NAME- MAREESH KUMAR ISSAR

NET ID- mi251

TITLE- ANALYSIS OF STRING SEARCHING ALGORITHMS

COURSE NAME- 16:332:573 DATA STRUCTURE AND ALGORITHMS

SUBMISSION DATE- 9 April, 2019

ABSTRACT

String searching algorithms are used to find the first occurrence (or more generally all occurrences) of a particular pattern in a given text. This paper investigates four string searching algorithms namely Brute force algorithm, Boyer-Moore algorithm, Rabin-Karp algorithm, and Knuth-Morris-Pratt algorithm. In this paper for each of the above mentioned algorithms, a detailed description of their working, implementation and properties is provided. Finally, a table summarizing all the key properties (searching time worst case, extra space requirements etc.) of these algorithms is provided to compare these algorithms.

INTRODUCTION

In today's information age, large amount of data is one of the key drivers that is helping us solve a wide variety of problems. In order to extract useful information from this data, various algorithms are used. One such important class of algorithms that finds its application in various scientific/engineering/commercial application is string searching algorithm, which is used finding the occurrence of a particular pattern in the data. Finding and replacing a text in a document using text processing programs is the most fundamental application of such string searching algorithms.

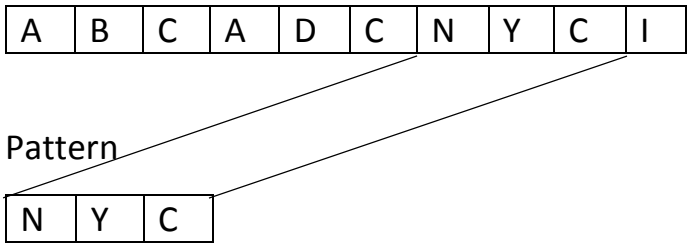
Other applications include searching for a particular pattern in DNA sequence, crawling a website to extract relevant data, spam filtering - by searching for most frequent keywords found commonly in spam emails, digital surveillance – deployment of string matching algorithms by a particular agency to scan the large amount of web traffic flowing through a location in order to identify and monitor threat emails [1].

The ultimate aim of any string searching algorithm is to provide a time and memory efficient solution to the issue of: -

Finding the first match (or more generally all occurrences) of a particular pattern (pat[0..M-1]) of length M in a text (txt[0..N-1]) of length N, where generally M<<N. All the elements present in the pattern and the text are drawn from a finite alphabet set of size R (Radix of the alphabet).

A majority of string searching algorithms match the pattern to the text by using a sliding window mechanism, where a window of the patterns slides along the text to find a match. For example,

Text



DESCRIPTION

All the algorithms discussed below find the position of the first occurrence of a pattern in a given text.

BRUTE FORCE ALGORITHM

This algorithm is the most basic implementation of a string searching algorithm.

WORKING

The Brute Force algorithm essentially tries to match the pattern (character by character) starting with each character present in the text [2]. If there is a mismatch, it goes back to starting position+1 in the text and starts the comparison again.

IMPLEMENTATION

The following is an implementation [3] of the algorithm in C++: -

```
int string_search_bf(string txt, string pat)
{
	int n = txt.length();
	int m = pat.length();
	int i, j;
	for (i = 0, j = 0; i < n && j < m; i++)
	{
		if (txt.at(i) == pat.at(j))
			j++;
		else
		{	i -= j;
			j = 0;
		}
```

```
        }
        if (j == m) return i - m;    // found position
        else        return n;        // not found
}
```

PROPERTIES

This algorithm requires no preprocessing of the pattern data but is usually slow when the text and the pattern are of large lengths and there are lots of common elements present in the text and the pattern. A major drawback of this algorithm is that whenever there is a mismatch, matching of the pattern with the text starts by going back in the text.

Average case: O(N) searching time.

