Optimization of Boyer-Moore-Horspool-Sunday Algorithm

Rionaldi Chandraseta - 13515077
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Bandung, Indonesia
rionaldi.chandraseta@gmail.com

***Abstract*—String matching is an important aspect in Computer Science. From search engines to bioinformatics, string matching algorithm is crucial in these fields. One of the widely used algorithm for string matching is the BM (Boyer-Moore) algorithm which dated back to 1977. Forty years later, improvements have been made for the BM algorithm, namely the BMH (Boyer-Moore-Horspool) algorithm and the BMHS (Boyer-Moore-Horspool-Sunday) algorithm. This paper will analyze the difference between Boyer-Moore algorithm and its improvements and present an optimization toward the BMHS algorithm.**

***Keywords*—Boyer-Moore, Boyer-Moore-Horspool, Boyer-Moore-Horspool-Sunday, Optimization, String Matching**

I. INTRODUCTION

Published in 1977, the BM (Boyer-Moore) algorithm is named after its founder, Robert S. Boyer and J Strother Moore. It is still an efficient string searching algorithm, and considered as the benchmark of string searching.

BM preprocesses the pattern that would be searched. This algorithm also brings a new perspective in string matching algorithm. Instead of processing the searched pattern from left to right, BM started the process from right to left. The unique approach is done to accommodate one of the shift rules in the algorithm, the good suffix rule as stated in reference [1].

Unfortunately, the good suffix rule is quite complex, both in concept and implementation. In 1980, Nigel Horspool proposed a simplified BM algorithm which does not require the good suffix rule. Reference [2] states that the tests done by Horspool show no significant differences between the original BM algorithm and the algorithm now known as BMH (Boyer-Moore-Horspool).

Ten years later, Daniel M. Sunday further improved the BMH algorithm. In reference [3], Sunday proposed that the shift amount is determined by the first character to the right of the text being processed. This caused the BMHS (Boyer-Moore-Horspool-Sunday) algorithm to be able to jump further than BMH, resulting in fewer comparisons and faster execution time.

The case of BMHS being faster than BMH and BM on average is true, but the BMH and BMHS algorithm would produce a really bad result compared to BM, both in number of comparisons and execution time, on certain cases. This is the consequence of removing the good suffix rule from calculations. On worst cases, the BMH and BMHS algorithm perform badly, in fact comparable to brute force algorithm. Thus, the need of an optimization to overcome the worst case scenario is needed.

II. THEORY

A. *Boyer-Moore Algorithm*

The BM algorithm commonly refers to the pattern being searched as “needle” and the text in which the pattern is searched as “haystack.” Unlike other common string matching algorithm, BM goes against the normal intuition of matching the string from left to right. The needle is lined up with the haystack, and the matching process starts at the end of the needle. The algorithm continues comparing each character toward the front end of the needle. If a mismatch occurs, the needle would be “shifted” right. The shift value depends on the BM shift rules.

The algorithm has two shift rules, both values are predetermined in preprocessing state. First shift rule is called the Bad Character rule, and the second rule is called the Good Suffix rule.

The bad character rule uses a last occurrence table. The table contains the index where each character last occurred in the needle.

TABLE 1 *Last occurrence table of the pattern "foxtrot"*

	pattern: foxtrot				
Character	f	o	x	t	r
Last Index	0	5	2	6	4

The good suffix rule uses a jump table generated from preprocessing the pattern. It is quite a complex concept, but helps eliminating common worst case scenario for BM algorithm. Basically, the algorithm searches for reoccurring substring in the needle. In the case of a mismatch, the algorithm could jump over the similar substring, reducing the number of comparisons.

TABLE 2 *Jump table of the pattern "foxtrot"*

	pattern: foxtrot						
Iteration	0	1	2	3	4	5	6
Jump	1	3	7	7	7	7	7

Note that iteration 0 in Table 2 indicates the jump value if mismatch occurred at the first matching process, which is the last character in the pattern (“T”), Index 1 is for when the mismatch occurred at the second matching process, (“OT”), and so on.

A mismatch on iteration 0 (the rightmost character in the needle) would generate a jump value of 1, this is because there is no suffix to be analyzed. The algorithm tries to find the first character to the left that is not the rightmost character, i.e. the pattern “ADD” have a jump value of 2 on iteration 0 because of the repeating “D” at the end of the string. The jump value of 2 ensures the next checked character is different than “D” that is known to be a mismatch.

