MAINTAINING SOFTWARE THROUGH BIT-PARALLELISM AND HASHING
THE PARAMETERIZED Q-GRAMS

Rajesh Prasad, Suneeta Agarwal, Sanjay Misra, Anuj Kumar Sharma, Alok Singh

In the software maintenance, it is often required to find duplicity present in the codes. Two code fragments are equivalent, if one can be transformed into the other via consistent renaming of identifiers, literals and variables. This equivalency can be detected by parameterized string matching. In this matching, a given pattern \( P \) is said to match with a substring \( t \) of the text \( T \), if there exists a one-to-one correspondence between symbols of \( P \) and symbols of \( t \). In this paper, we propose an efficient algorithm for this problem by using both the overlapping and non-overlapping \( q \)-gram. We show the effect of running time of the algorithm on increasing the duplicity present in the code.

**Keywords:** bit-parallelism, design of algorithm, hashing, plagiarism detection, \( q \)-gram, software maintenance, string matching

1 Introduction

Exact string matching problem is to find all the occurrences of a given pattern \( P[0…m-1] \) in the text \( T[0…n-1] \), where symbols of \( P \) and \( T \) are drawn from some finite alphabet of size. See [1, 4, 6, 8, 10, 11, 12, 14] for detail. Application area of string matching includes: computational biology, information retrieval, text editor, software maintenance, etc. In the software maintenance [2, 3], it is often required to find duplicity present in the codes. Two code fragments are equivalent, if one can be transformed into the other via consistency renaming of identifiers, literals and variables. This equivalency can be detected by parameterized string matching [2, 3]. In the parameterized string matching, we are given two different alphabets: \( \Sigma \) for fixed symbol alphabet and \( \Pi \) for parameterized alphabet. Symbols from \( \Sigma \) need not to be renamed, whereas symbols from \( \Pi \) may be renamed. A given pattern \( P \) is said to match with a substring \( t \) of the text \( T \), if there exist a one-to-one correspondence between symbols of \( P \) and symbols of \( t \). Many string matching rely on fairly large alphabet for good performance [15]. To make alphabet larger, the concept of \( q \)-gram [15] has been proposed. There are two ways to form \( q \)-grams: one is overlapping \( q \)-gram and the other is non-overlapping \( q \)-gram. In parameterized string matching, the word “abed” is transformed to “ab-cd” and in non-overlapping \( q \)-gram it is transformed to “ab-cd”.

In [1], a bit-parallel algorithm (Shift-or) for solving exact string matching has been presented. This algorithm runs in time \( O(n) \), when \( m \leq w \), where \( w \) is word length of computer used. In [16], this algorithm was further speeded-up by a factor of \( q' \), where \( q' \) is size of non-overlapping \( q \)-gram (also known as super alphabet). In [5], shift-or algorithm has been extended for parameterized string matching, which in [17] was further speeded-up by a factor of \( q' \) by using the concept of non-overlapping \( q \)-gram. In [18], an efficient algorithm using hashing the overlapping \( q \)-gram for solving exact string matching has been presented. This algorithm uses the concept of loop unrolling to speed-up the algorithm. In [9], Horspool algorithm was extended for parameterized string matching, which uses the concept of \( q \)-gram and only parameterized alphabet is used.

In this paper, we propose an efficient algorithm (FASTQGRAM) for the parameterized string matching problem by using both the overlapping and non-overlapping \( q \)-grams. We show the effect on running time of the algorithm on increasing the duplicity present in the code.

The article is organized as follows. In Section 2, we present the related concept and algorithm for exact and parameterized string matching. In Section 3, we present our proposed algorithm for parameterized string matching. In Section 4, we present experimental results. Finally we conclude in Section 5.

