Carta de apresentação

Santo André, 26 de fevereiro de 2016.

À Ilustríssima Pró-Reitora de Pesquisa,

Profª. Dra. Marcela Sorelli Carneiro Ramos

Encaminho o relatório do aluno **Rafael Cardoso da Silva** referente ao projeto de pesquisa junto ao programa de Iniciação Científica na modalidade **PIC** no edital **01/2015.**

O aluno está obtendo um bom desemprenho durante a pesquisa, estudando as literaturas fundamentais para obter o conhecimento necessário dos métodos que são empregados para a implementação do projeto. E aprendendo muitos conceitos referentes a como produzir um material acadêmico. Ele tem as suas dúvidas ocasionais sanadas em reuniões semanais, onde acompanho de perto o desenvolvimento do projeto, revisando e testando seus resultados.

Nome do Orientador: **Daniel M. Martin**

Rafael Cardoso da Silva

Um sistema para auxiliar na aprendizagem da disciplina Linguagens Formais e Autômatos

Santo André - SP, Brasil

2016

Rafael Cardoso da Silva

Um sistema para auxiliar na aprendizagem da disciplina Linguagens Formais e Autômatos

Relatório parcial como parte do projeto de Iniciação Científica apresentado a UFABC.

Universidade Federal do ABC

Bacharelado em Ciências da Computação

Orientador: Prof. Daniel M. Martin

Santo André - SP, Brasil

2016

Resumo

Resolver exercícios é uma prática fundamental para um aluno sobre os conceitos ministrado em aula. Mas a sua correção também faz parte, para que ele possa avaliar o seu aprendizado. Na UFABC, a disciplina de Linguagem Formais e Autômatos contempla vários exercícios que se tornam inviável a correção em tempo hábil para turmas grandes, pois admitem infinitas respostas. O objetivo deste projeto foi a criação e implementação de um sistema para aplicação e correção de exercícios. Envolvendo o estudo de métodos e algoritmos presentes na literatura, para o teste de equivalência entre a resposta do aluno e o gabarito. Além dos estudos para criação do sistema. Ao final do projeto, o sistema contemplará uma turma teste, afim de testar a qualidade do sistema e os resultados obtidos por eles ajudarão a compor o seu conceito final na disciplina.

Palavras-chave: autômato finito, equivalência de autômatos, minimização de autômato, Programação para Web

Sumário

1 Introdução

Resolução de exercícios é uma atividade fundamental para que um aluno possa praticar o que foi ministrado em aula. A correção dos exercícios também é importante para que o aluno possa avaliar seu aprendizado. Para turmas grandes, torna-se inviável corrigir rapidamente todos os exercícios propostos e devolvê-los em tempo hábil aos alunos. Na disciplina MC3106 – Linguagens Formais e Autômatos – vários exercícios da parte inicial da disciplina pedem como resposta um autômato finito determinístico. Tais exercícios são particularmente trabalhosos de se corrigir porque cada um deles admite infinitas respostas corretas e, por esse motivo, sua correção se torna exaustiva para o monitor ou docente responsável.

Com o intuito de agilizar essa tarefa, este projeto propõe o desenvolvimento de um sistema web que permita ao aluno inserir sua resposta-autômato por meio de uma interface intuitiva e fácil de ser utilizada e obter um *feedback* quase que imediatamente, um tempo drasticamente reduzido se comparado à correção manual. Acreditamos que um sistema como esse será de grande valia para o aprendizado dos alunos. Por outro lado, para o docente responsável, facilitará o processo de composição do conceito final na disciplina de Linguagens Formais e Autômatos.

Além do estudo de autômatos e linguagens regulares, a disciplina de Linguagens Formais e Autômatos trata de linguagens não-regulares, em especial das que podem ser descritas por gramáticas livres de contexto (GLC). Nesta segunda parte da disciplina, é também comum a ocorrência de exercícios que pedem, como resposta, uma gramática livre de contexto para descrever uma linguagem livre de contexto. Automatizar a correção deste tipo de exercício é impossível pois o problema geral de se determinar se duas gramaticas livres de contexto descrevem a mesma linguagem é um problema indecidível. Por outro lado, diversos tópicos em linguagens livres de contexto podem ser examinados de maneira automática, incluindo a equivalência de certas subclasses de gramáticas livres de contexto (NIJHOLT, 1981), a transformação de uma gramática livre de contexto qualquer para a Forma Normal de Chomsky (FNC), a busca a árvore de derivação para um texto segundo uma GLC na FNC, problemas de *parsing* em geral.

Se houver tempo, acreditamos que seria interessante agregar ao sistema alguns scripts de correção automática para classes de exercícios envolvendo gramáticas livres de contexto.

2 O principal problema

Um *autômato finito determinístico* (também chamado de AFD ou máquina de estados) consiste de um conjunto de estados não vazio Q, um alfabeto Σ, um estado inicial $s \in Q$, um conjunto de estados de aceitação $F \subseteq Q$ e uma função de transição $\delta\colon Q \times \Sigma \to Q$. É comum estender a função de transição para uma função $\hat{\delta}\colon Q \times \Sigma^* \to Q$ definida indutivamente por

(i) $\hat{\delta}(q, \varepsilon) = q$ para todo estado $q \in Q$;

(ii) $\hat{\delta}(q, \sigma) = \delta(q, \sigma)$ para todo estado $q \in Q$ e símbolo $\sigma \in \Sigma$;

(iii) $\hat{\delta}(q, \sigma_1 \cdots \sigma_k) = \delta(\hat{\delta}(q, \sigma_1 \cdots \sigma_{k-1}), \sigma_k)$ para todo estado $q \in Q$ e símbolos $\sigma_1, \ldots, \sigma_k \in \Sigma$.

Todo autômato pode ser representado por um diagrama de estados. Considere o diagrama da Figura 1 abaixo.

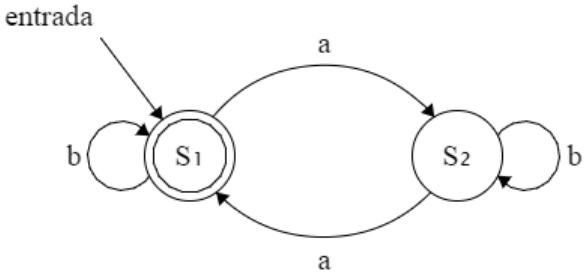


Figura 1 – Um autômato M_1.

