ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΝΑΠΤΥΞΗ ΒΙΒΛΙΟΘΗΚΗΣ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΤΑΙΡΙΑΣΜΑ ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ ΥΠΟΣΤΗΡΙΖΟΜΕΝΗ ΑΠΟ ΟΠΤΙΚΟΠΟΙΗΣΗ ΚΑΙ ΣΥΓΚΡΙΣΗ ΤΩΝ ΑΠΟΔΟΣΕΩΝ ΤΟΥΣ ΜΕ BENCHMARKS ΣΕ ΒΙΟΛΟΓΙΚΑ ΔΕΔΟΜΕΝΑ

Διπλωματική Εργασία

του

Αλέξανδρου Κοκοζίδη

Θεσσαλονίκη, Ιούνιος 2019

ΑΝΑΠΤΥΞΗ ΒΙΒΛΙΟΘΗΚΗΣ ΑΛΓΟΡΙΘΜΩΝ ΓΙΑ ΤΑΙΡΙΑΣΜΑ ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ ΥΠΟΣΤΗΡΙΖΟΜΕΝΗ ΑΠΟ ΟΠΤΙΚΟΠΟΙΗΣΗ ΚΑΙ ΣΥΓΚΡΙΣΗ ΤΩΝ ΑΠΟΔΟΣΕΩΝ ΤΟΥΣ ΜΕ BENCHMARKS ΣΕ ΒΙΟΛΟΓΙΚΑ ΔΕΔΟΜΕΝΑ

Αλέξανδρος Κοκοζίδης

Πτυχίο Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 2013

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπουσα Καθηγήτρια
κα Μαρία Σατρατζέμη

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ηη/μμ/εεεε

κα Σατρατζέμη Μαρία	κ. Κασκάλης Θεόδωρος	κα Κολωνιάρη Γεωργία
Καθηγήτρια	Αναπληρωτής Καθηγητής	Επίκουρη Καθηγήτρια
.....	.....	.....

ΑΛΕΞΑΝΔΡΟΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΚΟΚΟΖΙΔΗΣ, ΑΜ: 029/18

.....

Abstract

String searching, the act of finding a string in an even larger text, is an important and well-studied field in computer science, with applications ranging from simple tasks such as finding words in a text editor or performing a spell check, to more complex ones like system intrusion detection systems and DNA sequence matching. Their importance creates the need for easily understandable, efficient implementations of those algorithms. This thesis attempts to provide an in-depth insight in the 35 string search algorithms presented by Charras and Lecroq, by presenting their underlying functionality, implementing them according to the object-oriented paradigm in the Java programming language, comparing them on a custom, Java-based benchmark suite on huge biological sequence data and animating them by implementing a visualization suite.

Keywords: String searching, Pattern matching, String searching algorithms, Pattern matching algorithms, Java, Benchmarks, Bioinformatics benchmark, Visualization, Javascript, Brute Force, Deterministic Finite Automaton, Karp Rabin, Shift Or, Morris Pratt, Knuth Morris Pratt, Simon, Colussi, Galil Giancarlo, Apostolico Crochemore, Not So Naïve, Boyer Moore, Turbo Boyer Moore, Apostolico Giancarlo, Reverse Colussi, Horspool, Quick Search, Tuned Boyer Moore, Zhu Takaoka, Berry Ravindran, Smith, Raita, Reverse Factor, Turbo Reverse Factor, Forward DAWG Matching, Backward Nondeterministic DAWG Matching, Backward Oracle Matching, Galil Seiferas, Two Way, String Matching On Ordered Alphabets, Optimal Mismatch, Maximal Shift, Skip Search, KMP Skip Search, Alpha Skip Search

Contents

Images

1 Introduction

1.1 Problem Definition

String searching, or string matching, is the process of finding the occurrences of one or more, smaller strings, called patterns, in a larger string, called the text. String searching algorithms are an important class of string algorithms that implement above process. Assuming there is a text *T* of *n* characters and a pattern *P* of *m* characters, both of which are constructed of a finite set of characters called the alphabet Σ of size σ, the string searching algorithm aims to return the position of the first occurrence, or even the positions of all the occurrences, of *P* in *T*.

With more than 80 algorithms discovered since 1970 [1] and a continuous emergence of improved versions of them in academic literature, string searching algorithms are one of the most extensively studied branches of computer science. Their field of applications is broad. Ranging from simpler ones, like the word search functionality or the automated spell checker in a text editor, to more complex ones, such as plagiarism detection software, web search engines, system intrusion detection systems and DNA sequence matching in bioinformatics. The sheer amount and accelerating pace of growth of data found in those applications has made imperative the design of time-efficient string search algorithms.

Several aspects of the input data in a string searching procedure can impact the performance an algorithm can have in carrying out the process. Attributes such as the length of the pattern and the size of the alphabet (ASCII, english: {A..Za..z}, binary: {0,1}, biological: {A, C, G, T}) are exploited differently by various algorithms yielding different execution times. Additionally, most of the theoretical work in this field appears to concentrate on analyzing the asymptotic behavior of an algorithm with respect to the number of character comparisons performed in the searching process. The above two points showcase the need to perform comparative tests of the execution time of the algorithms on different scenarios in order to determine which algorithms are more efficient in each one of the presented situations.

In this thesis we are presenting the ported implementations of the 35 string searching algorithms, included in **http://www-igm.univ-mlv.fr/~lecroq/string/ [2]** in C language, using the Java programming language according to the object-oriented pattern presented by Sedgewick in **https://algs4.cs.princeton.edu/home/ [3]**. The object-

oriented environment showcases the distinct phases of the string searching procedure, namely preprocessing and searching. The implementations are also supported with a visualization suite, developed with web-technologies, which aims to provide an animated glimpse on how each algorithm behaves during execution. Finally, we use a string search benchmark suite, developed also in Java, to test the time efficiency of all implemented algorithms on a large amount of biological sequence data.

1.2 Structure

This thesis begins by introducing the basic notions of string searching algorithms in Chapter 2. This chapter also includes compact description of each one of the algorithms coupled with remarks regarding notable differences or additions in their implementations with respect to their C counterparts in the source web site. The full source code is available at the Appendix. In Chapter 3, the method for conducting the benchmarks is discussed, followed by the results, on individual and collective level, and their interpretation. Chapter 4 provides an overview of the general structure of the visualization suite, as well as certain notable implementation details regarding its components. Finally, Chapter 5 wraps up with conclusions concerning the whole process of understanding, implementing in Java, benchmarking and visualizing the string search algorithms provided.

2 Algorithm presentation

2.1 Introduction

In this chapter, we present the algorithms retrieved from [2] and considered in the process of creating their respective implementations in Java, comparing their time-efficiency with a benchmark suite and attempting to animate their functionality by implementing a visualization suite. Each algorithm has its own section consisting of a general description segment focusing in the key points behind its functionality. Additionally, an implementation-specific part is included. It attempts to highlight notable implementation details between our Java implementations and their respective C source code segments given also in [2]. Those details may refer to modifications performed on some parts of the C code for the sake of adapting each algorithm to the object-oriented development pattern, as well as to alterations for the sake of correctness of results.

Generally, two approaches are followed when an algorithm attempts to provide a solution for the string searching algorithm. They are differentiated by whether the pattern or the text is given first as input to the algorithm. Here we consider the first case, where the pattern is given as the sole argument to each algorithm's constructor, with the purpose of being preprocessed. The text is then provided as an argument to a search method that applies the searching procedure.

Another common point between the algorithms presented here is that they all follow the same general framework during execution. Specifically, the string searching procedure is split into two phases, the preprocessing and the searching. The preprocessing phase performs calculations that aim to speed up the actual searching. The searching phase comprises of utilizing what is often referred to as the sliding window. That window has the length of the pattern and is placed at the start of the text. Then the text characters aligned in the window are compared to their respective ones from the pattern. In case of mismatch the window slides to the right. This procedure is repeated until the window has reached the end of the text. The position of the window is denoted by the position of the left-most character of the text that is included by the window.

The most common way to categorize those algorithms is by the order the characters between the window and the pattern are compared. There are four distinct categories: from left to right, from right to left, in a specific order and in any order.

Generally, it is considered that the best results are extracted from algorithms that perform the comparisons from right to left. In the benchmarking section of this thesis, the results will be grouped also according to those categories. The algorithms are separated into the four groups as follows:

- From left to right: Deterministic Finite Automaton, Karp-Rabin, Shift Or, Morris-Pratt, Knuth-Morris-Pratt, Simon, Apostolico-Crochemore, Not So Naïve, Forward DAWG Matching, String Matching on Ordered Alphabets.
- For right to left: Boyer-Moore, Turbo Boyer-Moore, Apostolico-Giancarlo, Reverse Colussi, Zhu-Takaoka, Berry-Ravindran, Reverse Factor, Turbo Reverse Factor, Backward Nondeterministic DAWG Matching, Backward Oracle Matching.
- In a specific order: Colussi, Galil-Giancarlo, Galil-Seiferas, Two Way, Optimal Mismatch, Maximal Shift, Skip Search, KMP Skip Search, Alpha Skip Search.
- In no specific order (order does not matter): Brute Force, Horspool, Quick Search, Tuned Boyer-Moore, Smith, Raita.

All algorithms were implemented in Java in accordance with an object-oriented design pattern resembling the one presented in **https://algs4.cs.princeton.edu/home/** by Sedgewick. A separate class was created for each algorithm. For the sake of consistency, each one has the same interface of public methods. Specifically, each class provides:

- a constructor that accepts a *String* type argument corresponding to the pattern and eventually performs all the preprocessing phase of the algorithm
- a *search* method that accepts a *String* type argument corresponding to the text and that performs the searching procedure; it returns an integer-corresponding to the position of the first occurrence of the pattern in the text
- a *searchAll* method that accepts a *String* type argument corresponding to the text and that performs the searching procedure; it returns a *List* of *integers*, containing the positions of all the occurrences of the pattern in the text

Notably, there was not performed any kind of extensive optimization, since this was not the purpose of this thesis. However, several modifications were performed in most of the algorithms to compensate for a distinct aspect of the C language. Specifically, in C, each string of length x contains $x+1$ symbols, since the last symbol is always the termination symbol ('\0'). Since this does not hold for Java, several alterations had to be made to avoid the algorithms from crushing from indexes that would have been out of bounds. More specific details about where those modifications took place is mentioned in the implementation details segments of each algorithm in this chapter.

2.2 Terms

Some of the terms used during in this thesis are described below:

- The positions of the characters of a string s of length len are 0, 1, .. , len - 2, len-1.
- Σ denotes the alphabet of which the text and pattern are built and σ is the size of said alphabet.
- *P* or *pattern* is the string that we are searching for; the patterns length is denoted by *m.*
- *T* or *text* is the source string where we search for the pattern; the texts length is denoted by *n.*
- *P[i]* refers to the character of the pattern in position *i*; similarly, for *T[i]*.
- *Window*, or *search window* refers to the positions of the text where the pattern is currently aligned.
- *Attempt* refers to the action of comparing all the text characters included by the window with their respective ones from the pattern.
- *Substring* or *factor* of a string *s* is a string *z* if there exist strings *u,v* such that s = uzv.
- *Prefix* of a string *s* is a substring *u* of *s* such that $s = uv$, where *v* is also a (possibly empty) substring of *s*.
- *Suffix* of a string s is a substring *u* of *s* such that $s = xv$, where *v* is also a (possibly empty) substring of *s*.
- *Period* of a string *s* is an integer *p* such that $s[i] = s[i+p]$ *for* $0 \leq i < m-p$.
- *Per(s)* is the minimum period of string *s*.

- A string *s* is *basic* if it cannot be expressed as a power of another string; that is there is no string *z* and integer *k* such that $s = z^k$.
- A string *s* of length *len* is *periodic* if its minimum period *len(s)* is less or equal to *len*/2; it is *non-periodic* otherwise.
- A string *b* is a *border* of a string *s* if it is both a prefix and a suffix of *s.*
- A *reverse* of a string *s*, denoted with s^R, is the string *s[m-1]s[m-2]..s[1]s[0]*.

2.3 Brute-Force

2.3.1 Description

The Brute-Force (BF) [2] algorithm can be regarded as the simplest approach to solving the substring search problem. Its straightforward principal consists of scanning all the text characters sequentially from left to right, starting at the first character, and checking if the next m sequential characters, starting from the current character, match the pattern.

This procedure can be visualized, by placing a 'window' that has the same length as the pattern and is initially positioned at the first character of the text. Two actions can be triggered by the window; an attempt and a shift. During an attempt the pattern characters are compared one by one, from left to right with the text characters aligned with the window. If a mismatch occurs, the attempt stops and the window performs a shift. During a shift the window shifts one character to the left and triggers an attempt. This procedure is repeated until a match is found or the end of the text is reached. An execution example can be viewed in Figure 1.

This algorithm has a worst-case time complexity of O(nm) which can be witnessed when, at each attempt, the mismatch always occurs at the m-th character. For example, we can consider the text T: 'AAAAAA' and the pattern P: 'AAAB', where the first (m-1) characters of the pattern always match and there is always a mismatch at the last one.

Its advantages can be considered the lack of a pre-processing phase as well as the fact that it has no need for extra memory, however its inefficiency becomes apparent due to the backup that occurs after a mismatch, forcing the algorithm to compare the pattern again from its start after each shift.

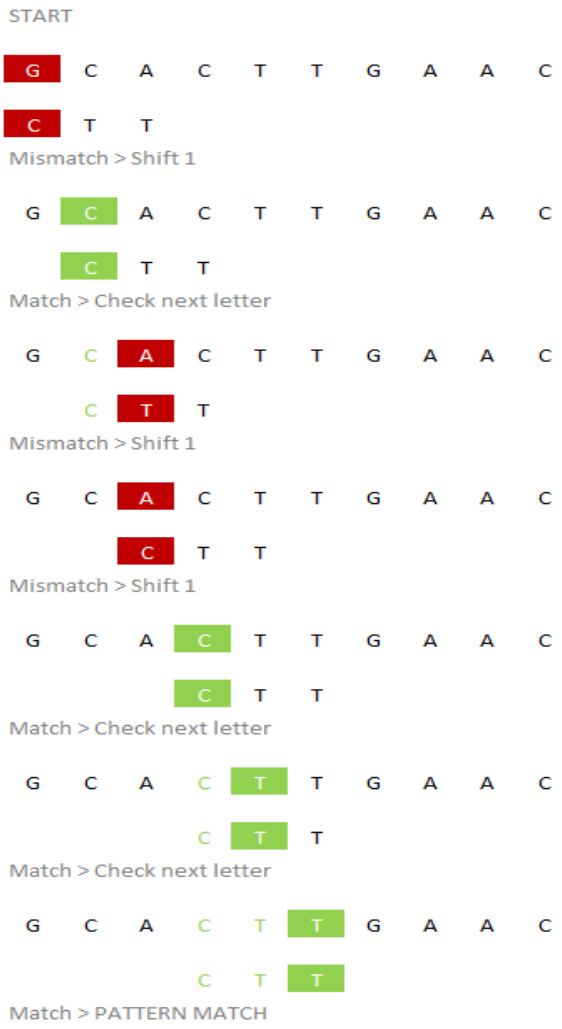

Figure 1: A Brute-Force execution example

2.3.2 Implementation Details

The Java implementation for BF does not include any notable changes in comparison with its C counterpart. The constructor's only responsibility is to store the pattern as an instance variable for multiple future searches, while the search method implements the searching logic by utilizing two loops. The outer that scans the text positions thus simulating the shifting window, and the inner that attempts to match the window to the pattern.

2.4 Deterministic Finite Automaton

2.4.1 Description

The Deterministic Finite Automaton (DFA) string search algorithm consists of two phases; a preprocessing phase, that includes building a deterministic finite automaton by preprocessing the pattern P, and a searching phase during which the text T is scanned with the constructed deterministic finite automaton [2].

A deterministic finite automaton is a finite state machine that outputs a unique run of the automaton for each input string, described by a set of transitions that connect each state and each symbol of the string, and expressed as a directed graph. Each state can be represented as a node and expresses a match of the symbol sequence up to this state. A transition can be represented as an edge and is assigned to a symbol, expressing the jump to the state representing the maximum prefix of the pattern that has been matched up to this symbol [4].

When building the automaton the number of states is (m + 1) consisting of the values [0 .. m]; the initial state is 0 and the final state is m. The automaton can be represented as a 2D array with columns the characters of the alphabet and rows the states the elements, with each cell denoting the next state transition resulting by the current character and the current state. The time complexity to construct the automaton is O(mσ).

An example of an automaton array for the pattern P = ‘abacaba’ and for alphabet Σ = {a, b, c}, and its state diagram are illustrated in Figure 2 and Figure 3 respectively.

	States			
Chars		a	b	c
	0	1	0	0
	1	1	2	0
	2	3	0	0
	3	1	2	4
	4	5	0	0
	5	1	6	0
	6	7	0	0
	7	1	2	0

Figure 2: Automaton array

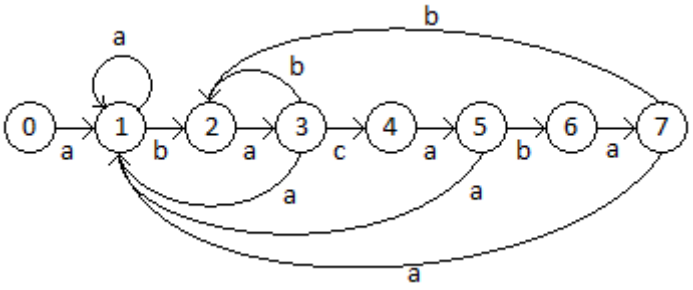

Figure 3: State diagram

In the automaton array, the entry for character ‘a’ and current state 0 contains the next state (1), which indicates that we have successfully matched the first letter of the pattern, and so on for every other entry.

Regarding the state diagram, there is an edge for each array entry except for the zero entries, which denote edges that point to the initial state (0) and are omitted for simplicity reasons in the state diagram.

Based on the above one can notice that, for example, state 3 indicates a match of the pattern prefix ‘aba’ up until the current scanned character in text T. In case the next character in T is ‘c’ then there is a match and the algorithm proceeds to the next state (4) and to the next character in T. If the next character is ‘a’ then the algorithm returns to state 1, meaning the maximum prefix of P that has been matched in T is ‘a’, and so we can move to the next character in T without having to back up in P. Similarly, if the next character is ‘b’ then we jump to state 2 as the maximum prefix of the pattern that has been matched is ‘ab’.

The searching phase begins from the initial state 0 and the first character of T. The characters of T are scanned sequentially only once and the next state is found, given the previous state and the currently scanned character. If the final state is reached, then there is a match. The time complexity for searching is O(n).

2.4.2 Implementation Details

To implement DFA in Java, the Automaton class was created to simulate the automaton data structure. The 2D transition array mentioned in the description is implemented with a one-dimensional array (target) and the appropriate mapping to access

its entries, implemented in the *setTarget* and *getTarget* methods. Also there are getter and setter methods for the boolean terminal array that denotes whether a state is terminal or not.

The constructor includes the initialization of the automaton as well as the preprocessing phase where the automaton is built. The search method scans T, passed as an argument, using the automaton for an occurrence of P.

2.5 Karp-Rabin

2.5.1 Description

The Karp-Rabin (KR) string matching algorithm has a similar premise to the BF algorithm, however it avoids the latter's quadratic number of character comparisons by first checking for each text position if the contents of the window resemble the contents of the pattern, and then comparing the characters one by one to confirm a match [5].

To perform this resemblance check it uses a hashing function. The hashing function for a string s of length m is calculated as follows [2]:

$$hash(s) = (s[0] \cdot 2^{m-1} + s[1] \cdot 2^{m-2} + ... + s[m-1] \cdot 2^0) MOD q$$

In this calculation, q is a very big integer.

The algorithm also includes a rehash function that efficiently 'rolls' along with the shifting window and avoids recalculating the whole new window when it shifts by one character. This is achieved by removing the hash value of the first character of the previous window and adding the hash value of the last character of the new window. The rehash function for a hash value 'h' of a string s of length 'm' with a previous first letter 'a' and new last letter 'b' is calculated as follows:

$$rehash(a, b, h) = ((h - a \cdot 2^{m-1})2 + b) MOD q$$

During the preprocessing phase, the hash value of the pattern is calculated once. The time complexity is O(m) and no extra space is needed.

2.5.2 Implementation Details

Regarding the implementation of the algorithm in Java there are three notable points to be made.

The hash value of the pattern is calculated at the constructor, while the hash value of the first m characters of the text is in the search function, since there is the point where the text is passed as a parameter.

The C function *memcmp* (Figure 4) was implemented as a static boolean method which returns true if the sequences of two specified character arrays starting from specified starting points and spanning a specified length are equal.

```
// Returns true if the sequential characters of two arrays, starting from
// specified indices to a common specified length, are equal; false otherwise.
private boolean memcmp(char[] x, int startX, char[] y, int startY, int length) {
    if (x == null)
        throw new IllegalArgumentException("first array is null");
    if (y == null)
        throw new IllegalArgumentException("second array is null");
    if (startX < 0 || startX >= x.length)
        throw new IllegalArgumentException(
                "start index of first array: " + startX + "should be between 0 and " + (x.length - 1));
    if (startY < 0 || startY >= y.length)
        throw new IllegalArgumentException(
                "start index of second array: " + startY + "should be between 0 and " + (y.length - 1));
    if (startX + length > x.length || startY + length > y.length)
        return false;
    for (int i = 0; i < length; ++i) {
        if (x[startX + i] != y[startY + i]) {
            return false;
        }
    }
    return true;
}
```

Figure 4: The *memcmp* Java implementation

A check was added that prevents an *OutOfBoundsException*. In the C language, all character arrays include a termination character '\0' at the last position, something that the C implementation takes into consideration. Contrary, in Java this is not the case, resulting in the insertion of this check in the search method to avoid accessing a position that is out of bounds (Figure 5).

```java
public int search(String txt) {
    if (txt == null)
        throw new IllegalArgumentException("text is null");
    if (txt.equals(""))
        throw new IllegalArgumentException("text is empty");

    final char[] text = txt.toCharArray();
    final int m = pattern.length;
    final int n = text.length;
    /* compute the hash code of the first m text characters */
    int txtHash = 0;
    for (int i = 0; i < m; ++i) {
        txtHash = (txtHash << 1) + text[i];
    }
    /* searching */
    for (int i = 0; i <= n - m; ++i) {
        if (patHash == txtHash) {
            if (memcmp(pattern, 0, text, i, m)) {
                return i;
            }
        }
        if (i == n - m) break; // prevents OutOfBoundsException
        txtHash = rehash(text[i], text[i + m], txtHash);
    }
    return n;
}
```

Figure 5: The Karp Rabin search method in Java with the added check that prevents OutOfBoundsException

2.6 Shift Or

2.6.1 Description

The Shift-Or (SO) string search algorithm utilizes bitwise operations to calculate and depict the search state as a number at each step. Its search phase is independent of the alphabet and pattern size, and for patterns smaller or equal to the word size of the machine doesn't require text buffering [2]. Like many known string search algorithms, it scans the positions of the text sequentially from left to right. Unlike most, it avoids the costly character comparisons by checking whether the state number indicates a match [6].

Specifically, this algorithm maintains the state of each position *j* of the text T in a bit vector *state* of length *m*. Each position *i* of the vector holds the value 0 if there is a match between prefix P[0..i] and T[j–i..j] (Figure 6). When the *m*-th bit is equal to 0 in text position *j*, a match is reported at position $j - m + 1$. In the example illustrated at Figure 7, we can see that $state_7[2] = 0$ and $state_{13}[2] = 0$ indicates matches at positions $7 - 3 + 1 = 5$ and $13 - 3 + 1 = 11$ respectively. In order to calculate the transition between

search states an auxiliary array *S* of size σ is needed, that stores the positions of each character of the alphabet in the pattern as a bit vector of length *m*.

Assuming the pattern is not greater than the word size of the machine, each bit vector of the *state* and *S* arrays can be calculated at constant time. Thus the preprocessing step including the construction of the *S* array takes time proportional to O(m + σ), and the searching phase consisting of transitioning between search states takes time proportional to O(n).

$$state_j[i] = \begin{cases} 0, & P[0..i] == T[j-i..j] \\ 1, & otherwise \end{cases}$$

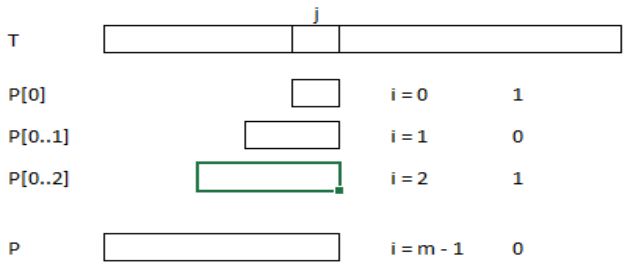

Figure 6: How the search state is expressed as a bit vector

			T												
			0	1	2	3	4	5	6	7	8	9	10	11	12	13
			G	A	G	C	T	G	A	T	C	C	G	G	A	T
	0	G	0	1	1	1	1	0	1	1	1	1	0	0	1	1
P	1	A	1	0	1	1	1	1	0	1	1	1	1	1	0	1
	2	T	1	1	1	1	1	1	1	0	1	1	1	1	1	0

Figure 7: A search state array example.

2.6.2 Implementation Details

Regarding the implementation, in comparison with the C counterpart, there are 2 notable differences.

The word size has been declared as a static variable equal to 31, as after extensive testing the algorithm as is produces correct results for pattern size, up to and including, 31.

In case of a pattern greater than the word size, the algorithm is directed to the use of a private method *searchLarge* (Figure 8). According to this method, when the word sized prefix of the pattern has been matched, then the rest *m* – *WORD_SIZE* letters are checked sequentially [1].

```
111    private int searchLarge(String txt) {
112        if (txt == null)
113            throw new IllegalArgumentException("text is null");
114        if (txt.equals(""))
115            throw new IllegalArgumentException("text is empty");
116
117        final char[] text = txt.toCharArray();
118        int n = text.length;
119        int state = ~0;
120        for (int i = 0; i < n; ++i) {
121            state = (state << 1) | S[text[i]];
122            if (state < lim) {
123                int k = 0;
124                int h = i - WORD_SIZE + 1;
125                while (k < m && h + k >= 0 && pattern[k] == text[h + k])
126                    ++k;
127                if (k == m)
128                    return i - WORD_SIZE + 1;
129            }
130        }
131        return n;
132    }
```

Figure 8: The *searchLarge* method

2.7 Morris-Pratt

2.7.1 Description

The Morris-Pratt (MP) algorithm is based on the same simple principle of step by step comparison that the BF algorithm relies on. However, it aims to exploit information gathered during the text scan, which is wasted in the case of the latter, by improving the length of the window shifts [2][7].

In the following example (Figure 9) of BF, we can see that after the mismatch the window shifts by only one character causing the algorithm to backtrack for the comparison attempt.

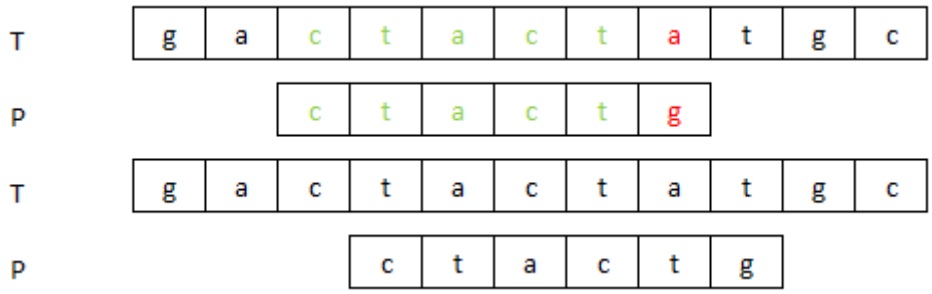

Figure 9: Example of Brute-Force backtracking after mismatch

The MP algorithm speeds up the searching process by increasing the size of the shifts whenever possible. Let's consider a common mismatch case. A prefix *u* of P has already been matched until a mismatch occurs between character *a* of T and character *b* P. Then we can find the longest suffix *v* of *u* that is also a prefix of *u* and shift the search window to the start of the suffix *v* (Figure 10). A portion like v is called a border, since it occurs at both ends of the text portion. The result of applying this logic to the example in Figure 9 is in Figure 11.

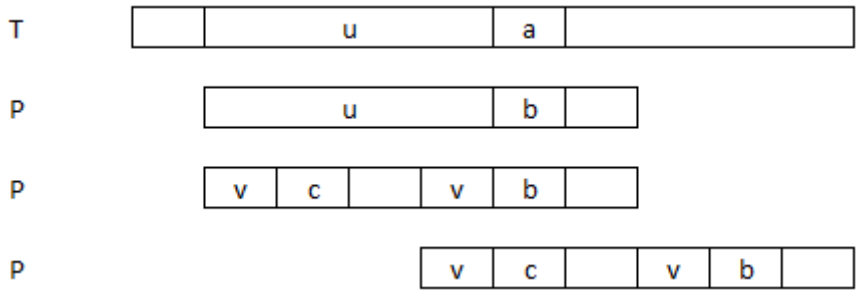

Figure 10: Window shifting in Morris-Pratt

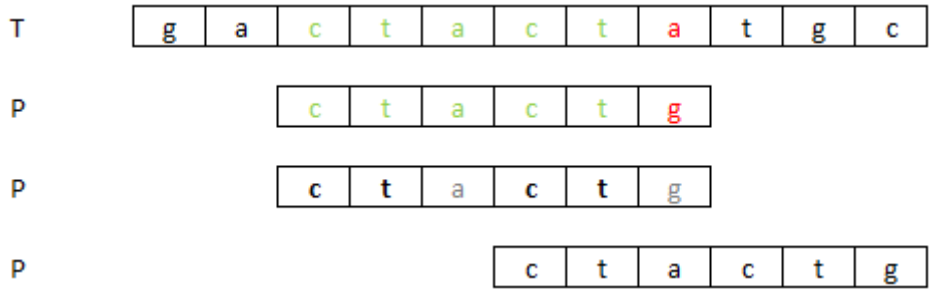

Figure 11: Window shifting in Morris-Pratt example

The process of finding *v* can be done by preprocessing the pattern. Specifically, we can maintain an array *mpNext* of size *m*, where for each position *i: 0 < i < m* we store the greatest *j: P[0..j - 1]* is the greatest suffix of *P[0..i - 1]* (or, put more simply, *j* is the maximal sequence of characters that is both a prefix and a suffix in *P[0..i - 1]*). *mpNext[0]* is initialized to 0.

This algorithm's preprocessing phase has an O(m) space and time complexity due to calculating *mpNext*, while the searching phase has an O(n + m) time complexity that is independent of alphabet size.

2.7.2 Implementation Details

There is one notable difference between the Java implementation and its C counterpart. The length of the array *mpNext* has been set to *m + 1* instead of *m* to avoid an *OutOfBoundsException*, something that is not necessary in the C implementation due to the termination character '\0' at the end of each string.

2.8 Knuth-Morris-Pratt

2.8.1 Description

The Knuth-Morris-Pratt (KMP) algorithm follows the same logic as Morris-Pratt. It offers a linear way of scanning the text just as the latter, aiming to further improve the length of the shifts [2].

Let's consider the case where the search window is placed at the T[j..j+m-1] factor of the text and the first mismatch happens between T[j+i] = a and P[i] = b, while the previous i characters (portion u) have already been matched (Figure 12).

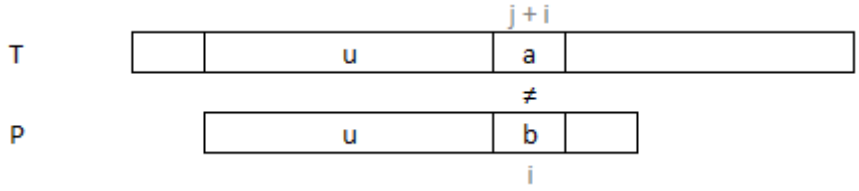

Figure 12: Mismatch case

Similarly, to Morris-Pratt, we can find the longest prefix v of u that is also a suffix of u (that is the border of u) and shift the window at the start of the suffix v. However, the length of the shift can be further improved by finding the border v that is succeeded by a character different from b, thus avoiding an immediate mismatch (Figure 13). Such a border is called a tagged border; that is a border that occurs at both ends of a text portion but succeeded by different characters.

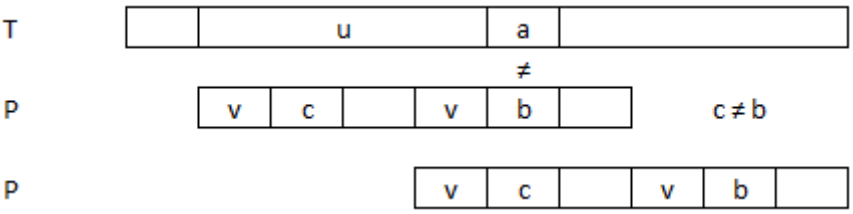

Figure 13: Knuth-Morris-Pratt window shift

The shifts can be calculated during preprocessing and stored in an array (namely kmpNext) of size m, taking time and space proportional to m. The searching phase consists of scanning the text sequentially from its first character and shifting appropriately in case of mismatch, and can be performed in time proportional to n + m.

2.8.2 Implementation Details

Implementation-wise, there are two differences between the Java and C programs.

The length of the array kmpNext has been increased from m to m + 1 to avoid an OutOfBoundsException resulting from the lack of '\0' character from the end of the strings in Java.

Additional boolean conditions have been added during the preprocessing of the algorithm, that also prevent OutOfBoundsExceptions (Figure 14).

```
void preKmp(char *x, int m, int kmpNext[]) {
   int i, j;

   i = 0;
   j = kmpNext[0] = -1;
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = kmpNext[j];
      i++;
      j++;
      if (x[i] == x[j])
         kmpNext[i] = kmpNext[j];
      else
         kmpNext[i] = j;
   }
}
```

```
private void preKmp() {
    final int m = pattern.length;
    int i, j;
    i = 0;
    j = -1;
    kmpNext[0] = -1;
    while (i < m) {
        while (j > -1 && pattern[i] != pattern[j]) {
            j = kmpNext[j];
        }
        i++;
        j++;
        if (i < m && j < m && pattern[i] == pattern[j]) {
            kmpNext[i] = kmpNext[j];
        } else {
            kmpNext[i] = j;
        }
    }
}
```

Figure 14: Additional check in KnuthMorrisPratt preprocessing method preKmp.

2.9 Simon

2.9.1 Description

The Simon (S) algorithm is a memory efficient implementation of Deterministic Finite Automaton (DFA) algorithm. Its principal relies on storing only a few significant edges of the automaton. Simon noticed that the significant edges can be grouped as follows [2]:

- Forward edges, that connect a prefix of *P* of length *k* to the prefix of length *k*+1; there are exactly *m* such edges.
- Backward edges, that connect a prefix of *P* of length *k* to a smaller non-zero prefix of *P*; there are at most *m* such edges.

The edges leading back to initial state, which are associated with a complete backtrack in the pattern after a mismatch, can be easily deduced and thus omitted.

Each state *i* indicates a match of a prefix of *P* of length *i*+1. This can be apparent from the example at Figure 15, where, indicatively, state 0 indicates a match of prefix 'g' of length 1, and so on. This means that state *i* can be labeled by *P[i]* (the last letter of the matched prefix of *P*). Thus, the forward edges can be easily deduced and need not to be stored. Finally, only the backward non-zero edges need to be stored, reducing the memory requirement from O(mσ) in DFA to O(m).

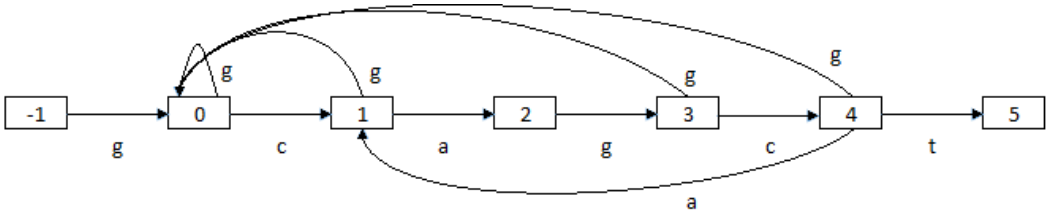

Figure 15: Simon algorithm state diagram example for the pattern 'gcagct'

To keep those edges, an array *L* of linked lists will be used. Since the initial state doesn't have backward edges, only m linked lists can be used. Each linked list will store the targets of their respective states' backward edges. Also, an integer variable *ell* is calculated such that *ell*+1 indicates the longest border of *P*. When searching for all occurrences of *P* in *T*, after a pattern match, the state will be updated with *ell*.

The preprocessing phase consists of calculating *L* and *ell* taking time and space proportional to O(*m*). The searching phase is like the DFA search and can be done in O(*m*+*σ*) time complexity.

2.10 Colussi

2.10.1 Description

The Colussi (C) [2][8] string search algorithm is an improvement of the Knuth-Morris-Pratt (KMP) algorithm. In preprocessing, it groups the pattern's positions into

two disjoint subsets: *noholes*, which are the positions for which *kmpNext[i]* is greater than -1, and *holes*, which are the remaining ones. In searching, each comparison attempt consists of two steps.

- First, the pattern positions of the *noholes* subset are scanned from left to right performing comparisons with the aligned text characters.
- Then, if there is no mismatch, comparisons are performed from right to left between the text characters aligned with the pattern positions of the *holes* subset.

This searching procedure has two advantages.

- If a mismatch occurs during the first step, the shift ensures that the text characters aligned with *nohole* pattern positions, which were compared during the previous attempt, will not be compared again (Figure 16).
- When a mismatch occurs during the second step it indicates that a suffix of the pattern matches a portion of the text. After the next shift it is ensured that the pattern prefix, which will still match a portion of the text, will not be compared again (Figure 17).

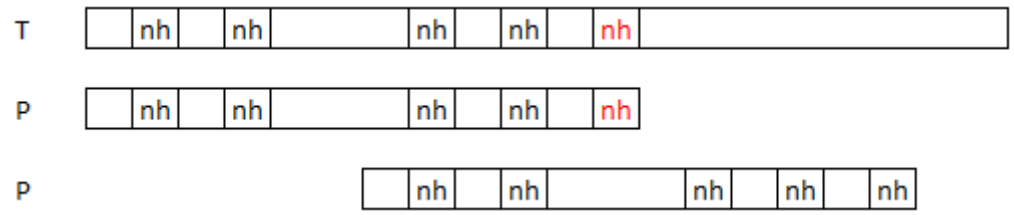

Figure 16: Shift after a mismatch with a *nohole* (nh). Comparisons are performed from left to right.

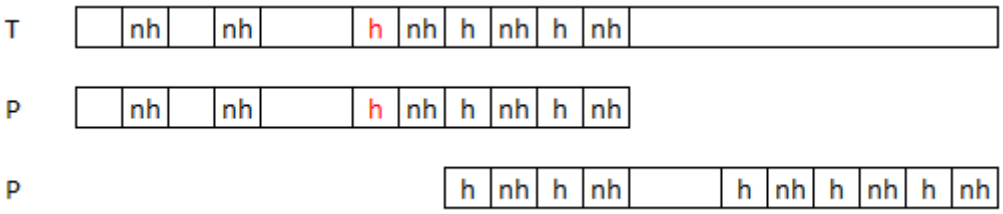

Figure 17: Shift after a mismatch with a *hole* (h). Comparisons are performed from right to left.

Colussi's preprocessing step takes time and space proportional to O(*m*)and its searching step takes time O(*n*), while it performs at most 1.5*n* text character comparisons.

2.10.2 Implementation Details

Porting Colussi algorithm from its C implementation to Java required some notable changes.

The sizes of the arrays *h, next, shift, hmax, kmin, nhd0, rmin* have been changed from *m* to *m+1* to avoid an *OutOfBoundsException.*

In the do..while loop in the computation of *hmax* some boolean conditions were added to also prevent an *OutOfBoundsException* (Figure 18).

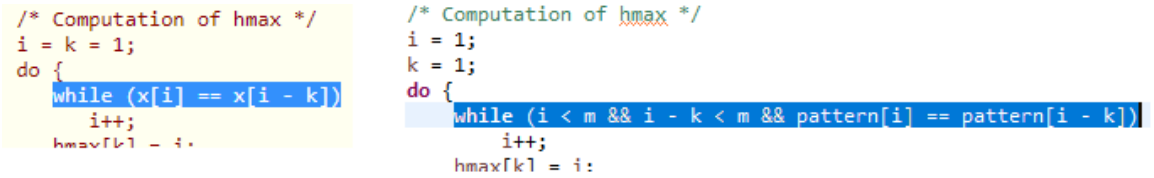

Figure 18: Comparison between C and Java implementations of Colussi

2.11 Galil-Giancarlo

2.11.1 Description

The Galil-Giancarlo algorithm is an improvement of the Colussi algorithm. It varies at the way the searching phase is performed for a pattern *P* that is not a power (not comprised) of a single character.

Specifically, let's assume that *l* is the greatest index in *P* such that *P[0..l]* is the longest prefix of the pattern that is comprised of the same character. Also, assume that in the previous comparison attempt, all the pattern *nohole* positions as well as a suffix of the pattern have been matched, indicating that, after the next shift, a prefix of the pattern will still match a factor of the text. So if the window is placed on *T[j..j+m-1]*, *P[0..last]* will match *T[j..j+last]*. After that in the next attempt, the characters starting from *T[last + 1]* will be scanned sequentially until the end is reached or a character $T[j+k] \neq P[0]$. In the last case there are two courses of action [2]:

- $P[l+1] \neq T[j+k]$ or only few of *P[0]* have been found($k \leq l$). In this case the window will be placed on *T[k+1..k+m]* and the search will continue similarly to Colussi starting from the first *nohole* position. At that point the memorized prefix of the pattern will be the empty string.

- *P[l+1] =T[j+k]* and enough of *P[0]* have been found (*k* > *l*). In this case the window will be placed on *T[k-l-1..k-l+m-2]* and the search will continue similarly to Colussi, staring from the second *nohole* position since *P[l+1]* is the first one. The memorized prefix will be *P[0..l+1]*.

Just like Colussi, this algorithm's preprocessing and searching phases take time proportional to O(*m*) and O(*n*) respectively, however the number of character comparisons is now reduced to 4n/3.

2.11.2 Implementation Details

Implementation-wise, there were some changes, regarding not only the correctness of the algorithm in the Java language but also its performance.

Since Galil-Giancarlo utilizes the same preprocessing method *preColussi* from the Colussi algorithm, the same changes were applied, regarding the change of all the size of the arrays in *preColussi* from *m* to *m+1*, to avoid an *OutOfBoundsException.*

Additionally, the call of the preprocessing method *preColussi*, was moved into the constructor of the class from its initial position inside the main body of the GG function from the C implementation. Also, three of the arrays constructed in *preColussi (h, shift, next)* are declared as instance variables as they are used in the search method. Those changes were carried out for performance reasons, since the three arrays as well as the integer value that *preColussi* returns are not changed throughout the searching phase and thus recalculating them at every search step would obviously impact negatively the execution time.

2.12 Apostolico-Crochemore

2.12.1 Description

The Apostolico-Crochemore [2][9] algorithm depends on the *kmpNext* table, from Knuth-Morris-Pratt, to perform the shifts, however utilizing a more sophisticated way of scanning the text. It introduces the variable *l* which is equal to the smallest pattern position of the different from *P[0]*, or 0 if the pattern is a power of a single character. At each attempt the comparisons are performed in the order: *l, l+1, l+2, … , m-2, m-1, 0, 1,*

l-1. In the searching phase, each step is described by a triad of indices (i, j, k) (Figure 19), where:

- *j* indicates that the window is placed on the text factor *T[j..j+m-1]*.
- *k* is responsible for comparisons between *P[0..k]* and *T[j..j+k]* for $0 \leq k < l$.
- *i* is responsible for comparisons between *P[l..i]* and *T[j..j+i]* for $l \leq i < m$

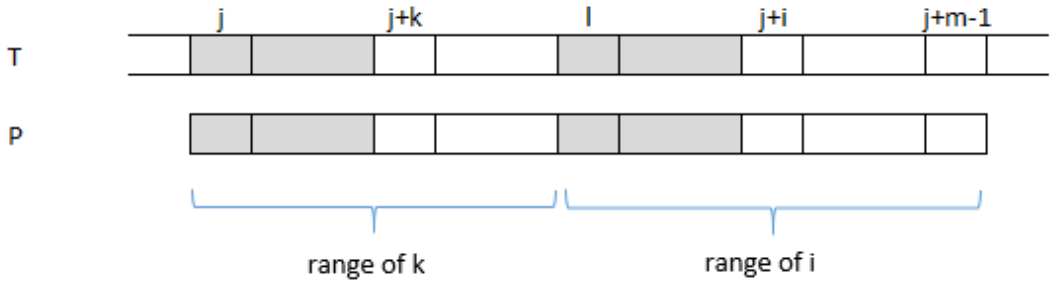

Figure 19: Scanning with the triad (i, j, k) in Apostolico-Crochemore

The transition to the next triad (*i*, *j*, *k*) is computed based on the value of *i*:

- $i = l$
  - if *P[i] = T[j+i]* then the next triad is (i+1, *j*, *k*)
  - else (*l*, *j*+1, max{0, *k*-1})
- $l < i < m$
  - if *P[i] = T[j+i]* then the next triad is (i+1, *j*, *k*)
  - else the outcome depends on the value of *kmpNext*
    - if *kmpNext[i]* $\leq l$ *then* (*l*, *j*+*i*-*kmpNext[i]*, *l*)
    - else (*kmpNext[i]*, *j*+*i*-*kmpNext[i]*, *l*)
- $i = m$
  - if $k < l$ and *P[k] = T[j+k]* then (*i*, *j*, *k*+1)
  - else if $k < l$ and *P[k]* $\neq$ *T[j+k]* then compute the triad as in the case where $l < i < m$
  - else report a match and compute the triad as in the case where $l < i < m$

The preprocessing phase consists of constructing *kmpNext* and computing *l* in time and space proportional to O(*m*). The searching phase can be done in O(*n*). The algorithm's number of total character comparisons is bounded by 1.5*n*.

2.12.2 Implementation Details

During the implementation of the algorithm in Java 2 changes were performed in comparison to its C counterpart, that prevent an *OutOfBoundsException*:

- The size of array *kmpNext* has been changed from *m* to *m+1*.
- A condition has been added in the computation of the *l* value (Figure 20)

```
/* Preprocessing */
preKmp(x, m, kmpNext);
for (ell = 1; x[ell - 1] == x[ell]; ell++);
if (ell == m)
   ell = 0;
```

```
/* Preprocessing */
preKmp();
for (ell = 1; ell < m && pattern[ell - 1] == pattern[ell]; ell++);
if (ell == m)
    ell = 0;
```

Figure 20: Apostolico-Crochemore different condition between C and Java implementation

2.13 Not So Naive

2.13.1 Description

The Not So Naïve algorithm is a variant of the Brute-force algorithm. It differentiates in the searching phase, where at each attempt the character comparisons are performed in the following order: 1, 2, .., m-2, m-1, 0. During preprocessing it calculates, in constant time, two shift values; *k* for after a mismatch and *ell* for after a pattern match. There are two distinct cases [2]:

- If $P[0] = P[1]$ then $k = 2$, $ell = 1$, meaning that if $P[1] \neq T[j+1]$ then the window shifts by 2, while if there is a pattern match then the window shifts by 1 (Figure 21).
- If $P[0] \neq P[1]$ then $k = 1$, $ell = 2$, meaning that if $P[1] \neq T[j+1]$ then the window shifts by 1, while if there is a pattern match then the window shifts by 2.

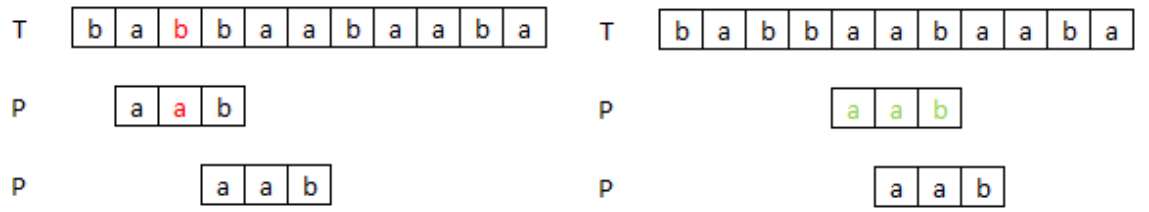

Figure 21: Not So Naive shifts after mismatch (left) and after pattern match (right) when P[0] = P[1].

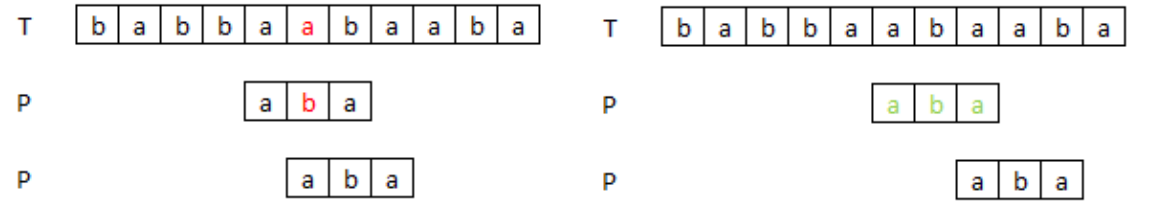

Figure 22: Not So Naive shifts after mismatch (left) and after pattern match (right) when P[0] ≠ P[1] (Figure 22).

Not So Naïve performs the preprocessing in constant time and its searching phase takes time similar to the Brute-Force's O(*mn*), however on average its performance is by coefficient sublinear.

2.13.2 Implementation Details

Regarding the algorithm's implementation in Java there were some noteworthy changes:

- The algorithm utilizes the static boolean method *memcmp* that was introduced in Karp-Rabin (Figure 4).
- The C implementation does not work correctly for patterns of length 1. This was resolved by reducing the searching for single character patterns to a simple Brute-Force search.
- A check was added in the constructor during the calculation of *k* and *ell*, that prevents an *OutOfBoundsException* for patterns of length 1 (Figure 23).

```
/* Preprocessing */
if (x[0] == x[1]) {
   k = 2;
   ell = 1;
}
else {
   k = 1;
   ell = 2;
}
```

```
/* Preprocessing */
if (m > 1 && pattern[0] == pattern[1]) {
    k = 2;
    ell = 1;
} else {
    k = 1;
    ell = 2;
}
this.ell = ell;
this.k = k;
```

Figure 23: Not So Naive added condition

2.14 Boyer-Moore

2.14.1 Description

Boyer-Moore is regarded the most performant string search algorithm in usual applications. Various simplifications of the algorithm are utilized in text editors to implement basic 'search' and 'substitute' functionality. Its principal relies on the idea that greater information is gained by performing the character comparisons of the pattern from right to left, beginning with the rightmost character [2][10]. This results to longer shifts and less character comparisons. The algorithm uses two shift tables for cases of mismatch or cases of pattern match.

- The good-suffix shift (or matching shift) table *bmGs*. Let's assume the window is placed on text factor *T[j..j+m-1]* and the first mismatch happens between *P[i]=a* and *T[j+i]=b*, meaning *P[i+1..m-1]=T[j+i+1..j+m-1]*. Then the shift table aligns the factor *T[j+i+1..j+m-1]* with its rightmost occurrence in *P* preceded by a character different from *a* (Figure 24). If there is no such occurrence then a suffix of the text factor is aligned with a prefix of the pattern (Figure 25).

