**CS4445b/9544b Analysis of Algorithms II 2016**

**Experimental Evaluation of String Matching Algorithms**

**James Crocker**

**250634027**

**Professor Roberto Solis-Oba**

**MC417**

**April 11th, 2016**

**Introduction**

The following report outlines the procedure taken to implement and subsequently analyze three different string matching algorithms. The algorithms chosen were the Naïve string matching algorithm, Knuth Morris Pratt (KMP) algorithm and the Las Vegas Pattern Matching algorithm described in Section 7.6 of Randomized Algorithms by Motwani and Raghavan. As a reference, the indexOf() method from the Java String class was also tested. For simplicity, the test cases used only strings of lowercase alphabetical characters and all algorithms terminated upon finding the first match (if it existed). By terminating on the first occurrence, none of the algorithms had to take on the overhead of storing the matching positions in an array or list and therefore any runtime measurement was closer to capturing the actual algorithm runtime.

String matching is a problem consisting of given two strings, a text string T=t1t2t3….tn and a pattern string P=p1p2p3….pm,find the occurrence of the pattern in the text. For example, given text “abcabcaabc” and a pattern “caa”, we can say an occurrence of the pattern occurs in the text at position 5. This example provided is simple, but consider a situation where the text string is in fact a document stretching over multiple pages and the pattern string is still relatively small. Although the same strategies can be used to find occurrences, a brute force approach becomes extremely tedious. This is where the importance of a good string matching algorithm comes in. In fact, string matching within a multi-page document is something we see every time we hit the “Ctrl+f” command in a word processor. Other applications include plagiarism detection, pattern matching in DNA sequences and digital forensics (SaiKrishna 219). The three string matching algorithms implemented are described below.

**Naïve String Matching Algorithm**

This algorithm represents a brutefore approach where the pattern is taken and compared to the text at every position. As a result, for a pattern of length ‘m’ and a text of length ‘n’, this algorithm has a running time of O(m(n-m)). Besides the simplicity of this algorithm, one of its key advantages is that it requires no pre-processing, which is needed by other algorithms that will be documented in this paper. Java’s built-in indexOf() method for strings actually uses this algorithm for its implementation.

**Knuth Morris Pratt (KMP) Algorithm**

The large downfall of the Naïve algorithm is backtracking of the pattern when a mismatch is found within the text. For example, consider the text “abcababcab” and the pattern “ababa”. Starting in position 3 of the text, the Naïve algorithm would see that the pattern matches the first 4 characters but there is a mismatch on the 5th character. We have just compared the pattern up to position 7 in the text, but because of the mismatch, we now need to backtrack and restart the comparison at position 4 of the text. KMP avoids this by using a ‘next table’ that determines where the next logical shift should be based on comparing the pattern string with itself. Continuing with the example, after performing the comparison on position 3 of the text, KMP would shift the pattern to position 5 and compare character 2 in the pattern with character 7 in the text. This shift is based on the ‘next table’ computed for this example and can be see in Figure 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 |
| P[i] | a | b | a | b | a |
| Next[i] | 0 | 0 | 1 | 2 | 3 |

Figure 1: Next Table for Pattern String "ababa"

Computation of this ‘next table’ requires O(m) time, while the string matching portion of the algorithm takes O(n) time.

**Las Vegas Pattern Matching**

This algorithm requires calculation of a fingerprint function F of the pattern P, and compares F(P) to F(T(j)) for each position j in the text T. If the two fingerprints are identical, then the pattern is compared to the text at position j for a match. To recalculate F(T(j)) for each position j, Equation (1) is used.

(1)

The prime number p is selected at random from a sequence of primes smaller than a threshold T (Motwani 171). This allows the probability of a false fingerprint match occurring to be O(1/n). For this algorithm, preprocessing takes O(m) since the fingerprint must be calculated for the pattern. The worst case running time for this algorithm is O(n+m).

