Efficient Generation of Super Condensed Neighborhoods

Luís M. S. Russo [1], Arlindo L. Oliveira

INESC-ID/IST, R. Alves Redol 9, 1000 LISBOA, PORTUGAL

Abstract

Indexing methods for the approximate string matching problem spend a considerable effort generating condensed neighborhoods. Condensed neighborhoods, however, are not a minimal representation of a pattern neighborhood. Super condensed neighborhoods, proposed in this work, are smaller, provably minimal and can be used to locate approximate matches that can later be extended by on-line search. We present an algorithm for generating Super Condensed Neighborhoods. The algorithm can be implemented either by using dynamic programming or non-deterministic automata. The complexity is $O(ms)$ for the first case and $O(kms)$ for the second, where m is the pattern size, s is the size of the super condensed neighborhood and k the number of errors. Previous algorithms depended on the size of the condensed neighborhood instead. These algorithms can be implemented using Bit-Parallelism and Increased Bit-Parallelism techniques. Our experimental results show that the resulting algorithms are fast and achieve significant speedups, when compared with the existing proposals that use condensed neighborhoods.

Key words: Suffix trees, edit distance, approximate string matching

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

Email address: `lsr@algos.inesc-id.pt` (Luís M. S. Russo).

[1] Supported by FCT, grant SFRH/BD/12101/2003 in project POCI 2010, Project BIOGRID POSI/SRI/47778/2002 and the Orient Foundation.

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 [1–3,5,10,11,15,19,20].

State of the art algorithms are hybrid, and divide their time into a *neighborhood generation* phase and a *filtration* phase [11,14].

The neighborhood generation phase is based on algorithms that traverse a suffix tree using dynamic programming to find the pattern matches [1,5,19]. For hybrid algorithms, however, the search space can be reduced. This is the idea of the Condensed neighborhood: a set of strings that compactly represent the whole neighborhood. In this work we propose an even more compact representative, the super condensed neighborhood.

The filtration phase uses the idea of filtration, a technique used for the on-line version of the problem. It consists in eliminating 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 neighborhoods, hence speeding up the algorithm. The most common filtration technique splits the pattern and later on tries to expand it around potential matches. The way the pattern is split determines the balancing point for the hybrid algorithm. Myers [11] and Baeza-Yates and Navarro [14] presented a detailed treatment of this subject. They also describe the limitations of the method, including the fact that above a given error level the complexity of the method becomes linear.

This paper is organized 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 and the notion of Super Condensed Neighborhood. In section 4 we present a high level description and complexity analysis of the algorithm for generating Super Condensed Neighborhoods. In section 5 we describe bit-parallel implementations of our algorithm. Section 6 presents the experimental results obtained with our implementations, and section 7 presents the conclusions.

2 Basic Concepts and Notation

We denote by ϵ the empty string; by $|S|$ the size of string S; by $S[i]$ the symbol at position i and by $S[i..j]$ the sub-string from position i to position j.

Definition 1 *The* edit *or* Levenshtein *distance,* $\mathrm{ed}(S, S')$*, between two strings*

Table 1
Table $D[i,j]$ for *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	$\boxed{2}$

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' can be computed by filling up a dynamic programming table $D[i,j] = ed(S[1..i], S'[1..j])$, constructed as follows:

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

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

Table 1 shows an example of the dynamic programming table D.

A different, yet related, approach for the computation of the edit distance is to use a non-deterministic automaton (NFA). We can use a NFA, denoted N_P^k, to recognize all the words that are within *edit* distance k from another string P. Figure 1 shows an automaton that recognizes words that are at distance at most one from *abbaa*, where Σ represents any symbol. It should be clear that the word *ababaa* is recognized by the automaton since $ed(abbaa, ababaa) = 1$. A comprehensive survey about these algorithms is available [13], and should be consulted for an in depth description.

Figure 2 shows the computation performed by automaton N_P^k when the input string is *ababaac*.

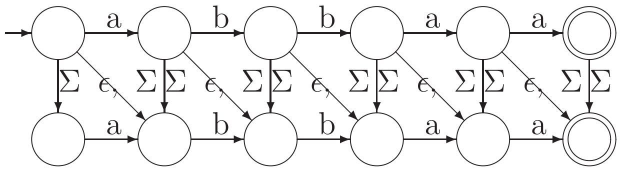

Fig. 1. Automaton N_P^k for *abbaa* with at most one error.

