UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

CAIO SOARES COSTA
Orientador: Rodrigo Geraldo Ribeiro

DESENVOLVIMENTO DE ALGORITMOS PARA CORREÇÃO AUTOMÁTICA DE EXERCÍCIOS SOBRE AUTÔMATOS FINITOS DETERMINÍSTICOS

Ouro Preto, MG
2023

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

CAIO SOARES COSTA

DESENVOLVIMENTO DE ALGORITMOS PARA CORREÇÃO AUTOMÁTICA DE EXERCÍCIOS SOBRE AUTÔMATOS FINITOS DETERMINÍSTICOS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Rodrigo Geraldo Ribeiro

Ouro Preto, MG
2023

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Caio Soares Costa

Desenvolvimento de algoritmos para correção automática de exercícios sobre autômatos finitos determinístico

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 28 de Agosto de 2023.

Membros da banca

Rodrigo Geraldo Ribeiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Reinaldo Silva Fortes (Examinador) - Doutor - Universidade Federal de Ouro Preto
Leonardo Vieira dos Santos Reis (Examinador) - Doutor - Universidade Federal de Juiz de Fora

Rodrigo Geraldo Ribeiro, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 28/08/2023.

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR 3 GRAU**, em 29/08/2023, às 08:52, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0577039** e o código CRC **10AF3EBA**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Resumo

Disciplinas que lecionam teoria de autômatos tem sido, tradicionalmente, ensinadas usando lápis e papel. Com isso, alunos podem ter mais dificuldades de entendimento desse conteúdo, além de ficarem frustrados por não aprenderem. Ademais, o fato de esses estudantes usarem papel e lápis, professores e tutores acabam gastando mais tempo na correção de exercícios dessa matéria. Dessa forma, foi proposto a criação de uma ferramenta usando a linguagem Racket para auxiliar alunos na aprendizagem de teoria de autômatos, sendo isso através de uma funcionalidade que faz a correção automática de exercícios sobre a criação de autômatos finitos determinísticos. A forma de auxiliar esses alunos é dando um contraexemplo que prova que o autômato criado pelo estudante esta incorreto. Isso é possível devido ás propriedades de fechamento que autômatos possuem.

Palavras-chave: Teoria de Autômatos. Autômatos Finitos Determinísticos. Contraexemplo. Propriedade de Fechamento

Lista de Ilustrações

Figura 2.1 – Imagem que representa um grafo geral usando Dot 14
Figura 3.1 – Autômato A 19
Figura 3.2 – Autômato G 19
Figura 3.3 – Autômato A 21
Figura 3.4 – Autômato G 21
Figura 3.5 – Imagem que representa um autômato finito determinístico 26

Lista de Algoritmos

1 Cáculo de Fatorial em Racket . . . 12
2 Grafo Geral em Dot . . . 13
3 Autômato em Dot . . . 23
4 Descrição de um AFD . . . 24
5 Macro . . . 24
6 Produto de AFDs . . . 26
7 Algoritmo para encontrar os estados alcançáveis a partir do estado inicial . . . 27
8 Interseção de 2 AFDs . . . 28
9 União de 2 AFDs . . . 29
10 Complemento de um AFD . . . 30
11 Inverso de uma transição . . . 30
12 Inverso de uma lista de transições . . . 30
13 Algoritmo de minimização . . . 31
14 Algoritmo de minimização . . . 31
15 Diferença entre Autômatos . . . 31
16 Transforma um Automato em um Grafo . . . 31
17 Encontra Caminho Entre um Vértice ao Vértice de Origem . . . 32
18 Palavras Aceitas . . . 32
19 Algoritmo Correção . . . 33
20 Autômato do JSON para Estrutura Racket . . . 35
21 Transição do JSON para Estrutura Racket . . . 36
22 Lista de Pares de Respostas . . . 36
23 Correção de Múltiplos Exercícios . . . 36

Lista de Abreviaturas e Siglas

ABNT	Associação Brasileira de Normas Técnicas
DECOM	Departamento de Computação
UFOP	Universidade Federal de Ouro Preto
AFD	Autômato finito determinístico
AFN	Autômato finito não-determinístico
JFLAP	Java Formal Language and Automata Package
AP	Autômato de pilha
JCT	Java Computability Toolkit
XML	Extensible Markup Language
IDE	Integrated Development Environment
Dot	Graph Description Language

Sumário

1 Introdução **1**
1.1 Objetivos 1
1.2 Organização do Trabalho 2
2 Revisão Bibliográfica **3**
2.1 Fundamentação Teórica 3
2.1.1 Linguagens Regulares e Autômatos Finitos 3
2.1.1.1 Autômatos Finitos Determinísticos 3
2.1.1.2 Autômatos Finitos Não-determinísticos 5
2.1.1.3 Equivalência entre AFNs e AFDs 6
2.1.1.4 Propriedades de Fechamento 7
2.1.1.5 Algoritmo de minimização de *Brzozowski* 10
2.2 Tecnologias utilizadas 12
2.2.1 Introdução à linguagem Racket 12
2.2.2 Introdução ao GraphViz e a linguagem Dot 13
2.3 Trabalhos Relacionados 14
2.3.1 *Java Formal Language and Automata Package* (JFLAP) 14
2.3.2 *FSA Simulator* 15
2.3.3 *The Java Computability Toolkit* (JCT) 15
2.3.4 *Automated Grading of DFA Constructions* 16
2.3.5 Experimentos em Salas de Aulas 16
3 Desenvolvimento **18**
3.1 Correção Automática de AFDs 18
3.2 Detalhes de Implementação 22
3.2.1 Visualização do Autômato Finito 22
3.2.1.1 Desenho do Autômato Finito 22
3.2.1.2 Descrição do Autômato 23
3.2.2 Operações em Autômatos Finitos 26
3.2.2.1 Funções sobre Propriedade de Fechamento 26
3.2.2.1.1 Produto de Dois AFDs 26
3.2.2.1.2 Interseção de Dois AFDs 27
3.2.2.1.3 União de Dois AFDs 28
3.2.2.1.4 Complemento de um AFDs 29
3.2.2.2 Função de Minimização 30
3.2.3 Algoritmo de Correção 31
3.2.3.1 Impressão de Palavras Encontradas 31
3.2.3.2 Execução de Correção de Múltiplos Exercícios 33

3.2.3.2.1 Formatação do Arquivo de Respostas 33
3.2.3.2.2 Correção de Múltiplos Exercícios 35
3.2.3.2.3 Exemplo de Execução 37
4 Considerações Finais 41

Referências 42

1 Introdução

A teoria de linguagens formais e autômatos é parte central do conteúdo de cursos de Ciência da Computação e Engenharia da Computação por fornecer a fundamentação matemática para o estudo de decidibilidade e intratabilidade de problemas (KOZEN, 1997; HOPCROFT; MOTWANI; ULLMAN, 2006). Usualmente, alunos de cursos de computação possuem dificuldades em compreender plenamente os conceitos da área devido ao nível de maturidade matemática e abstração envolvidos. Neste sentido, uma estratégia para mitigar esses problemas é o uso de muitos exemplos e a proposição de vários exercícios para solução por parte dos estudantes.

Porém, o uso de lápis e papel no ensino de teoria de autômatos tem sido o principal método de ensino utilizado. Como consequência, alunos ao tentar resolver exercícios podem se frustrar por não receber um *feedback* imediato indicando se sua solução está correta ou não, visto que muitas vezes a validação destes ocorre somente quando do momento de correção de exercícios avaliativos por parte do docente. Outro inconveniente é que autômatos para modelagem de processos de automação industrial ou outros problemas de cunho prático tendem a ser grandes, o que pode demandar muito tempo para elaboração de uma solução ou mesmo sua correção. Essa dificuldade faz com que docentes utilizem pequenos exemplos que não são úteis para ilustrar como o conteúdo pode ser utilizado para a solução de problemas práticos.

Uma alternativa para todas as questões levantadas é a utilização de ferramentas educacionais para auxiliar na solução de exercícios. Em especial, neste trabalho pretendemos desenvolver uma ferramenta para correção automática de exercícios sobre autômatos finitos determinísticos, um dos primeiros tópicos vistos em cursos sobre linguagens formais e autômatos.

1.1 Objetivos

Este trabalho tem como objetivo desenvolver uma ferramenta para correção automática de exercícios sobre autômatos finitos determinísticos. A ferramenta é ser capaz de produzir contra-exemplos, em caso do exercício submetido estar incorreto, para auxiliar o aluno na construção da solução adequada. O processo de correção será baseado em fundamentos matemáticos conhecidos da área de linguagens formais: propriedades de fechamento de linguagens regulares.

Dessa forma, objetivos do trabalho são:

- Desenvolver algoritmos que apliquem propriedades de fechamento nos autômatos
- Desenvolver um algoritmo de minimização de autômatos
- Desenvolver um algoritmo para correção automática de autômatos finitos determinísticos

1.2 Organização do Trabalho

Este trabalho monográfico é organizado em 4 capítulos, a saber:

Capítulo 1: Introdução.

Capítulo 2: Revisão Bibliográfica (com o referencial teórico, trabalhos relacionados e tecnologias utilizadas).

Capítulo 3: Desenvolvimento (material e métodos).

Capítulo 4: Conclusão (e trabalhos futuros).

Todo o código produzido como parte deste trabalho está presente no seguinte repositório on-line:

<https://github.com/lives-group/automata-language>

2 Revisão Bibliográfica

Neste capítulo apresentaremos os principais conceitos teóricos necessários para a compreensão do trabalho desenvolvido. A Seção 2.1 revisa conceitos sobre linguagens formais e autômatos utilizados na construção da ferramenta proposta e a Seção 2.3 discute trabalhos relacionados presentes da literatura.

2.1 Fundamentação Teórica

2.1.1 Linguagens Regulares e Autômatos Finitos

Inicia-se essa seção com alguns conceitos importantes da teoria de linguagens formais.

Definição 2.1 (Alfabeto)**.** Um alfabeto Σ é um conjunto finito não vazio de símbolos.

Definição 2.2 (Palavra)**.** Uma palavra é sequência finita de símbolos de um determinado alfabeto. A notação $|w|$ denota o número de símbolos que forma a palavra w e λ representa a palavra vazia, isto é, $|\lambda| = 0$.

Definição 2.3 (Linguagem)**.** Uma linguagem é um conjunto de palavras sobre um alfabeto. Definimos o conjunto de todas as palavras sobre um alfabeto Σ como Σ^*.

Exemplo 2.1. Exemplos de possíveis alfabetos são $\Sigma_1 = \{0, 1\}$ ou $\Sigma_2 = \{a, e, i, o, u\}$. Note que conjuntos como $\emptyset$ ou $\mathbb{N}$ não podem ser considerados como alfabetos.

