

REPORT

Project 3:Pattern Matching Algorithms

Submitted By

Abdullah Al Raqibul Islam (ID# 801151189)

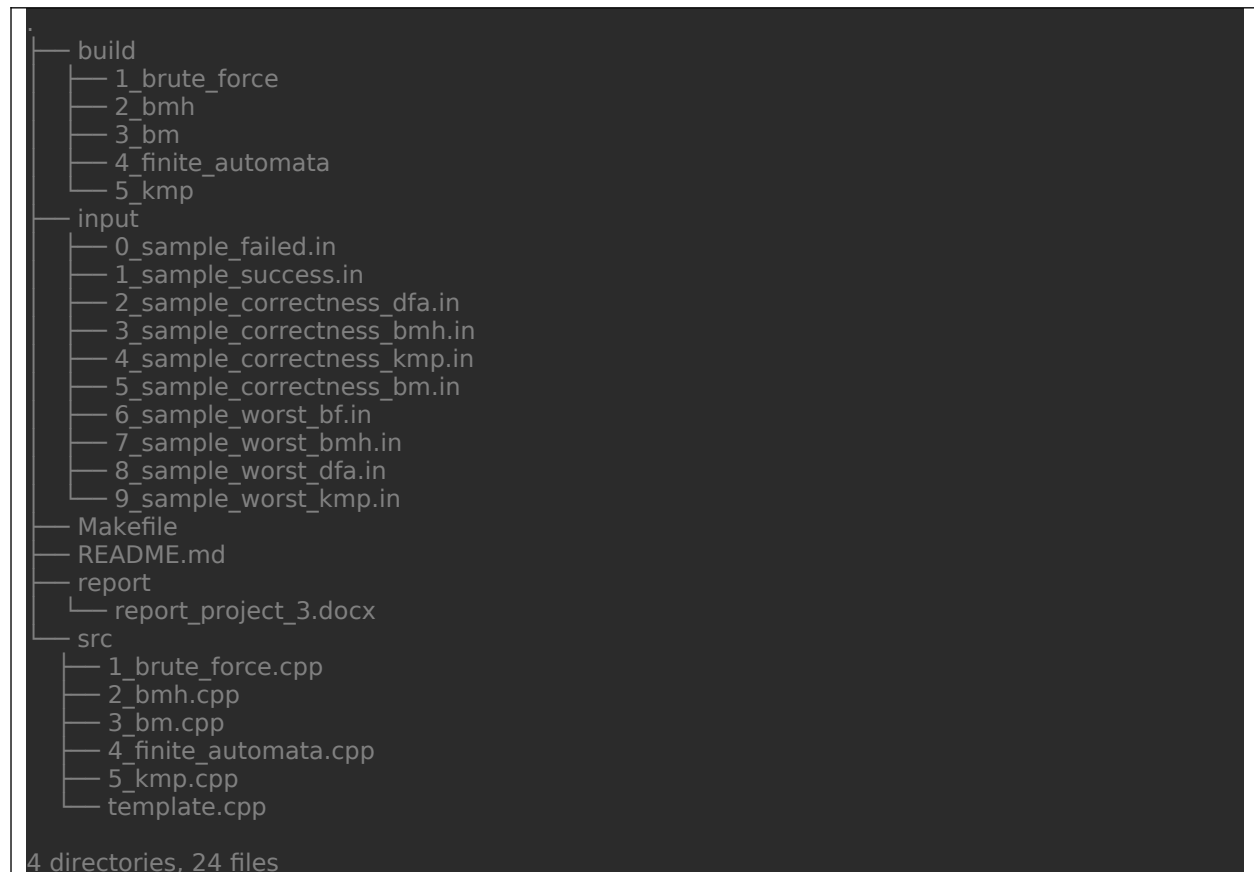
Project Objective

In this project, I have implemented five different pattern matching algorithms and compare their performance by calculating the number of character comparisons required to match the pattern in a text string. I have implemented the following pattern matching algorithms,

1. Brute-Force algorithm
2. Boyer-Moore-Horspool algorithm
3. Boyer-Moore algorithm
4. Finite automata for pattern matching
5. Knuth-Morris-Pratt algorithm

