

Application of String Matching and String Similarity for Shape Side Matching

Raden Francisco Trianto Bratadiningrat - 13522091

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
franciscotrianto@gmail.com:

Abstract— String matching is a crucial technique in computer science used extensively to compare and find relationships between strings. This paper presents a novel application of string-matching algorithms for the problem of shape-side matching, where altered-sided shapes are compared based on their side profiles to determine connectivity. The methodology involves converting side profiles into arrays of distances and subsequently into ASCII strings. Exact string matching algorithms, specifically Knuth Morris Pratt (KMP) and Boyer Moore (BM), are employed for efficient and accurate matching, while the Levenshtein Distance is utilized for approximate matching when exact matches are not feasible. Experimental results demonstrate the effectiveness of these approaches in accurately matching the sides of altered-sided shapes. This work not only solves the specific problem of shape-side matching but also establishes a foundation for further exploration in comprehensive shape matching.

Keywords—Shape Side Matching, String Matching, String Similarity, Design Algorithm

I. INTRODUCTION

Shape matching stands as a cornerstone in contemporary computer science, permeating various applications across diverse domains. Particularly pivotal in computer vision, this discipline facilitates object identification and classification within images and videos, underpinning crucial tasks like object recognition and scene understanding. However, conventional methods of shape matching, often reliant on geometric properties and transformations, confront challenges posed by real-world data variability, including changes in orientation, scale, and partial occlusions. In response, this paper proposes a pioneering approach integrating string matching and string similarity algorithms to address these complexities effectively.

Traditional shape-matching techniques, while proficient, can be computationally demanding and sensitive to data variations. This paper seeks to surmount these challenges by harnessing the power of string-matching algorithms like Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM), renowned for their efficacy in text-processing tasks. By incorporating these algorithms, alongside the Levenshtein Distance metric for approximate matching, the proposed methodology aims to offer a robust solution for efficiently and accurately matching the side profiles of altered-sided shapes. This innovative approach

not only resolves the specific problem of shape-side matching but also lays a solid foundation for further exploration in comprehensive shape matching.

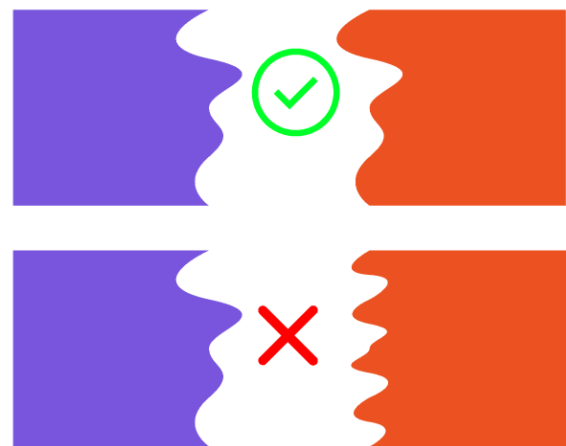


Fig. 1. Shape Side Matching
(source: writer archive)

II. STRING MATCHING

String matching is the process of comparing two different strings to find essential information about the relation between the two strings. It is one of the most used concepts in Computer Science because of just how much information is in a string.

String matching problem itself usually involves two elements [1]:

1. T: text, a long string with the length of n character
2. P: pattern, a string with the length of m character (assuming $m \ll n$), will be searched inside the text

There are two types of string matching: Exact String Matching Algorithms and Approximate String Matching Algorithms.

A. Exact String Matching

Exact String Matching Algorithms are algorithms used to find a perfect match of a pattern in a large string, usually a text

or sequence. Perfect match means that inside the text or sequence exists a pattern with every letter and order the same. Exact String Matching finds one or several instances of the pattern inside the text.

There are many string-matching algorithms. Some examples are Knuth Morris Pratt algorithm, Boyer Moore Algorithm, Rabin Karp Algorithm, Aho-Corasick Algorithm, and many more. But this paper will only use the Knuth Morris Pratt and Boyer Moore Algorithm, which are some of the most popular string matching algorithms.

