Documentação: Minimização de Autômatos Finitos Determinísticos (DFA)

Introdução

Autômatos Finitos Determinísticos (DFAs) são máquinas de estado finito utilizadas na teoria da computação e na ciência da computação para reconhecer padrões dentro de cadeias de caracteres. A minimização de DFAs é o processo de transformar um DFA em outro DFA equivalente que tenha o menor número possível de estados.

Esta documentação descreve o processo de minimização de DFAs, apresentando o código utilizado, os resultados obtidos (principalmente os diagramas dos autômatos antes e depois da minimização), e um manual do usuário para orientar a utilização do código.

Código Utilizado

O código a seguir define uma classe `DFA` que permite a criação, visualização e minimização de autômatos finitos determinísticos.

```
import networkx as nx

import matplotlib.pyplot as plt


# Define a classe DFA (Deterministic Finite Automaton)

class DFA:

    def __init__(self, states, alphabet, transition_function, start_state, accept_states):

        # Inicializa os estados do DFA

        self.states = states

        # Inicializa o alfabeto do DFA

        self.alphabet = alphabet

        # Inicializa a função de transição do DFA

        self.transition_function = transition_function

        # Inicializa o estado inicial do DFA

        self.start_state = start_state

        # Inicializa os estados de aceitação do DFA

        self.accept_states = accept_states


    # Método para remover estados inacessíveis

    def remove_inaccessible_states(self):
```

```
        # Conjunto para armazenar estados acessíveis
        accessible_states = set()
        # Fila de estados a serem verificados, começando pelo estado inicial
        queue = [self.start_state]
        while queue:
            # Remove o primeiro estado da fila
            state = queue.pop(0)
            if state not in accessible_states:
                # Adiciona estado ao conjunto de estados acessíveis
                accessible_states.add(state)
                # Para cada símbolo no alfabeto
                for symbol in self.alphabet:
                    # Obtém o próximo estado a partir da função de transição
                    next_state = self.transition_function.get((state, symbol))
                    if next_state and next_state not in accessible_states:
                        # Adiciona o próximo estado à fila se ainda não foi visitado
                        queue.append(next_state)
        # Atualiza os estados, função de transição e estados de aceitação com apenas os estados acessíveis
        self.states = accessible_states
        self.transition_function = {(state, symbol): next_state for (state, symbol), next_state in self.transition_function.items() if state in accessible_states and next_state in accessible_states}
        self.accept_states = {state for state in self.accept_states if state in accessible_states}

    # Método para minimizar o DFA
    def minimize(self):
        # Primeiro, remove estados inacessíveis
        self.remove_inaccessible_states()

        # Inicialmente, dois conjuntos: estados de aceitação e não aceitação
        P = [set(self.accept_states), set(self.states) - set(self.accept_states)]
        # Fila de conjuntos para verificar distinção
        W = [set(self.accept_states), set(self.states) - set(self.accept_states)]
```

```
while W:
    # Remove um conjunto da fila
    A = W.pop()
    for symbol in self.alphabet:
        # X é o conjunto de estados que transitam para um estado em A com o símbolo atual
        X = {state for state in self.states if self.transition_function.get((state, symbol)) in A}
        for Y in P[:]:
            # Interseção de X e Y
            intersection = X & Y
            # Diferença entre Y e X
            difference = Y - X
            if intersection and difference:
                # Remove Y de P
                P.remove(Y)
                # Adiciona interseção a P
                P.append(intersection)
                # Adiciona diferença a P
                P.append(difference)
                if Y in W:
                    # Remove Y de W
                    W.remove(Y)
                    # Adiciona interseção a W
                    W.append(intersection)
                    # Adiciona diferença a W
                    W.append(difference)
                else:
                    if len(intersection) <= len(difference):
                        # Adiciona menor conjunto a W
                        W.append(intersection)
                    else:
                        # Adiciona maior conjunto a W
                        W.append(difference)
```

