18P2824 – Maryam Ahmed Galal Nouh
18P3054 – Youssef Sherif Mohamed Youssef
18P6951 – Mostafa Hesham Abd El Raouf

Automata and Computability Course Project Report
NFA to DFA Converter

I. Relevance

As previously studied in the course, the NFA to DFA conversion is a very important step transforming a design to implementation.

II. Abstract

The idea of the project is to convert an NFA, that accepts epsilon transitions, and that allows choices on transitions in a state, to a DFA. Given the quintuple defining the NFA, its graph is constructed, as well as the graph of the converted DFA.

III. Introduction

A DFA, or a Deterministic Finite Automata, is a finite state machine with states, alphabet, a start state, a set of final states, and a transition table. The transition table represents the transitions allowed per state, when given each input symbol, leading to a next state, to which one moves to. For each states, transitions for all possible input symbols must be defined, with no space for decisions to be made, and where epsilon transitions are not possible.

Meanwhile, an NFA, short for Nondeterministic Finite Automata, is a finite state machine where, at a given state, given a single input, or none at all, one can move to another state (via the epsilon transition) or can be required to make a decision (as two transitions with the same input symbol can exist in a single state).

Usually, design is first done in NFA, then it is converted to DFA for less redundancy and better implementation.

We chose the NFA to DFA convertor as it was the first topic, tackled in the first couple of lectures, catching our attention, making us eager to learn more about the course and about this topic in specific. We also found this a great chance to learn more about it, as it meant more research and implementing what we have learned in the first part of the semester.

IV. Project Objective

Take an NFA, formally defined, as an input from the user and using the implemented algorithm, compute the epsilon closure, and find the equivalent DFA.

Moreover, we added an extra functionality to the code, that is, instead of taking the input from the user in the console and demand some restrictions in the input, we added a GUI that eases taking the input from the user in an organized manner. The output is also displayed in the GUI as two graphs, one for the NFA and another for the DFA.

V. Used Language

The main algorithm is implemented in Python.

VI. Approach

```
from graphviz import Digraph
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from tkinter import *
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

- Imports are placed at the top of the file:
    - `graphviz` is used to is used to visualize the NFA and DFA diagrams
    - `matplotlib` is used to place the diagrams in a pdf to be displayed the output.
    - `tkinter` is used for the graphical user interface, used to take input and display output to the user.

```
# Global vars to take input
statesString = ''
startStateString = ''
alphabetString = ''
finalStatesString = ''
deltaString = ''
```

- The global variables are used in order to store the values entered by the user from the GUI. We declare a variable for each element of the NFA's formal definition (Q, **Σ**, δ, q0, F):
    - `statesString`: all states of the NFA
    - `startStateString`: initial state of the NFA
    - `alphabetString`: alphabet that allows transitions between states.
    - `finalStatesString`: all final states (if there is more than one final state)
    - `deltaString`: the transition table of the NFA provided in a certain form.
        - (A,0,B) – state A goes to state B with input 0.

```
# Creating the NFA Class that will include the NFA Transition Table
class NFA:
    # Constructor of the class to identify the quintuple of the NFA
    def __init__(self, numOfStates, qStates, numOfAlphabet, alphabets, startState,
                 numOfFinalStates, finalStates, numOfAllTransitions, allTransitions):
        self.numOfStates = numOfStates
        self.qStates = qStates
        self.numOfAlphabet = numOfAlphabet
        self.alphabets = alphabets

        # Adding epsilon alphabet to the list
        # and incrementing the alphabet count
        self.alphabets.append('e')
        self.numOfAlphabet += 1
        # Identifying the start state
        self.startState = startState
        # Identifying the number of final states.
        self.numOfFinalStates = numOfFinalStates
        # Identifying the final transition states.
        self.finalStates = finalStates
        # Identifying the number of transitions
        self.numOfAllTransitions = numOfAllTransitions
        # Identifying the transition symbols
        self.allTransitions = allTransitions
        self.graph = Digraph()

