CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS

Recredenciado pela Portaria Ministerial nº 1.162, de 13/10/16, D.O.U. nº 198, de 14/10/2016
AELBRA EDUCAÇÃO SUPERIOR - GRADUAÇÃO E PÓS-GRADUAÇÃO S.A.

Pedro Sousa e Silva Cantuária

GRAPHLIVE: DESENVOLVIMENTO DE UM SISTEMA INTERATIVO PARA CRIAÇÃO DE GRAFOS

Palmas – TO

2020/2

Pedro Sousa e Silva Cantuária

GRAPHLIVE: DESENVOLVIMENTO DE UM SISTEMA INTERATIVO PARA CRIAÇÃO DE GRAFOS

Trabalho de Conclusão de Curso (TCC) II elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M.e Fabiano Fagundes.

Palmas – TO

2020/2

Pedro Sousa e Silva Cantuária

GRAPHLIVE: DESENVOLVIMENTO DE UM SISTEMA INTERATIVO PARA CRIAÇÃO DE GRAFOS

Trabalho de Conclusão de Curso (TCC) II elaborado e apresentado como requisito parcial para obtenção do título de bacharel em Ciência da Computação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. M.e Fabiano Fagundes.

Aprovado em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. M.e Fabiano Fagundes

Orientador

Centro Universitário Luterano de Palmas – CEULP/ULBRA

_____

Prof. M.e Jackson Gomes de Sousa

Centro Universitário Luterano de Palmas – CEULP/ULBRA

_____

Prof. D.ra Parcilene Fernandes de Brito

Centro Universitário Luterano de Palmas – CEULP/ULBRA

Palmas – TO

2020/2

RESUMO

CANTUARIA, Pedro Sousa e Silva, **GraphLive: Desenvolvimento de um Sistema Interatiovo para Criação de Grafos.** 2020. 38 f. Trabalho de Conclusão de Curso (Graduação) – Curso de Ciência da Computação, Centro Universitário Luterano de Palmas, Palmas/TO, 2020.

Grafos são estruturas de dados de extrema importância para a resolução de problema, pois são capazes de representar situações do mundo real de forma computacional. No entanto, nos cursos de computação, este tema de conteúdo extenso divide espaço com outros temas de mesma importância nas disciplinas de Estrutura de Dados, o que pode prejudicar o aprendizado. Portanto, o presente trabalho tem como proposta o desenvolvimento e implementação de um sistema web interativo para visualização dos conceitos de grafos funcionando de forma independente ou como um módulo de um Ambiente de Auxílio ao Ensino de Estrutura de Dados, que está sendo desenvolvido em paralelo a este trabalho. O resultado esperado neste trabalho é um sistema no qual o usuário poderá inserir comandos em forma de código em uma linguagem simplificada e em resposta visualizará as informações e características do grafo. A metodologia aplicada neste projeto utilizará ferramentas de desenvolvimento web para visualização dos grafos, suas informações e características, e um sistema auxiliar para a geração dos grafos. A entrada do usuário para criação dos grafos será via código, em uma linguagem simples compilada como parte deste projeto, visando aproveitar o contexto educacional no qual este trabalho está inserido e instigar a adaptabilidade do estudante ao uso de diferentes linguagens.

Palavras-chaves: Teoria dos Grafos. Visualização Interativa. Linguagem de Programação.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CEULP – Centro Universitário Luterano de Palmas

HTML – *HyperText Markup Language*

ANTLR4 – *Another Tool for Language Recognition 4*

SUMÁRIO

1 INTRODUÇÃO

Na computação, Estruturas de Dados são formas de armazenar e organizar dados de modo que possam ser acessados, analisados ou visualizados de forma eficiente. Por sua importância fazem parte do eixo básico das matrizes curriculares dos cursos da área.

As Diretrizes Curriculares Nacionais para os cursos de graduação em Computação, definido pelo Conselho Nacional de Educação homologado em oito de março de 2012, estabelece: “Os conteúdos tecnológicos e básicos comuns a todos os cursos (...) estruturas de dados e teoria dos grafos” (CNE/CES nº 136/2012, 2012, p.14). Em geral, o ensino da disciplina acontece de duas formas: inicialmente teórica, com apresentação dos diversos conceitos e visualização de imagens estáticas das estruturas, e em seguida, atividades práticas, com a implementação e validação dos resultados obtidos.

Estruturas de dados lidam com tipos abstratos de dados, acompanhando diversos conceitos, dos mais simples, como listas, pilhas e filas, até os mais complexos, como grafos, que possuem seu próprio ramo de estudo chamado Teoria dos Grafos. O estudo de grafos se destaca pela quantidade de conceitos e conteúdos existentes, como: grafos simples, completo, subgrafo, multigrafo, arvores, conjunto, e etc. Um conteúdo tão extenso dificilmente é apresentado de forma plena em um curso de graduação, limitando o ensino aos principais conceitos e utilizações.

Apesar de seu uso computacional, a Teoria dos Grafos é um ramo complexo da matemática que se utiliza de várias regras e fórmulas em seu conteúdo didático. De acordo com Madeira (2012, p. 1), a complexidade pode dificultar o entendimento e abstração necessários para o aprendizado por meio dos recursos habituais de ensino utilizados.

A oportunidade de testar na prática os conhecimentos teóricos adquiridos é essencial para um verdadeiro aprendizado do tema. Com um sistema onde a própria implementação das animações gráficas é facilitada a ponto de poder ser realizada pelo estudante, a absorção do funcionamento dos algoritmos será ainda mais intensa (GARCIA et al., 1997).

Existem diversas ferramentas para desenho de grafos disponíveis, tais como Graph Online (2015), Visualgo (2011), Odin (2012), mas cada qual com limitações em algum aspecto. Algumas em idiomas estrangeiros, outras de utilização complexa, mas todas com a mesma característica: apresentam apenas imagens do grafo, sem informações inerentes ao mesmo.

No ambiente o universitário, alunos dos cursos de computação são, desde o inicio, apresentados ao uso de linguagens de programação, assim este trabalho propõe a utilização de uma linguagem de programação simplificada para geração dos grafos e apresentação de suas características.

O sistema desenvolvido neste trabalho, o GraphLive, busca permitir ao usuário a oportunidade de testar e validar os conhecimentos teóricos sobre grafos. Os objetivos específicos para sua elaboração foram criar uma linguagem para descrição de grafos; analisar os grafos com base na entrada do usuário; e apresentar imagem do grafo e suas características.

Para o desenvolvimento deste projeto, foram utilizados ferramentas e conceitos de reconhecimento de linguagem natural e desenvolvimento de um compilador próprio com o ANTLR4. Além disso foram utilizados o GraphViz para geração das imagens dos grafos em conjunto com sua linguagem de descrição de grafos, DOT, obtida a partir da tradução da entrada do usuário.

Neste contexto, espera-se que o aluno possa ter contato com uma linguagem com atributos léxicos e sintaxe diferente do habitual sem perder o foco no aprendizado Teoria dos Grafos. Assim, o GraphLive poderá servir como um recurso valioso para professores e alunos para o aprendizado da disciplina.

2 REFERENCIAL TEÓRICO

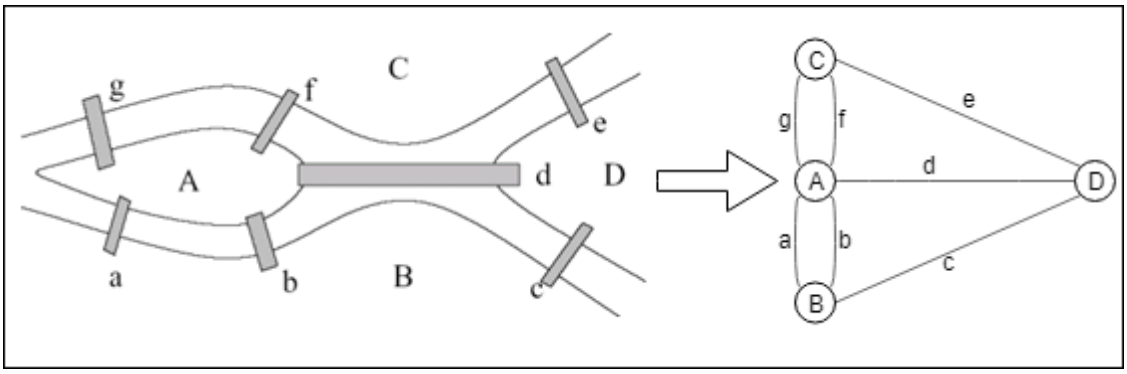
Esta seção apresenta os conceitos básicos de Teoria dos Grafos, utilizados no desenvolvimento e execução deste projeto. Dispondo de uma apresentação formal do conceito de grafos, suas principais características e tipos, além de apresentar algumas ferramentas de trabalhos relacionados.