2 Related concepts

2.1 Parameterized string matching problem

This section presents small introduction to parameterized string matching problem [2, 3]. Here we assume that all the symbols of \( P[0…m-1] \) and \( T[0…n-1] \) are taken from \( \Sigma \cup \Pi \), where \( \Sigma \) is fixed symbol alphabet of size \( \sigma \) and \( \Pi \) is parameter symbol alphabet of size \( \pi \). A pattern \( P \) matches the text substring \( T[j…j+m-1] \), for \( 0 \leq j \leq (n-m) \), if and only if \( \forall i \in \{0, 1, 2, … m-1\}, f(x_i) = T[j+i] \), where \( f(x) \) is a bijective mapping on \( \Sigma \cup \Pi \). There must be identity on \( \Sigma \) but need not be identity on \( \Pi \). For example, let \( P = XYABX \) on \( \Pi = \{A, B\} \) and \( \Pi' = \{X, Y, Z, W\} \). Pattern \( P \) matches the text substring \( ZWABZ \) with bijective mapping \( X \rightarrow Z \) and \( Y \rightarrow W \). This mapping can be simplified by prefix-encoding [3]. For any string \( S \), \( prev(S) \) maps its each parameter symbols to a non-negative integer \( p \), where \( p \) is
the number of symbols since the last occurrences of $s$ in $T$. The first occurrence of any parameter symbol in \textit{prev}-encoding is encoded as 0 and if $s \in \Sigma$ it is mapped to itself (i.e. to $s$). For example, \textit{prev}(P) = 00AB4 and \textit{prev}(ZWABZ) = 00AB4. With this scheme of \textit{prev}-encoding, the problem of the parameterized string matching can be transformed to the exact string matching problem, where \textit{prev}(P) is matched against \textit{prev}(T[...j+m–1]), for $0 \leq j \leq (n–m)$. The \textit{prev}(P) and the \textit{prev}(T[...j+m–1]) can be recursively updated as $j$ increases with the help of the following lemma [3].

Lemma 1: Let $S = \text{prev}(S)$. Then for $S' = \text{prev}(S[j...j+m–1])$ for all such that $S[i] \in \Pi$ it holds that $S'[i] = S'[i] \text{iff } S'[i] < m$, otherwise $S'[i] = 0$.

2.2 Bit-parallel algorithm (Shift-or)

This section presents shift-or [1] string matching algorithm for exact and parameterized string matching problem. First we define the following terms: (i) $b_w, b_{w+1}, ..., b_{w+m}$ denotes bits of computer word of length $w$. (ii) Exponentiation is used to denote bit repetition (e.g. $0^4 \equiv 00001$). C-like syntax is used for operations on the bits of computer words: "$^\wedge$" is for bit-wise or, "$\&$" for bit-wise and, "$^\wedge$" is for bit-wise xor, "$\cdot$" complements of all the bits. The shift left operation, "$<<$", moves all bits to the left by $r$ positions. If shift $[\text{prev}(P)] > 0$ then a shift of $\text{prev}(P)$ into $T$ is performed.

Lemma 2: Let $P = \text{prev}(P)$. Then for $S = \text{prev}(S[j...j+m–1])$ it holds that $S'[i] = S'[i] \text{iff } S'[i] < m$, otherwise $S'[i] = 0$.

2.3 q-grams

The main objective of using q-gram [15] in string matching algorithm is to make alphabet larger. When using q-grams we process q characters as a single character. There are two ways of transforming a string of characters into a string of q-grams. We can either use overlapping q-grams or non-overlapping q-grams. When using overlapping q-grams, a q-gram starts at every position of the original text while with non-overlapping q-grams, a q-gram starts in every position. For example transforming the word "abcd" into overlapping 2-grams results in the string "ab-bc-cd" and transforming it into non-overlapping 2-grams yields the string "ab-cd". In [16], non-overlapping q-gram was also known as super alphabet of size q. In [16], Shift-or algorithm has been extended to use the super alphabet and is speeded up by a factor of $q'$ (where $q'$ is size of super alphabet). In this extension, the number of bits required to represent vector $B$ is $m+q–1$, where $q$ is size of non-overlapping q-gram and $m$ is pattern length and $D$ is updated by: $D \leftarrow (D << q) \cdot B[T[i]]$. Let $C = \{c_1, c_2, ..., c_m\}$ is a super alphabet consisting of q consecutive symbols of $T$. Now, $B[C] = ((B[c_1] \cdot 1^q) << q-1) | (B[c_2] \cdot 1^q) << q-2) | ... | (B[c_m] \cdot 1^q)$. If after the $i$th step, any of the $m–1+q–2$ bits is zero, then pattern occurs with shift $l \times q–d–1$, where $d^b$ bit from right is zero.

2.4 Hashing the q-grams

In [18], Lecroq developed an algorithm for exact string matching, which considers substring of length $q$ (overlapping q-gram). Substrings $B$ of such a length are hashed using a hash function $h$ into integer values within 0 and 255. For $0 \leq c \leq 255$, $\text{Shift}[c] = \{ \{m–1–i \times c, i = \max \{0 \leq j \leq m–q+1 \mid h(P[j...j+q–1]) = c\} \}$.
shift of length $sh = m - 1 - i$ with $i = \max \{0, j \leq m - q \mid h(P[j \ldots j + q - 1]) = h(P[m - q + 1 \ldots m - 1])\}$.

