**DOKUZ EYLUL UNIVERSITY**

**ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**CME3203 THEORY OF COMPUTATION**

Finite Automata to RE Converter with Using GNFA

**By**

**Mehmet Hamdi USLU**

**Şeniz AKBULUT**

**Lecturers**

**Dr.Meltem YILDIRIM EKİCİ**

**IZMIR**

**27.01.2021**

**ABSTRACT**

In this report, the fundamentals and general features of a solution that considered as a homework within the scope of this course, which enables the conversion of a finite automata to a regular expression, with the help of a generalized nondeterministic finite automata will be discussed.

# CHAPTER ONE

ıntroductıon

In this solution , we show that any DFA can be converted into a regular expression. Our construction would work by allowing regular expressions to be written on the edges of the DFA, and then showing how one can remove states from this generalized automata (getting a new equivalent automata with the fewer states). In the end of this state removal process, we will remain with a generalized automata with a single initial state and a single accepting state, and it would be then easy to convert it into a single regular expression.

# CHAPTER TWO

DEFINITIONS

## FINITE AUTOMATION(FA)

Finite Automata(FA) is the simplest machine to recognize patterns.The finite automata or finite state machine is an abstract machine which have five elements or tuple. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically it is an abstract model of digital computer. FA is characterized into two types.(DFA and NFA)

Finite automations have the features listed below:

* Input
* Output
* States of automata
* State relation
* Output relation

A Finite Automata consists of the following :

* Q : Finite set of states.
* Σ : set of Input Symbols.
* q : Initial state.
* F : set of Final States.
* δ : Transition Function.

Formal specification of machine is

{ Q, Σ, q, F, δ }.

## DETERMINISTIC FINITE AUTOMATION(DFA)

A deterministic finite automaton (DFA)—also known as deterministic finite acceptor (DFA), deterministic finite-state machine (DFSM), or deterministic finite-state automaton (DFSA)—is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string.

A DFA is defined as an abstract mathematical concept, but is often implemented in hardware and software for solving various specific problems such as lexical analysis and pattern matching. For example, a DFA can model software that decides whether or not online user input such as email addresses are syntactically valid.

Briefly,In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.

One important thing to note is, there can be many possible DFAs for a pattern. A DFA with minimum number of states is generally preferred.

A deterministic finite automata is set of five tuples and represented as,

M={Q, Σ , δ , qo ,F}

Where,

Q: A non empty finite set of states present in the finite control(qo, q1, q2, …).  
Σ: A non empty finite set of input symbols.  
δ: It is a transition function that takes two arguments, a state and an input symbol, it returns a single state.  
qo: It is starting state, one of the state in Q.  
F: It is non-empty set of final states/ accepting states from the set belonging to Q.

## NONDETERMINISTIC FINITE AUTOMATION(NFA)

NFA refers to Nondeterministic Finite Automaton. A Finite Automata(FA) is said to be non deterministic, if there is more than one possible transition from one state on the same input symbol.

NFA is similar to DFA except following additional features:

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.

2. Ability to transmit to any number of states for a particular input.

However, these above features don’t add any power to NFA. If we compare both in terms of power, both are equivalent.

A non deterministic finite automata is also set of five tuples and represented as,

M={Q, Σ , δ , qo ,F}

Where,  
Q: A set of non empty finite states.  
Σ: A set of non empty finite input symbols.  
δ: It is a transition function that takes a state from Q and an input symbol from and returns a subset of Q.  
qo: Initial state of NFA and member of Q.  
F: A non-empty set of final states and member of Q.

One important thing to note is, in NFA, if any path for an input string leads to a final state, then the input string accepted.

Since all the tuples in DFA and NFA are the same except for one of the tuples, which is Transition Function (δ)

In case of DFA

δ : Q X Σ --> Q

In case of NFA

δ : Q X Σ --> 2Q

So mathematically, we can conclude that every DFA is NFA but not vice-versa. Yet there is a way to convert an NFA to DFA, so there exists an equivalent DFA for every NFA.