1) Brute Force Algorithm

String matching using the Brute-Force Algorithm is straightforward but inefficient. It checks each position in the text T to see if the pattern P starts in that position, comparing each letter and shifting one position to the right if it fails. The shift makes the algorithm do a lot of wasteful comparisons, making it very inefficient. In the worst case, the amount of character comparison needed is $m(n-m+1) = O(mn)$.

```

NOBODY NOTICED HIM
1 NOT
2 NOT
3 NOT
4 NOT
5 NOT
6 NOT
7 NOT
8 NOT

```

Fig. 2 Example of Brute-Force Algorithm
(source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>)

2) Knuth Morris Pratt (KMP) Algorithm

KMP is a string-matching algorithm that looks for the pattern in the text in a left-to-right order. KMP is different from Brute Force string matching in terms of shifting. KMP uses a more intelligent shift. It tries to shift the pattern as much as possible to avoid wasteful comparison yet still correctly checks the whole text. To do this KMP uses a Border Function to calculate the best amount of shift.

The border Function preprocesses the pattern to find the matches of the prefix of the pattern with the pattern itself. Border Function $b(k)$ is defined as the size of the largest prefix of pattern P $[0...k]$, which is also a suffix of P $[1...K]$.

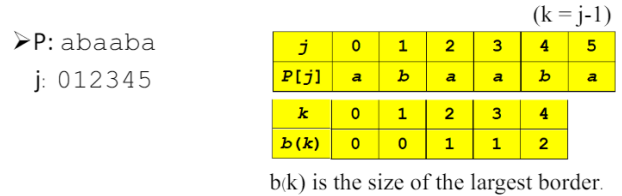


Fig. 3. Example of Border Function
(source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>)

The time complexity for KMP is $O(m + n)$, which is significantly faster than the Brute Force algorithm $O(mn)$. However, KMP does not work so well when the size of the alphabet increases, causing more chance of mismatch.

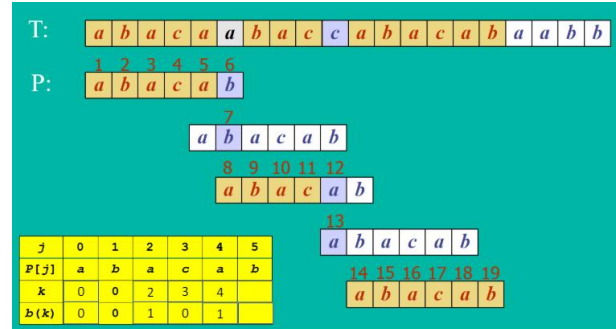


Fig. 4. Example of KMP Algorithm
(source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>)

3) Boyer Moore (BM) Algorithm

Boyer Moore Algorithm or BM is a string-matching algorithm based on two techniques: The looking-glass technique and the character-jump technique. The looking-glass technique finds the pattern P in text T by comparing P backward, starting from the end until the start of P. The character-jump technique is one of the ways to shift intelligently.

There are three cases of character jump-checked in order [1]:

1. If P contains x somewhere, try to shift P to the right until the last occurrence of x aligns with $T[i]$.
2. If P contains x somewhere, but a shift to the right to the last occurrence is impossible, then shift P right by one character to $T[i+1]$.
3. Lastly, if cases 1 and 2 do not apply, then shift $P[0]$ with $T[i+1]$

Like KMP, BM preprocesses the pattern P and the alphabet A to build a function called the last occurrence function $L()$. The last occurrence function, $L(x)$ and x is a letter in A, is defined as the largest index i such that $P[i] = x$, or -1 if no such index exists. The last occurrence function maps all the letters in A to an integer used for shifting.

