Comparing and Analyzing the Effectiveness of Various String Matching Algorithms Across Different Domains

Research Question - How do different string matching algorithms perform across various application domains, and what factors contribute to their varying effectiveness?

Words: 3514

Contents

1 Introduction

The impacts and applications of string matching algorithms extend far beyond their immediate textual applications; its quotidian use in domains such as bioinformatics, web crawling and security systems make it a pertinent area of study within computer science. There is also constant research being carried out to identify more efficient algorithms, and find ways in which to combine them with other disciplines to carry out certain tasks.

For instance, multiple studies have been recently published performing various comparative analyses of new string matching algorithms [4], and their applicability to a wide range of domains such as biological sequences [12], quantum computing [10], and instrusion detection systems [2]. These findings indicate the modern-day relevance and applicability of string matching algorithms and demonstrate its worthiness as a subject of investigation.

Similarly, this paper seeks to adopt an investigative style by examining **how different string matching algorithms perform across various application domains, and the factors that contribute to their varying effectiveness**.

To formulate a methodology that will directly address this aim, the research question can be broken down into three parts: measuring the performance of the mentioned string algorithms, analyzing the results by making links back to the theory in order to the understand the reasons behind their varying efficiency, and linking these to different domains.

To examine this subject, three popular string matching algorithms were programmed and tested on their ability to carry out three tasks from three different domains. They are then evaluated on a set of criteria to determine their efficiency and applicability. To maintain focus on the research question, the reasoning and rationale behind all the decisions will be explained to demonstrate how they directly address the question.

The choice of this research question is both relevant and justifiable, reflecting the persistent demand for evaluating and optimizing string matching algorithms across a range of applications.

2 Theoretical Background

2.1 Topic Overview

[9] String matching algorithms (often used interchangeably with string or pattern searching algorithms) are a subset of string algorithms. The term is composed of three parts: "string", "matching", and "algorithm". Strings refer to an abstract data type consisting of a sequence of characters. Matching relates to it's function to identify specific patterns within a larger dataset that match a criteria. As for algorithms, these can be defined as a set of instructions designed to carry out a function. These algorithms take in two inputs: a pattern (P), which is a sequence of characters being searched for within a larger sequence, a text (T). Within string matching problems, there exists two general variants [8]:

1. **Find occurences of a pre-defined pattern in a previously unseen dataset**
   Carried out using finite automata models [8] (an idealized machine used to recognize patterns in a given text; it accepts or rejects the input depending on whether the pre-defined pattern occurs in the text) or through the combinatorial properties of strings.

2. **Find occurences of any identifiable patterns in a given text**
   Carried out using finite automata and binary trees.

Linking into the mentioned problem variants, these algorithms can be further classified into two categories [3]: exact, and approximate string matching algorithms. As its name suggests, the former refers to finding occurences of a pattern that match it to the character, whereas the latter allows for some variation or deviation. For the purpose of this investigation, the first problem variant, as well as an exact-string matching approach was chosen for all the algorithms, both for simplicity and control. Three exact string matching algorithms were chosen for comparison based on their popularity: Naive, Knuth-Morris-Pratt, and Rabin-Karp. Prior to their explanations, it is important to note that the following notation will be used:

Table 1: String Matching Algorithm Notation

m	=	Length of P
n	=	Length of T
T_j	:	For an input text T where $0 \leq j \leq n-1$
P_i	:	For an input pattern P where $0 \leq i \leq m-1$

2.2 Naive Algorithm

[6] Also known as the "brute-force" approach, this algorithm compares the first indexes of both P and T. If P_i equals T_j then i and j are incremented and further compared. If P_i reaches $m-1$ then a variable containing the number of occurences of P is incremented and P_i is reset to $P_{i=0}$. On the other hand, if they do not match, then P_i is reset and j incremented. This repeats until $T_{j=m-1}$ is reached. Since there is no pre-processing phase, the only time complexity taken into account is during its matching phase, which is expected to be $O(n \cdot m)$ [8]. The algorithm is better expressed in pseudocode:

Algorithm 1: Naive

```
1  n ← T.length() ;
2  m ← P.length() ;

3  for i ← 0 to n − m do
4  |   while j < m and P[j] == T[i + j] do
5  |   |   j ← j + 1
6  |   end
7  |   if j == m then
8  |   |   output "Pattern found at index" + i
9  |   end
10 end
```

A practical example is also illustrated:

