A Comparative Study on Brute Force, Boyer-Moore, and Berry Ravindran Pattern Matching Algorithms

Deo Fetalvero
University of the Philippines Baguio
Governor Pack Rd
Baguio City 2600
dmfetalvero@up.edu.ph

Paul Naraval
University of the Philippines Baguio
Governor Pack Rd
Baguio City 2600
pdnaraval@up.edu.ph

Marielle Nolasco
University of the Philippines Baguio
Governor Pack Rd
Baguio City 2600
mtnolasco@up.edu.ph

ACM Reference format:
Deo Fetalvero, Paul Naraval, and Marielle Nolasco. 2018. A Comparative Study on Brute Force, Boyer-Moore, and Berry Ravindran Pattern Matching Algorithms. In *Proceedings of CMSC 199, Baguio City, Philippines, May 2018,* 4 pages.
DOI:

Abstract

This paper discusses and compares three fundamental pattern matching algorithms: brute force, Boyer-Moore, and Berry Ravindran.

1 INTRODUCTION

Pattern matching is an important concept in computer science due to its many technological applications such as DNA matching, voice recognition, face recognition, network security, and text processing.[1]
Pattern matching is finding one or all occurrences of a certain pattern p of length m in a text t of length n. [1] Pattern matching algorithms as is with all other algorithms base its performance by its time complexity. The time complexity of a pattern matching algorithm is based on its basic operation: comparison. A higher number of comparisons mean larger time complexity.[1] Thus, most pattern matching algorithms are concerned with minimizing the number of comparisons to get a better time efficiency.
This paper is concerned with discussing and illustrating three pattern matching algorithms: brute force, Boyer-Moore, and Berry Ravindran.

2 DEFINITION OF TERMS

- Text: a body of characters T to be matched upon.
- Pattern: a string of characters P which is to be matched with the text.
- Window: the current position k which $T[k+1]\dots T[k+m]$ is aligned with $P[1]\dots P[m]$, where m is the length of pattern P.
- Mismatch: when the characters $T[k+1]\dots T[k+m]$ does not match with all of $P[1]\dots P[m]$.

3 BRUTE FORCE

The brute force or naive method compares each character in the pattern p to each character in the text t. It compares p to t from left to right until it finds p or a mismatch occurs or there is an insufficient number of characters left in t. [2]

3.1 Algorithm

[4] The algorithm for the brute force method is given below:

```
BruteForceStringMatch(T[0...n-1],P[0...m-1])
//Input: An array T[0...n-1] of n characters
    representing a text and an array P[0...m-1] of m
    characters representing a pattern
//Output: The index of the first character in the text
    that starts a matching substring or -1 if the
    search is unsuccessful
    for i = 0 to n-m do
      j = 0
      while j < m and P[j] = T[i+j] do
        j = j+1
      if j = m return i
    return -1
```

3.2 Example

As an example, let $T[0...n-1]$ = "WE WANT TO EAT MORE ONIONS", with whitespace being included, and $P[0...m-1]$ = "ONION".

```
text:  WE WANT TO EAT MORE ONIONS
pat:   ONION
       ↑
```

Since $T[0] = W \neq O = P[0]$, the pattern is immediately shifted to the right without further comparison.

```
text:  WE WANT TO EAT MORE ONIONS
pat:    ONION
        ↑
```

Similarly with $T[1+0] = T[1] = E \neq O = P[0]$, the pattern is shifted to the right. The same process goes on until a match is found in $i = 20$.

```
text:  WE WANT TO EAT MORE ONIONS
pat:                       ONION
                           ↑↑↑↑↑
```

3.3 Worst Case

[5] The worst case occurs when the number of comparisons is maximized. From the algorithm above, the maximum number of comparisons made between each character from p to the characters in text t is m. This is attained when each character from p is compared to each corresponding character in t. Furthermore, the maximum number of comparisons between p and t is $n - m + 1$. This is attained due to the nature of the algorithm wherein it compares p to t at least once and shifting just once if a mismatch occurs. Thus, the maximum number of comparison is $(m)(n - m + 1)$ which implies that the time complexity of the worst case is $\Theta(nm)$.

4 HEURISTIC FUNCTIONS

In Computer Science, A.I. and Mathematical optimizations, a heuristic function is a function that takes the best alternative out of one or more alternatives at a branching step in an algorithm. A heuristic function's objective is to speed up an algorithm by leading it to a 'shortcut.' In some extreme cases these functions may not even improve the speed at which the algorithm converges to a solution and may add unnecessary load for the algorithm. A heuristic function may trade-off optimality, completeness, accuracy and precision depending on the implementation and objectives, in other cases heuristic functions may be generally benficial to the algorithm itself.

The following two exact string matching algorithms that will be discussed use heuristic functions to speed up the matching process.

5 BOYER-MOORE

The Boyer-Moore String Matching algorithm performs its match from right to left instead the usual left to right of its time. This right to left matching enabled Boyer-Moore algorithm to skip more characters than the other algorithms[3].