### 2.5 Horspool algorithm with q-grams

This section presents the Horspool algorithm for parameterized string matching [9]. In the parameterized matching problem, the last character alone never tells that there can’t be a match and even the last two characters do not indicate that the window cannot match. Therefore, a $q$-gram (overlapping) of $q$ character of the window is formed and the shift is based on it. In the parameterized matching problem, shifts are based on the last $q$-gram of the window, and we wish to make a shift that aligns it with the last $q$-gram of the pattern that $p$-matches it. As discussed in section 2.1, two strings $p$-match, if their predecessor strings match. Thus the algorithm indexes the table with the predecessor strings. Many solutions for calculating the indexes are given in [9]. One possible solution for calculating the indexes is to reserve enough bits for each character of the predecessor strings. Many solutions for calculating the indexes are given in [9]. One possible solution for calculating the indexes is to reserve enough bits for each character of the predecessor strings. The $i$th character of the predecessor string takes values between 0 and $i - 1$, so log, $i$ bits are needed to represent it.

For example, consider the text substring "abacab". The prev-encoded string is 00033. After converting into 0 and 1, we get the encoded string 00 01 11 01 01, which in decimal system is 27. Now this encoded pattern is matched using the simple Horspool algorithm.

### 3 Proposed algorithm

In this section, we present our proposed algorithm: FASTQGRAM, for parameterized string matching. Our algorithm uses both the overlapping and non-overlapping $q$-gram to speed-up the algorithm. It inherits the hashing feature from [18] and bit-parallelism feature on $q$-gram from [17]. Overlapping $q$-gram is used during hashing the parameterized $q$-gram and non-overlapping $q$-gram is used during the searching. The proposed algorithm is applicable only when $m \leq w$, where $w$ is word length of computer used. The algorithm consists of two phases: pre-processing and searching phase.

#### 3.1 Pre-processing phase

During pre-processing phase, a pattern $P[0 \ldots m - 1]$ is segmented into $m - q + 1$ distinct overlapping $q$-grams. These distinct $q$-grams are $prev$-encoded. After $prev$-encoding, they are hashed by a hash function [18] to get the desired shift. Another preprocessing of the pattern $P$ is needed for searching purpose. In this preprocessing, $prev$-encoding of the whole pattern is calculated and a bit-vector $B$ as discussed in section 2.2 is obtained. Algorithm 1 gives the pseudo code and Example 1 illustrates the pre-processing.

**Algorithm 1: Preprocessing** $(P, m, T, n)$ for $q = 3$

1. for $i \leftarrow 0$ to 255
2. do $shift[i] \leftarrow m - 2$
3. for $i \leftarrow 2$ to $m - 2$
4. do $P[i - 2 \ldots i] \leftarrow prev\text{-encoding}(P[i - 2 \ldots i])$
5. $h \leftarrow ((P[i - 2 \ldots i] \oplus P[i - 1 \ldots i] \oplus P[i])$
6. $shift[m \mod 256] \leftarrow m - 1 - i$
7. $P'[m - 3 \ldots m - 1] \leftarrow prev\text{-encoding}(P[m - 3 \ldots m - 1])$
8. $h \leftarrow ((P'[m - 3 \ldots m - 1] \oplus P'[m - 2 \ldots m - 1]) \oplus P'[m - 1]$
9. $sh \leftarrow shift[m \mod 256]$
10. $shift[h \mod 256] \leftarrow 0$
11. $P' \leftarrow prev\text{-encoding}(P)$
12. for $i \leftarrow 0$ to $\pi - 1$
13. do $pos[\pi] \leftarrow \infty$
14. for $i \leftarrow 0$ to $\sigma + m - 1$
15. do $B[A[i]] \leftarrow 0$
16. for $i \leftarrow 0$ to $m - 1$
17. do $B[P[i]] \leftarrow B[P[i]] \& \sim (1 << i)$
18. for $i \leftarrow 1$ to $m - 1$
19. do $B[A[\sigma + i]] \leftarrow B[A[\sigma + i]] \& (B[A[\sigma]] \mid (\sim 0 \ll i)))$

#### 3.1.1 Example 1

Consider the pattern $P = a b a b c$ on $A = \{a\}$ and $B = \{b, c\}$. Let us take $q = 3$. The distinct $3$-gram are: $a b a$, $b a b$ and $a b c$. The $prev$-encoded $3$-gram are: $a 0 a$, $a 2 a$ and $0 0 a$ respectively. The corresponding hash values are: $h(0 0 a) = 69$, $h(0 2 a) = 180$ and $h(0 0 a) = 20$. The shift corresponding to these hash values are: $shift(69) = 2$, $shift(180) = 1$, $shift(20) = 0$ and $sh = 3$. The bit-vector $B$ is given by $B[a] = 11111101$, $B[0] = 11011011$, $B[1] = 11111111$, $B[2] = 11101011$, $B[3] = 11111011$, $B[4] = 11111011$.