        # The following dictionary is used to store the indexes of the states
        self.statesDict = dict()
        for i in range(self.numOfStates):
            self.statesDict[self.qStates[i]] = i
        # The following dictionary is used to store the indexes of the alphabets
        self.alphabetsDict = dict()
        for i in range(self.numOfAlphabet):
            self.alphabetsDict[self.alphabets[i]] = i

        # The following dictionary is used for creating the transition table in the followig format:
        # [From State + Alphabet pair] -> [Set of To States]
        self.transitionsTable = dict()
        # This for loop gets the first part of the format: [From State + Alphabet pair]
        for i in range(self.numOfStates):
            for j in range(self.numOfAlphabet):
                self.transitionsTable[str(i) + str(j)] = []
        # This for loop gets the first part of the format: [Set of To States]
```

```
        for i in range(self.numOfAllTransitions):
            self.transitionsTable[str(self.statesDict[self.allTransitions[i][0]])
                                  + str(self.alphabetsDict[
                                          self.allTransitions[i][1]])].append(
                self.statesDict[self.allTransitions[i][2]])

    # Function that prints the NFA quintuple to the user in the following form:
    # Q: states
    # Σ: alphabets
    # q0: Start state
    # F: Final State
    # δ: transition table
    def __repr__(self):
        return "Q : " + str(self.qStates) + "\nΣ : "
        + str(self.alphabets) + "\nq0 : "
        + str(self.startState) + "\nF : " + str(self.finalStates) + \
        "\nδ : \n" + str(self.transitionsTable)

    # Function to perform epsilon closure of a state in the NFA that has epsilon as a transition symbol.
    def getEpsilonClosure(self, state):
        # This dictionary will be used to store all the visited states to avoid repitions.
        closure = dict()
        closure[self.statesDict[state]] = 0
        # This stack is used to get the next state to be visited
        closureStack = [self.statesDict[state]]

        # Check the capacity of the stack, if not empty execute the code inside.
        while len(closureStack) > 0:
            # Store the top of the stack in a variable to be examined.
            # The top of the stack represents the current visited state.
            cur = closureStack.pop(0)
            # Loop on the states that have epsilon as an input symbol
            for x in self.transitionsTable[str(cur) + str(self.alphabetsDict['e'])]:
                # Check if this state is not already visited before
                if x not in closure.keys():
                    # add the state to the dictionary
                    closure[x] = 0
                    # push the current state to the stack
                    closureStack.append(x)
            closure[cur] = 1
        return closure.keys()

    # A Function to return the name from set of states to display in the DFA diagram
    def getStateName(self, state_list):
        name = ''
        for x in state_list:
            name += self.qStates[x]
        return name

    # Function that identifies the final states in the DFA Diagram according to the number
    # of final states in NFA.
    def isFinalDFA(self, state_list):
        for x in state_list:
            for y in self.finalStates:
                if x == self.statesDict[y]:
                    return True
        return False
```

- Create a class to represent the NFA in total
    - Define its constructor, used to initialize the NFA given its states and their number, its alphabet and their number, the start state, the final states and their number, the transitions to construct delta, and their number. This makes up the formal definition of the NFA.
    - Add epsilon to the given list of alphabet and increment their number by one.
    - Identify the values of the quintuple of the NFA.
    - Store in the dictionary `states_dict` the index of the state and in the dictionary `alphabets_dict` the index of the alphabets.
        - This data structure is specifically chosen because it has a special functionality that is not provided in normal arrays and lists as it stores elements with a key that acts as a reference to each element inside the dictionary.
        - [state name] = state index
        - [alphabet symbol] = alphabet index
    - Create a dictionary, `transition_table`, to ease the search in the transition table, by making its key, the current state and the input symbol, and the next state

corresponding to the value stored. This dictionary will be the data structure that will allow us to easily deal with each incoming transition, filled with numbers only (index of state and alphabet rather than the actual symbol or letter).
    - [current state + input symbol] = [set of states]
- Implement a function, `__repr__()`, to represent the NFA formally by printing its formal definition in order (Q, **Σ**, δ, q0, F).
- Implement a function to compute the epsilon closure of each state, `getEpsilonClosure()`.
    - This is done by finding the states to which one can travel to, with no input symbols, only using epsilon.
    - The minimum epsilon closure is the state at hand itself.
    - For the transition (A, ε, B), the epsilon closure of A is A and B.
    - A dictionary is created to hold the epsilon closure for each state (ensures that a state is visited exactly once), and a stack is used to
- Implement `getStateName()` to create a string with the name used in the diagram for the DFA.
- Implement `isFinalDFA()` to find whether a state is final in the converted DFA or not.