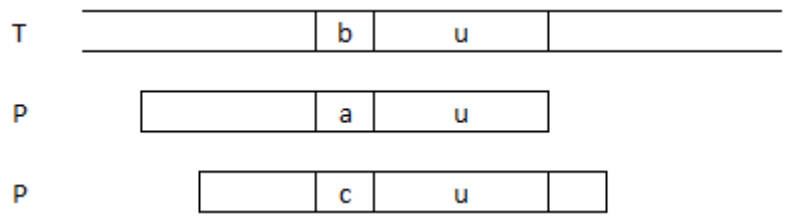

Figure 24: Good-suffix shift, portion u reoccurs in pattern with a different preceding character.

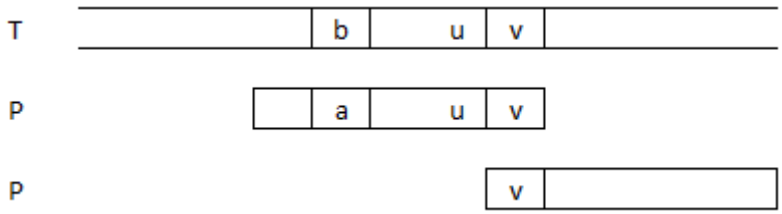

Figure 25: Good-suffix shift, portion u does not reoccur in pattern with a different preceding character.

- The bad-character (or mismatch shift) table *bmBc*. Let's assume the window is placed on text factor *T[j..j+m-1]* and the first mismatch happens between *P[i]=a* and *T[j+i]=b*, meaning *P[i+1..m-1]=T[j+i+1..j+m-1]*. Then the shift table aligns *b* with its rightmost occurrence in *P[0..m-2]* (Figure 26). If there is no such occurrence then the left end of the window is aligned with the character succeeding *b* (Figure 27).

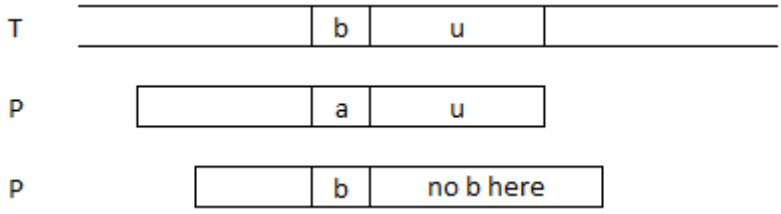

Figure 26: Bad-character shift, the window shifts to the rightmost occurrence in P[0..m-2].

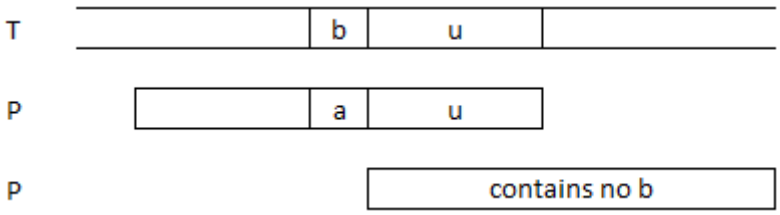

Figure 27: Bad-character shift, no other occurence of b.

Since the bad-character shift can be negative, after each mismatch the maximum between the good-suffix shift and the bad-character shift is taken.

The tables can be constructed during preprocessing taking time and space proportional to $O(m+\sigma)$. The searching phase maybe quadratic but the number of text

character comparisons is bounded by 3*n*.the algorithm is quite performant when dealing with large alphabets in comparison with the length of the pattern, and in searching patterns like $a^{m-1}b$.

2.15 Turbo Boyer-Moore

2.15.1 Description

Turbo Boyer-Moore is an improvement of Boyer-Moore. The main idea is to maintain in memory the text factor that matched a suffix of the pattern in the last comparison attempt, only after a good-suffix shift. Thus, the algorithm does not require extra preprocessing, just some constant extra space to save the text factor [2].

When such a factor is found, it can enable a regular shift or a turbo shift. A turbo shift may occur when the currently matched pattern's suffix *v* is shorter than the one from the previous attempt *u* (Figure 28). In this case let *uzv* be a suffix of the pattern and *a* and *b* be the characters that cause the current mismatch in the pattern and text respectively. This means that *av* is a suffix of both *P* and *u*. Let *p* be the distance between *a, b* in the text, then the *uzv* suffix of the pattern has a period of length *p*=|*zv*|, since *u* is a border of *uzv*, thus it cannot cover both occurrences of *a,b* at distance *p* in the text. The shortest possible shift is |*u*|-|*v*| is a turbo shift. If in the same case the bad-character shift is greater even than turbo-shift, the actual shift must be greater or equal than |*u*|+1.In the example in Figure 29 we assume that a good-suffix shift occurred therefore characters *c*, *d* are different. Given a shift greater than the turbo-shift and smaller than |*u*|+1 would align *c* and *d* with the same character in *v*. Thus here the shift length should be |*u*|+1.

The time and space complexity during preprocessing and searching is O($m+\sigma$) and O(n) respectively, but the number of text character comparisons bound is reduced to 2*n*.

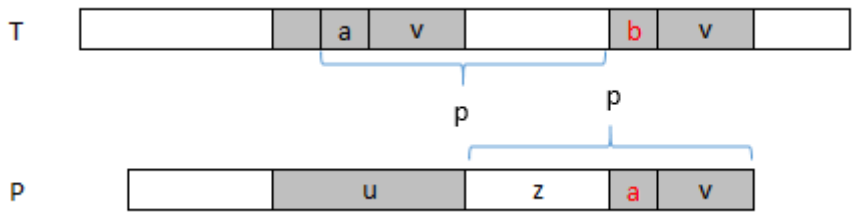

Figure 28: A turbo-shift can be enabled only when |v| < |u|

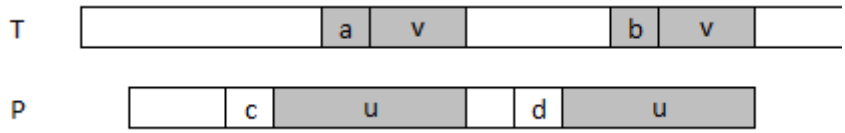

Figure 29: c is different from d so it won't be aligned with the same character in v

2.16 Apostolico-Giancarlo

2.16.1 Description

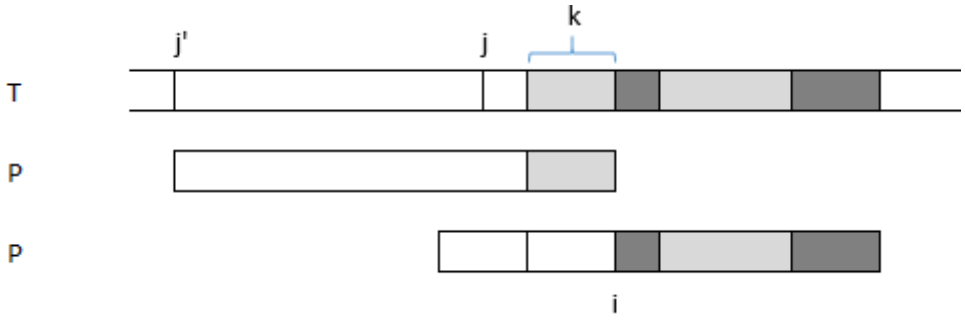

Figure 30: Typical situation in AG. Dark grey areas are factors that have been compared in the current attempt, and light grey ones are factors that have been skipped [2].

The Apostolico-Giancarlo [11] algorithm is a variant of the Boyer-Moore that is designed to store information about the suffix matches from previous attempts. It keeps the length of the longest suffix of the pattern that has been matched after each attempt, in a *skip* array. In a typical situation of Apostolico-Giancarlo (Figure 30), during an attempt for a window placed on $T[j'..j'+m-1]$, there is a match of a suffix of the pattern of length k ending at $i+j$, meaning $P[m-k..m-1] = T[j'+m-k..j'+m-1]$. Then, in a later attempt at position $j > j'$ a match occurs between $P[j-j'+m..m-1]$ and $T[j'+m-k..j+m-1]$. As seen in the Boyer-Moore chapter, the $suff[i]$ for $0 \leq i < m$ holds the length of the longest pattern suffix ending in i. Thus, after each matching of the text factor $T[j+i+1..j+m-1]$, there are four possible outcomes:

- $k > suff[i]$ *and* $suff[i] = i+1$: Meaning an occurrence of the pattern has been located at j. Then we set $skip[j+m-1]=m$ and the window is shifted by $bmGs[0]$ (the good-suffix shift array from Boyer-Moore).

- *k > suff[i] and suff[i]≤ i:* Meaning *P[i-suff[i]] ≠ T[j+i-suff[i]]*. We set *skip[j+m-1] = m – 1 – I + suff[i]* and shift using *bmBc[T[i+j-suff[i]]]* and *bmGs[i-suff[i]+1]*.
- *k < suff[i]:* Meaning P*[i-k] ≠ T[i+j-k]*. Then we set skip*[j+m-1] = m-1-i+k* and shift using bmBc*[T[i+j-k]] and* bmGs*[i-k+1]*.
- *k = suff[i]:* In this occasion, the text factor *T[j+i-k+1..j+i]* needs to be skipped, so the comparisons may continue for characters *T[j+i-k]* and *P[i-k]*.

Since only the latest m suffix lengths are needed each time, the size of the *skip* array can be reduced to *m.* The preprocessing phase's complexity is O(*m+σ*), similar to Boyer-Moore's and the number of text character comparisons is bounded by 1.5*n.*

2.16.2 Implementation Details

During porting the algorithm from C to Java, two static methods were implemented, equivalents for C's *memcpy* (Figure 31) and *memset* (Figure 32), regarding only the data types needed by the algorithm.

```
// similar to C's memcpy; copies a segment from one integer array to another
private static void memcpy(int[] dest, int d_i, int[] src, int s_i, int l) {
    if (dest == null)
        throw new IllegalArgumentException("destination array is null");
    if (src == null)
        throw new IllegalArgumentException("source array is null");
    if (l < 0) throw new IllegalArgumentException("length shoud be nonnegative");
    if (d_i < 0 || d_i >= dest.length)
        throw new IllegalArgumentException("destination start: " + d_i + " should be between 0 and " + (dest.length - 1));
    if (s_i < 0 || s_i > src.length)
        throw new IllegalArgumentException("source start: " + s_i + " should be between 0 and " + (src.length - 1));
    if (d_i + l > dest.length)
        throw new IllegalArgumentException("destination start: " + d_i + " + length: " + l + " is out of bounds");
    if (s_i + l > src.length)
        throw new IllegalArgumentException("source start: " + s_i + " + length: " + l + " is out of bounds");
    int[] temp = new int[l];
    for (int i = 0; i < l; ++i) {
        temp[i] = src[s_i + i];
    }

    for (int i = 0; i < l; ++i) {
        dest[d_i + i] = temp[i];
    }
}
```

Figure 31: Implementation of memcpy.

```
// similar to C's memset; sets given integer value to positions of an array
// dest starting from start and for length elements
private static void memset(int[] dest, int start, int length, int val) {
    if (dest == null)
        throw new IllegalArgumentException("destination array is null");
    if (start < 0 || start >= dest.length)
        throw new IllegalArgumentException("start: " + start + " should be between 0 and " + (dest.length - 1));
    if (start + length > dest.length)
        throw new IllegalArgumentException("start: " + start + " + length: " + length + " is out of bounds");
    for (int i = 0; i < length; ++i) {
        dest[start + i] = val;
    }
}
```

Figure 32: Implementation of memset.

2.17 Reverse Colussi

2.17.1 Description

Reverse Colussi [2][12] is an amelioration of the Boyer-Moore algorithm. It performs the comparisons in a specific order indicated by a preprocessed array *h*. Specifically, for each *i* where $0 \le i \le m$, two disjoint sets are defined:

- $Pos(i)=\{k: 0 \le k \le i \text{ and } P[i] = P[i-k]\}$
- $Neg(i)=\{k: 0 \le k \le i \text{ and } P[i] \neq P[i-k]\}$

Three auxiliary arrays are defined:

- *hmin[k]*, for $1 \le k \le m$, is the minimum integer l: $l \ge k-1$ and *Neg(i)* not containing k for $l \le i \le m-1$.
- *kmin[l]*, for $0 \le l \le m-1$, is the minimum integer k: $hmin[k]=l \ge k$ if such k exists; 0 otherwise.
- *Rmin[l]*, for $0 \le l \le m-1$, is the minimum integer k: $r > l$ and $hmin[r]=r$–1.

After *h[0]* is initialized to *m-1*, we increasingly select all indexes *h[0]*, .. , *h[d]* in increasing order of *kmin[l]* such that $kmin[h[i]] \neq 0$ and set $rcGs[i] = kmin[h[i]]$ for all $1 \le i \le d$. Then the indexes *h[d+1]*, .. , *h[m-1]* are increasingly selected setting $rcGs[i] = rmin[h[i]]$ for $d < I < m$. Finally, we set *rcGs[m]* to the period of the pattern.

The array *rcGs* is defined as: $rcGs[a, s] = min \{k: (k=m \text{ or } P[m-k-1] = a) \text{ and } (k > m–s-1 \text{ or } P[m-k-s-1]=P[m-s-1])\}$. To compute *rcGc* we define array *locc* such that for each alphabet character *c*, *locc[c]* is the position of its rightmost occurrence in *P[0..m-2]*; -1 if there is none. An array *link* indicates downward all the positions of the pattern characters.

Reverse Colussi's preprocessing phase takes $O(m^2)$ time and $O(m\sigma)$ space, while its searching phase is linear.

2.17.2 Implementation Details

Regarding the implementation of the algorithm in Java from C, to remedy the issue of the absence of the termination character '\0' and avoid an *OutOfBoundsException* the lengths of the auxiliary arrays have been altered. Specifically the size of the arrays *h, rcGs, hmin, kmin, link, locc, rmin* has been changed to *m+1* from *m*, and the array *rcBc*'s number of columns has been extended to *m+1* from *m*.

2.18 Horspool

2.18.1 Description

The Horspool algorithm [2][13] is a simplification of Boyer-Moore. It utilizes only the bad-character shift *bmBc* from the latter, while keeping the same right-to-left comparison order during attempts. This results in a quite simple to implement and efficient algorithm for large alphabets compared to the length of the pattern, like the ASCII table. The preprocessing can be done in O($m + \sigma$) time and O(σ) space. The searching phase may be quadratic at worst case, however the average number of text character comparisons is between $1/\sigma$ and $2/(\sigma + 1)$. A searching example is illustrated at Figure 33. Here the algorithm performs 4 + 2 + 1 + 1 + 6 = 14 character comparisons in total.

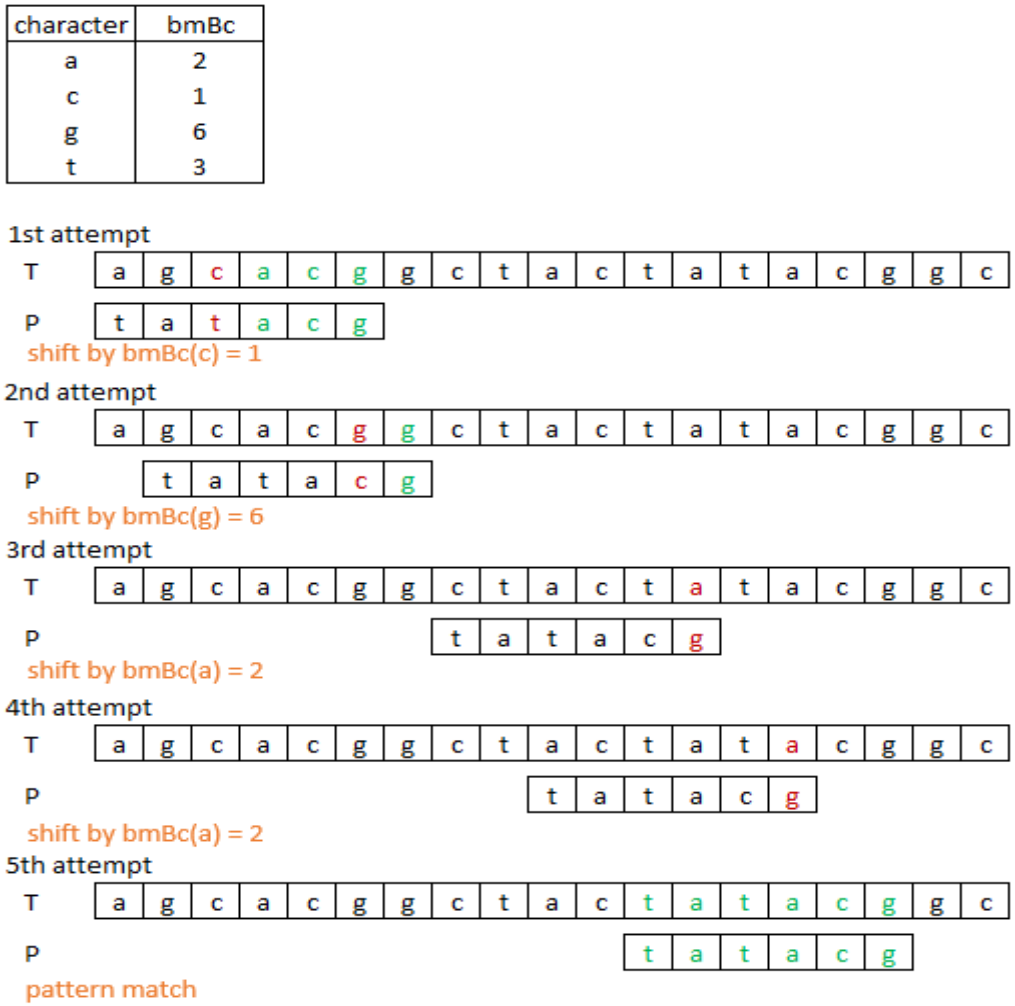

Figure 33: Horspool searching example.

2.18.2 Implementation Details

The Java implementation of the Horspool algorithm utilizes the *memcmp* static Boolean method that was introduced in the *Karp-Rabin* section (Figure 4).

2.19 Quick Search

2.19.1 Description

The Quick Search algorithm [2][14] is yet another simplification of Boyer-Moore that is quite performant regarding large alphabets and smaller patterns. Just as Horspool, it uses only the bad-character shift, but it allows the character comparisons to be performed at any order. Assuming the window is placed on text factor *T[j..j+m-1]* and since the length of a shift is always positive, one may notice that the character *T[j+m]* will be involved in the following attempt. This can be exploited by modifying the bad-character shift table *bmBc* of size σ, to take into account the last pattern character. Specifically, the shift table is calculated as follows: for each c in σ, $bmBc[c] = m - i$, *such that* $i = max\{pos: 0 \leq pos < m: P[pos] = c\}$; $m+1$ otherwise. Now, after a character mismatch, the shift will depend on the text character right after the rightmost end of the window. Compared to Horspool, Quick Search may produce shorter shifts however it compensates by performing fewer character comparisons. The preprocessing and searching phases have the same time and space complexities as their respective ones in Horspool. The example presented in the Horspool section (Figure 33) is revisited in Figure 34.The number of text character comparisons performed here are 1 + 1 +1 + 1 + 1 + 6 = 11, fewer that those performed in Horspool (14).

Figure 34: Quick Search searching example.

2.19.2 Implementation Details

The Java implementation of Quick Search utilizes the static boolean method *memcmp* introduced in the Karp-Rabin section, which returns true if two specified sequences of characters are identical. During the implementation, an additional condition (Figure 35) was added in the searching phase of the algorithm, which exits the scanning

loop after the attempt at position *j* = *n-m*. This is essential to avoid an out of bounds index, as the next shift is performed for the character at position *j* + *m*, which regarding the above position is out of bounds.

```
/* Searching */
j = 0;
while (j <= n - m) {
   if (memcmp(x, y + j, m) == 0)
      OUTPUT(j);
   j += qsBc[y[j + m]];
}
```

```
/* Searching */
for (int i = 0; i <= n - m; i += qsBc[text[i + m]]) {
    if (memcmp(pattern, 0, text, i, m))
        return i;
    if (i == n - m) break;
}
```

Figure 35: Added condition in Quick Search that prevents an OutOfBoundsException.

2.20 Tuned Boyer-Moore

2.20.1 Description

Tuned Boyer-Moore [2][15] is another performant, simplified version of Boyer-Moore. It attempts to reduce the most expensive part of a string search algorithm, the character comparisons, by doing several consecutive shifts at once using only the bad-character shift table *bmBc*. The algorithm focuses on trying to quickly find a match for the last pattern character *P[m-1]* by performing three shifts at once. To achieve this, the text is padded at its end with *m* occurrences of *P[m-1]* character. Additionaly, the actual shift of the last character, *bmBc[m-1]*, is stored separately in a variable *shift*, and then set to zero. This ensures that, in case one of the shifts matches the last character of the pattern, the following shifts will not skip it. When the last character is matched after the triad of shifts, an attempt follows in any order. In case of a mismatch the window shifts by *bmBc[c]* for each In case of a pattern match the window shifts by *shift*. Just like the previous variants of Boyer-Moore, this algorithm has a quadratic worst case time complexity but is quite efficient in practical terms. A searching example is illustrated in Figure 36.

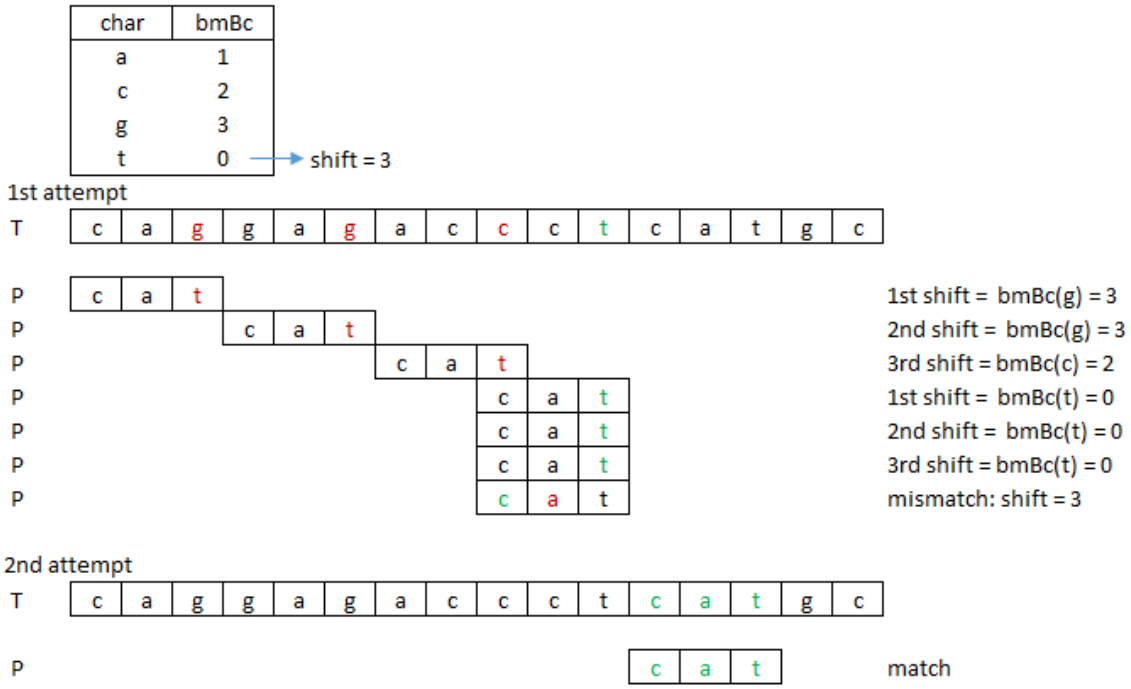

Figure 36: Tuned Boyer-Moore searching example.

2.21 Zhu-Takaoka

2.21.1 Description

The Zhu-Takaoka [2][16] is an implementation of a variant of the Boyer-Moore algorithm. It utilizes the good-suffix shift table along with a differentiated bad-character shift table *ztBc*, which contains the shift for two consecutive characters.

The algorithm performs the attempt comparisons from right to left. Assuming the window is positioned on text factor $T[j..j+m-1]$ and a mismatch occurs between $P[m-k]$ and $T[j+m-k]$ (meaning $P[m-k+1..m-1] = T[j+m-k+1..j+m-1]$) the shift is based on the two consecutive characters $T[j+m-2]$, $T[j+m-1]$. Zhu-Takaoka's bad character shift table is represented by a two-dimensional array that computes the rightmost occurrence of *ab* in $P[0..m-2]$, for each pair *a*,*b* from σ. Then $ztBc[a][b] = k$ iff:

- $k < m-2$ and $P[m-k..m-k+1] = ab$ and *ab* does not exist in $P[m-k+2..m-2]$

or

- $k = m-1$ and $P[0] = b$ and *ab* does not occur in $P[0..m-2]$
- $k = m$ and $P[0] \neq b$ and *ab* does not occur in $P[0..m-2]$

The algorithm performs the preprocessing phase in $O(m+\sigma^2)$ time and space complexity, and the searching phase in quadratic time complexity in the worst case.

2.21.2 Implementation Details

The Java implementation of Zhu-Takaoka has a major difference with respect to its C counterpart. During the searching phase, in the case of a mismatch when the next shift as calculated as the maximum between the good-suffix shift and the bad-character shift of the two consecutive characters *P[j+m-2]*, *P[j+m-1]*, the *if..else* has been modified as shown in Figure 37 to avoid an out of bounds index for *j+m-2*.

```
/* Searching */
j = 0;
while (j <= n - m) {
   i = m - 1;
   while (i < m && x[i] == y[i + j])
      --i;
   if (i < 0) {
      OUTPUT(j);
      j += bmGs[0];
   }
   else
      j += MAX(bmGs[i],
               ztBc[y[j + m - 2]][y[j + m - 1]]);
}
```

```
/* Searching */
j = 0;
while (j <= n - m) {
    i = m - 1;
    while (i >= 0 && pattern[i] == text[i + j])
        --i;
    if (i < 0)
        return j;
    else if (j + m - 2 >= 0)
        j += Math.max(bmGs[i], ztBc[text[j + m - 2]][text[j + m - 1]]);
    else
        j += bmGs[i];
}
```

Figure 37: Zhu-Takaoka: differences in searching phase between C and Java implementations.

2.22 Berry-Ravindran

2.22.1 Description

The Berry-Ravindran algorithm [2][17] is a simplification of the Zhu-Takaoka algorithm, resembling the simplified variant Quick Search of the Boyer-Moore algorithm that uses only the bad-character shift table. Berry-Ravindran differentiates by using a modified bad-character shift table *brBc* for the two consecutive characters immediately after the right end of the window. During preprocessing, *brBc* is constructed by setting for each pair *ab*, for *a*, *b* in σ, *brBc[a][b]* to:

- 1, if $P[m-1] = a$
- $m - i - 1$, if $P[i]P[i+1] = ab$
- $m + 1$, if $P[0] = b$
- $m + 2$, otherwise

During the searching phase, assuming the window is placed on text factor *T[j..j+m-1]* a shift *brBc[T[j+m]][T[j+m+1]]* will be performed. The bad-character shift can be preprocessed in space and time proportional to $O(m+ \sigma^2)$ and the searching can be done in $O(mn)$ in worst case.

2.22.2 Implementation Details

As mentioned above, during the searching phase the algorithm scans the text sequentially from the leftmost character applying the shift of the two consecutive characters, following the rightmost edge of the window in case of mismatch. The C implementation, in order to prevent an out of bounds text index, adds at the end of the string the null character (‘\0’), just right after the similar termination character. In the Java implementation, only one null character is added at the end of the character array of the text string. Also, the *while* loop that scans the text is modified as shown in Figure 38.This change performs the scan up to text position *n-m*-1 and, right after its termination, it performs an attempt on *T[n-m..n-2] (n-1* is the added null character*).* The algorithm has a O($m+\sigma^2$) time and space complexity and a O(*mn*) searching complexity.

```
/* Searching */
y[n + 1] = '\0';
j = 0;
while (j <= n - m) {
   if (memcmp(x, y + j, m) == 0)
      OUTPUT(j);
   j += brBc[y[j + m]][y[j + m + 1]];
}
```

```
/* Searching */
j = 0;
while (j < n - m) {
    if (memcmp(pattern, 0, text, j, m))
        return j;
    j += brBc[text[j + m]][text[j + m + 1]];
}
if (j == n - m)
    if (memcmp(pattern, 0, text, j, m))
        return j;
```

Figure 38: Berry-Ravindran: difference in the scan loop during the searching phase.

2.23 Smith

2.23.1 Description

Smith algorithm [2][18] is a hybrid of Horspool and Quick Search, which combines the best features of each. As mentioned above, Horspool’s shift is based on the first character in the window where a mismatch occurs, while Quick Search’s shift is done according to the first text character after the rightmost end of the window. However, as mentioned in the Quick Search section, Quick Search’s skip lengths might be shorter than those of Horspool. Smith remedies that by taking the maximum of the two shifts. The preprocessing phase of the algorithm consists of computing the two shift tables in O($m+\sigma$) time and O(σ) space. The searching phase is quadratic but with a good practical behavior.

2.23.2 Implementation Details

Since the Smith algorithm uses the Quick Search bad-character shift table which is accessed for the character immediately after the end of he right end of the window, the scan loop of the algorithm must be careful not to access an index out of bounds. In the C implementation one can notice that the scan loop reaches up until the position *n-m*, at which it will access the Quick Search shift table for *T[n]* (Figure 39). Since the strings in C end with the termination character '\0', this won't be a problem, however that is not the case in Java. The java implementation (Figure 40) has been slightly modified to scan up to *n-m-1* and then, right after loop termination to perform an attempt at text factor *T[n-m..n-1]*. This ensures that no *OutOfBoundsException* will be thrown.

```
/* Searching */
j = 0;
while (j<= n - m) {
   if (memcmp(x, y + j, m) == 0)
      OUTPUT(j);
   j += MAX(bmBc[y[j + m - 1]], qsBc[y[j + m]]);
}
```

Figure 39: C implementation of Smith's search phase.

```
/* Searching */
i = 0;
while (i < n - m) {
    if (memcmp(pattern, 0, text, i, m))
        return i;
    i += Math.max(bmBc[text[i + m - 1]], qsBc[text[i + m]]);
}
if (i == n - m)
    if (memcmp(pattern, 0, text, i, m))
        return i;
```

Figure 40: Java implementation of Smith's searching phase.

2.24 Raita

2.24.1 Description

The Raita [2][19] is an implementation of a variant of the Boyer-Moore algorithm that is quite performant when searching in a non-random text like an English text. Its main idea relies on taking advantage of the dependencies, which can form between successive characters in existing words, to plan the order of comparison during attempts. Noticeably, there are the stronger dependencies between close, neighboring characters

and weaker ones at word boundaries. Essentially, the algorithm, after a successful character match, aims to compare the character the dependencies of which are the weakest with respect to the already matched characters. Practically, the algorithm avoids comparing the characters of the window sequentially from left to right or from right to left, but aims to compare the last character, the first character and the middle character of the window, with the respective characters of the pattern in that order, and then all the rest characters. For the shifts the Boyer-Moore bad-character shift table is used. The algorithm performs the searching phase in quadratic time but is very efficient in practice on English texts. A searching example is illustrated in Figure 41.

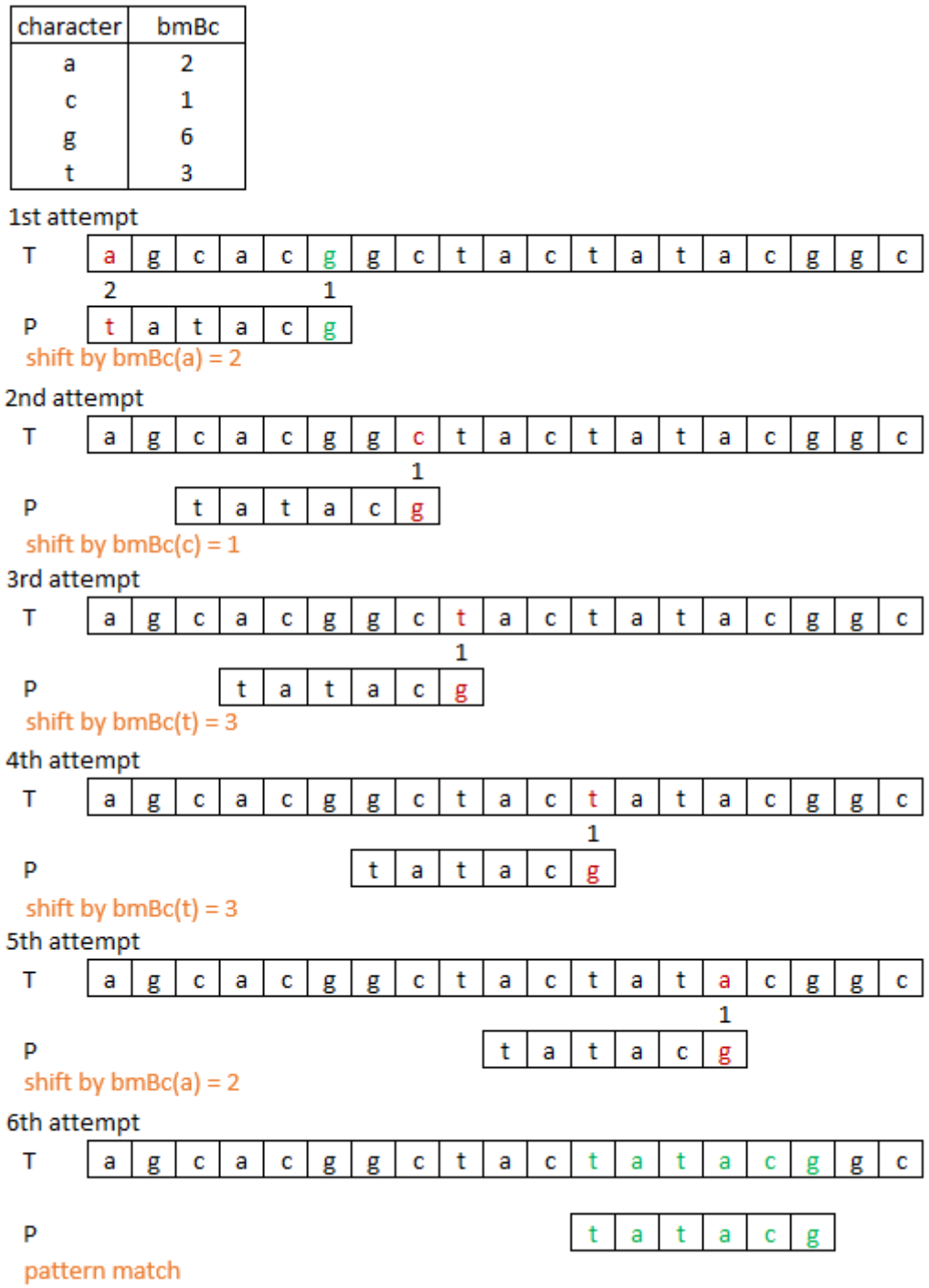

Figure 41: Raita searching example.

2.24.2 Implementation Details

Implementation-wise there was an important modification with respect to the program in C, regarding not the correctness of the algorithm but its efficiency as it was presented. The C implementation performs the character comparisons in the following order: last, middle, first, the rest (Figure 42). However, the order presented in [19] is the

following: last, first, middle, the rest; the latter is what has been implemented in Java (Figure 43).

```
/* Searching */
j = 0;
while (j <= n - m) {
   c = y[j + m - 1];
   if (lastCh == c && middleCh == y[j + m/2] &&
       firstCh == y[j] &&
       memcmp(secondCh, y + j + 1, m - 2) == 0)
      OUTPUT(j);
   j += bmBc[c];
}
```

Figure 42: Middle character is checked *before* first character in Raita's C implementation.

```
/* Searching */
i = 0;
while (i <= n - m) {
    char c = text[i + m - 1];
    if (lastChar == c && firstChar == text[i] && middleChar == text[i + m / 2]
            && memcmp(pattern, 1, text, i + 1, m - 2)) {
        return i;
    }
    i += bmBc[c];
}
```

Figure 43: Middle character is checked *after* first character in Raita's Java implementation.

2.25 Reverse Factor

2.25.1 Description

The Reverse Factor [2][20] is an implementation of a string search algorithm, efficient when dealing with long patterns and small alphabets. While Boyer-Moore and its variants attempt to match some suffixes of the pattern, this algorithm is designed to match prefixes of the pattern by scanning the window from right to left. This is achieved by constructing a smallest suffix automaton (or DAWG for Directed Acyclic Word Graph) for the reversed pattern.

During preprocessing, the smallest suffix automaton $S(P^R)$ for the reversed pattern is computed. That is a deterministic finite automaton, which recognizes all the suffixes of a string w and is denoted as $S(w) = \{\Sigma, S, s, F, \delta\}$, where:

- Σ is the alphabet
- S is the set of states
- s is the initial state
- F is the set of final states
- $\delta: S \times \Sigma = S$ is the transition function

Its main difference with the automaton presented in the Deterministic Finite Automaton algorithm is that there are no forward edges. An example is illustrated in Figure 44, where all the omitted edges are undefined. It can be constructed in time and space proportional to O(m).

During searching, the scanning of the window is performed by the smallest suffix automaton from right to left beginning from the initial state, proceeding as long as there is a defined transition for the current character in the current state. In case of a mismatch, the shift is easily computed to be the length of the longest prefix of the pattern that has been matched. This phase takes quadratic time in the worst case but it behaves optimally on average.

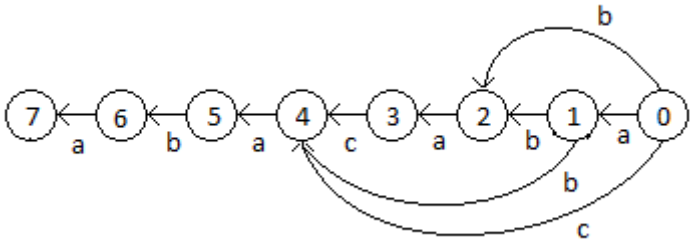

Figure 44: Smallest suffix automaton in Reverse Factor.

2.26 Turbo Reverse Factor

2.26.1 Description

The Turbo Reverse Factor [2] is an amelioration of the Reverse Factor algorithm that performs the searching phase in linear time in worst case. It takes advantage the fact that, after having matched a prefix u of P in the last attempt, in the next attempt when the right end of u has been reached, it is only needed to scan the rightmost half characters of u at worst case.

The preprocessing phase of the algorithm is the same as in Reverse Factor in computing the smallest suffix automaton. During the searching phase, let *disp(z, w)= d > 0* be the displacement of *z* in *w*, where *d* is the minimum position such that w[*m-d-*|*z*|-1] = *z*. In a typical situation, a prefix *u* has been matched in the text in the last attempt and in the current attempt the *m*-|*u*| characters right of *u* are about to be matched. Then there are four cases:

- *v* is not a factor of *P*, the shift is computed just like in Reverse Factor.
- *v* is a suffix of *P*, then the pattern has been matched.
- *v* is a factor but not a suffix of *P*, then it is needed to scan only the min{|*per(u)*|, |u|/2} rightmost characters of *u*.

As mentioned above, the searching phase is done in O(*n*), by performing at most 2*n* character inspections.

2.27 Forward Dawg Matching

2.27.1 Description

Forward Dawg Matching [2] is a string search algorithm that uses the smallest suffix automaton (or DAWG), introduced in the Reverse Factor section, to compute and store the largest factor of the pattern that ends at each position of the text. The preprocessing phase is similar to the one in Reverse Factor and consists of constructing the automaton, which also introduces the notion of longest state paths and suffix links. For every state *p*, let *length(p)* be the longest path from the initial state q_0 to *p*. Also let *S[p]* be the suffix link of state *p*. For *p* let *Path(p)=*(p_0,p_1, .. , p_l), be the suffix path such that p_0= *p*, where $1 \leq i \leq l$, p_i = *S*[p_i*-1]* and p_l = q_0. During searching phase, the text characters are scanned from left to right with *p* being the state of the current character. The state is updated with the target state of the first defined transition of the current character in *Path(p)*; if there is no such transition the state is updated with the initial state q_0. A pattern match is reported when *length(p) = m.*

The preprocessing phase takes time proportional to O(*m*) and the searching phase takes time O(*n*) performing exactly *n* character comparisons.

2.28 Backward Nondeterministic Dawg Matching

2.28.1 Description

Backward Nondeterministic Dawg Matching [2][21] is a variant of the Reverse Factor algorithm. It combines the advantages of the suffix automaton, which permits a search of a substring of the pattern and not only of a prefix or a suffix, with the efficiency of bit-parallelism similarly to the Shift-Or algorithm, making it quite performant when the pattern size is less than the word size of the computer. This achieved by computing the automaton in its simpler, nondeterministic form and can be simulated with bit-parallelism which intrinsically allows many operation to be performed at once.

Specifically, the algorithm uses a bit mask array B for each character of the alphabet c such that the i-th bit of B_c is set only if $P[i] = c$. Just like in Shift-Or, the state is stored in a bit vector d of size m, which should be less than the word size for the algorithm to be efficient. The bit d_i is activated when $P[m-i..m-1-i-k] = T[j+m-k..j+m-1]$. At each text position j, d id initialized to 1^{m-1} and updated with the formula $d = (d \& B[T[j]]) << 1$. If $d_{m-1} - 1$ at the m-th iteration then a pattern match is reported. When $d_{m-1} = 1$ during any iteration other than the m-th iteration, then a prefix of the pattern has been matched in the current window position j. The longest such prefix provides the next shift of the window.

2.29 Backward Oracle Matching

2.29.1 Description

Backward Oracle Matching [2][22] is another variant of the reverse Factor algorithm. It utilizes an elaborate structure called the factor oracle, which has $m+1$ states, is acyclic, recognizes at least all the suffixes of P and stores a linear number of transitions. The factor oracle is a compact deterministic finite automaton which recognizes all the suffixes of a string w and is denoted as $O(w) = \{\Sigma, Q, q_0, F, \delta\}$, where:

- Σ is the alphabet
- Q is the set of states
- q_0 is the initial state
- F is the set of final states

- $\delta: Q \times \Sigma = Q$ is the transition function

During preprocessing, the suffix oracle is constructed for the reverse pattern. Although it recognizes also some words that are not factors of the pattern, the only word of length greater or equal than *m* that it contains is the reverse pattern itself and thus can be used for pattern matching. This process is done is linear time and space.

The searching phase consists of scanning the characters of the window from right to left with the factor oracle staring from the initial state and it proceeds until no more transitions are defined for the current character. At this point, the prefix of the pattern that has been matched is a suffix of the scanned window, and is less than the path taken in the factor oracle from the initial state to the last defined state encountered. Thus the next shift can be easily computed. The searching process has a quadratic worst time complexity but is efficient in practice.

2.30 Galil-Seiferas

2.30.1 Description

Galil and Seiferas designed a linear time string search algorithm [23][2], which just as Karp-Rabin does not require dynamic storage allocation but also does not require any high level computational capabilities.

The algorithm introduces the use of variable *k*, which is proportional to its worst case running time, thus practically it should be a small value (k=4 is suggested by Galil, Seiferas). Also, let *reach* be a function such that $reach(i) = i + max\{i' \leq m - i: P[0..i'] = P[i+1..i'+i+1]\}$ for $0 \leq i < m$. Then a prefix $P[0..p]$ is considered to be a prefix period if it is basic, that is it cannot be written as a power of another word, and $reach(p) \geq kp$.

During the preprocessing phase, the algorithm attempts f uses methods *newP1*, *newP2* and *parse* to find a perfect factorization. That is a decomposition *uv* of *P* such that $u = P[0..s-1]$, $v = P[s..m-1]$. Method *newP1* stores the shortest prefix period of $P[s..m-1]$, method *newP2* stores the second shortest prefix period of $P[s..m-1]$ and method *parse* increments *s*. Thus right before searching we have:

- $P[s..m-1]$ has one prefix period at most.
- If $P[s..m-1]$ does not have a prefix period , its length is *p1*.
- $P[s..s+p1+q1-1]$ contains the shortest period of length *p1*.

- *P[s..s+p1+q1]* does not contain the shortest period of length *p1*.

An example of a perfect factorization is illustrated in Figure 45, where the pattern consists of *P[0..s-1]* and *P[s..m-1], P[s..m-1]* consists of $z^l z'az''$, *z* is basic, $|z| = p_1$, *z'* is a prefix of *z*, *z'a* is not a prefix of *z*, and $|z^l z'| = p1 + q1$.

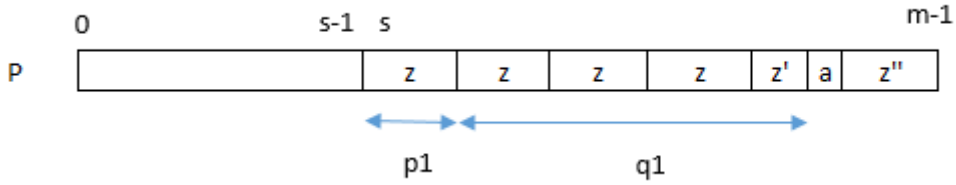

Figure 45: A perfect factorization.

During the searching phase, given the perfect factorization *uv*, the text is scanned for occurrences of *v* and for each one of them it looks if it is preceded by *u*. If *P[s..s+p1+q1-1]* has been matched then the window is shifted by *p1* and the attempt resumes with *P[s+q1];* otherwise if a mismatch occurs on *P[s+q]*, where $q \neq p1+q1$ then the window is shifted by *q/k+1* and the attempt resumes with *P[0]*. This phase yields a linear number of text character comparisons, bounded by 5*n*.

2.30.2 Implementation Details

Regarding the implementation of the algorithm in Java, the structure of the C code was maintained as is, with functions *newP1, newP2, parse* and *search (search* as private method to differentiate its signature from the public method *search* that returns the first occurrence*)* all implemented as methods. However, since we include two search methods, one that finds the first occurrence (*search*) and one that finds all the occurrences (*searchAll*), some modifications were included in order to avoid duplicated code. Both methods call the same *init* method that is the implementation of the function *GS.* Also, *t*wo instance variables have been introduced, an integer one for the result of *search* and a list of integers for the result of *searchAll.* The basic idea here is to use the same methods (*newP1, newP2, parse, search*) either we call *search* or *searchAll*, returning the appropriate result based on the public method that was called. To keep track of the called method, two flag variables were utilized.

A flag variable *searchAll*, that is true if we call the *searchAll* method and false if we call the *search*.

A flag variable *goOn* was introduced that defaults as true and becomes false when, during a call of the *search* method, the first match has been found. In the latter case, by checking this flag, the algorithm forces the method to stop the search and return. Since methods *newP1*, *newP2*, *parse* and *search* are heavily dependent on each other (Figure 46), a check for this flag has been added after each call on each of those methods, allowing for the searching procedure to end when a first match has been found.

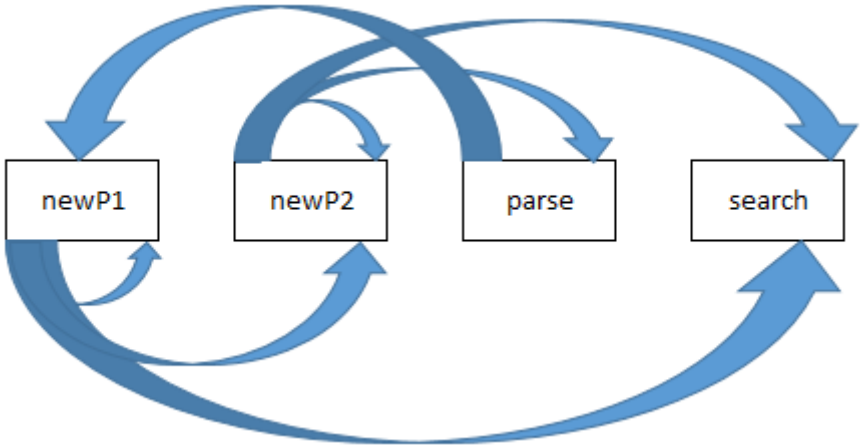

Figure 46: Dependent calls from each method in Galil-Seiferas.

2.31 Two Way

2.31.1 Description

The Two Way algorithm [2][24] presents an approach similar to Galil-Seiferas algorithm, but is conceptually simpler and needs only constant extra space. Its main principal is to factorize the pattern *P* in two parts, the left one (P_l) and the right one (P_r). Then, during the search phase the right part is scanned from left to right and then, if no mismatch occurs, the left part is scanned from right to left. The preprocessing phase aims to find a good factorization. This algorithm uses some notions in order to achieve this.

A repetition of a factorization *uv* is a factor *w* such that the two following properties hold:

- *w* is suffix of *u* or *u* is a suffix of *w*
- *w* is prefix of *v* or *v* is a prefix of *w*

This means that *w* occurs at both sides of the cutting point between *u* and *v*. the length of a repetition in *uv* is a local period and the minimum such value is the global period and

symbolized as *r(u, v).* Each factorization has at least one repetition, and one can see that $1 \leq r(u,v) \leq m$. When $r(u,v) = per(P)$ then we have a critical factorization. The algorithm chooses a critical factorization for the minimal $|P_l|$ such that $|P_l| < per(P)$. This computation requires finding the maximal suffixes for both orders ≤ and ~≤ and defining the length of the left factor P_l as the maximum of the lengths of the two suffixes. This can be done in linear time regarding the pattern size.

During the searching phase, as mentioned above, first the right part is scanned from left to right; in case of mismatch on the k-th character then the window shifts by k. Otherwise the algorithm proceeds scanning the left part from right to left, shifting the window by *per(P)* in case of mismatch. This phase runs in linear time performing at most $2n - m$ text character comparisons.

2.32 String Matching on Ordered Alphabets

2.32.1 Description

The String matching on Ordered Alphabets algorithm [2][25] exploits an ordered alphabet to offer a linear string search algorithm using only constant extra space. In a typical situation the window is placed on *T[j..j+m-1]*, a prefix *u* has been matched and mismatch occurs between *P[i]* and *T[j+i]*. In that case the algorithm attempts to find the period of *uT[j+i]*, or find an approximate value if it doesn't succeed.

The algorithm utilizes the notion of Maximal-Suffix decomposition (MS-decompositon). The MS-decomposition of a word *P* is defined to be tw^ew', meaning that

- $w^ew' = v$ is the maximal suffix of *P* in alphabetical order.
- *w* is basic
- $e \geq 1$
- *w'* is a proper prefix od *w*.

If *u* is a suffix of *w* then $per(P) = per(v) = |w|$, otherwise $per(P) > max\{|u|, min\{|v|, |tw^e|\}\} \geq |P|/2$.

The algorithm does not require a preprocessing step. Rather at each attempt it computes the maximal suffix of the matched prefix followed by the text character that

triggered the mismatch. In case of a mismatch of length *per(w)*, the maximal suffix does not need to be recalculated.

2.32.2 Implementation Details

One notable change in the implementation of the algorithm in Java from The C program, is that the variables *ip, jp, k, p* have been declared as arrays of integers of size 1 instead of integers, so that they can be passed by reference in the *nextMaximalSuffix* private method. This allows the method to change them without needing to declare them as instance variables, since they are different at every search step and are not reused.

2.33 Optimal Mismatch

2.33.1 Description

Optimal Mismatch [2][14] is another variant of the Quick Search algorithm that is quite efficient when dealing with large alphabets. Its main principle lies in choosing a scanning order that maximizes the possibility of a character mismatch at each text position. To achieve this, unlike the other algorithms stemming from the Boyer-Moore algorithm, where the scanning of the window is from right to left or can be done in any order, this one scans the pattern characters from the least frequent one to the most frequent one. This results in earlier mismatches and quicker shifts, and thus greater efficiency.

The preprocessing phase consists several steps. Initially, the frequencies of the letters of the alphabet in the text are calculated and the positions of the characters of the pattern are stored. Then the pattern characters are sorted according to two criteria; primarily in ascending order based on their individual frequency and, in case of a tie, in descending order based on their positions in the pattern (Figure 47). Finally, the bad-character shift table from Quick Search is constructed.

char	a	c	g	t
freq	23	6	15	9

i	0	1	2	3	4	5	6
P[i]	c	a	g	c	t	a	a
node.loc	3	0	4	2	6	5	1
node.c	c	c	t	g	a	a	a

Figure 47: Example of sorting the pattern characters after Optimal Mismatch preprocessing.

