Elaboração de algoritmos para criação de exercícios sobre Autômatos Finitos Determinísticos

Guilherme Augusto Anício Drummond do Nascimento[1], **Reinaldo Silva Fortes**[2], **Valéria de Carvalho Santos**[2], **Rodrigo Geraldo Ribeiro**[2]

[1] Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Ouro Preto
guilherme.drummond@aluno.ufop.edu.br
[2] Departamento de Computação, Universidade Federal de Ouro Preto
{valeriacs, reifortes, rodrigo.ribeiro}@ufop.edu.br

Abstract. *Theory of Computation is an important topic in Computer Science, and teaching it presents a significant challenge, as it involves many abstract and mathematical concepts. This complexity hinders and delays the adoption of educational activities that require a large number of questions, such as exercise lists or mock exams. In this context, automatic question generation has the potential to support student learning by focusing their attention on the presented material and reinforcing knowledge through the repetition of basic concepts across varied exercises. In this work, we present a tool integrated with Jupyter Notebook to generate exercises for the construction and minimization of deterministic finite automata and their corresponding answer, using regular expressions and a genetic algorithm to optimize the generation of questions with varying difficulty levels. Additionally, the proposed tool is capable of grading the generated exercises, presenting counterexamples that can help students understand why their submitted solution is incorrect.*

Resumo. *Teoria da Computação é um importante tópico para a Ciência da Computação e seu ensino é um grande desafio, uma vez que envolve muitos conceitos abstratos e matemáticos. Isso dificulta e atrasa a adoção de atividades educacionais que demandam muitas questões, como listas de exercícios ou simulados. Neste contexto, a geração automática de questões tem potencial para contribuir no aprendizado do estudante, ao focar a sua atenção no material apresentado e repetir conceitos básicos através de variados exercícios. Neste trabalho, apresentamos uma ferramenta, integrada ao Jupyter Nootebook, para gerar exercícios de construção de autômatos finitos determinísticos e seus respectivos gabaritos usando o conceito de expressões regulares e um algoritmo genético para otimizar a geração de questões com níveis variados de dificuldade. Adicionalmente, a ferramenta proposta é capaz de corrigir os exercícios gerados, apresentando contra-exemplos que podem ser utilizados pelo aluno para compreender o porquê da solução por ele elaborada não estar correta.*

1. Introdução

Teoria da Computação é um importante tópico para o entendimento da Ciência da Computação e faz parte dos currículos sugeridos pela Sociedade Brasileira de Computação [Zorzo et al. 2017]. Usualmente, o ensino deste assunto envolve conceitos matemáticos, como demonstrações e algoritmos [Mohammed 2020]. Tradicionalmente,

DOI: 10.5753/sbie.2025.12594

o ensino de Teoria da Computação é feito sem o auxílio de softwares, e os estudantes trabalham nos exercícios propostos usando papel e lápis.

O uso deste modelo tradicional de exercícios possui alguns problemas. Primeiro, para uma melhor compreensão do conteúdo, alunos necessitam fazer uma grande quantidade de exercícios e ter retorno sobre a correção de suas soluções, o que pode sobrecarregar o docente que não deve apenas atestar se uma resposta está incorreta mas também justificar o motivo do erro com o intuito de orientar o aluno na produção de uma solução adequada. Outro aspecto muito relevante a ser considerado é o esforço na criação de exercícios com diferentes níveis de dificuldade e em uma quantidade adequada.

Dessa forma, a *Geração Automática de Questões* (GAQ) surgiu como uma solução para estes desafios. As técnicas de GAQ estão interessadas na construção de algoritmos para produzir questões de boa qualidade a partir de fontes de conhecimento. De acordo com [Alsubait et al. 2012], pesquisas sobre GAQ remontam à década de 70 e atualmente vem ganhando importância com o surgimento de cursos online abertos e outras tecnologias de aprendizado digital [Qayyum and Zawacki-Richter 2018, Gaebel et al. 2014, Goldbach and Hamza Lup 2017].

A criação manual de exercícios que envolvem autômatos finitos determinísticos, abrangendo tanto sua construção quanto sua minimização, é um processo desafiador e intensivo em recursos. Teoria dos Autômatos, tópico tratado em Teoria da Computação, com seu caráter abstrato e matemático, frequentemente apresenta obstáculos significativos para o ensino e a aprendizagem. Um sistema de geração de exercícios dedicado a essa área, além de simplificar a elaboração de problemas práticos e teóricos, ofereceria a oportunidade de explorar uma ampla variedade de cenários de aprendizado.

Ao apresentar uma variedade de desafios, desde os fundamentos da construção de autômatos até a complexidade da minimização, a ferramenta pode incentivar os estudantes a praticar a aplicação dos conceitos teóricos, promovendo uma abordagem mais ativa e interativa. Além disso, com a integração com outro trabalho realizado por [Costa 2023], o sistema também consegue oferecer *feedback* imediato, permitindo aos estudantes aprimorarem suas habilidades e corrigirem equívocos de maneira eficiente.