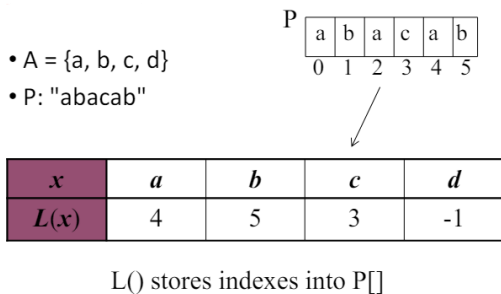


Fig. 5. Example of Last Occurrence Function
(source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>)

In the worst cases, Boyer Moore (BM) has a time complexity of $O(nm + A)$. In retrospect, BM is fast when the alphabet A is large but slow when the alphabet is small. Still, BM is significantly faster than brute-force string matching.

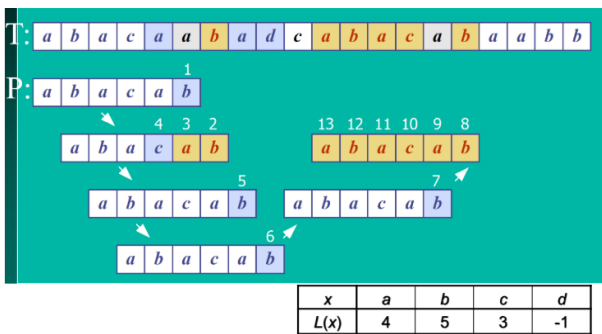


Fig. 6. Example of Boyer Moore Algorithm
(source: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>)

B. Approximate String Matching

Approximate String Matching Algorithms are algorithms used to search the substring of the input string. Suppose given two strings, text T [1...n] and pattern P [1...m]. The task is to find all the occurrences of patterns in the text whose edited distance to the pattern is at most k [2]. A few examples are the Levenshtein Distance, the Hamming Distance, the Bitmap Algorithm, and many others. In this paper, we will only use Levenshtein Distance.

1) Levenshtein Distance

Levenshtein Distance is the smallest number of insertions, deletions, and substitutions required to change one string into another [3]. Insertion is the operation to add a character to string A. Deletion is the operation to remove a character from string A. Substitutions is the operation to replace a character from string A with another character.

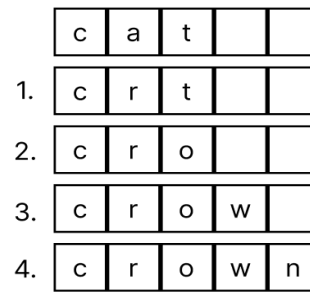


Fig. 7. Example of Levenshtein Distance for string "cat" and "crown"
(source: writer archive)

By using the Levenshtein Distance using iterative with the two-matrix approach, the time complexity needed to find the Levenshtein Distance is $O(m*n)$.

III. STRING SIMILARITY

String similarity is a way to find the similarity value between two different strings. There are many ways to find the similarity value, but for this paper, we will use the Levenshtein Distance to calculate the approximate similarity score of two different strings.

From section II: String Matching, we see that using Levenshtein Distance, we can get the smallest number of operations needed to change one string to the other string.

We can calculate the similarity between two strings of the Levenshtein Distance by using the formula below:

$$\text{similarity} = \{1 - (\text{Levenshtein distance} / \text{max length})\} \times 100$$

IV. SHAPE SIDE MATCHING

Shape matching is the process of comparing geometric shapes to find similarities or differences between them. It is a critical aspect in various fields such as computer vision, pattern recognition, medical imaging, and robotics. Shape matching techniques aim to determine how well two shapes correspond to each other, which can involve identifying exact matches, approximate matches, or partial matches.

In this paper, we will only dive into shape-side matching. Shape-side matching is the process of comparing shapes that originate from a sided shape called an altered-sided shape. For example, a rectangular that is altered on its sides but still has the side profile intact.



Fig. 8. Example of altered-sided shapes
(source: writer archive)

From Fig. 8. You can see that although not clear, all the shapes still have their side profile, but their side profile is now not just a simple straight line, but a pattern instead.

