Comparing the Performance of Reverse Colussi and Raita in Finding Indonesian Text

Jos Timanta Tarigan, Rahmat Fajar, Amer Sharif
Faculty of Computer Science and Information Technology, Universitas Sumatera Utara
jostarigan@usu.ac.id

Abstract. The availability of various string-matching algorithm raises the concern of finding the one with the best performance. With the growth of digital content, especially a text-based content, using an optimal algorithm is crucial to decrease the cost of performing a task. There are numerous studies using a specific content to evaluate the performance of ones algorithm. In this paper, we chose two commonly used string-matching algorithm; Reverse Colussi and Raita and compare the performance of these algorithm in finding text in Indonesian language. We specifically use Indonesian songs lyrics as our samples. To perform the test, we build and text-based song finder Android application. Our test result shows that Raita outperformed Reverse Colussi in most of the cases. This finding is due to the requirement of Reverse Colusi algorithm to have pre-processing step before performing the search.

1. Introduction
In modern society, the amount of digital content has increased significantly every day. It is easier to create, duplicate, and share the content with the help of internet connected mobile devices. This trend however has caused a need to create a better algorithm to perform various actions such as searching a pattern and extracting information.

There is numerous previous research that focused on observing text-based search algorithm and its performance. Charras et al. published a handbook that contains exact string matching algorithm available [1]. However, as listed by Faro et al., there are numerous new algorithms invented to increase the performance of the available string matching algorithm [2] and the problem that may arise with the current state of string-matching algorithm [3]. There are also various research focus on analyzing the performance of string-matching algorithm on a specific languages, topics, or properties such as Chinese strings [4], Quranic texts [5], and medical terms [6]. In this paper, we focus on performing a search on a text-based content using exact string-matching algorithm. Our research focus on two of the commonly known string-matching algorithm; Reverse Colussi algorithm and Raita algorithm. To help us to perform the test, we build an Android based application intended to search lyrics from a music based on a query entered by user.

2. String Matching Algorithm
There is various application that use string matching algorithm. Currently, string-matching algorithms are categorized into two categories: exact string-matching and inexact string matching (which usually called fuzzy string matching). The only difference between the two is the matching condition. Inexact
string matching allows text similar to a pattern to be treated as a match. The condition of similarity is varied depends on the algorithm used and user’s specific definition.

Before we go through the details on the algorithm, it is important to define a basic knowledge regarding terms and jargon used in this topic. In our paper, a pattern $x$ with size $m$ contains characters $x[0]$ to $x[m-1]$. The text $y$ with size $n$ where the search will be performed contains character $y[0]$ to $y[n-1]$. To avoid redundancy, we assume that $m > n$ and if both pattern and text does not follow this rule, we won’t perform the search and return no value. Moreover, it is important to know that our application does not perform any stemming to the query entered by user. Hence, the result contains the exact string entered by the user.

2.1. Raita Algorithm

Raita algorithm was first introduced by Timo Raita on his paper titled “Tuning the boyer-moore-horspool string searching algorithm” in 1992 [7]. As implied by the title of the paper, this algorithm is an optimization of the previously proposed boyer-moore-horspool algorithm introduced by R. Nigel Horspool in 1980 [8]. The implementation of the algorithm is quite straightforward. The algorithm started by performing a preprocessing phase to build a pattern window. This window will slide through the text during the search phase. The size of the window is $n+1$ which contains each character in the pattern and a value which represents the movement of the window. The value for the last character will be the length of the pattern ($m$). An additional character is added in the end of the window to represent characters that are not exist in the pattern. Table 1 below is an example the window for pattern “alive”.

$$bmBC[i] = m-i$$

where $m$ is the length of the pattern and $i$ is the index of the current character. However, the $i$ is only go from the first character to the second last character. The value for the last character will be the length of the pattern ($m$). An additional character is added in the end of the window to represent characters that are not exist in the pattern. Table 1 below is an example the window for pattern “alive”.

|   | a | l | i | v | e | * |
|---|---|---|---|---|---|---|
| $bmBC[i]$ | 4 | 3 | 2 | 1 | 5 | 5 |

Table 1. An Example of Pattern Window

The next step of the algorithm is to perform the search by sliding the window through the text. The movement of the window is based on the value of the character on the text. The image below shows the three step of sliding the window for pattern “alive” on text “we are alive”.

