Single and Multiple Pattern String Matching Algorithm

Course Project of CS4040 Bio Informatics

Final Report

Ammanamanchi Sai Karthik, Basireddy Praveen Kumar Reddy

April 6, 2018

Abstract

Background/Objectives: Information Retrieval Systems (IRS) are playing an eminent role in different applications like World Wide Web, DNA sequence retrieval, etc. Basically, the IRS systems use the string matching algorithms.
Methods/ Statistical Analysis: Since IRS uses string matching algorithms. If string matching algorithms quality is improved then automatically information retrieval system will achieve the most relevant results. For this retrieval purpose in this paper, single pattern and multiple pattern string matching algorithms are proposed.
Findings: To assess the efficiency of the proposed single pattern and multiple pattern string matching algorithm in this paper, DNA sequences of different monkeys datasets called Pan paniscus (2.71 Gb) are considered and different tetra patterns TAGA, TCTG are searched in this data sets. From the experimental results, it is observed that proposed single pattern and multiple pattern string matching algorithms outperforms compared to other well-known string matching algorithms.
Application/Improvements: It is also observed that multiple pattern string matching algorithm reduces search time and unnecessary comparisons when compared to single pattern string matching and other existing string matching algorithms. These proposed algorithms very useful when we searching about the multiple patterns.

1 Introduction

String matching is the process of searching for the occurrence of a specified pattern in a given text. It is one of the important aspects of many applications as said in the abstract. It is divided into single pattern and multiple patterns groups.
In many information retrieval applications it is necessary to be able to locate quickly some or all occurrences of user-specified patterns of words and phrases in text. This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords and phrases in an arbitrary text string.

2 Problem Statement

To improve the running times of standard single and multiple string matching algorithms.

Single pattern matching using naive algorithm,finite automata and rabin karp algorithm run with a worst case time complexity of O(n*m) n=length of genomic data and m=length of the pattern.

Multiple pattern matching using KMP algorithm which runs in linear time for a single pattern takes a time of

O(k*(n+m)) where k=number of patterns and n=length of genomic data and m=length of the longest pattern.

3 Literature Survey

Naive algorithm for single pattern matching considers the pattern to start at every index of the genetic sequence and executes the comparision function for the length of the pattern. This continues for all the indices of the genetic sequence.

Time Complexity: O(n*m)

n=length of genomic data and m=length of the longest pattern.

Finite Machine for single pattern matching builds a finte state automata on the pattern along with the transition function. The building of the transition function is complex and takes O(m^3 * no. of characters in pattern) which is high.

Time Complexity: O(m^3 * c)

n=length of genomic data, m=length of the longest pattern, c=number of characters in text and pattern.

Rabin Karp algorithm for single pattern matching calculates the score for pattern length blocks for every index of gene sequence and compares the score modulo prime number(p) to the score of the pattern modulo the prime number(p).

Time Complexity: This algorithm has an average case of O(n+m) but might run into its worst case of O(n*m). n=length of genomic data and m=length of the longest pattern.

3.1 Proposed Method

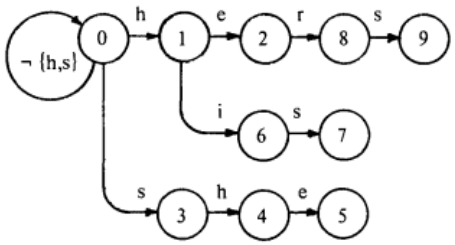
3.1.1 Single Pattern Matching Algorithm:

Knuth Morris Pratt:
Construct a longest proper suffix and prefix[lps] array for every index of pattern in O(m) time where m= length of pattern and then utilize this lps array while computing the gene sequence in such a way that on failure we do not go the start of the pattern but return to the index of previous match between the pattern and the gene sequence which is available from the lps array. This helps in restricting the time complexity to O(n+m) where n= length of the gene sequence.

3.1.2 Multiple Pattern Matching Algorithm:

Aho-Corasick Algorithm:
Consider a set of patterns p1,p2...pn and a gene sequence g.
Construct a trie for all the patternsp1,p2...pn and maintain a bit mapOUT to indicate if a pattern ends at that node.

Fig. 1. Pattern matching machine.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Now extend the trie to a finite state automata by creating the tranisition

table consisting of goto and failure transitions.
The goto transitions can be filled out by traversing the trie and the failure transitions can be filled out by doing a breadth first search on the trie.
Now the resulting finite state automata can be used for finding patterns in the gene sequence and when an accepting state is reached the corresponding patterns are obtained from the bit mapout and the index of the pattern in gene sequence is found out.
This algorithm gives a time complexity of O(n+m+z) where
n= length of the genetic sequence
m= total number of characters in all patterns
z= total number of occurences of words indexed