No autômato da Figura 1, temos $Q = \{S_1, S_2\}$, $\Sigma = \{a, b\}$, $s = S_1$, $F = \{S_1\}$ e a função de transição é dada pela tabela a seguir.

δ	**a**	**b**
S_1	S_2	S_1
S_2	S_1	S_2

Tabela 1 – Tabela de transição do M_1.

Nesse exemplo, se o AFD M_1 é alimentado com a palavra **aabab**, ele irá começar no estado S_1 e irá percorrer os estados S_2, S_1, S_1, S_2, S_2 nesta ordem, e irá rejeitar a palavra dada (pois o último estado $S_2 \notin F$). Se, para uma outra palavra w, a simulação tivesse terminado em S_1 (que pertence a F) diríamos que o autômato aceita a palavra w. Mais formalmente, se $M_1 = (Q, \Sigma, \delta, s, F)$ é um autômato finito determinístico, dizemos que M_1 *aceita* w se $\hat{\delta}(s, w) \in F$ e que M_1 *rejeita* w caso contrário.

A *linguagem reconhecida* por um autômato $M_1 = (Q, \Sigma, \delta, s, F)$ é o conjunto de palavras que são aceitas por esse autômato, ou seja, é definida pelo conjunto

$$L(M_1) = \{w \in \Sigma^* : \hat{\delta}(s, w) \in F\}.$$

No exemplo da Figura 1 acima, a linguagem reconhecida por por M_1 é $L(M_1) = \{w \in \Sigma^* : w$ tem número par de símbolos $\mathtt{a}\}$.

Vários tipos de exercícios pedem por uma resposta que é um autômato. Por exemplo:

(i) construir um autômato que reconheça a linguagem regular dada por um certo conjunto de palavras descrito matematicamente;

(ii) transformar um autômato finito não determinístico num determinístico equivalente;

(iii) criar um autômato que reconheça a mesma linguagem descrita por uma expressão regular dada;

(iv) construir o menor autômato finito determinístico que reconheça uma certa linguagem regular.

Cada um destes tipos de exercícios possuem centenas de casos particulares que os alunos podem fazer para praticar os conceitos de autômatos abordados na disciplina. Tais exercícios podem, por exemplo, ser encontrados abundantemente no livro *Introdução à Teoria da Computação* de M. Sipser (SIPSER, 2012). Outras referências importantes da disciplina que possuem exercícios semelhantes são (HOPCROFT; MOTWANI; ULLMAN, 2006) e (LINZ, 2011). Em todos os casos acima, contudo, há uma infinidade de respostas corretas para cada exercício. Por exemplo, considere o diagrama abaixo.

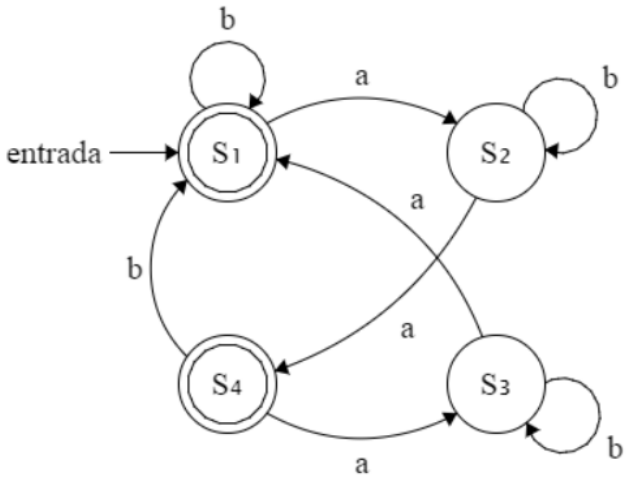


Figura 2 – Um autômato M_2.

Apesar de visivelmente maior e mais complexo que M_1, este autômato reconhece a mesma linguagem que o autômato M_1 da Figura 1. É possível demonstrar que, para cada linguagem regular, existem infinitos autômatos que a reconhecem. Como decidir então se o autômato-resposta dado por um aluno reconhece a linguagem pedida? No caso do sistema que iremos desenvolver isso se reduz ao seguinte problema:

Problema: Decidir se dois autômatos finitos determinísticos reconhecem a mesma linguagem.

3 Métodos para resolve o problema da equivalência

Determinar se dois autômatos finitos e determinísticos reconhecem linguagens diferentes é um problema bem resolvido. A ideia fundamental é tentar encontrar uma palavra w que é aceita por um autômato e rejeitada pelo outro. Se tal palavra puder ser encontrada, então as linguagens reconhecidas por esses autômatos são distintas. Caso tal palavra não exista, os autômatos reconhecem a mesma linguagem.

Mais formalmente, dizemos que dois estados q_1 e q_2 (no mesmo autômato ou não) são *equivalentes* se, para toda palavra $w \in \Sigma^*$ vale que $\hat{\delta}(q_1, w) \in F$ se e somente se $\hat{\delta}(q_2, w) \in F$, caso contrário q_1 e q_2 sao chamados distinguíveis. Dois autômatos serão, portanto, equivalentes se e somente se seus respectivos estados iniciais forem equivalentes no sentido que acabamos de definir.

Esse problema está intimamente relacionado com o problema de minimização de autômatos finitos determinísticos. Os primeiros algoritmos desenvolvidos para testar a equivalência de autômatos resolviam também o problema da minimização. Eles apareceram em (HUFFMAN, 1954) e (MOORE, 1956) e podem ser implementados sem muito esforço em tempo $O(n^4)$, onde n é o número total de estados. O algoritmo mais eficiente conhecido para minimização de autômatos (HOPCROFT, 1971) executa em tempo $O(n \log n)$. Posteriormente Hopcroft e Karp (HOPCROFT; KARP, 1971b) desenvolveram um algoritmo linear para testar a equivalência de autômatos.

3.1 Minimização de Moore

Nessa seção vamos investigar um algoritmo de minimização de autômato finito determinístico, mais especificamente o algoritmo de Minimizacao de Moore (HUFFMAN, 1954).

Para facilitar a compreensão utilizaremos como exemplo o autômato M_2 da Figura 2. Queremos determinar se M_2 é equivalente ao autômato M_1 da Figura 1.

O autômato M_2 é representado pela tupla $M_2 = (Q, \Sigma, \delta, s, F)$, onde temos $Q = \{S_1, S_2, S_3, S_4\}$, $\Sigma = \{a, b\}$, $s = S_1$, $F = \{S_1, S_4\}$ e a função de transição é dada pela tabela a seguir.

