Efficient Name Generation Using the Boyer-Moore Algorithm for Meaningful Combinations

Ellijah Darrellshane Suryanegara - 13522097
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): edarrell1202@gmail.com

Abstract— *This paper explores the application of the Boyer-Moore string matching algorithm in the context of generating names with specific meanings. The Boyer-Moore algorithm, known for its efficiency in string searching, leverages two heuristics—the Bad Character Heuristic and the Good Suffix Heuristic—to minimize the number of comparisons required during the matching process. We implement this algorithm to match user-defined keywords against name descriptions stored in a database, aiming to generate names that closely align with desired attributes such as "Strength," "Love," "Hope," "Beauty," and "Wisdom."*

Our results demonstrate that the Boyer-Moore algorithm's ability to efficiently handle large datasets and perform quick, accurate pattern matching makes it an ideal choice for applications requiring high performance in text searching. The generated names provide a meaningful and personalized experience for users, showcasing the practical utility of advanced string matching techniques in creative and data-driven applications.

Keywords—*Boyer-Moore Algorithm*, *String Matching*, *Pattern Matching*, *Name Generation*

I. INTRODUCTION

In the realm of computational linguistics and natural language processing, the challenge of generating names with specific meanings represents a fascinating intersection of creativity and algorithmic precision. This paper delves into the innovative application of string matching techniques to generate names that embody predetermined semantic attributes. The ability to generate meaningful names is not only an intriguing problem from a theoretical standpoint but also has practical implications in various domains, including brand creation, character naming in storytelling, and personalized content generation.

String matching, a fundamental concept in computer science, involves the identification and comparison of substrings within a larger string. Traditionally utilized for tasks such as text searching and pattern recognition, string matching algorithms have evolved to address more complex linguistic challenges. By leveraging these advanced algorithms, it is possible to systematically create names that align with specific phonetic and semantic criteria. This paper explores several methodologies, ranging from simple pattern matching to more sophisticated techniques that incorporate elements of machine learning and natural language understanding.

The significance of generating names with specific meanings extends beyond mere nomenclature. In branding, for instance, a name that resonates with the intended message or evokes particular emotions can greatly enhance a product's marketability. Similarly, in literature and media, a well-chosen name can add depth to a character, making them more memorable and relatable to the audience. This paper aims to demonstrate how string matching can be effectively employed to automate the creation of such meaningful names, thereby blending computational efficiency with the nuances of human creativity. Through detailed analysis and practical examples, we aim to showcase the potential of these techniques in various applications, ultimately contributing to the broader field of computational linguistics and creative automation.

II. THEORETICAL BASIS

A. String Matching

String matching is a fundamental concept in computer science that involves finding occurrences of a substring (often referred to as a "pattern") within a larger string (the "text"). This task is pivotal in various applications, including text editing, search engines, DNA sequencing, and network security. The primary goal of string matching is to efficiently locate all instances where the pattern appears in the text.

At its core, string matching can be approached through several algorithms, each varying in complexity and efficiency. The most straightforward method is the **naive algorithm**, which checks every possible position in the text to see if the pattern matches. While easy to understand and implement, the naive approach can be inefficient, especially for long texts and patterns, as it requires a comparison at every character position.

To address the inefficiencies of the naive algorithm, more sophisticated techniques have been developed. One such method is the Knuth-Morris-Pratt (KMP) algorithm, which preprocesses the pattern to create a partial match table (also

known as the "prefix function"). This table is used to skip unnecessary comparisons, thereby improving the search efficiency. KMP's time complexity is linear in relation to the length of the text and the pattern, making it significantly faster than the naive approach for larger inputs.

Another prominent algorithm is the Boyer-Moore algorithm, which preprocesses the pattern to generate two heuristic tables: the "bad character" and "good suffix" tables. These heuristics allow the algorithm to skip sections of the text, jumping over characters that have already been processed. Boyer-Moore is particularly efficient for longer patterns and texts because it minimizes the number of comparisons needed.

For more advanced and specific applications, such as genomic sequencing, suffix trees and arrays are utilized. These data structures provide a compact representation of all possible substrings of a text, enabling extremely fast substring searches. They are especially useful in scenarios where multiple queries need to be performed on the same text.

