Experimentation and Comparison with String Search Algorithms

Team - 6
Ezgi Yılmaz - 150150036
Muhammed Burak Buğrul - 150140015

Abstract
String-search is an old problem in the history of Computer Science. Through the time several solution has provided for that problem. In this report, we are presenting implementations of 4 of the string search algorithms and the experiment we have done on different input sets. We are comparing the time complexities of Naive Approach, Knutt-Morris-Pratt (KMP) Algorithm, Rabin Karp Algorithm, and Z Algorithm and making analysis about the results that we get.

Table of Contents

Introduction

Report Context

This report is prepared for “BLG 374E - Technical Communication for Computer Engineers” course term project.

Project Purpose and Context

We aim to implement 4 different string search algorithms and compare their performances on different test cases. After testing we will analyze the results and find in which condition which algorithms are more efficient. So, this research can be used for future tasks of string search in computer science related communities. We will add our analysis and results so people can choose a suitable algorithm.

Background

Naive Approach

Naive approach is the simplest and first-thought approach to the string-search problem. We take the target string and look every possible place that we can search in the string that will be searched. We began from the first character of the string and try to be fit our target in it, by iterating the whole string we are completing the search. We do not perform any preprocessing operations on the strings. The complexity of Naive Approach O(n.m) where n is the length of the string, m is the length of the target string.

Knutt-Morris-Pratt (KMP) Algorithm

Knutt Morris Pratt Algorithm uses a finite automata-based approach to the string search problem. It creates a DFA (Deterministic Finite Automata) as we send our text as an input if the text consists that our word [1]. It matches the prefix of the word with text similarly to Naive Approach but for better performance we do some preprocessing operations and construct a table that we stored the next starting point of matching. Then we can search our string in the text more efficiently. The complexity of preprocessing operations is O(m) and the complexity of the matching process is O(n) in total we can search in O(n+m) the whole text where n is the text length and m is the length of the string that we are searching.

Rabin-Karp Algorithm

Rabin Karp algorithm created by Richard M.Karp and Michael O. Kabin in 1987. The algorithm uses the string hashing for a faster search in the text [2]. If two string is identical than their hash values are identical as well. With this information if we hash both strings and compare text hash values with the pattern's hash value for all proper contiguous substrings, we can easily check if the text contains that word [4]. We need to hash the string as preprocessing part and its complexity is O(m), Complexity of the searching process is in best and average case is O(n+m), so our total complexity is O(n+m).

Z Algorithm

Z algorithm is similar to the KMP algorithm in the preprocessing phase and the searching phase. We store the length of the longest substring from the beginning of the text that is a prefix of our word in the preprocessing part. With Z algorithm we can search the prefixes of the word or the whole word if we want. In order to perform Z algorithms for string search, we can attach the pattern into the beginning of the text then perform classic Z algorithm on it. The complexity of the Z algorithm is O(m+n) where n is the length of the given text and m is the length of the string that will be searched [3].

Methods

Inputs

In order to compare our 4 algorithms in time complexity and in different circumstances, we generated different datasets that sized 1000, 10000, 100000 characters and search patterns that sized 33, 300, 333, 1000, 3000, 30000. We also considered the number of symbols that text consists of as an attribute of our input.

We generated our inputs with Python 3.7. The one can give input parameters like length of text, length of the pattern that will be searched and the number of different letters that will be used. You can see the random number generator code in Figure 3.1.1.

```python
# n -> length of text string
# m -> length of pattern string
# lim -> maximum number of characters from alphabet used
n, m, lim = map(int, sys.argv[1:4])
if lim == 2:

    cur = 0
    pattern = ''.join(choice(ascii_lowercase[:min(lim, len(ascii_lowercase))]) for _ in range(m))
    pattern = 'ab' * (m // 2)

    while cur + m <= n:
        if cur + 2 * m > n:
            print(pattern, end='')
        else:
            print(pattern[:-1], 'x', sep='', end='')
        cur += m

    if cur < n:
        print(''.join(choice(ascii_lowercase[:min(lim, len(ascii_lowercase))]) for _ in range(n - cur)))
    else:
        print()

    print(pattern)

else:
    print(''.join(choice(ascii_lowercase[:min(lim, len(ascii_lowercase))]) for _ in range(n)))
    print(''.join(choice(ascii_lowercase[:min(lim, len(ascii_lowercase))]) for _ in range(m)))

```

Figure 3.1.1 - Random Input Generator Implementation in Python 3.7

We generated 12 different inputs in total with the generator the specifications of the inputs are stated in Table 3.1.1.

Name	Text Length	Pattern Length	Number of Letters that Used
Input 1	1k	33	4
Input 2	1k	33	26
Input 3	1k	300	4
Input 4	1k	300	26
Input 5	10k	1k	4
Input 6	10k	1k	26
Input 7	10k	3k	4
Input 8	10k	3k	26
Input 9	100k	333	4
Input 10	100k	333	26
Input 11	100k	30k	4
Input 12	100k	30k	26

Table 3.1.1 - Specifications of the input files that generated

Algorithms

We implemented our algorithms with C++ 14. We run all algorithms on all input set and collect the results and analyze them. You can see our implementation in Figure 3.2.1 - Figure 3.2.4.

```
inline bool naive_search( string &text, string &pattern ) {

    for( int i=0 ; i <= (int)text.size() - (int)pattern.size() ; i++ ) {

        bool flag = true;

        for( int j=0 ; j < (int)pattern.size() ; j++ )
            if ( text[i + j] != pattern[j] ) {
                flag = false;
                break;
            }

        if (flag) {
            return true;
        }
    }

    return false;
}
```