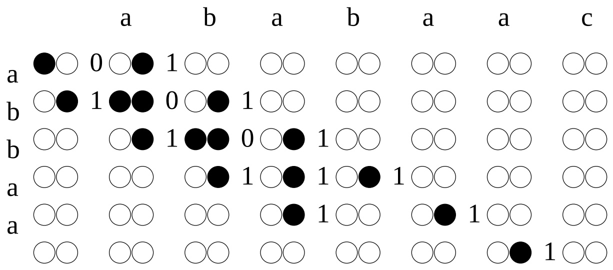

Fig. 2. Computation of *ababaac* using automaton N_P^k for *abbaa* with at most one error (flipped horizontally and rotated 90 degrees clockwise, active states marked in black).

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			
a	↑0	↖0	1 ⋮	⋱1	1 ⋮	⋱1	⋱1	1 ⋮			
1	1	0	1	0	1	0	0	1	0	0	0
b	↑0	0 ↑	↖0	1 ⋮	⋱1	1 ⋮	1 ⋮	⋱1			
2	2	1	0	1	0	1	1	1	0	0	1
b	↑0	0 ↑	0 ↑	↖0	1 ⋮	⋱1	1 ⋮	1 ⋮			
3	3	2	1	1	1	1	2	2	0	1	0
a	↑0	0 ↑	0 ↑	↖0	1 ⋮	⋱1	⋱1	1			
4	4	3	2	1	2	1	1	⋯ 2	0	1	1
a	↑0	0 ↑	0 ↑	0 ↑	↖0	1 ⋮	⋱1	⋱1			
5	$\boxed{5}$	$\boxed{4}$	$\boxed{3}$	$\boxed{2}$	$\boxed{2}$	$\boxed{2}$	$\boxed{1}$	$\boxed{2}$	1	0	0

Definition 2 *A cell in D is active iff its value is smaller or equal to k.*

Table 1 shows inactive cells shaded, when $k = 1$. Observe that the columns of N_P^k that correspond to inactive cells do not have any active state. (see figure 1).

A useful variation of table D is table $D'[i, j] = \min_{0 \leq l \leq j}\{ed(P[1..i], T[l\ ..\ j])\}$, computed as table D but setting $D[0, j] = 0$. (see table 2).

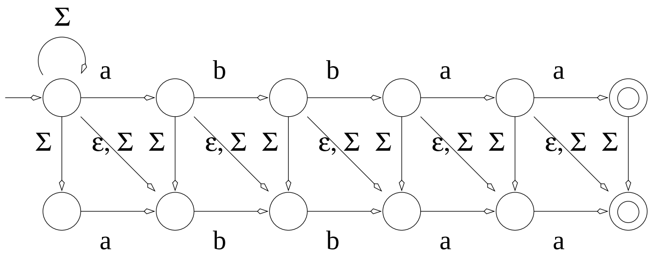

Fig. 3. Automaton N'^k_P for *abbaa* with at most one error.

According to the definition of D', the last line, $D'[m, j]$, stores the smallest edit distance between S and a sub-string of S' starting at some position l and ending at position 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 position 6. In particular, there are two such occurrences, *ababaa* and *abaa*. To compute the same result we can use an automata N'^k_P, built by adding a loop labeled with all the character in Σ to the initial state (see figure 3).

3 Indexed Approximate Pattern Matching

If we wish to find the approximate occurrences of P in T in sub-linear time (with $O(n^\alpha)$ complexity, for $\alpha < 1$) we need to use an index structure for T. Suffix arrays [14] and q-grams have been proposed in the literature [10,11]. An important class of algorithms for this problem are hybrid in the sense that they find a trade-off between neighborhood generation and filtration techniques.

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-neighborhood* of P.

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

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

The *k-neighborhood*, $U_k(S)$, turns out to be quite large. In fact $|U_k(S)| = O(|S|^k|\Sigma|^k)$ [18]. This motivates the notion of *condensed k-neighborhood* [11,14].

Definition 4 *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 $ed(S, S'') > k$.*

The *condensed k-neighborhood* $CU_k(P)$ can be generated by algorithm 1[2]. This algorithm can be used both with the dynamic programming method and the automata based approach [11,14].

