>

\( \text{LCS}(y, x) = \)"lge"

\( \text{Problem:} \) The \textit{Longest Common Subsequence} (LCS) of two strings, \( p \) and \( t \), is a subsequence of both \( p \) and of \( t \) of maximum possible length.

\( \text{Solution:} \) Using Dynamic Programming: We need to compute a matrix \( L[0..m, 0..n] \), where \( L_{i,j} \) represent the LCS for \( p_{1..i} \) and \( t_{1..j} \).

This is computed as follows:

\[
L[i,j] = \begin{cases} 
0, & \text{if either } i = 0 \text{ or } j = 0 \\
L[i-1, j-1] + 1, & \text{if } p_i = t_j \\
\max\{L[i-1, j], L[i, j-1]\}, & \text{if } p_i \neq t_j
\end{cases}
\]

\( \text{Pseudo-code:} \)

```plaintext
procedure LCS(p, t) {m = |p|, n = |t|}
begin
    for i ← 0 to m do L[i, 0] ← 0
    for j ← 0 to n do L[0, j] ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if p_i = t_j then L[i, j] ← L[i-1, j-1] + 1
            else
                if L[i, j-1] > L[i-1, j] then L[i, j] ← L[i, j-1]
                else L[i, j] ← L[i-1, j]
            od
        od
    end
    return L[m][n]
end
```

\( \text{Running time:} \) \( O(nm) \)
Example 1: \( p = \text{"survey"} \) and \( t = \text{"surgery"} \).

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\epsilon & s & u & r & i & g & e & r \\
0 & \epsilon & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & s & 0 & 1 & 1 & 1 & 1 & 1 \\
2 & u & 0 & 1 & 2 & 2 & 2 & 2 \\
3 & r & 0 & 1 & 2 & 3 & 3 & 3 \\
4 & v & 0 & 1 & 2 & 3 & 3 & 3 \\
5 & e & 0 & 1 & 2 & 3 & 3 & 4 \\
6 & y & 0 & 1 & 2 & 3 & 3 & 4 \\
\end{array}
\]

String \( t: \quad \text{surgery} \)

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{String } p: \quad \text{survey} \\
\end{array}
\]

LCS\((p, t) = \text{"surey"} \)

LLCS\((p, t) = L[6, 7] = 5 \)

Example 2: \( p = \text{"ttgatacatt"} \)

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\epsilon & g & a & a & t & a & a & g & a & c & c \\
0 & \epsilon & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & t & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & t & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & g & 0 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\
4 & a & 0 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 \\
5 & t & 0 & 1 & 2 & 2 & 3 & 3 & 3 & 3 & 3 \\
6 & a & 0 & 1 & 2 & 3 & 4 & 4 & 4 & 4 & 4 \\
7 & c & 0 & 1 & 2 & 3 & 4 & 4 & 4 & 5 & 5 \\
8 & a & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 \\
9 & t & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 \\
\end{array}
\]

LCS\((p, t) = ? \)

LLCS\((p, t) = L[9, 10] = 5 \)
Some More Definitions

The ordered pair of positions $i$ and $j$ of $L$, denoted $[i, j]$, is a match iff $x_i = y_j$.

If $[i, j]$ is a match, and an LCS $s_{i,j}$ of $x_1x_2...x_i$ and $y_1y_2...y_j$ has length $k$, then $k$ is the rank of $[i, j]$.

The match $[i, j]$ is $k$-dominant if it has rank $k$ and for any other pair $[i', j']$ of rank $k$, either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$.

Computing the $k$-dominant matches is all that is needed to solve the LCS problem, since the LCS of $x$ and $y$ has length $p$ iff the maximum rank attained by a dominant match is $p$.

A match $[i, j]$ precedes a match $[i', j']$ if $i < i'$ and $j < j'$.
Let $r$ be the total number of matches points, and $d$ be the total number of dominant points (all ranks). Then $0 \leq p \leq d \leq r \leq nm$.

Let $\mathcal{R}$ denote a partial order relation on the set of matches in $L$.

