Efficient Parameterized Matching Using Burrows-Wheeler Transform

Anjali Goel[1], Rajesh Prasad[2], Suneeta Agarwal[3], and Amit Sangal[4]
[1]Department of Computer Science and Engineering, Ajay Kumar Garg Engineering College, India
[2]Department of Computer Science and Engineering, Yobe State University, Nigeria
[3]Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, India
[4]Department of Computer Science and Engineering, Sunder Deep Engineering College, India

Abstract: *Two strings P[1, ..., m] and T[1, ..., n] with m≤n, are said to be parameterized match, if one can be transformed into the other via some bijective mapping. It is used in software maintenance, plagiarism detection and detecting isomorphism in a graph. In recent year, Directed Acyclic Word Graph (DAWG). Backward DAWG Matching (BDM) algorithm for exact string matching has been combined with compressed indexing technique: Burrows Wheeler Transform (BWT), to achieve less search time and small space. In this paper, we develop a new efficient Parameterized Burrows Wheeler Transform (PBWT) matching algorithm using the concept of BWT indexing technique. The proposed algorithm requires less space as compared to existing parameterized suffix tree based algorithm.*

Keywords: *Suffix array, burrow-wheeler transform, backward DAWG matching and parameterized matching.*

Received January 9, 2014; accepted December 23, 2014

1. Introduction

String matching is a problem of finding all the occurrences of a pattern P [1, ..., m] in the text T[1, ..., n], m≤n, over some finite alphabet set ∑. It has direct applicability to real world problems such as: DNA subsequence matching, digital libraries, multimedia and intrusion detection [3]. Parameterized string matching [1, 9] is a type of string matching, where symbols of text and patterns are consistently renamed. This renaming is done with the help of a one-one mapping. It is broadly used in software maintenance, plagiarism detection and detecting isomorphism in graph [8].

Baker [1] developed an algorithm for parameterized string matching. Her algorithm mainly uses the concept of suffix tree and has primarily been used in software maintenance.

Suffix Array (SA) [6], one of the indexing techniques which work on suffixes of a text T. It sorts the suffixes of text in lexicographical order [7]. Burrows-Wheeler Transformation (BWT) [3] is a reversible, lossless compression and indexing technique. It is reversible in nature because it is simply permutation of the letters of the text string. So, original text can be reconstructed from compressed text without loss of the information.

Directed Acyclic Word Graph (DAWG). Backward DAWG Matching (BDM) [5] is an average-optimal on-line string matching algorithm which performs matching from backward direction in the m-length text window. In succinct backward-DAWG-matching [5], BDM has been combined with BWT to achieve less search time and small space for exact matching. In 2009, BDM algorithm for exact string matching has been combined with BWT to achieve less search time and small space [5].

In this paper, we develop a new efficient Parameterized Burrows-Wheeler Transformation (PBWT) matching algorithm using the concept of BWT indexing technique. The proposed algorithm requires less space as compared to existing parameterized suffix tree based algorithm [1]. The proposed algorithm is also applicable for handling the multiple patterns simultaneously. To the best of our knowledge, BWT has not been applied on the parameterized matching in the past.

The Running time of our proposed algorithm is O(nm), which for large text length, is almost compatible to parameterized matching based on suffix tree indexing technique but it consumes $O(m^2)$ space which, for large text and small pattern, is very much less in comparison to parameterized suffix tree existing based approach (O(n)).

The paper is organized as follows. In the next section we describe related concepts. In section 3, we present our proposed algorithm: PBWT. Section 4 presents experimental setup and results. Finally, last section concludes the paper.

2. Related Concepts

2.1. Burrows-Wheeler Transform

BWT [3] is an indexing and compression algorithm which achieves high lossless compression ratio. BWT is reversible in nature because it is simply permutation of the letters of the text string. So, original text can be reconstructed from compressed text without loss of the information. BWT forms the cyclic rotations of text string after appending $ at the end of the text. Algorithm sorts cyclic rotated strings of text in lexicographical order. Last column of the sorted strings will be BWT compressed text.

Various BWT based compressors like bzip and szip are available [3]. Its computation function is similar to SA but consumes less memory so it has been adopted by various software as Bowtie, BWA, and SOAP2 [2].