Considere o alfabeto $\Sigma_1 = \{0, 1\}$. Exemplos de palavras sobre esse alfabeto são $w_1 = 110$, $w_2 = 00$ e $w_3 = \lambda$. Ainda considerando o alfabeto Σ_1, a seguir será apresentado alguns exemplos de linguagens: $\{110, 00, \lambda, 1\}$, $\emptyset$ e $\{0^n \mid n \geq 10\}$.

Intuitivamente, um autômato pode ser entendido como uma máquina abstrata para determinar se uma certa palavra pertence ou não a uma linguagem. A seguir, será apresentada a definição formal e exemplos de autômatos finitos.

2.1.1.1 Autômatos Finitos Determinísticos

Segundo (SIPSER, 2013) um AFD (Autômato Finito Determinístico) pode ser definido formalmente como uma quíntupla $M = (E, \Sigma, \delta, i, F)$, conforme a definição a seguir:

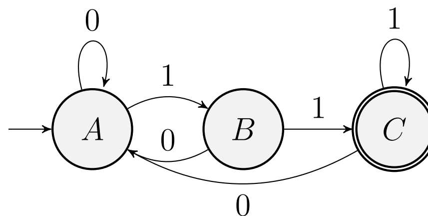
Definição 2.4 (Autômato Finito Determinístico)**.** Um AFD é uma quíntupla $(E, \Sigma, \delta, i, F)$, em que:

- E: é o conjunto de estados.

- Σ: é o alfabeto.
- $\delta : E$ x $\Sigma \rightarrow E$: função de transição, uma função total.
- $i \in E$: é o estado inicial.
- $F \subseteq E$: conjunto de estados finais.

Exemplo 2.2.

Considere a seguinte linguagem sobre $\Sigma = \{0, 1\}$: $L = \{0, 1\}^*\{11\}$. Ou seja, a linguagem de palavras sobre $\{0, 1\}$ que terminam em 11. Um AFD que reconhece palavras desta linguagem é apresentado a seguir.



Usualmente, AFDs são apresentados utilizando grafos direcionados em que nós denotam estados do autômato e arestas rotuladas sua função de transição. Formalmente, o AFD anterior é descrito pela seguinte quíntupla: $(\{A, B, C\}, \{0, 1\}, \delta, A, \{C\})$, em que a função de transição δ é definida pelas seguinte tabela:

δ	0	1
A	A	B
B	A	C
C	A	C

Para definir a linguagem aceita por uma AFD, é necessário estender a função de transição, que opera sobre símbolos de um alfabeto, para palavras. A definição a seguir apresenta este conceito.

Definição 2.5 (Função de transição estendida)**.** Seja $M = (E, \Sigma, \delta, i, F)$ um AFD qualquer. A função de transição estendida para M, $\hat{\delta} : E \times \Sigma^* \rightarrow E$, retorna o estado $e_2 \in E$ no qual a computação de uma palavra w termina em M ao ser inciado o processamento de w em um estado e. Formalmente, $\hat{\delta}$ é definido por recursão sobre a estrutura da palavra w.

$$\hat{\delta}(e, \lambda) = e \quad (1)$$
$$\hat{\delta}(e, ay) = \hat{\delta}(\delta(e, a), y), \ a \in \Sigma, y \in \Sigma^* \quad (2)$$

Exemplo 2.3. Considere o AFD apresentado no exemplo 2.2. Utilizando a função de transição estendida podemos obter o estado em que a computação deste AF termina com a palavra 011 ao iniciar seu processamento no estado inicial, como se segue:

$$\begin{aligned}
&\hat{\delta}(A, 011) && = \text{equação (2) de } \hat{\delta} \\
&\hat{\delta}(\delta(A, 0), 11) && = \text{def. de } \delta \\
&\hat{\delta}(A, 11) && = \text{equação (2) de } \hat{\delta} \\
&\hat{\delta}(\delta(A, 1), 1) && = \text{def. de } \delta \\
&\hat{\delta}(B, 1) && = \text{equação (2) de } \hat{\delta} \\
&\hat{\delta}(\delta(B, 1), \lambda) && = \text{def. de } \delta \\
&\hat{\delta}(C, \lambda) && = \text{equação (1) de } \hat{\delta} \\
&C
\end{aligned}$$

Utilizando a função de transição estendida, podemos definir a linguagem aceita por um AFD.

Definição 2.6 (Linguagem aceita por um AFD)**.** Seja $M = (E, \Sigma, \delta, i, F)$ um AFD qualquer. A linguagem aceita por M, $L(M)$, é definida como:

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(i, w) \in F\}$$

2.1.1.2 Autômatos Finitos Não-determinísticos

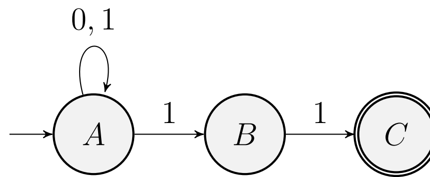
Ainda de acordo com (SIPSER, 2013) um AFN (Autômato Finito não-determinístico) é definido de maneira similar a um AFD. A principal diferença entre ambos está no fato que em um AFN uma transição pode produzir um conjunto de estados ao invés de um único como em um AFD.

Definição 2.7 (Autômato finito Não-determinístico)**.** Um AFN é uma quíntupla $(E, \Sigma, \delta, I, F)$, em que:

- E: é o conjunto de estados.
- Σ: é o alfabeto.
- $\delta : E \text{ x } \Sigma \rightarrow \mathcal{P}(E)$: função total de transição.
- $I \subseteq E$: conjunto de estados iniciais.
- $F \subseteq E$: conjunto de estados finais.

Exemplo 2.4. Para ilustrar a definição de AFN, considere a seguinte linguagem sobre $\Sigma = \{0, 1\}$:

$$L = \{0, 1\}^*\{11\}$$



isto é, palavras sobre $\{0, 1\}$ que terminam em 11. Um possível AFN para reconhecer esta linguagem é representado pelo seguinte diagrama de estados:

que pode ser descrito pela seguinte quíntupla $M = (\{A, B, C\}, \{0, 1\}, \delta, \{A\}, \{C\})$, em que:

δ	0	1
A	$\{A\}$	$\{A, B\}$
B	$\emptyset$	$\{C\}$
C	$\emptyset$	$\emptyset$

2.1.1.3 Equivalência entre AFNs e AFDs

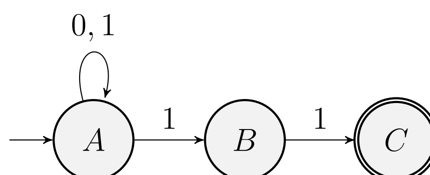
Em muitas situações, AFNs permitem uma representação concisa de reconhecedores para linguagens. Porém, essa facilidade de representação não implica que AFNs possuem um maior poder computacional que AFDs. É possível converter um AFN qualquer em um AFD que reconhece a mesma linguagem que o AFN original.

Definição 2.8 (AFD equivalente a um AFN)**.** Seja $M = (E, \Sigma, \delta, I, F)$ um AFN qualquer. O AFD $M' = (E', \Sigma, \delta', i', F')$ é equivalente ao AFN M, se:

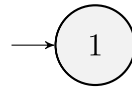
- $E' = \mathcal{P}(E)$
- $\delta'(X, a) = \bigcup_{e \in X} \delta(e, a)$, $\delta'(\emptyset, a) = \emptyset$, para $a \in \Sigma$.
- $i' = I$.
- $F' = \{X \subseteq E \,|\, X \cap F \neq \emptyset\}$.

O exemplo a seguir mostra como usar a definição2.8 para obter um AFD equivalente a um AFN.

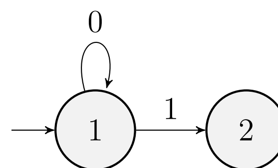
Exemplo 2.5. Considere novamente o seguinte AFN para a linguagem $\{0, 1\}^*\{11\}$:



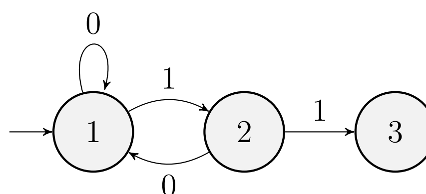
Usando a **Definição 2.8**, é possível construir um AFD equivalente ao AFN descrito acima. Contudo, como o conjunto de estados do AFD é $\mathcal{P}(E)$, este irá possuir um número exponencial de estados, sendo que muitos destes estados podem ser inalcançáveis a partir do estado inicial. Para evitar estes estados inúteis, pode-se adicionar estados sob demanda, começando pelo conjunto de estados iniciais do AFN. O rótulo 1 será adotado para denotar o conjunto $\{A\}$.



Note que no AFN, $\delta(A, 1) = \{A, B\}$ e $\delta(A, 0) = \{A\}$. Logo, deve-se incluir um estado relativo ao conjunto $\{A, B\}$, que possuirá o rótulo 2.

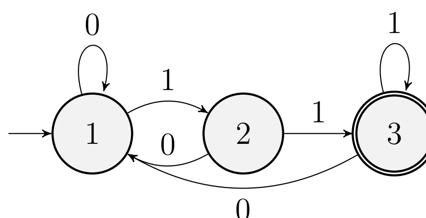


Assim, são finalizadas as transições a partir do estado 1, que representa o conjunto $\{A\}$. Para o estado 2, denotado por $\{A, B\}$, tem-se que: $\delta(\{A, B\}, 0) = \{A, B, C\}$ e $\delta(\{A, B\}, 1) = \{A\}$. Assim, é necessário adicionar um novo estado que corresponde ao conjunto $\{A, B, C\}$, sendo ele chamado de 3.



Por fim, ao incluir as transições para o estado 3, obtêm-se o AFD equivalente. As transições incluídas são: $\delta(\{A, B, C\}, 1) = \{A, B, C\}$ e $\delta(\{A, B, C\}, 0) = \{A\}$.

Como não há estados sem transições a serem incluídas, termina-se a construção. Agora, falta apenas indicar os estados finais, conforme abaixo.



2.1.1.4 Propriedades de Fechamento

Propriedades de fechamento são um conceito importante na teoria de autômatos pois ilustram como esse formalismo é composicional. Primeiramente, é necessário definir o conceito

de linguagem regular e na sequência será mostrado sobre quais operações a classe das linguagens regulares é fechada.

Definição 2.9 (Linguagem regular)**.** Seja $L \subseteq \Sigma^*$ uma linguagem qualquer sobre um alfabeto Σ. É dito que L é uma linguagem regular se existe um AFD M tal que $L(M) = L$.

