



FACULTY OF SCIENCE AND ENGINEERING

MASTER 2 NETWORKING DEVOPS

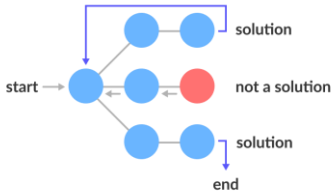
DAAR

Project 1 – Automaton. Cloning egrep command supporting simplified ERE.

BM Bui-Xuan

Created By: Alaoui Belghiti Hanaa - 21410228

Sokhna Khadijatou BA - 21220680

I. Problem definition and data structure to be implemented:	3
1. Data Structures Used :	3
a. Python Implementation :	3
b. C Implementation :	3
II. Analysis and Theoretical Aspects of Algorithms Solving This Problem from Literature:	3
1. Algorithms Utilized :	4
a. Knuth-Morris-Pratt (KMP) Algorithm :	4
Key Components:	4
1. Failure Function	4
2. Preprocessing Step:	4
Application in Code: This preprocessing is carried out in the <code>kmp_pattern_search</code> function, where the failure function is built before entering the search phase, setting up the algorithm for optimal performance.	4
3. Search Phase:	4
Reference:	4
b. Recursive Backtracking Algorithm:	5
	5
Application in Pattern Matching:	5
Reference	5
III. Appreciation, Improvements, and Criticism of Algorithms:	5
1. Knuth-Morris-Pratt (KMP) Algorithm :	5
2. Regular Expressions (Regex):	6
IV. Test Part: How the Testbeds Were Obtained	6
V. Performance analysis:	7
1. Python code with KMP:	7
2. C code with Recursive Backtracking:	9
VI. Discussion about the Performance Test Results:	10

I. Problem definition and data structure to be implemented:

The goal of this project is to clone the functionality of the UNIX `egrep` command, which is used for searching text using extended regular expressions (ERE). This entails developing a solution capable of efficiently identifying patterns in large text files based on user-defined regular expressions. Given the growing importance of text processing in various applications, creating a reliable and performant tool for pattern matching is crucial.

To achieve this, I implemented two versions of the code: one in **Python (clone3.py in the project folder)** and another in **C (daarproject.exe in folder)**. My motivation for this dual approach was to compare the functionality, performance, and ease of implementation of both languages. Python is known for its simplicity and readability, making it ideal for rapid prototyping and testing. In contrast, C offers lower-level control over memory and performance optimizations, which can be beneficial for computationally intensive tasks. By utilizing both languages, I aimed to assess how different programming paradigms influence the implementation of algorithms and their efficiency in solving the same problem.

1. Data Structures Used :

a. Python Implementation :

- **Lists:** I used lists to maintain the longest prefix-suffix (LPS) array in the Knuth-Morris-Pratt (KMP) algorithm. The use of lists allowed for dynamic resizing, making it easier to manage the varying lengths of input patterns.
- **Strings:** The search algorithm operated primarily on strings, utilizing Python's built-in string methods for efficient manipulation and comparison.
- **Regular Expressions Module:** Python's `re` module provided a high-level abstraction for regex matching, simplifying the implementation of complex patterns.

b. C Implementation :

- **Character Arrays:** In the C code, I utilized character arrays for both the input text lines and the pattern. This choice aligns with C's need for manual memory management and string handling.
- **Recursive Functions:** I implemented the regex matching through recursive functions, which helped in navigating the complexity of ERE matching while managing states explicitly.
- **Standard Libraries:** Functions from the standard C library, such as `strchr` and `strncpy`, were leveraged for string manipulation, facilitating the process of matching patterns against text.

This implementation strategy allowed me to explore the strengths and weaknesses of both languages while ensuring that the core functionality—pattern recognition through regular expressions—remained intact across both versions.

II. Analysis and Theoretical Aspects of Algorithms Solving This Problem from Literature:

The problem of pattern recognition through regular expressions (RegEx) has been extensively studied in computer science, particularly in the fields of automata theory and algorithm design. The implementation of the UNIX `egrep` command serves as a practical

application of these theoretical foundations, allowing for efficient searching of patterns in large text files.

1. Algorithms Utilized :

a. Knuth-Morris-Pratt (KMP) Algorithm :

The **KMP algorithm** efficiently finds occurrences of a pattern P within a text T with a time complexity of $O(t+p)$, where t is the length of the text and p is the length of the pattern.