Sendo assim, o principal objetivo deste trabalho é projetar e desenvolver uma ferramenta para criação automática de exercícios sobre construção e minimização de autômatos finitos determinísticos. Essa ferramenta produz uma série de exercícios e suas respectivas soluções.

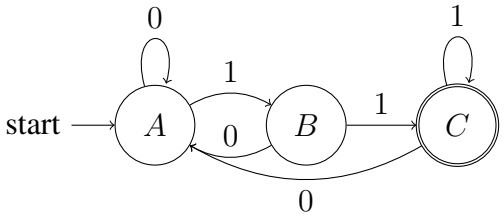
Este artigo está organizado da seguinte forma. A Seção 2 apresenta uma breve fundamentação teórica, necessária para a compreensão do trabalho. Na Seção 3 são apresentados os trabalhos relacionados a esta proposta. Uma visão geral da ferramenta é apresentada na Seção 4 e o desenvolvimento da ferramenta e problemas encontrados ao longo do caminho. Em seguida, na Seção 5 são apresentados os resultados e funcionalidades desenvolvidas. Por fim, a Seção 6 apresenta a conclusão e os trabalhos futuros.

O código-fonte da ferramenta está disponível no repositório do GitHub: *https://github.com/lives-group/automata-language*.

2. Fundamentação teórica

Intuitivamente, um autômato pode ser entendido como uma máquina abstrata para determinar se uma certa palavra pertence ou não a uma linguagem. Um Autômato Finito Determinístico (AFD) pode ser definido formalmente como uma quíntupla $M = (E, \Sigma, \delta, i, F)$, em que E é um conjunto de estados, Σ é um alfabeto, δ é uma função de transição, $i \in E$ é o estado inicial e $F \subseteq E$ é um conjunto de estados finais.

Para exemplificar esses conceitos, 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 denotam 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 pela seguinte tabela:

δ	0	1
A	A	B
B	A	C
C	A	C

Uma palavra é reconhecida por um AFD caso o processamento da palavra termine em um estado final do autômato. Diz-se que uma linguagem L é regular se existe um AFD M tal que $L(M) = L$, ou seja, se todas as palavras da linguagem são reconhecidas pelo AFD M. Dessa forma, tem-se que autômatos são uma maneira de especificar uma linguagem regular. Outro formalismo amplamente utilizado para definir linguagens regulares são as chamadas expressões regulares, que permitem uma especificação de linguagens utilizando uma notação algébrica. Por exemplo, $(0 + 1)^*11$ é uma expressão regular (ER) que define a mesma linguagem do AFD anterior. A equivalência entre expressões regulares e autômatos finitos é um resultado clássico da teoria da computação. Ao invés de utilizar a equivalência normalmente apresentada em livros clássicos [Sipser 2013], que, a partir de uma expressão regular, obtém um autômato não determinístico e, em seguida o converte em um AFD equivalente, usaremos um algoritmo baseado no conceito de derivadas, definido por [Brzozowski 1964].

Algoritmos genéticos (AG) referem-se a uma meta-heurística inspirada pelo processo de seleção natural que pertence à classe de algoritmos evolutivos. Dada uma população, indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos

aptos tendem a desaparecer. Analogamente, algoritmos genéticos, através de operações como seleção, mutação e recombinação, criam soluções cada vez melhores para um determinado problema, comumente de otimização ou busca.

3. Trabalhos relacionados

[Bezáková et al. 2020] afirmam haver um atraso entre um modelo ser introduzido e o aluno receber um retorno sobre tarefas relacionadas. Nesse tempo, o professor já avançou muito nos tópicos da disciplina e os alunos ficam progressivamente mais confusos. Somado a isso, as turmas podem ser grandes, dificultando ainda mais para que o professor tire as dúvidas de todos os alunos. [Rodger 2002] também afirma que estudantes aprendem melhor vendo representações (textuais, visuais ou animadas) de um conceito: livros didáticos podem oferecer representações textuais e algumas visuais, enquanto *softwares* podem fornecer representações visuais e animadas. No entanto, apenas observar não é o suficiente, os alunos devem ser capazes de interagir com o conceito de alguma forma e receber *feedback* para verificar sua compreensão. Dessa forma, nossa ferramenta visa fornecer *feedback* imediato (como contra-exemplos, caso a resposta esteja errada), assim como imagens dos autômatos (tanto dos enunciados, como daqueles que os próprios alunos construíram) e uma forma de verificar se uma palavra é aceita pelo autômato.