Code Repository Overview

I have used C++ as the programming language for this project. The project structure looks like this



```
.
├── build
│   ├── 1_brute_force
│   ├── 2_bmh
│   ├── 3_bm
│   ├── 4_finite_automata
│   └── 5_kmp
├── input
│   ├── 0_sample_failed.in
│   ├── 1_sample_success.in
│   ├── 2_sample_correctness_dfa.in
│   ├── 3_sample_correctness_bmh.in
│   ├── 4_sample_correctness_kmp.in
│   ├── 5_sample_correctness_bm.in
│   ├── 6_sample_worst_bf.in
│   ├── 7_sample_worst_bmh.in
│   ├── 8_sample_worst_dfa.in
│   └── 9_sample_worst_kmp.in
├── Makefile
├── README.md
├── report
│   └── report_project_3.docx
└── src
    ├── 1_brute_force.cpp
    ├── 2_bmh.cpp
    ├── 3_bm.cpp
    ├── 4_finite_automata.cpp
    ├── 5_kmp.cpp
    └── template.cpp

4 directories, 24 files
```

Here I am going to give brief description of the purpose of the sub-directories and files of this project:

- /build directory contains the executable program files.
- /input directory contains the sample pattern matching inputs. For every input set, I expect 2 lines of string. The first line indicate the text and the second line will contains the pattern to match in the text.
- Makefile will help compiling the project programs and place output executable files in the /build directory.
- README.md contains instructions to build the project and way of running the programs to test further.
- /report contains the project reports and helper files for it.
- /src holds the implementations of the algorithms to solve the pattern matching problem.

Note

For runtime analysis I have used standard notations, i.e. **m** will represent the pattern length and **n** will represent length of text. Also my implementations are guaranteed to work properly only for the upper case letters (in both text and pattern).

Project Run Instruction

```
# go to project directory
$ cd project_3

# build the project
$ make

# command to run solution for Brute-Force algorithm
# general instruction: $ ./build/1_brute_force < input_file > output_file
# here is a sample
$ ./build/1_brute_force < input/0_sample_failed.in > sample_bf.out

# command to run solution for Boyer-Moore-Horspool algorithm
# general instruction: $ ./build/2_bmh < input_file > output_file
# here is a sample
$ ./build/2_bmh < input/0_sample_failed.in > sample_bmh.out

# command to run solution for Boyer-Moore algorithm
```

```

# general instruction: $ ./build/3_bm < input_file > output_file
# here is a sample
$ ./build/3_bm < input/0_sample_failed.in > sample_bm.out

# command to run solution for Finite automation for pattern matching
# general instruction: $ ./build/4_finite_automata < input_file >
output_file
# here is a sample
$ ./build/4_finite_automata < input/0_sample_failed.in > sample_fa.out

# command to run solution for Knuth-Morris-Pratt algorithm
# general instruction: $ ./build/5_kmp < input_file > output_file
# here is a sample
$ ./build/5_kmp < input/0_sample_failed.in > sample_kmp.out

```

Sample Input Set

Here is the list of my sample input set that I have used to compare the performance of all the pattern matching algorithms mentioned earlier,

0	TETTHTEHHEEHTHTEEHETHHTHTHHEHTHTHT THE
1	TETTHTEHHEEHTHTEEHETHHTHTHHEHTHTHE THE
2	AABACAABABACAA ABABAC
3	TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT TCCTATTCTT
4	ABABABAABACABACABC ABACABC
5	CBBAABAABBCABAAABBBABBAAB ABBAAB
6	AAAAAAAAAAAAAAAAAAAAAAAAAB AAAAAAB
7	AAAAAAAAAAAAAAAAABAAAAAA BAAAAAA
8	ABABCABCDABCDEABCDEFABCDEFABCDEF ABCDEF
9	AAAAAABAAAAAABAAAAAABAAAAAA AAAAAA

Algorithm 1: Brute-Force Algorithm

Short description: In this task, I have implemented brute force algorithm for matching pattern in a text. This algorithm compares the pattern with the text for each possible shift of pattern relative to text, until either a match is found, or, all placements of the pattern have been tried.

Data-structure: In my solution, I have taken the input of both text and pattern as a C++ string. There is no other additional data structure required for this implementation.

Runtime analysis: The brute force algorithm runs in $O(nm)$.

Algorithm 2: Boyer-Moor-Horspool Algorithm

Short description: For this task, I have implemented Boyer-Moor-Horspool algorithm for matching pattern in a text. This algorithm required additional pre-processing on the pattern to make some intelligent shift instead of trying every possible shifts. For doing that, I have prepared a shift table to determine how much to shift the pattern when a mismatch occurs. Always makes a shift based on the shift table value of the text's character aligned with the last character of the pattern.

Data-structure: I have used an integer type array to represent the shift table. The length of table is 26 (considering upper case English characters) and contains the number of shift in case of failure.

