ISSN 1330-3651
UDC/UDK 004.416:004.021

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

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

Original scientific paper

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*

Održavanje softvera binarnim paralelizmom i pristupanje podacima parametariziranim *q*-gramima

Izvorni znanstveni članak

Pri održavanju softvera često je potrebno pronaći dupliciranost prisutnu u kodovima. Dva su fragmenta koda ekvivalentna ako se jedan može transformirati u drugi konzistentnim preimenovanjem identifikatora, znakova i varijabli. Ta se ekvivalencija može otkriti parametariziranim podešavanjem nizova. U tom podešavanju, kaže se da se zadani obrazac P slaže s podnizom t teksta T ako postoji jedan-prema-jedan slaganje između simbola koji pripadaju P i simbola koji pripadaju t. U ovom članku predlažemo učinkoviti algoritam za taj problem upotrebom q-grama sa i bez preklapanja. Pokazujemo djelovanje vremena izvršavanja algoritma na povećanje dupliciranosti prisutne u kodu.

Ključne riječi: *binarni paralelizam, hašingiranje, oblikovanje algoritma, otkrivanje plagijata, poklapanje niza, q-gram, softver za održavanje*

1 Introduction

Exact string matching problem is to find all the occurrences of a given pattern $P[0\ldots m-1]$ in the text $T[0\ldots n-1]$, 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 consistence 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: Σ for fixed symbol alphabet and Π for parameterized alphabet. Symbols from Σ need not to be renamed, whereas symbols from Π 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 overlapping 2-gram, the word "abcd" is transformed to "ab-bc-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 $P[0\ldots m-1]$ and $T[0\ldots n-1]$ are taken from $\Sigma\cup\Pi$, where Σ is fixed symbol alphabet of size σ and Π is parameter symbol alphabet of size π. A pattern P matches the text substring $T[j\ldots j+m-1]$, for $0 \leq j \leq (n-m)$, if and only if $\forall i \in \{0, 1, 2, \ldots m-1\}, f_j(P[i]=T[j+i])$, where $f_j(.)$ is a bijective mapping on $\Sigma\cup\Pi$. There must be identity on Σ but need not be identity on Π. 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 *prev*-encoding [3]. For any string S, *prev*(S) maps its each parameter symbols s to a non-negative integer p, where p is the number of symbols since the last occurrences of s in S. The first occurrence of any parameter symbol in *prev*-encoding is encoded as 0 and if $s \in \Sigma$ it is mapped to itself (i.e. to s). For example, *prev*(P) = 00AB4 and *prev*(ZWABZ) = 00AB4. With this scheme of *prev*-encoding, the problem of the parameterized string matching can be transformed to the exact string matching problem, where *prev*(P) is matched against *prev*($T\,[j\ldots j+m-1]$), for $0 \le j \le (n-m)$. The *prev*(P) and the *prev*($T\,[j\ldots j+m-1]$) can be recursively updated as j increases with the help of the following lemma [3].

Lemma 1: Let S' = prev(S). Then for S'' = prev($S\,[j\ldots j+m-1]$) for all i such that $S[i] \in \Pi$ it holds that $S''[i] = S'[i]$ 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) $b_{w-1}b_{w-2}\ldots b_1 b_0$ denotes bits of computer word of length w. (ii) Exponentiation is used to denote bit repetition (e. g. $0^4 1$=00001). C-like syntax is used for operations on the bits of computer words: "|" is for bit-wise or, "&" is for bit-wise and, ^ is bit-wise xor, "~" complements of all the bits. The shift left operation, "<<r", moves all bits to the left by 'r' and enters 'r' zeros in the right. Σ is an alphabet of size σ.

