**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 there 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 automation 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 graph inputs
* Makefile will help compiling the project programs and place output executable files in the /build directory
* /problem\_statement holds the problem statement of this project
* 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 programs to solve the problems of this project

**Note**

For runtime analysis I have used standard notations, i.e. B will represent number of nodes and **E** will represent number of edges.

**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/sample.in > sample\_bf.out  # command to run solution for Boyer-Moore algorithm # general instruction: $ ./build/2\_bmh < input\_file > output\_file # here is a sample $ ./build/2\_bmh < input/sample.in > sample\_bmh.out  # command to run solution for Boyer-Moore-Horspool algorithm # general instruction: $ ./build/3\_bm < input\_file > output\_file # here is a sample $ ./build/3\_bm < input/sample.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/sample.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/sample.in > sample\_kmp.out |

**Algorithm 1: Brute-Force Algorithm**

**Short description:** In this task, I have implemented brute force algorithm for matching pattern in a text.

**Data-structure:** In my solution,I have used …

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

**Sample Input Set**

Here is the list of my sample input set,

|  |  |
| --- | --- |
| 0 | TETTHTEHHEEHTHTEEHETHTHTHHEHTHTHT  THE |
| 1 | TETTHTEHHEEHTHTEEHETHTHTHHEHTHTHE  THE |
| 2 | AABACAABABACAA  ABABAC |
| 3 | TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT  TCCTATTCTT |
| 4 | ABABABAABACABACABC  ABACABC |
| 5 | CBBAABAABBCABAAABBBABBAAB  ABBAAB |
| 6 | AAAAAAAAAAAAAAAAAAAAAAAAB  AAAAAAAB |
| 7 | AAAAAAAAAAAAAAAAABAAAAAAA  BAAAAAAA |
| 8 | ABABCABCDABCDEABCDEFABCDEFGABCDEFGH  ABCDEFGH |
| 9 | AAAAAABAAAAAABAAAAAABAAAAAAA  AAAAAAA |

**Algorithm 2: Boyer-Moor-Horspool Algorithm**

**Short description:** For this task, I have implemented Boyer-Moor-Horspool algorithm for matching pattern in a text.

**Data-structure:** I have used …

**Runtime analysis:** The Boyer-Moor-Horspool algorithm’s preprocessing requires **O(S+m)**. In worst case, this algorithm runs in **O(nm)**, and in best cases it runs in **O(n/m)**.

**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 Automation | 33 | 33 | 12 | 38 | 18 | 25 | 25 | 25 | 35 | 28 |
| Knuth-Morris-Pratt | 44 | 43 | 16 | 52 | 23 | 33 | 42 | 25 | 41 | 46 |

**Algorithm 3: Boyer-Moor Algorithm**

**Short description:** In this task, I have implemented brute force algorithm for matching pattern in a text.

**Data-structure:** In my solution,I have used …

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

**Algorithm 4: Finite Automation for Pattern Matching**

**Short description:** In this task, I have implemented brute force algorithm for matching pattern in a text.

**Data-structure:** In my solution,I have used …

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

**Algorithm 5: Knuth-Morris-Pratt Algorithm**

**Short description:** In this task, I have implemented brute force algorithm for matching pattern in a text.

**Data-structure:** In my solution,I have used …

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

**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](http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm)

**Code**

**Implementation of algorithm 1:**

|  |
| --- |
| 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:**

|  |
| --- |
| #pragma warning ( disable : 4786 )  #include <iostream> #include <sstream>  #include <cstdio> #include <cstdlib> #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:**

|  |
| --- |
| #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:**

|  |
| --- |
| #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:**

|  |
| --- |
| 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; } |