DNA Sequence Matching

Maria Franchezka Sino
M.S. Computer and Information Science Student
Math and Computer Science Department
College of Science and Engineering
Southern Arkansas University
Magnolia, Arkansas 71753
Email: chezka@pm.me

String Matching is a very interesting problem in Computer Science and has various applications in different fields. There are different string matching algorithms and each has its own uses, and each of these algorithms are of different time complexities. String Matching is important in the field of bioinformatics where there is big data particularly in DNA Sequences. DNA is the blueprint of different organisms so a fast and efficient string matching algorithm is important to use for such a huge amount of data. This paper tackles three known string matching algorithms, and the Rabin-Karp algorithm is implemented and is used to a DNA sequence matching problem.

1 Introduction

String matching refers to searching for match(es) of a given pattern string in another string and there are a lot of different algorithms that efficiently processes this [1].

There are 2 types of string matching, namely, Exact and Approximate String Matching. Exact string matching searches for the exact occurrence of the pattern, whereas Approximate string matching allows for "inaccuracy" in searching. Additionally, string matching can be classified into Single Pattern and Multiple Pattern Matching. Single pattern matching searches for a single pattern on the text, while multiple pattern matching searches for multiple patterns in the text [2].

String matching has many applications. Soni and others listed the following where string matching can be applied:

1. Spell Checkers
2. Spam Filters
3. Intrusion Detection System
4. Search Engines
5. Plagiarism Detection
6. Bioinformatics/DNA Sequencing
7. Digital Forensics
8. Information Retrieval

As mentioned, string matching algorithms are used in the field of Bioinformatics, particularly in DNA Sequencing. Deoxyribonucleic acid, or DNA, is made up of four chemical bases - Adenine (A), Guanine (G), Cytosine (C) and Thymin (T). The sequence of these bases shows information on how an organism is "built" and maintained [3]. Organisms are made up of a huge number of DNA, thus the amount of DNA data to be processed grows fast so more efficient and faster methods are needed for DNA pattern matching [4].

This paper discusses the following String Matching Algorithms - Naïve, or brute force search, Boyer-Moore-Horspool algorithm and Rabin-Karp algorithm. This paper further discusses the Rabin-Karp algorithm and apply it to a DNA Pattern Matching example.

2 String Matching Algorithms

For the String Matching problems, let us denote T as the given string or body of text of length n, and P as the pattern to be search which is of length m. The string matching algorithm would determine whether P is in T, that is, for some index i,

$T[i] = P[0]$
$T[i+1] = P[1]$
...
$T[i+m-1] = P[m-1]$
or
$P = T[i...i+m-1]$

The output of these string matching algorithms can either be existence or non-existence of the pattern within a string, or the starting index of the matching pattern in the string [5].

2.1 Naïve or Brute-Force Search

The Brute Force algorithm or Naïve String Matching algorithm has no pre-processing phase. In this algorithm, the search window always shifts one position and it could return the value of which shift found a pattern match or a boolean value saying if it found the pattern or not. Figure 1 shows an example of the Brute-Force search.

Algorithm 1 Brute-Force Search

```
n ← len(T)
m ← len(P)
for i ← 0 to n − m do
    if P[i...m] = T[i+1...i+m] then
        return i
    end if
end for
return -1
```

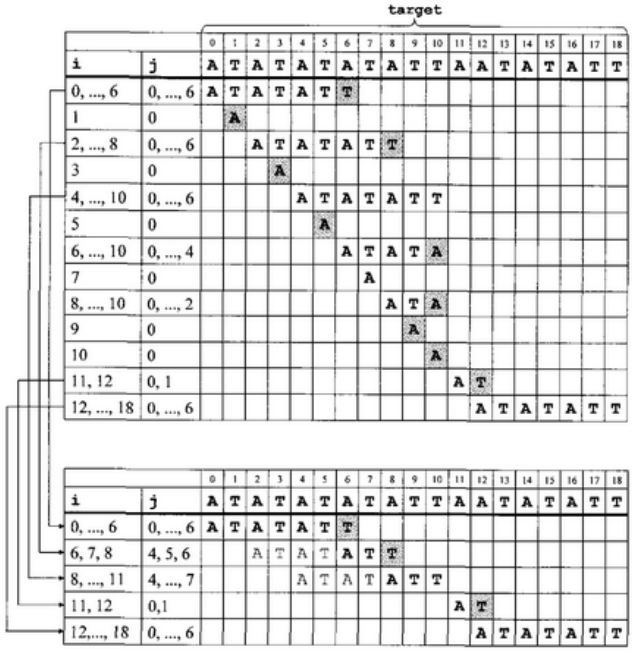

Fig. 1. Brute-Force Algorithm Example [6]

The time complexity of the Brute-Force algorithm is $O(m)$ at the best case, that is, the pattern is found from the start of the string and the algorithm checks for the pattern of length m. At the worst case, the string will be checked for the pattern $(m*(n-m+1))$ times. Thus, the worst-case running time for the Brute-Force algorithm is $O(mn)$.