[Chakraborty et al. 2011] listam diversas ferramentas de simulação de autômatos e algumas características sobre elas, como os tipos de autômatos suportados (autômatos finitos, autômatos de pilha, máquinas de Turing), suporte para não-determinismo, suporte para sub-máquinas e linguagem de implementação. Dentre eles, o único que suporta todas essas funcionalidades é o *Java Formal Languages and Automata Package* (JFLAP) [Rodger 2002] [1], além de ser uma das mais populares para o ensino de Teoria da Computação. Junto às funcionalidades já citadas, o JFLAP também oferece produto de autômatos, conversão de autômatos finitos não-determinísticos (AFN) para autômatos finitos determinísticos, conversão de AFD para gramáticas e vice-versa, entre outras. [Paiva et al. 2023] propõem uma estratégia utilizando três ferramentas (JFLAP, LFApp e LFAweb) e aponta que o JFLAP é a ferramenta mais completa, pois cobre mais de 90% dos conteúdos geralmente apresentados nas disciplinas de Teoria da Computação. No entanto, [Cassanho et al. 2024] apontam que embora a ferramenta se revela eficaz para o ensino, tem uma interface pouco amigável, com alguns alunos julgando-a ultrapassada, não intuitiva e que a documentação disponível é insuficiente e dificulta o uso eficiente da ferramenta.

O SimStudio [Chudá and Rodina 2010] oferece a possibilidade de descrever os autômatos tanto por meio de linguagens simbólicas quanto visualmente, criando o diagrama, porém não possui todas as funcionalidades que o JFLAP oferece. E o *Language Emulator* [Vieira et al. 2003] aceita as especificações de um autômato por meio de uma interface interativa e simula o seu comportamento, mas também não oferece todas as funções que o JFLAP oferece.

Os trabalhos anteriores não suportam a geração automática de questões, ou seja, todos os exercícios devem ser criados manualmente. [Adithi et al. 2015] levantam duas questões: (1) é possível o professor transmitir, de maneira não ambígua, a descrição do problema de uma forma segura (i.e., sem revelar sua própria solução)? (2) essa ferramenta

[1] Disponível em `https://jflap.org/`.

pode operar offline (i.e., sem conexão com a internet)? A questão (1) por si só pode ser resolvida permitindo que alunos enviem suas tentativas para um servidor, onde podem ser comparadas com a solução do professor para equivalência, porém tal *feedback* não pode ser obtido por alunos sem conexão confiável à internet.

Na linha de geração automática, [Shenoy et al. 2016] apresentam uma ferramenta que recebe um problema de construção de autômato finito determinístico P como entrada e gera um número arbitrário de problemas $P_1, P_2, ...$ em ordem decrescente de similaridade a P. A saída da ferramenta pode ser restringida a problemas que são mais fáceis, mais difíceis ou tão difíceis quanto P. A partir de uma base de problemas, composta de 107 problemas "semente", novos exercícios são gerados combinando e alterando esses problemas. Porém, é mostrado que esses exercícios podem gerar linguagens que não são regulares, ou seja, não seria possível construir um AFD para resolver o exercício.

Em contrapartida, nossa ferramenta consegue gerar um conjunto com uma quantidade de questões definidas pelo próprio aluno sem necessidade de uma base prévia de problemas, pois o gabarito da questão é gerado junto ao enunciado, mas não é inicialmente apresentado ao aluno, além de ter uma garantia de que a linguagem gerada sempre é regular e será possível construir um AFD. Dessa forma, o aluno pode gerar questões e receber *feedback* de forma *offline*, além de experimentar com diferentes expressões regulares e palavras próprias, já que a ferramenta consegue produzir AFD equivalentes à qualquer ER e validar se determinadas palavras pertencem ou não à linguagem definida pela ER (e pelo AFD).

[Casamaximo et al. 2024] apresentam um mapeamento sistemático da literatura sobre jogos educativos com potencial para abordar conceitos fundamentos de linguagens formais e autômatos, que utilizam elementos de jogos dinâmicos (como narrativas) e mecânicos, entre outros, para envolver os jogadores e facilitar a compreensão dos princípios teóricos subjacentes. Porém, os jogos tendem a abordar um conjunto mais limitado de conceitos em comparação com outros *softwares* educativos, por exemplo, pela necessidade de criar uma narrativa coesa para cada jogo e integrar os conceitos de forma significativa dentro desse contexto. Nosso trabalho se diferencia pelo foco na geração de questões e gabaritos de forma automática.

4. Visão geral da ferramenta proposta

Nesta seção é apresentada uma visão geral da ferramenta proposta neste trabalho. A Figura 1 ilustra os seus principais componentes. A ferramenta visa produzir dois tipos de questões: construir um AFD a partir de expressão regular e obter o autômato mínimo equivalente a um AFD. Para produção de ambos os tipos de questões, primeiro é necessário gerar uma expressão regular aleatória. A partir da representação da ER utilizando teoria de conjuntos, é possível criar o enunciado do primeiro tipo de questão, apresentando a ER ao aluno e solicitando que construa um AFD que aceite a mesma linguagem. Utilizando o método de construção baseada em derivadas apresentado por [Brzozowski 1964], é possível gerar um AFD equivalente à ER e, então, gerar uma imagem do AFD, que serve dois propósitos: como gabarito para questões de construção de AFDs e como parte do enunciado para questões de minimização de AFDs. A geração do gabarito e correção automática para questões de minimização serão abordadas em trabalhos futuros.