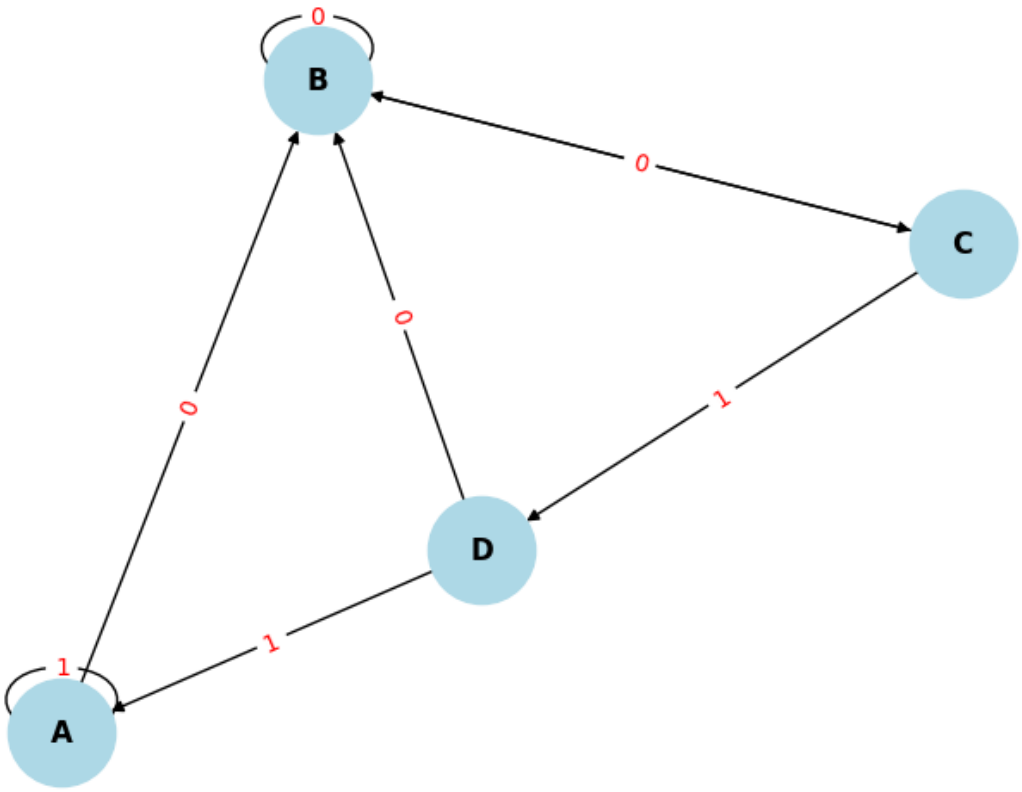
```
        # Criar novos estados
        new_states = {frozenset(s): i for i, s in enumerate(P)}
        # Nova função de transição
        new_transition_function = {}
        # Novo conjunto de estados de aceitação
        new_accept_states = set()

        for group in P:
            # Pega um representante do grupo
            representative = next(iter(group))
            # Estado novo correspondente ao grupo
            new_state = new_states[frozenset(group)]
            if representative in self.accept_states:
                # Se o representante é estado de aceitação, adiciona o novo estado aos estados de
aceitação
                new_accept_states.add(new_state)
            for symbol in self.alphabet:
                # Obtém o próximo estado a partir do representante
                next_state = self.transition_function.get((representative, symbol))
                if next_state:
                    # Encontra o grupo ao qual o próximo estado pertence
                    next_group = next(s for s in P if next_state in s)
                    # Define a nova transição
                    new_transition_function[(new_state, symbol)] = new_states[frozenset(next_group)]

        # Atualiza os estados com os novos estados
        self.states = set(new_states.values())
        # Atualiza a função de transição
        self.transition_function = new_transition_function
        # Atualiza os estados de aceitação
        self.accept_states = new_accept_states
        # Atualiza o estado inicial
        self.start_state = new_states[frozenset(next(s for s in P if self.start_state in s))]
```

```
    # Método para representar o DFA como string
    def __str__(self):
        return f"States: {self.states}\nAlphabet: {self.alphabet}\nStart state: {self.start_state}\nAccept states: {self.accept_states}\nTransitions: {self.transition_function}"

    # Método para desenhar o grafo do DFA
    def draw(self):
        # Cria um grafo direcionado
        G = nx.DiGraph()
        # Adiciona nós ao grafo
        G.add_nodes_from(self.states)
        for (state, symbol), next_state in self.transition_function.items():
            # Adiciona arestas ao grafo
            G.add_edge(state, next_state, label=symbol)

        # Layout do grafo
        pos = nx.spring_layout(G)
        # Desenha o grafo
        nx.draw(G, pos, with_labels=True, node_size=2000, node_color="lightblue", font_size=12, font_weight="bold", arrows=True)
        # Define os rótulos das arestas
        edge_labels = {(u, v): d['label'] for u, v, d in G.edges(data=True)}
        # Desenha os rótulos das arestas
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')
        # Exibe o grafo
        plt.show()
```

