NATIONAL INSTITUTE OF TECHNOLOGY, SILCHAR

Department of Computer Science and Engineering

Project Report on,

"APPLICATION OF DETERMINISTIC FINITE AUTOMATA IN TEXTUAL PATTERN SEARCHING, ITS PROPERTIES AND EQUIVALENCE WITH NON-DETERMINISTIC FINITE AUTOMATA"

Subject : Theory of Computation (CS-204)

Under the Supervision of
Dr. Shyamapada Mukherjee

Submitted by,
Name : Arvinder Singh
Scholar ID : 18-1-5-126

July 5, 2020

ABSTRACT

"In this report, I have tried to give a concise overview on one of the applications of finite automata in the realm of textual pattern searching.

Pattern searching is an important problem in computer science. When we search for a word or formally string in notepad, browser, database or in any word processor; pattern searching algorithms are used to show the search results. I have done some in depth mathematical analysis related to time complexity of the same. Also, a C++ program demonstration of how we can build the finite automata for real life application.

I have also described some properties of DFA in sufficient detail.

Later in the project I have shown equivalence of the DFA we made with another NFA.

This project is developed using one of the open source software LaTeX and programs shown are developed using CodeBlocks IDE and compiled using g++ 17 compiler."

Contents

1 Introduction

In computer science, string-matching algorithms are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. There are many applications of string matching like in browser, database or searching *text* in word processor.

An example of string searching is when the pattern and the searched text are arrays of elements belonging to finite set of alphabet Σ. Σ may belong to human laguage alphabets, for example it may belong to English alphabets *'a'* to *'z'* or binary digits such as 0 or 1 represented as $\Sigma = \{0, 1\}$.

1.1 Different types of Searching

String searching can be of the following types:

- Finding one or more occurrences of a single word in a text or string.
- Searching occurrences of string (set of words) in non contiguous fashion inside another text or string.
- Searching a same word with alternative spellings such as 'color' and 'colour'

Here, we use Finite Automata (FA) based algorithm to search for occurrences of a particular string of length m say *'string1'* in another text of length n say *'text1'*.

1.2 Finite state automaton based search

In a Naïve approach *(discussed in complexity section)* the time complexity may be very inefficient, of the order $\mathcal{O}(m.n)$ which can be drastically improved to $\mathcal{O}(n)$ searching time, using finite automaton based searching algorithm. This, motivates us to explore this domain.

The very nature of the *finite automata*, the transition of its states for every character it encounters makes this algorithm adequate for this application.

2 Background Theory

In Theory we would basically discuss, starting from formal mathematical definition of finite automata, idea for the automaton, functioning , construction and finally giving an example to understand the transitions.

2.1 Finite automata

A **deterministic finite accepter** or **dfa** say M is a $5 - tuple$ or quintuple defined by

$$M = (Q, \Sigma, \delta, q_0, F)$$

where,
Q is a finite set of **internal states**,
Σ is a finite set of symbols called the **input alphabet**,
$\delta : Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**,
$q_0 \in Q$ is the **initial state**,
$F \subseteq Q$ is a set of **final states**.

The transition from one internal state to another are governed by the transition function δ. It can be conveniently expressed as:

$$\delta(q_i, a) = q_j \tag{1}$$

The above represents that if the current state was q_i, then upon the consumption of an input symbol $a \in \Sigma$, the **dfa** will go into state q_j.

Similarly an extended transition function $\delta^* : Q \times \Sigma^* \rightarrow Q$ can be introduced which instead of consuming a single symbol or alphabet, consumes an array of symbols called *string*. Here, the change is in the second argument of the *LHS* of eq.(1).