δ	**a**	**b**
S_1	S_2	S_1
S_2	S_4	S_2
S_3	S_1	S_3
S_4	S_3	S_1

Tabela 2 – Tabela de transição do M_2.

Note que a definição de estados equivalentes na seção anterior não se traduz imediatamente em um algoritmo, pois não é possível testar se $\hat{\delta}(p, w) \in F$ para todas as palavras $w \in \Sigma^*$

No entanto, podemos desenvolver um algoritmo para determinar a equivalência entre estados p e q baseando-se no seguinte lema:

Lema 1. *Se $r = \delta(q, \sigma)$ e $s = \delta(p, \sigma)$ e se r e s são distinguíveis, então p e q são distinguíveis.*

Prova. Por hipótese dever haver uma palavra w que distingue r de s, ou seja, exatamente um dos estados $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$ é de aceitação. Então a palavra σw distingui p de q, já que $\hat{\delta}(p, \sigma w)$ e $\hat{\delta}(q, \sigma w)$ é o mesmo par de estados de $\hat{\delta}(r, w)$ e $\hat{\delta}(s, w)$. □

Entendido o processo de como verificar se um par de estados são distinguíveis, podemos ver o pseudo código que realiza as verificações para todos os possíveis pares, logo a baixo.

Algoritmo 1: Acha Pares Distinguíveis pela Minimização de Moore

Entrada: Q, Σ, δ, F

Saída: Tabela de distinguibilidade

```
1  início
2      para cada p ∈ Q faça
3          para cada q ∈ Q faça
4              D{p, q} ← ∅
5      para cada p ∈ F faça
6          para cada q ∈ Q\F faça
7              D{p, q} ← x
8      distinguiu_algo ← verdadeiro
9      enquanto distinguiu_algo faça
10         distinguiu_algo ← falso
11         para cada p ∈ Q faça
12             para cada q ∈ Q faça
13                 se D{p, q} = ∅ então
14                     para cada σ ∈ Σ faça
15                         se D{δ(p, σ), δ(q, σ)} ≠ ∅ então
16                             D{p, q} ← x
17                             distinguiu_algo ← verdadeiro
18     retorna D
```

Ao analisarmos este pseudo código de algoritmo de minimização vemos que este recebe como entrada o conjunto de estados Q, o alfabeto Σ, a tabela de transição δ e o conjunto de estados finais F de um autômato.

E contamos com o apoio de uma matriz quadrada D de dimensão $|Q|$, cujas células abaixo da diagonal principal serão utilizadas para representar todos os pares de estados possíveis. Inicialmente, será inicializada com valores vazios. Ao longo da execução do algoritmo, se um par for distinguido, então atribuiremos o simbolo "x" à célula correspondente a este par. Logo em seguida, preenchemos as células dos pares trivialmente distinguíveis, ou seja, pares que consistem de um estado final e um não final.

Então iniciam-se as rodadas de verificação, onde percorre-se D e para cada par $\{p, q\}$ com valor vazio, o algoritmo testa para cada carácter σ do conjunto Σ, se o par $\{p, q\}$ se torna distinguível pelo caractere σ, considerando $\hat{\delta}(p, w)$ e $\hat{\delta}(q, w)$ e o Lema 1.

Além disto, com o apoio da variável booleana *distinguiu_algo*, utilizada para

marcar se houve uma alteração em D, verificamos que as rodadas somente cessarão se, na mesma rodada, nenhum "x" é adicionado na tabela de distinguibilidade, concluindo assim que não existem mais pares à ser distinguídos. Retornando ao fim a tabela de distinguibilidade.

Concluímos que não pode haver mais de $O(n^2)$ verificações em uma rodada, assim certamente o limite superior para o número de verificações é $O(n^4)$, visto que o pior caso é atingido quando somente um único par de estados é distinguido por rodada, causando $O(n^2)$ rodadas com $O(n^2)$ verificações em cada, onde n é o tamanho do conjunto Q.

Então, para solucionamos o nosso problema de verificar se dois autômatos são equivalentes, podemos considerar um terceiro autômato que é a união disjunta dos dois autômatos dados e executamos o algoritmo acima sobre a união. E, ao final, apenas verificamos se a posição na tabela que representa o par de estados iniciais encontra-se vazia.

Para o nosso exemplo, ao unirmos o M_1 com o M_2, obtemos a seguinte tabela de transição .

δ	**a**	**b**
S_1	S_2	S_1
S_2	S_1	S_2
S_3	S_4	S_3
S_4	S_6	S_4
S_5	S_3	S_5
S_6	S_5	S_3

Tabela 3 – Tabela de Transição dos autômatos M_1 e M_2 unidos.

E ao executar o algoritmos de equivalência, obteremos a seguinte tabela de distinguibilidade.

S_2	x				
S_3		x			
S_4	x		x		
S_5	x		x		
S_6		x		x	x
	S_1	S_2	S_3	S_4	S_5

Tabela 4 – Tabela de distinguibilidade dos autômatos M_1 e M_2 unidos.

Basta agora verificar se a posição da matriz D correspondente ao par de estados iniciais está com o seu valor vazio, o que indica que os estados iniciais são equivalentes. Neste caso os autômatos M_1 e M_2 são equivalentes.

3.1.1 Detalhes de Implementação do algoritmo de minimização

Como visto na seção anterior, o que faz o Algoritmo 1 estar sobrecarregado é o fato de que em todas as rodadas de verificação da distinguibilidade dos pares ele procura pelo próximo par de estados que ainda não foi distinguido, que é feita pelas linhas 11, 12 e 13 do pseudo código anterior. Assim uma solução possível para melhorar a eficiência do algoritmo seria manter informações extras na memória de modo que, ao realizar uma verificação da distinguibilidade de um par, ele já deve ter conhecimento de modo rápido e direto qual é o próximo par que ainda não foi distinto à ser checado.

Assim, intuitivamente, se faz necessário o uso de uma estrutura de dados para abrigar os índices das células desta matriz D, que representa um par. E esta estrutura de dados terá como regra de inserção e remoção de elementos no formato de que o primeiro a entrar será o primeiro a sair, ou seja, uma fila. Então todos os elementos que estiverem contidos nesta fila, obrigatoriamente ainda não foram distinguidos.

Temos então que fazer algumas alterações no código do Algoritmo 1. Anteriormente à linha 9, em que se dá o inicio das rodadas de verificação, devemos inserir na fila todos os pares que não foram trivialmente distintos e ao final adicionamos um elemento chave, que tem a função de marcar o final de uma rodada.

