Improving Regular-Expression Matching on Strings Using Negative Factors

Xiaochun Yang
College of Information Science and Engineering
Northeastern University, China
yangxc@mail.neu.edu.cn

Bin Wang
College of Information Science and Engineering
Northeastern University, China
binwang@mail.neu.edu.cn

Tao Qiu
College of Information Science and Engineering
Northeastern University, China
qiutao@research.neu.edu.cn

Yaoshu Wang
College of Information Science and Engineering
Northeastern University, China
wangysneu@gmail.com

Chen Li
Dept. of Computer Science
UC Irvine, USA
chenli@ics.uci.edu

ABSTRACT

The problem of finding matches of a regular expression (RE) on a string exists in many applications such as text editing, biosequence search, and shell commands. Existing techniques first identify candidates using substrings in the RE, then verify each of them using an automaton. These techniques become inefficient when there are many candidate occurrences that need to be verified. In this paper we propose a novel technique that prunes false negatives by utilizing *negative factors*, which are substrings that *cannot* appear in an answer. A main advantage of the technique is that it can be integrated with many existing algorithms to improve their efficiency significantly. We give a full specification of this technique. We develop an efficient algorithm that utilizes negative factors to prune candidates, then improve it by using bit operations to process negative factors in parallel. We show that negative factors, when used together with necessary factors (substrings that must appear in each answer), can achieve much better pruning power. We analyze the large number of negative factors, and develop an algorithm for finding a small number of high-quality negative factors. We conducted a thorough experimental study of this technique on real data sets, including DNA sequences, proteins, and text documents, and show the significant performance improvement when applying the technique in existing algorithms. For instance, it improved the search speed of the popular Gnu Grep tool by 11 to 74 times for text documents.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing;

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, Performance

Keywords

Regular expression, long sequence, performance

1. INTRODUCTION

This paper studies the problem of efficiently finding matchings of a regular expression (RE) in a long string. The problem exists in many application domains. In the domain of bioinformatics, users specify a string such as `TAC(T|G)AGA` and want to find its matchings in proteins or genomes [6, 13]. Modern text editors such as *vi*, *emacs*, and *eclipse* provide the functionality of searching a given pattern in a text document being edited. The shell command *grep* is widely used to search plain-text files for lines matching a regular expression. For instance, the command "`grep ^a.ple fruits.txt`" finds lines in the file called `fruits.txt` that begin with the letter `a`, followed by one character, followed by the letter sequence `ple`. The ability of processing regular expressions has been integrated into the syntax of languages (e.g., Perl, Ruby, AWK, and Tcl) and provided by other languages through standard libraries in .NET, Java, and Python.

A simple method to do RE search on a string is to construct an automaton for the RE. For each position in the string, we run the automaton to verify if a substring starting from that position can be accepted by the automaton. Notice that the verification step can be computationally expensive. The main limitation of this approach is that we need to do the expensive verification step many times. Various algorithms have been developed to speed up the matching process by first identifying candidate occurrences in the string, and then verifying them one by one [3, 4, 14]. These algorithms identify candidate places based on certain substrings derived from the RE that have to appear in matching answers, such as a prefix and/or a suffix. For instance, each substring matching the RE `xy(a|b)*zw*` should start with `xy` (a prefix condition) and end with `zw` or `z` (suffix conditions). Such substrings, called *positive factors* throughout the paper, can be used to locate candidate occurrences. Each of them will be further verified using one or multiple automata. Although these algorithms can eliminate many starting positions in the string, their efficiency can still be low when the positive factors generate too many candidate occurrences, especially when the text is long.

Contributions: In this paper we study how to improve the efficiency of existing algorithms for searching regular expressions in strings. We propose a novel technique that provides significant pruning power by utilizing the fact that we can derive substrings from the RE that *cannot* appear in an answer. Such substrings are called *negative factors*. We give an analysis to show that many candidates can be pruned by negative factors (Section 3).

In Section 4, we study how to use negative factors to speed up the matching process of existing algorithms. We show that negative factors can be easily integrated into these methods, and propose two bit-operation algorithms to do efficient search. In Section 4.3 we study the benefits of using negative factors. One interesting result is that considering necessary factors (substrings that have to appear in every answer) only does not provide much filtering power in prefix-based approaches. However, when combining both negative factors and necessary factors, we can gain much more pruning power. In Section 5 we study the problem of deciding a set of high-quality negative factors. There can be many negative factors, and the selected ones can greatly affect the performance of the matching process. To achieve a high efficiency, we propose an algorithm that only considers partial strings, and present an algorithm that can do early termination in the process of deciding negative factors. In Section 6 we present experimental results on real data sets including DNA sequences, protein sequences, and text documents, and demonstrate the space and time efficiency of the proposed technique. For instance, it improved the search speed of the popular Gnu Grep tool by 11 to 74 times for texts.

2. PRELIMINARIES

2.1 Regular Expression Matching

Let Σ be a finite alphabet. A *regular expression* (RE for short) is a string over $\Sigma \cup \{\epsilon, |, \cdot, *, (,)\}$, which can be defined recursively as follows:

- The symbol ϵ is a regular expression. It denotes an empty string (i.e., the string of length zero).
- Each string $w \in \Sigma^*$ is a regular expression, which denotes the string set $\{w\}$.
- If e_1 and e_2 are regular expressions that denote the set R_1 and R_2, respectively, then
  - (e_1) is a regular expression that represents the same set denoted by e_1.
  - $(e_1 \cdot e_2)$ is a regular expression that denotes a set of strings x that can be written as $x = yz$, where e_1 matches y and e_2 matches z.
  - $(e_1|e_2)$ is a regular expression that denotes a set of strings x such that x matches e_1 or e_2.
  - (e_1^+) is a regular expression that denotes a set of strings x such that, for a positive integer k, x can be written as $x = x_1 \ldots x_k$ and e_1 matches each string x_i $(1 \leq i \leq k)$. We use $\epsilon|e^+$ to express a Kleene closure e^*. In this paper, we consider the general case e^*.

Given an RE Q, we use $R(Q)$ to represent the set of strings that can be accepted by the automaton of Q. We use $|Q|$ to express the number of characters that Q contains. We use l_{min} to represent the length of the shortest string(s) in $R(Q)$. For example, for the RE $Q = (\mathtt{G|T})\mathtt{A}^*\mathtt{GA}^*\mathtt{T}^*$, we have $|Q| = 6$ since it has six characters: `G`, `T`, `A`, `G`, `A`, and `T`. The set of strings $R(Q) = \{$`GG`, `TG`, `GAG`, `TAG`, `GGA`, `TGA`, `GGT`, `TGT`, `GAGT`, $\ldots\}$. We have $l_{min} = 2$, since its shortest strings `GG` and `TG` have the length 2.

For a text (sequence) T of the characters in Σ, we use $|T|$ to denote its length, $T[i]$ to denote its i-th character (starting from 0), and $T[i, j]$ to denote the substring ranging from its i-th character to its j-th character.

For simplicity, in our examples we focus on the domain of genome sequences, where Σ=$\{$`A`, `C`, `G`, `T`$\}$. We run experiments in Section 6 on other domains such as proteins and English texts, where the Σ has more characters.

Pattern Matching of an RE. Consider a text T of length n and an RE Q of length m. We say Q matches a substring $T[a, b]$ of T at position a if $T[a, b]$ belongs to $R(Q)$. The substring $T[a, b]$ is called an *occurrence* of Q in T. The problem of pattern matching for an RE is to find all occurrences of Q in T. Figure 1 shows an example text. Suppose $Q = (\mathtt{G|T})\mathtt{A}^*\mathtt{GA}^*\mathtt{T}^*$. Then Q matches T at position 3, and the substring $T[3, 6]$ is an occurrence of Q in T.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	C	T	A	G	A	C	G	T	T	A	A	T	T	T	A	C	G	T	A

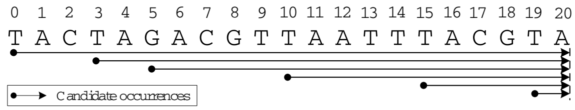
Figure 1: An example text.

2.2 Prefix-Based Approaches

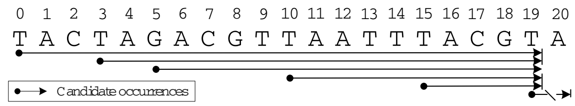
One naive approach for finding occurrences of an RE Q is to build an automaton for Q and run it from the beginning of the text T. An occurrence will be reported whenever a final state of the automaton is reached. The verification fails once a new feed character could not be accepted by the automaton. In this case, we move to the next position and repeat the verification process using the same automaton. We repeat the process for all starting positions in the text. Since the approach has to run the automaton for each position, it is very inefficient especially when T is very long (e.g., a chromosome of a human can contain 3 billion characters).

