Faster Generation of Super Condensed Neighbourhoods using Finite Automata

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. We present a new algorithm for generating super condensed neighbourhoods. Super condensed neighbourhoods have recently been presented as the minimal set of words that represent a pattern neighbourhood. These sets play an important role in the generation phase of hybrid algorithms for indexed approximate string matching. An existing algorithm for this purpose is based on a dynamic programming approach, implemented using bit-parallelism. In this work we present a bit-parallel algorithm based on automata which is faster, conceptually much simpler and uses less memory than the existing method.

1 Introduction and Related Work

Approximate string matching is an important subject in computer science, with applications in text searching, pattern recognition, signal processing and computational biology.

The problem consists in locating all occurrences of a given pattern string in a larger text string, assuming that the pattern can be distorted by errors. If the text string is long it may be infeasible to search it on-line, and we must resort to an index structure. This approach has been extensively investigated in recent years [1, 5, 6, 9, 13, 16, 17].

The state of the art algorithms are hybrid, and divide their time into a *neighbourhood generation* phase and a *filtration* phase [12, 9].

This paper is organised as follows: in section 2 we define the basic notation and the concept of strings and edit distance. In section 3 we present a high level description of hybrid algorithms for indexed approximate pattern matching. In section 4 we present previous work on the neighbourhood generation phase of hybrid algorithms. In section 5 we present our contribution, a new algorithm for generating Super Condensed Neighbourhoods. In section 6 we describe the bit-parallel implementation of our algorithm and present a complexity analysis. Section 7 presents the experimental results obtained with our implementation. Finally, section 8 presents the conclusions and possible future developments.

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

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 $|S|$.*

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 $i > j$. Additionally 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:

```
D S I
abcd
 bedf
```

$ed(abcd, bedf) = 3$

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:

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

The dynamic programming approach to the problem is the oldest approach to computing the edit distance. As such it has been heavily researched and many such algorithms have been presented, surveyed in [11].

One particularly important contribution was Myers proposal of an algorithm to compute the edit distance in a bit-parallel way [10]. The previous algorithm for computing Super Condensed Neighbourhoods [14] is based on this algorithm.

A different approach for the computation of the edit distance is to use a non-deterministic automaton(NFA). We can use a NFA to recognise all the words that are at *edit* distance k from another string P, denoted N_P^k. Figure 1 shows an automaton that recognises words that are at distance at most one from *abbaa*. It should be clear that the word *ababaa* is recognised by the automaton since $ed(abbaa, ababaa) = 1$.

To find every match of a string P in another string T we can build an automaton for P and restart it with every letter of T. This is equivalent to adding a loop labelled with all the character in Σ to the initial state. We shall denote this new automaton by $N_P'^k$.

[3] The notion of point is superfluous but useful since it provides a natural way to introduce automata states.

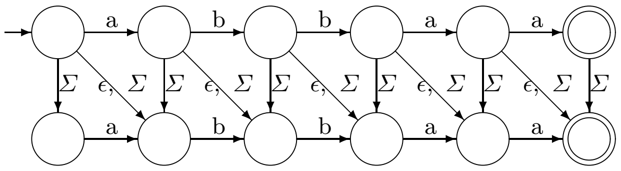

Fig. 1. Automaton for *abbaa* with at most one error.

3 Indexed Approximate Pattern Matching

If we wish to find the occurrences of P in T in sub-linear time, with a $O(|T|^{\alpha})$ complexity for $\alpha < 1$, we need to use an index structure for T. Suffix arrays [12] and q-grams have been proposed in the literature [6, 9]. An important class of algorithms for this problem are hybrid in the sense that they find a trade-off between neighbourhood generation and filtration techniques.

3.1 Neighbourhood 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 T. The set of generated words is the *k-neighbourhood* of P.

Definition 3. *The* k-neighbourhood *of S is $U_k(S) = \{S' \in \Sigma^* : ed(S, S') \leq k\}$*

Let us denote the language recognised by the automaton N_P^k as $L(N_P^k)$. It should be clear that $U_k(P) = L(N_P^k)$. Hence computing $U_k(P)$ is achieved by computing $L(N_P^k)$, this can be done by performing a DFS search in Σ^* that halts whenever all the states of N_P^K became inactive.

3.2 Filtration Techniques

The classic idea of filtration is to eliminate text areas, by guaranteeing that there is no match at a given point, using techniques less expensive than dynamic programming. Since this approach has the obvious drawback that it cannot exclude all such areas, the remaining points have to be inspected with other methods.

In the indexed version of the problem, filtration can be used to reduce the size of neighbourhoods, hence speeding up the algorithm.