Algorithm 1 Condensed Neighborhood 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 ∈ Σ do
 6:             p' ← UPDATE(p, z)
 7:             SEARCH(p', v.z)
 8:         end for
 9:     end if
10: end procedure
11: SEARCH(⟨0, 1, ..., m⟩, ε)
```

In the dynamic programming approach the search point (p) is a dynamic programming column of D associated to P. The IS_MATCH_POINT predicate checks whether the last cell is active. The EXTENDS_TO_MATCH_POINT predicate checks whether there are active cells in p. The UPDATE procedure computes the dynamic programming column that results from applying a to p.

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

In the automaton approach, 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.

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

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

SCU_1: abaa, abba, abbba, bbaa

CU_1: aabaa, ababaa, babbaa, bbbaa, aabbaa

U_1: abaaa, abbaa, abbaaa, abbaab, abbab, abbaba, abbbaa

Fig. 4. Figure representing the one-neighborhoods of *abbaa*.

Lemma 5 *If $S \in U_k(S')$ then some prefix of S is in $CU_k(S')$.*

We can generalize the idea and think of representing U_k by sub-strings instead of only by prefixes. This leads to the notion of *super condensed neighborhood* [16,17].

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 sub-string of S' then $ed(S, S'') > k$.*

The following lemma explains why the *super condensed neighborhood* represents the *k-neighborhood*.

Lemma 7 *If $S \in U_k(S')$ then some sub-string of S is in $SCU_k(S')$.*

Super condensed neighborhoods are, in fact, the smallest sets of strings that represent $U_k(S')$, because they are a minimal representation of $U_k(S')$. The following lemma explains this property.

Lemma 8 *If $C \subseteq U_k(S')$ and C represents $U_k(S')$ then $SCU_k(S') \subseteq C$.*

Proof This follows immediately from the fact that a word in $SCU_k(P)$ can only be represented by itself. Therefore, since C represents $U_k(S')$, it must contain $SCU_k(S')$. $\square$

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

Figure 4 shows an example of the *1-neighborhood*, the *1-condensed neighborhood* and the *1-super condensed neighborhood* of *abbaa*. Observe that $SCU_k(P) \subseteq CU_k(P) \subseteq U_k(P)$.

4 Computing Super Condensed Neighborhoods

We will now explain how to modify algorithm 1 to compute super condensed neighborhoods, using either dynamic programming or automata.

Definition 9 *A trace-back 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 trace-back for $D'[i,j]$ is the rightmost trace-back that $D'[i,j]$ has, i.e. vertical first, then diagonal and finally horizontal.

A canonical path is a path in D' made of canonical trace-backs. 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 starting position of a minimal match between S and a sub-string of S'.

Definition 10 *A cell $D'[i,j]$ is improper iff its canonical path is improper.*

The denomination improper is motivated by the following lemma.

Lemma 11 *If $D'[i,j]$ is an improper cell then $D[i,j] = D'[i,j]$.*

This is a direct consequence from the observation that improper cells start matching from the beginning of T 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 search point (p) to a column 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 need to be evaluated.