B. Boyer Moore

The Boyer-Moore algorithm combines two powerful techniques: the **Bad Character Heuristic** and the **Good Suffix Heuristic**. These heuristics can be utilized independently to search for a pattern within a text, but when combined, they form a highly efficient algorithm. To understand how these two independent methods work together in the Boyer-Moore algorithm, it's helpful to compare it with other string matching algorithms.

In contrast to the naive algorithm, which slides the pattern over the text one character at a time, and the KMP algorithm, which preprocesses the pattern to allow for shifts greater than one, the Boyer-Moore algorithm also preprocesses the pattern. It creates separate arrays for each of the two heuristics. During the search process, the pattern is shifted by the maximum distance suggested by either of the heuristics at each step. This means that the Boyer-Moore algorithm uses the greatest offset recommended by both heuristics to achieve efficient pattern matching.

A unique feature of the Boyer-Moore algorithm is that it starts matching the pattern from its last character rather than the first. In this discussion, we'll explore the Bad Character Heuristic, and the Good Suffix Heuristic will be covered in a subsequent discussion.

<u>*Bad Character Heuristic*</u>

The Bad Character Heuristic is based on a straightforward idea. The character in the text that does not match the current character of the pattern is referred to as the Bad Character. When a mismatch occurs, the algorithm shifts the pattern according to one of two criteria:

a. Case 1 – Mismatch becomes a match

When a mismatch occurs, the algorithm looks up the position of the last occurrence of the mismatched character within the pattern. If the mismatched character is present in the pattern, the pattern is shifted so that this character in the text aligns with its last occurrence in the pattern.

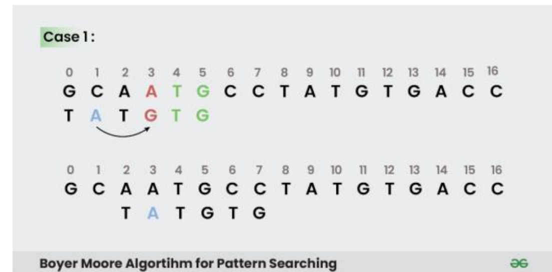


Image 1. Case 1 for Bad Character Heuristic in BM

In the above example, we got a mismatch at position **3**. Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

b. Case 2 – Pattern move past the mismatch character

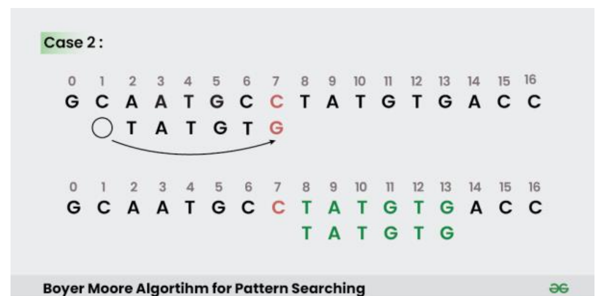


Image 2. Case 2 for Bad Character Heuristic in BM

Here we have a mismatch at position **7**. The mismatching character “C” does not exist in pattern before position 7 so we’ll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because “C” does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

<u>*Good Suffix Heuristic*</u>

The Good Suffix Heuristic is another component of the Boyer-Moore algorithm. Let's consider a substring ttt of the text TTT that matches a substring of the pattern PPP. When a mismatch occurs after this match, the pattern is shifted based on the following criteria:

1. Align another occurrence of ttt in PPP with ttt in TTT.
2. Align a prefix of PPP with the suffix of ttt.
3. Move PPP past ttt.

a. Case 1 – Another occurrence of t in P matched with t in T

The pattern PPP might have multiple occurrences of ttt. In such scenarios, the algorithm shifts the pattern to align the next occurrence of ttt in PPP with ttt in TTT. For example:

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Image 3. Case 1 for Good Suffix Heuristic in BM

In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t ("AB") in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore and not much effective

b. Case 2 – A prefix of P, which matches with suffix of t in T

It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some suffix of t matching with some prefix of P and try to align them by shifting P. For example –

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Image 4. Case 2 for Good Suffix Heuristic in BM

In above example, we have got t ("BAB") matched with P (in green) at index 2—4 before mismatch. But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix "AB" (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t "AB" starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

c. Case 3 – P moves past t

If the above two cases are not satisfied, we will shift the pattern past the t. For example –

i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

Image 5. Case 3 for Good Suffix Heuristic in BM

If above example, there exist no occurrence of t ("AB") in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5. We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.

