A picture containing font, logo, text, symbol

Description automatically generated

Marmara University   
Faculty of Engineering

CSE 2246

ALGORITHM ANALYSIS

**Homework 1 Report**

Instructure: ÖMER KORÇAK Date:17.05.2023

|  |  |  |  |
| --- | --- | --- | --- |
|  | Department | Student Id Number | Name & Surname |
| 1 | CSE | 150120022 | Tolga Fehmioğlu |
| 2 | CSE | 150121538 | Muhammed Enes Gökdeniz |
| 3 | CSE | 150121032 | Mehmet Toprak Balıkçı |
| 4 | CSE | 150121002 | Enes Torluoğlu |

**THE COMPLEXITY ANALYSİS OF ALGORITHMS**:

**a)Brute Force Approach:**

Given a pattern M characters in length, and a text N characters in length.

**Worst case**: Compares pattern to each substring of text of length M.

• Total number of comparisons: M (N-M+1) • Worst case time complexity: Ο(MN)

**Best case:** if pattern found: Finds pattern in first M positions of text.

• Total number of comparisons: M • Best case time complexity: Ο(M).

**Best case:** if pattern not found: Always mismatch on first character.

• Total number of comparisons: N • Best case time complexity: Ο(N).

**b) Horspool Algorithm:**

**Worst-case performance**: O(nm)

**Average performance**: Θ(n)

where the length of the pattern is m and the length of the search string is n.

**c) Boyer Moore’s Algorithm:**

**In the worst-case**: The performance of the Boyer-Moore-Horspool algorithm is O(mn), where m is the length of the substring and n is the length of the string.

**The average time** is O(n). **In the best case**, the performance is sub-linear, and is, in fact, identical to Boyer-Moore original implementation.

COMPLEXITY ANALYSIS TABLE:

|  |  |  |  |
| --- | --- | --- | --- |
| ALGORITHMS | WORST CASE | AVERAGE CASE | BEST CASE |
| Brute Force | Ο(MN) | Ο(MN) | Ο(M), Ο(N) |
| Horspool | O(nm) | O(n) | O(n) |
| Boyer Moore | O(nm) | O(n) | O(n) |

TIMING RESULTS:

1. **Bit Strings:**

Input Size vs Time (Pattern is constant)

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithms | Input Size | Time | Pattern  (101) |
| Brute Force | 1mb | 31 ms |
| Horspool | 1mb | 53 ms |
| Boyer Moore | 1mb | 50 ms |
| Brute Force | 2mb | 46 ms |
| Horspool | 2mb | 69 ms |
| Boyer Moore | 2 mb | 70 ms |
| Brute Force | 5 mb | 95  ms |
| Horspool | 5 mb | 117 ms |
| Boyer Moore | 5 mb | 131 ms |
| Brute Force | 10 mb | 161 ms |
| Horspool | 10 mb | 203 ms |
| Boyer Moore | 10 mb | 233 ms |
| Brute Force | 20mb | 161ms |
| Horspool | 20 mb | 203 ms |
| Boyer Moore | 20 mb | 233 ms |
| Brute Force | 50mb | 612 ms |
| Horspool | 50mb | 855 ms |
| Boyer Moore | 50 mb | 418 ms |
| Brute Force | 100 mb | 1833 ms |
| Horspool | 100 mb | 1693 ms |
| Boyer Moore | 100 mb | 2060 ms |

Pattern vs Time: ( The input is selected randomly which size is 10 mb)

|  |  |  |
| --- | --- | --- |
| Algorithms | Pattern | Time |
| Brute Force | **“10101”** | 152 ms |
| Horspool | **“10101”** | 208  ms |
| Boyer Moore | **“10101”** | 186  ms |
| Brute Force | **“010101101”** | 157  ms |
| Horspool | **“010101101”** | 202  ms |
| Boyer Moore | **“010101101”** | 144  ms |
| Brute Force | **“101010101001010110101”** | 120  ms |
| Horspool | **“101010101001010110101”** | 214 ms |
| Boyer Moore | **“101010101001010110101”** | 130 ms |
| Brute Force | **“10101010100101001010100101010101010101010101”** | 106 ms |
| Horspool | **“10101010100101001010100101010101010101010101”** | 199 ms |
| Boyer Moore | **“10101010100101001010100101010101010101010101”** | 48  ms |

Worst Case Scenario Sample:

(All text consisting of zeros 00000….)

Pattern is: “100”

Input Size is 50 mb

Brute Force: 57 ms

Horspool: 741 ms

Boyer Moore: 440 ms

Best Case Scenario Sample

(All text consisting of zeros 00000….)