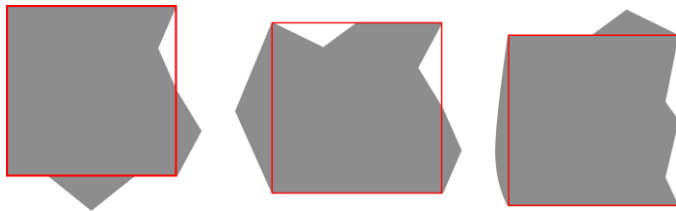


Fig. 9. Side profile of altered-sided shape (source: writer archive)

From Fig. 9. You can see that the altered-sided shapes still contain their side profiles. An altered-sided shape must follow the following rules:

- A shape is an altered-sided shape if it still contains the original shape side profile.
- A shape contains the original shape side profile if all the original shape's corner point still exists at the same position on the altered-sided shape.
- An altered-sided shape must not have a side pattern where the pattern has two different pixels at the same position depending on the side orientation (for example, column or row).

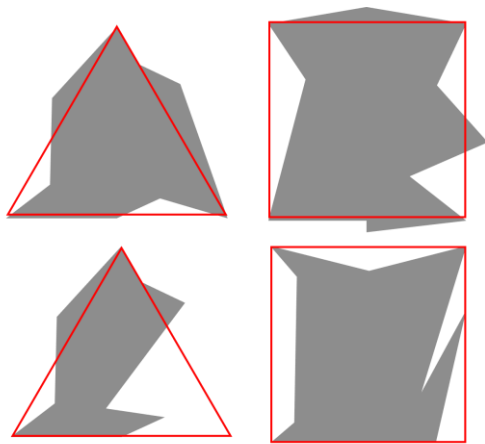


Fig. 10. Side profile of altered-sided shape. (a) Top left shape (b) top right shape (c) bottom left shape (d) bottom right shape (source: writer archive)

From Fig. 10, we see that (a) and (b) are altered-sided shapes, but (c) and (d) fail the rules set. (c) fails because the original shape side profile is not maintained, while (d) fails because the right side has a pattern where there are sections with two pixels of the pattern at the same height.

Shape-side matching involves two altered-sided shapes where one altered-sided shape will be connected to the other on one of the sides. In this paper, we will investigate how we can connect two altered-sided shapes on one of its sides using string matching as the base theory.

V. METHODOLOGY

Given an altered-sided shape A, find another altered-sided shape B, such that A and B can be connected at side SA for shape A and side SB for B, using String Matching. For simplicity, we will only be using an altered-sided shape that originates from a rectangle and has four sides (left, right, top, and bottom).

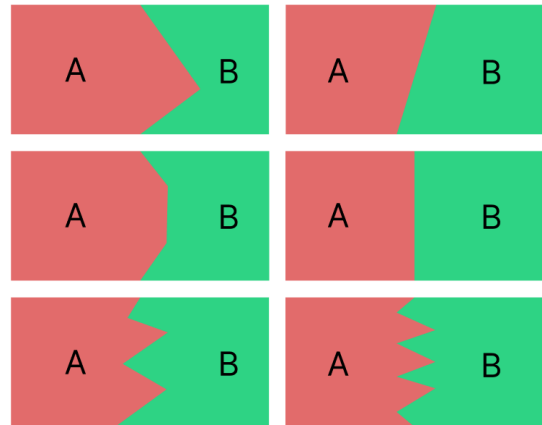


Fig. 11. Example of Shape Side Matching (source: writer archive)

To use String Matching, we first need to find a way to represent a side pattern in the form of a string. For our approach, we will use the distance from the edge of the shape to the first occurrence of a pixel in the pattern. For example, assuming the shapes are in .jpg image format, to calculate the distance of the pixel of the left side of a 4-sided shape, we will check each pixel from the left side of the shape to the first pixel that is not empty.

