SORBONNE UNIVERSITÈ

Computer Science

Development of Algorithms for Reticular Application

Preoject work on:

Automaton. Cloning egrep command supporting simplified ERE

Student:

Caterina Leonelli
Maria Cristina Centorame

Academic year 2023/2024

Chapter 1

Introduction

Regular Expressions (RegEx) and Extended Regular Expression (ERE) specifications play a fundamental role in computer science as they represent a formalized language that can be recognized by a deterministic finite automaton (DFA), based on principles from formal language theory. In this context, we focus on RegEx defined by ERE specifications, which are detailed in the following link: `http://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html`.

When dealing with a text file consisting of l lines, the task of finding a RegEx pattern within the file is typically approached by breaking it down into l individual searches, each targeting a different line. The overarching objective of this project is to devise a solution for this challenge. To accomplish this, we explored and implemented three distinct strategies: the Automata strategy based on Aho Ullman's principles, the Knuth-Morris-Pratt (KMP) algorithm, and the Radix Tree Search algorithm.

After implementing these three strategies, we conducted extensive testing using various input texts and different RegEx patterns. Our goal was to gather empirical data, which we subsequently analyzed. This analysis served two key purposes: firstly, it confirmed the practical applicability of our theoretical knowledge, and secondly, it shed light on the strengths and weaknesses inherent in each of the three approaches.

Finally, we have crafted a user-friendly interface that emulates the functionality of the UNIX egrep command. This interface empowers users to define a desired regular expression (regex) pattern, which they intend to locate within a provided input text. Upon execution, the program highlights the matched patterns within the text, presenting them in a distinctive red color for enhanced visibility and ease of identification.

Chapter 2

Strategies for pattern recognition by a RegEx

2.1 Automata Algorithm

2.1.1 Basic concepts of Automata Theory

Automata serve as abstract computational devices designed to tackle problem-solving tasks. In the realm of automata theory, a "problem" entails the determination of whether a given string belongs to a specific language, as explained in detail by Hopcroft, Motwani, and Ullman [1]. Consequently, automata prove to be highly suited for our project's objectives. Over the years, a variety of abstract machines have been proposed, ranging from the simplest combinatorial machines to the widely recognized Universal Turing Machine of today. In our project, we primarily refer to Finite Automata, characterized as fundamental combinatorial machines possessing a notion of "state." The state function within these automata calculates the subsequent state the automaton will transition to. However, it's important to note that Finite Automata suffer from finite memory constraints, imposing computational limitations in the process.

What makes this field intriguing is the direct equivalence between regular expressions and automata: both can represent the same language, yet they offer distinct perspectives. Regular expressions provide a descriptive view of the language, whereas automata offer an operational approach to dealing with it. In essence, regular expressions reveal "what" language can be obtained, whereas automata elucidate "how" to obtain a specific language.

A pivotal distinction arises between Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA). In DFA, the automaton cannot exist in multiple states simultaneously, whereas in NFA, it may concurrently occupy several states, rendering NFAs more efficient. Nondeterminism empowers us to "program" solutions to problems using a higher-level language. Subsequently, the NFA is "compiled" into a deterministic automaton, which can be executed on a conventional computer, as discussed in Hopcroft, Motwani, and Ullman [1].

2.1.2 Deterministic Finite Automata

A formal definition of a DFA is presented as a quintuple:

$$< I, S, sfn, SS, FS >$$

where:

- I represents the finite set of input symbols.
- S denotes the finite set of states, comprising a start state (SS), a set of final or accepting states (FS), and several intermediate states.
- $sfn : I \text{x} S \longrightarrow S$ signifies the state or transition function.

From this definition, we can discern that a finite automaton is an abstract machine equipped with a set of states, with its "control" transitioning from state to state in response to external inputs.

An alternate way to describe a DFA operationally is through a transition table, where rows correspond to states, columns correspond to inputs, and entries indicate whether a state is accepting and whether it is the start state.

2.1.3 Nondeterministic Finite Automata

The NDFA shares a very similar formal definition with the DFA, except the state function can return a set of states instead of a single state.

The reason for continuing to use NFAs is that they are often easier to construct than DFAs. Consequently, it is typically preferred to initially construct an NFA and then convert it into a DFA, rather than directly creating the DFA itself.