III. Analysis and Implementation

The name generator application described in the provided code leverages the Boyer-Moore algorithm to match meaningful keywords within name descriptions stored in a database. The primary goal is to find and generate names that closely align with the user-specified meanings.

Steps Involved:

1. **Input Handling**: The user inputs a desired meaning or set of attributes (e.g., "Strength Love Hope Beauty Wisdom") and specifies the gender of the names (male, female, or unisex).

2. **Tokenization**: The input string is tokenized into individual words, which are then used as search patterns.

3. **Database Query**: The application connects to a MySQL database containing names and their corresponding meanings. Depending on the specified gender, an appropriate query is executed to fetch relevant name-meaning pairs.

4. **Pattern Matching**: For each name-meaning pair retrieved from the database, the Boyer-Moore algorithm is used to count how many of the input words (patterns) are present in the meaning.

5. **Filtering and Combination**: The results are filtered to identify the names with the highest number of matching patterns. The best three-word name combinations are then generated based on these filtered results.

6. **Output**: The application outputs the names and their meanings, highlighting those that most closely match the desired attributes.

Breakdown

A. *Bad Character Heuristic*

```python
def bad_char_heuristic(pattern):
    """
    Generate the bad character heuristic table.
    """
    bad_char = [-1] * 256  # Initialize the table with -1

    for i in range(len(pattern)):
        bad_char[ord(pattern[i])] = i  # Fill the table with the last occurrence of each character in the pattern

    return bad_char
```

Image 6. Bad Char Heuristic function

This function initializes a table with 256 entries (for ASCII characters), setting each entry to -1. It then iterates through the pattern, updating the table with the index of the last occurrence of each character. This table is used to determine how far to shift the pattern when a mismatch occurs.

B. *Boyer-Moore Search*

```python
def boyer_moore_search(text, pattern):
    """
    Search for the pattern in the text using the Boyer-Moore algorithm.
    """
    m = len(pattern)
    n = len(text)

    bad_char = bad_char_heuristic(pattern)

    s = 0  # s is the shift of the pattern with respect to the text
    while s <= n - m:
        j = m - 1

        while j >= 0 and pattern[j] == text[s + j]:
            j -= 1

        if j < 0:
            return True  # Pattern found at index s
            s += (m - bad_char[ord(text[s + m])] if s + m < n else 1)
        else:
            s += max(1, j - bad_char[ord(text[s + j])])

    return False
```

Image 7. BM Search function

This function performs the Boyer-Moore search. It initializes the length variables for the pattern and the text, generates the bad character table, and sets the initial shift (s) to 0. It then enters a loop to slide the pattern over the text:

- **Matching from the End**: The algorithm starts comparing characters from the end of the pattern towards the beginning.
- **Mismatch Handling**: If a mismatch is found, the shift is determined by the bad character heuristic. The pattern is moved to align the mismatched character in the text with its last occurrence in the pattern.
- **Pattern Found**: If the entire pattern matches the text, the function returns True, indicating that the pattern is found at the current shift position.

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors*).

C. *Count Matching Words*

```python
def count_matching_words_boyer_moore(text, patterns):
    """
    Count how many words from the input are found in the combination using Boyer-Moore.
    """
    count = 0
    for pattern in patterns:
        if(boyer_moore_search(text, pattern)):
            count += 1

    return count
```

Image 8. Count Matching Words function

This function iterates over each input pattern and uses the Boyer-Moore search function to count how many of the patterns are found in the text (name meaning). The total count of matches is then returned.

D. *Filter Words*

```python
def filterWords(input_words, gender):
    """
    Search for the best three-word name combinations from the database meanings.
    """
    # Connect to the MySQL database
    cnx = mysql.connector.connect(user='root', password='root',
                                  host='localhost', database='names')
    cursor = cnx.cursor()

    # Execute the query to fetch names and meanings
    if(gender == "m"):
        query = "SELECT name, meaning FROM names WHERE gender = 'Boy'"
    elif(gender == "f"):
        query = "SELECT name, meaning FROM names WHERE gender = 'Girl'"
    elif(gender == "u"):
        query = "SELECT name, meaning FROM names WHERE gender = 'Unisex'"

    cursor.execute(query)

    results = []
    max_matches = 0
    for (name, meaning) in cursor:
        if meaning:
            matches = count_matching_words_boyer_moore(meaning.lower(), input_words)
            if(matches > max_matches):
                max_matches = matches
                results = []
                results.append((name, meaning))
            elif(matches == max_matches):
                results.append((name, meaning))

    cursor.close()
    cnx.close()

    return results

```

Image 9. Filter Words function

This function connects to the MySQL database and retrieves names and their meanings based on the specified gender. It then uses the Boyer-Moore algorithm to count the number of matching input words in each meaning. The names with the highest number of matches are stored in a results list.

