Efficient Exact Pattern-matching in Proteomic Sequences

Sérgio Deusdado[1], Paulo Carvalho[2]

[1]Department of Exact Sciences, ESA, Polytechnic Institute of Bragança, Portugal
[2]Department of Informatics, Engineering School, University of Minho, Portugal

Abstract. This paper proposes a novel algorithm for complete exact pattern-matching focusing the specificities of protein sequences (alphabet of 20 symbols) but, also highly efficient considering larger alphabets. The searching strategy uses large search windows allowing multiple alignments per iteration. A new filtering heuristic, named compatibility rule, contributed decisively to the efficiency improvement. The new algorithm's performance is, on average, superior in comparison with its best-rated competitors.

Keywords: exact pattern-match, searching algorithms

1 Introduction

Basically, a pattern search algorithm finds all instances of a string-pattern p of length m in a text x of length n, being $n \geq m$. Strings p and x are built over a finite set of characters in a given alphabet $\sum$ of size σ. From classic pattern-matching algorithms, KMP (Knuth-Morris-Pratt) [1] and BM (Boyer-Moore) [2] contributions improved significantly this nuclear computation recurrence in the late 1970s. The BM algorithm proceeds by sliding a search window of length m over the text. The text inside the window is checked against the pattern, from rightmost to leftmost character, and potentially encloses one pattern occurrence. Further investigation on BM caused new versions. The most relevant are Horspool's variant [3] and Sunday's Quick Search algorithm [4], but many more are included in the BM family algorithms.

A simplified searching strategy based on a text partitioning scheme and constant window shifts was presented in [5], improving performance comparatively with BM family algorithms. In this approach multiple alignments could occur per iteration.

Alternative pattern-matching algorithms use other approaches, such as suffix automata, bit parallelism or hashing. Representative examples are respectively, RF (Reverse Factor) [6], SO (Shift-Or) [7] and KR (Karp-Rabin) [8].

Recent algorithms follow hybrid approaches with refinements, and incorporate the best features of past algorithms to achieve better performance, being FJS [9] a significant hybrid example of heuristic based algorithms. Algorithms based on bit-parallelism are also part of the state-of-the-art, mainly on small alphabets. In this category the Backward Nondeterministic DAWG Matching (BNDM) algorithm [10] deserves mention, developed from the backward DAWG matching (BDM) algorithm

[11]. Further investigation in BNDM algorithm has produced a simplified and fastest version named SBNDM [12].

Recently, in [13], Lecroq proposed an adaptation of the Wu and Manber [14] multiple string matching algorithm to single string matching algorithm, including a new search strategy based on hashing *q*-grams.

For a comprehensive comparison of related algorithms we suggest [15] [16] [17].

In this paper, we propose a new algorithm for complete exact pattern-matching with a novel search logic responsible for superior performance and flexibility.

2 The New Algorithm: Description and Implementation

Combining known ideas like large search window, multiple alignments per iteration and constant shifts [5], with a novel search strategy and an innovative compatibility rule heuristic, the proposed algorithm extends searching flexibility and efficiency, keeping a low space complexity and simplicity. The new algorithm acts in two sequential phases, the pre-processing phase is devoted to analyze the pattern, and during the searching phase the text is iteratively scanned to identify pattern replicas. Let's analyze the lemma on which the searching strategy is based.

Lemma 1: If a pattern *p* of length *m* exists within a search window of *2m-1* contiguous characters, *p* includes necessarily the window's central character *x[cc]*. In consequence, the searching phase is focused primarily on *x[cc]*.
Proof: Being *2m-1* the length of the search window, any set of *m* consecutive characters within the window will include a common element - the central character of the window. By just verifying the *x[cc]* character is enough to determine whether or not *p* occurrences are possible.

