An Efficient Algorithm for Generating Super Condensed Neighborhoods

Luís M. S. Russo**[1] and Arlindo L. Oliveira[2]

[1] IST / INESC-ID, R. Alves Redol 9, 1000 LISBOA, PORTUGAL
aml@inesc-id.pt,
[2] lsr@algos.inesc-id.pt

Abstract. Indexing methods for the approximate string matching problem spend a considerable effort generating condensed neighborhoods. Here, we point out that condensed neighborhoods are not a minimal representation of a pattern neighborhood. We show that we can restrict our attention to super condensed neighborhoods which are minimal. We then present an algorithm for generating Super Condensed Neighborhoods. The algorithm runs in $O(m\lceil m/w\rceil s)$, where m is the pattern size, s is the size of the super condensed neighborhood and w the size of the processor word. Previous algorithms took $O(m\lceil m/w\rceil c)$ time, where c is the size of the condensed neighborhood. We further improve this algorithm by using Bit-Parallelism and Increased Bit-Parallelism techniques. Our experimental results show that the resulting algorithm is very fast.

1 Introduction

Approximate string matching is useful in areas of computer science as text searching, pattern recognition, signal processing and computational biology. The problem is to retrieve all segments, of a large text string whose *edit distance* to a shorter pattern string is at most k. If the text is large enough, an efficient algorithm must preprocess the text. This approach has been actively researched in recent years [1, 3, 8, 10, 14, 16, 17]. Hybrid algorithms that divide their time into a *neighborhood generation* phase and a *filtration* phase are the current state of the art.

In this paper we focus our attention on improving the *neighborhood generation* phase of such algorithms.

2 Basic Concepts and Notation

2.1 Strings

Definition 1. *A string is a finite sequence of symbols taken from a finite alphabet Σ. The empty string is denoted by ϵ. The size of a string S is denoted by $|S|$.*

** Supported by the Portuguese Science and Technology Foundation through program POCTI and project POSI/EEI/10204/2001 and Project BIOGRID POSI/SRI/47778/2002

By $S[i]$ we denote the symbol at position i of S and by $S[i..j]$ the substring from position i to position j or ϵ if $i > j$. Also we denote by $S\langle i\rangle$ the point[3] in between letters $S[i-1]$ and $S[i]$. $S\langle 0\rangle$ represents the first point and $S\langle i-1..j\rangle$ denotes $S[i..j]$.

2.2 Computing Edit Distance

Definition 2. *The* edit *or* Levenshtein *distance between two strings* ed(S, S') *is the smallest number of edit operations that transform S into S'. We consider as operations insertions (I), deletions (D) and substitutions (S).*

For example: $ed(abcd, bedf) = 3$

```
D S I
abcd
 bedf
```

The edit distance between strings S and S' is computed by filling up a dynamic programming table $D[i,j] = ed(S\langle 0..i\rangle, S'\langle 0..j\rangle)$, constructed as follows:

$$D[i,0] = i, \qquad D[0,j] = j$$
$$D[i+1,j+1] = D[i,j], \text{ if } S[i+1] = S'[j+1]$$
$$1 + \min\{D[i+1,j], D[i,j+1], D[i,j]\}, \text{otherwise}$$

Table 1 is an example of the dynamic programming table D. According to the definition,
$ed(abbaa, ababaac) = ed(S, S') = D[|S|, |S'|] = D[5,7] = 2$.

Table 1. Table $D[i,j]$ for *abbaa* and *ababaac*.

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**

2.3 Finding Approximate Matches

For the purpose of finding matches, a useful variation of this table is table $D'[i,j] = \min_{0 \le l \le j}\{ed(S\langle 0..i\rangle, S'\langle l \ .. \ j\rangle)\}$, computed as table D but setting $D[0,j] = 0$, as shown in table 2.

[3] The notion of point is superfluous but it helps in the definition of a simple and coherent notation.

Table 2. (Left)Table $D'[i,j]$ for *abbaa* and *ababaac*. (Section 3) Improper canonical paths are indicated by arrows, (Section 4) improper cell bits are indicated on trace-backs. (Right) Binary representation of column 1.