Pattern is: 111111111111111

Input Size is 50mb:

Brute Force: 57 ms

Horspool: 40 ms

Boyer Moore: 45 ms

So, as we can see from the Input Size vs Time table, where the pattern is constant and 101, for smaller input files brute force is more time efficient. As the input file size grows the time difference between Brute Force and the other algorithms decreases. However, we can’t see the advantage of Boyer Moore algorithm even for 100 mb files. But if we have look at the Pattern vs Time table, where the input file size is constant and 10 mb, Boyer Moore's advantage becomes apparent as the pattern size increases. For pattern “10101”, 5 bits pattern, Brute Force algorithm time is 152 ms while Boyer Moore algorithms time is 186 ms. But for the pattern size of 44, Brute Force time is 106 ms while Boyer Moore’s is only 48 seconds which is less than half of Brute Force Algorithm’s time. So, we can conclude that the bigger pattern size the more obvious Boyer Moore’s advantages. Also, for bigger file sizes Boyer Moore and Horspool’s advantage becomes more apparent. But for Horspool algorithm times in the second table, even though it compares less than Brute Force we can’t say it is quicker than Brute Force. This is most probably because of the bit pattern, since it is all 1s and 0s the algorithm can’t jump a lot and since looking up to shift table and reaching that value from memory takes time, it is not more advantageous than Brute Force. But for strings, since there are lots of different letters in a string, Horspool jumps more often than when we use bit pattern.

1. English Texts: We prepared 5 different input texts and different patterns based on these patterns. These are the time results of algorithms in different patterns.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Texts** | **Words** | **Brute Force** | **Horspool** | **Boyer Moore** |
| Text1 | existence | 36 ms | 29 ms | 30 ms |
| Text1 | because | 43 ms | 47 ms | 58 ms |
| Text1 | therefore | 51 ms | 29 ms | 30 ms |
| Text1 | beautiful | 33 ms | 27 ms | 29 ms |
| Text2 | existence | 51 ms | 47 ms | 56 ms |
| Text2 | because | 44 ms | 47 ms | 57 ms |
| Text2 | therefore | 57 ms | 45 ms | 54 ms |
| Text2 | beautiful | 43 ms | 60 ms | 55 ms |
| Text3 | existence | 28 ms | 39 ms | 42 ms |
| Text3 | because | 45 ms | 49 ms | 52 ms |
| Text3 | therefore | 46 ms | 36 ms | 39 ms |
| Text3 | beautiful | 36 ms | 37 ms | 40 ms |
| Text4 | existence | 26 ms | 22 ms | 23 ms |
| Text4 | because | 26 ms | 23 ms | 25 ms |
| Text4 | therefore | 27 ms | 21 ms | 23 ms |
| Text4 | beautiful | 28 ms | 22 ms | 23 ms |
| Text5 | existence | 25 ms | 19 ms | 21 ms |
| Text5 | because | 24 ms | 20 ms | 22 ms |
| Text5 | therefore | 25 ms | 19 ms | 21 ms |
| Text5 | beautiful | 32 ms | 19 ms | 21 ms |

As we can see in this string tables if the pattern and the file are English text, Boyer Moore and Horspool are much advantageous and most of the time quicker than the Brute Force. It is because while Brute Force goes for 1 by 1 comparison the other algorithms can jump lots of comparison and sometimes even whole word size. And this recovers the time lost in looking at tables and reaching the values in the memory.

**Comparison part**

1. **We used 4 different words as patterns, and 5 different text files. These numbers are comparisons in these algorithms.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word** | **Brute Force** | **Horspol** | **Boyer Moore** | **Text Name** |
| **existence** | 2299689 | 318129 | 302105 | text1.html |
| **because** | 2125528 | 380037 | 373631 | text1.html |
| **therefore** | 2298956 | 316470 | 302460 | text1.html |
| **beautiful** | 2126636 | 282753 | 282753 | text1.html |
| **existence** | 7003750 | 991500 | 938851 | text2.html |
| **because** | 6459590 | 1176803 | 1156715 | text2.html |
| **therefore** | 7054924 | 976174 | 933360 | text2.html |
| **beautiful** | 6457083 | 864532 | 864532 | text2.html |
| **existence** | 3839216 | 516893 | 497988 | text3.html |
| **because** | 3651562 | 701921 | 694454 | text3.html |
| **therefore** | 3822032 | 490112 | 476095 | text3.html |
| **beautiful** | 3652666 | 465885 | 465885 | text3.html |
| **existence** | 1565858 | 216177 | 205672 | text4.html |
| **because** | 1454705 | 259315 | 254964 | text4.html |
| **therefore** | 1572619 | 214046 | 205527 | text4.html |
| **beautiful** | 1454598 | 190930 | 190930 | text4.html |
| **existence** | 1239616 | 171242 | 162678 | text5.html |
| **because** | 1146829 | 204481 | 201114 | text5.html |
| **therefore** | 1240080 | 170281 | 162870 | text5.html |
| **beautiful** | 1147385 | 152343 | 152343 | text5.html |