Recent techniques are developed for identifying sequences that contain at least one matching of an RE among multiple sequences. They utilize certain features of the RE Q to improve the performance of the automaton-based methods. Their main idea is to use *positive factors*, which are substrings of Q that can be used to identify candidate occurrences of Q in T. For instance, Watson in [14] used prefixes of strings in $R(Q)$ to find maximal safe shift distances to avoid checking every position in T. A prefix w.r.t. an RE Q is defined as a prefix with length l_{min} of a string in $R(Q)$. For example, for the RE $Q = (\mathtt{G|T})\mathtt{A}^*\mathtt{GA}^*\mathtt{T}^*$, the prefixes w.r.t. Q are `GA`, `TA`, `GG`, and `TG`. There are other kinds of positive factors, which are discussed in Section 4.3.

Figure 2(a) shows the main idea of this approach using the running example. The substrings $T[0, 1]$, $T[3, 4]$, $T[5, 6]$, $T[10, 11]$, $T[15, 16]$, and $T[19, 20]$ are six matching prefixes for the RE Q. The approach only examines matching suffixes of the text T starting from these matching prefixes using the automaton of Q. The automaton keeps examining each matching suffix until it fails, and it reports an occurrence whenever a final state is reached. We call this kind of approach "algorithm PFILTER," where "P" stands for "Prefix."



(a) Using prefixes.



(b) Combination of prefixes and last matching suffix.

Figure 2: Checking candidate occurrences of the RE $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$.

An alternative approach is adopted in NR-grep [9, 11], which uses a sliding window of size l_{min} on the text T and recognizes reversed matching prefixes in the sliding window using a reversed automaton. Similar to the definition of prefix, a suffix w.r.t. an RE Q is defined as a suffix with length l_{min} of a string in $R(Q)$. For example, for the RE $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$, the suffixes w.r.t. Q are `TT`, `AT`, `AA`, `GA`, `GT`, `AG`, `GG`, and `TG`, and the matching suffixes are $T[4,5]$, $T[5,6]$, $T[8,9]$, $T[9,10]$, $T[11,12]$, $T[12,13]$, $T[13,14]$, $T[14,15]$, and $T[18,19]$. We call the suffix-based approach "algorithm SFILTER," which is also similar to the algorithm PFILTER. It runs a reversed automaton from the end position of each suffix to the beginning of the text.

2.3 Improving Prefix-Based Approaches Using Last Matching Suffix

In Figure 2(a), the matching prefix $T[19,20] = \texttt{TA}$ could not be used to produce an answer string in $R(Q)$ since it is not among the suffixes identified above from the RE. Therefore, we could use the last matching suffix to do an early termination in each verification step. Figure 2(b) shows the example of improving the algorithm PFILTER. As we can see, by using the last matching suffix $T[18,19] = \texttt{GT}$ in the text T, a verification can terminate early at position 19. We call this approach the "algorithm PS." It only verifies those substrings starting from every matching prefix S_p to the last matching suffix S_s if the starting position of S_p is less than or equals to the starting position of S_s. We call S_s a *valid matching suffix* and each S_p a *valid matching prefix* w.r.t. its valid matching suffix S_s. For example, the substring $T[18,19]$ is a valid matching suffix and the substrings $T[0,1]$, $T[3,4]$, $T[5,6]$, $T[10,11]$, $T[15,16]$ are valid matching prefixes, whereas $T[19,20]$ is an invalid matching prefix. The algorithm PS requires $O(m \cdot n \cdot n'_p)$ time to do verifications for n'_p valid matching prefixes of T, assuming m is $|Q|$, and the verification for each valid matching prefix requires $O(m \cdot n)$ time.

3. NEGATIVE FACTORS

In this section, we develop the concept of negative factor, which can be used to improve the performance of matching algorithms. Contrary to positive factors, a negative factor is a substring that *must not* appear in an occurrence. We show that a negative factor can not only prune unnecessary verifications, but also terminate verifications early. We first present a formal definition of negative factors, then show a good pattern to prune candidates.

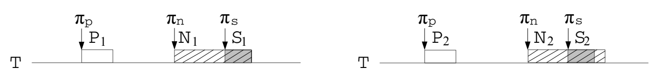
Definition 1. (Negative factor, or N-factor for short) Given a regular expression Q and a string w, a string w is called a *negative factor with respect to* Q, or simply a negative factor when Q is clear in the context, if there is no string $\Sigma^* w \Sigma^*$ does in $R(Q)$.

For a text T, an N-factor w.r.t. an RE Q must not appear in an answer to Q in T. For example, consider the RE $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$, the strings `C`, `AGG`, and `TTA` are N-factors, since they cannot appear in an answer as a substring.

LEMMA 1. *An N-factor w.r.t. an RE Q can not be a substring of a prefix or a suffix w.r.t. Q.*

THEOREM 1. *Given a text with length n, the number of N-factors w.r.t. an RE Q cannot be greater than $\sum_{i=1}^{n} |\Sigma|^i$.*

A PNS Pattern: Intuitively, we say a substring of T has a PNS pattern if it starts with a prefix of Q, has an N-factor in the middle, and ends with a suffix of Q. Formerly, let π_p, π_n, π_s be the set of starting positions of a matching prefix P, a matching N-factor N, and a matching suffix S in a text T, respectively. The substring $T[\pi_p, \pi_s + l_{min} - 1]$ conforms to a *PNS pattern* if N is a substring of $T[\pi_p, \pi_s + l_{min} - 1]$. Figure 3 shows that a substring conforms to a PNS pattern if and only if $\pi_p \leq \pi_n < \pi_s$ and $\pi_n + |N| \leq \pi_s + l_{min}$.



(a) A PNS pattern. (b) Not a PNS pattern.

Figure 3: A substring conforming to a PNS pattern iff $\pi_p \leq \pi_n < \pi_s$ and $\pi_n + |N| \leq \pi_s + l_{min}$.

Obviously, a substring of T conforming to a PNS pattern cannot be an occurrence of Q. Based on this observation, we can prune unnecessary verifications using N-factors. Figure 4 shows an example of the benefit by using PNS patterns.

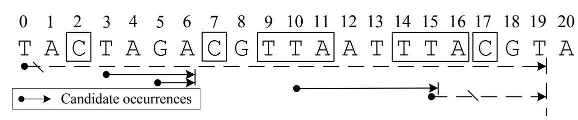


Figure 4: Using N-factors $T[2,2]$ and $T[17,17]$ to prune candidates of $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$. Compared with Figure 2(b), candidates $T[0,19]$ and $T[15,19]$ are pruned and the verifications starting from positions 3, 5, and 10 can be terminated early by using the N-factors $T[7,7]$ and $T[14,16]$, respectively.

Although the number of N-factors w.r.t. $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$ is large, we can still generate a small number of high-quality N-factors, such as {`C`, `AGG`, `ATA`, `ATG`, `GGG`, `GTA`, `GTG`, `TAT`, `TGG`, `TTA`, `TTG`}. (We will provide details in Section 5.) For the example in Figure 2(b), all the five candidate substrings conform to a PNS pattern, in which two of them can be pruned using the N-factors $T[2,2]$ and $T[17,17]$. For instance, the substring $T[15,19]$ is such a substring that consists of a matching prefix $T[15,16] = \texttt{TA}$, a matching suffix

$T[18,19]$ = GT, and an N-factor C at position 17. The N-factors can help us avoid the verifications of $T[0,19]$ and $T[15,19]$ (see Figure 2(b)). Furthermore, for the matching position of prefix $T[3,4]$ = TA, PNS can terminate the verification early when it meets the suffix $T[5,6]$ = GA, resulting in a better performance. Similarly, the verifications starting from the matching prefixes $T[5,6]$ = GA and $T[10,11]$ = TA can be terminated early at positions 6 and 15, respectively.

4. PRUNING UNNECESSARY CHECKS USING N-FACTORS

In this section we discuss how to combine N-factors with the algorithm PS to check candidate substrings that do not conform to a PNS pattern. In Section 4.1 we propose a merge algorithm to identify candidates and present two improved algorithms by using bit-parallel operations in Section 4.2. In Section 4.3 we show that negative factors, when used together with *necessary factors* (substrings that must appear in answers), can achieve better pruning power.

4.1 A Merging Algorithm

N-factors can be used to divide a text into a set of disjoint substrings, and we want to run the algorithm PS for each of them. Recall that the algorithm PS verifies substrings from every valid matching prefix to its corresponding valid matching suffix (see Section 2.3). For example, Figure 4 shows that the N-factors divide the text to disjoint substrings $T[0,1]$, $T[3,6]$, $T[8,10]$, $T[10,15]$, $T[15,16]$, and $T[18,20]$. Only $T[3,6]$ and $T[10,15]$ contain valid matching suffixes $T[5,6]$ and $T[14,15]$, respectively. The substrings $T[3,4]$ and $T[5,6]$ are valid matching prefixes w.r.t. the valid matching suffix $T[5,6]$, and $T[10,11]$ is a valid matching prefix w.r.t. the valid matching suffix $T[14,15]$, whereas $T[0,1]$, $T[15,16]$, and $T[19,20]$ are invalid matching prefixes.

In order to use N-factors in the PS approach, we sort starting positions of matches of prefixes, suffixes, and N-factors in the ascending order. A suffix trie of the text T can support this sorting operation with a large space overhead when T is long. To reduce the space cost, we can store the text T in a BWT format [7] and get an inverted list of starting positions for a substring α in a constant time ($O(|\alpha|)$) by simulating searches using BWT. For example, Figure 4 shows the inverted list of starting positions for the prefix TA is $\{0,3,10,15,19\}$ and the inverted list of starting positions for an N-factor C is $\{2,7,17\}$.