The searching phase is similar to Quick Search consisting of scanning the text positions and comparing following the order of the characters computed above. In case of a mismatch the bad-character table provides the next shift. This phase is quadratic in the worst case, but quite efficient in practice when dealing with large alphabets.

2.33.2 Implementation Details

Implementation-wise, several modifications where applied to the original C code, regarding not only the adaptation of the algorithm in Java but also its structure in general.

Since the algorithms in this thesis are implemented by preprocessing the pattern in the constructor and then providing the text as a parameter to the search method to launch a string search, the step of calculating the frequencies and sorting the characters was placed in the search method () and not in the constructor where the preprocessing usually takes place. Of course, the algorithm can be modified to preprocess first the text and then receive the pattern as a search parameter reducing the preprocessing time and allowing to search for multiple patterns in a source text more efficiently.

Concerning the implementation, the C structure *patternScanOrder* was implemented as a private static class *patternScanEmement* (Figure 48).

```
typedef struct patternScanOrder {
   int loc;
   char c;
} pattern;
```

```
private static class PatternScanElement{
    private char c;
    private int loc;
}
```

Figure 48: Comparison of the auxiliary structure *patternScanOrder* between C (left) and Java (right).

To manage the sorting part, a private static comparator class *ByFreqLoc* was created that implements the Comparator interface for sorting *patternScanEmement* instances, replacing the C function *optimalPcmp* (Figure 49).The comparator constructor takes a integer array of frequencies based on which it sorts the *patternScanEmement* instances first in ascending order based on their characters' (*c*) frequencies (*freq[c]*) and then in descending order according to their positions (*loc*) in the pattern in case of a tie.

```c
/* Optimal Mismatch pattern comparison function. */
int optimalPcmp(pattern *pat1, pattern *pat2) {
   float fx;

   fx = freq[pat1->c] - freq[pat2->c];
   return(fx ? (fx > 0 ? 1 : -1) :
               (pat2->loc - pat1->loc));
}
```

```java
private static class ByFreqLoc implements Comparator<PatternScanElement> {
    private final int[] freq;

    public ByFreqLoc(int[] freq) {
        this.freq = freq;
    }

    public int compare(PatternScanElement pse1, PatternScanElement pse2) {
        if (freq[pse1.c] > freq[pse2.c])
            return 1;
        if (freq[pse1.c] < freq[pse2.c])
            return -1;
        if (pse1.loc > pse2.loc)
            return -1;
        if (pse1.loc < pse2.loc)
            return 1;
        return 0;
    }
}
```

Figure 49: Comparison of the comparison functionality implementation between C (up) and Java (down).

.

Finally, since the bad-character shift array *qsBc* from Quick Search is utilized here, the same modification of the length of the array from *m* to *m+1* applies here as well.

2.34 Maximal Shift

2.34.1 Description

Maximal Shift [2][14] is the implementation of another variant of the Quick Search algorithm. Its approach is similar to the one mentioned in the Optimal Shift section. The window characters are scanned with a particular order aiming to increase efficiency and the shifts are performed according to the bad-character shift table introduced in the Quick Search section. The algorithm differentiates in the order that it employs to perform the comparisons in the window. Specifically, it sorts the characters of the pattern from the one that can lead to a longer shift, to the one that can lead to the shorter shift.

The preprocessing phase takes time and space proportional to $O(m^2+\sigma)$ and $O(m+\sigma)$ respectively, while the searching phase can be quadratic in the worst case.

2.35 Skip Search

2.35.1 Description

The Skip Search algorithm [2][26] utilizes buckets of character positions in the pattern to minimize work during each attempt and to maximize shift lengths.

During preprocessing phase, it creates a bucket for each character of the alphabet with its positions in the pattern in descending order (Figure 50). If a character *c* occurs *k* times in the pattern, then its bucket will contain all *k* values of its positions. This data structure can be represented by an array of linked lists, where each linked list is a bucket (Figure 51). This implementation is efficient for small alphabets, otherwise many buckets remain empty. The preprocessing step can be done in $O(m+\sigma)$ time and space complexity.

i	0	1	2	3	4	5	6
P[i]	t	c	a	c	a	g	a

char	bucket
a	{6, 4, 2}
c	{3, 1}
g	{5}
t	{0}

Figure 50: Example of buckets used in Skip Search for pattern = *tcacaga.*

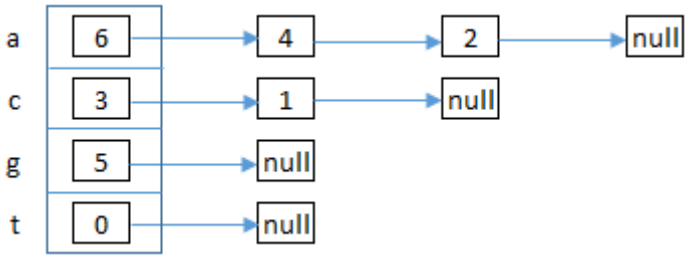

Figure 51: Data structure implementing of Skip Search buckets using an array of linked lists for pattern = *tcacaga*.

The searching phase consists of traversing each text position starting from position *m*-1 up to *n*-1. For each current character in position *j*, the positions are retrieved from their respective bucket to indicate the starting point of the window, meaning that if a character is located in positions pos1, pos2 and pos3 of the pattern then, during the window scanning, those positions will be considered in descending order to help determine the starting position of the window according to the formula *j* – *pos.* A searching example is illustrated at Figure 52. The searching phase is quadratic in worst case though efficient on practice.

```
j = m-1 = 6

          0  1  2  3  4  5  6  7  8  9  10 11
T         c  g  g  a  t  c  a  c  a  g  a  a

start = j - bucket(T[j]) = 6 - 6 = 0
P         t  c  a  c  a  g  a

start = j - bucket(T[j]) = 6 - 4 = 2
P               t  c  a  c  a  g  a

start = j - bucket(T[j]) = 6 - 2 = 4
P                  t  c  a  c  a  g  a
```

Figure 52: The searching phase of Skip Search.

2.36 KMP Skip Search

2.36.1 Description

KMP Skip Search [2][26] is an amelioration of the Skip Search algorithm, achieving a linear time searching performance. This is possible by utilizing the Morris-Pratt array *mpNext* and the Knuth-Morris-Pratt array *kmpNext.*

The preprocessing phase involves the construction of the two shift arrays. As mentioned in their respective sections, *mpNext*[*i*] ($1 \leq i \leq m$) contains the length of the longest border of *P[0..i-1]* with *mpNext[0] = -1, while kmpNext[i]* ($1 \leq i \leq m$) represents the length of the longest border of *P[0..i-1]* followed by a character different than *P[i]* with *kmpNext[0] = -1*. Finally the bucket list is computed. This phase runs in time and space proportional to $O(m+\sigma)$.

During the searching phase, in a typical situation of KMP Skip Search, the algorithm is on the text position *j* with *P[i] = T[j]* and *start = j – I* is the start of the window of the current attempt. Now let *wall* be the rightmost current scanned position meaning the previous *wall-start-1* characters of the window have been matched. From that position the comparisons between the pattern and the text are performed from left to right. Let *k* be the least position such that $k \geq wall - start$ and $P[k] \neq T[start+k]$ or $k = m$ if a pattern match has been reported in position *start.* Then we set *wall = start + k.* The algorithm then computes two new start positions for the window; a *skipStart* position, computed according to the *AdvancedSkip* algorithm and a start position *kmpStart* based on *kmpNext.* This results in one of the following cases:

- *skipStart* < *kmpStart,* resulting in computing a new *skipStart* value and the two start values are compared again.
- *kmpStart* < *skipStart* < *wall,* a new value *kmpStart* is computed from *mpNext* and the two start values are compared again.
- *skipStart* = *kmpStart,* results in a new attempt for *start = skipStart.*
- *kmpStart* < *wall* < *skipStart,* results in a new attempt for *start = skipStart.*

This phase is done in linear time proportional to the length of the text.

2.37 Alpha Skip Search

2.37.1 Description

Alpha Skip Search [1][2] is another improvement of the Skip Search algorithm, which achieves a linear time and space preprocessing phase regarding the length of the pattern. Just like Skip Search the algorithm uses buckets of positions. It differentiates, however, by using the buckets to store the positions of all factors of length $l = log_{\sigma}m$ of the pattern. In order to store the factors a trie structure is used, the leaves of which represent the pattern factors of length l and store the list of positions they appear on. This is achieved in linear time. During the searching phase, the algorithm inspects the lists of the text factors $T[j..j+l-1]$ for all $j = k..(m-l+1)$ for k in the range $T[1, floor\{ (n-l)/ m \}]$. This phase is quadratic but efficient in practice for small alphabets and large patterns.

3 Benchmarks

3.1 Methodology

To compare the algorithms presented we consider their execution time for finding a pattern in a larger text. The text used in the benchmarks is a genome sequence [27] consisting of 4 distinct characters A, C, G, T and containing about 4.5 million characters. A text of that scale was selected to stretch and challenge the performance capabilities of the algorithms.

Since each patterns size's impact varies during the searching phase of a string matching algorithm, this benchmark considers multiple pattern sizes grouped into two categories, the small sizes consisting of {2, 3, 4, 5, 6, 7, 8, 9, 10} and the larger sizes consisting of {10, 20, 40, 80, 160, 320, 640}. Each algorithm, for each pattern size, is tested against 10000, randomly selected form the above source text, patterns of current pattern size. A pattern sample of this size aims to compensate for the position factor impact, meaning that in the case we tested, for each pattern size, only one randomly selected pattern in such a large text, for one pattern size the random pattern could be located towards the end of the text, while for another it could be located in the beginning of the text resulting in misleading (as to the comparison of the algorithms) execution times. For each pattern size, all algorithms were tested on the same sequence of (10000) randomly generated patterns and the average of the respective execution times was stored referring to the current algorithm and the current pattern size (Figure 53).

The above procedure was applied twice, once for searching the first occurrence of each pattern in the text and once for searching all occurrences, yielding data for four major scenarios:

- Finding the first occurrence for small pattern sizes
- Finding the first occurrence for large pattern sizes
- Finding all occurrences for small patterns sizes
- Finding all occurrences for large patterns sizes

The results were organized and displayed in three sections. In the first section of individual results, the time of each algorithm on the above four scenarios is displayed in a line graph along with the minimum and average times of all algorithms per pattern size. In the second section, the comparative results of all algorithms are displayed in a bar

graph, sorted from fastest to slowest. Each bar corresponds to its corresponding algorithm's average performance on all pattern sizes. This bar chart has been done for each one of the previously mentioned four major scenarios. In the third section, the same collective results of each scenario were organized into the four categories according to which the algorithms are grouped concerning the order of comparisons performed (left to right, right to left, in a specific order, in any order).

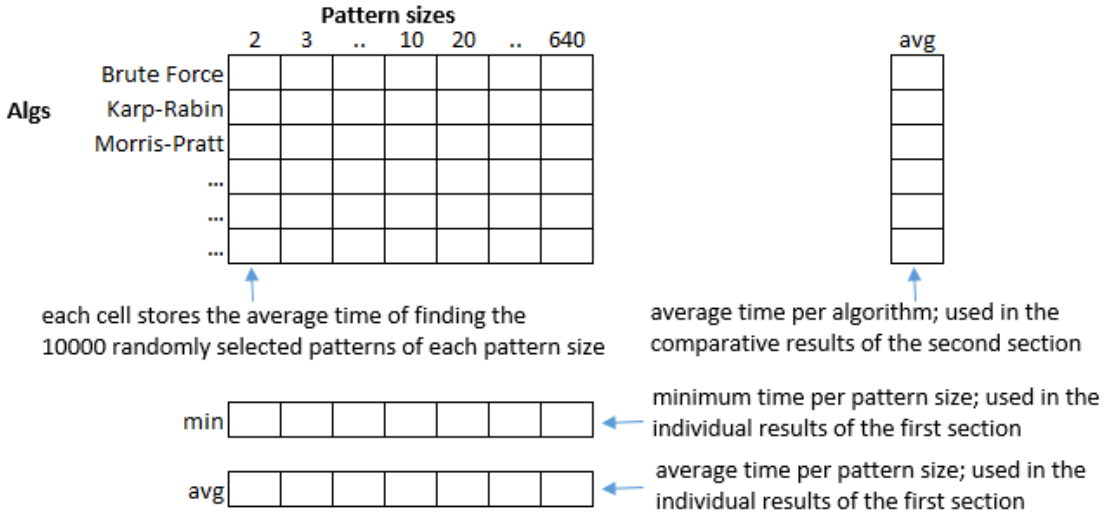

Figure 53: Organization of data in the benchmark suite.

3.2 Technical Details

To execute the above benchmarking procedure a benchmark suite was implemented in Java. The suite automatically tests all algorithms on all pattern sizes, as previously described, and, right after, automatically confirms the correctness of the results from each benchmark run. To achieve this the Java Reflection API has been utilized, which allows to modify the behavior of classes and methods in runtime.

Essentially, the benchmark program loops at each pattern size and produces a sequence of randomly selected patterns, substrings of the source text, of length equal to the current pattern size. Then it tests each algorithm on this sequence of patterns and stores the average time for each. Each algorithm instance is created at runtime via Reflection by looping in a String array that stores all the class names of the algorithms implemented in Java. Similarly, each search method call is performed via Reflection.

Right before each algorithm executes a string search for a pattern, a result correctness check occurs. Specifically, two instances are created with reflection, one for

the current algorithm and one for the Brute Force algorithm, the simplest string matching algorithm. Then the search of the pattern is executed, again with Reflection for both instances and the results are compared. If there is a mismatch, an Exception with a descriptive message is thrown and the execution of the whole benchmark procedure stops; otherwise, right after, the current algorithm is again tested on the pattern and the execution is timed. This part was no necessary for the benchmarking procedure, however it proved significantly useful in debugging the more complex string matching algorithm implementations.

Finally, all the results were exported in separate .csv files for each algorithm, as well as for the minimum and average values per pattern size and for the average time of each algorithm on all patterns.

The technical characteristics of the machine that performed the benchmarks were:

- Processor: AMD FX(tm)-6350 6 Core Processor @ 3.9 GHz
- RAM: 8 GB
- OS: Windows 10
- System type: 64-bit Operating System, x64-based precessor

3.3 Individual results

3.3.1 Brute Force

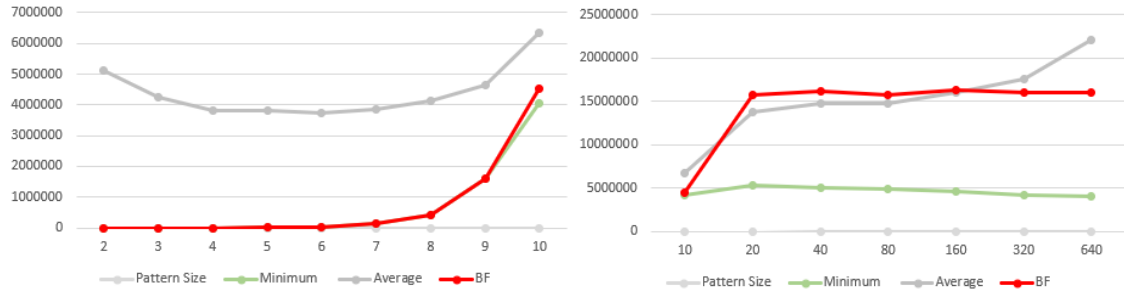

Figure 54: Brute Force - results of finding the first occurrence of small patterns (left) and large patterns (right).

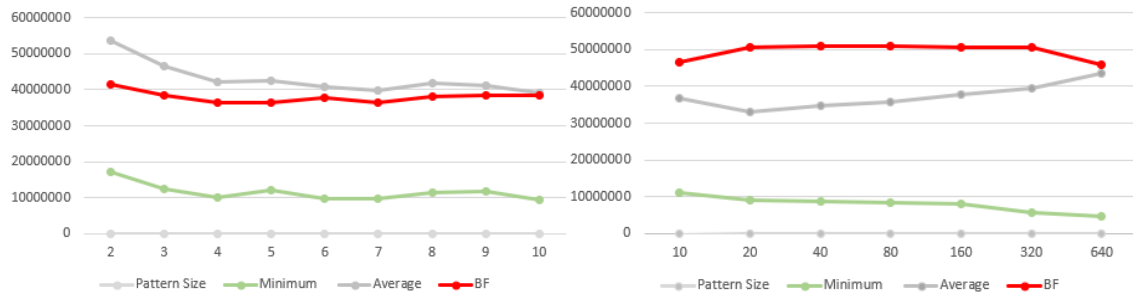

Figure 55: Brute Force - results of finding all occurrences of small patterns (left) and large patterns (right).

The Brute Force algorithm seems to be the most efficient approach when searching for the first occurrence of small patterns of length ranging from 2 to 9, while for larger patterns it performs similarly to the algorithms' average performance (Figure 54Figure 54: Brute Force - results of finding the first occurrence of small patterns (left) and large patterns (right).). This can be attributed to the lack of a preprocessing phase and the small size of the alphabet ($\sigma = 4$). Brute Force lacks preprocessing computations in contrary to almost all the other benchmark's algorithms. Also, most of the benchmark's algorithms use a shift table to perform jumps in the text when a mismatch occurs, the impact of which is relative to the size of the alphabet in the strings, since more distinct letters offer more shifts. The small alphabet used here presents only a few shifts, the effect of which is countered by the complex precomputations that add excessive overhead for small patterns.

Concerning the search for all occurrences, the algorithm performs a little better than the average algorithms' performance for small patterns but becomes slower for pattern sizes greater than 10, although not in an increasing factor (Figure 55).

3.3.2 Deterministic Finite Automaton

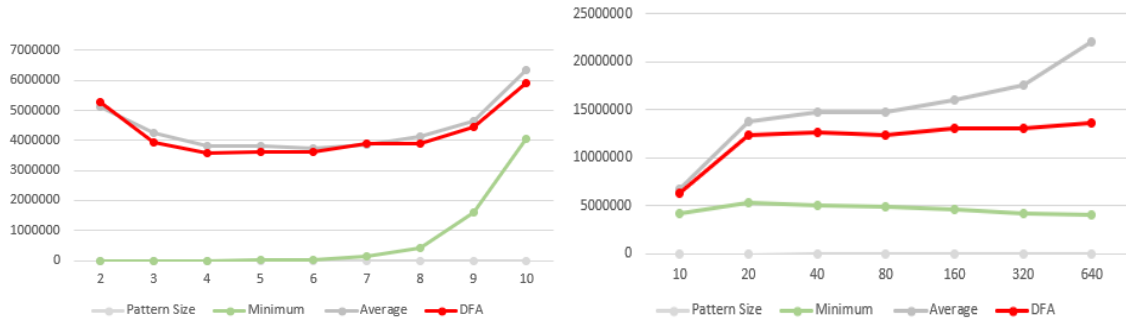

Figure 56: Deterministic Finite Automaton - results of finding the first occurrence of small patterns (left) and large patterns (right).

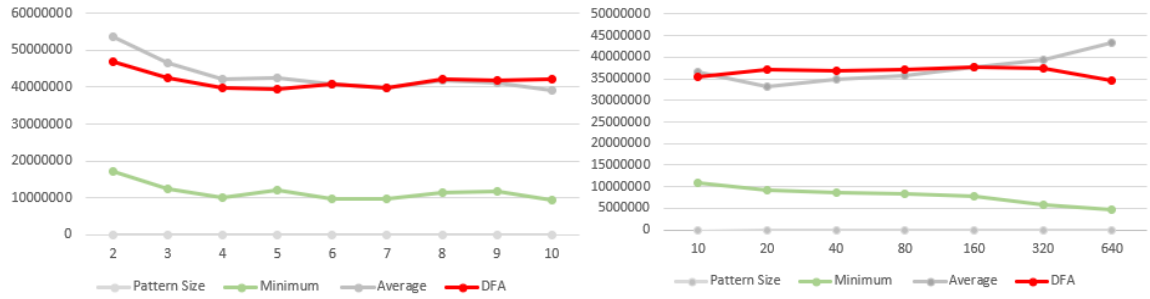

Figure 57: Deterministic Finite Automaton - results of finding all occurrences of small patterns (left) and large patterns (right).

The Deterministic Finite Automaton algorithm appears to be following closely the average algorithms' performance for either searching the first occurrence or searching for all occurrences of the pattern. One can notice that for large patterns (>160) the performance improves compared to the average.

3.3.3 Karp-Rabin

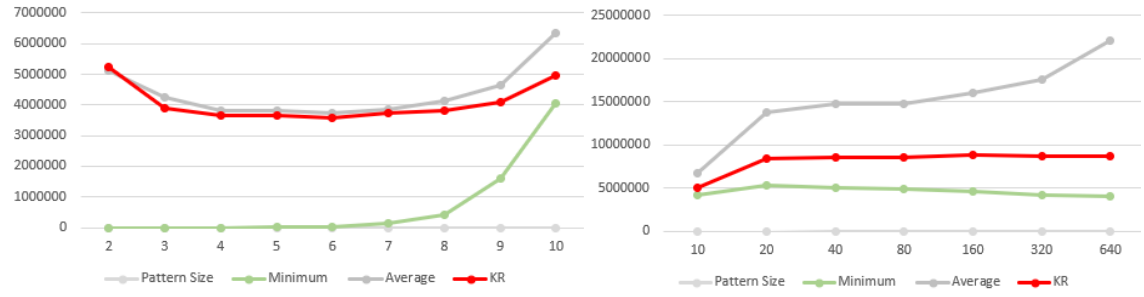

Figure 58: Karp-Rabin - results of finding the first occurrence of small patterns (left) and large patterns (right).

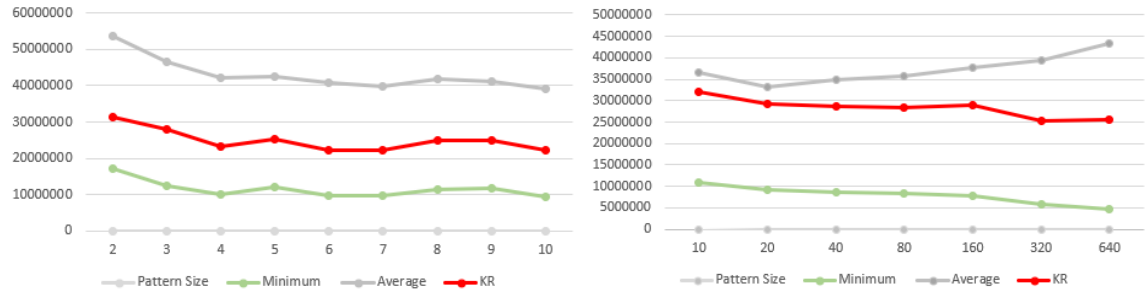

Figure 59: Karp-Rabin - results of finding all occurrences of small patterns (left) and large patterns (right).

One can notice that Karp-Rabin is more performant when searching for all occurrences of a pattern, especially for large patterns (Figure 59). It also presents a steady efficiency when searching for the first occurrence for large patterns compared to the increasing average (Figure 58). This can be attributed to its linear scanning of the text using the hash function that avoids quadratic comparisons, rendering the algorithm ideal for tasks as plagiarism detection.

3.3.4 Shift-Or

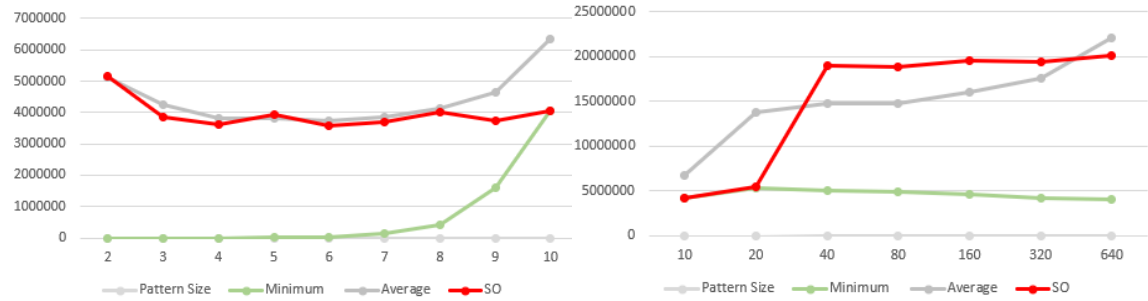

Figure 60: Shift-Or - results of finding the first occurrence of small patterns (left) and large patterns (right).

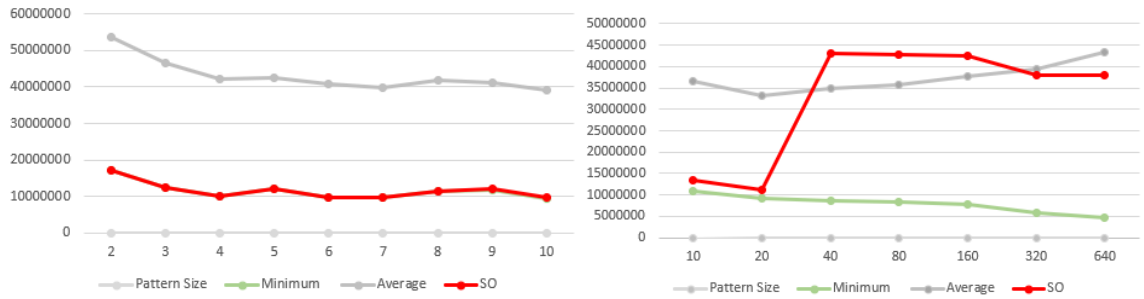

Figure 61: Shift-Or - results of finding all occurrences of small patterns (left) and large patterns (right).

Shift-Or implementation's use of bitwise operations render it the most efficient string search algorithm for finding all the occurrences of small patterns of length 2 to 10 and one of the fastest for up to word size length patterns (in this case 31). It seems also to be one of the most efficient approaches for searching the first appearance of patterns the length of which ranges from 8 to the word size. Since this implementation handles larger-than-the-word-size patterns by matching the first *word-size* characters and then sequentially comparing the rest, just like the Brute Force algorithm, its performance for patterns that exceed the word size in length, highly resembles Brute Force's.

3.3.5 Morris-Pratt

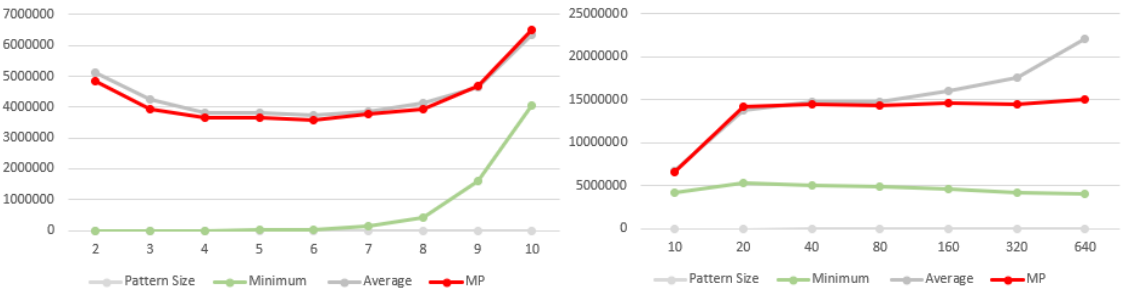

Figure 62: Morris-Pratt - results of finding the first occurrence of small patterns (left) and large patterns (right).

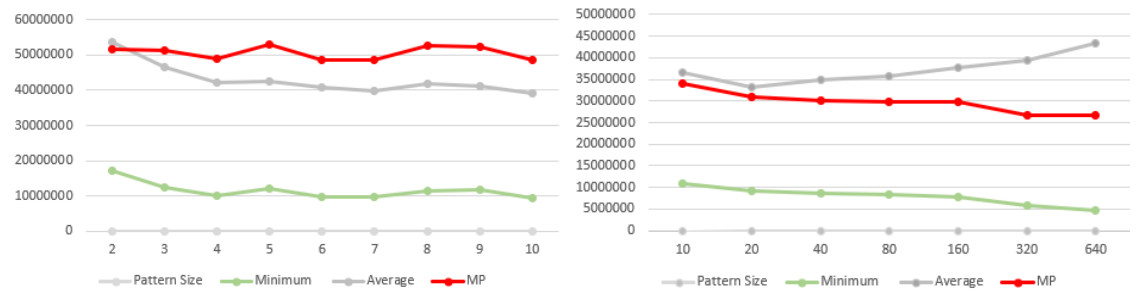

Figure 63: Morris-Pratt - results of finding all occurrences of small patterns (left) and large patterns (right).

The Morris-Pratt algorithm seems to perform well in searching for the first or all the appearances of large patterns, but appears relatively slow in finding all the occurrences of small patterns. This can be attributed to the fact that its preprocessing shift values are restricted by the small alphabet ($\sigma = 4$) encountered in this benchmark, since for a mismatch there are only 3 possible shifts.

3.3.6 Knuth-Morris-Pratt

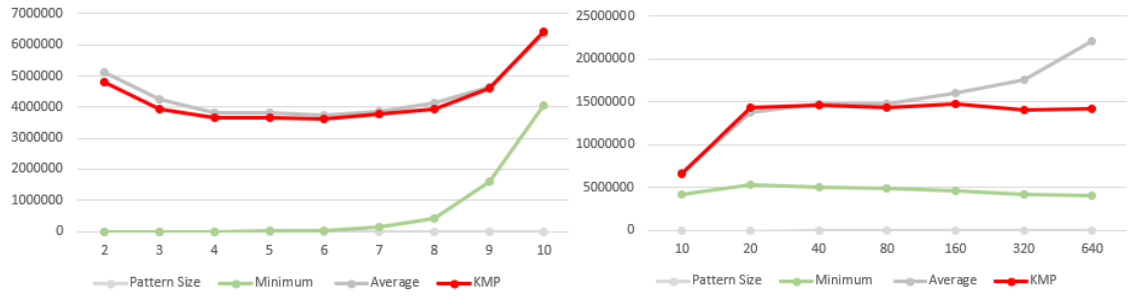

Figure 64: Knuth-Morris-Pratt - results of finding the first occurrence of small patterns (left) and large patterns (right).

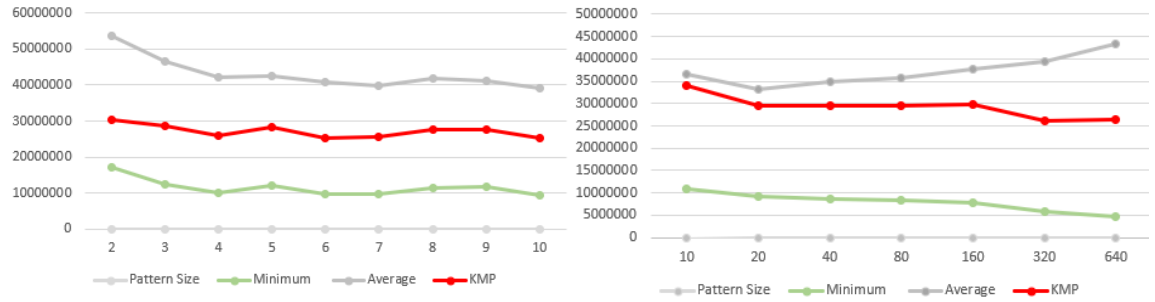

Figure 65: Knuth-Morris-Pratt - results of finding all occurrences of small patterns (left) and large patterns (right).

The Knuth-Morris-Pratt (KMP) seems to perform better that its predecessor Morris-Pratt regarding small patterns (Figure 65) being considerably faster than the average time of the benchmark in finding all the occurrences of a pattern. This can be attributed to the fact that KMP produces better shifts by considering tagged borders. Apart from that it appears to have the same efficiency in the other situation.

3.3.7 Simon

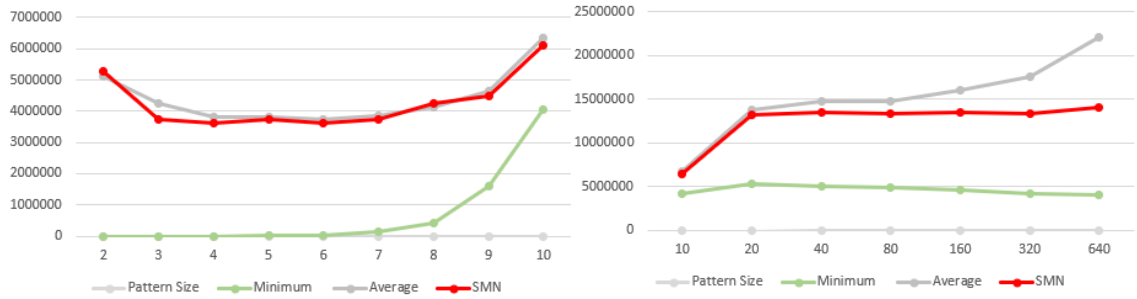

Figure 66: Simon - results of finding the first occurrence of small patterns (left) and large patterns (right).

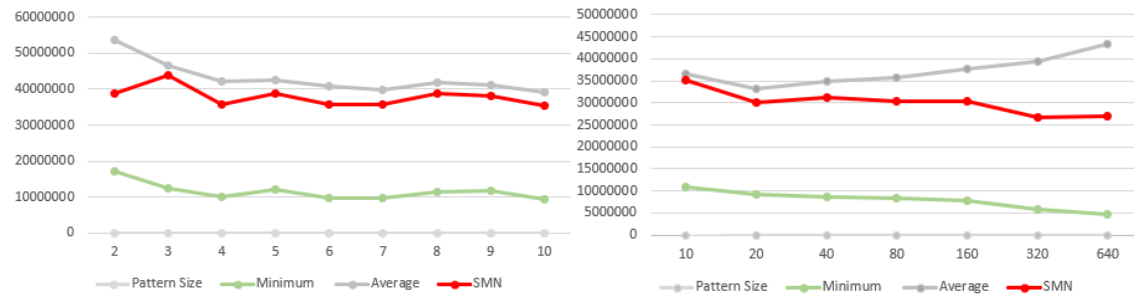

Figure 67: Simon - results of finding all occurrences of small patterns (left) and large patterns (right).

Although it mainly aims to be a more memory-efficient implementation of the Deterministic Finite Automaton (DFA) algorithm, Simon performs slightly better than the former during all occurrences searches regardless of pattern size (Figure 67), being in fact faster than the respective average benchmark time. This can be due to the algorithm storing only the significant edges that cannot be deduced otherwise. The performance remains approximately the same with DFA in the rest cases.

3.3.8 Colussi

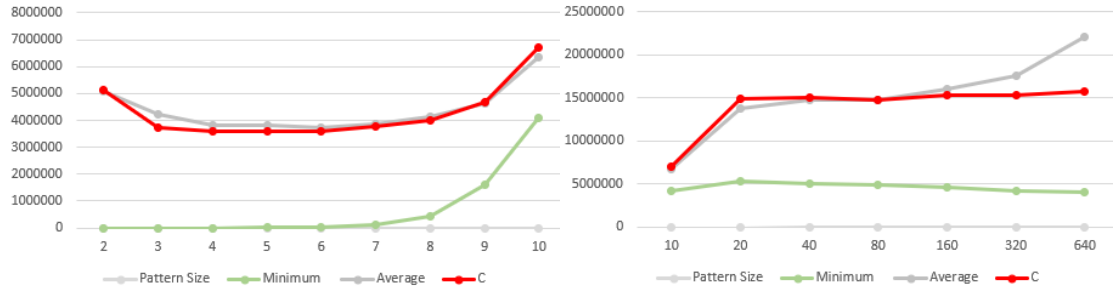

Figure 68: Colussi - results of finding the first occurrence of small patterns (left) and large patterns (right).

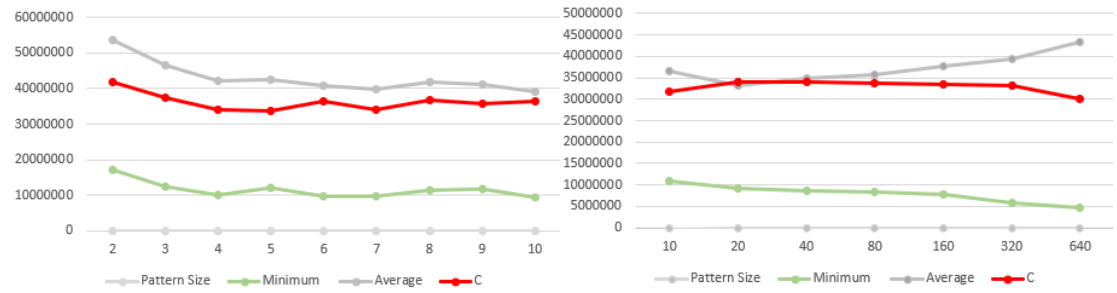

Figure 69: Colussi - results of finding all occurrences of small patterns (left) and large patterns (right).

The Colussi algorithm improves the Knuth-Morris-Pratt (KMP), by applying tighter conditions regarding selecting and performing the shifts. However it appears it performs slightly slower than KMP when searching for all occurrences of patterns (Figure 69), which may be due to its considerably more computational heavy preprocessing phase.

3.3.9 Galil-Giancarlo

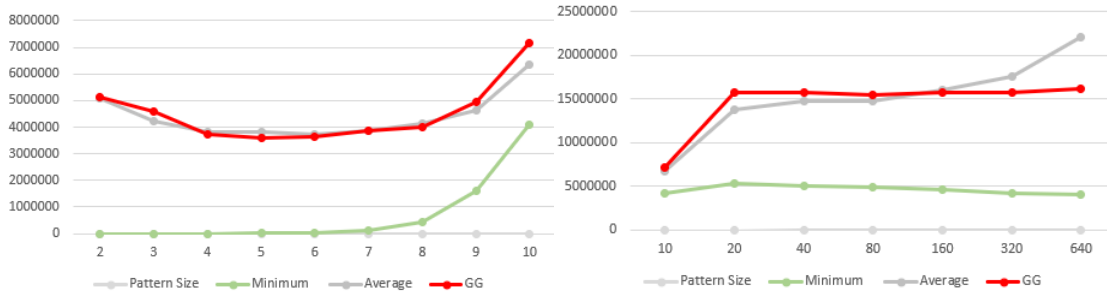

Figure 70: Galil-Giancarlo - results of finding the first occurrence of small patterns (left) and large patterns (right).

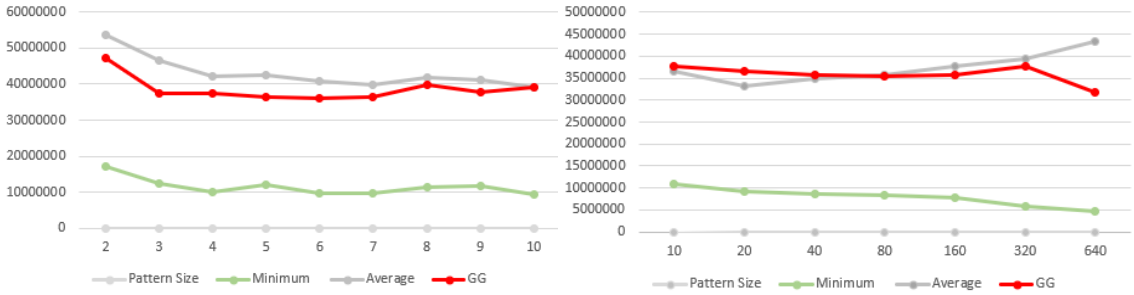

Figure 71: Galil-Giancarlo - results of finding all occurrences of small patterns (left) and large patterns (right).

One can notice that Galil-Giancarlo's performance resembles Colussi's regarding matching the first occurrence of a pattern regardless its size (Figure 70). Its efficiency however seems to slightly deteriorate during searching for all occurrences of a pattern, except for large patterns (>320) when it seems to be more efficient than Colussi (Figure 71).

3.3.10 Apostolico-Crochemore

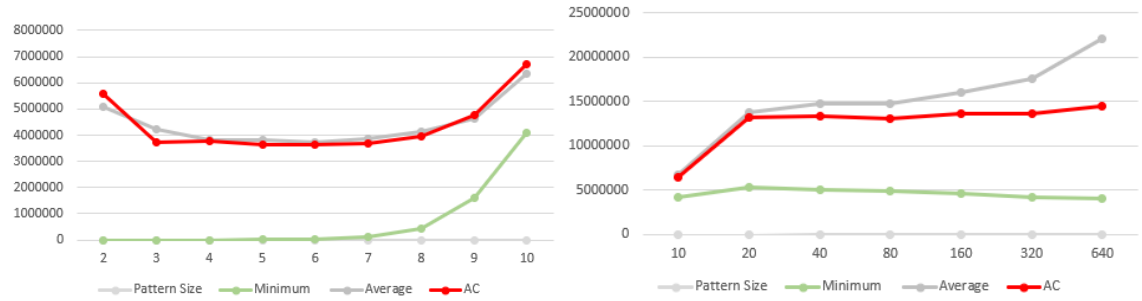

Figure 72: Apostolico-Crochemore - results of finding the first occurrence of small patterns (left) and large patterns (right).

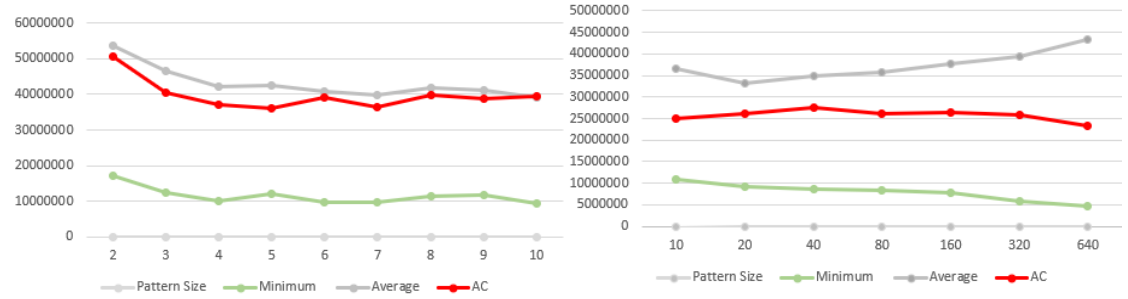

Figure 73: Apostolico-Crochemore - results of finding all occurrences of small patterns (left) and large patterns (right).

The Apostolico-Crochemore algorithm splits the pattern into two parts, the biggest prefix that is a power of a single character and the rest of the pattern, performing the comparisons first in the latter and lastly in the former, and utilizing the shift table from Knuth-Morris-Pratt (KMP) to perform the jumps. Generally it seems to perform better than the benchmark average in most of the occasions, especially when searching for all the occurrences for large patterns (Figure 73), outperforming KMP in this case.

3.3.11 Not So Naïve

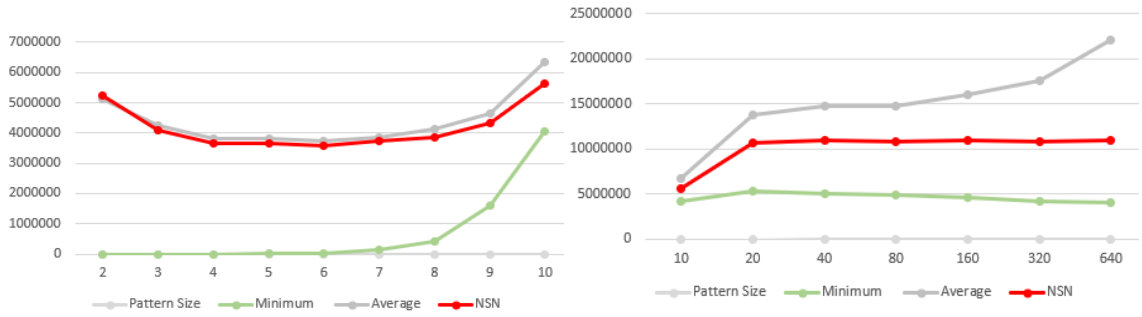

Figure 74: Not So Naive - results of finding the first occurrence of small patterns (left) and large patterns (right).

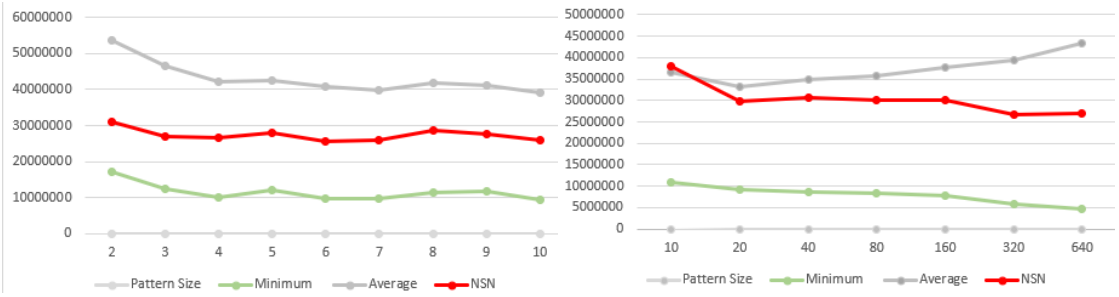

Figure 75: Not So Naive - results of finding all occurrences of small patterns (left) and large patterns (right).

Not So Naïve (NSN) is a variant of the Brute Force algorithm that slightly differentiates in the order the comparisons are performed. Notably, compared the latter, NSN doesn't exhibit the same efficiency in matching the first occurrence of small patterns (Figure 74), but it prevails in all the other cases (Figure 74, Figure 75). Specifically it behaves more efficiently in searching for all occurrences for small patterns and achieves times less than the benchmark average concerning searching for the first or all occurrences of large patterns.

3.3.12 Boyer Moore

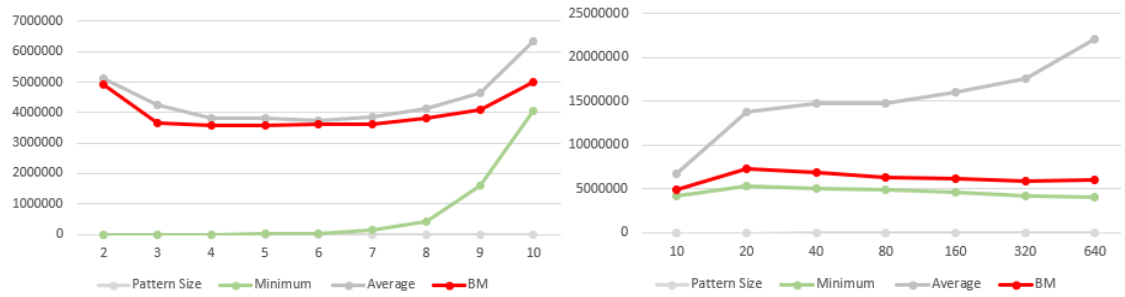

Figure 76: Boyer-Moore - results of finding the first occurrence of small patterns (left) and large patterns (right).

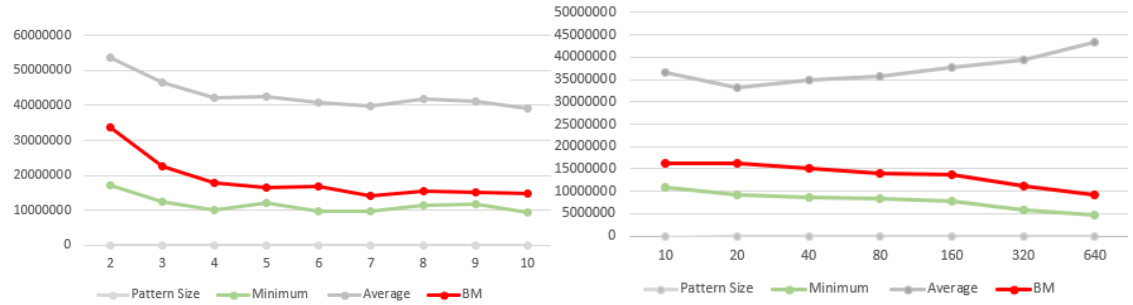

Figure 77: Boyer-Moore - results of finding all occurrences of small patterns (left) and large patterns (right).

The Boyer-Moore algorithm is considered one of the most efficient string search algorithms. Indeed, the algorithm's performance approaches the benchmark's minimum times in almost all the cases, improving as the length of the pattern increases (Figure 76, Figure 77). Remarkably, the algorithm's times are not so close to the minimum ones in the case of the first occurrence matching for small patterns (Figure 76), which have been recorded by Brute Force (Figure 54). As mentioned in the latter's results, its simple loop operations and lack of preprocessing phase prevail over Boyer-Moore's computationally heavier shift tables construction and overhead for small patterns representing a small alphabet ($\sigma = 4$).

3.3.13 Turbo Boyer-Moore

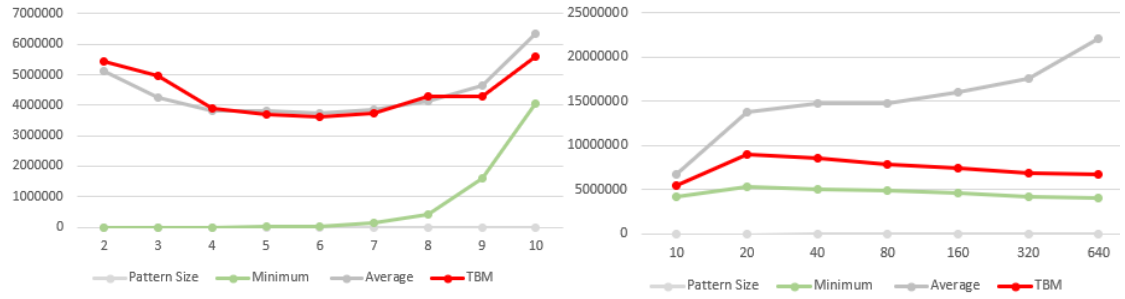

Figure 78: Turbo Boyer-Moore - results of finding the first occurrence of small patterns (left) and large patterns (right).

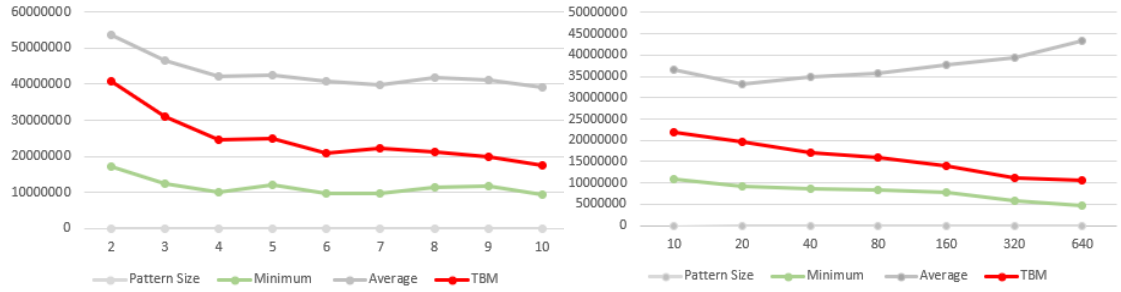

Figure 79: Turbo Boyer-Moore - results of finding all occurrences of small patterns (left) and large patterns (right).

Although Turbo Boyer-Moore is an improvement of the Boyer-Moore algorithm, it seems to perform slightly slower than the latter when searching for small patterns. Nevertheless, its performance still appears to improve as the size of the pattern increases.