2.2. Backward DAWG Matching (BDM)

BDM is an average-optimal on-line string matching algorithm which performs matching from backward direction (from right to left) in the m-length text window, where m is the pattern length. In this algorithm pattern pre-processing occur using DAWG. Algorithm form the DAWG for the reversed pattern. Using DAWG, we search longest suffix of the reversed pattern (prefix of original pattern) in the text window. BDM remembers longest suffix of the pattern, not the whole pattern. Detail is available in [5].

2.3. Parameterized Matching

Two strings P and T are said to be parameterized match [1, 9], if one can be transformed to other via some bijective mapping. This matching works on two disjoint alphabet sets: $\sum$, the fixed alphabet set and Π, the parameterized alphabet set. During matching, symbols from $\sum$ remains the same while symbols from Π may be consistently renamed. For example, let us assume the text T=XAXXAXAXA and P=XAXAXA with Π={X} and $\sum$={A}. Prev-encoding of pattern and text are prev(P)=0A2A2A and prev(T)=0A21A2A2A respectively. In prev encoding all $\sum$ set symbols remains same while Π set symbols renamed with non-negative integers. Parameterized matching is used to find all parameterized occurrences of a pattern in the text.

3. Proposed PBWT Algorithm

3.1. For Single Pattern

In [4], BDM algorithm for exact string matching has been combined with compressed indexing technique: BWT [3] to achieve less search time and small space. In this section, we propose a new algorithm: PBWT for single and multiple pattern parameterized matching algorithm using BDM algorithm and BWT indexing technique.

In the pre-processing step, we calculate prev-encodings of the pattern P (prev(P)) and the text T (prev(T)). Now append $ at the end of prev(P) as end of the file symbol, calculate the BWT compressed pattern and store in an array 'L'. We use the variables *s* representing the starting row in BWT matrix and variable *e* representing the ending row in a BWT matrix.

- *Working of the Algorithm is as Follows*: We start reading the last character (c) of the m-length text window (from right to left). Find corresponding prev-encoded value of (c) from prev(T). If this character belongs to set $\sum$ then check the occurrence of all possible combination of current sub-string read so far in the BWT matrix. But, if the character (c) belongs to the set Π then check the occurrence of substrings starting from actual and lowest parameterized value present in BWT matrix. Find the minimum starting index (s) for current substring in BWT matrix from both the indexes (calculated from actual and lowest parameterized value) and maximum ending index (e) in BWT matrix (calculated from actual and lowest parameterized value) where this string found in matrix. For each substring, find the starting index row and ending index row in BWT matrix. From the above founded starting indexes, choose minimum starting index (s). Similarly from the above founded ending index, store the maximum ending index (e) where current substring found in matrix. From compressed pattern array 'L', check the position of $ symbol and store its index in variable 'p'. If p is in the range ($s \leq p \leq e$), it shows that the current substring is an exact prefix of one of the suffix of prev(P) and hence it confirms the occurrence of the current sub-string as a factor of the pattern. Otherwise, for certain sub-string, no match is found as a prefix in the pattern. Similarly we will match every current substring of text as a prefix in the pattern. If no match is found till the beginning of the window, then the particular substring is not a factor of the pattern. Therefore, we shift the window completely. If match is found then shift the window by last founded longest suffix of the text. Examples 1 and 2 illustrate the algorithm.
- *Example 1*: Let us assume the text T=XAXXAXAXA and P=XAXAXA on Π={X} and $\sum$={A}. Pattern length (m=6) and text length (n=9). Prev-encoding of pattern and text are prev(P)=0A2A2A and prev(T)=0A21A2A2A respectively. To calculate the BWT of prev(P)$, we need to rotate the string left circularly upto length of prev(P)$ as shown in Figure 1 and then sort these rotated strings in lexicographical order as shown in Figure 2. We calculate the BWT of prev(P)$ shown above and store the last column of sorted string matrix in an array L=A$AA220. Position of $ in

p=2. Initialize variables i=1, shift=6. These variables will be updated at every window shift. Last character of the 6-length text window is X, its corresponding prev(T) value is 2.

0A2A2A$
A2A2A$0
2A2A$0A
A2A$0A2
2A$0A2A
A$0A2A2
$0A2A2A

Figure 1. Unsorted strings.

1	$0A2A2A
2	0A2A2A$
3	2A$0A2A
4	2A2A$0A
5	A$0A2A2
6	A2A$0A2
7	A2A2A$0

Figure 2. Sorted strings.

