KING ABDULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY
DIVISION OF MATHEMATICAL AND COMPUTER SCIENCES AND ENGINEERING
DESIGN AND ANALYSIS OF ALGORITHMS

Midterm Project Report

Diverse Approaches to Exact Pattern Matching

Affara, Lama
Almansour, Durrah
Al-Shedivat, Maruan
Chen, Gui
Fujii, Chisato
Rapakoulia, Trisevgeni

October 30, 2013

1 Introduction

Pattern matching remains to this day an extensively studied problem. The exponential growth of computing and data collection has increased the need for a more efficient solution to this problem. Pattern matching is applied in a wide range of disciplines such as Database queries, Text editors, two dimensional mesh, Bioinformatics, music content retrieval, MS word spell checker, digital libraries, and search engines.

One optimal algorithm that can be applied to all such applications which include different data formats has not been established. Therefore, we intend to test the performance of three different algorithms: Aho-Corasick, Suffix-tree, and Boyer-Moore algorithms. We will theoretically analyse and compare the time complexities of the above three algorithms. We will also apply them on texts, patterns, and alphabets of various types and lengths.

Pattern matching can be defined as follows [**?**]: Given a text T of length n and a pattern P of length k over some alphabet Σ, where n > k, the exact pattern matching problem consists of finding all occurrences z of the pattern P in the text T.

For example, let p = aba and t = abaababa over alphabet Σ = {a,b}. As illustrated in Figure 1.1, three occurrences of the considered pattern appear in the text, at locations 1, 4, and 6.

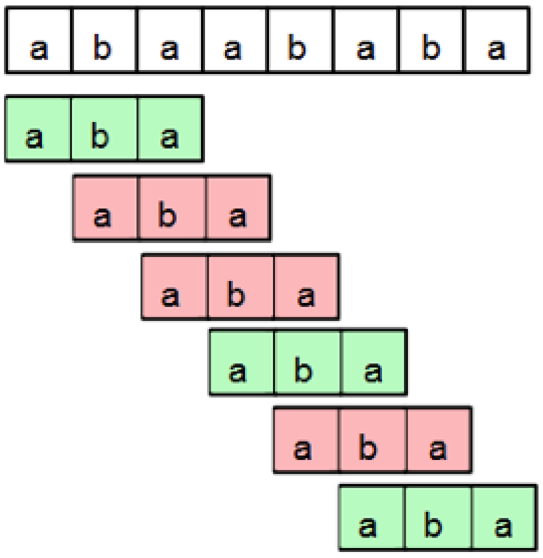
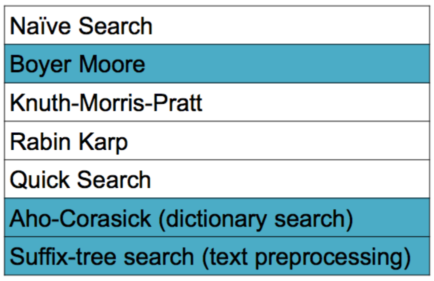


Figure 1.1: pattern matching instance

A variety of algorithms have been developed for exact pattern matching, the most famous of which are presented in Figure 1.2:

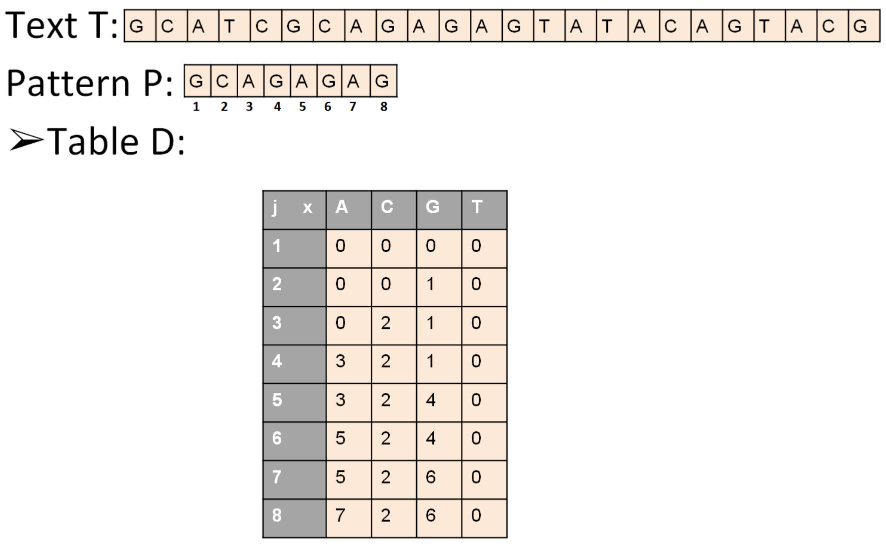


Naïve Search
Boyer Moore
Knuth-Morris-Pratt
Rabin Karp
Quick Search
Aho-Corasick (dictionary search)
Suffix-tree search (text preprocessing)

Figure 1.2: Algorithms for exact pattern matching

2 Boyer-Moore Algorithm

Boyer Moore algorithm [?] searches for all the occurrences of the pattern in the text. It is in some way similar to the naive search algorithm. Initially, it aligns the first character in P with the first character in T. The algorithm then compares characters between P and T sequentially from right to left. Once a mismatch occurs, a shift rule is applied thus moving the pattern by $s \geq 1$. The algorithm is basically divided into two stages: preprocessing and searching. There are two different preprocessing approaches in the literature: Bad Character Rule and Good Suffix Tree [?]. We decided to choose the Bad Character rule due to its simplicity and applicability to our dataset. In the following sections, we describe the two stages of the algorithm.


Text T: G C A T C G C A G A G A G T A T A C A G T A C G
Pattern P: G C A G A G A G
1 2 3 4 5 6 7 8
➢Table D:

j x	A	C	G	T
1	0	0	0	0
2	0	0	1	0
3	0	2	1	0
4	3	2	1	0
5	3	2	4	0
6	5	2	4	0
7	5	2	6	0
8	7	2	6	0

Figure 2.1: Table of preprocessing phase

2.1 PREPROCESSING STAGE

In the preprocessing stage, the algorithm makes use of the alphabet Σ and the pattern P. A two dimensional table D is constructed by processing the pattern according to the available alphabet. D is of size $k \times |\Sigma|$ where for each mismatch index in P, the position of rightmost occurrence of a character in Σ is stored. Figure **??** shows an example of the table stored by processing the pattern GCAGAGAG based on the DNA alphabet={A,C,G,T}. Starting from the last row corresponding to a mismatch occurring at position i=k in P, i=8 for this example, the algorithm scans P to find the rightmost index of the given character. In the below example, the last occurrence of A before position 8 is 7, G is 6, C is 2, and T is 0. It is important to note here that if a character does not exist in the pattern, its position in the table is always 0. The algorithm iterates from i=k to 1. If the mismatch occurs in position i-1, the algorithm updates only the value for the specific character placed in this position, A for this example, and all the other values remain the same as D[i,x]. The last occurrence of A before position 7 is 5, while G, C, and T stay the same.