The basic idea of utilizing N-factors in the PS approach is to find a left valid matching suffix and valid matching prefixes in the text for each N-factor. The formal algorithm, called PNS-Merge, is described in Algorithm 1 . It builds a min-heap H_N for all the N-factor lists, each of which is sorted in the ascending order. Let π_n be one element on the list of N. It pops π_n from the heap H_N and conduct a binary search for each suffix list specified by the largest element s_{max} ($< \pi_n + |N| - l_{min}$) to find a valid matching suffix starting at position s_{max} (lines 4 – 5). If such s_{max} exists, the algorithm PNS-Merge verifies the candidate occurrence $T[\pi_p, s_{max}]$ for each unprocessed element π_p on prefix lists that is not greater than s_{max}. When s_{max} could not be found, it removes all the elements that are not greater than π_n on the prefix lists since they are all starting positions of invalid matching prefixes (lines 6 – 13). The algorithm then pops the top element π_n from the heap H_N and repeats the above steps until the heap H_N is empty.

Algorithm 1: PNS-Merge

Input: An RE Q, Prefix lists $P_{set} = \{L_{P_1}, \ldots, L_{P_u}\}$, Suffix lists $S_{set} = \{L_{S_1}, \ldots, L_{S_v}\}$, Negative factor lists $L_{N_1}, \ldots, L_{N_w}$;

1 Calculate l_{min} given Q;
2 Insert the frontier records of $L_{N_1}, \ldots, L_{N_w}$ to a heap H_N;
3 **while** *H_N is not empty* **do**
4   Let π_n be the top element on H_N associated with an N-factor N;
5   $s_{max} \leftarrow$ FindMaxSuffix($S_{set}, \pi_n + |N| - l_{min}$);
6   **if** *s_{max} is found* **then**
7   Remove all elements that are less than s_{max} in suffix lists;
8   **foreach** $L_{P_i} (\in P_{set})$ **do**
9   **for** *element π_p ($\leq s_{max}$) in the list L_{P_i}* **do**
10   Verify($T[\pi_p, s_{max}]$);
11   Remove π_p from L_{P_i};
12   **else**
13   Remove all elements that are not greater than π_n in prefix lists;
14   Pop π_n from H_N;
15   Push next record (if any) on each popped list to H_N;
16 **if** *there exists a non-empty list in S_{set}* **then**
17   $s_{max} \leftarrow$ FindMaxSuffix($S_{set}, |T| - l_{min} + 1$);
18   Verify($T[\pi_p, s_{max}]$) for each π_p ($\leq s_{max}$) in prefix lists;

If there exist non-empty suffix lists, the algorithm finds the largest element s_{max} among them and verifies the candidate occurrence $T[\pi_p, s_{max}]$ for each remaining element $\pi_p (\leq s_{max})$ on the prefix lists (lines 16 – 18).

The algorithm requires $O(n_p + m_n \cdot \log l_n + m_s \cdot \log l_s)$ time to generate candidates, where n_p is the number of prefixes in T, m_n and m_s are the number of N-factors and suffixes, respectively, l_n and l_s are the average length of the inverted list of each N-factor and suffix, respectively. The average length of each verification has been reduced to $\frac{n}{m_n \cdot l_n}$ since N-factors could divide the text into $m_n \cdot l_n$ disjoint substrings. Therefore, the algorithm PNS-Merge only requires $O(\frac{m \cdot n}{m_n \cdot l_n} n_c)$ time to do verifications, where n_c is the number of candidates.[1]

4.2 Bit-Parallel Algorithms

In algorithm PNS-Merge, we need to find valid matching suffixes and their corresponding valid matching prefixes to construct candidate occurrences for verification. We can accelerate the process by introducing the bit-parallel operations. We propose two algorithms in Sections 4.2.1 and 4.2.2.

4.2.1 *The* PNS-BitC *Algorithm under the Constraint* $|N| \leq l_{min}$

We use a bit vector $N_s = d_0 \ldots d_n$ to represent all the occurrences of N-factors in the text T. We set $N_s[d_i] = 1$ ($d_0 \leq d_i < d_n$) if an N-factor starts at the position d_i in T. We set the last bit $N_s[d_n] = 1$ to mark the end of T. Consider our running example shown in Section 2. Given the RE Q = (G|T)A*GA*T* and the text shown in Figure 1, the third row in Table 1 shows an example N_s, which represents the occurrences of N-factors w.r.t. Q in the text T. Similarly,

[1]According to the analysis in Section 5.4 and experimental results in Section 6, n_c is much smaller than n_p.

Algorithm 2: PNS-BitC

```
   Input: Bit vectors P_s, S_s, N_s, l_min;
1  Calculate vectors S_sm and P_c using Equation 1;
2  t ← the position of last suffix using S_sm;
3  Change S_sm[t] from 1 to 0;   π_s ← t;   π_p ← 0;
4  repeat
5  |  t ← the position of last suffix using S_sm;
6  |  repeat
7  |  |  π_p ← the position of last prefix using P_c;
8  |  |  Change P_c[π_p] from 1 to 0;
9  |  |  Verify(T[π_p, π_s + l_min − 1]);
10 |  until π_p < t;
11 |  Change S_sm[t] from 1 to 0;   π_s ← t;
12 until there is no 1 in S_sm or P_c;
```

we use vectors P_s and S_s for the occurrences of prefixes and suffixes in T respectively, in which $P_s[d_i] = 1$ (or $S_s[d_i] = 1$) indicates a prefix (or suffix) starting at position d_i in T.

We first consider a simple case where the length of each N-factor is no greater than the length l_{min}. Under this constraint, it is obvious that the case shown in Figure 3(b) cannot happen, i.e., if positions π_p, π_n, and π_s satisfy the condition $\pi_p < \pi_n < \pi_s$, there exists a PNS pattern. Under this constraint, a valid matching suffix means $\pi_s < \pi_n$ since a matching suffix and a matching N-factor cannot start at the same position, which suggest we can just use N_s, P_s, and S_s to calculate candidate occurrences.

Algorithm 2 shows the algorithm PNS-BitC that generates a candidate occurrence using a valid matching suffix and a corresponding valid matching prefix to do verifications. In line 1, the algorithm calculates two vectors S_{s_m} and P_c using Equation 1. In S_{s_m}, a bit $S_{s_m}[d_i] = 1$ if d_i is the starting position of a valid matching suffix, otherwise $S_{s_m}[d_i] = 0$. In P_c, a bit $P_c[d_i] = 1$ if d_i is the starting position of a valid matching prefix. Otherwise $P_c[d_i] = 0$.

$$
\begin{array}{lrcl}
Intermediate: & A_0 & = & (N_s|S_s) - (N_s << 1), \\
 & S_{s_m} & = & (\sim A_0)\ \&\ S_s, \\
Intermediate: & B_0 & = & N_s - S_{s_m}, \\
 & P_c & = & P_s\ \&\ B_0.
\end{array} \tag{1}
$$

For illustration purposes, we use an intermediate vector A_0 to show how to calculate S_{s_m}. The bit $A_0[d_i]$ is set to 0 if d_i is the starting position of a valid matching suffix. For example, the bits $A_0[5]$, $A_0[14]$, and $A_0[18]$ in Table 1 are all 0, and they mark the three valid matching suffixes. If there is no matching suffix between any two adjacent matching N-factors starting at d_i and d_j, the algorithm PNS-BitC sets $A_0[d_i] = 0$ and $A_0[d_l] = 1$ ($d_i < d_l < d_j$). Then we calculate S_{s_m} using $(\sim A_0)\&S_s$.

Table 1: Generate candidate regions using bit operations under the constraint $|N| \le l_{min}$ ($l_{min} = 2$).

	Vector																					
	0					5					10					15					20	
P_s	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
S_s	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	1	0	0	0
N_s	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1
A_0	1	1	1	0	1	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
S_{s_m}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
B_0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1
P_c	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Correspondingly we use another intermediate vector B_0 to represent the starting positions of all the possible valid matching prefixes. Figure 5 shows an example of partial bits in the vector B_0. The "1" bits correspond to the starting positions of valid matching prefixes.

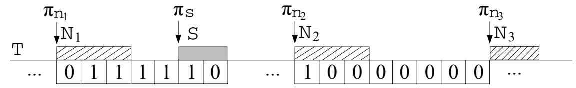


Figure 5: Explanation of the intermediate vector B_0.

The vector $P_c = P_s\ \&\ B_0$ denotes the starting positions of valid matching prefixes. Then the algorithm PNS-BitC gets each position pair π_p and π_s using bit operations (lines 2 and 7), which can be done in constant time.[2] It then verifies the corresponding candidate occurrence $T[\pi_p, \pi_s + l_{min} - 1]$ using P_c and S_{s_m} (line 9). For example, let $l_{min} = 2$. Table 1 shows the generated P_c and S_{s_m}. The candidate occurrences are $T[3, 6]$, $T[5, 6]$, and $T[10, 15]$, which are consistent with the candidate occurrences shown in Figure 4.

