An Efficient Algorithm for Generating Super Condensed Neighborhoods

Luís M. S. Russo* and Arlindo L. Oliveira

June 16, 2005

*Supported by the Portuguese Science and Technology Foundation through program POCTI and project POSI/EEI/10204/2001.

The *edit* distance between two strings *ed(*S, S'*)* is the smallest number of insertions (I), deletions (D) and substitutions (S) that transform S into S'.

For example: `D S I`

`abcd`

ed($abcd$, $bedf$) = 3 `bedf`

Computed by dynamic programming.

Table $D[i,j] = ed(S\langle 0..i\rangle, S'\langle 0..j\rangle)$ is built by the following local relations:

$$D[i,0] = i, \quad D[0,j] = j$$

$$D[i+1,\ j+1] = \begin{cases} D[i,j] & , \text{ if } S[i+1] = S'[j+1] \\ 1 + \min\{D[i+1,j], D[i,j+1], D[i,j]\} & , \text{otherwise} \end{cases}$$

Example: Table $D[i, j]$ for strings $abbaa$ and $ababaac$.

col	0	1	2	3	4	5	6	7
row		a	b	a	b	a	a	c
0	0	1	2	3	4	5	6	7
1 a	1	0	1	2	3	4	5	6
2 b	2	1	0	1	2	3	4	5
3 b	3	2	1	1	1	2	3	4
4 a	4	3	2	1	2	1	2	3
5 a	5	4	3	2	2	2	1	2

k - Error bound

Definition: A cell is active iff its value $\leq k$.

Example: take $k = 1$.

row \ col	0	1 a	2 b	3 a	4 b	5 a	6 a	7 c
0	0	1	2	3	4	5	6	7
1 a	1	0	1	2	3	4	5	6
2 b	2	1	0	1	2	3	4	5
3 b	3	2	1	1	1	2	3	4
4 a	4	3	2	1	2	1	2	3
5 a	5	4	3	2	2	2	1	2

Problem : Find matches of S in S' with at most k errors.

Solution : Table $D'[i,j] = \min_{0 \le l \le j}\{ed(S\langle 0..i\rangle, S'\langle l \ .. \ j\rangle)\}$.

$D'[i,0] = i, \quad D'[0,j] = 0$

$$D'[i+1, \ j+1] = \begin{cases} D'[i,j] & , \text{ if } S[i+1] = S'[j+1] \\ 1 + \min\{D[i+1,j], D[i,j+1], D[i,j]\} & , \text{otherwise} \end{cases}$$

Example: Found a match of $abbaa$ in $ababaac$ with 1 error.

col	0	1	2	3	4	5	6	7
		a	b	a	b	a	a	c
	0	0	0	0	0	0	0	0
a	1	0	1	0	1	0	0	1
b	2	1	0	1	0	1	1	1
b	3	2	1	1	1	1	2	2
a	4	3	2	1	2	1	1	2
a	5	4	3	2	2	2	1	2

Indexed Approximate Pattern Matching

Finding occurrences of S in S' in sub-linear time $O(|S'|^{\alpha})$ ($\alpha < 1$).

Use index structure like suffix arrays, q-grams.

Hybrid algorithms balance between neighbourhood generation and filtration techniques.

Neighbourhood Generation

Generate all the words at distance k from S look them up in the index.

k-neighbourhood of S is $U_k(S) = \{S' \in \Sigma^* : ed(S, S') \leq k\}$

$|U_k(S)| = O(|S|^k|\Sigma|^k)$ quite large.

1-neighbourhood of $abbaa$

U_1: abaa, abba, abbba, abbca, abcaa, abcbaa, acbaa, bbaa
aabaa, ababaa, acbbaa, babbaa, bbbaa, cabbaa, cbbaa, aabbaa
abaaa, abbaa, abbaaa, abbaac, abbaab, abbab, abbaba,
abbac, abbaca, abbbaa, abbcaa

Definition: The *condensed k-neighbourhood* (CU_k) of S is the largest subset of $U_k(S)$ such no proper prefix of an element is at distance $\leq k$ from S.

CU_1: abaa, abba, abbba, abbca, abcaa, abcbaa, acbaa, bbaa aabaa, ababaa, acbbaa, babbaa, bbbaa, cabbaa, cbbaa, aabbaa

U_1: abaaa, abbaa, abbaaa, abbaac, abbaab, abbab, abbaba, abbac, abbaca, abbbaa, abbcaa

Definition: The *super condensed k-neighbourhood* (SCU_k) of S is the largest subset of $U_k(S)$ such no proper substring of an element is at distance $\leq k$ from S.