While processing iteration 1, the algorithm assumes that the last character in the pattern matches correctly, thus the good suffix is “T”. Then, it searches for the nearest substring to the left containing “_T” where “_” is a random character except the character on index 1, which is “O”. In this particular case, the string “foxtrot” does have a recurring suffix. The substring “OT” and “XT” both has the suffix “T” with different first character. Hence, jump value of iteration 1 is the distance between “O” and “X” which equals 3.

Iteration 2 to 6 in Table 2 are filled with 7, which is the needle length. This is because the process could not find any reoccurring substring of “_OT”, “_ROT”, “_TROT”, and so on.

Based on reference [1], the string matching process would match the needle from right to left. In the case of a mismatch, the BM algorithm considers both shift rules, select the larger value, and shift the needle accordingly. The algorithm returns the text index where the needle successfully matched until the leftmost character, or -1 if no match is found.

TABLE 3 *An example of BM algorithm matching process, 4 shifts, 14 comparisons*

	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	O	T																	
2				F	O	X	T	R	O	T														
3											F	O	X	T	R	O	T							
4												F	O	X	T	R	O	T						
5																		F	O	X	T	R	O	T

B. *Boyer-Moore-Horspool Algorithm*

BMH uses the same core principle as BM in the matching process. However, BMH does not use the good suffix rule on calculating the shift value. The algorithm depends solely on a modified last occurrence table.

The last occurrence table is generated based on every character in the needle, with an exception for the last character. By doing this, if the last character only occurs once, the shift value is equal to the needle’s length. If the character is present elsewhere on the pattern, it would be shifted according to the last occurrence index.

Another main difference between BM and BMH is the way they use the last occurrence table. The original BM calculates the jump value based on the character in the haystack that causes mismatch. In BMH, the jump value is always calculated based on the character of the haystack that is aligned with the rightmost character of the needle as stated in reference [2].

The idea behind the algorithm is that if a mismatch occurred, the needle would be shifted right along the haystack. So, if a mismatch occurred, shifting the needle based on only the rightmost aligned character of the haystack would not skip over a possible match as it is checked after the shift.

Sometimes the calculated shift value is less than the shift value of BM, causing a longer search time due to more comparisons needed. But, on average case, the BMH algorithm is on par with BM. Do put in mind that the implementation of BMH is simpler than BM that uses the good suffix rule.

TABLE 4 *An example of BMH algorithm matching process, 5 shifts, 16 comparisons*

	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	O	T																	
2				F	O	X	T	R	O	T														
3							F	O	X	T	R	O	T											
4										F	O	X	T	R	O	T								
5																	F	O	X	T	R	O	T	
6																		F	O	X	T	R	O	T

C. *Boyer-Moore-Horspool-Sunday Algorithm*

Like the BMH, the BMHS also abandons the good suffix rule, and uses only the last occurrence table. Yet, unlike the BMH, the BMHS uses the original last occurrence table which does not exclude the last character of the needle.

The idea behind BMHS algorithm is pretty much the same as BMH, if a mismatch occurred, then the needle would have to be shifted to the right. BMHS literally takes this idea one step ahead.

BMH algorithm calculates the shift based on the text character aligned with the needle's rightmost character. Meanwhile, BMHS algorithm calculates it based on the character exactly one position to the right of the text character aligned with the needle's rightmost character. The jump value is calculated based on the last occurrence of that character.

The BMHS algorithm allows a maximum jump of needle's length + 1, making it able to do larger jumps compared to BM and BMH. Unfortunately, because the BMHS algorithm uses the character that is positioned exactly one after the pattern's rightmost character, the generated jump value could be less than BM and BMH. In the case where the character that is checked by BMHS exists in the pattern, and the mismatch happened on the rightmost character of the pattern, BMH would generate a larger jump value.

TABLE 5 *An example of BMHS algorithm matching process, 3 shifts, 11 comparisons*

	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	O	T																	
2									F	O	X	T	R	O	T									
3																	F	O	X	T	R	O	T	
4																		F	O	X	T	R	O	T

III. Worst And Average Case of BM-BMH-BMHS

Based on Table 3 and Table 5, it could be inferred that on average case, BMHS is more efficient than regular BM. BMHS has a larger maximum jump, and it results in fewer shifts and fewer comparisons in general.

However, BMHS abandoned the concept of good suffix that is present on the original BM algorithm. This causes the BMHS to have a very bad worst case scenario, which is in fact comparable to a brute force algorithm.