Para corrigir e apresentar contra-exemplos para as submissões para questões de

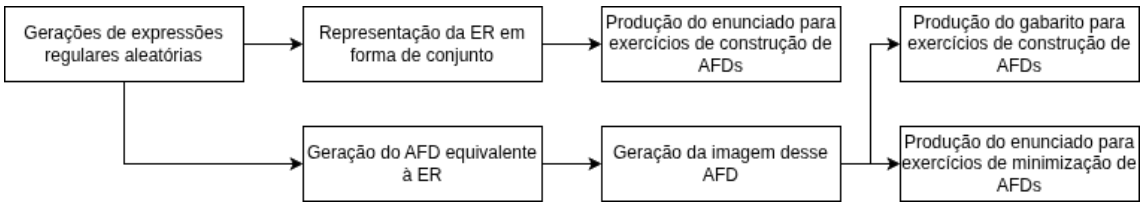


Figura 1. Visão geral da ferramenta

criação de autômatos, primeiro são criados dois autômatos: um que aceita apenas palavras incorretamente aceitas pelo autômato do aluno e outro que aceita apenas palavras incorretamente rejeitadas pelo autômato do aluno. A partir deles, usando algoritmos de caminhamento de grafo, é possível gerar e apresentar os contra-exemplos aos alunos.

Para a implementação da ferramenta proposta, foi utilizada a linguagem Racket [Flatt and PLT 2010, Version 8.17].

4.1. Operações sobre expressões regulares

Foram desenvolvidas também funções para simplificar expressões regulares, por meio da reescrita de igualdades entre as operações, como a união (ou interseção) com o conjunto vazio e a concatenação com a palavra vazia. Tais igualdades preservam a semântica da expressão obtida e evitam o acréscimo de estados desnecessários durante a criação do autômato.

Um componente importante do processo de simplificação de expressões regulares é determinar quando duas expressões regulares são ou não equivalentes. Intuitivamente, a função percorre a estrutura recursiva das expressões para determinar se estas são ou não equivalentes. Porém, simplesmente percorrer a estrutura recursiva de expressões para o teste de equivalência não é suficiente. Algumas operações sobre expressões regulares são comutativas (união e interseção) e associativas (união, interseção e concatenação) impedindo uma implementação direta do teste. Para contornar esse problema, foi adicionada uma etapa de reescrita que aninha, recursivamente, as sub expressões das operações de concatenação, união e interseção à direita e simplifica-as usando as funções citadas anteriormente.

Para união e interseção, dois passos adicionais são feitos: 1) a ordenação dos elementos de forma arbitrária (e igual para todas as ERs), para que o teste de equivalência possa ser feito percorrendo linearmente as expressões regulares e 2) a remoção de elementos duplicados, que são equivalentes. Sobre o ponto 1), a ordenação pode ser feita, pois essas operações são associativas e comutativas (a função impõe uma ordem sobre expressões regulares). A ordem estabelecida determina a seguinte prioridade: símbolo, concatenação, interseção, união, complemento, fecho de Kleene, λ e $\emptyset$. Expressões regulares do mesmo tipo são ordenadas lexicograficamente pelo primeiro símbolo que elas contêm. O ponto 2) é necessário, pois, embora as funções auxiliares mencionadas anteriormente já simplifiquem idempotência, como os elementos são aninhados a direita, elementos equivalentes podem ficar em níveis diferentes da árvore.

4.2. Geração de uma ER

Para produzir ERs foi adotada uma abordagem simples: expressões geradas aleatoriamente. Para isso, a biblioteca de testes baseados em propriedades, rackcheck[2] foi utilizada. A escolha de uso desta biblioteca é motivada pelo fato desta possuir geradores de valores aleatórios para diversos tipos primitivos da linguagem Racket e funções para combinar geradores para criar funções que produzem tipos de dados quaisquer.

4.3. Operações de *crossover* e mutação

Para explicar o funcionamento dessas operações, considere o seguinte: a estrutura de expressões regulares levam, de forma simples, a uma representação de árvore.
Exemplo 1.

A representação da expressão regular $(0+1)^*11$ *como uma árvore.*

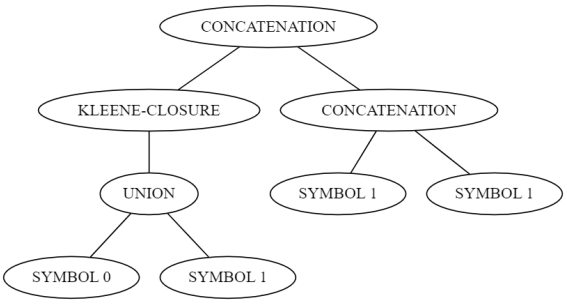


Figura 2. Árvore da expressão $(0+1)^*11$

Dessa forma, o *crossover* é feito trocando duas subárvores das árvores das expressões r e s.
Exemplo 2. *Considere as expressões* $r = (0+1)^*11$ *e* $s = 10+01$.

