UNIVERSIDADE FEDERAL DA BAHIA
INSTITUTO DE COMPUTAÇÃO

JEAN LOUI BERNARD SILVA DE JESUS

FAST PATTERN MATCHING IN SEQUENCES WITH CONDITIONS

A THEORETICAL MODEL FOR PERFECT PATTERN SEARCH

SALVADOR - BA
2023

JEAN LOUI BERNARD SILVA DE JESUS

FAST PATTERN MATCHING IN SEQUENCES WITH CONDITIONS

A THEORETICAL MODEL FOR PERFECT PATTERN SEARCH

Abstract:
This paper describes a new algorithm for fast pattern searching in sequences, allowing arbitrary numerical comparisons, and introduces the concept of a theoretical model for perfect pattern searching, where each element of the sequence is compared only once in all cases.

Keywords: pattern search, pattern matching, searching, dynamic comparisons, optimum algorithm, trie memory, regular expression, perfect model

SALVADOR - BA
2023

Table of Contents

Table of contents **3**

1 Summary **4**
1.1 Motivation 4
1.2 Introducing a new method to solve the addressed problem 4

2 Algorithm **5**
2.1 Representations 5
2.2 Step by step guide 5
2.3 Complexity 6
2.4 Applications 6
2.5 Theorical perfect pattern search model 6

3 Experiments and Results **7**
3.1 Comparing with the Naive Brute-force algorithm 7
3.1.1 Results of the Naive Brute-force algorithm 7
3.1.2 Results of the Jean-Loui-Bernard (JLB) algorithm 8
3.2 Comparing with the other fast algorithms (equality comparison) 8
3.2.1 Results of the Knuth-Morris-Pratt (KMP) algorithm 9
3.2.2 Results of the Booyer Moore (BM) algorithm 9
3.2.3 Results of the Rabin-Karp (RK) algorithm 10
3.3 Performance rate from the results 10

4 Conclusion **11**

References **11**

1 Summary

1.1 Motivation

Currently, the majority of fast pattern searching algorithms in sequences are designed only to compare exact occurrences to the pattern, such as Knuth–Morris–Pratt (KMP) [1], Boyer-Moore (BM) [2], Rabin-Karp (RK) [3], etc. Now, consider this following problem and the main issue addressed to this paper: in the numerical sequence below, we need to find all values that are lower than X and have the same number of digits.

```
Sequence = 9 9 5 9 6 4 9 7 8 0 1
X = 9 5 9 7
```

The occurrences found in this sequence are:

```
9596 at position 1
5964 at position 2
6497 at position 4
4978 at position 5
7801 at position 7
```

For this problem, which algorithm should we use?

We could even go further and consider other problems, such as searching for larger values or multiples of X — or even by looking for occurrences that have a specific relationship with X for example, X could be any validating formula $f : \Sigma \to \{0, 1\}$ that may behave differently for each subsequence or at each successful or unsuccessful step using a state-based system.

For searching for occurrences with specific characteristics, a solution would be the trivial algorithm with a complexity of $O(n \times m)$ — more commonly known as the naive brute-force algorithm. However, as of now, the existence of an efficient pattern searching algorithm that addresses this problem is unknown.

1.2 Introducing a new method to solve the addressed problem

In this paper, we will introduce a new method for fast pattern searching in sequences called Jean-Loui-Bernard (JLB), with a complexity of $O(n + m)$ to address the previously described problem. We will discover that this method not only allows conventional comparisons but also any kind of numerical comparison, including complex equations and functions that has a complexity of $O(1)$. Additionally, the method presented in this paper serves as a theoretical model for perfect pattern searching in sequences, where each element is compared only once in all cases during the search phase.

2 Algorithm

The JLB algorithm involves comparing the integer representation of a pattern P with the integer representation of each window of size $m = |P|$ in the sequence S. Similarly to the Rabin-Karp (RK) algorithm [3], the comparison between them must have a complexity of $O(1)$ and occur within the computer's registers.

