

Universidade Federal de Viçosa  
*Campus* Rio Paranaíba

Eduardo Henrique Silva Oliveira - 6032

Relatório do Projeto de Autômatos e Máquina de Turing

Rio Paranaíba - MG

2024

**Universidade Federal de Viçosa**  
***Campus* Rio Paranaíba**

Eduardo Henrique Silva Oliveira - 6032

**Relatório do Projeto de Autômatos e Máquina de Turing**

Trabalho apresentado para obtenção de créditos na disciplina SIN131 - Introdução à Teoria da Computação da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Henrique Aparecido Damaso de Melo.

**Rio Paranaíba - MG**

**2024**

# Sumário

<b>1</b>	<b>Introdução</b>	<b>4</b>
<b>2</b>	<b>Objetivos</b>	<b>4</b>
<b>3</b>	<b>Implementação</b>	<b>4</b>
3.1	Tecnologias, Linguagens e Frameworks Utilizados . . . . .	4
3.1.1	Python . . . . .	4
3.1.2	Flask . . . . .	5
3.1.3	Jinja2 . . . . .	5
3.1.4	HTML, CSS e JavaScript . . . . .	5
3.1.5	Graphviz . . . . .	5
3.2	Estrutura do Projeto . . . . .	5
3.3	app.py . . . . .	6
3.4	Implementação do app.py . . . . .	6
3.5	Principais Rotas . . . . .	6
3.5.1	Rota <code>processar_automato</code> . . . . .	6
3.5.2	Processamento de AFN e AFD . . . . .	8
3.5.3	Rota <code>processar_turing</code> . . . . .	8
3.6	automatos/ . . . . .	10
3.6.1	Implementação do Arquivo <code>afd.py</code> . . . . .	10
3.6.2	Estrutura da Classe AFD . . . . .	11
3.6.3	Métodos Principais . . . . .	11
3.6.4	Implementação do Arquivo <code>afn.py</code> . . . . .	12
3.6.5	Implementação do Arquivo <code>conversao.py</code> . . . . .	13
3.6.6	Função <code>completar_afd</code> . . . . .	13
3.6.7	Função <code>converter_afn_para_afd</code> . . . . .	14
3.6.8	Implementação do Arquivo <code>entrada.py</code> . . . . .	15
3.6.9	Função <code>receber_entrada_afn</code> . . . . .	15
3.6.10	Função <code>receber_entrada_afd</code> . . . . .	16

3.6.11	Implementação do Arquivo <code>minimizacao.py</code> . . . . .	17
3.6.12	Função <code>particionar_estados</code> . . . . .	17
3.6.13	Função <code>minimizar_afd</code> . . . . .	17
3.6.14	Explicação Detalhada da Função <code>minimizar_afd</code> . . . . .	19
3.6.15	O Algoritmo de Minimização de Estados . . . . .	19
3.6.16	Particionamento Inicial . . . . .	19
3.6.17	Laço <code>while</code> . . . . .	19
3.6.18	Refinamento dos Grupos de Estados . . . . .	20
3.6.19	Atualização dos Grupos . . . . .	20
3.6.20	Construção do AFD Minimizado . . . . .	20
3.6.21	Implementação do Arquivo <code>simulacao.py</code> . . . . .	21
3.6.22	Função <code>simular_afn</code> . . . . .	21
3.6.23	Função <code>simular_afd</code> . . . . .	22
3.6.24	<code>maquinaTuring/</code> . . . . .	22
3.6.25	<code>templates/</code> . . . . .	22
3.6.26	<code>static/</code> . . . . .	23
3.7	Detalhes da Implementação dos Autômatos . . . . .	23
3.7.1	Divisão em Módulos e Funções Específicas . . . . .	23
3.7.2	Uso de Classes e Abordagem de Implementação . . . . .	23
<b>4</b>	<b>Especificação da Máquina de Turing</b>	<b>23</b>
4.1	Problema 1: Incremento de Números Binários . . . . .	24
4.1.1	Descrição do Problema . . . . .	24
4.1.2	Definição do Alfabeto da Fita . . . . .	24
4.1.3	Definição dos Estados . . . . .	24
4.1.4	Regras de Transição . . . . .	24
4.1.5	Condições de Aceitação e Rejeição . . . . .	25
4.2	Problema 2: Reconhecimento de Linguagem Regular (Número Par de 'a's') .	25
4.2.1	Descrição do Problema . . . . .	25
4.2.2	Definição do Alfabeto da Fita . . . . .	25

4.2.3	Definição dos Estados . . . . .	25
4.2.4	Regras de Transição . . . . .	26
4.2.5	Condições de Aceitação e Rejeição . . . . .	27
4.3	Implementação da Máquina de Turing . . . . .	27
4.3.1	Diferença na Inicialização da Cabeça de Leitura . . . . .	29
<b>5</b>	<b>Conclusão</b>	<b>30</b>

# 1 Introdução

Este relatório descreve o desenvolvimento de uma aplicação web que implementa Autômatos Finitos Determinísticos (AFD), Autômatos Finitos Não-determinísticos (AFN), e uma Máquina de Turing. O projeto visa fornecer uma ferramenta interativa para a simulação e visualização desses modelos computacionais.

## 2 Objetivos

Os principais objetivos do projeto foram:

- Implementar a conversão de AFN para AFD.
- Simular a aceitação de palavras em AFNs e AFDs.
- Demonstrar a equivalência entre AFNs e AFDs convertidos.
- Implementar a minimização de AFDs.
- Desenvolver uma Máquina de Turing para simular operações computacionais específicas.

## 3 Implementação

### 3.1 Tecnologias, Linguagens e Frameworks Utilizados

Este projeto foi desenvolvido utilizando um conjunto de tecnologias e ferramentas que foram escolhidas para garantir eficiência, flexibilidade e facilidade de manutenção. Abaixo está uma visão geral das principais tecnologias, linguagens e frameworks utilizados:

#### 3.1.1 Python

Python foi a linguagem de programação principal utilizada no desenvolvimento do projeto.

### **3.1.2 Flask**

Flask foi o framework web escolhido para desenvolver a interface interativa do projeto. Flask é um microframework leve que permite o desenvolvimento rápido e eficiente de aplicações web. Ele foi utilizado para definir as rotas da aplicação, renderizar os templates HTML e conectar as funções de backend com a interface de usuário.

### **3.1.3 Jinja2**

Jinja2 é o mecanismo de templates utilizado em conjunto com Flask. Ele permite a geração dinâmica de HTML, facilitando a integração dos dados gerados pelo backend com a interface web. O uso de Jinja2 permitiu que o conteúdo dinâmico fosse facilmente renderizado nas páginas HTML, proporcionando uma experiência interativa ao usuário.