Formally, if $\delta(q_i, a) = q_j$ and $\delta(q_j, b) = q_k$, where $a, b \in \Sigma$ then,

$$\delta^*(q_i, ab) = q_k \tag{2}$$

which can be generalized for any arbitrary string w as,

$$\delta^*(q_i, w) = q_k \tag{3}$$

Also, the language accepted by a dfa $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

2.2 Idea for automaton

The basic idea is to build an automaton in which:

- Each character in the pattern to be matched has a state.
- Each match sends the automaton into a new state.
- If all the characters in the pattern has already been matched, the automaton enters the accepting state.
- Contrary to the above, if the automaton is not in the final state than it will return to a suitable state according to the current state and the input character. *This returned state reflects the maximum advantage we can take from the previous matching.*

2.3 Working

In Finite Automata based algorithm, we pre-process the pattern and build a 2-D array that represents a Finite Automata, in terms of implementation. Construction of the finite automata is the main tricky part of this algorithm. Once the finite automata is built, the searching is simple. In search, we simply need to start from the *start* state of the automata and the first character of the *text*. At every step, we consider next character of *text*, look for the next state in the built finite automata and move on to the next state, with respect to the transition function $\delta(q_i, a) = q_j$. If we reach a final state, then the pattern is found in the text and not, otherwise.

Given any input string w, over the alphabet Σ, a finite automata,

- starts reading from *start* state q_0, and
- reads the string w, character by character, changing state each character read.

When the finite automata is in state q_i and reads character $\sigma \in \Sigma$, it enters state $\delta(q, \sigma)$.

The finite automaton

- **accepts** the string w if it ends up in an accepting state, and
- **rejects** w if it does not end up in an accepting state.

2.4 Construction

We construct a finite automata for a pattern $P = p_1p_2...p_m$ in a text $T = t_1t_2...t_n$.

- The finite automata will have $m + 1$ states, which we number $0, 1, ..., m$
- State q_0 will be the starting state, and m will be the only accepting state.
- In general, the finite automata will be in state k if k characters of the pattern have been matched.

  In other words, the finite automata is in state k if the k most recently read characters of the *text* match the first k *pattern* characters.
- If the next *text* character t_{j+k} equals p_{k+1} (j *can be considered as offset*) , we have matched $k + 1$ characters, and the finite automata enters state $k + 1$.*i.e.,* $\delta(k, p_{k+1}) = k + 1$.
- If the next character t_{j+k} differs from p_{k+1}, then the finite automata enters a state $0, 1, 2, ..., k$, depending on how many initial characters of the pattern matched text characters ending with t_{j+k}.

  We shift the pattern, right till we obtain a match, or exhaust the pattern.

  If $\sigma = t_{j+k} \neq p_{k+1}$, then $\delta(k, \sigma)$= largest integer d such that,

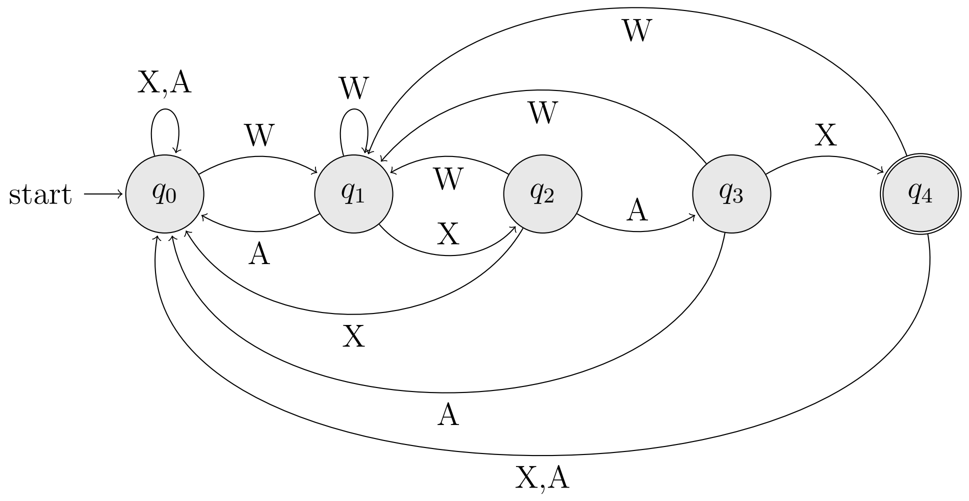
  $$t_{j+k-(d-1)}...t_{j+k-2}t_{j+k-1}t_{j+k} = p_1...p_{d-2}p_{d-1}p_d$$

  This makes it appear that $\delta(k, \sigma)$ depends on the subjects as well as the pattern, but recall that to be in state k to begin with we must have a match meaning $t_{j+i} = p_{i+1}$ for $i = 0, 1, 2, ...k - 1$.

