Shift State Exact String Matching Algorithm

Author : Shivraj Wabale
Systems Engineer
Visa Inc, Bangalore, India
Email: shiv.wabale@gmail.com, shwabale@visa.com

Mentor : Paul Payton
Principal Research Scientist, Visa Research
Visa Inc, Palo Alto, CA
Email: ppayton@visa.com

***Abstract*—This paper describes the use of bit shift and the bit operation to fast the process of String Matching Algorithms used in window shift approach. The paper presents the foundation on which other algorithms can be build on. The basic concept represents that the presented algorithm combined with any existing algorithm gives batter resutls. The paper tackles very fundamental problems in existing window shift string matching algorithms like attempts for most accurate window shift, compares the already processed characters or characters used in preprocessing stage. To build the foundation the paper presents the preprocessing phase is used to form shift stage(SS) table. The SS table is then used in phase two that is the search phase to estimate the exact shift. The SS table is formed from basic preprocessing concept of KMP string matching algorithm[1]. Preprocessing can also be applied to any algorithm falling into a category where preprocessing of the pattern(string-to-be-matched) is performed. The presented method is applicable to algorithms which searched from right to left of pattern. The paper represents the use of approximate finite automata concept. The paper also displays stochastic nature of deciding how to slide the comparison window is unpredictable. Presented concepts in this paper is pure theoretical algorithms. Performance comparisons are done against the average runtime over large text to be matched against.**

Keywords : String Matching Algorithm, Approximate finite Automata, Bit Operations, Computational Complexity.

I. Introduction

A. Problem statement

String Matching is an integral part of various computer application rather the most important. The various algorithms has been presented over the past years to optimize the process. The Shift State (SS) method of finding the exact match of the pattern involve two phases. The preprocessing phase to generate the preprocessed tables and the search phase to actually find the matches. The paper tackles following problems.

- Most of these string matching algorithms shift the window without checking whether the shift is accurate or not, which results in more number of unsuccessful attempts.
- None of the existing algorithms try for more accurate shifting before any attempt, except the Tuned Boyer-Moore algorithm[3] which blindly does three shifts in a row. Hence it also leads to some unsuccessful attempts.
- Most costly part of a string-matching algorithm is to check whether the characters of the pattern match the characters of the window. Most of the existing algorithms forget the previously matched characters, which eventually force the algorithm to match the characters of the window again. And some of the algorithms have tried to solve this problem by introducing constant extra space like in Turbo-BM algorithm [4]. But problem with this kind of algorithm is that, they do not consider the character or characters at immediate right of the window which are passed to precomputed functions.

Table I
SS Table

		G	C	A	G	A	G	A	G		
0	...0	0	0	0	0	0	0	0	0	1	MSB
0	...0	0	0	0	0	0	0	0	0	1	
0	...0	0	0	0	0	0	0	0	1	1	
0	...0	0	0	0	0	0	0	1	0	1	
0	...0	0	0	0	0	0	1	0	0	1	
0	...0	0	0	0	0	1	0	0	0	1	
0	...0	0	0	0	0	0	0	0	0	1	
0	...0	0	0	0	0	0	0	0	1	1	
1	...1	1	1	1	1	1	1	1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
1	...1	1	1	1	1	1	1	1	1	1	LSB
[31]	...	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	← SSv

B. Approaches

Construction of SS table in preprocessing phase acts as basic pillar for all the requirement. The construction of SS table is so strong that it fulfills all the necessary things to tackle the problems listed. The SS table itself can individually be used to shift the window. The processing of SS table element is done by bitwise operations. Finding next shift of window requires just to find next occurrence of set bit in corresponding element of SS table. This process is done by linear time complexity using bit operations. Similarly integration of the existing algorithm with presented methodologies is done by finding the shift by the preprocessed table of the existing algorithm and checking as well as correcting this shift by SS table.

II. SS table

Tab. I shows the formation of SS table of pattern 'GCAGAGAGAG'. Indicating this table as 32 elements of 32 bit variable SSv. The SSv points at the bottom right of the table, so the content of SSv[1] is $10000111111111111111111111111_2$ i.e.

285212671 and SSv[4] is $10011111111111111111111111111_2$ i.e. 83886079. Set bits in a element of SSv represent recurrence of right substring of pattern. For example 27^{th} bit of SSv[4] is set so $32 - 27 + 1 = 6^{th}$ element of pattern is recurrence. The construction of SSv is design considering to achieve the minimum time complexity in searching phase. This type construction lead to logarithmic complexity to find next possible shift.