### **3.1.4 HTML, CSS e JavaScript**

Para a construção da interface do usuário, foram utilizadas as linguagens padrão da web: HTML, CSS e JavaScript. O HTML foi utilizado para estruturar as páginas, o CSS para estilizar e proporcionar uma experiência visual agradável, e o JavaScript para adicionar interatividade e melhorar a experiência do usuário.

### **3.1.5 Graphviz**

Graphviz foi utilizado para a visualização dos autômatos gerados. Esta ferramenta permite a geração de gráficos a partir de descrições textuais, facilitando a visualização das transições e estados dos autômatos. A integração com Python foi realizada através da biblioteca ‘graphviz’, que permite a criação e manipulação de gráficos diretamente no código.

## **3.2 Estrutura do Projeto**

O projeto foi organizado de forma modular, com cada componente desempenhando uma função específica e bem definida. Essa abordagem não apenas facilita a manutenção do código, mas também permite uma melhor organização e reusabilidade das funcionalidades. Abaixo estão os principais módulos e diretórios que compõem o projeto:

### 3.3 app.py

O arquivo `app.py` é o núcleo da aplicação Flask, responsável por definir as rotas e controlar o fluxo principal do programa. Ele age como o ponto de entrada para o projeto, onde as diferentes funcionalidades do sistema são conectadas e gerenciadas. Este arquivo integra todas as partes do sistema, conectando as rotas com os módulos específicos para processar autômatos e a Máquina de Turing.

### 3.4 Implementação do app.py

O arquivo `app.py` é o núcleo da aplicação, responsável por gerenciar a interação entre a interface web e os módulos de processamento de autômatos e Máquina de Turing. Ele utiliza o framework Flask para definir as rotas e controlar o fluxo de dados.

### 3.5 Principais Rotas

#### 3.5.1 Rota processar\_automato

Essa rota é responsável por capturar os dados do autômato (AFN ou AFD) que o usuário inseriu no formulário, processá-los e gerar os resultados.

```
1 @app.route('/processar_automato', methods=['POST'])
2 def processar_automato():
3     tipo = request.form.get('tipo').lower()
4     estados = [estado.strip() for estado in request.form.get('estados').split(',')]
5     alfabeto = [simbolo.strip() for simbolo in request.form.get('alfabeto').split(',')]
6     transicoes_brutas = [transicao.strip() for transicao in request.form.get('transicoes').split(';') if transicao.strip()]
7     estado_inicial = request.form.get('estado_inicial').strip()
8     estados_aceitacao = [estado.strip() for estado in request.form.get('estados_aceitacao').split(',')]
9     palavras = [palavra.strip() for palavra in request.form.get('palavras').split(',')]
```

**Captura de Dados:** A rota `processar_automato` captura os dados submetidos pelo usuário através de `request.form.get()`. Cada campo do formulário é processado, com os valores sendo convertidos em listas ou strings conforme necessário.

- **Tipo de Autômato (tipo):** Determina se o autômato é AFN ou AFD.
- **Estados (estados):** Lista de estados do autômato, extraídos da entrada do usuário e limpos de espaços em branco.



- **Alfabeto** (alfabeto): Lista de símbolos que compõem o alfabeto do autômato.
- **Transições** (transicoes\_brutas): Conjunto de transições definidas pelo usuário, divididas e armazenadas para posterior processamento.
- **Estado Inicial** (estado\_inicial): O estado onde o autômato começa o processamento.
- **Estados de Aceitação** (estados\_aceitacao): Lista de estados que determinam a aceitação de uma palavra.
- **Palavras** (palavras): Lista de palavras que serão testadas no autômato.

Após capturar e processar os dados, a rota organiza as transições em um formato que o sistema possa utilizar.

```
1 transicoes = {}
2 for transicao in transicoes_brutas:
3     partes = transicao.split(',')
4     if len(partes) < 3:
5         continue
6     estado_de = partes[0].strip()
7     simbolo = partes[1].strip()
8     estado_para = partes[2].strip()
9     if tipo == 'afn':
10         if (estado_de, simbolo) not in transicoes:
11             transicoes[(estado_de, simbolo)] = set()
12             transicoes[(estado_de, simbolo)].add(estado_para)
13     else:
14         transicoes[(estado_de, simbolo)] = estado_para
```

**Organização das Transições:** A rota transforma as transições brutas inseridas pelo usuário em um formato estruturado, mapeando cada transição do estado atual para o próximo estado, dependendo do símbolo lido. No caso de um AFN, as transições podem levar a múltiplos estados, enquanto em um AFD, cada transição leva a um único estado.

### 3.5.2 Processamento de AFN e AFD

Dependendo do tipo de autômato escolhido, a função trata o AFN ou AFD de maneira apropriada, realizando conversões, simulações e minimizações.

```
1 if tipo == 'afn':
2     afn = AFN(set(estados), set(alfabeto), transicoes, estado_inicial, set(estados_aceitacao))
3     afd = converter_afn_para_afd(afn)
4     resultados_afn = {palavra: simular_afn(afn, palavra) for palavra in palavras}
5     resultados_afd = {palavra: simular_afd(afd, palavra) for palavra in palavras}
6     equivalente, palavra_diferente = verificar_equivalencia(afn, afd, palavras)
7     afd_minimizado = minimizar_afd(afd)
8
9     afd = converter_estados_legiveis(afd)
10    afd_minimizado = converter_estados_legiveis(afd_minimizado)
11
12    visualizar_automato(afn, 'static/visualizacao/afn')
13    visualizar_afd(afd, 'static/visualizacao/afd')
14    visualizar_afd(afd_minimizado, 'static/visualizacao/afd_minimizado')
15
16    return render_template('resultado.html', afn=afn.to_dict(), afd=afd.to_dict(), resultados_afn=resultados_afn,
17                          resultados_afd=resultados_afd, equivalente=equivalente,
18                          palavra_diferente=palavra_diferente, afd_minimizado=afd_minimizado.to_dict())
```

```
1 else:
2     afd = AFD(set(estados), set(alfabeto), transicoes, estado_inicial, set(estados_aceitacao))
3     resultados_afd = {palavra: simular_afd(afd, palavra) for palavra in palavras}
4     afd_minimizado = minimizar_afd(afd)
5
6     afd = converter_estados_legiveis(afd)
7     afd_minimizado = converter_estados_legiveis(afd_minimizado)
8
9     visualizar_afd(afd, 'static/visualizacao/afd')
10    visualizar_afd(afd_minimizado, 'static/visualizacao/afd_minimizado')
11
12    return render_template('resultado_afd.html', afd=afd.to_dict(), resultados_afd=resultados_afd, afd_minimizado=afd_minimizado.to_dict())
```

**Conversão e Simulação:** Se o tipo for AFN, o autômato é convertido para AFD, simulando ambos e verificando a equivalência. Caso o tipo seja AFD, apenas a simulação e minimização são feitas.