In general, the text string T can occupy the memory space of multiple words. The algorithm proceeds in the same fashion by computing S_{s_m} and P_c for all $\lceil \frac{n}{w} \rceil$ words, where w is the length of a word (e.g., 4 bytes). We only need to pay special attention to the case of processing (N_s<<1) and ($N_s - S_{S_m}$). For the current word N_s^{i-1}, when processing (N_s<<1), we need to maintain the first bit of its next word N_s^i and put it into the last bit of N_s^{i-1}. Similarly, the operation $N_s - S_{S_m}$ of the $(i-1)$st word is changed to $N_s^{i-1} - S_{S_m}^{i-1} - (N_s^i < S_{S_m}^i)$.

Let us analyze the space and time complexity of the algorithm. Let the word size be w. The algorithm requires $O(5\lceil \frac{n}{w} \rceil)$ to store the three input vectors and the two output vectors S_{S_m} and P_c. It takes $O(\lceil \frac{n}{w} \rceil)$ time to calculate S_{S_m} and P_c and $O(k_1 + k_2)$ time to generate candidate occurrences, where k_1 and k_2 are number of valid matching prefixes and valid matching suffixes, respectively. It requires the same verification time as the algorithm PNS-Merge.

4.2.2 *The* PNS-BitG *Algorithm without Constraints*

In Figure 3(b), the position relationship $\pi_p \le \pi_n < \pi_s$ does not guarantee a PNS pattern. It means that the comparison between the end positions of the N-factor N_2 and the suffix S_2 is also needed, i.e., both conditions $\pi_p \le \pi_n < \pi_s$ and $\pi_{n_e} \le \pi_{s_e}$ need to be satisfied to make sure that a PNS can be found, where π_{n_e} and π_{s_e} are the end positions of the N-factor N_2 and the suffix B_2.

We can get an end position π_{s_e} by shifting its corresponding starting position π_s to the right for $l_{min} - 1$ bits since we know the length of each suffix is l_{min}, i.e., the vector for the end positions of the occurrences of N-factors $S_e = S_s >> (l_{min} - 1)$. However, the length of N-factors could be different, so we have to use another vector N_e to represent the end positions of the occurrences of N-factors. Equation 2 shows the bit operations to calculate candidate occurrences.

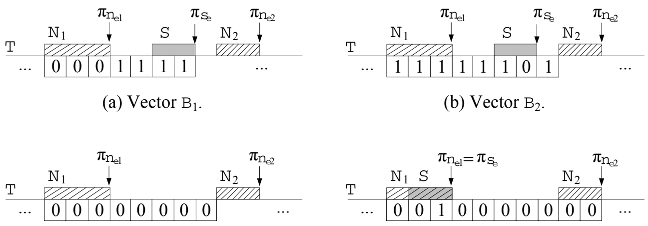
$$
\begin{array}{lrcl}
Intermediate: & A_1 & = & (N_e|S_e) - (N_e{<<}1), \\
 & S_{e_m} & = & (\sim A_1)\ \&\ S_e\ \&\ (\sim N_e), \\
Intermediate: & B_1 & = & (N_e - S_{e_m})\ \&\ (\sim N_e), \\
Intermediate: & B_2 & = & \sim ((S_{e_m}{<<}(l_{min}{-}1)){-}S_{e_m}), \\
Intermediate: & B_3 & = & (N_e\ \&\ A_1)\ |\ (N_e\ \&\ S_e), \\
Intermediate: & B_4 & = & (N_s - B_3)\ \&\ (\sim N_s), \\
Intermediate: & B_5 & = & (B_1\ \&\ B_2)\ |\ B_4, \\
 & P_c & = & P_s\ \&\ B_5.
\end{array} \tag{2}
$$

We consider the same example that matches the RE Q = `(G|T)A*GA*T*` on the text in Figure 1. Table 2 shows the

[2] http://graphics.stanford.edu/~seander/bithacks.html

input vectors and intermediate results of the bit operations in Equation 2.

Similar to the bit vector A_0 in Equation 1, in the intermediate bit vector A_1 in Equation 2, $A_1[d_i] = 0$ if d_i is the end position of a valid matching suffix. Then we can get the vector S_{e_m}, in which each bit 1 means an end position of a valid matching suffix. Notice that, different from S_{s_m} in Equation 1, the vector S_{e_m} needs to do an AND operation between $(\sim A_1)\&S_e$ and $(\sim N_e)$. Before we give the reason, let us consider the case where a matching suffix S has the same end position t with a matching N-factor N_1. According to the analysis in Section 4.1, this suffix S should not be used to generate candidate occurrences (see Figure 3(a)). In addition, if there is no suffix between the N-factor N_1 and its right neighbor N-factor N_2, then the bit $A_1[t]$ is 0. Therefore, the algorithm PNS-BitG may choose an invalid matching suffix setting $S_{e_m}[t] = 1$. In order to avoid choosing such invalid matching suffix S, we do the bit AND operation between $(\sim A_1)\&S_e$ and $(\sim N_e)$.



(a) Vector B_1. (b) Vector B_2. (c) Vector N_e& A_1. (d) Vector N_e& S_e.

Figure 6: Explanation of vectors.

For the purpose of calculating valid matching prefixes corresponding to each valid matching suffix, we use five intermediate vectors $B_1, \ldots, B_5$ to explain how the calculation works. Let S be the rightmost suffix between two N-factors N_1 and N_2. $B_1[d_i] = 1$ if $\pi_{n_{e1}} < d_i \leq \pi_{s_e}$. An example vector B_1 is shown in Figure 6(a). The vector B_2 sets all the bits to 1 except the bit $B_2[\pi_{s_e}]$, where π_{s_e} is the end position of a valid matching suffix. Figure 6(b) shows that $B_2[d_i] = 0$ $(\pi_{s_s} < d_i \leq \pi_{s_e})$, where π_{s_s} is the starting position of the valid matching suffix S.

Given two N-factors N_1 and N_2, if there is no suffix between them, we say N_1 is a *useless N-factor*. We use $B_3[\pi_{n_{e1}}] = 0$ to mark the useless N-factor N_1 and keep $B_3[\pi_{n_{e2}}] = 1$ at the same time. We use $N_e\&A_1$ to mark all the useless N-factors (see Figure 6(c)). However, a useless N-factor needs to be maintained if there is a suffix S happens ending at the same position with this N-factor. The reason is that a matching suffix $T[\pi_{s_s}, \pi_{s_e}]$ could be valid if there exists a prefix starting at position π_{p_s} and $\pi_{n_s} < \pi_{p_s} \leq \pi_{s_s}$, where π_{n_s} is the starting position of N_1 (see Figure 6(d)). We use $N_e\&S_e$ to specify all the same end positions of matching N-factors and valid matching suffixes. Furthermore we set each bit in $B_4[\pi_{n_s} + 1, \pi_{s_e}]$ to 1. Then we use the vector B_5 to combine all cases of possible valid matching prefixes. Finally, the vector P_c marks all valid matching prefixes.

Table 2 shows the generated S_{e_m} and P_c, and the candidate occurrences are consistent with those shown in Figure 4.

We call the corresponding algorithm PNS-BitG. After using Equation 2 to get the two vectors S_{e_m} and P_c, the algorithm gets each position pair π_s and π_p from S_{e_m} and P_c, respectively. It then generates candidates $T[\pi_p, \pi_s]$ to do verification. The algorithm PNS-BitG requires the same time complexity as the algorithm PNS-BitC, but needs $O(6\lceil \frac{n}{w} \rceil)$ to store one more vector N_e than the algorithm PNS-BitC.

Table 2: Generate candidate regions using bit operations without any constraint ($l_{min} = 2$).

	Vector																					
	0					5					10					15					20	
P_s	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
S_e	0	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	1	0	0
N_s	0	0	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	1	0	0	1
N_e	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	1	1
A_1	1	1	1	0	0	1	0	1	0	1	0	1	1	1	1	0	0	1	0	0	0	1
S_{e_m}	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
B_1	0	0	0	1	1	1	1	0	1	1	1	0	1	1	1	1	0	0	1	1	0	0
B_2	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1
B_3	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1
B_4	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
B_5	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0
P_c	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

4.3 Improving Pruning Power by Combining Negative Factors with Necessary Factors

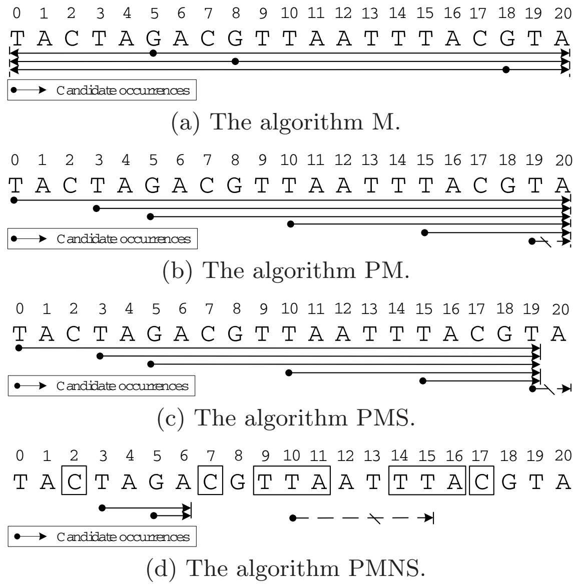
Besides prefixes and suffixes, there is another type of positive factors, called *necessary factor*. A necessary factor w.r.t. an RE Q is a substring that must appear in every matching substring in the text T. For instance, `G` is a necessary factor w.r.t. the RE Q = (`G|T`)`A`*`GA`*`T`*. GNU grep 2.0 [4] employs a different heuristic approach for finding necessary factors w.r.t. an RE Q. The neighborhoods of these necessary factors are then verified using a lazy deterministic automaton.