2.1. GRAFOS

A palavra Grafo é um estrangeirismo do inglês *graph* (gráfico), por conta de sua representação gráfica que ajuda a entender muitas de suas propriedades. Grafos podem ser classificados por diversos tipos a partir de suas características, numero de vértices e arestas, interconectividade e muito mais.

Teoria dos Grafos é um ramo da matemática que tem por objetivo estudar objetos combinatórios. Para Bondy et al. (1976), muitas situações do mundo real podem ser convenientemente descritas por um conjunto de pontos e linhas conectando certos pares destes pontos, formando um grafo.

Figura 1: Representação do problema das pontes de Königsberg



Fonte: adaptado de Bestavros (2012)

A figura 1 demonstra a utilização mais antiga registrada sobre a grafos, tida como a origem da Toria dos Grafos, desenvolvida por Leonhard Euler em 1736. Ainda de acordo com Bondy (1976), Euler provou que seria impossível atravessar as sete pontes da cidade de Köngsberg (na epoca território da Prússia, atual Kaliningrado na Rússia) percorrendo um caminho que não cruzavam mesmas pontes mais de uma vez.

Estudando o grafo referente ao problema, Euler chegou ao resultado de que o percurso proposto só seria possível se não houver vértices de graus ímpares em um grafo conexo, ou seja, para percorrer todos as arestas de um grafo apenas uma vez, é necessário um grafo conexto em que todos os seus vértices estejam relacionados pelo menos por duas arestas.

A teoria está relacionada a outros diversos ramos da matemática como a teoria de grupos, teoria de matrizes, análise numérica, probabilidade, topologia e análise combinatória. O fato é que a teoria dos grafos serve como um modelo matemático para qualquer sistema envolvendo uma relação binaria (RABUSKE, 1992, p.2).

Para Dovicchi (2007, p. 77), a terminologia utilizada na teoria dos grafos compreende as definições dos elementos, características e propriedades dos grafos, que são compostas por:

- Vértices (ou Nós): é a unidade fundamental das quais grafos são formados;
- Arestas: seguimento de linha que encontra dois vértices;
- Laços: são arestas que conectam apenas um vértice;
- Vértices Adjacentes: são dois vértices ligados por uma aresta;
- Arestas Paralelas: duas arestas que se conectam aos mesmos vértices;
- Vértice Isolado: um vértice que não se conecta a nenhum outro vértice;
- Grau (ou Valência): numero de arestas que se conectam a um dado vértice;
- Caminho**:** sequência de vértices conectados por arestas que conectam um vértice inicial a um vértice destino;

A disposição, ou ausência, destes elementos em um grafo caracterizam diferentes categorias ou tipos de grafos.

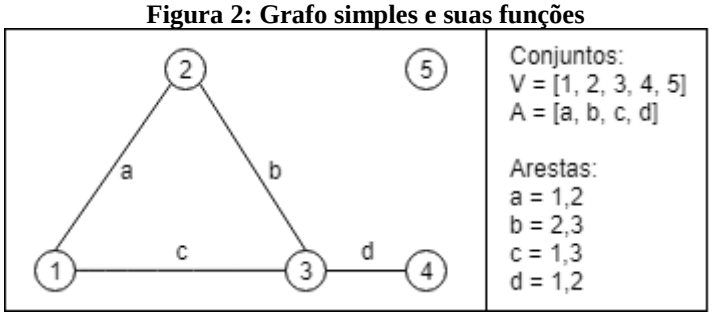
2.2. TIPOS DE GRAFOS

Existem diversos tipos de grafos que podem ser aplicados nas mais variadas situações, portanto um bom conhecimento dos tipos existentes é essencial para encontrar a melhor solução para problemas que podem ser resolvidos com esse tipo de estrutura de dados. A seguir serão apresentados os tipos mais comuns de grafos.

2.2.1. GRAFO SIMPLES

Segundo Costa (2011), um grafo é G é formado por um par (V(G), A(G)), onde V(G) é um conjunto não vazio e A(G) um conjunto de pares distintos não ordenados de elementos distintos de V(G). Abaixo está apresentado um modelo de grafo simples.

Figura 2: Grafo simples e suas funções

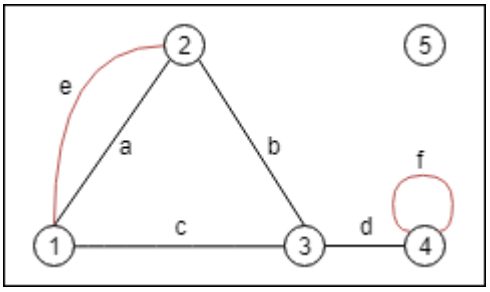


Na Imagem 2 pode-se identificar um grafo simples, composto por cinco vértices V(1, 2, 3, 4, 5) e quatro arestas A(a, b, c, d), onde cada uma das arestas é determinada por um par de vértices. Podem-se identificar algumas características neste grafo, como por exemplo: o grau dos vértices e a presença de um vértice isolado V(5), outras características serão apresentadas mais adiante.

2.2.2. MULTIGRAFO

São grafos que além das características de um grafo simples, podem possuir arestas paralelas, ou seja, arestas com os mesmos vértices finais, além da utilização de laços, conforme figura a seguir.

Figura 3: Um multigrafo

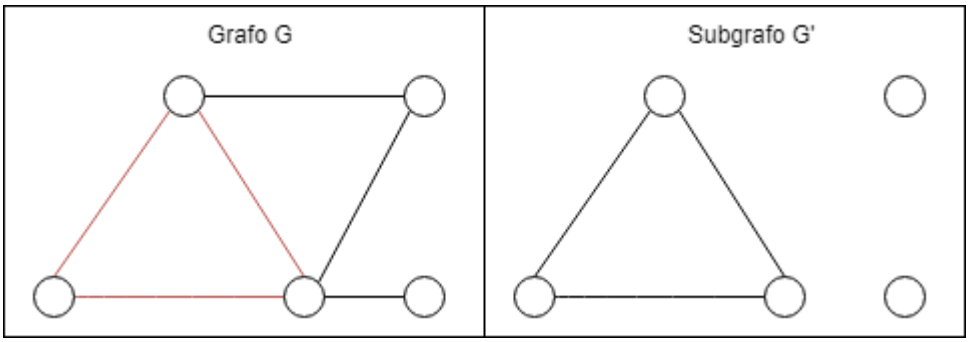


A Figura 3 apresenta um grafo com os mesmos vértices apresentado no grafo da Figura 2, mas com a adição de uma aresta paralela A(e) e de um laço A(f). De acordo com Diestel (2000, p 25) um grafo é essencialmente o mesmo que um multigrafo, mas sem laços ou arestas paralelas, eles podem supreendentemente ser mais comuns que grafos simples para a solução de problemas no mundo real.

2.2.3. SUBGRAFO

Um subgrafo de um grafo G é qualquer grafo G', tal que tanto os vértices quanto as arestas de G' façam parte de G. Um caminho $V_1...V_n$ num grafo G pode ser considerado um subgrafo de G (FEOFILOFF et al. 2004).

Figura 4: Um grafo e seu subgrafo

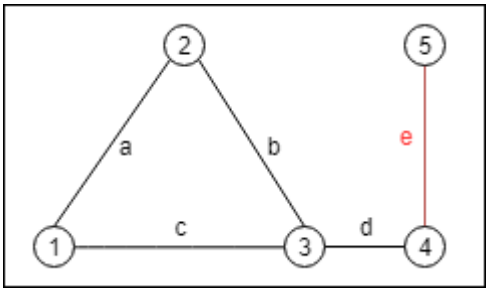


Em contraponto, o grafo G do qual o subgrafo G' faz parte pode ser considerado um supergrafo, ou seja, G' é subgrafo de G enquanto G é o supergrafo de G'.

2.2.4. GRAFO CONEXO

Um grafo conexo é um grafo no qual todos os vértices estão conectados por um caminho, por meio de uma ou mais arestas. A figura a seguir apresenta o grafo da figura 1 em um grafo conexo.

Figura 5: Grafo simples da Figura 1 transformado em grafo conexo

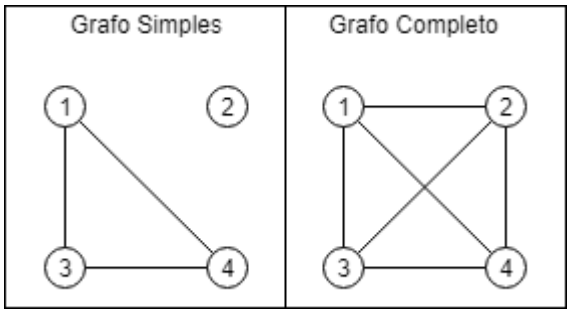


O grafo da Figura 5 apresenta a alteração necessária para a transformação do grafo da Figura 1, na qual o vértice V(5) não estava conectado a nenhum outro vértice, em um grafo conexo ao acrescentar a aresta A(e). Agora todos os vértices do grafo estão conectados a, pelo menos, um segundo vértice.