A summary of the algorithms analyzed in this paper are found in Table 1.

|  |  |  |
| --- | --- | --- |
| Name | Pre-processing Time | Worst Case Running Time |
| Naïve | None | O(m(n-m)) |
| KMP | O(m) | O(m + n) |
| Las Vegas | O(m) | O(m + n) |

Table 1: Worst Case Runtimes for Implemented Algorithms

All these algorithms were tested for inputs of different sizes and despite the theoretical runtimes, experimentally it was observed that the Naïve algorithm performed the best for practical inputs. This was attributed to the algorithm simplicity and low computation overhead.

**Results**

For all algorithms under test, no special data structures had to be developed as the majority of their design required only primitive data types and the use of loops. The implementation of the Naïve algorithm involved using nested loops. The pseudo code for this algorithm can be seen in Figure 2. Java’s indexOf() method uses a highly optimized version of this algorithm.

Algorithm: Naïve(text, pattern)

Input: Text string, pattern string

Output: Location of first occurrence of pattern in text, else -1

For i=0 to text.length() – pattern.length()

j=0

while j < pattern.length() and text[i + j] == pattern[j] then j++

if j == pattern.length() then return i

return -1

Figure 2: Naive Algorithm Pseudo Code

The KMP algorithm is more complex regarding implementation compared to the Naïve algorithm. The first part of the algorithm requires calculation of the ‘next table’ based on matching the pattern string against itself. The pseudo code for this algorithm is shown in Figure 3.

Algorithm: ComputePrefixFunction(pattern)

Input: Pattern string

Output: Next table

m = pattern.length()

next = array of size m

next[1] = 0

k = 0

for q = 2 to m

while k > 0 and pattern[k + 1] != pattern[q] then k = next[k]

if pattern[k + 1] == pattern[k] then ++k

next[q] = k

return next

Figure 3: KMP Next Table Computation (Cormen et al. 1006)

The second part of the KMP algorithm iterates along the text string using the ‘next table’ to locate a match. The pseudo code for this part of the algorithm is shown in Figure 4.

Algorithm: KMP(text, pattern)

Input: Text string, pattern string

Output: Location of first occurrence of pattern in text, else -1

n = text.length()

m = pattern.length()

next = ComputePrefixFunction(pattern)

q = 0

for i = 1 to n

while q > 0 and pattern[q + 1] != text[i] then q = next[q]

if pattern[q + 1] == text[i] then ++q

if q == m then return i – m

return -1

Figure 4: KMP Algorithm Pseudo Code (Cormen et al. 1005)

The LasVegas algorithm analyzed in this report first required the computation of fingerprints for both the pattern and text string. Figure 5 presents the pseudo code to calculate the initial finger prints.

Algorithm: CalculateFingerPrint(string, p)

Input: String string, prime number p

Output: Fingerprint calculated over the string

fp = 0

m = string.length()

for i = 0 and i < m

fp = fp + 26^(m – i) \* string[i].toInteger()

return fp % p

Figure 5: Fingerprint Pseudo Code Calculation

The LasVegas algorithm can then be implemented according to the pseudo code design shown in Figure 6. Note that both of these algorithms were formulated based on the design discussed on page 171 of Randomized Algorithms by Motwani and Raghavan.

Algorithm: LasVegas(text, pattern)

Input: Text string, pattern string

Output: Location of first occurrence of pattern in text, else -1

p = randomly chosen prime number less than N

patFP = CalculateFingerPrint(pattern, p)

txtFP = CalculateFingerPrint(text, p)

pos = 0

plen = pattern.length()

tlen = text.length()

while pos + plen <= tlen

if patFP == txtFP

if text == pattern at pos then return pos

txtFP = 26 \* (txtFP – 26^(plen – 1)\*pattern[pos]) + pattern[pos + plen]%p

pos = pos + 1

if text == pattern at pos then return pos

else return -1

Figure 6: Las Vegas Algorithm Pseudo Code

The first observation to make from the pseudocode for the Las Vegas algorithm is that 26 is used instead of 2 as an exponent base when calculating the fingerprints. In Randomized Algorithms by Motwani and Raghavan, the algorithm discussed used bit strings whereas the strings analyzed in this assignment used lowercase alphabetical character strings and therefore 26 was used because it represents all possible characters. An additional observation to make is that the pattern and text are compared to ensure a match after it is determined that both their fingerprints match.

