Topic: Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Course: Formal Languages & Finite Automata

Author: Cretu Dumitru and kudos to the Vasile Drumea with Irina Cojuhari

Theory

A finite automaton, akin to a state machine, represents processes with a defined start and end state, highlighting its deterministic nature. Non-determinism arises when a single transition can lead to multiple states, complicating the predictability of the system. Despite this, automata can be categorized as deterministic or non-deterministic, and through specific algorithms, determinism can be achieved by modifying the automaton's structure. This will be fursther explored in this laboratory work

Objectives:

```
a. Implement conversion of a finite automaton to a regular grammar.

b. Determine whether your FA is deterministic or non-deterministic.

c. Implement some functionality that would convert an NDFA to a DFA.

d. Represent the finite automaton graphically (Optional, and can be considered as a __*bonus point*__):

- You can use external libraries, tools or APIs to generate the figures/diagrams.

- Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.
```

Implementation description

I have created the following class to denote the language formed from the given grammar. it contains terminal, and nonterminal symbols as well as the rules through which words can be formed.

```
class Language():
    rules = {}
    words = []
    VN=[]
    VT=[]
    F=[]
    nfa=False
    def __init__(self,rules,VN,VT,F):
        self.rules=rules
        self.VN=VN
        self.VT=VT
        self.F=F
```

In order to keep a clean class structure I have created a function that given a grammar input(Ex below), would extract all of the symbols and the necessary rules.

```
 import collections

def grammar_to_language(grammar):
    terminals="abcdefghijklmnoprstqwxyz"
    nonterminals=terminals.upper()
    lines=grammar.split("\n")
    VN = lines[1].split("=")[1].translate({ord(c): None for c in "{} "}).split(",")[:-1]
    VT = lines[2].split("=")[1].translate({ord(c): None for c in "{} "}).split(",")[:-1]
    F = VT
    is_fa_grammar=0
    if "F =" in lines[3]:
        is_fa_grammar=1
        for i in range(3, len(lines)):
            line = lines[i]
            if not line:
                continue

            line = line.replace(".","").replace(" ","")
            line = line[:-1] if line[-1]=="," else line


            for char in VT:
                ind=VT.index(char)
                line = line.replace(VT[ind],terminals[ind])
            for char in VN:
                ind=VN.index(char)
                line = line.replace(VN[ind],nonterminals[ind])


            lines[i] = line
        for char in VT:
            ind=VT.index(char)
            VT[ind]=terminals[ind]
        for char in VN:
            ind=VN.index(char)
            VN[ind]=nonterminals[ind]

        if "F=" in lines[3]:
            F_part = lines[3].split("=")[1].strip().replace("{","").replace("}","").split(",")  # Extract the part after "F ="

            F = [char.strip() for char in F_part]

    grammar_rules = lines[3+is_fa_grammar:]
    rules = collections.defaultdict(list)

    for rule in grammar_rules:
        rule = rule.replace("→","=")
        if rule and "{" not in rule and "}" not in rule:
            lhs, rhs = rule.split("=")
            if is_fa_grammar:
                rhs=lhs.split(",")[1].replace(")","")+rhs
                lhs=lhs.split(",")[0].replace("δ(","")
            rules[lhs.strip()].append(rhs.strip())

    return rules,VN,VT,F
```

Yes the use of the collections library is not necessary, I just like the defaultdict data structure

Following that I have created a finite automata class:

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

from collections import defaultdict

class FiniteAutomaton:
    def __init__(self, grammar, F, initial="S"):
        self.grammar = grammar
        self.transitions = defaultdict(list)
        self.rules=defaultdict(dict)
        self.start_symbol = initial
        self.F=F
        self.convert()
        self._build_transitions()

    def _build_transitions(self):
        for non_terminal, productions in self.rules.items():
            # print(non_terminal,productions)
            for production in productions:
                symbol = productions[production]
                self.transitions[non_terminal].append((production,symbol))



    def convert(self):
        for state,transition in self.grammar.items():
            dic={}
            if type(transition)==list:
                for choice in transition:
                    print(choice)
                    upper = "".join([c for c in choice if c.isupper()])
                    if not upper:
                        dic[choice]=[choice]
                        continue
                    if choice.replace(upper,"") not in dic:
                        dic[choice.replace(upper,"")]=[upper]
                        continue

                    dic[choice.replace(upper,"")].append(upper)
                self.rules[state]=dic
            else:
                self.rules=self.grammar
```

Here, based on the previous grammar helper method that parses the given grammar string into a dictionary of rules a finite automata class has been defined. The additionalconvert method allows for the translation between the project structure used in the first laboratory to the second one, therefore the code as of this moment is perfectly compatible with old versions.

Following that we have the nfa to dfa detemrminer and convertor:

```
def which_type(lang):
    for lhs in lang.rules.keys():
        lhs_lower=[]
        for rhs in lang.rules[lhs]:
            lower=[c for c in rhs if c.islower()]
            lhs_lower.append(''.join(lower))
        if len(lhs_lower)!=len(set(lhs_lower)):
            lang.nfa=True

    if lang.nfa:
        print("This automata is a NFA!")
    else:
        print("This automata is a DFA!")
```

this checks if there is any ambiguity in the available choices and prints that the automata is an NFA if so.

Then there is the actual convertor and its helper function:

```
def convert_dfa(dfa):
    converted_dfa = {}
    for state, transitions in dfa.items():
        converted_transitions = []
        for symbol, next_state in transitions.items():
            converted_transitions.append(symbol + next_state)
        converted_dfa[state] = converted_transitions
    return converted_dfa

def convert_nfa_to_dfa(lang):
    dic = {}

    for lhs, rhs in lang.rules.items():
        for choice in rhs:
            upper = ""
            for c in choice:
                upper += c if c.isupper() else ""
            lower = ""
            for c in upper:
                lower += choice.replace(c, "")

            if lhs not in dic:
                dic[lhs] = {}
            if lower in dic[lhs]:
                dic[lhs][lower] += upper
                continue
            dic[lhs][lower] = upper
    dfa={}
    queue = [{'A'}]  # Start with the initial state
    visited = set()

    while queue:
        current_states = queue.pop(0)
        current_states_str = ''.join(sorted(current_states))
        if current_states_str in visited:
            continue
        visited.add(current_states_str)
        dfa[current_states_str] = {}
        dfa_state = {}

        for state in current_states:
            if state in dic:
                for symbol, next_state in dic[state].items():
                    if symbol not in dfa_state:
                        dfa_state[symbol] = next_state
                    else:
                        dfa_state[symbol] += next_state

        for symbol, next_states in dfa_state.items():
            next_state_set = ''.join(sorted(set(next_states)))
            dfa[current_states_str][symbol] = next_state_set
            if next_state_set not in visited:
                queue.append(next_state_set)