Worst case: O(MN) searching time. This occurs when characters in the text and the characters in the pattern differ only in the last character, for example

Text

C	C	C	C	C	C	C	C	C	C

Pattern

C	C	A

BOYER-MOORE ALGORITHM (USING BAD CHARACTER RULE)

Boyer-Moore algorithm was developed in 1977 by Robert S. Boyer and J. Strother Moore [4]. It is often considered as an efficient algorithm for implementation of string searching algorithms in text editors (for find and replace operations) and different programming languages (C++ Algorithm library) [2]. It has the following key properties: -

1. It preprocesses the pattern. This preprocessing helps in reducing the comparisons by skipping characters while matching.
2. It matches the pattern with the text starting from the right end of the pattern.

WORKING

This algorithm starts matching the right end of the pattern with the text. If a mismatch is found (i.e. bad character) then one of the following case is executed: -

CASE 1: - Mismatch character in the text is not present in the pattern.

Skip the text pointer by M+1 and then start comparing the pattern again from the right end.

CASE 2a: - Mismatch character in the text present in the pattern

Move the start of the text pointer such that it aligns with the position of the rightmost occurrence of that character in the pattern.

CASE 2b: - The rightmost occurrence of the mismatch character in the text is in the last position of the pattern

Increment the text pointer by 1.

In the preprocessing phase, an array right[R] is used to store the rightmost occurrence of each character present in the pattern. This array is used in the algorithm for finding the skip value.

IMPLEMENTATION

The following is an implementation [3] of the algorithm in C++: -

```
int string_search_bm(string txt, string pat)
{
	int n = txt.length();
	int m = pat.length();
	int skip;
	//Preprocessing started for the pattern
	const int R = 256; //Radix
	int right[R];
	//array for storing position of rightmost occurrences of the characters in the pattern
	for (int i = 0; i < R; i++)
		right[i] = -1;

	for (int i = 0; i < pat.length(); i++)
		right[pat.at(i)] = i;

	//Preprocessing completed
	for (int i = 0; i <= n - m; i += skip)
	{
		skip = 0;
		for (int j = m - 1; j >= 0; j--)
		{
			if (pat.at(j) != txt.at(i + j))//mismatch
			{
				skip = max(1, j - right[txt.at(i + j)]);
				break;
			}
		}
		if (skip == 0) return i;    // found
```

```
        }
        return n;                       // not found
}
```

PROPERTIES

An important characteristic of this algorithm is that, its average case behavior (N/M compare operations) is better than the linear dependence on N (this occurs if comparison is made with each character present in the text). The number of comparisons further decrease with the increase in length of the pattern. Also, it requires an extra space (R) for storing the last occurrence index of each character present in the pattern.

Average case: O(N/M) searching time. This occurs because on average the skip value can be taken to be equal to pattern length M.

Worst case: O(MN) searching time. This occurs when characters in the text and the characters in the pattern differ only in the first character, for example

Text

C	C	C	C	C	C	C	C	C	C

Pattern

A	C	C

RABIN-KARP ALGORITHM

Rabin-Karp algorithm was developed in 1987 by Richard M. Karp and Michael O. Rabin [5]. Apart from searching for a single pattern, it is widely used for multiple pattern searching [2]. This algorithm has the following key properties:

1. It preprocesses the pattern. This preprocessing helps to generate a hash value for the pattern which can then be used for matching with the hash value of the characters (of window size M) present in the text.
2. It uses the concept of rolling hash to compute the hash for the next window of M characters after the hash of the first M characters in the text is computed with the help of Horner method.

WORKING

This algorithm employs the concept of modular hashing, where

1. Hash of the pattern is computed.
2. Hash of the text (for each window of M characters) is computed.
3. If there is a hash match, then a both the sequences are compared character by character (Las Vegas implementation).

Modular arithmetic

Here, for the purpose of explaining the functioning of modulo, text as well as the pattern are considered to be of type integers (if they are of any other type, they can be easily converted to a corresponding integer value).

Instead of computing modulo for large number all at once, modulo of smaller numbers (generated by breaking a large number) is computed using the following formulas: -

$$(a+b) \bmod Q = \left((a \bmod Q) + (b \bmod Q)\right) \bmod Q$$

$$(a*b) \bmod Q = \left((a \bmod Q) * (b \bmod Q)\right) \bmod Q$$

In the above equations Q is a sufficiently large prime number.

