**Theory of Computation**

W**hat is TOC**?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

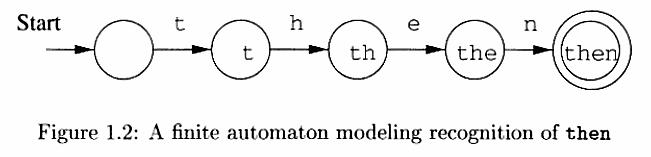
In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine. Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata. This automaton consists of

* **states** (represented in the figure by circles),
* and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.



**Introduction to formal proof:**

**Basic Symbols used :**

U – Union

∩- Conjunction

* - Empty String Φ – NULL set **7**- negation

**‘** – compliment= > implies

W**hat is TOC**?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

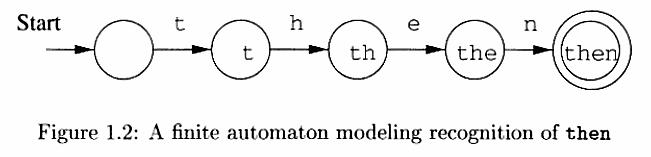
In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine. Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata. This automaton consists of

* **states** (represented in the figure by circles),
* and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.



**Introduction to formal proof:**

**Basic Symbols used :**

U – Union

∩- Conjunction

* - Empty String Φ – NULL set **7**- negation

**‘** – compliment= > implies

W**hat is TOC**?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

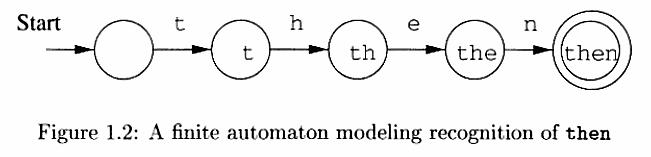
In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine. Automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata. This automaton consists of

* **states** (represented in the figure by circles),
* and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.



**Introduction to formal proof:**

**Basic Symbols used :**

U – Union

∩- Conjunction

* - Empty String Φ – NULL set **7**- negation

**‘** – compliment= > implies

**Strings or Words over Alphabet :**

A string or word over an alphabet is a finite sequence of concatenated symbols of .

**Example** : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alpha-bet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

**Length of a string :**

The number of symbols in a string w is called its length, denoted by |w|.

**Example :** | 011 | = 4, |11| = 2, | b | = 1

**Convention :** We will use small case letters towards the beginning of the English alphabetto denote symbols of an alphabet and small case letters towards the end to

denote strings over an alphabet. That is,

are strings.

**Some String Operations :**

 (symbols) and



Let and be two strings. The concatenation of x and y

denoted by xy, is the string . That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

**Example :** Consider the string011over the binary alphabet. All the prefixes, suffixes andsubstrings of this string are listed below.

Prefixes: , 0, 01, 011.

Suffixes: , 1, 11, 011.

Substrings: , 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and x 蝤 y.

In the above example, all prefixes except 011 are proper prefixes.

**Powers of Strings :** For any string x and integer , we use to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive

definition of as follows:

xx= e, if n = 0 ; otherwise xx=x xx-1

**Example :** If*x*=011,then = 011011011, = 011 and

**Powers of Alphabets :**

We write (for some integer k) to denote the set of strings of length k with symbols from . In other words,

= { w | w is a string over

and

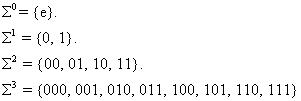
| w | = k}. Hence, for any alphabet,

denotes the set

of all strings of length zero. That is,

= { e }. For the binary alphabet { 0, 1 } we have

the following.



The set of all strings over an alphabet  is denoted by . That is,



The set  bols from

contains all the strings that can be generated by iteratively concatenating sym- any number of times.