3.3.14 Apostolico-Giancarlo

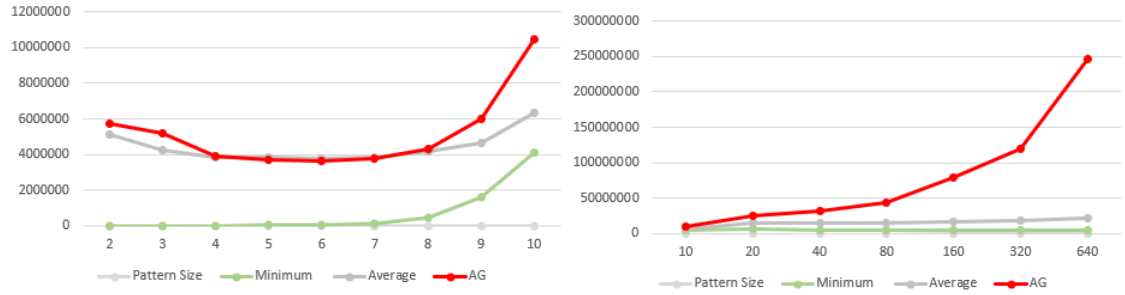

Figure 80: Apostolico-Giancarlo - results of finding the first occurrence of small patterns (left) and large patterns (right).

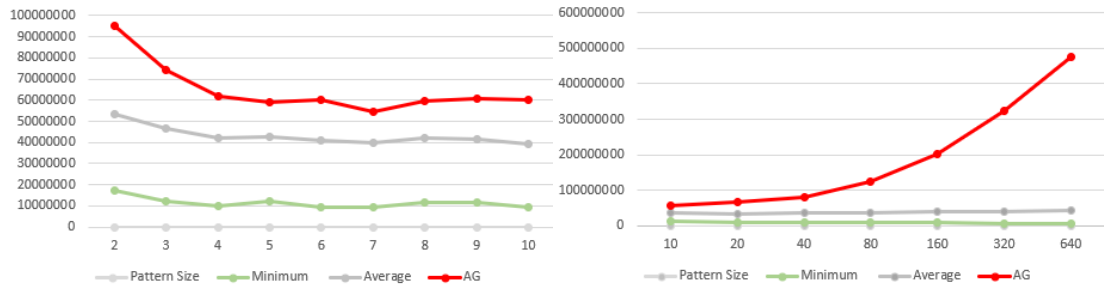

Figure 81: Apostolico-Giancarlo - results of finding all occurrences of small patterns (left) and large patterns (right).

One can notice that the Apostolico-Giancarlo algorithm, a variant of the Boyer Moore algorithm that can remember matched suffixes of the window, a quite slow performance when searching for large patterns (Figure 78, Figure 79); a performance that seems to rapidly deteriorate as long as the pattern length increases. During search for small patterns the performance of the algorithm approaches more the benchmark's average.

3.3.15 Reverse Colussi

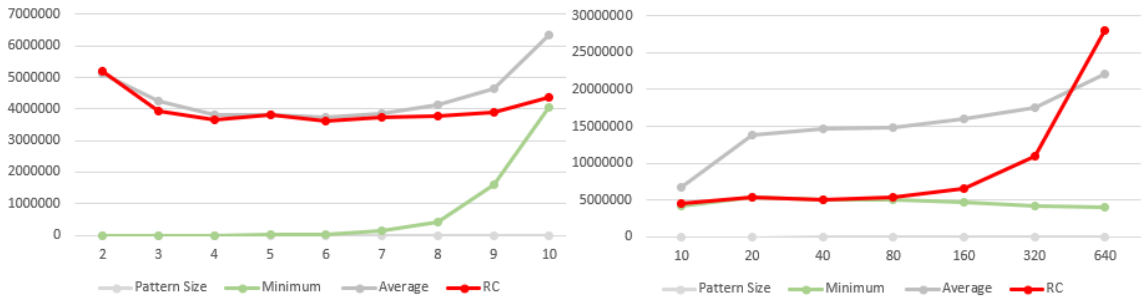

Figure 82: Reverse Colussi - results of finding the first occurrence of small patterns (left) and large patterns (right).

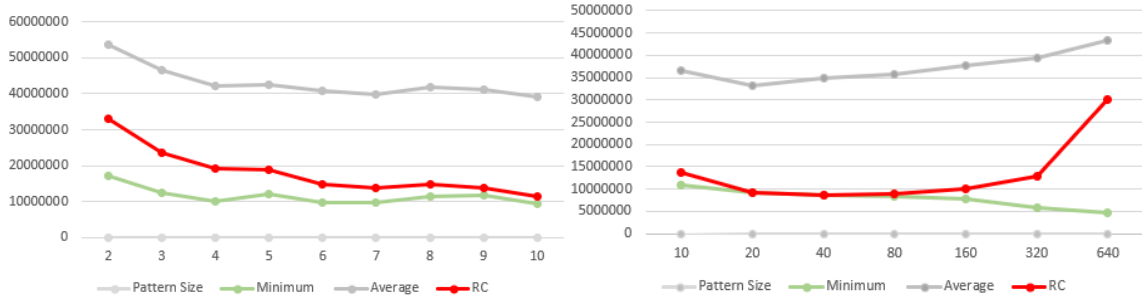

Figure 83: Reverse-Colussi - results of finding all occurrences of small patterns (left) and large patterns (right).

Reverse Colussi, which is an improved version of the Boyer-Moore algorithm, seems to clearly outperform the latter regarding medium to medium-large pattern sizes; specifically it approaches the benchmark's fastest times for pattern lengths ranging from 10 to 160. However its performance seems to rapidly deteriorate for any patterns larger than the aforementioned range.

3.3.16 Horspool

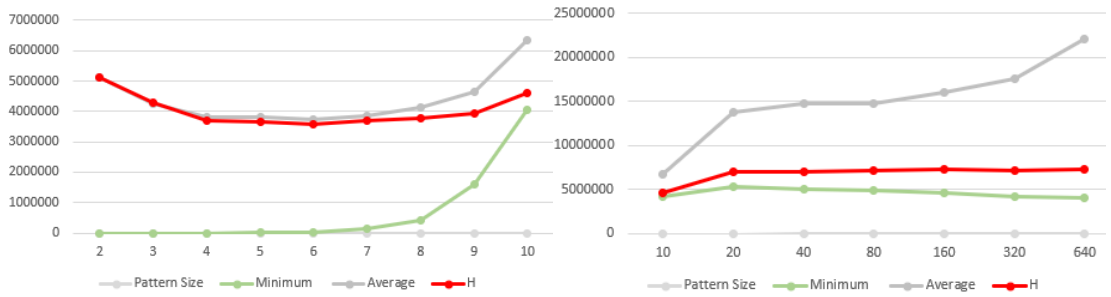

Figure 84: Horspool - results of finding the first occurrence of small patterns (left) and large patterns (right).

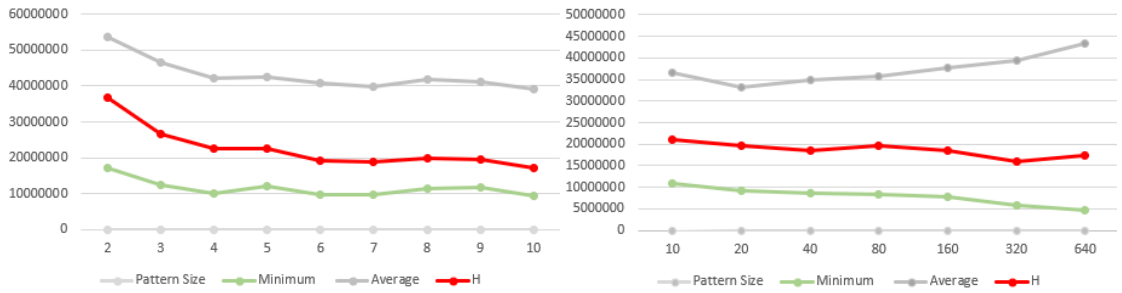

Figure 85: Horspool - results of finding all occurrences of small patterns (left) and large patterns (right).

The Horspool algorithm, a simplified implementation of Boyer-Moore which constructs and utilizes only the bad-character shift table, appears to not perform any better than Boyer-Moore in any of the cases of the benchmark. Nevertheless, it offers well below average searching times that don't increase as the pattern length increases.

3.3.17 Quick Search

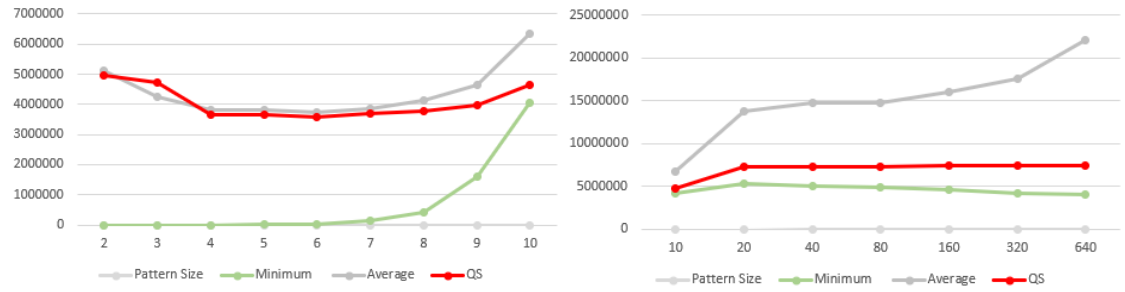

Figure 86: Quick Search - results of finding the first occurrence of small patterns (left) and large patterns (right).

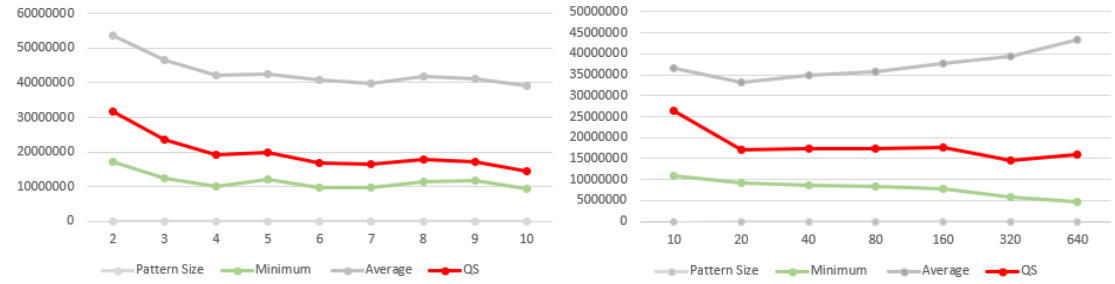

Figure 87: Quick Search - results of finding all occurrences of small patterns (left) and large patterns (right).

Quick Search, yet another simplification of the Boyer-Moore algorithm, exhibits a performance behavior that seems more similar to the latter's in comparison with Horspool. The algorithm is quite efficient in all pattern sizes for the small alphabet size that is used in this benchmark, displaying a stable performance as long as the pattern size increases.

3.3.18 Tuned Boyer-Moore

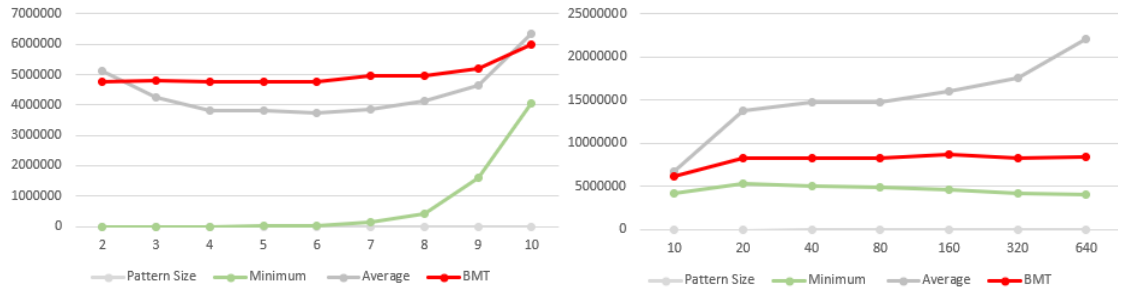

Figure 88: Tuned Boyer-Moore - results of finding the first occurrence of small patterns (left) and large patterns (right).

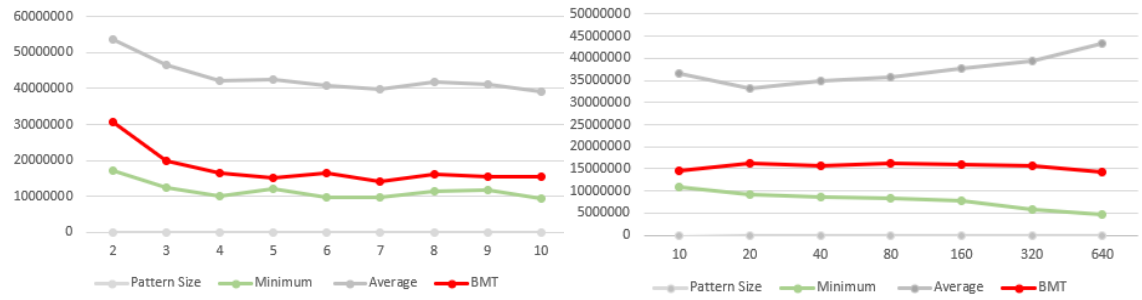

Figure 89: Tuned Boyer-Moore - results of finding all occurrences of small patterns (left) and large patterns (right).

The Tuned Boyer-Moore algorithm, another simplified version of Boyer-Moore that unrolls 3 shifts in a row before each attempt and uses only the bad-character shift table, displays a slightly worse behavior in the first occurrence search scenario for small patterns (Figure 88). Other than that, its performance in the rest of the cases resembles Boyer-Moore's, in that it offers search times well below the benchmark's average.

3.3.19 Zhu-Takaoka

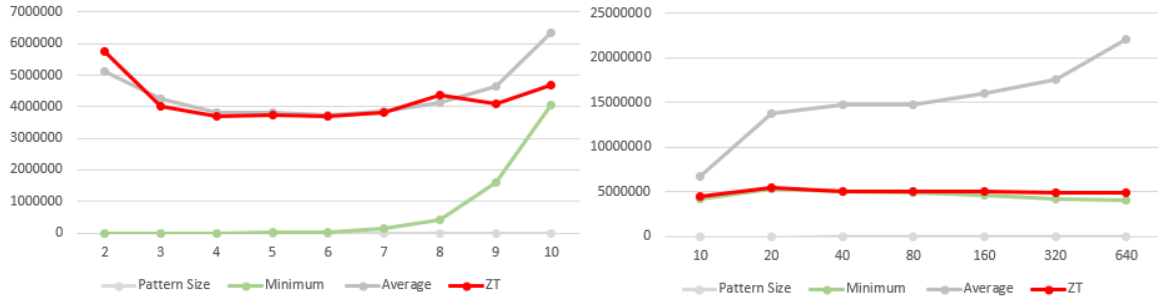

Figure 90: Zhu-Takaoka - results of finding the first occurrence of small patterns (left) and large patterns (right).

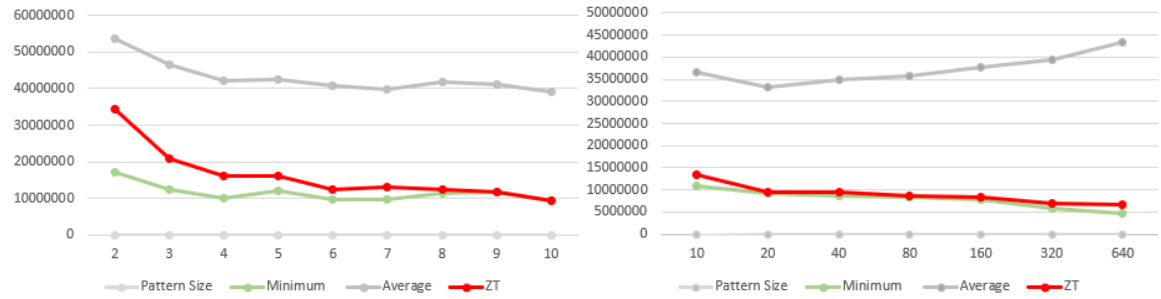

Figure 91: Zhu-Takaoka - results of finding all occurrences of small patterns (left) and large patterns (right).

Zhu Takaoka is another simplified implementation of the Boyer-Moore algorithm that uses the bad-character shift table at each mismatch for the two last consecutive characters of the window. One can notice it outperforms Boyer-Moore during large pattern searches as well as in all occurrences searches for small patterns (Figure 90, Figure 91). Additionally this algorithm appears to be the ideal option for first-occurrence search of patterns of size ranging between 10 and 80, as well as for all-occurrences search of patterns of size 9 and 160.

3.3.20 Berry-Ravindran

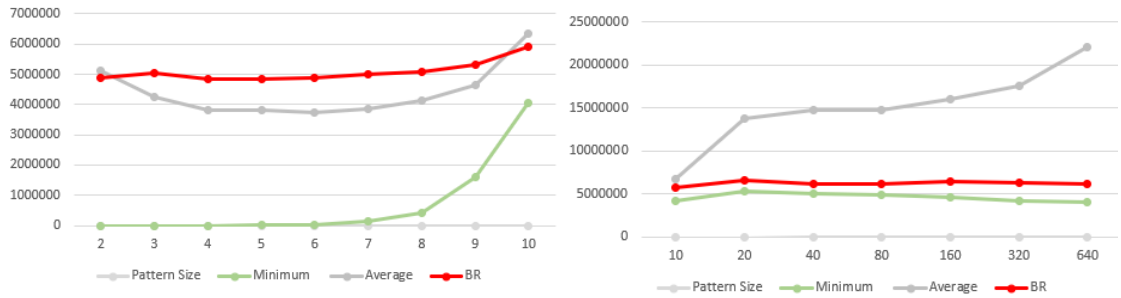

Figure 92: Berry-Ravindran - results of finding the first occurrence of small patterns (left) and large patterns (right).

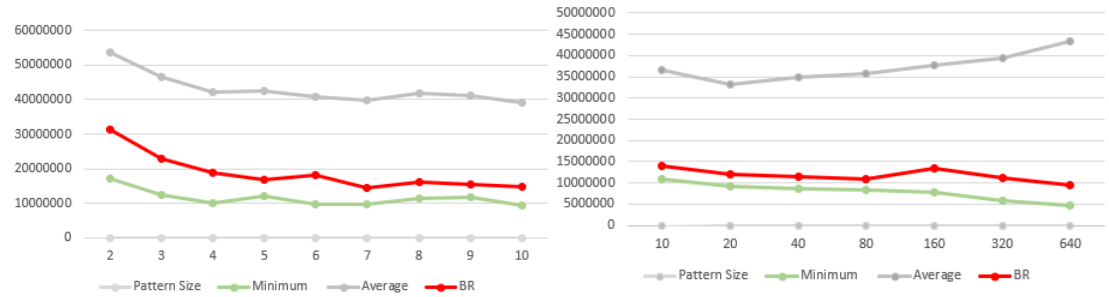

Figure 93: Berry-Ravindran - results of finding all occurrences of small patterns (left) and large patterns (right).

Berry-Ravindran is a hybrid of Quick Search and Zhu-Takaoka that uses the bad-character shift for the two consecutive characters right after the end of the window. It performs well in all benchmark cases except for first-occurrence matching of small patterns, where it displays times slower that the benchmark's average. However, in the rest cases it becomes more efficient as the pattern length increases.

3.3.21 Smith

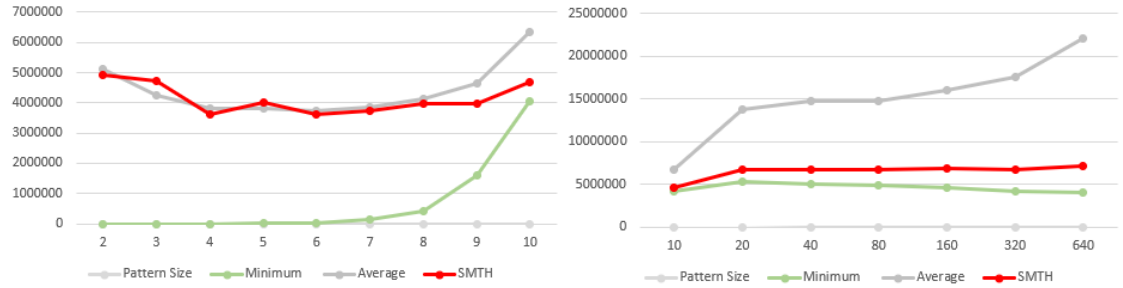

Figure 94: Smith - results of finding the first occurrence of small patterns (left) and large patterns (right).

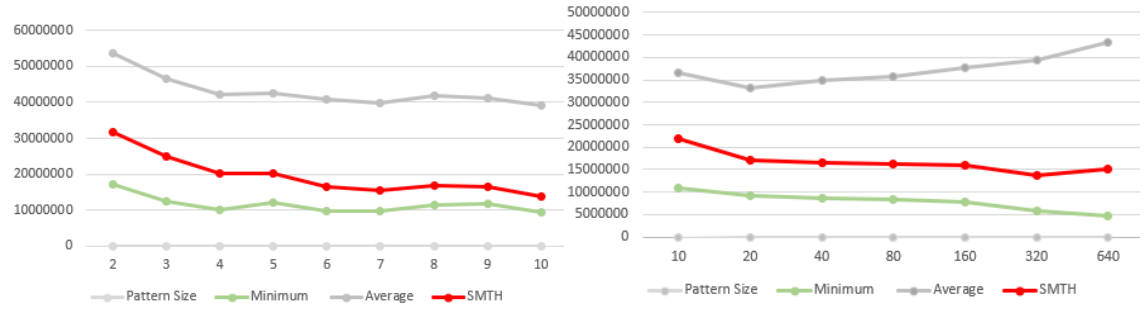

Figure 95: Smith - results of finding all occurrences of small patterns (left) and large patterns (right).

The Smith algorithm is a hybrid of Horspool and Quick Search takes the maximum shift of both bad-character shift tables. It seems to perform exactly the same as its two predecessors. Specifically, it is quite efficient with small sized patterns for either first-occurrence or all-occurrences searches.

3.3.22 Raita

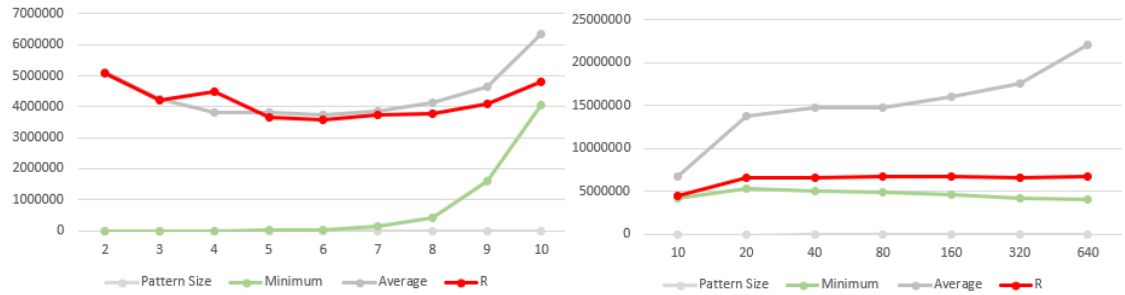

Figure 96: Raita - results of finding the first occurrence of small patterns (left) and large patterns (right).

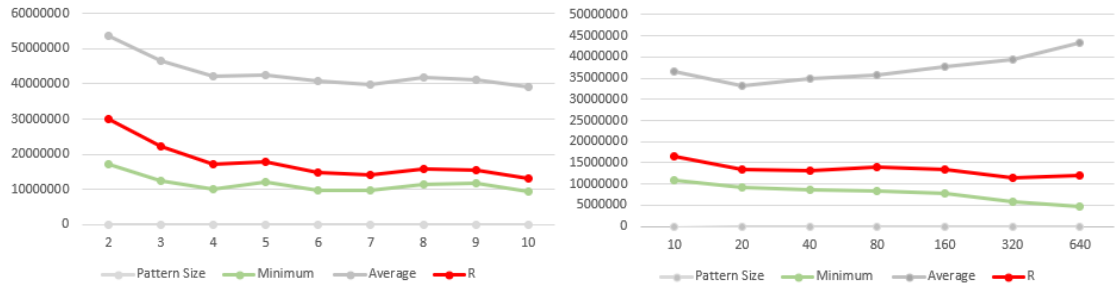

Figure 97: Raita - results of finding all occurrences of small patterns (left) and large patterns (right).

The Raita algorithm appears to be quite performant in any kind of search regardless of pattern size. Just like Horspool it utilizes only Boyer-Moore's bad-character table to perform the shifts, but also employs a specific order of comparisons during an attempt starting from the last character on to the first, the middle and finally the rest characters of the pattern. Notably, it performs, if not more, just as efficiently as Boyer-Moore using only one shift table and certainly outperforms Horspool that uses the same shift table. The algorithm seems to thrive particularly during searching all the occurrences of small patterns.

3.3.23 Reverse Factor

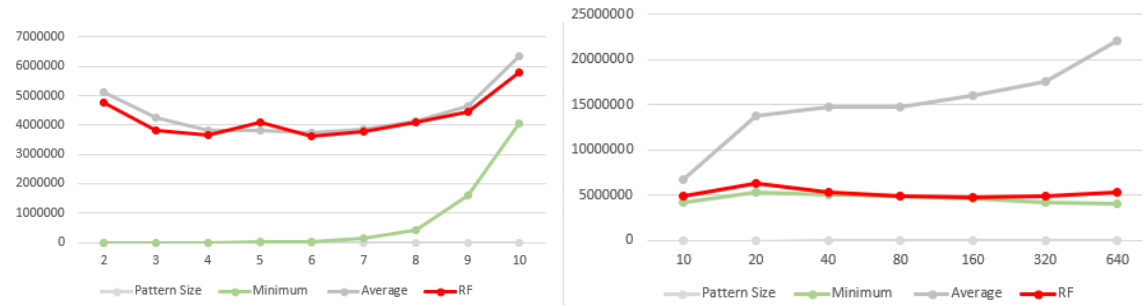

Figure 98: Reverse Factor - results of finding the first occurrence of small patterns (left) and large patterns (right).

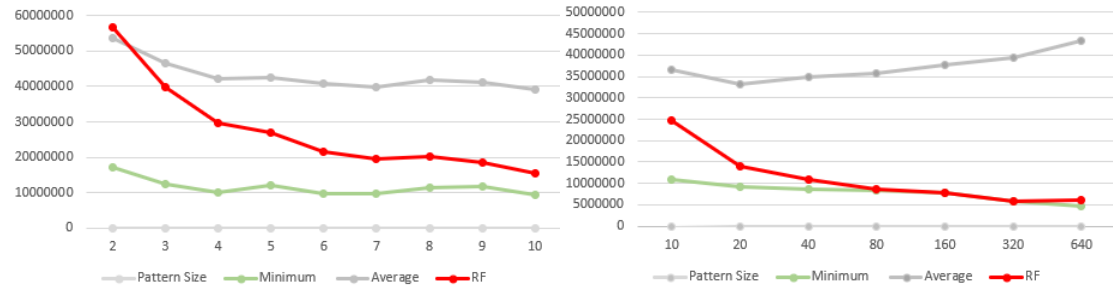

Figure 99: Reverse Factor - results of finding all occurrences of small patterns (left) and large patterns (right).

The Reverse Factor algorithm is a variant of Boyer-Moore implemented using a suffix automaton to store the most significant edges of the reversed pattern, thus avoiding using character comparisons, which are the most costly operations in a string search algorithm. This is reflected by its benchmark results, where for medium to large sized patterns (Figure 98, Figure 99) it performs the best searching times along with Zhu-Takaoka.

3.3.24 Turbo Reverse Factor

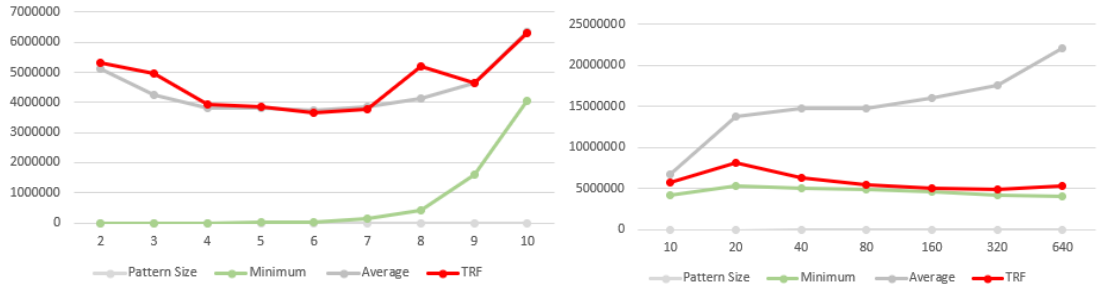

Figure 100: Turbo Reverse Factor - results of finding the first occurrence of small patterns (left) and large patterns (right).

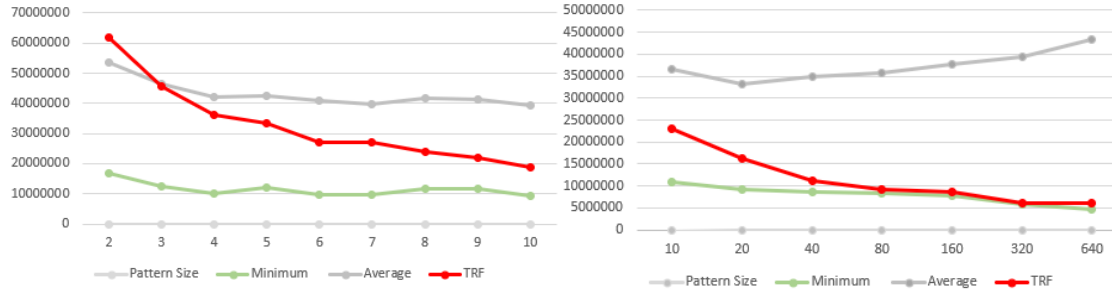

Figure 101: Turbo Reverse Factor - results of finding all occurrences of small patterns (left) and large patterns (right).

The Turbo Reverse Factor algorithm is quite performant in medium and large patterns of small alphabets (σ=4 here) and not so in cases with small patterns, just as its relative Reverse Factor. However although it is introduced as an amelioration to Reverse Factor there is no indication in the benchmark results of a notable improvement.

3.3.25 Forward DAWG Matching

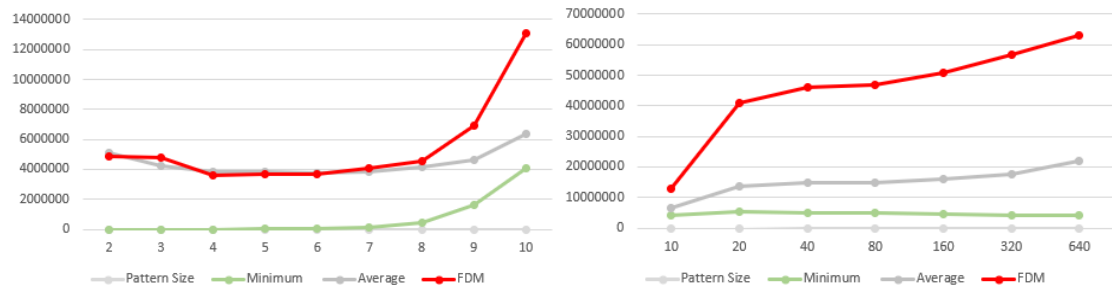

Figure 102: Forward DAWG Matching - results of finding the first occurrence of small patterns (left) and large patterns (right).

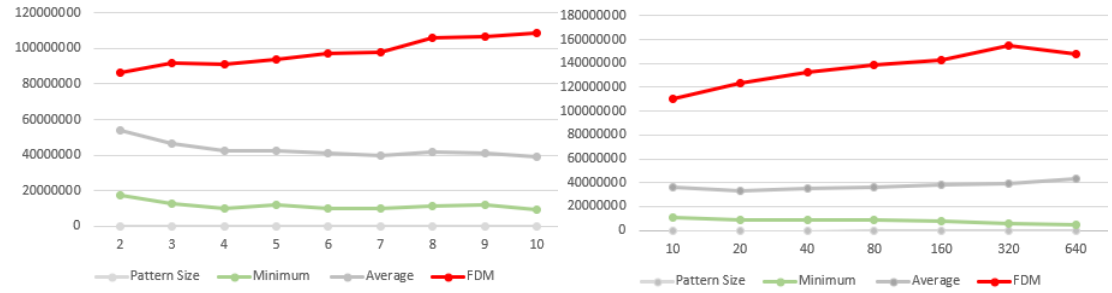

Figure 103: Forward DAWG Matching - results of finding all occurrences of small patterns (left) and large patterns (right).

Although the Forward DAWG Matching algorithm uses the suffix automaton that Reverse Factor introduced, it performs remarkably slow in almost all the cases except for the smaller patterns in a first-occurrence search.

3.3.26 Backward Nondeterministic DAWG Matching

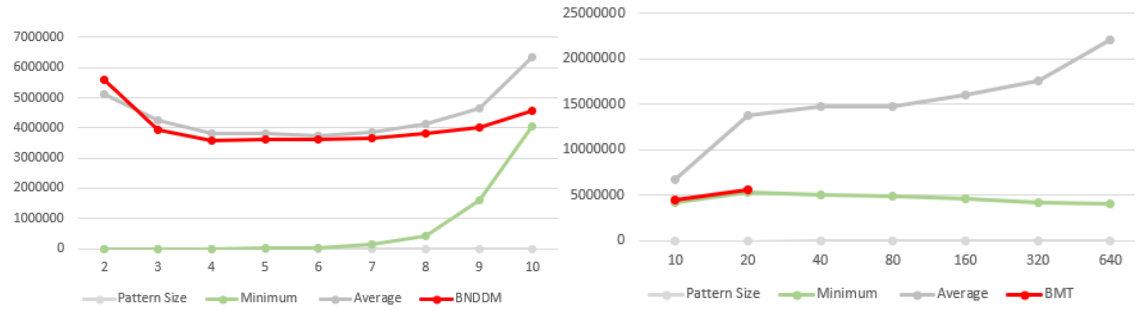

Figure 104: Backward Nondeterministic DAWG Matching - results of finding the first occurrence of small patterns (left) and large patterns (right).

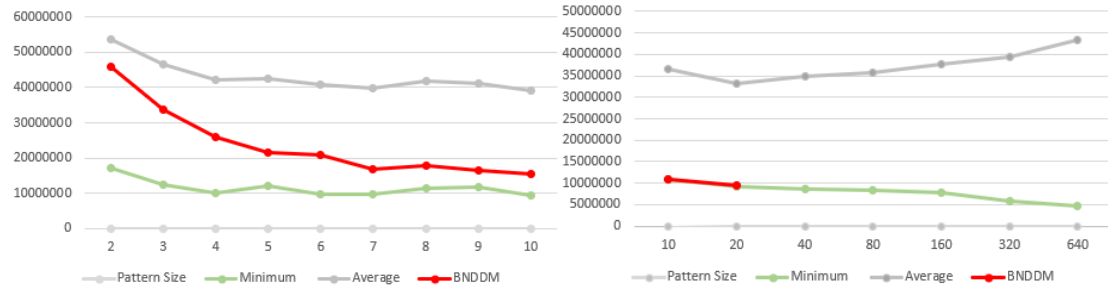

Figure 105: Backward Nondeterministic DAWG Matching - results of finding all occurrences of small patterns (left) and large patterns (right).

Since the Backward Nondeterministic DAWG Matching algorithm relies on a bitwise mask, where each bit corresponds to a character of the pattern, in order to store the current search state, and since the mask has to be stored as a number for computational efficiency, it is actually constricted by the word size of the computer (in this case 32). Meaning the algorithm can be used only for patterns of length up to the word size. Regarding its performance, one can notice that the algorithm approaches near to optimal efficiency in any search up until the word size pattern length, which can be attributed to the usage of bitwise operations to move between search states, similarly to the Shift Or algorithm.

3.3.27 Backward Oracle Matching

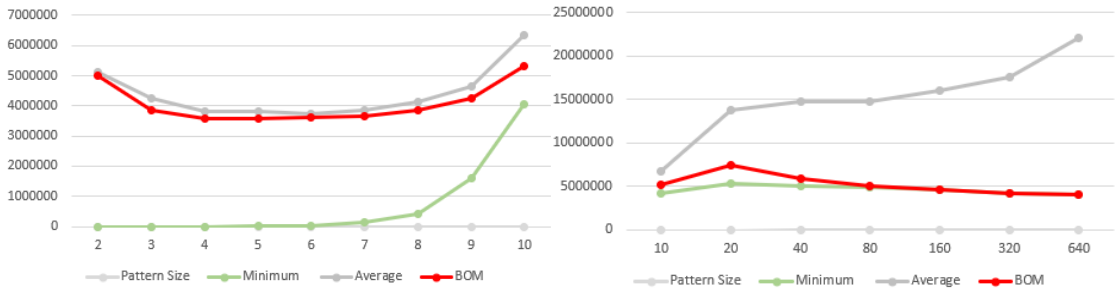

Figure 106: Backward Oracle Matching - results of finding the first occurrence of small patterns (left) and large patterns (right).

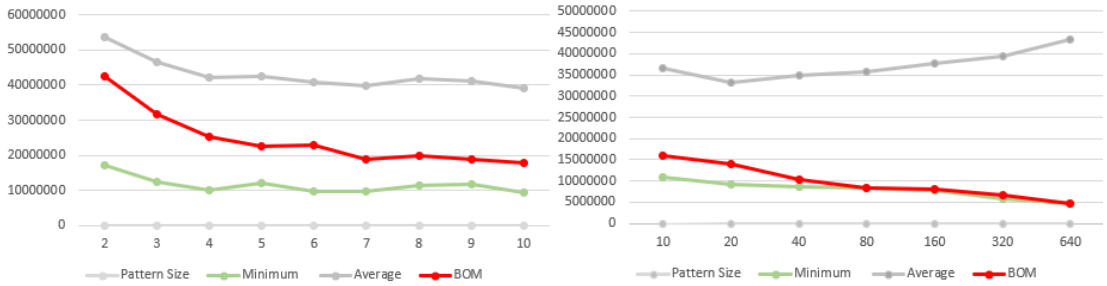

Figure 107: Backward Oracle Matching - results of finding all occurrences of small patterns (left) and large patterns (right).

The Backward Oracle matching is a variant of the Reverse Factor algorithm. It performs relatively efficiently, well below average, for small patterns. However just as Reverse Factor it thrives in large patterns, as it logged the best performance for patterns of length greater than 40, outperforming Reverse Factor for sizes over 320 where the latter slightly deteriorates.

3.3.28 Galil-Seiferas

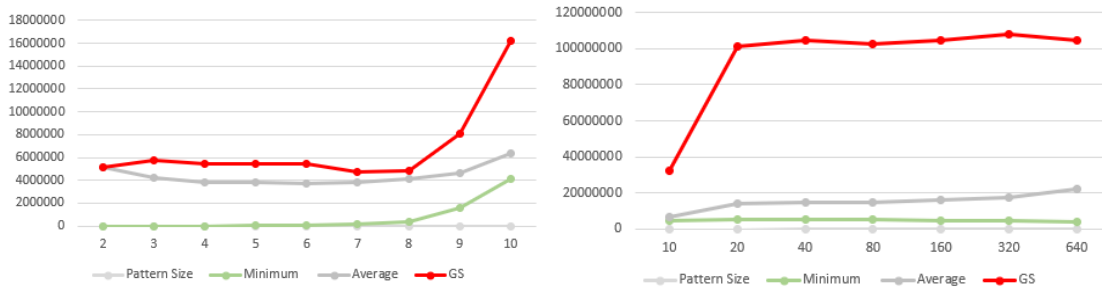

Figure 108: Galil-Seiferas - results of finding the first occurrence of small patterns (left) and large patterns (right).

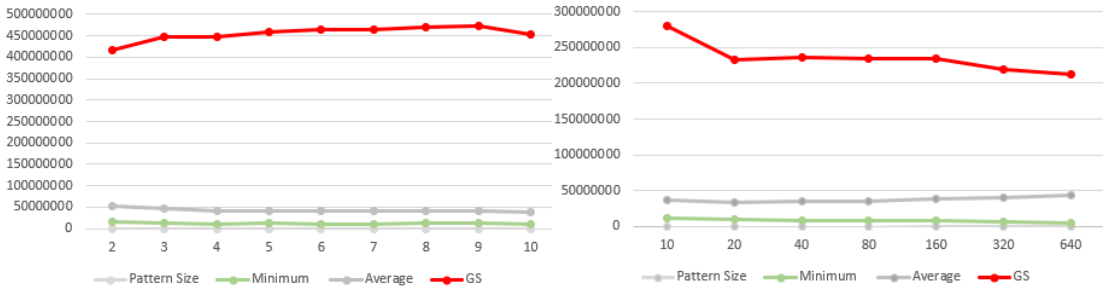

Figure 109: Galil-Seiferas - results of finding all occurrences of small patterns (left) and large patterns (right).

One can notice that the Galil-Seiferas algorithm is probably the least performant of the algorithms included in this benchmark. Notably, its performance is quite slow when searching for all occurrences of a pattern (Figure 109) as well as when looking for the first occurrence of a large pattern (Figure 108). Finally it manages to keep up with the average benchmark performance only for the smaller pattern sizes ranging from 2 to 8 (Figure 108).

3.3.29 Two Way

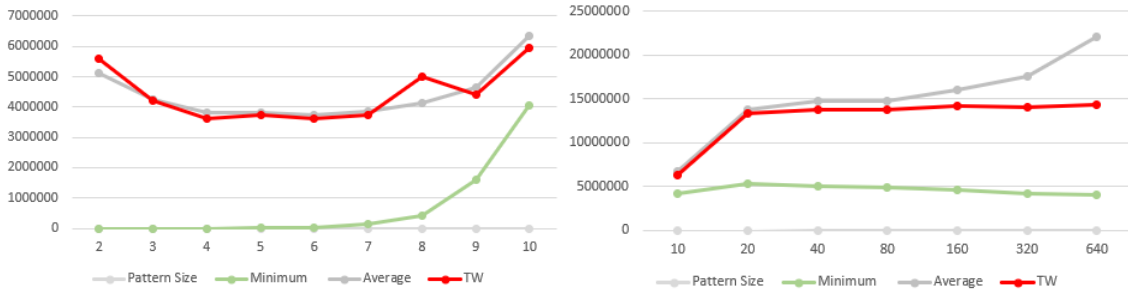

Figure 110: Two Way - results of finding the first occurrence of small patterns (left) and large patterns (right).

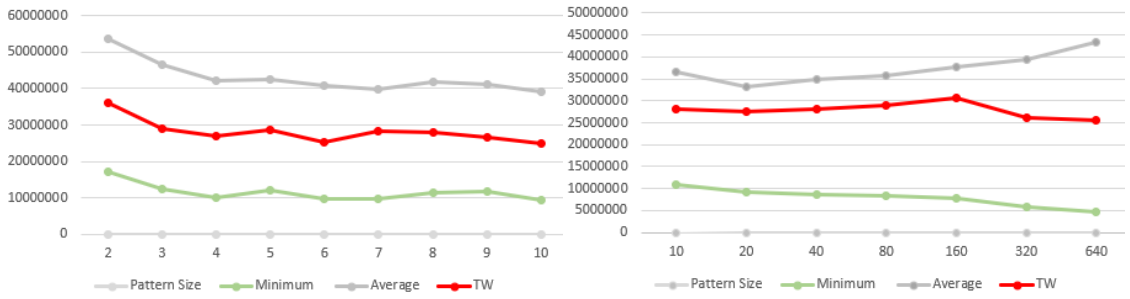

Figure 111: Two Way - results of finding all occurrences of small patterns (left) and large patterns (right).

The Two Way algorithm ranks rather averagely concerning first-occurrence searches (Figure 110), although it behaves well for larger patterns (>160). Additionally, its strength seems to lie in all-occurrences searches for small patterns ranging from 2 to 10, as well as for larger patterns (> 160).

3.3.30 String Matching on Ordered Alphabets

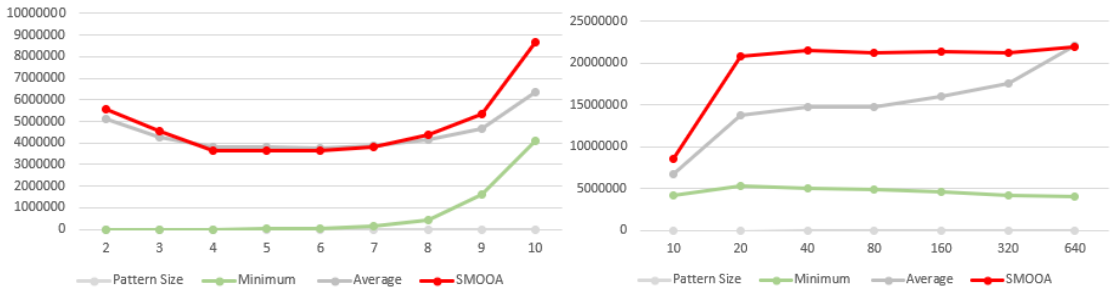

Figure 112: String Matching on Ordered Alphabets - results of finding the first occurrence of small patterns (left) and large patterns (right).

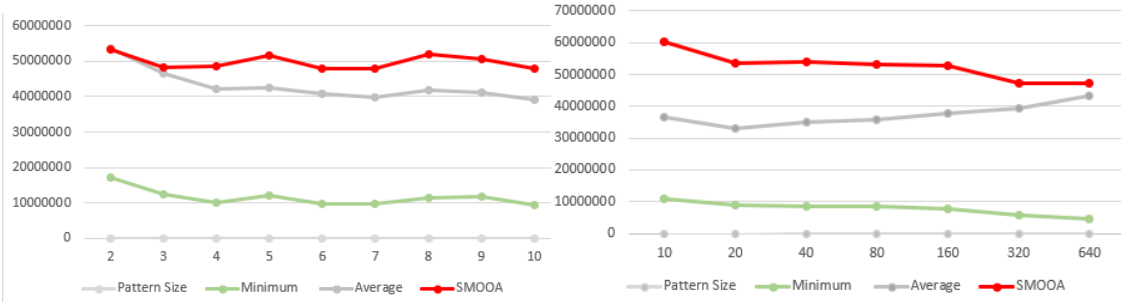

Figure 113: String Matching on Ordered Alphabets - results of finding all occurrences of small patterns (left) and large patterns (right).

Concerning first-occurrence searches, the String Matching on Ordered Alphabets algorithm performs averagely for smaller patterns (2-9) and for very large patterns

(>320), and not so efficiently for medium to large sized patterns (10-320) (Figure 112). Similarly, when searching for all pattern occurrences, it performs near the benchmark's average for very small patterns (2-4) and for very large patterns (>320) while it deteriorates for medium sized patterns (Figure 113).

3.3.31 Optimal Mismatch

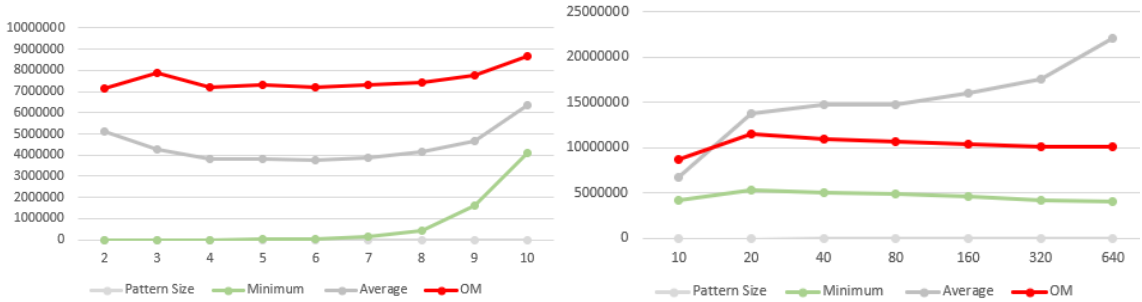

Figure 114: Optimal Mismatch - results of finding the first occurrence of small patterns (left) and large patterns (right).

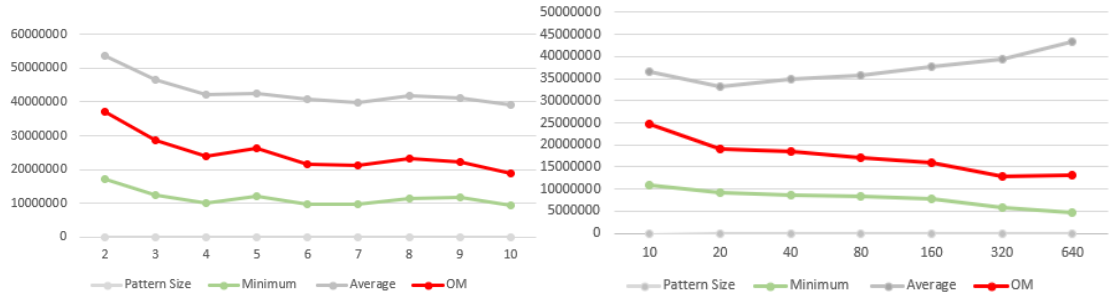

Figure 115: Optimal Mismatch - results of finding all occurrences of small patterns (left) and large patterns (right).

The Optimal Mismatch algorithm seems to be efficient for looking for all occurrences of a pattern regardless of pattern size and appears to execute faster as the pattern grows (Figure 115). Regarding searching for the first occurrence of a text, the algorithm achieves times above benchmark average for small patterns, and well below average for larger patterns (Figure 114).

3.3.32 Maximal Shift

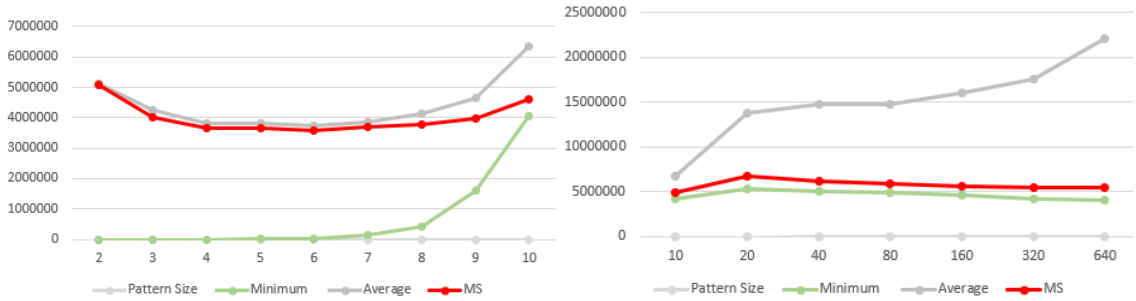

Figure 116: Maximal Shift - results of finding the first occurrence of small patterns (left) and large patterns (right).

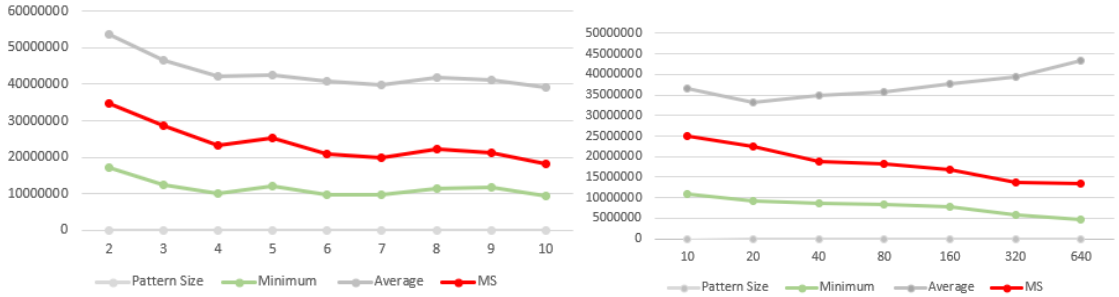

Figure 117: Maximal Shift - results of finding all occurrences of small patterns (left) and large patterns (right).

The Maximal Shift algorithm is quite efficient when searching for the first occurrence of a pattern, especially for large ones, where it almost approaches the benchmark's best performances (Figure 116). It ranks better than Optimal Mismatch in this case as it starts comparing the characters starting from the one that offers the largest shift and not the one that has the largest frequency (the impact of which is negated by the small alphabet size in this benchmark). The algorithm also performs relatively well for matching all the occurrences of a pattern regardless of size (Figure 117).