Before detailing each phase, it is fundamental to define some basic concepts:
Large search window with eventual multiple alignments: A new search window, containing *2m-1* characters and centered in *p[m]*, is only considered when an instance of *p[m]* is found, which means that at least, one alignment needs to be tested. Each window may contain a maximum of *m* alignments to test. More concretely, each window includes a maximum of alignments equal to *p[m]* occurrences in the pattern.
Central character (*cc*) as alignment reference: Based on the Lemma 1, only the central character of the window participates in all possible alignments of *p*, thus, it is considered the reference for alignments testing. As mentioned before, all the possible alignments will match a *x[cc]* equal to the character in *p[m]*. Whenever a window is established *x[cc]=p[m]*. The *cc* value is also used as progression variable, as the searching task ends when *cc>n*.
Precedent character of *cc* as a parallel filter: The *x[cc-1]* character of the search window is used as sentinel for the compatibility rule. Any character in the alphabet could be a future *x[cc-1]*. Since *x[cc]* is always equal to *p[m]*, a pattern replica can only occur if *x[cc-1]* matches any precedent of *p[m]* occurrences in the pattern. Therefore, it is valuable to pre-compute which values of *x[cc-1]* are plausible to consider further verifications, with this knowledge is possible to discard several alignments just examining *x[cc-1]*.This constitutes the basis for the compatibility rule.
Compatibility rule: The compatibility rule is a selective rule that inhibits tests in the incompatible alignments. As seen, the characters that precede the occurrences of *p[m]* in the pattern are important. These precedents are relevant because can be used to preview useful alignments compatibilities in the search phase, since future *x[cc-1]* characters will be aligned with the referred precedents. An alignment is compatible with a future *x[cc-1]* if it possesses an equivalence in the precedent. Therefore, the compatibility table contains, for each character in the alphabet, its list of compatible alignments. The incompatibility is determined if the *x[cc-1]* of the window does not precedes any occurrence of *p[m]* in the pattern. In searching phase, pre-processed compatibilities for the *x[cc]* under analysis are available, allowing selective alignments trials.
Regular shift of *m* characters per window: After a search window is established and tested, it is always possible a default shift of *m* characters to reposition the next window. In fact, a new pattern instance could

only occur beyond the last alignment tested, which means that, at least, it has to finish one character ahead. **Cyclic extra-shift:** All iterations begin with a cycle of extra-shifts. A pre-processed shift table, based on the BM's bad character rule, provides extra-shift values to rapidly find $p[m]$ occurrences in the text. Initially, $cc=m$ and, while $cc \leq n$ and $x[cc] \neq p[m]$, a shift cycle is maintained being cc successively incremented with $extra_shift(cc)$. When $x[cc]=p[m]$ the cycle is interrupted to establish a new window.

2.1 Pre-processing Phase

This phase is mainly related to knowledge gathering through pattern analysis and it is initiated with the extra-shift table computation. Basically, this table contains the maximum shift value for each pattern character, and m for the remaining characters of the alphabet that do not integrate the pattern. As the extra-shift function will be applied to the character that immediately follows the window, if that character matches the last character of the pattern then the shift value will be null. Otherwise, the maximum shift value is obtained by observing the distance from the last occurrence of a character in the pattern to m. Later, in the search phase it is possible to shift repeatedly the window analyzing only the shift table. While the central character (cc) of the next window does not match $p[m]$, or $extra_shift(cc)>0$, the next central character can be incremented iteratively without further verifications. When the window progression stops, then $x[cc]$ is necessarily equal to $p[m]$, hence only the last character alignments need pre-processing, reducing considerably the algorithm's space complexity. Considering $p=$*"Albert Einstein"*, with $m=15$, the resulting shift table is shown in Table 1.

Table 1. Shift table example for $p=$*"Albert Einstein"*.

ASCII	…	32	…	65	…	69	…	98	…	101	…	105	…	108	…	110	…	114	115	116	…
Char.	…	spc	…	A	…	E	…	b	…	e	…	i	…	l	…	n	…	r	s	t	…
Max. Shift	15	8	15	14	15	7	15	12	15	2	15	1	15	13	15	0	15	10	4	3	15

As the only alignments that will be necessary to study are those involving $p[m]$, the occurrences of $p[m]$ in p are registered in a vector in two ways: the number of occurrences in the pattern (first cell), and each specific position or index of occurrence (the following cells). Table 2 illustrates an example for the pattern $p=$*"Albert Einstein"*, where the character $p[m]=$*'n'* (ASCII code 110) has two occurrences, at 10th and 15th characters.