```
def convert():
    print("Start Conversion")
    global statesString, startStateString, deltaString, alphabetString, finalStatesString

    # mapping the strings to the content of the input from the GUI
    statesString = setOfStatesInput.get()
    startStateString = startStateInput.get()
    alphabetString = alphabetInput.get()
    finalStatesString = finalStatesInput.get()
    deltaString = deltaInput.get()

    # transform strings into arrays
    allStates = statesString.split(',')
    qNode = startStateString
    inputSymbols = alphabetString.split(',')
    qFinal = finalStatesString.split(',')
    transitionTable = [list(line.split(',')) for line in deltaString.split('|')]

    nfa = NFA(
        len(allStates),  # number of states
        allStates,  # array of states
        len(inputSymbols),  # number of alphabets
        inputSymbols,  # array of alphabets
        qNode,  # start state
        len(qFinal),  # number of final states
        qFinal,  # array of final states
        len(transitionTable),  # number of transitions
        transitionTable,
        # array of transitions with its element of type :
        # [from state, alphabet, to state]
    )
    # Making an object of Digraph to visualize NFA diagram
    nfa.graph = Digraph()

    # Creating the States in the NFA Diagram
    for x in nfa.qStates:
        # This if condition is used for displaying the circles in the diagram:
        # if the state is not a final state --> display it with a single circle.
        if x not in nfa.finalStates:
            nfa.graph.attr('node', shape='circle')
            nfa.graph.node(x)
        # if the state is a final state --> display it with double circles.
        else:
            nfa.graph.attr('node', shape='doublecircle')
            nfa.graph.node(x)
```

```
    # Creating the pointing arrow to the start state in NFA Diagram
    nfa.graph.attr('node', shape='none')
    nfa.graph.node('')
    nfa.graph.edge('', nfa.startState)

    # Create the edges between the states from the transitions array.
    for x in nfa.allTransitions:
        nfa.graph.edge(x[0], x[2], label=('ε', x[1])[x[1] != 'e'])

    # Generates a pdf that visualizes the NFA Diagram
    nfa.graph.render('nfa', view=False, format="png")

    # Making an object of Digraph to visualize DFA diagram
    dfa = Digraph()

    # Calling the function that performs the epsilon closure.
    epsilon_closure = dict()
    for x in nfa.qStates:
        epsilon_closure[x] = list(nfa.getEpsilonClosure(x))

    # The epsilon closure of the start state of the NFA will be the first state in the DFA.
    dfaSavedStack = list()
    # The previous list is created to store the states until the current is finished converting from NFA to DFA.
    dfaSavedStack.append(epsilon_closure[nfa.startState])

    # Condition that checks if the start state is also a final state.
    if nfa.isFinalDFA(dfaSavedStack[0]):
        dfa.attr('node', shape='doublecircle')
    else:
        dfa.attr('node', shape='circle')
    dfa.node(nfa.getStateName(dfaSavedStack[0]))

    # Identify the DFA start state by drawing an arrow
    dfa.attr('node', shape='none')
    dfa.node('')
    dfa.edge('', nfa.getStateName(dfaSavedStack[0]))

    # Create a list to keep track of DFA states
    dfaStates = list()
    dfaStates.append(epsilon_closure[nfa.startState])
    # The conversion will keep running until the array that stores the DFA states is empty
    while len(dfaSavedStack) > 0:
        # Popping the stack state after the other and storing it into a variable to
        # start making the conversion for that state.
        currentState = dfaSavedStack.pop(0)