3.3.33 Skip Search

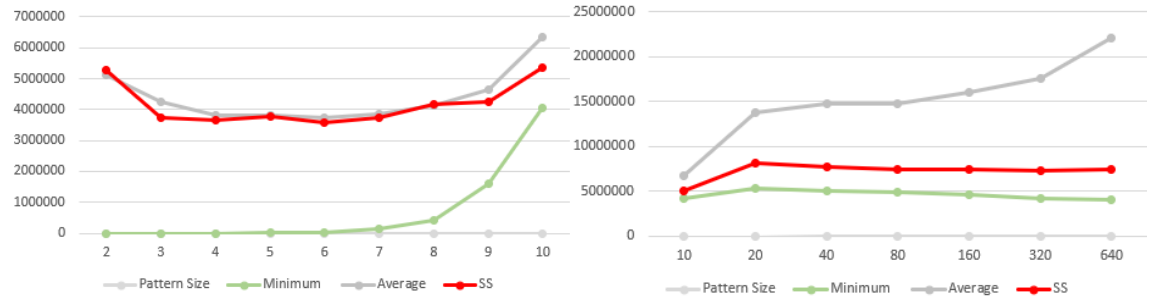

Figure 118: Skip Search - results of finding the first occurrence of small patterns (left) and large patterns (right).

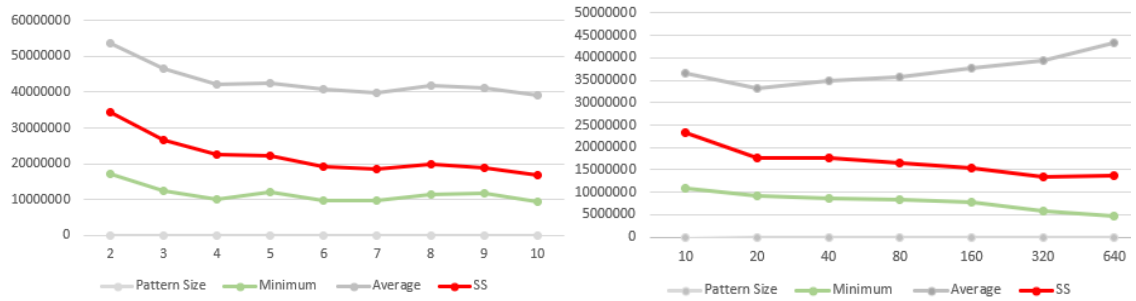

Figure 119: Skip Search - results of finding all occurrences of small patterns (left) and large patterns (right).

The Skip Search algorithm appears to be quite efficient regarding searching for all occurrences of a pattern, regardless of size (Figure 119). Furthermore, its performance seems to improve as the pattern length increases. The same can be said in the case of first-occurrence pattern searching, particularly for large patterns (Figure 118). Since the alphabet size is small ($\sigma = 4$), each character's position bucket is rarely empty, meaning that rarely there will be unused memory references.

3.3.34 KMP Skip Search

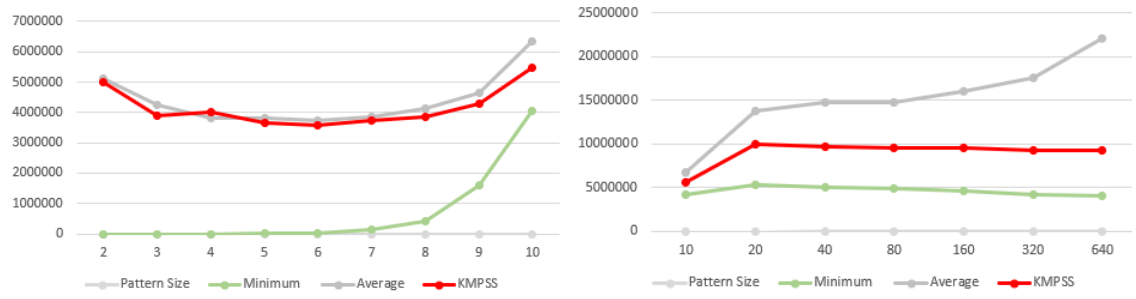

Figure 120: KMP Skip Search - results of finding the first occurrence of small patterns (left) and large patterns (right).

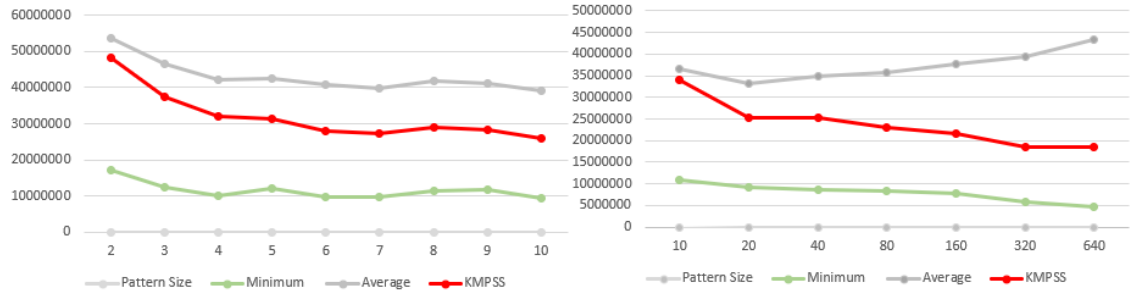

Figure 121: KMP Skip Search - results of finding all occurrences of small patterns (left) and large patterns (right).

The fact that KMP Skip Search is an improvement of the Skip Search algorithm, does not reflect in the benchmark results. Specifically, the algorithm seems notably slower when searching for large patterns and when performing an all-occurrences search with small patterns (Figure 121). However it is still a quite efficient algorithm ranking considerably better that the benchmark's average performance.

3.3.35 Alpha Skip Search

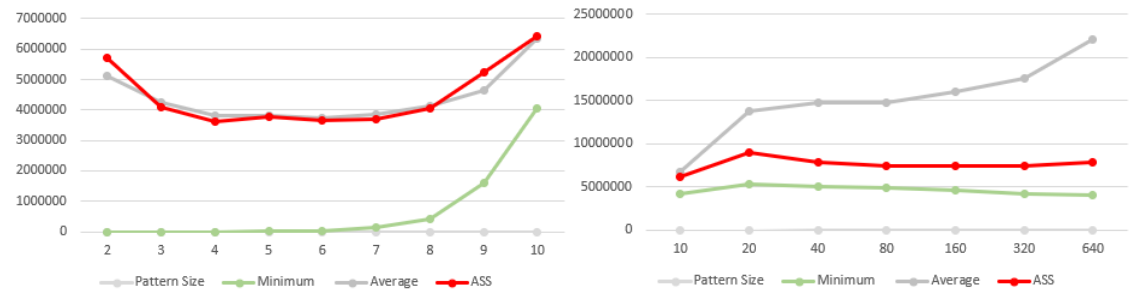

Figure 122: Alpha Skip Search - results of finding the first occurrence of small patterns (left) and large patterns (right).

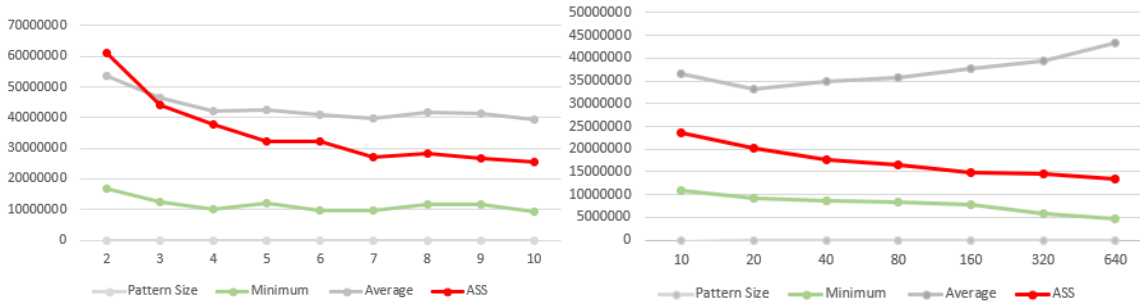

Figure 123: Alpha Skip Search - results of finding all occurrences of small patterns (left) and large patterns (right).

The Alpha Skip Search is another yet amelioration of Skip Search. One can notice that this version performs slightly better than KMP Skip Search for medium to very large patterns, while it seemingly deteriorates for the smaller ones (Figure 122, Figure 123).

3.4 Collective results

3.4.1 First occurrence – Small Patterns

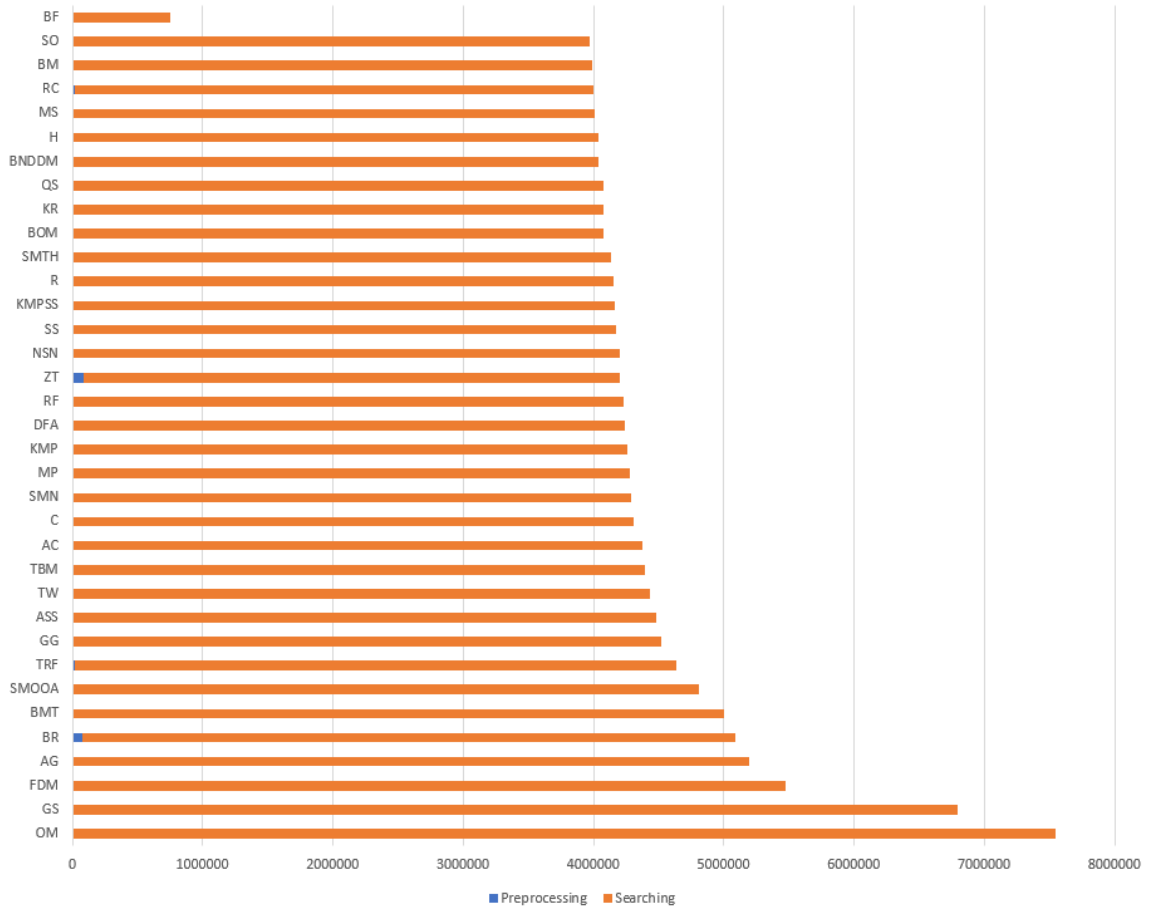

Figure 124: Comparative results for searching the first occurrence of a small pattern.

Regarding searching for the first occurrence of a small pattern, one can notice, that the Brute Force (BF) algorithm is a clear winner. This can be attributed to its lack of preprocessing phase and computational simplicity as well as the length of the patterns and the small size of the alphabet ($\sigma = 4$), which appear to negate the impact from whatever preprocessing computation from the rest of the algorithms. Notably, the most of the other algorithms seem to perform pretty close to each other, the frontrunners of which are Shift Or (SO) and Backward Nondeterministic DAWG Matching (BNDDM), which utilize bitwise operations, Boyer-Moore (BM) and its variants Reverse Colussi (RC), Maximal Shift (MS), Horspool (H) and Quick Search (QS), the hash function string search algorithm Karp-Rabin (KR) and Backward Oracle Matching (BOM). The least performant algorithms seem to be String Matching on Ordered Alphabets (SMOOA), Forward DAWG Matching (FDM), Galil-Seiferas (GS) as well as some of the variants of

Boyer-Moore: Tuned Boyer-Moore (BMT), Berry-Ravindran (BR), Apostolico-Giancarlo (AG) and Optimal Mismatch (OM).

3.4.2 First occurrence – Large Patterns

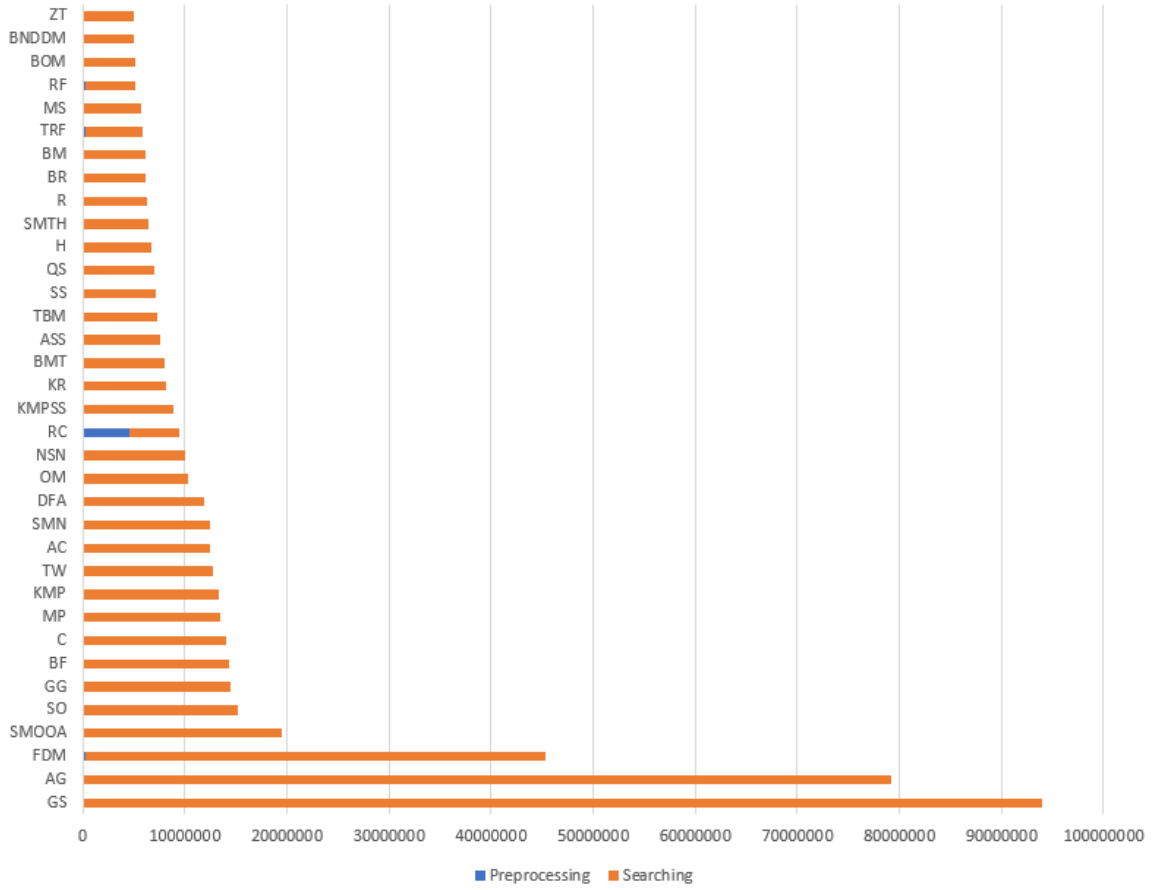

Figure 125: Comparative results for searching the first occurrence of a large pattern.

Contrary to the previous results for small patterns, searching the first occurrence for large patterns does not seem to have a clear winner. Again, one can distinguish classes of performances, the algorithms of which have relatively similar performances. The algorithms that appear to prevail are Zhu-Takaoka (ZT), Backward Nondeterministic DAWG Matching (BNDDM), Backward Oracle Matching (BOM) and Reverse Factor (RF). It has to be mentioned that BNDDM is restricted by the word size of the computer (here 32) so the performance shown above is only for pattern sizes 10 and 20. Remarkably, Brute Force algorithm, which prevailed for searching first-occurrences of small patterns, is now spotted in the at the bottom end of the second half of the

performance rankings. Regarding the least efficient algorithms, those are String Matching on Ordered Alphabets (SMOOA), Forward DAWG Matching (FDM), Apostolico-Giancarlo (AG) and Galil-Seiferas (GS).

3.4.3 All occurrences – Small Patterns

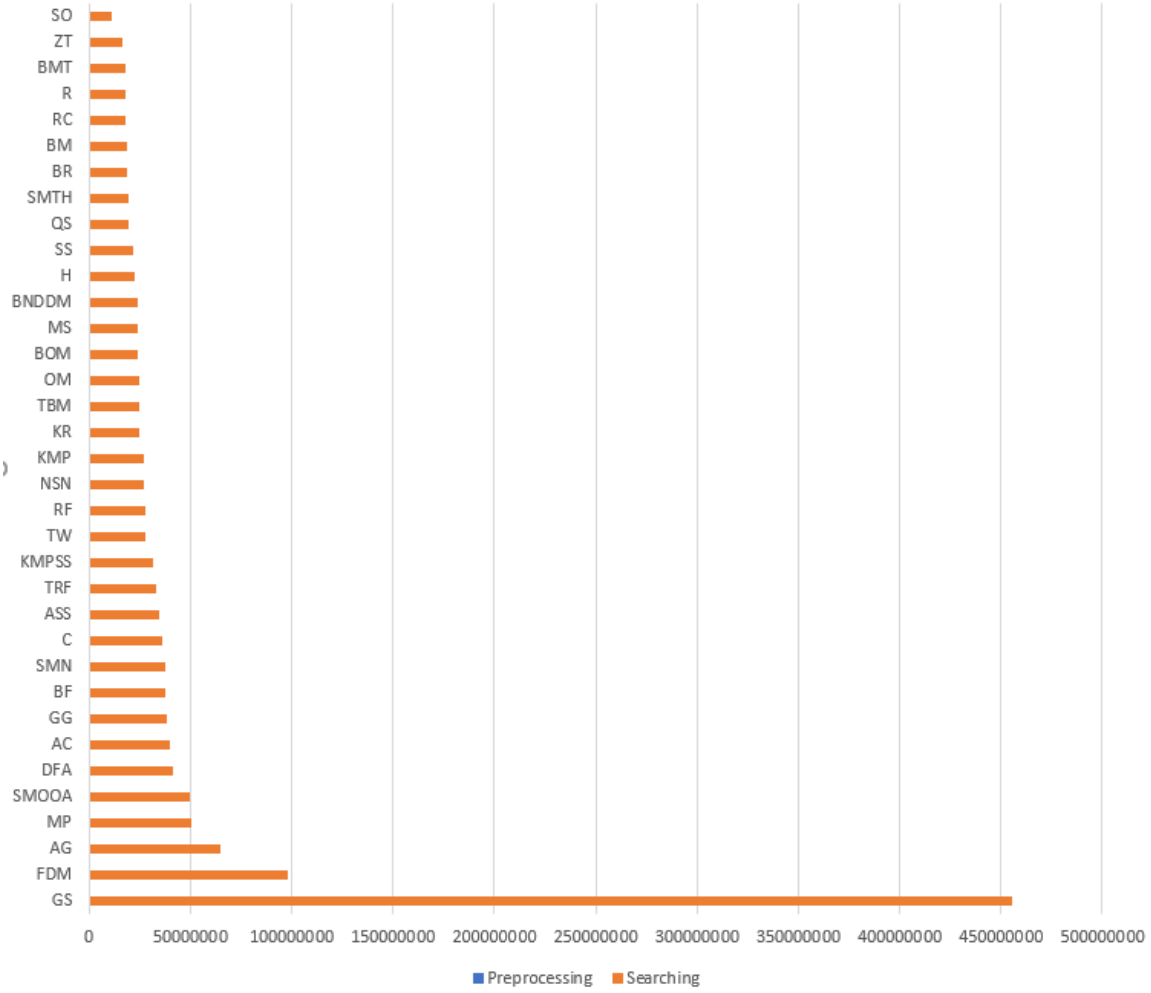

Figure 126: Comparative results for searching all the occurrences of a small pattern.

In the case of finding all occurrences of a small pattern seem to perform similarly except for a few outliers. The most efficient algorithm appears to be Shift Or (SO) followed closely by Zhu-Takaoka (ZT), Tuned Boyer-Moore (BMT), Raita (R) and Reverse Colussi (RC). Notably, Boyer-Moore and all of its variants (Raita (R), Turbo

Boyer-Moore (TBM), Reverse Colussi (RC), Horspool (H), Quick Search (QS), Smith (SMTH), Tuned Boyer-Moore (BMT), Zhu-Takaoka(ZT), Berry-Ravindran (BR), Optimal Mismatch (OM), Maximal Shift (MS)) except from Apostolico-Giancarlo (AG) are ranked in the first half of the performance comparison while Morris-Pratt and its variants (Knuth-Morris-Pratt (KMP), Colussi (C), Galil-Giancarlo (GG), Apostolico-Crochemore (AC)) are placed lower, in the second half. The worst performances have been logged by Galil-Seiferas (GS), Forward DAWG Matching (FDM) and Apostolico-Giancarlo (AG).

3.4.4 All occurrences – Large Patterns

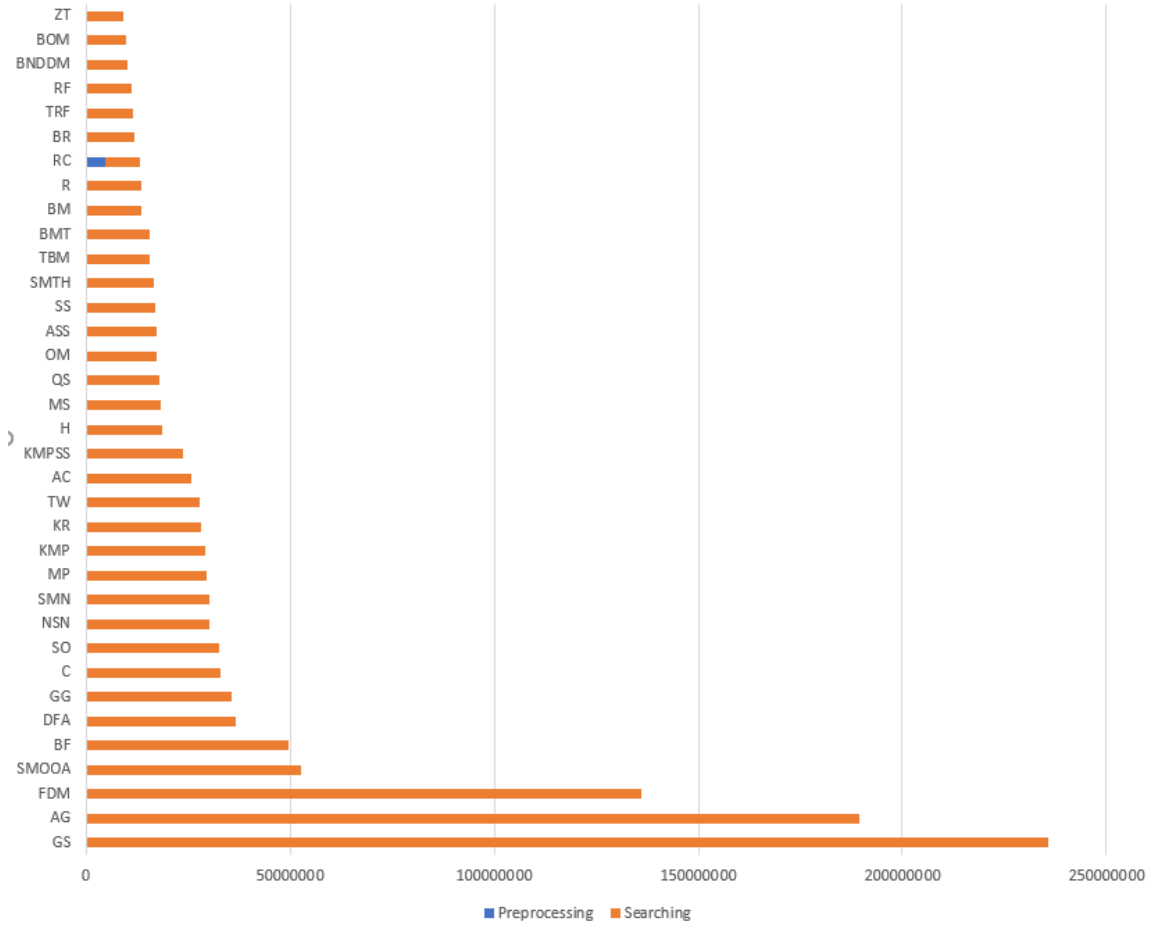

Figure 127: Comparative results for searching all the occurrences of a large pattern.

The algorithms that prevail in this case are Zhu-Takaoka (ZT), backward Oracle Matching (BOM), Backward Nondeterministic DAWG Matching (BNDDM) and Reverse Factor (RF). On the other side, the less efficient algorithms include Galil-Seiferas (GS), Apostolico-Giancarlo (AG), Forward DAWG Matching (FDM), String Matching on Ordered Alphabets (SMOOA) and Brute Force (BF). Notably, similarly to the case of the smaller patterns the algorithms of the Boyer-Moore family prevail over those of the Morris-Pratt family.

3.5 Collective results grouped by order of comparison

3.5.1 First occurrence – Small Patterns

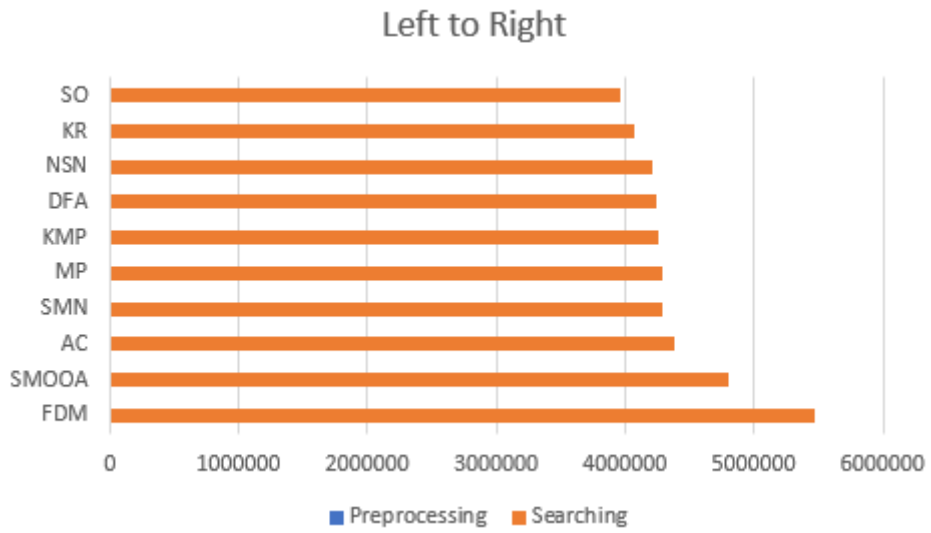

Figure 128: Searching the first occurrence of small patterns by comparing from left to right.

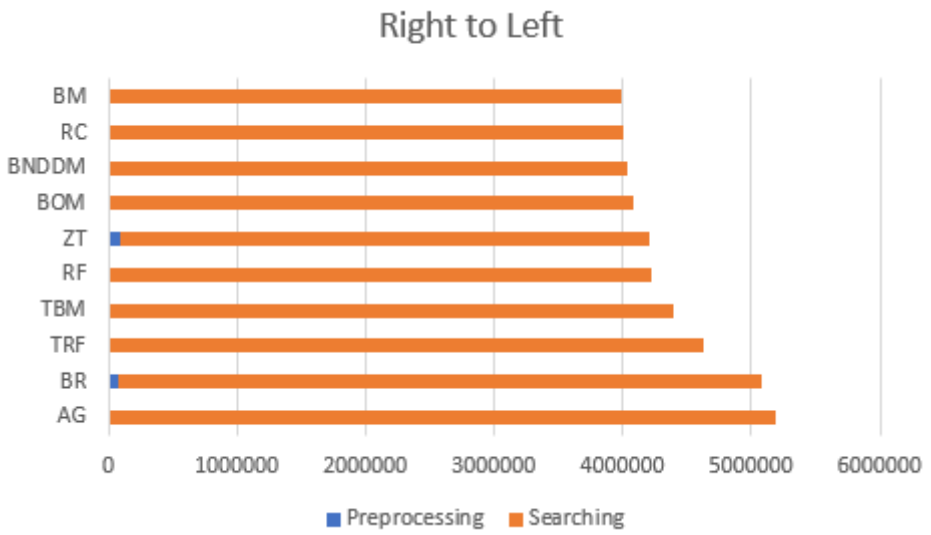

Figure 129: Searching the first occurrence of small patterns by comparing from right to left.

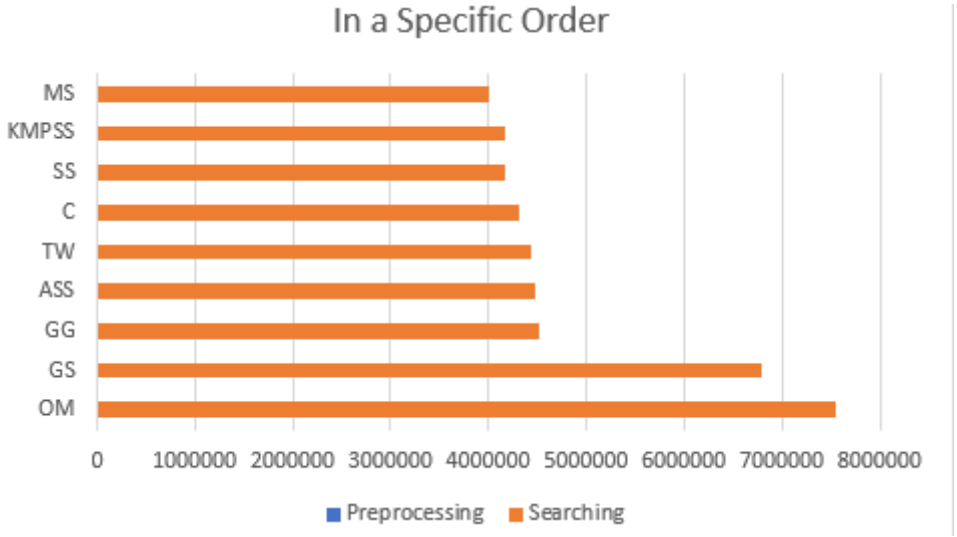

Figure 130: Searching the first occurrence of small patterns by comparing in a specific order.

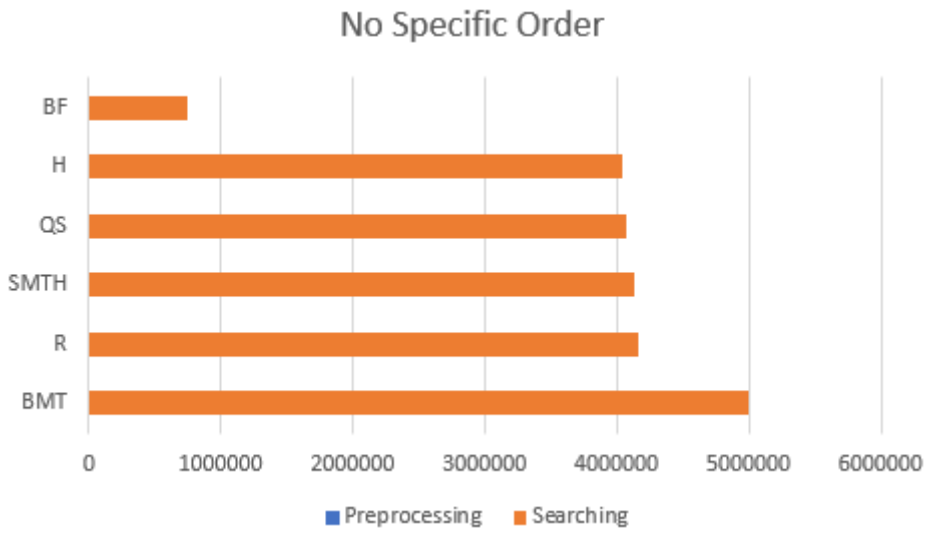

Figure 131: Searching the first occurrence of small patterns by comparing in no specific order.

When searching for the first occurrence of small patterns, the size of which ranges between 2 and 10, the best algorithms in each category of order of character comparison are the following:

- Left to right: Shift Or
- Right to left: Boyer-Moore
- In a specific order: Maximal Shift
- In no specific order: Brute Force

3.5.2 First occurrence – Large Patterns

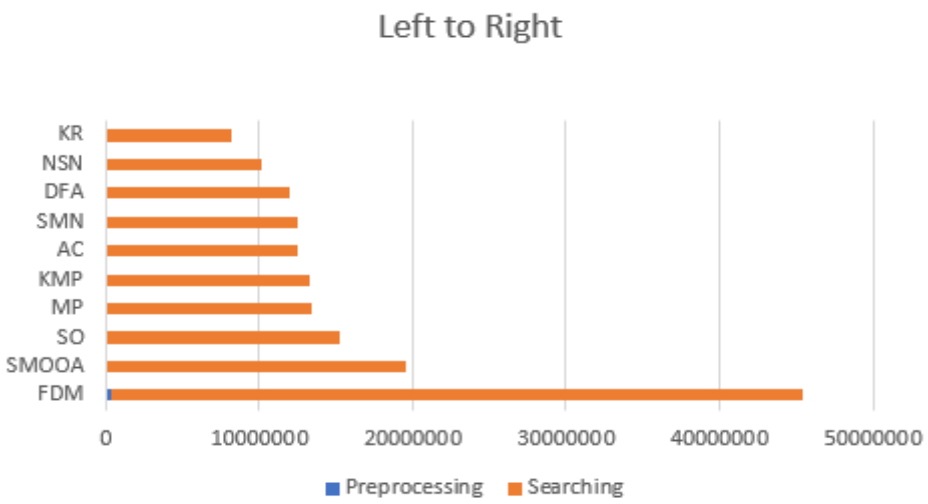

Figure 132: Searching the first occurrence of large patterns by comparing from left to right.

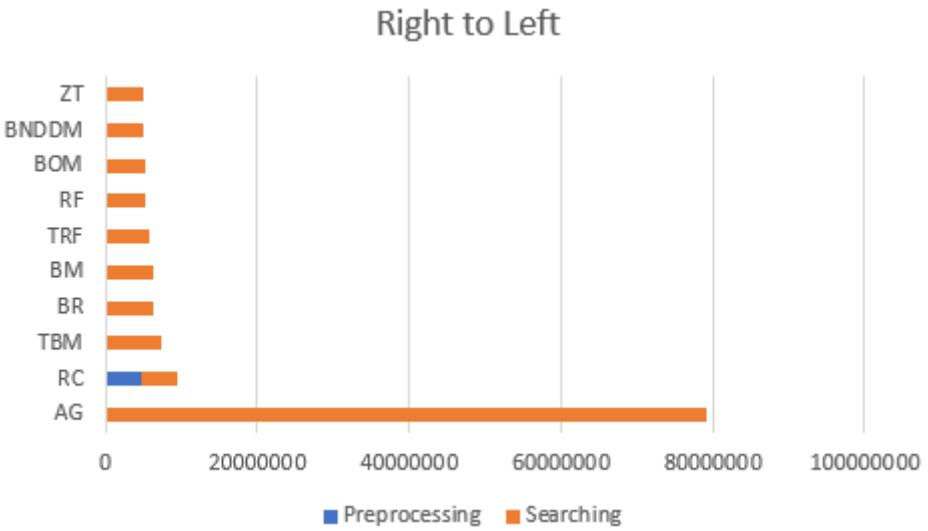

Figure 133: Searching the first occurrence of large patterns by comparing from right to left.

Figure 134: Searching the first occurrence of large patterns by comparing in a specific order.

Figure 135: Searching the first occurrence of large patterns by comparing in no specific order.

When searching for the first occurrence of large patterns, the size of which ranges between 10 and 640, the best algorithms in each category of order of character comparison are the following:

- Left to right: Karp-Rabin
- Right to left: Zhu-Takaoka
- In a specific order: Maximal Shift
- In no specific order: Raita

3.5.3 All occurrences – Small Patterns

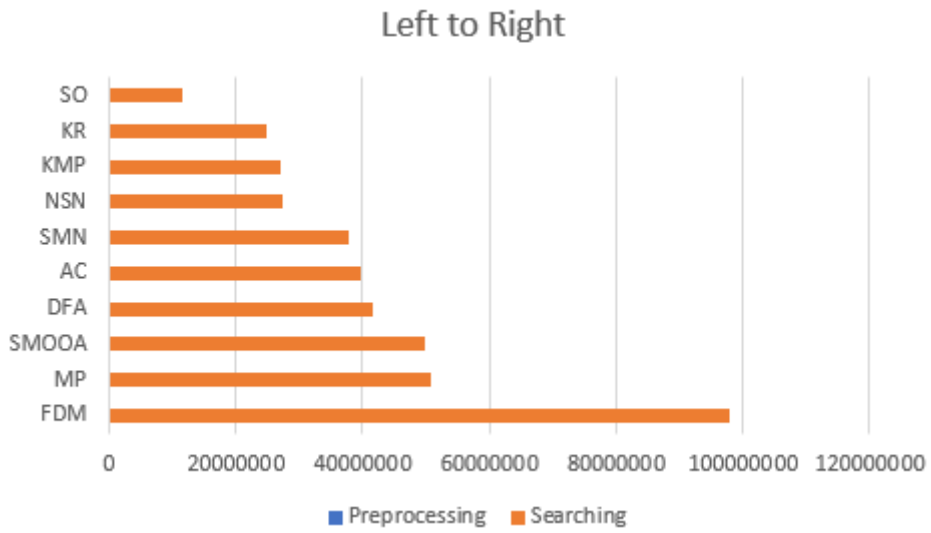

Figure 136: Searching all occurrences of small patterns by comparing from left to right.

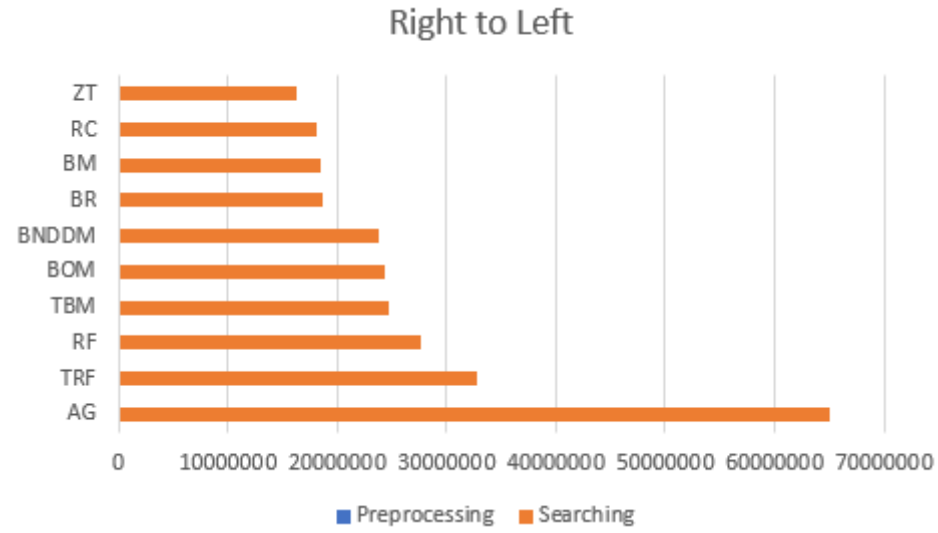

Figure 137: Searching all occurrences of small patterns by comparing from right to left.

Figure 138: Searching all occurrences of small patterns by comparing in a specific order.

Figure 139: Searching all occurrences of small patterns by comparing in no specific order.

When searching for all occurrences of small patterns, the size of which ranges between 2 and 10, the best algorithms in each category of order of character comparison are the following:

- Left to right: Shift Or
- Right to left: Zhu-Takaoka
- In a specific order: Skip Search
- In no specific order: Tuned Boyer-Moore

3.5.4 All occurrences – Large Patterns

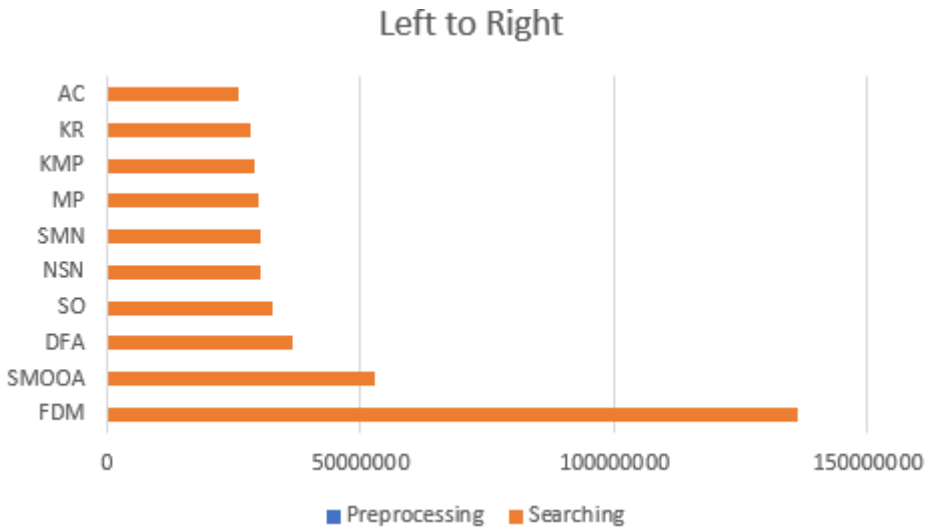

Figure 140: Searching all occurrences of large patterns by comparing from left to right.

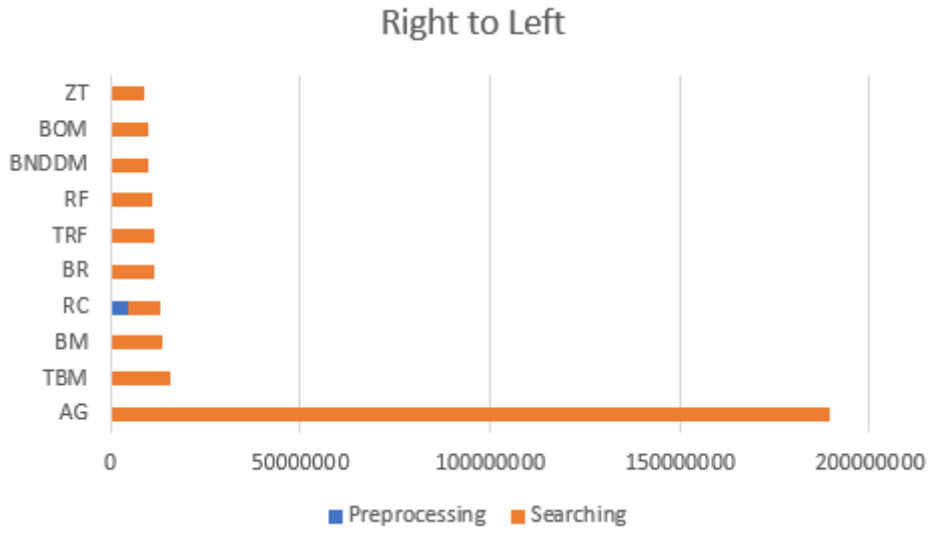

Figure 141: Searching all occurrences of large patterns by comparing from right to left.

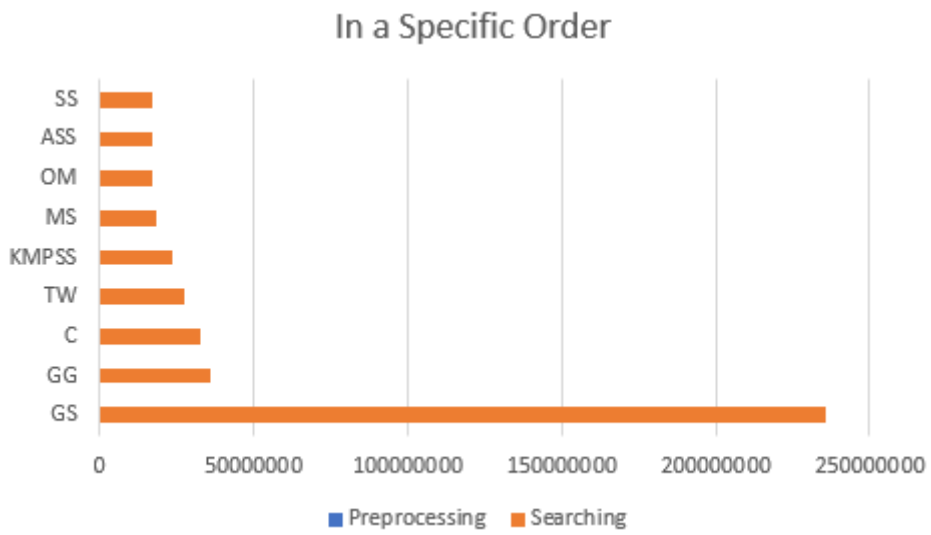

Figure 142: Searching all occurrences of large patterns by comparing in a specific order.

Figure 143: Searching all occurrences of large patterns by comparing in a specific order.

When searching for all occurrences of large patterns, the size of which ranges between 10 and 640, the best algorithms in each category of order of character comparison are the following:

- Left to right: Apostolico-Crochemore
- Right to left: Zhu-Takaoka
- In a specific order: Skip Search
- In no specific order: Raita

4 Visualization

4.1 Basic Description

This section offers an insight in the visualization software project regarding the string matching algorithms included in this thesis. The end program offers a simple visual representation of the string searching process for a specified algorithm (Figure 144). It was implemented using the web technologies HTML, CSS, JavaScript and the jQuery library, due to their flexibility in visualization purposes and the simplicity they offer in embedding their functionality in web pages.

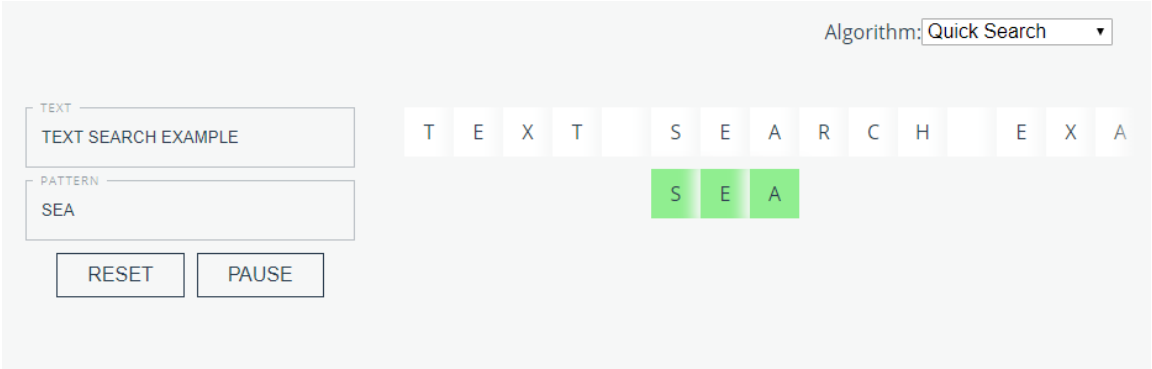

Figure 144: The visualization program's interface.

The interface of the visualization program consists of a header on top, a control panel to the left and a display panel to the right. The header contains a dropdown list that allows the selection of the algorithm, the functionality of which will be visualized. The control panel contains two input fields, one for the text and one for the pattern. It also offers two buttons; a reset button that initializes the used inputs in the display panel, and a start button that triggers the animation. The latter, once pressed changes to a pause button that, as it suggests, pauses the animation and reverts back to a start button and so on. The display panel is the view of the visualization process. It contains two rows of letter panels, the upper for the text characters and the lower for the pattern ones. The key actions that are visualized correspond to the following actions:

- Character match, denoted by changing the background to green of the letters that were matched between text and pattern.

- Character mismatch, denoted by changing the background to red of the letters that were mismatched between text and pattern.
- Window shift, indicated by sliding the pattern character row to the new window position.
- Pattern found, indicated by applying green background to all pattern letters.
- Pattern not found, indicated by applying red background to all pattern letters.

To perform the animations, contrary to what is displayed, the program does not execute the string search and the visualization in parallel. That would require severe interventions in each algorithm and applying a lot of logic in several points of their implementations resulting in complicated, unmaintainable and error-prone code. Instead, the key idea efficiently enabling the animation relies on first completing the execution of the algorithm, making it produce simple string queries that indicate one of the key actions to-be-visualized, mentioned above, and finally pass those queries in the sequence they were logged to the animation routine.

Specifically, at first, the string search algorithms are ported in JavaScript, which is relatively simple since it comes from the same programming language family as C and Java. Then, at key points the algorithm produces simple, predefined string queries (i.e. match 10 3) that describe the action to be visualized, and stores them in an array that is retrievable with a method (i.e. getQueries). All this happens when the *reset* button is pressed. When the *start* button is pressed the program traverses through the queries and visualizes them in the display panel.

4.2 Technical Details

As mentioned above the visualization suite was implemented in HTML, CSS, JavaScript and jQuery. In order to embed its functionality in a web page, the prerequisites are for the HTML file to load jQuery and the *script.js* file, the latter with a *type = "module"* attribute causing it to be treated as a module, and have one or more *div* container elements with the class *esmajs.* The program will then dynamically construct the previously mentioned structure in each one of those *div* elements. The *script.js* file is stored along with the other scripts in the web page project directory *js/Visualization/* except for the algorithm JavaScript files that are stored in the *js/Algorithms* directory.

The functionality of the whole visualization process starts in the *script.js* file, and is organized in various script files that act as modules by exporting their necessary functionality. To achieve separate visualization construction and control for different container elements, a very simplified version of the view-controller model was attempted to be created, by creating a 'view' for each container that constructs the appropriate structure and assigning it with a 'controller' that is restricted in managing only its functionality.

A brief description of the JavaScript modules used in this project follows below.

4.2.1 script.js

The script where it triggers the whole dynamic procedure of constructed the basic structure for any *div* containers that have the class *esmajs* and adding the visualization functionality. This script must be loaded in the HTML file of the web page using the script tag with the *type="module"* attribute. Practically it contains an anonymous function call that executes the *ESMAJS* function of the *esmajs.js* module.

4.2.2 esmajs.js

This module exports the *ESMAJS* function. Practically it locates every *div* element in the HTML file with the target class *esmajs* and for each one of them, it creates a 'view' by creating a *View* object (*view.js*) and a 'controller' by creating a *Controller* object (*controller.js*) based on the view, and finally it starts the 'controller'.