**Example :** If = { a, b }, then = { , a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}.

Please note that if , then  that is . It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention



The set of all nonempty strings over an alphabet is denoted by . That is,



Note that is infinite. It contains no infinite strings but strings of arbitrary lengths.

**Reversal :**

For any string

the reversal of the string is

.

An inductive definition of reversal can be given as follows:

**Languages :**

A language over an alphabet is a set of strings over that alphabet. Therefore, a



|  |  |  |  |
| --- | --- | --- | --- |
| language L is any subset of | | . That is, any | is a language. |
| **Example :** | |  |  |
| 1. | F is the empty language. |  |  |
| 2. | is a language for any | . |  |
| 3. | {e} is a language for any | . Note that, | . Because the language F does not |



contain any string but {e} contains one string of length zero.

1. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.
2. The set of all strings over {a, b, c} that starts with a.

**Convention :** Capital letters A, B, C, L, etc. with or without subscripts are normally used todenote languages.

**Set operations on languages :** Since languages are set of strings we can apply set operations tolanguages. Here are some simple examples (though there is nothing new in it).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Union :** A string |  |  |  |  |
|  |  | iff | or |  |
| **Example :** { 0, 11, 01, 011 } | | { 1, 01, 110 } = { 0, 11, 01, 011, 111 } | | |
| **Intersection :** | A string, | xϵ L1 | ∩ L2 | iff x ϵ L1 and x ϵ L2 . |
| **Example :** { 0, 11, 01, 011 } | | { 1, 01, 110 } = { 01 } | | |



Complement **:** Usually, is the universe that a complement is taken with respect to.



Thus for a language L, the complement is L(bar) =

{

| 

}.

**Example :** Let L = { x | |x| is even }. Then its complement is the language

{

| |*x*| is

odd }.

Similarly we can define other usual set operations on languages like relative com-plement, symmetric difference, etc.

**Reversal of a language :**

The reversal of a language *L*, denoted as , is defined as: .

**Example :**

****

|  |  |
| --- | --- |
| 1. Let L = { 0, 11, 01, 011 }. Then | = { 0, 11, 10, 110 }. |

2. Let L = { 

| n is an integer }. Then

= { 

| n is an integer }.

**Language concatenation :**

****

= { *xy* | and

The concatenation of languages

}.

and

is defined as

**Example :** {*a*,*ab*}{*b*,*ba*} = {*ab*,*aba*,*abb*,*abba*}.

Note that ,

1.  in general.
   1. 
   2. 

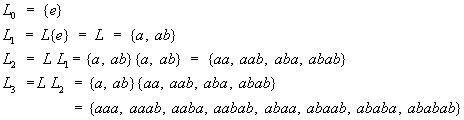
**Iterated concatenation of languages :** Since we can concatenate two languages, we also repeatthis to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation



L with itself n times. This is defined formally as follows:



**Example :** Let L = { a, ab }. Then according to the definition, we have



and so on.



**Kleene's Star operation :** The Kleene star operation on a language L, denoted as isdefined as follows :



= ( Union n in N )

= 

= { x | x is the concatenation of zero or more strings from L }

Thus is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define



* , i.e., all strings derivable by one or more concatenations of strings in L. That is



= (Union n in N and n >0)



=

**Example :** Let L = { a, ab }. Then we have,



=

* {*e*} {*a*, *ab*} {*aa*, *aab*, *aba*, *abab*} …

=



* {*a*, *ab*} {*aa*, *aab*, *aba*, *abab*} …



|  |  |  |
| --- | --- | --- |
| Note : is in | , for every language L, including *.* | |
| The previously introduced definition of | | is an instance of Kleene star. |

Finite Automation

(Generates) (Recognizes)

Grammar Language  Automata

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

* Every automaton consists of some essential features as in real computers. It has a mech-anism for reading input. The input is assumed to be a sequence of symbols over a given alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.
* The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an out-put sequence in response to an input sequence, then it is called a transducer(or automaton with output).
* The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet ( whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
* The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some de-fined manner determined by a transition function.