A seguir inicia-se as rodadas de verificação, em que será removido um a um os elementos da fila e verificamos primeiramente se este elemento se trata do elemento chave. Se sim, com o auxilio da variável booleana $distinguiu\_algo$, verificaremos se foi distinguido algum par de estados durante a rodada. Se não houve um novo par distinguido, então encerra-se as rodadas, caso contrario inserimos o elemento chave de volta a fila.

Mas caso não seja o elemento chave, tentamos verificar se o par que este elemento representa pode ser distinguido para todas os símbolos de Σ conforme o Lema 1. E se aquele par for distinguível, marcamos a celula de D como distinguido. Se e somente se o par não for distinguido, ele será novamente inserido na fila, para que na próxima rodada o torne a verificar.

Assim ao final das rodadas de verificação, os elementos contido na fila serão os representantes dos pares indistinguíveis.

Com esta estratégia, conseguiremos eliminar as verificações dos pares que já foram distinguidos durante as rodadas que o algoritmo executa. Utilizando mais memória para armazenar esta estrutura de dado que auxilia na verificação, mas proporcionando uma melhora significativa no desempenho do Algoritmo 1.

3.1.2 Recuperação de Palavra

Este sistema de apoio a aprendizagem tem como objetivo o rápido feedback sobre a resposta submetida pelo usuário. Com a ajuda do Algoritmo 1 conseguimos verificar se a resposta é equivalente ao gabarito perfeitamente. Mas, caso o Algoritmo 1 retornar que não são equivalentes, então se faz necessário de uma dica ao usuário para que ele tome conhecimento de seu erro. Esta dica será uma palavra $w \in \Sigma^*$, que distingue o estado inicial do autômato submetido do estado inicial do autômato gabarito, e ao ser percorrida pelo usuário, permite encontrar o erro cometido em sua resposta.

Então é necessária a implementação de um algoritmo que descubra qual é essa palavra que distingue os dois autômatos. Para que seja possível encontrar a palavra desejada é necessário modificar o Algoritmo 1, para que ele não somente descubra quais pares de estados são distinguíveis, mas também guarde os símbolos iniciais das palavras que as distinguem. Isso pode ser feiro alterando-se apenas as linhas 7 e 16 do algoritmo como indicado a seguir:

7. $D\{p, q\} \leftarrow \varepsilon$
16. $D\{p, q\} \leftarrow \sigma$

Então supondo que $D\{p, q\} \neq \emptyset$, temos o seguinte pseudo código.

Algoritmo 2: Acha a palavra que distingui os dois autômatos

Entrada: $D, \delta, \{p_0, q_0\}$
Saída: w

```
1 início
2     w ← ε
3     {p, q} ← {p_0, q_0}
4     σ ← D({p, q})
5     enquanto σ ≠ ε faça
6         w ← w · σ
7         {p, q} ← {δ(p, σ), δ(q, σ)}
8         σ ← D({p, q})
9     retorna w
```

O algoritmo recebe como entrada D, δ e $\{p_0, q_0\}$, que é a tabela de distinguibilidade gerado pelo Algoritmo 1 com as modificações sugeridas acima, a tabela de transição dos autômatos unidos e o par de estados iniciais dos autômatos, respectivamente.

O Algoritmo 2 irá coletando os símbolos contidos em células especificas de D e unindo-os ao conjunto w. Inciando pelo par $\{p_0, q_0\}$, descobrindo o simbolo que os distinguem e atribuindo a σ. E enquanto não encontrar o simbolo σ com o valor ε, unirá ao conjunto w o simbolo σ, em seguida descobre o próximo par de estados que o simbolo

σ levará utilizando a tabela de transição, ou seja, o novo par será $\{\delta(p,\sigma),\delta(q,\sigma)\}$, e por ultimo descobre qual é o simbolo que distinguiu este próximo par de estados.

Então será retornado o a palavra w, que é a palavra que distinguiu os dois autômatos que estávamos procurando.

3.2 Equivalência em tempo linear

A maioria dos algoritmos para teste de equivalência de autômatos finitos determinísticos na literatura tem crescimento assintótico proporcional ao quadrado do número de estados (HOPCROFT; MOTWANI; ULLMAN, 2006). Um algoritmo desenvolvido por Hopcroft e Karp (HOPCROFT; KARP, 1971a), para este mesmo fim, tem complexidade de $O(n \log^* n)$, onde n é o número de estados. Então estudaremos este algoritmo nesta seção.

Mas antes temos que entender a estrutura de dados utilizada para armazenar conjuntos disjuntos, conhecida popularmente como *Union-Find*, que será fundamental para o método de equivalência de Hopcroft e Karp.

3.2.1 Union-Find

Para o método que veremos na seção seguinte, envolve o agrupamento de elementos em uma coleção de conjuntos disjuntos[1], e há uma estrutura de dados que atende este requisito (CORMEN et al., 2001) e admite duas operações que será vital para a implementação do algoritmo de equivalência em tempo linear.

Na estrutura de dados de conjuntos disjuntos guarda-se um coleção de elementos disjuntos. Onde cada conjunto é identificado por um representante, que será um de seus elementos, não importando na maioria dos casos qual seja este elemento, somente que será sempre ele que atenderá quando for procurado o representante daquele conjunto.

Para se criar um conjunto utiliza-se a função `MAKE-SET(`x`)`, que aloca um conjunto contendo um elemento x na memória, este elemento será o representante deste conjunto, e claro x não deverá estar presente em nenhum outro conjunto.

A operação `FIND(`x`)` basicamente tem a função de dado um elemento x, encontrar a qual conjunto este elemento pertence, retornando o seu elemento representante.

A outra operação é o `MERGE(`x`, `y`)`, que tem a função de fundir os dois conjuntos que os elementos x e y pertencem em somente um na coleção.

Há mais de um modo de implementar esta estrutura, usando listas ligadas ou até lista de árvores, por exemplo.

[1] São quando os conjuntos não tem elementos em comum, ou seja, suas interseções é o conjunto vazio.

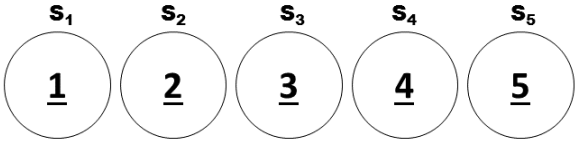
Com a lista ligada, temos o primeiro elemento de cada lista sendo o representante, e cada objeto desta lista tem um ponteiro para o próximo objeto e um ponteiro direto para o representante. Deste modo facilita a operação FIND por ter um ponteiro direto para o representante. Mas para o MERGE, não basta somente o ultimo objeto do primeiro conjunto apontar para o inicio do segundo conjunto, há a necessidade de atualizar o ponteiro para o novo representante de todos os elementos do segundo conjunto. Causando um aumento na complexidade do algoritmo.