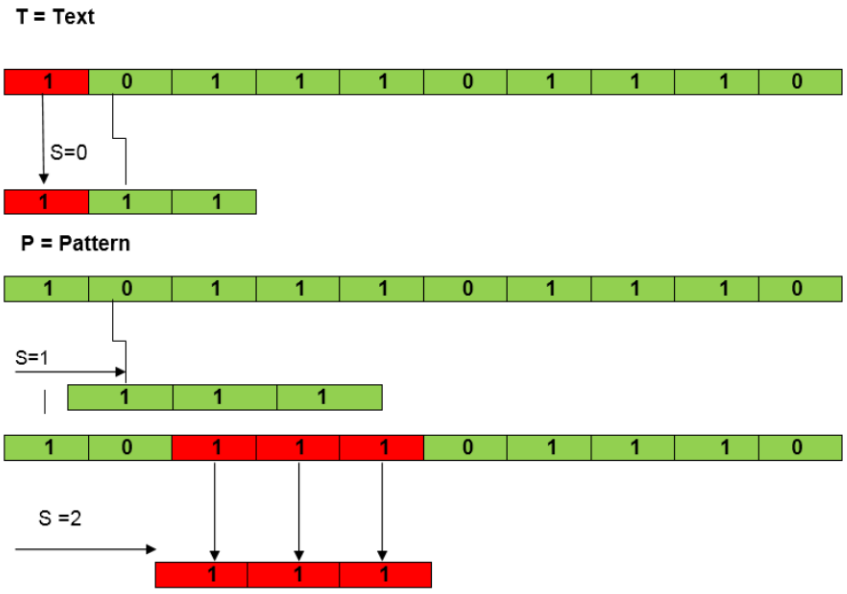


Figure 1: Illustrated Naive [1]

As can be seen, both indexes for the pattern (111) and the text (1011101110) begin at 0 and their first characters are compared. As they match, the indexes are incremented for both P and T and their respective characters matched. As they do not, the index for the pattern is reset to 0 and the text's index incremented. In the depicted example, this is repeated twice more before a match is found – everytime the index of the pattern increases, it matches with the increasing text index.

The best case scenario of this process is when $P_{i=0}$ is not present in T, yielding an $O(n)$ number of comparisons, making it particularly advantageous for solving problems with small n and m values. When larger sets are used however, the algorithm's efficiency significantly deteriorates. The worst case scenario for instance would be when all the characters for P and T are the same ($P_{i=0} \dots P_{m-1} = T_{j=0} \dots T_{n-1}$) or when all the characters for both P and T are the same except for their last characters (($P_{i=0} \dots P_{m-2} = T_{j=0} \dots T_{n-2}$) $\wedge$ ($P_{m-1} = T_{n-1}$)). This would yield an $O(m \cdot (n - m + 1))$ number of comparisons.

2.3 Knuth-Morris-Pratt Algorithm

[8] [5] This algorithm differs from the Naive approach by implementing an LPS table as part of the pre-processing phase. Its purpose is to keep track of the comparisons made; after a mismatch, it is used to calculate where to begin the next match without needing to reset P. The table is constructed by determining the largest prefix of P that matches its largest suffix – the main idea is to identify how many characters can be skipped, by not matching characters that have already been calculated to match (eliminating redundancy). Although appearing complicated, this can be better explained through pseudocode and a practical example represented through a truth table. Upon initalization, the LPS table is an array of 0s of length m. For demonstration purposes, P will equal "$ABABA$":

Index (i)	Pattern Character (P_i)	Length (of Suffix-Prefix match)	LPS Value ($LPS[i]$)
0	A	0	0
1	B	0	0
2	A	0	1
3	B	1	2
4	A	2	3

Table 2: Truth Table for Pattern "ABABA"

Algorithm 2: CalculateLPSArray

```
length ← 0 ;
LPS ← m ∗ [0] ;

while i < m do
    if P[i] == P[m] then
        length ← length + 1;
        LPS[i] ← 0;
        i ← i + 1;
    end
    else if length ≠ 0 then
        length ← LPS[length − 1];
    end
    else
        LPS[i] ← 0;
        i ← i + 1;
    end
end
return LPS
```

Afterwards, the main phase involves applying the calculated LPS table on T. The indexes i and j both begin at 0, and are incremented while P_i matches T_j. When a mismatch occurs, it becomes apparent that the characters in P match with the characters of T up to the mismatch ($P_{0...(i-1)} = T_{j-i...j-1}$). Moreover, from the LPS table, it is known that $LPS[i-1]$ represents the length of the longest part of the prefix and suffix of P. With these pieces of information, it can be concluded that the characters in $P_{0...i-1}$ do not need to be checked with $T_{j-i...j-1}$ since it is already known that they match. Thus, these can be skipped in both P and T. The exact process is presented in pseudocode here below:

Algorithm 3: KMP Match

```
n ← T.length() ;
m ← P.length() ;
LPS ← CalculateLPSArray(P) ;
i ← 0 ;
j ← 0 ;

while j < n do
    if P[i] == T[j] then
        i ← i + 1;
        j ← j + 1;
    end
    if i == m then
        output "Pattern found at index" +(j − i);
        i ← LPS[i − 1];
    end
    else if j < n and P[i] ≠ T[j] then
        if i ≠ 0 then
            i ← LPS[i − 1];
        end
        else
            j ← j + 1;
        end
    end
end
```

A practical example is also illustrated:

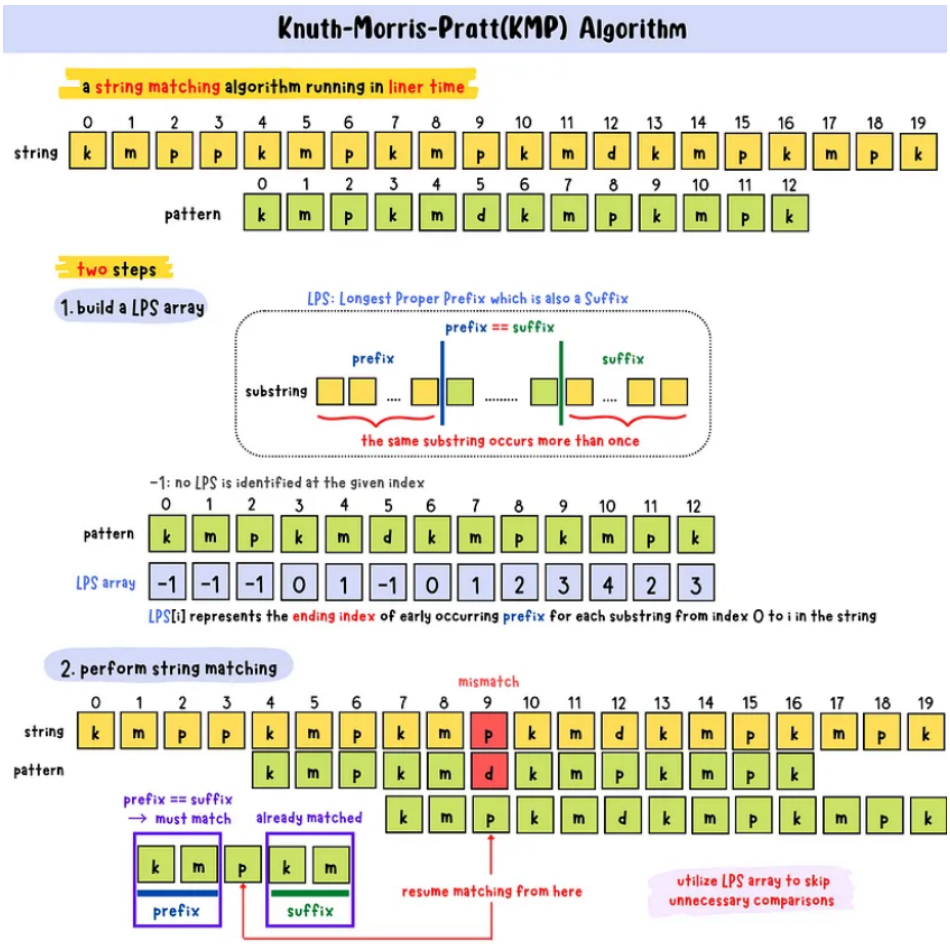


Figure 2: Illustrated KMP [11]

Firstly, the pattern is pre-processed to create an LPS table. This works by iterating through each character of the pattern, and determining its earliest occurrence within it. For example, beginning with "k", it is its first instance in the pattern and so there is no other earliest occurrence, so it is registered in the LPS table as -1, to indicate that there are not any. This repeats for the next 2 characters. Upon reaching index 3 with "k", it is noticed that its earliest instance now exists and is found at index 0, and so it is registered in the LPS table as so. This process continues for the rest of the pattern. Afterwards, the "longest proper prefix which is also a suffix" is found by taking the longest sub-list of consecutive LPS values. In this case it would be indexes 6 to 1 (kmpkmp). This array is then used as part of the string matching to skip the unnecessary comparisons (in the example, since it is known that the prefix equals the suffix, when kmpkmd is reached, the next 7 characters can be skipped, since based on the LPS table they are known not to

match).

By skipping the unecessary character comparisons, the algorithm proves to be much more efficient than the Naive approach. This is seen in its worst-case time complexity, which is $O(n + m)$ (best-case time complexity remains the same). As for the time complexities in the pre-processing and matching phases, they are $O(m)$ and $O(n)$ respectively.

