AutomaTop: Um Estudo sobre Autômatos e Máquinas de Turing

Yan Guilherme Viana Leite
5992

Vinicius de Oliveira Rocha
6036

Conteúdo

1 Introdução

Este trabalho tem como objetivo o estudo de autômatos, máquina de Turing e a implementação de um sistema para manipulação destes, além da melhora do entendimento deles para a computação. Este trabalho está sendo realizado no sétimo semestre de Sistemas de Informação, pela Universidade Federal de Viçosa campus Rio Paranaíba na disciplina de Teoria da Computação, ministrada pelo professor Pedro Damaso.

2 Estrutura do Sistema

O sistema está sendo estruturado usando a arquitetura cliente-servidor, onde há uma troca de informações entre os lados para que haja um bom desempenho e escalabilidade. A troca de informação entre os lados é feita através de arquivos no formato JSON (JavaScript Object Notation) que representa toda a estrutura do autômato.

3 Tecnologias Utilizadas

Para o desenvolvimento do back-end está sendo utilizado a linguagem de programação Java com o auxílio do framework Spring para facilitar a construção e configuração do projeto. Para o front-end, as tecnologias escolhidas são HTML 5, CSS 3 e JavaScript. O HTML é uma linguagem de marcação usada para estruturar toda a página de acesso do usuário. O CSS tem como objetivo alterar as formas dos elementos da tela para trazer uma boa experiência ao usuário.

O JavaScript é utilizado neste projeto para coletar todas as informações de entrada do usuário, além de fazer a dinamicidade da página, criando e removendo elementos. Outra função do JS é toda a comunicação com o servidor, que está sendo feita usando o AJAX, que processa as requisições ao servidor em segundo plano. Para salvar os autômatos criados, está sendo utilizado o SGBD H2 que armazena as informações somente na memória, ou seja, enquanto o back-end está ativo.

4 Back-End

4.1 Pacotes e Classes

Assim está estruturado nosso back-end, seguindo a metodologia MVC (Model, View, Controller). Apenas o View não está presente nesta implementação. O Controller tem o papel de mediar toda a comunicação entre o Model e o View. Model este que carrega consigo todas as regras de negócio e mantém restrito o acesso aos principais atributos. O pacote de config contém os arquivos de configuração do back-end.

A classe denominada AutomatoController é responsável por fazer a transferência dos dados entre o front e o back-end, contendo somente as informações que são úteis para esse compartilhamento.

Na pasta service estão as funcionalidades necessárias do projeto separadas para cada tipo de autômato. Algumas das funcionalidades são a escrita e a leitura de um registro no banco, a conversão entre tipos diferentes de autômatos, o teste de cadeias e a equivalência entre diferentes autômatos.