$q \leftarrow 1$;
$m \leftarrow PatternLenth$;
while $b < 32$ **do**
  $ssv[q] \leftarrow (1 \ll (32 - m)) - 1$;
  $q \leftarrow q + 1$;
end
$q \leftarrow m - 2$;
while $q \geq 0$ **do**
  $ssv[q] \leftarrow ssv[q] \vee (1 \ll (31 - m))$;
  while $k < m - 1$ *AND* $search[k] \neq search[q]$ **do**
    $k \leftarrow m - 1 - pi[k + 1]$;
  end
  if *search[k] = search[q]* **then**
    $k \leftarrow k - 1$;
  end
  $pi[q] \leftarrow m - 1 - k$;
  if *pi[q] = 0* **then**
    $ssv[pi[q]] \leftarrow ssv[pi[q]] \vee (1 \ll (31 - (m - 1 - q)))$;
  end
  $q \leftarrow q - 1$;
end

Algorithm 1: SS construction

The algorithms 1 shows the procedure to build SS table. As from the algorithm, lots of bits operations are used to construct the SS table. The bitwise use makes effective utilization of memory, hence has less space complexity and time complexity as $\Theta(m)$.

III. Searching

The SS table can be used to integrate with any existing algorithm and improves its run time performance. Integrating with Quick search algorithm makes it more effective. The algorithm show in 2 gives the idea how this methodology is combined.

Symbols in algorithm 2 represents :

- search[m], file[n] : string to be matched and matched against respectively.
- ss_com : variable used to store the matched characters.
- $BadCharacterShift$: Table used by Quick search algorithm.

The searching algorithm works from right to left of searching window. The variable ss_com's each bit is set for each matched variable, and for next searching iteration for every matches ss_com is left shifted by one bit. In this process once the least significant bit of ss_com is set then the next few

$i \leftarrow 0$;
while $i \leq n - m$ **do**
  for $j \leftarrow m - 1; j \geq 0; j - 1$ **do**
    if $ss\_com \wedge 1$ **then**
      $k \leftarrow shift(ss\_com)$;
      $j \leftarrow j - k$;
      $ss\_com \leftarrow 0$;
    end
    if $j < 0$ *OR* $search[j] \neq file[i + j]$ **then**
      break;
    end
    $ss\_com \leftarrow ss\_com \gg 1$;
  end
  if $j < 0$ **then**
    The pattern found;
  end
  $k \leftarrow BadCharacterShift$;
  if $k \leq j + 1$ *AND*
  $k \leq m$ *AND*
  $(ssv[m - 1 - j] \wedge (1 \ll ((31 - (m - 2 - j + k)) >$
  $0?(31 - (m - 2 - j + k)) : 0)))$ **then**
    $i \leftarrow i + k$;
    $ss\_com \leftarrow ((1 \ll (m - j)) - 1) \ll (k - 1)$;
  else
    $p \leftarrow k + 1$;
    while $k < j + 2$ **do**
      if
      $(ssv[m-1-j] \wedge (1 \ll (31-(m-2-j+k))))$
      AND $search[m - k] = file[i + m]$ **then**
        break;
      end
      $tp \leftarrow (32 - shift(ssv[m - 1 - j]$
      $\wedge((1 \ll (32 - (p))) - 1)))$;
      $k \leftarrow tp - (m - (j + 2))$;
      $ss\_com \leftarrow ((1 \ll (m - j)) - 1) \ll (k - 1)$;
      $p \leftarrow tp + 1$;
    end
    if $k \geq j + 2$ **then**
      $ss\_com \leftarrow 0$;
    end
    $i \leftarrow i + k$;
  end
end

Algorithm 2: SS Quick algorithm

number of characters comparisons are skipped. This number is calculated by total number of set bits in ss_com by using shift function. Shift function gives result in linear time by all bitwise operations using Single instruction, multiple data (SIMD) within a register (SWAR) technique [6].