2.4 Rabin-Karp Algorithm

[7] The most similar out of the two to the Naive approach, the Rabin-Karp algorithm has a similar process of checking every substring for a match. Unlike the Naive approach however, this algorithm compares "hash values". These are calculated using rolling hash functions, which allow for the removal and introduction of characters to the calculation, eliminating the need to recalculate the value entirely from scratch. To perform the hashing process on a substring, the following steps are carried out:

Rolling Hash Function

1. The base (b) and modulus (p) values are chosen (usually prime numbers).
2. A hash value is initialised to 0 and the initial hash value of P calculated (for each character in P, its contribution is added to the hash value as $c \cdot (b^{m-i-1})\%p$).
3. The hash value for the first substring of length m in T is calculated.
4. Everytime j is incremented, the contribution of the leftmost character is removed, the contribution of the rightmost character added.
5. Compare the text and pattern matches each time; if they match then it is a **potential match** (rolling function outputs can be the same for different inputs). To verify the match, a character match is performed.

The time complexity for the pre-processing phase (calculating P's hash value) is $O(m)$. As for the time complexities in the matching phase, they are $O(m \cdot n)$ for the worst case

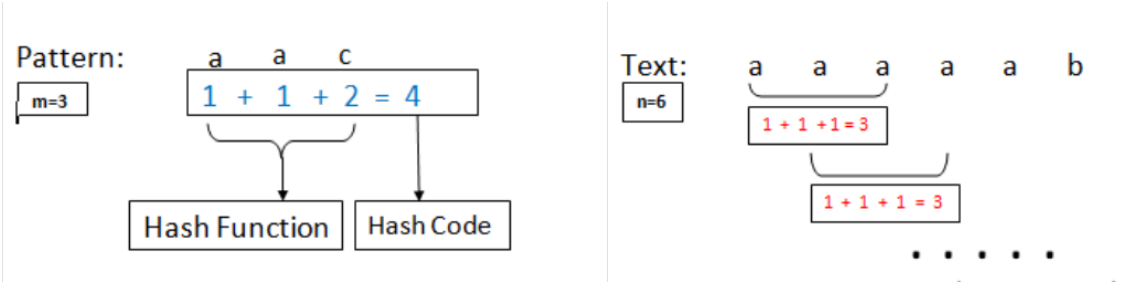
scenario and $O(n+m)$ for the expected case. Although this algorithm maintains the simplicity of the Naive algorithm all the while improving its efficiency, it has a disadvantage that, depending on the specific T case, may impact its efficiency – a spurious hit. This refers to the case where a hash value match is made between P and T but the characters do not match, increasing the time complexity. This can however be minimized by implementing a robust hash function.

The pseudocode for the algorithm is presented here below:

Algorithm 4: Rabin Karp

```
1  n ← T.length() ;
2  m ← P.length() ;
3  h ← 1 ;
4  hashT ← 0 ;
5  hashP ← 0 ;

6  for i ← 1 to m − 1 do
7  |  h ← (h · b)%p ;
8  end

9  for i ← 0 to m − 1 do
10 |  hashP ← (b · hashP + P[i])%p ;
11 |  hashT ← (b · hashT + T[i])%p ;
12 end

13 for i ← 0 to n − m do
14 |  match ← TRUE ;
15 |  for j ← 0 to m − 1 do
16 |  |  if P[j] ≠ T[i + j] then
17 |  |  |  match ← FALSE ;
18 |  |  end
19 |  end
20 |  if match == TRUE then
21 |  end
22 |  output "Pattern found at index " +i;
23 |  if i < n − m then
24 |  |  hashT ← (b · hashT + T[i]) + T[i + m]%p ;
25 |  |  if hashT < 0 then
26 |  |  |  hashT ← hashT + p ;
27 |  |  end
28 |  end
29 end
```

A practical example is also illustrated:



(a) Pattern Hash Code (b) Comparison With Text

Figure 3: Illustrated Rabin Karp [13]

To aid with comprehension, the rolling hash function used to calculate hash values is replaced with a simplified hashing function: each character is converted to its numerical position within the alphabet (e.g. a = 1, b = 2). Since it is known that the pattern length is $m = 3$, the hash values of groups of 3 characters are calculated from the text each time. Using the predefined assigned hash values for the characters, the pattern's hashcode is calculated to yield 4. This means that for each sub-group within the text that is examined, its hashcode should yield 4 to be deemed a potential match. In the example, the hashcode for the first group examined (aaa) is calculated (3). Since they do not match, the second sub-group is checked (aaa). This continues up until the end of the text, where no match is found.