2.2.5. GRAFO COMPLETO

Um grafo completo é um grafo simples em que todos os vértices são adjacentes aos outros vértices, ou seja, existe pelo menos uma aresta ligando quaisquer dois vértices do grafo.

Figura 6: Grafo simples e um grafo completo

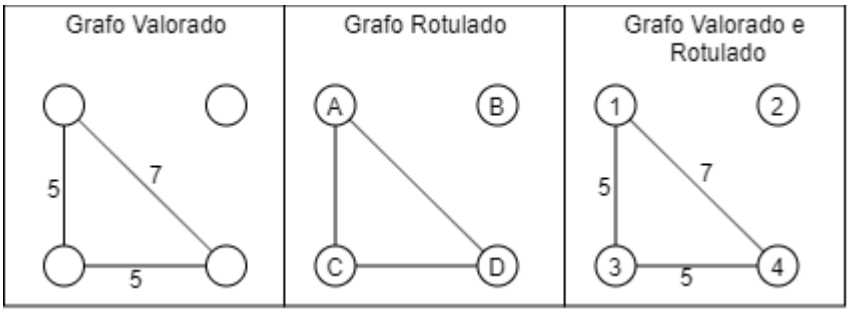


A quantidade de arestas de um grafo completo é determinada pela fórmula $\frac{n(n-1)}{2}$, onde n corresponde ao número de vértices do grafo. Assim, cada vértice estará conectado ao menos a n-1 arestas, para compreender qualquer par de vértices possível.

2.2.6. GRAFOS VALORADOS E ROTULADOS

Um grafo valorado é aquele que tem valores atribuídos à suas arestas enquanto um grafo rotulado possui um rótulo de identificação atribuído aos seus vértices. É comum um grafo valorado também ser rotulado, gerando um relacionamento entre estas características.

Figura 7: Um grafo valorado, um rotulado e um misto

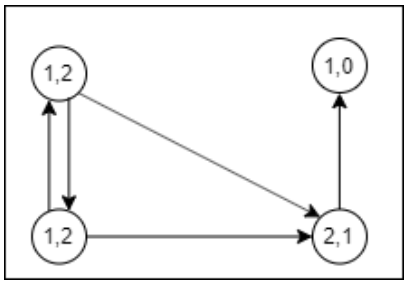


Estes tipos de grafos são uteis para a geração da Matriz de Adjacência. Os significados dos rótulos e dos valores são determinados pelo contexto ao qual o grafo está inserido, servindo como abstração de problemas do mundo real.

2.2.7. GRAFOS DIRECIONADOS OU DÍGRAFOS

Grafos direcionados ou dígrafos indicam em suas arestas a representação de seu sentido ou a direção a qual se conectam. Este tipo de grafo pode ter duas arestas ligando o mesmo par de vértices, mas em direções opostas, diferente de um grafo não-direcionado.

Figura 8: Um grafo direcionado com os graus de seus vértices



A Figura 8 apresenta um grafo direcionado com seus vértices indicando seus respectivos graus. O primeiro número indica o grau de saída do vértice, enquanto o segundo indica o grau de saída.

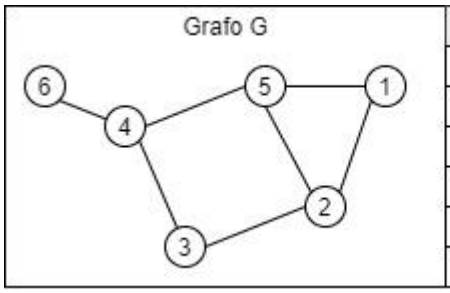
2.3. REPRESENTAÇÃO DOS GRAFOS

Embora a representação gráfica dos grafos seja muito útil na prática para modelar problemas do mundo real, facilitando sua leitura, ela não é adequada para representar computacionalmente sua estrutura. As formas mais comuns de se representar estruturas como grafos computacionalmente são através de Listas de Adjacência, Matriz de Adjacência e Matriz de Incidência.

2.3.1. LISTA DE ADJACÊNCIA

Para Dovicchi (p. 91) a representação de um grafo com poucas arestas é feita mais eficientemente por uma lista de Adjacências. Para cada vértice do grafo, a lista de adjacências apresenta uma coleção de todos seus vértices adjacentes.

Figura 9: Grafo simples e sua tabela de adjacência



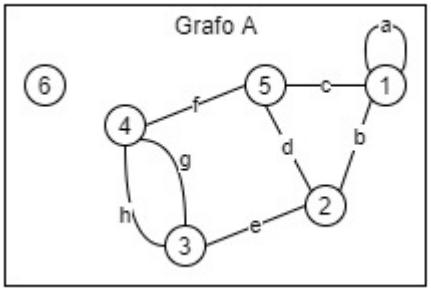
Vértice	Adjacência
1	2,5
2	1,3,5
3	2,4
4	3,5,6
5	1,2,4
6	4

A Figura 9 demostra um grafo acompanhado de sua lista de adjacência. A primeira coluna da lista representa os vértices, enquanto a segunda representa quais vértices são adjacentes a aquele vértice.

2.3.2. MATRIZ DE ADJACÊNCIA

Matrizes são entidades matemáticas usadas para representar dados em duas dimensões de ordenação (DOVICCHI, 2007). Uma matriz de adjacência representa o número de arestas que ligam dois vértices (BONDY, 1976). Também pode ser representar o valor da relação entre os mesmos para fins de apresentação e é recomendada para grafos pequenos, pois quanto maior de vértices, maior o tamanho da matriz.

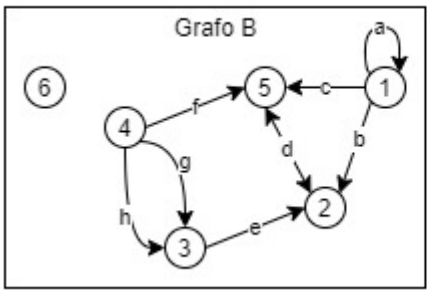
Figura 10: Um grafo e sua tabela de adjacência



ADJACÊNCIA						
Grafo A	1	2	3	4	5	6
1	2	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	2	0	0
4	0	0	2	0	1	0
5	1	1	0	1	0	0
6	0	0	0	0	0	0

A figura 10 apresenta um multigrafo não direcionado e rotulado, acompanhado de sua matriz de adjacência. As relações entre as arestas são descritas de forma numérica, em casos de laços, a aresta 1 se auto-relaciona duas vezes, uma para cada ponta da aresta. Já para arestas paralelas, o relacionamento entre dois vértices é igual ao numero de arestas paralelas, como o relacionamento entre os vértices 3 e 4, apresentados na figura. Para grafos não direcionados, matrizes de adjacência são simétricas ao longo de seu eixo diagonal principal, conforme explicitado em vermelho na figura.

Figura 11: Um digrafo e sua tabela de adjacência



ADJACÊNCIA						
Grafo B	1	2	3	4	5	6
1	1	1	0	0	1	0
2	0	0	0	0	1	0
3	0	1	0	0	0	0
4	0	0	2	0	1	0
5	0	1	0	0	0	0
6	0	0	0	0	0	0

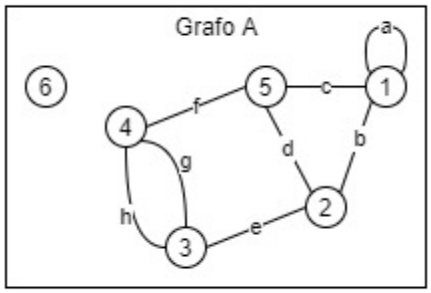
No caso de digrafos, a tabela de ajdacência relaciona quais vértices se direcionam a outro, de modo que, no caso apresentado na figura 11, vértices do eixo Y se direcionam aos vértices do eixo X, e são identificado pelo numero de relacionamentos. O mesmo não se aplica para vértices que apenas recebem o direcionamento de outros vértices, pois não formam um caminho válido para o relacionamento.

Diferente da tabela de adjacência de um grafo não direcionado, para dígrafos não há simetria entre o eixo principal diagonal, a não ser em caso onde todos as arestas sejam bidirecionais.

2.3.3. MATRIZ DE INCIDÊNCIA

Uma matriz de incidência é semelhante a uma matriz de adjacência, mas só podem ser aplicadas a grafos com vértices e arestas rotuladas, relacionando-os entre si, de forma que um eixo represente os vértices enquanto outro representa as arestas.