1. Character $r$ does not match the pattern. Hence, the window is moved 5 steps to the right.

2. Character $i$ does not match the pattern. Hence, the window is moved 2 steps to the right since $bmBC[2]$ for $i$ character is 2.

3. All the character is matched. There is no character left on the text. No need to slide the windows any further.
2.2. Reverse Colussi Algorithm

Reverse Colussi Algorithm is firstly introduced by Livio Colussi in 1994 [9]. This algorithm is a refinement of Boyer-Moore Algorithm. Similar to Raita Algorithm, the process of Rever Colussi contains two part: preprocessing and searching. The first step of the preprocessing is to create a located table. Given the pattern “alive” the located table is as follows:

| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| x[i] | a | l | i | v | e |
| Locc[c] | 3 | 2 | 1 | 0 | -1 |

The next step is to build two other tables, Reverse Colussi Bad Character (rcBC) and (rcGS) Reverse Colussi Good Suffixes. Since we have 5 character in the locc table, we should have 5 table for each of the character.

Table 3. The 5 tables of rcBC

| v | v | 5 | | | |
|---|---|---|---|---|---|
| i | v | 5 | | | |
| l | v | 5 | | | |
| a | v | 1 | | | |
| * | v | 1 | | | |

| v | i | 5 | | | |
|---|---|---|---|---|---|
| i | i | 5 | | | |
| l | i | 2 | | | |
| a | i | 2 | | | |
| * | i | 2 | | | |

| v | a | 4 | | | |
|---|---|---|---|---|---|
| i | a | 4 | | | |
| l | a | 4 | | | |
| a | a | 4 | | | |
| * | a | 4 | | | |

Combining the 5 tables, we’ll end up with this single rcBC table

Table 4. The combined rcBC Table of each character

| Char | Length |
|------|--------|
| v    | 5 5 5 1 1 |
| i    | 5 5 2 2 2 |
| l    | 5 3 3 3 3 |
| a    | 4 4 4 4 4 |
| *    | 5 5 5 5 5 |

We also have to establish an rcGS value of our patter that looks like table 5 below:

Table 5. rcGS Table for patter “alive”

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| p[i] | a | l | i | v | e | |
| link[i] | -1 | -1 | -1 | -1 | -1 | |
| hmin[i] | 0 | 4 | 4 | 4 | 4 | 4 |
| kmin[i] | 0 | 0 | 0 | 0 | 1 | 0 |
Now that we have established the rcGS and rcBC table, we can perform the searching by following this rule:

1. If the character has not been found previously, use value from rcBC.
2. If the character has been found at least once, use value from rcGS.

| 1 | w | e | a | r | e | a | l | i | v | e |
|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   | X |   |   |   |   |   |   |   |

rcBC[r][5] = rcBC[∗][5] = 5. So the window should slide 5 steps

| 2 | w | e | a | r | e | a | l | i | v | e |
|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   | X |   |   |   |   |   |   |   |

rcBC[i][5] = 2. So the window should slide 2 steps

| 3 | w | e | a | r | e | a | l | i | v | e |
|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   | 2 | 3 | 4 | 5 | 1 |

All the character is matched. There is no character left on the text. No need to slide the windows any further.

3. Test and Result

As previously stated, we perform the test by building an Android application that contains lyrics of Indonesian songs. The system contains 500 songs with an average 132 words and 850 characters. The application searching algorithm will go through each song and perform the selected algorithm on the song’s lyric. It is important to notice that the searching process does not end even though the pattern is found. The algorithm will go through the whole lyrics before going to the next song. This condition is important to avoid occurrence property to affect the running time result. Should the search ended after finding the pattern in a song, patterns with high occurrence in the beginning of the songs will tend to have a shorter running time compared to pattern with rare occurrence. Hence, our observation can focus in finding the relation between pattern length and running time in each algorithm.