**Visualização:** Os resultados, incluindo as visualizações gráficas dos autômatos, são gerados e enviados para serem exibidos na interface do usuário.

### 3.5.3 Rota processar\_turing

A rota `processar_turing` captura e processa as especificações de uma Máquina de Turing inseridas pelo usuário, e realiza simulações de problemas como incremento binário e reconhecimento de linguagem regular.

```

1 @app.route('/processar_turing', methods=['POST'])
2 def processar_turing():
3     tipo_problema = request.form.get('tipo_problema')
4     estados = request.form['estados'].split(',')
5     alfabeto_entrada = request.form['alfabeto_entrada'].split(',')
6     alfabeto_fita = request.form['alfabeto_fita'].split(',')
7     simbolo_vazio = request.form['simbolo_vazio']
8     estado_inicial = request.form['estado_inicial']
9     estados_finais = request.form['estados_finais'].split(',')
10    transicoes_brutas = request.form['transicoes'].strip().split('\n')

```

**Captura de Dados:** Similar à rota anterior, `processar_turing` captura as especificações da Máquina de Turing a partir do formulário:

- **Tipo de Problema** (`tipo_problema`): Define o problema a ser resolvido pela Máquina de Turing (incremento binário ou reconhecimento de linguagem).
- **Estados** (`estados`): Lista de estados da Máquina de Turing.
- **Alfabeto de Entrada** (`alfabeto_entrada`): Símbolos que a máquina pode ler da fita.
- **Alfabeto da Fita** (`alfabeto_fita`): Símbolos que a máquina pode escrever na fita.
- **Símbolo Vazio** (`simbolo_vazio`): Símbolo que representa espaços vazios na fita.
- **Estado Inicial** (`estado_inicial`): Estado de início da simulação.
- **Estados Finais** (`estados_finais`): Estados de aceitação da Máquina de Turing.
- **Transições** (`transicoes_brutas`): Regras de transição definidas pelo usuário.

**Organização das Transições:** As transições são organizadas de acordo com o formato de chave-valor, onde a chave é o par (estado atual, símbolo lido), e o valor é uma tupla contendo o próximo estado, o símbolo a ser escrito, e a direção do movimento da cabeça de leitura.

```

1 transicoes = {}
2 for transicao in transicoes_brutas:
3     partes = transicao.split('→')
4     estado_simbolo = partes[0].strip()
5     novo_estado_simbolo_direcao = partes[1].strip().split(',')
6     transicoes[estado_simbolo] = (novo_estado_simbolo_direcao[0].strip(),
7                                     novo_estado_simbolo_direcao[1].strip(),
8                                     novo_estado_simbolo_direcao[2].strip())

```

Por fim, a rota simula a Máquina de Turing de acordo com o problema selecionado, retornando o resultado da simulação para o usuário.

```

1 if tipo_problema == 'incremento_binario':
2     mt = MaquinaTuringIncrementoBinario(estados, alfabeto_entrada, alfabeto_fita, simbolo_vazio, transicoes, estado_inicial, estados_finais)
3     resultado, palavra_resultado = mt.executar(palavra_entrada)
4     return render_template('maquinaTuring.html', resultado=resultado, palavra_resultado=palavra_resultado)
5
6 elif tipo_problema == 'linguagem_par_a':
7     mt = MaquinaTuringLR(estados, alfabeto_entrada, alfabeto_fita, simbolo_vazio, transicoes, estado_inicial, estados_finais)
8     resultado, _ = mt.executar(palavra_entrada)
9     return render_template('maquinaTuring.html', resultado=resultado)

```

**Simulação e Retorno:** Dependendo do problema, a classe correspondente é utilizada para simular a operação da Máquina de Turing. O resultado é então renderizado na página `maquinaTuring.html`, mostrando se a palavra foi aceita e, no caso do incremento binário, exibindo o número incrementado.

## 3.6 automatos/

O diretório `automatos/` contém os módulos responsáveis por lidar com os Autômatos Finitos Determinísticos (AFD) e Autômatos Finitos Não-Determinísticos (AFN). A divisão em múltiplos arquivos foi feita para manter a clareza e a organização, facilitando a manutenção e expansão do código.

### 3.6.1 Implementação do Arquivo `afd.py`

O arquivo `afd.py` contém a classe `AFD`, que é responsável por modelar e manipular Autômatos Finitos Determinísticos (AFD). Este arquivo é crucial para o projeto, pois define as estruturas

de dados e os métodos que permitem a operação de AFDs dentro da aplicação.

### 3.6.2 Estrutura da Classe AFD

A classe AFD é implementada de forma a encapsular todos os elementos fundamentais de um AFD, como seus estados, alfabeto, transições, estado inicial e estados de aceitação.

```
1 def __init__(self, estados, alfabeto, transicoes, estado_inicial, estados_aceitacao):
2     self.estados = estados
3     self.alfabeto = alfabeto
4     self.transicoes = transicoes
5     self.estado_inicial = estado_inicial
6     self.estados_aceitacao = estados_aceitacao
```

**Inicialização (`__init__`):** No construtor, a classe recebe os parâmetros necessários para definir um AFD. Esses parâmetros incluem o conjunto de estados, alfabeto, transições (mapeadas como dicionário), estado inicial e estados de aceitação. A estrutura é desenhada para permitir uma fácil manipulação e simulação do autômato.

### 3.6.3 Métodos Principais

A classe AFD possui métodos importantes para verificar a estrutura do autômato, converter seus dados para outros formatos e representá-lo como uma string.

```
1 def esta_normalizado(self) → bool:
2     for estado in self.estados:
3         for simbolo in self.alfabeto:
4             if (estado, simbolo) not in self.transicoes:
5                 return False
6     return True
```

`esta_normalizado(self)` -> bool: Este método verifica se o AFD está "normalizado". Um AFD é considerado normalizado quando todos os estados possuem transições definidas para cada símbolo do alfabeto. Isso garante que o autômato possa processar qualquer entrada sem resultar em comportamentos indefinidos.

```
1 def to_dict(self):
2     return {
3         'estados': self.estados,
4         'alfabeto': self.alfabeto,
5         'transicoes': {str(k): v for k, v in self.transicoes.items()},
6         'estado_inicial': self.estado_inicial,
7         'estados_aceitacao': self.estados_aceitacao
8     }
```

`to_dict(self)`: Este método converte os dados do AFD em um dicionário. Isso é útil para exportar ou serializar a estrutura do autômato, especialmente quando se deseja exibir as informações na interface web ou salvar o estado atual do AFD.

```
1 def __str__(self):
2     return f"AFD(estados={self.estados}, alfabeto={self.alfabeto}, transicoes={self.transicoes}, estado_inicial={self.estado_inicial}, estados_aceitacao={self.estados_aceitacao})"
```

`__str__(self)`: Este método define a representação em string da classe AFD, facilitando a inspeção e depuração do autômato. Ele fornece uma visão geral dos estados, alfabeto, transições, estado inicial e estados de aceitação, todos formatados como uma string legível.

