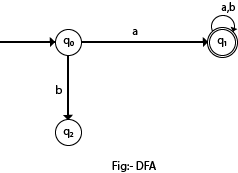
**Practical 2:**

**Aim: - Write a program to minimize DFA.**

**Theory: -**

* DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
* In DFA, there is only one path for specific input from the current state to the next state.
* DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
* DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.



## **Formal Definition of DFA**

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

Transition function can be defined as:

1. δ: Q x ∑→Q

## **Graphical Representation of DFA**

A DFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

### Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

**Solution:**

Transition Diagram:

Deterministic finite automata

**Transition Table:**

|  |  |  |
| --- | --- | --- |
| Present State | Next state for Input 0 | Next State of Input 1 |
| →q0 | q0 | q1 |
| q1 | q2 | q1 |
| \*q2 | q2 | q2 |

# **Minimization of DFA**

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

**Step 1:** Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

**Step 2:** Draw the transition table for all pair of states.

**Step 3:** Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

**Step 4:** Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

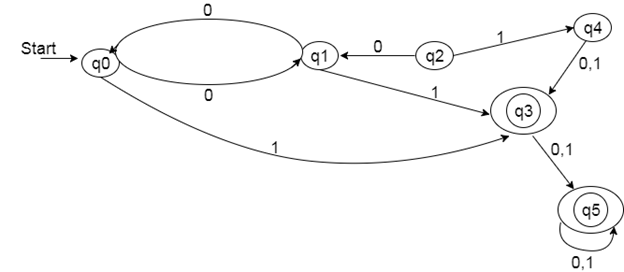
That means, find the two states which have the same value of a and b and remove one of them.

**Step 5:** Repeat step 3 until we find no similar rows available in the transition table T1.

**Step 6:** Repeat step 3 and step 4 for table T2 also.

**Step 7:** Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

### Example:



**Solution:**

**Step 1:** In the given DFA, q2 and q4 are the unreachable states so remove them.

**Step 2:** Draw the transition table for the rest of the states.

|  |  |  |
| --- | --- | --- |
| State | 0 | 1 |
| →q0 | q1 | q3 |
| q1 | q0 | q3 |
| \*q3 | q5 | q5 |
| \*q5 | q5 | q5 |

**Step 3:** Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

|  |  |  |
| --- | --- | --- |
| State | 0 | 1 |
| q0 | q1 | q3 |
| q1 | q0 | q3 |

2. Another set contains those rows, which starts from final states.

|  |  |  |
| --- | --- | --- |
| State | 0 | 1 |
| q3 | q5 | q5 |
| q5 | q5 | q5 |

**Step 4:** Set 1 has no similar rows so set 1 will be the same.

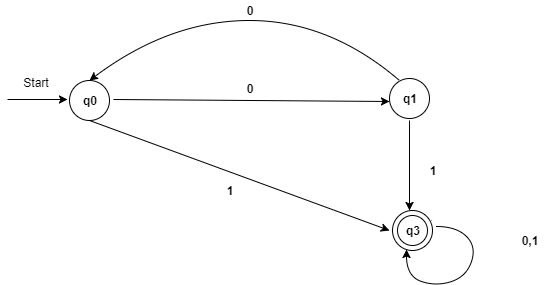
**Step 5:** In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

|  |  |  |
| --- | --- | --- |
| State | 0 | 1 |
| q3 | q3 | q3 |

**Step 6:** Now combine set 1 and set 2 as:

|  |  |  |
| --- | --- | --- |
| State | 0 | 1 |
| →q0 | q1 | q3 |
| q1 | q0 | q3 |
| \*q3 | q3 | q3 |

**Now it is the transition table of minimized DFA.**



**Code:-**

**Disjointset.py**

class DisjointSet(object):

def \_\_init\_\_(self,items):

self.\_disjoint\_set = list()

if items:

for item in set(items):

self.\_disjoint\_set.append([item])

def \_get\_index(self,item):

for s in self.\_disjoint\_set:

for \_item in s:

if \_item == item:

return self.\_disjoint\_set.index(s)

return None

def find(self,item):

for s in self.\_disjoint\_set:

if item in s:

return s

return None

def find\_set(self,item):

s = self.\_get\_index(item)

return s+1 if s is not None else None

def union(self,item1,item2):

i = self.\_get\_index(item1)

j = self.\_get\_index(item2)

if i != j:

self.\_disjoint\_set[i] += self.\_disjoint\_set[j]

del self.\_disjoint\_set[j]

def get(self):

return self.\_disjoint\_set

**DFA.py**

from collections import defaultdict

from disjoint\_set import DisjointSet

class DFA(object):

def \_\_init\_\_(self,states\_or\_filename,terminals=None,start\_state=None, transitions=None,final\_states=None):

#if values in graph file

if terminals is None:

self.\_get\_graph\_from\_file(states\_or\_filename)

#if manual values

else:

assert isinstance(states\_or\_filename,list) or isinstance(states\_or\_filename,tuple)

self.states = states\_or\_filename

assert isinstance(terminals,list) or isinstance(terminals,tuple)

self.terminals = terminals

assert isinstance(start\_state,str)

self.start\_state = start\_state

assert isinstance(transitions,dict)

self.transitions = transitions

assert isinstance(final\_states,list) or isinstance(final\_states,tuple)

self.final\_states = final\_states

def \_remove\_unreachable\_states(self):