Table 2. Pattern study detailing $p[m]=$*'n'* occurrences for $p=$*"Albert Einstein"*.

Occurrences	Occurrences' Indexes			
2	10	15	0	…

The pre-processing phase could now attain the compatibility rule's analysis, which represents the classification of the pattern alignments as compatibles or incompatibles under certain circumstances. The compatibility rule plays a major role in the proposed algorithm, being crucial to assure both performance and flexibility. Since the proposed algorithm uses a search window of $2m-1$ characters, frequently examines multiple alignments within the window, thus the compatibility rule aims to reduce the number of attempts per iteration inhibiting incompatible alignments. It is possible to

study all the pattern alignments of *p[m]*, grouping them (see Table 3), and analyze the conditions that should occur later in the search phase to effectively test an alignment.

Reusing *p="Albert Einstein"*, we have two alignments for the character in *p[m]*, as shown in Table 3. In the next phase, for each search window will be considered a central character (*x[cc]*) as reference for *p[m]* alignments, and if *x[cc]* will always meet a character *p[m]*, the previous character (*x[cc-1]*) will always meet the precedent characters of *p[m]* occurrences, previewed by the compatibility rule. Therefore, these characters can be used to differentiate the possible alignments as compatibles or incompatibles facing a future *x[cc-1],* enabling the prevention of redundant processing. In the example in Table 3, the alignments are only compatible with a *x[cc-1]='i'* (ASCII code 105), any different *x[cc-1]* means incompatibility.

Table 3. Alignments with the character in *p[m]* for *p="Albert Einstein"*.

		A	l	b	e	r	t		E	i	n	s	t	e	i	n											
							A	l	b	e	r	t		E	i	n	s	t	e	i	n						
															cc-1	cc											
a	n	d		t	h	i	s		r	e	p	r	e	s	e	n	t	s		t	h	e		t	e	x	t

The compatibility rule table (see Table 4) is pre-processed and will contain the compatible alignments' specifications, supplying the necessary parameterization to proceed with alignment tests in the searching phase. The compatibility rule is not effective to dismiss alignments when *x[cc]=p[1]* since no previous character exist. In these cases no alignments can be excluded. The alignments' indexes are stored backwards; this is required to find eventual pattern replicas in the correct sequence. In fact, greater indexes correspond to earlier pattern occurrences.

Table 4. Compatibility pre-processed table for *p="Albert Einstein"*.

ASCII	...	105	…
Align. 1	0	15	0
Align. 2	0	10	0
…	…	...	...
Align. n	0	0	0

2.2 Searching Phase

The searching phase is based on alignment trials over the iterative searching windows used to discover instances of the pattern within the text. A new window is centered in *cc*, and is only established if the detection of an eventually viable alignment occurs, this constitutes the primary filter. Initially *cc=m* and an extra-shit cycle is performed until *x[cc]=p[m]* or *cc>n*. In the best case the searching phase will end without testing any alignment and using always the maximal shift. However, in the average-case, after a short shift cycle the first window is established, then the characters *x[cc]* and *x[cc-1]* are used to evaluate the compatibility of the alignments to test them selectively. No matter the number of alignments to test within a window, the characters *x[cc]* and *x[cc-1]* involving all pattern occurrences will be tested only

once, saving redundant computation and enabling parallel verification. The validation relies on an efficient search strategy, derived from the principles of the Lemma 1.

Note that the last pattern's character is always an eventual alignment but not always a compatible one. The remaining occurrences of *p[m]* in the pattern are the other candidates to compatible alignments. By consulting the compatibility table where the compatible alignments for a particular *x[cc-1]* are described, further tests are performed selectively avoiding excessive computation. By only testing the character *x[cc-1]* it is possible to avoid several character comparisons to decide which ones are not viable. This feature contributes decisively to enhance efficiency.