2.1.4 From the RegEx to the Automata

For our project, we have implemented the following algorithm to create an automata:

Algorithm 1 Automata Algorithm

1: **procedure** AUTOMATA(Regex $regex$)
2: $regexTree \leftarrow$ CONVERTTOREGEXTREE($regex$)
3: $nfa \leftarrow$ CONVERTTONFA($regexTree$)
4: $dfa \leftarrow$ CONVERTTODFA(nfa)
5: $minimizedDFA \leftarrow$ MINIMIZEDFA(dfa)
6: **return** $minimizedDFA$
7: **end procedure**

In summary, if the RegEx is not reduced to a series of concatenations, we first transform it into a syntax tree (1) then into a non-deterministic finite automaton (NDFA) with ϵ-transitions according to Aho-Ullman algorithm (2) then into a deterministic finite automaton (DFA) with the subset method (3) then into an equivalent automaton with a minimum number of states (4) and finally the automaton is used to test whether a suffix of a line of the input text is recognizable by this automaton (5).

The complexity of this algorithm is $O(n^2)$ with n as the text length.

2.2 Knuth-Morris-Pratt(KMP) algorithm

2.2.1 KMP approach

To perform pattern matching, one of the first procedure that comes to mind is usually to apply the so-called "naive algorithm"which has a worst case complexity of $O(m(n-m+1))$, indeed it's not a good solution for cases in which there are many matching characters followed by a mismatching one. Therefore, an improved string matching algorithm was developed by Donald Knuth, Vaughan Pratt, and James H. Morris to efficiently search for occurrences of a pattern (substring) within a text (string) by exploiting the basic idea of avoiding superfluous comparisons of characters. The KMP algorithm includes a pre-processing phase which makes use of a partial match table storing information about the pattern structure and then proceeds searching in the text with the advantage of knowing the count of characters to be skipped each time a mismatch is encountered.

2.2.2 KMP implementation

The KMP implementation consists of the following steps:

- ◇ Preprocessing phase:

A prefix-table is implemented as an array $lps[]$ of the same length M of the pattern $pat[]$ in which for each subpattern $pat[0...i]$ where i goes from 0 to $M-1$, $lps[i]$ stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern $pat[0..i]$. For programming convenience the $lps[]$ array is shifted one bit to the right by adding the "-1"in the first position(see Algorithm 2).

In addition, in Algorithm 3 the CarryOver is computed: it's an optimized version of the LPS array that works as a sliding window and where the "-1"stands for an already processed character.

e.g an example of lps and CarryOver computation for "mamamia" is
$lps = [-1, 0, 0, 1, 2, 3, 0, 0]$, and $CarryOver = [-1, 0, -1, 0, 0, 3, 0, 0]$.
This program takes $O(m)$ units of time, where m is the pattern length.

◇ Searching phase:

The KMP-algorithm goes through the text from left to right by using the index i and scans the pattern with the index j. As soon as it encounters a mismatch, it shifts to the right and fetches a new pattern index from the prefix table $j \leftarrow \text{table}[j-1]$ as long as j is not 0 so that it avoids checking characters that it already knows will match and at the same time it skips all the positions where no matches can be made(see Algorithm 4).

This program takes $0(n)$ units of time, where n is the text length. For a deeper understanding, see chapter 32.4 from [2]

Finally, the whole algorithm has a complexity of $O(m+n)$.

Algorithm 2 LPS implementation

```
1:  procedure computeLPSArray(pattern, M)
2:      len ← 0                                  ▷ length of the previous longest prefix suffix
3:      lps ← an array of length M               ▷ initialize lps[] with zeros
4:      i ← 1
5:      while i < M do                           ▷ the loop calculates lps[i] for i = 1 to M-1
6:          if pattern[i] = pattern[len] then
7:              len ← len + 1
8:              lps[i] ← len
9:              i ← i + 1
10:         else
11:             if len ≠ 0 then
12:                 len ← lps[len − 1]
13:             else
14:                 lps[i] ← 0
15:                 i ← i + 1
16:             end if
17:         end if
18:     end while
19:     lps.insert(0, −1)                        ▷ shift the entire array by one position forward
20:     return lps
21: end procedure
```

Algorithm 3 CarryOver implementation

```
1: procedure computeCarryOverArray(pattern, M, lps)        ▷ This is an optimization of the computeLPSArray function
2:     for i ← 1 to M − 1 do
3:         if pattern[i] = pattern[lps[i]] then
4:             if lps[lps[i]] = −1 then
5:                 lps[i] ← −1
6:             else
7:                 lps[i] ← lps[lps[i]]
8:             end if
9:         else
10:            lps[i] ← lps[i]
11:        end if
12:    end for
13:    return lps
14: end procedure
```