TABLE 6 *An example of finding the pattern "ABBBB" with BM*

	B	B	B	B	B	B	B	B	B	B
1	A	B	B	B	B					
2						A	B	B	B	B

TABLE 7 *An example of finding the pattern "ABBBB" with BMHS*

	B	B	B	B	B	B	B	B	B	B
1	A	B	B	B	B					
2		A	B	B	B	B				
3			A	B	B	B	B			
4				A	B	B	B	B		
5					A	B	B	B	B	
6						A	B	B	B	B

In Table 6, the BM algorithm only needs to shift 1 time and compare 10 characters to check the string and to know that there is no match for the pattern "ABBBB". On the other hand, from Table 7, the BMHS algorithm needs to shift 5 times and performs 30 comparisons before determining that the pattern does not exist in the string.

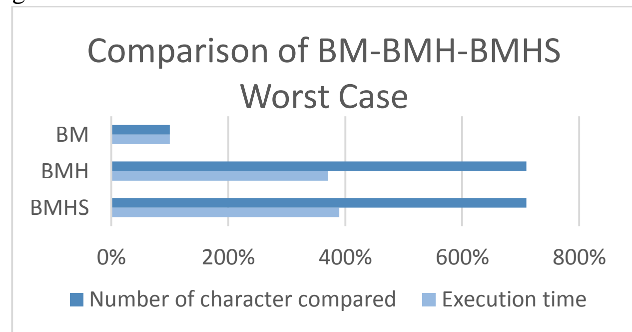
The BM algorithm uses the good suffix rule to figure out that the pattern does not have other occurrence of "_BBBB" and generated a jump value of 5. While the bad character rule generates a jump value of 1, the algorithm would select the larger jump value, which is 5.

BMHS however, does not have access to the good suffix rule. The first matching process would fail at text index 0. The algorithm calculates the jump value based on the character at text index 5, which is "B", and this would return the value of 1. This also applies for the next processes after the shifts. Because the mismatch occurred right on the pattern's leftmost character, the number of comparisons increases by the pattern's length for each iteration.

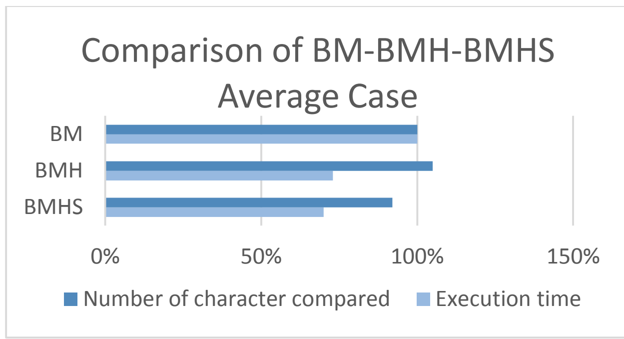
Of course, one could argue that on common cases of string matching this worst case does not come up regularly. That is also true for most string matching usage, such as a find function in text processor, or a search engine, where the worst case of BMHS is not prone to happen.

How about some more specialized fields such as DNA sequence matching? In such field, a searched pattern might be similar to the string that is being searched.

The following test was done with an Intel i7 processor, 8 GB RAM, Windows 10 Operating System, and all algorithms coded in C++. The test data is a 64 KB text designed for the worst case of BM algorithm and its variations. The searched pattern is 7 character long, with guaranteed mismatch on leftmost character.



GRAPH 1 *Number of character comparison and execution time of BM-BMH-BMHS algorithms' worst case*



GRAPH 2 *Number of character comparison and execution time of BM-BMH-BMHS algorithms' average case*

The data on Graph 1 is presented by referencing the BM algorithm as 100% in both number of character compared and execution time. The number of character compared in BMH and BMHS is around 7 times of BM. This is as expected since the pattern is 7 character long. The execution time is faster in BMH probably because the BMHS needs to access and lookup the next-to-last character in the text instead of the mismatched character in BMH.

Nevertheless, it would be unfair for BMH and BMHS if the algorithms are benchmarked based on their worst case scenario. The second test data uses a 710 KB text filled with regular English words. The searched pattern is 7 characters long.

From Graph 2, it could be concluded that BMH and BMHS is actually faster in execution time against BM. The BMH algorithm actually compares more character than BM, but has a faster execution time. This is also understandable because BM calculate two jump values and determine the maximum value between the two generated values. The time taken to do this is minuscule, but when repeated over and over again several thousands or millions times throughout the process, this time adds up and slows down the process.