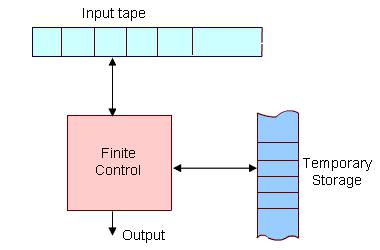


Figure 1: The figure above shows a diagrammatic representation of a generic automa-tion.

Operation of the automation is defined as follows.

At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automa-ton then moves to some other integral (or remain in the same state) as defined by the transition function. The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed. The automation may also produce some output during this transition. The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next ( as defined by the transition function) is called a *move.* Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

**Finite Automata**

Automata (singular : automation) are a particularly simple, but useful, model of compu-tation. They were initially proposed as a simple model for the behavior of neurons.

**States, Transitions and Finite-State Transition System :**

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

*Transitions* are changes of states that can occur spontaneously or in response to inputs to thestates. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).

Some examples of state transition systems are: digital systems, vending machines, etc. A system

containing only a finite number of states and transitions among them is called

1. *finite-state transition system*.

Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

**Deterministic Finite (-state) Automata**

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an in-put string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deter-ministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each

cell holds a symbol from .

1. A *tape head* for reading symbols from the tape
2. A *control* , which itself consists of 3 things:
3. finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
4. a current state, initially set to a start state,

* a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summariz-

ing all the above we can formulate the following formal definition:

**Deterministic Finite State Automaton :** A Deterministic Finite State Automaton (DFA) isa 5-tuple :



* *Q* is a finite set of states.
* is a finite set of input symbols or alphabet

 is the “next state” transition function (which is total ). Intuitively,

function that tells which state to move to in response to an input, i.e., if M is in

is

a

state q and sees input a, it moves to state

.

* is the start state.
* is the set of accept or final states*.*

**Acceptance of Strings :**

A DFA accepts a string

if there is a sequence of states

in

*Q*

such that

1. is the start state.

2. for all .

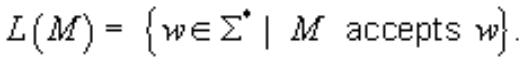


3.

**Language Accepted or Recognized by a DFA :**

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and

is denoted by

 i.e.

The

notion

of

acceptance can also be made more precise by extending the transition function

.

**Extended transition function :**

Extend (which is function on symbols) to a function on strings, i.e. .



That is,  is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:

The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

L(M) = { | *M* accepts *w* }

= {| }

**Example 1 :**

****

is the start state



It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1

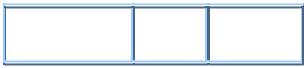
We can describe the same DFA by transition table or state transition diagram as follow-ing:

**Transition Table :**

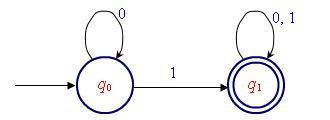
****

0  1





It is easy to comprehend the transition diagram.



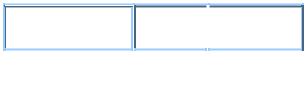
**Explanation :** We cannot reach find state w/0 or in the i/p string. There can be any no.

of 0's at the beginning. ( The self-loop at  on label 0 indicates it ). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

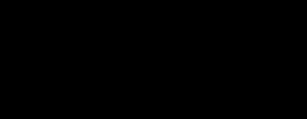
**Transition table :**

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the “next state”).

* Rows correspond to states,
* Columns correspond to input symbols,
* Entries correspond to next states
* The start state is marked with an arrow
* The accept states are marked with a star (\*).



0  1



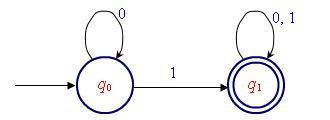
**(State) Transition diagram :**

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.