For this reason, the pattern size must be smaller than $\log_{|A|} 2^R$, where A is the ordered set of all possible symbols that a valid occurrence in P can contain, and R is the size of the hardware register that will perform the comparison. In other words, the pattern size is upper-bounded by the processor architecture and the efficiency of this algorithm depends exclusively on the environment in which it will be executed. We will return to this issue in the section 2.4.

2.1 Representations

The sequence S must be a list of integer values. If S is a character sequence, each character must be converted to its respective integer representation, based on its position $\{0, ..., |A| - 1\}$ in the alphabet A. We can perform the conversion during the search phase, using a function $r : A \to \mathbb{N}$. The calculation of the integer representation of P — as well as the frame initialization — is given by this formula: $P_r = P_{m-1} \times |A|^{m-1} + ... P_1 \times |A|^1 + P_0 \times |A|^0$.

In this paper, we will assume P as a fixed pattern to be compared. However, it could be replaced by a validating function $f : \mathbb{N} \to \{0, 1\}$.

2.2 Step by step guide

See the step-by-step guide of the algorithm JLB below. Check out a basic implementation of this method by clicking on this link: https://github.com/JeanExtreme002/JLB-Algorithm.

1. Convert the pattern P to an integer P_r;
2. Create an integer variable $Frame$;
3. Read and store at $Frame$ the first valid subsequence — contains only valid symbols stipulated in the alphabet A — with m elements of S;
4. Compare $Frame$ with P_r — use any comparison function $f : \mathbb{N}, \mathbb{N} \to \{0, 1\}$ in $O(1)$ time to check if it is an occurrence;
5. Shift the digits of $Frame$ to the left — subtract $Frame_{m-1} \times |A|^{m-1}$ and multiply by $|A|$;
6. Read the next unread value in the sequence and add it to $Frame$. If the value is invalid — symbol is not contained in the alphabet A — go back to the step 3;
7. Repeat the three previous steps until reaching the end of S.

2.3 Complexity

The calculation of P_r has a complexity of $O(m)$ because we need to read and compute all m elements of P. In the search phase, the shifting and comparison procedures occurs in $O(1)$ time. These procedures are performed for each element of S. Therefore, the search phase occurs in $O(n)$ time, where $n = |S|$.

A significant advantage of this method over others is that, since the search always involves reading the next element in the sequence and never going back to a previously read element, it is guaranteed that the search will always be linear. It means that each element of S is compared only once and the efficiency of the algorithm depends on the hardware. This way, we conclude that the JLB algorithm has a complexity of $O(n+m)$ and we introduce the concept of a theoretical model for perfect pattern search in the section 2.5.

2.4 Applications

We employ this method for problems involving searches for small patterns, particularly if the problem requires alternative ways of pattern comparison or dynamic comparisons, given that an excellent feature of this method over others is allowing any kind of comparison in $O(1)$ time, such as finding lower or higher values than P, multiples of P, or even comparing the value based on the result of a complex formula.

For the proper functioning of the method, it is required for arithmetic operations and comparisons of values to occur within the computer's registers in $O(1)$ time. It means that the value of P_r is upper-bounded by the size of the computer's register — section 2. Therefore, in conventional computers, the JLB algorithm is only applicable when the pattern we are searching for is small enough. On a second thought, in supercomputers, we can extend the register capacity, allowing us to compare larger frames in the sequence, making the method more efficient and suitable.

2.5 Theorical perfect pattern search model

The perfect pattern search model is one that has linear complexity and the number of comparisons for each element of the sequence S is always ≤ 1. In practice, the method presented in this paper has a limitation on the size of the pattern, as seen in the section 2 and section 2.4. However, this limitation is at the hardware-level — on the software-level, assuming that we can always increase the size of the registers when needed, the algorithm still keeps working properly, with a perfectly linear search — section 2.3. Then, we can say the JLB algorithm is a software-level idealization of a perfect pattern search model.