## GENERALIZED NONDETERMINISTIC FINITE AUTOMATION(GNFA)

Generalized nondeterministic finite automaton (GNFA), also known as an expression automaton or a generalized nondeterministic finite state machine, is a variation of a nondeterministic finite automaton (NFA) where each transition is labeled with any regular expression. The GNFA reads blocks of symbols from the input which constitute a string as defined by the regular expression on the transition. There are several differences between a standard finite state machine and a generalized nondeterministic finite state machine. A GNFA must have only one start state and one accept state, and these cannot be the same state, whereas a NFA or DFA both may have several accept states, and the start state can be an accept state. A GNFA must have only one transition between any two states, whereas a NFA or DFA both allow for numerous transitions between states. In a GNFA, a state has a single transition to every state in the machine, although often it is a convention to ignore the transitions that are labelled with the empty set when drawing generalized nondeterministic finite state machines.

A GNFA can be defined as a 5-tuple, (S, Σ, T, s, a), consisting of

* a finite set of states (S);
* a finite set called the alphabet (Σ);
* a transition function (T : (S ∖ {a}) × (S ∖ {s}) → R);
* a start state (s ∈ S);
* an accept state (a ∈ S);

where R is the collection of all regular expressions over the alphabet Σ.

The transition function takes as its argument a pair of two states and outputs a regular expression (the label of the transition). This differs from other finite state machines, which take as input a single state and an input from the alphabet (or the empty string in the case of nondeterministic finite state machines) and outputs the next state (or the set of possible states in the case of nondeterministic finite state machines).

A DFA or NFA can easily be converted into a GNFA and then the GNFA can be easily converted into a regular expression by repeatedly collapsing parts of it to single edges until S = {s, a}. Similarly, GNFAs can be reduced to NFAs by changing regular expression operators into new edges until each edge is labelled with a regular expression matching a single string of length at most 1. NFAs, in turn, can be reduced to DFAs using the powerset construction. This shows that GNFAs recognize the same set of formal languages as DFAs and NFAs.

## REGULAR EXPRESSION

A regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. “find and replace”-like operations. Regular expressions are a generalized way to match patterns with sequences of characters. It is used in every programming language like C++, Java and Python. Regex are used in Google analytics in URL matching in supporting search and replace in most popular editors like Sublime, Notepad++, Brackets, Google Docs and Microsoft word.

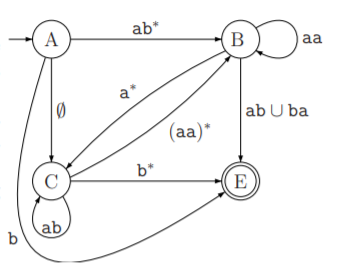
# CHAPTER THREE

**PLANNING AND STEPS**

**DEFINITION OF A GNFA**

A Generalized NFA consider an NFA N where we allowed to write any regular expression on the edges, and not only just symbols. The automata is allowed to travel on an edge, if it can matches a prefix of the unread input, to the regular expression written on the edge. We will refer to such an automata as a NFA (generalized non-deterministic finite automata

Thus, the NFA of the above,



accepts the string abbbbaaba, since



To simplify , we would enforce the following conditions:

(C1) There are transitions going from the initial state to all other states, and there are no transitions into the initial state.

(C2) There is a single accept state that has only transitions coming into it (and no outgoing transitions).

(C3) The accept state is distinct from the initial state.

(C4) Except for the initial and accepting states, all other states are connected to all other states via a transition. In particular, each state has a transition to itself.

When you can not actually go between two states, a NFA has a transitions labelled with ∅, which will not match any string of input characters. We do not have to draw these transitions explicitly in the state diagrams.

**3.1 Conversıon**

We will convert a DFA to a regular expression as follows:

* Convert DFA to a NFA, adding new initial and final states.
* Remove all states one-by-one, until we have only the initial and final states.
* Output regex is the label on the (single) transition left in the NFA. (The word regex is just a shortcut for regular expression.)