This is done because relying on the comparison of the fingerprints alone can result in a false match. The probability of a false match occurring can be significantly reduced if the correct value of the prime number p is chosen. This is demonstrated in the analysis below.

Consider the integer summation P calculated over the pattern string, and the integer summation T[i] calculated over the text string at position i before the modulus is taken to calculate the fingerprints. The probability of a false match occurring can be written as follows:

Defining A = P – T[i], this can be rewritten as:

By the Prime Number Theorem the number of primes less than N is approximately equal to .

Recognizing that the maximum value A can take is equal to the sum of the geometric series of 26 up to the power m+1, we can write this value as follows:

Therefore, there must be less than 5(m+1) prime divisors of A and the probability of a false match is given by:

By choosing a threshold N=n2mlogn2m, the probability of a false match occurring becomes O(1/n). In the context of applying the ‘Ctrl+f’ operation on a 1000 word essay, this means there is a 0.1% of a false match occurring which for any practical case is more than suitable.

While implementing the Las Vegas matching algorithm described in Randomized Algorithms by Motwani and Raghavan, it was realized that a discrepancy exists between section 7.6 where this algorithm is described and section 7.4 where the fingerprinting function is described. In order to agree with the notation described in section 7.6, the finger print function should be implemented using Equation (2) below rather than Equation (3) provided in section 7.4.

In order to ensure an accurate runtime for each algorithm was captured, all runtimes in this report are averaged over 1000 runs. For testing, two different types of text string inputs were provided to each algorithm. First a text string consisting of the sequence “abcdefghijklmnopqrstuvwxyz” continuously repeated for 500,000 characters was used. The second text string was the sequence “abc” repeated for 500,000 characters. Additionally, 1,000,000, 1,500,000 …. 5,000,000 length versions of these same sequences were used. The strings were chosen to be this long to ensure that a reasonable amount of time was spent running each algorithm; the string matching algorithms were able to handle small text strings in too short a time to measure. The end of each sequence terminated with an identical match to the pattern string that was used thus forcing each string matching algorithm to search through the entire text string sequence to locate a match. For all sequence lengths, and both text sequence types, the pattern strings “abcc”, “abcabcc” and “abcabcabcc” were used for comparison. These three pattern strings were chosen because they are able to capture the affect of the algorithm run times for increasing lengths of the pattern string (i.e. m). Additionally, for the KMP algorithm, the repeated “abc” sequence within these pattern strings would demonstrate the efficiency of the ‘next table’. Considering these patterns along with the text string, the text sequence consisting of “abc..xyz” would demonstrate a more practical situation since a partial match would occur every 26 characters. The other text sequence formed by repeating “abc” would resemble more closely the worst case scenario, since depending on the pattern string input, a partial match could be generated as frequently as every 3 characters.

The results of running the algorithms for the “abc…xyz” text sequence can be seen in Figure 7. The key in the figure uses (1), (2) and (3) to identify the pattern string used for the corresponding graph line, where (1) is “abc”, (2) is “abcabcc” and (3) is “abcabcabcc”.