2. There is a directed edge from node q to node p labeled a iff  . (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)

1. There is an arrow with no source into the start state.
2. Accepting states are indicated by double circle.



|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 5. |  |
| 6. | Here is an informal description how a DFA operates. An input to a DFA can be any | | |
|  | string. | Put a pointer to the start state q. Read the input string w from left | |
|  | to right, one symbol at a time, moving the pointer according to the transition | |  |
|  | function, | . If the next symbol of w is a and the pointer is on state p, move the | |
|  | pointer to | . When the end of the input string w is encountered, the pointer is on | |
|  | some state, r. The string is said to be accepted by the DFA if | | and |
|  | rejected if | . Note that there is no formal mechanism for moving the pointer. | |
| 7. | A language | is said to be regular if L = L(M) for some DFA M. |  |

**Non-Deterministic Finite Automata**

Nondeterminism is an important abstraction in computer science. Importance of nondeterminism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. ( Travelling salesman, Hamiltonean cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because

the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.

In the context of FA nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

* multiple next state.
* - transitions.

**Multiple Next State :**

* In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in ).
* This means that - in a state *q* and with input symbol a - there could be one, more

than one or zero next state to go, i.e. the value of is a subset of *Q*. Thus = which means that any one of could be the next state.

* The zero next state case is a special one giving =, which means that there is no next state on input symbol when the automata is in state *q*. In such a case, we may think that the automata "hangs" and the input will be rejected.

**- transitions :**

In an -transition, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one

or more states. This is written formally as implying that the next

state could by any one of w/o consuming the next input symbol.

**Formal definition of NFA** :

Formally, an NFA is a quituple where *Q*, , , and *F* bear the same meaning as for a DFA, but , the transition function is redefined as follows:



where *P*(*Q*) is the power set of *Q* i.e. .

**The Langauge of an NFA** :

From the discussion of the acceptance by an NFA, we can give the formal definition of a language accepted by an NFA as follows :

If is an NFA, then the langauge accepted by *N* is writtten as *L*(*N*) is

given by .

That is, *L*(*N*) is the set of all strings *w* in such that contains at least one accepting state.

Removing ϵ-transition:

- transitions do not increase the power of an *NFA* . That is, any - *NFA* ( *NFA* with transition), we can always construct an equivalent *NFA* without -transitions. The

equivalent *NFA* must keep track where the *NFA* goes at every step duringcomputation. This can be done by adding extra transitions for removal of every - transitions from the - *NFA* as follows.

If we removed the - transition from the - *NFA* , then we need to moves

from state *p* to all the state on input symbol which are reachable from state q (in the - *NFA* ) on same input symbol *q*. This will allow the modified *NFA* to move from state *p* to all states on some input symbols which were possible in case of -*NFA* on the same input symbol. This process is stated formally in the following theories.

Theorem if *L* is accepted by an - *NFA N* , then there is some equivalent without transitions accepting the same language *L*

*Proof:*

*Let *be the given with



We construct

Where, for all and and



Other elements of *N'* and *N*

We can show that i.e. *N'* and *N* are equivalent.

We need to prove that 

 i.e.



We will show something more, that is,

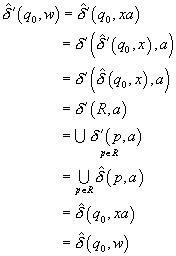


We will show something more, that is, 

Basis : , then 

But by definition of .

Induction hypothesis Let the statement hold for all with .



By definition of extension of 

By inductions hypothesis.

Assuming that



By definition of 

Since 

To complete the proof we consider the case

When i.e. then

and by the construction of wherever constrains a state in *F*.

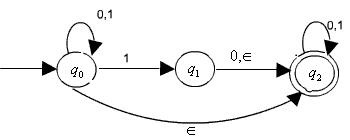
If (and thus is not in *F* ), then with leads to an accepting state in *N'* iff it lead to an accepting state in *N* ( by the construction of *N'* and *N* ).

