Pattern Matching Using *n*-gram Sampling Of Cumulative Algebraic Signatures : Preliminary Results

Witold Litwin[1], Riad Mokadem, Philippe Rigaux & Thomas Schwarz[2]

Extended Abstract

We propose a novel string (pattern) matching algorithm called *n-gram search*. We intend it for the records stored once and searched many times in a database or a file, especially organized into a Scalable Distributed Data Structure, (SDDS), over a grid or a structured P2P net. We presume that the records are encoded into their *cumulative algebraic signatures,* providing incidental confidentiality of stored data. The search starts with pre-processing the pattern, calculating the *logarithmic algebraic signature (LAS)* of the pattern and the *LASs* of every *n*-gram in it. The value of $n \geq 1$ is a parameter that one may tune. The search attempts to match the *LASs* of *n*-grams in the pattern towards dynamically calculated *LASs*, sampled over *n*-grams in the records. A mismatch generates a shift of up to *K*-*n* symbols towards next sample, where *K* is the pattern length. The whole process is parallel over the SDDS servers and does not require any local decoding. For an *M*-symbol long record, the unsuccessful search, measured as number of match attempts, costs $O((M\text{-}K) / (K\text{-}n+1))$. The 2-grams should typically suffice, leading to $O((M\text{-}K) / (K\text{-}1))$. We show that the algorithm particularly efficient for larger strings and records, i.e., with e-documents or DNA data. Preliminary results show then the *n*-gram search about $(K - n + 1)$ faster than our previous algorithms and among the fastest known, e.g., probably often faster than Boyer-Moore.

Key words

SDDS, grid, structured P2P, scalable distributed pattern matching, algebraic signatures.

1 Algebraic Signatures

Let G be a $GF(2^f)$, i.e., a Galois Field with the elements $0,1\ldots2^{f-1}$. We call these elements *symbols* and naturally consider them as bytes (f = 8) or words (f = 16 or 32 etc). Let S_K be a string of K symbols $p_1..p_K$. Let α be a primitive element in G. The *cumulative* (1-symbol) *algebraic signature (CAS) of* S_K, [LMS5], noted here *CAS* (S_K), is the string noted $S_K' = p'_1..p'_K$ where for every $i = 1,2\ldots K$:

$$p'_i = p_1\alpha + \ldots + p_i\alpha^i.$$

Here the addition and the multiplication are in *G*. Accordingly, p'_i is CAS of the prefix S_i of S_K ending with p_i. We write p'_i = CAS (S_i) or simply p'_i = CAS (p_i).

The CAS replaces each individual symbol *p* with another symbol *p'* whose value encodes not only *p*, but also some knowledge of several symbols preceding *p*. The rationale is that a single comparison of two symbols may indicate now not only whether these symbols are different or equal, but also whether two entire strings of the same length and ending with the compared symbols differ or are likely to be equal. This property is of obvious potential interest for

[1] Université Paris Dauphine
[2] Santa Clara University

string searches. In particular one may immediately observe that the complexity of prefix match operation, i.e., of the search for a record with a given prefix, of theoretically any length $l \leq K$, becomes independent of l and reaches O (1). This, because, for an unsuccessful search, it suffices to compare only the last symbol of the searched prefix with that in the record at the same offset. Any prefix match attempt in the original record (non-encoded) could need up to O (l) comparisons.

The CASs have several practical properties completing the rationale. First we have for every p'_i with $i > 1$:

$p'_i = p'_{i\text{-}1} + p_i\alpha^i = s\ (p_{i\text{-}1}) + p_i\alpha^i$.

We can thus reuse the signature of the previous symbol to encode the next symbol. This leads to O (K) (linear) complexity of the CAS encoding . Next, for every i, we have:

$p_i = (p'_i - p'_{i\text{-}1})\ /\ \alpha^i = \ (p'_i + p'_{i\text{-}1})\ /\ \alpha^i = (p'_i + p'_{i\text{-}1}) * \alpha^{f-1-i} = (p'_i \text{ XOR } p'_{i\text{-}1}) * \alpha^{f-1-i}$

This property provides also O (K) speed to the *CAS* decoding. It results from the well-known properties of any *GF*.

