The Study On The Applicability Of AHO-CORASICK Algorithm In Identifying Tests' Validity

Ed O. Omictin III [1]
iced_3@yahoo.com

Rodrigo Gante Jr. [2]
rodgantejr@gmail.com

Robby Rosa P. Villaflores [3]
robzvil@gmail.com

Ma. Bryne Catherine M. Marchan [4]
bryne_m7@yahoo.com

Rodolfo T. Noblefranca Jr. [5]
jayarnob@hotmail.com

Abstract

Aho-Corasick Algorithm (ACA) is a kind of dictionary-matching algorithm that locates elements of finite set of strings within an input text. It matches all patterns "at once", so the complexity of the algorithm is linear in the length of the patterns plus the length of the searched text plus the number of output matches. This paper discusses the applicability of Aho-Corasick algorithm in identifying test validity using the standard Guidelines in Evaluating Tests. A proposed Quiz-Zone system was developed in order to evaluate and test the applicability of the algorithm used. Quiz-Zone allows the user to create exam that will check the test's validity. It also allows the user to choose five types of exam namely: Matching Type, Multiple Choice, Essay, True or False and Short-Answer. The researchers revealed that there are some rules in identifying test validity that ACA can't be applied.

Keywords : Aho Corasick Algorithm, string-matching algorithm, test validity

1. INTRODUCTION AND THEORETICAL BACKGROUND

In creating exam teachers, are supposed to follow the guidelines in evaluating a good exam which usually they took for granted. Unfortunately, most teachers just produce an exam without thinking of the guidelines. When they make another set of exam, they seek for old test paper to get question. This study provides teacher a validated exam and hassle free task in creating exam. They do not need to think of the guidelines because the system already evaluates the questions whether it follows the guidelines. They could also retrieve validated questions from the database and creates exam from the retrieve questions and view the answers of that exam.

Aho-Corasick algorithm is a string algorithm created by Alfred V. Aho and Margaret J. Corasick. It is a kind of dictionary-matching algorithm that locates elements of finite set of strings within an input text. It matches all patterns "at once", so the complexity of the algorithm is linear in

[1] *Department of Information Technology, College of Computer Studies, Silliman University*
[2] *Department of Computer Science, College of Computer Studies, Silliman University*
[3] *Department of Computer Science, College of Computer Studies, Silliman University*
[4] *Department of Computer Science, College of Computer Studies, Silliman University*
[5] *Department of Computer Science, College of Computer Studies, Silliman University*

the length of the patterns plus the length of the searched text plus the number of output matches. Since, all matches are found; there can be a quadratic number of matches in every substring string e.g. dictionary = a, aa, aaa, aaaa and input string is aaaa. (Aho & Corasick, June 1975).

Aho-Corasick algorithm is used on multiple-string searching operations. Several algorithms on string matching are also being implemented. But Aho-Corasick is most commonly used for its automaton which can be implemented efficiently. The methods of performing useful operations on strings by exploiting the sequence, or syntactic ordering, of their symbols are considered useful in the field of text-processing, information retrieval, text editing and word-processing, linguistic analysis, and also areas in molecular biology such as genetic sequence analysis.

One of the most common problems involving strings is that of searching occurrences of a given pattern as a substring of a larger text string. According to Graham Stephen, author of String Searching Algorithms, the extension of the string-matching problem involves searching the text string for an occurrence of any set of N pattern strings, X = {z1, z2,…,zn}. These problems can be accomplished more efficiently in time by employing a pattern-matching technique, or automaton, to search for the pattern strings simultaneously like that of Aho-Corasick algorithm. (Stephen, 1994).

2. OBJECTIVES OF THE STUDY

This study aimed to prove how Aho-Corasick algorithm is used in identifying the validity of tests basing on the selected rules. Specifically, it strives to answer the following questions: a) How the principles of Aho-Corasick algorithm is used in determining the validity based on the selected rules in creating tests? and b) How to design a user interface and system features which are detailed while maintaining high understandability in the test builder application?

3. SCOPE AND LIMITATION

The main focus of the study was to investigate how Aho-Corasick algorithm is applied in identifying the validity of the test. Specifically the study focused on development of a system named Quiz-zone that enables the user to create a test and allows the user to choose five types of exam namely: Matching Type, Multiple Choice, Essay, True or False and Short-Answer. The proposed system analyzes the validity of the test in accordance to Aho-Corasick algorithm. Listed below are the selected rules in making exams depending on the type of test, which are the basis in the formulation of the algorithm, that are included in the evaluation of the applicability of Aho-Corasick algorithm:

True-False Items:

- In arranging the items avoid the regular recurrence of “true” and “false” statements.
- Score is number of correct answers.

Matching Items:

- There should be two columns; under column “A” is the stimuli which should be longer and more descriptive than the responses under column “B”.
- Use larger or smaller number of responses than premises, and permit the responses to be used more than once.

- Place the responses in alphabetical, numerical or chronological order.
- Put all the matching items on the same page.

Multiple Choice:

- Avoid using the alternative “all of the above” and use “none of the above” with extreme caution.
- Random occurrence of responses should be employed.

Short Answer Items:

- Blank should be of equal lengths.
- Place the blank near or at the end of the statement.

4. RESEARCH COMPONENT

4.1. Interpretation

Table 1 lists all the rules that are included in the implementation of the algorithm and its corresponding interpretation by the researchers. The interpretations are the basis in the conversion of the rules into pseudocodes and implementation of the algorithm.