SCU_1: abaa, abba, abbba, abbca, abcaa, abcbaa, acbaa, bbaa

CU_1: aabaa, ababaa, acbbaa, babbaa, bbbaa, cabbaa, cbbaa, aabbaa

U_1: abaaa, abbaa, abbaaa, abbaac, abbaab, abbab, abbaba, abbac, abbaca, abbbaa, abbcaa

Algorithm 1 Condensed Neighbourhood Generator Algorithm

```
procedure Search(Search State s, Current String v)
    if Is_Match_State(v) then
        Report(v)
    else if Extends_To_Match_State(s) then
        for z ∈ Σ do
            s' ← Update(s, z)
            Search(s', v.z)
        end for
    end if
end procedure
Search(⟨0, 1, ..., |P|⟩, ε)
```

s - dynamic programming column of D.

Is_Match_State - is the last cell active ?

Extends_To_Match_State - Are there active cells ?

Update - apply a to s.

Take s as column 5.

Is_Match_State - is false since $D[5,5]$ is inactive.

Extends_To_Match_State - is true since $D[4,5]$ is active.

row \ col		0	1 a	2 b	3 a	4 b	5 a	6	7
0		0	1	2	3	4	5		
1	a	1	0	1	2	3	4		
2	b	2	1	0	1	2	3		
3	b	3	2	1	1	1	2		
4	a	4	3	2	1	2	1		
5	a	5	4	3	2	2	2		

Is_Match_State - is true since $D[5,6]$ is active.

Column 7 is never evaluated.

row \ col	0	1 a	2 b	3 a	4 b	5 a	6 a	7
0	0	1	2	3	4	5	6	
1 a	1	0	1	2	3	4	5	
2 b	2	1	0	1	2	3	4	
3 b	3	2	1	1	1	2	3	
4 a	4	3	2	1	2	1	2	
5 a	5	4	3	2	2	2	1	

skipping case b.

Is_Match_State - is false since $D[5,6]$ is inactive.

Extends_To_Match_State - is false since there are no active cells.

row \ col		0	1 a	2 b	3 a	4 b	5 a	6 c	7
0		0	1	2	3	4	5	6	
1	a	1	0	1	2	3	4	5	
2	b	2	1	0	1	2	3	4	
3	b	3	2	1	1	1	2	3	
4	a	4	3	2	1	2	1	2	
5	a	5	4	3	2	2	2	2	

The previous algorithm generated $ababaa$ as a member of $CU_1(abbaa)$.

But $ab\mathit{aba}a$ is not in $SCU_1(abbaa)$.

How to change the previous algorithm to generate SCU_k?

Let us look at table D'.

abbaa is at distance 1 from ababaa but abbaa is also at distance 1 from abaa.

col	0	1	2	3	4	5	6
		a	b	a	b	a	a
	0	0	0	0	0	0	0
a	1	0	1	0	1	0	0
b	2	1	0	1	0	1	1
b	3	2	1	1	1	1	2
a	4	3	2	1	2	1	1
a	5	4	3	2	2	2	1

Definition: A traceback is a pointer to a predecessor cell, classified as:

vertical $D'[i+1,j] \rightarrow D'[i,j]$ iff $D'[i+1,j] = 1 + D'[i,j]$

diagonal $D'[i+1,j+1] \rightarrow D'[i,j]$ iff
$D'[i+1,j+1] = 1 + D'[i,j]$ or $S[i+1] = S'[j+1]$

horizontal $D'[i,j+1] \rightarrow D'[i,j]$ iff $D'[1,j+1] = 1 + D'[i,j]$

Definition: A canonical traceback for $D'[i,j]$ is the first traceback that $D'[i,j]$ has got in the ordering above.

Definition: A canonical path is a path in D' made of canonical tracebacks.

Obs: Canonical paths show the rightmost position of a minimal match between S and S'.

Definition: An improper canonical path is a path that ends in $D[0,0]$.

Definition: An improper cell has an improper canonical path.

Idea: restrict our algorithm to improper cells.