Consider now the substring $S_{k,l} = p_k \dots p_l$ of S with $1 < k < l < K$. To find the algebraic signature that it would have if it was instead a string $S_{l\text{-}k+1} = p_1 \dots p_{l\text{-}k+1}$, it suffices to compute:

$AS\ (S_{l\text{-}k+1}) = (p'_l\ XOR\ \ p'_{k\ \text{-}\ 1})\ /\ \alpha^{k\text{-}1}$

This property let us to efficiently match the pattern to any substring within the visited record. For the fast calculus of multiplications and of divisions in $GF(2^f)$ with $f = 8$ or $f = 16$ the log_α and $antilog_\alpha$ tables seems the most convenient tool. See [LMS5a] for details. In our case above we have the formula:

$AS\ (S_{l\text{-}k+1}) = antilog_\alpha\ [\ (log_\alpha\ (p'_l\ XOR\ \ p'_{k\ \text{-}\ 1})\ \text{-}\ \ k+1) \bmod 2^{f-1}]$

Similar transformations hold for the other formulae above. Notice that for the logs, the additions/subtractions are the usual ones.

2 The n-gram Search

2.1 Preprocessing

Let S'_k be the CAS of the searched string S_K (the pattern). Let n be a parameter. We basically consider that $n = 2$ below. An n-gram within S_k is any substring $p_i \cdots p_{i+n\text{-}1}$; $i = 1 \dots K\text{-}n+1$. We define similarly the n-grams in the visited record. Once called, the algorithm starts by pre-processing S_K as follows.
- If S is some string, let the *logarithmic algebraic signature* (*LAS*) of S be $log_\alpha AS$ (S). We put $LAS\ (p_{K\text{-}n+1} \cdots p_K)$, i.e., log_α of the signature of the terminate n-gram of S_K into some variable V. For any other n-gram, except the rightmost one, we hash *LAS* into the following table T.
- T has L entries: $T\,[0 \dots L\text{-}1]$. We fix L as $K + 2\delta$ with δ chosen somehow arbitrarily as $\delta = Int\ (0.25K)$. If P is an n-gram then we hash P to $T\,(i)$ with $i =$ AS (P) mod (K + &). The rationale for our choice of δ is to have only a few collisions on the primary entries, thus $T\,[0...K+\delta\text{-}1]$. We use the last δ entries of T for the overflows, managed using the well-known separate chaining collision resolution method.
- Each entry $T\,(i)$ is the triplet denoted (s, p, d). We have:
  - s is *LAS* of the rightmost n-gram in S_K hashed to $T\,(i)$. Different *LAS*'s, of different n-grams thus, may indeed hash to the same i. A choice different from ours could lead to an incorrect search result, as it will appear.

- p is the offset of the hashed n-gram with respect to that of the terminal n-gram in S_K. We use p as basis for the shifts within the searched record described below.

- d is zero or is the pointer towards the next overflow location on the collision chain starting at $T(i)$. Thus $T(i+d)$ contains the log_α signature etc. of an n-gram P' of S_K, hashed to $T(i)$ that happens to already store the data of n-gram P, also hashed there, while having a signature different of that of P'.

2.2 Processing

We describe the processing for a single record R of length M. In an SDDS and SDDS-2005 our description corresponds to the processing of the non-key field only. Each R is supposed stored pre-encoded into its *CAS* (R).