Out of the three algorithms, BMHS achieved the least number of character compared and also the least execution time. The small number of compared character is caused by the larger maximum jump value that the BMHS has over BM and BMH. The execution time is also affected by the less number of comparison made. As BMHS uses only the least occurrence table like BMH, the execution time is also faster than BM.

IV. Optimizing BMHS Algorithm

BMHS algorithm is often faster than BM and BMH in average case. This is not the case when facing the worst case scenario though. There are several ideas to further optimize the BMHS algorithm.

A. *The Pincer Method*

The pincer method is based on the pincer movement that is used in military to describe a maneuver in which the enemy is attacked from both flanks (sides). In the world's history, some decisive victories were achieved by using this method, i.e. battle of Manzikert and battle of Stalingrad.

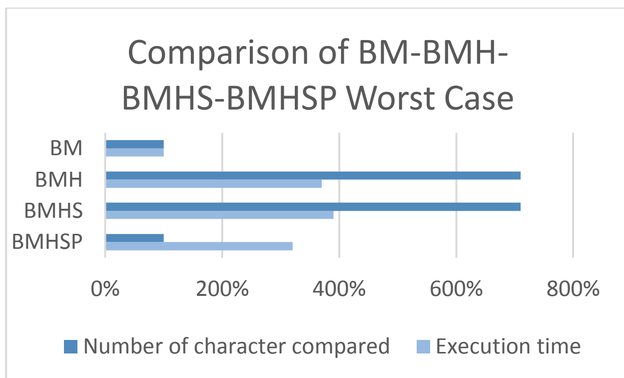
The BMHS algorithm could avoid its common worst case scenario by using the core principle of this pincer movement. The idea is to alternately check from both the left side and the right side.

BMHS does not require a fixed searching pattern from right to left like the original BM algorithm. No matter where the mismatch is found, BMHS always use the next-to-last character to calculate the jump value.

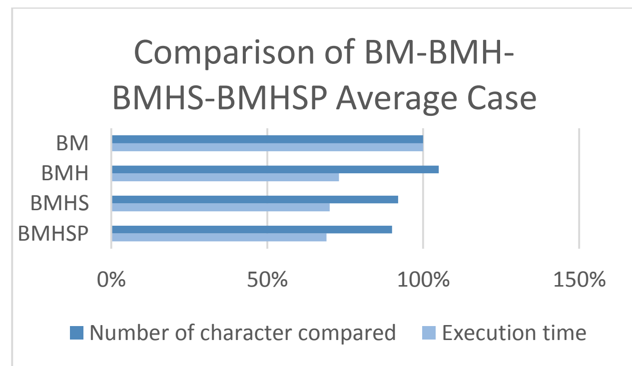
The modified BMHS algorithm, dubbed BMHSP (Boyer-Moore-Horspool-Pincer), would check the leftmost character in the pattern, followed by the rightmost character, then the second character, and so on

TABLE 8 *An example of BMHSP algorithm matching process, 3 shifts, 11 comparisons*

	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	O	T																	
2									F	O	X	T	R	O	T									
3																	F	O	X	T	R	O	T	
4																		F	O	X	T	R	O	T



GRAPH 3 *Number of character comparison and execution time of BM-BMH-BMHS-BMHSP algorithms' worst case*



GRAPH 4 *Number of character comparison and execution time of BM-BMH-BMHS-BMHSP algorithms' average case*

The data in Graph 3 is interesting to say the least. The BMHSP actually compares the same number of character as the BM. This is because the mismatch always occur at the leftmost character. The BM algorithm always checks it last, but the good suffix rule would jump ahead as long as the pattern's length. Meanwhile, BMHSP algorithm always found the mismatch on first character check, but the algorithm is unable to advance further than 1 step.

One possible reason of the high execution time despite a low number of compared character is the calculation of two pointers used to determine which character to check on the left and the right side. The pointers are recalculated after every shift, this causes performance reduction as the shift is practically done as many times as the text's length.

The result looks more promising in Graph 4, where the BMHSP algorithm runs faster than BMHS. Lower number of compared character is achieved thanks to the pincer method. On average, a mismatch is detected earlier by checking the front and end part of the pattern alternatingly.

Like all algorithm, the pincer method also has its own worst case, and it is no better than the BMHS. The worst case is when the different character happens to be in the middle of the pattern.