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

According to the definition, line $D'[|S|, j]$ stores the smallest edit distance between S and a substring of S' starting at some point l and ending at j. Suppose we want to find all occurrences of *abbaa* in *ababaac* with at most one error. By looking at row $D'[5, j]$ we find out that such occurrences can end only in point 6. In particular there are two such occurrences *ababaa* and *abaa*.

Definition 3. *A cell in D or D' is active iff its value is smaller than k.*

Take $k = 1$ for our example. In tables 1 and 2 inactive cells are shaded.
A complete up to date survey on this problem has been presented by Navarro [12].

3 Indexed Approximate Pattern Matching

3.1 Overview

If we wish to find the occurrences of P in T in sub-linear time, i.e. $O(|T|^{\alpha})$ for $\alpha < 1$, we can use an index structure for T. Suffix arrays [13] and q-grams have been proposed in the literature [8, 10].

These algorithms are hybrid in the sense that they find a tradeoff between neighborhood generation and filtration techniques.

3.2 Neighborhood Generation

A first and simple-minded approach to the problem consists in generating all the words at distance k from P and looking them up in the index.

Definition 4. *The* k-neighborhood *of* S *is* $U_k(S) = \{S' \in \Sigma^* : ed(S, S') \leq k\}$

Since $U_k(S)$ turns out to be quite large ($|U_k(S)| = O(|S|^k|\Sigma|^k)$) [15], we restrict our attention to the *condensed k-neighborhood*. .

Definition 5. *The* condensed k-neighborhood *of* S, $CU_k(S)$ *is the largest subset of* $U_k(S)$ *whose elements* S' *verify the following property: if* S'' *is a proper prefix of* S' *then* $ed(S, S'') > k$.

Algorithm 1 generates $CU_k(P)$ [2]. [4]

Algorithm 1 Condensed Neighborhood Generator Algorithm

```
1: procedure SEARCH(Search State s, Current String v)
2:     if IS_MATCH_STATE(s) then
3:         REPORT(v)
4:     else if EXTENDS_TO_MATCH_STATE(s) then
5:         for z ∈ Σ do
6:             s' ← UPDATE(s, z)
7:             SEARCH(s', v.z)
8:         end for
9:     end if
10: end procedure
11: SEARCH(⟨0, 1, ..., |P|⟩, ϵ)
```

The search state (s) is a dynamic programming column of D associated to P. The IS_MATCH_STATE predicate checks whether the last cell is active. The EXTENDS_TO_MATCH_STATE predicate checks whether there are active cells in s. The UPDATE procedure computes the dynamic programming column that results from applying a to s.

For example, if s is column 5 of table 1, then the IS_MATCH_STATE predicate returns false, since cell $D[5, 5]$ is inactive. The EXTENDS_TO_MATCH_STATE, on the other hand, returns true, since cell $D[4, 5]$ is active. The UPDATE procedure computes column 6 from column 5 and a. When s is column 6, the IS_MATCH_STATE evaluates to true and the algorithm reports *ababaa* as being at distance 1 from *abbaa*. This way column 7 never gets evaluated. Let us skip line 5 for $z = b$. If $z = c$ and s is column 5 then the UPDATE procedure returns $\langle 6, 5, 4, 3, 2, 2\rangle$. In this case both the IS_MATCH_STATE and the EXTENDS_TO_MATCH_STATE predicates fail and the search backtracks.

The reason why the *condensed neighborhood* is important is that it represents the *k-neighborhood*.

Lemma 1. *If* $S \in U_k(S')$ *then some prefix of* S *is in* $CU_k(S')$.

[4] We can shortcut the generate and search cycle by running algorithm 1 on the index structure. For example in the suffix tree this can be done by using a tree node instead of v.

We can generalize the idea and think of representing U_k by substrings instead of only by prefixes. This leads to the notion of *super condensed neighborhood*.

Definition 6. *The* super condensed k-neighborhood *of S, $SCU_k(S)$ is the largest subset of $U_k(S)$ whose elements S' verify the following property:*
if S'' is a proper substring of S' then $ed(S, S'') > k$.