Table 1.
List of Rules in Identifying Test Validity

Type of test	Rules	Interpretation
Multiple Choice	1.) Avoid using the alternative “All of the above” and use “None of the above” with extreme caution.	Do not use “All of the above” and “None of the above”
	2.)Random occurrence of responses should be employed	Responses should not be used consecutively, allow only three consecutive occurrences of responses.
Matching Items	1.) There should be two columns; under column ‘A’ are the stimuli which should be longer and more descriptive than the responses under column ‘B’	There are two columns – column A and column B. Under column A, the input string should be longer and more defined that in column B.
	2.) Use larger or smaller number of responses than premises, and permit the responses to be used more than once.	The number of responses should be lesser or greater than number of responses, say if the there are 5 items then the responses should be either 3 or
	3.) Place the responses in alphabetical, numerical or chronological order	The responses should be in order alphabetically or numerically.
	4.) Put all of the matching items in the same page.	Matching items should be in one page.
Short Answer	1.) Leave only one blank.	In every item, there should only have one blank (6 series of lines).
	2.) Blank should be of equal length.	The blank should have the same size or length(6 series of lines)
	3.) Place the blank near or at the end of the statement.	The blank could either be at the start or at the end of the statement.
True-False	1.) In arranging the items avoid the regular recurrence of “true” and “false” statements.	Do not use regular recurrence of “true” and “false” statement. Regular occurrence of this pattern: TFTFTFTF, FTFTFTFT and for instance in a 10 items exams then the answers are all true or all false, it would then not be accepted.
	2.) Score is the number of correct answers. (This holds true to all objective type of tests)	This will depends on the instruction given by the user. Note: The researchers will only tackle test validity.

4.2 Matching Finite States

Aho-Corasick builds a finite state of every inputted text which is considered as strings. It does the process of matching finite states. Following are sample question and diagrams to further explain how Aho-Corasick is used in the study:

Type of test: Multiple Choice

Question: What is the name of the penguin mascot of Linux?

a.) Tax b.) Tux c.) Dux d.) None of the above

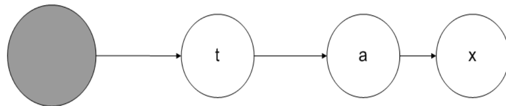

Figure 1. Finite State Diagram 1

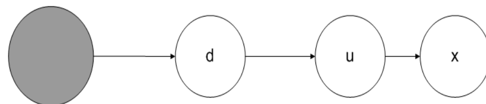

Figure 2. Finite State Diagram 2

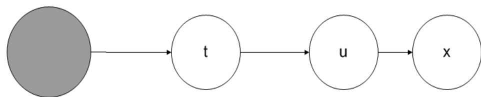

Figure 3. Finite State Diagram 3

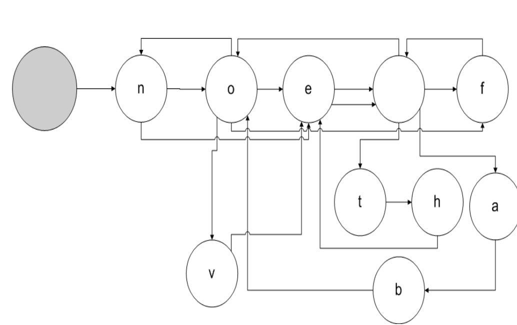

Figure 4. Finite State Diagram

There are four choices in the Multiple Choice question; the choices are converted into finite state. In figure 1, the word tax is converted into finite state wherein the shaded circle serves as the start node. Each node contains every inputted letter, meaning one letter in one node. The first letter "t" will be inputted and stored in the next node until the succeeding letters are accomplished. Figures 2 and 3 are similar to the flow of the nodes in figure 1. In fig 4, the choice given is "none of the above", the letter "n" enters in the transition node, next is another node for the letter "o", since the next letter is "n" it goes back to the previous node that has letter "n", next is letter "e" it will create another node because letter "e" does not exist yet. Another node is being created for the space between the word "none" and "of" but since the next letter is "o" and that letter already exist then it goes back again to the node that has letter "o", connecting to that will be the node for letter "f" then back again to the "space" node and will have another node "t" then another node for "h" and goes back again to the node that has letter "e" then back to space node and make another node for "a" then "b" until it goes to node "o" then "v" and finally it goes back to node "e". The node "e" is the accepting state.

After performing the finite state choices, Aho-Corasick does the next process. There is already an existing FNS which is "None of the above" and "All of the above". This FNS will be compared with the FNS choices. If one of the existing FNS will match to the FNS choices then an error occurs because it violates the rules in making Multiple Choice tests. That method is called Aho-Corasick since it matches finite states.

4.3. Algorithms Implemented

The following are the algorithms of the goto and failure functions:

Algorithm 1: Construction of the goto function.

Input: Set of keywords K={y1,y2…yk}.

Output: Goto function g and a partially computed output function.