### 3.6.4 Implementação do Arquivo `afn.py`

O arquivo `afn.py` é responsável por modelar Autômatos Finitos Não-Determinísticos (AFN). A estrutura da classe AFN é semelhante à da classe AFD no arquivo `afd.py`, uma vez que ambos os tipos de autômatos compartilham várias características fundamentais. No entanto, a classe AFN não possui a função `esta_normalizado`, que é específica para autômatos determinísticos.

### 3.6.5 Implementação do Arquivo `conversao.py`

O arquivo `conversao.py` desempenha um papel crucial no projeto, pois contém funções responsáveis por converter um Autômato Finito Não-Determinístico (AFN) em um Autômato Finito Determinístico (AFD) e por garantir que o AFD esteja completo. Além disso, o arquivo inclui funções para melhorar a legibilidade dos estados do AFD.

### 3.6.6 Função `completar_afd`

```
1  def completar_afd(afd: AFD) → AFD:
2
3      estado_neutro = 'estado_neutro'
4      novos_estados = afd.estados | {estado_neutro}
5      novas_transicoes = afd.transicoes.copy()
6
7      # Adiciona transições para o estado neutro
8      for estado in afd.estados:
9          for simbolo in afd.alfabeto:
10             if (estado, simbolo) not in novas_transicoes:
11                 novas_transicoes[(estado, simbolo)] = estado_neutro
12
13     # Estado neutro transita para ele mesmo para qualquer símbolo
14     for simbolo in afd.alfabeto:
15         novas_transicoes[(estado_neutro, simbolo)] = estado_neutro
16
17     return AFD(
18         estados=novos_estados,
19         alfabeto=afd.alfabeto,
20         transicoes=novas_transicoes,
21         estado_inicial=afd.estado_inicial,
22         estados_aceitacao=afd.estados_aceitacao
23     )
```

A função `completar_afd` tem como objetivo garantir que o AFD resultante seja completo, ou seja, que para cada estado e símbolo do alfabeto, exista uma transição definida. Caso contrário, um estado "neutro" é adicionado para absorver todas as transições indefinidas. Este estado neutro transita para ele mesmo em qualquer símbolo, garantindo que o AFD esteja

sempre definido.

### 3.6.7 Função converter\_afn\_para\_afd

```
1 def converter_afn_para_afd(afn: AFN) → AFD:
2
3     # Converte um AFN para um AFD equivalente.
4     estados_dfa = set()
5     transicoes_dfa = {}
6     estados_aceitacao_dfa = set()
7
8     # Estado inicial do AFD é um conjunto contendo o estado inicial do AFN
9     estados_para_processar = [frozenset([afn.estado_inicial])]
10    estados_dfa.add(frozenset([afn.estado_inicial]))
11
12    # Processa todos os estados
13    while estados_para_processar:
14        estado_atual = estados_para_processar.pop()
15
16        for simbolo in afn.alfabeto:
17            novo_estado = frozenset()
18            for subestado in estado_atual:
19                if (subestado, simbolo) in afn.transicoes:
20                    novo_estado |= afn.transicoes[(subestado, simbolo)]
21
22            if novo_estado:
23                transicoes_dfa[(estado_atual, simbolo)] = novo_estado
24                if novo_estado not in estados_dfa:
25                    estados_dfa.add(novo_estado)
26                    estados_para_processar.append(novo_estado)
27
28                if novo_estado & afn.estados_aceitacao:
29                    estados_aceitacao_dfa.add(novo_estado)
30
31    afd = AFD(
32        estados=estados_dfa,
33        alfabeto=afn.alfabeto,
34        transicoes=transicoes_dfa,
35        estado_inicial=frozenset([afn.estado_inicial]),
36        estados_aceitacao=estados_aceitacao_dfa
37    )
38
39    return completar_afd(afd)
```



A função `converter_afn_para_afd` é responsável por converter um AFN em um AFD equivalente. Esse processo envolve:

- **Criação de Estados do AFD:** Cada estado do AFD é representado como um conjunto de estados do AFN. O estado inicial do AFD é o conjunto que contém apenas o estado inicial do AFN.
- **Construção de Transições:** Para cada estado e símbolo do alfabeto, as transições são calculadas com base nos estados alcançáveis no AFN. Os novos estados gerados são adicionados ao conjunto de estados do AFD.
- **Identificação de Estados de Aceitação:** Se qualquer estado no conjunto atual for um estado de aceitação no AFN, o novo estado correspondente no AFD também será um estado de aceitação.
- **Completeness do AFD:** Após a conversão, a função `completar_afd` é chamada para garantir que o AFD resultante esteja completo.

### 3.6.8 Implementação do Arquivo `entrada.py`

O arquivo `entrada.py` é responsável por receber os dados de entrada fornecidos pelo usuário e transformá-los em objetos que representam os autômatos. Ele contém funções essenciais que convertem os dados brutos recebidos em instâncias dos autômatos (AFN e AFD), prontos para serem processados pelas demais funções do projeto.

### 3.6.9 Função `receber_entrada_afn`

A função `receber_entrada_afn` transforma os dados fornecidos pelo usuário em um objeto da classe AFN. Essa função realiza as seguintes operações:

- **Estados:** Converte a lista de estados fornecida em um conjunto, removendo espaços em branco e garantindo que cada estado seja único.
- **Alfabeto:** Converte a lista de símbolos do alfabeto em um conjunto.

```

1 def receber_entrada_afn(dados):
2
3     estados = set(estado.strip() for estado in dados["estados"])
4     alfabeto = set(simbolo.strip() for simbolo in dados["alfabeto"])
5     transicoes = {(estado.strip(), simbolo.strip()): set(destino.strip() for destino in destinos)
6                   for (estado, simbolo), destinos in dados["transicoes"].items()}
7     estado_inicial = dados["estado_inicial"].strip()
8     estados_aceitacao = set(estado.strip() for estado in dados["estados_aceitacao"])
9
10    afn = AFN(estados, alfabeto, transicoes, estado_inicial, estados_aceitacao)
11    return afn
12

```

- **Transições:** Converte as transições fornecidas, que estão no formato de dicionário, em um mapeamento de pares (estado, símbolo) para um conjunto de estados de destino. Essa estrutura é fundamental para capturar a natureza não-determinística do AFN.
- **Estado Inicial e Estados de Aceitação:** Define o estado inicial e os estados de aceitação, garantindo que estejam corretamente formatados.