Já se utilizarmos uma estrutura de árvore, em que cada elemento será um objeto que representa um nó na árvore. Então este nó somente basta ter um ponteiro para seu nó pai, e o representante será aquele que apontar para ele mesmo. A operação MERGE se torna mais simples, mas o ponteiro da raiz do segundo apontar para a raiz do primeiro conjunto. Já o FIND terá que percorrer a partir do pais deste elemento, todos os pais de seus ancestrais, até encontrar a raiz, que é o representante deste conjunto. Isso pode se tornar custoso caso a altura da árvore seja muito grande, por isso uma otimização da estrutura se faz necessária, utilizando um método de compactação do caminho até a raiz.

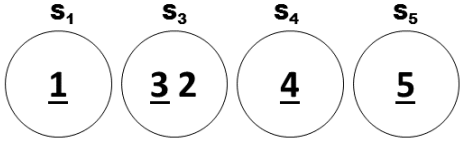
Cada implementação tem a sua particularidade e a escolha depende de seu uso, para o Algoritmo de Hopcroft e Karp, será implementada como modo de árvores, que discutiremos e analisaremos melhor nas seções seguintes.

Para ilustrar melhor esta estrutura, temos o seguinte exemplo:

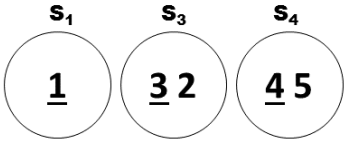
- Criamos uma coleção de conjuntos disjuntos $\Psi = \{S_1, S_2, S_3, S_4, S_5\}$, onde para cada conjunto S_i temos o elemento i contido nele e este será o representante do conjunto disjunto. Criando $\{\{\underline{1}\}, \{\underline{2}\}, \{\underline{3}\}, \{\underline{4}\}, \{\underline{5}\}\}$. Os elementos sublinhados caracteriza o representantes daquele conjunto.



- MERGE(2, 3) resulta em $\{\{\underline{1}\}, \{2, \underline{3}\}, \{\underline{4}\}, \{\underline{5}\}\}$.



- MERGE(5, 4) resultado em $\{\{\underline{1}\}, \{2, \underline{3}\}, \{\underline{4}, 5\}\}$.



- FIND(2) será retornado o elemento 3.
- FIND(3) será retornado o elemento 3.

3.2.2 Algoritmo de Hopcroft e Karp

Nesta seção estudaremos o algoritmo de Hopcroft e Karp para o teste de equivalência em tempo linear (HOPCROFT; KARP, 1971a). O algoritmo utilizas-se da estrutura de dados pilha e uma estrutura de dados popularizada como Union-Find para armazenar conjuntos disjuntos (CORMEN et al., 2001). Em especial esta última estrutura de dados tem fundamental importância no algoritmo para testar a equivalência dos autômatos, pois armazenará no mesmo conjunto os estados que são equivalentes.

Vejamos a seguir o algoritmo.

Algoritmo 3: Teste de Equivalência de Hopcroft e Karp

Entrada: $Q, \delta, \{p_0, q_0\}$
Saída: Conjuntos disjuntos

```
1  início
2      conjuntos.INIT(|Q|)
3      conjuntos.MERGE(p_0, q_0)
4      pilha.empilha({p_0, q_0})
5      enquanto pilha ≠ ∅ faça
6          {p, q} ← pilha.desempilha()
7          para cada σ ∈ Σ faça
8              r ← conjuntos.FIND( δ(p, σ) )
9              s ← conjuntos.FIND( δ(q, σ) )
10             se r ≠ s então
11                 conjuntos.MERGE(r, s)
12                 pilha.empilha({r, s})
13     retorna conjuntos
```

A estrutura de dados Union-Find armazena conjuntos disjuntos na memória, e deseja-se executar essencialmente duas operações: MERGE(i, j), une o conjunto i ao conjunto j, e FIND(i) busca o conjunto ao qual pertence o elemento i.

O algoritmo inicia-se criando a estrutura de dados Union-Find, que atribuímos o nome de *conjuntos*, e terá o tamanho igual ao tamanho do conjunto Q, que é novamente a união dos dois autômatos que queremos testar sua equivalência. Assim a estrutura conterá $|Q|$ conjuntos e em cada um contendo um único elemento que corresponde a um estado de Q, e cada conjunto é denominado pelo nome do elemento que é o representante daquele conjunto. Ou seja, cada estado de Q está em seu próprio conjunto, sendo ele mesmo o

representante e nome deste conjunto.

Realizaremos a primeira fusão, entre os conjunto p_0 e q_0, pois supomos que se os autômatos são equivalentes, então seus estados iniciais também deverão ser equivalentes. EM seguida utilizaremos uma estrutura de dados de Pilha para armazenar o par destes dois estado iniciais.

Assim inicia-se as verificações até que esta pilha esteja vazia. Primeiro desempilhamos um par da pilha e atribuímos a variáveis $\{p, q\}$. Então para cada simbolo σ do alfabeto Σ será feito algumas etapas. Buscamos qual é o conjunto que contém o estado que foi levado pela função de transição δ ao entrar com o simbolo σ e denominaremos estes de r e s. E verificaremos se $r \neq s$, ou seja, se o conjunto r é diferente do conjunto s. Se sim, então uniremos estes dois conjuntos e empilhamos o par $\{r, s\}$ na pilha.

Assim o algoritmo somete terminará ao analisar todos os estados para todas as letras do alfabeto, retornando ao final o *conjuntos*.

Então para constatar a equivalência, basta examinar em cada conjunto q da lista de conjuntos disjuntos se $q \subset F$ ou $q \subset Q \backslash F$. Ou seja, os dois autômatos são equivalentes se, e somente se em nenhum conjunto da lista há estados de aceitação e de não aceitação no mesmo conjunto.

Os passos das linhas 2, 3, 4 e a verificação final desta lista, são executadas em uma quantidade limitada por uma constante de tempo n, onde n é o número de estados de Q.