Method: Assuming that output(s) is empty when state s is first created, and g(s,a)=fail if a is undefined or ifg(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 (yi)
For all a such that g(0,a) =fail do g(0,a) ← 0
End
```

```
Procedure enter(a1,a2,…am)
Begin
state ← 0; j ← 1
while g (state, aj ) ≠ fail do
   begin
state ← g (state, a) )
j ← j + 1
   end
for p ← j until m do
   begin
newstate ← newstate + 1
g (state, ap ) ← newstate
state ← newstate
          end
output(state) ← { a1, a2 . . . am}
end
```

Algorithm 2: Construction of the failure function.

Input: Goto function g and output function from Algorithm 1.

Output: Failure function and output function output.

Method:

```
begin
   queue ← empty
   for each a such that g(O, a) = s ≠ 0 do
begin
   queue ← queue U  {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 U {s }
  state ← f(r)
  while g(state, a) = fail do state ← f ( s t a t e )
  f(s) ← g(state, a)
  output(s) ← output(s) U output ( f ( s ) )
     end
  end
end
```

However, the failure function can be eliminated. As stated by Aho and Corasick, a deterministic set of states S and a next move function δ such that for each state S and input symbol a, δ(s,a) is a state in S. Deterministic finite automaton makes exactly one state transition on each input symbol. By using the next move function δ of an appropriate deterministic finite automaton in place of the goto function. Below is the algorithm in eliminating failure function.

```
Algorithm 2: Construction of a deterministic finite automaton
Input: Goto function g from Algorithm 1 and failure function f from Algorithm 2.
Output: Next move function δ.
Method:
begin
queue ← empty
for each symbol a do
  begin
    δ (0, a) ← g(O, a)
    if g (0, a) ≠ 0 then queue ← queue U {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
    ifg(r, a) = s ≠ fail do
        begin
      queue ← queue U {s}
          δ(r, a) ← s
        end
    else δ (r, a) ← δ (f(r), a)
    end
    end
```

4.4 Deterministic Finite Automations, Transition Tables and Algorithms

Illustrated in succeeding pages are the deterministic finite automatons, transition tables and algorithms of the rules listed in Table 1. The diagrams explains where each input x goes through the state S. To fully understand the diagram, a transition table is presented to give clarity to the flow of the input x. All the transition tables presented in the next pages has 6 columns, namely: Inputs, Current (current state), Answer, Remaining (remaining inputs), Move (next state), Matched ("+" accepted, "-" not accepted).

4.4.1 Multiple Choice

Figure 5 illustrates a diagram with 20 states. States S, 4, 8,12,16,20 are non final states, the rest are all final states. It will only accept multiple occurrences of the answers "a" "b" "c" "d" "e" more than that it will no longer accept it , meaning all the transitions goes to the non final states. For instance, inputs "bcdaaaa" when the first input is "b" transition goes to state 5. From state 5 if input is "c" it goes to state 9. In state 9, the next input is "d" it goes to state 13 and in state 13, when the next input is "a" it goes to state 1 and again if the input is "a" it goes to state 2. If in the event that in state 2 the next input is "a" then transition goes to state 3. In this case the last input will not be accepted anymore.

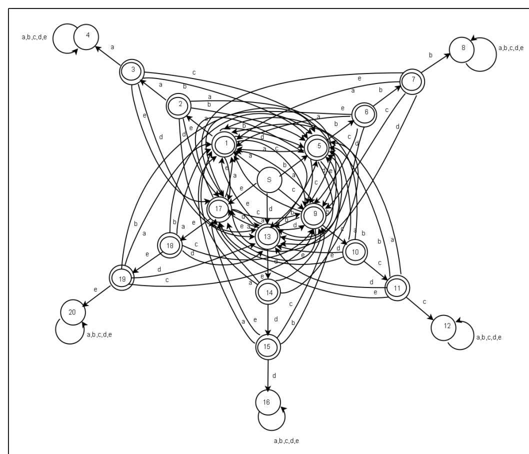

Figure 5. DFS of AC algorithm in Multiple Choice for Rule 2

Table 2 is shown to give clarification on the diagram presented above. Input = “bcdaaaa”, it will be then tokenized by characters. Starting from the start state when the input is ‘b’ a will transition move to state 1 and the remaining characters would be ‘cdaaaa’. The process continues until all the letters are accepted. Same steps will be followed in input ‘bcdc’.

Table 2
Transition Table of Multiple Choice for input bcdaaaa and bcdc

Inputs	Current	Answer	Remaining	Move	Matched
bcdaaaa	S	b	Cdaaaa	5	+
	5	c	Daaaa	9	+
	9	d	Aaaa	17	+
	17	a	Aaa	1	+
	1	a	Aa	2	+
	2	a	A	3	+
	3	a	None	4	+
b c d c	S	b	Cdc	5	+
	5	c	Dc	9	+
	9	d	C	13	+
	13	c	None	9	+

In the pseudocode state is initialized to zero, while input is not equal to empty then it continues to a switch statement that allows the user to choose letters from a to e. In case a if state is equal to zero then it goes to state 1 and so on if it is state 5,6,7,9,10,11,13,14,15,17,18 and 19 it stays at state 1. Same thing happens to case b,c,d, and e. It only matters on the inputs based on Figure 2.1.