A classe das linguagens regulares é fechada sobre diversas operações. Para fins deste trabalho monográfico, será apresentado o fechamento sobre as operações de união, interseção, complementação e reverso. Seguindo (KOZEN, 1997), será definido o fechamento sobre a união e interseção em termos da operação de produto, descrita a seguir.

Definição 2.10 (Produto de AFDs)**.** Sejam $M_1 = (E_1, \Sigma, \delta_1, i_1, F_1)$ e $M_2 = (E_2, \Sigma, \delta_2, i_2, F_2)$ dois AFDs quaisquer. O Produto de M_1 por M_3 é definido como sendo o AFD $M_3 = (E_3, \Sigma, \delta_3, i_3, F_3)$, em que:

- $E_3 = E_1 \times E_2$
- $\delta_3((e_1, e_2), a) = (\delta_1(e_1, a), \delta_2(e_2, a))$
- $i_3 = (i_1, i_2)$
- $F_3 = F_1 \times F_2$, se é desejado obter o AFD para $L(M_1) \cap L(M_2)$ ou $F_3 = (E_1 \times F_2) \cup (F_1 \times E_2)$ se é desejado um AFD para $L(M_1) \cup L(M_2)$.

O fechamento sobre a complementação consiste em inverter estados finais e não finais em um AFD.

Definição 2.11 (Complemento de AFD)**.** Sejam $M = (E, \Sigma, \delta, i, F)$ um AFD que reconhece a linguagem $L(M)$. O AFD $M' = (E, \Sigma, \delta, i, E - F)$ reconhece $\overline{L(M)}$.

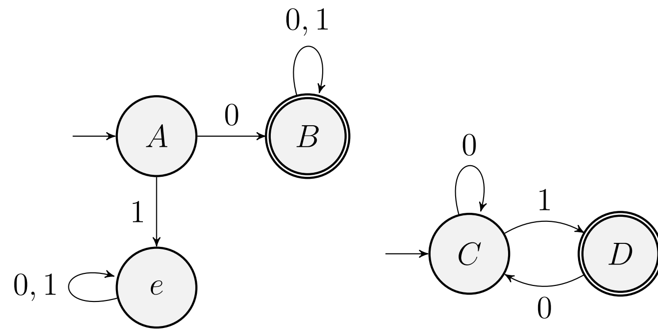
Note que a partir do fechamento sobre a interseção e complementação, temos que a classe das linguagens regulares é fechada sobre a diferença pois para quaisquer conjuntos A e B temos que $A - B = A \cap \overline{B}$.

Outra propriedade de fechamento de linguagens regulares que será importante neste trabalho é a de reverso, descrita a seguir.

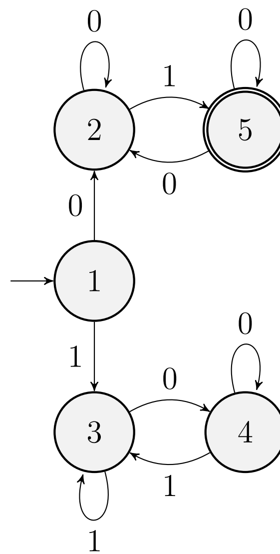
Definição 2.12 (Reverso de um AFD)**.** Seja $M = (E, \Sigma, \delta, i, F)$ um AFD qualquer que aceita a linguagem $L(M)$. O AFN $M' = (E, \Sigma, \delta', F, \{i\})$ aceita a linguagem $L(M)^R$, em que:

$$\delta'(e', a) = \{e \in E \mid \delta(e, a) = e'\}$$

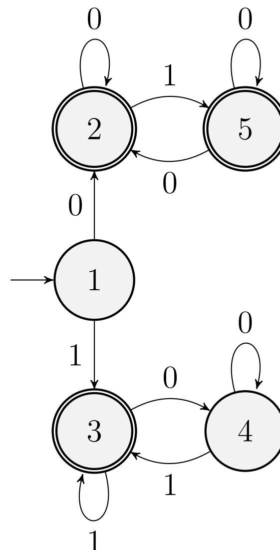
Exemplo 2.6. Para ilustrar o uso de propriedades de fechamento para composição de autômatos, considere os seguintes AFDs: o primeiro reconhece $L_1 = \{0\}\{0, 1\}^*$ e o segundo reconhece $L_2 = \{0, 1\}^*\{1\}$.



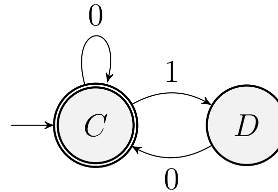
Aplicando a construção de produto, pode-se criar AFDs para $L_1 \cap L_2$ e $L_1 \cup L_2$ que são apresentadas a seguir. Considere as seguintes abreviações para os conjuntos de estados: 1 para o conjunto $\{A, C\}$, 2 para $\{B, C\}$, 3 para $\{e, D\}$, 4 para $\{e, C\}$ e 5 para $\{B, D\}$. O próximo autômato representa um AFD para $L_1 \cap L_2$.



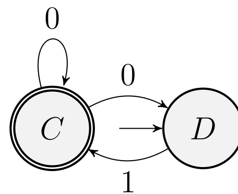
O resultado para $L_1 \cup L_2$, será:



Para construir a complementação de um AFD basta marcar como finais todos os estados não finais e como não finais os antigos estados finais. O autômato resultado para $\overline{L_2}$ é :



Finalmente, o seguinte AFN é obtido ao se usar a propriedade de reverso sobre o AFD para L_2:



2.1.1.5 Algoritmo de minimização de *Brzozowski*

O algoritmo apresentado por *Brzozowski*, é um dos algoritmos de simples implementação que permite produzir o AFD mínimo equivalente a um AFD fornecido como entrada. Sejam $reverse$ uma função para se obter o AFN correspondente ao reverso de um AFD e $subset$ uma função que converte um AFN em um AFD[1], o algoritmo de minimização de Brzozowski é definido como:

$$minimize(M) = subset(reverse(subset(reverse(M))))$$

O primeiro passo do algoritmo é calcular o reverso do AFD fornecido como entrada e em seguida, converter o AFN resultante em outro AFD. A partir do AFD obtido no primeiro passo, repetimos o processo e o AFD resultante será o AFD mínimo equivalente ao AFD original.

Para auxiliar na compreensão do funcionamento do algoritmo considere o seguinte exemplo:

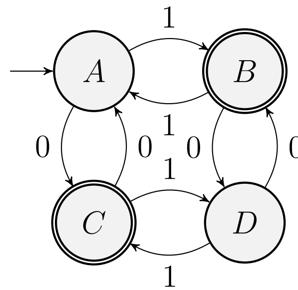
Exemplo 2.7.

Considere a linguagem $L \subseteq \{0, 1\}^*$

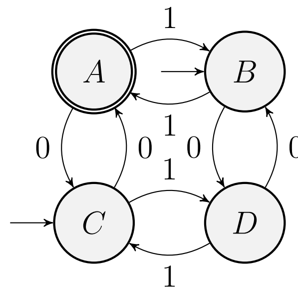
$L = \{w \in \{0, 1\}^* | \exists k. |w| = 2k + 1\}$

Dessa forma, temos o seguinte autômato que representa L.

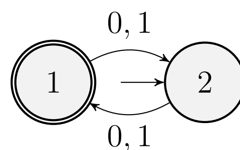
[1] O processo de conversão de um AFN em um AFD é conhecido na literatura como *subset-construction* (KOZEN, 1997).



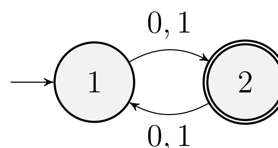
O primeiro passo da execução do Algoritmo de *Brzozowski* é fazer o reverso do autômato que se deseja encontrar o mínimo. Portanto, deve-se inverter o sentido das transições e tornar os estados finais em iniciais e todos os inicias em finais. Dessa forma, será obtido o seguinte autômato resultado do primeiro passo:



O segundo passo presente no algoritmo de *Brzozowski* é converter o AFN gerado pelo passo anterior em um AFD. Para isso, será usado a **Definição 2.8**. Com isso, obtém-se um estado inicial com o conjunto de estados inicias $\{B, C\}$ que será renomeado para 1. Além disso, considerando as transições do novo estado 1, será obtido um novo estado, renomeado de 2, com o conjunto de estados $\{A, D\}$ que será o estado final. Assim ficaria o autômato resultante:



Dessa forma, é encontrado o autômato mínimo do inverso do autômato inicial. Como descrito anteriormente, para se obter o autômato mínimo do autômato inicial, basta executar os dois primeiros passos novamente. Portanto, ao se reverter o AFD anterior, tem-se o seguinte resultado:



Contudo, como o autômato é já é um AFD, não se faz necessário nesse exemplo específico converter um AFN em um AFD. Assim, é obtido um AFD mínimo equivalente ao autômato inicial.

2.2 Tecnologias utilizadas

Nesta seção abordaremos as tecnologias utilizadas para o desenvolvimento da ferramenta proposta neste trabalho. Primeiramente, apresentaremos uma breve introdução à linguagem Racket e na sequência a ferramenta GraphViz, utilizada para elaboração de representações gráficas de AFDs.

2.2.1 Introdução à linguagem Racket

O Racket é uma linguagem de programação funcional dinamicamente tipada e consiste de um dialeto moderno da conhecida linguagem LISP. Assim como outros dialetos de LISP, Racket possui uma sintaxe simples e um sistema de macros que permite incluir novos elementos sintáticos à linguagem.

Uma característica importante de Racket é que sua sintaxe base utiliza as chamadas *S-expressions*, em que o código fonte possui uma estrutura simples que torna a representação de sua árvore de sintaxe como uma lista de símbolos. Ilustraremos esses conceitos utilizando alguns exemplos. A seguir, apresentamos a definição de uma função para cálculo do fatorial utilizando Racket:

Algoritmo 1: Cáculo de Fatorial em Racket

```
1        (define (factorial n)
2             (if (= n 0)
3                 1
4                 (* n (factorial (- n 1)))))
5
```

A palavra reservada `define` inicia a definição de uma função de nome `fatorial`, que recebe um argumento de nome `n`. Em linguagens da família LISP, utilizamos parêntesis para representar a chamada de funções, que devem sempre ser feitas como um prefixo de seus argumentos. Por exemplo, a expressão aritmética $n - 1$ é representada, em Racket, como `(- n 1)`, em que o parêntesis é utilizado para representar a chamada da função `-` sobre os argumentos `n` e `1`. Adicionalmente, essa estrutura da chamada de funções permite enxergá-la como uma lista de símbolos em que o primeiro é a função `-`; o segundo, a variável `n` e o último o número 1. Essa representação é o que permite que linguagens da família LISP possuam bons recursos para meta-programação. Em particular, Racket combina o mecanismo de macros com um sistema de módulos para permitir que um mesmo programa seja desenvolvido em diferentes linguagens, que internamente são todas traduzidas para Racket.

