WORD SEARCH USING PARALLEL PROGRAMMING IN PYTHON

A PROJECT REPORT

Submitted by

Mahesh Sharma
(17BCE0619)
Ishan Kaushik
(17BCE2025)
Simriti Koul
(17BCE2211)
Apoorva Bansal
(17BCE2226)

CSE4001 Parallel and Distributed Computing
Slot – L29+L30

Computer Science and Engineering
November 2019

CONTENTS

1. ABSTRACT

An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem. There are various types of algorithms present and are thus utilized with various different functionality. Various types of algorithms are:

- Backtracking algorithms.
- Dynamic programming algorithms
- Branch and bound algorithms.
- Brute force algorithms.
- Randomized algorithms.
- String Matching Algorithms.

These are just the basic types of algorithm and they are further classified upon their functionality. Word Search algorithms are the algorithms which are used in everyday life, from our phone to our personal computers everything has use of such an algorithm. Word Searching, which comes under the class of string-matching algorithms is a very important subject in the field of text processing. These algorithms lie at the fundamentals of many practical software relying on efficient searching of occurrences of a particular word.

Here we are going to discuss about 'Knuth–Morris–Pratt-algorithm' and how it helps solving the problem of word or string searching in an algorithm. In both serial and parallel computing, this algorithm can be implemented and the differences can be noticed on both ends. Thus, the final conclusion of this project will be implementing the KMP algorithm in both serial and parallel way and thus concluding which is faster.

2. INTRODUCTION

2.1 Algorithms

While our interaction with computers and other technology has evolved in the recent years, the main form of communication with a computer still at heart remains by text. Data is still stored in the text format and is being processed by software in this format. Traditionally, word searching is done using serial computing and the performance using a single core is quite limited, hence we would like to implement one of the popular algorithms used of string matching and parallelize it to improve the performance. In computing, approximate string matching (often conversationally stated as fuzzy string searching) is that the technique of finding strings that match a pattern more or less (rather than exactly). the matter of approximate string matching is often divided into 2 subproblems: finding approximate substring matches within a given string and finding wordbook strings that match the pattern more or less.
Traditionally, word searching is done using serial computing and the performance using a single core is quite limited, hence we would like to implement one of the popular algorithms used of string matching and parallelize it to improve the performance.

By the number of patterns used, the algorithms are further divided.

2.1.1 Single Pattern Algorithm

Let m be the length of the pattern, n be the length of the searchable text and $k = |\Sigma|$ be the size of the alphabet.

Algorithm	Preprocessing time	Matching time[1]	Space
Naïve string-search algorithm	none	Θ(nm)	none
Rabin–Karp algorithm	Θ(m)	average Θ(n + m), worst Θ((n–m)m)	O(1)
Knuth–Morris–Pratt algorithm	Θ(m)	Θ(n)	Θ(m)
Boyer–Moore string-search algorithm	Θ(m + k)	best Ω(n/m), worst O(mn)	Θ(k)
Bitap algorithm (*shift-or*, *shift-and*, *Baeza–Yates–Gonnet*)	Θ(m + k)	O(mn)	
Two-way string-matching algorithm	Θ(m)	O(n+m)	O(1)
BNDM (Backward Non-Deterministic Dawg Matching)	O(m)	O(n)	
BOM (Backward Oracle Matching)	O(m)	O(n)	

2.1.2 Algorithms using Finite set of patterns

a. Aho–Corasick string matching algorithm (extension of Knuth-Morris Pratt)

b. Commentz-Walter algorithm (extension of Boyer-Moore)