```
Psuedocode for Multiple choice rule # 2:
state=0;
while(input≠empty){
switch(input)
case 'a': if(state==0) state=1;
if(state==1) state=2;
if(state==2) state=3;
if(state==3) state=4;
if(state==5 or 6 or 7 or 9 or 10 or 11 or
13 or 14 or 15 or 17 or 18 or 19) state=1;
case 'b': if(state==0) state=5;
if(state==5) state=6;
if(state==6) state=7;
if(state==7) state=8;
if(state==1 or 2 or 3 or 9 or 10 or 11 or
13 or 14 or 15 or 17 or 18 or 19) state=5;
case 'c': if(state==0) state=9;
if(state==9) state=10;
if(state==10) state=11;
if(state==11) state=12;
if(state==1 or 2 or 3 or 5 or 6 or 7 or 13
or 14 or 15 or 17 or 18 or 19) state=9;
case 'd': if(state==0) state=13;
if(state==13) state=14;
if(state==14) state=15;
if(state==15) state=16;
if(state==1 or 2 or 3 or 5 or 6 or 7 or 9
or 10 or 11 or 17 or 18 or 19) state=9;
case 'e': if(state==0) state=17;
if(state==17) state=18;
if(state==18) state=19;
if(state==19) state=20;
if(state==1 or 2 or 3 or 5 or 6 or 7 or 9
or 10 or 11 or 13 or 14 or 15) state=9;
```

Figure 6 is a diagram for rule number one in multiple choice wherein every occurrence of “none of the above” “all of the above” statements are not allowed. In this diagram it has 20 states. States 1-17 are all final states and the rest are non final state. From state 0 if input is “xl_ofthbv” transition goes to state 1 else input is “a” it goes to state 2 else input is “n” it goes to state 18. From state 1 if input is “xl_ofthbv” transition goes to state 2. State 1 and state 2 has a loop its “xl_ofthbv”

for state 1 and “a” for state 2. From state 18 if input is “o” transition goes to state 19, from state 19 it goes to state 20 if input is “n”, in state 20 if input is “e” transition goes to state 4. Going back to state 2 if input is “l” it goes to state 3 else input is “x_of vltenhbrv” it goes to state 1. From state 3 if input is “l” then it goes to state 4 else input is “x_of vltenhbrv” it goes to state 1. From state 4 if input is “_” it goes to state 5, else input is “xofvltenhbrv” it goes to state 1. In state 5 if input is “o” it goes to state 6 else input is “x_fvltenhbrv” it goes to state 1. State 6, if input is “f” it goes to state 7 else input is “x_ovltenhbrv” it goes to state 1. State 7, if input is “_” it goes to state 8 else input is “x_ofvlenhbrv” it goes to state 1. In state 8 if input is “t” then it goes to state 9 else it goes to state 1. State 9, input is “h” it goes to state 10 else it goes to state 1. In state 10 if input is “e” it goes to state 11 else it goes to state 1. Looking at state 11, if input is “_” it goes to state 12 else it goes to state 1. State 12, input is “a” it goes to state 13 else it goes to state 1. State 13, input is “b” it goes to state 14 else it goes to state 1. In state 14, if input is “o” transition goes to state “15” else it goes to state 1. In state 15, if input is “v” it goes to state 16 else it goes to state 1. From state 16 if input is “e” it goes to state 17 where in state 17 has a loop “xlf_othbrv”. States 3-17 passes a transition “a” to state 2.

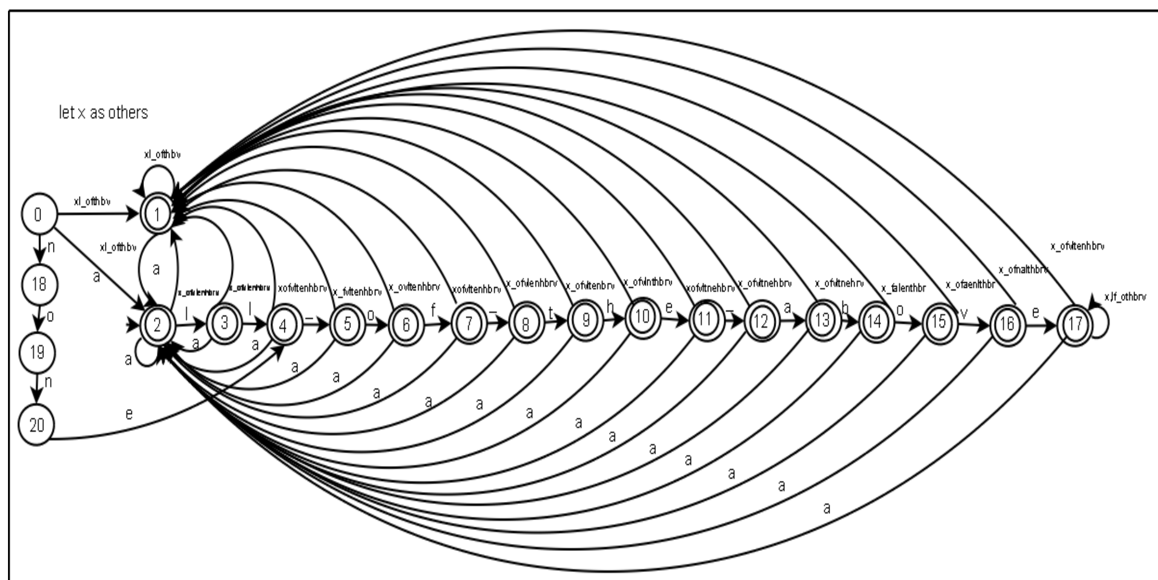

Figure 6. DFS of AC algorithm of Multiple Choice for Rule # 1

Table 3 shows the input string “world” and “all of the above” is accepted or rejected. In the table, it is clearly shown where every input character goes starting from the start state.

When the input string is “world”, the “world” is tokenized by letters. Starting from the start state when the input character is ‘w’ it goes to state 1 and the remaining letters would be “orld” the process continues until all the letters is accepted and is marked by a plus sign. On the other hand, if the input string is “All of the above” denoted by A, again starting at the start state it goes to state 2 and is rejected- represented by minus sign.

Table 3.
Transition Table of Multiple Choice for input string “world” and“All of the above”

Inputs	Current	Answer	Remaining	Move	Matched
w	S	w	orld	1	+
o	1	o	rld	1	+
r	1	r	ld	1	+
l	1	l	d	1	+
d	1	d	None	1	+
A	S	A	None	2	-

Shown on the next succeeding sections are pseudocodes when the input string is “all of the above” and “none of above”. Let x as the input string and S as state. When state is equal to zero then it continues to the conditional statement when input is not equal to empty then it loops starting from the 1 until it reaches the desired number of items. Every item is evaluated if the list of choices has “All of the above” or “None of the above” and if it detects that there are choices inputted as mentioned then it displays an error message containing that it violates the rules of validity.

As shown in the pseudocodes x is initialized as input and s as state. When state is equal to zero continues to the if statement if input is not equal to empty it loops from 1 until to the nth items while inside the loop it evaluates if the inputted string is A or B which is the “All of the above” and “None of the above” it gives an error message otherwise it prints the inputted string x. After it is evaluated it goes out from the loop statement and evaluates if the state S is equal to 2 then it is not accepted else there is no input.

Pseudocode in Multiple Choice for rule # 1:

```
Input (none, all of the above)
Let x as input and s as state
State = 0
if input ≠empty {
for i =1 until n
do
if (x=A or x=B)
print <not accepted>
else
print(x)
}
}
If (s==2)
Print <not accepted>
Else
Print <no input>
```

4.4.2 Matching Type

In matching type, the rules 1 and 2 can’t be solved by Aho-Corasick Algorithm. Researchers tried to look for another algorithm to solve the problem but they didn’t found any. Probably, there is but it tales lot of time and ample study to look for another algorithm to solve the problem. Researchers were given a limited time to work for the project, so instead of consuming all the time in looking for another algorithm, they decided to make their own solution for the problem.

In this rule, the researchers used structure for colA it initialized that there are 50 capacities that can be stored same as with colB while colA[t] is equal to empty t as the index number for colA it increments the value of t as the user inputs on the other hand in colB[n] is not equal to empty, it increments n, n as the index of colB. It continues to evaluate if the index t is less than n.

Rule 1.) There should be two columns; under column “A” are the stimuli which should be longer and more descriptive than the responses under column “B”.

Pseudo codes:

```
Struct String {
ColA [50];
ColB [50];
};
While(ColA[t]=empty){
T++;
}
While(ColB[n]≠ empty){
N++;
}
If (t<n) <error>;
```

In this rule the researcher’s initialized iteminput is equal to 10, ansinput is equal to 15, x and y is equal to 0. Then x as iteminput minus 2 and y as iteminput+2 if ansinput is less than x or ansinput is greater than y then it displays an error message else it is accepted.

Rule 2.) Use a larger or smaller number of responses than premises, and then permit responses to be used more than once.

Pseudo codes:

```
Ex. items inputted: 10
responses inputted: 15
iteminput=10;
ansinput=15;
x=0;
y=0;
x=iteminput-2;
y=iteminput+2;
if(ansinput<x or ansinput>y) <error>
else
<accepted>
```

4.4.3 True or False

Figure 7 demonstrates the transitions following the rules with only 5 successive “true” or “false” answer is allowed. Answering in more than the allowable answer would mean a violation of the rule. Likewise, the pattern TFTFTF or FTFTFT, are allowed for the answers. From “S” which is the start state, if the input is “T” it goes to state 1. In state1 if the input is “T” it goes to state 9, from state 9 input is “T” it goes to state 10, from state 10 if input is “T” it goes to state 11, from state 11, if input is “T” it goes to state 12. States 1, 9,10,11,12 are all final states. In state 12, if input is “T” it goes to state 13 which is a non final state. State 13 has a loop of T or F. States 1, 9,10,11,12, if all the input of these states is “F” their transition will go to state 5. Going back to the S state if the input is “F” it goes to state 5, from state 5 if input is “F” it goes to state 14, if the input is “F” again in state 14, then goes to state 15, then if input is “F” it goes to state 16, from state 16 if input is “F” it goes to state 17. States 5,14,15,16,17 are all final states meaning all the inputs of these states are accepted and if the input of these states are “T” all there transitions goes to state 1. Going back to state 17, if input is “F” it goes to state 18 that this state has a loop T of F. From state 1, if input is “F” transition goes to state 2. From state 2 “T” it goes to state 3 else input is “F” transition goes to state 14. From state 3 if input is “F” it goes to state 4 else it goes to state 9. From state 4 if input is “T” it goes to state 19 that performs a loop T or F else it goes to state 14. From state 5 if input is “T” it goes to state 6, from state

6 if input is “F” it goes to state 7 else it goes to state 9. From state 7 if input is “T” it goes to state 8 else it goes to state 14. From state 8 if the input is “F” it goes to state 20 that performs a loop T or F else it goes to state 1.

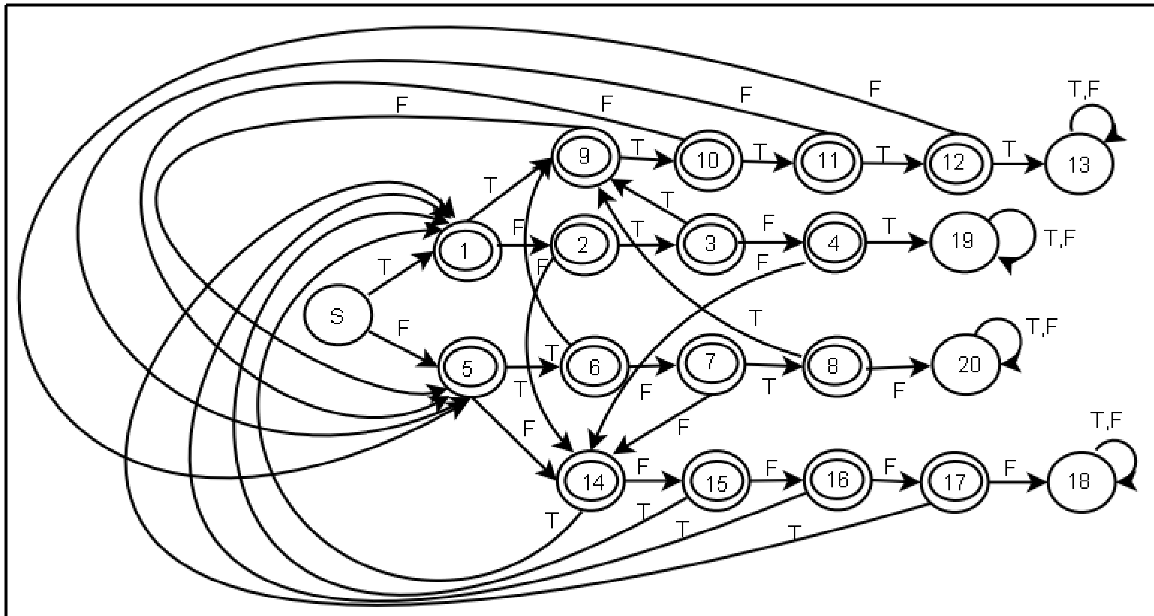

Figure 7. DFS of AC algorithm in True/False for Rule 1

4.4.4 Short Answer

The figure 8 above has 4 states. From start state if the input is 6 underscore it will go to state 1 and it will be accepted since state 1 is a final state. From state 1 if input is another underscore the transition will go to state 3 which is a non final state. In state 1 if the input is “x” (others) it will go to state 2 a non final state. Going back to the start state if the input is “x”(others) it will go to state 2. In state 2 there is a transition of “_,x” that goes to state 2 itself, called loop transition.

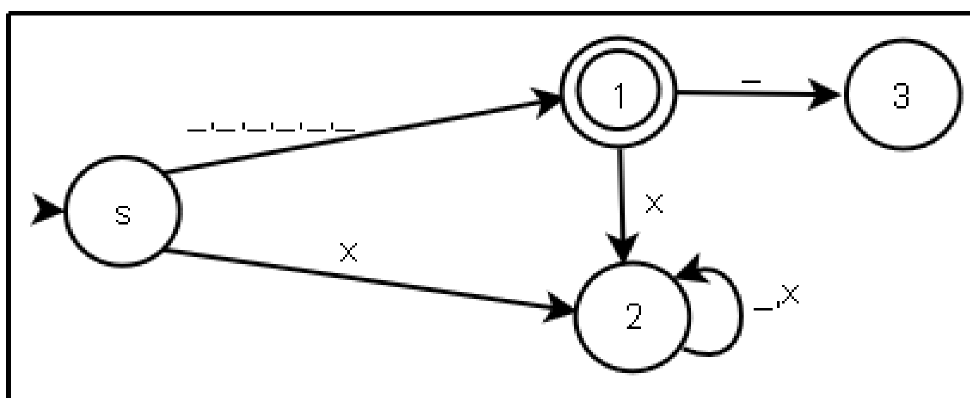

Figure 8. DFS of AC algorithm in Short Answer rule # 2

The transition on table 4 is shown for better understanding. Form start state if input is ‘_’ transition move to state 1 and the remaining input is ‘_____’ (6 underscore). The same process is followed on the next inputs.

Table 4
Transition Table of Short answer for input ______

	Current	Inputs	Remaining	Move	Matched
______	S	_	_____	1	+
_____	1	_	____	1	+
____	1	_	___	1	+
___	1	_	__	1	+
__	1	_	_	1	+
_	1	_	None	1	+