2.1.1 Boyer-Moore-Horspool Algorithm

The Boyer-Moore Algorithm is an efficient string-searching algorithm which preprocesses the pattern (P) but not the string (T). This algorithm is good to use for search patterns that are significantly shorter than the string. To summarize this algorithm, the end of the pattern is being matched instead of the head, and the search skips along multiple characters unlike the previous algorithm that only goes to the immediate character next to the current initial character [7].

The Boyer-Moore-Horspool Algorithm, or simply Horspool's algorithm, is a simplification of the Boyer-Moore Algorithm [8]. The pre-processing phase for this algorithm creates a Shift Table that contains the number of characters that can be skipped when the pattern search phase encounters a mismatch.

Algorithm 2 Shift Table for Horspool's Algorithm

Input: P, alphabet of possible characters
Output: $Table$ = table of the size of the alphabet of possible characters

```
for i ← size − 1 do
    Table[i] ← m
end for
for j ← m − 2 do
    Table[P[j]] ← m − 1 − j
end for
return Table
```

Algorithm 3 Horspool's Matching

Input: P, T
Output: Index of match if found;
-1 otherwise

```
j ← 0
while j + m ≤ n do
    if P[m − 1] = T[j + m − 1 then
        i ← m − 2
        while i ≥ 0 and P[i] = T[j + i] do
            i ← i − 1
        end while
        if i = −1 then
            return j
        end if
    end if
    j ← j + Table[T[j + m − 1]]
end while
return n
```

The worst-case time complexity of this algorithm is $O(mn)$, and its average time is $O(n)$ [9].

2.1.2 Rabin-Karp algorithm

Rabin-Karp algorithm is another string search algorithm that is more efficient. Similar to the Brute-Force algorithm, the search windows moves by one, but instead of checking all the characters per move, it computes for the hash value of the substring with the same number of characters as the pattern from the current index. Only when this hash value is matches will the algorithm check for the individual character match [10]. We will discuss more about this algorithm deeper and implement this in Python to a DNA matching problem.

3 Algorithm Analysis of the Rabin-Karp algorithm

As mentioned in the previous section, the Rabin-Karp algorithm uses a hash function and also makes use of the rolling hash technique. The hash function maps an object to a value, and the rolling hash calculates a new hash value without having to recalculate the value for the whole string. Basically, this technique lets us calculate the new hash using the old hash [11].

For the Python implementation of this algorithm, we

would be using the following hash :

$$H_n = (d * H_{n-1} + c_n) \bmod q \quad (1)$$

where q is a prime number, $c_1, ..., c_n$ are the input characters and n is the length of the string. This is computed iteratively from 0 to $n-1$ and letting the initial $H_0 = 0$.

This pre-processing in the Rabin-Karp algorithm, which is computing for the hash, has time complexity of $O(m)$.

The algorithm of the search phase is as follows:

Algorithm 4 Rabin-Karp Algorithm Search Phase

Input: P, T
Output: Index of match if found;
-1 otherwise

```
1:  d ← number of possible characters
2:  q ← a prime number
3:  p ← hash(P,d,q)
4:  s ← hash(T[0 : len(P)],d,q)
5:  h ← d^(m−1) mod q
6:  for i ← 0 to n−m do
7:    if p=s then
8:      for j ← range(len(P)) do
9:        if T[i+j] ≠ P[j] then
            break
10:       end if
11:       j+ = 1
12:       if j=m then
13:         return i
14:       end if
15:     end for
16:   end if
17:   if i < n−m then
18:     s ← (d*(s−T[i]*h)+T[i+m]) mod q
19:   end if
20: end for
21: return -1
```

Overall, the best-case time complexity of Rabin-Karp is $O(m+n)$. This is because the rolling has take $O(n)$ time and if the hash codes match, m letters have to be checked to make sure that the pattern indeed found a match. The worst-case of the Rabin-Karp algorithm happens when each step results in a hash code match, but checking the m characters of the string does not result in a pattern match. The running time for this case is $O(mn)$. This depends on the hash function used in the pre-processing phase.