We perform our test by picking the pattern that appeared at least once on the list. We pick 10 patterns with different length to see how the increment of pattern length affect the running time. We perform the test once for each algorithm and compare both results to find the one with better performance. Figure 1 below is the test result of test.
As shown in the previous figure, it is clear that Raita algorithm is a much better algorithm especially for a long pattern. Our test result shows that after 12 characters, the performance of raita outperforms reverse colussi. This behavior is expected since the search algorithm of reverse colussi is more complex than raita. However, it is also important to notice that reverse colussi has a better performance while searching a shorter pattern.

To validate our test, we also picked a different set of patterns with similar length and perform multiple tests. The result shows a similar pattern to the first test; Raita algorithm outperformed Reverse Colussi algorithm in a longer pattern.

4. Conclusion
In this research, we compare the performance of two exact string-matching algorithm; Raita and Reverse Colussi, in finding pattern in Indonesian language. We perform the test by building an Android-based song’s lyric search application. We collected 500 Indonesian songs as the sample of our test. The test result shows that Raita outperforms Reverse Colussi in longer pattern. We performed multiple tests and the result is similar. It is safe to say that Raita is averagely a better algorithm to perform exact string-matching.

While the current result is adequate to find the better algorithm between the two, further observation is required to find the behaviour of the algorithm based on the pattern property. Based on the description of the algorithm, we can predict that the structure of the pattern may contribute to the result. Pattern and text with repetitive structures may have a different performance compared to the one without any repetition. We have yet to perform this observation and planning to increase the test case to include this property.

Acknowledgment
This research is made possible by the aid of Universitas Sumatera Utara as the institutions of the authors. The author would like to thank the head and staff of "Lembaga Penelitian USU (Research Center of Universitas Sumatera Utara)", Prof. Dr. Erman Munir, MSc, and staff for all the support during this research. The author would also like to thank the Dean of Faculty of Computer Science and Information Technology University of Sumatera Utara, Prof. Dr. Opim Salim Sitompul, M.Sc for their widest to support this work.

References
[1] C. Charras and T. Lecroq, Handbook of exact string matching algorithms. London: Kings College Publ, 2004.
[2] S. Faro and T. Lecroq, “The exact online string matching problem: A review of the most recent results,” ACM Computing Surveys, vol. 45, no. 2, pp. 1–42, Feb. 2013.

[3] S. Faro and T. Lecroq, “The Exact String Matching Problem: a Comprehensive Experimental Evaluation,” arXiv:1012.2547 [cs], Dec. 2010.

[4] Zhou Xueguang and Zhang Huanguo, “Flexible pattern matching algorithm in Chinese strings,” in 2008 27th Chinese Control Conference, Kunming, China, 2008, pp. 610–614.

[5] G. A. Gilkar, S. Hakak, A. Kamsin, M. M. Rahman, and M. Rahman, “An Exact matching approach to enhance retrieval process for Quranic texts,” in Proceedings of the 4th ACM International Conference of Computing for Engineering and Sciences on - ICCES’18, Kuala Lumpur, Malaysia, 2018, pp. 1–4.

[6] C. Lovis and R. H. Baud, “Fast Exact String Pattern-matching Algorithms Adapted to the Characteristics of the Medical Language,” J Am Med Inform Assoc, vol. 7, no. 4, pp. 378–391, 2000.

[7] T. Raita, “Tuning the boyer-moore-horspool string searching algorithm,” Software: Practice and Experience, vol. 22, no. 10, pp. 879–884, Oct. 1992.

[8] R. N. Horspool, “Practical fast searching in strings,” Software: Practice and Experience, vol. 10, no. 6, pp. 501–506, Jun. 1980.

[9] L. Colussi, “Fastest Pattern Matching in Strings,” Journal of Algorithms, vol. 16, no. 2, pp. 163–189, Mar. 1994.