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.

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

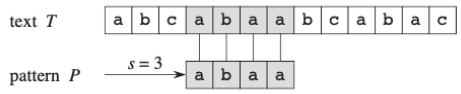

Figure 1: An example of the string-matching problem. Given the text $T = abcabaabcabac$ and the pattern $P = abaa$, we want to find all the occurrences of P in T. The pattern occurs only once in T, starting at shift $s = 3$.

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

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 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

The worst-case running time of the naive string-matching algorithm is $\Theta((n-m+1)m)$, which is $\Theta(nm)$. The worst case occurs when the first $m-1$ characters of the pattern match the first $m-1$ characters of the text, but the last characters do not match. In this case, the algorithm performs $n-m+1$ comparisons, each requiring m character comparisons.

The best-case running time of the naive string-matching algorithm is $\Theta(n)$. The best case occurs when the first character of the pattern does not occur in the text. In this case, the algorithm performs $n-m+1$ comparisons, each requiring 1 character comparison.

The average-case running time of the naive string-matching algorithm is $\Theta(n)$. The average case occurs when the pattern does not occur in the text. In this case, the algorithm performs $n - m + 1$ comparisons, each requiring m character comparisons.

4 String matching with finite automata

Some algorithms implement a finite automata - a simple machine for processing information - that scans the text string T for all occurrences of the pattern P. 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. The time to build the automaton can be large if Σ is large.

4.1 Finite automata

Finite automata

5 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm improves the running time of the naive algorithm by using information about the pattern itself to avoid some comparisons. 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

Description

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

The worst-case running time of the Knuth-Morris-Pratt algorithm is $\Theta(n)$. The worst case occurs when the first character of the pattern does not occur in the text. In this case, the algorithm performs n comparisons, each requiring 1 character comparison.

The best-case running time of the Knuth-Morris-Pratt algorithm is $\Theta(n)$. The best case occurs when the pattern does not occur in the text. In this case, the algorithm performs n comparisons, each requiring 1 character comparison.

The average-case running time of the Knuth-Morris-Pratt algorithm is $\Theta(n)$. The average case occurs when the pattern does not occur in the text. In this case, the algorithm performs n comparisons, each requiring 1 character comparison.

6 Differences between naive string-matching and KMP algorithms

6.1 Preprocessing

Preprocessing

6.2 Time complexity

		Matching time		
Algorithm	**Preprocessing time**	**Worst-case**	**Best-case**	**Average-case**
Naive	0	$\Theta((n-m+1)m)$	$O((n-m+1)m)$	$O((n-m+1)m)$
KMP	$\Theta(m)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Table 1: Time complexity of the naive and KMP algorithms.

7 Tests and results

7.1 Expected performances

8 Conclusions