We can also compute the *super condensed neighborhoods* using automata. For this purpose, we define a new automaton, $N_P''^k$ that results from N_P^k by adding a new initial state with a loop labeled by all the characters of Σ linked to the old initial state by a transition also labeled by all the characters of Σ. An example of $N_P''^k$ is shown in fig. 5. The language recognized by $N_P''^k$ consists of all the strings that have a **proper** suffix S'' such that $ed(P, S'') \leq k$.

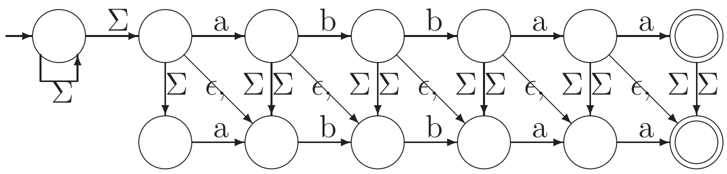

Fig. 5. Automaton $N_P''^k$ for *abbaa* that matches every proper suffix.

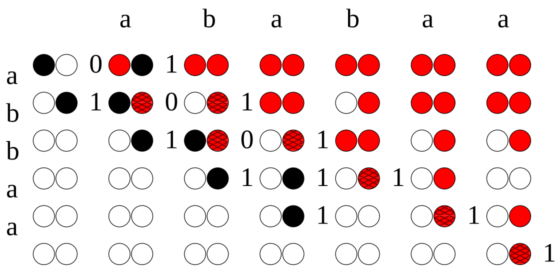

Fig. 6. Computation of *ababaac* using automaton N_P^k (black) and $N_P''^k$ (Grey).

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

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

sub-strings The definition of $L(N_P^k) \backslash L(N_P''^k)$ will yield the subset of $L(N_P^k)$ 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 $L(N_P^k)$ that does not contain a string and a proper <u>sub-string</u> 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.

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 the string *ababaa* is also not 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, because $ed(abab, abb) = 1$. This state is also active in $N_P''^k$ since $ed(ab, abb) = 1$ and *ab* is a proper suffix of *abab* (figure 6).

4.1 Complexity Analysis

The algorithm for generating $CU_k(P)$ runs in $O(m^2|CU_k(P)|)$ for the dynamic programming version and in $O(km^2|CU_k(P)|)$ for the automata version. Myers [11] showed how to reduce this to $O(m|CU_k(P)|)$ and $O(km|CU_k(P)|)$ respectively. The idea is that when all active cells of a column are equal to k we no longer need to compute the dynamic programming table. In our example after computing *aba* all active cells are equal to 1. At that point the only possible way to extend *aba* into a word of $CU_k(P)$ is by a suffix of P, corresponding to an active cell. In our example the possible suffixes are *baa*, *aa* and *a*, resulting in *ababaa*, *abaaa* and *abaa*. The problem is that *abaaa* does not belong to $CU_k(P)$ since *abaa* does. Myers showed a way to solve this based on the failure links of the Knuth-Morris-Pratt algorithm. Recently Hyyrö presented an improved version that achieves the same result in a sequential way that is relevant for bit-parallel algorithms [8].

The algorithm for the generation of $|SCU_k(P)|$ presented in this article runs in $O(m^2|SCU_k(P)|)$ for the dynamic programming version and in $O(km^2|SCU_k(P)|)$ for the automata version. It can be improved in the same way to $O(m|SCU_k(P)|)$ and $O(km|SCU_k(P)|)$ respectively. It was shown by Myers [11] that $|CU_k(P)| = O(n^{pow(m/k)})$, 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 neighborhood* which means that $|SCU_k(P)| = O(n^{pow(m/k)})$. However our results do show a practical improvement in speed, since $|SCU_k(P)| \leq |CU_k(P)|$.

5 Bit Parallel Implementation

5.1 Bit Parallel Dynamic Programming

Myers presented a way to parallelize the computation of D and D' [12] 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 application the $\lceil m/w \rceil$ typically takes the value 1 since $m = \Theta(\log_\sigma n)$ for hybrid algorithms. This leads to a complexity of $O(m)$. Hyyrö presented a modification of Myers algorithm [6] that we will now describe and extend to solve our problem.

Ukkonen was the first to notice the following properties of D and D' [18]:

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 [12,6].