Generally, a necessary factor (substring) divides Q into a left part and a right part. Two automata are constructed for verification in both directions. Figure 7(a) shows an example of this approach, call M algorithm. Since `G` is a necessary factor, the algorithm M builds an automaton A_r for the right part of the RE Q, i.e., `GA`*`T`*, and another automaton A_l for the left part of Q, i.e., (`G|T`)`A`*`G`. It then runs A_r on the suffixes of T starting at positions 5, 8, and 18, and runs A_l on the prefixes starting at these positions.

Now we analyze the effect of introducing necessary factors into the algorithms P, PS, and PNS. We call the corresponding algorithms PM, PMS, and PMNS, respectively. (In the case where there are no necessary factors, the approaches based on necessary factors can not be used.) Generally, an occurrence of Q in the text T must contain at least one necessary factor. Otherwise, the corresponding candidate can be pruned without verification. Now we show an interesting observation that using both negative factors and necessary factors together (i.e., algorithm PMNS) can enhance the filtering power significantly.

Figures 7(b) and 7(c) show the examples of introducing necessary factors into the algorithms P and PS, respectively. In Figures 7(b), the algorithm PM can prune the candidate $T[19, 20]$ since it does not contain a necessary factor. However, in Figure 7(c), the algorithm PMS cannot prune any more candidates because the necessary factor `G` at position 18 appears in the last matching suffix $T[18, 19]$ = `GT`. Generally, it has a high probability that a necessary factor appears in the late part of T, that is, each candidate occurrence might have a high probability to contain this necessary factor and could not be pruned.

However, the algorithm PNS generates candidates in a relatively short interval, in which at least one necessary factor is expected to be found, otherwise it must not be an occurrence. For example, the substring $T[10, 15]$ shown in



(a) The algorithm M.

(b) The algorithm PM.

(c) The algorithm PMS.

(d) The algorithm PMNS.

Figure 7: Checking candidate occurrences of the RE $Q = (\mathtt{G|T})\mathtt{A}^*\mathtt{GA}^*\mathtt{T}^*$ using necessary factors, which are shown in bold font.

Figure 7(d) is unnecessary to be verified using the algorithm PMNS since it does not contain any necessary factor.

We can design algorithms using the ideas of PNS-Merge and bit operations to combine necessary factors with PNS patterns. Due to space limitation we do not give the detail of these algorithms. In Section 6 we show our experimental results of the algorithm PMNS by using bit operations with constraints $|N| \leq l_{min}$, which is called PMNS-BitC.

5. CHOOSING GOOD N-FACTORS

The number of N-factors w.r.t. an RE could be large, and different N-factors could have different improvements on the matching performance of algorithms based on PNS patterns. For example, given an RE $Q = \mathtt{C}^*\mathtt{AAA}$, both `G` and `CGA` are N-factors w.r.t. Q. It is obvious that any occurrence of `CGA` in a text T also contains an occurrence of `G` in T. Thus the N-factor `CGA` does not provide more filtering power than `G`. So a natural question is how to choose a small number of high-quality N-factors to maximize search performance. In this section we develop techniques to solve this problem. In Section 5.1 we show how to define a finite set of high-quality N-factors, called "core N-factors," and show an upper bound on the number of core N-factors w.r.t. an RE Q. Then in Section 5.2 we describe the challenge of efficiently constructing the core N-factors, and propose an efficient algorithm for constructing core N-factors. In Section 5.3 we develop a technique to speed up the generation of core N-factors by doing early termination.

5.1 Core N-Factors

Definition 2. (Core N-factor). An N-factor w.r.t. an RE Q is called a *core N-factor* if each of its proper subsequences is not an N-factor w.r.t. Q.[3]

For example, for the RE $Q = \mathtt{C}^*\mathtt{AAA}$, the set of core N-factors w.r.t. Q is {`G`, `T`, `AC`, `AAAA`}. The substring `GA` is not a core N-factor since its subsequence `G` is an N-factor.

In order to compute an upper bound on the number of core N-factors w.r.t. an RE Q, we use a *factor automaton* [2] to check whether a string is an N-factor. A factor automaton is an automaton representing the set of all positive factors w.r.t. Q. For a given RE Q, we first construct a nondeterministic factor automaton A_f. Then A_f can be further transformed to a unique minimal deterministic factor automaton A_{f_m}, which accepts exactly the same set of strings [5]. Using A_{f_m} we can prove an upper bound on the number of core N-factors.

Theorem 2. *Let Q be an RE and A_{f_m} be its minimized deterministic factor automaton. The length of a core N-factor w.r.t. Q cannot be greater than the number of states in the longest acyclic path of A_{f_m}.*

Lemma 2. *The number of core N-factors w.r.t. an RE Q is upper bounded by $|Q|^2$.*

Proof. An upper bound on the number of states in the longest acyclic path of the minimized deterministic factor automaton is the square of the number of characters that Q contains, i.e., $|Q|^2$. Therefore, the length of any core N-factor should be no greater than $|Q|^2$. □

5.2 Constructing Core N-Factors Online

One naive way to construct core N-factors is to enumerate strings with a length $\leq |Q|^2$ and check if they are core N-factors one by one (see Algorithm 3). The check process of a string s includes two phases: (i) the function CheckSubSequence checks if a subsequence of s is an N-factor (line 4), and (ii) otherwise, check if s itself is an N-factor, which can be determined by running the factor automaton $A_f(Q)$ (line 6).

Algorithm 3: NaiveCore

Input: Alphabet Σ, an RE Q, a factor automaton $A_f(Q)$;
Output: A set of core N-factors C_N;
1 $C_N \leftarrow \emptyset$;
2 **for** *length* $l \leftarrow 1; l \leq |Q|^2; l++$ **do**
3   **for** *each string* $s \in \Sigma^l$ **do**
4   FOUND $\leftarrow$ CheckSubSequence(C_N, s);
5   **if** *FOUND is false* **then**
6   **if** *s cannot be accepted by $A_f(Q)$* **then**
7                 $C_N \leftarrow C_N \cup \{s\}$;
8 return C_N;

It can be time consuming to enumerate all the strings with a length $\leq |Q|^2$ and check each of them, especially when we have to do it for each search query. For instance, we need to enumerate 340 strings within 16 iterations for the RE $Q = \mathtt{C}^*\mathtt{AAA}$. Next we present a more efficient method to construct core N-factors to reduce the computational cost.

Instead of enumerating all the strings (see line 3 in Algorithm 3), we can use two properties of core N-factors to generate them with a length l using a smaller set of strings with length $l - 1$. Based on Definition 2, we show the properties and how to utilize them to improve the performance.

[3]We distinguish substring and subsequence in this paper. A substring of a string s has consecutive characters of s, while the characters in a subsequence may not be consecutive in s.

(1) *Generate core N-factors using substrings w.r.t. an RE Q.*

Property 1. If a string x is a core N-factor, then its prefix $x[0, |x| - 2]$ and suffix $x[1, |x| - 1]$ can be accepted by the factor automaton $A_f(Q)$.

This property helps us improve the performance of computing N-factors by "joining" a set of strings with length $l-1$ to generate strings with length l. The formal definition of *string-join* is given below.

Definition 3. (String-join) Given two strings s_1 and s_2, the *string-join* of s_1 and s_2, denoted by $\widehat{s_1 s_2}$, is computed as follows. If $|s_1| = |s_2| = l - 1$ and $s_1[1, l-2] = s_2[0, l-3]$, then $\widehat{s_1 s_2}[0, l-2] = s_1$ and $\widehat{s_1 s_2}[l-1] = s_2[l-2]$; otherwise, $\widehat{s_1 s_2} = \emptyset$.

Definition 4. (String set self-join) Let $S = \{s_1, \ldots, s_k\}$ be a set of strings of length $l - 1$. The *string set self-join* of S, denoted by $\widehat{S}$, is a set of non-empty strings $\widehat{s_i s_j}$ $(1 \leq i, j \leq k)$.

For example, let $S = \{s_1, s_2\}$, $s_1 =$ `ACG` and $s_2 =$ `CGT`, then $\widehat{S} = \{$`ACGT`$\}$.

Compared with Algorithm 3, we can get the same set of core N-factors with length l by self-joining the set of strings with length $l-1$, each of which can be accepted by the factor automaton $A_f(Q)$.

