Experiment no – 02

Aim: Write a program to minimize given DFA.

Theory Explanation :

- 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.

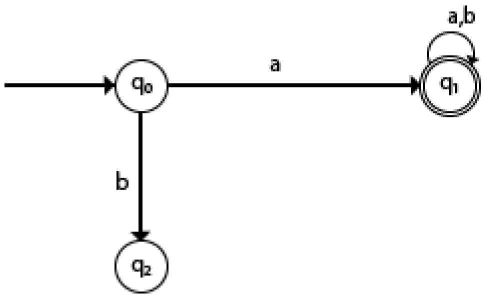


Fig:- DFA

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:

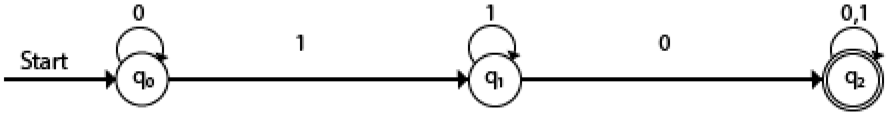
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

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

Solution:

Transition Diagram:



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

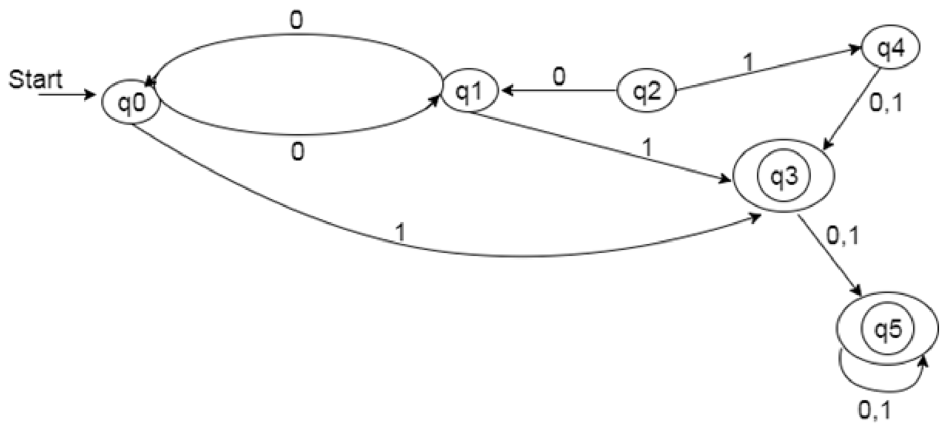
That means, find the two states which have the same value of a and b and remove one of them.

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.

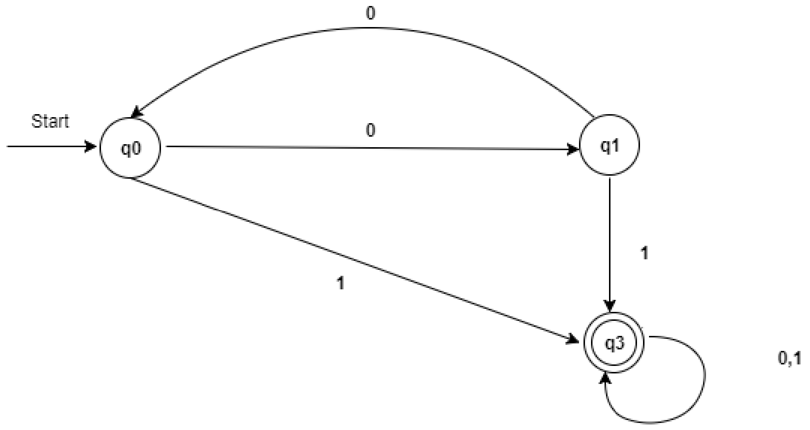
Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

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.



Program:

1. **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 terminals is None:
      self._get_graph_from_file(states_or_filename)
    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 __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)
```

```
    print(dfa)
    dfa.minimize()
    print(dfa)
```

2. **Disjoint.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
```

3. Graph Input

```
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:

```
('1', '3') : ['a']
('1', '2') : ['b']
('2', '1') : ['b']
('2', '4') : ['a']
('3', '4') : ['b']
('3', '5') : ['a']
('4', '4') : ['a', 'b']
('5', '3') : ['a']
('5', '2') : ['b']
5 states. 2 final states. start state - 1

('2', '4') : ['a']
('2', '1') : ['b']
('1', '2') : ['b']
('1', '3') : ['a']
('4', '3') : ['b']
('4', '2') : ['a']
('3', '3') : ['a', 'b']
4 states. 1 final states. start state - 2
```

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