        # Loop over all the alphabets for the states stored in the currentState variable
        # to get the corresponding transitions of those states in the DFA
        for al in range(nfa.numOfAlphabet - 1):
            # A set to check if the epsilon closure of the current state is empty or not
            closureOutput = set()
            for x in currentState:
                # Perform union operation between the states found in the epsilon closure of the current state.
                closureOutput.update(
                    set(nfa.transitionsTable[str(x) + str(al)]))

            # Condition to ensure that the epsilon closure of the new set is not empty.
            if len(closureOutput) > 0:
                # Create a set to store the states the current state will be transitioned to.
                toState = set()
                for x in list(closureOutput):
                    toState.update(set(epsilon_closure[nfa.qStates[x]]))

                # Check if the to state already exists in DFA and if not then add it
                if list(toState) not in dfaStates:
                    dfaSavedStack.append(list(toState))
                    dfaStates.append(list(toState))

                    # This consition checks if this state is a final state or not
                    # in order to identify if it should be surrounded by double circles or
                    # a single circle.
                    if nfa.isFinalDFA(list(toState)):
                        dfa.attr('node', shape='doublecircle')
                    else:
                        dfa.attr('node', shape='circle')
                    dfa.node(nfa.getStateName(list(toState)))

                # Draw an edge from the current state to the corresponding to state.
                dfa.edge(nfa.getStateName(currentState),
                         nfa.getStateName(list(toState)),
                         label=nfa.alphabets[al])
```

```
        # Else, the current state has an empty epsilon closure, then it
        # will be represented as a phi (ϕ) state.
        else:

            # Condition to make sure there weren't any dead states present
            # before this one.
            # 1- if there wasn't any phi state beofre then we create a new one
            if (-1) not in dfaStates:
                dfa.attr('node', shape='circle')
                dfa.node('ϕ')

                # The phi state will have all transitions looping on itself.
                for alpha in range(nfa.numOfAlphabet - 1):
                    dfa.edge('ϕ', 'ϕ', nfa.alphabets[alpha])

                # Adding -1 to list to mark that dead state is present
                dfaStates.append(-1)

            # 2- Else, we add this current state to the phi state.
            dfa.edge(nfa.getStateName(currentState, ),
                     'ϕ', label=nfa.alphabets[al])

    # Generates a pdf and opens the DFA diagram.
    dfa.render('dfa', view=False, format="png")
    # make new figure with 2 sub-figures
    # each sub-figure can have an image in it
    fig = plt.figure()
    image1 = plt.subplot(121)
    image2 = plt.subplot(122)

    # read the image files (png files preferred)
    img_source1 = mpimg.imread('nfa.png')
    img_source2 = mpimg.imread('dfa.png')
    # put the images into the window
    _ = image1.imshow(img_source1)
    _ = image2.imshow(img_source2)