Optamos por utilizar a linguagem Racket para o desenvolvimento deste trabalho devido à facilidade de criar novas linguagens e seu IDE extensível, Dr. Racket. Novas linguagens desenvolvidas em Racket possuem suporte automático deste IDE, diminuindo assim, a barreira para utilização de linguagens desenvolvidas utilizando Racket.

2.2.2 Introdução ao GraphViz e a linguagem Dot

GraphViz é um conjunto de ferramentas para o dsenho de grafos (ELLSON et al., 2002). A ferramenta permite que grafos simples ou mais complexos sejam criados e visualizados de maneira rápida e fácil. Neste trabalho, utilizaremos essa ferramenta como uma forma de produzir representações visuais de autômatos.

O Graphviz utiliza a linguagem Dot, que tem como finalidade criar grafos e diagramas de forma declarativa. Com ela é possível criar driagramas de fluxo, árvores, diagramas de classes, diagramas de rede, grafos e entre outros. A partir da descrição de um grafo, o Graphviz produz uma imagem utilizando diversas heurísticas para desenho de grafos (ELLSON et al., 2002).

O seguinte exemplo, apresenta um trecho de código Dot. A especificação representa um grafo não direcionado (declarado usando a palavra reservada `graph`) de nome `G` e, na sequência, especifica suas arestas. Cada aresta é representada pelos rótulos dos nós de suas extremidades e pelo token `--`, usado para definir aresta não direcionadas.

Algoritmo 2: Grafo Geral em Dot

```
1 graph G {
2   A -- B;
3   A -- C;
4   B -- C;
5   B -- D;
6   C -- D;
7 }
8
```

A partir da especificação anterior, o Graphviz produz a seguinte imagem em formato png:

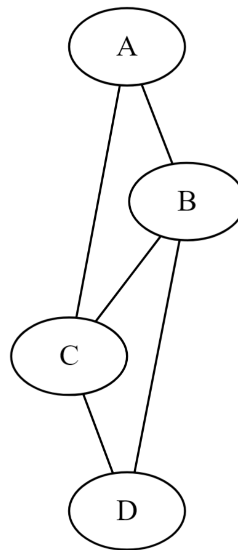


Figura 2.1 – Imagem que representa um grafo geral usando Dot

2.3 Trabalhos Relacionados

Nesta seção serão apresentadas ferramentas que também tem o intuito de auxiliar no ensino das áreas de linguagens formais e autômatos. Além disso, serão apresentadas as funcionalidades de cada *software* e relatar como foi o uso de algumas ferramentas na sala de aula.

2.3.1 *Java Formal Language and Automata Package* (JFLAP)

Gramond e Rodger (1999) apresentaram o JFLAP como uma ferramenta capaz de auxiliar o ensino de disciplinas que contem o tema de teoria de autômatos uma vez que, essa matéria geralmente é lecionada sem o uso de computadores fazendo com que os alunos não tenham um retorno imediato ao fazer exercícios de fixação. Dessa forma, o *software* apresentado tem como solução para esse problema, uma representação visual alternativa e a chance de criar e simular essa representações.

Ainda em seu texto, (GRAMOND; RODGER, 1999) descreve JFLAP como uma ferramenta para criar e simular diversas versões de AFD (Autômato Finito Determinístico), AFN (Autômato Finito não-determinístico), AP (Autômato de Pilha), Máquinas de Turing com uma ou duas fitas. Ademais, o usuário consegue criar grafos que representam as transições do autômato, renomeá-las, inserir dados de entrada e, por fim, simular a execução da máquina ou do autômato. Além disso, o *software* também tem a funcionalidade de fazer transformações em linguagens formais, sendo elas: converter uma AFN em uma AFD, minimizar uma AFD, transformar um AFN em uma gramática regular e um transformar uma gramática regular em uma AFN. É também descrita a forma de se utilizar cada uma das funcionalidades citadas.

Por fim, (GRAMOND; RODGER, 1999) fornece exemplos de maneiras de como usar o

JFLAP em sala de aula. Além disso, relata a experiência de alguns alunos ao usar a ferramenta para estudos tanto em sala de aula quanto em atividades avaliativas ou exercícios de fixação.

2.3.2 *FSA Simulator*

Grinder (2002) descreve o desenvolvimento do *FSA Simulator*, um programa escrito em Java que possibilita que estudantes de ciência da computação façam experimentos com autômatos finitos. Embora existam *softwares* similares, este em questão possui duas funcionalidades diferentes dos demais, sendo elas: a possibilidade de o usuário criar um autômato finito de forma visual e testar se uma sequência de símbolos arbitrária é aceita naquele autômato; comparar dois autômatos e dizer se eles aceitam a mesma linguagem.

A primeira versão descrita possuía vários autômatos finitos criados previamente para servir de exemplos aos usuários. Apesar de a ferramenta aceitar AFD quanto AFN, os resultados dos testes de aceitação *strings* para autômatos determinísticos são mais confiáveis.

Após um reestruturação do projeto, foi crida uma segunda versão do *FSA Simulator*. Assim o *software* passou a salvar os autômatos criados em arquivos XML ou em bancos de dados ou em servidores usando *web services*. Além disso, os testes de aceitação de uma linguagem descrita por um AFN foram aprimoradas e seus resultados passaram a ser sempre corretos. Ademais, o programa conta com animações que mostram a passo a passo da execução de um autômato. Uma outra funcionalidade presente nessa ferramenta é a possibilidade de detectar se dois autômatos finitos reconhecem a mesma linguagem. Essa função faz com que os alunos possam verificar se o autômato criado aceita a mesma linguagem que é esperada durante o seu estudo.

2.3.3 *The Java Computability Toolkit* (JCT)

O JCT (*The Java Computability Toolkit*), descrito por (ROBINSON et al., 1999), é um *software* visual e interativo que tem como intuito auxiliar o ensino de estudantes das disciplinas relacionadas a teoria da computação. Geralmente o ensino de tal campo da computação é feito usando lápis e papel que, frequentemente, possibilita que os alunos se confundam no processo de criar autômatos ou Máquinas e acabam não entendendo corretamente o conceito que estão sendo ensinados. Além disso, verificar se um autômato ou uma Máquina de Turing estão corretas pode ser uma tarefa complexa e pode consumir bastante tempo.

O programa foi criado usando a biblioteca *Java Swing* para construir os componentes visuais. Ele possibilitou que o JCT pudesse ser executado em navegadores, o que facilitaria o uso dessa ferramenta. Além disso, o *software* possui as seguintes funcionalidades:

- Conjunto de propriedades de fechamento de autômatos finitos.
- Uma representação única dos autômatos como uma placa de circuito.

- Mostrar visualmente o passo a passo da execução de um autômato a patir de uma entrada arbitrária.
- Especificação de uma Máquina de Turing em alto nível.
- Métodos para simplificar sintaticamente Máquinas de Turing.
- Possibilidade de salvar, carregar e imprimir os autômatos ou as Máquinas de Turing.

Com essas funcionalidades, o JCT pode auxiliar os alunos durante seus estudos das áreas de teoria da computação minimizando o tempo gasto em certas atividades e facilitando a compreensão do funcionamento de autômatos e Máquinas de Turing.

2.3.4 *Automated Grading of DFA Constructions*

Alur et al. (2013) expõe uma abordagem para avaliar automaticamente a construção de autômatos finitos determinísticos (AFD) por alunos do curso de ciência da computação. O artigo tem como objetivo fornecer um *feedback* instantâneo e preciso aos alunos, os informando se os autômatos criados por eles esta muito longe da solução correta e, como consequência, mostrar o porquê ele esta incorreto. Dessa forma, alunos que usaram essa ferramenta, podem corrigir seus autômatos finitos mais rapidamente.

A ideia proposta usa um algoritmo de comparação de autômatos para verificar a equivalência entre a solução do aluno e a solução esperada. O algoritmo utilizado é baseado na minimização de um AFD para simplificar os autômatos e facilitar a comparação. O algoritmo também usa técnicas de poda para reduzir o tempo de processamento e tornar a abordagem escalável para grandes conjuntos de dados.

É relatado resultados experimentais que mostram que a abordagem proposta é precisa e escalável para uma ampla gama de problemas de construção de AFD. Além disso, é explicado no artigo que essa abordagem aumenta a eficiência e a eficácia do ensino de linguagens formais e autômatos, permitindo que os instrutores forneçam *feedback* mais rápido e detalhado aos alunos.

2.3.5 Experimentos em Salas de Aulas

D'antoni et al. (2015) apresenta uma pesquisa que tem como intuito criar uma ferramenta de *feedback* automático para correção de AFD e avaliar se essa ferramenta pode auxiliar os estudantes a aprenderem a construir autômatos finitos de forma mais correta. Além disso, a ferramenta desenvolvida produzirá um *feedback* dizendo se o autômato criado está correto, gera um contraexemplo caso esteja incorreto e dicas de como o usuário pode corrigir esse autômato. O estudo envolveu alunos do curso de Ciência da Computação de uma universidade dos Estados Unidos, em que esses estudantes tiveram uma semana para completar uma atividade que era constituída em diversos exercícios de construir autômatos. Ademais, foram divididos em dois

grupos de alunos para resolver essa atividade, um que fez o uso da ferramenta e outro grupo que não teve acesso a ela.

Os resultados encontrados no artigo indicam que o grupo que teve acesso à ferramenta obteve um desempenho significativamente melhor do que o grupo que não teve acesso ao *software* ao executarem uma tarefa que constituía em construir um autômato finito determinístico. Além disso, os alunos do grupo experimental tiveram uma percepção positiva ao usar a ferramenta.

Por fim, os responsáveis pela escrita do artigo concluem que o *feedback* automático pode ser uma ferramenta eficaz para ajudar estudantes a construir autômatos finitos, melhorando seu desempenho e percepção de se utilizar ferramentas de aprendizado.

Em um outro artigo, o (GRINDER, 2003) avalia de forma empírica a efetividade de se usar o *FSA Simulator*, descrito em 2.3.2, e sua funcionalidade de fazer comparações entre dois autômatos. Este artigo tem o intuito de, por meio de experimentos com alunos usando o *software* citado, validar se houve uma melhora na compreensão desses estudantes na hora de criar um autômato para descrever uma linguagem. Um experimento foi executados no laboratório de ciência da computação na Universidade Estadual de Montana.