**Lemma 1.1.** A DFA M can be converted into an equivalent NFA G.

**Proof:** We can consider M to be an NFA. Next, we add a special initial state qinit that is connected to the old initial state via ε-transition. Next, we add a special final state qfin, such that all the final states of M are connected to qfin via an ε-transition. The modified NFA M0 has an initial state and a single final state, such that no transition enters the initial state, and no transition leaves the final state, thus M0 comply with conditions (C1–C3) above. Next, we consider all pair of states x, y ∈ Q(M0), and if there is no transition between them, we introduce the transition 

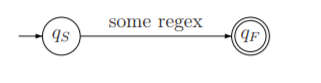
The resulting NFA G from M0 is now compliant also with condition (C4).

It is easy now to verify that G is equivalent to the original DFA M.

We will remove all the intermediate states from the GNFA, leaving a NFA with only initial and final states, connected by one transition with a (typically complex) label on it. The equivalent regular expression is obvious: the label on the transition.

**Lemma 1.2.** Given a NFA N with k = 2 states, one can generate an equivalent regular expression.

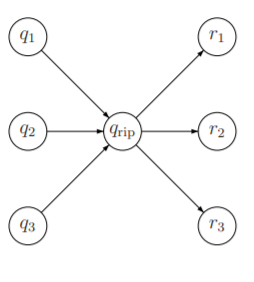
**Proof:** A NFA with only two states (that comply with conditions (C1)-(C4)) have the following form.



The NFA has a single transition from the initial state to the accepting state, and this transition has the regular expression R associated with it. Since the initial state and the accepting state do not have self loops, we conclude that N accepts all words that matches the regular expression R. Namely,

L(N) = L(R).

We first describe the construction. Since k > 2, there is at least one state in N which is not initial or accepting, and let qrip denote this state. We will “rip” this state out of N and fix the NFA, so that we get a NFA with one less state. Transition paths going through qrip might come from any of a variety of states q1, q2, etc. They might go from qrip to any of another set of states r1, r2, etc. For each pair of states qi and ri , we need to convert the transition through qrip into a direct transition from qi to ri.



To understand how this works, let us focus on the connections between qrip and two other specific states qin and qout. Notice that qin and qout might be the same state, but they both have to be different from qrip.

The state qrip has a self loop with regular expression Rrip associated with it. So, consider a fragment of an accepting trace that goes through qrip.

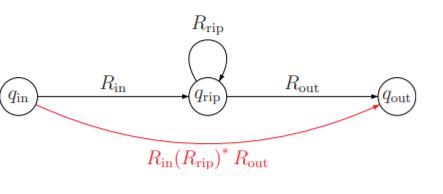
It transition into qrip from a state qin with a regular expression Rin and travels out of qrip into state qout on an edge with the associated regular expression

being Rout. This trace, corresponds to the regular expression Rin followed by 0 or more times of traveling

on the self loop (Rrip is used each time we traverse the loop), and then a transition out to qout using the regular expression Rout. As such, we can introduce a direct transition from qin to qout with the regular

expression:

R = Rin(Rrip) ∗ Rout.



Clearly, any fragment of a trace traveling qin → qrip → qout can be replaced by the direct transition So, let us do this replacement for any two such stages, we connect them directly via a new transition, so that they no longer need to travel through qrip.

Clearly, if we do that for all such pairs, the new automata accepts the same language, but no longer need to use qrip. As such, we can just remove qrip from the resulting automata. And let M0 denote the resulting automata.

The automata M0 is not quite legal, yet. Indeed, we will have now parallel transitions because of the above process (we might even have parallel self loops). But this is easy to fix: We replace two such parallel transitions .by a single transition 

As such, for the triple qin, qrip, qout, if the original label on the direct transition from qin to qout was originally Rdir, then the output label for the new transition (that skips qrip) will be



Clearly the new transition, is equivalent to the two transitions it replaces. If we repeat this process for all the parallel transitions, we get a new NFA M which has k − 1 states, and furthermore it accepts exactly the same language as N.