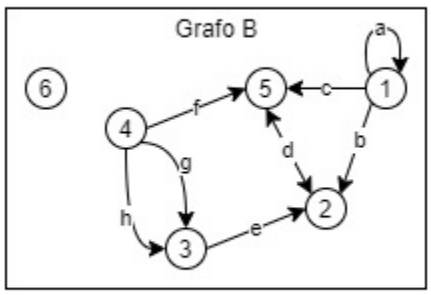
Figura 12: Um grafo e sua Matriz de Incidência



INCIDÊNCIA						
Grafo A	1	2	3	4	5	6
a	2	0	0	0	0	0
b	1	1	0	0	0	0
c	1	0	0	0	1	0
d	0	1	0	0	1	0
e	0	1	1	0	0	0
f	0	0	0	1	1	0
g	0	0	1	1	0	0
h	0	0	1	1	0	0

Para grafos não direcionados, a matriz de incidência representa sempre a quantidade de vezes que uma aresta se conecta a um vértice. Para laços, uma mesma aresta se conecta ao mesmo vértice duas vezes, como representado figura 12, no relacionamento entre a aresata “a” com o vértice “1”.

Figura 13: Um digrafo e sua Matriz de Incidência



INCIDÊNCIA						
Grafo B	1	2	3	4	5	6
a	1	0	0	0	0	0
b	0	1	0	0	0	0
c	0	0	0	0	1	0
d	0	1	0	0	1	0
e	0	1	0	0	0	0
f	0	0	0	0	1	0
g	0	0	1	0	0	0
h	0	0	1	0	0	0

Semelhante ao que ocorro em matrizes de incidência quando se trata de dígrafos, matrizes de adjacência represntam o relacionamento entre arestas com os vértices que estão se

direcionando. Como exemplo apresentado na figura 13, a aresta “f” se direciona apenas ao vértice 5, logo, não se relaciona com o sua origem, o vértice 4.

2.4. TRABALHOS RELACIONADOS

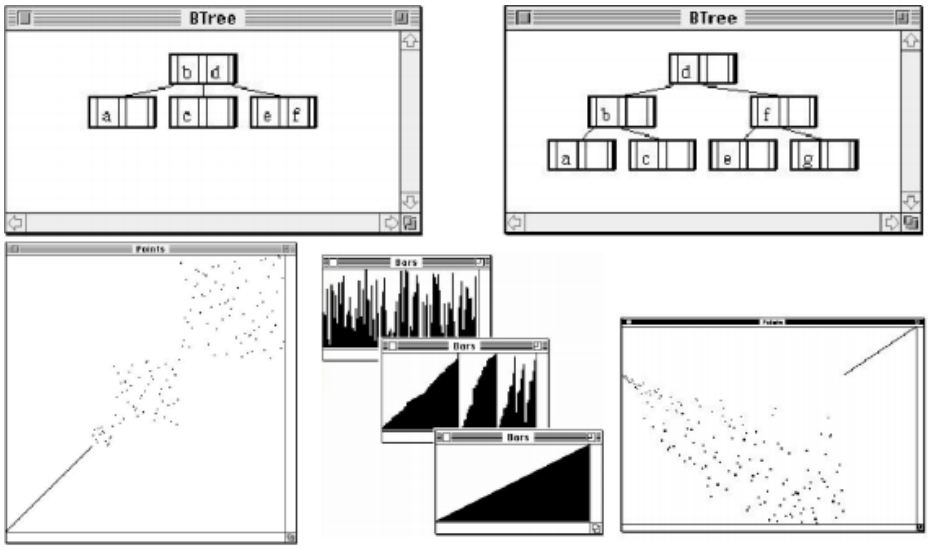
Nesta seção são descritos alguns trabalhos relacionados que influenciaram este trabalho.

2.4.1. ASTRAL: UM AMBIENTE PARA ENSINO DE ESTRUTURAS DE DADOS ATRAVÉS DE ANIMAÇÕES DE ALGORITMOS

Em Garcia et al. (1997) foi apresentado o projeto do ambiente Astral, desenvolvido em Pascal, que visou atender as necessidades de preparação de diversos exercícios de implementação de estrutura de dados, incluindo um módulo, através de animações.

O ambiente Astral foi desenvolvido para arquitetura Macintosh, visando as facilidades oferecidas por esta plataforma na época quanto à implementação de aplicativos gráficos. O ambiente Macintosh foi pioneiro na utilização de interfaces gráficas de usuário, contando com um ambiente mais robusto para desenvolvedores.

Figura 14: Tela de resultado do sistema Astral



Fonte: adaptado de Garcia et al. (1997)

A Figura 14 apresenta, da esquerda para direita e cima para baixo: uma árvore binária, antes e depois de uma inserção no nó raiz, a animação de um algoritmo QuickSort, três estágios do algoritmo MergeSort e por fim, resultado da animação do algoritmo HeapSort.

No que tange a linguagem de programação, o ambiente foi desenvolvido em Pascal, considerado pelos autores como a linguagem mais adequada ao ensino de programação básica, no contexto da época de seu desenvolvimento.

Voltado mais especificamente para a visualização de modo animado de grafos e árvores, o ambiente contava com a participação ativa do usuário. Para sua utilização, o usuário recebia um arquivo de texto explicativo e um aplicativo-exemplo para análise e replicação com seus próprios dados de entrada, que eram carregados de volta no ambiente para gerar as estruturas definidas.

De acordo com os autores uma sensível melhoria de desempenho foi percebida com a utilização da ferramenta em comparação com semestres anteriores à sua utilização, apesar de indicarem a necessidade de estudos mais laborados para quantificar essa melhoria.

2.4.2. ODIN - AMBIENTE WEB DE APOIO AO ENSINO DE ESTRUTURAS DE DADOS LISTA ENCADEADA

Neste trabalho de Madeira et al (2012) foi apresentado o sistema Odin, criado em conjunto entre os departamentos de Ciência da Computação da Universidade do Extremo Sul Catarinense (UNESC). O objetivo do trabalho era criar um ambiente de apoio ao ensino de lista encadeada, na disciplina de estrutura de dados.

O ambiente foi desenvolvido para operar em navegadores web utilizando duas linguagens diferentes: C++ para implementação das listas, por ser o padrão utilizado no curso de Ciência da Computação da UNESC, e Java para a construção da interface, em razão de sua operacionalidade e fácil utilização em navegadores.

Para sua utilização, o usuário conta com uma tela composta por três partes: uma contendo o código fonte de cada função da lista, outra para apresentação do grafo e a ultima contendo os menus e botões que permitem a interação do usuário com o ambiente.

O trabalho buscou utilizar critérios ergonômicos possibilitando a demonstração gráfica das funções dos algoritmos de lista encadeada. Ao final do projeto é salientada a dificuldade encontrada para relacionar os conceitos de informática na educação.

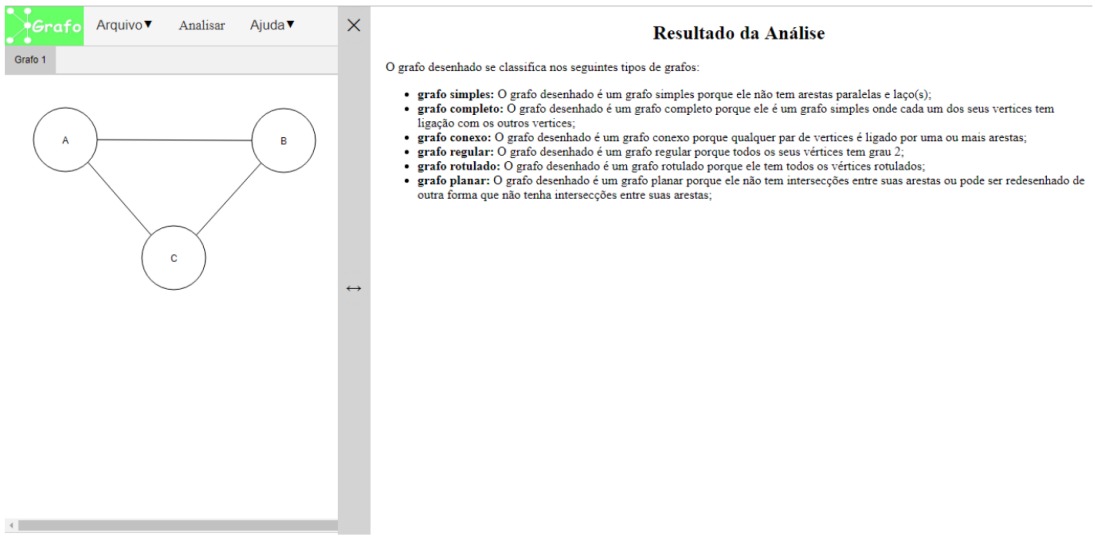
2.4.3. CRIAÇÃO DE UMA FERRAMENTA INFORMATIVA SOBRE TEORIA DOS GRAFOS

Nesta monografia, Silva Filho (2017), também aluno do CEULP/ULBRA, apresenta o resultado de seu desenvolvimento de uma ferramenta semelhante à proposta neste trabalho: uma ferramenta para visualização de grafos.