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 computation. 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, except for cell $D'[0, 0]$, all other $D'[0, j]$ cells are inactive. However special care must be taken to update the proper cells of column 1. This can be done by 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 improper cell of column 3 is $D'[3, 3]$. What line 23 does is to check whether the canonical trace-back of $D'[3, 4]$ is non-horizontal. An horizontal canonical trace-back 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 dependencies to the list of improper cells. If a cell has a vertical canonical trace-back 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.

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

procedure INITIALIZE(Pattern P)
    $VP \leftarrow (1^m)_2$
    $VN \leftarrow (0^m)_2$
    For $z \in \Sigma$ **Do** $PM_z \leftarrow (0^m)_2$
    For $1 \leq i \leq m$ **Do** $PM_{P[i]} \mid\ \leftarrow 2^{i-1}$
    $CP1 \leftarrow 0$
    $VAL_0 \leftarrow (10101010\ \ldots)_2$
    $VAL_1 \leftarrow (01100110\ \ldots)_2$
    $\vdots$
    return $VP, VN, CP1,\ VAL_0,\ \ldots,\ VAL_{\lceil \log m \rceil - 1}$
end procedure
procedure UPDATE($(VP,\ VN,\ CP1,\ VAL_0,\ \ldots,\ VAL_{\lceil \log m \rceil - 1}),\ z)$
    $D0 \leftarrow (((PM_z\ \&\ VP) + VP){}^\wedge VP) \mid PM_z \mid VN$
    $HP \leftarrow VN \mid\ \sim (D0 \mid VP)$
    $HN \leftarrow VP\ \&\ D0$
    $VAL_0,\ \ldots,\ VAL_{\lceil \log m \rceil - 1} \leftarrow$ CARRY_EFFECT($HP,\ HN,\ VAL_0,\ \ldots)$
    $VP \leftarrow (HN << 1) \mid \sim (D0 \mid (HP << 1))$
    $VN \leftarrow (HP << 1)\ \&\ D0$
    $CP1 \leftarrow$ UPDATE_PROPER_CELLS($CP1,\ PM_z,\ HN,\ VN,\ VP$)
    return $VP,\ VN,\ CP1,\ VAL_0,\ \ldots,\ VAL_{\lceil \log m \rceil - 1}$
end procedure
procedure UPDATE_PROPER_CELLS($CP1,\ PM,\ HN,\ VN,\ VP$)
    $CP1 \leftarrow ((PM \mid HN \mid\ \sim VN)\ \&\ ((CP1 << 1) \mid 1)) \mid CP1$
    $CP1 \mid\ \leftarrow VP$
    $CP1 \leftarrow (CP1 >> 1)$
    $CP1 \leftarrow (CP1 + 1){}^\wedge CP1$
    return $CP1$
end procedure
procedure CARRY_EFFECT($HP,\ HN,\ VAL_0,\ \ldots,\ VAL_{\lceil \log m \rceil - 1}$)
    $carry \leftarrow HP \mid HN$
    $VAL_0 \leftarrow carry{}^\wedge VAL_0$
    $carry\ \& \leftarrow HN{}^\wedge VAL_0$
    $VAL_1 \leftarrow carry{}^\wedge VAL_1$
    $carry\ \& \leftarrow HN{}^\wedge VAL_1$
    $\vdots$
    $VAL_{\lceil \log m \rceil - 1} \leftarrow carry{}^\wedge VAL_{\lceil \log m \rceil - 1}$
end procedure

Keeping track of which cells are active can be done in several ways. The two most significant are:

WHILE Keeping a pointer to the lowest active cell and moving upwards.
CARRY Storing the values of D' in computer words.

The pointer solution is as far as we know the standard solution to this problem. Observe that the active improper cells of D' appear in contiguous cells. If from one column to the next we keep track of the lowest improper active cell we can determine whether that cell belongs to the last line of D', in which case the IS_MATCH_POINT procedure returns true. According to the adjacency properties of D' presented above, to find the lowest active improper cell of a column we can start at the cell that is diagonally adjacent to the lowest active improper cell of the previous column and move upwards until we find an active improper cell or a proper cell. If a proper cell is found, then the EXTENDS_TO_MATCH_POINT procedure returns false; otherwise it returns true. Computing a dynamic programming table with this method requires $O(m\lceil m/w\rceil + k)$ time since updating the lowest active improper cell amortizes to k. The generation of the *super condensed neighborhood* requires $O((m\lceil m/w\rceil + k)|SCU_k(P)|)$ time.