### 3.2 Searching phase

Set $T[n \ldots n + m - 1]$ to $P$ in order to avoid testing the end of the text, but exit the algorithm only when an occurrence of $P$ is found. The searching phase of the algorithm consists in reading substrings $B$ of length $q$. If $shift(h(B)) > 0$, then a shift of length $shift(h(B))$ is applied. Otherwise, when $shift(h(B)) = 0$, the pattern $P$ is checked in the text by using non-overlapping $q$-gram technique. By using non-overlapping $q$-gram in searching phase, search time can be reduced. Algorithm 2 gives the pseudo code.

**Algorithm 2: Searching** $(B, P, m, T, n)$ for $q = 3$

1. $T[n \ldots n + m - 1] \leftarrow P$
2. $j \leftarrow m - 1$
3. while TRUE
4. do $sh \leftarrow 1$
5. while $sh \neq 0$
6. do $T[j - 2 \ldots j] \leftarrow prev\text{-encoding}(P[j - 2 \ldots j])$
7. $h \leftarrow ((T[j - 2 \ldots j] \oplus T[j]) \oplus T[j])$
8. $sh \leftarrow shift(h \mod 256)$
9. $j \leftarrow j + sh$
10. if $j < n$
11. then NON-OVERLAP-QGRAMS($T, P, m, q$)
12. $j \leftarrow j + sh$
13. else RETURN

**NON-OVERLAP-QGRAMS** ($T, P, m, q$)

1. $D \leftarrow 0$
2. Take $prev$-encoding of $m$-length window $T[j + m - 1 \ldots j + 1]$
3. Let $C \leftarrow (B[T[j - m + 1] \ll (q - 1)) \mid (B[T[j - m + 1]] \ll (q - 2)) \mid (B[T[j - m + 1]])$
4. Update $D$ by: $D \leftarrow (D < 3) \mid C$
5. If after scanning whole the $m$-length window, $d^{th}$ bit become zero in step $l$, then REPORT OCCURRENCE AT $l \times s - d$
4 Experimental results

We have implemented our proposed algorithm: FASTQGRAM and existing algorithms: Horspool [6] and PSO [5] in C++, compiled with GCC 4.2.4 compilers on the Pentium 4, 2.14 GHz processor (word length \( w = 32 \)) with 512 MB RAM, running ubuntu 10.04. A DNA file of size 40 MB is taken from the file ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/other/. The patterns and text are chosen from the set \( \{A, C, G, T\} \). Fig. 1(a) and 1(b) shows the running time of algorithms for varying pattern length, by keeping \( \Sigma = \{a, t\}, \Pi = \{c, g\} \) and \( \Sigma = \{a\}, \Pi = \{c, g, t\} \) respectively. It shows that on increasing the pattern length, FASTQGRAM performs better. Fig. 1(c) shows the running time of algorithm FASTQGRAM by increasing value of \( q \), where \( \Sigma = \{a, t\} \) and \( \Pi = \{c, g\} \).

5 Conclusion

In this paper, we have developed a new algorithm (FASTQGRAM) for parameterized string matching. We compare the proposed algorithm with parameterized shift-or (PSO) and Parameterized Horspool algorithm. From Fig. 1(a) and 1(b), it is clear that (i) on increasing the pattern length, the proposed algorithm performs better than Horspool and is comparable to PSO (ii) on increasing the duplicity in the code (i.e. on increasing size of the set), time increases. From Fig. 1(c), it is clear that on increasing the value of \( q \), time decreases and the best value is obtained when \( q \) is nearly equal to half of the pattern length.

6 References

Authors’ addresses

Rajesh Prasad
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Alahabad, India

Prof. Suneeta Agarwal
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Alahabad, India

Prof. (Dr.) Sanjay Misra
Department of Computer Engineering
Faculty of Engineering
Atilim University
Ankara, Turkey

Anuj Kumar Sharma
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Alahabad, India

Alok Singh
Department of Computer Science and Engineering
Motilal Nehru National Institute of Technology
Alahabad, India