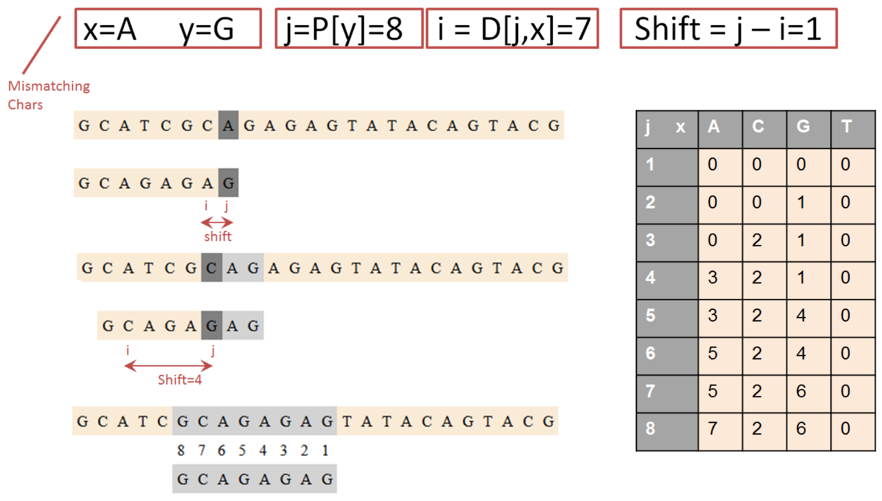
2.2 SEARCHING PHASE

In the searching phase, the algorithm shifts the pattern and sequentially matches it with the aligned text. Starting from the rightmost character in P, the algorithm checks the aligned character in T. If the pair of characters are matching, it sequentially continues the check to the next left character. If a mismatch occurs at position j in P, the algorithm shifts P according to the mismatched character x in the text. For example, if at position j, the text contains a character that is not found in P, the pattern is shifted by j. However, if the mismatched character is found in P, the pattern is shifted by j-i, where i corresponds to the rightmost occurrence of this character in P. The index i of the last occurrence is retrieved from table D (i=D[j,x] where x is the character in the text). Figure **??** shows the searching phase for the example shown in the previous section. The pseudocode of Boyer Moore algorithm is shown below.

```
Algoritm: BMMatch(T, P):
Input: Text T (n characters) and pattern P (k characters)
Output: List I of indexes of occurences of P in T

D=Preprocess(P)
l=k
j=k
z=0
repeat
    if P[j] = T[l] then
        if j = 0 then
            I[z]=l;
            z=z+1
        else {check next character}
            l = l - 1
            j = j - 1
    else { P[j] <> T[l] shift the pattern}
        x=T[l]
        i=D[j,x]
        l = l + k - j - 1 {reset l to position of P in T}
        l=l+ j - i
        j =k
until l > n
return "There is no substring of T matching P."
```

In the searching phase, the algorithm needs shift the pattern and sequentially match it with the aligned text. Starting from the rightmost character in P, the algorithm checks the aligned character in T. If the pair of characters are matching, it sequentially continues the check to the next left character. If a mismatch occurs at position j in P, the algorithm needs to shift P according to the mismatched character in the text. For example, if at position j, the text contains a character that is not found in P, the pattern should be shifted by j. However, if the mismatched character is found in P, the pattern should be shifted by j-i, where i corresponds to the rightmost occurrence of this character in P. The index i of the last occurrence is retrieved from table D and i=D[j,x] where x is the character in the text. Figure 2.2 shows the searching phase for the example shown in the previous section.



x=A y=G
j=P[y]=8
i = D[j,x]=7
Shift = j – i=1
Mismatching Chars
G C A T C G C A G A G A G T A T A C A G T A C G
G C A G A G A G
i j
shift
G C A T C G C A G A G A G T A T A C A G T A C G
G C A G A G A G
i j
Shift=4
G C A T C G C A G A G A G T A T A C A G T A C G
8 7 6 5 4 3 2 1
G C A G A G A G

j x	A	C	G	T
1	0	0	0	0
2	0	0	1	0
3	0	2	1	0
4	3	2	1	0
5	3	2	4	0
6	5	2	4	0
7	5	2	6	0
8	7	2	6	0

Figure 2.2: Searching Phase

2.3 Complexity

In this section we show the time complexity for both the preprocessing and searching phases of the algorithm. For the preprocessing stage, a table of $k \times |\Sigma|$ is stored in memory, so the space complexity is $O(k \times |\Sigma|)$. Initially, the first row is initialized to zeros. At each step, one value is updated in the table and the rest of the values are copied from the row before it. So at each step i<k, $|\Sigma|$ operations are done. Thus, the preprocessing time complexity is $O(k \times |\Sigma|)$.

For the searching phase, the text T is compared to pattern P from right to left. The worst time occurs when the shifts are only one character at a time and the algorithm would be similar to naive search with a complexity of $\mathrm{O}(k \times n)$. However, the shifts employed by this algorithm allow it to have a sublinear complexity especially in the case of large alphabet and random strings.

3 AHO-CORASICK ALGORITHM

Aho-Corasick algorithm [?] is a generalization of the Knuth-Morris-Pratt algorithm. It takes one or more patterns (a dictionary) to be searched in the text. Suppose, the total length of all patterns is k. Then, the algorithm pre-processes the patterns and constructs a Deterministic Finite Automaton (DFA) [?] in O(k) time. The obtained DFA processes a text and reports pattern occurrences in linear time too – O(n + z), where z is the number of occurrences to be found. The O(z) term is for that we assume that we can report all the occurrences in linear time. The example of the DFA is given in the figure 3.1.