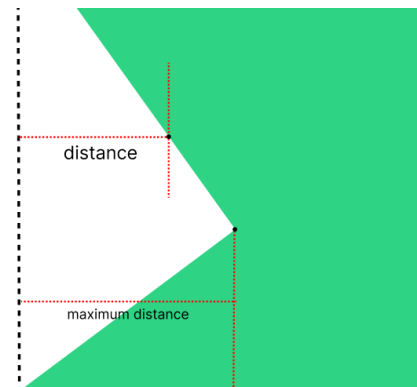


Fig. 12. Calculating the distance of each pixel from the edge (source: writer archive)

The result of calculating the distance is an array of integers. For the example of Fig 12, the distance will be in the form of an array with height amount of length.

Table 1. Converting a shape pattern into an array of distances

Shape	A Sample of Array filled with Distance
-------	--



	0, 0, 1, 2, 3, 3, 4, 5, 5, 6, 7, 8,
	8, 9, 10, 10, 11, 12, 13, 13,
	14, 15, 15, 16, 17, 18, 18, 19,
	20, 20, 21, 22, 23, 23, 24, 25,
	25, 26, 27, 28, 28, 29, 30, 30,
	31, 32, 33, 33, 34, 35, 35, 36,
	37, 38, 38, 39, 40, 41, 41, 42,
	43, 43, 44, 45, 46, 46, 47, 48,
	48, 49, 50, 51, 51, 52, 53, 53,
	54, 55, 56, 56, 57, 58, 58, 59,
	60, 61, 61, 62, 63, 63, 64, 65

Table 2. Converting a shape into an ASCII string

Shape	A Sample of ASCII string
	<pre> ♫☼⊙▶◀↕↕!!¶§—↕↕↕↓→└ └↔↔▲▲▼0123345567889::;< ==>??@ABBCDDEFGGHIJK LLMNNOPQRSSTUVVWX XYZ[[\]]^_`abcdeefghhijklm mnoopqrrsttuvwwxyyz{ }~△ </pre>

We need to make sure all the patterns are one uniform pattern where every array starts from the top or left side of the pattern. We also want to check that the array only stores the flattened pattern. This check will make the stored information all-important to identify a pattern.

Using this array of integers, we can convert it into 8-bit ASCII. But, because 8-bit ASCII can only convert integers from 0 to 255, we need to add a limitation on how big the pattern can be. The maximum distance of the pattern must be below 256. In cases where the maximum distance exceeds the limit, we can manage it by first removing the pattern from the start until the pattern is valid. Secondly, we remove all the patterns after the valid pattern where the pattern is no longer valid. Though this is just one way of solving it, the consequence is that information about the pattern may be lost. The maximum distance is shown in Fig. 12.

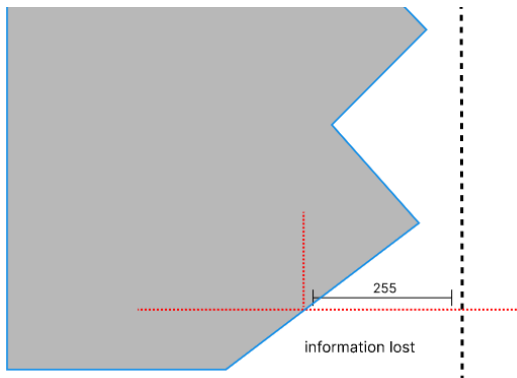


Fig. 13. Information lost from ASCII limitation (source: writer archive)



From Fig 13 you can see that the bottom part of the pattern is lost due to the maximum distance being above the limit of ASCII characters. By assuming the max distance is below 256, we can be sure that no information about the pattern is lost. With that assumption, we can then convert the array of distances into an ASCII string.

Now that we can convert all sides of every shape into a string of ASCII characters, we can use string matching and similarity to solve the problem. The resulting string from shape A and side SA, as well as shape B and side SB, is not yet ready to be used for string matching. As defined in section II, pattern P has a length of m and text T has a length of n where m is much smaller than n ($m \ll n$). Since the pattern converted is the whole image, it means that m and n if using the original ASCII string, could be around the same size, not fulfilling $m \ll n$.

