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

1.2 References

Chapter 32 of the book *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (Third Edition).

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.

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.
We say that the pattern P occurs with shift s in the text T if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$.

2.2 Example

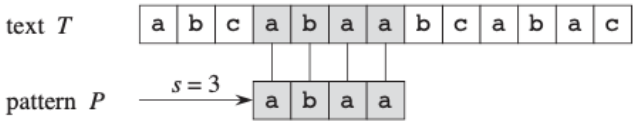

Figure 1: An example of the string-matching problem. Given the text $T = abcabaabcabac$ and the pattern $P = abaa$, we want to find all the occurrences of P in T. The pattern occurs only once in T, starting at shift $s = 3$.

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

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 Description

The naive string-matching algorithm proceeds sliding a "template" containing the pattern over the text, noting for which shifts all of the characters in the template match the corresponding characters in the text.
The for loop consider each possible shift explicitly, and the if-condition implicitly loops to check corresponding characters position until all position match successfully or until a mismatch is found.

3.2 Pseudocode

Data: Text T of size n and pattern P of size m
Result: All valid shifts with which P occurs in T

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

3.3 Time complexity

To analyse the time complexity of the algorithm, we can study separately the case in which the pattern occurs only once within the text and the case in which it occurs several times.
In the first case we can identify three situations: the case in which the pattern is at the beginning of the text (best-case), the case in which it is at the end of the text (worst-case) and the case in which it is anywhere else in the text (average-case).

- **Best-case:** the pattern is found immediately at the beginning of the text. In this case the complexity is $\Theta(m)$, where m is the length of the pattern.

- **Worst-case:** the pattern is found at the end of the text. In this case the complexity is $\Theta((n-m+1)m)$, where n is the length of the text and m is the length of the pattern.
- **Average-case:** the pattern is found anywhere else in the text. In this case the complexity is $O((n-m+1)m)$, where n is the length of the text and m is the length of the pattern.

In the second case, on the other hand, we find only a single case, since the algorithm is forced to view the entire text anyway (this is similar to the worst-case treated in the previous situation). In fact, in this case the complexity is $O((n-m+1)m)$.

4 String matching with finite automata

Some algorithms, in order to reduce the time complexity of the matching process, use a finite automaton to implement a **preprocessing function** that examines the pattern P.
There's in fact a pattern matching technique, the **string matching with finite automata**, that employs finite automata to efficiently search for occurrences of a given pattern within a larger text. These string-matching automata examine each text character exactly once, taking constant time per text character. The matching time used, after preprocessing the pattern to build the automaton, is therefore $\Theta(n)$, where n is the length of the text. Although it is independent of the size of the text, the time to build the automaton can be large if Σ is large. In fact, the time to process the pattern using the function δ is $O(m|\Sigma|)$, where m is the length of the pattern and $|\Sigma|$ is the size of the alphabet.

4.1 Finite automaton

Concept:

- A finite automaton is a simple machine for processing information, used to recognize patterns within input sequences.
- It's a computational model consisting of a finite set of states and transitions between those states based on input symbols.

- The transition function δ defines the ways in which one goes from one state to another.
- In this context, it's used to recognize whether the pattern occurs in the text.

Process:

- Preprocessing: construct a finite automaton that represents the pattern to be matched.
- Matching: use the finite automaton to scan through the text and identify potential matches.

Advantages:

- Efficient for multiple searches of the same pattern within different texts.
- Requires preprocessing only for the pattern.

Implementation:

- Construction of Finite Automaton: this involves building a transition function δ that defines the behavior of the automaton when reading characters of the pattern.
- Matching: traverse the text while updating the automaton's state based on the characters encountered. If the automaton reaches an accepting state, a match is found.

5 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm is a string-matching algorithm that uses finite automata to perform string matching. In fact, the Knuth-Morris-Pratt algorithm improves the running time of the naive algorithm by using information about the pattern itself to avoid some comparisons, using a preprocessing function π.
The key idea is that when a mismatch occurs, the pattern itself contains enough information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

5.1 Description

It preprocesses the pattern to create a "failure function" that determines potential shifts in case of mismatches during pattern matching. This information allows the algorithm to skip unnecessary comparisons, enabling faster pattern identification within the text. The KMP algorithm scans the text, leveraging the failure function to avoid redundant checks, and reports all positions where the pattern is found within the text.

5.2 Pseudocode

Data: Text T of size n and pattern P of size m
Result: All valid shifts with which P occurs in T

```
1  KMP-MATCHER(T,P)
2  n ← T.length
3  m ← P.length
4  π ← COMPUTE-PREFIX-FUNCTION(P)
5  q ←0
6  for i ← 1 to n do
7  |  while q > 0 and P[q + 1] ≠ T[i] do
8  |  |  q ← π[q];
9  |  end
10 |  if P[q + 1] = T[i] then
11 |  |  q ← q + 1;
12 |  end
13 |  if q = m then
14 |  |  print "Pattern occurs with shift" i − m;
15 |  |  q ← π[q];
16 |  end
17 end
```

Data: Pattern P of size m
Result: Prefix function π

```
1  COMPUTE-PREFIX-FUNCTION(P)
2  m ← P.length
3  let π[1..m] be a new array;
4  π[1] ← 0;
5  k ← 0;
6  for q ← 2 to m do
7  |  while k > 0 and P[k + 1] ≠ P[q] do
8  |  |  k ← π[k];
9  |  end
10 |  if P[k + 1] = P[q] then
11 |  |  k ← k + 1;
12 |  end
13 |  π[q] ← k;
14 end
15 return π;
```

5.3 Time complexity

Let us analyse the time complexity of the KMP algorithm by examining the preprocessing and matching phases separately.

We know that KMP uses a preprocessing function π, based on the concept of finite automata, but implements it in such a way as to be able to compute the transition function δ efficiently ("on the fly" as needed). In fact, since $\pi[1..m]$ is the array in which the values of the function of the same name are saved, and since $\pi[1..m]$ has only m entries, this allows the KMP to save a factor of $|\Sigma|$ in the preprocessing time. The time complexity of the preprocessing function π of the KMP is therefore $\Theta(m)$.

The time for the matching phase is therefore $\Theta(n)$, as already mentioned in the Section 4 (String matching with finite automata). This is because the preprocessing function allows the algorithm to scroll through the text by examining the characters only once, and consequently the complexity is reduced only to the text size, which is n.

6 Differences between naive string-matching and KMP algorithms

6.1 Preprocessing

Algorithm	**Preprocessing time**
Naive	0
KMP	$\Theta(m)$

Table 1: Preprocessing time complexity.

6.2 Matching time complexity

Pattern with only one occurrence in the text:

	Matching time		
Algorithm	**Worst-case**	**Best-case**	**Average-case**
Naive	$\Theta((n-m+1)m)$	$O((n-m+1)m)$	$O((n-m+1)m)$
KMP	$\Theta(n)$	$O(n)$	$O(n)$

Table 2: Matching time complexity (case 1).

Pattern with multiple occurrences in the text:

Algorithm	**Matching time**
Naive	$\Theta((n-m+1)m)$
KMP	$\Theta(n)$

Table 3: Matching time complexity (case 2).

7 Tests and results

Tests and results
- TO BE WRITTEN -

7.1 Expected performances

Expected performances
- TO BE WRITTEN -

7.2 Results

Results
- TO BE WRITTEN -

8 Conclusions

Conclusions
- TO BE WRITTEN -