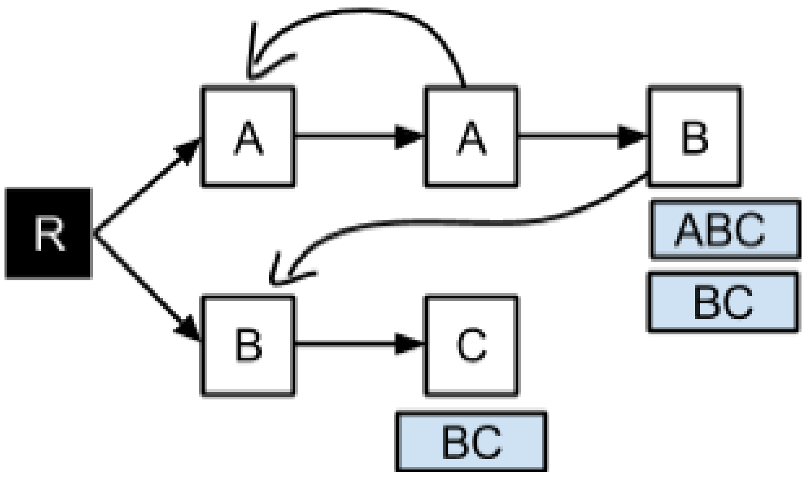


Figure 3.1: A DFA representing a set of patterns AAB, BC

3.1 CONSTRUCTION OF DFA

Each node consists of a character is represents, a flag signifying whether or not it is a dictionary node, and a set of the patterns it represents.
A dictionary node is a special kind of node which marks the end of a pattern or more, and those patterns are stored in the set of patterns it represents.

1. Forward Edge: Connecting a node representing character at position i in pattern p to the one representing the character at position i+1 in the same pattern.
2. Reverse Edge: Connecting a node to another node representing the same value in a higher level (closer to the root) provided they have the same value parents, which is insured during the building.

3.1.1 Setting Up the Nodes and Forward Edges

1. First a root node is created and set to be the current node. The first pattern is selecter and a pointer is set to point to the first character in it.
2. If the current node does not have a forward edge to a node representing the current character, then such a node is created and a forward edge connects the current node to it. The new node becomes the the current node.
3. If the current node has a forward edge to a node representing the current character, then that node becomes the current node.
4. The pointer in the pattern moves to the next character
5. If there are no more characters in the pattern, mark the node as dictionary node with the corresponding pattern, then select the next pattern in the list and make the pointer point to the first character in it. Set the current node to the root and repeat steps 2 through 5 until there are no more patterns in the list.

3.1.2 Setting Up the Reverse Edges

Expanding a node means dequeuing it and enqueuing its child nodes.

1. Start from the node and expand it into its children, put the child nodes into a queue.
2. Point the reverse edges of the children of the root back to the root.
3. Expand the node at the front of the queue.
4. For each node x of the children of the newly expanded node do the following:
   a) Set the parent of the newly expanded node as the current node.
   b) If the current node has a child node with the same value as node x, point x's reverse edge to that child, and if this child node happens to be a dictionary node for pattern p, mark node x as a dictionary of the same patter p.
   c) If this child node does not have the same value as node x, check that it is the root, and if so, point x's reverse edge to it. Otherwise go to the node pointed to by the reverse edge of the current node set it as the current node and repeat 4.b and 4.c.

3.1.3 Searching

The search algorithm outputs any patterns it find.
input: string of length n.

1. set the current node to root node
2. for every character c of the input string:

a) if the current node has a forward edge pointing to a node x with the value of c, set the current node to be node x.

b) if there is not such a forward edge, then if the current node is not the root, set the node pointed to by the reverse edge as the current node and repeat step 2.a. if the current node is the root, skip the rest of this step and step 2.c,

c) If the current node is a dictionary node, output the patterns it represents.