E. Generate Full Names

```python
def generateFullNames(input_words, gender):
    """
    Generate full names that fit most of the meanings defined
    """
    best_combinations = filterWords(input_words, gender)

    if(len(best_combinations) >= 3):
        full_names = generate_combinations(best_combinations, 3)
        maxMatchedMeaning = 0
        pickedNames = []
        for fullName in full_names:
            allMeaning = ""
            allName = ""
            for name in fullName:
                allName += name[0] + " "
                allMeaning += name[1] + "; "

            matchedMeaning = count_matching_words_boyer_moore(allMeaning.lower(), input_words)
            if(matchedMeaning > maxMatchedMeaning):
                maxMatchedMeaning = matchedMeaning
                pickedNames = []
                pickedNames.append((allName.strip(), allMeaning.strip(), matchedMeaning))
            elif(matchedMeaning == maxMatchedMeaning):
                pickedNames.append((allName.strip(), allMeaning.strip(), matchedMeaning))


        for res in pickedNames:
            print(f"Name: {res[0]}\nMeaning: {res[1]}\nMatches: {res[2]}\n")

    else:
        print("")
        for combination in best_combinations:
            print(f"Name: {combination[0]}\nMeaning: {combination[1]}\n")

```

Image 10. Generate Full Names function

This function generates the final full names based on the best-matching combinations. It filters the names using the Boyer-Moore algorithm and then generates three-word name combinations. It calculates the number of matches for each combination's combined meaning and selects the names with the highest number of matching attributes.

<u>*Test Cases*</u>

- Female
  a. TC1

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) f
Meanings: beauty wisdom wind
Generating names...

Name: Auriah
Meaning: Morning star; aurora of beauty and wind;
```

  b. TC2

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) f
Meanings: flower light hope
Generating names...

Name: Asjeah
Meaning: A healer; light of hope and precious being
```

  c. TC3

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) f
Meanings: water calm god hope
Generating names...
Name: Ahuic Aquielle Chantol
Meaning: Name of the Aztec goddess of the running water in rivers, streams, and waves; A water goddess; A mermaid; A stone placed in water; having
 a calm nature;
Name: Ahuic Atziri Chantol
Meaning: Name of the Aztec goddess of the running water in rivers, streams, and waves; Water jewel; Water goddess; A stone placed in water; having
 a calm nature;
Name: Aquielle Atziri Chantol
Meaning: A water goddess; A mermaid; Water jewel; Water goddess; A stone placed in water; having a calm nature;
```

- Male
  a. TC1

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) m
Meanings: Warrior Strength Hope
Generating names...

Name: Andrew
Meaning: Man; Manly; Brave; Strength; Warrior
```

  b. TC2

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) m
Meanings: Brave Hope Light
Generating names...
Name: Angshu Anirjita Ansu
Meaning: Ray of Light; Hope; Chance; Brave; Strong; Priceless; Unconquered; Enlightened one; Ray of Light; Hope; Radiant Light Beam; Glow;
```

  c. TC3

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) m
Meanings: brave strong hope
Generating names...
Name: Danushith Anik Anirjita
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; Brave; Strong; Priceless; Unconquered; Enligh
tened one;
Name: Danushith Anik Anker
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; A manly harvester; Strong and brave;
Name: Danushith Anik Ansela
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; A strong man who is brave;
Name: Danushith Anik Antarah
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; Hero-like; Brave; Strong;
Name: Danushith Anik Antreas
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; A brave, strong warrior;
Name: Danushith Anik Apasanka
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; Brave; Fearless; Strong;
Name: Danushith Anik Arbaz
Meaning: Powerful; Strong; Brave and Courageous; Mighty; One who is strong and brave like a soldier; Noble; Brave; Strong;
Name: Danushith Anirjita Anker
Meaning: Powerful; Strong; Brave and Courageous; Mighty; Brave; Strong; Priceless; Unconquered; Enlightened one; A manly harvester; Strong and bra
ve;
```