A set of matches such that in any pair one of the matches always precedes the other in $\mathcal{R}$ constitutes a \textit{chain} relative to the partial order relation $\mathcal{R}$.

A set of matches such that in any pair neither element of the pair precedes the other in $\mathcal{R}$ constitutes an \textit{antichain}.

Sankoff and Sellers (1973) observed that the LCS problem translates to finding a longest \textit{chain} in the poset of matches induced by $\mathcal{R}$.

A decomposition of a poset into antichains partitions the poset into the minimum possible number of antichains.
Here we will make use of word-level parallelism in order to compute the matrix $L$ more efficiently.

The algorithm is based on the $O(1)$-time computation of each column in $L$ by using a bit-parallel formula under the assumption that $m \leq w$, where $w$ is the number of bits in a machine word or $O(nm/w)$-time for the general case.

An interesting property of the LCS allows to represent each column in $L$ by using $O(1)$-space. That is, the values in the columns (rows) of $L$ increase by at most one. i.e. $\Delta L[i, j] = L[i, j] - L[i - 1, j] \in \{0, 1\}$ for any $(i, j) \in \{1..m\} \times \{1..n\}$.

In other words $\Delta L$ will use the relative encoding of the dynamic programming table $L$.

$\Delta L'$ is defined as $\text{NOT } \Delta L$. 
Example: $x = \text{"ttgatacatt"}$ and $y = \text{"gaataagacc"}$.

(a) Matrix $L$

(b) Matrix $\Delta L$

Matrix $\Delta L'$

$\Delta L'_6 = <011000101>$
First we compute the array $M$ of the vectors that result for each possible text character. If both the strings $x$ and $y$ range over the alphabet $\Sigma$ then $M[\Sigma]$ is defined as $M[\alpha]_i = 1$ if $y_i = \alpha$ else 0.

Example: $x$= “ttgatacatt” and $y$= “gaataagacc”.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Matrix M

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>G</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Matrix $M'$

**Basic steps of the algorithm**

1. Computation of $M$ and $M'$

2. Computation of matrix $\Delta L' (L)$ as follows:

$$L = \begin{cases} 2^n - 1, & \text{for } j = 0 \\ (L_{j-1} + (L_{j-1} \text{ AND } M(y_j))) \text{ OR } (L_{j-1} \text{ AND } M'(y_j)), & \text{for } j \in \{1..n\} \end{cases}$$

3. Let LLCS be the number of times a carry took place.
Pseudo-code

LLCS(x, y) ▷ n = |y|, m = |x|, p = 0
begin
▷ Preprocessing
for i ← 1 until m do
M[α](i) ← y_i = α
M′[α](i) ← y_i ≠ α
▷ Initialization
L_0 = 2^n - 1
▷ TheMainStep
for j ← 1 until n do
L_j ← (L_{j-1} + (L_{j-1} AND M[y_j])) OR (L_{j-1} AND M′[y_j])
if L_j(m + 1) = 1 then p++
return p
end

Illustration of ΔL'_4 Computation
for x = "gaataagacc" and y = "ttgacatt".
\[
L_4 \leftarrow (L_3 + (L_3 \& M_T)) \mid (L_3 \& M_T')
\]

<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_3)</td>
<td>1 1 1 0 1 0 0 1 1 &amp;</td>
<td>(M_T)</td>
<td>1 0 0 0 1 0 0 1 1</td>
</tr>
<tr>
<td>(L_3)</td>
<td>1 1 1 0 1 0 0 1 1 &amp;</td>
<td>(M_T)</td>
<td>0 1 1 1 0 1 1 0 0</td>
</tr>
<tr>
<td>(1)</td>
<td>1 0 0 0 1 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>0 1 1 1 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>1 0 1 1 1 0 0 1 1 0</td>
<td>(3)</td>
<td>1 0 1 1 1 0 0 1 1 0</td>
</tr>
<tr>
<td>(4)</td>
<td>0 1 1 0 0 0 0 0 0</td>
<td>(4)</td>
<td>0 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Automata for Addition

Experimental Results

- Text Length
- Time (in micro secs.)

Graph showing the relationship between text length and time.