3 Experimental Methodology

3.1 Measuring Performance

To measure the detailed algorithms' performance, their mentioned pseudocode algorithms were programmed in *Python* 3.12. Python was chosen due to it's wide array of third-party modules, three of which are used as part of the implementation: *timeit*, *matplotlib*, and *random*. *timeit* is used to measure the algorithms' runtimes for each algorithm, *matplotlib* is used to generate the graphical plots, and *random* is used to generate the patterns. An experimental methodology was chosen as their is limited secondary data available due to the high specificity of the task. Furthermore, this methodology provides

increased control over variable manipulation. Using their implementations, the following variables will be measured:

1. **Dependent Variables**

   Pre-processing runtime (ms) - The amount of time taken to prepare the data prior to matching.

   Matching runtime (ms) - The amount of time taken to search a pattern P in a text T.

   Total number of comparisons - The total number of references made to the text (until all instances of the pattern are found)

   Total number of comparisons with unmatching characters - The total number of references made to the text for characters that do not match (until all instances of the pattern are found).

2. **Control Variables**

   Language - Depending on whether the language employs a compiler or an interpetor, runtime speeds can differ. Moreover, the type of programming language (procedural, functional, etc.) can have an impact on how the code is run, further impacting runtime speed.

   Pattern - More specifically pattern length and occurrence within the text. By controlling this factor, it is ensured that variations in number of comparisons and runtime are due to the algorithms themselves.

   Prime Number - An additional input is required for Rabin-Karp's algorithm - a prime number (q). For all trials q will equal to 101 to limit its' effect on the results.

The algorithms are then evaluated on their performance based on a set of constructed criteria:

Criteria	Testing
Time complexity	Analyzing program structure.
Space complexity	Analyzing program structure.
Simplicity	Subjective qualitative judgement.
Applicability	More specifically **pattern length** and **occurrence** within the text. By controlling this factor, it is ensured that variations in number of comparisons and runtime are due to the algorithms themselves.

Table 3: Evaluation Criteria

3.2 Dataset

Two pieces of primary data were collected for the experiment, the text and the patterns to match. The same text (3000 character long English text) was used across tests for all three algorithms. As for the patterns, 30 were obtained (generated pseudorandomly from text) each with a length of m from 1 to 20 to yield a total of 600 data points. The high data yield will allow for more reliable and justified judgements on trends to be made. The choice of 30 patterns was made as it was found to be an appropriate quantity for a text of this length. As for the 20 lengths of m, this was done as it would provide a wide range of results to observe a trend without compromising on runtime. To account for edge cases, one pattern not found within the text was included.

3.3 Programs

3.3.1 Pseudorandom Pattern Generator

```
import random

patterns = []

for pattern_num in range(30):
  for pattern_length in range(1,21):
    random_initial_position = random.randint(0, 2999)
    new_pattern = text[random_initial_position : random_initial_position
                                        + pattern_length]
    patterns.append(new_pattern)
```

3.3.2 Conducting Matches

Total and unmatching matches were calculated within the actual algorithm functions.

```
import timeit

patterns = sorted(patterns, key=len)
q = 101

kmp_preprocessing_runtimes = []
rabin_karp_preprocessing_runtimes = []

naive_runtimes = []
kmp_runtimes = []
rabin_karp_runtimes = []

total_matches_naive = []
total_matches_kmp = []
total_matches_rabin_karp = []

unmatching_matches_naive = []
unmatching_matches_kmp = []
unmatching_matches_rabin_karp = []

for pattern in patterns:
  # Preprocessing runtimes
  start = timeit.default_timer()
  computeLPSArray(pattern, len(pattern), [0]*len(pattern))
  stop = timeit.default_timer()
  runtime = stop - start
  kmp_preprocessing_runtimes.append(runtime)

  start = timeit.default_timer()
  rolling_hash(pattern)
  stop = timeit.default_timer()
  runtime = stop - start
  rabin_karp_preprocessing_runtimes.append(runtime)

  # Matching runtimes
  start = timeit.default_timer()
  kmp(pattern, text)
  stop = timeit.default_timer()
  runtime = stop - start
  kmp_runtimes.append(runtime)

  start = timeit.default_timer()
  naive(pattern, text)
  stop = timeit.default_timer()
  runtime = stop - start
  naive_runtimes.append(runtime)

  start = timeit.default_timer()
  rabin_karp(pattern, text, q)
  stop = timeit.default_timer()
  runtime = stop - start
  kmp_runtimes.append(runtime)
```