Algorithm 4 KMP Algorithm

```
1: procedure KMPSearch(pattern, txt)
2:     M ← length of pattern
3:     N ← length of txt
4:     matching_indexes ← empty list
5:     j ← 0                                                ▷ index for pattern[]
6:     lps ← computeLPSArray(pattern, M)
7:     lps ← computeCarryOverArray(pattern, M, lps)
8:     i ← 0                                                ▷ index for txt[]
9:     while i < N do
10:        if pattern[j] = txt[i] then
11:            i ← i + 1
12:            j ← j + 1
13:        end if
14:        if j = M then
15:                                                         ▷ Found pattern at index
16:            matching_indexes.append(i − j)
17:            j ← lps[j − 1]
18:        end if
19:        if i < N and pattern[j] ≠ txt[i] then
20:            if j ≠ 0 then
21:                j ← lps[j − 1]
22:            else
23:                i ← i + 1
24:            end if
25:        end if
26:    end while
27:    return matching_indexes
28: end procedure
```

2.3 Radix-tree algorithm

In the specific case in which the RegEx is not only reduced to a series of concatenations, but also composed exclusively of alphabetic characters, another strategy can be taken into account. The procedure (see Algorithm 5) is as follows:

- we are given a text to be processed by storing all the useful words in the text into a table of indices contained in a cache-file.
- the cache-file is then stored in a radix tree search structure
- finally, the algorithm proceeds searching for the Regex by browsing the radix tree.

Algorithm 5 RadixTree Search Algorithm

```
procedure RadixTreeSearch(String fileName, String pattern )
    root ← RadixNode()                                          ▷ Build the Radix tree
    cache_file_lines ← open(filename, "r").read().split("\n")
    for line, in cache_file_lines: do
        words ← line.split()
        for index, token in enumerate(words) do▷ Insert all the useful words into the Radix tree
            root.insert_token(token, index)
        end for
        matching_indices ← []                                   ▷ Initialize empty list for matching indices
        for component in components do                          ▷ Search for each component of the pattern
            founded_indices ← root.search_tokens(pattern)
            matching_indices.extend(founded_indices)            ▷ Extend the list of matching indices
with the currently found indices
        end for
        colored_words ← color_matched_words(cache_file, matching_indices)
        Print "".join(colored_words)
```

Radix search encompasses a range of data structures designed to facilitate the search for strings viewed as sequences of digits within a given base or radix. The fundamental concept revolves around grouping strings with common prefixes under a single node. We construct a tree where each internal node can have as many children as there are unique characters in the alphabet.

One valuable characteristic of radix trees lies in their key ordering which is not arbitrary, as is the case with hash tables; instead, keys are ordered based on bitwise lexicographical sorting.

In addition, Radix trees exhibit several noteworthy characteristics that set them apart from comparison-based search trees:

the height and complexity of radix trees are primarily influenced by key length rather than the number of elements within the tree; Radix trees don't necessitate rebalancing operations, and all insertion orders produce the same tree structure; the path leading to a leaf node encodes the key associated with that leaf, enabling implicit key storage and the ability to reconstruct keys from paths. The complexity of radix trees is $O(m)$, where m is the pattern length. These observations imply that radix trees, especially those with a broad range, can be more efficient than traditional search trees. For a deeper insight into the topic see [3]

Chapter 3

Performance analysis

As stated earlier in the introduction, the algorithms described in the preceding sections can prove valuable in various applications, depending on their time complexity, the need of a preprocessing phase and pattern structure.
In this section, we will demonstrate the time complexity results obtained by alternately varying the lengths of both the pattern and the input text used in our algorithm. We conducted performance tests using the following texts from the provided dataset `http://www.gutenberg.org/`:

→ 'Book About Babylone.txt'

→ 'Pony Tracks.txt'

These tests were carried out to assess the algorithm's performance under various conditions.
The longest text used for the tests is 100000 characters long and the longest pattern is 5000 long.

Specifically, the Automata approach demonstrates significant utility when searching for multiple patterns within the same text. While the construction of the automaton may initially require some time and complexity, it offers the advantage of reusability for multiple searches. However, it may not be the most efficient option for single pattern searches.
In the context of the Automata approach, our focus is narrowed down to specific elements, which include parentheses, alternatives, concatenations, the asterisk (for repetition), the period (representing the universal character), and any ASCII letter.
The Automata approach exhibits a time complexity of $O(n^2)$(n is the text length), and even in cases involving patterns consisting solely of concatenation, it escalates to a complexity of $O(m(n - m))$(n is the text length, m is the pattern length). This suggests the suitability of an alternative strategy, namely the KMP algorithm due to its straightforward implementation and reliably linear time complexity.