Let S' be the set of strings that can be accepted by $A_f(Q)$, where each string in S' has a length $l-1$, and $S = \widehat{S'}$. We use $S_Q(\Sigma^l) \subseteq \Sigma^l$ and $S_Q(S) \subseteq S$ to represent strings that can be accepted by $A_f(Q)$, and use $S_C(\Sigma^l) \subseteq \Sigma^l$ and $S_C(S) \subseteq S$ to represent the core N-factors.

THEOREM 3. *The two sets $S_Q(\Sigma^l)$ and $S_Q(S)$ are equivalent, and the two sets $S_C(\Sigma^l)$ and $S_C(S)$ are also equivalent.*

(2) *Avoid invoking* CHECKSUBSEQUENCE *for every generated string.*

Property 2. Given a core N-factor x with a length greater than 1, let s be a string whose prefix $s[0, |s| - 2]$ and suffix $s[1, |s| - 1]$ can be accepted by $A_f(Q)$. If x is subsequence of s, then $x[0] = s[0]$ and $x[|x| - 1] = s[|s| - 1]$.

Let $|x| = m$ $(1 < m \leq n)$. We construct string s as follows. Suppose S' is a set of strings with length $l - 1$ that can be accepted by $A_f(Q)$. Let $S = (\widehat{S'})$ and $s \in S$. We know there must exist two strings $s_1, s_2 \in S'$, such that $s_1 = s[0, l-2]$ and $s_2 = s[1, l-1]$. Assuming $x[0] \neq s[0]$, since x is a subsequence of a string $s \in S$, we know x should be a subsequence of $s[1, l-1]$, i.e., x is a subsequence of s_2. It contradicts to the setting that any core N-factor is not a subsequence of a string in S. Therefore, $x[0]$ must be equivalent to $s[0]$. Similar, $x[m-1] = s[l-1]$.

Algorithm 4 is an improved algorithm by considering the above two properties in Algorithm 3. According to Property 1, the algorithm QUICKCORE in Algorithm 4 initially generates strings with length 1 and maintains those that can be accepted by the factor automaton $A_f(Q)$. Through self-joining the retained strings with length $l - 1$, the algorithm generates the strings with one more character (lines 3 – 16).

Furthermore, when there exists a core N-factor x that does not satisfy $s[0] = x[0]$ and $s[l-1], x[|x|-1]$, then we do

Algorithm 4: QUICKCORE

Input: Alphabet Σ, an RE Q, a factor automaton $A_f(Q)$;
Output: A set of core N-factors;
1 $C_N \leftarrow \emptyset$; $S \leftarrow \Sigma$;
2 Create a hash table H_T;
3 **for** *length* $l \leftarrow 1; l \leq |Q|^2; l++$ **do**
4   **for** *each string* $s \in S$ **do**
5   FOUND $\leftarrow$ false;
6   **if** $l > 2$ **then**
7   New a string y, $y[0] \leftarrow s[0]$ and $y[1] \leftarrow s[l-1]$;
8   **if** y *is in* H_T **then**
9   FOUND $\leftarrow$ CHECKSUBSEQUENCE(C_N, s);
10   **if** *FOUND is false* **then**
11   **if** s *cannot be accepted by* $A_f(Q)$ **then**
12         $C_N \leftarrow C_N \cup \{s\}$; $S \leftarrow S - \{s\}$;
13   Insert y into H_T, where $y[0] \leftarrow s[0]$ and $y[1] \leftarrow s[l-1]$;
14   **else**
15       $S \leftarrow S - \{s\}$;
16   $S \leftarrow \widehat{S}$;
17 return C_N;

not need to invoke the function CHECKSUBSEQUENCE based on Property 2 (lines 7 – 9) . In the above example for Q = `C`*`AAA`, among the 65 strings, only 8 strings need to be further checked using the function CHECKSUBSEQUENCE.

5.3 Early Termination of Constructing Core N-Factors

We observe that the upper bound $|Q|^2$ on the length of an N-factor in Algorithm 4 is loose, which can result in many iterations in the algorithm (see line 2). As we can see in Table 3, all core N-factors have been generated before the fifth iteration. However, the absence of incremental core N-factors at an iteration cannot guarantee that there will not be any more core N-factors generated in the following iterations, i.e., a new core N-factor can still be generated even though nothing is produced in the previous iterations. So the question is whether there exists a tighter upper bound on the length using which we can terminate the iterations early. Next we show such a tighter bound.

Table 3: Early termination for Q = `C`*`AAA`, where the underlined strings are core N-factors.

Iteration	Processing strings using Algorithm 4	Processing strings using Algorithm 5
1	A, C, $\underline{\text{T}}$, $\underline{\text{G}}$	A, C, $\underline{\text{T}}$, $\underline{\text{G}}$
2	AA, CA, CC, $\underline{\text{AC}}$	AA, CA, $\underline{\text{AC}}$
3	AAA, CAA, CCA, CCC	AAA, CAA
4	$\underline{\text{AAAA}}$, CAAA, CCAA, CCCA, CCCC	$\underline{\text{AAAA}}$, CAAA
5	CCAAA, CCCAA, CCCCA, CCCCC	$\emptyset$
6	CCCAAA, CCCCAA, CCCCCA, CCCCCC	
...	...	
16	...	

Observation 1. Let C be the set of characters in e^*, and k be the summation of frequencies of characters of C in Q. For any string s that matches the pattern e^* and $|s| > k$, we cannot find any string s' to make $\widehat{ss'}$ an N-factor.

As we can see in the second column in Table 3, Algorithm 4 keeps generating strings and checks if some of them are N-factors. In the second iteration, the string CC is used to generate strings CCA and CCC in the third iteration, and generate string CCAA, CCCA and CCCC in later iterations. As we know, CC is not an N-factor because C* exists in the RE Q. Any generated string based on CC cannot be an N-factor, since multiple Cs can always be accepted by the factor automaton $A_f(Q)$. Therefore, it is not needed to keep CC in the second iteration.

Based on this observation in Algorithm 5, we propose the algorithm EarlyCore to do early termination in the construction of core N-factors without any false dismissal. The algorithm first counts the frequency of each character in each Kleene closure e^* in Q (lines 2 – 7) and then uses it to check if a generated string needs to be reserved for self-joining in the next iteration (line 12).

Algorithm 5: EarlyCore

Input: Alphabet Σ, an RE Q, a factor automaton $A_f(Q)$;
Output: A set of core N-factors;
1 $C_N \leftarrow \emptyset$; $S \leftarrow \Sigma$; $C_S \leftarrow \emptyset$;
2 **for** *each expression* e^* *in* Q **do**
3   Let $C \leftarrow$ the set of characters in e;
4   Let $freq(C) \leftarrow 0$;
5   **for** *each character* c *in* e **do**
6     $freq(C) \leftarrow freq(C)$ + number of c in Q;
7   Insert C into C_S;
8 $l \leftarrow 1$;
9 **while** S *is not empty* **do**
10   **for** *each string* $s \in S$ **do**
11   **if** $\exists C \in C_S$ *and* $s \in C^l$ **then**
12   **if** $l = freq(C) + 1$ **then** // bound of C
13         $S \leftarrow S - \{s\}$;
14   ... // see lines 5 - 15 in Algorithm 4
15   $S \leftarrow \widehat{S}$; $l \leftarrow l + 1$;
16 **return** C_N;

Table 3 shows the number of iterations using Algorithm 4 and Algorithm 5, respectively. Algorithm 5 only needs 4 iterations compared with the 16 iterations in Algorithm 4.

Theorem 4. *The set S in Algorithm 5 always converges to an empty set.*

5.4 Pruning Power of N-factors

In this section, we analyze the pruning power of N-factors. We first present a theoretical analysis, then give an experimental result on a real data set.

Consider a substring $T[a, d]$ conforming to a PNS pattern, in which the substring $T[a, b]$ is a matching prefix and the substring $T[c, d]$ is a matching suffix ($a \le b \le c \le d$). Let $p_1(n)$ denote the probability that the length of $T[b, c]$ is equal to n and $p_2(n)$ denote the probability that there exists at least one N-factor matching in $T[b, c]$. Then the probability of filtering any prefix matching in T using N-factors can be calculated as follows:

$$p_f = \sum_{n=0}^{|T|-2\cdot l_{min}} p_1(n) \times p_2(n). \qquad (3)$$

We first calculate $p_1(n)$. Let $S = \{S_1, \ldots, S_h\}$ be the set of suffixes w.r.t. the RE Q. As defined in [14], each suffix in S has the same length l_{min}. Let $B_n(h, l_{min})$ denote the number of substrings $T[b, c]$ with a length n such that any suffix in S is not a substring of $T[b, c]$. Then we could get the following recurrence function for $n \ge 0$:

$$B_n(h, l_{min}) = \begin{cases} |\Sigma|^n & \text{if } n < l_{min}, \\ |\Sigma|\cdot B_{n-1}(h, l_{min}) - h\cdot B_{n-l_{min}}(h, l_{min}) & \text{otherwise.} \end{cases}$$

Then we have

$$p_1(n) = \frac{B_n(h, l_{min})}{|\Sigma|^n} \times \frac{h}{|\Sigma|^{l_{min}}}. \qquad (4)$$

Similarly, let $N = \{N_1, \ldots, N_k\}$ be the set of N-factors w.r.t. the RE Q and each N-factor in N has a length l'. Then we have

$$p_2(n) = 1 - \frac{B_n(k, l')}{|\Sigma|^n}. \qquad (5)$$

6. EXPERIMENTS

In this section, we present experimental results of the N-factor technique on multiple real data sets.