3.3.3 Generating Graphical Plots

Pattern length vs runtimes:

```
\begin{singlespace}
import matplotlib.pyplot as plt

pattern_lengths = [len(i) for i in patterns]

plt.figure(figsize=(8, 6))
plt.scatter(pattern_lengths, naive_runtimes, marker='.', label="Naive")
plt.scatter(pattern_lengths, rabin_karp_runtimes, marker='.', label="
                                    Rabin Karp")
plt.scatter(pattern_lengths, kmp_runtimes, marker='.', label="KMP")

plt.title('Pattern Length vs Runtimes')
plt.xlabel('Pattern Length')
plt.ylabel('Runtime (ms)')

plt.legend()
plt.grid()
plt.show()
\begin{singlespace}
```

Barchart for preprocessing runtimes:

```
\begin{singlespace}
avg_preprocessing_rabin_karp = sum(rabin_karp_preprocessing_runtimes) /
                                    600
avg_preprocessing_kmp = sum(kmp_preprocessing_runtimes) / 600

algorithms = ['Rabin-Karp', 'KMP']
preprocessing_runtimes = [avg_preprocessing_rabin_karp,
                                    avg_preprocessing_kmp]

fig, ax = plt.subplots()
bar_container = ax.bar(algorithms, preprocessing_runtimes, color=['y', '
```

```
                                    g'])
ax.set(ylabel='runtime (ms)', title='Average Preprocessing Runtimes for
                                    Rabin-Karp and KMP')
ax.bar_label(bar_container, fmt=lambda x: f'{x}')
\begin{singlespace}
```

Pattern length vs total number of comparisons:

```
\begin{singlespace}
plt.figure(figsize=(8, 6))
plt.scatter(pattern_lengths, total_matches_naive, marker='.', label="
                                    Naive")
plt.scatter(pattern_lengths, total_matches_rabin_karp, marker='.', label
                                    ="Rabin Karp")
plt.scatter(pattern_lengths, total_matches_kmp, marker='.', label="KMP")

plt.title('Pattern Length vs Total Number of Comparisons')
plt.xlabel('Pattern Length')
plt.ylabel('Total Number of Comparisons')

plt.legend()
plt.grid()
plt.show()
\begin{singlespace}
```

Pattern length vs number of unmatching comparisons:

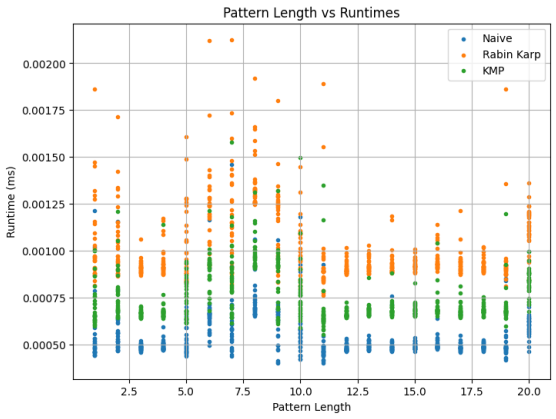
```
\begin{singlespace}
plt.figure(figsize=(8, 6))
plt.scatter(pattern_lengths, unmatching_matches_naive, marker='.', label
                                    ="Naive")
plt.scatter(pattern_lengths, unmatching_matches_rabin_karp, marker='.',
                                    label="Rabin Karp")
plt.scatter(pattern_lengths, unmatching_matches_kmp, marker='.', label="
                                    KMP")

plt.title('Pattern Length vs Number of Unmatching Comparisons')
```

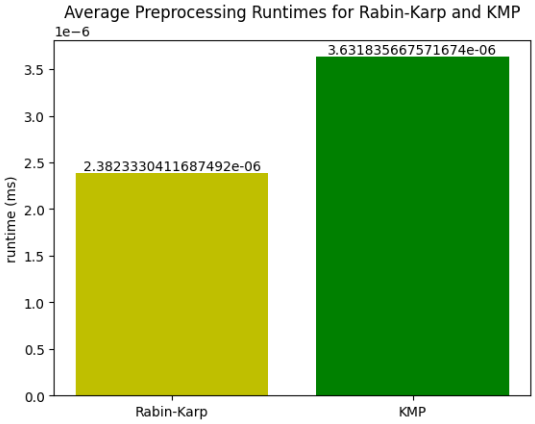
```
plt.xlabel('Pattern Length')
plt.ylabel('Number of Unmatching Comparisons')


plt.legend()
plt.grid()
plt.show()
\begin{singlespace}
```