4 Experimental Results

Below is the code snippet for hash function given a text.

```
def calc_hash(text, d, q):
    hash_text = 0

    for i in range(len(text)):
        hash_text = (d * hash_text + \
            ord(text[i]))% q

    return hash_text
```

Below is the code for the Rabin-Karp algorithm that uses the hash function defined above:

```
def search(pattern, string):

    d = 4
    q = 11
    m = len(pattern)
    n = len(string)
    h = pow(d, m-1) % q

    # calculate hash of pattern and
    # first m characters of string
    p = calc_hash(pattern,d,q)
    s = calc_hash(string[0:m],d,q)

    for i in range(0,n-m):
        if p == s:
            for j in range(m):
                # not a match
                if string[i+j] != \
                    pattern[j]:
                        break
            j+=1
            if j==m:
                return i
        if i < n-m:
            s = (d*(s-ord(string[i]) \
                *h) + ord(string[i+m]))\
                % q
    return -1
```

The following code goes through a fasta file which contains multiple DNA sequences and finds a match for a DNA sequence pattern in each of the DNA sequence in the file. For example, sequence ID 1105513 has a DNA sequence that has 1,420 chemical bases. The search pattern is as follows:

GGTAACGGCCCACCAAGGCGACGACGGGTAGC
TGGTCTGAGAGGATGGCCAGCCACATTGGGACTG

Using the code implemented, the following result was obtained when algorithm found the pattern within the sequence

```
key was found at 1105513 (Line, column):
140412 , 195
```

The end results of parsing through the fasta file that contains 99,322 sequences shows the total execution time of the pattern search using Rabin-Karp algorithm in seconds.

```
Total Number of read: 99322
Total execution time: 57.8159...
```

5 Conclusions

String matching is an important problem in computer science, and has various applications in a wide variety of fields. As how it is nowadays, data we collect is getting bigger fast so we need fast and efficient algorithms to process this data. For example, DNA sequences of a big number of different organisms is already big data and DNA pattern matching is used a lot in the field of genomics. The different algorithms that were discussed have different computing times and even ones that are exponentially "slower" as data gets bigger can still be used in smaller string matching problems.

References

[1] Black, Paul. "string matching."*Dictionary of Algorithms and Data Structures*, 15 July 2019, www.nist.gov/dads/HTML/stringMatching.html. Accessed 3 Dec. 2019.

[2] Soni, Kapil Kumar, et al. "Importance of String Matching in Real World Problems." *International Journal Of Engineering And Computer Science*. vol. 3, iss. 6, 2014, www.researchgate.net/publication/304305210_Importance_of_String_Matching_in_Real_World_Problems, Accessed 3 Dec. 2019.

[3] "What is DNA?" *Genetics Home Reference*, 26 Nov. 2019, ghr.nlm.nih.gov/primer/basics/dna. Accessed 3 Dec. 2019.

[4] Bhukya, Raju, and Somayajulu, DVLN. "Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair." *International Journal of Computer Applications (0975 –8887)*. vol. 17, no. 8, 2011, pdfs.semanticscholar.org/eead/61406ece398aa21667a410c6bf667d152f05.pdf, Accessed 3 Dec. 2019.

[5] Gope, Ashish Prosad and Behera, Rabi Narayan. "A Novel Pattern Matching Algorithm in Genome Sequence Analysis". *International Journal of Computer Science and Information Technologies*. vol. 5, 2014.

[6] Kasahara, Masahiro and Morishita, Shinichi. *Large-Scale Genome Sequence Processing*. London, Imperial College Press, 2006.

[7] Wikipedia Contributors. "Boyer–Moore String-Search Algorithm." *Wikipedia*, Wikimedia Foundation, 3 Dec. 2019, en.wikipedia.org/wiki/Boyer–Moore_string-search_algorithm. Accessed 9 Dec. 2019.

[8] Wikipedia Contributors. "Boyer–Moore–Horspool Algorithm." *Wikipedia*, Wikimedia Foundation, 4 Dec. 2019, en.wikipedia.org/wiki/Boyer–Moore–Horspool_algorithm. Accessed 9 Dec. 2019.

[9] Saldaña, Francisco. "The Boyer-Moore-Horspool Algorithm - Nearsoft." *Nearsoft*, 13 Dec. 2018, nearsoft.com/blog/the-boyer-moore-horspool-algorithm/. Accessed 9 Dec. 2019.

[10] "Rabin-Karp Algorithm." *Tutorialspoint.Com*, 2018, www.tutorialspoint.com/Rabin-Karp-Algorithm. Accessed 9 Dec. 2019.

[11] "Rabin-Karp Algorithm." *Brilliant Math Science Wiki*, brilliant.org/wiki/rabin-karp-algorithm/. Accesses 9 Dec. 2019.