This character belongs to set Π, therefore, we check the occurrence of substring starting from lowest parameterized value 0 present in BWT matrix and actual parameterized value 2. For each substring starting from both of the parameterized value, we find the minimum starting index row and maximum ending index row in BWT matrix. Update the variables for substring (0): s=2, e=2, for the substring (2): s=3, e=4, minimum s=2, maximum e=4 and Shift=5 (because character X is proper suffix of the pattern). Now check the second last character of the window which is A, it belongs to set ∑, so we check all locations where current substring (A2 and A0) found in BWT matrix. Update variable for substring (A2): s=6, e=7, and substring A0 is not found. This substring (AX) is not proper suffix of the pattern, so, variable Shift will not be updated. In a similar fashion, check occurrence of all characters in BWT upto mismatch or beginning of window is found. When we read fourth character (from right) of the text window, mismatch is found. So, shift the window from the last updated shift variable value which is 3 after adding value of variable i which is initially 1. Now update all variables: i=4, shift=6, s=1 and e=7, because window is changed. Now start reading the last character of 6-length text window which is A, it belongs to set ∑, so we check all locations where substring found in BWT matrix. Update the variables: s=5, e=7. This substring (A) is not proper suffix of pattern. So, variable Shift will not be updated. Check another second last character is X, so check all locations where substring (0A and 2A) is found in BWT matrix. Update variables for substring (0A): s=2, e=2 and for substring (2A): s=3, e=4, minimum s=2, maximum e=4, shift=4 (because substring (XA) is proper suffix of the pattern). Similarly, check another character A, so we check all locations where substring found in BWT matrix. Update variables: s=6, e=7. This substring is not proper suffix of the pattern so shift variable will not be updated. Similarly we will check till the beginning of window or mismatch found. Finally, in the end of this window pattern match with shift=3.

- *Example 2*: Let us assume the Text T=XABXXABX and Pattern P=XABX on the Π={X} and ∑={A, B}, where |P|=4, |T|=8, prev(P)$=0AB3$ and prev(T)=0AB31AB3. Calculate BWT of prev(P)$ and store the compressed pattern in the array L as calculated (3$B0A). Position of $ in p=2. Initialise the variables i=1 and shift 4, these variables will be shifted at every window shift. Now start reading with the last character of the window that is T[4]= X, take Π values and check the occurrence of current substring starting from 0 and 3. Then calculate the "s" and "e" values as 2, 3 respectively, shift will be updated by 3 because this substring is the proper suffix of our pattern. Subsequently check occurrences of all the variables in this window. Proper match will be found at starting position 1 and 4 with updated shift variable for further match in the same text. Algorithm 3 and Figure 3 illustrates the program in C and flowchart respectively.

Algorithm 3: *PBWT (P, T)*

```
# P is a patter, of length m and T is a text of length n
#C []  is an array holding current substring
#verify()= verifies the  current substring is proper suffix of the pattern
#l[k]is an array for checking the last character of m- length text window
PREVT[]= represent the prev-encoding of the text
shift= movement of window by last founded suffix of the text
$= end of the file symbol
while(l[k]!='$')
        {
        k++; p=k+1;
        while(i<=n-m+1)
                {
                h=0;
                 j=m;
                shift=m;
                s=1;
                e=m+1
                while((s<=e)&&j>0)
                        {
                        c=PREVT[i+j-2];
                        C[h++]=c;
                        C[h]='\0'
                        verify();
                        j=j-1;
                        if(s<=p && p<=e)
                                {
                                if (j>0)
                                shift=j;
                                else
                                print("Report Match at
                            Position i");
                                }
                        else i=i+ shift;
                        }
                }
        }
 }
```