3 Experiments and Results

As we know that the number of comparisons per element of the sequence is constant in the JLB algorithm, the idea of this section is evaluating its runtime and compare its results with those of other algoritms. All algorithms will be implemented faithfully in the C++ programming language and we will generate random patterns and sequences, performing 30 times each test with pair (M, N) and taking the average between them, for the following alphabets:

- Binary Alphabet $A = \{0, 1\}$
  - random patterns of sizes $M \in \{4, 8, 16, 32\}$
  - random sequences of sizes $N \in \{100K, 1M, 10M, 100M\}$
- Digit Alphabet $A = \{0, ..., 9\}$
  - random patterns of sizes $M \in \{3, 5, 10, 15\}$
  - random sequences of sizes $N \in \{100K, 1M, 10M, 100M\}$

In this experiment, we expect a linear growth in the runtime of our method as the sequence size increases. Since the comparison is done in $O(1)$ time for any frame, we also expect that the runtime will not vary as the alphabet size or the pattern size increases. The tests will be conducted on the Replit platform, using its basic hardware specifications (free plan), assuming inputs in the proper format, as seen in the section 2.1.

3.1 Comparing with the Naive Brute-force algorithm

Here, we will compare the runtime results of our method with the results of the trivial algorithm — which up to now is the only known algorithm able to solve the addressed problem — searching for values less than the pattern.

You can access the implementation of the naive brute-force algorithm in the C++ language clicking on this link: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/. However, to search for values smaller than the pattern, we needed to make a slight modification to the code by changing the comparison method to less-than.

3.1.1 Results of the Naive Brute-force algorithm

Execution times of Naive Brute-force algorithm in seconds (binary alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 4$	0.0062	0.0278	0.2629	3.2542
$M = 8$	0.0055	0.0220	0.2646	3.0079
$M = 16$	0.0057	0.0275	0.2762	3.2592
$M = 32$	0.0045	0.0332	0.2796	2.9259

Execution times of Naive Brute-force algorithm in seconds (digit alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 3$	0.0007	0.0272	0.2216	2.6257
$M = 5$	0.0055	0.0206	0.1930	2.2566
$M = 10$	0.0067	0.0181	0.2294	2.0671
$M = 15$	0.0012	0.0218	0.1918	2.2333

3.1.2 Results of the Jean-Loui-Bernard (JLB) algorithm

Execution times of Jean-Loui-Bernard algorithm in seconds (binary alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 4$	0.0001	0.0153	0.1769	1.7695
$M = 8$	0.0003	0.0134	0.1457	1.5720
$M = 16$	0.0001	0.0182	0.1510	1.4941
$M = 32$	0.0000	0.0122	0.1413	1.5086

Execution times of Jean-Loui-Bernard algorithm in seconds (digit alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 3$	0.0006	0.0368	0.2074	1.6215
$M = 5$	0.0000	0.0147	0.1689	1.5292
$M = 10$	0.0001	0.0156	0.1489	1.5293
$M = 15$	0.0000	0.0143	0.1773	1.6301

3.2 Comparing with the other fast algorithms (equality comparison)

Although the KMP, BM and RK algorithms are not competitors for solving the addressed problem of using alternative comparison methods [1][2][3] — they are used only to find exact occurrences of the pattern — we will compare their runtime with the JLB's runtime by applying the same tests done for the naive brute-force algorithm, using equality comparison at our method.

Below are the links to the codes used for implementing the algorithms in the C++ language:

- Knuth-Morris-Pratt (KMP): https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- Boyer-Moore (BM): https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
- Rabin-Karp (RK): https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

3.2.1 Results of the Knuth-Morris-Pratt (KMP) algorithm

Execution times of Knuth-Morris-Pratt algoritm in seconds (binary alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 4$	0.0066	0.0229	0.2394	2.6110
$M = 8$	0.0010	0.0236	0.2370	2.4234
$M = 16$	0.0007	0.0363	0.2483	2.4986
$M = 32$	0.0011	0.0319	0.2475	2.4337

Execution times of Knuth-Morris-Pratt algorithm in seconds (digit alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 3$	0.0045	0.0117	0.1465	1.3191
$M = 5$	0.0000	0.0051	0.1146	1.3751
$M = 10$	0.0002	0.0124	0.1113	1.3329
$M = 15$	0.0001	0.0125	0.1367	1.2786