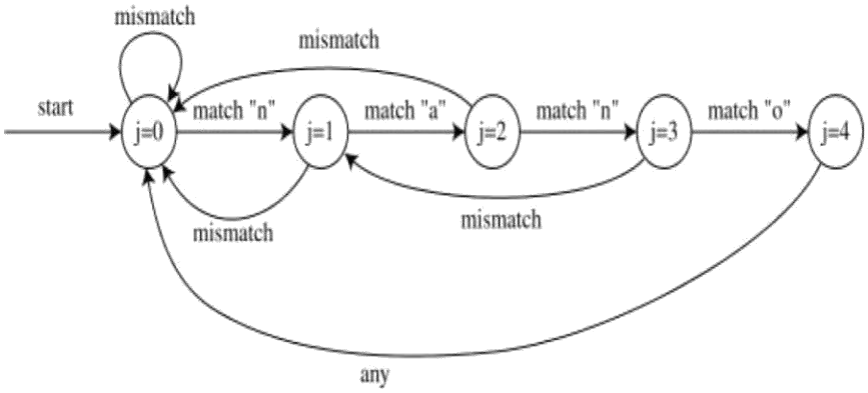
c. Rabin–Karp string search algorithm

2.1.3 Algorithms using infinite set of patterns

Naturally, the patterns cannot be enumerated finitely in this case. They are represented usually by a regular grammar or regular expression. In this project we will use Knuth- Morris- Algorithm, and thus identify in which case the kmp algorithm works better, i.e. in serial computing or parallel computing.

2.2 KMP Algorithm

The KMP algorithm works on the principle that whenever a mismatch occurs, the word itself embodies sufficient information to tell where the next match could begin thus improving the worst-case complexity. The KMP matching algorithmic rule uses degenerating property (pattern having same sub-patterns showing quite once within the pattern) of the pattern and improves the worst-case complexness to O(n). The fundamental plan behind KMP's algorithmic rule is: whenever we tend to observe a couple (after some matches), we tend to already apprehend a number of the characters within the text of consequent window. We tend to profit of this data to avoid matching the characters that we all know can anyway match.



3. LITERATURE REVIEW

Here we shall discuss the most widely used algorithms for string matching

3.1 Naïve string-search algorithm

This is a very basic string-matching algorithm which maintains window that slides over the text one by one and checks for a match. If not found, the window slides one more time and checks again. This process is repeated for the rest of the document.

Performance: Θ(nm)

3.2 Rabin-Karp algorithm

This algorithm uses hashing to find a set of pattern string inputted as text. It also tends to ignore case and punctuation so it is not practical to search for a single string. A practical application of this algorithm is for plagiarism detection where a

large number of input strings in given and they can be searched for in the source material.

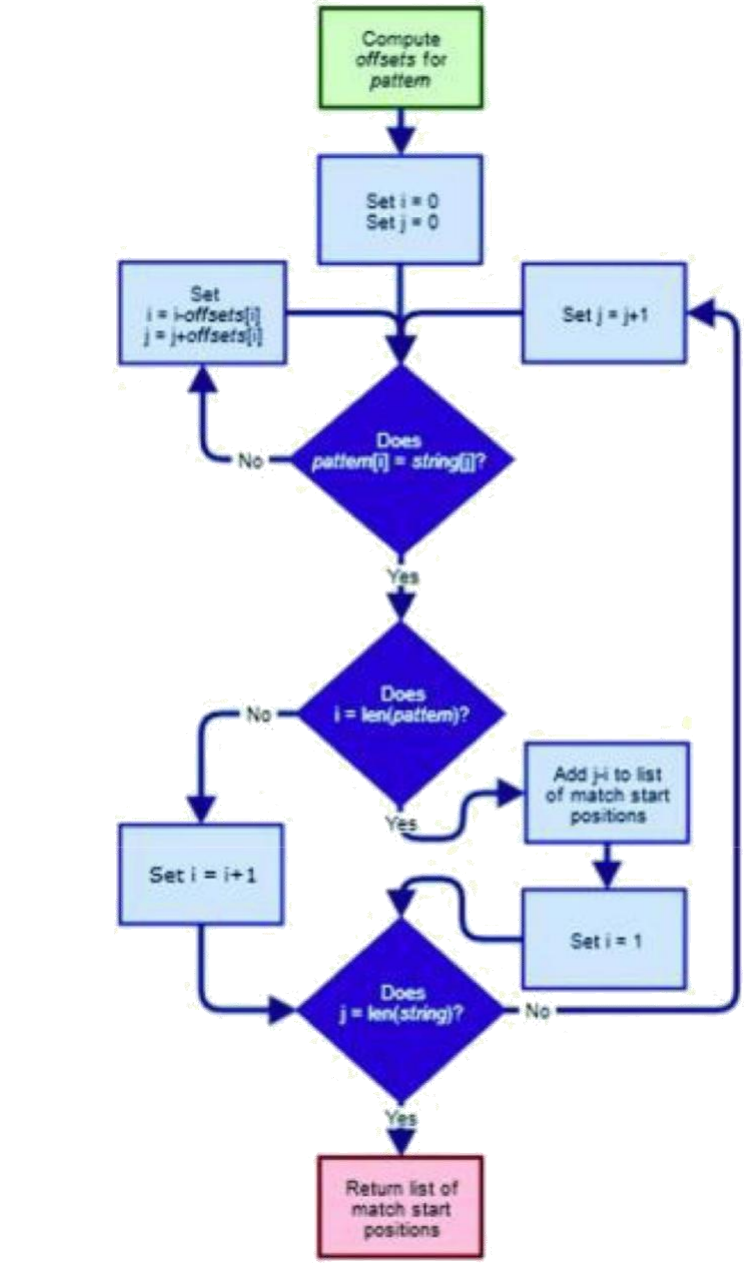
Performance (Worst Case): $\Theta((n-m)m)$