1. Let $R^n{}_i$ denote the n-gram in R ending with p_i. Likewise, $S^n{}_i$ is the n-gram in the pattern. We typically call it simply S. We start with the matching for $i = K$. We consider that the n-grams match iff their *AS*'s, hence *LAS*'s, are equal (no bad jokes, please). We thus test whether $V = LAS\ (R^n{}_K)$. We calculate the latter using the algebraic formulae of the previous section. Thus, if $p^R{}_i$ is the symbol at the offset i in R we simply have:

$LAS\ (R^n{}_K) = p^R{}_K$ XOR $p^R{}_{K-n}$.

2. If these n-grams match, we compare the *AS*'s of the entire S and of $R_{1,K}$, i.e., we test $LAS\ (S) = p^R{}_K$. If OK, we finally test for the collision, by matching every corresponding symbol in (encoded) S and R. If this last test is also OK, the search is successful.
3. Otherwise, we hash $LAS\ (R^n{}_K)$ into i and we test the *LAS* against s's in the chain starting at $T(i)$. If we do not find any (equality) match, then the search shifts forward in R by $K - n + 1$ positions. We will now thus compare S to the substring $p'_{K-n+1}\ldots p'_{2K-n+1}$ in R. Otherwise the search shifts by $p\ (j)$ positions where j is the index of the matching entry in T. Notice that this shift has to be smaller than the previous one, by at least one.
4. The shift means that we basically loop over steps (1) to (3) for every new substring. The difference is that to calculate the signature of the terminal n-gram in the substring we XOR with p'_{K-n} and divide the result by α^{K-n}. We perform the latter using the logarithm, as explained in the previous section.
5. The looping ends when either the search is successful or the last shift aligning S on the suffix of R still shows no match. This means an unsuccessful search. Notice that the last shift may bounce a part of S beyond R. If it happens, the attempt leading to this shift is the last to consider. Otherwise, the (last) attempt uses the *LSA* of the entire pattern only.

Example. We first consider the search in a French text, choosing arbitrarily R = 'Universite de Technologie Paris Dauphine' and S = 'Dauphine', Figure 1. We choose n =2, i.e. digram search. We start with the pre-processing of 'Dauphine'. We put aside the *LAS* of the terminal digram 'ne' in a variable for fast access. Since the length of the pattern is $K = 8$, the choice of dim $[T] = 12$ should suffice with the hash mod 10[3]. We thus hash into T, one after the other, *LAS's* of the digrams 'in', 'hi', 'ph'…'Da'. We do not show the result here. We notice however that no digrams collided, rendering the same *LAS*. If it happened, the later one would not enter T, as if it was simply a repetition of the former digram in the pattern.

The search using the digrams needs then 6 attempts and thus 5 shifts, Figure 1a. This, assuming no collision of a visited *LAS* with some in T, what we did not test. We underlined the examined digrams. We show the successive shifts one after the other in two lines under the pattern. The 5th attempt visiting 'up' finds it in T at the offset 4 with respect to the end of

[3] The choice of 2^i primary location, e.g., 16 instead of 10, could lead to faster hashing, but at this stage we neglect such optimisations.

the pattern. After the shift, the pattern aligns on the suffix of the record. This attempt calculates the signature of the entire pattern only.

```
(a)  Universite de Technologie Paris Dauphine
     Dauphine      Dauphine      Dauphine
            Dauphine      Dauphine   Dauphine

(b)  Universite de Technologie Paris Dauphine
     Dauphine  Dauphine   Dauphine   Dauphine
       Dauphine   Dauphine       Dauphine
```

Figure 1 *n*-gram search in (encoded) French text using (a) *n* = 2 and (b) *n* =1.

The scan using $n = 1$ needs 7 attempts, Figure 1b. Notice that Boyer-Moore algorithm, [BM77], would need the same number of attempts. Whether it could be in the end faster or slower using wall clock speed remains to be studied.

