#sudo pip install graphviz

#sudo apt install python-pydot python-pydot-ng graphviz #if first instalation doent work

from display\_automaton import export\_automaton

# A tree-based representation of regular expressions

class RegEx:

def \_\_init\_\_(self, symbol, children = []):

self.symbol = symbol

self.children = children

def to\_string(self):

if self.symbol == "\*":

return "(" + self.children[0].to\_string() + ")\*"

elif self.symbol == "+":

return self.children[0].to\_string() + "+" \

+ self.children[1].to\_string()

elif self.symbol == ".":

return self.children[0].to\_string() + self.children[1].to\_string()

else:

return self.symbol

# Question 4

# Output an epsilon-NFA equivalent to the regular expression

def to\_enfa(self):

enfa = ENFA([0, 1], [0], [1], [], [])

enfa.convert\_reg\_ex(0, 1, self)

return enfa

# Non-deterministic finite automata with epsilon transitions

class ENFA:

def \_\_init\_\_(self, all\_states, initial\_states, final\_states,

alphabet, edges):

# States: a set of integers

self.all\_states = set(all\_states)

# The alphabet: a set of strings

# "" stands for epsilon

self.alphabet = set(alphabet)

self.alphabet.add("")

# Initial and final states: two sets of integers

self.initial\_states = set(initial\_states).intersection(self.all\_states)

self.final\_states = set(final\_states).intersection(self.all\_states)

# There must be an initial state; if there isn't, an initial state 0

# is added

if not self.initial\_states:

self.initial\_states.add(0)

self.all\_states.add(0)

# Edges: a dictionnary (origin, letter): set of destinations

self.next\_states = {(state, letter): set()

for state in self.all\_states

for letter in self.alphabet}

for edge in set(edges):

if (edge[0] in self.all\_states) and (edge[2] in self.all\_states) \

and (edge[1] in self.alphabet):

self.next\_states[(edge[0], edge[1])].add(edge[2])

# Question 1

# Add a new state to the automaton

def new\_state(self):

new\_state = max(self.all\_states) + 1

self.all\_states.add(new\_state)

for letter in self.alphabet:

self.next\_states[(new\_state, letter)] = set()

return new\_state

# Question 2

# Add a new letter 'letter' to the automaton

def new\_letter(self, letter):

if not letter in self.alphabet:

for state in self.all\_states:

self.next\_states[state, letter] = set()

self.alphabet.add(letter)

# Question 3

# Insert the automaton matched to the regular expression 'reg\_ex'

# between the two states 'origin' and 'destination' according to

# Thompson's algorithm

def convert\_reg\_ex(self, origin, destination, reg\_ex):

if reg\_ex.symbol == ".":

# 2 new states

mid\_state\_1 = self.new\_state()

mid\_state\_2 = self.new\_state()

# 1 new edge

self.next\_states[(mid\_state\_1, "")].add(mid\_state\_2)

# 2 recursive calls

self.convert\_reg\_ex(origin, mid\_state\_1, reg\_ex.children[0])

self.convert\_reg\_ex(mid\_state\_2, destination, reg\_ex.children[1])

elif reg\_ex.symbol == "+":

# 4 new states

high\_state\_1 = self.new\_state()

high\_state\_2 = self.new\_state()

low\_state\_1 = self.new\_state()

low\_state\_2 = self.new\_state()

# 4 new edges

self.next\_states[(origin, "")].add(high\_state\_1)

self.next\_states[(high\_state\_2, "")].add(destination)

self.next\_states[(origin, "")].add(low\_state\_1)

self.next\_states[(low\_state\_2, "")].add(destination)

# 2 recursive calls

self.convert\_reg\_ex(high\_state\_1, high\_state\_2, reg\_ex.children[0])

self.convert\_reg\_ex(low\_state\_1, low\_state\_2, reg\_ex.children[1])

elif reg\_ex.symbol == "\*":

# 2 new states

mid\_state\_1 = self.new\_state()

mid\_state\_2 = self.new\_state()

