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: not enough

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 [5], and their applicability to a wide range of domains such as biological sequences [6], quantum computing [7], and instrusion detection systems [8]. 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**. 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.

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 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.

2 Theoretical Background

2.1 Topic Overview

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

1 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 [2]:

1. **Find occurences of a pre-defined pattern in a previously unseen dataset**

   Carried out using finite automata models [4] (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 [1]: 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

[9] 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 $O(n \cdot m)$ [10]. The algorithm is better expressed in pseudocode:

Algorithm 1: Naive

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

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 ($P_{i=0} \ldots P_{m-1} = T_{j=0} \ldots T_{n-1}$) or when all the characters for both P and T are the same except for their last characters ($(P_{i=0} \ldots P_{m-2} = T_{j=0} \ldots T_{n-2}) \wedge (P_{m-1} = T_{n-1})$). This would yield an $O(m \cdot (n - m + 1))$ number of comparisons.

2.3 Knuth-Morris-Pratt Algorithm

[10] [11] 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 $P_{0\ldots i-1}$ do not need to be checked with $T_{j-i\ldots j-1}$ 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: 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
```

2.4 Rabin-Karp Algorithm

3 Experimental Methodology

3.1 Research Question Decomposition

3.2 Measuring Performance

3.3 Application to Domains

4 Experimental Results Analysis

5 Conclusion

6 References

Question 1 *(with optional title)*

In congue risus leo, in gravida enim viverra id. Donec eros mauris, bibendum vel dui at, tempor commodo augue. In vel lobortis lacus. Nam ornare ullamcorper mauris vel molestie. Maecenas vehicula ornare turpis, vitae fringilla orci consectetur vel. Nam pulvinar justo nec neque egestas tristique. Donec ac dolor at libero congue varius sed vitae lectus. Donec et tristique nulla, sit amet scelerisque orci. Maecenas a vestibulum lectus, vitae gravida nulla. Proin eget volutpat orci. Morbi eu aliquet turpis. Vivamus molestie urna quis tempor tristique. Proin hendrerit sem nec tempor sollicitudin.