3.3 Knuth–Morris–Pratt algorithm

The KMP algorithm works on the principle that whenever a mismatch occurs, the word itself embodies sufficient information to tell where the next match could begin thus improving the worst-case complexity.

Performance: $\Theta(n)$

3.3.1 Flow Chart of KMP


Compute offsets for pattern
Set i = 0
Set j = 0
Set i = i-offsets[i]
j = j+offsets[i]
Set j = j+1
Does pattern[i] = string[j]?
No
Yes
Does i = len(pattern)?
No
Yes
Add j-i to list of match start positions
Set i = i+1
Set i = 1
Does j = len(string)?
No
Yes
Return list of match start positions

3.3.2 Advantages

a. The running time of KMP algorithm is optimal (O(m+n)), which is very fast compared to other string-matching algorithms.

b. The algorithm never needs to move backwards in the input text T. it makes the algorithm good for processing very large files.

3.3.3 Disadvantage

Doesn't work so well as the size of the alphabets increases, because more chances of mismatch occur.

3.4 Boyer-Moore String-search algorithm

The Boyer-Moore Algorithm is considered the most efficient string-searching algorithm and is used as a standard benchmark for comparing the efficiency of other algorithms. This algorithm preprocesses the string being searched for and uses information gathered during the preprocessing step to skip sections of text and has better performance as the input string gets longer.

4. INSTALLING PYTHON

Step 1: Install Homebrew (Part 1)

To get started, you first want to install Homebrew:

Open a browser and navigate to http://brew.sh/. After the page has finished loading, select the

Homebrew bootstrap code under "Install Homebrew". Then hit Cmd+C to copy it to the clipboard.

Make sure you've captured the text of the complete command because otherwise the installation will fail.

Now you need to open a Terminal window, paste the Homebrew bootstrap code, and then hit Enter. This +will begin the Homebrew installation.

If you're doing this on a fresh install of macOS, you may get a pop-up alert asking you to install Apple's "command line developer tools". You'll need those to continue with the installation, so please confirm the dialog box by clicking on "Install".

Step 2: Install Homebrew (Part 2)

You can continue installing Homebrew and then Python after the command line developer tools installation is complete:

Confirm the "The software was installed" dialog from the developer tools installer.

Back in the terminal, hit Enter to continue with the Homebrew installation. Homebrew asks you to enter your password so it can finalize the installation. Enter your user account password and hit Enter to continue. Depending on your internet connection, Homebrew will take a few minutes to download its required files. Once the installation is complete, you'll end up back at the command prompt in your terminal window.

Whew! Now that the Homebrew package manager is set up, let's continue on with installing Python 3 on your system.

Step 3: Install Python

Once Homebrew has finished installing, return to your terminal and run the following command:

This will download and install the latest version of Python. After the Homebrew brew install command finishes, Python 3 should be installed on your system. You can make sure everything went correctly by testing if Python can be accessed from the terminal:

1. Open the terminal by launching Terminal.app.
2. Type pip3 and hit Enter.

3 You should see the help text from Python's "Pip" package manager. If you get an error message running pip3, go through the Python install steps again to make sure you have a working Python installation.

5. PARALLELIZATION IN PYTHON

Multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Local by using sub-processes instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The multiprocessing module also introduces APIs which do not have analogs in the threading module. A prime example of this is the Pool object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using Pool.