The *super condensed neighborhood* represents the *k-neighborhood* as follows:

Lemma 2. *If $S \in U_k(S')$ then some substring of S is in $SCU_k(S')$.*

In our example *ababaa* and *abaa* are in the *condensed neighborhood* of *abbaa*, but only *abaa* is in the *super condensed neighborhood*.

It is easy to see that the *Super Condensed k-neighborhood* is minimal, since any subset of $U_k(P)$ that represents it (as in lemma 2) must contain $SCU_k(P)$, for a word in $SCU_k(P)$ can only be represented by itself.

Definition 7. *A traceback is a pointer from cell $D'[i, j]$ to a predecessor neighbor cell, given by the following conditions:*

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* $D'[1, j+1] = 1 + D'[i, j]$

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

A canonical path is a path in D' made of canonical tracebacks. We refer to a canonical path as improper if it ends in $D[0, 0]$ (see table 2). The idea behind canonical paths is that they always show the rightmost position of a minimal match between S and a substring of S'.

Definition 8. *A cell $D'[i, j]$ is improper iff its canonical path is improper.*

The denomination improper is motivated by the following lemma.

Lemma 3. *If $D'[i, j]$ is an improper cell then $D[i, j] = D'[i, j]$.*

This is a a direct consequence from the observation that improper cells start matching from point 0 just like the cells in D. In fact the converse of the lemma is also true.

Computing the *super condensed neighborhood* can also be done by algorithm 1 but we change our states to columns of D' and restrict our attention to improper active cells.

Observe that, in this version of the algorithm, the string *ababaa* is no longer reported. In fact it can be seen that in column 4 of table 2 there are no active improper cells and hence neither column 5 nor column 6 get evaluated.

A theoretical time analysis shows that this new algorithm runs in $O(m^2|SCU_k(P)|)$, while the previous algorithm takes $O(m^2|CU_k(P)|)$.

In [10] Myers proved that $|CU_k(P)| = O(|P|^{pow(|P|/k)})$, where:
$pow(\alpha) = \log_{|\Sigma|} \frac{(\alpha^{-1}+\sqrt{1+\alpha^{-2}})+1}{(\alpha^{-1}+\sqrt{1+\alpha^{-2}})-1} + \alpha \log_{|\Sigma|}(\alpha^{-1} + \sqrt{1+\alpha^{-2}}) + \alpha$

It is hard to improve on this bound for the *super condensed neighborhood* so $|SCU_k(P)| = O(|P|^{pow(|P|/k)})$. However there is a clear practical improvement, as we show in the results section.

4 Bit Parallel Implementation

Myers presented a way to parallelize the computation of D and D' [11] that reduces the complexity of computing a dynamic programming table to $O(m\lceil m/w \rceil)$ were w is the size of the computer word. In our aplication the $\lceil m/w \rceil$ tipicaly takes the value 1 since $m = \Theta(\log_\sigma n)$ for hybrid algorithms.This leads to a complexity of $O(m)$.

Heikki Hyrrö presented a modification of Myers algorithm [5] that we will now describe and extend to solve our problem.

Ukkonen was the first to notice the following properties of D and D' [15]:

Diagonal Property $D[i+1, j+1] - D[i, j] = 0$ or 1
Vertical Adjacency Property $D[i+1, j] - D[i, j]$ = -1, 0 or 1
Horizontal Adjacency Property $D[i, j+1] - D[i, j]$ = -1, 0 or 1

The following bit-vectors can then be used to represent and compute columns of D'.

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 $z \in \Sigma$

The above bit-vectors are packed in computer words along i, i.e. by columns. In algorithm 2 we show how to compute a column of D'.

The procedure UPDATE of algorithm 2 is essentially the algorithm explained in the original work on bit parallelism [11, 5].

We will now show how the UPDATE_PROPER_CELLS procedure works. We define an improper cell vector $CP1$ to account for improper cells.

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

Table 2 shows an example of this computaion. Since we assume the bit vectors are of size m we can't store this information for the cells in row 0. This is not a big problem since apart, except for cell $D'[0, 0]$, all other $D'[0, j]$ cells are inactive. Special care must hence be taken to update the proper cells of column 1. This can be done by suffices changing the 1 in line 26 for VP &1.