    formatted_dfa = convert_dfa(dfa)
    return dfa
```

The convert_Dfa function just helps with formating the actual state of the graph representation, and the convert_nfa_to_dfa function does all the actual converting to a DFA

Next we have the get_automata function that generates a graph using networkx for the given finiteAutomata. the red vertexes represent terminal states, and the skyblue ones are Non terminal. The respective states are connected to each other via transitions

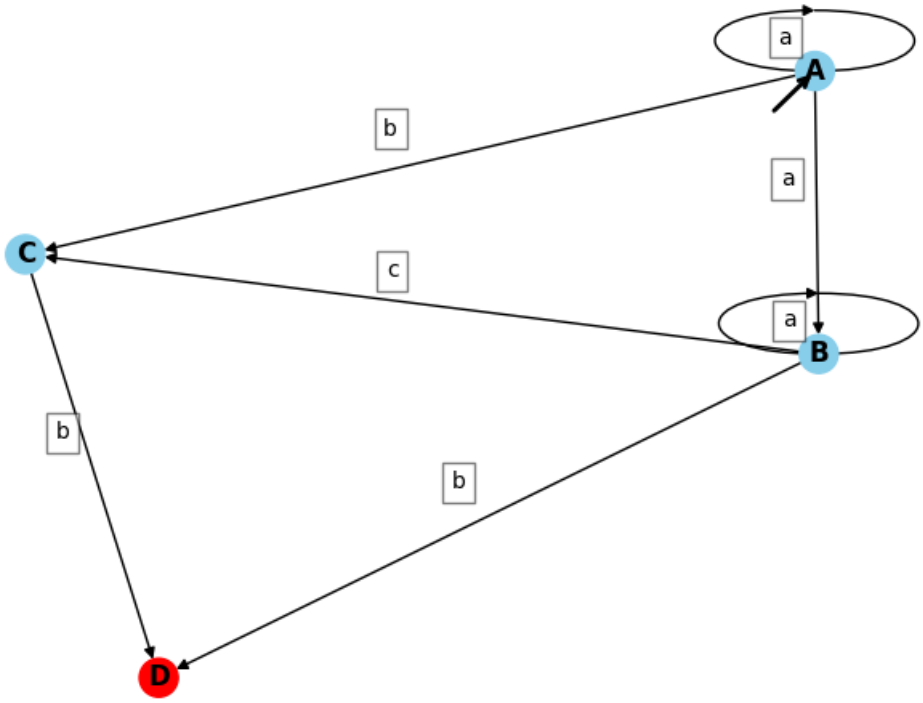
```
def getAutomata(self,title):
        initial=self.start_symbol
        G = nx.DiGraph()
        edge_list = []
        node_colors = {}

        for node, edges in self.transitions.items():
            print(node,edges)
            if any(final in node for final in self.F):
                node_colors[node]="red"
            else:
                node_colors[node]="skyblue"

            for edge in edges:
                if type(edge[1]) == str:
                    G.add_edge(node, edge[1], label=edge[0])
                    edge_list.append(edge[0])
                    if any(final in edge[1] for final in self.F):
                        node_colors[edge[1]]="red"
                    else:
                        node_colors[edge[1]]="skyblue"
                    continue

                for choice in edge[1]:
                    G.add_edge(node, choice, label=edge[0])
                    edge_list.append(edge[0])

                    if any(final in choice for final in self.F):
                        node_colors[choice]="red"
                    else:
                        node_colors[choice]="skyblue"


        node_colors = node_colors.values()
        pos = nx.spring_layout(G)

        nx.draw(G, pos, with_labels=True, node_color=node_colors, font_size=12, font_weight="bold")
        for i,edge in enumerate(G.edges()):
            x0, y0 = pos[edge[0]]
            x1, y1 = pos[edge[1]]
            x, y = (x0 + x1) / 2, (y0 + y1) / 2

            x -= 0.02
            y += 0.08


            plt.text(x, y, edge_list[i], bbox=dict(facecolor='white', alpha=0.5), horizontalalignment='center')
        for i,node in enumerate(G.nodes()):
            if initial in node:
                x, y = pos[node]
                plt.annotate("", xy=(x, y), xytext=(x-20, y-20),
                 textcoords="offset points", ha="center", va="center",
                 arrowprops=dict(arrowstyle="->", linewidth=2))