Improper cells marked with 0's.

col	0	1	2	3	4	5	6	7
	a		b	a	b	a	a	c
	0	0	0	0	0	0	0	0
a	$\uparrow^0$ 1	$\nwarrow^0$ 0	1 ⋮ 1	1.. 0	1 ⋮ 1	1.. 0	1.. 0	1 ⋮ 1
b	$\uparrow^0$ 2	0 $\uparrow$ 1	$\nwarrow^0$ 0	1 ⋮ 1	1.. 0	1 ⋮ 1	1 ⋮ 1	1.. 1
b	$\uparrow^0$ 3	0 $\uparrow$ 2	0 $\uparrow$ 1	$\nwarrow^0$ 1	1 ⋮ 1	1.. 1	1 ⋮ 2	1 ⋮ 2
a	$\uparrow^0$ 4	0 $\uparrow$ 3	0 $\uparrow$ 2	$\nwarrow^0$ 1	1 ⋮ 2	1.. 1	1.. 1	1 ... 2
a	$\uparrow^0$ 5	0 $\uparrow$ 4	0 $\uparrow$ 3	0 $\uparrow$ 2	$\nwarrow^0$ 2	1 ⋮ 2	1.. 1	1.. 2

Idea: Use the same algorithm with new predicates.

s - dynamic programming column of D'.

Is_Match_State - is the last cell active and improper ?

Extends_To_Match_State - Are there improper active cells ?

Update - apply a to s.

In our example the algorithm backtracks in column 4.

Bit Parallel Implementation

Vertical Positive $VP[i+1,j] = 1$ iff $D[i+1,j] - D[i,j] = 1$

Vertical Negative $VN[i+1,j] = 1$ iff $D[i+1,j] - D[i,j] = -1$

Horizontal Positive $HP[i,j+1] = 1$ iff $D[i,j+1] - D[i,j] = 1$

Horizontal Negative $HN[i,j+1] = 1$ iff $D[i,j+1] - D[i,j] = -1$

Diagonal Zero $D0[i+1,j+1] = 1$ iff $D[i+1,j+1] = D[i,j]$

Pattern Match Vectors $PM_z[i] = 1$ iff $P[i] = z$, for each $z \in \Sigma$

Algorithm 2 C-style Bit-Parallel Algorithm

1: **procedure** Update(Previous Column (VP, VN), Letter z)
2: $D0 \leftarrow (((PM_z \ \& \ VP) + VP) \hat{} VP) | PM_z | VN$
3: $HP \leftarrow VN | \sim (D0 | VP)$
4: $HN \leftarrow VP \ \& \ D0$
5: $VP \leftarrow (HN << 1) | \sim (D0 | (HP << 1))$
6: $VN \leftarrow (HP << 1) \ \& \ D0$
7: **return** VP, VN
8: **end procedure**

Improper Cells $CP1[i,j] = 1$ iff $D'[i,j]$ is a proper cell.

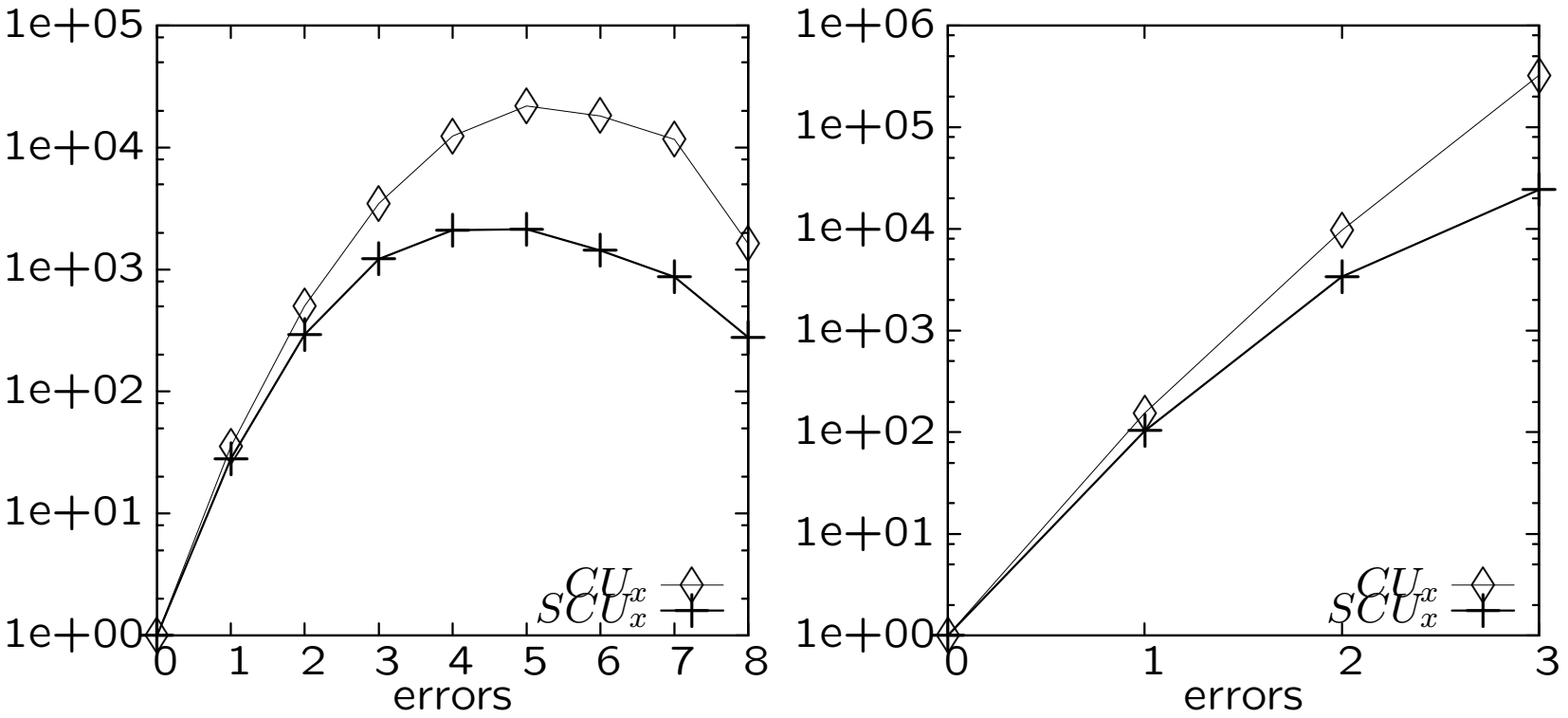
Algorithm 3 C-style Bit-Parallel Algorithm

```
1: procedure Update_Proper_Cells(CP1, PM, HN, VN, VP)
2:     CP1 ← ((PM|HN| ~ VN) & ((CP1 << 1)|1))|CP1      ▷ horizontal dep ?
3:     CP1| ← VP                                        ▷ add vertical deps
4:     CP1 ← (CP1 >> 1)
5:     CP1 ← (CP1 + 1)^CP1   ▷ propagate improper cells and clean up
6:     return CP1
7: end procedure
```