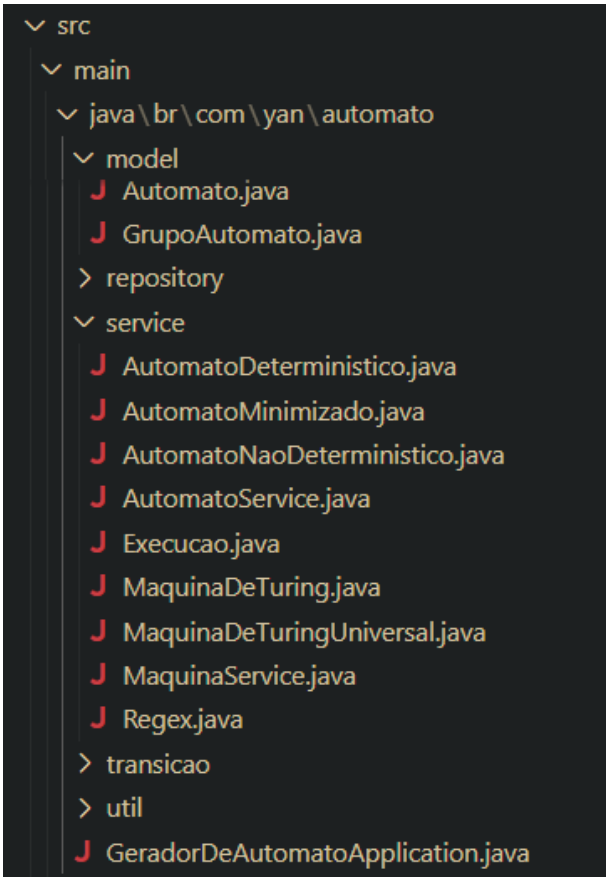

Figura 1: Pacotes e Classes

4.2 Classes Principais

4.2.1 Automato

Na imagem abaixo é possível ver como os atributos da classe foram escolhidos, principalmente na questão do tipo. Um autômato finito é definido por um conjunto de estados, um alfabeto, um conjunto de transições, um estado inicial e um conjunto de estados de aceitação.

- Os estados são um conjunto do tipo `String`, denominados por "q1, q2, q3,...".
- O alfabeto é um conjunto de `Character`, uma vez que um `char` representa um símbolo do alfabeto.
- Para as transições, a solução foi usar um `Map` que é definido por chave e valor. A chave (`String`) representa o estado atual do autômato e o valor é representado por outro `Map`, que utiliza a chave (`char`) para representar o símbolo lido e o valor (`String`) representa o conjunto de estados de destino dado aquela chave.
- Para o AFN, a mudança ocorre apenas no valor do `Map` interno, que passa a ser uma coleção de `String`, devido à natureza do AFN.
- Os estados de aceitação são formados por um conjunto de `String`, que é selecionado a partir do conjunto de estados.

- O estado inicial é composto apenas por uma `String`, respeitando a restrição de ser apenas um estado do conjunto inicial.

```java
@JsonTypeInfo(
        use = JsonTypeInfo.Id.NAME,
        property = "tipo"
)
@JsonSubTypes({
        @JsonSubTypes.Type(value = AutomatoDeterministico.class, name = "AFD"),
        @JsonSubTypes.Type(value = AutomatoNaoDeterministico.class, name = "AFN"),
})
@Document(collection = "automato")
@JsonIgnoreProperties(ignoreUnknown = true)
public abstract class Automato {
    @Id
    private String id;
    protected String estadoInicial;
    protected Set<String> estadosAceitacao = new HashSet<>();
    protected String nome;
```

Figura 2: Classe Automato

4.2.2 Equivalencia

A classe EquivalenciaService está vinculada com a equivalência entre autómatos, onde a função testarEquivalencia(...) realiza dois testes para determinar se os dois autómatos são de fato equivalentes.

As classes AlfabetoValidator e LinguagemValidator implementam a função validade, herdada de EquivalenceInterface, que, respectivamente, valida a equivalencia do alfabeto e da linguagem dos autómatos.

```java
@Override
public TesteEquivalencia validate(Automato automato1, Automato automato2) {
    Set<String> linguagem = gerarLinguagem(automato1.getId(), automato2.getId());
    for(String cadeia: linguagem){
        if(!automato1.validarProcessar(cadeia).equals(automato2.validarProcessar(cadeia))){
            return new TesteEquivalencia(sucesso:false, "Falhou na cadeia: " + cadeia, nome:"Reconhe
        }
    }
    return new TesteEquivalencia(sucesso:true, message:"Sim", nome:"Reconhecem mesma linguagem?");
}
```

Figura 3: Método validate()

4.2.3 Pacote Service

O pacote service contem todas as funcionalidades disponíveis no sistema, a classe AutomatoService cuida das funcionalidades que são iguais a todos os autómatos (conversão, minimização, ...). As outras classes de automatos e regex cuidam apenas das funcionalidades especificas de cada um.

As classes AutomatoDeterministico e AutomatoNaoDeterministico possuem os métodos especificos para cada tipo de autómato, cada um cuidando das especificações de sua respectiva classe.

Na imagem 4 é possível analisar como o método save(...) funciona. Ele recebe como parâmetro um objeto do tipo DTO (Data Transfer Object) e atribui os seus valores a classe do modelo através dos métodos getters e setters.

```
public Automato save(Automato automato){
    return repository.save(automato);
}
```

Figura 4: Método save()

4.2.4 AutomatoController

A classe AutomatoController está encarregada de controlar as requisições do front-end, acessar o automato no BD, salvar um automato no BD, dentre outras.