In Boyer-Moore string searching algorithm, the pattern $pat[1...patlen]$ of length $patlen$ is searched in string $string[1...stringlen]$ of length $stringlen$. For this part of the paper, let $char$ be defined as the $patlen^{th}$ character of $string$ when pat is placed on top of the leftmost part of $string$, $char$ is the character of $string$ aligned with the last character of pat. Boyer and Moore uses a function to quickly traverse $string$. It is called the Bad Character Shift, which was initially referred to as $delta_1$.[3]

5.1 Algorithm

[3]The Boyer-Moore searches using the following algorithm:

```
        $stringlen \leftarrow$ length of $string$.
        $i \leftarrow patlen$.
top:  if $i > stringlen$ then return false
        $j \leftarrow patlen$.
loop: if $j = 0$ then return $j + 1$.
        if $string[i] = pat[j]$
          then
          $j \leftarrow j - 1$.
          $i \leftarrow i - 1$.
          goto loop.
          close;
        $i \leftarrow i + badChar(\ string[i]\ )$
        goto top
```

The function $badChar$ is defined by Boyer and Moore as

$$badChar = \begin{cases} patlen & \text{does not occur in } patlen \\ 2 * patlen - j & \text{otherwise; where } pat(j) \text{ is the rightmost occurrence of } char \text{ in } pat \end{cases}$$

5.2 Example

To make this algorithm clearer, the same example as before will be used.

```
string:  WE WANT TO EAT MORE ONIONS
pat:     ONION
             ↑
```

As $string[5] = A$ is not in the pattern, pat is shifted by $patlen = 5$ to the right.

```
string:  WE WANT TO EAT MORE ONIONS
pat:          ONION
                  ↑
```

This time $string[10] = O$ can be found in pat with the rightmost occurrence at $j = 4$. Hence the pattern is shifted to the right by $5 - 4 = 1$.

```
string:  WE WANT TO EAT MORE ONIONS
pat:           ONION
                   ↑
```

This setting is done until $i = 25$

```
text:  WE WANT TO EAT MORE ONIONS
pat:                       ONION
                           ↑↑↑↑↑
```

5.3 Worst Case

The worst case of Boyer-Moore algorithm occurs when for every alignment $pat[1...patlen]$ and $string[i - patlen + 1 \ldots i]$, every element is compared with only the first elements not being equal. Hence per alignment, $patlen$ comparisons will be done. Also, the pattern is only shifted to the right by one. Since *string* has the length *stringlen* and *pat* is *patlen* long, there will be (*stringlen* − *patlen* + 1) alignments. Therefore, the maximum number of comparisons is (*patlen*) × (*stringlen* − *patlen* + 1), consequently it has a complexity of Θ(*patlen* × *stringlen*).[2]

6 BERRY-RAVINDRAN

Based on a plethora of algorithms that was brought about by Boyer-Moore Exact String Matching Algorithm, Berry and Ravindran designed an algorithm which does shifting considering the two consecutive characters to the right of the window.

6.1 Algorithm

(1) Let P be the pattern string to find. Let T be the text to be matched upon. Let n be the length of text T where $T[1]$ is the first character and $T[n]$ is the last character of text T. Let m be the length of the pattern such that $P[m]$ is the last character and $P[1]$ is the first character of pattern P.
(2) Create and initialize the Shift Table. The two dimensional array, ST (Shift Table), of size at most $(m+1) \times (m+1)$ will store the shift values for all pairs of characters. The ST will be initialised with values of $m+2$.
(3) Create and fill the CON Table. As the index of the ST is of type integer, we need to convert the pairs of characters into pairs of integers. This is done by defining an array of ASCII character set size called CON with each entry initialised to 0.
(4) For each character c in the pattern the right most position (numbering from the right, starting with 1) is entered in the corresponding location in CON ($CON[c]$ = right most position of c).
(5) For every index i starting at 1, reading from left to right and finish at $m+1$, do steps 6–9.
(6) For each of characters $P[i]$ and $P[i-1]$ find the values of indices a and b from $CON[P[i-1]]$ and $CON[P[i]]$ respectively, in the CON table. If any of $P[i]$ or $P[i-1]$ does not exist, for every index q available to ST table, set $b = q$ or $a = q$, respectively and do step 7 otherwise, if both exist, proceed.
(7) In the shift table ST at (a, b), set the value with the formula:

$$ST(a, b) = m + 2 - i$$

(8) Create the index k. Set a new variable $k = 0$.
(9) For every character $P[i]$ (index i) starting at $P[m]$ (index m) and finishing at $P[1]$ (index 1), do step 10
(10) Compare the characters $P[i]$ with $T[k+i]$. If they're not the same (mismatch), skip to step 11. If they're the same, continue searching and once each pairs of characters are all the same, declare a match at $T[k+1]$ to $T[k+m]$ and stop the program.
(11) Find the shift value s. The value of a shift for the pair $T[k+m+1]$ and $T[k+m+2]$ is $s = ST(CON[T[k+m+1]], CON[T[k+m+2]])$.
(12) Add the new shift value to k. Set $k = k + s$.
(13) Check the alignment window. Check if the new $k + m > n$. If true, declare that no match was found and stop the program, otherwise repeat steps 10–14.