Já o tempo de execução da interações da linha 5 até a 12, tem o tempo limitado pelo número de vezes que os pares são removidos da pilha. Que por sua vez, o número de pares retirados da pilha é limitado por n, pois cada vez que um par é inserido na pilha, dois conjuntos são fundidos, assim o número total de conjuntos é diminuído em um. Desde a inicialização, são definidos n conjuntos e no máximo $n - 1$ pares são colocados na pilha.

Então o tempo de execução deste algoritmo para testar a equivalência de dois autômatos finitos é delimitados por uma constante vezes o produto do tamanho do alfabeto pelo o número de estados, ou seja, $O(c \times |\Sigma| \times |Q|)$.

Mas como utilizamos a estrutura de dados Union-Find temos o consumo de tempo limitados superiormente por $O(n \log^* n)$.

3.2.3 Detalhes de implementação do algoritmo de Teste de Equivalência de Hopcroft e Karp

O algoritmo utiliza-se de duas estruturas de dados, uma pilha para armazenar o próximo par a ser analisado e o Union-Find para armazenar conjuntos disjuntos.

A pilha utilizada na implementação é a oferecida pela biblioteca padrão da linguagem. Já o conjunto disjunto pode ser simplificada, pois sua utilização neste algoritmo,

a estrutura de dados não necessita ter alocação de memória dinâmica. Logo, para sua implementação basta a alocação de um único vetor sequencial de tamanho igual ao passado por parâmetro no método INIT. Para isto, os atributos padrão destra estrutura de dados serão simplificados, provendo assim uma economia de memória.

Cada índice deste vetor, representará um elemento a ser agrupado em conjuntos, que por sua vez representa um estado da união dos dois autômatos. O valor daquele elemento no vetor representará o seu pai. E quando este valor apontar para si mesmo, significa que este elemento é o elemento raiz, também chamados de representante do conjunto que ele está contido. O atributo *rank* não se fará necessário para esta implementação simplificada. Esta estrutura se assemelhará a uma árvore, em que cada elemento há somente o ponteiro para seu pai.

Assim com esta estrutura, podemos implementar somente os métodos necessários para manipular os conjuntos disjuntos que o algoritmo necessita.

Os passos para a implementação da função MERGE serão as seguintes: recebido dois índices por parâmetro, buscamos os conjuntos que os contêm com auxilio da função FIND e por fim fazemos o primeiro conjunto ser subconjunto do segundo, ou seja, a raiz do primeiro terá seu valor apontando para a raiz do segundo conjunto.

Já o método FIND, recebe um elemento por parâmetro, e retorna o índice do elemento raiz (o representante) do conjunto que contem o elemento passado como parâmetro.

Para testar a equivalência de dois autômatos, inicialmente supomos que eles são equivalentes, logo os seus estados iniciais são equivalentes. Por conta disto acontece a primeira chamada do método MERGE para o par $\{p_0, q_0\}$ correspondente ao par de estados inciais dos dois autômatos. E assim inicia-se as verificações até que não haja mais conjuntos a serem unidos.

Há ainda a possibilidade de otimizar este código, pois ao final da execução do algoritmo, temos que verificar todos os conjuntos resultantes da estrutura de dados Union-Find. Se durante a execução do Algoritmo 3 for haver uma união de conjuntos, basta verificarmos cada um dos representantes se um é estado de aceitação e o outro não. Então comprovando que aqueles dois autômatos não são equivalentes, não necessitando terminar todas as interações para verificar todos os conjuntos somente ao final. Então se encontrado uma união heterogênea, o algoritmo deve cessar. E acontecerá esta verificação anteriormente as linhas 3 e 11 do Algoritmo 3 nos conjuntos que pretendem ser unidos.

3.2.4 Recuperação de palavra

Para poder encontrar uma palavra que demonstre a distinção dos dois autômatos, a partir da otimização proposta na seção anterior, facilmente percebemos que, se o algoritmo for terminado por tentar unir dois conjuntos heterogêneos, então exite uma palavra que os

distingue. Basta agora recuperar esta palavra, e para isto temos que recuperar todas as uniões que foram efetuadas até o momento do término.

Uma possível solução, que foi implementada, foi que a cada união de conjuntos devemos guardar em uma nova pilha as informações a respeito desta união. Então armazenamos a tupla $\{p, q, \sigma, r, s\}$, que basicamente são as variáveis daquela interação, p e q é o par que estava sendo verificado pelo simbolo σ, que causou a união do par r e s.

O pseudo código a seguir veremos a implementação da recuperação desta palavra.

Algoritmo 4: Recuperação da palavra que os distinguem

Entrada: $pHist$
Saída: w

```
1 início
2     w ← ε
3     enquanto pHist ≠ ∅ faça
4         {p, q, σ, r, s} ←pHist.desempilha()
5         w ← σ · w
6         enquanto (pHist.topo().r ≠ p) | (pHist.topo().s ≠ q) faça
7             pHist.desempilha()
8     retorna w
```

Assim teremos uma pilha com o histórico das uniões. Então se o Algoritmo 3 terminou por ter encontrado uma união não desejada, basta agora ir retirando os elementos desta pilha histórico até esvaziar, adicionando à variável w o simbolo que causou a união. E por fim encontrar o par de estados que uniu este par sucessor, dispensando os elementos que não estamos procuramos.

Poderíamos abstrair melhor este algoritmo como se a cada verificação a partir da raiz ($\{p_0, q_0\}$) no Algoritmo 3 fosse uma busca em profundidade, se interrompida em algum ponto, então a recuperação da palavra seria um caminho de volta até a raiz, formando a palavra que distinguiu os dois autômatos.

4 O sistema web

Não podemos negar que a internet está difundida na sociedade, esta em todos os lugares, desde a computadores até televisores e celulares. Com base nesta plataforma é possível desenvolver inúmeras aplicações, pois o requisitos fundamental é estar conectado à internet. Ela propõe uma experiência de uso semelhante de um aplicativo para *desktop*, se opondo a necessidade de instaladores e dependência de um sistema operacional como os programas tradicionais são.

Um sistema na web se baseia em um servidor, que atenderá todos as requisições de serviços de um cliente via rede, e o cliente por meio de um navegador irá interagir com as informações. Tornando-se muito prático e acessível.

Pensando em unir o útil ao agradável, o aluno poder praticar o conceito aprendido em aula e obter o retorno imediato é de grande valia. Por isso utilizaremos a plataforma web para desenvolver este sistema. Que será acessado pelos usuários, que serão os alunos daquela disciplina, e o administrador, que será o professor que ministra a disciplina.