O resultado final é a criação de uma instância de AFN, que é retornada para ser utilizada nas etapas subsequentes de processamento.

### 3.6.10 Função receber\_entrada\_afd

```

1 def receber_entrada_afd(dados):
2
3     estados = set(estado.strip() for estado in dados["estados"])
4     alfabeto = set(simbolo.strip() for simbolo in dados["alfabeto"])
5     transicoes = {(estado.strip(), simbolo.strip()): destino.strip()
6                   for (estado, simbolo), destino in dados["transicoes"].items()}
7     estado_inicial = dados["estado_inicial"].strip()
8     estados_aceitacao = set(estado.strip() for estado in dados["estados_aceitacao"])
9
10    afd = AFD(estados, alfabeto, transicoes, estado_inicial, estados_aceitacao)
11    return afd

```

A função `receber_entrada_afd` segue uma lógica semelhante à de `receber_entrada_afn`, mas adaptada para a natureza determinística do AFD:

- **Transições:** Diferente do AFN, onde cada par (estado, símbolo) pode mapear para vários estados, no AFD cada par mapeia para um único estado de destino.

Essa função retorna uma instância de AFD criada a partir dos dados de entrada, pronta para ser processada.

### 3.6.11 Implementação do Arquivo `minimizacao.py`

O arquivo `minimizacao.py` contém a implementação da função de minimização de Autômatos Finitos Determinísticos (AFD). A minimização é um processo crucial na teoria dos autômatos, pois permite reduzir o número de estados de um AFD sem alterar a linguagem que ele reconhece. O objetivo é encontrar um AFD equivalente, mas com o menor número de estados possível.

### 3.6.12 Função `particionar_estados`

```
1 def particionar_estados(afd: AFD, estado_neutro: str) -> list:
2
3     # Particiona os estados do AFD em dois conjuntos: aceitação e não aceitação, removendo o estado_neutro.
4     P = [afd.estados_aceitacao, afd.estados - afd.estados_aceitacao]
5     return [p - {estado_neutro} for p in P]
```

Essa função particiona os estados do AFD em dois conjuntos principais: estados de aceitação e estados de não-aceitação. Esse particionamento é o primeiro passo do algoritmo de minimização, que busca agrupar estados que se comportam de maneira idêntica para qualquer entrada.

- **Particionamento Inicial:** Cria dois grupos de estados: os que são de aceitação e os que não são.
- **Remoção do Estado Neutro:** Garante que o `estado_neutro` não faça parte de nenhum grupo após o particionamento.

### 3.6.13 Função `minimizar_afd`

```

1 def minimizar_afd(afd: AFD) → AFD:
2
3     # Minimiza o AFD dado usando o algoritmo de minimização de estados.
4     estadoNeutro = "estadoNeutro"
5
6     # Completa o AFD se necessário
7     if not afd.esta_normalizado():
8         afd = completar_afd(afd)
9
10    # Particiona os estados, excluindo o estadoNeutro
11    gruposEstados = particionar_estados(afd, estadoNeutro)
12    estadosProcessar = [afd.estados_aceitacao.copy()]
13
14    # Processo de minimização
15    while estadosProcessar:
16        estadosAtuais = estadosProcessar.pop()
17        for simbolo in afd.alfabeto:
18            estadosAlvo = {estado for estado in afd.estados if afd.transicoes.get((estado, simbolo)) in estadosAtuais}
19            for grupo in gruposEstados[:]:
20                estadosComuns = estadosAlvo & grupo
21                estadosRestantes = grupo - estadosAlvo
22                if estadosComuns and estadosRestantes:
23                    gruposEstados.remove(grupo)
24                    gruposEstados.append(estadosComuns)
25                    gruposEstados.append(estadosRestantes)
26                if grupo in estadosProcessar:
27                    estadosProcessar.remove(grupo)
28                    estadosProcessar.append(estadosComuns)
29                    estadosProcessar.append(estadosRestantes)
30            else:
31                if len(estadosComuns) ≤ len(estadosRestantes):
32                    estadosProcessar.append(estadosComuns)
33                else:
34                    estadosProcessar.append(estadosRestantes)
35
36    # Remover todas as referências ao estadoNeutro antes de criar o AFD minimizado
37    afd = limpar_estado_neutro_completo(afd)
38
39    # Criar os estados e transições minimizadas
40    estadosMinimizados = {frozenset(grupo) for grupo in gruposEstados}
41    estadoInicialMinimizado = next(frozenset(grupo) for grupo in estadosMinimizados if afd.estado_inicial in grupo)
42    estadosAceitacaoMinimizados = {frozenset(grupo) for grupo in estadosMinimizados if grupo & afd.estados_aceitacao}
43
44    transicoesMinimizadas = {}
45    for grupo in estadosMinimizados:
46        for estado in grupo:
47            for simbolo in afd.alfabeto:
48                estadoDestino = afd.transicoes.get((estado, simbolo))
49                if estadoDestino and estadoDestino ≠ estadoNeutro:
50                    for grupoDestino in estadosMinimizados:
51                        if estadoDestino in grupoDestino:
52                            transicoesMinimizadas[(grupo, simbolo)] = grupoDestino
53                            break
54
55    # Cria o AFD minimizado
56    afdMinimizado = AFD(
57        estados=estadosMinimizados,
58        alfabeto=afd.alfabeto,
59        transicoes=transicoesMinimizadas,
60        estado_inicial=estadoInicialMinimizado,
61        estados_aceitacao=estadosAceitacaoMinimizados
62    )
63
64    # Aplicar limpeza final ao AFD minimizado
65    return limpar_estado_neutro_completo(afdMinimizado)

```

A função `minimizar_afd` aplica o algoritmo de minimização de estados ao AFD. Este é um dos processos mais importantes na teoria dos autômatos, pois garante que o AFD resultante seja o menor possível em termos de número de estados, mantendo a mesma linguagem.

- **Normalização e Estado Neutro:** O AFD é normalizado (completo) se necessário, e o `estadoNeutro` é utilizado para lidar com transições não definidas.
- **Particionamento e Refinamento:** Estados são agrupados inicialmente em dois grupos (aceitação e não-aceitação), e então refinados com base em suas transições. Estados que não podem ser distinguidos uns dos outros são mesclados.
- **Construção do AFD Minimizado:** Novos estados e transições são construídos a partir dos grupos de estados resultantes. O `estado_neutro` é removido, e o AFD minimizado é criado e retornado.