'''

Removes states that are unreachable from the start state

'''

g = defaultdict(list)

for k,v in self.transitions.items():

g[k[0]].append(v)

# do DFS

stack = [self.start\_state]

reachable\_states = set()

while stack:

state = stack.pop()

if state not in reachable\_states:

stack += g[state]

reachable\_states.add(state)

self.states = [state for state in self.states if state in reachable\_states]

self.final\_states = [state for state in self.final\_states if state in reachable\_states]

self.transitions = { k:v for k,v in self.transitions.items() \

if k[0] in reachable\_states}

def minimize(self):

self.\_remove\_unreachable\_states()

def order\_tuple(a,b):

return (a,b) if a < b else (b,a)

table = {}

sorted\_states = sorted(self.states)

# initialize the table

for i,item in enumerate(sorted\_states):

for item\_2 in sorted\_states[i+1:]:

table[(item,item\_2)] = (item in self.final\_states) != (item\_2\

in self.final\_states)

flag = True

# table filling method

while flag:

flag = False

for i,item in enumerate(sorted\_states):

for item\_2 in sorted\_states[i+1:]:

if table[(item,item\_2)]:

continue

# check if the states are distinguishable

for w in self.terminals:

t1 = self.transitions.get((item,w),None)

t2 = self.transitions.get((item\_2,w),None)

if t1 is not None and t2 is not None and t1 != t2:

marked = table[order\_tuple(t1,t2)]

flag = flag or marked

table[(item,item\_2)] = marked

if marked:

break

d = DisjointSet(self.states)

# form new states

for k,v in table.items():

if not v:

d.union(k[0],k[1])

self.states = [str(x) for x in range(1,1+len(d.get()))]

new\_final\_states = []

self.start\_state = str(d.find\_set(self.start\_state))

for s in d.get():

for item in s:

if item in self.final\_states:

new\_final\_states.append(str(d.find\_set(item)))

break

self.transitions = {(str(d.find\_set(k[0])),k[1]):str(d.find\_set(v)) \

for k,v in self.transitions.items()}

self.final\_states = new\_final\_states

def show(self):

temp = defaultdict(list)

for k,v in self.transitions.items():

temp[(k[0],v)].append(k[1])

return temp

def \_\_str\_\_(self):

'''

String representation

'''

num\_of\_state = len(self.states)

start\_state = self.start\_state

num\_of\_final = len(self.final\_states)

return '{} states. {} final states. start state - {}'.format( \

num\_of\_state,num\_of\_final,start\_state)

def \_get\_graph\_from\_file(self,filename):

'''

Load the graph from file

'''

with open(filename,'r') as f:

try:

lines = f.readlines()

states,terminals,start\_state,final\_states = lines[:4]

if states:

self.states = states[:-1].split()

else:

raise Exception('Invalid file format: cannot read states')

if terminals:

self.terminals = terminals[:-1].split()

else:

raise Exception('Invalid file format: cannot read terminals')

if start\_state:

self.start\_state = start\_state[:-1]

else:

raise Exception('Invalid file format: cannot read start state')

if final\_states:

self.final\_states = final\_states[:-1].split()

else:

raise Exception('Invalid file format: cannot read final states')

lines = lines[4:]

self.transitions = {}

for line in lines:

current\_state,terminal,next\_state = line[:-1].split()

self.transitions[(current\_state,terminal)] = next\_state

except Exception as e:

print("ERROR: ",e)

if \_\_name\_\_ =="\_\_main\_\_":

filename = 'graph'

dfa = DFA(filename)

x = dict(dfa.show())

#initial

for key,value in x.items():

print(key, ':', value)

print(dfa, "\n")

dfa.minimize()

#after minimizing

x = dict(dfa.show())

for key,value in x.items():

print(key, ':', value)

print(dfa)

**Graph Input:-**

## Input Graph Format

1. Space separated list of states. (**Q**)
2. Space separated list of terminals. (**∑**)
3. Start state. (**q0**)
4. Space separated list of final states. (**F**)
5. N lines of 3 space separated symbols A, b and C representing transition  **A→C. (δ)**

**b**

1 2 3 4 5

a b

1

1 5

1 a 3

1 b 2

2 b 1

2 a 4

3 b 4

3 a 5

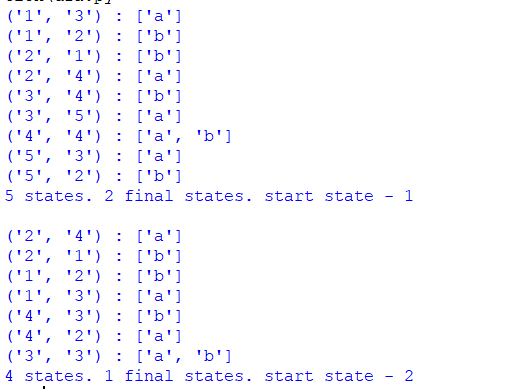
4 a 4

4 b 4

5 a 3

5 b 2

**Output:-**

****

**Conclusion:-**

A minimized DFA was successfully observed from 5 total states to 4 states.

**References :-**

<https://github.com/navin-mohan/dfa-minimization>

<https://www.javatpoint.com/minimization-of-dfa>

https://www.javatpoint.com/deterministic-finite-automata