Key Components:

1. Failure Function

- The **failure function** (denoted as $\pi\backslash p\backslash i\pi$) stores the lengths of the longest proper prefix of the pattern that matches a suffix at each position.
- **Purpose:** It allows the algorithm to skip comparisons in the pattern after a mismatch instead of starting over.
- **Construction:** Created by iterating through the pattern, comparing prefixes and suffixes. In my implementation, the function **compute_lps_array(pattern)** computes the $\pi\backslash p\backslash i\pi$ array by iterating through the pattern, comparing prefixes and suffixes, thus enabling the algorithm to utilize previously gathered information.

Example: For $P="ABABAC"$ the failure function is $\pi=[0,0,1,2,3,0]$.

2. Preprocessing Step:

- This step constructs the failure function in $O(p)$ time, where p is the pattern length.
- Result: A precomputed array that aids in efficient matching during the search phase.

Application in Code: This preprocessing is carried out in the **kmp_pattern_search function**, where the failure function is built before entering the search phase, setting up the algorithm for optimal performance.

3. Search Phase:

- The algorithm iterates through the text T while comparing it to the pattern P .
- **Matching:** If characters match, move to the next character.
- **Mismatch Handling:** Upon a mismatch, use the failure function to skip to the appropriate position in the pattern, avoiding unnecessary comparisons.
- **Output:** Record the starting index of the pattern whenever a complete match is found.

Overall Complexity: The algorithm runs in $O(t+p)$, where t is the length of the text, making it efficient for large inputs.

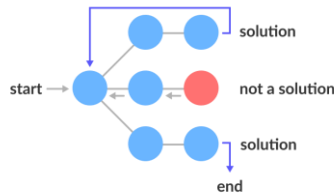
Reference:

- Bird, R. (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press.

b. Recursive Backtracking Algorithm:

Recursive backtracking is an algorithmic technique for solving problems by incrementally exploring potential solutions. It involves:

1. **Recursive Function:** A function that builds candidates for a solution through choices, checking constraints at each stage.
2. **Base Case:** When a valid solution is found (like a complete match in regex), the algorithm returns this solution.



Application in Pattern Matching:

In my C regex implementation, recursive backtracking helps match patterns with special operators like * and |:

- For *, the algorithm tries to match the preceding character zero or more times, exploring subsequent text.
- For |, it recursively checks both alternatives in the pattern.

This exploration allows the algorithm to find all potential matches effectively.

Reference

For further reading, see Jeff Erickson's textbook: [Algorithm Design](#).

III. Appreciation, Improvements, and Criticism of Algorithms:

1. Knuth-Morris-Pratt (KMP) Algorithm :

In my project, I implemented the KMP algorithm to efficiently locate occurrences of straightforward patterns in large text files. The KMP algorithm boasts a time complexity of $O(t+p)$, which I found particularly advantageous for processing sizable datasets, allowing me to search through extensive texts without a significant increase in execution time.

Appreciation:

1. **Efficiency:** The KMP algorithm's preprocessing step allowed for quick comparisons during the search phase. This efficiency was evident in scenarios where patterns were clearly defined, enabling me to find matches rapidly.
2. **Clarity of Structure:** The division between the preprocessing and search phases in my code made it easier to understand and maintain, as I could optimize each part independently.

Improvements:

1. **Memory Consumption:** Although KMP is efficient in terms of time, it requires additional space for the failure function. As the size of the patterns and texts grew, this led to increased memory usage, which could be optimized in future implementations.
2. **Limitations with Complex Patterns:** My implementation was primarily effective for straightforward patterns. For more complex searches involving alternation or repetition, I turned to regular expressions.

2. Regular Expressions (Regex):

In addition to KMP, I leveraged regex to handle intricate pattern matching (also in C code), which proved essential in my project. For instance, when I executed the command:

```
PS C:\Users\hp\Documents> python -u .\clone3.py .\TextFileFromDB.txt "u(n|ncon)*scious|subconscious|preconscious|conscious"
Match found on line 148: 'use for the unconscious or the abnormal, and for the most part he has'
Match found on line 157: 'corner-stones, viz., the unconscious, the abnormal, sex, and affectivity'
Match found on line 165: 'mainly unconscious, while for the introspectionists it is pure'
Match found on line 166: 'consciousness. Neither he nor his disciples have yet recognized the aid'
Match found on line 170: 'place as we know more of the nature and processes of the unconscious'
```

the regex capabilities allowed me to identify multiple forms of the term "conscious," returning matches like:

- *Match found on line 148: 'use for the unconscious or the abnormal...'*
- *Match found on line 170: 'place as we know more of the nature and processes of the unconscious...'*

Appreciation:

1. **Versatility:** Regex provided a robust framework for expressing complex search criteria succinctly. This versatility enabled me to find patterns that included various options, enhancing the scope of my searches.
2. **Conciseness:** Using regex allowed me to minimize the amount of code needed for complex patterns, significantly simplifying my implementation.