$(\sim PM \ \& \ \sim HN \ \& \ VN)$

col	0	1	2	3	4	5	6	7
		a	b	a	b	a	a	c
	0	0	0	0	0	0	0	0
a	$\uparrow^0$ 1	$\nwarrow^0$ 0	1 ⋮ 1	1.. 0	1 ⋮ 1	1.. 0	1.. 0	1 ⋮ 1
b	$\uparrow^0$ 2	0 $\uparrow$ 1	$\nwarrow^0$ 0	1 ⋮ 1	1.. 0	1 ⋮ 1	1 ⋮ 1	1.. 1
b	$\uparrow^0$ 3	0 $\uparrow$ 2	0 $\uparrow$ 1	$\nwarrow^0$ 1	1 ⋮ 1	1.. 1	1 ⋮ 2	1 ⋮ 2
a	$\uparrow^0$ 4	0 $\uparrow$ 3	0 $\uparrow$ 2	$\nwarrow^0$ 1	1 ⋮ 2	1.. 1	1.. 1	1 ... 2
a	$\uparrow^0$ 5	0 $\uparrow$ 4	0 $\uparrow$ 3	0 $\uparrow$ 2	$\nwarrow^0$ 2	1 ⋮ 2	1.. 1	1.. 2

Neighbourhood Sizes:



$|P| = 16$ and $|\Sigma| = 2$(left). $|P| = 6$ and $|\Sigma| = 16$(right).

Errors	1	2	3	4	5	6	7	8
$CU - time(ms)$	1.31	1.59	3.24	7.95	13.15	11.71	7.15	4.21
$SCU - time(ms)$	1.28	1.54	2.38	3.28	3.59	3.27	2.98	2.41

Average time of Myers algorithm for $|P| = 16$ and $|\Sigma| = 2$

Bit-parallel and increased bit-parallel algorithms in milliseconds.

	$\|\Sigma\| = 2$		$\|\Sigma\| = 4$	
	$k = 2$	$k = 4$	$k = 2$	$k = 4$
CU_k	0.036	0.013	1.038	20.459
SCU_k-WHILE	0.013	0.005	0.378	0.356
SCU_k-NFA	0.012	0.008	0.293	0.566
SCU_k-CARRY	0.012	0.004	0.297	0.312
SCU_k-INC-WHILE	0.011	0.004	0.254	0.225
SCU_k-INC-NFA	0.009	0.006	0.132	0.249
SCU_k-INC-CARRY	0.009	0.003	0.125	0.142

Questions ?

Acknowledgements:

Eugene Myers, prototype.

Gonzalo Navarro and Heikki Hyyrö, remarks.

The Portuguese Science and Technology Foundation.

The Orient Foundation.

Thanks