For computing the hash of the first M elements of the text Horner's method is used. Horner's method is used for the evaluation of a polynomial in linear time.

Computing the modular hash function

Let x_i denotes the character at the i[th] position in the text and R, the Radix, then hash h_i is given by

$$h_i = x_i R^{M-1} + x_{i+1} R^{M-2} \ldots + x_{i+M-1} R^0 \ (\bmod Q)$$

For computing the value of h_{i+1} (rolling hash), we could use the value h_i computed in the previous step,

$$h_{i+1} = (h_i - x_i R^{M-1})\, R + x_{i+M} \ (\bmod Q)$$

This algorithm can be implemented in the following two versions: -

1. Monte Carlo version- In this case we return as soon as the hash of the pattern matches with the hash of a sequence in the text. This version is guaranteed fast but not always correct.
2. Las Vegas version- In this case we compare each character in the text and the pattern after their hash match. This version is guaranteed to provide correct answer but is not always fast.

IMPLEMENTATION

The following is a Las Vegas implementation [3] of the algorithm in C++: -

```
// Compute hash for key[0..m-1].
long h(string key, int m)
{
	int R = 256;
	long q = 2147483647;

	long h = 0;
	for (int j = 0; j < m; j++)
		h = (R * h + key.at(j)) % q;
	return h;
}

// Las Vegas version: does pat[] match txt[i..i-m+1] ?
bool check(string txt, string pat, int i)
{
	int m = pat.length();

	for (int j = 0; j < m; j++)
		if (pat.at(j) != txt.at(i + j))
			return false;
	return true;
}

int string_search_rkf(string txt, string pat)
{
	int n = txt.length();
	int m = pat.length();

	int R = 256;//Radix
	long q = 2147483647;//Large prime number


	// precompute R^(m-1) % q for use in removing leading digit
	long RM = 1;
	for (int i = 1; i <= m - 1; i++)
		RM = (R * RM) % q;

	long patHash = h(pat, m);

	if (n < m) return n;

	long txtHash = h(txt, m);
```

```
        // check for match at offset 0
        if ((patHash == txtHash) && check(txt,pat, 0))
               return 0;

        // check for hash match; if hash match, check for exact match
        for (int i = m; i < n; i++)
        {
               // Remove leading digit, add trailing digit, check for match.
               txtHash = (( ( txtHash-(txt.at(i-m)*RM) ) )*R + txt.at(i)) % q;

               // match
               int offset = i - m + 1;
               if ((patHash == txtHash) && check(txt,pat,offset))
                      return offset;
        }

        // no match
        return n;
}
```

PROPERTIES

Las Vegas version of the algorithm needs character wise comparison to guard against collisions whenever there is a hash match. A large prime number (of size MN^2) decreases the probability of false collisions to about 1/N. In practice, the probability of collisions in about 1/Q. The rolling hash feature allows O(1) computing after the first hash is computed.

Average case: O(M+N) searching time.

Worst case: O(MN) searching time. This occurs when the pattern and the text differ only in the last character i.e. there are multiple hash matches followed by single character compares (similar to the worst case of Brute Force algorithm).

Text

C	C	C	C	C	C	C	C	C	C

Pattern

C	C	A

KNUTH-MORRIS-PRATT ALGORITHM

The Knuth-Morris-Pratt algorithm was developed in 1977 by Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt [6]. This algorithm has the following key properties: -

1. It preprocesses the pattern. This preprocessing helps in skipping the characters in the text that were already matched with those present in the pattern.
2. It uses Deterministic Finite Automaton (DFA) of the pattern to identify the next element to compare and thus avoids going back to the previously compared positions of the text.

WORKING

This algorithm uses DFA at its core for all the computation [1], thus understanding the creation and working of the DFA forms the core of understanding the working of this algorithm.

The following table shows the representation of a DFA

	0	1	2	3	4	5	6
	A	C	A	C	A	G	A
A	1	1	3	1	5	1	7
C	0	2	0	4	0	4	0
G	0	0	0	0	0	6	0

Table 1: Tabular representation of DFA for the pattern ACACAGA.