Notice also that digrams here perform also better by one attempt than 3-grams (not shown). All this illustrates the tunable behaviour of the algorithm. It is, besides, the only tunable pattern match algorithm we are aware of. In our case, $n = 2$ is optimal, otherwise, if a single symbol is highly discriminative, $n = 1$ may be a better option. Generally, larger n appears preferable for patterns longer with respect to the size of the alphabet used. The symbols repeat then many times in the pattern. For $n = 1$, a typical shift moves in this case the search forward by a small fraction of the pattern only. A larger n, keeps the average pace close to $K - n + 1$. Recall that to calculate and manipulate an *LSA* for an *n*-gram in *R* or *S* costs the same for any n under the consideration.

The example at

```
                   AGACAGAT AGACAGAT
                AGACAGAT AGACAGAT

(c) AGCATATAAAGCGAGTGCGGAGCAT
         AGACAGAT   AGACAGAT
          AGACAGAT    AGACAGAT
          AGACAGAT    AGACAGAT
           AGACAGAT     AGACAGAT
             AGACAGAT    AGACAGAT
              AGACAGAT
                AGACAGAT
```

Figure 2 illustrates this facet of our approach for a DNA sequence. It is inspired by the one in [CL4]. We have four letter alphabet of nucleotides: A, C, G, T. The pattern is 'AGACAGAT'. Choosing $n = 1$ leads to 12 attempts. Choosing $n = 2$ reduces the search cost three times to 4 attempts. Unlike for the text, the best choice is here however $n = 3$. It leads to 3 attempts only, hence accelerates the search by factor of four. No larger n does better. Notice that Boyer-Moore would need again the same number of attempts as for $n = 1$. It thus should be typically much slower than our trigram search. The comparative analysis in depth remains however to be studied.

3 Performance

If we choose n so that n-grams are selective, then the signature of the visited one usually will not be in *T*. The typical shift will be the longest possible under the method which is K-n+1.

Our basic choice of *n* = 2, i.e., of using digrams, amounts to *K* -1. The reason for this choice is that it should be typically more selective than of *n* = 1, or much more selective, as we have just seen. The ratio may be about the square of matching probability. The latter choice may then lead to many successful matches in *T*. This lowers the average shift length and increases uselessly the search cost. Assuming good practical selectivity of the digrams, the unsuccessful search speed of our algorithm should be *O* ((*M* – *K*) / (*K* -1)) attempts. This cost is the basic component of the overall search cost over the SDDS bucket, (the database or the file…), when, as usual, only a small fraction of existing records matches. The same cost characterizes the scan for all the occurrences of the pattern in a record. The pattern pre-processing itself costs *O* (2*K* – *n* + 1). It involves, we recall, the calculus of the LAS of the pattern and of its *n*-grams.

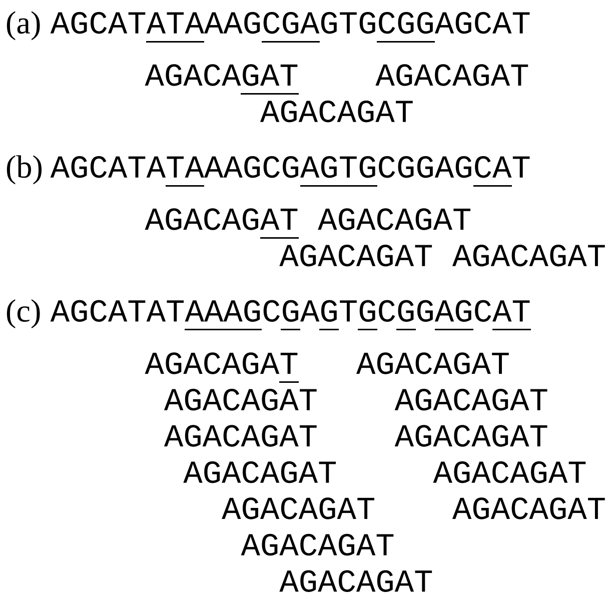

Figure 2 *n*-gram search in (encoded) DNA sequence for (a) *n* = 3, then (b) *n* =2 and (c) *n* =1.