The idea of the CARRY solution is to store the values of D' in an unorthodox way. Values are stored across computer words and not in a single one. This solution requires $\lceil m/w\rceil\lceil\log m\rceil$ 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. This requires $O(\lceil m/w\rceil\lceil\log m\rceil)$ time.

It is enough to identify active cells whose value is k. In our example this can be done by evaluating $\sim$ VAL_0 & VAL_1 & $\sim$ VAL_2. With the CARRY method a dynamic programming table can be computed in $O(m\lceil m/w\rceil\lceil\log m\rceil)$ time and the *super condensed neighborhood* in $O(m\lceil m/w\rceil\lceil\log m\rceil|SCU_k(P)|)$ time.

A final improvement is to adapt the previous algorithm so that it works in an increased bit-parallelism fashion [9]. 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 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. It is also necessary to copy the values of the D' column $|\Sigma|$ times into the computer word just before instruction 7. This is done by $>>$ and $|$ operations.

Algorithm 3 describes the details of implementation of the necessary predicates using an NFA.

The F_i computer words store the N_P^k automaton states for row i. The S_i computer words store the $N_P^{\prime\prime k}$automaton states for row i.

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

Algorithm 3 Bit-Parallel version of the Algorithm. N_P^k represented by F_i and $N_P^{\prime\prime k}$ by S_i. Bitwise operations in C-style.

procedure IS_MATCH_POINT(Search Point $F_0, \ldots, F_k, S_0, \ldots, S_k$)
    return $F_k \&\& !(S_k \& 10^m)$
end procedure
procedure EXTENDS_TO_MATCH_POINT($F_0, \ldots, F_k, S_0, \ldots, S_k$)
    return $((F_0 \& \sim S_0) \mid \ldots \mid (F_k \& \sim S_k)) \&\& !(S_k \& 10^m)$
end procedure
procedure UPDATE(Search Point $F_0, \ldots, F_k, S_0, \ldots, S_k$, letter z)
    $F'_0 \leftarrow (F_0 << 1) \ \& \ PM_z$
    $S'_0 \leftarrow ((S_0 << 1) \mid 1) \ \& \ PM_z$
    for $i \leftarrow 0, k$ **do**
        $F'_{i+1} \leftarrow ((F_{i+1} << 1) \& PM_z) \mid F_i \mid (F_i << 1) \mid (F'_i << 1)$
        $S'_{i+1} \leftarrow ((S_{i+1} << 1) \& PM_z) \mid S_i \mid (S_i << 1) \mid (S'_i << 1)$
    end for
    return $F'_0, \ldots, F'_k, S'_0, \ldots, S'_k$
end procedure

Since the UPDATE and EXTENDS_TO_MATCH_POINT procedures run in $O(k\lceil m/w \rceil)$ the final algorithm takes $O(k\lceil m/w \rceil \ m \ |SCU_k(P)|)$. This is a conservative bound since it is easy to modify the algorithm so that it runs in $O((k\lceil m/w \rceil + m) \ |SCU_k(P)|)$, using the KMP failure links.

We also implemented a version based on Navarro and Baeza-Yates [4] variation of the NFA. The procedures are implemented in a similar way and the resulting algorithm runs in $O(\lceil k(m-k)/w \rceil \ m \ |SCU_k(P)|)$. This 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[3].

[3] Actually this could be reduced but the gains would be marginal.

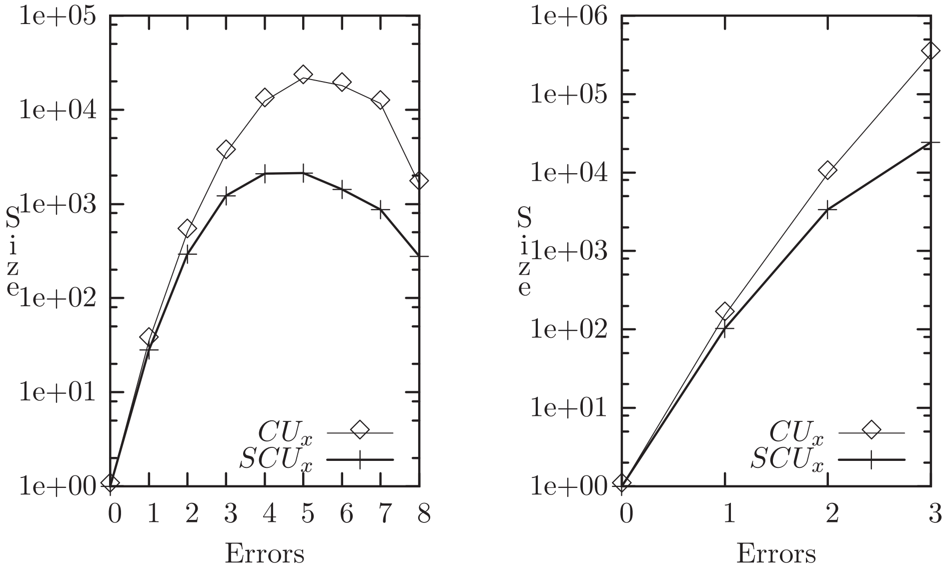

Fig. 7. Average size of the condensed neighborhood versus the super condensed neighborhood for $|P| = 16$ and $|\Sigma| = 2$ (left), $|P| = 6$ and $|\Sigma| = 16$ (right)

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

Table 3
Average size of CU_k vs SCU_k for $|P| = 8$.