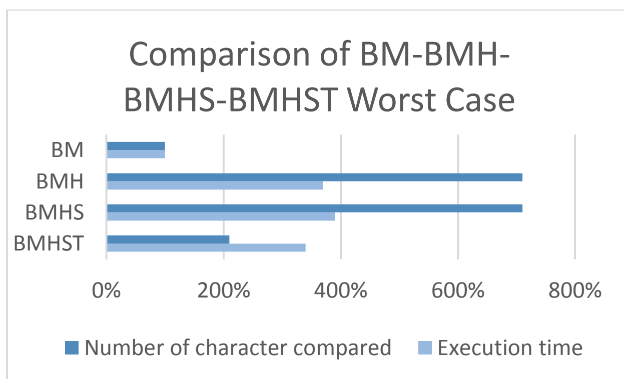
B. The Tri-Point Check

The worst case scenario for the pincer method sparked an idea that could help in optimizing the average case of the BMHS. Instead of checking only the leftmost and the rightmost character of the pattern on the beginning, why not check the middle character of the pattern too.

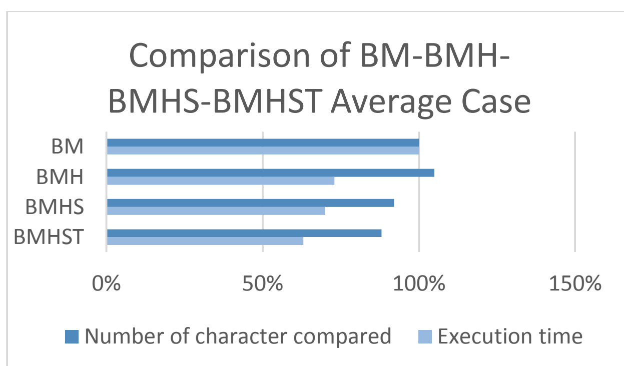
The BMHS algorithm is modified, and referred to as BMHST (Boyer-Moore-Horspool-Tri), to check the middle character first after every shift, before proceeding to check with the pincer method. This further optimizes the average case because the number of substring that has the same first character, middle character, and last character but still differs in other indexes is much smaller than those that has the same first and last character.

TABLE 9 *An example of BMHST algorithm matching process, 3 shifts, 10 comparisons*

	J	U	L	I	E	T	T	H	O	T	E	L	T	A	N	G	O	F	O	X	T	R	O	T
1	F	O	X	T	R	O	T																	
2									F	O	X	T	R	O	T									
3																	F	O	X	T	R	O	T	
4																		F	O	X	T	R	O	T



GRAPH 5 *Number of character comparison and execution time of BM-BMH-BMHS-BMHST algorithms' worst case*



GRAPH 6 *Number of character comparison and execution time of BM-BMH-BMHS-BMHST algorithms' average case*

According to Graph 5, there is no improvement in the worst case over BMHS. The number of compared character increases because BMHST actually matches the middle character before the leftmost character. Surprisingly, the execution time is not far off the BMHSP, which might support the previous statement that the execution time is affected by the calculation of two pointers that is used to determine the next character index to be checked.

On average case, Graph 6 shows that BMHST is actually slightly faster than BMHSP. A slightly worse performance on worst case scenario but slightly better performance on average case scenario makes BMHST the preferred algorithm over BMHSP.

C. The Random Pivot

Inspired by the concept of quick sort algorithm, where the pivot is selected at random to minimize the occurrence of its worst case scenario, the BMHS algorithm could also benefit from its own worst case scenario by selecting which character to check first at random.

The implementation is fairly simple, after every shift, generate a random number between 0 and the pattern's length and start comparing the characters around it. For example, if the selected random number is n, the next character to be checked is n+1 and n-1.

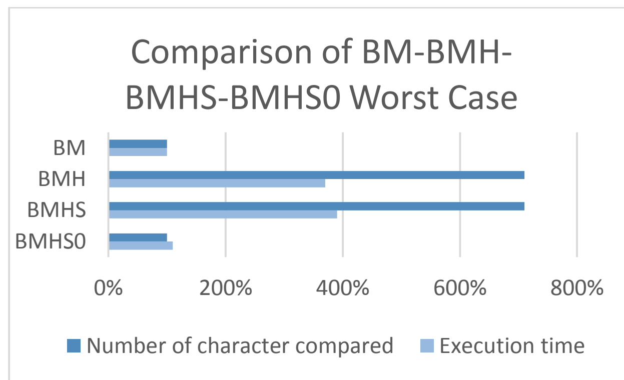
BMHS algorithm with the random pivot method achieved, on average, 400% the execution time of BM algorithm. The original BMHS algorithm runs at 90% the execution time of BM, so the random pivot method causes the process to run up to 4.5 times slower.