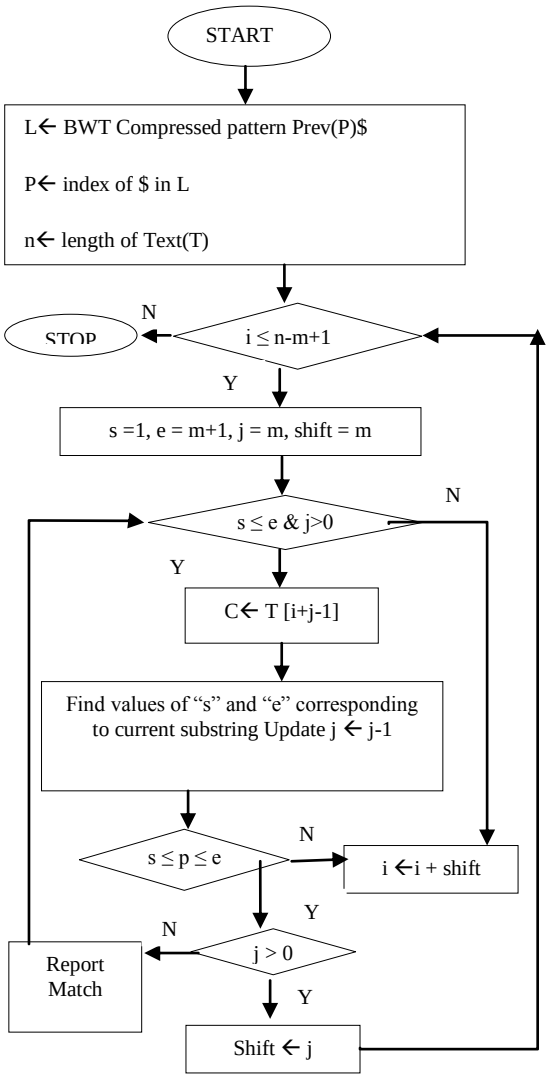


Figure 3. Flow chart showing the working of the algorithm PBWT.

- *Analysis*: The running time of our proposed algorithm is O(nm), but it consumes O(m^2) space which, for large text and small pattern, is very much less in comparison to parameterized suffix tree existing based approach (O(n)). For single pattern, experimental results show that increase in pattern size and file size, running time increases.

3.2. For Multiple Patterns

The algorithm proposed in section 3.1 works for multiple patterns also. The new algorithm is called as MPBWT. As a pre-processing step: we calculate prev-encodings of all the patterns: P_1, P_2, …, P_n. (where n>0) in prev(P_1), prev(P_2), …, prev(P_n) and of the text T (prev(T)). Now append $ at the end of each prev-encoded patterns as end of the file symbol and concatenate each patterns to get prev($P_1$$), prev($P_2$$), …, prev($P_n$$). We calculate the BWT compressed pattern of prev(P) and store in an array 'L', whose length is r(m+1), m being the pattern length and r being the number of patterns. We use the variables *s* representing the starting row in BWT matrix and variable *e* representing the ending row in a BWT matrix.

Working of the algorithm is follows: we start reading the last character (c) of the m-length text window (from right to left). Find corresponding prev-encoded value of (c) from prev(T). If this character belongs to set $\sum$ then check the occurrence of all possible combination of current sub-string read so far in the BWT matrix. But, if the character (c) belongs to the set Π then check the occurrence of substrings starting from actual and lowest parameterized value present in BWT matrix (as done in single pattern). Find the minimum starting index (s) for current substring in BWT matrix from both the indexes (calculated from actual and lowest parameterized value) and maximum ending index (e) in BWT matrix (calculated from actual and lowest parameterized value) where this string found in matrix. For each substring, find the starting index row and ending index row in BWT matrix. From the above founded starting indexes, choose minimum starting index (s). Similarly from the above founded ending index, store the maximum ending index (e) where current substring found in matrix.

Working of the algorithm is same as the previous one. We start reading the last character (c) of the m-length text window (from right to left). Find corresponding prev-encoded value of (c) from prev(T). Check the occurrence of all possible combination of current sub-string read so far in the BWT matrix. For every substring, check $ is in this range ($rank_{\$}$ (L, e)-$rank_{\$}$ (L, s-1) >0), it means there exist some prefix of pattern which occur in this range, Where $rank_{\$}$ (L, e) represents the total number of occurrences of character $ in L up to length e and $rank_{\$}$ (L, e) represents the total number of occurrences of character $ in L up to length s-1. When $ is in this range it shows that current substring must match a prefix of one of the suffix of prev(P) and hence it confirms the occurrence of the current sub-string as a factor of the pattern. Otherwise, for certain sub-string, no match is found as a prefix in the pattern. Similarly we will match every current substring of the text as a prefix in the pattern. If no match is found till the beginning of the window, then the particular substring is not a factor of the pattern. Therefore, we shift the window completely. If match is found then shift the window by last founded longest suffix of the text.