Also, if ( , thus *w* is accepted by N' iff *w* is accepted by *N* (iff )

If (and, thus in *M* we load in *F* ), thus is accepted by both *N'* and *N* .

Let . If *w* cannot lead to in *N* , then . (Since can add transitions to get an accept state). So there is no harm in making an accept state in *N'*.

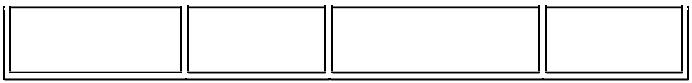
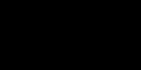
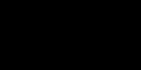
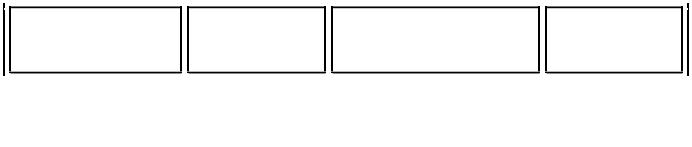
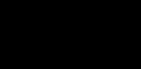
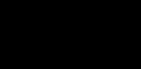
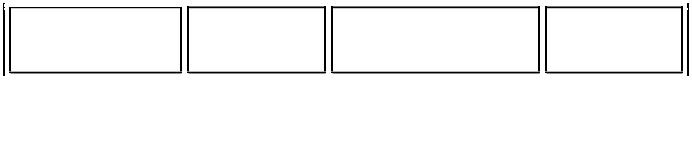
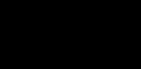
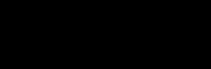
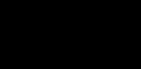
Ex: Consider the following *NFA* with - transition.



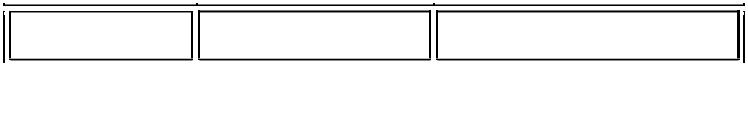
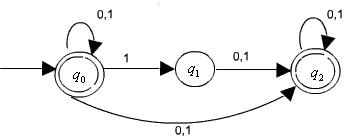
Transition Diagram 



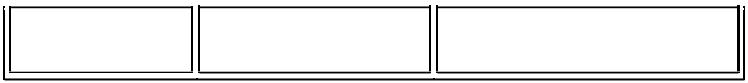
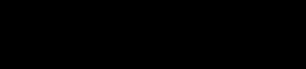
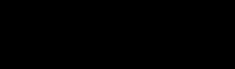
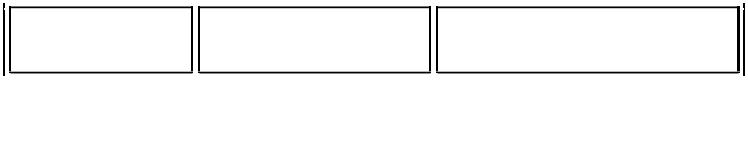
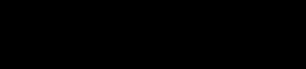
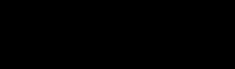
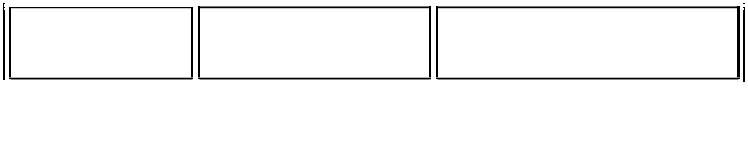
0 1



Transition diagram for ' for the equivalent *NFA* without - moves



0 1



Since the start state *q*0 must be final state in the equivalent *NFA* .

Since and and we add moves and in the equivalent *NFA* . Other moves are also constructed accordingly.