As shown in algorithm 4 the function $shift$ works in linear time. Basically function $shift$ returns:

$$\lceil \log_2(x) \rceil, \forall x \in N \mid x < 2^{32} - 1$$

```
i ← 0;
while i ≤ n − m do
    for j ← m − 1; j ≥ 0; j − 1 do
        if ss_com ∧ 1 then
            k ← shift(ss_com);
            j ← j − k;
            ss_com ← 0;
        end
        if j < 0 OR search[j] ≠ file[i + j] then
            break;
        end
        ss_com ← ss_com ≫ 1;
    end
    if j < 0 then
        The pattern found;
    end
    k ← Berry − Ravindran;
    if k ≤ j + 1 AND
    k ≤ m AND
    (ssv[m − 1 − j] ∧ (1 ≪ ((31 − (m − 2 − j + k)) >
    0?(31 − (m − 2 − j + k)) : 0))) then
        i ← i + k;
        if k > 1 then
            ss_com ←ss_com
            ∨((1 ≪ (m − j)) − 1) ≪ (k − 1);
        end
    else
        p ← k + 1;
        while k < j + 2 do
            if
            (ssv[m−1−j]∧(1 ≪ (31−(m−2−j+k))))
            AND search[m − k] = file[i + m] AND
            search[m − k + 1] = file[i + m + 1] then
                break;
            end
            tp ← (32 − shift(ssv[m − 1 − j]
            ∧((1 ≪ (32 − (p))) − 1)));
            k ← tp − (m − (j + 2));
            ss_com ← ((1 ≪ (m − j)) − 1) ≪ (k − 1);
            if k > 1 then
                ss_com ← ss_com ∨ (1 ≪ (k − 2));
            end
            p ← tp + 1;
        end
        if k ≥ j + 2 then
            ss_com ← 0;
        end
        i ← i + k;
    end
end
```

Algorithm 3: SS Berry-Ravindran algorithm

```
function ONES32(x)
    x ← x − ((x ≫ 1) ∧ 0x55555555)
    x ← (((x ≫ 2) ∧ 0x33333333) + (x ∧ 0x33333333))
    x ← (((x ≫ 4) + x) ∧ 0x0f0f0f0f)
    x ← x + (x ≫ 8)
    x ← x + (x ≫ 16)
    return (x ∧ 0x0000003f)
end function
function SHIFT(ss)
    ss ← ss ∨ (ss ≫ 1)
    ss ← ss ∨ (ss ≫ 2)
    ss ← ss ∨ (ss ≫ 4)
    ss ← ss ∨ (ss ≫ 8)
    ss ← ss ∨ (ss ≫ 16)
    return ones32(ss)
end function
```

Algorithm 4: Shift

IV. Methodology

Considering pattern (text to be matched) as GCAGAGAG and the text to be matched against as GCATCGCAGAGAG-TATACAGTACG. Then total number of character comparisons done by algorithm 2 is 9. The characters' comparison done only by Quick Search algorithm is 15 much larger that than integrated algorithm. The comparison table II shows that iterations performed during each search phase. As table indicates each matched character in last iteration and a character or characters (in case where multiple characters are used like in Zhu-Takaoka algorithm[9] and Berry-Ravindran algorithm[11]) used for finding next attempt are being skipped in next iteration. In attempt 4, five characters' comparisons are skipped among which **AGAG** are four characters matched in last attempt and a character **A** used to find next attempt. This shows how the character comparisons are minimized in each attempt. Algorithm presented in 3 shows how to integrate with the algorithms which uses multiple character to find next shift. The algorithm also check its accuracy before starting next attempt. For given example lets consider E as expected shift calculated from bad character table of Quick Search algorithm and let L be number of character matched in last attempt. Now L index of SS (SSv[L]) variable's E+L bit from top is checked, if is it set then shift is accurate otherwise not. In third attempt, E = 2 and L = 4, then SSv[4] variable's 26^{th} bit is set so it is accurate shift. If it was not accurate then next shift is calculated from next set bit of corresponding SS variable from top with one additional character comparisons. But this extra character comparison is compromised by setting bit of ss_com variable i.e. it won't be compared again in next attempt.

V. Performance

The table IV gives the total number of character comparison of various algorithms [12]. And table III gives total character comparison done by three integrated algorithms. As from the tables minimum character comparison done for this example without SS Techniques is 10 from Tuned Boyer Moore [13].

Table II
COMPARISONS TABLE

Text:	G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G	
Attempt 1								1																	
	G	C	A	G	A	G	A	G																	Shift by 1
Attempt 2							3	2	-																
		G	C	A	G	A	G	A	G																Shift by 2
Attempt 3							5	-	-	-	4														
				G	C	A	G	A	G	A	G														Shift by 2
Attempt 4						8	7	-	-	-	-	-	6												
						G	C	A	G	A	G	A	G												Shift by 9
Attempt 5																						9			
															G	C	A	G	A	G	A	G			END

- : skipping of character comparison
_ : mismatched character

And the SS Techniques are used with existing algorithm then result in much less characters' comparisons.