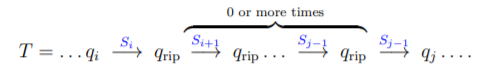
**Lemma 1.3.** Given a NFA N with k > 2 states, one can generate an equivalent NFA M with k – 1 states.

**Proof:** Since k > 2, N contains least one state in N which is not accepting, and let qrip denote this state.

We will “rip” this state out of N and fix the NFA, so that we get a NFA with one less state. For every pair of states qin and qout, both distinct from qrip, we replace the transitions that go through qrip with direct transitions from qin to qout, as described in the previous section.

**Correctness.** Consider an accepting trace T for N for a word w. If T does not use the state qrip than the same trace exactly is an accepting trace for M. So, assume that it uses qrip, in particular, the

trace looks like



Where SiSi+1 . . . , Sj is a substring of w. Clearly, Si ∈ Rin, where Rin is the regular expression associated with the transition qi → qrip. Similarly, Sj−1 ∈ Rout, where Rout is the regular expression associated with the transition qrip → qj . Finally,

Si+1 Si+2 · · · Sj−1 ∈ (Rrip) ∗ , where Rrip is the regular expression associated with the self loop of qrip. Now, clearly, the string SiSi+1 . . . , Sj matches the regular expression Rin(Rout) ∗ Rout. in particular, we can replace this portion of the trace of T by



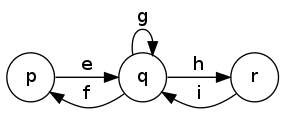
This transition is using the new transition between qi and qj introduced by our construction. Repeating this replacement process in T till all the appearances of qrip are removed, results in an accepting trace Tb of M. Namely, we proved that any string accepted by N is also accepted by M. We need also to prove the other direction. Namely, given an accepting trace for M, we can rewrite it into an equivalent trace of N which is accepting. This is easy, and done in a similar way to what we did above. Indeed, if a portion of the trace uses a new transition of M (that does not appear in N), we can place it by a fragment of transitions going through qrip. In light of the above proof, it is easy and we omit the straightforward but tedious details.

**Theorem 1.4.** Any DFA can be translated into an equivalent regular expression.

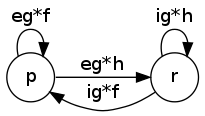
**Proof:** Indeed, convert the DFA into a NFA N. As long as N has more than two states, reduce its number of states by removing one of its states using **Lemma 1.3.** Repeat this process till N has only two states. Now, we convert this NFA into an equivalent regular expression using **Lemma 1.2.**

**3.2 State removal method and examples**

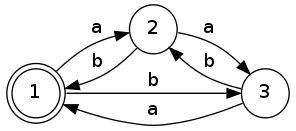
The idea is to consider regular expressions on edges and then removing intermediate states while keeping the edges labels consistent. The main pattern can be seen in the following to figures. The first has labels between p,q,r that are regular expressions e,f,g,h,i and we want to remove q.



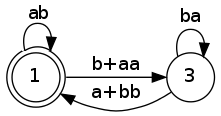
Once removed, we compose e,f,g,h,i together (while preserving the other edges between p and r but this is not displayed on this):



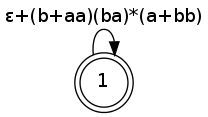
As an example :



we successively remove q2:



and then q3:



then we still have to apply a star on the expression from q1 to q1. In this case, the final state is also initial so we really just need to add a star:

(ab+(b+aa)(ba)∗(a+bb))∗

# CHAPTER FOUR

**PSEUDO CODES OF SOLUTION**

**3.2 Pseudo code of algorıthm**

|  |
| --- |
| transformdfatonfa() |
| String[] startx = *List.Get()*  *start* = startx[1]  String[] acceptx = *List.Get()*  *accept* = acceptx[1]  String[] alphabetparse = *List.Get()*  *alphabet* = alphabetparse[1]  String[] statespase = *List.Get()*  String statesnp = statespase[1]  String[] states = statesnp split by (",")  Add *statelist* ("qs")  for i = 0 to states.length  Add *statelist ,* (states[i])  Add *statelist* ("qe");  *arraysize* = Size of *statelist*  for i = 4 to Size of *list*  String lol =Get i.th member of *list*  Add *transactionlist* (lol) |