Figure 7: Experimental Algorithm Runtimes for "abc..xyz" Text String

One of the interesting observations from the above figure, and consistent with all data recorded, is how much more efficient the indexOf() algorithm is at string matching compared to the other algorithms implemented. Despite this algorithm using a variation of the Naïve algorithm which is supposed to run in O(m(n-m)) time, it was consistently able to perform string matching in a fraction of the time of the other algorithms tested. This is likely because the implementation of this algorithm is highly optimized for performance. It is interesting to acknowledge as well that the next fastest algorithm for these inputs was the Naïve algorithm. Again, this is another algorithm that has runtime of O(m(n-m)) and despite this it was able to run faster than both KMP and the Las Vegas algorithm. This can be attributed to two factors. The first is that the Naïve algorithm has a much simpler implementation than both KMP and the LasVegas algorithm, on each loop iteration where there are no matches, there are a maximum of 4 comparisons, 3 value assignments and 2 calculations. Meanwhile, KMP which has a less computationally demanding implementation compared to LasVegas, has a minimum 5 comparisons, 2 value assignments, and 1 calculation. Additionally, KMP has the added overhead of calculating the ‘next table’, although this is not a significant contributor if we observe the runtime delta for smaller text inputs. The worst case for the Naïve algorithm is when it finds a partial match between the pattern and text, followed by a single character that does not match and therefore must subsequently backtrack along the text to restart the comparisons. Considering that the text input for these tests would only match the pattern on every 26th character, there was a very small set of opportunities where the Naïve algorithm would have to backtrack and for the most part, this algorithm could iterate along the text string in O(n) time. Likewise, for KMP, since there was a very small set of comparisons where the pattern partially matched the text, this algorithm would have spent most of its computations iterating along the text string. Additionally, since there were few matches, this algorithm would have been unable to take advantage of the ‘next table’, which is where most of the runtime of this algorithm is saved. The Las Vegas algorithm was the worst performing of those tested, although it was observed to run in linear time as expected, the large number of computations needed to shift the fingerprint along the text for comparison came at a higher time cost.

Two final observations can be made from the above test results. First, all lines appear to come from the graph origin. This is evidence that for large string inputs, the affect of any preprocessing needed for algorithms like KMP and Las Vegas is negligible. The second observation is that despite increasing the length of the pattern, there is no obvious increase in the algorithm runtime. This can be attributed to the nature of the text and pattern used. Regardless of whether “abcc”, “abcabcc” or “abcabcabcc” was used as input, since the text consisted of the repeating sequence “abc..xyz”, the pattern would only ever match up to the first 3 characters. Therefore, extending the length of the pattern input had no affect.

The results of running the algorithms for a repeating “abc” text sequence can be seen in Figure 8. The key in the figure uses (1), (2) and (3) to identify the pattern string used for the corresponding line, where (1) is “abc”, (2) is “abcabcc” and (3) is “abcabcabcc”.

Figure 8: Experimental Algorithm Runtimes for "abc" Text String

Once again, the indexOf() algorithm was the fastest running string matching algorithm, however this was only true for the “abcc” pattern string. As expected, when the length of the pattern string increased, the runtime of the algorithm also increased as demonstrated by the blue line in the graph moving vertically upwards. The same can be observed for the Naïve string matching algorithm. The combination of text and pattern inputs for these tests was especially good at identifying the weak points of these algorithms because of the recurring “abc” pattern in both strings. This forced the algorithms to waste a significant amount of time comparing two strings that did not match. Unlike these algorithms though, KMP actually improved its running time with longer inputs. This is a unique result due to the combination of string inputs for this test, and should not be generalized to thinking that KMP runs faster with larger pattern string inputs. In this case since the larger pattern string input was a concatenation of the smaller pattern string and its prefix, this allowed the creation of a ‘next table’ which could effectively skip all but the last 4 characters in the pattern when performing a comparison. This allowed the algorithm to run faster than both indexOf() and the Naïve algorithm on all but the “abcc” pattern which is when the ‘next table’ had no advantage since “abcc” does not contain a prefix of itself. Again, as observed in the first sequence of test data, the LasVegas algorithm was the worst performing. One important observation should be made, and that is this algorithm had a constant runtime regardless of the size of the pattern input and number of partial matches that occurred between the pattern and text. This a testament to the robustness of using an integer fingerprint to compare two strings; regardless of how large a partial match existed between the two strings, the integer fingerprint could mask this. As a result, the Las Vegas algorithm’s runtime is almost exclusively attributed to moving the fingerprint along the text string.