In the shift-or algorithm [1], a non-deterministic finite automata (NFA) automaton of a given pattern $P[0\ldots m-1]$ is constructed in the pre-processing phase. The automaton has states 0, 1, 2…m, with state 0 as initial state, state m as final state and $\forall\, i = 0\ldots m-1$ and there is a transition from state i to state i+1 for character $P[i]$ of the pattern P. In addition, there is a transition for every $c \in \Sigma$ from and to the initial state. This algorithm builds a table B having one bit mask entry for each $c \in \Sigma$. For $0 \le i \le m-1$, the mask $B[c]$ has i^{th} bit set to 0 iff $P[i] = c$ otherwise it is 1. If the i^{th} bit in $B[c]$ is 0, then in the automaton, there is a transition from the state i to i+1 with character c. For searching, algorithm needs a bit mask D so that the i^{th} bit of this mask is set to 0, if and only if state i in NFA is active. For each text symbol c the state vector D is updated by $D \leftarrow (D << 1) \mid B[c]$. If after processing the i^{th} symbol of the text, the $(m-1)^{\text{th}}$ bit of D is 1, then there is an occurrence of P with shift $i-m$.

In [5], the shift-or algorithm has been extended to parameterize string matching (PSO). It has been extended in the following way:

(i) The pattern P is encoded by *prev*-encoding and stored as *prev*(P).

(ii) For all j = 0, 1, 2...$n-m$, *prev*($T\,[j\ldots j+m-1]$) can be efficiently *prev*-encoded by lemma 1.

(iii) The table B is built such that all the parameterized pattern prefixes can be searched in parallel. To simplify indexing into array B, it is assumed that $\Sigma = \{0, 1\ldots\sigma-1\}$, and *prev*-encoded parameter offsets are mapped into the range $\{\sigma\ldots\sigma+m-1\}$. For this purpose, an array $A[0\ldots\sigma+m-1]$ is formed, in which the positions $0\ldots\sigma-1$ are occupied by element of Σ and the rest positions are occupied by *prev*-encoded offsets.

Searching for $P' = prev(P)$ in $T' = prev(T)$ can't be done directly as explained below. Let the pattern P = XAXX and the text T = ZZAZZAZZ. P' = 0A21 and T' = 01A21A21. Obviously, P has two overlapping parameterized occurrences in T (one with shift = 1 and another with shift = 4) but P' does not have any occurrences in T'. The problem occurs because of when searching for all the m prefixes of the text in parallel, then some non-zero encoded offset p in T' should be interpreted as zero in some case. For example, when searching for P' in $T'[1\ldots4]$ = 1A21, 1(from left) should be zero. The solution to this problem is that the lemma 1 should be applied in parallel to all m-length sub-strings of T'. This has been achieved by [5] in the following way. The bit vector $B[A[\sigma+i]]$ is the match vector for A[i], where $0 \le i \le \sigma$+m–1. If the j^{th} bit of this vector is zero, it means that $P'[j] = A[i]$. If any of the i^{th} least significant bit of $B[A[\sigma]]$ is zero then corresponding bit of $B[A[\sigma+i]]$ is also cleared. This can be achieved as: $B[A[\sigma+i]] \leftarrow B[A[\sigma+i]]$ & $(B[A[\sigma]] \mid (\sim 0 << i))$ which signifies that for $\sigma \le i \le \sigma+m-1$, $A[i]$ is treated as $A[i]$ for prefixes whose length is greater than A[i] and as zero for shorter prefixes thus satisfies lemma 1.

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 q^{th} 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: $D \leftarrow (D << q) \mid B[T[i]]$. Let $C = \{c_1, c_2, c_3\ldots c_q\}$ is a super alphabet consisting of q consecutive symbols of T. Now, $B[C] = ((B[c_1]\ \&\ 1^m) << q-1) \mid (B[c_2]\ \&\ 1^m) << q-2) \mid\ldots\mid (B[c_q]\ \&\ 1^m)$. If after the l^{th} step, any of the $m-1$ to $m+q-2$ bits is zero, then pattern occurs with shift $l \times q-d-1$, where d^{th} 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 $c \le 255$,

$$\text{Shift}[c] = \begin{cases} m-1-i, i = \max\{0 \le j \le m-q+1 \mid h(P[j...j+q\text{-}1]) = c\} \\ m-q \quad \text{when such an i does not exist} \end{cases}$$

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 naively checked in the text T. In this case, a shift of length sh is applied where $sh=m-1-i$ 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 transform the q-grams into predecessor strings and then to reserve enough bits for each character of the predecessor strings in the index. The i^{th} character of the predecessor string takes values between 0 and $i-1$, so $\log_2 i$ bits are needed to represent it.