6. IMPLEMENTATION

6.1 Sequential Execution

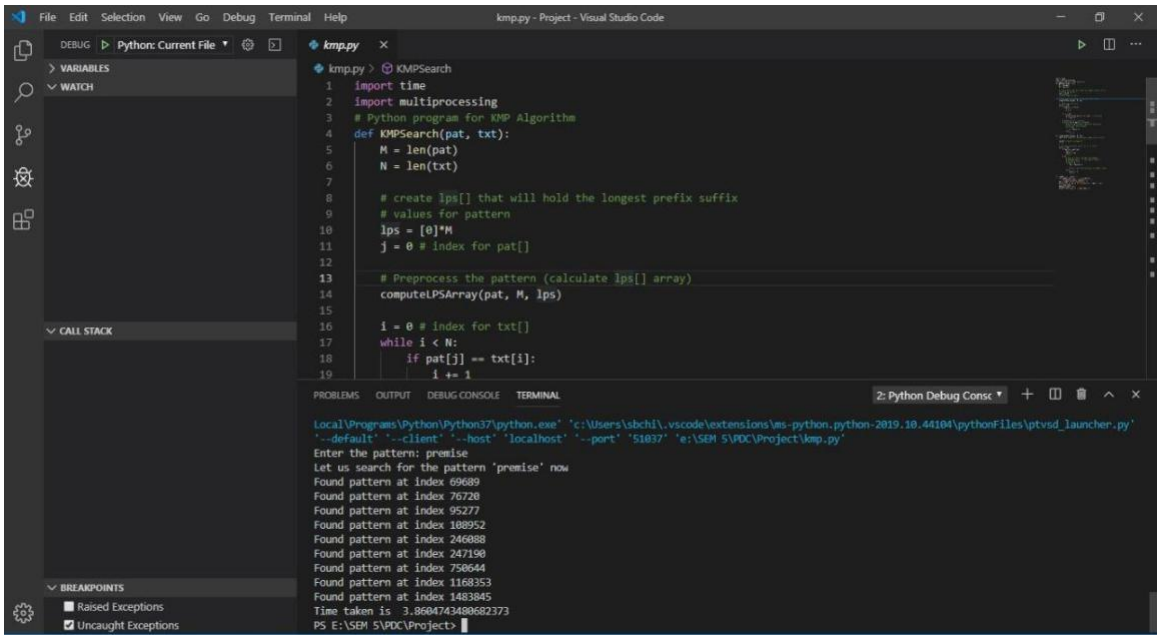


Figure 1 – Output of Sequential Execution

6.2 Parallel Execution

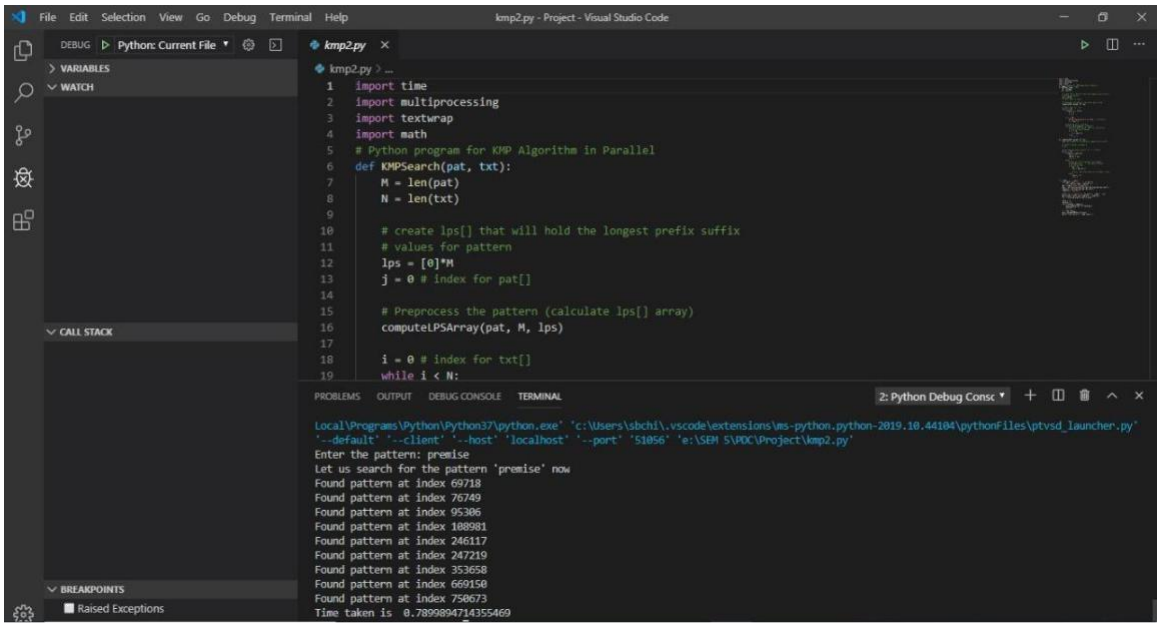


Figure 2 – Output of Parallel Execution

6.3 Evaluation Result – Line Graph

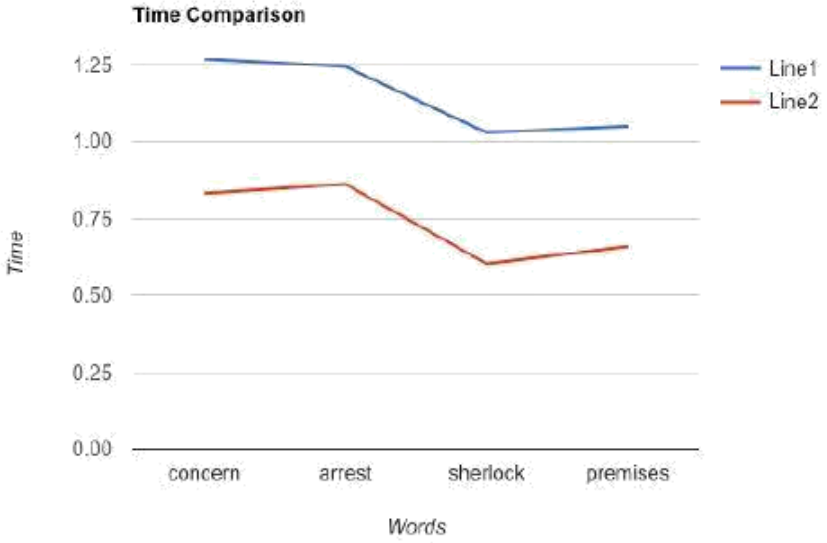


Figure 3 – Running time Comparison of Serial and Parallel implementation

6.4 Evaluation Result – Bar Graph

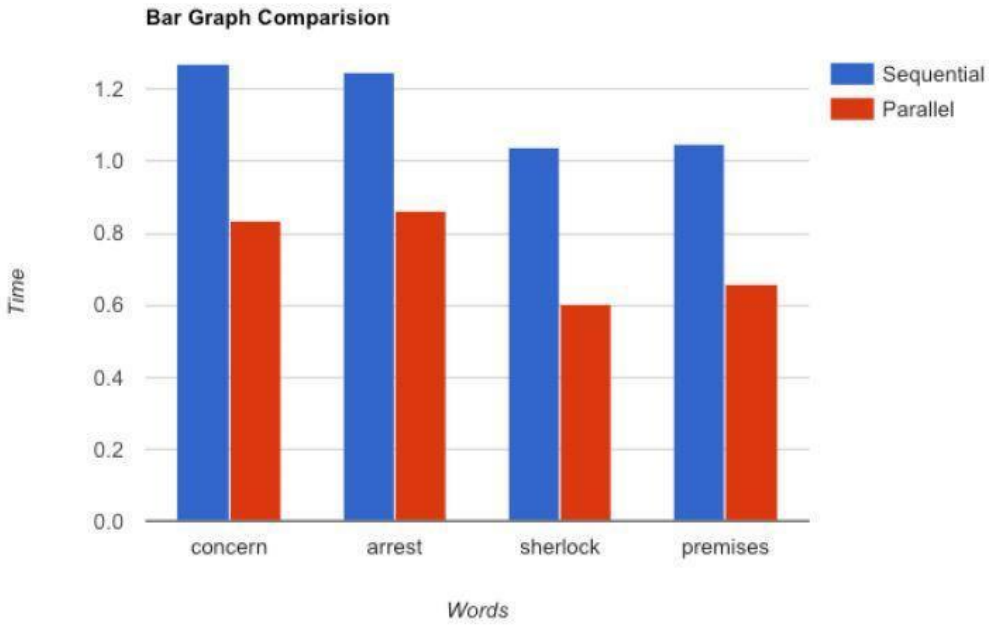


Figure 4 - Running time Analysis of Serial and Parallel implementation

6.5 Performance parameters

- Speed – Speed of executing the program increases by 3.5 times the original speed on parallelisation.
- Size of the image – Different Images have different sizes, parallelisation makes it quicker as the images are being executed in different cores at the same time, whereas in the normal case it is linear and done one at a time.
- Length of the hashed string - We use BytesIO to open each Image in the array in binary format and encrypt it using Base64 encryption, This creates a unique hashed string for each image.
- Number of cores present in the PC.
  Number of threads that can be processed at the same time.
  (Multi threading)

A technique by which a single set of code can be used by several processors at different stages of execution.

7. CONCLUSION