Experiment Setup. We conducted the experiments on three public data sets, including Human Genome, Protein sequences, and English texts.

- Human Genome: The genomic sequence (GRCh37) was assembled from a collection of DNA sequences, which consisted of 24 chromosomes with a length varying from 48 million to 249 million.[4]

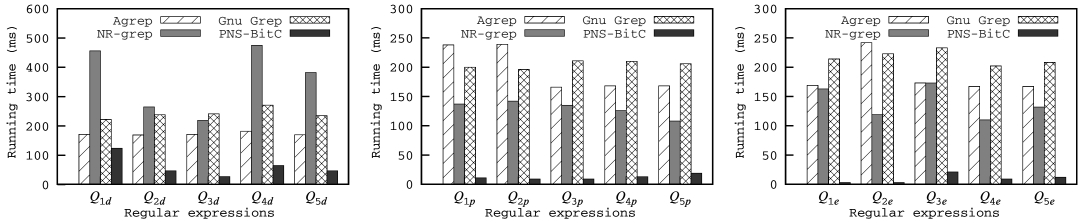
- Protein sequences: We adopted the database Pfam 26.0 that contains a large amount of protein families and is composed of Pfam-A and Pfam-B.[5] The symbol set consists of all the capital English letters, excluding "O" and "J". We randomly picked text with a length varying from 101 to 9143 from Pfam-B.

- English texts: We used DBLP-Citation-network[6], which included 1,632,442 blocks, each of which corresponds to one paper. Each block contains several attributes of a paper, e.g., title, authors, abstract, etc. We extracted the abstract from every block. The symbol set consisted of 52 English letters, plus 10 digits.

We extracted several subsequences of length ranging from 10 million to 100 million from the Human Genome. For the other two data sets, the size of each data set varied from 10MB to 100MB. Since there is no "random" regular expression [12], we manually synthesized REs to cover different properties of the regular expression (see Table 4). The size of these REs was from 6 to 21 and l_{min} varied from 2 to 5. We ran each algorithm using the corresponding REs on different data sets. These REs were used as queries to compare the performance of different algorithms.

All the algorithms were implemented using GNU C++. The experiments were run on a PC with an Intel 3.10GHz Quad Core CPU i5 and 8GB memory with a 500GB disk, running a Ubuntu (Linux) 64-bit operating system. All index structures were in memory.

[4]http://hgdownload.cse.ucsc.edu/goldenPath/hg18
[5]ftp://ftp.sanger.ac.uk/pub/databases/Pfam/releases
[6]http://arnetminer.org/DBLP_Citation



(a) DNA sequences. (b) Protein sequences. (c) English text.

Figure 8: Performance comparison of different algorithms.

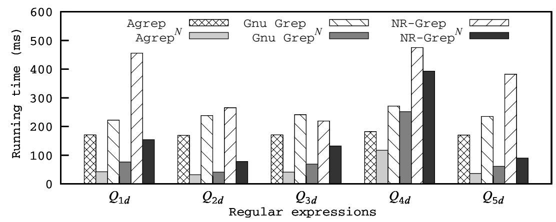
Table 4: Synthesized regular expressions.

<table>
<tr><th>Data sets</th><th>Regular expressions Q</th></tr>
<tr><td>DNA</td><td>Q_{1_d} = C(TA)*(G)(TT)*(A|G)
Q_{2_d} = (CT|GT|AT)(CT)(A|T)*A
Q_{3_d} = ((TA)|(GG))*(GA)(AT|GC)
Q_{4_d} = (TG)*(C|A)(C)(GA)*(T)
Q_{5_d} = (TG)*(C|A)(C)GA*T</td></tr>
<tr><td>Protein</td><td>Q_{1_p} = (VYL|VAP|DD|LR)(PST)*(TT|FA|EMLA)
Q_{2_p} = (EV|NV|SS|PQ)(LSI)*(VR|SR|SV|VI)
Q_{3_p} = (EL)*(VR)(L|E)*G
Q_{4_p} = (LQ|LA)A*(V|L)
Q_{5_p} = (SL|LA)(A|L)*(EL)(S|L)*</td></tr>
<tr><td>English text</td><td>Q_{1_e} = (e|a|i)*(re|to)s
Q_{2_e} =(this|This|That|that)*(is)(c|d|e|f|g)
Q_{3_e} = (Auto|auto)*(ma|ta)
Q_{4_e} = S(e|a|i)*st
Q_{5_e} = (pa)t*(er|n)</td></tr>
</table>

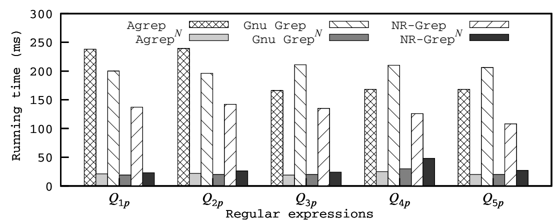
Comparison of RE Matching Algorithms. Recall that the existing algorithms Agrep, Gnu Grep, and NR-grep are developed for matching REs on a set of short sequences. When a sequence contains more than one occurrences of a query, only the first occurrence of the query is returned. For the purpose of comparability, we modified the source code of them so that they can find all the occurrences as our algorithms do.

Figure 8 shows the performance comparison of Agrep, Gnu Grep, NR-grep, and our algorithms on the three data sets. Each data set contained sequences with length 50 million. The algorithm PNS-BitC achieved the best time performance. For instance, when querying Q_{3_d} on DNAs, PNS-BitC took 27ms only, compared to the 171ms, 241ms, and 219ms using Agrep, Gnu Grep, and NR-grep, respectively. The superiority was even more evident on English texts, where PNS-BitC was 74 times faster than the popular Gnu Grep tool, e.g., 3ms versus 242ms, 223ms, and 119ms for query Q_{2_e}. The difference was due to the fact that the pruning power of N-factors increased as the size of Σ increased.

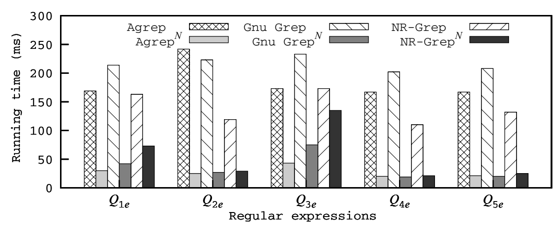
Improving Existing Algorithms Using N-Factors. To evaluate the benefits of N-factors, we modified the three existing algorithms to utilize the N-factors. Figure 9 shows the benefits of N-factors. Each superscript N means that the algorithm was using N-factors. It can be seen that the modified algorithms achieved a better performance than the original algorithms. For instance, for the DNA data set, the N-factor improved the performance of the algorithms by about 3 times. Figure 9(a) shows that for the query Q_{1_d}, Agrep^N reduced the time of Agrep from 171ms to 42ms, Gnu Grep^N reduced the time of Gnu Grep from 222ms to 76ms, and NR-grep^N reduced the time of NR-grep from 456ms to 154ms. Figure 9(b) shows that Agrep and Gnu Grep were improved by more than 10 times when using N-factors for queries Q_{1_p} and Q_{2_p}.



(a) DNA sequences.



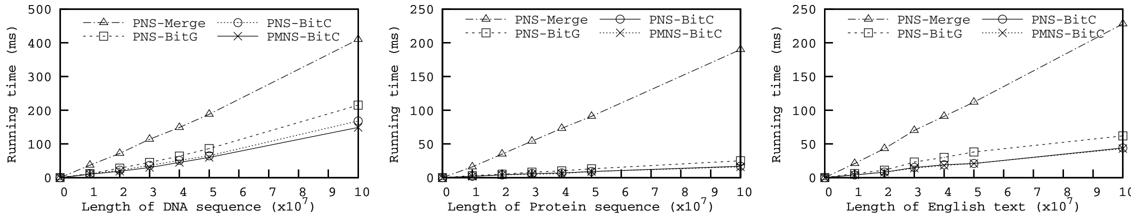
(b) Protein sequences.



(c) English texts.

Figure 9: Improving existing approaches by using N-factors.

Scalability of Using N-Factors. Figure 10 shows the slowly increased running time when we increased the length of the sequence for different algorithms of using N-factors. We can see that the bit-parallel algorithms performed much better than the algorithm PNS-Merge. The algorithm PMNS-BitC was the most efficient one. For instance, when the length of the sequence was 100 million, the running time were 149ms, 168ms, and 215ms, respectively, for DNA sequences, the running time were 16ms, 17ms, and 25ms, respectively, for protein sequences, and the running time were 43ms, 44ms, and 62ms, respectively, for English texts.



(a) DNA sequences. (b) Protein sequences. (c) English texts.