- *Analysis*: The running time of our proposed algorithm is O(r n m), but it consumes O(r m^2) space which, for large text and small pattern, is very much less in comparison to parameterized suffix tree existing based approach (O(n)).

4. Experimental Results

We have implemented our proposed algorithms PBWT and MPBWT in C (some coding part shown below), compiled with Borland compiler version 3.0.

Experiments are performed on Intel(R) Core(TM)2 Duo CPU T6400@2.00 GHz with 3 GB RAM, running Window 7 Ultimate. We are using DNA database sequences, random file of 8 English alphabets and random file of 26 English alphabets for analysis purpose with varied file size. We are calculating the time for different pattern size and file size with values of parameterized alphabet set Π. The execution time of algorithms is measured through CPU time

Table 1 shows the running time of PBWT algorithm on single, varying pattern size and file size on DNA alphabet. Experimental results show that: increase in pattern size with file size, running time increases for single pattern.

Table 1. Running time (in seconds) of PBWT algorithm for various pattern sizes and file size on DNA alphabet {A, C, G, T}.

File Size	Size of Π	Pattern size = 5	Pattern size = 10	Pattern size = 15
5 KB	Π=1	0.219780	0.604396	1.373636
	Π=2	0.219780	0.604396	1.318681
	Π=3	0.274725	0.604396	1.318681
	Π=4	0.274725	0.604396	1.263736
10KB	Π=1	0.384615	1.208791	2.692308
	Π=2	0.494505	1.263736	2.692308
	Π=3	0.439560	1.208791	2.197802
	Π=4	0.494505	1.263736	2.637363
15 KB	Π=1	0.604396	1.868132	3.846154
	Π=2	0.714286	1.813187	3.846154
	Π=3	0.714286	1.868132	3.956044
	Π=4	0.824176	1.868132	3.846154
20KB	Π=1	0.714286	2.307692	4.890110
	Π=2	0.834066	2.307692	4.835165
	Π=3	0.879121	2.362637	5.000000
	Π=4	1.043956	2.362637	4.780220

Table 2 shows the running time (in seconds) of PBWT for multiple patterns with various file size on DNA alphabet. As pattern size and file size increases, the time for matching increases. For multiple pattern, with increase the number of patterns with file size, running time increases.

Table 2. Running time (in seconds) of MPBWT algorithm for multiple pattern (fixed size) and various files size on DNA alphabet.

File Size	Size of Π	No. of Pattern = 3	No. of Pattern = 4	No. of Pattern = 5
5 KB	Π=1	2.527473	5.274725	8.736264
	Π=2	2.527473	5.274725	8.626374
	Π=4	2.527473	5.329670	8.681319
	Π=8	3.571429	7.692308	12.307692
10KB	Π=1	5.000000	10.769231	17.417582
	Π=2	5.219780	10.824176	17.802198
	Π=4	5.164835	10.769231	17.472527
	Π=8	6.978022	15.164835	24.725275
15 KB	Π=1	7.747253	15.934066	26.208791
	Π=2	7.527473	14.120879	25.989011
	Π=4	7.637363	15.769231	26.098901
	Π=8	10.000000	22.747253	37.087912
20KB	Π=1	9.450549	13.076923	32.087912
	Π=2	9.395604	19.835165	32.362637
	Π=4	9.505495	19.670330	32.197802
	Π=8	13.076923	28.791209	47.417582

5. Conclusions

In this paper, we proposed a new algorithm: PBWT for single and multiple patterns using BWT indexing technique. The proposed algorithm asymptotically requires less space as compared to existing algorithm: parameterized suffix tree algorithm. Experimental results in Table 1 show that with increase in pattern size and file size, running time increases for single pattern. Table 2 show that with the increase in number of patterns and file size, running time increases for multiple patterns. Figure 4 showing the experimental screenshots of PBWT algorithm for pattern size 5 and file size 5KB on DNA alphabet {A, C, G, T}. Table 3 shows that the running time of PBWT algorithm decreases with the increasing alphabet size.

Table 3. Running time (in seconds) of PBWT algorithm with increasing file size (Keeping pattern size = 10 and Π = 2 fixed).