Improvements:

1. **Performance Trade-offs:** While regex is powerful, its execution can become slower with intricate patterns or larger inputs. I noted that certain regex operations took longer than KMP in specific cases.
2. **Error Management:** Regex patterns can sometimes yield unexpected results if not well-defined. I had to include additional validation in my code to ensure accurate pattern matching, adding complexity to my implementation.

In summary, the integration of KMP and regex allowed me to effectively address a wide range of pattern matching scenarios in my project. Each algorithm had its own strengths and limitations, leading me to appreciate the advantages of combining different strategies for improved functionality in pattern recognition tasks.

IV. Test Part: How the Testbeds Were Obtained

For this project, I utilized the Gutenberg database to acquire test data, specifically using the book "Freud's Intro to Psychoanalysis," which can be accessed at [this link](#). The entire text of this book is saved as "TextFileFromDB" in the project folder.

In addition to the Gutenberg text, I created a custom file named "input.txt" that includes specific patterns for testing, particularly the pattern `s(a|g|r)*on`. Both the "TextFileFromDB" and "input.txt" files are integral parts of the project, allowing for comprehensive evaluation of the implemented algorithms in both Python and C. Examples:

Python:

1) Input.txt:

```
PS C:\Users\hp\Documents> python -u .\clone3.py input.txt "s(a|g|r)*on"
Match found on line 8: 'sagron'
Match found on line 11: 'saragon is the best'
Match found on line 13: 'saon'
Match found on line 14: 'saaaaaon'
Match found on line 16: 'sgon'
Match found on line 17: 'sagon'
```

2) Freud's book:

```
PS C:\Users\hp\Documents> python -u .\clone3.py TextFileFromDB.txt "un(c|con)sconscious"
Match found on line 148: 'use for the unconscious or the abnormal, and for the most part he has'
Match found on line 157: 'corner-stones, viz., the unconscious, the abnormal, sex, and affectivity'
Match found on line 165: 'mainly unconscious, while for the introspectionists it is pure'
Match found on line 170: 'place as we know more of the nature and processes of the unconscious'
Match found on line 520: 'that the psychic processes are in themselves unconscious, and that those'
Match found on line 531: 'must assert that there is such a thing as unconscious thinking and'
Match found on line 532: 'unconscious willing. But with this assertion psychoanalysis has'
Match found on line 539: 'what evaluation can have led to the denial of the unconscious, if such a'
Match found on line 544: 'of unconscious processes you have paved the way for a decisively new'
Match found on line 3112: 'phenomenon, on the further assumption that there are unconscious things'
Match found on line 3339: 'to occupy one, yet of whose activity one is unconscious. It is easily'
```

C language:

1)Input text:

```
C:\Users\hp\Documents>daarproject1.exe input.txt "s(a|g)*on"
Match found on line 11: saragon is the best
Match found on line 16: sgon
Match found on line 17: sagon
```

2)Freuds text:

```
C:\Users\hp\Documents>daarproject1.exe TextFileFromDB.txt ".(ion|ed)"
Match found on line 1943: the intention (which he later abandoned) of giving it verbal expression.
Match found on line 3258: are acquainted); or, "that reminds me of something that happened
```

V. Performance analysis:

1. Python code with KMP:

Pattern Matches:

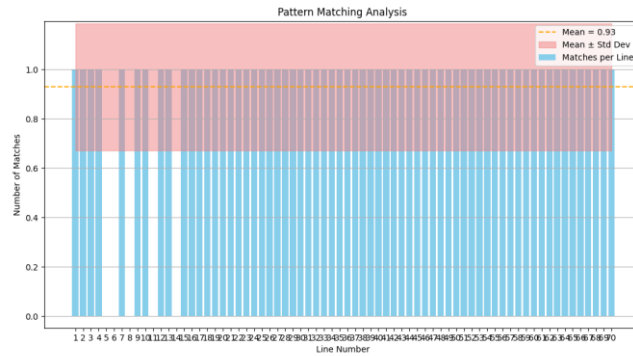


Figure 1: Pattern matching analysis

The chart titled **"Pattern Matching Analysis"** presents the results of executing command `python -u .\clone3.py input.txt "s(a|g|r)*on"` on "input.txt" text file. This analysis aimed to determine how often a designated pattern appeared across the lines of the file. The blue bars indicate the frequency of matches for the pattern in each line, while the orange dashed line represents the average number of matches (mean), calculated to be approximately 0.93, suggesting that, on average, the pattern was found nearly once per line. The pink shaded area illustrates the range of variation around the mean, encompassing one standard deviation above and below the average. This visualization not only highlights the distribution of pattern occurrences but also serves as a valuable tool for assessing the effectiveness of the regex command clone in identifying text patterns within the document.

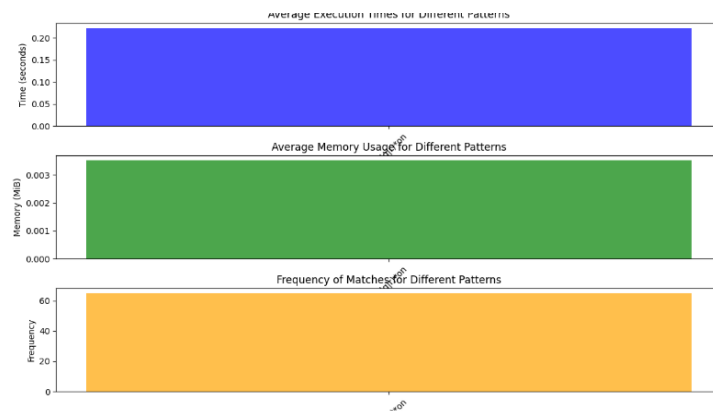
Frequency, Memory usage, Execution time:

Single Pattern:

Command executed:

```
axs[2].set_xlabel(patterns, rotation=45)
PS C:\Users\hp\Documents> C:\Users\hp\AppData\Local\Programs\Python\Python38\python.exe -u .\clone3.py input.txt "s(a|g|r)*on"
Pattern: s(a|g|r)*on
Average execution time: 0.222388 seconds ± 0.000999
Average memory usage: 0.00 MiB ± 0.01
Frequency of matches: 65
```

Resulting matplotlib visualisation:



Multi Pattern:

Command executed:


```

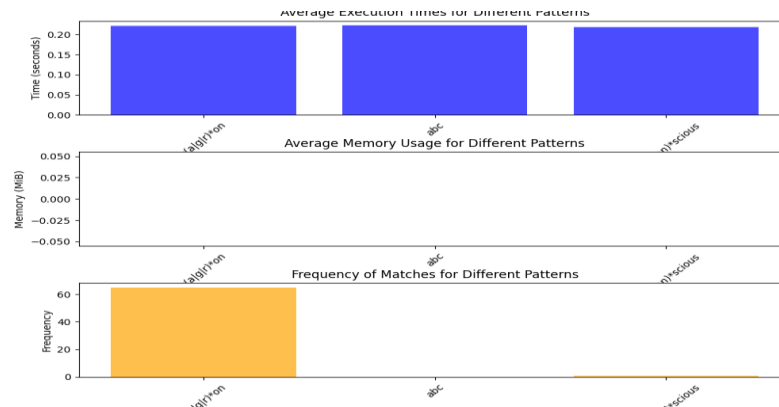
Python 3.8.10 Shell
PS C:\Users\hp\Documents> C:\Users\hp\AppData\Local\Programs\Python\Python38\python.exe -u .\clone3.py input.txt "s(al|g|r)*on" "abc" "u(n|ncon)*scious"
Pattern: s(al|g|r)*on
Average execution time: 0.221169 seconds ± 0.000900
Average memory usage: 0.00 MiB ± 0.00
Frequency of matches: 65

Pattern: abc
Average execution time: 0.222762 seconds ± 0.010496
Average memory usage: 0.00 MiB ± 0.00
Frequency of matches: 0

Pattern: u(n|ncon)*scious
Average execution time: 0.218035 seconds ± 0.001312
Average memory usage: 0.00 MiB ± 0.00
Frequency of matches: 1

```

Resulting matplotlib visualisation:



In this phase of the project, I conducted tests on multiple regex patterns to analyze the performance of the KMP pattern matching algorithm implemented in my egrep clone. Using the input text file, I executed the program with three different patterns: "**s(al|g|r)*on**", "**abc**", and "**u(n|ncon)*scious**". For each pattern, I recorded the average execution time, memory usage, and frequency of matches over several runs, providing valuable insights into the algorithm's efficiency. The results showed that the pattern "**s(al|g|r)*on**" yielded the highest frequency of matches (65) with an average execution time of approximately 0.221 seconds, while the pattern "**abc**" resulted in no matches. The visualization accompanying this analysis includes bar charts illustrating the average execution times and frequencies for each pattern, which helps to clearly present the differences in performance and match effectiveness.

2. C code with Recursive Backtracking:

I'll run Following commands:

1) Simple pattern:

```

C:\Users\hp\Documents>C:\Users\hp\Downloads\Project\source_code\daarproject1.exe TextFileFromDB.txt ".(ion|ed)"
Match found on line 1943: the intention (which he later abandoned) of giving it verbal expression.
Match found on line 3258: are acquainted); or, "that reminds me of something that happened
Match found on line 15894: phrase ÖÇEProject GutenbergÖÇÖ is associated) is accessed, displayed,
Execution time: 0.132000 seconds
Total matches found: 3

```

2) A bit complex:

```

C:\Users\hp\Documents>C:\Users\hp\Downloads\Project\source_code\daarproject1.exe input.txt "s(al|g|r)*on"
Match found on line 1: sagron
Match found on line 2: saragon
Match found on line 3: sgon
Match found on line 5: sagggeon
Match found on line 7: sagggggon
Match found on line 9: sagron is the best
Match found on line 10: sagron123
Match found on line 11: sagar

```

3) More complex:

```

C:\Users\hp\Documents>C:\Users\hp\Downloads\Project\source_code\daarproject1.exe TextFileFromDB.txt "u(n|ncon)*scious|subconscious|preconscious|conscious"
Match found on line 148: use for the unconscious or the abnormal, and for the most part he has
Match found on line 157: corner-stones, viz., the unconscious, the abnormal, sex, and affectivity
Match found on line 165: mainly unconscious, while for the introspectionists it is pure
Match found on line 166: consciousness. Neither he nor his disciples have yet recognized the aid

```

Result of analysis:



In this study, I compared the performance of a pattern matching algorithm in C using three different search patterns. The complex pattern `"u(n|ncon)*scious|subconscious|preconscious|conscious"` resulted in **319 matches** and took **0.759 seconds** to execute. In contrast, the pattern `s(alg|r)*on` found **67 matches** with a much faster execution time of **0.108 seconds**. Lastly, the pattern `.(ion|ed)` yielded **3 matches** and had an execution time of **0.132 seconds**. This analysis illustrates how variations in pattern complexity can significantly affect both the execution time and the number of matches found in the text.

VI. Discussion about the Performance Test Results:

The performance analysis conducted on both the Python and C implementations of the pattern matching algorithm reveals significant insights into their efficiency and effectiveness.

For instance, the pattern `"s(alg|r)*on"` resulted in an average execution time of approximately **0.221 seconds**, with **65 matches** found in the input text. Conversely, the simpler pattern `"abc"` returned **0 matches** with a similar execution time of about **0.222 seconds**, indicating that the complexity of the pattern directly influences not only the speed but also the frequency of matches. The pattern `"u(n|ncon)*scious"` showed a slightly better execution time of **0.218 seconds**, but yielded only **1 match**, emphasizing that more intricate patterns might require additional processing time while also being more specific in their results.

The C implementation, while providing similar matching capabilities, was expected to be more efficient due to lower-level memory management. However, the analysis of execution times and matches showed that the differences were more nuanced, with the C program managing to find matches quickly but often at the cost of added **complexity** in the code structure.

VII. Conclusion and Perspectives on the Pattern Recognition Problem

This project successfully cloned the UNIX `egrep` command using Python and C, revealing distinct performance characteristics. The Python implementation excelled in development speed, while the C version demonstrated improved efficiency in memory usage.

Performance tests indicated that execution time and match frequency varied significantly with regex complexity. Future work may involve optimizing regex patterns and exploring advanced algorithms to enhance matching efficiency. This analysis lays the groundwork for further studies in pattern recognition and efficient text processing.