Figure 10: Scalability of approaches using N-factors.

Pruning Power of N-Factors. In Section 5.4 we analyzed the pruning power of N-factors. In order to calculate the probability value p_f in Equation 3 to see the pruning power of using N-factors for DNA sequences, we used three regular-expression workloads with query lengths 8, 12, and 15, respectively, each of which contained 100 REs, and $l_{min} = 5$. Each RE in a workload generated different numbers of suffixes and N-factors with different lengths. We ran the REs in each workload on DNA sequences with 50 million characters and calculated the average probability p_f. Figure 11 shows the pruning power of using N-factors. The dotted lines represent the computed values p_f, which ranged from 70.071% to 73.781% in the three workloads.

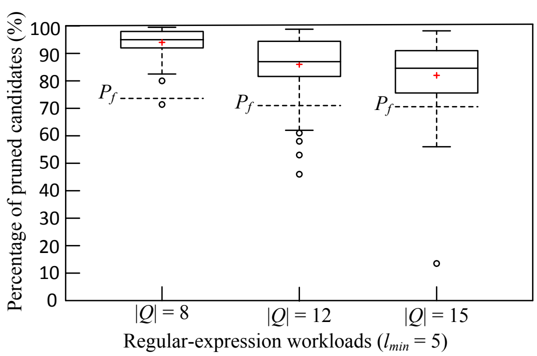


Figure 11: The ability of pruning false negatives using N-factors on DNA sequences.

Each box region in the figure represents the pruning power in the range between the first quartile (the top 25% REs) and the third quartile (the top 75% REs). The line in the box is the median value of the pruning power. The plus sign and the circles indicate the mean value of the pruning power and the outliers respectively. For the workload of $|Q| = 8$, the top 25% REs in the workload could prune 98.331% of the false negatives (i.e., substrings that do not need to be verified) and the top 75% REs could prune 92.008% of the false negatives. The other two workloads also provided significantly high pruning power. As we can see, the value p_f for each workload was lower than the experimental mean value. The reason is that our analysis is based on the assumption that data is evenly distributed, which may not be true in real data sets.

We got similar results for proteins and English texts (For space reason, we do not show the details.). The average probability value p_f for protein sequences was 72.115% when $|Q| = 19$ and $l_{min} = 4$, and the average probability value p_f for English texts was 61.870% when $|Q| = 23$ and $l_{min} = 4$.

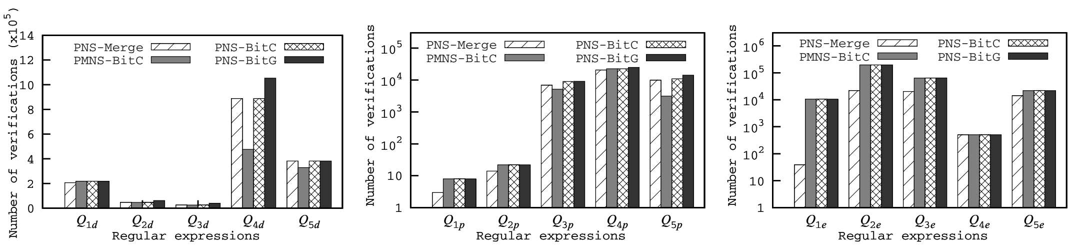
We then tested the number of verifications when using different N-factor-based algorithms on the three data sets of size 100MB. As we can see from Figure 12, the algorithm PMNS-BitC, which considers the necessary factors, required fewer verifications in most cases. For example, in Figure 12(a), when the query was Q_{4_d}, the number of verifications using PMNS-BitC was $475,926$, about the half of the number of the other algorithms. However, in most cases, the filtering advantage of PMNS-BitC was not evident. For example, in Figure 12(c), the number of verifications of algorithm PMNS-BitC was $63,861$, which was similar to the number of algorithms PNS-BitC and PNS-BitG, and was even higher than that of PNS-Merge.

Construction of N-Factors. We compared the construction time of N-factors using different N-factor construction algorithms and give the experiment results in Table 5. We did not use the algorithm NaiveCore in the experiments due to its very poor performance. The time of the algorithm QuickCore was not stable, and it varied from 0.05ms to 648.08ms. The reason was that the number of sequences to be processed could grow exponentially if there exists a Kleene closure of size one. The algorithm EarlyCore was much more efficient than QuickCore. It was stable since it avoids generating N-factors due to Kleene closure. In the best situation, it only took 0.05ms to construct the N-factors, and the worst construction time was just 1.66ms.

7. RELATED WORK

Traditional techniques of finding occurrences of an RE Q of length m in a text T of length n is to convert Q to an automaton and run it from the beginning of T. An occurrence will be reported whenever a final state of the automaton is reached [1, 4, 8].

Recent techniques utilize positive factors of the RE Q to improve the traditional automata-based techniques. For instance, MutliStringRE [14] uses prefixes, Gnu Grep [4] uses necessary factors, and NR-grep [9, 11] uses reversed prefixes to identify initial matchings and then verify them using the automata (see Section 2.2 for details). All these approaches support finding initial matchings of positive factors. MutliStringRE and Gnu Grep use Shift-And algorithm [15], and NR-grep uses BNDM [10], where BNDM is a bit-parallel implementation of the reverse automaton. The above approaches cannot be directly used to find all occurrences of an RE in a long text (sequence), since they are mainly developed for identifying the sequences that contain at least one matching of the RE Q among a set of sequences. Agrep [15] is a different approach that supports approximate matching of regular expressions within a specified search region and



(a) DNA sequences. (b) Protein sequences. (c) English texts.

Figure 12: Comparison of verification numbers.

Table 5: Time for constructing Core N-factors.

Data set: DNA			Data set: Protein			Data set: English text		
REs	QuickCore	EarlyCore	REs	QuickCore	EarlyCore	REs	QuickCore	EarlyCore
Q_{1_d}	18.74ms	0.15ms	Q_{1_p}	2.02ms	1.99ms	Q_{1_e}	528.99ms	0.63ms
Q_{2_d}	648.08ms	0.09ms	Q_{2_p}	0.87ms	0.86ms	Q_{2_e}	1.71ms	1.66ms
Q_{3_d}	0.17ms	0.11ms	Q_{3_p}	13.37ms	0.16ms	Q_{3_e}	0.45ms	0.43ms
Q_{4_d}	0.05ms	0.05ms	Q_{4_p}	0.17ms	0.12ms	Q_{4_e}	215.80ms	0.37ms
Q_{5_d}	14.89ms	0.19ms	Q_{5_p}	429.48ms	1.01ms	Q_{5_e}	0.65ms	0.44ms

returns matching occurrences exactly. Compared to these existing approaches, our main contribution is to use negative factors to improve the matching performance.

8. CONCLUSION

In this paper, we proposed a novel technique called N-factor and developed algorithms to improve the performance of matching a regular expression to a sequence. We gave a full specification of this technique, and conducted experiments to compare the performance between our algorithms and existing algorithms,such as Agrep, Gnu grep, and NR-grep. The experimental results demonstrated the superiority of our algorithms. We also extended Agrep, Gnu grep, and NR-grep with the N-factor technique, and showed great performance improvement.

9. ACKNOWLEDGMENTS

The work is partially supported by the National Basic Research Program of China (973 Program) (No. 2012CB316201), the National NSF of China (Nos. 60973018, 61272178), the Joint Research Fund for Overseas Natural Science of China (No. 61129002), the Doctoral Fund of Ministry of Education of China (No. 20110042110028), the National Natural Science of China Key Program (No. 60933001), the National Natural Science Foundation for Distinguished Young Scholars (No. 61025007), and the Fundamental Research Funds for the Central Universities (No. N110804002).

10. REFERENCES

[1] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915 – 936, 1996.

[2] M. Šimánek. The factor automaton. *Kybernetika*, 38(1):105 – 111, 2002.

[3] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two strings matching algorithms. *Algorithmica*, 12(4/5):247 – 267, 1994.

[4] GNUgrep. ftp://reality.sgiweb.org/freeware/relnotes/fw-5.3/fw_gnugrep/gnugrep.html.

[5] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[6] L. F. Kolakowski, J. Leunissen, and J. E. Smith. Prosearch: Fast searching of protein sequences with regular expression patterns related to protein structure and function. *Biotechniques*, 13:919 – 921, 1992.

[7] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

[8] M. Mohri. String matching with automata. *Nordic Journal of Computing*, 4(2):217 – 231, 1997.

[9] C. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265 – 1312, 2001.

[10] C. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5:4, 2000.

[11] C. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proceedings of WAE'01, Lecture Notes in Computer Science 2141*, pages 1 – 12, 2001.

[12] C. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89 – 116, 2004.

[13] R. Staden. Screening protein and nucleic acid sequences against libraries of patterns. *J. DNA Sequencing Mapping*, 1:369 – 374, 1991.

[14] B. W. Watson. A new regula grammar pattern matching algorithm. In *Proceedings of the 4th Annual European Sysmposium, Lecture Notes in Computer Science 1136*, pages 364 – 377. Springer-Verlag, 1996.

[15] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83 – 91, 1992.