4.2.3 view.js

This module exports a *View* function. The latter can be called with a valid, not self-enclosing tag element as an argument, appending in it the basic structure of the visualization interface. Specifically, it initializes the text and pattern either from the attributes *data-text* and *data-pattern*, if they exist in the element passed as argument, or from the constants *defaultText, defaultPattern* otherwise. Eventually it returns an object offering three functions related to each view, a *get* function that receives a selector as an argument and returns its correspondent element(s) from this specific view only, a *getText*

function that returns the text of the view and a *getPattern* function that returns the pattern of the view.

4.2.4 controller.js

This module exports a *Controller* function. It receives a *View* object as an argument and creates the general functionality for this specific object only. It returns an object that offers the function *start* which triggers the dynamic construction of the functionality. Practically, it dynamically fills the dropdown list with the available algorithms, sets up the letter panels for the current text and pattern and creates an *AnimationControlPanel* (*AnimationControlPanel.js*) object that offers functionality to the control panel's buttons, *reset, start and pause.*

4.2.5 AnimationControlPanel.js

This module exports an *AnimationControlPanel* function, which receives as argument a *View* object and returns an object offering 2 methods, *start* and *stop* which correspond to the *start* and *pause* buttons. Specifically, the function creates an *AnimationController* (*AnimationController.js*) object with its *View* object as argument. The function additionally retrieves the queries array for the current algorithm and processes each query string into a *Query* (*query.js*) object. Finally, it keeps track of which query should be executed each time.

4.2.6 AnimationController.js

This script exports an *AnimationController* function which takes a *View* object as an argument and returns an object providing the function *animate*. This function receives a *Query* (*query.js*) object and executes the corresponding animation in the display panel. All possible animation sub-functions are implemented here.

4.2.7 constants.js

This module stores all global constants, such as the target element class name (*esmajs*), the selectors and class names of the key components in the dynamically created

view and the HTML string of the view structure itself. Notably, the use of id's is avoided so that the functionality can be applicable in multiple target elements in the same page (since only an element can have a specific id in the same page). All these constants are stored as attributes of a constant object, which has been subjected to the *Object.freeze* function that prevents its contents to change afterwards.

4.2.8 query.js

This module exports the class *Query*. Its constructor takes a query string as an argument, which is then parsed and stored in an array. The keyword of the query as well as the rest arguments are retrievable from outside the class with the function *get*.

4.2.9 utils.js

As its name suggests, this module provides utility functions throughout the whole execution phase. Those consist mainly of type checking function like *isString, isNumber* and similar others.

5 Conclusion

This thesis attempted to provide an in-depth, software-development-focused insight in understanding the functionality and measuring the efficiency of 35 string searching algorithms presented by Charras and Lecroq. This was achieved by applying the multilevel process of presenting each algorithm's underlying functionality, implementing them in Java programming language, implementing a benchmark suite to compare their performance on biological data and implementing a visualization platform to animate their functionality.

The algorithms considered here follow the string searching method that receives first the pattern as input and then the text. Their process consists of two phases; first preprocessing the pattern and then performing the search on the text. During the searching phase they utilize the 'sliding window' mechanism, where the pattern is initially aligned with the leftmost end of the text. Then the corresponding characters are compared and in case of mismatch the window slides to the right, repeating above procedure until a match is found or the end of the text is reached.

All algorithms were ported to Java programming language from their C counterparts. All the implementations follow the object-oriented design pattern presented by Sedgewick. Besides the constructor, the algorithms share a common interface of public classes, including *search*-which returns the position of the first occurrence of the pattern in the text-and *searchAll*-which returns a list of the positions of all occurrences. The preprocessing phase was encapsulated in the constructors while the searching phase was placed in the two search methods. Several modification were made and logged, regarding the adaptation of the algorithms in the object-oriented paradigm as well as the correctness of the results.

The efficiency of the algorithms was measured according to their execution times. To run the intended tests, a benchmark suite was implemented, also in Java, which automatically subjects each algorithm class to the designed tests, confirms the correctness of the results and finally exports the results in an organized format to the appropriate files. The benchmark suite relies on Java's Reflection API which allows one to modify and manipulate the behavior of classes and methods during runtime. This approach was preferred because it saved a remarkable amount of code and development

time, as otherwise it would be required to inject a benchmark segment of code in each one of the algorithms.

The benchmark algorithm tested the algorithms on huge biological sequences data consisting of the small alphabet (A, C, G, T) on 4 general scenarios:

- Finding the first occurrence of small patterns
- Finding the first occurrence of large patterns
- Finding all occurrences of small patterns
- Finding all occurrences of large patterns

The results were displayed in three sections; an individual one, which showcases in a line graph the performance of each algorithm along with the minimum and average performances of all algorithms included; a collective one, where the average performance of each algorithm is presented in a bar chart; and finally a repetition of the collective one where the results of each of the four scenarios are grouped according to how the algorithm performs the character comparisons at each attempt (left to right, right to left, in a specific order, in no specific order).

The individual results included the following results. When searching for the first occurrence of small patterns up to 10 length, the Brute Force algorithm clearly is the winner, while for larger patterns (length between 10 -640) Reverse Colussi, Zhu-Takaoka, Reverse Factor, Turbo Reverse Factor, Backward Nondeterministic DAWG Matching and Backward Oracle Matching logged the best times. When searching for all occurrences for small patterns (up to 10 length) Shift Or prevails over every other algorithm, while for large patterns of length between 10 and 640, Reverse Colussi, Zhu-Takaoka, , Reverse Factor, Turbo Reverse Factor, Backward Nondeterministic DAWG Matching and Backward Oracle Matching appear to be the most performant.

The collective results, which were based on the average execution time of each algorithm on every pattern size used, produced the following results. When searching for the first occurrence of small patterns of length up to 10, the best average performance was logged by Brute Force, Shift Or, Boyer-Moore, Reverse Colussi and Maximal Shift. Searching for the first occurrence of large patterns, of length between 10 and 640, on average, proved Zhu-Takaoka, Backward Nondeterministic DAWG Matching, Backward Oracle Matching, Reverse Factor and Maximal Shift to be the frontrunners. Regarding searching for all occurrences of small patterns of length up to 10, the best choices would be Shift Or , Zhu-Takaoka, Tuned Boyer-Moore, Raita and Reverse Colussi. While

searching for all occurrences in large patterns of size between 10 and 640, Zhu-Takaoka, Reverse Factor, Turbo Reverse Factor, Backward Nondeterministic DAWG Matching and Backward Oracle Matching seem to be more efficient.In this last case the Backward Nondeterministic DAWG Matching algorithm can be applied only for patterns up to the length of computer's word size (here 32). Remarkably, it has to be mentioned that all the algorithms of the Boyer-Moore family, besides Apostolico-Giancarlo clearly outperformed all the algorithms of the Morris-Pratt family.

The collective results for each scenario, where grouped into the four categories concerning the character comparison order of each algorithm. The best results are displayed in the following table:

	Left to Right	Right to Left	In a Specific Order	No Specific Order
First occurrence - Small patterns	Shift Or	Boyer-Moore	Maximal Shift	Brute Force
First occurrrence - Large patterns	Karp-Rabin	Zhu-Takaoka	Maximal Shift	Raita
All occurrences - Small patterns	Shift Or	Zhu-Takaoka	Skip Search	Boyer-Moore Tuned
All occurrences - Large patterns	Apostolico-Crochemore	Zhu-Takaoka	Skip Search	Raita

Figure 145: Best performing algorithms in each scenario for each character comparison order group.

Finally, in order to provide a better glimpse of the functionality of the string searching algorithms, a visualization suite was implemented that offers an animated view of the string searching process. The suite was developed using the web technologies HTML, CSS, JavaScript and jQuery, making it easily embeddable in web pages. A demo view can be seen in the static web page **esmaj.surge.sh**, which was deployed using *Surge*, a tool that simplifies publishing static web pages.

6 References

[1] Faro, Simone and Lecroq, Thierry. SMART, String Matching Algorithms Research Tool - http://www.dmi.unict.it/~faro/smart/algorithms.php

[2] Charras, Christian and Lecroq, Thierry. ESMAJ - http://www-igm.univ-mlv.fr/~lecroq/string/

[3] Sedgewick, R and Wayne, K. (2011). Algorithms. 4th ed. Addison-Wesley Professional - https://algs4.cs.princeton.edu/home/

[4] Deterministic Finite Automaton. In Wikipedia. Retrieved February 1, 2019, from: https://en.wikipedia.org/wiki/Deterministic_finite_automaton#cite_note-Cai-9

[5] KARP R.M., RABIN M.O., 1987, Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2):249-260.

[6] BAEZA-YATES, R.A., GONNET, G.H., 1992, A new approach to text searching, Communications of the ACM . 35(10):74-82.

[7] MORRIS (Jr) J.H., PRATT V.R., 1970, A linear pattern-matching algorithm, Technical Report 40, University of California, Berkeley.

[8] COLUSSI L., 1991, Correctness and efficiency of the pattern matching algorithms, Information and Computation 95(2):225-251

[9] APOSTOLICO A., CROCHEMORE M., 1991, Optimal canonization of all substrings of a string, Information and Computation 95(1):76-95.

[10] BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

[11] Maxime Crochemore, Christophe Hancart, Thierry Lecroq. A unifying look at the ApostolicoGiancarlo string-matching algorithm. Journal of Discrete Algorithms, Elsevier, 2003, 1 (1), pp.37-52. ff10.1016/S1570-8667(03)00005-4ff. ffhal-00619563

[12] COLUSSI L., 1994, Fastest pattern matching in strings, Journal of Algorithms. 16(2):163-189.

[13] HORSPOOL R.N., 1980, Practical fast searching in strings, Software - Practice & Experience, 10(6):501-506.

[14] SUNDAY D.M., 1990, A very fast substring search algorithm

[15] HUME A. and SUNDAY D.M. , 1991. Fast string searching. Software - Practice & Experience 21(11):1221-1248.

[16] ZHU R.F., TAKAOKA T., 1987, On improving the average case of the Boyer-Moore string matching algorithm, Journal of Information Processing 10(3):173-177.
[17] BERRY, T., RAVINDRAN, S., 1999, A fast string matching algorithm and experimental results, in Proceedings of the Prague Stringology Club Workshop`99, J. Holub and M. Simánek ed., Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, 1999, pp 16-26.
[18] SMITH P.D., 1991, Experiments with a very fast substring search algorithm, Software - Practice & Experience 21(10):1065-1074.
[19] RAITA T., 1992, Tuning the Boyer-Moore-Horspool string searching algorithm, Software - Practice & Experience, 22(10):879-884.
[20] LECROQ T., 1992, A variation on the Boyer-Moore algorithm, Theoretical Computer Science 92(1):119--144.
[21] NAVARRO G., RAFFINOT M., 1998. A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching, In Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science 1448, Springer-Verlag, Berlin, 14-31.
[22] ALLAUZEN C., CROCHEMORE M., RAFFINOT M., 1999, Factor oracle: a new structure for pattern matching, in Proceedings of SOFSEM'99, Theory and Practice of Informatics, J. Pavelka, G. Tel and M. Bartosek ed., Milovy, Czech Republic, Lecture Notes in Computer Science 1725, pp 291-306, Springer-Verlag, Berlin.
[23] GALIL Z., SEIFERAS J., 1983, Time-space optimal string matching, Journal of Computer and System Science 26(3):280-294.
[24] CROCHEMORE M., PERRIN D., 1991, Two-way string-matching, Journal of the ACM38(3):651-675.
[25] CROCHEMORE M., 1992, String-matching on ordered alphabets, Theoretical Computer Science 92(1):33-47.
[26] CHARRAS C., LECROQ T., PEHOUSHEK J.D., 1998, A very fast string matching algorithm for small alphabets and long patterns, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching , M. Farach-Colton ed., Piscataway, New Jersey, Lecture Notes in Computer Science 1448, pp 55-64, Springer-Verlag, Berlin.
[27] Yoon, S. H., Ha, S. M., Kwon, S., Lim, J., Kim, Y., Seo, H. and Chun, J. (2017). Introducing EzBioCloud: A taxonomically united database of 16S rRNA and whole

genome assemblies. Int J Syst Evol Microbiol. 67:1613-1617. https://www.ezbiocloud.net/genome/explore?puid=172783

7 Appendix

7.1 Algorithm Implementations

7.1.1 Brute Force

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code BruteForce} class finds the occurrencies of a pattern
string in a
 * text string.
 * <p>
 * This implementation provides methods for retrieving the first
occurence of
 * the pattern in the text and for retrieving all the occurencies of
the pattern
 * in the text. This implementation takes time proportional to
<em>nm</em>,
 * where <em>n</em> is the length of the text and <em>m</em> is the
length of
 * the pattern. The expected text characters comparisons are
<em>2n</em>.
 * </p>
 *
 */
public class BruteForce {

      private final String pat;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat the pattern string
       */
      public BruteForce(String pat) {
            if (pat == null)
                  throw new IllegalArgumentException("pattern is
null");
            if (pat.equals(""))
                  throw new IllegalArgumentException("pattern is
empty");

            this.pat = pat;
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
```

```
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final int m = pat.length();
            final int n = txt.length();
            for (int i = 0; i <= n - m; ++i) {
                  int j;
                  for (j = 0; j < m && pat.charAt(j) == txt.charAt(i +
j); ++j)
                        ;
                  if (j >= m) {
                        return i;
                  }
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the pattern
string in the text
       * string.
       *
       * @param txt the text string
       * @return the indices of all the occurrences of the pattern
string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            List<Integer> list = new ArrayList<>();
            final int m = pat.length();
            final int n = txt.length();
            for (int i = 0; i <= n - m; ++i) {
                  int j;
                  for (j = 0; j < m && pat.charAt(j) == txt.charAt(i +
j); ++j)
                        ;
                  if (j >= m) {
                        list.add(i);
                  }
            }
            return list;
      }
      }
```

7.1.2 Deterministic Finite Automaton

```
      package com.accel.stringsearch;

      import java.util.*;

      import com.accel.utils.Automaton;


      /**
```

```
       * The {@code DeterministicFiniteAutomaton} class finds the
occurrences of a
       * pattern string in a text string.
       * <p>
       * This implementation uses a minimal deterministic automaton.
This
       * implementation provides methods for retrieving the first
occurRence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. This implementation takes time proportional to
<em>n</em>, where
       * <em>n</em> is the text's length. The preprocessing phase takes
space and time
       * proportional to <em>mσ</em>, where <em>m</em> is the length of
the pattern
       * and <em>σ</em> is the length of the alphabet.
       * </p>
       *
       */
      public class DeterministicFiniteAutomaton {
            private static final int ASIZE = 256; // alphabet size

            private final char[] pattern; // pattern
            private final Automaton automaton; // automaton

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public DeterministicFiniteAutomaton(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  this.automaton  =  new  Automaton(m  +  1,  (m  +  1)  *
ASIZE);
                  preAut();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
```

```
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int n = txt.length();
                  final int m = pattern.length;

                  for (int state = automaton.getInitial(), i = 0; i <
n; ++i) {
                        state = automaton.getTarget(state, text[i]);
                        if (automaton.isTerminal(state)) {
                              return i - m + 1;
                        }
                  }

                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int n = txt.length();
                  final int m = pattern.length;

                  for (int state = automaton.getInitial(), i = 0; i <
n; ++i) {
                        state = automaton.getTarget(state, text[i]);
                        if (automaton.isTerminal(state)) {
                              list.add(i - m + 1);
                        }
                  }

                  return list;
            }

            /**
             * Builds the automaton from the pattern.
             */
            private void preAut() {
                  final int m = pattern.length;
                  int state = automaton.getInitial();
                  for (int i = 0; i < m; ++i) {
                        int   oldTarget   =   automaton.getTarget(state,
pattern[i]);
```

```
                        int target = automaton.newVertex();
                        automaton.setTarget(state, pattern[i], target);
                        automaton.copyVertex(target, oldTarget);
                        state = target;
                  }
                  automaton.setTerminal(state);
            }

      }
```

7.1.3 Karp-Rabin

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.List;

      /**
       * The {@code KarpRabin} class finds the occurrencies of a
pattern string in a
       * text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurence of
       * the pattern in the text and for retrieving all the occurencies
of the pattern
       * in the text. This implementation uses a hashing function. It
takes time
       * proportional to <em>nm</em>, where <em>n</em> is the length of
the text and
       * <em>m</em> is the length of the pattern. The expected running
time is
       * proportional to <em>n + m</em>. The preprocessing phase takes
<em>m</em> time
       * and constant space.
       * </p>
       *
       */
      public class KarpRabin {

            private final char[] pattern;
            private final int d;
            private final int patHash;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat the pattern string
             */
            public KarpRabin(String pat) {
                  if (pat == null)
                        throw new IllegalArgumentException("pattern is
null");
                  if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  /* compute d = 2^(m-1) with the left-shift operator
*/
                  int d = 1;
```

```
                for (int i = 1; i < m; ++i) {
                      d = d << 1;
                }
                this.d = d;

                // compute the hash code of the pattern
                int patHash = 0;
                for (int i = 0; i < m; ++i) {
                      patHash = (patHash << 1) + pattern[i];
                }
                this.patHash = patHash;
          }

          /**
           * Returns the index of the first occurrence of the pattern
string in the text
           * string.
           *
           * @param txt the text string
           * @return the index of the first occurrence of the pattern
string in the text
           *         string; n if no such match
           */
          public int search(String txt) {
                if (txt == null)
                      throw new IllegalArgumentException("text is
null");
                if (txt.equals(""))
                      throw new IllegalArgumentException("text is
empty");

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                /* compute the hash code of the first m text
characters */
                int txtHash = 0;
                for (int i = 0; i < m; ++i) {
                      txtHash = (txtHash << 1) + text[i];
                }
                /* searching */
                for (int i = 0; i <= n - m; ++i) {
                      if (patHash == txtHash) {
                            if (memcmp(pattern, 0, text, i, m)) {
                                  return i;
                            }
                      }
                      if (i == n - m) break; // prevents
OutOfBoundsException
                      txtHash = rehash(text[i], text[i + m],
txtHash);
                }
                return n;
          }

          /**
           * Returns the indices of all the occurrences of the
pattern string in the text
           * string.
           *
```

```
             * @param txt the text string
             *  @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  /* compute the hash code of the first m text
characters */
                  int txtHash = 0;
                  for (int i = 0; i < m; ++i) {
                        txtHash = (txtHash << 1) + text[i];
                  }
                  /* searching */
                  for (int i = 0; i <= n - m; ++i) {
                        if (patHash == txtHash) {
                              if (memcmp(pattern, 0, text, i, m)) {
                                    list.add(i);
                              }

                        }
                        if (i == n - m) break; // prevents
OutOfBoundsException
                        txtHash = rehash(text[i], text[i + m],
txtHash);
                  }
                  return list;
            }

            // rehashing function
            private int rehash(char c1, char c2, int oldHash) {
                  return ((oldHash - (c1) * d) << 1) + c2;
            }

            // Returns true if the sequential characters of two arrays,
starting from
            // specified indices to a common specified length, are
equal; false otherwise.
            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  if (x == null)
                        throw new IllegalArgumentException("first array
is null");
                  if (y == null)
                        throw new IllegalArgumentException("second
array is null");
                  if (startX < 0 || startX >= x.length)
                        throw new IllegalArgumentException(
                                    "start index of first array: " +
startX + "should be between 0 and " + (x.length - 1));
```

```
                  if (startY < 0 || startY >= y.length)
                        throw new IllegalArgumentException(
                                    "start index of second array: " +
startY + "should be between 0 and " + (y.length - 1));
                  if (startX + length > x.length || startY + length >
y.length)
                        return false;
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }
      }
```

7.1.4 Shift Or

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.List;

      /**
       * The {@code ShiftOr} class finds the occurrences of a pattern
string in a text
       * string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. This implementation is efficient if the pattern
length is no
       * longer than the memory-word size of the machine. It takes time
proportional
       * to <em>n</em>, where <em>n</em> is the length of the text. The
preprocessing
       * phase takes <em>m + σ</em> time and space where <em>m</em> is
the length of
       * the pattern and σ is the alphabet size.
       * </p>
       *
       */
      public class ShiftOr {

            private static final int WORD_SIZE = 31; // word size
            private static final int ASIZE = 256; // radix

            private final char[] pattern;
            private final int m; // pattern length
            private final int[] S; // positions of the characters in
the pattern
            private int lim; // limit

            /**
             * Preprocesses the pattern string.
             *
             * @param pat the pattern string
             */
            public ShiftOr(String pat) {
                  if (pat == null)
```

```
                        throw new IllegalArgumentException("pattern is
null");
                if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");

                this.pattern = pat.toCharArray();
                this.m = pat.length();
                this.S = new int[ASIZE];
                // preprocessing
                for (int i = 0; i < ASIZE; ++i) {
                        S[i] = ~0;
                }
                this.lim = 0;
                for (int i = 0, j = 1; i < m; ++i, j <<= 1) {
                        S[pat.charAt(i)] &= ~j;
                        this.lim |= j;
                }
                this.lim = ~(this.lim >> 1);
        }

        /**
         * Returns the index of the first occurrence of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the index of the first occurrence of the pattern
string in the text
         *         string; n if no such match
         */
        public int search(String txt) {
                if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");
                if (m > WORD_SIZE)
                        return searchLarge(txt);

                final char[] text = txt.toCharArray();
                int n = text.length;
                int state = ~0;
                for (int i = 0; i < n; ++i) {
                        state = (state << 1) | S[text[i]];
                        if (state < lim)
                                return i - m + 1;
                }
                return n;
        }

        /**
         * Returns the indices of all the occurrences of the
pattern string in the text
         * string.
         *
         * @param txt the text string
         * @return the indices of all the occurrences of the
pattern string in the text
```

```
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");
                  if (m > WORD_SIZE)
                        return searchAllLarge(txt);

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  int n = text.length;
                  int state = ~0;
                  for (int i = 0; i < n; ++i) {
                        state = (state << 1) | S[text[i]];
                        if (state < lim)
                              list.add(i - m + 1);
                  }
                  return list;
            }

            private int searchLarge(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");

                  final char[] text = txt.toCharArray();
                  int n = text.length;
                  int state = ~0;
                  for (int i = 0; i < n; ++i) {
                        state = (state << 1) | S[text[i]];
                        if (state < lim) {
                              int k = 0;
                              int h = i - WORD_SIZE + 1;
                              while (k < m && h + k >= 0 && pattern[k]
== text[h + k])
                                    ++k;
                              if (k == m)
                                    return i - WORD_SIZE + 1;
                        }
                  }
                  return n;
            }

            private List<Integer> searchAllLarge(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
```

```
                int n = text.length;
                int state = ~0;
                for (int i = 0; i < n; ++i) {
                      state = (state << 1) | S[text[i]];
                      if (state < lim) {
                            int k = 0;
                            int h = i - WORD_SIZE + 1;
                            while (k < m && h + k >= 0 && pattern[k]
== text[h + k])
                                  ++k;
                            if (k == m)
                                  list.add(i - WORD_SIZE + 1);
                      }
                }
                return list;
          }

     }
```

7.1.5 Morris-Pratt

```
     package com.accel.stringsearch;

     import java.util.*;

     /**
      * The {@code MorrisPratt} class finds the occurrences of a
pattern string in a
      * text string.
      * <p>
      * This implementation provides methods for retrieving the first
occurrence of
      * the pattern in the text and for retrieving all the occurrences
of the pattern
      * in the text. It takes time proportional to <em>n + m</em>,
where <em>n</em>
      * is the length of the text and <em>m</em> is the length of the
pattern. It
      * performs at most <em>2n - 1</em> character comparisons. The
preprocessing
      * phase takes <em>m</em> time and space.
      * </p>
      *
      */
     public class MorrisPratt {
           private char[] pattern; // pattern
           private int[] next; // shift array

           /**
            * Preprocesses the pattern string.
            *
            * @param pat
            *            the pattern string
            */
           public MorrisPratt(String pat) {
                 if      (pat      ==      null)      throw      new
IllegalArgumentException("pattern is null");
                 if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                 this.pattern = pat.toCharArray();
                 preMp();
```

```
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = txt.length();
                  int i = 0;
                  int j = 0;
                  while (j < n) {
                        while (i > -1 && pattern[i] != text[j])
                              i = next[i];
                        i++;
                        j++;
                        if (i >= m) {
                              return j - i;
                        }
                  }
                  return n;
            }

            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             *  @return  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = txt.length();
                  int i = 0;
                  int j = 0;
```

```
                while (j < n) {
                        while (i > -1 && pattern[i] != text[j])
                                i = next[i];
                        i++;
                        j++;
                        if (i >= m) {
                                list.add(j - i);
                                i = next[i];
                        }
                }
                return list;
        }

        private final void preMp() {
                final int m = pattern.length;
                next = new int[m + 1];
                next[0] = -1;
                int j = -1;
                int i = 0;
                while (i < m) {
                        while (j > -1 && pattern[i] != pattern[j]) {
                                j = next[j];
                        }
                        next[++i] = ++j;
                }
        }
}
```

7.1.6 Knuth-Morris-Pratt

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code KnuthMorrisPratt} class finds the occurrences of a
pattern string
 * in a text string.
 * <p>
 * This implementation provides methods for retrieving the first
occurrence of
 * the pattern in the text and for retrieving all the occurrences of
the pattern
 * in the text. It takes time proportional to <em>n + m</em>, where
<em>n</em>
 * is the length of the text and <em>m</em> is the length of the
pattern, and it
 * ia independent of the alphabet size. The preprocessing phase takes
<em>m</em>
 * time and space.
 * </p>
 *
 */
public class KnuthMorrisPratt {
      private char[] pattern; // pattern
      private int[] kmpNext; // shift array

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
```

```
       */
      public KnuthMorrisPratt(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
            if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

            this.pattern = pat.toCharArray();
            final int m = pattern.length;
            kmpNext = new int[m + 1];
            preKmp();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i = 0;
            int j = 0;
            while (j < n) {
                  while (i > -1 && pattern[i] != text[j])
                        i = kmpNext[i];
                  i++;
                  j++;
                  if (i >= m) {
                        return j - i;
                  }
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the indices of all the occurrences of the pattern
string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
```

```
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i = 0;
            int j = 0;
            while (j < n) {
                  while (i > -1 && pattern[i] != text[j])
                        i = kmpNext[i];
                  i++;
                  j++;
                  if (i >= m) {
                        list.add(j - i);
                        i = kmpNext[i];
                  }
            }
            return list;
      }

      // preprocessing
      private void preKmp() {
            final int m = pattern.length;
            int i, j;

            i = 0;
            j = -1;
            kmpNext[0] = -1;
            while (i < m) {
                  while (j > -1 && pattern[i] != pattern[j]) {
                        j = kmpNext[j];
                  }
                  i++;
                  j++;
                  if (i < m && j < m && pattern[i] == pattern[j]) { //
CHANGED:: if(pattern[i) == pattern[j))
                        kmpNext[i] = kmpNext[j];
                  } else {
                        kmpNext[i] = j;
                  }
            }
      }
      }
```

7.1.7 Simon

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code Simon} class finds the occurrences of a pattern string in
a text
 * string.
 * <p>
 * This implementation is an economical implementation of the minimal
 * Deterministic Finite Automaton. This implementation provides methods
for
```

```
 * retrieving the first occurrence of the pattern in the text and for
retrieving
 * all the occurrences of the pattern in the text. This implementation
takes
 * time proportional to <em>m + n</em>, where <em>n</em> is the length
of the
 * text and <em>m</em> is the length of the pattern, and is independent
from the
 * alphabet size.There are at most 2<em>n</em> + 1text character *
comparisons
 * during the search phase. The preprocessing phase takes space and
time
 * proportional to <em>m</em>.
 * </p>
 *
 */
public class SiMoN {
      private final char[] pattern;
      private final int ell;
      private final Cell[] L;

      private static class Cell {
            int element;
            Cell next;
      }

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public SiMoN(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
            if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

            this.pattern = pat.toCharArray();
            final int m = pattern.length;
            this.L = new Cell[m];
            /* Preprocessing */
            this.ell = preSimon();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
```

```
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            for (int state = -1, i = 0; i < n; ++i) {
                  state = getTransition(state, text[i]);
                  if (state >= m - 1) {
                        state = ell;
                        return (i - m + 1);
                  }
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the indices of all the occurrences of the pattern
string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            for (int state = -1, i = 0; i < n; ++i) {
                  state = getTransition(state, text[i]);
                  if (state >= m - 1) {
                        list.add(i - m + 1);
                        state = ell;
                  }
            }
            return list;
      }

      // preprocessing
      private int preSimon() {
            final int m = pattern.length;
            int ell;
            Cell cell;

            for (int i = 0; i < m - 2; ++i) {
                  L[i] = null;
            }
            ell = -1;
            for (int i = 1; i < m; ++i) {
                  int k = ell;
                  cell = (ell == -1 ? null : L[k]);
```

```
                ell = -1;
                if (pattern[i] == pattern[k + 1]) {
                      ell = k + 1;
                } else {
                      setTransition(i - 1, k + 1);
                }
                while (cell != null) {
                      k = cell.element;
                      if (pattern[i] == pattern[k]) {
                            ell = k;
                      } else {
                            setTransition(i - 1, k);
                      }
                      cell = cell.next;
                }
          }
          return ell;

    }

    private int getTransition(int p, char c) {
          final int m = pattern.length;
          Cell cell = new Cell();

          if (p < m - 1 && pattern[p + 1] == c) {
                return p + 1;
          } else if (p > -1) {
                cell = L[p];
                while (cell != null) {
                      if (pattern[cell.element] == c) {
                            return cell.element;
                      } else {
                            cell = cell.next;
                      }
                }
                return -1;
          } else {
                return -1;
          }
    }

    private void setTransition(int p, int q) {
          Cell cell = new Cell();
          cell.element = q;
          cell.next = L[p];
          L[p] = cell;
    }

}
```

7.1.8 Colussi

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code Colussi} class finds the occurrences of a pattern string
in a text
 * string.
 * <p>
```

```
 * This implementation is a refinement of the Knuth, Morris and Pratt
algorithm.
 * This implementation provides methods for retrieving the first
occurrence of
 * the pattern in the text and for retrieving all the occurrences of
the pattern
 * in the text. This implementation takes time proportional to
<em>n</em>, where
 * <em>n</em> is the length of the text. The preprocessing phase takes
space and
 * time proportional to <em>m</em>, where <em>m</em> is the length of
the
 * pattern. There are at most <em>(3/2)n</em> character comparisons.
 * </p>
 *
 */
public class Colussi {

      private final char[] pattern; // pattern
      private final int[] h;
      private final int[] next;
      private final int[] shift;
      private int nd;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public Colussi(String pat) {
            if (pat == null)
                  throw new IllegalArgumentException("pattern is
null");
            if (pat.equals(""))
                  throw new IllegalArgumentException("pattern is
empty");

            this.pattern = pat.toCharArray();
            final int m = pattern.length;
            this.h = new int[m + 1];
            this.next = new int[m + 1];
            this.shift = new int[m + 1];
            /* Processing */
            this.nd = preColussi();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
            if (txt == null)
```

```
                  throw new IllegalArgumentException("text is null");
            if (txt.equals(""))
                  throw new IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            int i, j, last;
            i = 0;
            j = 0;
            last = -1;
            while (j <= n - m) {
                  while (i < m && last < j + h[i] && pattern[h[i]] ==
text[j + h[i]]) {
                        i++;
                  }
                  if (i >= m || last >= j + h[i]) {
                        i = m;
                        return j;
                  }
                  if (i > nd) {
                        last = j + m - 1;
                  }
                  j += shift[i];
                  i = next[i];
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the indices of all the occurrences of the pattern
string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null)
                  throw new IllegalArgumentException("text is null");
            if (txt.equals(""))
                  throw new IllegalArgumentException("text is empty");

            final List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            int i, j, last;
            i = 0;
            j = 0;
            last = -1;
            while (j <= n - m) {
                  while (i < m && last < j + h[i] && pattern[h[i]] ==
text[j + h[i]]) {
                        i++;
                  }
                  if (i >= m || last >= j + h[i]) {
```

```
                        i = m;
                        list.add(j);
                  }
                  if (i > nd) {
                        last = j + m - 1;
                  }
                  j += shift[i];
                  i = next[i];
            }
            return list;
      }

      // preprocessing
      private int preColussi() {
            final int m = pattern.length;
            int i, k, nd, q, r, s;
            int[] hmax = new int[m + 1];
            int[] kmin = new int[m + 1];
            int[] nhd0 = new int[m + 1];
            int[] rmin = new int[m + 1];
            /* Computation of hmax */
            i = 1;
            k = 1;
            do {
                  while (i < m && i - k < m && pattern[i] == pattern[i
- k])
                        i++;
                  hmax[k] = i;
                  q = k + 1;
                  while (hmax[q - k] + k < i) {
                        hmax[q] = hmax[q - k] + k;
                        q++;
                  }
                  k = q;
                  if (k == i + 1) {
                        i = k;
                  }
            } while (k <= m);

            /* Computation of kmin */
            for (int j = 0; j < m; ++j) {
                  kmin[j] = 0;
            }
            for (i = m; i >= 1; --i) {
                  if (hmax[i] < m) {
                        kmin[hmax[i]] = i;
                  }
            }

            /* Computation of rmin */
            r = 0;
            for (i = m - 1; i >= 0; --i) {
                  if (hmax[i + 1] == m) {
                        r = i + 1;
                  }
                  if (kmin[i] == 0) {
                        rmin[i] = r;
                  } else {
                        rmin[i] = 0;
                  }
```

```
            }

            /* Computation of h */
            s = -1;
            r = m;
            for (i = 0; i < m; ++i) {
                  if (kmin[i] == 0) {
                        h[--r] = i;
                  } else {
                        h[++s] = i;
                  }
            }
            nd = s;

            /* Computation of shift */
            for (i = 0; i <= nd; ++i) {
                  shift[i] = kmin[h[i]];
            }
            for (i = nd + 1; i < m; ++i) {
                  shift[i] = rmin[h[i]];
            }
            shift[m] = rmin[0];

            /* Computation of nhd0 */
            s = 0;
            for (i = 0; i < m; ++i) {
                  nhd0[i] = s;
                  if (kmin[i] > 0) {
                        ++s;
                  }
            }

            /* Computation of next */
            for (i = 0; i <= nd; ++i) {
                  next[i] = nhd0[h[i] - kmin[h[i]]];
            }
            for (i = nd + 1; i < m; ++i) {
                  next[i] = nhd0[m - rmin[h[i]]];
            }
            next[m] = nhd0[m - rmin[h[m - 1]]];

            return nd;

      }
      }
```

7.1.9 Galil-Giancarlo

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code GalilGiancarlo} class finds the occurrences of a pattern
string in
 * a text string.
 * <p>
 * This implementation is a refinement of the Colussi algorithm. This
 * implementation provides methods for retrieving the first occurrence
of the
 * pattern in the text and for retrieving all the occurrences of the
pattern in
```

```
 * the text. This implementation takes time proportional to <em>n</em>,
where
 * <em>n</em> is the length of the text. The preprocessing phase takes
space and
 * time proportional to <em>m</em>, where <em>m</em> is the length of
the
 * pattern. There are at most <em>(4/3)n</em> character comparisons.
 * </p>
 *
 */
public class GalilGiancarlo {

      private final char[] pattern; // pattern
      private final int[] h;
      private final int[] next;
      private final int[] shift;
      private final int nd;

      /**
       * Preprocessing.
       *
       * @param pat
       *            the pattern string
       */
      public GalilGiancarlo(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
            if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

            this.pattern = pat.toCharArray();
            final int m = pattern.length;
            this.h = new int[m + 1]; // CHANGED FROM: int[m]
            this.next = new int[m + 1]; // CHANGED FROM: int[m]
            this.shift = new int[m + 1]; // CHANGED FROM: int[m]
            nd = preColussi();           // moved here in preprocessing
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            int i, j, k, ell, last;
            boolean heavy;
```

```
            for (ell = 0; ell < m - 1 && pattern[ell] == pattern[ell +
1]; ell++)
                  ;
            if (ell == m - 1) {
                  /* Searching for a power of a single character */
                  for (j = ell = 0; j < n; ++j) {
                        if (pattern[0] == text[j]) {
                              ++ell;
                              if (ell >= m) {
                                    return j - m + 1;
                              }
                        } else {
                              ell = 0;
                        }
                  }
            } else {
                  /* Searching */
                  i = 0;
                  j = 0;
                  heavy = false;
                  last = -1;
                  while (j <= n - m) {
                        if (heavy && i == 0) {
                              k = last - j + 1;
                              while (pattern[0] == text[j + k]) {
                                    k++;
                              }
                              if (k <= ell || pattern[ell + 1] !=
text[j + k]) {
                                    i = 0;
                                    j += (k + 1);
                                    last = j - 1;
                              } else {
                                    i = 1;
                                    last = j + k;
                                    j = last - (ell + 1);
                              }
                              heavy = false;
                        } else {
                              while (i < m && last < j + h[i] &&
pattern[h[i]] == text[j + h[i]]) {
                                    ++i;
                              }
                              if (i >= m || last >= j + h[i]) {
                                    i = m;
                                    return j;
                              }
                              if (i > nd) {
                                    last = j + m - 1;
                              }
                              j += shift[i];
                              i = next[i];
                        }
                        heavy = (j > last ? false : true);
                  }
            }
            return n;
      }
```

```
      /**
       * Returns the indices of all the occurrences of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the indices of all the occurrences of the pattern
string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int n = text.length;
            final int m = pattern.length;
            int i, j, k, ell, last;
            boolean heavy;

            for (ell = 0; ell < m - 1 && pattern[ell] == pattern[ell +
1]; ell++)
                  ;
            if (ell == m - 1) {
                  /* Searching for a power of a single character */
                  for (j = ell = 0; j < n; ++j) {
                        if (pattern[0] == text[j]) {
                              ++ell;
                              if (ell >= m) {
                                    list.add(j - m + 1);
                              }
                        } else {
                              ell = 0;
                        }
                  }
            } else {
                  /* Searching */
                  i = 0;
                  j = 0;
                  heavy = false;
                  last = -1;
                  while (j <= n - m) {
                        if (heavy && i == 0) {
                              k = last - j + 1;
                              while (pattern[0] == text[j + k]) {
                                    k++;
                              }
                              if (k <= ell || pattern[ell + 1] !=
text[j + k]) {
                                    i = 0;
                                    j += (k + 1);
                                    last = j - 1;
                              } else {
                                    i = 1;
                                    last = j + k;
                                    j = last - (ell + 1);
```

```
                        }
                        heavy = false;
                    } else {
                        while (i < m && last < j + h[i] &&
pattern[h[i]] == text[j + h[i]]) {
                            ++i;
                        }
                        if (i >= m || last >= j + h[i]) {
                            list.add(j);
                            i = m;
                        }
                        if (i > nd) {
                            last = j + m - 1;
                        }
                        j += shift[i];
                        i = next[i];
                    }
                    heavy = (j > last ? false : true);
                }
            }
            return list;
      }

      // preprocessing
      private int preColussi() {
            final int m = pattern.length;
            int i, k, nd, q, r, s;
            int[] hmax = new int[m + 1]; // CHANGED FROM: int[m]
            int[] kmin = new int[m + 1]; // CHANGED FROM: int[m]
            int[] nhd0 = new int[m + 1]; // CHANGED FROM: int[m]
            int[] rmin = new int[m + 1]; // CHANGED FROM: int[m]

            /* Computation of hmax */
            i = 1;
            k = 1;
            do {
                  while (i < m && i - k < m && pattern[i] == pattern[i
- k]) {
                        i++;
                  }
                  hmax[k] = i;
                  q = k + 1;
                  while (hmax[q - k] + k < i) {
                        hmax[q] = hmax[q - k] + k;
                        q++;
                  }
                  k = q;
                  if (k == i + 1) {
                        i = k;
                  }
            } while (k <= m);

            /* Computation of kmin */
            for (int j = 0; j < m; ++j) {
                  kmin[j] = 0;
            }
            for (i = m; i >= 1; --i) {
                  if (hmax[i] < m) {
                        kmin[hmax[i]] = i;
                  }
```

```
            }

            /* Computation of rmin */
            r = 0;
            for (i = m - 1; i >= 0; --i) {
                  if (hmax[i + 1] == m) {
                        r = i + 1;
                  }
                  if (kmin[i] == 0) {
                        rmin[i] = r;
                  } else {
                        rmin[i] = 0;
                  }
            }

            /* Computation of h */
            s = -1;
            r = m;
            for (i = 0; i < m; ++i) {
                  if (kmin[i] == 0) {
                        h[--r] = i;
                  } else {
                        h[++s] = i;
                  }
            }
            nd = s;

            /* Computation of shift */
            for (i = 0; i <= nd; ++i) {
                  shift[i] = kmin[h[i]];
            }
            for (i = nd + 1; i < m; ++i) {
                  shift[i] = rmin[h[i]];
            }
            shift[m] = rmin[0];

            /* Computation of nhd0 */
            s = 0;
            for (i = 0; i < m; ++i) {
                  nhd0[i] = s;
                  if (kmin[i] > 0) {
                        ++s;
                  }
            }

            /* Computation of next */
            for (i = 0; i <= nd; ++i) {
                  next[i] = nhd0[h[i] - kmin[h[i]]];
            }
            for (i = nd + 1; i < m; ++i) {
                  next[i] = nhd0[m - rmin[h[i]]];
            }
            next[m] = nhd0[m - rmin[h[m - 1]]];

            return nd;

      }
}
```

7.1.10 Apostolico-Crochemore

```
      package com.accel.stringsearch;
```

```
      import java.util.ArrayList;
      import java.util.List;

      /**
       * The {@code ApostolicoCrochemore} class finds the occurrences
of a pattern
       * string in a text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. This implementation takes time proportional to
<em>n</em>, where
       * <em>n</em> is the length of the text.There are at most
<em>n(3/2)</em> text
       * character * comparisons during the search phase. The
preprocessing phase
       * takes space and time proportional to <em>m</em>, where
<em>m</em> is the
       * length of the pattern.
       * </p>
       *
       */
      public class ApostolicoCrochemore {

            private final int[] kmpNext;
            private final char[] pattern;
            private final int ell;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat the pattern string
             */
            public ApostolicoCrochemore(String pat) {
                  if (pat == null)
                        throw new IllegalArgumentException("pattern is
null");
                  if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");

                  final int m = pat.length();
                  this.pattern = pat.toCharArray();
                  int ell;
                  this.kmpNext = new int[m + 1]; // CHANGED :: new
int[m]
                  /* Preprocessing */
                  preKmp();
                  for (ell = 1; ell < m && pattern[ell - 1] ==
pattern[ell]; ell++);
                  if (ell == m)
                        ell = 0;
                  this.ell = ell;
            }

            /**
```

```
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");

                  int i, j, k;
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  /* Searching */
                  i = ell;
                  j = 0;
                  k = 0;
                  while (j <= n - m) {
                        while (i < m && pattern[i] == text[i + j]) {
                              ++i;
                        }
                        if (i >= m) {
                              while (k < ell && pattern[k] == text[j +
k]) {
                                    ++k;
                              }
                              if (k >= ell)
                                    return j;
                        }
                        j += (i - kmpNext[i]);
                        if (i == ell) {
                              k = Math.max(0, k - 1);
                        } else {
                              if (kmpNext[i] <= ell) {
                                    k = Math.max(0, kmpNext[i]);
                                    i = ell;
                              } else {
                                    k = ell;
                                    i = kmpNext[i];
                              }
                        }
                  }
                  return n;
            }

            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt the text string
```

```
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");

                  List<Integer> list = new ArrayList<>();
                  int i, j, k;
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  /* Searching */
                  i = ell;
                  j = 0;
                  k = 0;
                  while (j <= n - m) {
                        while (i < m && pattern[i] == text[i + j]) {
                              ++i;
                        }
                        if (i >= m) {
                              while (k < ell && pattern[k] == text[j +
k]) {
                                    ++k;
                              }
                              if (k >= ell)
                                    list.add(j);
                        }
                        j += (i - kmpNext[i]);
                        if (i == ell) {
                              k = Math.max(0, k - 1);
                        } else {
                              if (kmpNext[i] <= ell) {
                                    k = Math.max(0, kmpNext[i]);
                                    i = ell;
                              } else {
                                    k = ell;
                                    i = kmpNext[i];
                              }
                        }
                  }

                  return list;
            }

            // preprocessing
            private void preKmp() {
                  final int m = pattern.length;
                  int i, j;

                  i = 0;
                  j = -1;
                  kmpNext[0] = -1;
                  while (i < m) {
                        while (j > -1 && pattern[i] != pattern[j]) {
```

```
                              j = kmpNext[j];
                        }
                        i++;
                        j++;
                        if (i < m && j < m && pattern[i] == pattern[j])
{ // CHANGED:: if(pat.charAt(i) == pat.charAt(j))
                              kmpNext[i] = kmpNext[j];
                        } else {
                              kmpNext[i] = j;
                        }
                  }
            }

      }
```

7.1.11 Not So Naïve

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       *  The  {@code  NotSoNaive}  class  finds  the  occurrences  of  a
pattern string in a
       * text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. This implementation takes time proportional to
<em>nm</em>,
       * where <em>n</em> is the length of the text and <em>m</em> is
the length of
       * the pattern. On average it is slightly (by coefficient) sub-
linear.
       * </p>
       *
       */
      public class NotSoNaive {
            private final char[] pattern; // pattern
            private final int ell;
            private final int k;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public NotSoNaive(String pat) {
                  if (pat == null)
                        throw  new  IllegalArgumentException("pattern  is
null");
                  if (pat.equals(""))
                        throw  new  IllegalArgumentException("pattern  is
empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  int ell, k;
                  /* Preprocessing */
```

```
                if (m > 1 && pattern[0] == pattern[1]) {
                      k = 2;
                      ell = 1;
                } else {
                      k = 1;
                      ell = 2;
                }
                this.ell = ell;
                this.k = k;
          }

          /**
           * Returns the index of the first occurrence of the pattern
string in the text
           * string.
           *
           * @param txt
           *            the text string
           * @return the index of the first occurrence of the pattern
string in the text
           *         string; n if no such match
           */
          public int search(String txt) {
                if (txt == null)
                      throw   new   IllegalArgumentException("text   is
null");
                if (txt.equals(""))
                      throw   new   IllegalArgumentException("text   is
empty");
                if (pattern.length == 1) {
                      return                                          new
BruteForce(String.copyValueOf(pattern)).search(txt);
                }

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int skip = 0;

                for (int i = 0; i <= n - m; i += skip) {
                      if (m == 1 && pattern[0] == text[i]) {
                            return i;
                      }
                      if (pattern[1] != text[i + 1]) {
                            skip = k;
                      } else {
                            if (memcmp(pattern, 2, text, i + 2, m -
2) && pattern[0] == text[i]) {
                                  return i;
                            }
                            skip = ell;
                      }
                }
                return n;
          }
          /**
           *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
           * string.
           *
```

```
             * @param txt
             *            the text string
             *  @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");
                  if (pattern.length == 1) {
                        return                                    new
BruteForce(String.copyValueOf(pattern)).searchAll(txt);
                  }

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int skip = 0;

                  for (int i = 0; i <= n - m; i += skip) {
                        if (pattern[1] != text[i + 1]) {
                              skip = k;
                        } else {
                              if (memcmp(pattern, 2, text, i + 2, m -
2) && pattern[0] == text[i]) {
                                    list.add(i);
                              }
                              skip = ell;
                        }
                  }
                  return list;
            }

            // Returns true if the sequential characters of two arrays,
starting from
            // specified indices to a common specified length, are
equal; false otherwise.
            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
      //          if (x == null)
      //                throw new IllegalArgumentException("first array
is null");
      //          if (y == null)
      //                throw    new    IllegalArgumentException("second
array is null");
      //          if (startX < 0 || startX >= x.length)
      //                throw new IllegalArgumentException(
      //                            "start index of first array: " +
startX + " should be between 0 and " + (x.length - 1));
      //          if (startY < 0 || startY >= y.length)
      //                throw new IllegalArgumentException(
      //                            "start index of second array: " +
startY + " should be between 0 and " + (y.length - 1));
      //          if (startX + length > x.length || startY + length >
y.length)
```

```
      //                return false;
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }
      }
```

7.1.12 Boyer-Moore

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code BoyerMoore} class finds the occurrences of a
pattern string in a
       * text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. It takes time proportional to <em>nm</em> in the
worst case and
       * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
       * <em>m</em> is the length of the pattern. The preprocessing
phase takes <em>m
       * + σ</em> time and space, where σ is the size of the alphabet.
The character
       * comparisons are <em>3n</em> in the worst case when searching
for a non
       * periodic pattern.
       * </p>
       *
       */
      public class BoyerMoore {

            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] bmGs;
            private final int[] bmBc;
            private final int[] suff;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public BoyerMoore(String pat) {
                  if      (pat      ==      null)      throw      new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
```

```
                this.bmGs = new int[m];
                this.bmBc = new int[ASIZE];
                this.suff = new int[m];

                /* Preprocessing */
                preBmGs();
                preBmBc();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j;

                j = 0;
                while (j <= n - m) {
                      for (i = m - 1; i >= 0 && pattern[i] == text[i
+ j]; --i)
                            ;
                      if (i < 0) {
                            return j;
                      } else {
                            j += Math.max(bmGs[i], bmBc[text[i + j]]
- m + 1 + i);
                      }
                }
                return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
```

```
                if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                List<Integer> list = new ArrayList<>();
                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j;

                j = 0;
                while (j <= n - m) {
                      for (i = m - 1; i >= 0 && pattern[i] == text[i
+ j]; --i)
                            ;
                      if (i < 0) {
                            list.add(j);
                            j += bmGs[0];
                      } else {
                            j += Math.max(bmGs[i], bmBc[text[i + j]]
- m + 1 + i);
                      }
                }

                return list;
          }

          // construct bad character shift array
          private void preBmBc() {
                final int m = pattern.length;
                for (int i = 0; i < ASIZE; ++i) {
                      bmBc[i] = m;
                }
                for (int i = 0; i < m - 1; ++i) {
                      bmBc[pattern[i]] = m - i - 1;
                }
          }

          private void suffixes() {
                final int m = pattern.length;
                int f, g;

                suff[m - 1] = m;
                g = m - 1;
                f = 0;
                for (int i = m - 2; i >= 0; --i) {
                      if (i > g && suff[i + m - 1 - f] < i - g) {
                            suff[i] = suff[i + m - 1 - f];
                      } else {
                            if (i < g) {
                                  g = i;
                            }
                            f = i;
                            while (g >= 0 && pattern[g] == pattern[g
+ m - 1 - f]) {
                                  --g;
                            }
                            suff[i] = f - g;
                      }
                }
```

```
            }

            // construct good suffix shift array
            private void preBmGs() {
                  final int m = pattern.length;
                  int i, j;
                  suffixes();
                  for (i = 0; i < m; ++i) {
                        bmGs[i] = m;
                  }
                  j = 0;
                  for (i = m - 1; i >= 0; --i) {
                        if (suff[i] == i + 1) {
                              for (; j < m - 1 - i; ++j) {
                                    if (bmGs[j] == m) {
                                          bmGs[j] = m - 1 - i;
                                    }
                              }
                        }
                  }
                  for (i = 0; i <= m - 2; ++i) {
                        bmGs[m - 1 - suff[i]] = m - 1 - i;
                  }