        man = plt.get_current_fig_manager()
        man.set_window_title(title)
        plt.show()
```

You might notice the extensive used of the separate matplotlib functions, that is due to the limitations of networkx in regards to the placement of edge labels in relation to the actual edges.

This is an example usage:

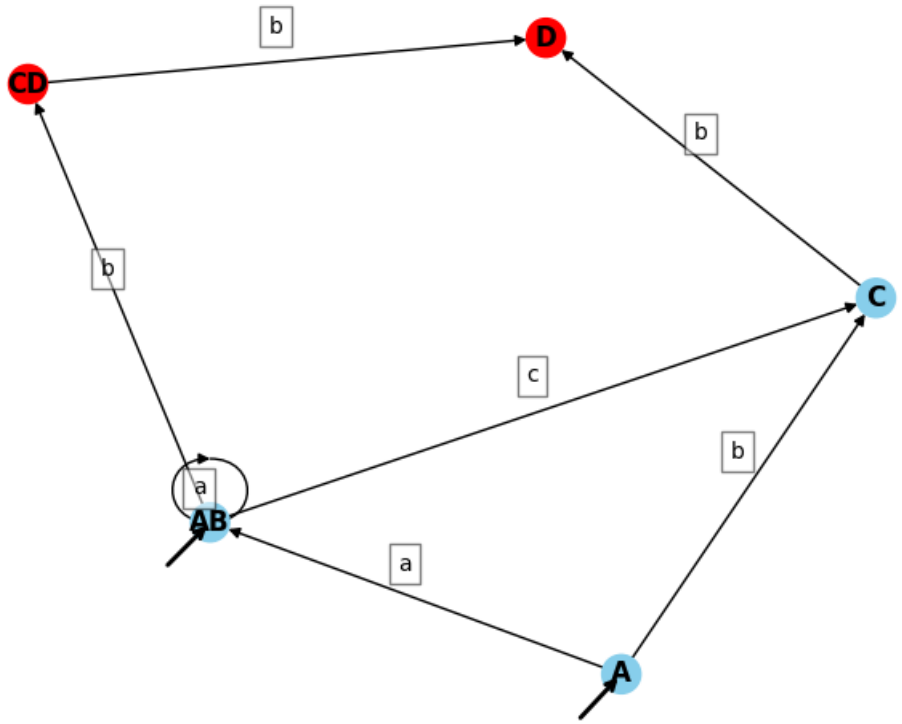
```
condition="""Variant 28
Q = {q0,q1,q2,q3},
Σ = {a,b,c},
F = {q3},
δ(q0,a) = q0,
δ(q0,a) = q1,
δ(q1,a) = q1,
δ(q1,c) = q2,
δ(q1,b) = q3,
δ(q0,b) = q2,
δ(q2,b) = q3.
"""
rules, VN, VT, F = grammarHelper.grammar_to_language(condition)



lang=language.Language(rules,VN,VT,F)

fa= FiniteAutomaton(rules,F=lang.F,initial='A')
fa.getAutomata("FA for language(NFA/DFA)")
dfa_rules=nfa_dfa.convert_nfa_to_dfa(lang)
dfa = FiniteAutomaton(dfa_rules,F=lang.F,initial='A')
dfa.getAutomata("FA for language, DFA")
```

This is the graph output of the program:



In the figure above we have the pure untouched graph of the automata described in the condition

```
condition="""Variant 28
Q = {q0,q1,q2,q3},
∑ = {a,b,c},
F = {q3},
δ(q0,a) = q0,
δ(q0,a) = q1,
δ(q1,a) = q1,
δ(q1,c) = q2,
δ(q1,b) = q3,
δ(q0,b) = q2,
δ(q2,b) = q3.
"""
```



And in this picture is illustrated the DFA-Converted FA, after all the computations.

As for the cpnsole output, besides the word generation and checking if an word can be generated via the given FA we have the NFA/DFA Decider this is its result:

```
This automata is a NFA!
```

The input automata describes an NFA

Now let us discuss the chomsky classification of grammars. Here is my implementation of the classification method in the FiniteAutomata Class