Runtime analysis: The Boyer-Moor-Horspool algorithm's preprocessing requires $O(m)$. In worst case, the matching algorithm runs in $O(nm)$, and in best cases it runs in $O(n/m)$.

Algorithm 3: Boyer-Moor Algorithm

Short description: In this task, I have implemented Boyer-Moor algorithm for matching pattern in a text. This algorithm required two additional pre-processing on the pattern to make more intelligent shift (instead of trying every possible options) comparing to Boyer-Moor-Horspool algorithm. For doing that, I have prepared two shift tables, one for bad symbol shift and another one for good suffix

shift. In case of mismatch, we consider the best of the two shift table values to determine how much to shift the pattern.

Data-structure: In my solution, I have used two array to handle the bad symbol and good suffix scenarios.

Runtime analysis: Both of the pre-processing (bad symbol shift table and good suffix shift table) required $O(m)$. In worst case, $O(nm)$ comparisons are necessary. If there are only a constant number of matches of the pattern in the text, it performs in $O(n)$ comparisons. If the alphabet is large compared to the length of the pattern, the algorithm performs $O(n/m)$ comparison on the average. This is because often a shift by m is possible due to the bad character heuristics.

Algorithm 4: Finite Automation for Pattern Matching

Short description: In this task, I have implemented Finite Automation algorithm for matching pattern in a text. This algorithm preprocessed a simple state machine for processing information – that scans the text string for all occurrences of the pattern. The state machine then performs like this,

- if there is a match, move to a new state
- otherwise, make a transition to a state to avoid repeated comparisons

Data-structure: In my solution, I have used two additional array to keep the KMP prefix function value and states of the finite automation.

Runtime analysis: To prepare the state machine, I have used the assistance of KMP prefix function. The overall preprocessing requires $O(m + S*m)$ where S represents the number of unique characters. The matching algorithm requires $O(n)$ comparisons.

Algorithm 5: Knuth-Morris-Pratt Algorithm

Short description: In this task, I have implemented Knuth-Morris-Pratt algorithm for matching pattern in a text. The prefix function that I have implemented for assisting the pattern matching, is an online algorithm, i.e. it processes the data as it

arrives – for example, you can read the string characters one by one and process them immediately, finding the value of prefix function for each next character.

Data-structure: In my solution, I have used an additional array to keep the prefix function value which will then help the matching algorithm.

Runtime analysis: The preprocessing part will take $O(m)$, and matching part required $O(m+n)$.

Algorithm Performance Analysis (w.r.t. comparison required):

Algorithm	0	1	2	3	4	5	6	7	8	9
Brute-Force Algorithm	49	49	18	56	35	39	144	25	62	91
Boyer-Moor-Horspool	22	24	10	38	13	31	25	89	13	10
Boyer-Moor	22	24	10	20	13	24	25	31	13	10
Finite Automaton	33	33	12	38	18	25	25	25	35	28
Knuth-Morris-Pratt	44	43	16	52	23	33	42	25	41	46

Test platform

- Processor: Intel(R) Xeon(R) CPU E5-2620 2.00GHz (12 Core)
- Linux version: 5.0.0-27-generic
- g++ version: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

Further Work

In this work, my intention was to compare the performance of different pattern matching algorithms w.r.t. the required character comparisons to find the the match. It is further possible to generate large test cases and compare the runtime of these pattern matching algorithms.

References

- Book: Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- CP-Algorithms (Prefix function): <https://cp-algorithms.com/string/prefix-function.html>
- Boyer-Moore algorithm: www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm

Code

Implementation of algorithm 1: Brute-Force algorithm

```
string text, pattern;
int text_len, pattern_len;
int number_of_comparison;

int brute_force_matching() {
    number_of_comparison = 0;
    for(int i=0; i<=(text_len-pattern_len); i+=1) {
        int j = 0;
        while(j<pattern_len && text[i+j] == pattern[j]) {
            j+=1;
            number_of_comparison += 1;
        }
        if(j == pattern_len) return i;

        number_of_comparison += 1; //for failed match
    }
    return -1;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    //load input after this line
    getline(cin, text);
    getline(cin, pattern);
    text_len = text.length();
    pattern_len = pattern.length();

    double st = clock();
    int match_found = brute_force_matching();
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;

    if(match_found == -1) printf("pattern not matched in the text; # of comparison required: %d\n",
number_of_comparison);
    else printf("pattern matched in the text at text position: %d; # of comparison required: %d\n",
match_found, number_of_comparison);

    return 0;
}
```