Regarding algorithm improvements, there is not much in the way of modifications that can be done to the Naïve algorithm because it is inherently already simple. Originally this algorithm was implemented using Java’s String class, and therefore the charAt() method was uses for character access. Upon replacing the String class with a character array and using the standard array index method to access characters, it was observed that the runtime of the algorithm was reduced by approximately 27%. Similarly, KMP does not allow much with regarding improvement because the algorithm is already well defined and therefore only performance optimizations were made to this algorithm. The Las Vegas algorithm also underwent performance improvements, the most important of which was performing the 26^m calculation outside the loop responsible for recalculating the fingerprint on each iteration. Originally this calculation was performed inside the loop, and it was observed that the runtime of this algorithm increased with increasing pattern length. Moving this calculation outside the loop removed this affect. Additionally, this algorithm was originally described as using a randomly generated prime number p to perform the modulus when calculating the fingerprint. This was implemented using the BigInteger(int b,int c, Random r) constructor, where the resulting BigInteger calculated is prime with certainty c, and is of bit length b. Although this does not have a significant affect for larger inputs, for smaller sized inputs the random number generation accounted for as much as 60% of the algorithm runtime. In Randomized Algorithms by Motwani and Raghavan, the motivation for using a randomly generated prime number was to defend against an adversary should the bits be flipped when comparing two fingerprints. In the string matching application, this is not as relevant especially considering that the set of possible input characters for these tests was limited to lowercase alphabetical characters. This made the calculation of the string fingerprints more similar to that of a hashing problem. Therefore to reduce the overhead of generating a random prime number, a constant value of 31 was used for p. Referring back to the earlier analysis that determined the probability of a false match occurring between two fingerprints was O(1/n), this depended on picking a random prime less than the threshold N=n2mlogn2m. Assuming that the average length of a word is 8 characters, this would require that the text string exceed 23 characters in length if p=31 is used. This is a reasonable scenario.

**Conclusion**

The goal of this project was to implement various string matching algorithms and compare their performance to the Las Vegas algorithm described in Randomized Algorithms by Motwani and Raghavan. This goal was achieved by implementing various algorithms whose run time varied from O(m(n-m)) to O(m + n). The Las Vegas algorithm was hampered by requiring a significant amount of computations on each loop iteration to recalculate the fingerprint at a new position along the text. Although the runtime of the algorithm is O(m+n), it only takes O(m) time to calculate the initial finger prints based on the pattern length, after this it is a stronger function of the text string length. This was observed during experimentation by consistent runtimes regardless of the pattern string input. In order to improve this algorithm, the prime number p used to calculate the fingerprint can be hardcoded rather than generated randomly, but this only has an impact for smaller inputs. In order to improve the runtime for larger input sizes, the calculations needed to shift the fingerprint along the text need to be optimized.

KMP was the best performing algorithm when the text consisted of many partial matches with the pattern string. This was expected because KMP relies on similarities between prefixes of the pattern string and itself, so that when a mismatch is located, no backtracking occurs. This algorithm did however perform the second slowest when subjected to inputs of few partial matches because it was no longer able to rely on the ‘next table’. The design of this algorithm also makes it difficult to improve short of optimizing the operations to reduce the overhead on each loop iteration.

Perhaps the most surprising of the results observed was how fast the indexOf() method and Naïve algorithm were at string matching. Theoretically, these algorithms have O(m(n-m)) runtimes and are therefore perceived as being the slowest, but practically their implementations are so simple and computationally inexpensive that they proved to be the fastest of the string matching algorithms implemented. KMP did have faster runtimes for larger inputs when many partial matches existed between the text and pattern, but practically it is much more likely to encounter text and patterns that have fewer partial matches. It was in this case that these algorithms demonstrated superior performance. Since the implementation of the Naïve algorithm is simple and essentially a pair of nested loops, there is little that can be modified, let alone improved for this algorithm.

**References**

Cormen, Thomas H., et al. *Introduction to Algorithms*. Third ed. Cambridge, MA: MIT, 2009.

Print.

Motwani, Rajeev, and Raghavan, Prabhakar. *Randomized Algorithms*. Cambridge: Cambridge

UP, 1995. Print.

SaiKrishna, Vidya. "String Matching and Its Applications in Diversified Fields." *Internation*

*Journal of Computer Science* 1st ser. 9.1 (2012): 219-26. *IJCSI.org*. Web. 15 Mar. 2016.

<http://ijcsi.org/papers/IJCSI-9-1-1-219-226.pdf>.