Knutt-Morris-Pratt algorithm has been successfully implemented by proving that the parallel program was 1.53 times faster when compared to the normal program (when a string is considered). The same algorithm was extended to images, where images can be searched using the same concept. We use BytesIO to open each Image in the array in binary format and encrypt it using Base64 encryption, This creates a unique hashed string for each image. The dataset and the input image are

stored in different folders and are extracted into 2 different arrays and after encoding to base64 we pass it to the KMP search function already defined at the earlier stage. A little knowledge on encryption and core python was also gained.

8. RESULTS AND DISCUSSION

From the result we got, we can clearly say that KMP algorithm used in parallel computing gives faster result than compared to its implementation in serial.

9. REFERENCES

1. AHO, A.V., 1990, Algorithms for finding patterns in strings. in Handbook of Theoretical Computer Science, Volume A, Algorithms and complexity, J. van Leeuwen ed., Chapter 5, pp 255-300, Elsevier, Amsterdam.
2. AOE, J.-I., 1994, Computer algorithms: string pattern matching strategies, IEEE Computer Society Press.
3. BAASE, S., VAN GELDER, A., 1999, Computer Algorithms: Introduction to Design and Analysis, 3rd Edition, Chapter 11, Addison-Wesley Publishing Company.
4. BAEZA-YATES R., NAVARRO G., RIBEIRO-NETO B., 1999, Indexing and Searching, in Modern Information Retrieval, Chapter 8, pp 191-228, Addison-Wesley.
5. BEAUQUIER, D., BERSTEL, J., CHRÉTIENNE, P., 1992, Éléments d'algorithmique, Chapter 10, pp 337-377, Masson, Paris.
6. CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L., 1990. Introduction to Algorithms, Chapter 34, pp 853-885, MIT Press.