The single purpose of line 23 is to discover whether the first improper cell in a column will became proper in the next column. For example, in table 2, the first

Algorithm 2 Bit-Parallel Algorithm, bitwise operations in C-style.

```
1: procedure INITIALIZE(Pattern P)
2:     VP ← (1^m)_2
3:     VN ← (0^m)_2
4:     For z ∈ Σ Do PM_z ← (0^m)_2
5:     For 1 ≤ i ≤ m Do PM_{P[i]}| ← 2^{i−1}
6:     CP1 ← 0
7:     VAL_0 ← (10101010 ...)_2
8:     VAL_1 ← (01100110 ...)_2
9:     ⋮
10:    return VP, VN, CP1, VAL_0, ..., VAL_{⌈log m⌉−1}
11: end procedure
12: procedure UPDATE( Previous Column (VP, VN, CP1, VAL_0, ..., VAL_{⌈log m⌉−1}),
        Letter z)
13:    D0 ← (((PM_z & VP) + VP)^VP)|PM_z|VN
14:    HP ← VN| ∼ (D0|VP)
15:    HN ← VP & D0
16:    VAL_0, ..., VAL_{⌈log m⌉} ← CARRY_EFFECT( HP, HN, VAL_0, ..., VAL_{⌈log m⌉−1})
17:    VP ← (HN << 1)| ∼ (D0|(HP << 1))
18:    VN ← (HP << 1) & D0
19:    CP1 ← UPDATE_PROPER_CELLS(CP1, PM_z, HN, VN, VP)
20:    return VP, VN, CP1, VAL_0, ..., VAL_{⌈log m⌉−1}
21: end procedure
22: procedure UPDATE_PROPER_CELLS(CP1, PM, HN, VN, VP)
23:    CP1 ← ((PM|HN| ∼ VN) & ((CP1 << 1)|1))|CP1
24:    CP1| ← VP
25:    CP1 ← (CP1 >> 1)
26:    CP1 ← (CP1 + 1)^CP1
27:    return CP1
28: end procedure
29: procedure CARRY_EFFECT(HP, HN, VAL_0, ..., VAL_{⌈log m⌉−1})
30:    carry ← HP | HN
31:    VAL_0 ← carry^VAL_0
32:    carry & ← HN^VAL_0
33:    VAL_1 ← carry^VAL_1
34:    carry & ← HN^VAL_1
35:    ⋮
36:    VAL_{⌈log m⌉−1} ← carry^VAL_{⌈log m⌉−1}
37: end procedure
```

improper cell of column 3 is $D'[3,3]$. What line 1 does is to check whether the canonical traceback of $D'[3,4]$ is non-horizontal. An horizontal canonical traceback respects the condition $\sim PM\& \sim HN\&VN$, (see cells $D'[3,6]$, $D'[3,7]$, $D'[4,6]$ and $D'[4,7]$).

Line 24 of algorithm 2 adds the vertical dependences to the list of improper cells since, if a cell has a vertical canonical traceback to a proper cell, then it is also proper. By introducing vertical dependencies line 24 also activates some unnecessary bits. In order to determine which bits actually represent improper cells we shift $CP1$ (line 25) and send a carry through it (line 26). The carry stops in the last improper cell. Finally we clean up the unnecessary bits and restore the ones eliminated by the carry by doing a xor with the previous $CP1$ (line 26). The $\sim CP1$ provides a mask of improper cells.

Keeping track of which cells are active can be done in several ways.

WHILE Keeping a pointer to the lowest active cell and moving upwards.

NFA Using a bit parallel implementation of an NFA for approximate pattern matching [17] similar to [13].

CARRY Storing the values of D' in computer words.

The pointer solution is as far as we know the standard solution to this problem. If k is small the NFA solution becomes viable since it uses k state vectors, i.e. computer words.

The idea of the CARRY solution is to store the values of D' in an unorthodox way. Values are stored across computer words instead of in a single one. This solution requires $\lceil \log |P| \rceil$ computer words, the VAL vectors. We define $VAL_k[i,j]$ as the $(k+1)$ digit in the binary representation of $D'[i,j]$. For an example, see table 2.