A figura 5 é a representação do endpoint "/create" que é a escrita de um registro no banco de dados, ela recebe o retorno da função representada na imagem 3, salva em um objeto para ser retornado ao back-end com o status code 200, que significa um OK, ou seja, transação bem sucedida.

```
@PostMapping("/save")
public Automato create(@RequestBody Automato automato) {
    return automatoService.save(automato);
}
```

Figura 5: Método create()

O endpoint que busca por todos os registros no banco, é denominado de "/findAll", ele está representado na figura 5. Ele usa o método implementado na classe de serviço que apenas retorna todos os registros do banco, com esse retorno o endpoint cria uma lista de autômatos e manda de volta ao front-end.

```
@GetMapping("/findAll")
public List<Automato> findAll() {
    List<Automato> automatos = automatoService.findAll();
    return automatos;
}
```

Figura 6: Método findAll()

4.2.5 model/Automato

A classe autómato, dentro do pacote model está encarregada de cuidar do arquivo .json que será gravado no mongoDB. São gravados os atributos de ID, nome, estado inicial e aceitação e também as transições entres os estados.

Também serve como uma estrutura base para diferentes tipos de autômatos, fornecendo métodos de validação, processamento de cadeias e gerenciamento de estados. Ela é

projetada para ser estendida por classes concretas que implementarão os comportamentos específicos para autômatos determinísticos e não determinísticos.

```java
@JsonSubTypes({
        @JsonSubTypes.Type(value = AutomatoDeterministico.class, name = "AFD"),
        @JsonSubTypes.Type(value = AutomatoNaoDeterministico.class, name = "AFN"),
})
@Document(collection = "automato")
@JsonIgnoreProperties(ignoreUnknown = true)
public abstract class Automato {
    @Id
    private String id;
    protected String estadoInicial;
    protected Set<String> estadosAceitacao = new HashSet<>();
    protected String nome;
```

Figura 7: Classe Automato Modelo JSON

4.2.6 Regex

Está classe trata das expressões regulares, transformando uma expressão regular em um AFN, por meio do algoritmo de thompson.

gerarAFN(...): Método principal que, dado uma expressão regular, limpa a expressão (removendo espaços e barras invertidas), adiciona concatenações implícitas e constrói o AFN correspondente.

construirAFN(...): Constrói o AFN a partir da expressão regular utilizando duas pilhas: uma para armazenar os AFNs parciais e outra para operadores. Conforme a expressão é processada, os AFNs são combinados (por concatenação, alternância ou fechamento de Kleene) até que o AFN final seja gerado.

```java
@Autowired
private AutomatoService automatoService;

public AutomatoNaoDeterministico gerarAFN(String expressaoRegular) {
    // Limpa a expressão removendo quebras de linha, espaços em branco extras e barras invertidas
    String expressaoLimpa = expressaoRegular.replaceAll(regex:"\\s+", replacement:"").replace(target
    // Insere concatenação implícita
    String expressaoComConcatenacao = adicionarConcatenacaoImplicita(expressaoLimpa);
    return construirAFN(expressaoComConcatenacao);
```

Figura 8: Método gerarAFN()

5 Front-End

5.1 Automato

Exibindo detalhes dos automatos: estado inicial, estados de aceitação, transições, e uma representação gráfica (usando viz.js).