Implementation of algorithm 2: Boyer-Moore-Horspool algorithm

```
#pragma warning ( disable : 4786 )

#include <iostream>
#include <sstream>
```

```

#include <stdio>
#include <stdlib>
#include <cmath>
#define scale(x) (x-'A')
#define rev_scale(x) (x+'A')

const int MAX = 1000005;
const int inf = (1 << 28);

string text, pattern;
int text_len, pattern_len;
int shift_table[30];
int number_of_comparison;

void build_shift_table() {
    for(int i=0; i<30; i+=1) shift_table[i] = pattern_len;
    for(int j=1, i=pattern_len-2; i>=0; j+=1, i-=1) {
        if(shift_table[scale(pattern[i])] == pattern_len) shift_table[scale(pattern[i])] = j;
    }
}

void print_shift_table() {
    for(int i=0; i<26; i+=1) {
        printf("%c:%d ", rev_scale(i), shift_table[i]);
    }
    printf("\n");
}

int horspool_matching() {
    build_shift_table();

    number_of_comparison = 0;
    int i = pattern_len - 1;
    while(i < text_len) {
        int j = 0;
        while(j<pattern_len && pattern[pattern_len-1-j] == text[i-j]) {
            j+= 1;
            number_of_comparison += 1;
        }
        if(j == pattern_len) return (i - pattern_len + 1);

        number_of_comparison += 1; //for failed match
        i += shift_table[scale(text[i])];
    }
    return -1;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    getline(cin, text);
    getline(cin, pattern);
    text_len = text.length();
    pattern_len = pattern.length();

```

```

double st = clock();
int match_found = horspool_matching();
cerr << (clock() - st) / CLOCKS_PER_SEC << endl;

if(match_found == -1) printf("pattern not matched in the text; # of comparison required: %d\n",
number_of_comparison);
else printf("pattern matched in the text at text position: %d; # of comparison required: %d\n",
match_found, number_of_comparison);

return 0;
}

```

Implementation of algorithm 3: Boyer-Moore algorithm

```

#define scale(x) (x-'A')
#define rev_scale(x) (x+'A')

const int MAX = 1000005;
const int inf = (1 << 28);

string text, pattern;
int text_len, pattern_len;
int bs_shift_table[30], gs_shift_table[MAX], pos[MAX];
int number_of_comparison;

int _max(int a, int b) {
    return a > b ? a : b;
}

void build_bad_symbol_shift_table() {
    for(int i=0; i<30; i+=1) bs_shift_table[i] = pattern_len;
    for(int j=1, i=pattern_len-2; i>=0; j+=1, i-=1) {
        if(bs_shift_table[scale(pattern[i])] == pattern_len) bs_shift_table[scale(pattern[i])] = j;
    }
}

void full_suffix_match() {
    int i = pattern_len, j = pattern_len+1;
    pos[i] = j;
    while(i > 0) {
        while ((j <= pattern_len) && (pattern[i-1] != pattern[j-1])) {
            if(gs_shift_table[j] == 0) gs_shift_table[j] = j-i;
            j = pos[j];
        }
        i -= 1;
        j -= 1;
        pos[i] = j;
    }
}

void partial_suffix_match() {
    int i, j;
    j = pos[0];
    for(i=0; i <= pattern_len; i+=1) {
        if(gs_shift_table[i] == 0) gs_shift_table[i]=j;
        if(i == j) j = pos[j];
    }
}

```

```

}

void build_good_suffix_shift_table() {
    full_suffix_match();
    partial_suffix_match();
}

void print_good_suffix_shift_table() {
    for(int i=0; i<=pattern_len; i+=1) {
        printf("%d: %d\n", i, gs_shift_table[i]);
    }
}

void print_bad_symbol_shift_table() {
    for(int i=0; i<26; i+=1) {
        printf("%c:%d ", rev_scale(i), bs_shift_table[i]);
    }
    printf("\n");
}

int bm_matching() {
    build_bad_symbol_shift_table();
    build_good_suffix_shift_table();

    number_of_comparison = 0;
    int i = pattern_len - 1;
    while(i < text_len) {
        int j = 0;
        while(j<pattern_len && pattern[pattern_len-1-j] == text[i-j]) {
            j+= 1;
            number_of_comparison += 1;
        }
        if(j == pattern_len) return (i - pattern_len + 1);

        number_of_comparison += 1; //for failed match
        i += _max(gs_shift_table[pattern_len-j], (bs_shift_table[scale(text[i-j])] - j));
    }
    return -1;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    getline(cin, text);
    getline(cin, pattern);
    text_len = text.length();
    pattern_len = pattern.length();

    double st = clock();
    int match_found = bm_matching();
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;

    if(match_found == -1) printf("pattern not matched in the text; # of comparison required: %d\n",
number_of_comparison);
    else printf("pattern matched in the text at text position: %d; # of comparison required: %d\n",