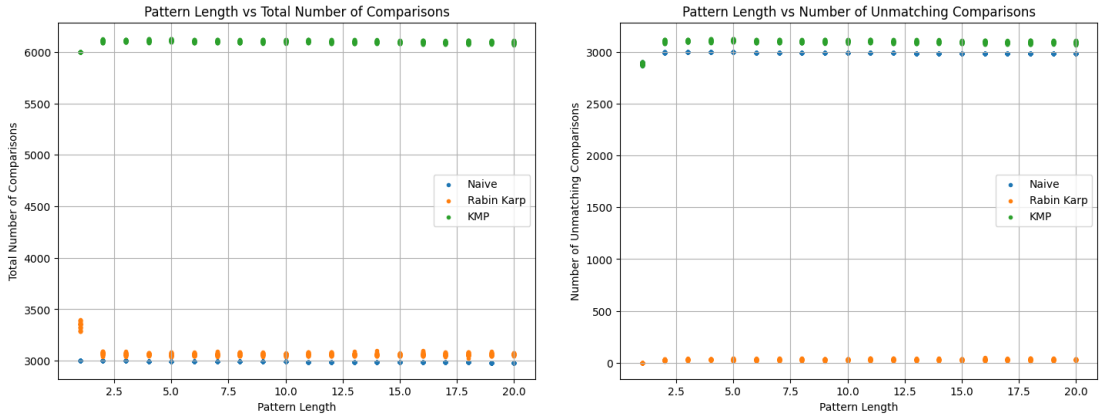
4 Experimental Results

4.1 Graphical Presentation

Since the number of datapoints is too high to be presented as a table, only graphical representations will be presented. These allows for a more intuitive and visual view of the results which in turn aids in identifying patterns and trends. As demonstrated with the programs, these produced the four following graphs which are presented below in their most appropriate presentation forms:







Each of the graphs above (except the barchart) depict the three algorithms' performance based on the examined dependent variables. The plots are depicted as scatter plots as no observable trend was yet inferred. It should also be noticed that their is no third bar to the barchart as the Naive algorithm does not have a preprocessing stage.

4.2 Data Analysis

This section will be structured by algorithm; in each sub-section, the algorithm's performance depicted on each of the four plots will be examined, as well as its trend and how it can be explained. Afterwards, it will be judged against the criteria and be considered with the examined performance to make an overall judgement that will be applied to the three domains: DNA sequencing, plagiarism detection and search engines.

4.2.1 Naive Algorithm

Firstly, it should be noticed across the runtimes that all three algorithms seem to spike in runtime around the 7.5 length region. This may be an anomaly that might have occurred due to the specific characteristic of that particular pseudorandomly generated pattern. Out of the three, the Naive algorithm performed the best, with the lowest average runtime. Considering the simplicity of it, this performance is unexpected, and might be attributed to a the dataset - this can be examined further by using a wider set of data to confirm its unusual performance. This unexpected trend continues throughout the third plot, with it having an average total number of comparisons lying close to 3000, which

remains stable throughout, demonstrating a linear trend with a gradient of 0. In the number of unmatching comparisons, it performs marginally better than KMP, however in comparison to Rabin Karp it performs much worse, and this can be attributed to its brute force method.

In comparison with the criteria, its time and space complexity are $O(n - m + 1)$ and $O(1)$ respectively. This will be put into perspective with the other algorithms at the end of the analyses. Its simplicity out of the three is already known to be the most. As for its application to domains, it was demonstrated to perform well with smaller datasets, however the areas of applications generally concern themselves with massive databases, hence it would not be appropriate to use this algorithm on any of these applications.

4.2.2 KMP Algorithm

For the matching runtimes, KMP is relatively in the middle compared to the two others, with its trend following the same similar spikes. Its preprocessing time did not follow the same trend, with it performing 52.5% worse than Rabin Karp. Its total number of comparisons was also significantly higher than the other algorithms, sitting at an average close to 6000. The final plot also depicts the same trend, with its average number of unmatching comparisons being marginally above 3000. These results can be explained in the opposite way to Naive - KMP's performance improves in comparison to other algorithms as the pattern lengths increase, however with small datasets it is not the most efficient.

This is further justified in its time and space complexities which are both $O(m)$ for the preprocessing stage, and $O(n)$ with $O(1)$ for the matching stages. Its simplicity can be examined in two areas: the LPS table and the matching. The LPS table is relatively understandable and can be easily visualised. The latter however is more complicated, as it is more difficult to imagine how the LPS table fits onto the text to skip comparisons. Its notable characteristic of the LPS table, aswell as its suggested improved performance with large datasets renders it a more appropriate method to use with the mentioned domains.

4.2.3 Rabin Karp Algorithm

For this last algorithm, it is first seen that it spends the most amount of time in the matching stage, with it having the worst runtime. Its average preprocessing runtime is much improved in comparison to KMP's and its matching one, which can be attributed to the low time cost of the hashing function. In the last two plots, it can be seen to excel in terms of the number comparisons. The average total lies around 3000 which is close to Naive, and it has a negligible number of unmatching comparisons, which is the most ideal.