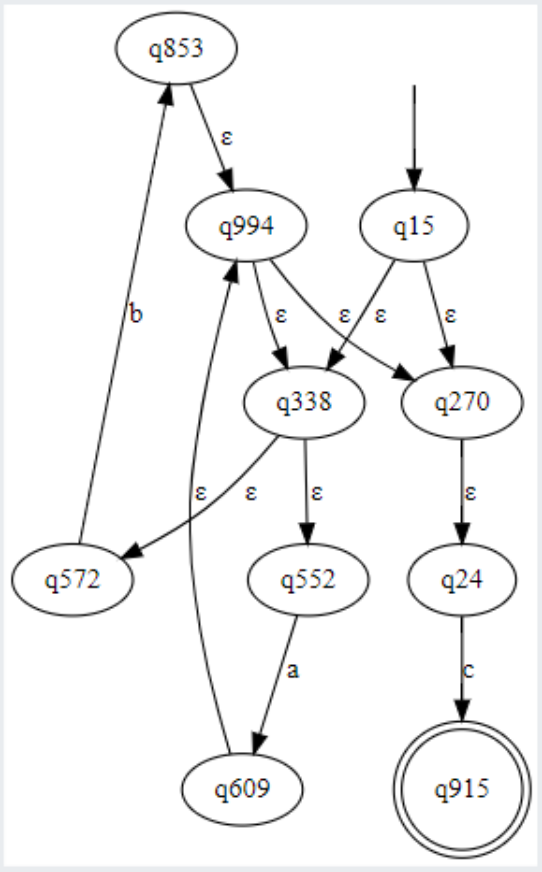

Figura 9: Representação Gráfica AFN

5.2 Equivalencia

Exibindo um teste de equevalencia: comparando os automatos 0*.1.(0+1)* e (a+b)*.c

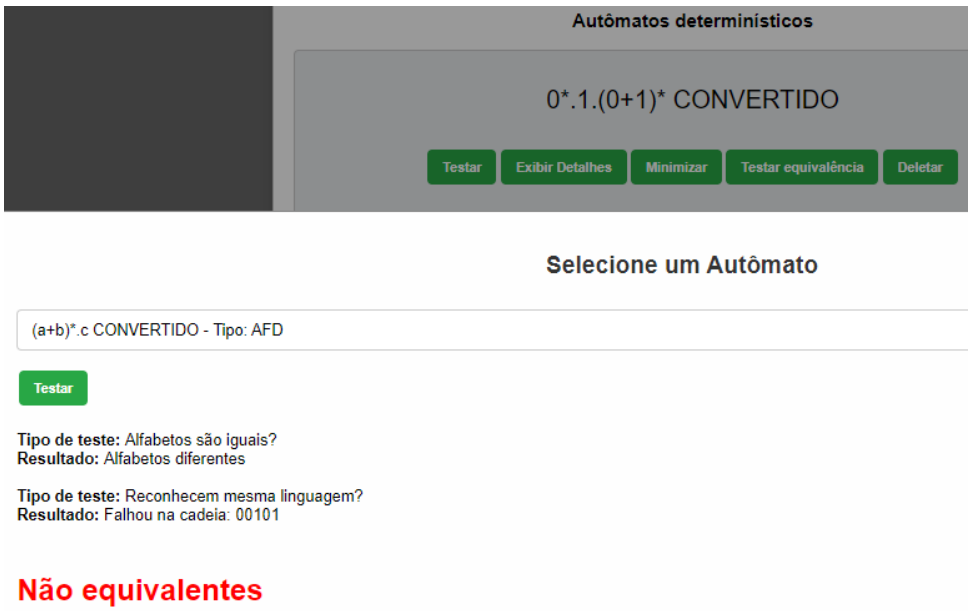

Figura 10: Teste de Equivalencia

5.3 Maquina de Turing

Exibindo detalhes das maquinas de turing: mostra os detalhes de cada máquina, formatado em uma tabela para as transições e uma representação gráfica (usando viz.js).

Nome: palindromo binario

Tipo: MT

Estados Q: q0, q1, q2, q3, q4, q5, q6, q7

Alfabeto Σ: 0, 1, x, y, -

Transições:

δ	0	1	x	y	-
->q0	q1, -, R	q2, -, R	-	-	q8, -, S
q1	q3, 0, R	q3, 1, R	-	-	q8, -, S
q2	q4, 0, R	q4, 1, R	-	-	q8, -, S
q3	q3, 0, R	q3, 1, R	-	-	q5, -, L
q4	q4, 0, R	q4, 1, R	-	-	q6, -, L
q5	q7, -, L	-	-	-	-
q6	-	q7, -, L	-	-	-
q7	q7, 0, L	q7, 1, L	-	-	q0, -, R