- Unisex
  a. TC1

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) u
Meanings: noble lion god
Generating names...
Name: Adalay Aeriel Arayah
Meaning: God is my refuge; Noble one; Lion of God; God has seen; Lion of God;
Name: Adalay Aeriel Arel
Meaning: God is my refuge; Noble one; Lion of God; Lion of God; Hero;
Name: Adalay Aeriel Areya
Meaning: God is my refuge; Noble one; Lion of God; Lion of God;
Name: Adalay Aeriel Ariea
Meaning: God is my refuge; Noble one; Lion of God; Lioness; Air; Noble; Honorable;
Name: Adalay Aeriel Ariely
Meaning: God is my refuge; Noble one; Lion of God; Noble or lion-like;
Name: Adalay Aeriel Arim
Meaning: God is my refuge; Noble one; Lion of God; Lion of God; Brave; Skilled;
Name: Adalay Aeriel Ariyon
Meaning: God is my refuge; Noble one; Lion of God; Lion of God;
Name: Adalay Aeriel Arlind
Meaning: God is my refuge; Noble one; Lion of God; Golden lion; Noble and courageous;
Name: Adalay Aeriel Aryah
Meaning: God is my refuge; Noble one; Lion of God; Noble, lion;
```

  b. TC2

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) u
Meanings: Celestial Graceful Kind Noble
Generating names...

Name: Aliece
Meaning: Graceful; Kind-hearted; Noble

Name: Aliese
Meaning: Graceful; Noble; Kind
```

  c. TC3

```
==================================================
                 NAME GENERATOR
==================================================
Gender: Male(M) / Female(F) / Unisex(U) u
Meanings: Leader Strong Brave Inspire
Generating names...
Name: Arko Breland Cardin
Meaning: Brave; Strong; Leader; Strong; Brave; Leader; Strong, brave, leader, determined;
```

IV. CONCLUSION

The Boyer-Moore algorithm's integration into the name generator application demonstrates its capability to efficiently match patterns within large texts. By leveraging the bad character and good suffix heuristics, the algorithm minimizes

unnecessary comparisons and accelerates the search process. In this context, the algorithm is used to identify names that closely align with user-defined meanings, resulting in a more personalized and meaningful name generation process. This approach highlights the versatility and efficiency of the Boyer-Moore algorithm in real-world applications, particularly in tasks involving large datasets and the need for rapid, accurate pattern matching

V. ACKNOWLEDGMENT

I am profoundly grateful to God Almighty, creator of the universe, for His guidance throughout the journey of crafting this paper. I would also like to express my sincere gratitude all the following individuals who have played pivotal roles in the completion of this paper:

1. Dr. Ir. Rinaldi Munir, M.T., my dedicated class professor and course coordinator, whose guiding and syllabus have been instrumental in providing invaluable insights that helped put the trajectory of this research endeavor on course.
2. My esteemed colleagues of IF'22, whose collaborative spirit and shared enthusiasm fostered an enriching academic environment, stimulating meaningful discussions and enhancing the overall research experience.
3. Last but not least, my heartfelt appreciation goes to my parents for their enduring support, encouragement, and understanding. Their unwavering belief in my academic pursuits has been a constant source of inspiration, and I am truly grateful for their love and encouragement throughout this academic journey.

VI. REFERENCES

[1] GeeksForGeeks, *Boyer Moore Algorithm | Good Suffix heuristic*. October 2023. Accessed through https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/ on June 11th, 2024.

[2] GeeksForGeeks, *Boyer Moore Algorithm for Pattern Searching*. March 2024. Accessed through https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ on June 11th, 2024.

[3] GeeksForGeeks, *Boyer Moore Algorithm for Pattern Searching*. March 2024. Accessed through https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/ on June 11th, 2024.

[4] Munir, Rinaldi, *Pencocokan String (String/Pattern Matching)*. Institut Teknologi Bandung, 2021. Accessed through https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf on June 11th, 2024.

[5] https://www.momjunction.com/baby-names as data source. Accessed on June 12th, 2024.

LINK OF GITHUB AND YOUTUBE

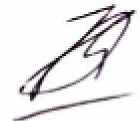
https://github.com/HenryofSkalitz1202/NameGenerator

https://youtu.be/77bVz15UyM4

STATEMENT OF ORIGINALITY

I hereby declare that this paper is an original composition of my own, not of any adaptation or translation from the authored works of others, and free from plagiarism.

Bandung, 12 Juni 2024



Ellijah Darrellshane Suryanegara
13522097