#### 3.6.14 Explicação Detalhada da Função `minimizar_afd`

#### 3.6.15 O Algoritmo de Minimização de Estados

O objetivo do algoritmo é identificar estados equivalentes que podem ser combinados, ou seja, estados que se comportam da mesma maneira para todas as entradas e, portanto, não precisam ser distinguidos.

#### 3.6.16 Particionamento Inicial

A função começa particionando os estados do AFD em dois grupos principais:

- **Estados de Aceitação:** Estados que pertencem ao conjunto de estados finais do AFD.
- **Estados de Não-Aceitação:** Todos os outros estados.

#### 3.6.17 Laço `while`

O laço `while` refina os grupos de estados até que nenhum grupo possa ser dividido mais.

```
while estadosProcessar:
```

```
    estadosAtuais = estadosProcessar.pop()
```

Neste ponto, o algoritmo processa um grupo de estados (chamados **estadosAtuais**) que precisa ser analisado. Ele tenta dividir esse grupo em subgrupos menores, baseando-se nas transições para outros estados sob cada símbolo do alfabeto.

### 3.6.18 Refinamento dos Grupos de Estados

Para cada símbolo do alfabeto, o algoritmo verifica para quais estados cada estado em **estadosAtuais** se transita. Se essas transições levarem para diferentes grupos de estados, então o grupo **estadosAtuais** é dividido.

- **estadosAlvo**: Conjunto de estados que, a partir de **estado**, transitam para algum estado dentro de **estadosAtuais** com o símbolo atual.
- **estadosComuns**: Estados que, ao receber o símbolo atual, transitam para um estado dentro de **estadosAtuais**.
- **estadosRestantes**: Estados que, ao receber o símbolo atual, transitam para fora de **estadosAtuais**.

Se **estadosComuns** e **estadosRestantes** forem não vazios, isso significa que o grupo original (antes indivisível) pode ser subdividido em dois novos grupos: **estadosComuns** e **estadosRestantes**.

### 3.6.19 Atualização dos Grupos

Após dividir um grupo, o algoritmo verifica se o grupo original estava na lista **estadosProcessar**. Se estava, o algoritmo substitui o grupo original por **estadosComuns** e **estadosRestantes** para processamento futuro. Se não, o grupo menor é adicionado à lista de **estadosProcessar** para refinamento posterior.

### 3.6.20 Construção do AFD Minimizado

Depois que todos os grupos são refinados até o ponto onde não podem mais ser divididos, o AFD minimizado é construído. Cada grupo de estados refinados se torna um único estado no novo AFD.

Finalmente, o AFD minimizado é retornado, garantindo que o autômato resultante seja o mais simples possível em termos de número de estados, mas ainda assim reconheça a mesma linguagem que o AFD original.

### 3.6.21 Implementação do Arquivo `simulacao.py`

O arquivo `simulacao.py` contém funções essenciais para simular a execução de autômatos finitos determinísticos (AFD) e não determinísticos (AFN). A simulação é a responsável por verificar se uma determinada palavra é aceita pelo autômato, ou seja, se a palavra pertence à linguagem reconhecida pelo autômato.

### 3.6.22 Função `simular_afn`

```
1 def simular_afn(afn, palavra):
2
3     # Simula a execução de um AFN em uma palavra fornecida.
4     estados_atuais = {afn.estado_inicial}
5     for simbolo in palavra:
6         proximos_estados = set()
7         for estado in estados_atuais:
8             proximos_estados.update(afn.transicoes.get((estado, simbolo), []))
9         estados_atuais = proximos_estados
10    return bool(estados_atuais & afn.estados_aceitacao)
```

A função `simular_afn` é responsável por simular a execução de um AFN em uma palavra fornecida.

- **Processo de Simulação:** A simulação começa com o estado inicial do AFN. Para cada símbolo na palavra de entrada, a função verifica quais são os próximos estados possíveis a partir dos estados atuais, considerando as transições definidas.
- **Transições:** Como o AFN permite transições para múltiplos estados, a função mantém um conjunto de `estados_atuais`, que são atualizados a cada símbolo processado.
- **Aceitação da Palavra:** Após processar todos os símbolos, a função verifica se algum dos estados atuais é um estado de aceitação. Se for, a palavra é aceita pelo autômato.

### 3.6.23 Função `simular_afd`

```
1 def simular_afd(afd, palavra):
2
3     # Simula a execução de um AFD em uma palavra fornecida.
4     estado_atual = afd.estado_inicial
5     for simbolo in palavra:
6         estado_atual = afd.transicoes.get((estado_atual, simbolo))
7         if estado_atual is None:
8             return False
9     return estado_atual in afd.estados_aceitacao
```

A função `simular_afd` simula a execução de um AFD em uma palavra fornecida.

- **Processo de Simulação:** Diferente do AFN, o AFD tem exatamente uma transição para cada símbolo no alfabeto a partir de cada estado. A função começa no estado inicial e segue as transições para cada símbolo na palavra de entrada.
- **Transições:** Se em algum ponto não houver uma transição definida para um símbolo (ou seja, o resultado da transição for `None`), a palavra é rejeitada imediatamente.
- **Aceitação da Palavra:** Se após processar todos os símbolos, o autômato se encontrar em um estado de aceitação, a palavra é aceita.

### 3.6.24 `maquinaTuring/`

Este diretório é dedicado à implementação das Máquinas de Turing. Ele contém classes específicas para resolver problemas como o incremento de números binários e o reconhecimento de linguagens regulares. A separação dessas classes em arquivos distintos permite que cada problema seja tratado de forma isolada, tornando o código mais modular e organizado.

### 3.6.25 `templates/`

O diretório `templates/` armazena os arquivos HTML usados como templates na aplicação Flask. Esses templates são responsáveis pela interface gráfica que o usuário interage, permi-



tindo a inserção de dados e a visualização dos resultados das simulações.

### **3.6.26 static/**

O diretório `static/` contém todos os arquivos estáticos, como folhas de estilo CSS e as imagens geradas para a visualização dos autômatos. Manter esses arquivos em um diretório separado permite uma organização mais limpa e facilita a referência a esses recursos dentro dos templates HTML.

## **3.7 Detalhes da Implementação dos Autômatos**

A implementação dos autômatos foi estruturada para garantir modularidade, clareza e reusabilidade do código, quando necessário.

### **3.7.1 Divisão em Módulos e Funções Específicas**

A implementação dos autômatos foi dividida em vários módulos, cada um focado em uma parte específica da funcionalidade. Essa abordagem modular facilita a manutenção do código e permite que diferentes partes do sistema sejam desenvolvidas e testadas independentemente.