-closures:

The concept used in the above construction can be made more formal by defining the -closure for a state (or a set of states). The idea of -closure is that, when moving

from a state *p* to a state *q* (or from a set of states Si to a set of states Sj ) an input , we need to take account of all -moves that could be made after the transition. Formally, for a given state *q*,



-closures:

Similarly, for a given set 

-closures:



So, in the construction of equivalent *NFA N'* without -transition from any *NFA* with moves. the first rule can now be written as



Equivalence of *NFA* and *DFA*

It is worth noting that a *DFA* is a special type of *NFA* and hence the class of languages accepted by *DFA* s is a subset of the class of languages accepted by *NFA* s. Surprisingly, these two classes are in fact equal. *NFA* s appeared to have more power than *DFA* s because of generality enjoyed in terms of -transition and multiple next states. But they are no more powerful than *DFA* s in terms of the languages they accept.

Converting *DFA* to *NFA*

Theorem: Every *DFA* has as equivalent *NFA*

Proof: A *DFA* is just a special type of an *NFA* . In a *DFA* , the transition functions is

defined from whereas in case of an *NFA* it is defined from and

be *a DFA* . We construct an equivalent *NFA *as follows.



i. e



If and

All other elements of *N* are as in *D*.

If then there is a sequence of states such that



Then it is clear from the above construction of *N* that there is a sequence of states (in *N*)

such that and and hence 

Similarly we can show the converse.

Hence , 

Given any *NFA* we need to construct as equivalent *DFA* i.e. the *DFA* need to simulate the behaviour of the *NFA* . For this, the *DFA* have to keep track of all the states where the NFA could be in at every step during processing a given input string.

There are possible subsets of states for any *NFA* with *n* states. Every subset corresponds to one of the possibilities that the equivalent *DFA* must keep track of. Thus, the equivalent *DFA* will have states.

The formal constructions of an equivalent *DFA* for any *NFA* is given below. We first consider an *NFA* without transitions and then we incorporate the affects of transitions later.

Formal construction of an equivalent *DFA* for a given *NFA* without transitions.

Given an without - moves, we construct an equivalent *DFA*

**

as follows



i.e.



(i.e. every subset of *Q* which as an element in *F* is considered as a final stat

in *DFA D* )



for all and 

where 

That is, 

To show that this construction works we need to show that *L(D)=L(N)* i.e.



Or,

We will prove the following which is a stranger statement thus required.



Proof : We will show by inductions on 

Basis If =0, then *w =*

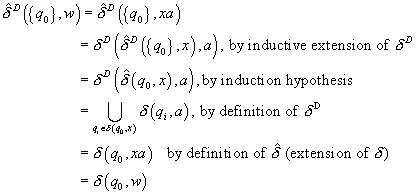
*So, *by definition.

Inductions hypothesis : Assume inductively that the statement holds of length less than or equal to *n*.

Inductive step

Let , then with 

Now,



Now, given any *NFA* with -transition, we can first construct an equivalent *NFA* without -transition and then use the above construction process to construct an equivalent *DFA* , thus, proving the equivalence of *NFA* s and *DFA* s..

It is also possible to construct an equivalent *DFA* directly from any given *NFA* with - transition by integrating the concept of -closure in the above construction.

Recall that, for any 

- closure :



In the equivalent *DFA* , at every step, we need to modify the transition functions to keep track of all the states where the *NFA* can go on -transitions. This is done by

replacing by -closure , i.e. we now compute at every step as follows:



Besides this the initial state of the *DFA D* has to be modified to keep track of all the states that can be reached from the initial state of *NFA* on zero or more -transitions.

This can be done by changing the initial state to -closure ( ) .