As Figuras 3 e 4 apresentam as árvores das expressões r *e* s*, respectivamente. Os nós preenchidos em cinza são os nós raízes das sub árvores trocadas.*

As Figuras 5 e 6 apresentam as árvores geradas pelo crossover: $r' = (10)^*11$ *e* $s' = 0+1+01$.

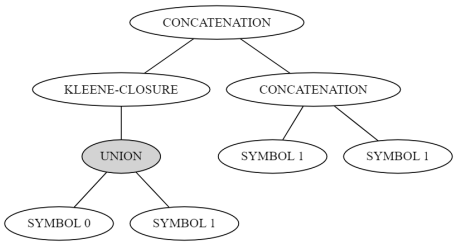


Figura 3. Árvore da expressão $r = (0+1)^*11$

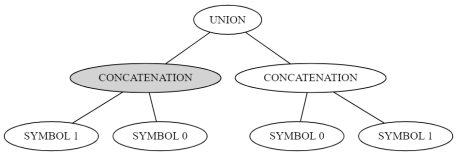


Figura 4. Árvore da expressão $s = 10+01$

E a mutação é feita trocando a operação, caso seja um nó interno da árvore, ou o símbolo, caso seja um nó folha.

[2]Disponível em `https://docs.racket-lang.org/rackcheck/index.html`.

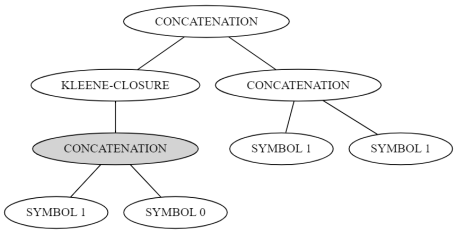


Figura 5. Árvore da expressão $r' = (10)^*11$

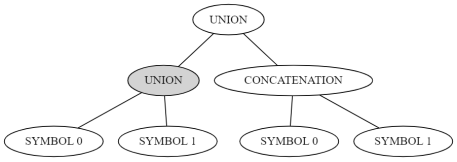


Figura 6. Árvore da expressão $s' = 0 + 1 + 01$

Exemplo 3. *Considere a expressão* $r = (0 + 1)^*11$.

A Figura 7 apresenta a árvore dessa expressão e a Figura 8 apresenta o resultado da mutação, $r' = (0 + 1)^* + 11$. *O nó preenchido em cinza foi o nó que sofreu mutação.*

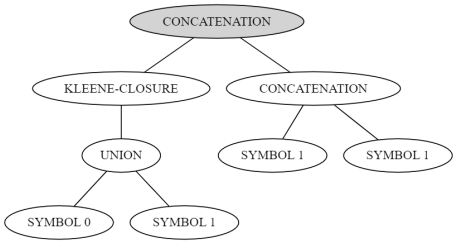


Figura 7. Árvore da expressão $r = (0 + 1)^*11$

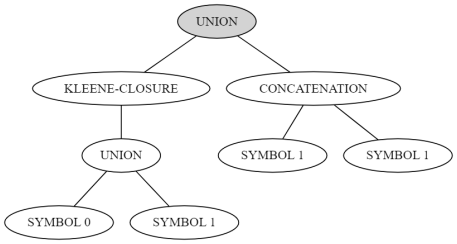


Figura 8. Árvore da expressão $r' = (0 + 1)^* + 11$

A seguir, é apresentada uma forma alternativa para realizar o *crossover*: dada duas expressões r e s e os nós A_r, B_r, A_s e B_s, de forma que B_r esteja na sub árvore cuja raiz é A_r e B_s esteja na sub árvore cuja raiz é A_s, trocando o trecho das árvores que se encontra entre A_r e B_r e entre A_s e B_s nas duas árvores, incluindo A_r, B_r, A_s e B_s.

Exemplo 4. *Considere as expressões* $r = (11 + 0)^*0$ *e* $s = (\neg(000))^*$.

As Figuras 9 e 10 apresentam as árvores das expressões r *e* s*, respectivamente. Os nós preenchidos em cinza são os nós que delimitam o trecho que será trocado.*

As Figuras 11 e 12 apresentam as árvores geradas pelo crossover: $r' = \neg(110)0$ *e* $s' = (((0 + 0)0)^*)^*$.

4.4. Criação das questões

Por terem enunciados similares, apenas as expressões são necessárias para criação das questões. Basta variar a expressão sobre a qual o exercício trata para criar novos exercícios.

