String Matching Algorithms

Naive and KMP algorithms implementation and analysis

Elia Innocenti

November 2023

Index

1 Introduction

In this report we would like to compare two algorithms for the **string-matching problem**, namely the **naive string-matching algorithm** and the **Knuth-Morris-Pratt algorithm**.

1.1 Hardware and software specifications

Below are the specifications of the machine used to conduct the tests:

- **CPU:** Apple M1 SoC (System on Chip) with an 8-core CPU (4 high-performance and 4 high-efficiency cores), 7-core GPU, and Neural Engine
- **RAM:** 8GB unified memory
- **OS:** Sonoma 14.0
- **Python version:** 3.11

2 The string matching problem

2.1 Definition

The **string-matching problem** is the problem of finding all occurrences of a pattern P in a text T.

We formalize the string-matching problem as follows.
We assume that the test is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of T and P are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, c, \ldots, z\}$, the set of lowercase English letters. The character arrays T and P are often called strings of character.

In fact, the string-matching problem can also be seen as the problem of finding all valid shifts with which a given pattern P occurs in a text T.

2.2 Example

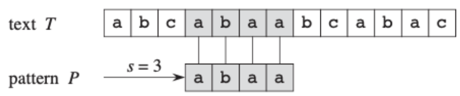

Fig. 1: An example of the string-matching problem. Given the text T = *abcabaabcabac* and the pattern P = *abaa*, we want to find all the occurrences of P in T. The pattern occurs only once in T, starting at shift $s = 3$.

$$\begin{cases} 0 \leq s \leq n - m \\ \text{T[s+1..s+m]} = \text{P[1..m]} \\ \text{T[s+j]} = \text{P[j] for } (1 \leq j \leq m) \end{cases}$$

2.3 Practical application

String matching algorithms have many practical applications in different fields:

- Text search: Find the occurrences of words or phrases within documents or long texts.
- Natural language processing: Analysing text to recognise language patterns, lexical analysis and parsing of texts.
- Computational biology: Analysing DNA, RNA and protein sequences to identify biological patterns and correlations.
- Log and structured text analysis: Recognise patterns in large datasets, such as system logs, sensor records, etc.
- Data filtering and analysis: They allow information to be extracted from large volumes of structured and unstructured data.
- Search engines: Working behind the scenes to identify the most relevant matches between user search queries and content.
- Data compression: Used in compression algorithms to find and replace repeated patterns.
- Computer security: Identify signatures and patterns of attacks by detecting byte sequences or malicious behaviour in data.

3 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ values of s.

3.1 Pseudocode

Data: Text T of size n and pattern P of size m

```
NAIVE-STRING-MATCHER(T,P)
n ← T.length
m ← P.length
for s ← 0 to n − m do
    if P[1..m] = T[s + 1..s + m] then
        print "Pattern occurs with shift" s;
    end
end
```

4 String matching with finite automata

5 Knuth-Morris-Pratt algorithm

5.1 Pseudocode

Data: Text T of size n and pattern P of size m

```
KMP-MATCHER(T,P)
n ← T.length
m ← P.length
π ← COMPUTE-PREFIX-FUNCTION(P)
q ←0
for i ← 1 to n do
    while q > 0 and P[q + 1] ≠ T[i] do
        q ← π[q];
    end
    if P[q + 1] = T[i] then
        q ← q + 1;
    end
    if q = m then
        print "Pattern occurs with shift" i − m;
        q ← π[q];
    end
end
```

```
COMPUTE-PREFIX-FUNCTION(P)
m ← P.length
let π[1..m] be a new array;
π[1] ← 0;
k ← 0;
for q ← 2 to m do
    while k > 0 and P[k + 1] ≠ P[q] do
        k ← π[k];
    end
    if P[k + 1] = P[q] then
        k ← k + 1;
    end
    π[q] ← k;
end
return π;
```

6 Tests and results

6.1 Expected performances

7 Conclusions