Overall, Rabin Karp represents the ideal compromise between performance and the other evaluation criteria. It has a time and space complexity of $O(m)$ and $O(1)$ in the preprocessing stage, and an average of $O(m+n)$ and $O(1)$ in the matching stage. As was demonstrated, it works well with smaller datasets, and its algorithm design suggests it would be the same with larger ones. The only realy difficulty with its comprehension is the hash function, however this can be visualised for better understanding. Its performance however may vary depending on characteristics of text, which will vary the number of spurious hits obtained. However, the general results demonstrate stability, going in hand with reliability. This links to the domains, as its demonstrated characteristics are ideal for applications areas of this genre.

5 Conclusion

Linking the examined information together, the evaluated criteria for each algorithm is presented here below:

Algorithm	Naive	KMP	Rabin Karp
Preprocessing Time Complexity	N/A	$O(m)$	$O(m)$
Preprocessing Space Complexity	N/A	$O(m)$	$O(1)$
Matching Time Complexity	$O(n-m+1)$	$O(n)$	$O(m+n)$
Matching Space Complexity	$O(1)$	$O(1)$	$O(1)$
Simplicity	3	2	2
Applicability	1	2	3

Table 4: Evaluated Criteria

Overall, taking into consideration all the criteria points, Rabin Karp is the best performer across all these factors, followed by KMP and Naive. This directly addresses and answers the research question, as various tests were conducted on the algorithms to determine performance, the obtained results were graphed and analysed by linking back to the theory to examine the underlying factors for their varying performances. Finally, these were linked to the three domains, to understand how and whether they should be applied to these areas.

6 Further Research Opportunities

One research opportunity concerns the q variable (prime number) that was controlled in the experimental methodology. It's exact influence on the results was not determined, so a suggestion for a further investigation would be exploring how varying the magnitude of q affects the 4 dependent variables. Another research opportunity would be investigating more string matching algorithms and evaluating them, as having a broader view of a wider variety of algorithms can help in making better judgements as to which should be chosen for which specific purpose or domain.

References

[1] URL: https://www.javatpoint.com/daa-naive-string-matching-algorithm.

[2] K Azarudeen et al. “A novel approach for pattern string matching in Intrusion detection system”. In: *Journal of Physics: Conference Series* 1916.1 (2021), p. 012007. DOI: 10.1088/1742-6596/1916/1/012007.

[3] eswithareddy. *Applications of string matching algorithms*. Nov. 2022. URL: https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/.

[4] Margaret Gao et al. “Comparisons of classic and quantum string matching algorithms”. In: *Proceedings of the 4th International Conference on Advanced Information Science and System* (2022). DOI: 10.1145/3573834.3574498.

[5] GeeksforGeeks. *KMP algorithm for pattern searching*. May 2023. URL: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/?ref=lbp.

[6] GeeksforGeeks. *Naive algorithm for pattern searching*. Oct. 2023. URL: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/?ref=lbp.

[7] GeeksforGeeks. *Rabin-Karp algorithm for pattern searching*. Sept. 2023. URL: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/?ref=lbp.

[8] Georgy Gimelfarb. *String matching algorithms - auckland*. URL: https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf.

[9] harendrakumar123. *Introduction to pattern searching - data structure and algorithm tutorial*. May 2023. URL: https://www.geeksforgeeks.org/introduction-to-pattern-searching-data-structure-and-algorithm-tutorial/.

[10] Thierry Lecroq. *Optimal-Hash Exact String Matching Algorithms*. 2023. arXiv: 2303.05799 [cs.DS].

[11] Claire Lee. *Knuth-Morris-Pratt(KMP) algorithm String matching*. Oct. 2022. URL: https://yuminlee2.medium.com/knuth-morris-pratt-kmp-algorithm-string-matching-fb2a3ec6d682.

[12] Muhammad Yusuf Muhammad et al. “COMPARATIVE ANALYSIS OF BIT-PARALLEL STRING PATTERN MATCHING ALGORITHMS FOR BIOLOGICAL SEQUENCES”. In: *Operational Research in Engineering Sciences: Theory and Applications* 6.1 (Apr. 2023). URL: https://oresta.org/menu-script/index.php/oresta/article/view/554.

[13] Bhakti Thaker. *Rabin-Karp, the string matching algorithm.* Sept. 2019. URL: https://bhaktithaker.medium.com/rabin-karp-the-string-matching-algorithm-ca0ba1f8e5fa.