### **3.7.2 Uso de Classes e Abordagem de Implementação**

O uso de classes no projeto, como AFN e AFD, permite encapsular os dados e comportamentos relacionados a cada tipo de autômato. Essa abordagem orientada a objetos facilita a manutenção do código e promove a reutilização de funções em diferentes contextos.

## **4 Especificação da Máquina de Turing**

Neste projeto, a Máquina de Turing foi implementada para resolver dois problemas computacionais específicos: o incremento de números binários e o reconhecimento de uma linguagem regular (número par de 'a's). A seguir, está a especificação formal para cada um desses problemas.

## 4.1 Problema 1: Incremento de Números Binários

### 4.1.1 Descrição do Problema

O objetivo deste problema é criar uma Máquina de Turing que seja capaz de incrementar um número binário em 1. O número binário é representado como uma sequência de 0s e 1s na fita da máquina, e a máquina deve modificar essa sequência para representar o próximo número na sequência binária.

### 4.1.2 Definição do Alfabeto da Fita

- **Alfabeto de entrada:**  $\{0, 1\}$
- **Alfabeto da fita:**  $\{0, 1, \_ \}$  (onde  $\_$  representa o símbolo vazio)

### 4.1.3 Definição dos Estados

- **Estados:**  $\{q_0, q_{Aceita}\}$
- **Estado Inicial:**  $q_0$
- **Estado de Aceitação:**  $q_{Aceita}$

### 4.1.4 Regras de Transição

As regras de transição para a máquina de incremento binário são as seguintes:

- $(q_0, 1) \rightarrow (q_0, 0, L)$ 
  - Se a máquina está no estado  $q_0$  e lê um 1, ela substitui o 1 por 0 e move a cabeça para a esquerda. Isso simula o processo de carry-over, onde o 1 deve ser "virado" para 0, e o próximo dígito à esquerda deve ser processado.
- $(q_0, 0) \rightarrow (q_{Aceita}, 1, S)$ 
  - Se a máquina está no estado  $q_0$  e lê um 0, ela o substitui por 1, completando o incremento do número binário. A máquina então para o movimento da cabeça (S de Stop) e entra no estado de aceitação.

- $(q_0, \_) \rightarrow (q_{\text{Aceita}}, 1, S)$ 
  - Se a máquina está no estado  $q_0$  e encontra o símbolo vazio, significa que todos os dígitos à esquerda são 1s. Nesse caso, o incremento é feito adicionando um 1 no início da sequência, e a máquina para no estado de aceitação.

#### 4.1.5 Condições de Aceitação e Rejeição

A máquina aceita a palavra (ou seja, incrementa corretamente o número binário) quando chega ao estado  $q_{\text{Aceita}}$  após processar todos os dígitos binários. Não há condição explícita de rejeição; qualquer palavra que possa ser processada pela máquina resultará em um incremento binário válido.

## 4.2 Problema 2: Reconhecimento de Linguagem Regular (Número Par de 'a's)

### 4.2.1 Descrição do Problema

Este problema visa implementar uma Máquina de Turing que reconheça palavras pertencentes a uma linguagem regular que contém um número par de 'a's. A máquina deve percorrer a palavra e determinar se a quantidade de 'a's é par, aceitando ou rejeitando a palavra com base nessa verificação.

### 4.2.2 Definição do Alfabeto da Fita

- **Alfabeto de entrada:**  $\{a, b\}$
- **Alfabeto da fita:**  $\{a, b, \_ \}$  (onde  $\_$  representa o símbolo vazio)

### 4.2.3 Definição dos Estados

- **Estados:**  $\{q_0, q_1, q_{\text{Aceita}}, q_{\text{Rejeita}}\}$
- **Estado Inicial:**  $q_0$
- **Estado de Aceitação:**  $q_{\text{Aceita}}$

- **Estado de Rejeição:**  $q_{\text{Rejeita}}$

#### 4.2.4 Regras de Transição

As regras de transição para a máquina de reconhecimento de linguagem regular são as seguintes:

- $(q_0, a) \rightarrow (q_1, a, R)$ 
  - Se a máquina está no estado  $q_0$  e lê um 'a', ela mantém o 'a' na fita, move a cabeça para a direita e muda para o estado  $q_1$ . Este estado é usado para alternar entre a contagem par e ímpar de 'a's.
- $(q_0, b) \rightarrow (q_0, b, R)$ 
  - Se a máquina está no estado  $q_0$  e encontra um 'b', ela simplesmente ignora e continua para a direita no estado atual, já que a presença de 'b's não afeta a contagem de 'a's.
- $(q_1, a) \rightarrow (q_0, a, R)$ 
  - Se a máquina está no estado  $q_1$  e lê um 'a', ela mantém o 'a' na fita, move a cabeça para a direita e retorna ao estado  $q_0$ . Isso alterna o estado da máquina a cada 'a' lido, permitindo a contagem do número de 'a's.
- $(q_1, b) \rightarrow (q_1, b, R)$ 
  - O mesmo comportamento de ignorar 'b's se aplica aqui, mantendo o estado atual.
- $(q_0, -) \rightarrow (q_{\text{Aceita}}, -, S)$ 
  - Se a máquina está no estado  $q_0$  e encontra o símbolo vazio ( $-$ ), significa que a palavra terminou e o número de 'a's é par. A máquina então muda para o estado  $q_{\text{Aceita}}$ .
- $(q_1, -) \rightarrow (q_{\text{Rejeita}}, -, S)$

- Se a máquina está no estado  $q_1$  e encontra o símbolo vazio ( $\_$ ), significa que a palavra terminou com um número ímpar de 'a's. A máquina então muda para o estado  $q_{Rejeita}$ .

#### 4.2.5 Condições de Aceitação e Rejeição

A palavra é aceita se a máquina termina no estado  $q_{Aceita}$ , indicando que o número de 'a's na palavra é par. Se a máquina termina no estado  $q_{Rejeita}$ , a palavra é rejeitada, indicando que o número de 'a's é ímpar.