Presented in figure 9 is a diagram with 21 states. Only state 7 and 15 are the final states, the rest are all none final. State 0, if input is "_" it goes to state 1 else input is "s" it goes to state 8 which performs a loop "s". From state 1, if input is "_" it goes to state 2. In state 2 if input is "_" it goes to state 3, in state 3 if the input is "_" it goes to state 4. In state 4 if input is "_" it goes to state 5. From state 5, if the input is "_" it goes to state 6. From state 6, if input is "s" transition goes to state 7 which perform a loop "s", else transition goes to state 9. From state 7 if input is "_" it goes to state 19. States 6 and 9 both performs a loop "s,_". States 1, 2,3,4,5, if input in this state is "s" it goes to state 17 wherein it also performs a loop "s,_". Going back to state 8 if input is "_" it goes to state 10. In state 10 if input is "_" it goes to state 11, from state 11 if input is "_" it goes to state 12, from state 12 if input is "_" it goes to state 13. In state 13 if input is "_" it goes to state 14, from state 14 if input is "_" it goes to state 15. In state 15 is input is "_" transitions goes to state 18 which performs a loop "_". States 15 and 18, if their inputs are "s" both transitions goes to state 16. State 16 has a loop "s" and if input is "_" it goes to state 20 which has a loop that performs the input "s, _". States 10, 11,12,13,14, if all the inputs of these states is "s" all the transitions goes to state 17.

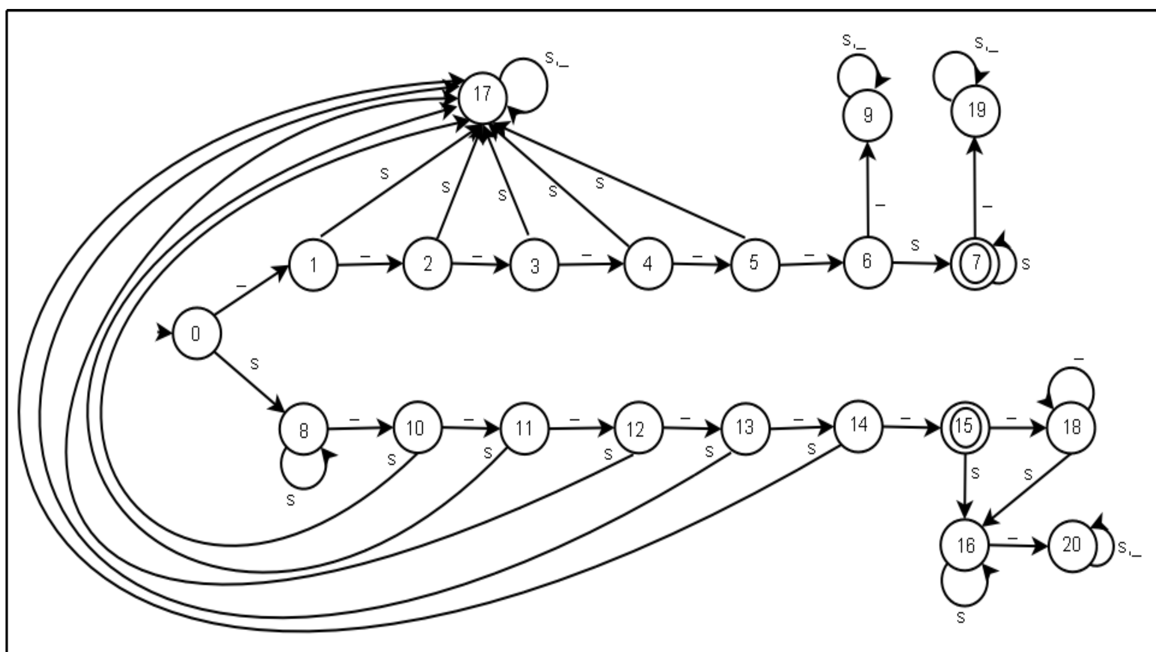

Figure 9. DFS of AC algorithm of Short Answer for Rule#1 and 3

There are number of table presented bellow with different sample inputs. All the process of the following transition tables are the same from other transition tables presented above.

Sample Inputs: _S_

Table 5.

Transition Table of Short Answer for input _S_

	Current	**Inputs**	**Remaining**	**Move**	**Matched**
_S_	S	_	S_	1	+
S_	1	S	___	3	+
_	3	_	None	5	-

Sample Inputs: SSS

Table 6.

Transition Table of Short Answer for input SSS

	Current	Inputs	Remaining	Move	Matched
SSS	S	_	SS	1	+
SS	1	S	S	3	+
S	3	_	None	5	-

Sample Inputs: _SS

Table 7.

Transition Table of Short Answer for input _SS

	Current	Inputs	Remaining	Move	Matched
_ S S	S	_	S S	1	+
S S	1	S	S	3	+
S	3	S	N o n e	3	+

As shown in the tables 5-7, c and state is initialized to zero and x as others. While input is not equal to empty it goes to switch statement. When the input is an underscore it is in state 1 if the input is others it goes to state 2 then it increments the value of c. If c is not equal to 6 or state is equal to 2 then it displays an error message otherwise it is accepted.

4.5 Design Architecture

Figure 10 below illustrates the flow of the system, "Quiz-zone". As you can see, the data (question) formulated by the user is passed on to the tokenizer that splits data into a set of characters. The tokenized data is again passed on to the pattern matching machine. The pattern matching machine is composed of the DFA, goto and output function. The pattern matching machine is employed in the identifying test validity. The construction of the DFA in the pattern matching machine is a built-in DFA. Through the DFA, the inputted data is evaluated using the goto function and if it fits to the rules of validity then it will display the data entered by the user otherwise it displays an error message that the data entered is invalid. User will need to comply the error first before he/she can continue the process.

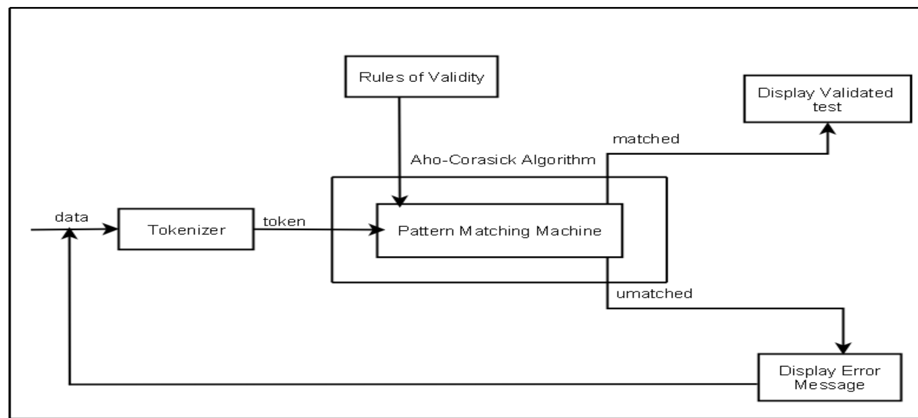

Figure 10. Design Architecture

5. TEST RESULTS

Several testing were made to prove how accurate the applicability of Aho-Corasick algorithm in identifying test validity. The testers created exam, with different number of questions and performed a number of test repetitions to it. They inserted questions to an exam from the Question bank manually, inserted questions in random, a combination of manual and random as well as the combination of random to manual insertion of questions. As a result, they put "P" under the column of "Applicability of Aho-Corasick Algorithm" meaning as they do the testing, the testers found out that the rules of validity, only those applicable, were 100 % solved by the algorithm. Sample Test Result under True or False is shown in table 8.

Table 8.