Foram criados dois grupos de estudantes em que, metade dos alunos usaram o *FSA Simulator* durante as aulas da disciplina de teoria da computação e a outra metade não fez o uso da ferramenta. Foi feita uma avaliação com esses alunos ao final do curso, que era constituída por questionários e testes sobre conceitos ensinados durante as aulas de teoria da computação e autômatos finitos.

Os resultados encontrados no artigo mostram que os alunos que usaram o *software* tiveram um melhor desempenho nos testes aplicados do que os alunos que não usaram o *FSA Simulator*. Além disso, os estudantes que usaram o software relataram que se sentiram mais engajados nas aulas e acharam o aprendizado mais fácil.

O artigo conclui que, a ferramenta é útil para o ensino de teoria da computação e autômatos finitos e proporciona uma experiência visual e interativa durante as aulas causando assim, uma compreensão maior sobre esses conceitos abstratos.

3 Desenvolvimento

Este capítulo visa descrever como foi desenvolvimento as funcionalidades da ferramenta proposta para correção automática de exercícios sobre autômatos finitos. Iniciamos a Seção 3.1 que descreve o algoritmo proposto para correção automática de autômatos finitos determinísticos em termos de propriedades de fechamentos de linguagens regulares e apresenta um exemplo deste processo. A Seção 3.2 discute como o Graphviz foi utilizado para produzir imagens de autômatos e o estado atual de implementação da ferramenta.

3.1 Correção Automática de AFDs

Intuitivamente, o processo de correção de autômatos finitos recebe como entrada dois argumentos: um autômato considerado correto e que denominaremos por gabarito (representado pela letra G) e um autômato que representa a solução submetida por um aluno (representado pela letra A). O processo de correção é baseado no fato de que a solução do aluno será correta se este reconhece a mesma linguagem do autômato de gabarito. Para determinar que o autômato do aluno aceita a mesma linguagem do gabarito, utilizaremos a seguinte estratégia:

Definição 3.1 (Equivalência entre linguagens)**.** Seja L_A a linguagem aceita pelo autômato do aluno e L_G a linguagem aceita pelo autômato gabarito. Podemos dizer que $L_A = L_G$ se $L_A - L_G = \emptyset$ e $L_G - L_A = \emptyset$.

De maneira simples, a linguagem $L_A - L_G$ denota o conjunto de palavras aceitas pela solução do aluno e não aceitas pelo gabarito, isto é, o conjunto de palavras aceitas incorretamente pela solução submetida. Por sua vez, o conjunto $L_G - L_A$ denota as palavras incorretamente recusadas pelo autômato do aluno. Dessa forma, temos que a solução do aluno só pode ser considerada correta se $L_A - L_G = \emptyset$ e $L_G - L_A = \emptyset$.

Uma vez que a classe das linguagens regulares é fechada sobre a interseção e complementação, temos que esta classe de linguagens é também fechada sobre a diferença de conjuntos pois $A - B \equiv A \cap \overline{B}$. Desta forma, a partir dos autômatos de gabarito e do autômato submetido pelo aluno podemos construir autômatos para:

- Palavras incorretamente aceitas pela solução do aluno: para isso basta construir um autômato para $L_A - L_G$.
- Palavras incorretamente recusadas pela solução do aluno: para isso basta construir um autômato para $L_G - L_A$.

Teremos que a solução do aluno estará correta se ambas as linguagens forem vazias, isto é se os autômatos resultantes de $L_A - L_G$ e $L_G - L_A$ não possuírem estados finais. Caso a solução do aluno seja inválida, um dos dois autômatos terá algum estado final e poderemos computar um contra-exemplo de sua solução utilizando um algoritmo para produzir caminhos entre o estado inicial e algum dos estados finais.

O exemplo a seguir ilustra essas ideias.

Exemplo 3.1 (AFDs Equivalentes)**.**

Considere a linguagem $L \subseteq \{0,1\}^*$

$L = \{w \in \{0,1\}^* | \exists k. |w| = 2k\}$ Considere os autômatos a seguir como A, criado por um aluno e G, considerado gabarito, respectivamente:

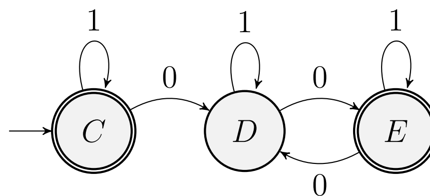


Figura 3.1 – Autômato A

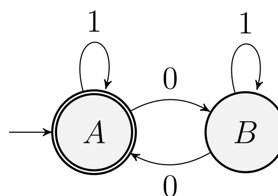
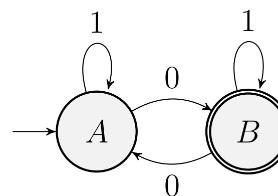
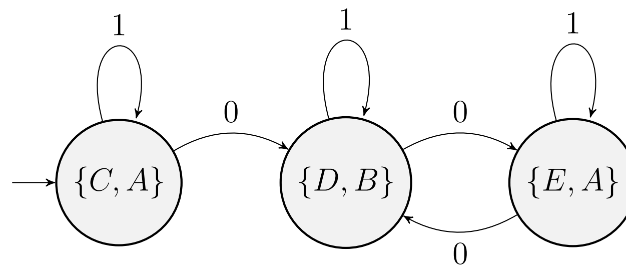


Figura 3.2 – Autômato G

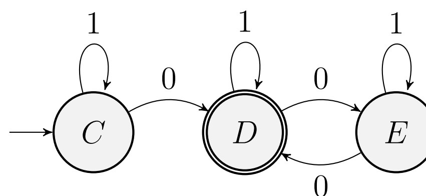
Como o $A - G = A \cap \overline{G}$, é necessário, primeiro, gerar o autômato complemento de G que é :



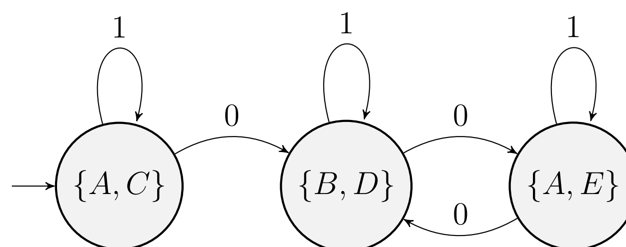
A partir disso, o resultado de $A \cap \overline{G}$ é um autômato vazio como pode-se visualizar no autômato abaixo, ou seja, não possui estado final. Isso quer dizer que não há palavras que A reconhece que G não reconhece.



Além da operação de $A - G$, é necessário fazer também a operação $G - A$, que se traduz em $G \cap \overline{A}$. Assim como anteriormente, é preciso conhecer o autômato complemento. Porém, nesse caso será o complemento de A, que é :



A partir disso, o resultado de $G \cap \overline{A}$ é um autômato vazio como pode-se visualizar no autômato abaixo, ou seja, não possui estado final. Isso quer dizer que não há palavras que A não reconhece que G reconhece.



Portanto, é possível afirmar que A é equivalente a G uma vez que, os dois autômatos aceitam a mesma linguagem como foi mostrado pelas operações anteriores.

Exemplo 3.2 (AFDs Não Equivalentes)**.**

Considere a linguagem $L \subseteq \{0, 1\}^*$

$L = \{w \in \{0, 1\}^* | \exists k.|w| = 2k\}$

Considere os autômatos a seguir como A, criado por um aluno e G, considerado gabarito, respectivamente:

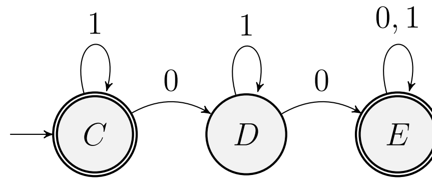


Figura 3.3 – Autômato A

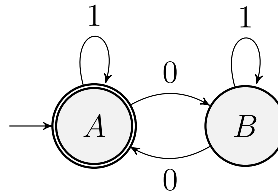
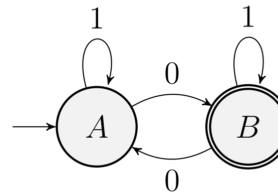
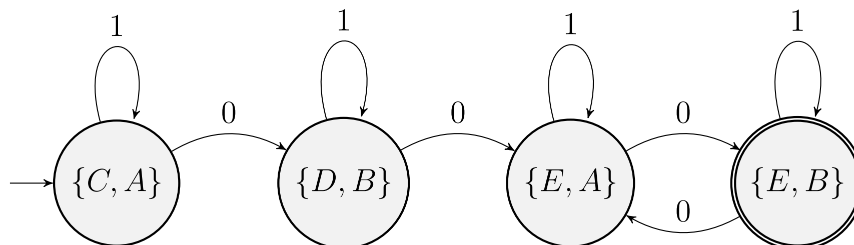


Figura 3.4 – Autômato G

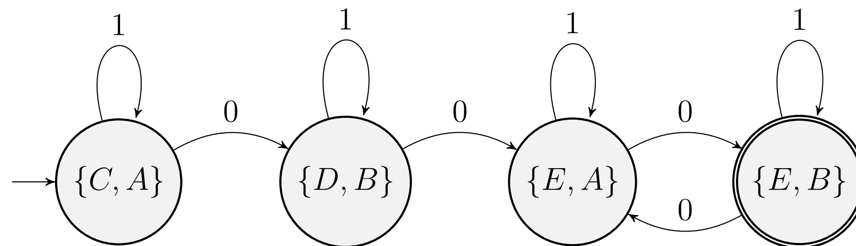
Como o $A - G = A \cap \overline{G}$, é necessário, primeiro, gerar o autômato complemento de G que é :



A partir disso, o resultado de $G \cap \overline{A}$ é:



Ao minimizar o AFD resultante, será o obtido o mesmo autômato anterior, uma vez que ele já é mínimo. Além disso, é possível perceber que o autômato resultante não é vazio, o que quer dizer que o AFD A reconhece palavras que G não reconhece. Para encontrar os contraexemplos que provam que A e G não são equivalentes basta obter uma palavra processado por um caminho entre o estado inicial e o estado final. Um possível contra-exemplo seria a palavra `000`, que será retornada ao estudante que poderá usá-la para depurar sua solução.



3.2 Detalhes de Implementação

Esta seção é responsável por mostrar as funcionalidades já implementadas na ferramenta além de explicar, através dos códigos, como cada uma delas foi desenvolvida. A Seção 3.2.1 descreve a integração do GraphViz com o IDE de Racket, o Dr.Racket para visualização de autômatos especificados textualmente. Em seguida, descrevemos a implementação de algoritmos para propriedades de fechamento (Seção 3.2.2).