For example, consider the text substring. "abcab". The prev-encoded string is 00033. After converting into 0 and 1, we get the encoded string 0 00 11 011, 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: preprocessing and searching phase.

3.1 Pre-processing phase

During preprocessing 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$_q$ (P, m, T, n) for q=3

```
for i ←0 to 255
    do shift[i]  m–2
for i  2 to m–2
    do P'[i–2…i]  prev-encoding(P[i–2…i])
        h  ((P'[i–2]2 + P'[i–1])2) + P'[i]
        shift[h mod 256]  m–1–i
P'[m–3…m–1]  prev-encoding(P[m–3…m–1])
h  ((P'[m–3]2 + P'[m–2])2) + P'[m–1]
sh1  shift[h mod 256]
shift[h mod 256]0
P' ←prev-encode(P)
for i ← 0 to π–1
    do pos[Π[i]] ← –∞
for i ← 0 to σ+m–1
    do B[A[i]] ← ~0
for i ← 0 to m–1
    do B[P[i]] ← B[P[i]] & ~ (1 << i)
for i ← 1 to m–1
    do B[A[σ+i]] ← B[A[σ+i]] & (B[A[σ]] | (~0 <<i))
```

3.1.1 Example 1

Consider the pattern P = a b a b c on = $\{a\}$and $\Pi = \{b, c\}$. Let us take q=3. The distinct 3-grams are: a b a, b a b and a b c. The *prev*-encoded 3-grams are: "a 0 a", "0 a 2" and "a 0 0" respectively. The corresponding hash values are: h(a 0 a) = 69, h(0 a 2) = 180 and h(a00) = 20. The shift corresponding to these hash values are: shift(69)=2, shift(180)=1, shift(20)=0 and sh_1=3. The bit-vector B is given by $B[a]$ = 1111010, $B[0]$ = 1101101, $B[1]$ = 1111111, $B[2]$ = 1110101, $B[3]$ = 1111101, $B[4]$ = 1111101.

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$_q$ (P, m, T, n) for q=3

```
T[n…n+m–1] ← P
j ← m–1
while TRUE
 do sh ← 1
    while sh ≠ 0
        do T'[j–2…j] ← prev-encoding(P[j–2…j])
            h ← ((T'[j–2]2 + T'[j–1])2) + T'[j]
            sh ← shift[h mod 256]
            j ← j + sh
    if j < n
      then NON-OVERLAP-QGRAMS(T, P, m, q)
          j ← j + sh1
    else RETURN
```