            }

      }
```

7.1.13 Turbo Boyer-Moore

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code TurboBM} class finds the occurrences of a pattern
string in a text
       * string.
       * <p>
       * This implementation is a variant of the Boyer-Moore algorithm.
This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It takes time proportional to <em>n</em> in the
worst case, where
       * <em>n</em> is the length of the text. The preprocessing phase
takes <em>m +
       * ◆</em> time and space, where ◆ is the size of the alphabet.
The character
       * comparisons are <em>2n</em> in the worst case.
       * </p>
       *
       */
      public class TurboBM {
            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] bmGs;
            private final int[] bmBc;
            private final int[] suff;
```

```
            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public TurboBM(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  this.bmGs = new int[m];
                  this.bmBc = new int[ASIZE];
                  this.suff = new int[m];

                  /* Preprocessing */
                  preBmGs();
                  preBmBc();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, u, v, bcShift, shift, turboShift;

                  j = u = 0;
                  shift = m;
                  while (j <= n - m) {
                        i = m - 1;
                        while (i >= 0 && pattern[i] == text[i + j]) {
                              --i;
                              if (u != 0 && i == m - 1 - shift) {
                                    i -= u;
                              }
                        }
                        if (i < 0) {
                              return j;
                        } else {
                              v = m - 1 - i;
```

```
                        turboShift = u - v;
                        bcShift = bmBc[text[i + j]] - m + 1 + i;
                        shift = Math.max(turboShift, bcShift);
                        shift = Math.max(shift, bmGs[i]);
                        if (shift == bmGs[i])
                              u = Math.min(m - shift, v);
                        else {
                              if (turboShift < bcShift)
                                    shift = Math.max(shift, u +
1);
                              u = 0;
                        }
                  }
                  j += shift;
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the
pattern string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the indices of all the occurrences of the
pattern string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null) throw new
IllegalArgumentException("text is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i, j, u, v, bcShift, shift, turboShift;

            j = u = 0;
            shift = m;
            while (j <= n - m) {
                  i = m - 1;
                  while (i >= 0 && pattern[i] == text[i + j]) {
                        --i;
                        if (u != 0 && i == m - 1 - shift) {
                              i -= u;
                        }
                  }
                  if (i < 0) {
                        list.add(j);
                        shift = bmGs[0];
                        u = m - shift;
                  } else {
                        v = m - 1 - i;
                        turboShift = u - v;
                        bcShift = bmBc[text[i + j]] - m + 1 + i;
                        shift = Math.max(turboShift, bcShift);
```

```
                        shift = Math.max(shift, bmGs[i]);
                        if (shift == bmGs[i])
                              u = Math.min(m - shift, v);
                        else {
                              if (turboShift < bcShift)
                                    shift = Math.max(shift, u +
1);
                              u = 0;
                        }
                  }
                  j += shift;
            }
            return list;
      }

      // construct bad character shift array
      private void preBmBc() {
            final int m = pattern.length;
            for (int i = 0; i < ASIZE; ++i) {
                  bmBc[i] = m;
            }
            for (int i = 0; i < m - 1; ++i) {
                  bmBc[pattern[i]] = m - i - 1;
            }
      }

      private void suffixes() {
            final int m = pattern.length;
            int f, g;

            suff[m - 1] = m;
            g = m - 1;
            f = 0;
            for (int i = m - 2; i >= 0; --i) {
                  if (i > g && suff[i + m - 1 - f] < i - g) {
                        suff[i] = suff[i + m - 1 - f];
                  } else {
                        if (i < g) {
                              g = i;
                        }
                        f = i;
                        while (g >= 0 && pattern[g] == pattern[g
+ m - 1 - f]) {
                              --g;
                        }
                        suff[i] = f - g;
                  }
            }

      }

      // construct good suffix shift array
      private void preBmGs() {
            final int m = pattern.length;
            int i, j;
            suffixes();
            for (i = 0; i < m; ++i) {
                  bmGs[i] = m;
            }
            j = 0;
```

```
                for (i = m - 1; i >= 0; --i) {
                      if (suff[i] == i + 1) {
                            for (; j < m - 1 - i; ++j) {
                                  if (bmGs[j] == m) {
                                        bmGs[j] = m - 1 - i;
                                  }
                            }
                      }
                }
                for (i = 0; i <= m - 2; ++i) {
                      bmGs[m - 1 - suff[i]] = m - 1 - i;
                }

          }

     }
```

7.1.14 Apostolico-Giancarlo

```
     package com.accel.stringsearch;

     import java.util.ArrayList;
     import java.util.List;

     /**
      * The {@code ApostolicoGiancarlo} class finds the occurrences of
a pattern
      * string in a text string.
      * <p>
      * This implementation is a variant of the Boyer-Moore algorithm.
This
      * implementation provides methods for retrieving the first
occurrence of the
      * pattern in the text and for retrieving all the occurrences of
the pattern in
      * the text. It takes time proportional to <em>n</em>, where
<em>n</em> is the
      * length of the text. The preprocessing phase takes <em>m +
σ</em> time and
      * space, where σ is the size of the alphabet. The character
comparisons are
      * <em>3n/2</em> in the worst case.
      * </p>
      *
      */
     public class ApostolicoGiancarlo {

          private static final int ASIZE = 256;

          private final char[] pattern;
          private final int[] bmGs;
          private final int[] bmBc;
          private final int[] suff;
          private final int[] skip;

          /**
           * Preprocesses the pattern string.
           *
           * @param pat the pattern string
           */
          public ApostolicoGiancarlo(String pat) {
```

```
                if (pat == null)
                      throw new IllegalArgumentException("pattern is
null");
                if (pat.equals(""))
                      throw new IllegalArgumentException("pattern is
empty");

                this.pattern = pat.toCharArray();
                final int m = pattern.length;
                this.bmGs = new int[m];
                this.bmBc = new int[ASIZE];
                this.suff = new int[m];
                this.skip = new int[m];

                /* Preprocessing */
                preBmGs();
                preBmBc();
          }

          /**
           * Returns the index of the first occurrence of the pattern
string in the text
           * string.
           *
           * @param txt the text string
           * @return the index of the first occurrence of the pattern
string in the text
           *         string; n if no such match
           */
          public int search(String txt) {
                if (txt == null)
                      throw new IllegalArgumentException("text is
null");
                if (txt.equals(""))
                      throw new IllegalArgumentException("text is
empty");

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, k, s, shift;

                j = 0;
                while (j <= n - m) {
                      i = m - 1;
                      while (i >= 0) {
                            k = skip[i];
                            s = suff[i];
                            if (k > 0)
                                  if (k > s) {
                                        if (i + 1 == s)
                                              i = (-1);
                                        else
                                              i -= s;
                                        break;
                                  } else {
                                        i -= k;
                                        if (k < s)
                                              break;
                                  }
```

```
                        else {
                              if (pattern[i] == text[i + j])
                                    --i;
                              else
                                    break;
                        }
                  }
                  if (i < 0) {
                        return j;
                        // skip[m - 1] = m;
                        // shift = bmGs[0];
                  } else {
                        skip[m - 1] = m - 1 - i;
                        shift = Math.max(bmGs[i], bmBc[text[i +
j]] - m + 1 + i);
                  }
                  j += shift;
                  memcpy(skip, 0, skip, shift, (m - shift));
                  memset(skip, m - shift, shift, 0);
            }
            return n;
      }

      /**
       * Returns the indices of all the occurrences of the
pattern string in the text
       * string.
       *
       * @param txt the text string
       * @return the indices of all the occurrences of the
pattern string in the text
       *         string
       */
      public List<Integer> searchAll(String txt) {
            if (txt == null)
                  throw new IllegalArgumentException("text is
null");
            if (txt.equals(""))
                  throw new IllegalArgumentException("text is
empty");

            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i, j, k, s, shift;

            j = 0;
            while (j <= n - m) {
                  i = m - 1;
                  while (i >= 0) {
                        k = skip[i];
                        s = suff[i];
                        if (k > 0)
                              if (k > s) {
                                    if (i + 1 == s)
                                          i = (-1);
                                    else
                                          i -= s;
                                    break;
```

```
                                    } else {
                                          i -= k;
                                          if (k < s)
                                                break;
                                    }
                              else {
                                    if (pattern[i] == text[i + j])
                                          --i;
                                    else
                                          break;
                              }
                        }
                        if (i < 0) {
                              list.add(j);
                              skip[m - 1] = m;
                              shift = bmGs[0];
                        } else {
                              skip[m - 1] = m - 1 - i;
                              shift  =  Math.max(bmGs[i],  bmBc[text[i  +
j]] - m + 1 + i);
                        }
                        j += shift;
                        memcpy(skip, 0, skip, shift, (m - shift));
                        memset(skip, m - shift, shift, 0);
                  }

                  return list;
            }

            // construct bad character shift array
            private void preBmBc() {
                  final int m = pattern.length;
                  for (int i = 0; i < ASIZE; ++i) {
                        bmBc[i] = m;
                  }
                  for (int i = 0; i < m - 1; ++i) {
                        bmBc[pattern[i]] = m - i - 1;
                  }
            }

            private void suffixes() {
                  final int m = pattern.length;
                  int f, g;

                  suff[m - 1] = m;
                  g = m - 1;
                  f = 0;
                  for (int i = m - 2; i >= 0; --i) {
                        if (i > g && suff[i + m - 1 - f] < i - g) {
                              suff[i] = suff[i + m - 1 - f];
                        } else {
                              if (i < g) {
                                    g = i;
                              }
                              f = i;
                              while (g >= 0 && pattern[g] == pattern[g
+ m - 1 - f]) {
                                    --g;
                              }
                              suff[i] = f - g;
```

```
                        }
                  }

            }

            // construct good suffix shift array
            private void preBmGs() {
                  final int m = pattern.length;
                  int i, j;
                  suffixes();
                  for (i = 0; i < m; ++i) {
                        bmGs[i] = m;
                  }
                  j = 0;
                  for (i = m - 1; i >= 0; --i) {
                        if (suff[i] == i + 1) {
                              for (; j < m - 1 - i; ++j) {
                                    if (bmGs[j] == m) {
                                          bmGs[j] = m - 1 - i;
                                    }
                              }
                        }
                  }
                  for (i = 0; i <= m - 2; ++i) {
                        bmGs[m - 1 - suff[i]] = m - 1 - i;
                  }

            }

            // memcpy
            private void memcpy(int[] dest, int d_i, int[] src, int
s_i, int l) {
                  int[] temp = new int[l];
                  for (int i = 0; i < l; ++i) {
                        temp[i] = src[s_i + i];
                  }

                  for (int i = 0; i < l; ++i) {
                        dest[d_i + i] = temp[i];
                  }
            }

            // memset
            private void memset(int[] dest, int start, int length, int
val) {
                  for (int i = 0; i < length; ++i) {
                        dest[start + i] = val;
                  }
            }

      }
```

7.1.15 Reverse Colussi

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code ReverseColussi} class finds the occurrences of a
pattern string in
       * a text string.
```

```
       * <p>
       *  This implementation is a refinement of the Boyer-Moore
algorithm. This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       *  the text. This implementation takes time proportional to
<em>n</em>, where
       * <em>n</em> is the length of the text. The preprocessing phase
takes space
       * proportional to <em>mσ</em>, where <em>m</em> is the length of
the pattern
       *  and σ is the alphabet size, and time proportional to
<em>m^2</em>. There are
       * at most <em>2n</em> character comparisons.
       * </p>
       *
       */
      public class ReverseColussi {
            private static final int ASIZE = 256; // alphabet size

            private final char[] pattern; // pattern
            private final int[] h;
            private final int[][] rcBc;
            private final int[] rcGs;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public ReverseColussi(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  this.h = new int[m + 1]; // CHANGED FROM: int[m]
                  this.rcBc = new int[ASIZE][m + 1]; // CHANGED FROM:
int[m]
                  this.rcGs = new int[m + 1]; // CHANGED FROM: int[m]
                  /* Processing */
                  preRc();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
```

```
            public int search(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int n = text.length;
                  final int m = pattern.length;
                  int i, j, s;

                  j = 0;
                  s = m;
                  while (j <= n - m) {
                        while (j <= n - m && pattern[m - 1] != text[j +
m - 1]) {
                              s = rcBc[text[j + m - 1]][s];
                              j += s;
                        }
                        for  (i  =  1;  i  <  m  &&  j  +  h[i]  <  n  &&
pattern[h[i]] == text[j + h[i]]; ++i)
                              ;
                        if (i >= m && j <= n - m) // CHANGED :: (i >=
m)
                              return j;
                        s = rcGs[i];
                        j += s;
                  }
                  return n;
            }

            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             *  @return  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int n = text.length;
                  final int m = pattern.length;
                  int i, j, s;

                  j = 0;
                  s = m;
                  while (j <= n - m) {
                        while (j <= n - m && pattern[m - 1] != text[j +
m - 1]) {
                              s = rcBc[text[j + m - 1]][s];
```

```
                        j += s;
                    }
                    for (i = 1; i < m && j + h[i] < n &&
pattern[h[i]] == text[j + h[i]]; ++i)
                        ;
                    if (i >= m && j <= n - m) // CHANGED :: (i >=
m)
                        list.add(j);
                    s = rcGs[i];
                    j += s;
                }
                return list;
            }

            // preprocessing
            private void preRc() {
                final int m = pattern.length;
                int a, i, j, k, q, r, s;

                int[] hmin = new int[m + 1];
                int[] kmin = new int[m + 1];
                int[] link = new int[m + 1];
                int[] locc = new int[ASIZE];
                int[] rmin = new int[m + 1];

                /* Computation of link and locc */
                for (a = 0; a < ASIZE; ++a)
                    locc[a] = -1;
                link[0] = -1;
                for (i = 0; i < m - 1; ++i) {
                    link[i + 1] = locc[pattern[i]];
                    locc[pattern[i]] = i;
                }

                /* Computation of rcBc */
                for (a = 0; a < ASIZE; ++a)
                    for (s = 1; s <= m; ++s) {
                        i = locc[a];
                        j = link[m - s];
                        while (i - j != s && j >= 0)
                            if (i - j > s)
                                i = link[i + 1];
                            else
                                j = link[j + 1];
                        while (i - j > s)
                            i = link[i + 1];
                        rcBc[a][s] = m - i - 1;
                    }

                /* Computation of hmin */
                k = 1;
                i = m - 1;
                while (k <= m) {
                    while (i - k >= 0 && pattern[i - k] ==
pattern[i])
                        --i;
                    hmin[k] = i;
                    q = k + 1;
                    while (hmin[q - k] - (q - k) > i) {
                        hmin[q] = hmin[q - k];
```

```
                        ++q;
                    }
                    i += (q - k);
                    k = q;
                    if (i == m)
                        i = m - 1;
                }

                /* Computation of kmin */
                for (i = 0; i < m; i++)
                    kmin[i] = 0;
                for (k = m; k > 0; --k)
                    kmin[hmin[k]] = k;

                /* Computation of rmin */
                r = 0;
                for (i = m - 1; i >= 0; --i) {
                    if (hmin[i + 1] == i)
                        r = i + 1;
                    rmin[i] = r;
                }

                /* Computation of rcGs */
                i = 1;
                for (k = 1; k <= m; ++k)
                    if (hmin[k] != m - 1 && kmin[hmin[k]] == k) {
                        h[i] = hmin[k];
                        rcGs[i++] = k;
                    }
                i = m - 1;
                for (j = m - 2; j >= 0; --j)
                    if (kmin[j] == 0) {
                        h[i] = j;
                        rcGs[i--] = rmin[j];
                    }
                rcGs[m] = rmin[0];
            }
        }
```

7.1.16 Horspool

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code Horspool} class finds the occurrences of a pattern
string in a
 * text string.
 * <p>
 * This implementation is a simplification of the Boyer-Moore
algorithm. This
 * implementation provides methods for retrieving the first
occurrence of the
 * pattern in the text and for retrieving all the occurrences of
the pattern in
 * the text. It takes time proportional to <em>nm</em> in the
worst case and
 * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
```

```
       * <em>m</em> is the length of the pattern. The preprocessing
phase takes <em>m
       * + σ</em> time and <em>σ</em> space, where σ is the size of the
alphabet. The
       *   character   comparisons   are   between   <em>1/σ</em>   and
<em>2/(σ+1)</em> on
       * average.
       * </p>
       *
       */
      public class Horspool {

            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] bmBc;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public Horspool(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if          (pat.equals(""))          throw          new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  this.bmBc = new int[ASIZE];
                  preBmBc();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  char cPat, cTxt;

                  for (int i = 0; i <= n - m; i += bmBc[cTxt]) {
                        cPat = pattern[m - 1];
                        cTxt = text[i + m - 1];
```

```
                        if (cPat == cTxt && memcmp(pattern, 0, text, i,
m - 1))
                              return i;
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null) throw new
IllegalArgumentException("text is null");
                  if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  char cPat, cTxt;

                  for (int i = 0; i <= n - m; i += bmBc[cTxt]) {
                        cPat = pattern[m - 1];
                        cTxt = text[i + m - 1];
                        if (cPat == cTxt && memcmp(pattern, 0, text, i,
m - 1))
                              list.add(i);
                  }
                  return list;
            }

            // construct bad character shift array
            private void preBmBc() {
                  final int m = pattern.length;
                  for (int i = 0; i < ASIZE; ++i) {
                        bmBc[i] = m;
                  }
                  for (int i = 0; i < m - 1; ++i) {
                        bmBc[pattern[i]] = m - i - 1;
                  }
            }

            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }
```

```
}
```

7.1.17 Quick Search

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code QuickSearch} class finds the occurrences of a
pattern string in a
 * text string.
 * <p>
 * This implementation is a simplification of the Boyer-Moore
algorithm. This
 * implementation provides methods for retrieving the first
occurrence of the
 * pattern in the text and for retrieving all the occurrences of
the pattern in
 * the text. It takes time proportional to <em>nm</em> in the
worst case and
 * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
 * <em>m</em> is the length of the pattern. The preprocessing
phase takes <em>m
 * + σ</em> time and <em>σ</em> space, where σ is the size of the
alphabet. It
 * is very fast in practice for short patterns and large
alphabets.
 * </p>
 *
 */
public class QuickSearch {

      private static final int ASIZE = 256;

      private final char[] pattern;
      private final int[] qsBc;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public QuickSearch(String pat) {
            if      (pat      ==      null)      throw      new
IllegalArgumentException("pattern is null");
            if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

            this.pattern = pat.toCharArray();
            this.qsBc = new int[ASIZE];
            preQsBc();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
```

```
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  for (int i = 0; i <= n - m; i += qsBc[text[i + m]]) {
                        if (memcmp(pattern, 0, text, i, m)) {
                              return i;
                        }
                        if (i == n - m) {
                              break;
                        }
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  for (int i = 0; i <= n - m; i += qsBc[text[i + m]]) {
                        if (memcmp(pattern, 0, text, i, m)) {
                              list.add(i);
                        }
                        if (i == n - m) {
                              break;
                        }
                  }
                  return list;
            }

            // preprocessing
```

```
            private void preQsBc() {
                  final int m = pattern.length;
                  for (int i = 0; i < ASIZE; ++i)
                        qsBc[i] = m + 1;
                  for (int i = 0; i < m; ++i)
                        qsBc[pattern[i]] = m - i;
            }

            // returns true if elements of two arrays-starting from
specified indices to a
            // common length- are equal; false otherwise
            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }

      }
```

7.1.18 Tuned Boyer-Moore

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code TunedBoyerMoore} class finds the occurrences of a
pattern string in a
       * text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. It takes quadratic time in the worst case, but
has a very good
       * practical behaviour.
       * </p>
       *
       */

      public class BoyerMooreTuned {

          private static final int ASIZE = 256;

          private final char[] pat;
          private final int[] bmBc;
          private final int shift;

          /**
           * Preprocesses the pattern string.
           *
           * @param pat the pattern string
           */
          public BoyerMooreTuned(String pat) {
            if        (pat       ==        null)        throw        new
IllegalArgumentException("pattern is null");
```

```
                if          (pat.equals(""))          throw          new
IllegalArgumentException("pattern is empty");

                this.pat = pat.toCharArray();
                final int m = pat.length();
                this.bmBc = new int[ASIZE];
                preBmBc();
                shift = bmBc[this.pat[m - 1]];
                bmBc[this.pat[m - 1]] = 0;

            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *          string; n if no such match
             */
            public int search(String txt) {
              if (txt == null) throw new IllegalArgumentException("text
is null");
                if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                final int m = pat.length;
                final int n = txt.length();
                int j, k;
                final char[] txtArr = new char[n + m];

                for (int i = 0; i < n; ++i) {
                    txtArr[i] = txt.charAt(i);
                }
                Arrays.fill(txtArr, n, n + m, pat[m - 1]);
                j = 0;
                while (j < n) {
                    k = bmBc[txtArr[j + m -1]];
                    while (k !=  0) {
                        j += k;
                        k = bmBc[txtArr[j + m - 1]];
                        j += k;
                        k = bmBc[txtArr[j + m - 1]];
                        j += k;
                        k = bmBc[txtArr[j + m - 1]];
                    }
                    if (memcmp(pat, 0, txtArr, j, m - 1) && j < n) { //
CHANGED :: j < n - m  CHANGED :: j < n
                        return j;
                    }
                    j += shift;                           /* shift */
                }
                return n;
            }

            /**
             * Returns the indices of all the occurrences of the pattern
string in the text
```

```
           * string.
           *
           * @param txt the text string
           * @return the indices of all the occurrences of the pattern
string in the text
           *         string
           */
          public List<Integer> searchAll(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

              List<Integer> list = new ArrayList<>();
              final int m = pat.length;
              final int n = txt.length();
              int j, k;
              final char[] txtArr = new char[n + m];

              for (int i = 0; i < n; ++i) {
                  txtArr[i] = txt.charAt(i);
              }
              Arrays.fill(txtArr, n, n + m, pat[m - 1]);
              j = 0;
              while (j < n) {
                  k = bmBc[txtArr[j + m -1]];
                  while (k !=  0) {
                      j += k;
                      k = bmBc[txtArr[j + m - 1]];
                      j += k;
                      k = bmBc[txtArr[j + m - 1]];
                      j += k;
                      k = bmBc[txtArr[j + m - 1]];
                  }
                  if (memcmp(pat, 0, txtArr, j, m - 1) && j < n) { //
CHANGED :: j < n - m  CHANGED :: j < n
                      list.add(j);
                  }
                  j += shift;                           /* shift */
              }
              return list;
          }

          // construct bad character shift array
          private void preBmBc() {
              final int m = pat.length;
              for (int i = 0; i < ASIZE; ++i) {
                  bmBc[i] = m;
              }
              for (int i = 0; i < m - 1; ++i) {
                  bmBc[pat[i]] = m - i - 1;
              }
          }

          // returns true if elements of two arrays-starting from
specified indices to a common length- are equal; false otherwise
          private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
```

```
                            return false;
                        }
                    }
                    return true;
                }

        }
```

7.1.19 Zhu-Takaoka

```
        package com.accel.stringsearch;

        import java.util.*;

        /**
         *  The {@code ZhuTakaoka} class finds the occurrences of a
pattern string in a
         * text string.
         * <p>
         * This implementation is a variant of the Boyer-Moore algorithm.
         * This implementation provides methods for retrieving the first
occurrence of
         * the pattern in the text and for retrieving all the occurrences
of the pattern
         * in the text. It takes time proportional to <em>nm</em> in the
worst case and
         * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
         * <em>m</em> is the length of the pattern. The preprocessing
phase takes <em>m
         * + σ^2</em> time and space, where σ is the size of the
alphabet.
         * </p>
         *
         */

        public class ZhuTakaoka {

            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] bmGs;
            private final int[] suff;
            private final int[][] ztBc;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat the pattern string
             */
            public ZhuTakaoka(String pat) {
              if        (pat        ==        null)        throw        new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                final int m = pat.length();
                this.pattern = pat.toCharArray();
                this.bmGs = new int[m];
                this.suff = new int[m];
                this.ztBc = new int[ASIZE][ASIZE];
```

```
            /* Preprocessing */
            preZtBc();
            preBmGs();
        }

        /**
         * Returns the index of the first occurrence of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the index of the first occurrence of the pattern
string in the text
         *         string; n if no such match
         */
        public int search(String txt) {
          if (txt == null) throw new IllegalArgumentException("text
is null");
                if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i, j;

            j = 0;
            while (j <= n - m) {
                i = m - 1;
                while (i >= 0 && pattern[i] == text[i + j]) {
                    --i;
                }
                if (i < 0) {
                    return j;
                }
                else if (j + m - 2 >= 0) {
                      j += Math.max(bmGs[i],
                            ztBc[text[j + m - 2]][text[j + m - 1]]);
                }
                else {
                    j += bmGs[i];
                }
            }
            return n;
        }

        /**
         * Returns the indices of all the occurrences of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the indices of all the occurrences of the pattern
string in the text
         *         string
         */
        public List<Integer> searchAll(String txt) {
          if (txt == null) throw new IllegalArgumentException("text
is null");
```

```
                if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i, j;

            j = 0;
            while (j <= n - m) {
                i = m - 1;
                while (i >= 0 && pattern[i] == text[i + j]) {
                    --i;
                }
                if (i < 0) {
                    list.add(j);
                    j += bmGs[0];
                }
                else if (j + m - 2 >= 0) {
                      j += Math.max(bmGs[i],
                            ztBc[text[j + m - 2]][text[j + m - 1]]);
                }
                else {
                    j += bmGs[i];
                }
            }
            return list;
        }

        private void preZtBc() {
            final int m = pattern.length;
            for (int i = 0; i < ASIZE; ++i) {
                for (int j = 0; j < ASIZE; ++j) {
                    ztBc[i][j] = m;
                }
            }
            for (int i = 0; i < ASIZE; ++i) {
                ztBc[i][pattern[0]] = m - 1;
            }
            for (int i = 1; i < m - 1; ++i) {
                ztBc[pattern[i - 1]][pattern[i]] = m - 1 - i;
            }
        }

        private void suffixes() {
            final int m = pattern.length;
            int f, g;

            suff[m - 1] = m;
            g = m - 1;
            f = 0;
            for (int i = m - 2; i >= 0; --i) {
                if (i > g && suff[i + m - 1 - f] < i - g) {
                    suff[i] = suff[i + m - 1 - f];
                } else {
                    if (i < g) {
                        g = i;
                    }
                    f = i;
```

```
                    while (g >= 0 && pattern[g] == pattern[g + m - 1
- f]) {
                        --g;
                    }
                    suff[i] = f - g;
                }
            }

        }

        // construct good suffix shift array
        private void preBmGs() {
            int j;
            final int m = pattern.length;

            suffixes();

            for (int i = 0; i < m; ++i) {
                bmGs[i] = m;
            }
            j = 0;
            for (int i = m - 1; i >= 0; --i) {
                if (suff[i] == i + 1) {
                    for (; j < m - 1 - i; ++j) {
                        if (bmGs[j] == m) {
                            bmGs[j] = m - 1 - i;
                        }
                    }
                }
            }
            for (int i = 0; i <= m - 2; ++i) {
                bmGs[m - 1 - suff[i]] = m - 1 - i;
            }

        }

    }
```

7.1.20 Berry-Ravindran

```
    package com.accel.stringsearch;

    import java.util.*;

    /**
     * The {@code BerryRavindran} class finds the occurrences of a
pattern string in
     * a text string.
     * <p>
     * This implementation is a hybrid of the Quick Search and Zhu-
Takaoka
     *  algorithms.  This  implementation  provides  methods  for
retrieving the first
     * occurrence of the pattern in the text and for retrieving all
the occurrences
     * of the pattern in the text. It takes time proportional to
<em>nm</em> in the
     * worst case and <em>n/m</em> in the best case, where <em>n</em>
is the length
     * of the text and <em>m</em> is the length of the pattern. The
preprocessing
```

```
       * phase takes <em>m + σ^2</em> time and space, where σ is the
size of the
       * alphabet.
       * </p>
       *
       */

      public class BerryRavindran {

            private static final int ASIZE = 256;
            private static char NULL_CHARACTER = '\0';

            private final char[] pattern;
            private final int[][] brBc;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public BerryRavindran(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  this.brBc = new int[ASIZE][ASIZE];

                  /* Preprocessing */
                  preBrBc();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final int n = txt.length();
                  final char[] text = new char[n + 1];
                  for (int i = 0; i < n; ++i) {
                        text[i] = txt.charAt(i);
                  }
                  text[n] = NULL_CHARACTER;
                  final int m = pattern.length;
                  int j;
```

```
                j = 0;
                while (j < n - m) {
                        if (memcmp(pattern, 0, text, j, m)) {
                                return j;
                        }
                        j += brBc[text[j + m]][text[j + m + 1]];
                }
                if (j == n - m) {
                        if (memcmp(pattern, 0, text, j, m)) {
                                return j;
                        }
                }
                return n;
        }

        /**
         * Returns the indices of all the occurrences of the
pattern string in the text
         * string.
         *
         * @param txt
         *            the text string
         * @return the indices of all the occurrences of the
pattern string in the text
         *         string
         */
        public List<Integer> searchAll(String txt) {
                if (txt == null) throw new
IllegalArgumentException("text is null");
                if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                List<Integer> list = new ArrayList<>();
                final int n = txt.length();
                final char[] text = new char[n + 1];
                for (int i = 0; i < n; ++i) {
                        text[i] = txt.charAt(i);
                }
                text[n] = NULL_CHARACTER;
                final int m = pattern.length;
                int j;

                j = 0;
                while (j < n - m) {
                        if (memcmp(pattern, 0, text, j, m)) {
                                list.add(j);
                        }
                        j += brBc[text[j + m]][text[j + m + 1]];
                }
                if (j == n - m) {
                        if (memcmp(pattern, 0, text, j, m)) {
                                list.add(j);
                        }
                }
                return list;
        }

        private void preBrBc() {
                final int m = pattern.length;
```

```
                for (int a = 0; a < ASIZE; ++a) {
                      for (int b = 0; b < ASIZE; ++b) {
                            brBc[a][b] = m + 2;
                      }
                }
                for (int a = 0; a < ASIZE; ++a) {
                      brBc[a][pattern[0]] = m + 1;
                }
                for (int i = 0; i < m - 1; ++i) {
                      brBc[pattern[i]][pattern[i + 1]] = m - i;
                }
                for (int a = 0; a < ASIZE; ++a) {
                      brBc[pattern[m - 1]][a] = 1;
                }
          }

          // returns true if elements of two arrays-starting from
specified indices to a
          // common length- are equal; false otherwise
          private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                for (int i = 0; i < length; ++i) {
                      if (x[startX + i] != y[startY + i]) {
                            return false;
                      }
                }
                return true;
          }
     }
```

7.1.21 Smith

```
     package com.accel.stringsearch;

     import java.util.*;

     /**
      * The {@code Smith} class finds the occurrences of a pattern
string in a
      * text string.
      * <p>
      * This implementation is a hybrid of the Horspool and the
QuickSearh algorithms;
      * it takes the maximum of the Horspool bad-character shift
function and the
      * Quick Search bad-character shift function.
      * This implementation provides methods for retrieving the first
occurrence of
      * the pattern in the text and for retrieving all the occurrences
of the pattern
      * in the text. The preprocessing phase takes <em>m
      * + σ</em> time and <em>σ</em>space, where σ is the size of the
alphabet.
      * </p>
      *
      */

     public class SMiTH {

         private static final int ASIZE = 256;
```

```
          private final char[] pattern;
          private final int[] bmBc;
          private final int[] qsBc;

          /**
           * Preprocesses the pattern string.
           *
           * @param pat the pattern string
           */
          public SMiTH(String pat) {
            if        (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

              this.pattern = pat.toCharArray();
              this.bmBc = new int[ASIZE];
              this.qsBc = new int[ASIZE];

              preBmBc();
              preQsBc();
          }

          /**
           * Returns the index of the first occurrence of the pattern
string in the text
           * string.
           *
           * @param txt the text string
           * @return the index of the first occurrence of the pattern
string in the text
           *         string; n if no such match
           */
          public int search(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
                if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

              final char[] text = txt.toCharArray();
              final int m = pattern.length;
              final int n = text.length;
              int i;

              i = 0;
              while (i < n - m) {
                  if (memcmp(pattern, 0, text, i, m)){
                      return i;
                  }
                  i += Math.max(bmBc[text[i + m - 1]], qsBc[text[i +
m]]);
              }
              if (i == n - m) {
                  if (memcmp(pattern, 0, text, i, m)){
                      return i;
                  }
              }
              return n;
          }
```

```
          /**
           * Returns the indices of all the occurrences of the pattern
string in the text
           * string.
           *
           * @param txt the text string
           * @return the indices of all the occurrences of the pattern
string in the text
           *         string
           */
          public List<Integer> searchAll(String txt) {
            if (txt == null) throw new IllegalArgumentException("text
is null");
                if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

              List<Integer> list = new ArrayList<>();
              final char[] text = txt.toCharArray();
              final int m = pattern.length;
              final int n = text.length;
              int i;

              i = 0;
              while (i < n - m) {
                  if (memcmp(pattern, 0, text, i, m)){
                      list.add(i);
                  }
                  i += Math.max(bmBc[text[i + m - 1]], qsBc[text[i +
m]]);
              }
              if (i == n - m) {
                  if (memcmp(pattern, 0, text, i, m)){
                      list.add(i);
                  }
              }
              return list;
          }

          // construct bad character shift array
          private void preBmBc() {
              final int m = pattern.length;
              for (int i = 0; i < ASIZE; ++i) {
                  bmBc[i] = m;
              }
              for (int i = 0; i < m - 1; ++i) {
                  bmBc[pattern[i]] = m - i - 1;
              }
          }

          // preprocessing
          private void preQsBc() {
              final int m = pattern.length;
              for (int i = 0; i < ASIZE; ++i)
                  qsBc[i] = m + 1;
              for (int i = 0; i < m; ++i)
                  qsBc[pattern[i]] = m - i;
          }

          // returns true if elements of two arrays-starting from
specified indices to a common length- are equal; false otherwise
```

```
          private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }

      }
```

7.1.22 Raita

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code Raita} class finds the occurrencies of a pattern
string in a
       * text string.
       * <p>
       * This implementation first compares the last pattern character,
then the
       * first and finally the middle one before actually comparing the
others.
       * This implementation provides methods for retrieving the first
occurence of
       * the pattern in the text and for retrieving all the occurencies
of the pattern
       * in the text. It takes time proportional to <em>nm</em> in the
worst case and
       * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
       * <em>m</em> is the length of the pattern. The preprocessing
phase takes <em>m
       * + σ</em> time and <em>σ</em> space, where σ is the size of the
alphabet.
       * </p>
       *
       */

      public class Raita {

          private static final int ASIZE = 256;

          private final char[] pattern;
          private final int[] bmBc;
          private final int firstChar, middleChar, lastChar;

          /**
           * Preprocesses the pattern string.
           *
           * @param pat the pattern string
           */
          public Raita(String pat) {
            if        (pat        ==        null)        throw        new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");
```

```
            final int m = pat.length();
            this.pattern = pat.toCharArray();
            this.bmBc = new int[ASIZE];

            /* Preprocessing */
            preBmBc();
            this.firstChar = pattern[0];
            this.middleChar = pattern[m / 2];
            this.lastChar = pattern[m - 1];
        }

        /**
         * Returns the index of the first occurrence of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the index of the first occurrence of the pattern
string in the text
         *         string; n if no such match
         */
        public int search(String txt) {
          if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i;

            i = 0;
            while (i <= n - m) {
                char c = text[i + m - 1];
                if (lastChar == c && firstChar == text[i] &&
middleChar == text[i + m / 2] && memcmp(pattern, 1, text, i + 1, m -
2)) {
                    return i;
                }
                i += bmBc[c];
            }
            return n;
        }

        /**
         * Returns the indices of all the occurrences of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the indices of all the occurrences of the pattern
string in the text
         *         string
         */
        public List<Integer> searchAll(String txt) {
          if (txt == null) throw new IllegalArgumentException("text
is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");
```

```
            List<Integer> list = new ArrayList<>();
            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i;

            i = 0;
            while (i <= n - m) {
                char c = text[i + m - 1];
                if (lastChar == c && firstChar == text[i] &&
middleChar == text[i + m / 2] && memcmp(pattern, 1, text, i + 1, m -
2)) {
                    list.add(i);
                }
                i += bmBc[c];
            }
            return list;
        }

        // construct bad character shift array
        private void preBmBc() {
            final int m = pattern.length;
            for (int i = 0; i < ASIZE; ++i) {
                bmBc[i] = m;
            }
            for (int i = 0; i < m - 1; ++i) {
                bmBc[pattern[i]] = m - i - 1;
            }
        }

        // returns true if elements of two arrays-starting from
specified indices to a common length- are equal; false otherwise
        private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                for (int i = 0; i < length; ++i) {
                    if (x[startX + i] != y[startY + i]) {
                        return false;
                    }
                }
                return true;
            }
    }
```

7.1.23 Reverse Factor

```
    package com.accel.stringsearch;

    import java.util.*;

    import com.accel.utils.Graph;
    import com.accel.utils.SuffixAutomaton;

    /**
     * The {@code ReverseFactor} class finds the occurrences of a
pattern string in
     * a text string.
     * <p>
     * This implementation provides methods for retrieving the first
occurrence of
     * the pattern in the text and for retrieving all the occurrences
of the pattern
```

```
       * in the text. It takes time proportional to <em>nm</em>, where
<em>n</em> is
       * the length of the text and <em>m</em> is the length of the
pattern. The
       * preprocessing phase takes <em>m</em> time and space. The
algorithm is optimal
       * at average.
       * </p>
       *
       */

      public class ReverseFactor {

            private static final int ASIZE = 256;

            private final char[] pattern;
            private final char[] patternR;
            private final int init;
            private final SuffixAutomaton aut;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat the pattern string
             */
            public ReverseFactor(String pat) {
                  if (pat == null)
                        throw new IllegalArgumentException("pattern is
null");
                  if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");

                  final int m = pat.length();
                  this.pattern = pat.toCharArray();
                  /* Preprocessing */
                  aut = new SuffixAutomaton(2 * (m + 2), 2 * (m + 2) *
ASIZE);
                  patternR = reverse();
                  buildSuffixAutomaton();
                  init = aut.getInitial();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");
```

```
                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, state, shift;

                /* Searching */
                j = 0;
                while (j <= n - m) {
                      i = m - 1;
                      state = init;
                      shift = m;
                      while (i + j >= 0 && aut.getTarget(state,
text[i + j]) != Graph.UNDEFINED) {
                            state = aut.getTarget(state, text[i +
j]);
                            if (aut.isTerminal(state)) {
                                  shift = i;
                            }
                            --i;
                      }
                      if (i < 0) {
                            return j;
                      }
                      j += shift;
                }
                return n;
          }

          /**
           * Returns the indices of all the occurrences of the
pattern string in the text
           * string.
           *
           * @param txt the text string
           * @return the indices of all the occurrences of the
pattern string in the text
           *         string
           */
          public List<Integer> searchAll(String txt) {
                if (txt == null)
                      throw new IllegalArgumentException("text is
null");
                if (txt.equals(""))
                      throw new IllegalArgumentException("text is
empty");

                List<Integer> list = new ArrayList<>();
                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, state, shift, period;

                /* Searching */
                period = m;
                j = 0;
                while (j <= n - m) {
                      i = m - 1;
                      state = init;
                      shift = m;
```

```
                        while (i + j >= 0 && aut.getTarget(state,
text[i + j]) != Graph.UNDEFINED) {
                              state = aut.getTarget(state, text[i +
j]);
                              if (aut.isTerminal(state)) {
                                    period = shift;
                                    shift = i;
                              }
                              --i;
                        }
                        if (i < 0) {
                              list.add(j);
                              shift = period;
                        }
                        j += shift;
                  }
                  return list;
            }

            private void buildSuffixAutomaton() {
                  final int m = pattern.length;
                  int art, init, last, p, q, r;
                  char c;

                  init = aut.getInitial();
                  art = aut.newVertex();
                  aut.setSuffixLink(init, art);
                  last = init;
                  for (int i = 0; i < m; ++i) {
                        c = patternR[i];
                        p = last;
                        q = aut.newVertex();
                        aut.setLength(q, aut.getLength(p) + 1);
                        aut.setPosition(q, aut.getPosition(p) + 1);
                        while (p != init && aut.getTarget(p, c) ==
Graph.UNDEFINED) {
                              aut.setTarget(p, c, q);
                              aut.setShift(p, c, aut.getPosition(q) -
aut.getPosition(p) - 1);
                              p = aut.getSuffixLink(p);
                        }
                        if (aut.getTarget(p, c) == Graph.UNDEFINED) {
                              aut.setTarget(init, c, q);
                              aut.setShift(init, c, aut.getPosition(q)
- aut.getPosition(init) - 1);
                              aut.setSuffixLink(q, init);
                        } else {
                              if (aut.getLength(p) + 1 ==
aut.getLength(aut.getTarget(p, c))) {
                                    aut.setSuffixLink(q,
aut.getTarget(p, c));
                              } else {
                                    r = aut.newVertex();
                                    aut.copyVertex(r, aut.getTarget(p,
c));
                                    aut.setLength(r, aut.getLength(p) +
1);
                                    aut.setSuffixLink(aut.getTarget(p,
c), r);
                                    aut.setSuffixLink(q, r);
```

```
                                    while (p != art &&
aut.getLength(aut.getTarget(p, c)) >= aut.getLength(r)) {
                                          aut.setShift(p, c,
aut.getPosition(aut.getTarget(p, c)) - aut.getPosition(p) - 1);
                                          aut.setTarget(p, c, r);
                                          p = aut.getSuffixLink(p);
                                    }
                              }
                        }
                        last = q;
                  }
                  aut.setTerminal(last);
                  while (last != init) {
                        last = aut.getSuffixLink(last);
                        aut.setTerminal(last);
                  }
            }

            private char[] reverse() {
                  final int m = pattern.length;
                  final char[] patternR = new char[m + 1];
                  for (int i = 0; i < m; ++i) {
                        patternR[i] = pattern[m - 1 - i];
                  }
                  patternR[m] = '\0';
                  return patternR;
            }
      }
```

7.1.24 Turbo Reverse Factor

```
      package com.accel.stringsearch;

      import java.util.*;

      import com.accel.utils.Graph;
      import com.accel.utils.SuffixAutomaton;

      /**
       * The {@code TurboReverseFactor} class finds the occurrences of
a pattern
       * string in a text string.
       * <p>
       * This implementation is a refinement of the Reverse Factor
algorithm. This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It takes time proportional to <em>n</em>, where
<em>n</em> is the
       * length of the text.The preprocessing phase takes <em>m</em>
time and space,
       * where <em>m</em> is the length of the pattern. At worst case
<em>2n</em> text
       * character comparisons are performed but the algorithm is
optimal at average.
       * </p>
       *
       */

      public class TurboReverseFactor {
```

```
            private static final int ASIZE = 256;

            private final char[] pattern;
            private final char[] patternR;
            private int[] mpNext; // shift array
            private final int init;
            private final int period;
            private final SuffixAutomaton aut;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public TurboReverseFactor(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  final int m = pat.length();
                  this.pattern = pat.toCharArray();
                  this.mpNext = new int[m + 1]; // TODO :: check if
needs to be m
                  aut = new SuffixAutomaton(2 * (m + 2), 2 * (m + 2) *
ASIZE);
                  patternR = reverse();
                  buildSuffixAutomaton();
                  init = aut.getInitial();
                  preMp();
                  period = m - mpNext[m];
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, shift, u, periodOfU, disp, state, mu;

                  i = 0;
                  shift = m;
                  j = 0;
```

```
                  while (j <= n - m) {
                        i = m - 1;
                        state = init;
                        u = m - 1 - shift;
                        periodOfU = (shift != m ? m - shift - mpNext[m
- shift] : 0);
                        shift = m;
                        disp = 0;
                        while (i > u && aut.getTarget(state, text[i +
j]) != Graph.UNDEFINED) {
                              disp += aut.getShift(state, text[i + j]);
                              state = aut.getTarget(state, text[i +
j]);
                              if (aut.isTerminal(state)) {
                                    shift = i;
                              }
                              --i;
                        }
                        if (i <= u) {
                              if (disp == 0) {
                                    return j;
                                    // shift = period;
                              } else {
                                    mu = (u + 1) / 2;
                                    if (periodOfU <= mu) {
                                          u -= periodOfU;
                                          while (i > u &&
aut.getTarget(state, text[i + j]) != Graph.UNDEFINED) {
                                                disp +=
aut.getShift(state, text[i + j]);
                                                state =
aut.getTarget(state, text[i + j]);
                                                if
(aut.isTerminal(state)) {
                                                      shift = i;
                                                }
                                                --i;
                                          }
                                          if (i <= u) {
                                                shift = disp;
                                          }
                                    } else {
                                          u = u - mu - 1;
                                          while (i > u &&
aut.getTarget(state, text[i + j]) != Graph.UNDEFINED) {
                                                disp +=
aut.getShift(state, text[i + j]);
                                                state =
aut.getTarget(state, text[i + j]);
                                                if
(aut.isTerminal(state)) {
                                                      shift = i;
                                                }
                                                --i;
                                          }
                                    }
                              }
                        }
                        j += shift;
                  }
```

```
                return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                if (txt == null) throw new
IllegalArgumentException("text is null");
                if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                List<Integer> list = new ArrayList<>();
                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, shift, u, periodOfU, disp, state, mu;

                i = 0;
                shift = m;
                j = 0;
                while (j <= n - m) {
                    i = m - 1;
                    state = init;
                    u = m - 1 - shift;
                    periodOfU = (shift != m ? m - shift - mpNext[m
- shift] : 0);
                    shift = m;
                    disp = 0;
                    while (i > u && aut.getTarget(state, text[i +
j]) != Graph.UNDEFINED) {
                        disp += aut.getShift(state, text[i + j]);
                        state = aut.getTarget(state, text[i +
j]);
                        if (aut.isTerminal(state)) {
                            shift = i;
                        }
                        --i;
                    }
                    if (i <= u) {
                        if (disp == 0) {
                            list.add(j);
                            shift = period;
                        } else {
                            mu = (u + 1) / 2;
                            if (periodOfU <= mu) {
                                u -= periodOfU;
                                while (i > u &&
aut.getTarget(state, text[i + j]) != Graph.UNDEFINED) {
                                    disp +=
aut.getShift(state, text[i + j]);
```

```
                                                state                   =
aut.getTarget(state, text[i + j]);
                                                if
(aut.isTerminal(state)) {
                                                      shift = i;
                                                }
                                                --i;
                                          }
                                          if (i <= u) {
                                                shift = disp;
                                          }
                                    } else {
                                          u = u - mu - 1;
                                          while     (i     >     u     &&
aut.getTarget(state, text[i + j]) != Graph.UNDEFINED) {
                                                disp                    +=
aut.getShift(state, text[i + j]);
                                                state                   =
aut.getTarget(state, text[i + j]);
                                                if
(aut.isTerminal(state)) {
                                                      shift = i;
                                                }
                                                --i;
                                          }
                                    }
                              }
                        }
                        j += shift;
                  }
                  return list;
            }

            private void buildSuffixAutomaton() {
                  final int m = pattern.length;
                  int art, init, last, p, q, r;
                  char c;

                  init = aut.getInitial();
                  art = aut.newVertex();
                  aut.setSuffixLink(init, art);
                  last = init;
                  for (int i = 0; i < m; ++i) {
                        c = patternR[i];
                        p = last;
                        q = aut.newVertex();
                        aut.setLength(q, aut.getLength(p) + 1);
                        aut.setPosition(q, aut.getPosition(p) + 1);
                        while  (p  !=  init  &&  aut.getTarget(p,  c)  ==
Graph.UNDEFINED) {
                              aut.setTarget(p, c, q);
                              aut.setShift(p,  c,  aut.getPosition(q)  -
aut.getPosition(p) - 1);
                              p = aut.getSuffixLink(p);
                        }
                        if (aut.getTarget(p, c) == Graph.UNDEFINED) {
                              aut.setTarget(init, c, q);
                              aut.setShift(init,  c,  aut.getPosition(q)
- aut.getPosition(init) - 1);
                              aut.setSuffixLink(q, init);
```

```
                        } else {
                              if     (aut.getLength(p)     +     1     ==
aut.getLength(aut.getTarget(p, c))) {
                                    aut.setSuffixLink(q,
aut.getTarget(p, c));
                              } else {
                                    r = aut.newVertex();
                                    aut.copyVertex(r,   aut.getTarget(p,
c));
                                    aut.setLength(r, aut.getLength(p) +
1);
                                    aut.setSuffixLink(aut.getTarget(p,
c), r);
                                    aut.setSuffixLink(q, r);
                                    while      (p      !=      art      &&
aut.getLength(aut.getTarget(p, c)) >= aut.getLength(r)) {
                                          aut.setShift(p,             c,
aut.getPosition(aut.getTarget(p, c)) - aut.getPosition(p) - 1);
                                          aut.setTarget(p, c, r);
                                          p = aut.getSuffixLink(p);
                                    }
                              }
                        }
                        last = q;
                  }
                  aut.setTerminal(last);
                  while (last != init) {
                        last = aut.getSuffixLink(last);
                        aut.setTerminal(last);
                  }
            }

            private char[] reverse() {
                  final int m = pattern.length;
                  final char[] patternR = new char[m + 1];
                  for (int i = 0; i < m; ++i) {
                        patternR[i] = pattern[m - 1 - i];
                  }
                  patternR[m] = '\0';
                  return patternR;
            }

            private void preMp() {
                  final int m = pattern.length;
                  mpNext[0] = -1;
                  int j = -1;
                  int i = 0;
                  while (i < m) {
                        while (j > -1 && pattern[i] != pattern[j]) {
                              j = mpNext[j];
                        }
                        mpNext[++i] = ++j;
                  }
            }
      }
```

7.1.25 Forward DAWG Matchinng

```
      package com.accel.stringsearch;

      import java.util.*;
```

```
import com.accel.utils.Graph;
import com.accel.utils.SuffixAutomaton;


/**
 * The {@code ForwardDawgMatching} class finds the occurrences of
a pattern
 * string in a text string.
 * <p>
 * This implementation provides methods for retrieving the first
occurrence of
 * the pattern in the text and for retrieving all the occurrences
of the pattern
 * in the text. It takes time proportional to <em>n</em>, where
<em>n</em> * is
 * the length of the text. It performs exactly n text character
inspections.
 * </p>
 *
 */

public class ForwardDawgMatching {

      private static final int ASIZE = 256;

      private final char[] pattern;
      private final int init;
      private final SuffixAutomaton aut;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public ForwardDawgMatching(String pat) {
            if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
            if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

            final int m = pat.length();
            this.pattern = pat.toCharArray();
            aut = new SuffixAutomaton(2 * (m + 2), 2 * (m + 2) *
ASIZE);
            buildSuffixAutomaton();
            init = aut.getInitial();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
```

```
            public int search(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int j, ell, state;

                  /* Searching */
                  ell = 0;
                  state = init;
                  for (j = 0; j < n; ++j) {
                        if     (aut.getTarget(state,     text[j])     !=
Graph.UNDEFINED) {
                              ++ell;
                              state = aut.getTarget(state, text[j]);
                        } else {
                              while      (state      !=      init      &&
aut.getTarget(state, text[j]) == Graph.UNDEFINED) {
                                    state = aut.getSuffixLink(state);
                              }
                              if   (aut.getTarget(state,   text[j])   !=
Graph.UNDEFINED) {
                                    ell = aut.getLength(state) + 1;
                                    state     =     aut.getTarget(state,
text[j]);
                              } else {
                                    ell = 0;
                                    state = init;
                              }
                        }
                        if (ell == m) {
                              return j - m + 1;
                        }
                  }
                  return n;
            }

            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             *  @return  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
```

```
                final int m = pattern.length;
                final int n = text.length;
                int j, ell, state;