3.2.1 Visualização do Autômato Finito

Ao pensar em um ferramenta de correção automática de autômatos finitos, primeiro é necessário pensar como fazer que o usuário do programa consiga visualizar o autômato que ele quer descrever. Após chegar a uma conclusão de como tornar aquele autômato finito visível, é necessário avaliar uma forma em que o usuário do *software* consiga descrever o autômato desejado. Esta seção é responsável por explicar como foram construídas as soluções para os problemas citados.

3.2.1.1 Desenho do Autômato Finito

Para representar os autômatos finitos foi utilizada a biblioteca Graphviz, que tem como principal funcionalidade desenhar grafos a partir de uma linguagem chamada Dot. Dessa forma, a partir do que for descrito pelo usuário será criado um código em Dot para montar o autômato. Para isso, será mostrado um exemplo de código em Dot para criar um autômato e como ficará o autômato resultante do código.

Exemplo 3.3.

Considere a seguinte linguagem sobre $\Sigma = \{0, 1\}$

$L = \{0, 1\}^*\{11\}$. Ou seja, uma linguagem que termine com 11.

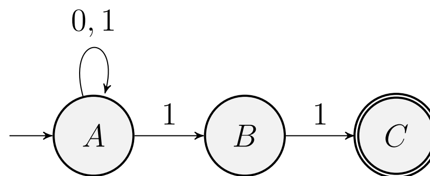
Dessa forma, o seguinte código será o responsável por montar o autômato finito correspondente a essa linguagem.

Algoritmo 3: Autômato em Dot

```
digraph nfa {
      rankdir=LR;
      node [shape=circle];
      start [shape=point];
      C [shape=doublecircle];
      start -> A;
      A -> A [label="0,1"];
      A -> B [label="1"];
      B -> C [label="1"];
}

```

Esse código irá gerar o seguinte autômato:



3.2.1.2 Descrição do Autômato

Para que o usuário da ferramenta pudesse descrever o autômato de forma simples e intuitiva foi utilizada uma funcionalidade da linguagem Racket chamada *macro* (uma extensão para o compilador de Racket). Dessa forma, seria possível criar uma sintaxe, dentro da linguagem, que indicaria que um autômato finito está sendo descrito passando toda a sua quíntupla com: conjunto de estados, alfabeto, transições, estado inicial ou conjunto de estados inicias e conjunto de estados finais. O exemplo a seguir mostra como é a definição de um autômato dentro da ferramenta:

Exemplo 3.4.

Considere a seguinte linguagem sobre $\Sigma = \{0, 1\}$

$L = \{0, 1\}^*\{11\}$. Ou seja, uma linguagem que termine com 11.

Algoritmo 4: Descrição de um AFD

```
(define M
  (dfa A (C)
        (A : 0 -> A)
        (A : 1 -> B)
        (B : 0 -> A)
        (C : 0 -> A)
        (B : 1 -> C)
        (C : 1 -> C)))

```

Na primeira linha do código acima, é definida a variável que armazenará o autômato. Logo em seguida, na segunda linha, é indicado pelo comando *dfa* que um autômato finito determinístico será definido (DFA é o equivalente de AFD em inglês. Na mesma linha, há a definição de qual será o estado inicial, caracterizado pelo símbolo *A*. Continuando na segunda linha, o próximo argumento do comando é uma lista com o símbolo *C*, indicando a lista com o conjunto de estados finais. A seguir, da linha 3 a última linha com código, são representadas as transições do autômato, onde o primeiro símbolo dentro dos parênteses indicam o estado de origem de transição, o símbolo de dois pontos indicam que o próximo símbolo será o que o autômato processará e por fim, após ->, o símbolo que irá representar o estado de destino daquela transição.

A partir da descrição feita, é usada o seguinte *macro* para transformar o que foi escrito pelo usuário em uma variável manipulável dentro da ferramenta.

Algoritmo 5: Macro

```
(define-syntax (dfa stx)
  (syntax-parse stx #:datum-literals (: ->)
    [(_ start:id (end:id ...)
    [state:id : sym:expr -> next:id] ...)
    #:fail-when (check-duplicates
    (syntax->datum #'(list (cons 'state 'sym) ...)))
        "This finite machine is non-deterministic."
     #'(complete (fa 'dfa
                    (remove-duplicates
                        (append (syntax->datum #'(state ...))
                           (syntax->datum #'(next ...))
                           (syntax->datum #'(end ...))
                           (syntax->datum #'(start))))
                    (remove-duplicates
                        (syntax->datum #'(sym ...)))
                    (list (cons (cons 'state 'sym) 'next) ...)
                    'start
                    (list 'end ...)))]))

```

O autômato gerado pelo *macro* mostrado anteriormente deverá gerar uma variável com as seguintes características:

```
1       (fa dfa (A B C) (0 1) (((A . 0) . A) ((A . 1) . B) ((B . 0) . A)
    ((C . 0) . A) ((B . 1) . C) ((C . 1) . C)) A (C))
2
```

Onde os dois primeiros valores indicam que a variável se trata de um autômato finito e um AFD, respectivamente. O próximo valor é o conjunto de estados do autômato, representado por uma lista de símbolos. A próxima lista é o alfabeto aceito pelo autômato. O próximo valor armazenado é uma lista com todas as transições do autômato, sendo cada transição representada por um par de par, o primeiro valor do par interno indica o estado de origem, o segundo valor do par interno indica o símbolo que será processado e, por fim, o segundo valor do par externo indica o estado de destino da transição. Os dois últimos valores presentes na variável representam o estado inicial e conjunto de estados finais, respectivamente.

Esse autômato M gera a imagem 3.5 dentro da ferramenta

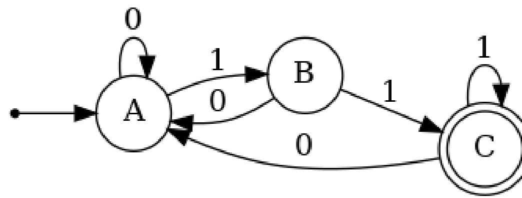


Figura 3.5 – Imagem que representa um autômato finito determinístico

3.2.2 Operações em Autômatos Finitos

Além da descrição a visualização de um autômato finito, é interessante para que a ferramenta auxilie ainda mais os seus usuários, que seja possível aplicar algumas das propriedades de fechamento, é importante que haja a possibilidade de minimizar um autômato e claro, que seja possível fazer a correção automática de um autômato.

3.2.2.1 Funções sobre Propriedade de Fechamento

Funções que aplicam as propriedades de fechamento, definidas em 2.1.1.4, são algumas das funcionalidades presentes na ferramenta. Cada uma das propriedades de fechamento foi criada individualmente de maneira que o usuário da ferramenta possam usar elas separadamente e possam ver o autômato finito resultante delas.

3.2.2.1.1 Produto de Dois AFDs

O código a seguir é responsável por gerar o produto de dois AFDs:

Algoritmo 6: Produto de AFDs

```
1  ; generate a list with a cartesian product of all transitions of two dfa
2  (define (dfa-product dfa1 dfa2)
3    (apply append (map (lambda (s1)
4                          (apply append (map (lambda (s2)
5                               (map (lambda (symb)
6                                       (product-transitions
7                                          (dfa-delta dfa1)
8                                          (dfa-delta dfa2)
9                                          s1 s2 symb))
10                                   (dfa-sigma dfa1)))
11                            (dfa-states dfa2))))
12                        (dfa-states dfa1))))
13
```

Essa é a principal função para a propriedade de fechamento do produto. Nela, como descrito no comentário da função, é gerada uma lista com o produto cartesiano de dois AFDs (`dfa1, dfa2`) com todas as transições possíveis. Isso faz com que sejam criados estados que

não são alcançáveis a partir do estado inicial. Para isso, foi criada uma função que indica apenas os estados que são alcançáveis a partir do estado inicial, sendo ela desenvolvida da seguinte forma:

Algoritmo 7: Algoritmo para encontrar os estados alcançáveis a partir do estado inicial

```
; generate the list of states that are reachable from the start state
(define (reachable-states transitions states)
  (define reachable-states
    (append states (filter symbol?
                    (append
                      (map (lambda (t)
                        (if (member? (car (car t)) states)
                          (cdr t)
                          #f))
                      transitions)))))

  (if (equal?
        (remove-duplicates reachable-states)
        (remove-duplicates states))
      (remove-duplicates states)
      (reachable-states transitions
        (filter symbol? reachable-states))))

```

A ideia desse código é criar uma função recursiva que inicia com uma lista que contêm o estado inicial e, a cada iteração, são incluídos os estados que podem ser alcançados por um passo de transição. Além disso, é utilizada uma variável de critério de parada que foi chamada de *states*, ela receberá o resultado da iteração anterior. Assim, quando a variável usada para critério de parada não for alterada em relação a iteração anterior, a função termina.

Dessa forma, é possível gerar todas as transições e estados do produto de dois AFDs. Contudo, não é possível visualizar o desenho somente do produto uma vez que, é necessário para a função que desenha que os estados finais sejam descritos. Como indicar os estados finais do produto de dois AFDs caracterizam ou a interseção ou a união de dois AFDs, não pode-se considerar um produto de dois AFDs.

3.2.2.1.2 Interseção de Dois AFDs

O código a seguir é responsável por gerar a interseção de dois AFDs:

Algoritmo 8: Interseção de 2 AFDs

```
; generate the list of transitions that are reachable from start state
(define (reachable-trasitions states transitions)
  (filter (lambda (t)
        (member? (car (car t)) states)) transitions))

; generate a list with all final states considering a intersection of
   two dfas
(define (dfa-intersection-states dfa1 dfa2)
  (convert-string-symbol (combination-states (dfa-final dfa1)
                                             (dfa-final dfa2))))

; generate a intersection of two dfas
(define (mk-intersection-dfa dfa1 dfa2)
  (define start (product-start dfa1 dfa2))
  (define sigma (dfa-sigma dfa1))
  (define final (dfa-intersection-states dfa1 dfa2))
  (define aux-delta (dfa-product dfa1 dfa2))
  (define states
    (reachable-states aux-delta (list start)))
  (define delta (reachable-trasitions states aux-delta))
  (mk-dfa states sigma delta start final))

```

Como é possível observar, essa função cria um novo AFD com as características necessárias para ser a interseção de dois AFDs. Além disso, o código também usa funções que são do produto de dois autômatos finitos. Isso pode ser observado nas linhas 13, 16 e 18. Existem algumas outras funções auxiliares mas que não há a necessidade de mostrar aqui porque não tem impacto direto na construção da interseção.