The random number used in the algorithm is generated by using the *rand()* function. The function is called after every shift. It turns out that the process of generating a random number, then calculating its modulo, to make sure it is in range, takes a lot of time. The idea might work if the random number generator is as fast as or nearly as fast as assigning a number to a variable. Sadly, this seems improbable in C++ language.

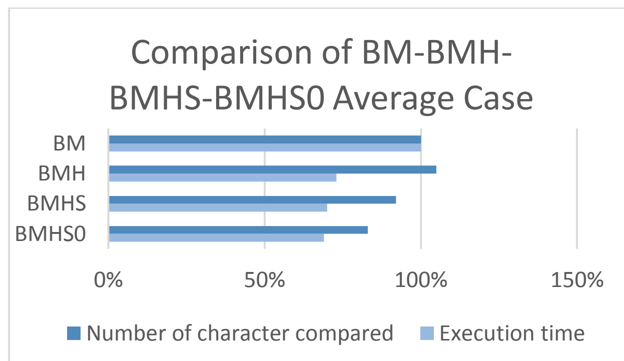
D. The Good Suffix

Disappointed by the worst case results of the previous ideas, the search for an algorithm to match the original BM has gone full circle. If you cannot beat them, join them.

The combination of BM algorithm's good suffix rule would drastically reduce both the number of comparison and execution time in BMHS worst case scenario. The modified algorithm, codenamed BMHS0 (Boyer-Moore-Horspool-0) for going back to its root, is expected to run as fast as BMHS in average case thanks to its capability to jump one character more than BM. However, the good suffix rule requires the searching process to be done sequentially from the rightmost to the leftmost character.



GRAPH 7 *Number of character comparison and execution time of BM-BMH-BMHS-BMHS0 algorithms' worst case*



GRAPH 8 *Number of character comparison and execution time of BM-BMH-BMHS-BMHS0 algorithms' average case*

The example of BMHS0 algorithm would be equal to Table 5 in average case and Table 6 in worst case.

In Graph 7, it could be seen that the amount of compared character in BMHS0 is equal to the BM algorithm. Strangely, the execution time constantly averages slightly higher than BM. A possible cause is that the BMHS 0 algorithm needs to calculate the index of the character that is positioned one index to the right of the rightmost character. As stated before, the time taken to do this is minuscule, but would add up if done repeatedly.

On average case, the algorithm compares less characters than BMHS, and nearly 20% less than the BM algorithm according to Graph 8. The running time does not get much improvement over BMHS algorithm despite having less characters to be checked. This may be caused by the index calculation in the previous paragraph, and also the time taken to compare the jump values generated by the last occurrence table and the good suffix rule.

V. Conclusion

The BMHST is a pretty good optimization that also has its own worst case that performs worse than BM, but it occurs far less often than the regular worst case of BMHS. The random character selection method is actually better in evading the worst case scenario. However, the time it takes to randomize a number in C++ makes the algorithm far slower than other alternatives.

The best optimization is achieved in BMHS0 algorithm that combines BM's good suffix rule and BMHS ability to jump further. This combination drastically decreases the worst case scenario impact on BMHS, while also improving the result on average case for BM. Going back to use the good suffix rule does increase the complexity of the code, but it pays off in the execution time.

VII. Acknowledgment

The author would like to thank Dr. Nur Ulfa Maulidevi, S.T., M.Sc. and Dr. Ir. Rinaldi Munir, M.T. for the comprehension of algorithm strategies through all the lectures and projects in this semester. The author would also like to say thank you to all the family members and friends for their support throughout the making of this paper.

References

[1] R. Boyer and J. Moore, "A Fast String Searching Algorithm", *Communications of the ACM*, vol. 20, no. 10. 1977.
[2] R. Horspool, "Practical Fast Searching in Strings", *Software: Practice and Experience*, vol. 10, no. 6. 1980.
[3] D. Sunday, "A Very Fast Substring Search Algorithm", *Communications of the ACM*, vol. 33, no. 8. 1990.

Declaration

I hereby declare that this paper is of original work, neither an adaptation, nor a translation of any existing paper, and not an act of plagiarism.

Bandung, 17 Mei 2017

Rionaldi Chandraseta - 13515077