To do string matching, instead of samples of the pattern to be used as the pattern P for string matching. We can do this by simply taking some amount of sample with a certain size. The more the sample used, the more accurate the result. The same with the amount of character used as the pattern also matters for the accuracy of the result. In this paper, we will choose 16 samples or patterns with each sample containing 5 characters. Those amounts are the result of experiments we have conducted.

Using the 16 patterns, we can use the KMP or BM algorithm to compare all the shapes in the dataset, trying to find the best matching shape. The amount of successful KMP or BM string matching is counted for each side of a shape. We keep track of the shape B and side SB that have the greatest number of successful patterns. We can set a threshold for the number of successful patterns so it must be above 6 counts to check if it is truly the best matched. Through experiment, 6 is the best threshold to validate if KMP or BM is successful in finding the best match.

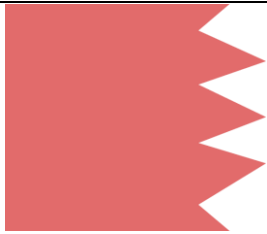

Table 3. Shape Matching using KMP and BM

Shape A side right	Result: Shape B side left
	
Total successful KMP: 7	
Total successful BM: 7	

From Table 3 we can see that both KMP and BM successfully string match 7 patterns out of the total 16 patterns. Since the limit is 6, we can say that both KMP and BM successfully found the best match.

If the string matching fails to find an exact match or a match that is accurate enough, then we can use String Similarity to find the closest match. We will be using Levenshtein Distance to calculate the similarity between two strings originating from shape A side SA and shape B side SB without needing to extract 16 small patterns created for the string matching. Using Levenshtein Distance, we can find the best shape with the highest amount of similarity value using the formula provided in section III.

Table 4. Shape Matching using Levenshtein Distance Similarity







Shape A side right	Result: Shape B side left
	
Similarity Score: 0.534	










By combining both string matching and similarity, using this methodology, we should be able to solve shape-side matching. Note that our approach has many limitations like the limit of 8-bit ASCII characters. Even then we are still able to shape side matches according using string-matching.

VI. TESTING

To assess our theory of solving shape-side matching using string matching and similarity, we implement a program that uses the same methodology to solve the problem. See Appendix A for the source code and documentation.

Table. Testing using the writer's implementation.

No	Input	KMP algorithm	BM algorithm
1.			
2.			

3.			
4.			
5.			

From Table 5 we see that our implementation successfully shows that our methodology is correct. We can use string matching and similarity for shape-side matching. Although the writer's implementation only handles altered-sided shapes that originated from a rectangular, we see that our approach is indeed one of the ways to solve shape-side matching.

VII. CONCLUSION

In this paper, we successfully demonstrated the application of string-matching techniques to the problem of shape-side matching. By representing the side profiles of altered-sided shapes as strings, we were able to leverage both exact and approximate string-matching algorithms to identify matching sides. The use of Knuth Morris Pratt (KMP) and Boyer Moore (BM) algorithms provided efficient and accurate results for exact matches, while the Levenshtein Distance offered a robust method for finding approximate matches when exact matching was not feasible.

Our methodology included converting side profiles into arrays of distances and then into ASCII strings, ensuring uniformity and maximizing the utility of string-matching algorithms. Through experiments, we validated that this approach could successfully match sides of altered-sided shapes, with both KMP and BM algorithms identifying the correct matches when given appropriate patterns.

The success of this approach opens further opportunities to extend the methodology to solve the entire shape-matching problem. By applying string-matching techniques to all sides of a shape and exploring more advanced or alternative encoding methods, we can enhance the accuracy and efficiency of matching complex shapes. This lays the groundwork for future research and practical applications in fields such as computer vision, pattern recognition, and robotics.

In conclusion, the integration of string-matching algorithms into shape-side matching has proven effective and promising. This research not only addresses the specific problem of side matching but also provides a solid foundation for tackling the broader challenge of comprehensive shape matching.