A ferramenta é voltada especificamente para grafos, para utilização em navegadores Web e desenvolvida primariamente em HTML e *JavaScript*. A escolha do ambiente se deu pela disponibilidade alcançada por um ambiente Web e pelas vantagens do elemento SVG do HTML.

Durante sua utilização, são apresentados ao usuário uma tela e alguns botões, que permitem o desenho livre dos grafos, indicando as posições dos vértices, seus respectivos valores e as conexões das arestas. Em seguida, com o grafo desenhado, o sistema apresenta uma segunda tela, contendo as informações e características do grafo desenhado.

Figura 15: Tela de resultado do sistema Astral



Fonte: adaptado de Silva Filho (2017).

Como resultado a ferramenta criada pelo autor permite ao usuário, em um módulo, desenhar grafos manualmente e em outro visualizar informações sobre o grafo desenhado. Silva Filho (2017) também propõe a complementação de seu trabalho com a implementação de funções sobre grafos, como algoritmos de busca.

O projeto proposto neste trabalho se diferencia deste pela utilização de ferramentas de desenvolvimento mais atualizadas, como: a plataforma Angular, linguagem Typescript e a ferramenta Graphviz, que serão descritos na sessão Materiais e Métodos. Além das ferramentas, este projeto propõe a integração com um ambiente de auxílio ao ensino e a utilização de uma linguagem de programação simplificada como entrada para geração dos grafos.

3 MATERIAIS E MÉTODOS

As subseções a seguir apresentam materiais e procedimentos utilizados para o desenvolvimento deste trabalho.

3.1. MATERIAIS

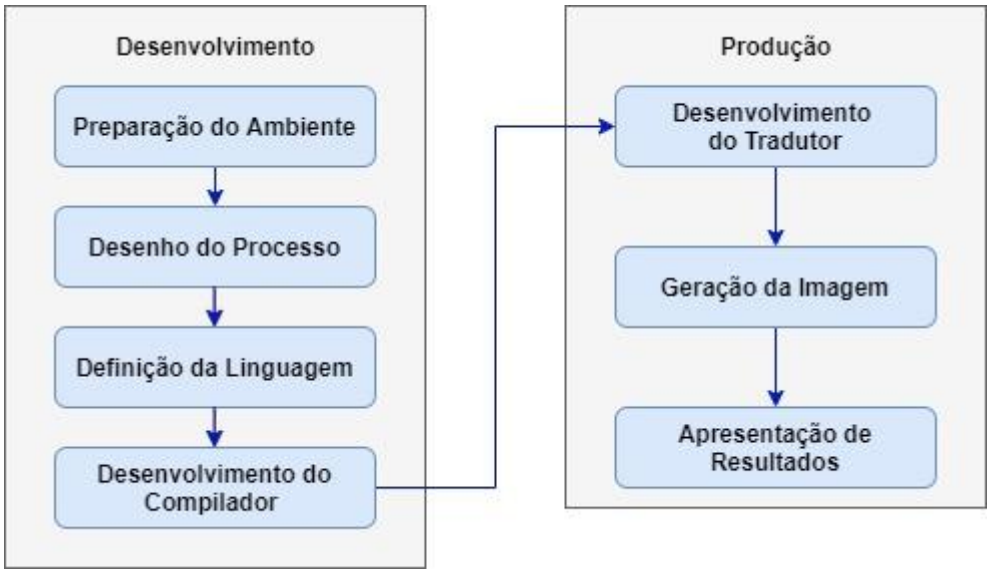
Abaixo são especificadas as linguagens de programação e os softwares que serão utilizados no desenvolvimento do trabalho:

- **HTML** – (*HyperText Markup Language*) ou Linguagem de Marcação de Hipertexto está atualmente em sua quinta versão e será utilizada para a produção de páginas web e criação de documentos que possam ser lidos por praticamente qualquer navegador;
- **CSS** – (*Cascading Style Sheets*) é um mecanismo para adicionar estilos (cores, fontes, espaçamento, etc) a um documento web e, neste projeto, será utilizado para formatação de documentos HTML;
- **Bootstrap -** é um *framework* livre para desenvolvimento de interfaces e *front-end*, que oferece uma enorme variedade de plug-ins e temas. É de fácil integração com as linguagens utilizadas para o desenvolvimento da ferramenta;
- **JQuery** – biblioteca de código aberto de funções *JavaScript/TypeScript* que interage com o HTML para simplificar os *scripts* interpretados pelo navegador do cliente;
- **Visual Studio Code** – editor de código fonte desenvolvido pela Microsoft, livre e de código aberto, com suporte para depuração, Git, realce de sintaxe, complementação de código e muito mais;
- **JSON Viewer** – extensão para o Chrome para visualização de formato JSON, utilizado para testar os retornos do formato;
- **Draw.io** – software livre o online para criação de fluxogramas, diagramas de processos, gráficos organizacionais, UML, entre outros. Utilizado para criar diagramas utilizados na documentação do projeto;
- **Graphviz** – Pacote de ferramentas de código aberto para desenhar gráficos especificados na linguagem DOT. Utilizado para gerar grafos a partir de uma entrada de texto em DOT;
- **ANTLR4** – (*ANother Tool for Language Recognition 4*) é um gerador de *parser* (analisador) para criar linguagens. Neste contexto será utilizado para converter o código de entrada do usuário em linguagem DOT;

3.2. MÉTODOS

Esta seção descreve a metodologia adotada para a realização deste trabalho, que serão apresentados em detalhes no capitulo a seguir.

Figura 16: Fluxograma de Desenvolvimento



As informações apresentadas na Figura 16 representam cada passo da metodologia utilizada. Como pode ser identificado, o desenvolvimento foi dividido em duas grandes etapas, compostos por processos menores.

A primeira etapa é definida por processos necessários para o desenvolvimento, não sendo necessária para a utilização do projeto final, ou seja, a primeira etapa desenvolve as ferramentas necessárias para o desenvolvimento da segunda. O primeiro destes processos foi a preparação do ambiente, instalação de programas e sistemas necessários. Em seguida foi realizado um planejamento do processo de desenvolvimento, definindo os principais pontos de execução.

No processo de definição da gramática utilizada foi realizada a análise de funcionamento da linguagem DOT e planejada a linguagem utilizada pelo usuário do GraphLive. Esta primeira etapa finalizou-se com a criação do compilador gerado pelo ANTLR4 baseado na gramática estabelecida no processo anterior. A linguagem de programação definida para o compilador foi *JavaScript*, por sua integração com o ambiente WEB, apesar disso, o ANTLR4 possibilita a utilização de várias outras linguagens.

A segunda etapa foi definida pela elaboração da página principal em HTML, na qual um algoritmo *JavaScript* recebe a entrada do usuário e envia para o compilador, que por sua vez fragmenta o código utilizando uma Árvore de Sintaxe Abstrata gerada pelo compilador, traduzindo-os de acordo com as regras pré-estabelecidas na gramatica. Também é nesta etapa

que se realiza a verificação do grafo descrito com base em regras estabelecidas pela teoria dos grafos, cujos principais algoritmos são apresentados.

Na sequância foi apresentada a geração da imagem do grafo, seguida da do processo de apresentação das informações gerais para o usuário.

4 RESULTADOS E DISCUSSÃO

Esta seção tem por objetivo apresentar detalhes do processo de desenvolvimento deste trabalho, descrever sua utilização e apresentar os resultados obtidos.

