PAF KIET

Karachi Institute of Economics and Technology

Automata Project Report

Project Title: DFA Minimization

Class ID: 110664

SUBMITTED TO: SIR AZIZ MEHMOOD FAROOQI.

GROUP MEMBERS	NAME	STUDENT ID
1	MUHAMMAD MAAZ	9906
2	MUHAMMAD MAAZ	9967
3	MOUNIS UMER	

AUTOMATA PROJECT

DFA MINIMIZATION:

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states

Minimization of DFA:

Suppose there is a DFA D < {Q, Σ, q, F, δ} > which recognizes a language L. Then the minimized DFA D < {Q, Σ, q, F, δ} > can be constructed for language Las:

Step 1: We will divide O (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called Po.

Step 2: Initialize k = 1

Step 3: Find Pk by partitioning the different sets of P&-i. In each set of P&-t, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in Pk.

Step 4: Stop when Pk = Pk-1 (No change in partition)

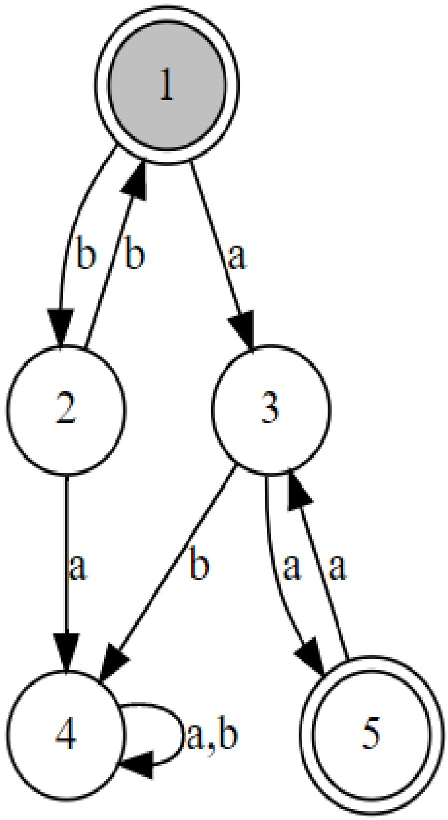
Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no, of sets in PR.

FIVE TUPLES:

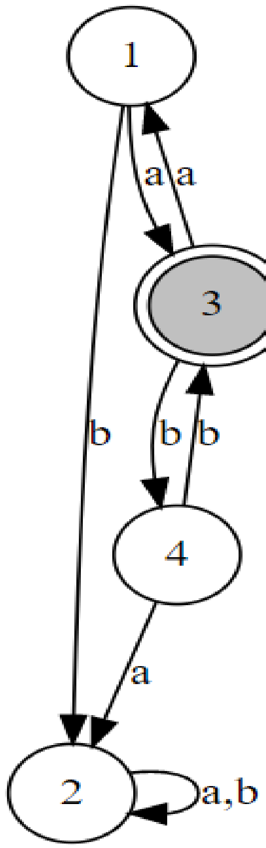
DFA consists of 5 tuples {Q, Σ, q, F, δ}.
Q : set of all states.
Σ : set of input symbols. (Symbols which machine takes as input)
q : Initial state. (Starting state of a machine)
F : set of final state.
δ : Transition Function, defined as δ : Q X Σ --> Q

Example:

DFA:



Minimized DFA:



Input Graph Format:

Space separated list of states. ()

Space separated list of terminals. ()

Start state(

)

Space separated list of final states. ()

N line of 3 space separated symbols, and representing transition. ()

CODE:

```
[ ]  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
```

```
from collections import defaultdict

import networkx as nx
import matplotlib.pyplot as plt
from graphviz import Source


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 draw(self):
  '''
  Draws the dfa using networkx and matplotlib
  '''
  g = nx.DiGraph()

  for x in self.states:
      g.add_node(x,shape='doublecircle'
       if x in self.final_states
       else 'circle'
      ,fillcolor='grey'
       if x == self.start_state
        else 'white',style='filled')

    temp = defaultdict(list)
  for k,v in self.transitions.items():
      temp[(k[0],v)].append(k[1])

  for k,v in temp.items():
      g.add_edge(k[0],k[1],label=','.join(v))


  return Source(nx.drawing.nx_agraph.to_agraph(g))
```

```python
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}
```

```python
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
```

```python
        # 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)
  # print(d.get())
  # 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
```

```python
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)
```

```python
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)
```

```
[ ] import networkx as nx
    import matplotlib.pyplot as plt

    %matplotlib inline
```

```
[ ] # with open('Graph', 'w') as writefile:
    #     writefile.write("This is line A")
    with open('Graph', 'r') as testwritefile:
        print(testwritefile.read())
```

```
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
```

```
[ ] filename = 'Graph'
    dfa = DFA(filename)
    dfa.draw()
```

```
ERROR: not enough values to unpack (expe
```

```
[ ] # minimize dfa
    dfa.minimize()
    print(dfa)
```

```
4 states. 1 final states. start state - 3
```

```
[ ] # draw minimized dfa
    dfa.draw()
```