Foram definidas três funções objetivo para o desenvolvimento do algoritmo genético, para gerar questões com dificuldade variada. A dificuldade da questão é definida pelo número de estados do autômato equivalente à expressão regular do enunciado da questão. Um número de estados menor ou igual a quatro é considerado um exercício fácil, de cinco a oito estados é considerado um exercício médio e de nove a doze é considerado um exercício difícil. O limite de 12 estados para questões difíceis existe para

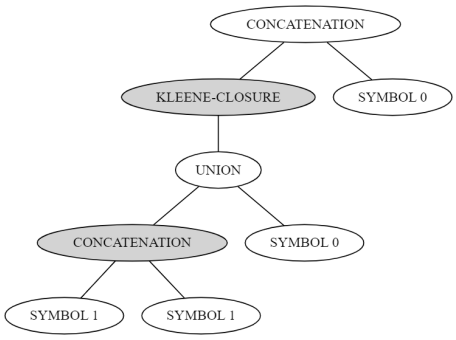


Figura 9. Árvore da expressão $r = (11 + 0)^*0$

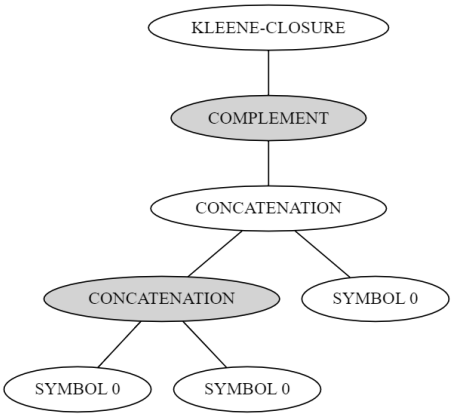


Figura 10. Árvore da expressão $s = (\neg(000))^*$

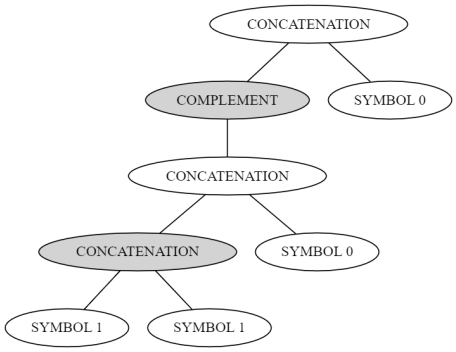


Figura 11. Árvore da expressão $r' = \neg(110)0$

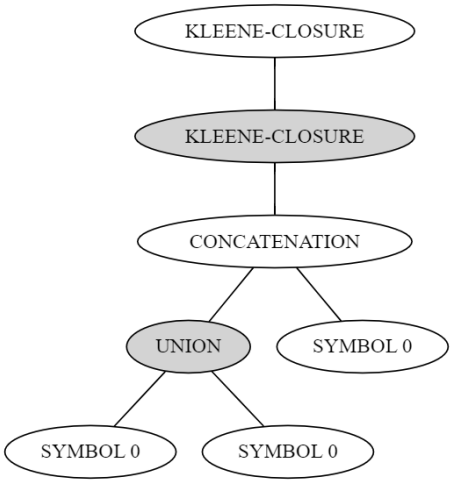


Figura 12. Árvore da expressão $s' = (((0 + 0)0)^*)^*$

não criar autômatos com estados demais, o que torna a resolução do exercício em um processo mais tedioso do que desafiador. Não foram encontradas na literatura formas de avaliar a dificuldade da criação do AFD equivalente à expressão regular e, por isso, foi utilizado esse método. Porém, usando esse método, autômatos finitos com muitos estados que apresentam a forma de uma lista seriam categorizados como difíceis, mesmo que na realidade sejam simples, por serem lineares.

5. Testes e resultados

Nesta seção são apresentados os testes realizados para avaliar a eficácia do algoritmo genético desenvolvido. O objetivo principal desses testes foi gerar um conjunto diversificado de questões, com a proporção de questões fáceis, média e difíceis escolhidas pelo usuário. Por exemplo, em uma lista de 10 exercícios, o aluno pode querer 3 questões fáceis, 3 médias e 4 difíceis. Para os testes e resultados apresentados, todas as questões geradas são de nível fácil, com o intuito de facilitar a avaliação e comparação dos resul-

tados. Esta escolha não afeta nos resultados de outras dificuldades, pois uma falha na geração de questões de um nível seria reproduzida em todos os níveis, mudando apenas a função objetivo do AG, que avalia o total de nós do autômato equivalente à ER.

Os indivíduos da população inicial são as diferentes expressões regulares geradas aleatoriamente. O *crossover* consiste em, dada duas expressões r e s, trocar subexpressões delas, gerando duas novas expressões r' e s', enquanto a mutação consiste em trocar uma operação ou um símbolo da expressão.

Com o intuito de variar a geração das questões, o algoritmo genético é executado uma vez para cada questão da lista a ser gerada, variando a dificuldade das questões geradas para manter a proporção escolhida pelo usuário, e o melhor indivíduo da população final é escolhido para entrar na lista.

A escolha do nó onde as operações de *crossover* e mutação acontecem é aleatória e os resultados iniciais podem ser observados na Tabela 1. Inicialmente, devido a utilização da função de reescrita de ER, que aninhava as expressões para a direita, um número n entre 1 e a altura da folha mais a direita é gerado e o n-ésimo nó no caminho formado pelos galhos mais à direita na árvore é escolhido para ser o ponto.