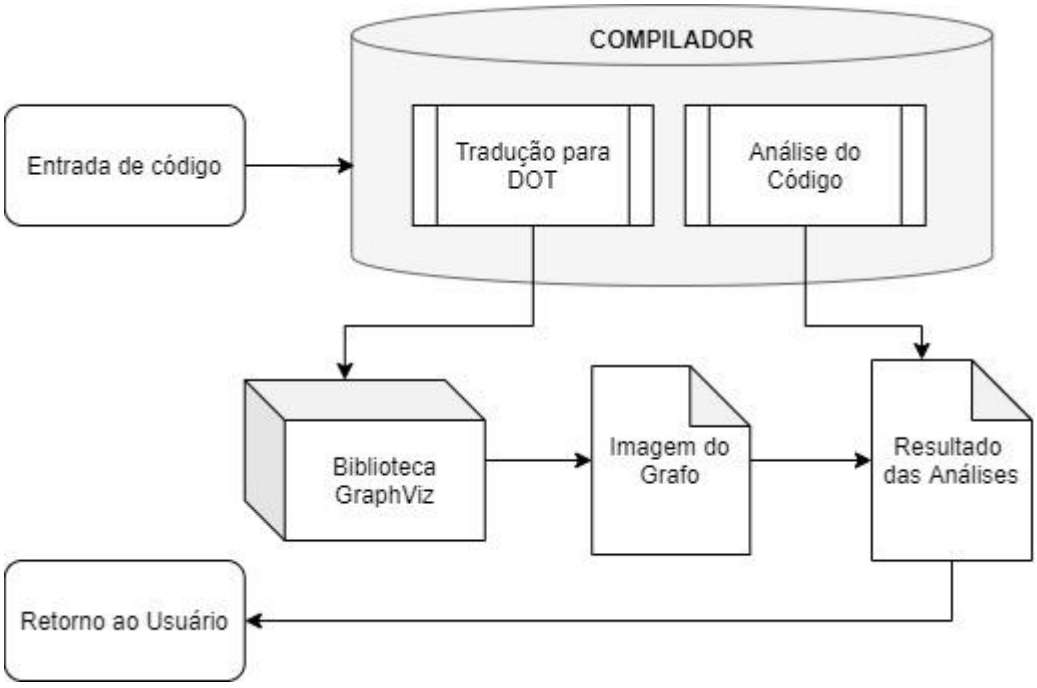
4.1. PREPARAÇÃO DO AMBIENTE

Esta primeira etapa compreendeu a preparação do ambiente de desenvolvimento e instalação das ferramentas necessárias para a construção deste projeto. Em destaque, a instalação do ANTLR4 exige a criação de varáveis de ambiente e a instalação de um servidor web integrada ao editor Visual Code. Ambos os processos foram necessários apenas para desenvolvimento do projeto, não sendo necessários para utilização final da ferramenta.

4.2. DESENHO DO PROCESSO

Nesta etapa foi elaborado o formato pelo qual o sistema funciona, com apresentação dos processos que são executados e em sua sequência, conforme apresentado na figura abaixo.

Figura 17: Fluxograma de Execução



Em linhas gerais, conforme apresentado na figura 17, a utilização do GraphLive ocorre da seguinte forma:

- o usuário realiza entrada de texto com o código na linguagem desenvolvida.
- o compilador recebe a entrada e realiza duas funções:
    - Traduz o código para a linguagem de descrição de grafos do GraphViz, a linguagem DOT;
    - E realiza as análises do código de entrada de acordo com parâmetros pré-estabelecidos referentes às características dos grafos;
- o código traduzido para a linguagem DOT é exportado para uma biblioteca do GraphViz que gera a imagem do grafo em SVG.
- a imagem gerada e o resultado das análises são retornados ao usuário.

Uma vez que o processo foi definido, foi possível seguir o desenvolvimento da ferramenta, seguindo para a definição da gramática para tradução da linguagem de entrada.

4.3. DEFINIÇÃO DA GRAMÁTICA

O processo de definição da gramática começou pelo estudo da linguagem DOT, para qual a linguagem utilizada pelo GraphLive será traduzida.

O GraphViz é uma ferramenta para desenho de grafos de alto nível utilizando a linguagem DOT. Apesar de ser capaz de gerar grafos altamente elaborados, o GraphViz não gera informações sobre eles, e a linguagem DOT é muito complexa e verbosa para estudantes, de modo que o aprendizado desta linguagem poderia comprometer o aprendizado de grafos.

Na linguagem DOT, todo grafo deve ser iniciado pela identificação de grafo direcionado ou não-direcionado, com a utilização da função ***graph*** ou ***digraph*** respectivamente. Esta característica não foi passada à linguagem do GraphLive, pois a identificação do tipo de grafo é esperada como resultado da entrada.

Os vértices são declarados manualmente em cada relacionamento, o relacionamento indicado por setas e suas características são indicadas após o relacionamento de forma descritiva. A imagem a seguir apresenta um exemplo de grafo descrito na linguagem DOT.

Figura 18: Exemplo de descrição de grafo na linguagem DOT

```
digraph teste {
    rankdir="LR"
    size="8"
    PALMAS -> PARAISO [label = 60]
    PALMAS -> PORTO [label = 70] [dir = both]
    PARAISO -> PORTO [label = 150]
}
```

Como evidenciado na figura 18, a descrição repetitiva dos vértices pode ocasionar grafos duplicados por conta de erros de digitação. O relacionamento é sempre feito da esquerda para direita, mesmo quando se trata de um relacionamento bidirecional.

Ao fim de cada relacionamento são definidas as características como valor (***label***) e se necessário a função de indicação bidirecional (***both***) e pode-se notar termos de configuração do grafo, ***rankdir*** para alinhamento e ***size*** para tamanho da fonte. Além destes, existem muitos outros termos de configuração e características, mas são desnecessárias para o escopo deste projeto.

A linguagem desenvolvida para o GraphLive busca reduzir inconsistências referentes à erro de digitação de código repetido, além de limitar o uso de termos de configuração de imagem que não são relevantes para o processo, utilizando-se de uma forma padrão e mais intuitiva. Os principais pontos para a definição da linguagem foram:

- o grafo sempre deve ser iniciado com a função ***graph***, independente de seu direcionamento;
- identificação dos vértices através de variáveis, evitando repetição constante dos nomes dos vértices;
- utilização para o relacionamento as variáveis para representar tais vértices;
- utilização de esquema de traços para indicar a aresta que fará o relacionamento entre os nós, identificando o valor entre os traços, se houver;
    - “--” para relacionamento simples;
    - “-X-” para relacionamentos simples com valor, onde “X” pode ser substituído por qualquer valor;
    - “-->” para relacionamento direcionado ao nó à direita sem valor de aresta;
    - “-X->” para relacionamento direcionado ao nó à direita com valor de aresta;
    - “<-->” para relacionamento bidirecional sem valor de aresta;
    - “<-X->” para relacionamento bidirecional com valor de aresta.

A seguir, segue um comparativo da linguagem DOT com a linguagem GraphLive:

Figura 19: Código DOT e equivalente na linguagem GraphLive.

DOT	GraphLive

digraph teste {
rankdir="LR"
size="8"
PALMAS -> PARAISO [label = 60]
PALMAS -> PORTO [label = 70] [dir = both]
PARAISO -> PORTO [label = 150]
}
	graph teste:
	A = "PALMAS"
	B = "PARAISO"
	C = "PORTO"
	A -60-> B
	A <-70-> C
	B -150-> C

A figura 19 revela a simplificação do código quanto à descrição dos relacionamentos. A linguagem GraphLive apresenta os relacionamento de forma mais intuitiva e prática, ocultando termos desnecessários para construção dos grafos no escopo definido. A figura abaixo apresenta a gramática utilizada para gerar a linguagem acima.

Figura 20: Código DOT e equivalente na linguagem GraphLive.

```
grammar GraphG;

prog: graph* EOF;
graph: graph_nome declaracao+ relacao+;
graph_nome: 'graph ' nome ':';
nome: ALFANUM;
label: ALFANUM;
declaracao: nome '=' '"' label '"';
direcao: '<' | '>';
aresta: '--'|('-'nome'-');
relacao: label direcao? aresta direcao? label;


fragment LETRA: [a-zA-Z];
fragment DIGITO: [0-9];
ALFANUM: (LETRA|DIGITO|'_')+;
WS: [ \t\r\n]+ -> skip;
```

A figura 20 apresenta a gramática *GraphG* utilizada na construção do compilador no qual cada ***token*** define uma sequência de caracteres e limitadores que podem ser identificados pelo analisador léxico.

Na gramática *GraphG* os ***tokens*** são definidas as regras de análise (*parser*) e regras léxicas (*lexer*) do texto de entrada, iniciando pelas regras de análise, que indica o que cada ***token*** representa e do que é composto.

Por exemplo, o ***token*** *graph* é composto por outros ***tokens***: *graph_nome*, *declaracao*, e *relacao*. Estes ***tokens***, por sua vez, são compostos por outros ***tokens*** ou por componentes léxicos, indicados no final da gramática. Já os componentes léxicos são compostos por caracteres reconhecidos pela linguagem.