**2- Comparison numbers for bit patterns, different input sizes.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of comparisons | | | | | |
| Pattern | Patern size | Brute Force | Horspol | Boyer Moore | İnput |
| **100100** | 6 | 4127193 | 2423651 | 2031179 | 2MB |
| **100100** | 6 | 12384573 | 9233661 | 6402707 | 6MB |
| **100100** | 6 | 20640909 | 15397817 | 10675707 | 10MB |
| **1010101010** | 10 | 4190519 | 3142180 | 980522 | 2MB |
| **1010101010** | 10 | 12573954 | 9427953 | 2936739 | 6MB |
| **1010101010** | 10 | 20948393 | 15705799 | 4891551 | 10MB |
| **100101010011100** | 15 | 4192853 | 3143756 | 1205559 | 2MB |
| **100101010011100** | 15 | 12581051 | 9428590 | 3619689 | 6MB |
| **100101010011100** | 15 | 20967856 | 15724256 | 6030402 | 10MB |
| **1111111111** | 10 | 4189796 | 942849 | 478091 | 2MB |
| **1111111111** | 10 | 12565036 | 2837473 | 1429901 | 6MB |
| **1111111111** | 10 | 20938035 | 4740329 | 2382705 | 10MB |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of comparisons | | | | | |
| Pattern | Patern size | Brute Force | Horspol | Boyer Moore | İnput |
| **100100** | 6 | 4127193 | 2423651 | 2031179 | 2MB |
| **101010** | 6 | 4128554 | 3080413 | 1717655 | 2MB |
| **001001** | 6 | 4128158 | 2424034 | 1753170 | 2MB |
| **000100** | 6 | 4128260 | 3079789 | 1968849 | 2MB |
| **111111** | 6 | 4128238 | 1422999 | 913123 | 2MB |
| **000000** | 6 | 4127295 | 1420546 | 912865 | 2MB |
| **1010101010** | 10 | 4190519 | 3142180 | 980522 | 2MB |
| **1000100101** | 10 | 4190770 | 3141036 | 1374256 | 2MB |
| **1011011011** | 10 | 4189012 | 3139944 | 1327127 | 2MB |
| **1000100100** | 10 | 4190770 | 3141454 | 1468433 | 2MB |
| **0011001011** | 10 | 4189004 | 3139603 | 1365080 | 2MB |
| **0100011000** | 10 | 4191143 | 2484933 | 1423089 | 2MB |
| **0110110111** | 10 | 4188711 | 2487595 | 1230576 | 2MB |
| **1111111111** | 10 | 4189796 | 942849 | 478091 | 2MB |
| **0000000000** | 10 | 4188522 | 946860 | 475602 | 2MB |
| **100101010011100** | 15 | 4192853 | 3143756 | 1205559 | 2MB |
| **100111000110110** | 15 | 4191603 | 3143935 | 1378672 | 2MB |
| **111001110011011** | 15 | 4193513 | 3143376 | 1391725 | 2MB |
| **100001000010100** | 15 | 4194015 | 3145018 | 1110075 | 2MB |
| **001100011100010** | 15 | 4193079 | 3145598 | 1289546 | 2MB |
| **010011000101000** | 15 | 4193388 | 3145036 | 1433066 | 2MB |
| **010101010101010** | 15 | 4194178 | 3146197 | 622561 | 2MB |
| **111111111111111** | 15 | 4193730 | 653530 | 300503 | 2MB |
| **000000000000000** | 15 | 4192255 | 654207 | 299171 | 2MB |

How The Code Works?

As you start the code it asks you for type of the text, type 1 for English text, or 2 for bit strings. After this you are asked to read a file, or you want to create a random one. Than it prints out the Bad Table, Good Suffix Table. And at the end occurrence, total comparison numbers and the running time for each algorithms.

Output Example:

**Which type of texts do you want to search(enter 1(English Text) or 2(Bit Strings)): 1**

**Do you want to read a file or create a random text(if the choice is reading enter yes otherwise no): yes**

**Whats the name of the file that you want to read(ex:english.html or give its path(ex: English Text Samples/englishText1.html)): English Text Samples/text2.html**

**finished searching**

**BadTable:**

**t(a)->3**

**t(s)->5**

**t(e)->1**

**t(v)->2**

**t(\*)->6**

**t(l)->4**

**GoodSuffixTable**

**k is 1 d2(1) is 5**

**k is 2 d2(2) is 5**

**k is 3 d2(3) is 5**

**k is 4 d2(4) is 5**

**k is 5 d2(5) is 5**

**k is 6 d2(6) is 5**

**Brute Force Occurrence is 89**

**Brute Force Total comparison is 6697818**

**Brute Force running time is 79 milliseconds**

**HorsPool Occurrence is 89**

**HorsPool Total comparison is 1310739**

**HorsPool running time is 57 milliseconds**

**Boyer Moore Occurrence is 89**

**Boyer Moore comparison is 1310739**

**Boyer Moore running time is 82 milliseconds**