```
    def clasify(self):
        global_tier=3
        for non_terminal, productions in self.grammar.items():

            for production in productions:
                tier=3

                elements = [char for choice in production for char in choice]

                terminals = [x for x in elements if x.islower()]

                non_terminals = [x for x in elements if x.isupper()]


                elements_lhs = [char for choice in non_terminal for char in choice]
                non_terminals_lhs = [x for x in elements_lhs if x.isupper()]

                tier = 2 if (len(non_terminals) > 1 or len(terminals) > 1) else tier
                tier = 1 if len(non_terminals_lhs)>1 else tier
                tier = 0 if non_terminals_lhs == [] else tier

                global_tier=min(global_tier,tier)
        return global_tier
```

The code was developed in great part using this as basis:

Type	Grammar	Production rules
Type 0	unrestricted	$\alpha \to \beta$
Type 1	context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
Type 2	**context-free**	$A \to \gamma$
Type 3	regular	$A \to aB$ or $A \to Ba$

in order to make sure that this properly works, I have set up the following testing suite:

```
 import automata
import grammarHelper
tests = [
    {
        "grammar": """Variant 21:
        VN={S, B, C, D},
        VT={a, b, c},
        P={
            S → aB
            B → bS
            B → aC
            B → b
            C → bD
            D → a
            D → bC
            D → cS
        }

        """,
        "expected_tier": 3  # Expected tier level for this grammar
    },
    {
        "grammar": """Variant 21:
        VN={S, A, B},
            VT={a, b},
            P={
                S → aS
                S → A
                A → bB
```

```
            B → a
            B → b
        }
    """,
    "expected_tier": 3  # Expected tier level for this grammar
},
{
    "grammar": """Variant 21:
    VN={S, A, B, C},
        VT={a, b},
        P={
            S → aA
            A → bB
            A → B
            B → aC
            C → b
        }
    """,
    "expected_tier": 3  # Expected tier level for this grammar
},
{
    "grammar": """Variant 21:
    VN={X, Y},
        VT={e, a, b},
        P={
            X → e
            X → a
            X → aY
            Y → b
        }
    """,
    "expected_tier": 3  # Expected tier level for this grammar
},
{
    "grammar": """Variant 21:
    VN={S, X},
        VT={e, a, b, c},
        P={
            S → Xa
            X → a
            X → aX
            X → abc
            X → e
        }
    """,
    "expected_tier": 2  # Expected tier level for this grammar
},{
    "grammar": """Variant 21:
    VN={A, B},
        VT={a, b, c},
        P={
            AB → AbBc
            A → bcA
            B → b
        }
    """,
    "expected_tier": 1  # Expected tier level for this grammar
},{
    "grammar": """Variant 21:
    VN={A, B},
        VT={a, b, c},
        P={
            S → ACaB
            Bc → acB
            CB → DB
```

```
                aD → Db
            }
        """,
        "expected_tier": 1  # Expected tier level for this grammar
    },

    ]

for test in tests:
    grammar = test["grammar"].strip()
    expected_tier = test["expected_tier"]
    rules, VN, VT, F = grammarHelper.grammar_to_language(grammar)
    fa = automata.FiniteAutomaton(rules, F)
    actual_tier = fa.clasify()
    result= actual_tier == expected_tier
    if not result:
        print(f"XXXX - Test failed: Expected tier {expected_tier}, got {actual_tier}")
    else :
        print(f"       Test completed: Expected tier {expected_tier}, got {actual_tier}")
```

And this is the output:

```
       Test completed: Expected tier 3, got 3
      Test completed: Expected tier 3, got 3
      Test completed: Expected tier 3, got 3
      Test completed: Expected tier 3, got 3
      Test completed: Expected tier 2, got 2
      Test completed: Expected tier 1, got 1
      Test completed: Expected tier 1, got 1
```

Since all tests randomly found on the internet have succeded I dare say that the condition has been achieved and My algorithm can Classify grammars using the Chomsky classification

Conclusions

In the end we have managed to determine if a Finite Automata is Deterministic or Nondeterministic and Classify it according to Chomsky's rules. Following that We have developed an algorithm to transform an NFA into a DFA, by getting rid of all ambiguities. These processes have been illustrated above via pictures and examples.

References

Documentation of networkx again Difference between Type 0 and Type 1 in the Chomsky Hierarchy