Table III
SS TECHNIQUES

Algorithm	Total characters' comparisons
SS techniques with Quick Search algorithm	9
SS techniques with Berry-Ravindran algorithm	8
SS techniques with Zhu-Takaoka algorithm	9

Considering pattern and text as

- Patter : GCAGAGAG
- Text : GCATCGCAGAGAGTATACAGTACG

As the most costly part of string matching algorithms is characters' comparisons. Hence the performance improvement is done on basis of number of characters comparisons. The randomly generated pattern with $\sigma = 4$of length from 1 to 31 and text both of characters *ACTG* are tested against the integrated and actually algorithms. The graph 1 shows how the integrated algorithm actually reduces the total characters' Comparisons. Also shown on the graph are variations around the regression lines indicating best and worst run times. These variations increase as string length increases. There is some overlap of the variations, indicating the stochastic nature of deciding how to slide the comparison window does not always work. This graphs shows total percent deduction as 14.1%.Table V shows the total percent deduction of characters' comparison for few integrated algorithm against the actual algorithm.

VI. RESULTS

As from the performance analysis, it shows that the presented algorithm helps to reduce the characters' comparisons. Since this comparisons are very costly, techniques presented

Table IV
PERFORMANCE

Algorithm	Total characters' comparisons
Brute Force	30
Deterministic Finite Automaton	24
Morris-Pratt	29
Knuth-Morris-Pratt	18
Simon	24
Colussi	20
Galil-Giancarlo	19
Apostolico-Chrochemore	20
Not So Nave	27
Boyer-Moore	17
Turbo Boyer-Moore	15
Apostolico-Giancarlo	15
Reverse Colussi	16
Horspool	17
Quick Search	15
Tuned Boyer Moore	10
Zhu-Takaoka	12
Berry-Ravindran	16
Smith	15
Raita	18
Reverse Factor	17
Turbo Reverse Factor	13
Forward Dawg Matching	24
Backward Nondeterministic Dawg Matching	15
Backward Oracle Matching	17
Galil-Seiferas	25
Two Way	20
String Matching on Ordered Alphabet	27
Maximal Shift	12
Skip Search	14
KMP Skip Search	14
Alpha Skip Search	18

Table V
PERCENT DEDUCTION

Algorithm	Deduction
Quick Search algorithm	21.71%
Berry-Ravindran algorithm	14.1%
Zhu-Takaoka algorithm	21.41%

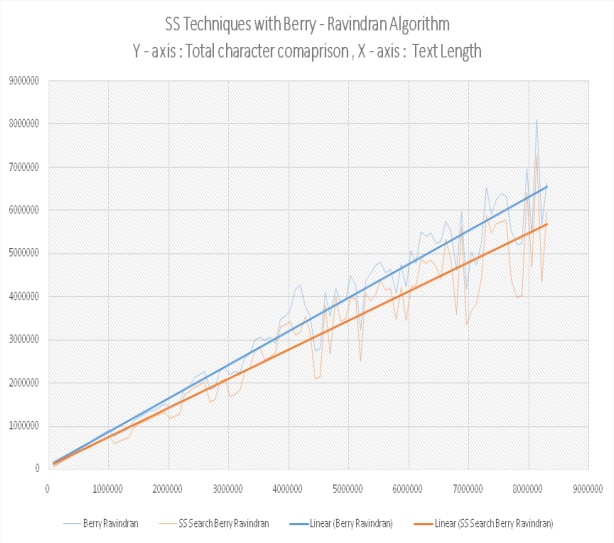

Figure 1. Performance Comparison

helps to reduce this. Which effectively increases the performance. As techniques presented are for limited characters from 1 to 31 in size. This limit can easily increases to 64, 128, etc. by modifying lengths of SS table, ss_com and limit of *Shift* function or by comparing part by part. The ss_com requires to set its bit before attempting next next attempts. If the algorithms uses more than one characters to find next possible shift then all the corresponding characters position should be set in it.

As this Methodologies uses SS table for various purpose like to find accuracy, if not accurate then finding next possible shift and could act as preprocessing table by itself. This multipurpose SS table just need constant extra phase, this preprocessing of SS table is performed in $O(n+\sigma)$ time and space complexity. Time complexity of $O(n)$ is required to in searching phase.