We improved this to $O((\lceil k(m-k)/w\rceil \ + m)\ |SCU_k(P)|)$ using an approach similar to the one followed by Myers but found no difference in practice, since $O(\lceil k(m-k)/w\rceil)$ is constant for small patterns, which is the case for hybrid algorithms.

6 Experimental Results

We investigated the ratio between the average size of the condensed neighborhood versus the size of super condensed neighborhood (i.e. the number of possible different strings). The results are shown in figure 7 and table 3. These results were obtained by generating the neighborhoods of 50 random patterns.

We tested our NFA approach by analyzing its impact in the hybrid index [14]. Since we are only interested in the neighborhood generation phase, we set the j option of the index to 1, preventing the pattern from getting split. This was tested in a 800MHz Power PC G3 processor with 512K level 2 cache 640MB

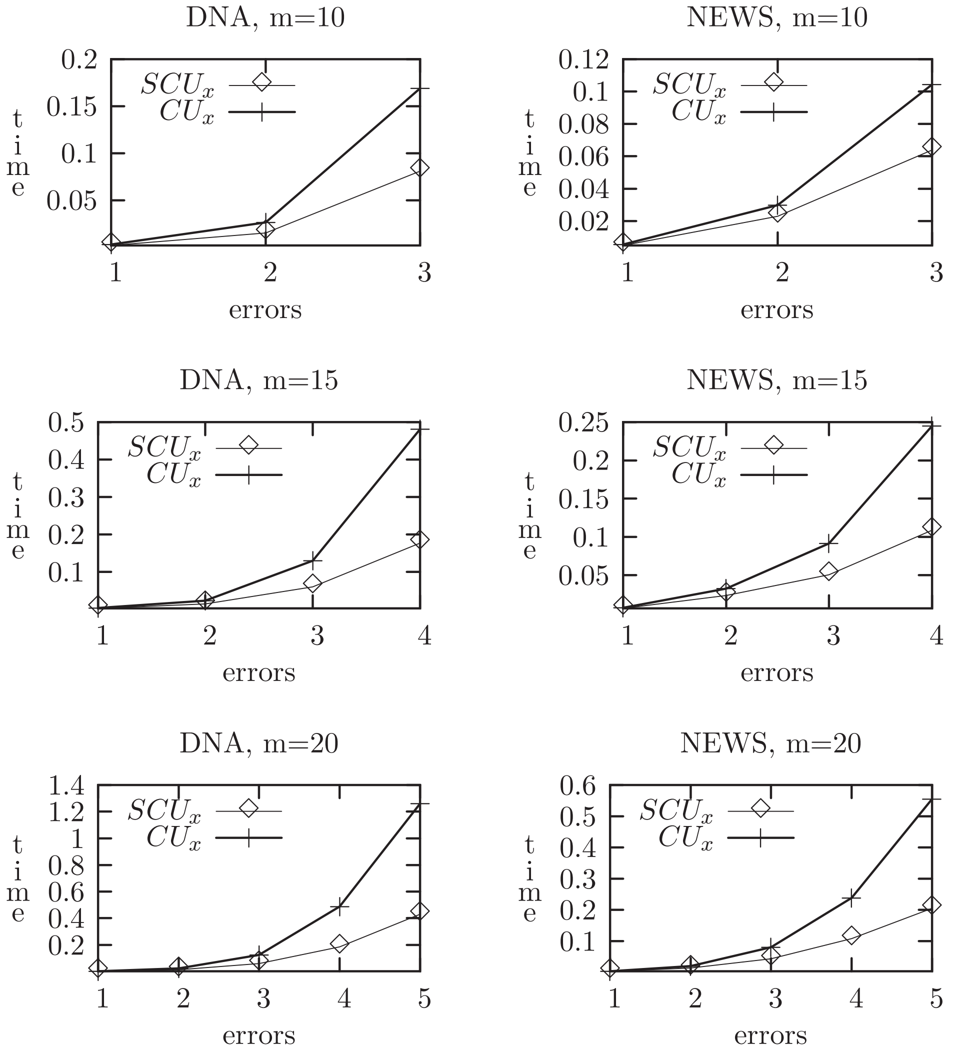

Fig. 8. Hybrid-Index. The left column shows the average time (in seconds) for searching in DNA data and the right column shows the average time for searching in the Newsgroups data. The pattern size is indicated by m.

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 [4].

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 [21], 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 8.

	$\|\Sigma\| = 2$		$\|\Sigma\| = 4$	
	$k = 2$	$k = 4$	$k = 2$	$k = 4$
CU_k	0.036	0.013	1.038	20.459
SCU_k-NFA	0.0043	0.0026	0.1048	0.171
SCU_k-WHILE	0.013	0.005	0.378	0.356
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-CARRY	0.009	0.003	0.125	0.142

Table 4
Bit-parallel and increased bit-parallel algorithms in milliseconds for $|P| = 8$.

In table 4 we present a comparison between the different methods of generating *super condensed neighborhoods* for patterns of size 8. The first row shows the times needed to generate *Condensed Neighborhoods* using the bit-parallel NFA. The next three rows show the times needed to generate *Super Condensed Neighborhoods* with our three alternatives. The two final rows show the times needed to generate *Super Condensed Neighborhoods* using increased bit parallelism. The NFA was implemented using Wu and Manber bit-parallel algorithm. Using these results we can rank our methods in order of decreasing performance as NFA, WHILE and CARRY. The results show that not even the increased bit parallelism approaches were able to outperform the NFA approach. This was expected since, as was pointed out by Navarro [13], NFA simulation is the fastest method for short patterns, which is the case for hybrid indexes. In fact it is also not surprising that the CARRY method outperformed slightly the WHILE method since the actual implementation of the WHILE method uses a higher number of branching instructions which compensate for the $O(\lceil \log m \rceil)$ factor of the CARRY method.

The results clearly show 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.

7 Conclusions

In this work, we addressed the problem of indexed approximate pattern matching by restricting our attention to the generation of super condensed neighborhoods. We have shown that this leads to 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 [14]. 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 [7]. Our algorithm generalizes all these cases.