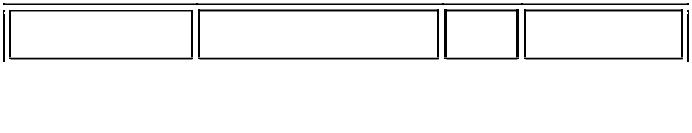
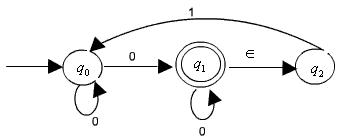
It is clear that, at every step in the processing of an input string by the *DFA D* , it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any

-*NFA*

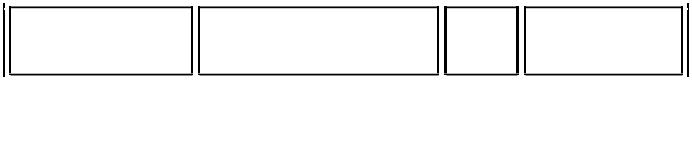
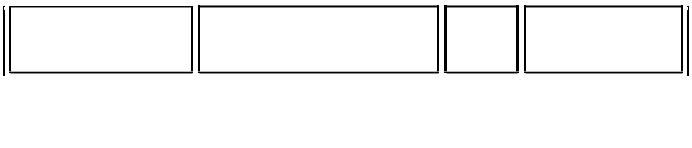
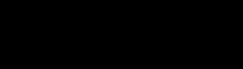
If the number of states in the *NFA* is *n* , then there are states in the *DFA* . That is, each state in the *DFA* is a subset of state of the *NFA* .

But, it is important to note that most of these states are inaccessible from the start state and hence can be removed from the *DFA* without changing the accepted language. Thus, in fact, the number of states in the equivalent *DFA* would be much less than .

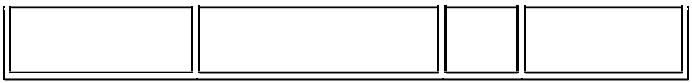
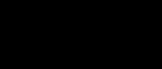
Example : Consider the NFA given below.



0 1

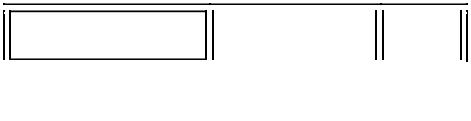


{}

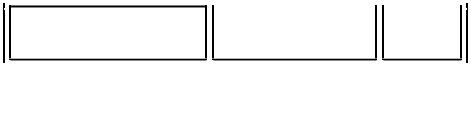


Since there are 3 states in the NFA

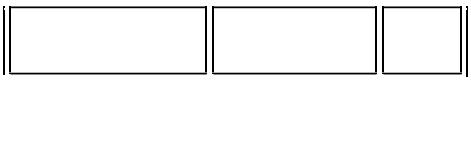
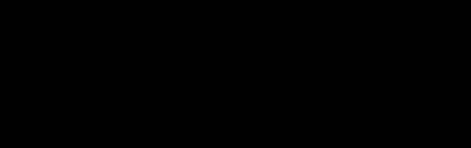
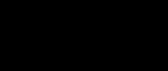
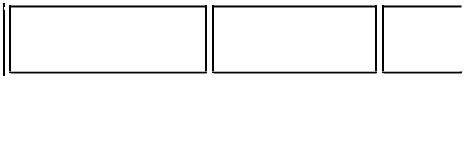
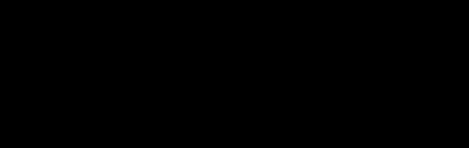
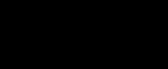
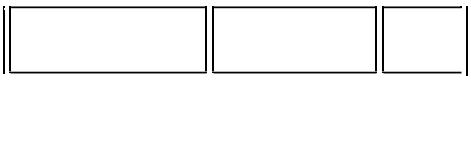
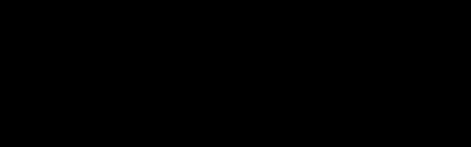
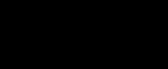
There will be states (representing all possible subset of states) in the equivalent *DFA* . The transition table of the *DFA* constructed by using the subset constructionsprocess is produced here.