When all the compatible alignments are tested the iteration is terminated. The searching phase ends when all the text has been scanned. An implementation proposal for the new algorithm's searching phase, in C language, is presented bellow.

```
void DC_Search(char *text, long n, char *pattern, unsigned int m)
{int cc,na,precedent,ia,iap,j,prefix;
 char b;
 b=pattern[m-1]; //Last character in pattern
 cc=m-1; // First cc
 while ((text[cc]!=b) && (cc<=n)) cc+=xshift[text[cc]];//1st extra-shift cycle
 while (cc<=n)
   {precedent=text[cc-1];
    ia=1;
    while ((na=compatibility[precedent][ia])>0)
     {prefix=na-2; // Prefix length
      iap=cc-prefix-1; // Position to align the pattern
      j=0;
      while ((j<prefix) && (text[iap+j]==pattern[j])) j++; // Prefix test
      if (j>=prefix)
          {j=prefix+2; //cc-1 and cc are pre-tested, advance to suffix
           while ((j<m) && (text[iap+j]==pattern[j])) j++;   // Suffix test
           if (j>=m) printf ("\nPattern at position %d.", iap);
          }
      ia ++;  // Advance to the following compatible alignment
     }
    cc+=m;  // regular shift
    while ((text[cc]!=b) && (cc<=n)) cc+=xshift[text[cc]];//Extra-shift cycle
   }
}
```

An illustrative example is now provided to better understand the behavior of the new algorithm during the processing phase. The searching example uses the text *x*="*This text includes the pattern Albert Einstein once.*", with *n*=52 and the pattern *p*="*Albert Einstein*", with *m*=15, already analyzed in the pre-processing phase section.

The first *cc* is initialized with *m*, thus *x[cc]='u'* (see Fig. 1) but, while $x[cc] \neq p[m]$ the extra-shift table is recurrently consulted to increment *cc* in order to rapidly find the next occurrence of *p[m]* in the text.

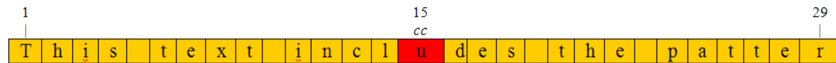

Fig. 1. The beginning of the first iteration and the first shift cycle.

Consulting Table 1, *extra_shift('u')* is 15, thus next *cc=30*. As the new *x[cc]='n'=p[m]*, the first search window is established (see Fig. 2). Analyzing the compatibility conditions within the first window (see Table 4), the compatibility rule states, facing a *precedent='r'*, that no alignments are compatible, and therefore, no further tests are needed.

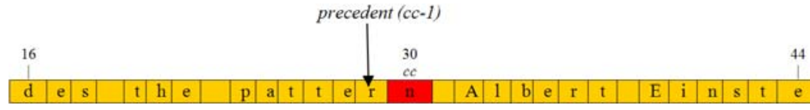

Fig. 2. The first search window is established, the characters *x[cc]* and *x[cc-1]* are analyzed.

The constant shift component is applied (advancing *m* characters), so the new *cc=45* and the next iteration begins with *x[cc]=’i’* (see Fig. 3).

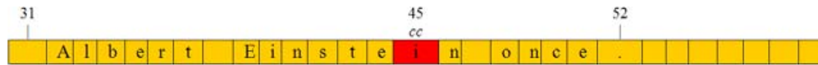

Fig. 3. The beginning of the second iteration and the second shift cycle.

Consulting Table 1, *extra_shift(‘i’)* is 1, thus next *cc=46*. As *x[cc]=’n’=p[m]*, the second search window is established (see Fig. 4).

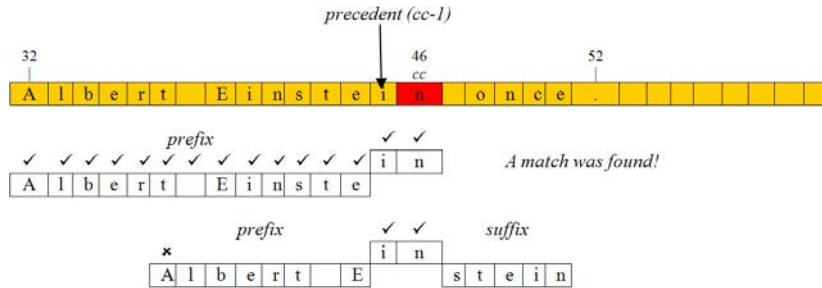

Fig. 4. The second window is established, the characters *x[cc]* and *x[cc-1]* are analyzed.