    # hide axis and show window with images
    image1.axis("off")
    image2.axis("off")
    plt.show()
```

- A function is made for the conversion from NFA to DFA.
    - Note: before the conversion, the NFA diagram is constructed and generated in pdf format, while the DFA diagram is made post-conversion.
    - Steps of the conversion algorithm:
        1. Create an object from the class implementing the NFA, initialized with the values entered by the user.
        2. Construct the NFA graph and display it in pdf format.
        3. Create another graph, currently empty, to be filled later with the DFA.
        4. Find the epsilon closure of each single state in the NFA and set the DFA's start state as the first epsilon closure value.
        5. To evaluate each DFA state, store them in a stack.
        6. To later graph the DFA, store all DFA states in an array.
        7. Loop over all states in the stack, removing the last added state to be evaluated as the current state.
        8. For each state, loop over the input symbols to study the next state.
        9. Compute once again the epsilon closer of the current state, and check if it is not empty:
            a. Create another set for the next state.
            b. Add it to the stack and the final DFA states only if it does not already exist there.
            c. Add a node and its corresponding edge to the graph.
        10. Else, it is empty, labeled as phi
            a. Add it to the final DFA states only if it does not already exist there.

b. Add a node and an edge from the current state leading to phi in the graph.
c. All transitions from phi lead to itself.

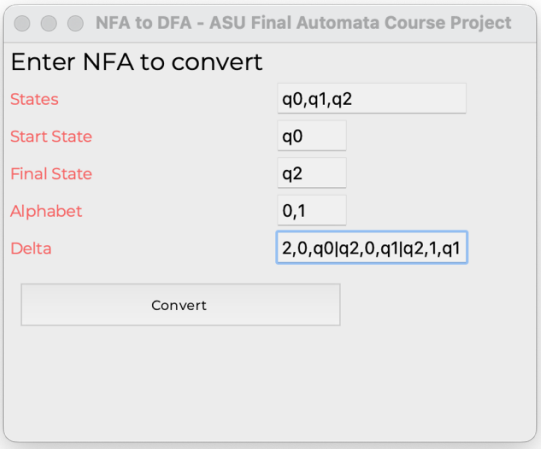
11. Construct the graphs.

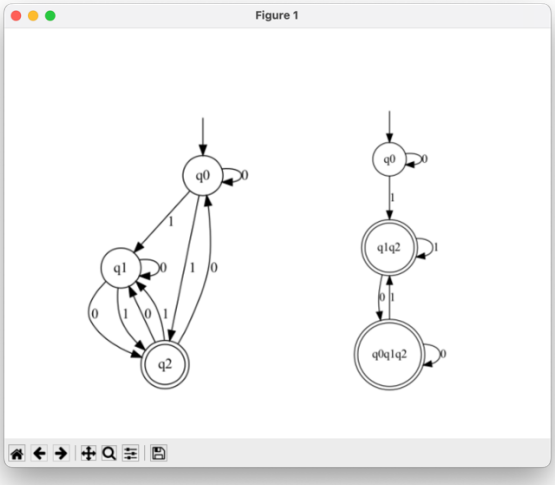
```
# GUI
root = Tk()
root.geometry(str(400) + "x" + str(300))
root.title("NFA to DFA - ASU Final Automata Course Project")

Label(root, text="Enter NFA to convert", font=("Montserrat", 18), fg='#000000').grid(column=1, row=1, padx=2,
                                                                                     sticky="w")

Label(root, text="States", font=("Montserrat", 12), fg='#f66666').grid(column=1, row=2, padx=2, sticky="w")
setOfStatesInput = Entry(root, width=15, justify="left", bg='#f0f0f0')
setOfStatesInput.grid(column=2, row=2, padx=2, sticky="w")

Label(root, text="Start State", font=("Montserrat", 12), fg='#f66666').grid(column=1, row=4, padx=2, sticky="w")
startStateInput = Entry(root, width=5, justify="left", bg='#f0f0f0')
startStateInput.grid(column=2, row=4, padx=2, sticky="w")

Label(root, text="Final State", font=("Montserrat", 12), fg='#f66666').grid(column=1, row=6, padx=2, sticky="w")
finalStatesInput = Entry(root, width=5, justify="left", bg='#f0f0f0')
finalStatesInput.grid(column=2, row=6, padx=2, sticky="w")

Label(root, text="Alphabet", font=("Montserrat", 12), fg='#f66666').grid(column=1, row=8, padx=2, sticky="w")
alphabetInput = Entry(root, width=5, justify="left", bg='#f0f0f0')
alphabetInput.grid(column=2, row=8, padx=2, sticky="w")

Label(root, text="Delta", font=("Montserrat", 12), fg='#f66666').grid(column=1, row=10, padx=2, sticky="w")
deltaInput = Entry(root, width=15, justify="left", bg='#f0f0f0')
deltaInput.grid(column=2, row=10, padx=2, sticky="w")

convertBtn = Button(root, text="Convert", width=30, height=2, font=("Montserrat", 10),
                    command=lambda: convert())
convertBtn.grid(column=1, row=12, columnspan=3, sticky="w", padx=10, pady=10)