2.5 Finite Automata example

In this subsection, I have given an example of finite automata construction for recognition of the pattern 'WXAX'.

Upon following either the transition table or the automata we see that an example string 'WAAX' is rejected, whereas 'WXAX' is accepted.



Present State	W	A	X
q_0	q_1	q_0	q_0
q_1	q_1	q_0	q_2
q_2	q_1	q_3	q_0
q_3	q_1	q_0	q_4
q_4	q_1	q_0	q_0

Table 1: Transition Table for finite automata

We can clearly observe from the construction of the above *dfa* that if the length of the string to be recognized is m then, we require a dfa with $m + 1$ states in total.

Also, we can observe intuitively the fact that as we move on scanning the text to be scanned for the *string*, for every character we encounter we change the state depending on the transition δ, giving us time complexity of the order $\mathcal{O}(n)$ *(refer to complexity analysis discussed in* ***Section*** 4*)*.

3 Algorithm

In this section I have given algorithms for answer generation, *i.e.,* generating matches which comprises mainly of the following functions: FIND-ANS(T, P), FIND-TRANSITION-TABLE$(P, trans, m)$ and NEXT-STATE$(P, m, state, c)$.

Algorithm

- FIND-ANS(T, P) // $T \to$ Text, $P \to$ Pattern

```
1: $m \leftarrow P.length$
2: $n \leftarrow T.length$
3: initialize $trans[m+1][|\Sigma|]$// 0 based indexing,
                                  quantity inside bracket
                                  represents size only
4: FIND-TRANSITION-TABLE$(P, trans, m)$
5: for $i = 0$ to $n - 1$
6:     $state = trans[state][T(i)]$
7:     if $state == m$ then
8:         print ("Match found at index = $i - (m - 1)$")
```

- FIND-TRANSITION-TABLE$(P, trans, m)$

```
1: for $state = 0$ to $m$
2:     for $c = 0$ to $|\Sigma| - 1$
3:         $trans[state][c]$=NEXT-STATE$(P, m, state, c)$
```

- NEXT-STATE$(P, m, state, c)$

```
1: if $state == m$ and $c == P(state)$ then
2:     return $state + 1$
3: for $next = state$ to 1
4:     if $P(next - 1) == c$ then
5:         for $i = 0$ to $next - 2$
6:             if $P(i) == P(state - (next - 1) + 1)$ then
7:                 break
8:         if $i == next - 1$ then
9:             return $next$
10: return 0
```

This works absolutely fine while searching for *string* in *text* separated by space.
I have also, given a code demonstration in **Section** 5 along with the test case demonstrating the same.

4 Complexity Analysis

Considering the Naïve approach, the time complexity of the algorithm is very poor. Since, this algorithm checks for patterns in the given *text* in iterations. Consider, any general character of the *text* σ, we start matching *string* from this point, if it succeeds *i.e.,* pattern is found in the *text* then it simply reports it. Otherwise we start checking from the next character in *text*. Thus this gives worst case complexity of $\mathcal{O}(m.n)$ which is indeed poor.

On the other hand,the string-matching automaton is very efficient, it examines each character in the text exactly once and reports all the valid shifts of states in *dfa* in $\mathcal{O}(n)$. This implies that search takes $\mathcal{O}(n)$ time since each character is examined once.

Observing the algorithms mentioned in the respective algorithm section, we see that the NEXT-STATE$(P, m, state, c)$ takes $\mathcal{O}(m^2)$ to compute next state for a given state and a character say σ or say to compute the transition $\delta(q_i, \sigma)$.