```

```

match_found, number_of_comparison);

    return 0;
}

```

Implementation of algorithm 4: Finite automation

```

#define scale(x) (x-'A')
#define rev_scale(x) (x+'A')

const int MAX = 1000005;
const int inf = (1 << 28);

string text, pattern;
int text_len, pattern_len;
int dfa[30][MAX], pi[MAX];
int number_of_comparison;

//kmp_failure_function_fast
void prefix_function() {
    pi[0] = 0;
    for(int i=1; i<pattern_len; i+=1) {
        int j = pi[i-1];

        while(j>0 && pattern[i] != pattern[j]) j = pi[j-1];
        if(pattern[i] == pattern[j]) j += 1;
        pi[i] = j;
    }
}

void calculate_dfa_table() {
    prefix_function();
    for (int i = 0; i < pattern_len; i+=1) {
        for (int c = 0; c < 26; c+=1) {
            if (i > 0 && rev_scale(c) != pattern[i]) dfa[c][i] = dfa[c][pi[i-1]];
            else dfa[c][i] = i + (rev_scale(c) == pattern[i]);
        }
    }
}

void print_dfa_table() {
    for(int i=0; i<26; i+=1) {
        for(int j=0; j<pattern_len; j+=1) {
            printf("%d ", dfa[i][j]);
        }
        printf("\n");
    }
}

int dfa_matching() {
    calculate_dfa_table();

    number_of_comparison = 0;
    int current_state = 0;
    for(int i=0; i<text_len; i+=1) {
        number_of_comparison += 1;
        current_state = dfa[scale(text[i])][current_state];
    }
}

```

```

        if(current_state == pattern_len) return (i - pattern_len + 1);
    }
    return -1;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    getline(cin, text);
    getline(cin, pattern);
    text_len = text.length();
    pattern_len = pattern.length();

    double st = clock();
    int match_found = dfa_matching();
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;

    if(match_found == -1) printf("pattern not matched in the text; # of comparison required: %d\n",
number_of_comparison);
    else printf("pattern matched in the text at text position: %d; # of comparison required: %d\n",
match_found, number_of_comparison);

    return 0;
}

```

Implementation of algorithm 5: Knuth-Morris-Pratt algorithm

```

const int MAX = 1000005;
const int inf = (1 << 28);

string text, pattern;
int text_len, pattern_len;
int pi[MAX];
int number_of_comparison;

void kmp_failure_function_naive() {
    for(int i=0; i<pattern_len; i+=1) {
        for(int j=i; j>=0; j-=1) {
            if(pattern.substr(0, j) == pattern.substr(i-j+1, j)) {
                pi[i] = j;
                break;
            }
        }
    }
}

void kmp_failure_function_fast() {
    pi[0] = 0;
    for(int i=1; i<pattern_len; i+=1) {
        int j = pi[i-1];

        while(j>0 && pattern[i] != pattern[j]) j = pi[j-1];
        if(pattern[i] == pattern[j]) j += 1;
    }
}

```

```

        pi[i] = j;
    }
}

void print_kmp_failure_table() {
    for(int i=0; i<pattern_len; i+=1) printf("%d ", pi[i]);
    printf("\n");
}

int kmp_matching() {
    kmp_failure_function_fast();

    number_of_comparison = 0;
    int i = 0, j = 0;
    while(i < text_len) {
        number_of_comparison += 1;
        if(text[i] == pattern[j]) {
            if(j == pattern_len-1) return i - j;
            else {
                i += 1;
                j += 1;
            }
        }
        else {
            if(j > 0) j = pi[j - 1];
            else i += 1;
        }
    }
    return -1;
}

int main() {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int i, j, k;
    int test, t = 0, kase = 0;

    //load input after this line
    getline(cin, text);
    getline(cin, pattern);
    text_len = text.length();
    pattern_len = pattern.length();

    double st = clock();
    int match_found = kmp_matching();
    cerr << (clock() - st) / CLOCKS_PER_SEC << endl;

    if(match_found == -1) printf("pattern not matched in the text; # of comparison required: %d\n",
number_of_comparison);
    else printf("pattern matched in the text at text position: %d; # of comparison required: %d\n",
match_found, number_of_comparison);

    return 0;
}

```