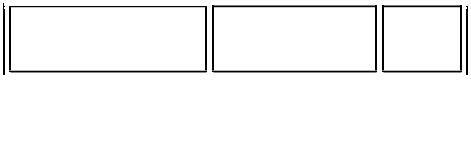
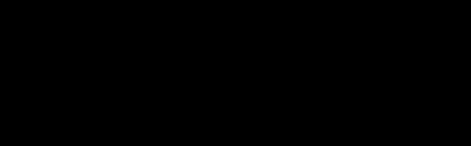
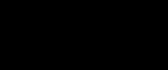
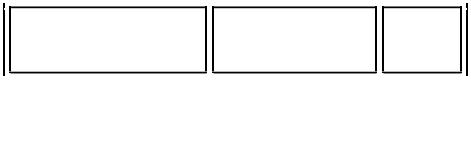
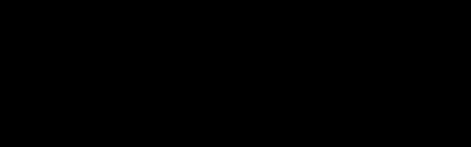
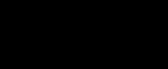
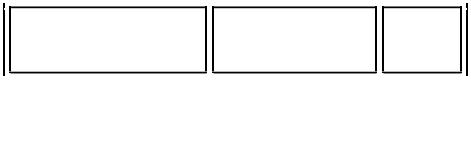
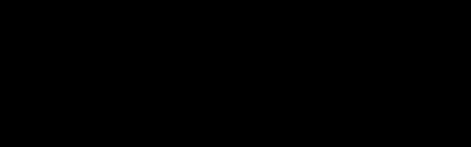
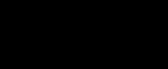
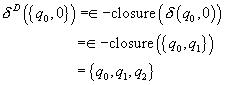
|  |  |  |  |
| --- | --- | --- | --- |
| 0 |  | 1 | The start state of the *DFA* is - closures |
|  |  |  |



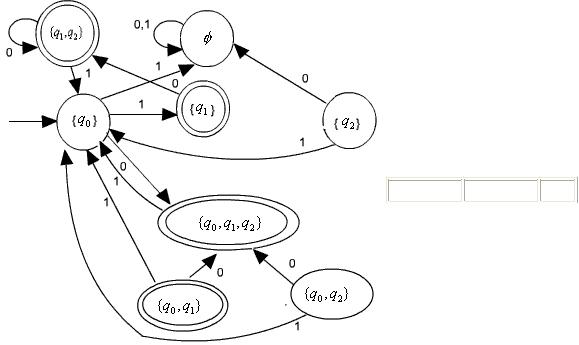
 The final states are all those subsets that contains (since in the *NFA*).



|  |  |
| --- | --- |
| { } | Let us compute one entry, |
|  |



 Similarly, all other transitions can be computed

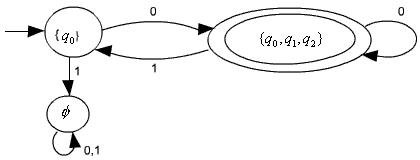


0 1



**Corresponding Transition fig. for DFA.**Note that states

are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



It is interesting to note that we can avoid encountering all those inaccessible or unnecessary states in the equivalent *DFA* by performing the following two steps inductively.

1. If is the start state of the NFA, then make - closure ( ) the start state of the equivalent *DFA* . This is definitely the only accessible state.
2. If we have already computed a set of states which are accessible. Then

. compute because these set of states will also be accessible.