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

```
D ← ~0
    Take prev-encoding of m-length
window T [j–m+1...j]
    Let C ← (B[T [j–m+1]]<<q–1) | (B[T [j–
m+1]]<<q–2) | (B[T [j–m+1]])
    Update D by: D ← (D<<3) | C
    If after scanning whole the m-length
window, d th bit become zero in step l, then
REPORT OCCURRENCE AT l×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 $\Pi = \{c, g\}$.

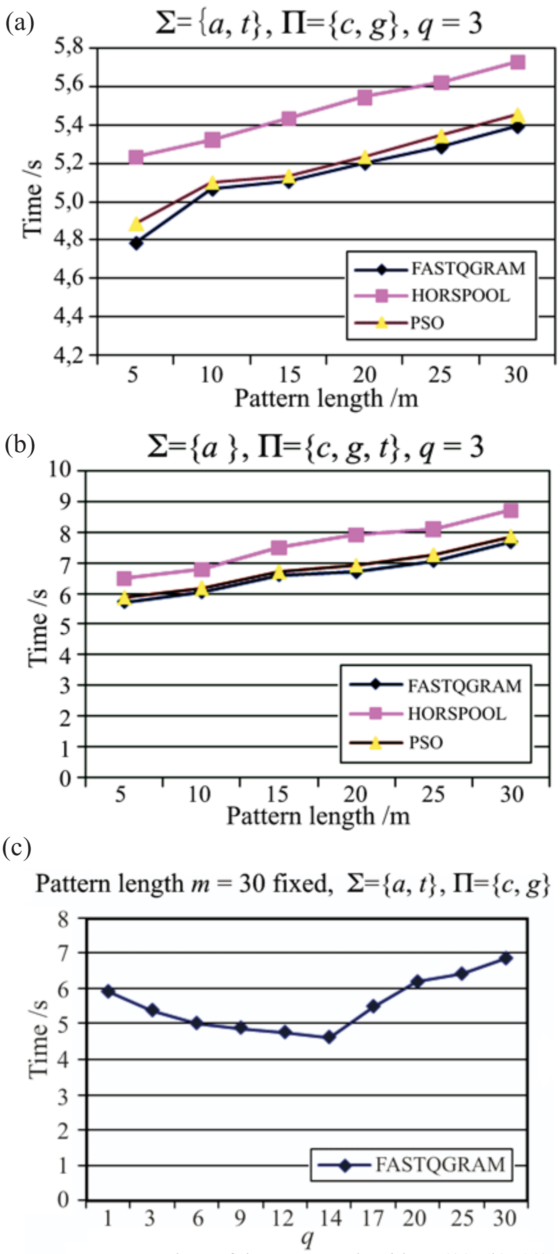

Figure 1 Comparison of time among algorithms ((a), (b), (c)): PSO, FASTQGRAM and HORSPOOL

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

[1] Baeza-Yates, R. A.; Gonnet, G. H. A new approach to text searching. // Communication of ACM, 35, 10(1992), 74-82.
[2] Baker, B.S. Parameterized duplication in string: algorithm and application in software maintenance. // SIAM J. Computing, 26, 5(1997), 1343-1362.
[3] Baker, B. S. Parameterized diff. In proc. 10th Symposium on Discrete Algorithm (SODA), (1999), 854-855.
[4] Boyer, R. S.; Moore, J. S. A fast string-searching algorithm. // Communication of ACM, 20, 10(1977), 762-772.
[5] Fredriksson, K.; Mozgovoy, M. Efficient parameterized string matching. // Information Processing Letters, 100, 3(2006), 91-96.
[6] Horspool, R. N. Practical fast searching in strings. // Software - Practice and Experience, 10, 6(1980), 501-506.
[7] Prasad, R.; Agarwal, S. A new parameterized string matching algorithm by combining bit-parallelism and suffix automata. In proc. of 8th IEEE International Conference on Computer and Information Technology, Sydney, Australia. (2008), 778–783.
[8] Raita, T. Tuning the Boyer-Moore-Horspool string searching algorithm. // Software - Practice and Experience, 22, 10(1992), 879-884.
[9] Salmela, L.; Tarhio, J. Fast Parameterized Matching with q-grams. // Journal of discrete algorithm 6, 3(2008), 408-419.
[10] Smith, P. D. Experiments with a very fast substring search algorithm. // Software - Practice and Experience, 21, 10(1991), 1065-1074.
[11] Sunday, D. M. A very fast substring search algorithm. // Communications of the ACM, 33, 8(1990), 132-142.
[12] Wu, S.; Manber, U. Fast text searching allowing errors. // Communication of the ACM, 35, 10(1992), 83-91.
[13] Kulekci, M.O. BLIM: A New Bit-Parallel Pattern Matching Algorithm Overcoming Computer Word Size Limitation. // Mathematics in Computer Science, 3, 4(2010), 407-420.
[14] Navarro, G.; Raffinot, M. Fast and Flexible String Matching by Combining Bit-parallelism and Suffix automata. // ACM Journal of Experimental Algorithms, 5, 4(2000), 1-36.
[15] Salmela, L.; Tarhio, J.; Kytojoki, J. Multi-pattern String matching with q-gram. Journal of Experimental Algorithmics, 11, (2006), 1-19.
[16] Fredriksson, K. Shift-or String matching with super alphabets. // Information processing letter, 87, 4(2003), 201-204.
[17] Prasad, R.; Agarwal, S. Parameterized Shift-and String matching algorithm using super alphabet. In proc. of the International Conference on Computer and Communication Engineering, Malaysia, (2008), 937-942
[18] Lecroq, T. Fast Exact String matching algorithms. // Information Processing Letter, 102, 6(2007), 229-235.

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