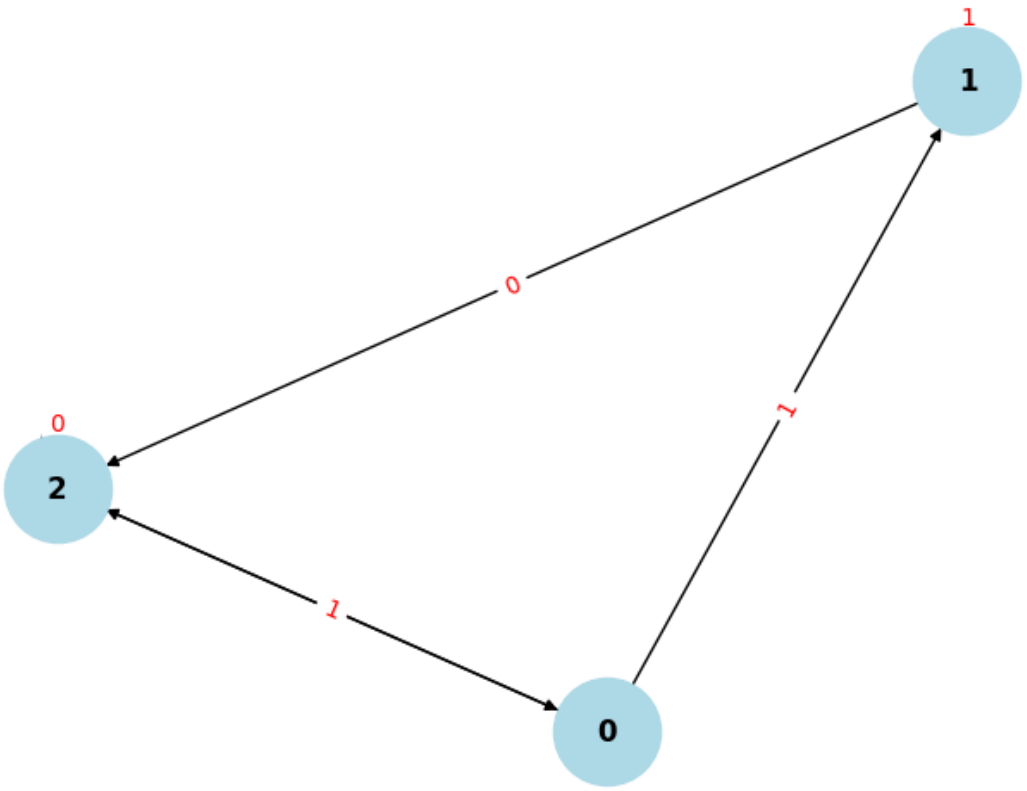
Plots:

Exemplo 1:

Original

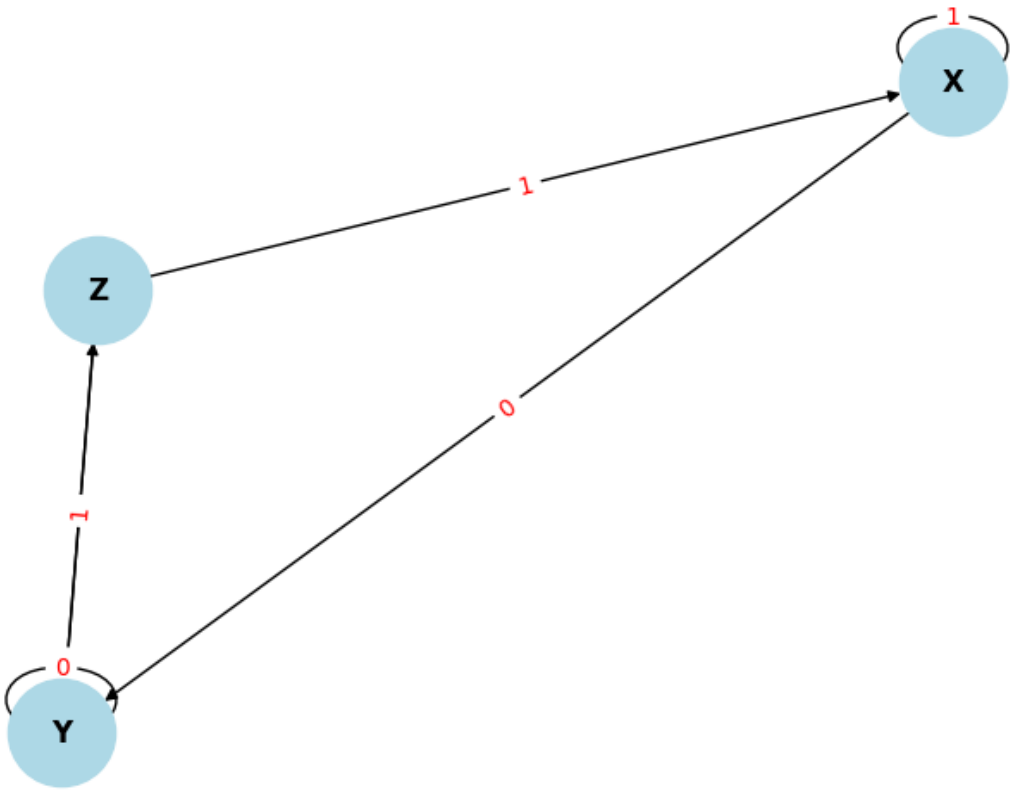


Minimizado

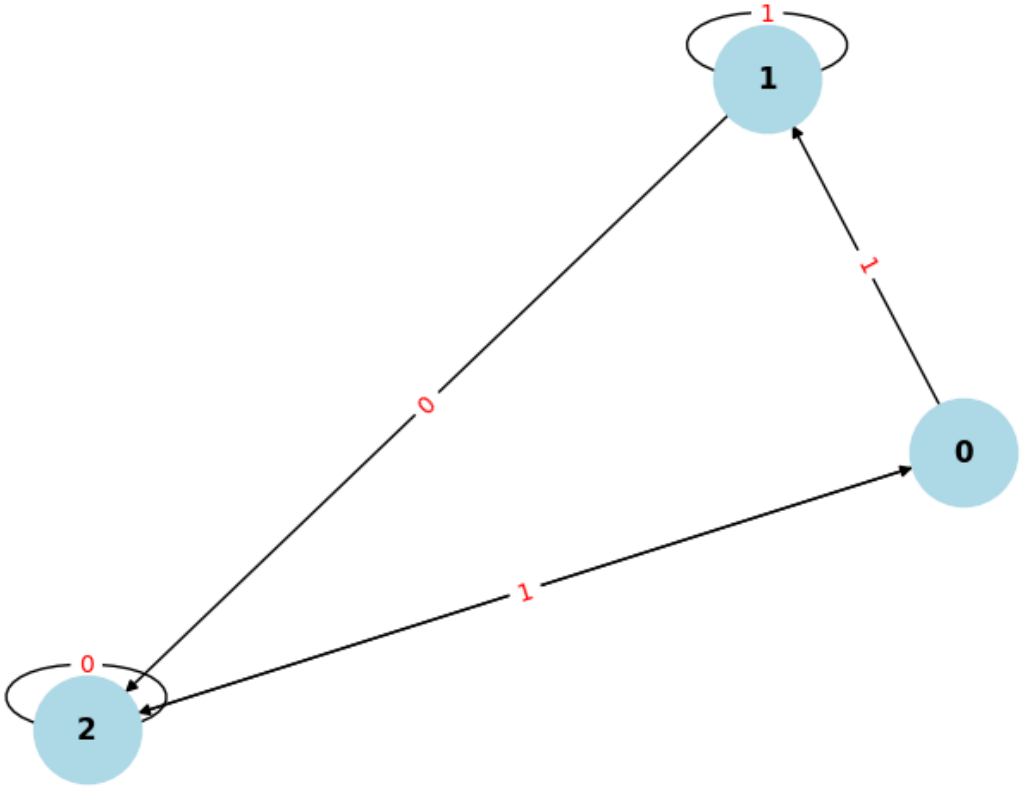


Exemplo 2:

Original

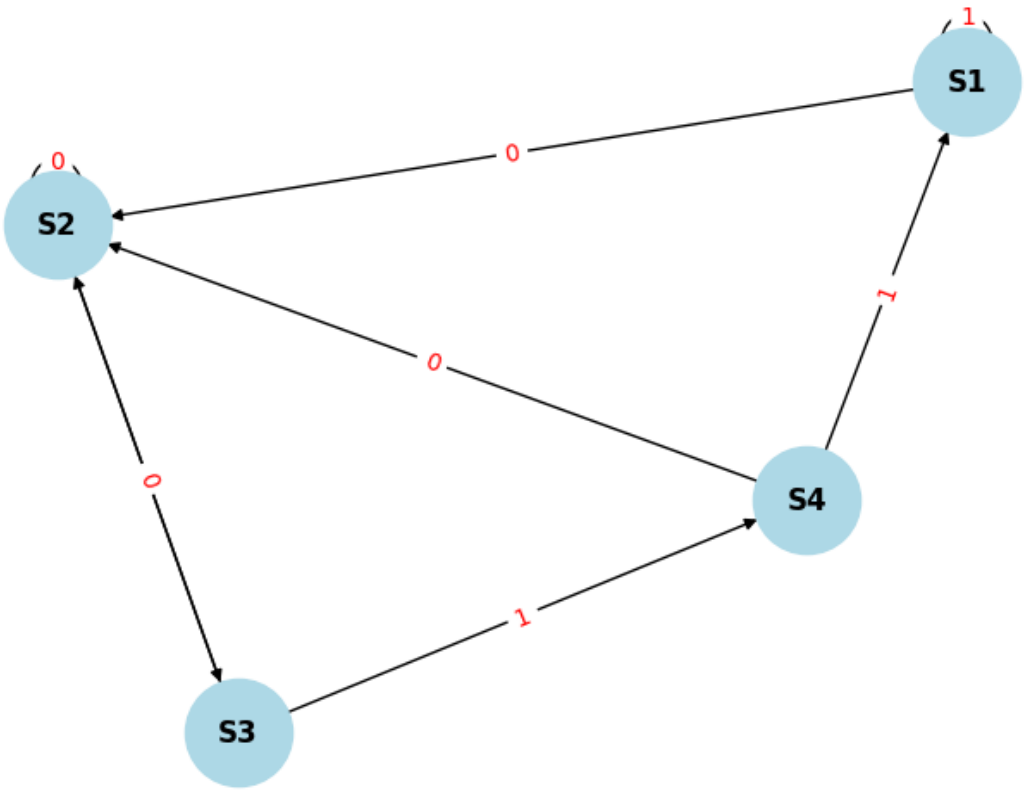


Minimizado

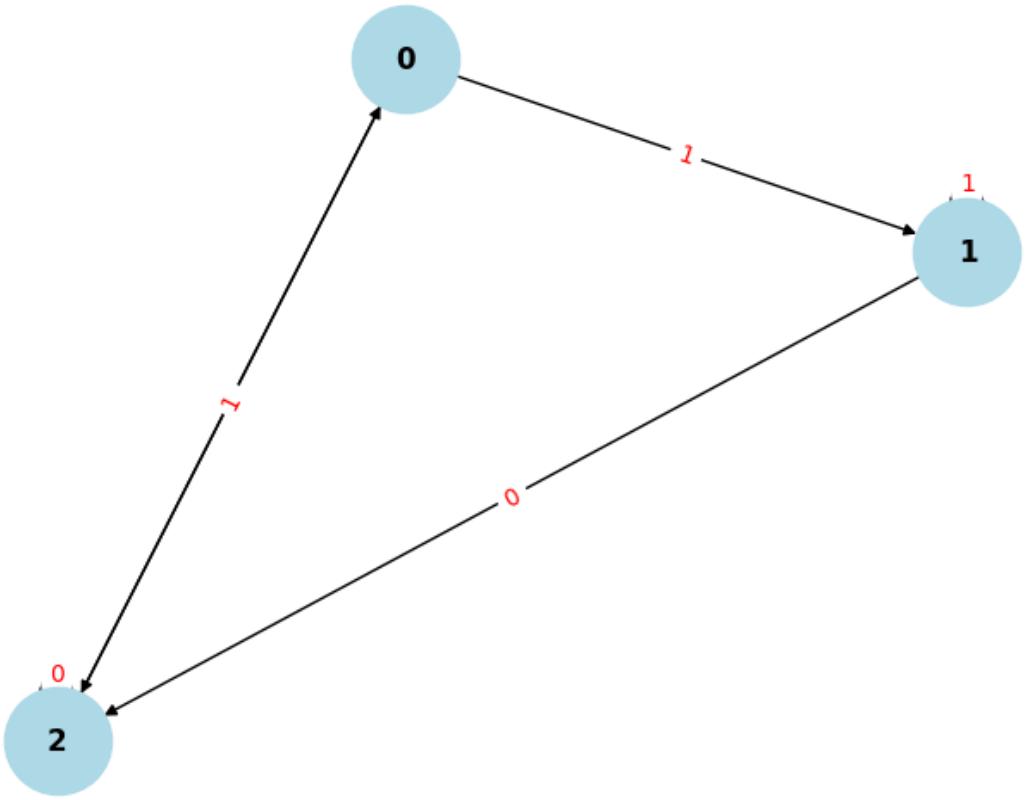


Exemplo 3

Original



Minimizado



Exemplos de Uso

Exemplo 1: Cadeias que terminam com "01"

DFA Original

```
states = {'A', 'B', 'C', 'D'}
alphabet = {'0', '1'}
transition_function = {
    ('A', '0'): 'B',
    ('A', '1'): 'A',
    ('B', '0'): 'B',
    ('B', '1'): 'C',
    ('C', '0'): 'B',
    ('C', '1'): 'D',
    ('D', '0'): 'B',
    ('D', '1'): 'A'
}
start_state = 'A'
accept_states = {'C'}

dfa1 = DFA(states, alphabet, transition_function, start_state, accept_states)
print("Original DFA 1:")
print(dfa1)
dfa1.draw()
```

DFA Minimizado

```
dfa1.minimize()
print("\nMinimized DFA 1:")
print(dfa1)
dfa1.draw()
```

Exemplo 2: Cadeias que contêm "01" no final

DFA Original

```
states = {'X', 'Y', 'Z'}
alphabet = {'0', '1'}
transition_function = {
    ('X', '0'): 'Y',
```

```
    ('X', '1'): 'X',
    ('Y', '0'): 'Y',
    ('Y', '1'): 'Z',
    ('Z', '0'): 'Y',
    ('Z', '1'): 'X'
}
start_state = 'X'
accept_states = {'Z'}

dfa2 = DFA(states, alphabet, transition_function, start_state, accept_states)
print("Original DFA 2:")
print(dfa2)
dfa2.draw()
```

DFA Minimizado

```
dfa2.minimize()
print("\nMinimized DFA 2:")
print(dfa2)
dfa2.draw()
```

Exemplo 3: Cadeias que terminam com "01" (estrutura diferente)

