Instituto Tecnológico y de Estudios Superiores de Monterrey

Jesús Ramírez Delgado

A01274723

Implementacion de Metidos Computacionales

Actividad Integradora 1: Convertir una expresión regular a DFA

La situación para resolver en esta actividad integradora es la siguiente:

1. Desarrolla un programa que permita convertir un expresión regular a un autómata finito determinístico. El programa deberá implementar el algoritmo visto en clase: expresión regular -> autómata finito no determinístico -> autómata finito determinístico.
2. El programa recibe de entrada el alfabeto y la expresión regular. Por ejemplo:

Alphabet: ab

RegEx: (a|b)*abb

3. El programa deberá desplegar tanto en autómata finito no determinístico como el autómata finito determinístico.

----RESULTS----

INPUT:

(a|b)*abb

NFA:

0 => [(1, 'a')]

1 => [(5, '#')]

2 => [(3, 'b')]

3 => [(5, '#')]

4 => [(0, '#'), (2, '#')]

5 => [(4, '#'), (7, '#')]

6 => [(4, '#'), (7, '#')]

7 => [(8, '#')]

8 => [(9, 'a')]

9 => [(12, '#')]

12 => [(13, 'b')]

13 => [(16, '#')]

16 => [(17, 'b')]

Accepting state: 17

DFA:

A => [('B', 'a'), ('C', 'b')]

B => [('B', 'a'), ('D', 'b')]

C => [('B', 'a'), ('C', 'b')]

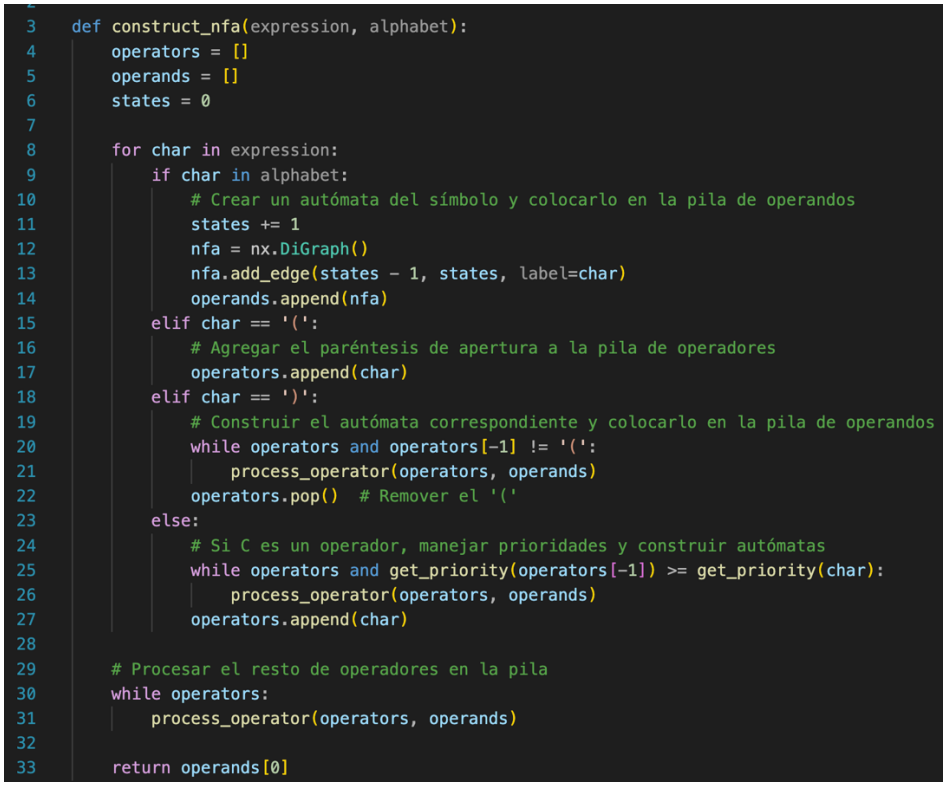
D => [('B', 'a'), ('E', 'b')]

E => [('B', 'a'), ('C', 'b')]

Accepting states: ['E']