3.2 Theoretical Analysis

k = total length of all patterns.
n = number of characters in input string
z = pattern occurrences in the text

3.2.1 Preprocessing

Preprocessing is done in two steps: A Deterministic Finite Automaton (DFA) is built according to the patterns as described in the previous section. Setting up nodes and creating forward edges requires k time.
Traversing the DFA to add the reverse edges involves going forward k times and going up the reverse edges of a number of nodes. This number is bounded by the k, the total length of patterns since in the worst case a node at the end of each pattern will point back to the root. This gives a total of 2k for this step.
Hence, the preprocessing yields a runtime of 3*k = O(k).

3.2.2 Searching

Each character in the input text must be compared to the content of a node once, therefore a minimum of n computations is needed for comparisons, while outputting a pattern set of a dictionary node takes constant time. However, since pattern sets are output whenever they are found in the text, then the number of times the algorithm outputs a pattern, must be taken into consideration. The worst case is when patterns are found at virtually every position of the input string. For example, let the set of patterns be $\left\{a, a^2, ..., a^m\right\}$ where m is the length of the longest pattern, and let the text by a^n, such that a^i is i a's. This can also be represented by alphabet in $|\Sigma| = \{a\}$

When n = 1, A(1) outputs $\{a\}$, the result of the first node.
When n = 2, A(2) = $\{a\}$ + $\left\{a, a^2\right\}$ = A(1) + $\left\{a, a^2\right\}$.
Assume that when n = i, then A(i) = $\{a\}$+ $\left\{a, a^2\right\}$ + ... + $\left\{a, a^2, ..., a^i\right\}$.
Then when n = i+1, A(i+1) = $\{a\}$ + $\left\{a, a^2\right\}$ +... + $\left\{a, a^2, ..., a^i\right\}$ + $\left\{a, a^2, ..., a^i, a^(i+1)\right\}$ which can be rewritten as A(i+1) = A(i) + $\left\{a, a^2, ..., a^i, a^(i+1)\right\}$

This shows that in the worst case, the number of pattern occurrences in the text, denoted as z, is the dominating factor in the time complexity, yeilding O(n+z). In the worst case the term

z grows similarly to the sum of running time of applying Naive Search on each pattern in the pattern set, which will be $k \times n$.

4 Suffix-tree Pattern Matching

As the third part of our comparative study, we propose a pattern matching algorithm that relies on so called suffix tree data structure. Suffix tree is a well known data structure which is commonly used in industrial and scientific applications of pattern matching today. Being a concise representation of a text composed over a finite alphabet, suffix tree, or more general suffix automaton, is a common way to compress large amounts of textual information, and it is widely used also in databases. In our study, we will perform tests of suffix tree base pattern matching algorithm, and compare it against the aforementioned Boyer-More and Aho-Corasick algorithms, trying to get an insight on when should one chose either of these algorithms.

Below, we will introduce first *suffix trie* – an auxillary data structure – which we will enhance into suffix tree. Along this way, we will show that suffix tree construction algorithm is a linear time algorithm, and that suffix tree is a linear-memory data structure, following the Ukkonen's construction algorithm [**?**]. Suffix tree construction is the preprocessing step for the pattern matching algorithm. We also, describe how to use a suffix tree of a text to find out if a pattern matches the text, and show that procedure also takes pattern length linear time.

4.1 Suffix Trie

Being able to consider suffix tries, and following [**?**], we first introduce the notations. Let text T be a string $T = t_1 t_2 \dots t_n$ over an alphabet Σ; $T_i = t_i \dots t_n$ where $1 \le i \le n+1$ is a suffix of the text; lets also assume that $T_{n+1} = \varepsilon$, i.e. the empty suffix. Lets denote the set of all the suffixes of the text T by $\sigma(T)$. We name the following set of objects a *suffix trie* of a text T

$$STrie(T) = \left\{Q \cup \{\perp\}, root, F, g, f\right\} \tag{4.1}$$

which is a deterministic finite-state automaton (DFA) which has a tree-shaped transition graph representing the trie for $\sigma(T)$. It is augmented with so called *suffix function* f and with an auxiliary state $\perp$. In the presented notations, the set Q is the set of all the states of $STrie(T)$, which could be put into one-to-one correspondence with the substrings of T, i.e. with all such x that $T = a\ x\ v$, where a is some prefix of T, and v is some suffix of T. The initial state $root$ corresponds to the empty string ε, and the set of final states F corresponds to all the suffixes $\sigma(T)$. Finally, the transition function g is defined as $g(\bar{x}, a) = \bar{y}$ for all $\bar{x}, \bar{y} \in Q$ such that $y = xa$, where $a \in \Sigma$; $g(\perp, a) = root$ for all $a \in \Sigma$.