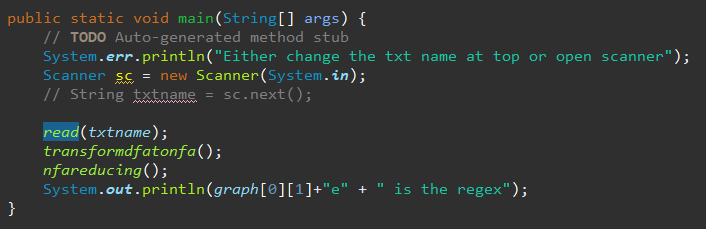
|  |
| --- |
| nfareducing() |
| *graph* ;  *graphchanged* ;  for i = 0 to *transactionlist*.size()  String[] parsing = *transactionlist*.get(i).split(",")  int c = Integer.*valueOf*(parsing[0].substring(1, 2))  String[] lol = parsing[1].split("=")  int g = Integer.*valueOf*(lol[1].substring(1, 2))  if (lol != null)  if (*graph*[c][g] != null)  *graph*[c][g] += lol[0]  else  *graph*[c][g] = lol[0]  int x = *start*.substring(1, 2))  int y = *accept*.substring(1, 2))  *graph*[0][x] = "e”  *graph*[*arraysize* - 1][y] = "e"  for i = 0 to < *graph*.length  for j = 0 to *graph*[0].length  if (*graph*[i][j] != null)  Print (*graph*[i][j] + " ")  else  Print ("0 ")  Call *newtoreg*() |
|  |

|  |
| --- |
| newtoreg () |
| for i = 1 to 2  for j = i to *graph*[0].length-1  String self = ""  if (*graph*[i][i] != null )  if (*graph*[i][i].length() >1)  self ="(" +*graph*[i][i] + ")"+ "\*"  else  self =*graph*[i][i] + "\*"  String loop = "";  if (*graph*[i][j] != null && *graph*[j][i] != null && i != j)  if (self.equals(""))  loop = *graph*[i][j] + self + *graph*[j][i]  else  loop = *graph*[i][j] + *graph*[j][i]  String thereisone =""  if (*graph*[0][j] != null)  thereisone = *graph*[0][j]  String goingright = "";    if (*graph*[i][j + 1] != null)  goingright = *graph*[i][j + 1]  String goingd = "";  if (*graph*[i+1][j] != null)  goingd = *graph*[i+1][j]  if (*work* ==0)  *graphchanged* [0][j] = thereisone+self + goingright + loop  *work*++;  for k = 2 to *graph*.length  for k2 = 2 to *graph*[0].length  *graphchanged*[k-1][k2-1] = *graph*[k][k2]  *graph* = *graphchanged*;  *work*=0;  for x = 0 to *graphchanged*.length; x++)  for j = 0 to *graphchanged*[0].length  if (*graph*[x][j] != null)  Print (*graphchanged*[x][j] + " ")  else  Print ("0 ")  *arraysize*--  *graphchanged* = new String[*arraysize* - 1][*arraysize* - 1]  if (*graph*.length<3)  Print ("stop")    else  Call *newtoreg*() |