The number of states in the DFA = number of characters in the pattern+1(stop state). It has a stop state, which indicates that a match for the pattern has been found in the text. States in a DFA are also the representation of the longest match between the pattern and the text.

DFA Creation

CASE 1: Character matching to the current state found in the text

Go to the next state in the DFA for the pattern.

CASE 2: Character matching to the current state not found in the text

Copy the contents of the states (except those which satisfy CASE 1) from the state X (state where the machine would be if the pattern was shifted from its starting position by 1)

IMPLEMENTATION

The following is an implementation [3] of the algorithm in C++: -

```
int string_search_kmp(string txt, string pat)
```

```
{
	int n = txt.length();
	int m = pat.length();
	int i, j;
    int R = 256;

	vector<vector<int> > dfa(R);
	for (int i = 0; i < R; i++)
	{
		// size of column
		int col = m;

		// declare  the i-th row to size of column
		dfa[i] = vector<int>(col);
		for (int j = 0; j < col; j++)
			dfa[i][j] = 0;
	}

	// preprocessing
    // building DFA from pattern
	dfa[pat.at(0)][0] = 1;

	for (int x = 0, j = 1; j < m; j++)
	{
		for (int c = 0; c < R; c++)
			dfa[c][j] = dfa[c][x];     // Copy mismatch cases.

		dfa[pat.at(j)][j] = j + 1;   // Set match case.
		x = dfa[pat.at(j)][x];     // Update restart state.
	}

	for (i = 0, j = 0; i < n && j < m; i++)
	{
		j = dfa[txt.at(i)][j];
	}

	if (j == m) return i - m;    // found
	return n;                    // not found
}
```

PROPERTIES

This algorithm is well suited for use in the case when there are a lot of redundancies between the characters of the pattern and the text. Also, it requires extra space (RM) for storing the DFA of the pattern.

Average case: O(N) searching time.

Worst case: O(N) searching time.

ANALYSIS

The table below provides an overall analysis of the different string searching algorithms analyzed in this paper. Here, the following conventions are considered: -

1. Length of the pattern is M.
2. Length of the text is N.
3. Alphabet size (Radix) is R.

Algorithm	Remark	Searching time (average case)	Searching time (worst case)	Extra space requirements
Brute Force algorithm	-	O(N)	O(MN)	O(1)
Boyer-Moore algorithm	Bad character rule	O(N/M)	O(MN)	O(R)
Rabin-Karp algorithm	Las Vegas version	O(N+M)	O(MN)	O(1)
Knuth-Morris-Pratt algorithm	DFA based	O(N)	O(N)	O(RM)

Table 2: Analysis of string searching algorithms.

CONCLUSION

By analyzing the working of the different string searching algorithms presented in this paper, we find that we can achieve performance of O(N) or better in the average case by preprocessing the pattern, since the pattern is already known to us. Also if the algorithm doesn't require backing up in the input text (like the Knuth-Morris-Pratt algorithm) during pattern searching then such algorithms can be used for searching patterns in input data streams. The advantage of Boyer-Moore algorithm over the other algorithms is its low implementation complexity and its sublinear average case searching time. In the Rabin-Karp algorithm by selecting a sufficiently large prime

number (size MN^2) we can decrease the probability of false collisions to about 1/N. The Knuth-Morris-Pratt algorithm provides good performance even when the pattern and the text have multiple common characters.

REFERENCES

[1]. https://algs4.cs.princeton.edu/lectures/53SubstringSearch.pdf

[2]. http://www-igm.univ-mlv.fr/~lecroq/string/node2.html#SECTION0020

[3]. https://algs4.cs.princeton.edu/53substring/

[4]. Boyer, Robert S., and J. Strother Moore. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

[5]. Karp, Richard M., and Michael O. Rabin. "Efficient randomized pattern-matching algorithms." *IBM journal of research and development* 31.2 (1987): 249-260.

[6]. Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." *SIAM journal on computing*6.2 (1977): 323-350.