root.mainloop()
```

- Build the GUI to take the input from the user.

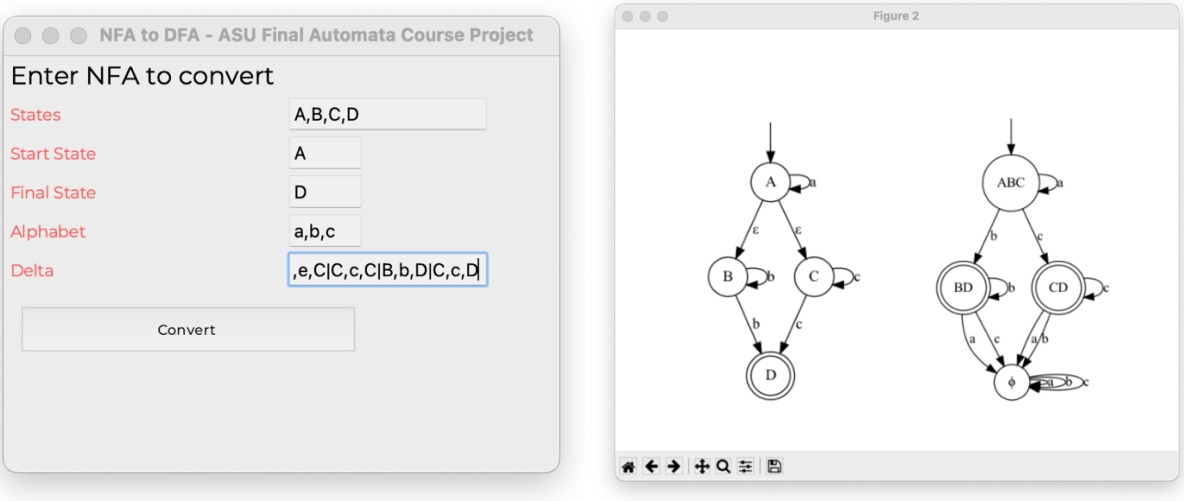
VII. Results and Analysis

Example #1:

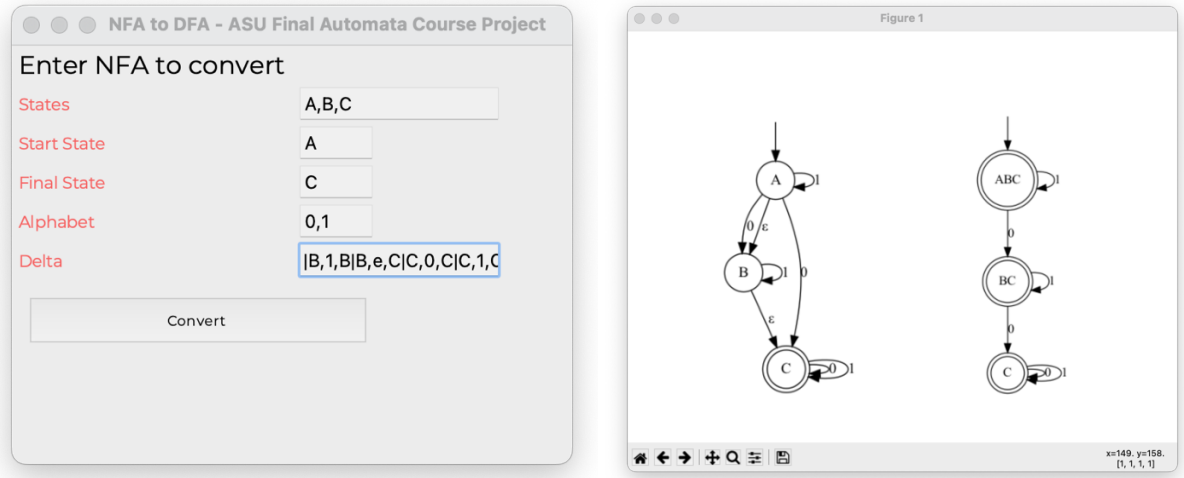




Example #2:



Example #3:



VIII. Conclusion

Through this project, we were able to apply our programming skills in python to implement a topic we studied and were tested on; NFA to DFA conversion, applying the steps learnt at the beginning of the semester, while also learning more about epsilon closure and how to implement it. And finally, we were able to increase our knowledge in GUI design to be able to take input and display output to the user.

This project allowed us to demonstrate the clear steps for this conversion, checking for all special cases, and ensuring unflawed equivalence between the given NFA and the generated DFA, allowing the program to generate what we call the model answer to such a problem.