DFA Original

```
states = {'S1', 'S2', 'S3', 'S4'}
alphabet = {'0', '1'}
transition_function = {
    ('S1', '0'): 'S2',
    ('S1', '1'): 'S1',
    ('S2', '0'): 'S2',
    ('S2', '1'): 'S3',
    ('S3', '0'): 'S2',
    ('S3', '1'): 'S4',
    ('S4', '0'): 'S2',
    ('S4', '1'): 'S1'
}
start_state = 'S1'
accept_states = {'S3'}
```

```
dfa3 = DFA(states, alphabet, transition_function, start_state, accept_states)

print("Original DFA 3:")

print(dfa3)

dfa3.draw()
```

DFA Minimizado

```
dfa3.minimize()

print("\nMinimized DFA 3:")

print(dfa3)

dfa3.draw()
```

Manual do Usuário

1. **Instalação das Dependências:** Certifique-se de ter o `networkx` e o `matplotlib` instalados. Utilize os comandos abaixo para instalar:

   pip install networkx matplotlib

2. **Criação de um DFA:** Para criar um DFA, instancie a classe `DFA` com os estados, alfabeto, função de transição, estado inicial e estados de aceitação.

   dfa = DFA(states, alphabet, transition_function, start_state, accept_states)

3. **Visualização do DFA:** Use o método `draw()` para visualizar o DFA.

   dfa.draw()

4. **Minimização do DFA:** Utilize o método `minimize()` para minimizar o DFA.

   dfa.minimize()

5. **Visualização do DFA Minimizado:** Após a minimização, use novamente o método `draw()` para visualizar o DFA minimizado.

   dfa.draw()