The most common filtration technique splits the pattern according to the following lemma:

Lemma 1. *If* ed(S,S')$\leq k$ *and* $S = S_1 x_1 S_2 x_2 \ldots S_{l-1} x_{l-1} S_l$ *then* S_h *appears in* S' *with at most* $\lfloor k/l \rfloor$ *errors for some* h.

This lemma was presented by Navarro and Baeza-Yates [12]. Myers had also presented a similar proposition [9].

4 Neighbourhood Analysis

The *k-neighbourhood*, $U_k(S)$, turns out to be quite large. In fact $|U_k(S)| = O(|S|^k|\Sigma|^k)$ [15] and therefore we restrict our attention to the *condensed k-neighbourhood* [9, 12].

Definition 4. *The* condensed k-neighbourhood *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 $ed(S, S'') > k$.*

The generation of $CU_k(P)$ can be done using automaton N_P^k by testing the words of Σ^* obtained from a DFS traversal of the lexicographic tree. The search backtracks whenever all states of N_P^k became inactive or a final state becomes active.

The second criterion guarantees that no generated word is a prefix of another one.

Algorithm 1 generates $CU_k(P)$ by performing a controlled DFS that does not extend words of $L(N_P^k)$ found in the process [2]. [4]

Algorithm 1 Condensed Neighbourhood Generator Algorithm

```
1: procedure SEARCH(Search Point $p$, Current String $v$)
2:     if IS_MATCH_POINT($p$) then
3:         REPORT($v$)
4:     else if EXTENDS_TO_MATCH_POINT($p$) then
5:         for $z \in \Sigma$ do
6:             $p' \leftarrow$ UPDATE($p, z$)
7:             SEARCH($p', v.z$)
8:         end for
9:     end if
10: end procedure
11: SEARCH($\langle 0, 1, \ldots, |P| \rangle, \epsilon$)
```

The search point p is a set of active states of N_P^k. The IS_MATCH_POINT predicate checks whether some state of p is a final state. The EXTENDS_TO_MATCH_POINT predicate checks whether p is non-empty. The UPDATE procedure updates the active states of p by processing character z with N_P.

It has been noted [14] that the *condensed neighbourhood* still contains some words that can be discarded without missing any matches.

[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.

Definition 5. *The* super condensed k-neighbourhood *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 $ed(S, S'') > k$.*

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

Figure 2 shows an example of the *1-neighbourhood*, the *1-condensed neighbourhood* and the *1-super condensed neighbourhood* of *abbaa*. Observe that $SCU_k(P) \subseteq CU_k(P) \subseteq U_k(P)$.

SCU_1: abaa, abba, abbba, bbaa
CU_1: aabaa, ababaa, babbaa, bbbaa, aabbaa
U_1: abaaa, abbaa, abbaaa, abbaab, abbab, abbaba, abbbaa

Fig. 2. Figure representing the one-neighbourhoods of *abbaa*.

The *Super Condensed k-neighbourhood* is minimal in the sense that we can't have a set with a smaller number of words that can be used in the search without missing matches [14].

H. Hyyrö and G. Navarro [6] presented the notion of artificial *prefix-stripped length-q* neighbourhood, that is smaller than the condensed neighbourhood but it is not minimal.

5 Computing Super Condensed Neighbourhoods using Finite Automata

We now present the main contribution of this paper, a new approach to compute *super condensed neighbourhoods.*

In order to compute the *super condensed neighbourhood* we define a new automaton. Consider the automaton N''^k_P that results from N^k_P by adding a new initial state with a loop labelled by all the characters of Σ linked to the old initial state by a transition also labelled by all the characters of Σ. An example of N''^k_P is shown in fig. 3. The language recognised by N''^k_P consists of all the strings that have a **proper** suffix S'' such that $ed(P, S'') \leq k$.

The set $L(N^k_P)\backslash L(N''^k_P)$ is <u>not</u> a *super condensed neighbourhood* by the following two reasons:

prefixes Some words might still be prefixes of other words. For example both *abaa* and *abaaa* belong to $L(N^k_{abbaa})\backslash L(N''^k_{abbaa})$. This can be solved when performing the DFS traversal of the lexicographic tree, as before.

substrings The definition of $L(N^k_P)\backslash L(N''^k_P)$ will yield the subset of $L(N^k_P)$ such that no proper <u>suffix</u> is at distance at most k from P. But this is not what we want, since we desire a subset of $L(N^k_P)$ that does not contain a string and a proper substring of that string. In order to enforce this requirement we must stop the DFS search whenever a final state of $N_P''^k$ is reached.