Indeed, as shown in Figure 3.1, the KMP algorithm exhibits a linear behavior concerning the length of the input text. If we also account for the pattern preprocessing phase, it has a complexity of $O(n + m)$, which remains linear concerning the pattern length as well. However, when considering only the matching phase, KMP operates with constant time regardless of the pattern's length. And this constitutes a significant advantage over the automata approach, as visible in Figure 3.4.
Here it appears that the KMP line in the left graph remains constant due to the significant difference in slope compared to the automata line, however when we refer to Figure 3.3, the true slope of the KMP becomes apparent in comparison to the Radix tree line.

On the other hand, the Radix Search Tree proves its efficiency when searching for patterns with shared prefixes or implementing autocomplete functionality. It excels in scenarios involving substantial datasets containing words or strings that exhibit common prefixes. When comparing all the three algorithms(see Figure 3.4), an interesting observation to note is that both the KMP and Radix Tree algorithms perform well with longer patterns precisely because the first maintains a constant behavior and the second has a low-slope line with respect to the pattern length, whereas the automata approach is notably affected by the length of the pattern thus becoming slower on larger ones.

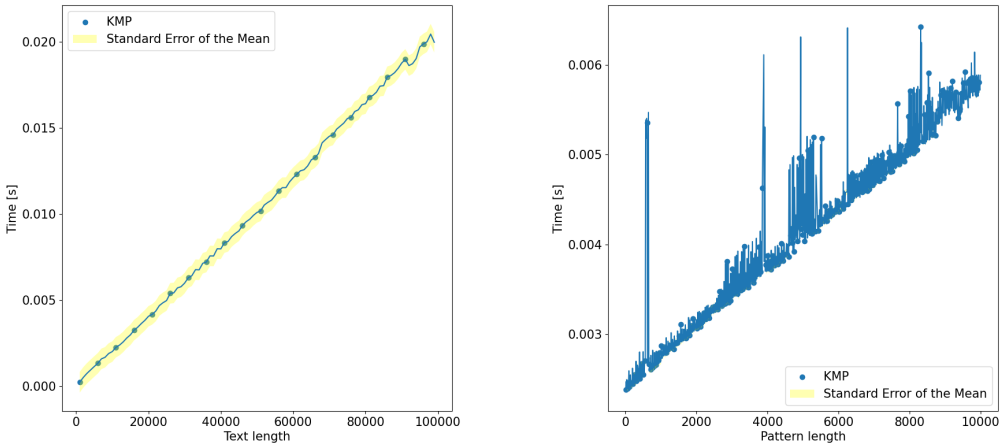

Figure 3.1: KMP execution time based on text length(left) and pattern length(right).
Note that the right-hand graph takes into account also the time complexity due to the preprocessing phase.

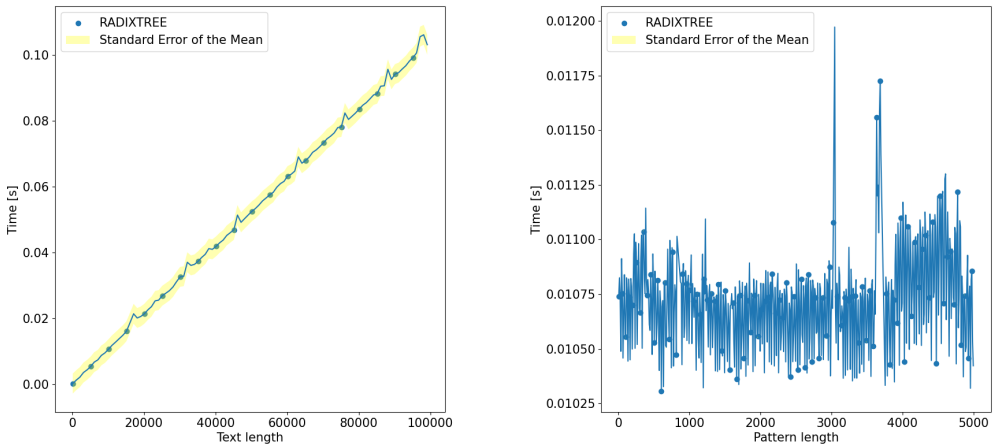

Figure 3.2: Radix tree execution time based on text length(left) and pattern length(right). Note that in the right-hand graph, the curve appears to be constant, but if the algorithm is stressed sufficiently, a linear trend can be identified.