We compute this,*i.e.,* $\delta(q_i, \sigma)$ for every *state* and character σ in the FIND-TRANSITION-TABLE$(P, trans, m)$ function which takes $\mathcal{O}((m + 1)|\sigma|)$. However, in total we have,

$$\begin{aligned} &\mathcal{O}((m + 1)|\sigma|).\mathcal{O}(m^2) \\ &= \mathcal{O}(m^3|\Sigma|) \end{aligned}$$

Building the transition array*(finite automata)* is a bit expensive as the FIND-ANS(T, P) funciton in the algorithm section above takes $\mathcal{O}(m^3|\Sigma|)$, where m is the length of the pattern and $|\Sigma|$ is the total possible characters in the *'pattern and the text'*. However, since size m is generally small the overall complexity is not much affected by the cube factor.

If we treat number of characters *i.e.,* $|\Sigma|$ as a constant, however it can be as large as 256(*considering all the ASCII characters*), we have a time complexity of $\mathcal{O}(m^3)$, which is not bad considering the pattern to be searched to be fairly short(*which is the case in general application*), compared to the actual text to be scanned which is the main concern.

So, we can conclude that the overall complexity of the algorithm *i.e.,* construction of finite automata plus searching is $\mathcal{O}(m^3 + n)$, if $|\Sigma|$ is treated as constant.

5 Code

Below is a $C++$ code.

It consists of all three algorithms exactly as displayed in algorithms section. Further, the code mentioned below is well described with the help of comments.

Listing 1: C++ code for searching

```cpp
#include<bits/stdc++.h>
using namespace std;

int nextState(string pat, int m, int state, int c)
{
  if(state<m && c==pat[state]) return state+1;
  for(int nx = state; nx>0; nx--){
  if(pat[nx-1]==c){
   int i = 0;
   for(i=0;i<nx-1;i++)
    if(pat[i]!=pat[state-(nx-1)+i])
     break;
    if(i==nx-1) return nx;
  }
 }
return 0;
}

void find_trans(string pat, vector<vector<int>> &trans, int m)
{
for(int state = 0; state<=m; state++)
 for(int c = 0; c<256; c++)
  trans[state][c] = nextState(pat, m, state, c);
  //comparing all 256 ASCII characters

}

void find_(string text, string pat)
{
int m = pat.length() , n = text.length();
//Creating a transition table for jumps
```

```
    vector<vector<int>> trans(m+1,vector<int> (256));
    //256—>represents the no. of characters
    find_trans(pat,trans,m);
    //function call to generate answer for transition table.
    int state = 0;
    //0—>represents initial state
    for(int i=0;i<n;i++){
     state = trans[state][text[i]];
     //Reading current literal and taking current state
     if(state == m) cout<<"Match found at index= "<<i-(m-1)<<endl;
     }
    }
    //Below is the driver program
    int main()
    {
    string text, pat;
    //Where text,pat—> text and pattern to be scanned for the match
    getline (cin,text);
    getline (cin,pat);
    //User input
    find_(text,pat);
    //function to generate answer as per the algorithm
    }
```

Test Cases

Input:

Text:WXAXAXAXWXAXWXAAXAWXAWXAWXAXAWXWAXWXAXWAX...
AWXAXWXXWAXXWA

Pattern:WXAX

Output:

```
Match found at index= 0
Match found at index= 8
Match found at index= 24
Match found at index= 34
Match found at index= 42
```

Input:

Text:This is the string to be searched.

Pattern:string

Output:

```
Match found at index= 12
```

6 Equivalence with NFA

We have seen the deterministic(*dfa*) version of the finite automata, in *dfa a unique transition is defined for each state and each input symbol.* Formally, δ is a total function. This is what makes such automata deterministic. Situations where more than one transition is possible make such automata nondeterministic in nature.

However, computers are deterministic in nature and practical realization of *nfa* is impractical. Here, we therefore largely focus on theoretical aspects only.