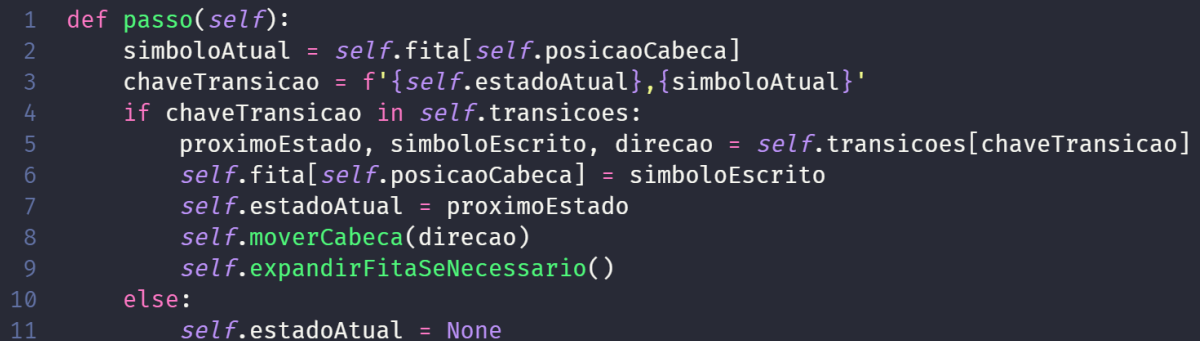
### 4.3 Implementação da Máquina de Turing

A seguir, apresento trechos do código e uma explicação do seu funcionamento.

```
1 def __init__(self, estados, alfabetoEntrada, alfabetoFita, simboloVazio, transicoes, estadoInicial, estadosFinais):
2     self.estados = estados
3     self.alfabetoEntrada = alfabetoEntrada
4     self.alfabetoFita = alfabetoFita
5     self.simboloVazio = simboloVazio
6     self.transicoes = transicoes
7     self.estadoInicial = estadoInicial
8     self.estadosFinais = estadosFinais
9     self.fita = []
10    self.estadoAtual = self.estadoInicial
11    self.posicaoCabeca = 0
```

Figura 1: Inicialização da Máquina de Turing

**Explicação:** Este trecho inicializa a Máquina de Turing com todos os parâmetros necessários. Ele define os estados, alfabetos, transições, e outros elementos essenciais para a operação da máquina. É aqui que o estado inicial e a posição da cabeça de leitura/gravação são configurados, preparando a máquina para processar a palavra de entrada.



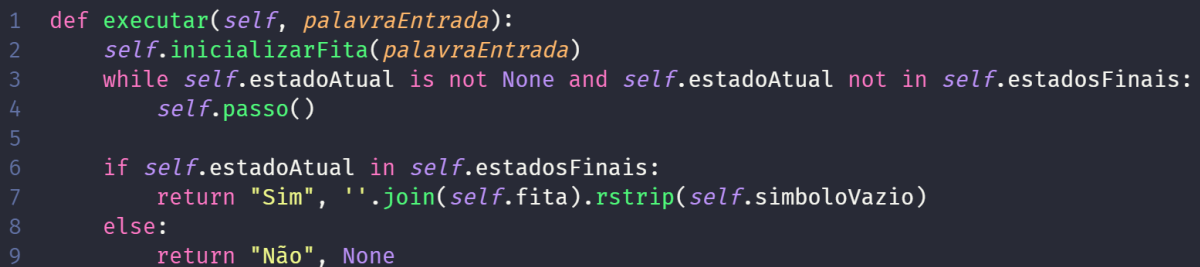
```

1  def passo(self):
2      simboloAtual = self.fita[self.posicaoCabeca]
3      chaveTransicao = f'{self.estadoAtual},{simboloAtual}'
4      if chaveTransicao in self.transicoes:
5          proximoEstado, simboloEscrito, direcao = self.transicoes[chaveTransicao]
6          self.fita[self.posicaoCabeca] = simboloEscrito
7          self.estadoAtual = proximoEstado
8          self.moverCabeca(direcao)
9          self.expandirFitaSeNecessario()
10     else:
11         self.estadoAtual = None

```

Figura 2: Execução de um Passo da Máquina de Turing

**Explicação:** Este método é fundamental para a operação da máquina, executando um único passo do processamento. Ele verifica o símbolo atual sob a cabeça de leitura, consulta as transições possíveis, e então atualiza o estado da máquina e a fita conforme necessário. Se não houver uma transição válida, a execução para, indicando que a palavra foi processada ou que a máquina não pôde continuar.



```

1  def executar(self, palavraEntrada):
2      self.inicializarFita(palavraEntrada)
3      while self.estadoAtual is not None and self.estadoAtual not in self.estadosFinais:
4          self.passo()
5
6      if self.estadoAtual in self.estadosFinais:
7          return "Sim", ''.join(self.fita).rstrip(self.simboloVazio)
8      else:
9          return "Não", None

```

Figura 3: Execução Completa da Máquina de Turing

**Explicação:** Este é o ponto de entrada para a execução completa da Máquina de Turing. O método inicializa a fita com a palavra de entrada e executa passos até que a máquina chegue a um estado final ou não possa mais prosseguir. O resultado é uma indicação se a palavra foi aceita e, se aplicável, a palavra resultante após o processamento.

A implementação foi projetada para seguir o conceito clássico de uma Máquina de Turing, com foco na modularidade e clareza. Cada componente (fita, cabeça, transições) é tratado separadamente, o que facilita o entendimento e a manutenção do código. A função

de execução permite uma simulação completa da máquina, o que é essencial para garantir que o incremento binário seja realizado corretamente.

#### 4.3.1 Diferença na Inicialização da Cabeça de Leitura

A única diferença entre as implementações da `MaquinaTuringIncrementoBinario` e da `MaquinaTuringLR` reside na posição inicial da cabeça de leitura na fita.

**MaquinaTuringIncrementoBinario:** Na implementação da `MaquinaTuringIncrementoBinario`, a cabeça de leitura é inicialmente posicionada no último bit da palavra de entrada, utilizando a seguinte linha de código:

```
self.posicaoCabeça = len(palavraEntrada) - 1
```

Isso ocorre porque o processo de incremento binário começa do bit menos significativo, ou seja, da direita para a esquerda. É necessário verificar e manipular os bits na extremidade direita da palavra primeiro, pois é ali que o incremento pode causar uma "propagação" de valores, alterando os bits subsequentes.

**MaquinaTuringLR:** Já na implementação da `MaquinaTuringLR`, a cabeça de leitura é inicialmente posicionada no começo da fita:

```
self.posicaoCabeça = 0
```

Esta escolha é apropriada para o reconhecimento de linguagem regular, onde a máquina começa a leitura da esquerda para a direita. Nesse caso, a palavra de entrada é processada a partir do primeiro símbolo, o que é necessário para verificar propriedades como a paridade de 'a's na palavra.

**Conclusão:** A posição inicial da cabeça de leitura é ajustada de acordo com a natureza do problema a ser resolvido. No incremento binário, o foco está no final da palavra, enquanto no reconhecimento de linguagem regular, o processamento começa desde o início.

## 5 Conclusão

Este projeto demonstrou a viabilidade de implementar um simulador interativo para autômatos e Máquina de Turing. O desenvolvimento modular e a utilização de Flask como framework permitiram uma estrutura clara e fácil de manter, além de fornecer uma ferramenta útil para estudantes na área de Teoria da Computação.