Tabela 1. Exemplos de expressões de nível fácil geradas com simplificação das expressões e apenas um ponto de *crossover*.

Identificador	Expressão
1	λ
2	$(1)^*$
3	λ
4	$(0 + (11 \,\&\, (1)^*) + \lambda)$
5	$(1)^*$

Tabela 2. Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 1.

	Média	Desvio padrão
Número de estados	3	1
Iterações até convergência	4	2

A Tabela 1 apresenta o resultado de uma execução do algoritmo genético para gerar questões de nível fácil. É possível ver que embora o algoritmo tenha gerado 10 ERs, existem apenas 3 expressões distintas. Para tentar aumentar a variedade nas expressões geradas, o *crossover* foi modificado para ser feito em 2 pontos das expressões.

A Tabela 2 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis. Ela apresenta a média aritmética e desvio padrão do número de estados dos AFDs equivalentes às expressões geradas. A geração final é a quantidade de iterações executadas pelo algoritmo genético até sua convergência. A média de estados mostra que as expressões geradas são satisfatórias, atendendo o requisito para questões fáceis. Além disso, o número pequeno de iterações executadas mostra que o problema é relativamente simples de resolver, obtendo soluções satisfatórias para o problema. Porém, como dito antes, as expressões geradas apresentam muitas repetições.

A Tabela 3 mostra algumas expressões de dificuldade fácil geradas por esse método. Ela mostra que, embora a variedade de expressões geradas tenha aumentado, as expressões ainda se repetem.

A Tabela 4 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis, usando o método descrito anteriormente. A média de estados para questões fáceis ficou acima do limite esperado para questões fáceis, gerando mais questões de dificuldade média. Além disso, o número de iterações executadas mostra que dessa forma, o algoritmo precisou executar bem mais iterações e, em alguns casos, alcançou o número máximo de iterações antes de convergir.

Tabela 3. Exemplos de expressões de nível fácil geradas com simplificação das expressões e dois pontos de *crossover*.

Identificador	Expressão
1	$((0 + \lambda))^*$
2	$(0 + \lambda)$
3	$((0 + ((0 + 00))^*) \,\&\, (1)^*)$
4	$\neg((0 + 1))((0 + 1))^*$
5	λ

Tabela 4. Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 3.

	Média	Desvio padrão
Número de estados	3	6
Iterações até convergência	6	3

Durante os testes, foi observado que a reescrita estava diminuindo o tamanho das expressões, devido às simplificações realizadas. Também foi observado que a escolha de caminhar pelos galhos mais a direita fez com que o início das expressões (o lado esquerdo da árvore) não mudasse ao longo das gerações do algoritmo genético. Esses dois fatores fizeram com que o algoritmo genético convergisse muito rápido e gerasse muitas expressões idênticas ao final da execução.

Para tentar reduzir esse efeito, as expressões não são mais reescritas durante a execução do algoritmo genético e o método para escolher o ponto para *crossover* e mutação foi alterado para trocar trechos das árvores que se encontram entre 2 pontos escolhidos de forma aleatória. Isso gerou mais diversidade nas expressões geradas na população final do algoritmo genético, mas frequentemente essas expressões podiam ser simplificadas para outras expressões que acabavam repetindo, i.e., mesmo as expressões sendo diferentes, ainda eram equivalentes. Para tentar aumentar o número de expressões não equivalentes, foram feitos diversos ajustes na frequência das operações e nos símbolos que podem ser gerados durante a criação da população inicial.

A Tabela 5 apresenta os resultados dessas mudanças. A Tabela 6 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis. A média de estados mostra que as expressões geradas são satisfatórias, atendendo

o requisito para questões fáceis. E o número pequeno de iterações executadas mostra que com esse método o algoritmo voltou a convergir mais rapidamente.

Tabela 5. Exemplos de expressões de nível fácil geradas sem simplificação das expressões e *crossover* por caminho.

Identificador	Expressão
1	$(0 + ((0 + \lambda) \,\&\, (1 + \lambda)))$
2	$(0 + 1)$
3	$\neg(\lambda)(1)^*$
4	$(0 + ((0 + \lambda) \,\&\, (1 + \lambda)))\neg(0)$
5	$(0 + ((0 + \lambda) \,\&\, (1)^*))\neg(0)$

Tabela 6. Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 5.

	Média	Desvio padrão
Número de estados	3	0,8
Iterações até convergência	4	2

6. Conclusões

Durante o desenvolvimento deste trabalho, foi explorada a criação de uma ferramenta dinâmica para auxiliar na compreensão de AFDs. A implementação bem-sucedida de um sistema capaz de gerar expressões regulares aleatórias, com níveis variados de dificuldade, e criar autômatos equivalentes promove uma abordagem prática e interativa para o ensino-aprendizagem de conceitos abstratos e desafiadores. Além disso, a visualização das estruturas por meio de imagens reforça a compreensão visual e abre caminho para treinamento da minimização dos autômatos.