Now, the suffix function f is defined for each state $\bar{x} \in Q$ as follows. If $\bar{x}$ is not $root$, then $x = az$ for some $a \in \Sigma$, and we set $f(\bar{x}) = \bar{z}$. If it is $root$, $f(root) = \perp$.

To on-line construct a suffix trie reading a text T from left to right symbol by symbol, we can apply the following simple algorithm. Suppose, we have already read some prefix of the text $T^i = t_1 \dots t_i$, and have already constructed a $STrie(T^i)$. Now, we read the next symbol t_{i+1} from the text, and we need to add it to the structure and obtain $STrie(T^{i+1})$. One can notice, that all the new suffixes of T^{i+1} could be obtained by concatenation of t_{i+1} symbol to all the previous suffixes, i.e.

$$\sigma(T^{i+1}) = \sigma(T^{i})t_i \cup \varepsilon.$$

1.

4.2 Suffix Tree

The suffix tree for the string S of length n is defined as a tree such that:

- The tree has exactly n leaves numbered from 1 to n.
- Except for the root, every internal node has at least two children.
- Each edge is labeled with a non-empty substring of S.
- No two edges starting out of a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i..n]$, for i from 1 to n.

4.3 DEFINITIONS OF SOME NOTATIONS

For a text, let $T = t_1 t_2 ... t_3$ be a string over an alphabet Σ. And each string $T_i = t_i t_{i+1} ... t_n$ where $1 \le i \le n+1$ is a $suffix$ of string T. In particular, $T_{n+1} = \varepsilon$ is the $empty$ suffix. We define suffix tree $STree(T)$ of T to be a data structure that represents $STre(T)$ in space linear in the length $|T|$ of T. And we denote it as $STree(T) = (Q^{'} \cup \{\perp\}, root, g^{'}, f^{'})$.Set $Q^{'}$ consists of all branching states (states from which there are at least two transitions) and all leaves (states from which there are no transitions) of $STrie(T)$. By definition, $root$ is included in the branching states. The other states of $STrie(T)$ (the states other than $root$ and $\perp$ from which there is exactly one transition) are called implicit states as states of $STree(T)$; they are not explicitly present in $STree(T)$. Here $\perp$ is an auxiliary state.

5 Dataset

We will apply these algorithms on an available dataset of DNA sequences and natural language text, to find specific sequence motifs in the first, and words and phrases in the latter. With regards to the biological dataset, we selected the DNA sequence of human chromosome 1 in fasta format as a searching test. The DNA alphabet consists of four elements, $\Sigma = \{A, C, G, T\}$, that represent the four nitrogenous bases. We also chose three different patterns for our experimental measurements. As for the natural language data set, we will search for specific phrases of varying lengths. We will compare the performance in the two data sets to highlight the effect of the size of the alphabet and pattern length. Applying the algorithms on these datasets will show practical results on the analytical complexities that we examined and the difference in performance of the three above mentioned algorithms on the tested domains of application.

6 DISCUSSION

Each algorithm has a one time pre-processing cost. Aho Corasick algorithm has linear search cost to the length of the input text and the occurrences of the patterns in the input text.
When the alphabet in $|\Sigma|$ has a single character, the term z in Aho Corasick grows similarly to the sum of running time of applying Naive Search on each pattern in the pattern set, which will be $k \times n$, resulting in the worst case running time for Aho Corasick. However, such a small alphabet has a positive effect on Boyer-Moore, resulting in a smaller preprocessing time.

	Boyer-Moore	Aho Corasick	Suffix Tree
Pre-processing	$O(\|\Sigma\|)$	$O(k)$	O()
Searching	$O(k \times m)$	$O(n + z)$	O()