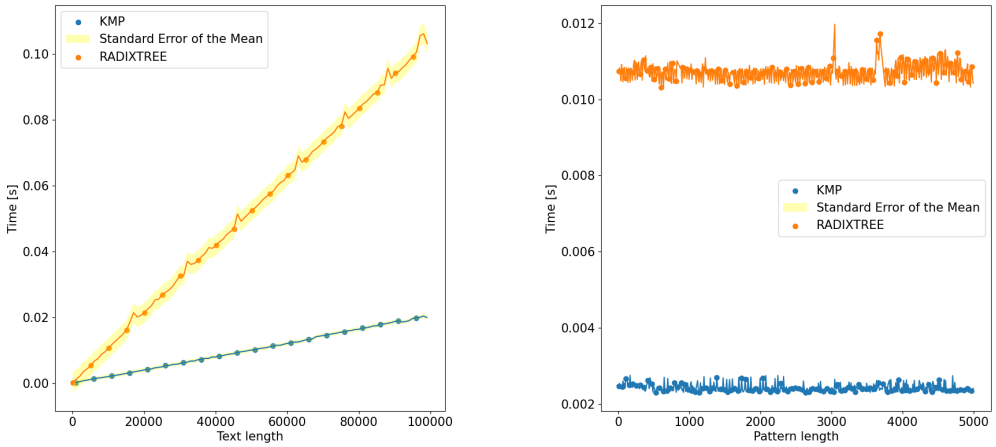

Figure 3.3: KMP and Radix tree execution time based on text length(left) and pattern length(right).

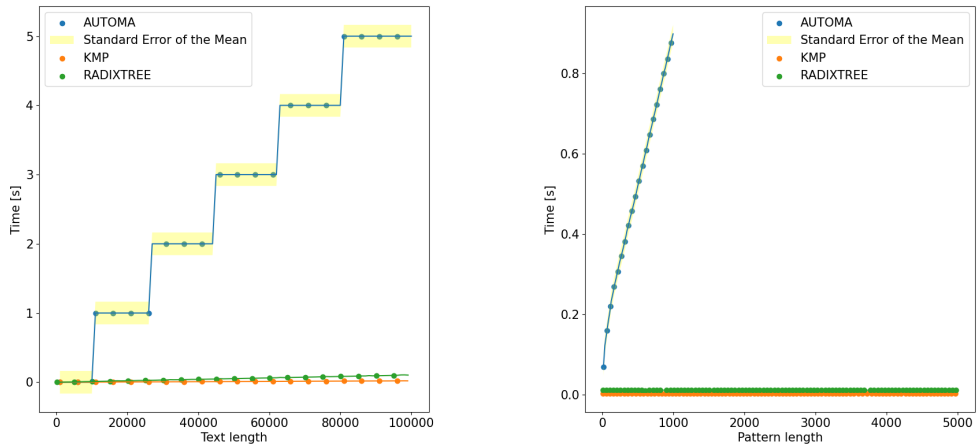

Figure 3.4: Automata, KMP and Radix tree execution time based on text length(left) and pattern length(right).

Conclusions

In this study, we explored and implemented three distinct algorithms for pattern recognition within text files: the Automata approach, the Knuth-Morris-Pratt (KMP) algorithm, and the Radix Tree Search algorithm. Our analysis and experimentation aimed to shed light on the practical applicability and strengths and weaknesses of each approach. Our conclusions have been that the choice of algorithm for pattern recognition depends on the specific application requirements, time complexity considerations, and the characteristics of the input data. Each algorithm has its strengths and limitations, and selecting the most suitable one is essential for achieving efficient and accurate pattern recognition.

We have also developed a user-friendly interface that emulates the functionality of the UNIX egrep command, allowing users to define regular expression patterns for locating within input text files. This interface highlights matched patterns within the text, enhancing visibility and ease of identification.

Overall, this study provides valuable insights into the performance and practicality of various pattern recognition algorithms, enabling informed decision-making when addressing pattern recognition challenges in real-world applications.

Bibliography

[1] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3nd ed.* Pearson-Addison-Wesley, 2018.

[2] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms, Fourth Edition.*

[3] Thomas Neumann Viktor Leis, Alfons Kemper. The adaptive radix tree: Artful indexing for main-memory databases.