Updating the VAL vectors is a matter of simulating the carry effect of the ALU. This is implemented in the CARRY_EFFECT procedure. We propagate the addition and subtraction carries in the same word.

It is enough to identify active cells whose value is k. In our example this can be done by evaluating $\sim \mathrm{VAL}_0$ & VAL_1 & $\sim \mathrm{VAL}_2$.

A final improvement is to adapt the previous algorithm so that it works in an increased bit-parallelism fashion [7]. The idea of increased bit parallelism is to tile the computer word with more than one D' column and compute more than one D' column per instruction. In this approach the algorithm that is used is essentially the same but one must redefine the "+", ">>", "<<" operations to respect the column boundaries. The 1's must also be replaced accordingly.

Our approach was to move the instruction 6 of algorithm 1 to the exterior of the for cycle (instruction 5). In this case we had to make the UPDATE procedure update the column for all the letters of Σ. This was done by concatenating all the PM_z vectors into a single PM vector. We also had to copy the values of the D' column $|\Sigma|$ times into the computer word just before instruction 7. This was done by >> and | operations.

5 Experimental Results

We investigated the ratio between the average size of the condensed neighborhood versus the size of super condensed neighborhood (figure 1). This was done by generating the neighborhoods of 50 random patterns.

Next we verified that this ratio among averages translated into the time performance of the algorithm for patterns of size 16 where no partition occurs (table 3). This was done by comparing Myers implementation with ours. For this we tested 1000 random patterns on random text. These implementations do not yet resort to bit-parallelism.

These tests were performed in a 2.40 Ghz Intel Xeon processor, 4 GB RAM, Linux OS 2.4.20-28.7smp and gcc 2.96.

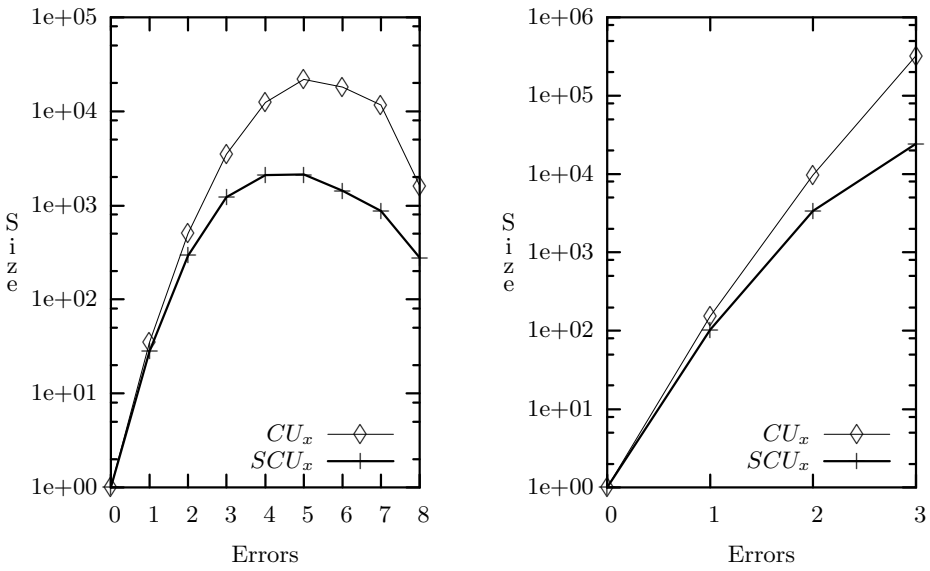

Fig. 1. Average size of the condensed neighborhood versus the super condensed neighborhood for $|P| = 16$ and $|\Sigma| = 2$ (left), $|P| = 6$ and $|\Sigma| = 16$ (right)

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

Table 3. Average time of Myers algorithm for $|P| = 16$ and $|\Sigma| = 2$

Finally we tested the bit-parallel version and the increased bit-parallel version for patterns of size 8 using a 800MHz PowerPC G3 processor with 512K level 2 cache 640MB SDRAM, Mac Os X 10.2.8 and gcc 3.3. The results are shown in Table 4. [5]

Table 4. 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

The first row shows the times needed to generate *Condensed Neighborhoods* while the next three rows show the times needed to generate *Super Condensed Neighborhoods* with our three alternatives. The three final rows show the times needed to generate *Super Condensed Neighborhoods* using increased bit parallelism.