7. CROCHEMORE, M., 1997. Off-line serial exact string searching, in Pattern Matching Algorithms, ed. A. Apostolico and Z. Galil, Chapter 1, pp 1-53, Oxford University Press.
8. CROCHEMORE, M., HANCART, C., 1999, Pattern Matching in Strings, in Algorithms and Theory of Computation Handbook, M.J. Atallah ed., Chapter 11, pp 11-1--11-28, CRC Press Inc., Boca Raton, FL.
9. CROCHEMORE, M., LECROQ, T., 1996, Pattern matching and text compression algorithms, in CRC Computer Science and Engineering Handbook, A. Tucker ed., Chapter 8, pp 162-202, CRC Press Inc., Boca Raton, FL.
10. CROCHEMORE, M., RYTTER, W., 1994, Text Algorithms, Oxford University Press.
11. GONNET, G.H., BAEZA-YATES, R.A., 1991. Handbook of Algorithms and Data Structures in Pascal and C, 2nd Edition, Chapter 7, pp. 251-288, Addison-Wesley Publishing Company.
12. GOODRICH, M.T., TAMASSIA, R., 1998, Data Structures and Algorithms in JAVA, Chapter 11, pp 441-467, John Wiley & Sons.
13. GUSFIELD, D., 1997, Algorithms on strings, trees, and sequences: Computer Science and Computational Biology, Cambridge University Press.
14. HANCART, C., 1992, Une analyse en moyenne de l'algorithme de Morris et Pratt et de ses raffinements, in Théorie des Automates et Applications, Actes des 2e Journées FrancoBelges, D. Krob ed., Rouen, France, 1991, PUR 176, Rouen, France, 99-110.
15. HANCART, C., 1993. Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte, Ph. D. Thesis, University Paris 7, France.