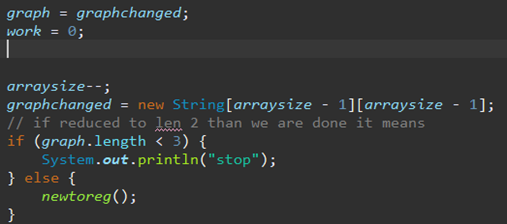
# CHAPTER FIVE

**CODING**

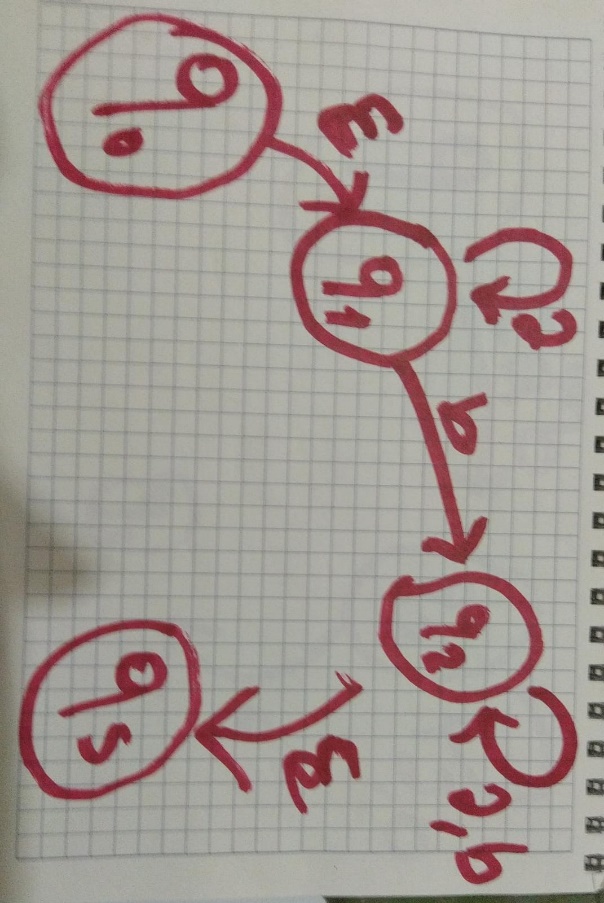
Firstly we need to read given txt and parse it by our need to create a state diagram. In the main funcion we read the txt by read function first, and call transformdfatonfa funtion that parses the read txt values from a list to used in nfa reducing function which is the fucntion that creates the state table and sents it to the newtoreg. After we call newtoreg function it recursively calls it self until state table size not smaller than two.