3.2 Pseudo Code

3.2.1 KMP algorithm:

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed,
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

```
algorithm kmp_table:
    input:
        an array of characters, W (the word to be analyzed)
        an array of integers, T (the table to be filled)
    output:
        nothing (but during operation, it populates the table)

    define variables:
        an integer, pos ← 1 (the current position we are computing in T)
        an integer, cnd ← 0 (the zero-based index in W of the next character

    let T[0] ← -1

    while pos < length(W) do
        if W[pos] = W[cnd] then
            let T[pos] ← T[cnd], pos ← pos + 1, cnd ← cnd + 1
        else
            let T[pos] ← cnd

            let cnd ← T[cnd] (to increase performance)

            while cnd >= 0 and W[pos] <> W[cnd] do
                let cnd ← T[cnd]

            let pos ← pos + 1, cnd ← cnd + 1

    let T[pos] ← cnd (only need when all word occurrences searched)
```

3.2.2 Aho-Corasick algorithm:

Algorithm 1. Pattern matching machine.
Input. A text string $x = a_1 a_2 \cdots a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function *output*, as described above.
Output. Locations at which keywords occur in x.
Method.

```
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, a_i) = fail do state ← f(state)
            state ← g(state, a_i)
            if output(state) ≠ empty then
                begin
                    print i
                    print output(state)
                end
        end
end
```

Algorithm 2. Construction of the goto function.
Input. Set of keywords $K = \{y_1, y_2, \ldots, y_k\}$.
Output. Goto function g and a partially computed output function *output*.
Method. We assume $output(s)$ is empty when state s is first created, and $g(s, a) = fail$ if a is undefined or if $g(s, a)$ has not yet been defined. The procedure $enter(y)$ inserts into the goto graph a path that spells out y.

```
begin
    newstate ← 0
    for i ← 1 until k do enter(y_i)
    for all a such that g(0, a) = fail do g(0, a) ← 0
end

procedure enter(a_1 a_2 ··· a_m):
begin
    state ← 0; j ← 1
    while g(state, a_j) ≠ fail do
        begin
            state ← g(state, a_j)
            j ← j + 1
        end
    for p ← j until m do
        begin
            newstate ← newstate + 1
            g(state, a_p) ← newstate
            state ← newstate
        end
    output(state) ← {a_1 a_2 ··· a_m}
end
```

Algorithm 3. Construction of the failure function.
Input. Goto function g and output function *output* from Algorithm 2.
Output. Failure function f and output function *output*.
Method.

```
begin
  queue ← empty
  for each a such that g(0, a) = s ≠ 0 do
    begin
      queue ← queue ∪ {s}
      f(s) ← 0
    end
  while queue ≠ empty do
    begin
      let r be the next state in queue
      queue ← queue − {r}
      for each a such that g(r, a) = s ≠ fail do
        begin
          queue ← queue ∪ {s}
          state ← f(r)
          while g(state, a) = fail do state ← f(state)
          f(s) ← g(state, a)
          output(s) ← output(s) ∪ output(f(s))
        end
    end
end
```

Algorithm 4. Construction of a deterministic finite automaton.
Input. Goto function g from Algorithm 2 and failure function f from Algorithm 3.
Output. Next move function δ.
Method.

```
begin
  queue ← empty
  for each symbol a do
    begin
      δ(0, a) ← g(0, a)
      if g(0, a) ≠ 0 then queue ← queue ∪ {g(0, a)}
    end
  while queue ≠ empty do
    begin
      let r be the next state in queue
      queue ← queue − {r}
      for each symbol a do
        if g(r, a) = s ≠ fail do
          begin
            queue ← queue ∪ {s}
            δ(r, a) ← s
          end
        else δ(r, a) ← δ(f(r), a)
    end
end
```

3.3 Results

In general a monkey chromosome contains 10 (TAGA, TCAT, GAAT, AGAT, AGAA, GATA, TATC, CTTT, TCTG and TCTA) Complex DNA Index Structures (CODIS), here these 10 CODIS are considered as search patterns.
To assess the efficiency of the proposed string matching algorithms, all the chromosomes of Pan paniscus (2.71 Gb are considered as data sets.Implementation is in C++ and the program was run on Intel quad core@ 2.2Ghz and 8GB of RAM.

Algorithm	Rabin Karp	KMP	Aho-Corasick
Pan Paniscus PanPan	88.2444	53.2859	35.4649

4 Conclusion

The experimental results have shown that the single pattern string matching algorithm(KMP algorithm) reduced the search time when compared with other string matching algorithms.
Whereas, the multiple string matching algorithms out-perform in terms of search time as compared to proposed single pattern and existing string matching algorithms.

References

[1] Knuth D, James H, Morris Jr, Pratt V *Fast pattern matching in strings*, SIAM Journal on Computing, 1977.

[2] Aho AV, Corasick MJ. *Efficient string matching: An aid to bibliographic search*, Communications of the ACM, 1975.

[3] Karp RM, Rabin MO. *Efficient randomized pattern-matching algorithms.*, CIBM Journal of Research and Development, 1987.

[4] Chinta Someswara Rao, K. Butchi Raju *Single and Multiple Pattern String Matching Algorithm.*, Indian Journal of Science and Technology, 2016.