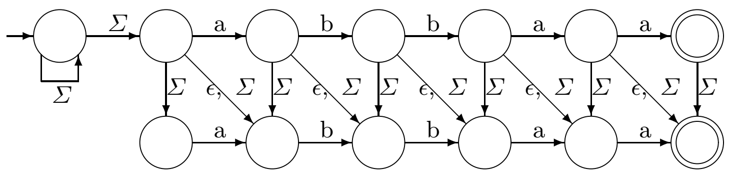

Fig. 3. Automaton $N_P''^k$ for *abbaa* that matches every proper suffix.

A point p in the DFS search of the lexicographical tree now corresponds to two sets of states, one for N_P^k and one for $N_P''^k$. The IS_MATCH_POINT predicate checks that no active state of $N_P''^k$ is final and that there is one active state of N_P^k that is final. The EXTENDS_TO_MATCH_POINT checks that no active state of $N_P''^k$ is final and that there is one active state of N_P^k that is inactive in $N_P''^k$. The UPDATE procedure updates both automata using letter z.

Observe that, in this version of the algorithm, the string *ababaa* is no longer reported. In fact the DFS search backtracks after having reached *abab*. After reading *abab* the only active state of N_P^k is the one corresponding to *abb* on the second column, since $ed(abab, abb) = 1$. This state is also active in $N_P''^k$ since $ed(ab, abb) = 1$ and *ab* is a proper suffix of *abab*. Interestingly, the dynamic programming DFS for this same example backtracked only after reaching *ababa*. Clearly, the dynamic programming algorithm could be improved to backtrack sooner but it is conceptually much simpler to use the automata approach.

It was shown by Myers [9] that $|CU_k(P)| = O(|P|^{pow(|P|/k)})$, where:
$pow(\alpha) = \log_{|\Sigma|} \frac{(\alpha^{-1}+\sqrt{1+\alpha^{-2}})+1}{(\alpha^{-1}+\sqrt{1+\alpha^{-2}})-1} + \alpha \log_{|\Sigma|}(\alpha^{-1} + \sqrt{1+\alpha^{-2}}) + \alpha$

We establish no new worst case bound for the size of the *super condensed neighbourhood* so $|SCU_k(P)| = O(|P|^{pow(|P|/k)})$. However ours results do show a practical improvement in speed.

6 Bit Parallel Implementation and Complexity Analysis

We implemented N_P^k and $N_P''^k$ by using bit-parallelism techniques that have been proposed for $N_P'^k$ [17, 4].

Algorithm 2 describes the details of implementation of the necessary predicates.

The F_i computer words store the N_P^k for row i. The S_i computer words store the $N_P''^k$automata states for row i. The $B[z]$ computer words stores the bit mask of the positions of the letter z in P.

Our implementation of the Wu and Manber algorithm stores the first column of the automata. Furthermore for automata $N_P''^k$ we don't need to store the artificial state that was inserted, since it is sufficient to initialise the S_i state vectors to zero.

Algorithm 2 Bit-Parallel of the Algorithm. N_P^k represented by F_i and $N_P''^k$ by S_i. Bitwise operations in C-style.

```
 1: procedure IS_MATCH_POINT(Search Point $F_0, \ldots, F_k, S_0, \ldots, S_k$)
 2:     return $F_k \&\& !S_k$
 3: end procedure
 4: procedure EXTENDS_TO_MATCH_POINT(Search Point $F_0, \ldots, F_k, S_0, \ldots, S_k$)
 5:     return $((F_0 \& \tilde{} S_0) | \ldots | (F_k \& \tilde{} S_k)) \&\& !S_k$
 6: end procedure
 7: procedure UPDATE(Search Point $F_0, \ldots, F_k, S_0, \ldots, S_k$, letter $z$)
 8:     $F_0' \leftarrow (F_0 << 1) \& B[z]$
 9:     $S_0' \leftarrow ((S_0 << 1)|1) \& B[z]$
10:     for $i \leftarrow 0, k$ do
11:         $F_{i+1}' \leftarrow ((F_{i+1} << 1) \& B[z]) | F_i | (F_i << 1) | (F_i' << 1)$
12:         $S_{i+1}' \leftarrow ((S_{i+1} << 1) \& B[z]) | S_i | (S_i << 1) | (S_i' << 1)$
13:     end for
14:     return $F_0', \ldots, F_k', S_0', \ldots, S_k'$
15: end procedure
```

Since the UPDATE and EXTENDS_TO_MATCH_POINT procedures run in $O(k\lceil |P|/w \rceil)$ the final algorithm takes $O(k\lceil |P|/w \rceil \ |P| \ s)$ where $s = |SCU_k(P)| = O(|P|^{pow(|P|/k)})$ and w is the size of the computer word. This is a conservative bound since it is easy to modify the algorithm so that it runs in $O((k\lceil |P|/w \rceil + |P|)s)$. This is achieved by using the KMP failure links and was first presented by Myers [9]. Recently Heikki Hyyrö presented way of achieving the same result in a sequential way that is relevant for bit-parallel algorithms [8].