APPENDIX

Appendixes A: documentation of the writer's implementation of Shape Side Matching for rectangular shape origin.

ACKNOWLEDGMENT

The writer would like to thank all IF2211 lecturers, especially Dr. Ir. Rinaldi Munir, M.T. as the lecturer in our class K01 of IF2211 for "Strategi Algoritma" or Algorithm Design Strategies, for teaching and supporting all the students in creating these papers to contribute to the field of Computer Science. I have gained a much better understanding of string matching and string similarities and their application to Shape Matching. I also would like to thank Dr. Ir. Rinaldi, M.T., for providing students with plentiful resources on Algorithm Design Strategies on his website.

REFERENCES

- [1] Geeksforgeeks, Applications of String Matching Algorithms. Retrieved June 11, 2024, from <https://www.geeksforgeeks.org/applications-of-string-matching-algorithms>.
- [2] Munir, Rinaldi, "Pencocokan String (String/Pattern Matching)," IF2211 Strategi Algoritma. Bandung, West Java, 2021. Retrieved June 11, 2024, from <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf>.
- [3] Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999, "Levenshtein distance", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed. 15 May 2019. Retrieved June 11, 2024, from: <https://www.nist.gov/dads/HTML/Levenshtein.html>.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024



Raden Francisco Trianto Bratadiningrat dan 13522091

APPENDIX A

You can find the writer's implementation and documentation below.

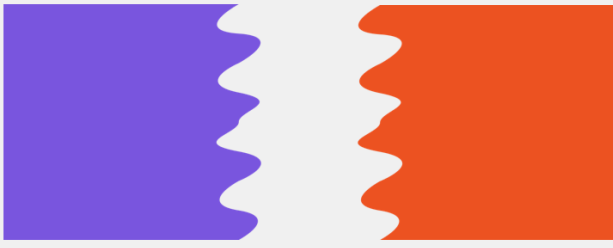
<https://github.com/NoHaitch/Shape-Side-Matching-for-rectangular-shape-origin>

Using KMP search

Shape Side Matching

Select a shape (.png):
Select Shape
right

KMP Search | BM Search
Best match: left-side-7.png
Algorithm used: KMP
Side used: left

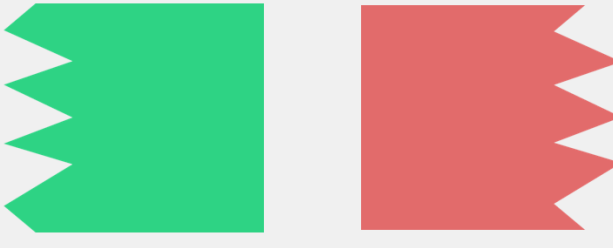


Shape Matching Time: 5.5062 seconds

Shape Side Matching

Select a shape (.png):
Select Shape
left

KMP Search | BM Search
Best match: right-side-6.png
Algorithm used: KMP
Side used: right




Shape Matching Time: 5.5926 seconds

Using BM Search

Shape Side Matching

Select a shape (.png):
Select Shape
right

KMP Search | BM Search
Best match: left-side-4.png
Algorithm used: BM
Side used: left

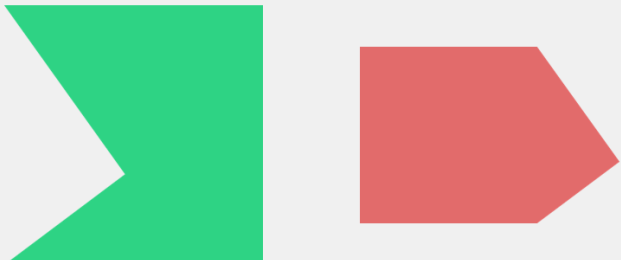


Shape Matching Time: 0.0490 seconds

Shape Side Matching

Select a shape (.png):
Select Shape
left

KMP Search | BM Search
Best match: right-side-1.png
Algorithm used: BM
Side used: right



Shape Matching Time: 5.5309 seconds