Test Result of True or False

Test number	Number of test performed	Number of questions	Applicability of Aho-Corasick Algorithm
1	2	5	P
2	4	6	P
3	1	7	P
4	2	12	P
5	3	8	P
6	2	9	P
7	2	6	P
8	1	10	P
9	1	8	P
10	3	13	P
11	2	20	P
12	1	11	P
13	2	10	P
14	1	12	P
15	1	16	P
16	1	13	P
17	2	14	P
18	3	11	P

Table 8.
Test Result of True or False (cont')

19	2	15	P
20	2	8	P
21	2	16	P
22	1	17	P
23	2	18	P
24	2	19	P
25	2	20	P
26	2	22	P
27	2	23	P
28	2	25	P
29	1	24	P
30	2	25	P

6. CONCLUSION

In terms of applying Aho-Corasick algorithm, researchers found out that there are some rules in which the algorithm can't be applied. Out of 11 rules identified, only 7 rules are applicable by ACA. Researchers made their own solution for those rules that can't be solve by the chosen algorithm.

Table 9.
Applicability of Aho-Corasick Algorithm

Type of test	Rules in Identifying Test Validity	Applicability of AC Algorithm
Multiple Choice	1.) Avoid Using the Alternative "All of the Above" and use "None of the "with extreme caution	Applicable
	2.) Random occurrence of responses should be employed	Applicable
Matching Items	1.) There should be two columns; under column "A" are the stimuli which should be longer and more descriptive than the responses under column "B"	Not applicable
	2.) Use a larger or smaller number of responses than premises, and permit the responses to be used more than once.	Not applicable
	3.) Place the responses in alphabetical, numerical, or chronological order.	Applicable
	4.) Put all matching items in the same page.	Not applicable
Short Answer	1.) Leave only one blank.	Applicable
	2.) Blank should be of equal lengths.	Applicable
	3.) Place the blank near or at the end of the statement.	Applicable
True or False	1.) In arranging the items avoid the regular recurrence of "True" and "False" statements.	Applicable
	2.) Score is number of correct answers.	Not applicable

Table 9 shows the applicability of Aho-Corasick algorithm in identifying test validity based on the type of tests. There are some rules that are mark "not applicable", since it can't be solve by Aho-Corasick Algorithm. As summarize in Table 9, there are three rules in matching items, and one in true or false types of test that Aho-Corasick Algorithm is not applicable.

7. RECOMMENDATION

Based on the result of this study, not all rules were solved by the algorithm used, it's better to have not only Aho-Corasick algorithm but also to use or searched for another algorithm that is applicable to solve all the rules in identifying test validity. It is further recommended for future study the inclusion of other rules in identifying tests's validity using ACA or any other string-marching algorithms to have an integrative and psychologically validated test questionnaire.

References

Aho,A. & Corasick, M. (1975, June).Efficient String Matching: An Aid to Bibliographic Search. USA: *Communications of the ACM.* p. **18.**

Boss, H. & Huang, K. (2004). Network Intrusion Detection System. *Technical Report Journal*, p. 2.

Knipp, C. T.(2006, September). Physics Education Research's Study on the Reliability and Validity of Exams. *American Physics Journal*, p. 10.

Stephen, G. (1994). String Searching Algorithms. USA:World Scientific Publishing Co.

FScreation,Inc. (2003). ExamView. Retrieved from http://www.examview.com.

Kojm, T. (2002, May 8). Clam Antivirus. Retrieved from http://www.clamav.net/.

Microsoft ® Encarta ® 2006. © 1993-2005 Microsoft Corporation.

Murherjee,T. (2005, April 6). Multiple-pattern matching in LZW Compressed Files using Aho-Corasick Algorithm. Retrieved from .http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/9633/30443/01402239.pdf?arnumber=1402239).

Stahlberg, M.(2008, April 9). Antivirus Engine of F-Secure Corporation.http://www.fsecure.com/weblog/archives/00001421.html.