We also implemented a version based on Navarro and Baeza-Yates [3] variation of the NFA. The procedures are implemented in a similar way and the resulting algorithm runs in $O(\lceil k(|P| - k)/w \rceil \ |P| \ s)$. We improved this to $O((\lceil k(|P| - k)/w \rceil \ + |P|) \ s)$ using an approach similar to the one followed by Myers but found no time difference in practice. Usually $O(\lceil k(|P| - k)/w \rceil)$ is approximately constant for small patterns, which is the case for hybrid algorithms. We usually split the pattern into pieces of size $\Theta(\log_\sigma |T|)$.

In previous work [14], we reported a complexity of $O(|P| \lceil |P|/w \rceil s)$ which was too pessimistic, since it did not take into account the possible reduction in complexity that is possible to achieve by applying the method based on the KMP failure links of Myers [9].

Once again the Baeza-Yates and Navarro algorithm usually doesn't store the states below the first diagonal including the diagonal. We don't need to keep track of the states below the diagonal but we do need to keep track of the diagonal [5].

[5] Actually this could be reduced but the gains would be practically none.

7 Experimental Results

We tested our approach by analysing its impact in the hybrid index [12]. Since we are only interested in the neighbourhood generation phase we set the j option of the index to 1, preventing the pattern from getting split.

Tests were run in a 800MHz Power PC G3 processor with 512K level 2 cache 640MB SDRAM, Mac Os X 10.2.8 and gcc 3.3.

Our implementation was based on the original implementation of Navarro and Baeza-Yates. The NFA is also based on the variation presented by Navarro and Baeza-Yates [3].

For each $(|P|, k)$ combination we tested 100 patterns randomly selected from the text and computed the average time to search for those patterns. The patterns were taken with sizes 10, 15 and 20. We used two source texts, an English text [18], that consists of cleaned up newsgroups text and a DNA file of 5.6 Mb, from the *S. cerevisiae* (baker's yeast) genome. Results are shown in figure 4.

We generated random patterns of size 8 for alphabets of size 2 and 4. The average size results are shown in table 1 and the time to generate the neighbourhoods blindly without the text are shown in table 2

Table 1. Average size of CU_k vs SCU_k.

	$\|\Sigma\| = 2$		$\|\Sigma\| = 4$	
	$k = 2$	$k = 4$	$k = 2$	$k = 4$
CU_k	67	42	810	21430
SCU_k	22	14	320	591

Table 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-CARRY	0.012	0.004	0.297	0.312
SCU_k-INC-CARRY	0.009	0.003	0.125	0.142
SCU_k-NFA	0.0043	0.0026	0.1048	0.171

The first row shows the times needed to generate *Condensed Neighbourhoods*, while the next three rows show the times needed to generate *Super Condensed Neighbourhoods*. The second and third rows were obtained using the algorithms based on dynamic programming while the last line corresponds to the algorithm