VII. Conclusion and Future Scope

The new developed string matching techniques are compared with various algorithms. This comparison shows great improvement in performance considering characters matching as most costly part. The performance are evaluated over number of randomly generated characters and found to be consistent in improvement. As the bit operations are fast which makes this techniques very fast and real time. The successful improvement shows that this techniques are useful to improve the performance of various existing algorithms or this techniques itself can be use as one of the string matching algorithms. Easy modification can enable it to work upon larger patterns.

References

[1] Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". SIAM Journal on Computing. 6 (2): 323350. doi:10.1137/0206024.

[2] Aoe, Jun-ichi. Computer algorithms: string pattern matching strategies. Vol. 55. John Wiley Sons, 1994.

[3] Tuned Boyer Moore Algorithm Fast string searching, HUME A. and SUNDAY D.M., Software - Practice Experience 21(11), 1991, pp. 1221-1248. Adviser: R. C. T. Lee Speaker: C. W. Cheng National Chi Nan University.

[4] Crochemore, Maxime, et al. "Speeding up two string-matching algorithms." Algorithmica 12.4-5 (1994): 247-267.

[5] CROCHEMORE, M., LECROQ, T., 1996, Pattern matching and text compression algorithms, in CRC Computer Science and Engineering Handbook, A. Tucker ed., Chapter 8, pp 162-202, CRC Press Inc., Boca Raton, FL.

[6] Fisher, Randall James. "General-purpose SIMD within a register: Parallel processing on consumer microprocessors." (2003).

[7] Patterson, David A., and John L. Hennessy. Computer organization and design: the hardware/software interface. Newnes, 2013.

[8] Yu, Wenhua. Advanced FDTD Methods: Parallelization, Acceleration, and Engineering Applications. Artech House, 2011.

[9] ZHU R.F., TAKAOKA T., 1987, On improving the average case of the Boyer-Moore string matching algorithm, Journal of Information Processing 10(3):173-177.

[10] Cormen, Leiserson, and Charles Leiserson. "Rivest, Introduction to algorithms." (1990).

[11] BERRY, T., RAVINDRAN, S., 1999, A fast string matching algorithm and experimental results, in Proceedings of the Prague Stringology Club Workshop'99, J. Holub and M. Simnek ed., Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, 1999, pp 16-26.

[12] Charras, Christian, and Thierry Lecroq. Handbook of exact string matching algorithms. King's College, 2004.

[13] Hume, Andrew, and Daniel Sunday. "Fast string searching." Software: Practice and Experience 21.11 (1991): 1221-1248.

[14] Baeza-Yates, Ricardo, and Gonzalo Navarro. "A faster algorithm for approximate string matching." Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg, 1996.

[15] Charras, Christian, Thierry Lecrog, and Joseph Daniel Pehoushek. "A very fast string matching algorithm for small alphabets and long patterns." Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg, 1998.

[16] Crochemore, Maxime, Wojciech Rytter, and Maxime Crochemore. Text algorithms. Vol. 698. New York: Oxford University Press, 1994.

[17] Crochemore, Maxime. "Off-line serial exact string searching." Pattern matching algorithms. Oxford University Press, 1997.

[18] Sunday, Daniel M. "A very fast substring search algorithm." Communications of the ACM 33.8 (1990): 132-142.

[19] Breslauer, Dany. E cient String Algorithmics. Diss. COLUMBIA UNIVERSITY, 1992.

[20] Charras, Christian, Thierry Lecrog, and Joseph Daniel Pehoushek. "A very fast string matching algorithm for small alphabets and long patterns." Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg, 1998.

[21] Lecroq, Thierry. "A variation on the Boyer-Moore algorithm." Theoretical Computer Science 92.1 (1992): 119-144.

[22] Warren, Henry S. "" This is the first book that promises to tell the deep, dark secrets of computer arithmetic, and it delivers in spades. It contains every trick I knew plus many, many more. A godsend for library developers, compiler writers, and lovers of elegant hacks, it deserves a spot on your shelf right next to Knuth."-Josh Bloch."

[23] Stein, Josef. "Computational problems associated with Racah algebra." Journal of Computational Physics 1.3 (1967): 397-405.

[24] Cain, Thomas R., and Alan T. Sherman. "How to break Gifford's Cipher." Cryptologia 21.3 (1997): 237-286.

[25] Marchenko, V. A. "The generalized shift, transformation operators, and inverse problems." Mathematical Events of the Twentieth Century. Springer Berlin Heidelberg, 2006. 145-162.

[26] Hardy, Darel W., Fred Richman, and Carol L. Walker. Applied algebra: codes, ciphers and discrete algorithms. CRC Press, 2011.