4.4. DESENVOLVIMENTO DO COMPILADOR

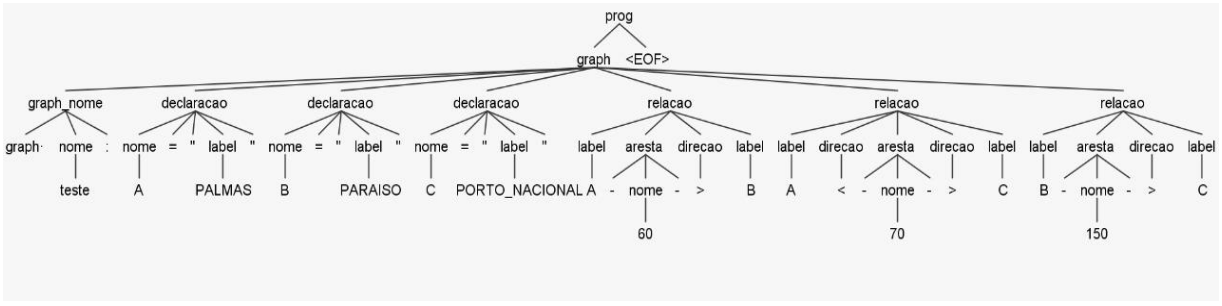
Neste projeto, o compilador foi desenvolvido com o ANTLR4 para executar duas funções: a primeira é traduzir o código de entrada na linguagem GraphViz para a linguagem DOT, e a segunda é realizar as verificações do grafo descrito.

Para a realização do primeiro passo, foi necessário definir uma gramática contendo todas as regras da linguagem, seguindo o padrão da ferramenta. A partir desta gramática é gerado um ***lexer*** (ou analisador léxico) e um ***parser*** (ou analisador sintático).

Em seguida o ***lexer*** consegue identificar, a partir de uma entrada de texto, quais palavras do texto correspondem a qual ***token***, ou seja, se as palavras utilizadas fazem parte da gramática estabelecida.

A partir da identificação dos ***tokens***, é realizada a análise sintática pelo ***parser*** que identifica se os as palavras chaves estão estruturadas da forma organizada pela gramática. Como resultado desta análise, é gerado uma Arvore de Sintaxe Abstrata (ou AST, do inglês *Abstract Syntaxe Tree*), distribuindo as palavras-chaves assimiladas em seus nós a partir das regras estabelecidas pela gramática.

Figura 21: AST gerada a partir da gramática e do texto de entrada.



Os nós folha da Arvore de Sintaxe Abstrata correspondem às palavras identificadas pelo *parser e,* conforme apresentado na figura 21, são distribuído entre pelos nós representando os *tokens*. Uma vez que a gramática foi testada e atende a todas as regras definidas, o ANTLR4 pode importá-la para utilização como uma biblioteca *JavaScript*.

Figura 22: Trecho do *parser* importado.

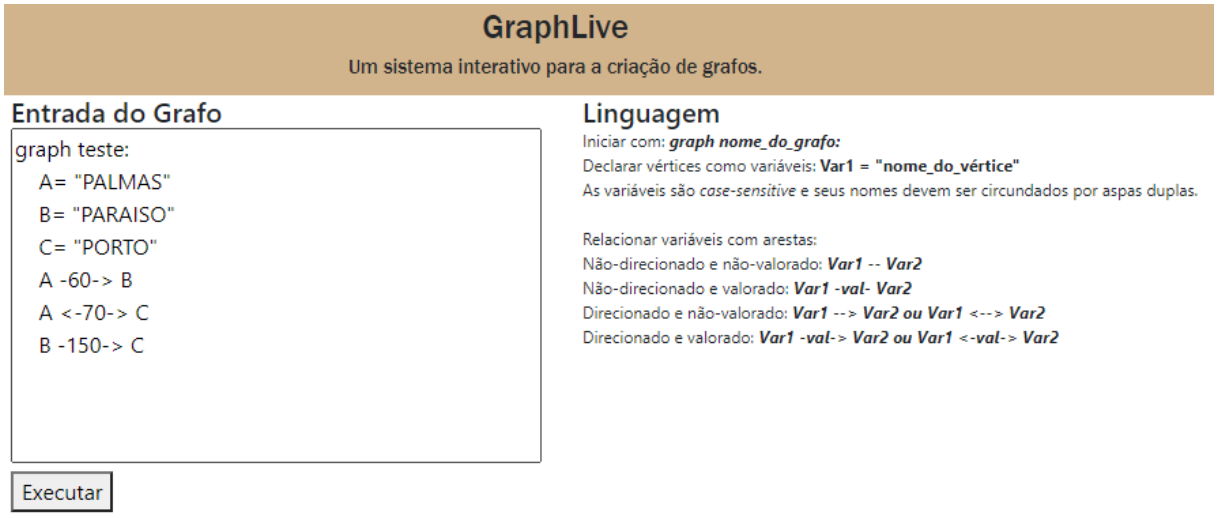
```
// Enter a parse tree produced by GraphGParser#label.
HtmlGraphGListener.prototype.enterLabel = function(ctx) {
};

// Exit a parse tree produced by GraphGParser#label.
HtmlGraphGListener.prototype.exitLabel = function(ctx) {

};

// Enter a parse tree produced by GraphGParser#declaracao.
HtmlGraphGListener.prototype.enterDeclaracao = function(ctx) {

};

// Enter a parse tree produced by GraphGParser#direcao.
HtmlGraphGListener.prototype.enterDirecao = function(ctx) {

};
```

O *parser* importado é capaz de percorrer a AST com funções executadas sempre que um nó é visitado, uma vez para chegada e uma vez para saída, conforme trecho apresentado na figura 22. Os nós são visitados de forma recursiva, e é com estas funções que é possível realizar a análise e tradução do código de entrada.

Todo o processo descrito até agora foi necessário para a obtenção do *parser* importado como compilador. Todo o restante do projeto utiliza este arquivo como base.

O usuário utiliza uma página HTML criada para o recebimento do código na linguagem criada. Uma breve introdução da linguagem desenvolvida é apresentada e um código de exemplo é pré-gravado em sua execução.

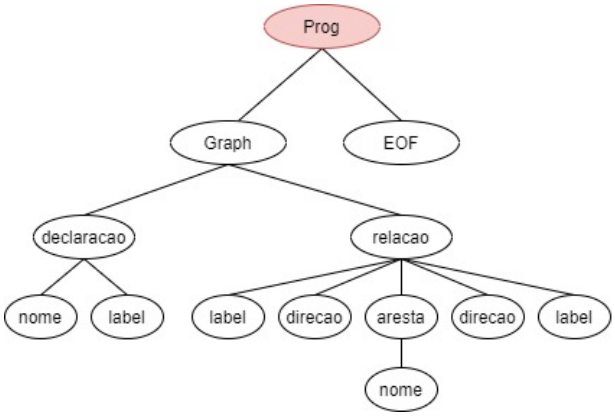
Figura 23: Tela de entrada do usuário



Uma vez que o código de descrição do grafo tenha sido inserido, o usuário deve clicar no botão “Executar”, como apresentado na figura 23. Uma função em *JavaScript* faz a leitura do código de entrada e o chama os métodos gerados pelos arquivos exportados pelo ANTLR4, para execução do compilador.

Ao entrar no compilador, o código de entrada é distribuído pelos nós da Arvore de Sintaxe Abstrata, nesta próxima etapa, o *parser* a percorre executando funções de entrada e saída, que podem ser utilizadas dependendo da necessidade. Cada função definida com o nome *enter* e *exit* seguidos pelo nome do *token*, apresentados na figura 20, cujos principais serão apresentados a seguir.

Figura 24: Função *enterProg* e sua posição na AST.

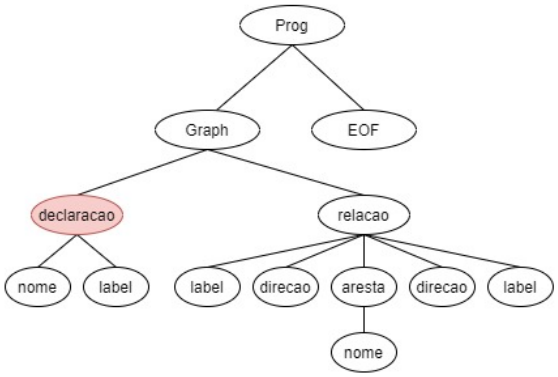



```
// Enter a parse tree produced by GraphGParser#prog.
HtmlGraphGListener.prototype.enterProg = function(ctx) {

    ctx.dir = 0;
    ctx.arestas = 0;
    ctx.vertices = [];
    ctx.qtNodes = 0;
    ctx.graus = [];
    ctx.labels = [];
    ctx.adjacencia = [];
    ctx.listaArestas = [];

};
```

A primeira chamada de função ocorre na entrada do nó raiz, ou seja, a função *enterProg*, na qual são definidas algumas variáveis que serão utilizadas. As funções de entrada são mais utilizadas para declaração de variáveis, já que os nós filhos ainda não foram visitados para terem algum resultado válido.

Em seguida foi feita a identificação dos nós declarados no código de entrada, neste caso, o algoritmo foi executado na função de saída ***exitDeclaracao***, conforme imagem abaixo.

Figura 25: Função exitD*eclaracao* e sua posição na AST.