El desarrollo de esta situación problema se llevó a cabo en Python. Utilice la librería **networkx** que es utilizada para facilitar el uso de varias estructuras de datos como grafos. En este caso utilizo un grafo de la biblioteca llamado DiGraph que van de un nodo inicial a un nodo final

1. El primer paso en el código fue crear la función **construct_nfa** que va a tomar la expresión regular y **alphabet** que servirán para construir el autómata.



```python
def construct_nfa(expression, alphabet):
    operators = []
    operands = []
    states = 0

    for char in expression:
        if char in alphabet:
            # Crear un autómata del símbolo y colocarlo en la pila de operandos
            states += 1
            nfa = nx.DiGraph()
            nfa.add_edge(states - 1, states, label=char)
            operands.append(nfa)
        elif char == '(':
            # Agregar el paréntesis de apertura a la pila de operadores
            operators.append(char)
        elif char == ')':
            # Construir el autómata correspondiente y colocarlo en la pila de operandos
            while operators and operators[-1] != '(':
                process_operator(operators, operands)
            operators.pop()  # Remover el '('
        else:
            # Si C es un operador, manejar prioridades y construir autómatas
            while operators and get_priority(operators[-1]) >= get_priority(char):
                process_operator(operators, operands)
            operators.append(char)

    # Procesar el resto de operadores en la pila
    while operators:
        process_operator(operators, operands)

    return operands[0]
```

2. El siguiente paso fue declarar la función **get_priority** que está encargada de darle un valor jerárquico a los operadores para la expresión regular.

```
35  def get_priority(operator):
36      priorities = {'*': 3, '.': 2, '|': 1}
37      return priorities.get(operator, 0)
```

3. Ahora se agregó la función **process operator**, esta función toma dos listas **operators** y **operands** que son dos pilas de operadores y operandos. Esta función realiza las siguientes funciones:
    a. **operator = operators.pop()**: Se extrae el último operador de la pila de operadores **operators**.
    b. Si el operador es extraído se empieza a construir el autómata.
        i. Se extrae ultimo operando de la pila de operandos que representa al autómata que cerrara.
        ii. Se crea un nuevo grafo (DiGraph) en ayuda de la biblioteca.
        iii. Se agregar las aristas para el cierre.
        iv. Se agregan aristas que permitan repetir el autómata original cero o más veces.
        v. Se agregan las aristas que permitan pasar del estado inicial al estado final y del estado final al autómata original.
        vi. Se agrega una arista que permite la transición del estado inicial a el autómata original.
        vii. El resultado se coloca en la pila de **operands**.
    c. Si el operador extraído es '.' (concatenación) o '|' (unión), se construye el autómata correspondiente al operador.
        i. Se extraen los dos últimos operandos de la pila, que representan los dos autómatas en la operación.
        ii. Se crea un nuevo grafo para representar el automata restante.
        iii. Se agregar aristas para permitir transiciones de estado inicial hacia los autómatas originales.
        iv. El autoamta resultante **nfa** se coloca en la pila de operandos.

```python
def process_operator(operators, operands):
    operator = operators.pop()
    if operator == '*':
        # Operador de cierre de Kleene (*)
        nfa_operand = operands.pop()
        nfa = nx.DiGraph()
        nfa.add_edge(1, 2, label='ε')
        nfa.add_edge(2, 2, label='ε')
        nfa.add_edge(1, 3, label='ε')
        nfa.add_edge(3, 2, label='ε')
        nfa.add_edge(3, 4, label='ε')
        nfa.add_edge(2, 4, label='ε')
        nfa.add_edge(4, 5, label='ε')
        nfa.add_edge(5, 6, label='ε')
        nfa.add_edge(6, 3, label='ε')
        nfa.add_edge(6, 5, label='ε')
        operands.append(nfa)
    elif operator in ['.', '|']:
        # Operadores de concatenación (.) y unión (|)
        nfa_operand2 = operands.pop()
        nfa_operand1 = operands.pop()
        nfa = nx.DiGraph()
        nfa.add_edges_from([(x[0], x[1], y) for x, y in nfa_operand1.edges.items()])
        nfa.add_edges_from([(x[0] + len(nfa_operand1), x[1] + len(nfa_operand1), y) for x, y in nfa_operand2.edges.items()])
        nfa.add_edge(0, 1, label='ε')
        nfa.add_edge(0, len(nfa_operand1) + 1, label='ε')
        nfa.add_edge(len(nfa_operand1), len(nfa.nodes), label='ε')
        nfa.add_edge(len(nfa_operand1) + len(nfa_operand2), len(nfa.nodes), label='ε')
        operands.append(nfa)
```