We will see theoretically how, with *nfa* we would have realized pattern matching *i.e.,* what we just did with *dfa.* Formally,

A **nondeterministic finite accepter** or **nfa** say M is a $5-tuple$ or quintuple defined by

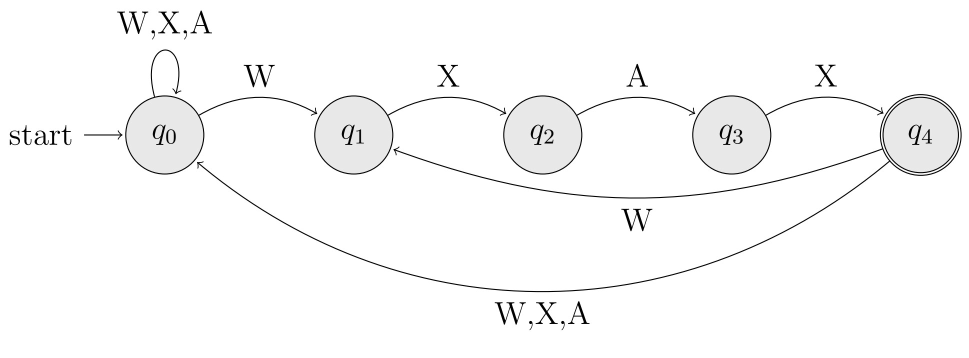
$$M = (Q, \Sigma, \delta, q_0, F)$$

where,
Q is a finite set of **internal states**,
Σ is a finite set of symbols called the **input alphabet**,
$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$ is the **transition function**,
$q_0 \in Q$ is the **initial state**,
$F \subseteq Q$ is a set of **final states**.

Here, a transition like the following is also possible,

$$\delta(q_i, \sigma) = \{q_u, q_v, ...\}, \sigma \in (\Sigma \cup \{\lambda\})$$

The *dfa* in section 2.5 can be conveniently expressed with its equivalent *nfa* as follows:



Tracing the above *nfa* we find that it accepts the string 'WXAX'

7 Further Work

The string matching algorithm is very efficient and searching only takes $\mathcal{O}(n)$ and construction process of the finite automata takes $\mathcal{O}(m^3|\Sigma|)$. We have already discussed this in details in previous sections.

The above algorithm is very efficient one but it can be further improved to work in $\mathcal{O}(m + n)$, this can be done using the idea of longest prefix suffix approach used in *KMP(Knuth–Morris–Pratt) algorithm.* This can reduce the time required for construction of finite automata to $\mathcal{O}(m|\Sigma|)$. Although this may seem to improves the time complexity by a lot of margin, its practical implications are only when the phrases to be searched for are comparable to n, which is not generally the case, at least in general use, also of the application mentioned.

Pattern matching is an important area of study in computer science, there are different kinds of searching algorithms in existence, a basic classification is as the following: **Single-pattern algorithms** which this project was based on, **algorithms using a finite set of patterns** it is a kind of dictionary-matching algorithm that locates elements of a finite set of strings *(the "dictionary")* within an input text, and lastly **algorithms using an infinite set of patterns**, here patterns cannot be enumerated finitely in this case they are usually represented using regular grammar or regular expression.

8 Conclusion

We see have seen how we will practically implement pattern matching using *dfa*, as well as simply in theory with *nfa* construction. In the theory part *i.e.,* **Section** 2 we have analyzed the theory governing the finite automata, its essential properties, construction, working and application.

In **Algorithm** *i.e.,* **Section** 3 we have developed the algorithm required for our purpose. In the following section we did *time complexity* analysis in **Section** 4 of all the functions and developed a working *C++* program and the code is well commented, described in **Section** 5 for the same along with outputs. In the next section we saw theoretical model of string-pattern matching using *nfa*. Also, in **Section** 7 we have discussed what are further developments that we can make in our algorithm, as well as discussed briefly the further classifications of the pattern matching.

In this project our scope was confined to application of finite automata with regards to searching patterns. However, there are lots of application of finite automata a few other than pattern matching are: designing of lexical analysis of a compiler, text parsing, regular expression matching, natural language processing,etc. These are only to mention a few, and for the rest, the limit is where our imagination will take us to!

References

[1] https://en.wikipedia.org/wiki/String-searching_algorithm

[2] https://www.geeksforgeeks.org/algorithms-gq/pattern-searching/

[3] https://www.eecs.wsu.edu/~ananth/CptS317/Lectures/FiniteAutomata.pdf

[4] Formal Lagnuages and Automata Theory, Peter Linz, 6th edition

[5] https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/5.pdf