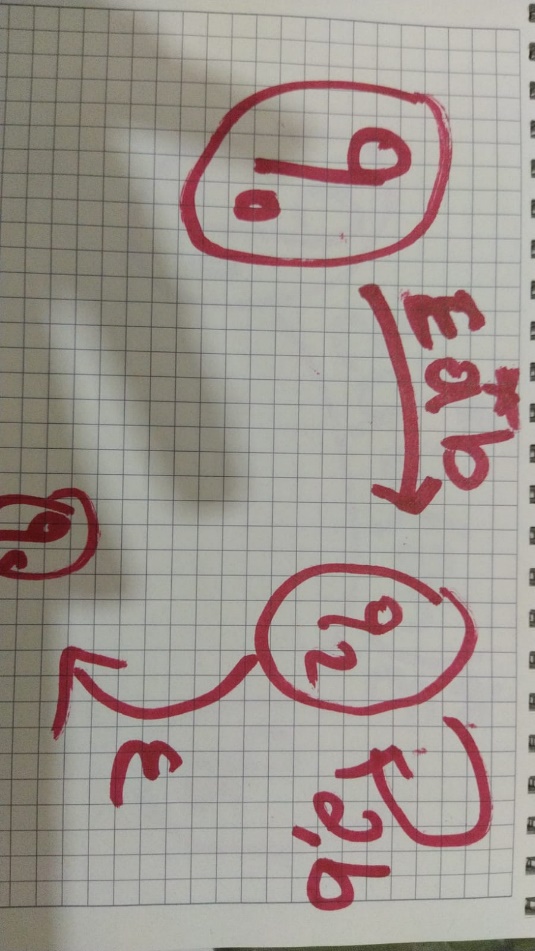


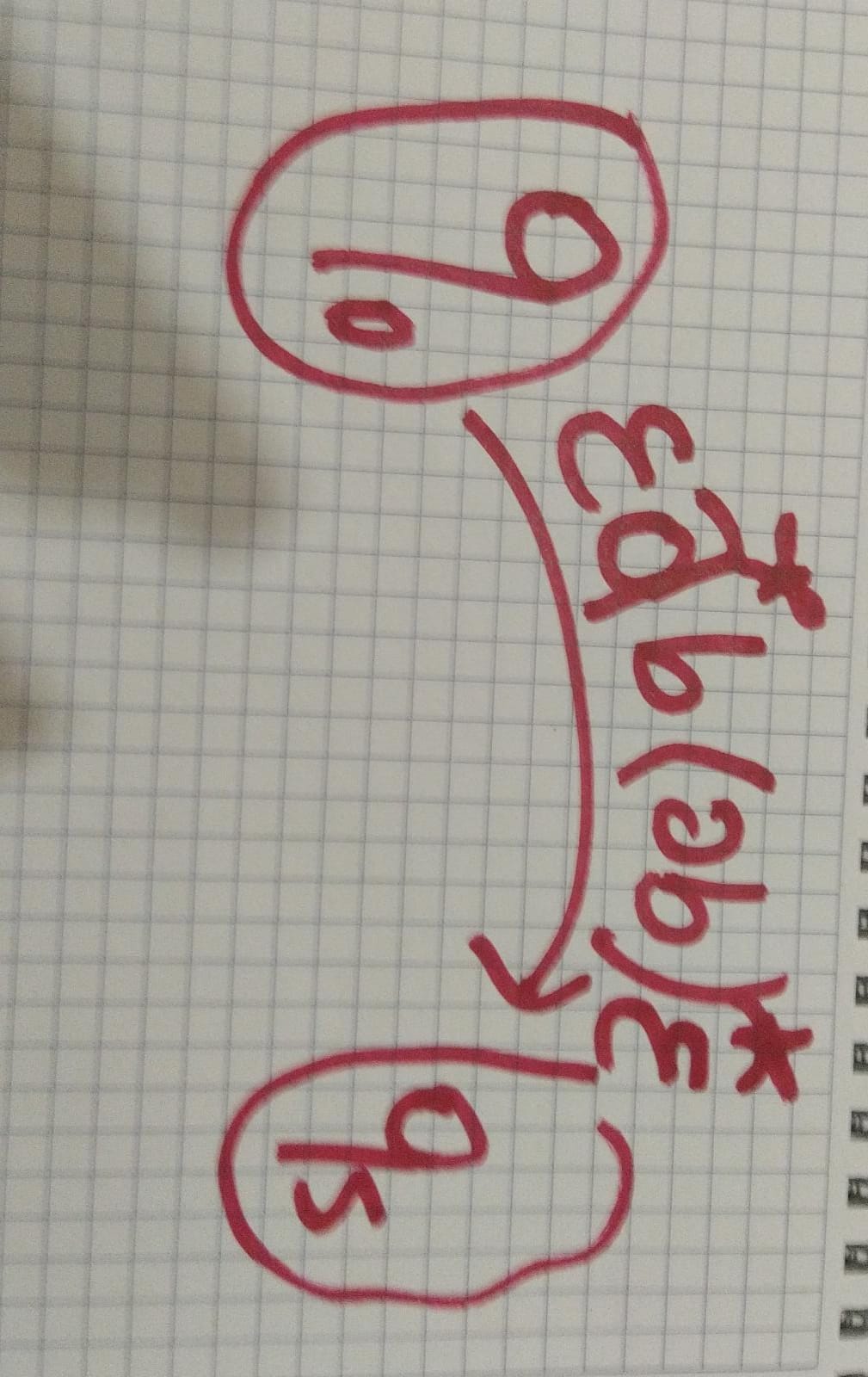
Newtoreg function get the graph 2d array which holds our state tables. When first working the rows and columns 0,1 is merged to eleminate a state. Which includes processes of checking for self return to tables when arr(i,j) matches in for loop when having equal values, arr(i,j)= arr(j,i) condition which means there is a loop from eleminated state to next state and passed on to graphchanged array. Sther states and their data that are not yet reduce is passed on to graphchanged[i - 1][j - 1] and lastly the graph is equalised to graphchanged and recursive call occurs.



GNFA Shema







**SCREENSHOTS**