6.2 Example

For an example, the pattern 'onion' is tested with the text 'we want to test with onion'. The example is referenced to Thomas Berry's Dissertation[2]. Let P be the pattern 'onion' and T be the text 'we want to test with onion'. Now let m be the length of the pattern which is 5 and n be the length of the text which is 26. Further, the CON table is generated as follows:

Character	...	a	b	...	h	i	j
ASCII Value	...	97	98	...	104	105	106
CON	...	0	0	...	0	3	0

Character	...	n	o	p	...
ASCII Value	...	110	111	112	...
CON	...	1	2	0	...

The ST table is all initialized with values of $m+2$ which is 7. The Shift Values will be entered in the following order:

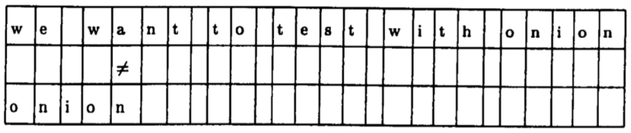
(1) For CON[(any)] and CON[o], ST((any) , 2) = 6
(2) For CON[n] and CON[i], $ST(1,3) = 5$
(3) For CON[i] and CON[o], $ST(3,2) = 4$
(4) For CON[o] and CON[n], $ST(2,1) = 3$
(5) For CON[o] and CON[n], $ST(2,1) = 2$
(6) For CON[n] and CON[(any)], ST(1, (any)) = 1

The ST table would look like the following after the values are entered. The table is accessed by row and then by column such that the value of $ST(i, j)$ of any i or j is at row i and column j.

	0	1	2	3
0	7	7	6	7
1	1	1	1	1
2	7	2	6	7
3	7	7	3	7

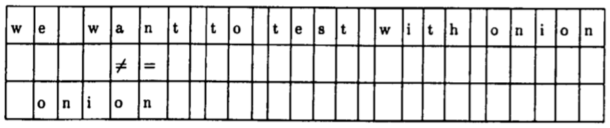
The preprocessing section of the algorithm is now over. String matching can now be initiated. The ST and CON tables will now be used as a heuristic to speed up the string matching process.

In the following tables, the remaining steps are executed until a 'match' or 'no matches' declaration can be inferred. The first row shows the text and the third row shows the position of the pattern. The second row shows whether the aligned pattern and text characters match (=) or mismatch (≠) as the comparisons are made.



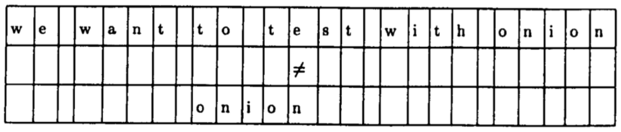
w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				≠																					
o	n	i	o	n																					

Mismatch shift on $ST(CON[\mathbf{n}], CON[\mathbf{t}]) = ST(1, 0) = 1$



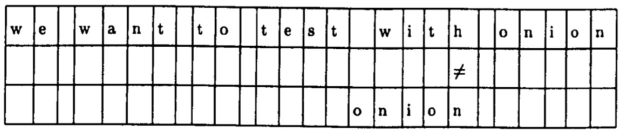
w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
				≠	=																				
	o	n	i	o	n																				

Mismatch shift on $ST(Con[\mathbf{t}], CON[\ ]) = ST(0,0) = 7$



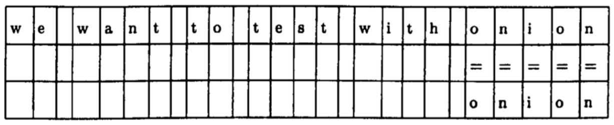
w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
												≠													
								o	n	i	o	n													

Mismatch shift on $ST(Con[\mathbf{s}], CON[\mathbf{t}]) = ST(0,0) = 7$



w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																			≠						
															o	n	i	o	n						

Mismatch shift on $ST(Con[\ ], CON[\mathbf{o}]) = ST(0,2) = 6$



w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																					=	=	=	=	=
																					o	n	i	o	n

So the pattern 'onion' has been matched and the test is exhausted

Further, the word 'onion', in a text length of 26, was found using only 10 comparisons.

REFERENCES

[1] Amjad Hudaib, Rola al-khalid, Mariam Abd Alfattah Itriq, and Dima Suleiman. 2015. Four Sliding Windows Pattern Matching Algorithm. *Journal of Software Engineering and Applications* 8, 3 (March 2015), 154–165.

[2] Thomas Berry. 2002. *Algorithm Engineering: String processing.* Ph.D. Dissertation. Advisor(s) Dr. S. Ravindran.

[3] Robert Boyer, and J Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (October 1977), 762–772.

[4] GeeksforGeeks. n.d.. Searching for Patterns — Set 1 (Naive Pattern Searching). (n.d.). Retrieved May 16, 2018 from https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/

[5] Academic Stuffs. September 12,2012. Brute Force(Naive) String Matching Algorithm. (September 12,2012). Retrieved May 16, 2018 from http://somemoreacademic.blogspot.com/2012/09/brute-forcenaive-string-matching.html