One aspect specific to the method is the possibility of a collision between *n*-grams in the pattern on their LSA's. We recall that this happens if two different digrams have the same signature. Collisions obviously slow the average search speed. We recall from the theory of algebraic signatures in [LS4] that the probability of a collision is zero if two *n*-grams differ by only one symbol (the algebraic signatures were the first signature scheme known for this property and its more general formulation for *l*-symbol signatures not dealt with here.). Otherwise the collision probability for *n* > 1 is 2^{-f}, e.g., 1/256 for byte based texts (ASCII, EBCDIC…) and 1 / 64K for Unicode. Also, *n*-grams with two symbols switched, the digrams like 'xy' and 'yx' especially, never collide.

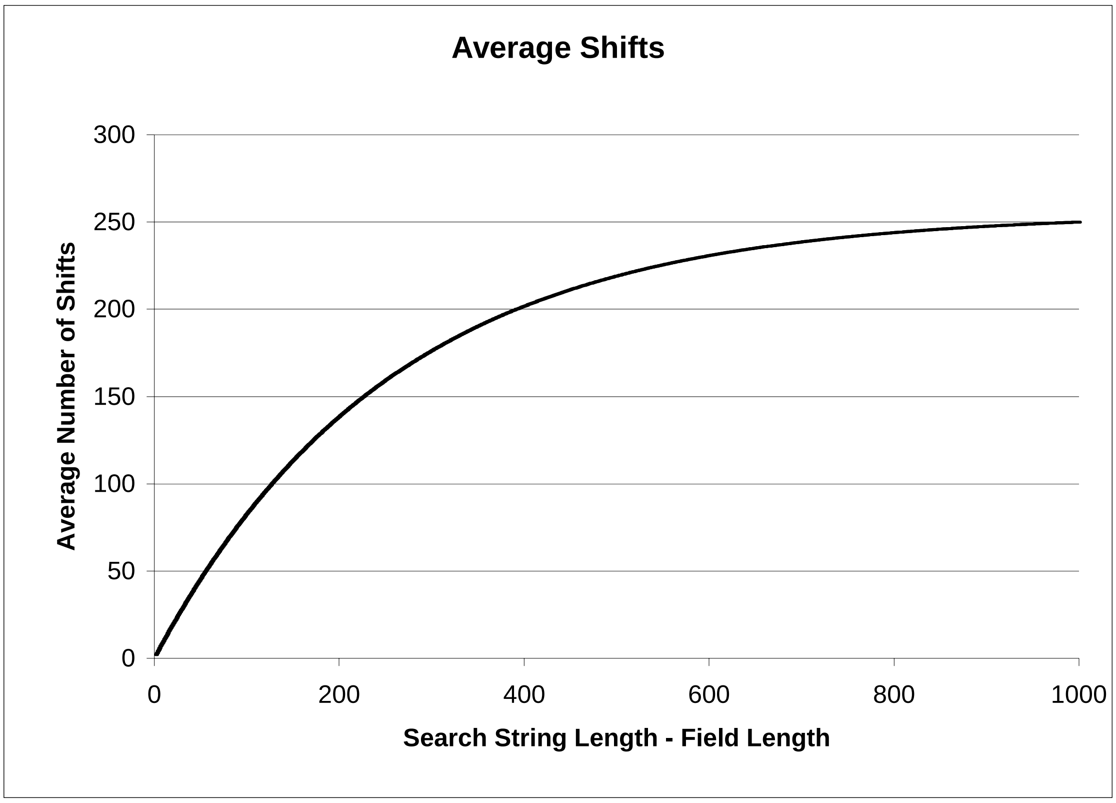

Figure 3 : Average shifts for DNA (one nucleotide per byte) for n-grams with n ≥ 4.

DNA data is often stored as a byte string with characters 'A','C','G', and 'T'. We found an encoding of these characters that guarantees that there are no collisions for *n*-grams with n ≤ 4. In addition, we found both theoretically and practically, that the size of the shifts goes to 255.004 with increasing size *n* while for small *n* the shift size is approximately *n*+1. As a consequence, only very few (namely 1/(*n*+1) or 1/255.004) of all locations in the DNA string need to be searched. Figure 3 shows our result for large *n*.