Accordingly with the compatibility rule (see Table 4), facing a *precedent=’i’*, it is necessary to test two compatible alignments, the first includes the pattern aligned with *x[cc]* by the 15th character, and the second includes the pattern aligned with *x[cc]* by the 10th character. Character comparisons are initiated to confirm or not prefix and suffix correspondences with the text. For the first alignment all characters match, therefore a pattern occurrence is revealed. The second alignment verification fails at the first character comparison. As the next *cc* will exceed *n*, the searching is finished.

3 Experimental Results and Comparisons

For simplicity reasons, all performance comparisons will be confined to four reference searching algorithms: the BMH algorithm, generically considered the fastest of all classical algorithms; the FJS algorithm (late 2005), also an important reference in terms of performance in general purpose exact pattern-matching based on heuristics; the SBNDM algorithm (2003), because it combines the advantages of bit parallelism and shifting heuristics achieving top performance when $m \leq \omega$, being ω the number of bits in a computer word (typically 32 or 64); and finally the WML algorithm, recently published (2007) and presenting top results, in particular on small

size alphabets. As all contenders are 1-gram based, and being WML, at least, a 2-gram algorithm, it was limited to *q=2* version (WML2) for comparison fairness.

For equity reasons, all the tested algorithms, including the proposed algorithm, referred here as DC, were coded in C language and compiled with *gcc* using full optimization. Alternative algorithms' implementations used in tests were obtained as follows: BMH is included in [17], FJS is provided by the authors in [9], SBNDM and WML2 implementations used were a courtesy of the authors. Performance tests were executed using a system based on an Intel Pentium IV - 3,4 GHz - 512KB cache - 1GB DDR-RAM, under Windows XP Professional SP2 OS. Execution times were collected in milliseconds using the *timeGetTime()* function, from *libwinmm.a* library.

The participant algorithms were tested searching protein data ($\sigma=20$) and natural language (ASCII) data ($\sigma=256$). The proteins sequence used was the result of merging four of the largest proteomes available, namely *Homo Sapiens*, *C. Elegans*, *A. Thaliana* and *Mouse Musculus* proteomes, obtained from Integr8 databases (www.ebi.ac.uk/integr8/). FASTA tags were cleared in the merged file, conserving only the amino-acid sequences, resulting in nearly 50 MB of raw data. The natural language text resulted from a compilation of 37 e-books, including authors like Charles Dickens, Victor Hugo, Sir Arthur Conan Doyle, Jules Verne, etc., based on ASCII plain text and obtained from Project Gutenberg (www.gutenberg.org), being the merged text length also about 50 MB. For each data type, a pattern collection containing 600 different patterns, based on 100 samples by length class, with *m=4, 8, 16, 32, 64* and *128*, were randomly generated (except for natural language patterns) and stored in a file for test purposes. The natural language patterns consist of English words when $m\leq 8$, the larger patterns are complete or incomplete sentences randomly chosen from the text.

We have run 1200 tests for each algorithm, employing 100 pattern samples multiplied by 6 different pattern lengths multiplied by 2 alphabet types. The following tables (Tables 5 and 6) contain a summary of the obtained results, consisting of the mean execution time in milliseconds, comprising pre-processing and searching phase.

Table 5. Runtimes for proteins data ($\sigma=20$), using several merged proteomes (~50MB).

	DC		BMH		FJS		SBNDM		WML2	
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank
4	**111**	**1**	149	4	120	2	132	3	235	5
8	**67**	**1**	86	3	77	2	96	4	111	5
16	**44**	**1**	56	3	52	2	61	5	60	4
32	32	2	42	5	40	4	**31**	**1**	37	3
64	**26**	**1**	33	4	32	3	$>\omega$	?	27	2
128	**22**	**1**	31	4	30	3	$>\omega$	?	**22**	**1**

Table 6. Runtimes for natural language ($\sigma=256$), using e-books compilation (~50MB).