O administrador adicionará no sistema listas de questões, que os alunos devem responder e o próprio sistema atribuirá as correções das respostas submetidas pelos alunos.

Inicialmente, as classes de exercícios que o sistema deverá atender tem como entrada de resposta um autômato. Então há a necessidade de utilizar um método, ou um formulário, para receber esta resposta, que poderia ser descrito matematicamente, mas não seria nada intuitivo para o aluno.

Como um autômato finito é definido como um modelo matemático de uma máquina de estados finitos. E máquinas de estados finitos podem ser representado por um diagrama de estados. Assim este diagrama deverá conter os estados, transições de estados por simbolo do alfabeto, um marcação para se o estado é de aceitação e qual o estado é o inicial.

Necessitamos então desde método de entrada, e na seção seguinte detalharemos como solucionamos este requisito.

4.1 Interface de entrada

Após uma pequena pesquisa na internet, encontramos um formulário para construção de uma máquina de estados finitos desenvolvida por Evan Wallace (WALLACE, 2010).

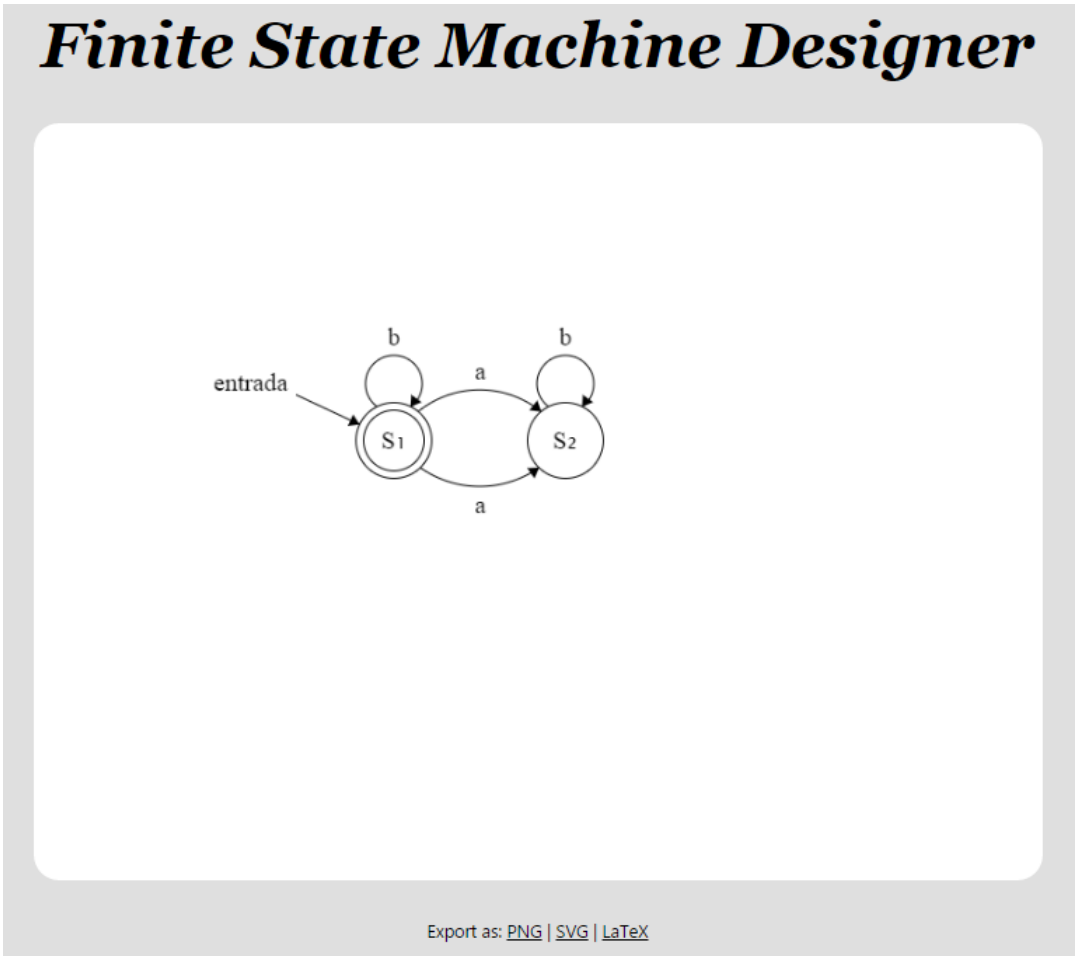


Figura 3 – Construtor de máquinas de estados.

Este construtor trata-se de um script feito com *HTML5*[1] e *JavaScript*[2] que utiliza o elemento *CANVAS*[3], possibilitando a interação do mouse e teclado para construir máquinas de estados finitos.

Exigindo somente fazer as adequações para as necessidades do sistema, pois o script tem licença MIT (MIT, 1988). Permitindo assim, sem limitação, o direito de usar, copiar, modificar, mesclar , publicar, distribuir, sub-licenciar como bem entender.

Primeiramente foi estudado o código fonte do script disponível no repositório pessoal do Evan Wallace[4], afim de entender melhor seus atributos e métodos que interagem com o *CANVAS*.

[1] HTML5 (Hypertext Markup Language, versão 5) é uma linguagem para estruturação e apresentação de conteúdo para a web.
[2] Linguagem de programação para paginas HTML e web, executada pelo navegador do cliente.
[3] Um elemento do HTML para exibições gráficas em uma página web.
[4] Disponível em: <https://github.com/evanw/fsm>.

O construtor tem os seguintes atributos e interações com o *CANVAS*:

- Criação dos estados com duplo clique;
- Criação das transições com a tecla shift e o arrastar do mouse;
- Arrastar um objeto com o mouse;
- Deletar um objeto selecionando e clicando com a tecla delete;
- Fazer um estado ser de aceitação com um duplo clique;
- Números subscrito com *underline*, do seguinte modo "S_0", para criar "S_0";
- Letras gregas, do seguinte modo "\beta" para criar "β";
- Exportar como PNG, SVG e LaTeX a máquina de estados;
- Manipulação dos objetos em memória da máquina de estados;
- Desenho dos objetos no elemento *CANVAS*;
- Salvamento automático dos objetos no *CANVAS* para o *Local Storage* do navegador;
- Recuperação automática dos objetos do *Local Storage* do navegador para o *CANVAS*;

Dentre as funções destacadas acima, somente as funções de exportar, de salvar e recuperar automaticamente não nos interessa, portanto foi desabilitado esses recursos do script original. Já que para este sistema o ideal é que a partir dos objetos que representa a máquina de estados que o script manipula, extraímos as suas descrições matemáticas e validamos se estão corretas conforme a definição de um Autômato Determinístico Finito visto na seção 2.