4 Related Work

SDDS-2005 already allows storing the SDDS files with records encoded into their *CAS*'s. The goal was the protection against incidental viewing of data at the servers, while preserving the parallel distributed non-key scan capability at the servers. With *CAS* encoding, it is impossible to incidentally see the content of a record, e;g., while studying a storage dump of some utility after a crash. One needs to find the α value, not stored at the servers, and (voluntarily) decode the records. This is analogous to open a sealed envelope in real life with someone's else letter, with all the legal consequences of eavesdropping, if caught.

SDDS-2005 offers several string search algorithms, including for the pattern matching. The latter uses a Karp-Rabin alike sequential shift, [KR87], [CL4]. The scan calculus is different, especially since our data are encoded. Its complexity is nevertheless the same: $O\,(M - K)$. Experiments with the Karp-Rabin version defined in [CL4] have shown the actual time also to quite similar[4]. It appears in fact slightly faster in our case for patterns longer than 32 bytes. This algorithm requires about twice less pre-processing of the pattern than the *n*-gram search. It only calculates the *LAS* of the pattern, with the complexity of $O\,(K)$. The *n*-gram search should thus be usually substantially faster within each record for $K >> 1$, and $M >> K$. As we

[4] It is not the only version. For instance, Rivest in his book presents, what he calls a slightly different Rabin-Karp algorithm. Another known variant, e.g., taught by Ingold (Fribourg U.), uses a division by a large prime.

have seen, the improvement ratio should be close to $K - 1$, hence linear with pattern length. The actual timing per record over the scan of the whole bucket remains to be found. One can expect a smaller ratio in practice, because of the time to move among the records the bucket structure. We recall that this one is a RAM B-tree in SDDS 2005.

In the open literature, pattern matching is among the fundamental problems of computer science with several prominent contributions, [CL4]. The popular algorithms do not require any specific record pre-processing (encoding). The Boyer-Moore algorithm seems particularly efficient in practice. Its scan and shift complexity can be about ours for $n = 1$, namely, whenever the "bad character" shift becomes prominent with respect to the "good suffix" suffix case. In all the already discussed conditions, the choice of the best $n > 1$ should make an average *n*-gram shift perhaps several times longer. The *n*-gram search time should then be often faster. Likewise, it should then outperform the other well-known algorithms. As we said, the precise comparison remains however an open research problem.

5 Conclusion

The algebraic signature based *n*-gram pattern matching appears particularly efficient for longer patterns in the context of the database search. It should be a useful new tool for text, image, DNA… string search, especially in the scalable distributed (P2P, grid…) environment. It seems possibly faster than the prominent algorithms. It also enhances the data security. Further work should precisely determine its best application conditions.

Acknowledgments. The support for this work came from EGov IST project.

References

[BM77] Boyer R.S., Moore J.S. A fast string searching algorithm. CACM, 20, 1977.

[CL4] Crochemore, M., Lecroq, T. Pattern matching and text compression algorithms. Computer Science and Engineering Handbook. A. Tucker (ed.) CRC Press Inc., 2004

[KR87] Karp, R. M., Rabin, M. O. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31, 2, March 1987.

[LMS5] W Litwin, R.Mokadem & Th.Schwarz. Cumulative Algebraic Signatures for fast string search. Protection Against Incidental Viewing and Corruption of Data in an SDDS. VLDB DBIS-P2P 2005, Springer, LNCS series.

[LMS5a] W Litwin, R. Moussa, T Schwartz. LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure. ACM-TODS, Sept. 2005.

[LS4] Litwin, W., Schwarz, Th. Algebraic Signatures for Scalable Distributed Data Structures. IEE Intl. Conf. On Data Eng., ICDE-04, 2004.