	DC		BMH		FJS		SBNDM		WML2	
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank
4	**94**	**1**	127	4	95	2	101	3	197	5
8	**58**	**1**	75	3	65	2	84	4	96	5
16	**44**	**1**	55	3	52	2	64	4	52	2
32	**31**	**1**	39	4	39	4	37	3	32	2
64	25	2	32	3	29	2	$>\omega$	?	**24**	**1**
128	**17**	**1**	21	4	19	3	$>\omega$	?	**17**	**1**

Considering the 12 different competitions, DC collects 10 winnings. DC is clearly dominant and, in general terms, stands as the most efficient algorithm for alphabets with $\sigma \geq 20$. Protein sequences are searched, on average, 20% faster using DC.

4 Discussion and Conclusions

We have presented a novel algorithm oriented to exact pattern-matching in protein sequences or larger alphabets' based sequences. As the results demonstrate, the new algorithm is highly efficient and flexible, standing as the best choice when $\sigma \geq 20$.

The proposed algorithm is heuristic based, not limited in pattern length, introducing new heuristics and a novel search strategy. The most valuable contribution is the compatibility rule which enhances the multi-alignments windows searching strategy. The compatibility rule enhances the filtering capability and is particularly useful in presence of long patterns as it allows parallel verifications to decide selectively the useful alignments to test. Furthermore, the new algorithm includes a shift cycle to enhance performance when dealing with large alphabets, where alignments occur more rarely.

The complexity analysis for the proposed algorithm suggests a sub-linear behavior in the average case but, further analysis is necessary to theoretically demonstrate it.

In summary, attending to the innovative search strategy and the achieved performance, the proposed algorithm is a relevant contribution regarding flexible and efficient exact pattern-matching in protein sequences or natural language texts.

References

1. D. E. Knuth, J. H. Morris, and V. R. Pratt: Fast pattern matching in strings. *SIAM J. Comput.*, vol. 6(2), pp. 323--350 (1977)
2. R. S. Boyer and J. S. Moore: A fast string searching algorithm. *Commun. Assoc. Comput. Mach.*, vol. 20(10), pp. 762--772 (1977)
3. R. N. Horspool: Practical fast searching in strings. *Software - Practice & Experience*, vol. 10(6), pp. 501--506 (1980)
4. D. M. Sunday: A very fast substring search algorithm. *Commun. Assoc. Comput. Mach.*, vol. 33(8), pp. 132--142 (1990)
5. S. Kim: A new string-pattern matching algorithm using partitioning and hashing efficiently. *Journal of Experimental Algorithmics (JEA)*, vol. 4(2) (1999)
6. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: Speeding up Two String Matching Algorithms. *Algorithmica*, vol. 12(4/5), pp. 247--267 (1994)
7. R. A. Baeza-Yates and G. H. Gonnet: A new approach to text searching. *Commun. ACM*, vol. 35(10), pp. 74--82 (1992)
8. R. M. Karp and M. O. Rabin: Efficient randomized pattern-matching algorithms. *IBM J Res Dev.*, vol. 31(2), pp. 249--260 (1987)
9. F. Franek, C. G. Jennings, and W. F. Smyth: A Simple Fast Hybrid Pattern-Matching Algorithm. *Lecture Notes in Computer Science*, vol. 3537, pp. 288--297 (2005)
10. G. Navarro and M. Raffinot: Fast and Flexible String Matching by Combining Bitparallelism and Suffix automata. *ACM Journal of Experimental Algorithms*, vol. 5(4), pp. 1--36 (2000)
11. M. Crochemore and W. Rytter: *Text algorithms*. Oxford University Press (1994)
12. H. Peltola and J. Tarhio: Alternative Algorithms for Bit-Parallel String Matching. in Proceedings of SPIRE '03, 10th Symposium on String Processing and Information Retrieval (2003)
13. T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, vol. 102, pp. 229--235 (2007)
14. S. Wu and U. Manber: A fast algorithm for multi-pattern searching. TR-94-17, Department of Computer Science, University of Arizona, Tucson (1994)
15. P. D. Michailidis and K. G. Maragaritis: On-line String Matching Algorithms: Survey and Experimental Results. *International Journal of Computer Mathematics*, vol. 76(4), pp. 411--434 (2001)
16. T. Lecroq: Experimental Results on String Matching Algorithms. *Software - Pratice and Experience*, vol. 25(7), pp. 727--765 (1995)
17. B. Smyth: *Computing Patterns in Strings*. Pearson Addison-Wesley (2003)