Tables 4,3 and the graphics in Figure 1 show clearly the advantages of the techniques described in this work, both in terms of the neighborhood size and the speedup obtained by the bit parallel algorithms.

6 Conclusions and Future Work

In this work, we propose to address the problem of indexed approximate pattern matching by restricting our attention to super condensed neighborhoods. We have shown that this entailed a significant time improvement that was verified by experimental results.

Arguments of the same nature have been used before. In fact an early exploit of the Super Condensed Neighborhood idea was an heuristic used in [13]. The idea was that it is enough to find those matches to P that begin by matching one of its first $k + 1$ characters. The condition obviously guarantees that in column 1 there will be no improper active cells. A refinement of this idea has also been presented in [6]. Our algorithm generalizes all these cases.

More recently the authors of [8] presented the notion of artificial *prefix-stripped length-q* neighborhood, that modifies the condensed neighborhood in

[5] Please note that some of the results in table 4 are out of the sub-linear region of the index. This will be corrected by extending our prototype to larger $|\Sigma|$. The graphics in figure 1 show that this should produce even better results.

a way that adapts to Myers algorithm but that it not minimal. The notion of *super condensed neighborhood* had in fact been considered by the previous authors [6] and also in [4].

We proposed an algorithm for generating *super condensed neighborhoods* that adapts very well to a bit-parallel and increased bit-parallel approaches. To achieve this we proposed a new way of managing the active cells that clearly outperformed previous methods and adapted much better to increased bit-parallelism.

The results show that the use of *Super Condensed Neighborhood* speeds up the generation of the neighborhood by a significant factor that increases with the alphabet size and the error level.

Finally we would like to point out that this work is by no means finished. Our prototype must be extended to deal with larger $|\Sigma|$ and tested on the hybrid index [13]. The algorithm should also benefit greatly from an improvement like the one proposed in [9], specially since our binary representation is suitable for the necessary test predicates and this reduces the theoretical complexity truly to $O(ms)$. This would minimize the effect of the copy phase. Additionally our approach to increased bit-parallelism is still a bit naive. In particular we believe that there should be a way to squeeze more bits into the computer word, eventually by altering the copy phase.

Acknowledgments

We are grateful to Eugene Myers for providing us access to his prototype. We also thank Gonzalo Navarro and Heikki Hyyrö for their suggestions and remarks.

References

1. R. Baeza-Yates. Text retrieval: Theory and practice. volume I, pages 465–476. 12th IFIP World Computer Congress, Elsevier Science, 1992.
2. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, (35(10)):74–82, 1992.
3. A.L. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM95)*, LNCS 937, pages 41–54. Springer, 1995.
4. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1999.
5. H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
6. H. Hyyrö. *Practical Methods for Approximate String Matching.* PhD thesis, University of Tampere, 2003.
7. H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate string matching. In *Proc. 3rd Workshop on Efficient and Experimental Algorithms (WEA'04)*, LNCS 3059, pages 285–298, 2004.

[6] personal communication

8. H. Hyyrö and G. Navarro. A practical index for genome searching. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 341–349. Springer, 2003.
9. Heikki Hyyrö. An improvement and an extension on the hybrid index for approximate string matching. In *Proceedings of the 11th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 3246, pages 208–209. Springer, 2004.
10. E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica*, (12):345–374, 1994.
11. G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic programming. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM98)*, LNCS 1448, pages 1–13. Springer-Verlag, 1998.
12. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
13. G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.
14. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.
15. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, pages 132–137, 1985.
16. E. Ukkonen. Approximate string matching over suffix trees. volume 684 of *LNCS 2857*, pages 228–242. Procedings of the 4th Annual Symposium on Combinatorical Pattern Matching (CPM93), Springer, 1993.
17. S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, (35(10)):83–91, 1992.