Estado inicial: q0

Estados de aceitação: q8

Figura 11: Tabela de Transições

6 Maquina de Turing

Nos optamos por desenvolver uma maquina de turing universal, devido as capacidades de simular qualquer outra maquina de turing já que não limitada por um conjunto especifico de regras ou linguagem.

```java
@Document(collection = "maquinasDeTuring")
public class MaquinaDeTuring{
    @Id
    private String id;
    private String estadoInicial;
    private Set<String> estadosAceitacao = new HashSet<>();
    private String nome;
    private Character x;
    private Character y;
    private Set<Character> alfabetoFita;
    private Map<String, Map<Character, TransicaoMT>> transicoes = new HashMap<>
```

Figura 12: Classe Maquina de Turing

6.1 Exemplo 1

6.1.1 Descrição do Problema

O problema consiste em reconhecer a linguagem formada por cadeias da forma $a^n b^n$, onde $n \geq 1$. A linguagem contém cadeias como "ab", "aabb", "aaabbb", e assim por diante. A Máquina de Turing deve verificar se o número de caracteres a é igual ao número de caracteres b e se todos os a vêm antes dos b.

6.1.2 Alfabeto da Fita

O alfabeto da fita consiste nos símbolos $\{a, b, x, y, -\}$, onde a e b são os símbolos da linguagem, x e y são marcadores auxiliares usados pela Máquina de Turing durante o processo de cálculo, e $-$ representa o símbolo de espaço em branco (ou vazio).

6.1.3 Estados

A Máquina de Turing pode ser definida com os seguintes estados:

- Estados Q: q0, q1, q2, q3

6.1.4 Regras de Transição

As regras de transição podem ser descritas da seguinte forma:

δ	a	b	x	y	-
->q0	q1, x, R	-	-	q3, y, R	-
q1	q1, a, R	q2, y, L	-	q1, y, R	-
q2	q2, a, L	-	q0, x, R	q2, y, L	-
q3	-	-	-	q3, y, R	q4, -, R

Figura 13: Tabela de Transições

6.1.5 Condições de Aceitação

A Máquina de Turing aceita a cadeia se ela tiver marcado todos os a e b corretamente, isto é, se o número de a for igual ao número de b, e todos os a vierem antes de todos os b.

6.2 Exemplo 2

6.2.1 Descrição do Problema

O problema consiste em reconhecer se uma cadeia de caracteres formada pelos símbolos 0 e 1 pertence à linguagem dos palíndromos, ou seja, cadeias que são lidas da mesma forma de frente para trás e de trás para frente. Exemplos incluem "0110"e "010". A Máquina de Turing deve verificar se a sequência de caracteres é um palíndromo.

6.2.2 Alfabeto da Fita

O alfabeto da fita consiste nos símbolos $\{0, 1, x, y, -\}$, onde 0 e 1 são os símbolos da linguagem, x e y é um marcador auxiliar utilizado pela Máquina de Turing durante o processo de verificação, e $-$ representa o símbolo de espaço em branco.

6.2.3 Estados

A Máquina de Turing pode ser definida com os seguintes estados:

- Estados Q: q0, q1, q2, q3, q4, q5, q6, q7

6.2.4 Regras de Transição

As regras de transição podem ser descritas da seguinte forma:

δ	0	1	x	y	-
->q0	q1, -, R	q2, -, R	-	-	q8, -, S
q1	q3, 0, R	q3, 1, R	-	-	q8, -, S
q2	q4, 0, R	q4, 1, R	-	-	q8, -, S
q3	q3, 0, R	q3, 1, R	-	-	q5, -, L
q4	q4, 0, R	q4, 1, R	-	-	q6, -, L
q5	q7, -, L	-	-	-	-
q6	-	q7, -, L	-	-	-
q7	q7, 0, L	q7, 1, L	-	-	q0, -, R

Figura 14: Tabela de Transições

6.2.5 Condições de Aceitação

A Máquina de Turing aceita a cadeia se, após comparar e marcar todos os caracteres correspondentes, restarem apenas espaços em branco na fita. Isso indica que a cadeia lida é a mesma de trás para frente, caracterizando um palíndromo.