                /* Searching */
                ell = 0;
                state = init;
                for (j = 0; j < n; ++j) {
                      if     (aut.getTarget(state,     text[j])     !=
Graph.UNDEFINED) {
                            ++ell;
                            state = aut.getTarget(state, text[j]);
                      } else {
                            while      (state      !=      init      &&
aut.getTarget(state, text[j]) == Graph.UNDEFINED) {
                                  state = aut.getSuffixLink(state);
                            }
                            if   (aut.getTarget(state,   text[j])   !=
Graph.UNDEFINED) {
                                  ell = aut.getLength(state) + 1;
                                  state     =     aut.getTarget(state,
text[j]);
                            } else {
                                  ell = 0;
                                  state = init;
                            }
                      }
                      if (ell == m) {
                            list.add(j - m + 1);
                      }
                }
                return list;
          }

          private void buildSuffixAutomaton() {
                final int m = pattern.length;
                int art, init, last, p, q, r;
                char c;

                init = aut.getInitial();
                art = aut.newVertex();
                aut.setSuffixLink(init, art);
                last = init;
                for (int i = 0; i < m; ++i) {
                      c = pattern[i];
                      p = last;
                      q = aut.newVertex();
                      aut.setLength(q, aut.getLength(p) + 1);
                      aut.setPosition(q, aut.getPosition(p) + 1);
                      while  (p  !=  init  &&  aut.getTarget(p,  c)  ==
Graph.UNDEFINED) {
                            aut.setTarget(p, c, q);
                            aut.setShift(p,  c,  aut.getPosition(q)  -
aut.getPosition(p) - 1);
                            p = aut.getSuffixLink(p);
                      }
                      if (aut.getTarget(p, c) == Graph.UNDEFINED) {
                            aut.setTarget(init, c, q);
                            aut.setShift(init,  c,  aut.getPosition(q)
- aut.getPosition(init) - 1);
```

```
                                aut.setSuffixLink(q, init);
                          } else {
                                if     (aut.getLength(p)     +     1     ==
aut.getLength(aut.getTarget(p, c))) {
                                      aut.setSuffixLink(q,
aut.getTarget(p, c));
                                } else {
                                      r = aut.newVertex();
                                      aut.copyVertex(r,   aut.getTarget(p,
c));
                                      aut.setLength(r, aut.getLength(p) +
1);
                                      aut.setSuffixLink(aut.getTarget(p,
c), r);
                                      aut.setSuffixLink(q, r);
                                      while      (p      !=      art      &&
aut.getLength(aut.getTarget(p, c)) >= aut.getLength(r)) {
                                            aut.setShift(p,             c,
aut.getPosition(aut.getTarget(p, c)) - aut.getPosition(p) - 1);
                                            aut.setTarget(p, c, r);
                                            p = aut.getSuffixLink(p);
                                      }
                                }
                          }
                          last = q;
                    }
                    aut.setTerminal(last);
                    while (last != init) {
                          last = aut.getSuffixLink(last);
                          aut.setTerminal(last);
                    }
              }
        }
```

7.1.26 Backward Nondeterministic DAWG Matching

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.List;

      import com.accel.exceptions.ExceedingWordSizeException;

      /**
       * The {@code BackwardNonDeterministicDawgMatching} class finds
the occurrences
       * of a pattern string in a text string.
       * <p>
       *  This implementation is a variant of the Reverse Factor
algorithm. This
       *  implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It is efficient if the pattern length is no longer
than the
       * memory-word size of the machine.
       * </p>
       *
       */

      public class BackwardNonDeterministicDawgMatching {
```

```
            private static final int ASIZE = 256;
            private static final int WORD_SIZE = 31;

            private final char[] pattern;
            private final int[] B;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public BackwardNonDeterministicDawgMatching(String pat) {
                  if (pat == null)
                        throw new IllegalArgumentException("pattern is
null");
                  if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");
                  if (pat.length() > WORD_SIZE)
                        throw new ExceedingWordSizeException(
                                    "pattern length: " + pat.length() +
" is greater than word size: " + WORD_SIZE);

                  this.pattern = pat.toCharArray();
                  this.B = new int[ASIZE];
                  preBNDDM();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, last, d;

                  /* Searching phase */
                  j = 0;
                  while (j <= n - m) {
                        i = m - 1;
                        last = m;
                        d = ~0;
```

```
                        while (i >= 0 && d != 0) {
                              d &= B[text[j + i]];
                              i--;
                              if (d != 0) {
                                    if (i >= 0) {
                                          last = i + 1;
                                    } else {
                                          return j;
                                    }
                              }
                              d <<= 1;
                        }
                        j += last;
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                  if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, last, d;

                  /* Searching phase */
                  j = 0;
                  while (j <= n - m) {
                        i = m - 1;
                        last = m;
                        d = ~0;
                        while (i >= 0 && d != 0) {
                              d &= B[text[j + i]];
                              i--;
                              if (d != 0) {
                                    if (i >= 0) {
                                          last = i + 1;
                                    } else {
                                          list.add(j);
                                    }
                              }
                              d <<= 1;
                        }
```

```
                        j += last;
                }
                return list;
        }

        private void preBNDDM() {
                final int m = pattern.length;
                int s, i;
                s = 1;
                for (i = m - 1; i >= 0; --i) {
                        B[pattern[i]] |= s;
                        s <<= 1;
                }
        }
}
```

7.1.27 Backward Oracle Matching

```
package com.accel.stringsearch;

import java.util.ArrayList;
import java.util.List;

import com.accel.utils.Graph;

/**
 * The {@code BackwardOracleMatching} class finds the occurrences
of a pattern
 * string in a text string.
 * <p>
 * This implementation provides methods for retrieving the first
occurrence of
 * the pattern in the text and for retrieving all the occurrences
of the pattern
 * in the text. It takes time proportional to <em>nm</em>, where
<em>n</em> is
 * the length of the text and <em>m</em> is the length of the
pattern. The
 * preprocessing phase takes <em>m</em> time and space. The
algorithm is optimal
 * at average.
 * </p>
 *
 */
public class BackwardOracleMatching {

        private final char[] pattern;
        private final boolean[] T;
        private final Cell[] L;

        private static class Cell {
                private int element;
                private Cell next;
        }

        /**
         * Preprocesses the pattern string.
         *
         * @param pat the pattern string
         */
        public BackwardOracleMatching(String pat) {
                if (pat == null)
```

```
                        throw new IllegalArgumentException("pattern is
null");
                if (pat.equals(""))
                        throw new IllegalArgumentException("pattern is
empty");

                final int m = pat.length();
                this.pattern = pat.toCharArray();
                /* Preprocessing */
                T = new boolean[m + 1];
                L = new Cell[m + 1];
                oracle();
        }

        /**
         * Returns the index of the first occurrence of the pattern
string in the text
         * string.
         *
         * @param txt the text string
         * @return the index of the first occurrence of the pattern
string in the text
         *         string; n if no such match
         */
        public int search(String txt) {
                if (txt == null)
                        throw new IllegalArgumentException("text is
null");
                if (txt.equals(""))
                        throw new IllegalArgumentException("text is
empty");

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, p, q, shift;

                /* Searching */
                j = 0;
                while (j <= n - m) {
                        i = m - 1;
                        p = m;
                        shift = m;
                        while (i + j >= 0 && (q = getTransition(p,
text[i + j])) != Graph.UNDEFINED) {
                                p = q;
                                if (T[p] == true) {
                                        shift = i;
                                }
                                --i;
                        }
                        if (i < 0) {
                                return j;
      //                        shift = period;
                        }
                        j += shift;
                }
                return n;
        }
```

```
            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt the text string
             *  @return  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, p, q, shift, period;// , state, shift/* ,
period */;

                  /* Searching */
                  j = 0;
                  period = 0;
                  while (j <= n - m) {
                        i = m - 1;
                        p = m;
                        shift = m;
                        while  (i  +  j  >=  0  &&  (q  =  getTransition(p,
text[i + j])) != Graph.UNDEFINED) {
                              p = q;
                              if (T[p] == true) {
                                    period = shift;
                                    shift = i;
                              }
                              --i;
                        }
                        if (i < 0) {
                              shift = period;
                              list.add(j);
                        }
                        j += shift;
                  }
                  return list;
            }

            private void oracle() {
                  final int m = pattern.length;
                  int i, p, q;
                  int[] S = new int[m + 1];
                  char c;

                  q = 0;
                  S[m] = m + 1;
                  for (i = m; i > 0; --i) {
                        c = pattern[i - 1];
```

```
                        p = S[i];
                        while (p <= m && (q = getTransition(p, c)) ==
Graph.UNDEFINED) {
                              setTransition(p, i - 1);
                              p = S[p];
                        }
                        S[i - 1] = (p == m + 1 ? m : q);
                  }
                  p = 0;
                  while (p <= m) {
                        T[p] = true;
                        p = S[p];
                  }
            }

            private int getTransition(int p, char c) {
                  Cell cell;

                  if (p > 0 && pattern[p - 1] == c)
                        return (p - 1);
                  else {
                        cell = L[p];
                        while (cell != null)
                              if (pattern[cell.element] == c)
                                    return (cell.element);
                              else
                                    cell = cell.next;
                        return (Graph.UNDEFINED);
                  }
            }

            private void setTransition(int p, int q) {
                  Cell cell = new Cell();
                  cell.element = q;
                  cell.next = L[p];
                  L[p] = cell;
            }
      }
```

7.1.28 Galil-Seiferas

```
      package com.accel.stringsearch;

      import java.util.*;

      /**
       * The {@code GalilSeiferas} class finds the occurrences of a
pattern string in
       * a text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. It takes time proportional to <em>n</em>, where
<em>n</em> is
       * the length of the text and. The preprocessing phase takes
<em>m</em> time and
       * constant space, where <em>m</em> is the length of the pattern.
There are at
       * most <em>5n</em> character comparisons at worst case.
       * </p>
```

```
 *
 */

public class GalilSeiferas {
      private static final int K = 4;

      private final char[] pattern;
      private char[] text;
      private final int m;
      private int n, p, p1, p2, q, q1, q2, s;
      private List<Integer> list;
      private int result;
      private boolean findAll, goOn;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public GalilSeiferas(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
            if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

            m = pat.length();
            pattern = pat.toCharArray();
      }

      /**
       * Returns the index of the first occurrence of the pattern
string in the text
       * string.
       *
       * @param txt
       *            the text string
       * @return the index of the first occurrence of the pattern
string in the text
       *         string; n if no such match
       */
      public int search(String txt) {
            if (txt == null) throw new
IllegalArgumentException("text is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            text = txt.toCharArray();
            n = text.length;
            findAll = false;
            result = n;
            goOn = true;
            init();
            return result;
      }

      /**
       * Returns the indices of all the occurrences of the
pattern string in the text
       * string.
```

```
             *
             * @param txt
             *            the text string
             *  @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  list = new ArrayList<>();
                  text = txt.toCharArray();
                  n = text.length;
                  findAll = true;
                  goOn = true;
                  init();
                  return list;
            }

            private void init() {
                  p = q = s = q1 = p2 = q2 = 0;
                  p1 = 1;
                  newP1();
            }

            private void newP1() {
                  while (s + p1 + q1 < m && pattern[s + q1] ==
pattern[s + p1 + q1]) {
                        ++q1;
                  }
                  if (p1 + q1 >= K * p1) {
                        p2 = q1;
                        q2 = 0;
                        newP2();
                        if (!goOn) {
                              return;
                        }
                  } else {
                        if (s + p1 + q1 == m) {
                              search();
                              if (!goOn) {
                                    return;
                              }
                        } else {
                              p1 += (q1 / K + 1);
                              q1 = 0;
                              newP1();
                        }
                  }
            }

            private void newP2() {
                  while (s + p2 + q2 < m && pattern[s + q2] ==
pattern[s + p2 + q2] && p2 + q2 < K * p2) {
                        ++q2;
                  }
                  if (p2 + q2 == K * p2) {
```

```
                        parse();
                        if (!goOn) {
                              return;
                        }
                  } else if (s + p2 + q2 == m) {
                        search();
                        if (!goOn) {
                              return;
                        }
                  } else {
                        if (q2 == p1 + q1) {
                              p2 += p1;
                              q2 -= p1;
                        } else {
                              p2 += (q2 / K + 1);
                              q2 = 0;
                        }
                        newP2();
                  }
            }

            private void parse() {
                  while (true) {
                        while (s + p1 + q1 < m && pattern[s + q1] ==
pattern[s + p1 + q1]) {
                              ++q1;
                        }
                        while (p1 + q1 >= K * p1) {
                              s += p1;
                              q1 -= p1;
                        }
                        p1 += (q1 / K + 1);
                        q1 = 0;
                        if (p1 >= p2) {
                              break;
                        }
                  }
                  newP1();
            }

            private void search() {
                  while (p <= n - m) {
                        while (s + q < m && p + s + q < n && pattern[s
+ q] == text[p + s + q]) {
                              ++q;
                        }
                        if (q == m - s && memcmp(pattern, 0, text, p, s
+ 1)) {
                              if (findAll) {
                                    list.add(p);
                              } else {
                                    result = p;
                                    goOn = false;
                                    return;
                              }
                        }
                        if (q == p1 + q1) {
                              p += p1;
                              q -= p1;
                        } else {
```

```
                        p += (q / K + 1);
                        q = 0;
                  }
            }
      }

      // returns true if elements of two arrays-starting from
specified indices to a
      // common length- are equal; false otherwise
      private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
            for (int i = 0; i < length; ++i) {
                  if (pattern[startX + i] != y[startY + i]) {
                        return false;
                  }
            }
            return true;
      }
}
```

7.1.29 Two Way

```
package com.accel.stringsearch;

import java.util.*;

/**
 * The {@code TwoWay} class finds the occurrences of a pattern
string in a text
 * string.
 * <p>
 * This implementation provides methods for retrieving the first
occurrence of
 * the pattern in the text and for retrieving all the occurrences
of the pattern
 * in the text. It takes time proportional to <em>n</em>, where
<em>n</em> is
 * the length of the text. The preprocessing phase takes
<em>m</em> time and
 * constant space, where m is the size of the pattern. The
maximum number of
 * text character comparisons. is <em>2n - m</em>. This algorithm
requires an
 * ordered alphabet.
 * </p>
 *
 */
public class TwoWay {

      private final char[] pattern;
      private final int ell;
      private int per;

      /**
       * Preprocesses the pattern string.
       *
       * @param pat
       *            the pattern string
       */
      public TwoWay(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
```

```
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  int[] maxSuf = maxSuf();
                  int[] maxSufTilde = maxSufTilde();
                  if (maxSuf[0] > maxSufTilde[0]) {
                        ell = maxSuf[0];
                        per = maxSuf[1];
                  } else {
                        ell = maxSufTilde[0];
                        per = maxSufTilde[1];
                  }
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, memory;
                  if (memcmp(pattern, 0, pattern, per, ell + 1)) {
                        j = 0;
                        memory = -1;
                        while (j <= n - m) {
                              i = Math.max(ell, memory) + 1;
                              while (i < m && pattern[i] == text[i +
j])
                                    ++i;
                              if (i >= m) {
                                    i = ell;
                                    while (i > memory && pattern[i] ==
text[i + j])
                                          --i;
                                    if (i <= memory)
                                          return j;
                                    j += per;
                                    memory = m - per - 1;
                              } else {
                                    j += (i - ell);
                                    memory = -1;
                              }
                        }
                  } else {
                        per = Math.max(ell + 1, m - ell - 1) + 1;
```

```
                        j = 0;
                        while (j <= n - m) {
                              i = ell + 1;
                              while (i < m && pattern[i] == text[i +
j])
                                    ++i;
                              if (i >= m) {
                                    i = ell;
                                    while (i >= 0 && pattern[i] ==
text[i + j])
                                          --i;
                                    if (i < 0)
                                          return j;
                                    j += per;
                              } else
                                    j += (i - ell);
                        }
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null) throw new
IllegalArgumentException("text is null");
                  if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, memory;
                  if (memcmp(pattern, 0, pattern, per, ell + 1)) {
                        j = 0;
                        memory = -1;
                        while (j <= n - m) {
                              i = Math.max(ell, memory) + 1;
                              while (i < m && pattern[i] == text[i +
j])
                                    ++i;
                              if (i >= m) {
                                    i = ell;
                                    while (i > memory && pattern[i] ==
text[i + j])
                                          --i;
                                    if (i <= memory)
                                          list.add(j);
                                    j += per;
                                    memory = m - per - 1;
```

```
                        } else {
                              j += (i - ell);
                              memory = -1;
                        }
                  }
            } else {
                  per = Math.max(ell + 1, m - ell - 1) + 1;
                  j = 0;
                  while (j <= n - m) {
                        i = ell + 1;
                        while (i < m && pattern[i] == text[i +
j])
                              ++i;
                        if (i >= m) {
                              i = ell;
                              while (i >= 0 && pattern[i] ==
text[i + j])
                                    --i;
                              if (i < 0)
                                    list.add(j);
                              j += per;
                        } else
                              j += (i - ell);
                  }
            }
            return list;
      }

      /* Computing of the maximal suffix for <= */
      private int[] maxSuf() {
            final int m = pattern.length;
            int ms, p; // the two variables that are returned
            int j, k;
            char a, b;
            int[] ret = new int[2];

            ms = -1;
            j = 0;
            k = p = 1;
            while (j + k < m) {
                  a = pattern[j + k];
                  b = pattern[ms + k];
                  if (a < b) {
                        j += k;
                        k = 1;
                        p = j - ms;
                  } else if (a == b)
                        if (k != p)
                              ++k;
                        else {
                              j += p;
                              k = 1;
                        }
                  else { /* a > b */
                        ms = j;
                        j = ms + 1;
                        k = p = 1;
                  }
            }
            ret[0] = ms;
```

```
            ret[1] = p;
            return ret;
        }

        /* Computing of the maximal suffix for >= */
        private int[] maxSufTilde() {
            final int m = pattern.length;
            int ms, p; // the two variables that are returned
            int j, k;
            char a, b;
            int[] ret = new int[2];

            ms = -1;
            j = 0;
            k = p = 1;
            while (j + k < m) {
                a = pattern[j + k];
                b = pattern[ms + k];
                if (a > b) {
                    j += k;
                    k = 1;
                    p = j - ms;
                } else if (a == b)
                    if (k != p)
                        ++k;
                    else {
                        j += p;
                        k = 1;
                    }
                else { /* a < b */
                    ms = j;
                    j = ms + 1;
                    k = p = 1;
                }
            }
            ret[0] = ms;
            ret[1] = p;
            return ret;
        }

        // returns true if elements of two arrays-starting from
specified indices to a
        // common length- are equal; false otherwise
        private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
            for (int i = 0; i < length; ++i) {
                if (x[startX + i] != y[startY + i]) {
                    return false;
                }
            }
            return true;
        }

    }
```

7.1.30 String Matching on Ordered Alphabets

```
    package com.accel.stringsearch;

    import java.util.*;
```

```
      /**
       * The {@code StringMatchingOnOrderedAlphabets} class finds the
occurrences of a
       * pattern string in a text string.
       * <p>
       * This implementation provides methods for retrieving the first
occurrence of
       * the pattern in the text and for retrieving all the occurrences
of the pattern
       * in the text. It takes time proportional to <em>n</em>, where
<em>n</em> is
       * the length of the text. The maximum number of text character
comparisons is
       * <em>6n + 5</em>. This algorithm requires an ordered alphabet.
       * </p>
       *
       */
      public class StringMatchingOnOrderedAlphabets {

            private final char[] pattern;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public StringMatchingOnOrderedAlphabets(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j;
                  int[] ip, jp, k, p;

                  ip = new int[1];
```

```
                  jp = new int[1];
                  k = new int[1];
                  p = new int[1];
                  /* Searching */
                  ip[0] = -1;
                  jp[0] = 0;
                  k[0] = 1;
                  p[0] = 1;
                  i = j = 0;
                  while (j <= n - m) {
                        while (i + j < n && i < m && pattern[i] ==
text[i + j])
                              ++i;
                        if (i == 0) {
                              ++j;
                              ip[0] = -1;
                              jp[0] = 0;
                              k[0] = p[0] = 1;
                        } else {
                              if (i >= m)
                                    return j;
                              nextMaximalSuffix(text, j, i + 1, ip, jp,
k, p);
                              if (ip[0] < 0 || (ip[0] < p[0] &&
memcmp(text, j, text, j + p[0], ip[0] + 1))) {
                                    j += p[0];
                                    i -= p[0];
                                    if (i < 0)
                                          i = 0;
                                    if (jp[0] - ip[0] > p[0])
                                          jp[0] -= p[0];
                                    else {
                                          ip[0] = -1;
                                          jp[0] = 0;
                                          k[0] = p[0] = 1;
                                    }
                              } else {
                                    j    +=    (Math.max(ip[0]    +    1,
Math.min(i - ip[0] - 1, jp[0] + 1)) + 1);
                                    i = jp[0] = 0;
                                    ip[0] = -1;
                                    k[0] = p[0] = 1;
                              }
                        }
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
```

```
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j;
                  int[] ip, jp, k, p;

                  ip = new int[1];
                  jp = new int[1];
                  k = new int[1];
                  p = new int[1];
                  /* Searching */
                  ip[0] = -1;
                  k[0] = p[0] = 1;
                  i = j = jp[0] = 0;
                  while (j <= n - m) {
                        while (i + j < n && i < m && pattern[i] ==
text[i + j])
                              ++i;
                        if (i == 0) {
                              ++j;
                              ip[0] = -1;
                              jp[0] = 0;
                              k[0] = p[0] = 1;
                        } else {
                              if (i >= m)
                                    list.add(j);
                              nextMaximalSuffix(text, j, i + 1, ip, jp,
k, p);
                              if  (ip[0]  <  0  ||  (ip[0]  <  p[0]  &&
memcmp(text, j, text, j + p[0], ip[0] + 1))) {
                                    j += p[0];
                                    i -= p[0];
                                    if (i < 0)
                                          i = 0;
                                    if (jp[0] - ip[0] > p[0])
                                          jp[0] -= p[0];
                                    else {
                                          ip[0] = -1;
                                          jp[0] = 0;
                                          k[0] = p[0] = 1;
                                    }
                              } else {
                                    j    +=    (Math.max(ip[0]    +    1,
Math.min(i - ip[0] - 1, jp[0] + 1)) + 1);
                                    i = jp[0] = 0;
                                    ip[0] = -1;
                                    k[0] = p[0] = 1;
                              }
                        }
                  }
                  return list;
            }

            /* Compute the next maximal suffix. */
```

```
            private void nextMaximalSuffix(char[] x, int xIndex, int m,
int[] i, int[] j, int[] k, int[] p) {
                  char a, b;
                  int xLength = x.length;
                  if (i.length != 1 || j.length != 1 || k.length != 1
|| p.length != 1) {
                        throw new IllegalArgumentException("all array
arguments must have exactly one element");
                  }

                  while (j[0] + k[0] < m && xIndex + i[0] + k[0] <
xLength && xIndex + j[0] + k[0] < xLength) {
                        a = x[xIndex + i[0] + k[0]];
                        b = x[xIndex + j[0] + k[0]];
                        if (a == b)
                              if (k[0] == p[0]) {
                                    (j[0]) += p[0];
                                    k[0] = 1;
                              } else
                                    ++(k[0]);
                        else if (a > b) {
                              (j[0]) += k[0];
                              k[0] = 1;
                              p[0] = j[0] - i[0];
                        } else {
                              i[0] = j[0];
                              ++(j[0]);
                              k[0] = p[0] = 1;
                        }
                  }
            }

            // returns true if elements of two arrays-starting from
specified indices to a
            // common length- are equal; false otherwise
            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  for (int i = 0; i < length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  return true;
            }

      }
```

7.1.31 Optimal Mismatch

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.Arrays;
      import java.util.Comparator;
      import java.util.List;

      /**
       * The {@code OptimalMismatch} class finds the occurrences of a
pattern string in a
       * text string.
       * <p>
```

```
       * This implementation is a variant of the QuickSearch algorithm.
This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It takes time proportional to <em>nm</em> in the
worst case and
       * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
       * <em>m</em> is the length of the pattern. The preprocessing
phase takes
       * <em>m^2 + σ</em> time and <em>m + σ</em> space, where σ is the
size of the
       * alphabet.
       * </p>
       *
       */
      public class OptimalMismatch {
            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] qsBc;

            private static class PatternScanElement {
                  private char c;
                  private int loc;
            }

            private     static     class     ByFreqLoc     implements
Comparator<PatternScanElement> {
                  private final int[] freq;

                  public ByFreqLoc(int[] freq) {
                        this.freq = freq;
                  }

                  public     int     compare(PatternScanElement     pse1,
PatternScanElement pse2) {
                        if (freq[pse1.c] > freq[pse2.c])
                              return 1;
                        if (freq[pse1.c] < freq[pse2.c])
                              return -1;
                        if (pse1.loc > pse2.loc)
                              return -1;
                        if (pse1.loc < pse2.loc)
                              return 1;
                        return 0;
                  }
            }

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public OptimalMismatch(String pat) {
                  if      (pat      ==      null)      throw      new
IllegalArgumentException("pattern is null");
```

```
                if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

                this.pattern = pat.toCharArray();
                this.qsBc = new int[ASIZE];
                preQsBc();
          }

          /**
           * Returns the index of the first occurrence of the pattern
string in the text
           * string.
           *
           * @param txt
           *            the text string
           * @return the index of the first occurrence of the pattern
string in the text
           *         string; n if no such match
           */
          public int search(String txt) {
                if (txt == null) throw new
IllegalArgumentException("text is null");
                if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                final int[] freq = new int[ASIZE];
                final Comparator<PatternScanElement> byFreqLoc;
                final PatternScanElement[] pse = new
PatternScanElement[m];
                final int[] adaptedGs = new int[m + 1];
                int i, j;

                /* Preprocessing */

                // get frequencies
                for (char c : text) {
                      freq[c]++;
                }

                // create comparator based on frequencies
                byFreqLoc = new ByFreqLoc(freq);

                orderPattern(pattern, pse, byFreqLoc);
                preAdaptedGs(pattern, adaptedGs, pse);

                /* Searching */
                j = 0;
                while (j <= n - m) {
                      i = 0;
                      while (i < m && pse[i].c == text[j +
pse[i].loc])
                            ++i;
                      if (i >= m)
                            return j;
                      if (j < n - m)
                            j += Math.max(adaptedGs[i], qsBc[text[j +
m]]);
```

```
                        else
                              j += adaptedGs[i];
                  }
                  return n;
            }

            /**
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if (txt == null) throw new
IllegalArgumentException("text is null");
                  if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  final int[] freq = new int[ASIZE];
                  final Comparator<PatternScanElement> byFreqLoc;
                  final PatternScanElement[] pse = new
PatternScanElement[m];
                  final int[] adaptedGs = new int[m + 1];
                  int i, j;

                  /* Preprocessing */

                  // get frequencies
                  for (char c : text) {
                        freq[c]++;
                  }

                  // create comparator based on frequencies
                  byFreqLoc = new ByFreqLoc(freq);

                  orderPattern(pattern, pse, byFreqLoc);
                  preAdaptedGs(pattern, adaptedGs, pse);

                  /* Searching */
                  j = 0;
                  while (j <= n - m) {
                        i = 0;
                        while (i < m && pse[i].c == text[j +
pse[i].loc])
                              ++i;
                        if (i >= m)
                              list.add(j);
                        if (j < n - m)
                              j += Math.max(adaptedGs[i], qsBc[text[j +
m]]);
                        else
```

```
                        j += adaptedGs[i];
                }
                return list;
        }

        /* Construct an ordered pattern from a string. */
        private    static    void    orderPattern(char[]    pattern,
PatternScanElement[] pse, Comparator<PatternScanElement> cmptor) {
                final int m = pattern.length;
                for (int i = 0; i < m; ++i) {
                        PatternScanElement        x        =        new
PatternScanElement();
                        x.c = pattern[i];
                        x.loc = i;
                        pse[i] = x;
                }
                Arrays.sort(pse, cmptor);
        }

        /*
         * Constructs the good-suffix shift table from an ordered
string.
         */
        private  static  void  preAdaptedGs(char[]  pattern,  int[]
adaptedGs, PatternScanElement[] pse) {
                final int m = pattern.length;
                int lshift, i, ploc;

                adaptedGs[0] = lshift = 1;
                for (ploc = 1; ploc <= m; ++ploc) {
                        lshift  =  matchShift(pattern,  ploc,  lshift,
pse);
                        adaptedGs[ploc] = lshift;
                }
                for (ploc = 0; ploc < m; ++ploc) {  //
                        lshift = adaptedGs[ploc];
                        while (lshift < m) {
                                i = pse[ploc].loc - lshift;
                                if (i < 0 || pse[ploc].c != pattern[i])
                                        break;
                                ++lshift;
                                lshift   =   matchShift(pattern,   ploc,
lshift, pse);
                        }
                        adaptedGs[ploc] = lshift;
                }
        }

        /*
         * Find the next leftward matching shift for the first ploc
pattern elements
         * after a current shift or lshift.
         */
        private static int matchShift(char[] pattern, int ploc, int
lshift, PatternScanElement[] pse) {
                final int m = pattern.length;
                int i, j;

                for (; lshift < m; ++lshift) {
                        i = ploc;
```

```
                        while (--i >= 0) {
                              if ((j = (pse[i].loc - lshift)) < 0)
                                    continue;
                              if (pse[i].c != pattern[j])
                                    break;
                        }
                        if (i < 0)
                              break;
                  }
                  return (lshift);
            }

            // preprocessing
            private void preQsBc() {
                  final int m = pattern.length;
                  for (int i = 0; i < ASIZE; ++i)
                        qsBc[i] = m + 1;
                  for (int i = 0; i < m; ++i)
                        qsBc[pattern[i]] = m - i;
            }

      }
```

7.1.32 Maximal Shift

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.Arrays;
      import java.util.Comparator;
      import java.util.List;

      /**
       * The {@code MaximalShift} class finds the occurrences of a
pattern string in a
       * text string.
       * <p>
       * This implementation is a variant of the QuickSearch algorithm.
This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It takes time proportional to <em>nm</em> in the
worst case and
       * <em>n/m</em> in the best case, where <em>n</em> is the length
of the text and
       * <em>m</em> is the length of the pattern. The preprocessing
phase takes
       * <em>m^2 + σ</em> time and <em>m + σ</em> space, where σ is the
size of the
       * alphabet. It has a quadratic waorst case time complexity.
       * </p>
       *
       */
      public class MaximalShift {
            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] qsBc;
            private final int[] minShift;
```

```
            private   static   class   PatternScanElement   implements
Comparable<PatternScanElement> {
                  private char c;
                  private int loc;

                  public int compareTo(PatternScanElement that) {
                        if (this.c > that.c)
                              return 1;
                        if (this.c < that.c)
                              return -1;
                        if (this.loc > that.loc)
                              return -1;
                        if (this.loc < that.loc)
                              return 1;
                        return 0;
                  }
            }

            private     static     class     ByFreqLoc     implements
Comparator<PatternScanElement> {
                  private final int[] minShift;

                  public ByFreqLoc(int[] minShift) {
                        this.minShift = minShift;
                  }

                  public    int    compare(PatternScanElement    pse1,
PatternScanElement pse2) {
                        if (minShift[pse1.c] > minShift[pse2.c])
                              return -1;
                        if (minShift[pse1.c] < minShift[pse2.c])
                              return 1;
                        if (pse1.loc > pse2.loc)
                              return -1;
                        if (pse1.loc < pse2.loc)
                              return 1;
                        return 0;
                  }
            }

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public MaximalShift(String pat) {
                  if      (pat      ==      null)      throw      new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  this.minShift = new int[m];
                  computeMinShift();
                  this.qsBc = new int[ASIZE];
                  preQsBc();
            }
```

```
            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  final Comparator<PatternScanElement> byShiftLoc;
                  final      PatternScanElement[]      pse      =      new
PatternScanElement[m];
                  final int[] adaptedGs = new int[m + 1];
                  int i, j;

                  /* Preprocessing */

                  // create comparator based on frequencies
                  byShiftLoc = new ByFreqLoc(minShift);

                  orderPattern(pattern, pse, byShiftLoc);
                  preAdaptedGs(pattern, adaptedGs, pse);

                  /* Searching */
                  j = 0;
                  while (j <= n - m) {
                        i = 0;
                        while   (i   <   m   &&   pse[i].c   ==   text[j   +
pse[i].loc])
                              ++i;
                        if (i >= m)
                              return j;
                        if (j < n - m)
                              j += Math.max(adaptedGs[i], qsBc[text[j +
m]]);
                        else
                              j += adaptedGs[i];
                  }
                  return n;
            }

            /**
             *  Returns  the  indices  of  all  the  occurrences  of  the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
```

```
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  final int[] freq = new int[ASIZE];
                  final Comparator<PatternScanElement> byFreqLoc;
                  final     PatternScanElement[]     pse     =     new
PatternScanElement[m];
                  final int[] adaptedGs = new int[m + 1];
                  int i, j;

                  /* Preprocessing */

                  // get frequencies
                  for (char c : text) {
                        freq[c]++;
                  }

                  // create comparator based on frequencies
                  byFreqLoc = new ByFreqLoc(freq);

                  orderPattern(pattern, pse, byFreqLoc);
                  preAdaptedGs(pattern, adaptedGs, pse);

                  /* Searching */
                  j = 0;
                  while (j <= n - m) {
                        i = 0;
                        while   (i   <   m   &&   pse[i].c   ==   text[j   +
pse[i].loc])
                              ++i;
                        if (i >= m)
                              list.add(j);
                        if (j < n - m)
                              j += Math.max(adaptedGs[i], qsBc[text[j +
m]]);
                        else
                              j += adaptedGs[i];
                  }
                  return list;
            }

            /* Construct an ordered pattern from a string. */
            private    static    void    orderPattern(char[]    pattern,
PatternScanElement[] pse, Comparator<PatternScanElement> cmptor) {
                  final int m = pattern.length;
                  for (int i = 0; i < m; ++i) {
                        PatternScanElement        x        =        new
PatternScanElement();
                        x.c = pattern[i];
```

```
                        x.loc = i;
                        pse[i] = x;
                  }
                  Arrays.sort(pse, cmptor);
            }

            /*
             * Constructs the good-suffix shift table from an ordered
string.
             */
            private static void preAdaptedGs(char[] pattern, int[]
adaptedGs, PatternScanElement[] pse) {
                  final int m = pattern.length;
                  int lshift, i, ploc;

                  adaptedGs[0] = lshift = 1;
                  for (ploc = 1; ploc <= m; ++ploc) {
                        lshift = matchShift(pattern, ploc, lshift,
pse);
                        adaptedGs[ploc] = lshift;
                  }
                  for (ploc = 0; ploc < m; ++ploc) { //
                        lshift = adaptedGs[ploc];
                        while (lshift < m) {
                              i = pse[ploc].loc - lshift;
                              if (i < 0 || pse[ploc].c != pattern[i])
                                    break;
                              ++lshift;
                              lshift = matchShift(pattern, ploc,
lshift, pse);
                        }
                        adaptedGs[ploc] = lshift;
                  }
            }

            /*
             * Find the next leftward matching shift for the first ploc
pattern elements
             * after a current shift or lshift.
             */
            private static int matchShift(char[] pattern, int ploc, int
lshift, PatternScanElement[] pse) {
                  final int m = pattern.length;
                  int i, j;

                  for (; lshift < m; ++lshift) {
                        i = ploc;
                        while (--i >= 0) {
                              if ((j = (pse[i].loc - lshift)) < 0)
                                    continue;
                              if (pse[i].c != pattern[j])
                                    break;
                        }
                        if (i < 0)
                              break;
                  }
                  return (lshift);
            }

            // preprocessing
```

```
	private void preQsBc() {
		final int m = pattern.length;
		for (int i = 0; i < ASIZE; ++i)
			qsBc[i] = m + 1;
		for (int i = 0; i < m; ++i)
			qsBc[pattern[i]] = m - i;
	}

	private void computeMinShift() {
		final int m = pattern.length;
		int i, j;

		for (i = 0; i < m; ++i) {
			for (j = i - 1; j >= 0; --j)
				if (pattern[i] == pattern[j])
					break;
			minShift[i] = i - j;
		}
	}

}
```

7.1.33 Skip Search

```
package com.accel.stringsearch;

import java.util.ArrayList;
import java.util.List;

/**
 * The {@code SkipSearch} class finds the occurrences of a pattern string in a
 * text string.
 * <p>
 * This implementation uses buckets of positions for each character of the
 * alphabet. This implementation provides methods for retrieving the first
 * occurrence of the pattern in the text and for retrieving all the occurrences
 * of the pattern in the text. It takes time proportional to <em>nm</em> in the
 * worst case and <em>n/m</em> in the best case, where <em>n</em> is the length
 * of the text and <em>m</em> is the length of the pattern. The preprocessing
 * phase takes <em>m + σ</em> time and space, where σ is the size of the
 * alphabet. There are <em>O(n)</em> expected text character comparisons.
 * </p>
 *
 */
public class SkipSearch {
	private static final int ASIZE = 256;

	private final char[] pattern;
	private final Cell[] z;

	private static class Cell {
		private int element;
		private Cell next;
```

```
        }

        /**
         * Preprocesses the pattern string.
         *
         * @param pat
         *            the pattern string
         */
        public SkipSearch(String pat) {
            if (pat == null) throw new
IllegalArgumentException("pattern is null");
            if (pat.equals("")) throw new
IllegalArgumentException("pattern is empty");

            this.pattern = pat.toCharArray();
            this.z = new Cell[ASIZE];
            preSs();
        }

        /**
         * Returns the index of the first occurrence of the pattern
string in the text
         * string.
         *
         * @param txt
         *            the text string
         * @return the index of the first occurrence of the pattern
string in the text
         *         string; n if no such match
         */
        public int search(String txt) {
            if (txt == null) throw new
IllegalArgumentException("text is null");
            if (txt.equals("")) throw new
IllegalArgumentException("text is empty");

            final char[] text = txt.toCharArray();
            final int m = pattern.length;
            final int n = text.length;
            int i;
            /* Searching */
            for (i = m - 1; i < n; i += m)
                for (Cell ptr = z[text[i]]; ptr != null; ptr =
ptr.next)
                    if (memcmp(pattern, 0, text, i -
ptr.element, m))
                        return i - ptr.element;
            return n;
        }

        /**
         * Returns the indices of all the occurrences of the
pattern string in the text
         * string.
         *
         * @param txt
         *            the text string
         * @return the indices of all the occurrences of the
pattern string in the text
         *         string
```

```
             */
            public List<Integer> searchAll(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i;
                  /* Searching */
                  for (i = m - 1; i < n; i += m)
                        for (Cell ptr = z[text[i]]; ptr != null; ptr =
ptr.next)
                              if   (memcmp(pattern,   0,   text,   i   -
ptr.element, m))
                                    list.add(i - ptr.element);
                  return list;
            }

            // preprocessing
            private void preSs() {
                  final int m = pattern.length;

                  for (int i = 0; i < m; ++i) {
                        Cell ptr = new Cell();
                        ptr.element = i;
                        ptr.next = z[pattern[i]];
                        z[pattern[i]] = ptr;
                  }
            }

            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  int i;
                  for (i = 0; i < length && startX + i < x.length &&
startY + i < y.length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  if (i < length && x.length - startX != y.length -
startY) {
                        return false;
                  }
                  return true;
            }

      }
```

7.1.34 KMP Skip Search

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.List;

      /**
       * The {@code KMPSkipSearch} class finds the occurrences of a
pattern string in
```

```
       * a text string.
       * <p>
       *  This implementation is an improvement of the SkipSearch
algorithm. It uses
       * buckets of positions for each character of the alphabet. This
implementation
       * provides methods for retrieving the first occurrence of the
pattern in the
       * text and for retrieving all the occurrences of the pattern in
the text. It
       * takes time proportional to <em>nm</em> in the worst case and
<em>n/m</em> in
       * the best case, where <em>n</em> is the length of the text and
<em>m</em> is
       * the length of the pattern. The preprocessing phase takes <em>m
+ σ</em> time
       * and space, where σ is the size of the alphabet.
       * </p>
       *
       */
      public class KMPSkipSearch {
            private static final int ASIZE = 256;

            private final char[] pattern;
            private final int[] mpNext;
            private final int[] kmpNext;
            private final int[] list;
            private final int[] z;

            /**
             * Preprocesses the pattern string.
             *
             * @param pat
             *            the pattern string
             */
            public KMPSkipSearch(String pat) {
                  if       (pat       ==       null)       throw       new
IllegalArgumentException("pattern is null");
                  if         (pat.equals(""))         throw         new
IllegalArgumentException("pattern is empty");

                  this.pattern = pat.toCharArray();
                  final int m = pattern.length;
                  this.mpNext = new int[m + 1];
                  preMp();
                  this.kmpNext = new int[m + 1];
                  preKmp();
                  this.list = new int[m];
                  for (int i = 0; i < list.length; ++i) {
                        list[i] = -1;
                  }
                  this.z = new int[ASIZE];
                  for (int i = 0; i < z.length; ++i) {
                        z[i] = -1;
                  }
                  z[pattern[0]] = 0;
                  for (int i = 1; i < m; ++i) {
                        list[i] = z[pattern[i]];
                        z[pattern[i]] = i;
                  }
```

```
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if       (txt       ==       null)       throw       new
IllegalArgumentException("text is null");
                  if          (txt.equals(""))          throw          new
IllegalArgumentException("text is empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int i, j, k, kmpStart, per, start, wall;

                  /* Searching */
                  wall = 0;
                  per = m - kmpNext[m];
                  i = j = -1;
                  do {
                        j += m;
                  } while (j < n && z[text[j]] < 0);
                  if (j >= n)
                        return n;
                  i = z[text[j]];
                  start = j - i;
                  while (start <= n - m) {
                        if (start > wall)
                              wall = start;
                        k = attempt(text, pattern, start, wall);
                        wall = start + k;
                        if (k == m) {
                              return start;
                        } else
                              i = list[i];
                        if (i < 0) {
                              do {
                                    j += m;
                              } while (j < n && z[text[j]] < 0);
                              if (j >= n)
                                    return n;
                              i = z[text[j]];
                        }
                        kmpStart = start + k - kmpNext[k];
                        k = kmpNext[k];
                        start = j - i;
                        while (start < kmpStart || (kmpStart < start &&
start < wall)) {
                              if (start < kmpStart) {
                                    i = list[i];
                                    if (i < 0) {
```

```
                                        do {
                                              j += m;
                                        } while (j < n && z[text[j]]
< 0);
                                        if (j >= n)
                                              return n;
                                        i = z[text[j]];
                                  }
                                  start = j - i;
                            } else {
                                  kmpStart += (k - mpNext[k]);
                                  k = mpNext[k];
                            }
                      }
                }
                return n;
          }

          /**
           * Returns the indices of all the occurrences of the
pattern string in the text
           * string.
           *
           * @param txt
           *            the text string
           * @return the indices of all the occurrences of the
pattern string in the text
           *         string
           */
          public List<Integer> searchAll(String txt) {
                if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                List<Integer> result = new ArrayList<>();
                final char[] text = txt.toCharArray();
                final int m = pattern.length;
                final int n = text.length;
                int i, j, k, kmpStart, per, start, wall;

                /* Searching */
                wall = 0;
                per = m - kmpNext[m];
                i = j = -1;
                do {
                      j += m;
                } while (j < n && z[text[j]] < 0);
                if (j >= n)
                      return result;
                i = z[text[j]];
                start = j - i;
                while (start <= n - m) {
                      if (start > wall)
                            wall = start;
                      k = attempt(text, pattern, start, wall);
                      wall = start + k;
                      if (k == m) {
                            result.add(start);
                            i -= per;
```

```
                        } else
                              i = list[i];
                        if (i < 0) {
                              do {
                                    j += m;
                              } while (j < n && z[text[j]] < 0);
                              if (j >= n)
                                    return result;
                              i = z[text[j]];
                        }
                        kmpStart = start + k - kmpNext[k];
                        k = kmpNext[k];
                        start = j - i;
                        while (start < kmpStart || (kmpStart < start &&
start < wall)) {
                              if (start < kmpStart) {
                                    i = list[i];
                                    if (i < 0) {
                                          do {
                                                j += m;
                                          } while (j < n && z[text[j]]
< 0);
                                          if (j >= n)
                                                return result;
                                          i = z[text[j]];
                                    }
                                    start = j - i;
                              } else {
                                    kmpStart += (k - mpNext[k]);
                                    k = mpNext[k];
                              }
                        }
                  }
                  return result;
            }

            // builds mpNext
            private final void preMp() {
                  final int m = pattern.length;
                  mpNext[0] = -1;
                  int j = -1;
                  int i = 0;
                  while (i < m) {
                        while (j > -1 && pattern[i] != pattern[j]) {
                              j = mpNext[j];
                        }
                        mpNext[++i] = ++j;
                  }
            }

            // builds kmpNext
            private void preKmp() {
                  final int m = pattern.length;
                  int i, j;

                  i = 0;
                  j = -1;
                  kmpNext[0] = -1;
                  while (i < m) {
                        while (j > -1 && pattern[i] != pattern[j]) {
```

```
                              j = kmpNext[j];
                        }
                        i++;
                        j++;
                        if (i < m && j < m && pattern[i] == pattern[j])
{ // CHANGED:: if(pattern[i) == pattern[j))
                              kmpNext[i] = kmpNext[j];
                        } else {
                              kmpNext[i] = j;
                        }
                  }
            }

            private static int attempt(char[] y, char[] x, int start,
int wall) {
                  final int m = x.length;
                  int k;

                  k = wall - start;
                  while (k < m && x[k] == y[k + start])
                        ++k;
                  return k;
            }

      }
```

7.1.35 Alpha Skip Search

```
      package com.accel.stringsearch;

      import java.util.ArrayList;
      import java.util.List;

      import com.accel.utils.Graph;
      import com.accel.utils.Trie;

      /**
       * The {@code AlphaSkipSearch} class finds the occurrences of a
pattern string
       * in a text string.
       * <p>
       * This implementation is an improvement of the Skip Search
Algorithm; it uses
       * buckets of positions for each factor of length log(base )m.
This
       * implementation provides methods for retrieving the first
occurrence of the
       * pattern in the text and for retrieving all the occurrences of
the pattern in
       * the text. It takes time proportional to <em>nm</em>, where
<em>n</em> is the
       * length of the text and <em>m</em> is the length of the
pattern. The
       * preprocessing phase takes <em>m</em> time . There are
<em>O(log(base )m*(n /
       * (m - log(base )m)))</em> expected text character comparisons.
       * </p>
       *
       */
      public class AlphaSkipSearch {
            private static final int ASIZE = 256;
```

```
	private final char[] pattern;
	private final Graph trie;
	private final Cell[] z;
	private int logM, root, node;

	private static class Cell {
		private int element;
		private Cell next;
	}

	/**
	 * Preprocesses the pattern string.
	 *
	 * @param pat the pattern string
	 */
	public AlphaSkipSearch(String pat) {
		if (pat == null)
			throw new IllegalArgumentException("pattern is
null");
		if (pat.equals(""))
			throw new IllegalArgumentException("pattern is
empty");

		this.pattern = pat.toCharArray();
		final int m = pattern.length;
		int i, temp, size, pos;
		int art, childNode, lastNode;

		logM = 0;
		temp = m;
		while (temp > ASIZE) {
			++logM;
			temp /= ASIZE;
		}
		if (logM == 0)
			logM = 1;

		/* Preprocessing */
		size = 2 + (2 * m - logM + 1) * logM;
		trie = new Trie(size, size * ASIZE);
		z = new Cell[size];
		root = trie.getInitial();
		art = trie.newVertex();
		trie.setSuffixLink(root, art);
		node = trie.newVertex();
		trie.setTarget(root, pattern[0], node);
		trie.setSuffixLink(node, root);
		for (i = 1; i < logM; ++i)
			node = addNode(trie, art, node, pattern[i]);
		pos = 0;
		setZ(node, pos);
		pos++;
		for (i = logM; i < m - 1; ++i) {
			node = trie.getSuffixLink(node);
			childNode = trie.getTarget(node, pattern[i]);
			if (childNode == Graph.UNDEFINED)
				node = addNode(trie, art, node,
pattern[i]);
			else
				node = childNode;
```

```
                        setZ(node, pos);
                        pos++;
                  }
                  node = trie.getSuffixLink(node);
                  childNode = trie.getTarget(node, pattern[i]);
                  if (childNode == Graph.UNDEFINED) {
                        lastNode = trie.newVertex();
                        trie.setTarget(node, pattern[m - 1], lastNode);
                        node = lastNode;
                        node = addNode(trie, art, node, pattern[i]);
                  } else
                        node = childNode;
                  setZ(node, pos);
            }

            /**
             * Returns the index of the first occurrence of the pattern
string in the text
             * string.
             *
             * @param txt the text string
             * @return the index of the first occurrence of the pattern
string in the text
             *         string; n if no such match
             */
            public int search(String txt) {
                  if (txt == null)
                        throw   new   IllegalArgumentException("text   is
null");
                  if (txt.equals(""))
                        throw   new   IllegalArgumentException("text   is
empty");

                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int shift, j, k, b;
                  Cell current;
                  /* Searching */
                  shift = m - logM + 1;
                  for (j = m + 1 - logM; j < n - logM; j += shift) {
                        node = root;
                        for  (k  =  0;  node  !=  Graph.UNDEFINED  &&  k  <
logM; ++k)
                              node = trie.getTarget(node, text[j + k]);
                        if (node != Graph.UNDEFINED)
                              for (current = z[node]; current != null;
current = current.next) {
                                    b = j - current.element;

                                    if   (pattern[0]   ==   text[b]   &&
memcmp(pattern, 1, text, b + 1, m - 1))
                                          return b;
                              }
                  }
                  return n;
            }

            /**
```

```
             * Returns the indices of all the occurrences of the
pattern string in the text
             * string.
             *
             * @param txt
             *            the text string
             * @return the indices of all the occurrences of the
pattern string in the text
             *         string
             */
            public List<Integer> searchAll(String txt) {
                  if      (txt      ==      null)      throw      new
IllegalArgumentException("text is null");
                  if         (txt.equals(""))         throw         new
IllegalArgumentException("text is empty");

                  List<Integer> list = new ArrayList<>();
                  final char[] text = txt.toCharArray();
                  final int m = pattern.length;
                  final int n = text.length;
                  int shift, j, k, b;
                  Cell current;
                  /* Searching */
                  shift = m - logM + 1;
                  for (j = m + 1 - logM; j < n - logM; j += shift) {
                        node = root;
                        for (k = 0; node != Graph.UNDEFINED && k <
logM; ++k)
                              node = trie.getTarget(node, text[j + k]);
                        if (node != Graph.UNDEFINED)
                              for (current = z[node]; current != null;
current = current.next) {
                                    b = j - current.element;
                                    if   (pattern[0]   ==   text[b]   &&
memcmp(pattern, 1, text, b + 1, m - 1))
                                          list.add(b);
                              }
                  }
                  return list;
            }

            private void setZ(int node, int i) {
                  Cell cell = new Cell();
                  cell.element = i;
                  cell.next = z[node];
                  z[node] = cell;
            }

            /*
             * Create the transition labelled by the character c from
node node. Maintain
             * the suffix links accordingly.
             */
            private int addNode(Graph trie, int art, int node, char c)
{
                  int childNode, suffixNode, suffixChildNode;

                  childNode = trie.newVertex();
                  trie.setTarget(node, c, childNode);
                  suffixNode = trie.getSuffixLink(node);
```

```
                  if (suffixNode == art)
                        trie.setSuffixLink(childNode, node);
                  else {
                        suffixChildNode   =   trie.getTarget(suffixNode,
c);
                        if (suffixChildNode == Graph.UNDEFINED)
                              suffixChildNode   =   addNode(trie,   art,
suffixNode, c);
                        trie.setSuffixLink(childNode, suffixChildNode);
                  }
                  return (childNode);
            }

            private boolean memcmp(char[] x, int startX, char[] y, int
startY, int length) {
                  int i;
                  for (i = 0; i < length && startX + i < x.length &&
startY + i < y.length; ++i) {
                        if (x[startX + i] != y[startY + i]) {
                              return false;
                        }
                  }
                  if (i < length && x.length - startX != y.length -
startY) {
                        return false;
                  }
                  return true;
            }
      }
```