Com a integração ao IRacket[3], um *kernel* de Racket para Jupyter, foi gerado um Jupyter Notebook contendo exercícios de dificuldades variadas, com o total de exercícios de cada dificuldade escolhidos pelo próprio aluno, o que pode aumentar ainda mais a utilidade e impacto da ferramenta, possibilitando uma adaptação mais personalizada às necessidades de cada estudante e ampliando as possibilidades de interação com o ambiente de aprendizado. Por fim, também é possível integrar novos tipos de questões e seus gabaritos à ferramenta, como minimização de AFDs e converter AFN para AFD, entre outros.

Como trabalhos futuros, pretendemos avaliar o impacto do uso da ferramenta em cursos de Teoria da Computação e continuar seu desenvolvimento, avaliando outros métodos para geração de questões com diferentes níveis de dificuldade e desenvolver a geração do gabarito e correção automática para questões de minimização.

O código-fonte da ferramenta está disponível no repositório do GitHub: *https://github.com/lives-group/automata-language*.

[3]Disponível em `https://docs.racket-lang.org/iracket/index.html`.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. À Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), Código do Financiamento APQ-03665-22.

Referências

Adithi, G., Adiga, A., Pavithra, K., Vasisht, P. P., and Kumar, V. (2015). Secure, offline feedback to convey instructor intent. In *2015 IEEE Seventh International Conference on Technology for Education (T4E)*, pages 105–108, Warangal, India. IEEE.

Alsubait, T., Parsia, B., and Sattler, U. (2012). Automatic generation of analogy questions for student assessment: an ontology-based approach. *Research in Learning Technology*, 20.

Bezáková, I., Hemaspaandra, E., Lieberman, A., Miller, H., and Narváez, D. E. (2020). Prototype of an automated feedback tool for intro cs theory. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1311, New York, NY, USA. Association for Computing Machinery.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494.

Casamaximo, R., Silva, P., Michels, J. F., and Barbosa, C. R. (2024). Avaliação de jogos digitais no ensino de linguagens formais e autômatos. In *Anais do XXXV Simpósio Brasileiro de Informática na Educação*, pages 538–550, Porto Alegre, RS, Brasil. SBC.

Cassanho, L., Michels, J. F., and Barbosa, C. R. (2024). Avaliação do jflap para ensino de autômatos. In *Anais do XXXV Simpósio Brasileiro de Informática na Educação*, pages 513–524, Porto Alegre, RS, Brasil. SBC.

Chakraborty, P., Saxena, P. C., and Katti, C. P. (2011). Fifty years of automata simulation: a review. *acm inroads*, 2(4):59–70.

Chudá, D. and Rodina, D. (2010). Automata simulator. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10, page 394–399, New York, NY, USA. Association for Computing Machinery.

Costa, C. S. (2023). Desenvolvimento de algoritmos para correção automática de exercécios sobre autômatos finitos determinísticos.

Flatt, M. and PLT (2010). Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc. `https://racket-lang.org/tr1/`.

Gaebel, M., Kupriyanova, V., Morais, R., and Colucci, E. (2014). E-learning in european higher education institutions: Results of a mapping survey conducted in october-december 2013. *European University Association*.

Goldbach, I. and Hamza Lup, F. (2017). Survey on e-learning implementation in eastern-europe spotlight on romania. In *The Ninth International Conference on Mobile, Hybrid, and On-line Learning, eLmL*, number 2, pages 05–13, Nice, França.

Mohammed, M. K. O. (2020). Teaching formal languages through visualizations, simulators, auto-graded exercises, and programmed instruction. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 1429, New York, NY, USA. Association for Computing Machinery.

Paiva, P., Souza, M., and Terra, R. (2023). Ferramentas de apoio para a disciplina de linguagens formais e autômatos: uma proposta de uso. In *Anais do XXXIV Simpósio Brasileiro de Informática na Educação*, pages 1698–1709, Porto Alegre, RS, Brasil. SBC.

Qayyum, A. and Zawacki-Richter, O. (2018). *Open and distance education in Australia, Europe and the Americas: National perspectives in a digital age*. Springer Nature, Singapore.

Rodger, S. H. (2002). Using hands-on visualizations to teach computer science from beginning courses to advanced courses. In *Second Program Visualization Workshop*, number 14, pages 103–112, Hornstrup Center, Dinamarca.

Shenoy, V., Aparanji, U., Sripradha, K., and Kumar, V. (2016). Generating dfa construction problems automatically. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 32–37, Mumbai, India. IEEE.

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

Vieira, L. F. M., Vieira, M. A. M., and José, N. (2003). Language emulator, uma ferramenta de auxílio no ensino de teoria da computação.

Zorzo, A. F., Nunes, D., Matos, E., Steinmacher, I., de Araujo, R. M., Correia, R., and Martins, S. (2017). Referenciais de formação para os cursos de graduação em computação.