3.2.2.1.3 União de Dois AFDs

O código a seguir é responsável por gerar a união de dois AFDs:

Algoritmo 9: União de 2 AFDs

```
1 ; generate the list of transitions that are reachable from start state
2 (define (reachable-trasitions states transitions)
3   (filter (lambda (t)
4         (member? (car (car t)) states)) transitions))
5
6 ; generate a list with all final states considering a union of two dfas
7 (define (dfa-union-states dfa1 dfa2)
8   (define list-states (product-states dfa1 dfa2))
9   (define final-states (append (dfa-final dfa1) (dfa-final dfa2)))
10   (remove-duplicates (flatten
11                       (map (lambda (state)
12                              (find-union-states list-states state))
    final-states))))
13
14 ; generate a union of two dfas
15 (define (mk-union-dfa dfa1 dfa2)
16   (define start (product-start dfa1 dfa2))
17   (define sigma (dfa-sigma dfa1))
18   (define final (dfa-union-states dfa1 dfa2))
19   (define aux-delta (dfa-product dfa1 dfa2))
20   (define states
21     (reachable-states aux-delta (list start)))
22   (define delta (reachable-trasitions states aux-delta))
23   (mk-dfa states sigma delta start final))
24
```

Como é possível observar, essa função cria um novo AFD com as características necessárias para ser a união de dois AFDs. Assim como na interseção, o código usa funções que são do produto de dois autômatos finitos. Isso pode ser observado nas linhas 16, 19 e 17. Existem algumas outras funções auxiliares mas que não há a necessidade de mostrar aqui porque não tem impacto direto na construção da união.

3.2.2.1.4 Complemento de um AFDs

O código a seguir é responsável por gerar a complementação de um AFD:

Algoritmo 10: Complemento de um AFD

```
; generate a complement of a dfa
(define (complement dfa)
  (define new-final (set-diff (dfa-states dfa) (dfa-final dfa)))
  (mk-dfa (dfa-states dfa)
          (dfa-sigma dfa)
          (dfa-delta dfa)
          (dfa-start dfa)
          new-final))

```

Essa função é mais simples que as demais, uma vez que, é necessário apenas tornar os estados não finais em finais e os que, anteriormente, eram finais em não finais. Isso é feito utilizando uma função da própria linguagem que retorna uma lista somente com os elementos que não estão em uma outra lista. Assim como na união e na interseção, é criada um novo AFD com as características necessárias para ser a complementação de um AFD anterior.

3.2.2.2 Função de Minimização

As funções que aplicam a minimização de um autômato finito são a implementação Racket do algoritmo de *Brzozowski*, descrito em na seção 2.1.1.5. O código é bem direto. A primeira função `revert-transition`, inverte o sentido de uma única transição.

Algoritmo 11: Inverso de uma transição

```
;; revert a single transition
(define (revert-transition t)
  (match t
    [(cons (cons o s) t) (cons (cons t s) o)]))

```

Por sua vez, a função `revert-delta` inverte todas as transições, aplicando a função `revert-transition` a lista de transições de um AFD.

Algoritmo 12: Inverso de uma lista de transições

```
(define (revert-delta delta)
    (combine-domain (map revert-transition delta)))

```

Finalmente, a função `revert-dfa` cria uma AFN, sendo ele o inverso do AFD passado como argumento.

Algoritmo 13: Algoritmo de minimização

```
(define (revert-dfa dfa)
  (mk-nfa
    (dfa-states dfa)
    (dfa-sigma dfa)
    (revert-delta
     (remove-duplicates (dfa-delta dfa)))
    (dfa-final dfa)
    (list (dfa-start dfa))))
```

Usando a função de reverse, podemos implementar o algoritmo de minimização como sendo a composição de produzir o reverso de um AFD e após converter o AFN resultante em AFD duas vezes como apresentado no código a seguir:

Algoritmo 14: Algoritmo de minimização

```
(define (minimization fa)
  (define step1 (nfa->dfa (revert-dfa fa)))
  (nfa->dfa (revert-dfa step1)))
```

3.2.3 Algoritmo de Correção

Os códigos a seguir são responsáveis por executar a correção automática de autômatos finitos determinísticos como descrito na Seção 3.1.

3.2.3.1 Impressão de Palavras Encontradas

As funções descritas nessa seção tem como intuito mostrar o passo a passo para encontrar as palavras de contra-exemplo encontradas no algoritmo de correção.

Algoritmo 15: Diferença entre Autômatos

```
(define (automaton-difference dfa1 dfa2)
    (mk-intersection-dfa dfa1 (complement dfa2)))

```

O Algoritmo 15 cria um novo autômato, sendo a diferença entre dois AFDs.

Algoritmo 16: Transforma um Automato em um Grafo

```
(define (transform-dfa-graph dfa)
    (map (lambda (t)
        (list (cdr (car t)) (car (car t)) (cdr t))) (dfa-delta dfa)))

```

O Algoritmo 16 transforma as transições de um autômato em uma estrutura de dados que posteriormente será usada para criar um grafo. A estrutura criada é uma lista, em que, o

seu primeiro elemento representa o peso da aresta, ou seja, o símbolo que será processado. O segundo elemento descreve o vértice origem (estado origem) e o terceiro elemento corresponde ao vértice destino (estado destino).

Algoritmo 17: Encontra Caminho Entre um Vértice ao Vértice de Origem

```
1 (define (find-path predecessors state word graph)
2     (if (hash-ref predecessors state)
3     (find-path predecessors
4       (hash-ref predecessors state)
5       (append word
6         (list
7           (edge-weight graph
8                       (hash-ref predecessors state) state)))
9       graph)
10    word))
11
12 (define (acceptance-words predecessors final-states graph)
13    (define words
14    (map (lambda (state)
15            (find-path predecessors state empty graph))
16         final-states))
17    (map (lambda (letters)
18         (apply string-append
19           (map (lambda (letter)
20             (format "~a" letter)) letters)))
21       words))
22
```

O Algoritmo 17 encontra o caminho entre um vértice do grafo até o único vértice que não possui um vértice antecessor. Esse vértice sem antecessor sempre representa o estado inicial de um autômato. Além disso, através do caminho encontrado, concatena-se os pesos das arestas do grafos. Como elas representam os símbolos aceitos pelo autômato é possível formar as palavras aceitas por aquele autômato.

Algoritmo 18: Palavras Aceitas

```
1 (define (find-acceptance-words dfa)
2     (define graph (weighted-graph/directed
3     (transform-dfa-graph dfa)))
4     (define-values (distance predecessors)
5     (bfs graph (dfa-start dfa)))
6     (acceptance-words predecessors (dfa-final dfa) graph))
7
```

O Algoritmo 18 cria um grafo direcionado com peso em suas arestas a partir de um AFD. Além disso, ele gera uma lista com os antecessores de cada vértice do grafo, usando o algoritmo

de BFS(Busca em Largura). Com a lista de antecessores, é possível encontrar algumas palavras que o autômato aceita, usando a ideia de que cada peso de aresta representa um símbolo de aceitação.

Algoritmo 19: Algoritmo Correção

```
(define (automaton-correction answer feedback)
    (define dfa1 (automaton-difference answer feedback))
    (define dfa2 (automaton-difference feedback answer))
    (cond
    [(and
      (not (empty? (dfa-final dfa1)))
      (not (empty? (dfa-final dfa2)))
        (displayln (string-append
            (format "~a: ~a" "Your automaton is accepting a word(s) that it should not" (find-acceptance-words dfa1))
            "\n"
            (format "~a: ~a" "Your automaton is not accepting word(s) that it should" (find-acceptance-words dfa2)))))]
    [(not (empty? (dfa-final dfa1))) (format "~a: ~a" "Your automaton is accepting a word(s) that it should not" (find-acceptance-words dfa1))]
    [(not (empty? (dfa-final dfa2))) (format "~a: ~a" "Your automaton is not accepting word(s) that it should" (find-acceptance-words dfa2))]
    [else (displayln "Your automaton accept the same words of the feedback")]))

```

O Algoritmo 19 executa todos os passos necessários para que se mostre para o aluno se o autômato criado por ele esta correto ou não. Primeiro é feita a diferença entre o AFD de resposta do aluno e o do gabarito. Após isso, é feita a verificação se os autômatos gerados pelas diferenças são vazios. Caso não sejam vazios, é impresso no console algumas palavras que são aceitas por esses AFDs.

3.2.3.2 Execução de Correção de Múltiplos Exercícios

Nesta seção será mostrado como é possível executar a correção de múltiplos exercícios, disponibilizando os autômatos de resposta do aluno e os AFDs de gabarito por meio de um arquivo JSON.

3.2.3.2.1 Formatação do Arquivo de Respostas

Para que seja possível descrever os autômatos em JSON, foi necessário criar uma formatação específica. No arquivo, obtém-se uma lista com vários exercícios que contêm as informações

de qual exercício se trata, as características de um autômato de resposta do aluno e as características do AFD de gabarito. Dessa forma, o json precisa estar no seguinte formato:

```
[
    {
        "question": "question1",
        "answer": {
            "type": "dfa",
            "states": ["A", "B"],
            "start": "A",
            "final": ["B"],
            "sigma": [0, 1],
            "delta": [
                {
                    "origin-state": "A",
                    "destiny-state": "B",
                    "symbol": 0
                }
            ]
        },
        "feedback": {
            "type": "dfa",
            "states": ["C", "D"],
            "start": "C",
            "final": [ "D"],
            "sigma": [0, 1],
            "delta": [
                {
                    "origin-state": "C",
                    "destiny-state": "D",
                    "symbol": 0
                }
            ]
        }
    }
]
```

O atributo *question* é um identificador de qual exercício se trata. O atributo *answer* é um objeto que representa o automato de resposta do aluno e o atributo *feedback* é um objeto que representa o AFD gabarito.

Olhando para os objetos que representam autômatos, encontram-se os seguintes atributos que tem como significado:

- *type*: o tipo de autômato finito descrito
- *states*: é uma lista com todos os estados existentes no autômato

- *start*: é o estado inicial do autômato, podendo ser uma lista de estados iniciais caso se trate de um AFN
- *final*: é uma lista com todos os estados finais de um autômato finito
- *delta*: é uma lista com todas as transições do autômato

O atributo *delta* também se trata de um objeto, dessa forma, ele também possui atributos. O significado de cada um é:

- *origin-state*: estado de origem da transição
- *destiny-state*: estado de destino da transição
- *symbol*: símbolo que pode ser processado na transição

3.2.3.2.2 Correção de Múltiplos Exercícios

Para corrigir diversos exercícios em lote foi criado um código que lê um JSON, como o definido na Seção 3.2.3.2.1, e corrige os exercícios um a um imprimindo no terminal os contraexemplos encontrados, caso não haja contraexemplo, é feita a impressão de uma mensagem indicando que a resposta do aluno esta correta.