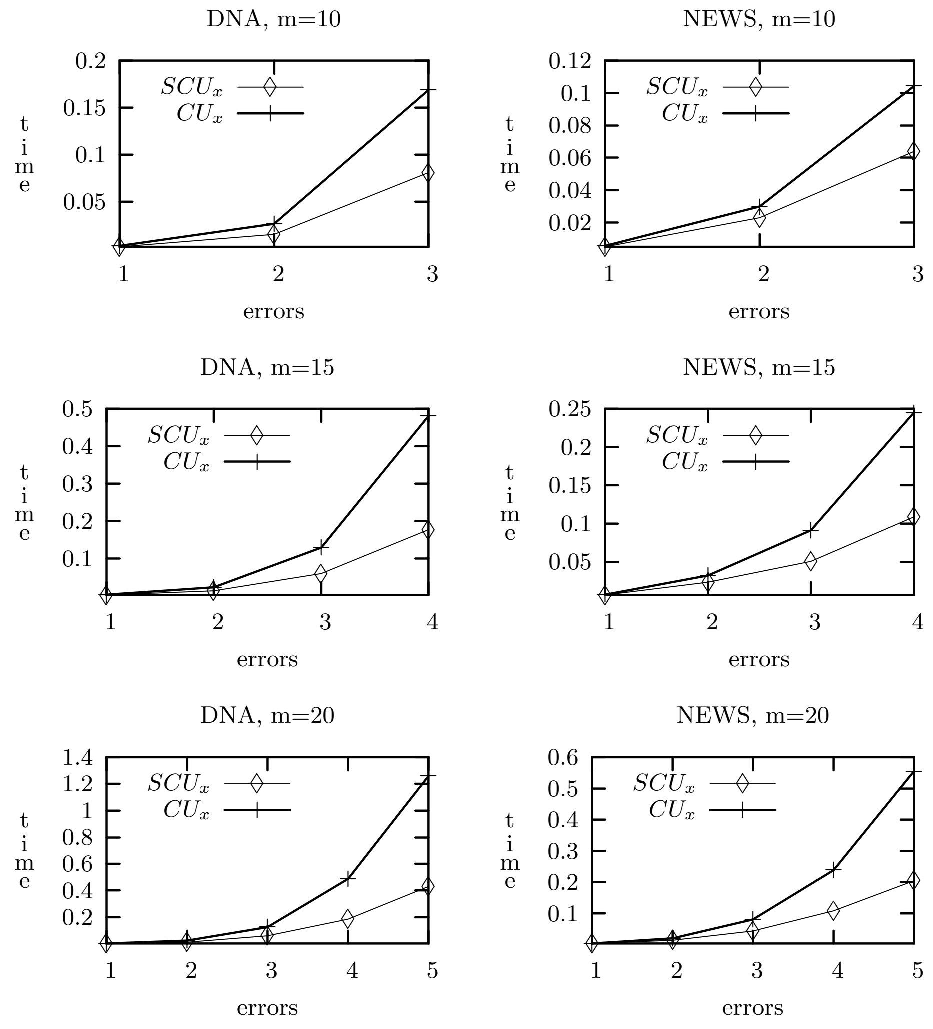

Fig. 4. The left column shows the average time (in seconds) for searching in DNA data and the right column shows the average time for searching in the Newsgroups data. The pattern size is indicated by m.

described in this article implemented using Wu and Manber bit-parallel algorithm.

8 Conclusions

In this work we proposed a new algorithm for the generation of *super condensed neighbourhoods* and used it to show the practical gains of using *super condensed neighbourhoods* instead of *condensed neighbourhoods*.

We also compared the algorithms we presented with the ones that existed based on dynamic programming. As expected, results favour this new approach. However it was pointed out by Heikki Hyyrö [7] that when generating the neighbourhoods the main time factor corresponds to accessing the index in memory and not in how we compute the *edit* distance. This means that using NFA's or dynamic programming makes little practical difference. This is also an argument in favour of this algorithm since it is conceptually much simpler than the one based on dynamic programming.

Acknowledgements

We thank Gonzalo Navarro and Baeza-Yates for kindly lending us their implementation of the hybrid index. We also thank Gonzalo Navarro and Heikki Hyyrö for their suggestions and remarks. We thank Gene Myers for suggestions, corrections and remarks.

References

1. Ricardo A. Baeza-Yates. Text-retrieval: Theory and practice. In Jan van Leeuwen, editor, *IFIP Congress (1)*, volume A-12 of *IFIP Transactions*, pages 465–476. North-Holland, 1992.
2. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
3. Ricardo A. Baeza-Yates and Gonzalo Navarro. A faster algorithm for approximate string matching. In Daniel S. Hirschberg and Eugene W. Myers, editors, *CPM*, volume 1075 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 1996.
4. Ricardo A. Baeza-Yates and Gonzalo Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. Archie L. Cobbs. Fast approximate matching using suffix trees. In Zvi Galil and Esko Ukkonen, editors, *CPM*, volume 937 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1995.
6. 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.
7. Heikki Hyyrö. *Practical Methods for Approximate String Matching*. PhD thesis, Faculty of Information of the University of Tampere, 2003.

8. 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.
9. E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica*, (12):345–374, 1994.
10. Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In Martin Farach-Colton, editor, *CPM*, volume 1448 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1998.
11. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
12. G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.
13. 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.
14. Luís M. S. Russo and Arlindo L. Oliveira. An efficient algorithm for generating super condensed neighborhoods. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *CPM*, volume 3537 of *Lecture Notes in Computer Science*, pages 104–115. Springer, 2005.
15. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, pages 132–137, 1985.
16. Esko Ukkonen. Approximate string-matching over suffix trees. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *CPM*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1993.
17. Sun Wu and Udi Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
18. file 20ng-train-all-terms from `http://www.gia.ist.utl.pt/~acardoso/datasets`