Figure 3.2.1 - Implementation of Naive Approach in C++ 14

```
inline bool kmp( string &text, string &pattern ) {

    int next[(int)pattern.size() + 1];
    int result = 0;

    memset(next, 0, sizeof next);

    for( int i=1 ; i < (int)pattern.size() ; i++ ) {

        int j = next[i + 1];

        while( j > 0 && pattern[i] != pattern[j] )
            j = next[j];

        if( j > 0 || pattern[i] == pattern[j] )
            next[i + 1] = j + 1;
    }

    for( int i=0, j=0 ; i < (int)text.size() ; i++ ) {

        if( text[i] == pattern[j] ) {

            j++;

            if( j == (int)pattern.size() ) {
                return true;
            }
        } else if( j > 0 ) {
            j = next[j];
            i--;
        }
    }

    return false;
}
```

Figure 3.2.2 - Implementation of the Knutt-Morris-Pratt (KMP) Algorithm in C++ 14

```cpp
inline bool rabin_karp(string const& t, string const& s) {

    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();

    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;

    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(T + 1, 0);

    for (int i = 0; i < T; i++)
        h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;

    long long h_s = 0;

    for (int i = 0; i < S; i++)
        h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

    for (int i = 0; i + S - 1 < T; i++) {

        long long cur_h = (h[i+S] + m - h[i]) % m;

        if (cur_h == h_s * p_pow[i] % m)
            return true;
    }

    return false;
}
```

Figure 3.2.3 - Implementation of Rabin Karp Algorithm in C++ 14.

```cpp
int z[MAXN];    // Z array, z[i] holds maximum length we can match
                //with prefix of string s starting from ith indice

char s[MAXN];   // Initial string

inline int z_function(string &text, string &pattern) {

    int l = 0, r = 0;                                   // Left and right boundaries of current prefix match
    int result = 0;

    string s = pattern + "#" + text;

    for( int i = 1 ; i < (int)s.size() ; i++ ) {

        z[i] = max(0, min(z[i - l], r - i + 1));        // We can use previously calculated values for
                                                        //new z[i] value, but we can not go beyond right boundary
                                                        //without checking values

        while (i + z[i] < (int)s.size() && s[i + z[i]] == s[z[i]])  // Checking values beyond right boundary in
                                                                    //a simple way
            z[i]++;

        if( i + z[i] - 1 > r ) {                        // Setting new boundaries
            l = i;
            r = i + z[i] - 1;
        }

        if (z[i] == (int)pattern.size()) {
            return true;
        }
    }

    return false;
}
```

Figure 3.2.4 - Implementation of the Z Algorithm in C++ 14.

We used for clock() function from the ctime library of the C++ for the time measurements. In every test sequence we started the clock right before the preprocessing and/or searching operations began and we stopped it right after it.

Results

We tested every input that we stated in Table 3.1.1 for our 4 algorithms and the results of the experiments are stated in Table 4.1.

Input/Algorithm	Naive Approach	KMP Algorithm	Rabin Karp Algorithm	Z Algorithm
Input 1	$3.7x10^{-5}$ s	$4.1x10^{-5}$ s	$7.5x10^{-5}$ s	$6.1x10^{-5}$ s
Input 2	$2.6x10^{-5}$ s	$3.9x10^{-5}$ s	0.000111 s	$5.5x10^{-5}$ s
Input 3	$5.2x10^{-5}$ s	$5.4x10^{-5}$ s	$6.3x10^{-5}$ s	$5.1x10^{-5}$ s
Input 4	$3.3x10^{-5}$ s	$4.9x10^{-5}$ s	0.000121 s	$5.7x10^{-5}$ s
Input 5	0.00175 s	0.000322 s	0.000302 s	0.000542 s
Input 6	$5.7x10^{-5}$ s	$8.8x10^{-5}$ s	0.000255 s	0.000309 s
Input 7	0.00263 s	$9.8x10^{-5}$ s	0.000611 s	0.000229 s
Input 8	$3.6x10^{-5}$ s	0.000316 s	0.000352 s	0.000284 s
Input 9	0.004994 s	0.000176 s	0.006337 s	0.002075 s
Input 10	0.000173 s	0.000232 s	0.005382 s	0.001676 s
Input 11	0.237052 s	0.000234 s	0.00583 s	0.001586 s
Input 12	0.000154 s	0.00086 s	0.002925 s	0.005271 s

Table 4.1 - Experiment Results (in seconds)

Analysis

After we did the experiment with 12 different input set we compare the results and make some deductions about the results and algorithms. Despite the performance on the smaller inputs, we can clearly see that the Naive Approach cannot make an efficient performance in the bigger inputs. The other algorithms have the same time complexity O(n+m) we can see the results are very close, but since we added an attribute to our input set like the number of letters you can see small differences between the algorithms. Also since the hash cost does not change that much, Rabin Karp Algorithm is more independent from the input, in other words, it is more stable.

Evaluation

We divided our project into 4 parts:

- Generating Inputs
- Implementing the Algorithms
- Testing and Saving the Results
- Writing the Final Report

and we successfully completed our all tasks on time. We progressed according to our RACI chart. We progressed our tasks in the given order and used the technologies that we stated in the report and the proposal.

Conclusion

If we summarize the whole report we can say that we experimented 4 algorithms 3 of them run in linear time in addition we tested their dependencies on the input specifications and we presented the results of the experiment and our analysis in this report.

References

[1] Knuth, D. E., Morris, Jr., J. H., & Pratt, V. R. (1977). Fast Pattern Matching in Strings. SIAM Journal on Computing, 6(2), 323–350.

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

[3] Z-function and its calculation (n.d.). Retrieved from https://cp-algorithms.com/string/z-function.html

[4] String Hashing (n.d.) Retrieved from https://cp-algorithms.com/string/string-hashing.html