Assim foram feitas as adequações necessárias, para quando o usuário submeter seu autômato resposta, um novo script criado e mesclado com o script original, terá a função de extrair os dados do diagrama de estado desenhado pelo usuário no *CANVAS* e enviar para o sistema avaliar a sua resposta, que por sua vez utilizará um dos algoritmos analisados nas seções anteriores.

4.2 Breve descrição do sistema

Nesta seção veremos uma breve descrição de como o sistema será. O sistema é composto por páginas web, onde cada uma terá uma função, dependendo do tipo de usuário. Como dito anteriormente, termos dois tipos de usuários, o **aluno** e o **professor**.

A primeira página do sistema será o de **Logar**, onde o usuário poderá inserir suas credenciais (nome e senha) para ter acesso ao sistema.

Se caso for um aluno, será redirecionado para a lista de exercícios a ele atribuído, onde poderá escolher qual questão deseja responder. E ao clicar em um item desta lista, o redirecionará para a página que contém o enunciado do exercício e o logo abaixo o método de entrada para sua resposta. Ao submeter a resposta, o sistema recebe a resposta desenhada pelo aluno e o retorna para a página que lista os exercícios, para que o aluno possa responder as outras, enquanto o sistema avalia a sua resposta.

Já se caso for o professor, ele tem a permissão de administrar o sistema. Então, ele poderá gerenciar os usuários e as questões. Para adicionar questões ao banco de questões, será necessário o preenchimento de um formulário para compor o enunciado da questão e ao fim o autômato gabarito, para assim ser utilizado no algoritmo de teste de equivalência junto a uma possível resposta. E também contará com uma visualização geral do desemprenho dos alunos, afim de avalia-los.

Os próximos passos deste projeto serão a respeito da implementação desde sistema e detalhamento das ações e particularidades de cada página. E a sua codificação, utilizando *PHP*[5] com conexão ao *MySQL*[6]. Além da inserção de uma coletânea de questões no banco de dados do sistema.

Ainda mais, poderá haver a oportunidade da próxima turma da Disciplina de Linguagens Formais e Autômatas se tornarem a turma teste do sistema. Concedendo assim, grandes benefícios ao projeto, fornecendo informações sobre a qualidade do sistema e possíveis erros para a sua apropriada correção.

[5] PHP (PHP: Hypertext Preprocessor) é uma linguagem de script open source, muito utilizada e especialmente adequada para o desenvolvimento web.

[6] O MySQL é um sistema de gerenciamento de banco de dados.

5 Cronograma do projeto

O orientador está acompanhando o progresso em reuniões semanais, esclarecendo duvidas e complementando os conceitos estudados.

O projeto se constitui dos seguintes itens abaixo.

1. Leitura da bibliografia (capítulos de livros e artigos citados).
2. Implementação dos algoritmos que decidem a equivalência de dois AFDs em C/C++.
3. Implementação dos algoritmos de minimização de AFD (para armazenamento econômico dos gabaritos) em C/C++.
4. Design e construção do banco de dados de exercícios e gabaritos.
5. Adequação do script de Evan Wallace às necessidades deste projeto.
6. Implementação dos algoritmos que convertem uma expressão regular num AFN e um AFN num AFD.
7. Elaboração do relatório parcial.
8. Sistema de criação e gerenciamento de usuários.
9. Sistema de logs de submissões.
10. Finalização da versão para testes e correções de possíveis erros.
11. Elaboração do relatório final.

A continuidade do projeto será conforme a Tabela 5 abaixo (os números à esquerda indicam as tarefas enumeradas a acima). Os itens concluídos estão omitidos.

	fev	**mar**	**abr**	**mai**	**jun**	**jul**
4	x	x	x			
6	x	x				
8		x	x			
9			x	x		
10				x	x	x
11				x	x	x

Tabela 5 – Cronograma

Referências

CORMEN, T. H. et al. *Introduction to algorithms*. Second. [S.l.]: MIT Press, Cambridge, MA; McGraw-Hill Book Co., Boston, MA, 2001. xxii+1180 p. ISBN 0-262-03293-7. Citado 2 vezes nas páginas 14 e 16.

HOPCROFT, J. An n log n algorithm for minimizing states in a finite automaton. In: *Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971)*. [S.l.]: Academic Press, New York, 1971. p. 189–196. Citado na página 8.

HOPCROFT, J.; KARP, R. *A Linear Algorithm for Testing Equivalence of Finite Automata*. [S.l.], 1971. Citado 2 vezes nas páginas 14 e 16.

HOPCROFT, J. E.; KARP, R. M. A linear time algorithm for testing equivalence of finite automata. *Technical report of Cornell University*, p. 71–114, 1971. Citado na página 8.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. Automata theory, languages, and computation. *International Edition*, v. 24, 2006. Citado 2 vezes nas páginas 6 e 14.

HUFFMAN, D. A. The synthesis of sequential switching circuits. I, II. *J. Franklin Inst.*, v. 257, p. 161–190, 275–303, 1954. ISSN 0016-0032. Citado na página 8.

LINZ, P. *An introduction to formal languages and automata*. [S.l.]: Jones & Bartlett Publishers, 2011. Citado na página 6.

MIT. *The MIT Licence*. 1988. <http://opensource.org/licenses/MIT>. Citado na página 21.

MOORE, E. F. Gedanken-experiments on sequential machines. In: *Automata studies*. [S.l.]: Princeton University Press, Princeton, N. J., 1956, (Annals of mathematics studies, no. 34). p. 129–153. Citado na página 8.

NIJHOLT, A. The equivalence problem for ll- and lr-regular grammars. In: GéCSEG, F. (Ed.). *Fundamentals of Computation Theory*. Springer Berlin Heidelberg, 1981, (Lecture Notes in Computer Science, v. 117). p. 291–300. ISBN 978-3-540-10854-2. Disponível em: <http://dx.doi.org/10.1007/3-540-10854-8_32>. Citado na página 4.

SIPSER, M. *Introduction to the Theory of Computation*. [S.l.]: Cengage Learning, 2012. Citado na página 6.

WALLACE, E. *Finite State Machine Designer*. 2010. <http://madebyevan.com/fsm/>. [Online; acessado 4 de dezembro de 2015]. Citado na página 20.