3.2.2 Results of the Booyer Moore (BM) algorithm

Execution times of Boyer-Moore algorithm in seconds (binary alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 4$	0.0062	0.0233	0.2378	2.4463
$M = 8$	0.0007	0.0101	0.2046	2.4003
$M = 16$	0.0010	0.0312	0.1957	2.8818
$M = 32$	0.0009	0.0277	0.2466	2.2169

Execution times of Boyer-Moore algorithm in seconds (digit alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 3$	0.0001	0.0207	0.1695	0.7237
$M = 5$	0.0000	0.0017	0.0359	0.4989
$M = 10$	0.0000	0.0012	0.0253	0.3546
$M = 15$	0.0048	0.0010	0.0299	0.2869

3.2.3 Results of the Rabin-Karp (RK) algorithm

Execution times of Rabin-Karp algorithm in seconds (binary alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 4$	0.0025	0.0365	0.3390	3.4902
$M = 8$	0.0074	0.0380	0.3151	3.5081
$M = 16$	0.0062	0.0396	0.3437	3.4695
$M = 32$	0.0058	0.0333	0.3376	3.6023

Execution times of Rabin-Karp algorithm in seconds (digit alphabet)				
Sizes: N / M	100K chars	1M chars	10M chars	100M chars
$M = 3$	0.0054	0.0443	0.2897	3.0687
$M = 5$	0.0011	0.0253	0.2823	3.1057
$M = 10$	0.0010	0.0387	0.3449	3.0341
$M = 15$	0.0058	0.0335	0.3012	3.0793

3.3 Performance rate from the results

Average performance rate related to JLB (binary alphabet)				
Ratio	100K chars	1M chars	10M chars	100M chars
Naive / JLB	∞	1.9227	1.7775	1.9683
KMP / JLB	∞	1.9667	1.5939	1.5756
BM / JLB	∞	1.5653	1.4474	1.5769
RK / JLB	∞	2.5316	2.1861	2.2285

Average performance rate related to JLB (digit alphabet)				
Ratio	100K chars	1M chars	10M chars	100M chars
Naive / JLB	∞	1.2063	1.2083	1.4541
KMP / JLB	∞	0.5834	0.7258	0.8421
BM / JLB	∞	0.2062	0.3420	0.2951
RK / JLB	∞	1.9370	1.7708	1.9491

Because the results of the tests for 100K characters are not very precise, such column was ignored in the performance rate calculation above.

4 Conclusion

As we observed in the test results, for the binary alphabet, our method proved to be more efficient than all other algorithms and more efficient than the naive brute-force algorithm. We also observed that the runtime of our method tends to grow linearly as the size of the sequence increases and the runtime did not vary significantly with changes in the alphabet size or in the pattern size, as expected.

For the naive brute-force algorithm tests, the results using the binary alphabet were better than the tests using the digit alphabet. As the experiment uses random inputs with well-distributed characters, the larger the alphabet set, the lower the probability of extensive comparisons in the naive brute-force algorithm (it performs only reading and comparison of characters, while our method involves additional operations, then the cost per character is lower for the trivial algorithm). In tests using the binary alphabet, the probability of success in comparing each character is always 50%. On the other hand, for an alphabet with more than 2 characters, the probability may vary depending on the character of the pattern. That is, $Prob(S_i < P_k) = \frac{P_k}{|A|}$. In the worst case, the naive brute-force algorithm will perform $n \times m$ comparisons, while our method will only perform n comparisons.

For exact pattern matching, using the digit alphabet, our method achieved runtime higher but close to the Knuth-Morris-Pratt algorithm for large sequences — with a performance rate tending to increase as the size of the sequence increases — and runtime lower than the Rabin-Karp algorithm. However, our method demonstrated to be extremely slower than the Boyer Moore algorithm.

Unfortunately, we are unable to achieve more precise results due to a lack of hardware, and with the basic hardware specification of the Replit platform we were able to build patterns of at most 15 digits, not being possible to verify whether there would be a significant performance rate gain over the growth of the pattern size, as expected in theory. Nevertheless, we hope that the information contained in this paper be useful for the implementation of more performant systems through the use of the JLB algorithm, which allows for multiple kinds of comparisons at a very low runtime.

References

[1] Donald Ervin Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Association for Computing Machinery*, 20(10):762–772, oct 1977.

[3] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.