4. Funcion **printNFA** que da como salida el automata NFA construido.

```python
def print_nfa(nfa):
    print("NFA:")
    for state, neighbors in nfa.adjacency():
        transitions = [(neighbor, data['label']) for neighbor, data in neighbors.items()]
        print(f"{state} => {transitions}")
```

5. Posteriormente para iniciar a convertir un NFA a un DFA se usó la función **épsilon_closure** que calcula la cerradura épsilon de los estados del automata NFA.

```python
def epsilon_closure(nfa, states):
    closure = set(states)
    stack = list(states)
    while stack:
        state = stack.pop()
        for neighbor in nfa[state]:
            label = nfa[state][neighbor].get('label')
            if label == 'ε' and neighbor not in closure:
                closure.add(neighbor)
                stack.append(neighbor)
    return closure
```

6. Parte del algoritmo de Thomson es el movimiento **move** el cual se implementó en la función de mismo nombre.

```python
def move(nfa, states, symbol):
    moves = set()
    for state in states:
        for neighbor in nfa[state]:
            label = nfa[state][neighbor].get('label')
            if label == symbol:
                moves.add(neighbor)
    return moves
```

7. El último paso para convertir un autómata NFA a DFA fue con la función **convert_to_dfa.** Esta funcion realiza:
    a. Crea un grafo vacio y una lista para almacenar los estados del DFA.
    b. Calcula la cerradura épsilon del estado inicial.
    c. Agrega el estado inicial al grafo.
    d. Se crea una cola para procesar los estados.
    e. Se obtienen los siguientes estados.
    f. Se calcula la cerradura épsilon de los siguientes estados.
    g. Si el conjunto de estados no se encuentra se agrega al DFA y la cola.
    h. Calcula los estados de aceptación.

```python
def convert_to_dfa(nfa, alphabet):
    dfa = nx.DiGraph()
    dfa_states = []
    start_state = epsilon_closure(nfa, [0])
    dfa_states.append(tuple(sorted(start_state)))
    dfa.add_node(0)

    queue = [start_state]
    while queue:
        current_state = queue.pop(0)
        for symbol in alphabet:
            new_state = epsilon_closure(nfa, move(nfa, current_state, symbol))
            if not new_state:
                continue
            if tuple(sorted(new_state)) not in dfa_states:
                dfa_states.append(tuple(sorted(new_state)))
                queue.append(new_state)
                dfa.add_node(len(dfa_states) - 1)
            dfa.add_edge(dfa_states.index(tuple(sorted(current_state))), dfa_states.index(tuple(sorted(new_state))), label=symbol)

    accepting_states = [i for i, state in enumerate(dfa_states) if any(x in state for x in nfa.nodes())]
    return dfa, accepting_states
```

8. Ahora creamos una función que servirá como función principal, que será responsable la ejecución e implementación de lo previamente visto. También esta encargada de la interacción con el usuario y de la salida.

```python
def main():
    alphabet = input("Alphabet: ").strip()
    regex = input("RegEx: ").strip()
    nfa = construct_nfa(regex, alphabet)
    print("----RESULTS----")
    print("NFA:")
    print_nfa(nfa)

    dfa, accepting_states = convert_to_dfa(nfa, alphabet)
    print_dfa(dfa, accepting_states)

if __name__ == "__main__":
    main()
```

Esta actividad integradora fue sumamente retadora y a mi consideración las más complicada de resolver en todos los aspectos. A pesar de eso considero que la solución es competente o al menos me deja un aprendizaje de la teoría de autómatas y ayudar a comprender mas sobre lenguajes, compiladores etc…