# 4 new edges

self.next\_states[(origin, "")].add(mid\_state\_1)

self.next\_states[(mid\_state\_2, "")].add(destination)

self.next\_states[(origin, "")].add(destination)

self.next\_states[(mid\_state\_2, "")].add(mid\_state\_1)

# 1 recursive call

self.convert\_reg\_ex(mid\_state\_1, mid\_state\_2, reg\_ex.children[0])

else:

# 1 new letter

self.new\_letter(reg\_ex.symbol)

# 1 new edge

self.next\_states[(origin, reg\_ex.symbol)].add(destination)

# Question 5

# Returns the epsilon forward closure of a state 'origin'

def epsilon\_reachable(self, origin):

reached\_states = {origin}

old\_states = set()

fixed\_point\_reached = False

# Using a fixed-point iterative computation

while not fixed\_point\_reached:

old\_states = reached\_states.copy()

fixed\_point\_reached = True

for state in old\_states:

for destination in self.next\_states[(state, "")]:

if not destination in old\_states:

fixed\_point\_reached = False

reached\_states.add(destination)

return reached\_states

# Question 6

# Returns a NFA equivalent to the epsilon NFA by performing a backward

# removal of epsilon transitions

def to\_nfa(self):

edges\_nfa = []

new\_final = []

for state in self.all\_states:

for reached in self.epsilon\_reachable(state):

# Computing the new final states

# May add duplicates: we don't care, explicit testing would

# be just as expensive

if reached in self.final\_states:

new\_final.append(state)

for letter in self.alphabet:

for target in self.next\_states[(reached, letter)]:

# Adding the new edges

edges\_nfa.append((state, letter, target))

return NFA(self.all\_states, self.initial\_states, new\_final,

self.alphabet, edges\_nfa)

# Non-deterministic finite automaton

class NFA:

def \_\_init\_\_(self, all\_states, initial\_states, final\_states,

alphabet, edges):

# States: a set of integers

self.all\_states = set(all\_states)

# The alphabet: a set of strings

# "" stands for epsilon

self.alphabet = set(alphabet)

if "" in self.alphabet:

self.alphabet.remove("")

# Initial and final states: two sets of integers

self.initial\_states = set(initial\_states).intersection(self.all\_states)

self.final\_states = set(final\_states).intersection(self.all\_states)

# There must be an initial state; if there isn't, an initial state 0

# is added

if not self.initial\_states:

self.initial\_states.add(0)

self.all\_states.add(0)

# Edges: a dictionnary (origin, letter): set of destinations

self.next\_states = {(state, letter): set()

for state in self.all\_states

for letter in self.alphabet}

for edge in set(edges):

if (edge[0] in self.all\_states) and (edge[2] in self.all\_states) \

and (edge[1] in self.alphabet):

self.next\_states[(edge[0], edge[1])].add(edge[2])

# Testing regular expressions

a = RegEx("a", [])

b = RegEx("b", [])

c = RegEx("c", [])

a\_star = RegEx("\*", [a])

bc = RegEx(".", [b, c])

e = RegEx("+", [a\_star, bc])

print("Expects (a)\*+bc:", e.to\_string())

# Printing the patterns for Thompson's algorithm

E\_star = a\_star.to\_enfa()

export\_automaton(E\_star, "aut\_star")

E\_conc = RegEx(".", [a, b]).to\_enfa()

export\_automaton(E\_conc, "aut\_conc")

E\_plus = RegEx("+", [a, b]).to\_enfa()

export\_automaton(E\_plus, "aut\_plus")

# Testing Thompson's algorithm

E = e.to\_enfa()

export\_automaton(E, "E")

# Testing the ENFA class

A = ENFA([0, 1, 2], [0], [1], ["a", "b"], [(0, "a", 1), (0, "", 1),

(1, "b", 2), (2, "", 0)])

export\_automaton(A, "A")

# A.new\_state()

# export\_automaton(A, "A\_new")

# Testing the epsilon closure

print("Expects {0, 1}:", A.epsilon\_reachable(0))

print("Expects {0, 1, 2}:", A.epsilon\_reachable(2))

# Testing the removal of epsilon transitions

B = A.to\_nfa()

export\_automaton(B, "B")