File Size	Alphabet size = 4	Alphabet size = 8	Alphabet size = 26
10KB	1.263736	0.989011	0.879121
15KB	1.868132	1.373626	1.318681
20KB	2.362637	1.813187	1.648352

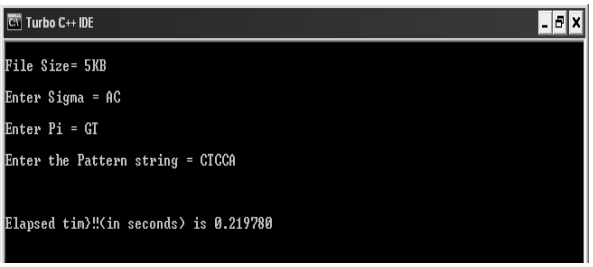


Figure 4. Experimental screenshots of PBWT algorithm for pattern size 5 and file size 5KB on DNA alphabet {A, C, G, T}.

The running time of our proposed algorithm PBWT is O(nm), which for large text length, is almost compatible to parameterized matching based on suffix tree indexing technique but it consumes $O(m^2)$ space which, for large text and small pattern, is very much less in comparison to parameterized suffix tree existing approach, which is O(n).

References

[1] Baker B., "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM Journal of Computing*, vol. 26, no. 5, pp. 1343-1362, 1997.

[2] Beller T., Zwerger M., Gog S., and Ohlebusch E., "Space-Efficient Construction of the Burrows-Wheeler Transform," *in Proceedings of International Symposium on String Processing and Information Retrieval*, Jerusalem, pp. 5-16, 2013.

[3] Burrows M. and Wheeler D., "A Block-sorting Lossless Data Compression Algorithm," Technical Report 124, DEC Systems Research Centre, 1994.

[4] Faro S. and Lecroq T., "An Efficient Matching Algorithm for Encoded DNA Sequences and Binary Strings," *in Proceedings of Combinatorial Pattern Matching*, Lille, pp. 106-115, 2009.

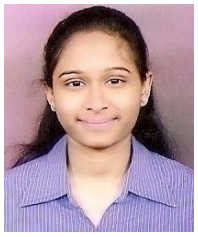
[5] Fredriksson K., "Succinct Backward-DAWG-Matching," *Journal of Experimental Algorithmics*, vol. 13, no. 1, pp. 8- 26, 2009.

[6] Goel A. and Prasad R., "Efficient Record Matching using Indexing Techniques and Deduplication," *International Journal of Computational Vision and Robotics*, vol. 4, no. 1-2, pp. 75-85, 2014.

[7] Huynh T., Hon W., Lam T., and Sung W., "Approximate String Matching using Compressed Suffix Arrays," *Theoretical Computer Science*, vol. 352, no. 1-3, pp. 240-249, 2006.

[8] Mendivelso J., Kim S., Elnikety S., He Y., Hwang S., and Pinzon Y., "Solving Graph Isomorphism Using Parameterized Matching," *in Proceedings of the 20th International Symposium on String Processing and Information Retrieva*, Jerusalem, pp. 230-242, 2013.

[9] Prasad R., Sharma A., Singh A., Agarwal S., and Misra S., "Efficient Bit-Parallel Multi-Patterns Approximate String Matching Algorithms," *Scientific Research and Essays*, vol. 6, no. 4, pp. 876-881, 2011.



Anjali Goel holds a B. Tech in Computer Science and Engineering from Sunder Deep Engineering College, Ghaziabad, India. She has completed her M. Tech. in Computer Science and Engineering from AKGEC, Ghaziabad, India.

Rajesh Prasad is currently, working as Professor and Head in the Department of Computer Science, Yobe State University, Damaturu, Nigeria. He received his M. Tech (SE) and Ph. D (CSE) from MNNIT, Allahabad, India. He is active member of IEEE.

Suneeta Agarwal is currently, working as a Professor in the Department of Computer Science at MNNIT, Allahabad, India. She received BSc, MSc & M.Tech (CS) degrees in 1973, 1975 and 2007 respectively. She did Ph. D from IIT, Kanpur in 1980. She is active member of IEEE, ISTE and CSI.

Amit Sangal received his B. Tech in Computer Science and Engineering from Sunder Deep Engineering College, Ghaziabad, India. He is SIX Sigma, ISTQB certified and GATE qualified.