Algoritmo 20: Autômato do JSON para Estrutura Racket

```
1
2 (define (mk-automato-json json-data)
3   (define states (map string->symbol (hash-ref json-data 'states)))
4   (define start
5     (if (list? (hash-ref json-data 'start))
6       (map string->symbol (hash-ref json-data 'start))
7       (string->symbol (hash-ref json-data 'start))))
8   (define final (map string->symbol (hash-ref json-data 'final)))
9   (define sigma (hash-ref json-data 'sigma))
10  (define delta (transform-json-delta (hash-ref json-data 'delta)))
11  (if (string=? "dfa" (hash-ref json-data 'type))
12    (mk-dfa states sigma delta start final)
13    (mk-nfa states sigma delta start final)))
14
```

O Algoritmo 20 cria um autômato finito baseado nas descrições definidas no JSON utilizando um *hash* como estrutura de leitura do JSON.

Algoritmo 21: Transição do JSON para Estrutura Racket

```
(define (transform-json-delta transitions)
  (map (lambda (t)
      (cons
        (cons
          (string->symbol (hash-ref t 'origin-state))
          (if (string? (hash-ref t 'symbol))
            (string->symbol (hash-ref t 'symbol))
            (hash-ref t 'symbol)))
        (string->symbol (hash-ref t 'destiny-state)))) transitions))

```

O Algoritmo 21 transforma uma lista de objetos que representam transições no JSON para uma estrutura que descreve transições para o algoritmo de correção.

Algoritmo 22: Lista de Pares de Respostas

```
(define (mk-pairs-answer json-data)
  (map (lambda (automato)
    (cons (cons
        (mk-automato-json (hash-ref automato 'answer))
        (mk-automato-json (hash-ref automato 'feedback)))
      (hash-ref automato 'question))) json-data))

```

O Algoritmo 22 gera uma lista de pares, que contem como primeiro valor do par, um par com a reposta e o gabarito e como segundo valor o identificador do exercício.

Algoritmo 23: Correção de Múltiplos Exercícios

```
(define (run-answers-batch json-data)
  (define pairs-answer (mk-pairs-answer json-data))
  (map (lambda (pair)
    (displayln
      (string-append "\n" (cdr pair)))
    (automaton-correction
        (car (car pair))
        (cdr (car pair)))) pairs-answer))

```

Por fim, o Algoritmo 23 executa o algoritmo de correção 19 para cada resposta e gabarito encontrado no JSON mostrando primeiro de qual exercício se trata e o resultado da correção logo em seguida.

3.2.3.2.3 Exemplo de Execução

Com o intuito de auxiliar a compreensão do funcionamento do algoritmo de correção, considere os Exemplos 3.1 e 3.2 demonstrados na Seção 3.1.

Exemplo 3.5. O Exemplo 3.1 cria dois AFDs que são equivalentes. Dessa forma, o resultado esperado para a execução do algoritmo de correção deve ser:
`Your automaton accept the same words of the feedback.`

Para executar o algoritmo do exemplo 3.1, primeiro é necessário descrever ambos os autômatos no arquivo JSON. Ele deverá ser da seguinte maneira:

```
[
    {
        "question": "AFDs Equivalentes",
        "answer": {
            "type": "dfa",
            "states": ["C", "D", "E"],
            "start": "C",
            "final": ["C", "E"],
            "sigma": [0, 1],
            "delta": [
                {
                    "origin-state": "C",
                    "destiny-state": "C",
                    "symbol": 1
                },
                {
                    "origin-state": "C",
                    "destiny-state": "D",
                    "symbol": 0
                },
                {
                    "origin-state": "D",
                    "destiny-state": "D",
                    "symbol": 1
                },
                {
                    "origin-state": "D",
                    "destiny-state": "E",
                    "symbol": 0
                },
                {
                    "origin-state": "E",
                    "destiny-state": "E",
                    "symbol": 1
                },
                {
```

```
37                      "origin-state": "E",
38                      "destiny-state": "D",
39                      "symbol": 0
40                  },
41              ]
42          },
43          "feedback": {
44              "type": "dfa",
45              "states": ["A", "B"],
46              "start": "A",
47              "final": ["A"],
48              "sigma": [0, 1],
49              "delta": [
50                  {
51                      "origin-state": "A",
52                      "destiny-state": "A",
53                      "symbol": 1
54                  },
55                  {
56                      "origin-state": "A",
57                      "destiny-state": "B",
58                      "symbol": 0
59                  },
60                  {
61                      "origin-state": "B",
62                      "destiny-state": "B",
63                      "symbol": 1
64                  },
65                  {
66                      "origin-state": "B",
67                      "destiny-state": "A",
68                      "symbol": 0
69                  }
70              ]
71          }
72      }
73  ]
```

Nesse JSON o autômato descrito no objeto `answer` corresponde ao autômato A do Exemplo 3.1 e o objeto `feedback` corresponde ao AFD G do mesmo exemplo. Ao se executar o algoritmo, usando a IDE DrRacket ou via terminal o seguinte resultado será impresso no terminal: `Your automaton accept the same words of the feedback.`

Exemplo 3.6. O Exemplo 3.2 cria dois AFDs que não são equivalentes. Dessa forma, o resultado esperado para a execução do algoritmo de correção deve ser:
`AFDs Não Equivalentes`

```
'("Your automaton is accepting a word(s) that it should not: (000)")
```

Para executar o algoritmo do Exemplo 3.2, primeiro é necessário descrever ambos os autômatos no arquivo JSON. Ele deverá ser da seguinte maneira:

```
1  [
2      {
3          "question": "AFDs Nao Equivalentes",
4          "answer": {
5              "type": "dfa",
6              "states": ["C", "D", "E"],
7              "start": "C",
8              "final": ["C", "E"],
9              "sigma": [0, 1],
10             "delta": [
11                 {
12                     "origin-state": "C",
13                     "destiny-state": "C",
14                     "symbol": 1
15                 },
16                 {
17                     "origin-state": "C",
18                     "destiny-state": "D",
19                     "symbol": 0
20                 },
21                 {
22                     "origin-state": "D",
23                     "destiny-state": "D",
24                     "symbol": 1
25                 },
26                 {
27                     "origin-state": "D",
28                     "destiny-state": "E",
29                     "symbol": 0
30                 },
31                 {
32                     "origin-state": "E",
33                     "destiny-state": "E",
34                     "symbol": 1
35                 }
36             ]
37         },
38         "feedback": {
39             "type": "dfa",
40             "states": ["A", "B"],
41             "start": "A",
42             "final": ["A"],
43             "sigma": [0, 1],
```

```
44            "delta": [
45                {
46                    "origin-state": "A",
47                    "destiny-state": "A",
48                    "symbol": 1
49                },
50                {
51                    "origin-state": "A",
52                    "destiny-state": "B",
53                    "symbol": 0
54                },
55                {
56                    "origin-state": "B",
57                    "destiny-state": "B",
58                    "symbol": 1
59                },
60                {
61                    "origin-state": "B",
62                    "destiny-state": "A",
63                    "symbol": 0
64                }
65            ]
66        }
67    }
68 ]
```

Nesse JSON o autômato descrito no objeto `answer` corresponde ao autômato A do Exemplo 3.2 e o objeto `feedback` corresponde ao AFD G do mesmo exemplo. Ao se executar o algoritmo, usando a IDE DrRacket ou via terminal o seguinte resultado será impresso no terminal:

```
AFDs Não Equivalentes
'("Your automaton is accepting a word(s) that it should not: (000)")
```

4 Considerações Finais

Neste trabalho monográfico apresentamos o desenvolvimento de uma ferramenta para correção automática de exercícios sobre autômatos finitos determinísticos. A estratégia de correção possui uma sólida fundamentação matemática: o uso de propriedades de fechamento de linguagens regulares, algoritmos de minimização de autômatos e o uso de algoritmos clássicos da teoria de grafos para o processo de construção de contra-exemplos, em caso da solução proposta por um aluno ser incorreta.

Trabalhos futuros envolvem: 1) Tornar a ferramenta disponível publicamente como um pacote da linguagem Racket, permitindo seu uso por interessados em teoria de linguagens formais. 2) Utilizar a ferramenta em cursos introdutórios de teoria da computação e avaliar seu impacto no aprendizado do conteúdo da disciplina. 3) Elaboração de um parser para um formato de descrição de autômatos e integração do algoritmo de correção ao Jupyter notebooks, para permitir a solução de exercícios e sua respectiva correção.

Referências

ALUR, R.; D'ANTONI, L.; GULWANI, S.; KINI, D. Automated grading of dfa constructions. In: *IJCAI '13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. [s.n.], 2013. p. 1976–1982. ISBN 978-1-57735-633-2. Disponível em: <https://www.microsoft.com/en-us/research/publication/automated-grading-dfa-constructions/>.

D'ANTONI, L.; KINI, D.; ALUR, R.; GULWANI, S.; VISWANATHAN, M.; HARTMANN, B. How can automatic feedback help students construct automata? *ACM Trans. Comput.-Hum. Interact.*, Association for Computing Machinery, New York, NY, USA, v. 22, n. 2, mar 2015. ISSN 1073-0516. Disponível em: <https://doi.org/10.1145/2723163>.

ELLSON, J.; GANSNER, E.; KOUTSOFIOS, L.; NORTH, S. C.; WOODHULL, G. Graphviz—open source graph drawing tools. In: MUTZEL, P.; JÜNGER, M.; LEIPERT, S. (Ed.). *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 483–484. ISBN 978-3-540-45848-7.

GRAMOND, E.; RODGER, S. H. Using jflap to interact with theorems in automata theory. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, p. 336–340, mar 1999. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/384266.299800>.

GRINDER, M. T. Animating automata: A cross-platform program for teaching finite automata. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 34, n. 1, p. 63–67, feb 2002. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/563517.563364>.

GRINDER, M. T. A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 35, n. 1, p. 157–161, jan 2003. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/792548.611958>.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321455363.

KOZEN, D. C. *Automata and Computability*. 1st. ed. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN 0387949070.

ROBINSON, M. B.; HAMSHAR, J. A.; NOVILLO, J. E.; DUCHOWSKI, A. T. A java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In: *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 1999. (SIGCSE '99), p. 105–109. ISBN 1581130856. Disponível em: <https://doi.org/10.1145/299649.299704>.

SIPSER, M. Introduction to the theory of computation (3rd international ed.). *Cengage Learning*, 2013.