More recently the authors of [10] presented the notion of *artificial prefix-stripped length-q neighborhood*, that modifies the condensed neighborhood in a way that adapts to Myers algorithm but that is not minimal. Therefore, according to lemma 8 the *super condensed neighborhood* is never larger than the *artificial prefix-stripped length-q neighborhood*, and in some cases it is strictly smaller [4]. The notion of *super condensed neighborhood* has in fact been considered by Hyyrö and Navarro [5] but no algorithm, for computing this neighborhood, was available until now.

We proposed an algorithm for generating *super condensed neighborhoods* that adapts very well to bit-parallel and increased bit-parallel approaches and can be implemented either by dynamic programming or using NFA's.

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

Acknowledgments

We are grateful to Eugene Myers for providing us access to his prototype and for suggestions, corrections and remarks. We also thank Gonzalo Navarro and Heikki Hyyrö for suggestions and remarks. We thank Gonzalo Navarro and Ricardo Baeza-Yates for making available their implementation of the hybrid index.

Parts of the work reported in the article have been previously presented in CPM [16] and SPIRE [17].

[4] For example for DNA when $P = atcg$ and $k = 1$ the string *aatcg* belongs to the *artificial prefix-stripped length-q neighborhood* but not to the *super condensed neighborhood*.
[5] Personal communication.

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. A unified view to string matching algorithms. In Keith G. Jeffery, Jaroslav Král, and Miroslav Bartosek, editors, *SOFSEM*, volume 1175 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[3] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[4] 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.

[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] Heikki Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, Department of Computer and Information Sciences, University of Tampere, 2001.

[7] Heikki Hyyrö. *Practical Methods for Approximate String Matching*. PhD thesis, University of Tampere, 2003.

[8] Heikki Hyyrö. An improvement and an extension on the hybrid index for approximate string matching. In Alberto Apostolico and Massimo Melucci, editors, *SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 208–209. Springer, 2004.

[9] Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro. Increased bit-parallelism for approximate string matching. In Celso C. Ribeiro and Simone L. Martins, editors, *WEA*, volume 3059 of *Lecture Notes in Computer Science*, pages 285–298. Springer, 2004.

[10] Heikki Hyyrö and Gonzalo Navarro. A practical index for genome searching. In Mario A. Nascimento, Edleno Silva de Moura, and Arlindo L. Oliveira, editors, *SPIRE*, volume 2857 of *Lecture Notes in Computer Science*, pages 341–349. Springer, 2003.

[11] Eugene W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

[12] 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.

[13] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[14] Gonzalo Navarro and Ricardo A. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[15] Gonzalo Navarro, Ricardo A. Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[16] 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.

[17] Luís M. S. Russo and Arlindo L. Oliveira. Faster generation of super condensed neighbourhoods using finite automata. In Mariano P. Consens and Gonzalo Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 246–255. Springer, 2005.

[18] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[19] 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.

[20] Sun Wu and Udi Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[21] `http://www.gia.ist.utl.pt/~acardoso/datasets`.