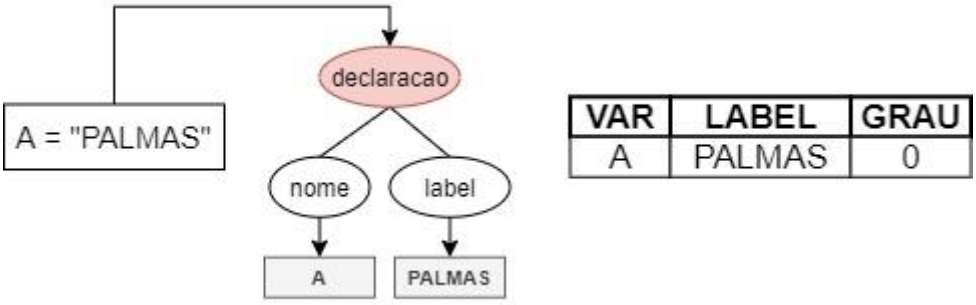



```
// Algoritmo na saida do nó Declaracao para criar um array com todos os nós declarados;
HtmlGraphGListener.prototype.exitDeclaracao = function(ctx) {

    var dataLabel = [];

    for(var i=0; i<ctx.children.length; i++){

        if(ctx.children[i] instanceof GraphGParser.GraphGParser.NomeContext){

            var dataset = {'node':ctx.children[i].getText(), 'grau':0}
            this.graus.push(dataset);
            this.qtNodes ++;
            dataLabel.push(ctx.children[i].getText());

        }

        if(ctx.children[i] instanceof GraphGParser.GraphGParser.LabelContext){
            dataLabel.push(ctx.children[i].getText())
        }
    }

    this.labels.push(dataLabel);

};
```

Uma vez que se trata de uma função de saída, os nós filhos (***nome*** e ***label***) já foram percorridos, seus valores estão disponíveis. Dentro de um laço de repetição os nós filhos são verificados com funções definidas.

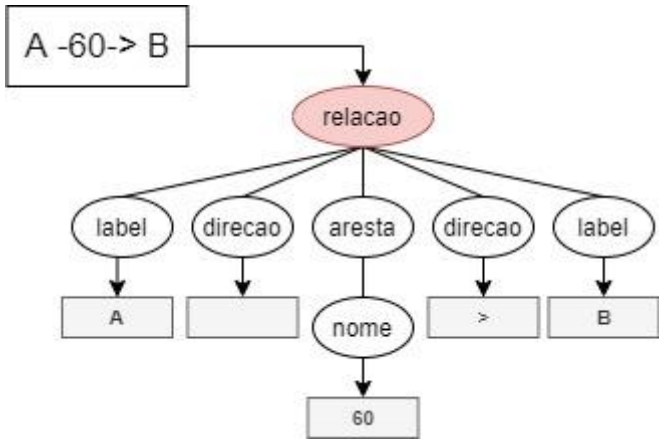
Para o nó ***nome*** foi criado um conjunto de dados com a variável do vértice e o valor de seu grau, iniciado em 0, em seguida o conjunto foi armazenado em um *array* declarado no início das funções AST. Além disso, a quantidade de nós foi contada para retorno ao usuário.

Figura 26: Exemplo de declaração e sua distribuição na AST.



Conforme explicitado na figura 26, para cada declaração de vértice no código de entrada, é feita a identificação do mesmo no *array* de vértices, que é utilizado na próxima etapa. A seguir, na função ***exitRelacao*** é feita a leitura e verificação das relações indicadas pelo usuário através do código de entrada. Primeiramente, são analisados os conteúdos do primeiro e ultimo filhos do nó, que são sempre as declarações das variáveis referentes aos vértices da relação, conforme apresentado na representação do trecho da AST abaixo.

Figura 27: Exemplo de uma relação e sua distribuição na AST.



Na sequência, para cada variável de vértice da relação é criada uma variável local para armazená-los temporariamente. Em seguida os outros filhos são percorridos analisando seu tipo com operadores gerados automaticamente pelo ANTL4 a partir da gramática estabelecida.

Figura 28: Função de validação dos nós filhos do tipo *direcao*.

```
if(ctx.children[i] instanceof GraphGParser.GraphGParser.DirecaoContext){
    ctx.relacao += ctx.children[i].getText();
}
```

Assim que o nó filho do nó ***relação*** foi identificado como uma instância de ***direção***, ou seja, um nó do tipo direção, seu conteúdo foi adicionado a uma variável local. Esta ação pode ocorrer uma ou duas vezes em cada relacionamento, indicando a direção do mesmo, seja para esquerda ou direita.

A última validação dos nós filhos realizada nessa etapa é identificação do nó como instancia de ***aresta*** que pode conter ou não um valor de relação entre os vértices. Caso haja um valor dado ao relacionamento, o mesmo será adicionado a uma variável local.

O relacionamento então é nomeado, identificado pela letra ‘a’ seguida de um numero sequencial e armazenado em uma lista de identificadores de arestas, por exemplo: a1, a2, a3...

Ainda na função ***exitDeclaracao***, após a verificação de todos os nós filhos, foi declarada uma variável ***saída*** contendo uma *string* vazia. Seu conteúdo será preenchido a partir das variáveis locais obtidas durante o processo de verficação anterior iniciando o processo de tradução para a linguagem DOT.

A variável contendo a referência do primeiro vértice do relacionamento é identificada na lista vértices declarada no inicio do processo. O nome do vértice referente a essa variável é

então adicionado à *string* ***saída***. Em seguida é feita a validação quanto à identificação de direcionais na relação, obtidos na variável local.

Caso a variável esteja vazia, ou seja, se não houve direcionamento, um conjunto de traços (--) é adicionado à *string* ***saída***, caso contrário, o valor adicionado é o traço seguido do sinal "maior que" (->). Em sequência, a mesma verificação do primeiro vértice é feita no segundo, e seu nome referenciado também é adicionado à ***saída***.

Continuando a tradução da relação entre vértices, a variável local declarada a partir do nome do relacionamento é verificada, caso não esteja vazia, o texto "***[label = aresta: valor]***" é concatenado à *string* saída, de forma que o trecho *aresta* é substituído pelo identificador da aresta e o trecho *valor*, pelo conteúdo do nome do relacionamento.

O mesmo processo se repete, verificando a variável local referente aos direcionamentos identificados. Caso seu conteúdo seja a *string* "<>" representando relacionamento bidirecional, o texto "***[dir = both]***" também é concatenado à *string*, finalizando com o resultado explicitado na imagem a seguir.

Figura 29: Resultados obtidos na execução da função *exitDeclaracao*.

ENTRADA	SAÍDA
A <-60-> B	PALMAS -> PARAISO [label: "a1: 60"] [dir: both]

LISTA	FORMATO	VALORES
Relacionamento	{Vértice 1, Nome, Vértice 2, Aresta}	{A, 60, B, a1}
Graus	{Vértice, Label, Grau}	{A, PALMAS, 1} {B, PARAISO, 1}

Como saída, o nó ***exitDeclaração*** retorna as informações apresentadas na figura 29: a *string* ***saída*** contendo o relacionamento traduzido para DOT, inclusões no *array* de relacionamentos e o *array* de vértices com seus *labels* e graus.

Seguindo a AST, a execução retorna ao nó ***Prog*** cujos filhos realizaram as declarações de vértices e de relacionamentos, além de ser o ultimo nó antes de retornar ao usuário. nesta etapa a função ***exitProg*** finaliza a tradução do código de entrada para a linguagem DOT.

Caso algum dos relacionamentos contenha sinais direcionais, o grafo será considerado um grafo direcionado, ou um ***digraph*** como é identificado na linguagem DOT, caso contrário, será considerado um ***graph***.

Figura 30: Texto padrão DOT concatenado com *string saida*.

```
if(qtDir > 0){
    saida = 'digraph '+nome+' {\n rankdir=LR; \n size="8";\n '+ saida
}
if(qtDir == 0){
    saida = 'graph '+nome+' {\n rankdir=LR; \n size="8";\n '+ saida
}
saida += '\n}'

this.value.push(saida);
```

O texto padrão é concatenado ao nome do grafo definido no código de entrada e com a *string* ***saída*** que contem a concatenação dos relacionamentos traduzidos na etapa anterior.

Com todas as informações obtidas e catalogadas em *arrays* para retorno ao usuário, mas algumas verificações sobre as características do grafo descrito são realizadas. Estas características são identificadas em um ultimo *array* de dados.

Após este processo, a *string* traduzida, e as listas de variáveis identificadas são retornadas a pagina do usuário para procedimentos finais.

4.5. GERAÇÃO DA IMAGEM

O GraphViz fornece uma biblioteca para geração de grafos tendo como entrada um código na linguagem DOT, que no caso foi obtida pela tradução realizada pelo compilador. O código traduzido é passado a esta biblioteca que retorna a imagem do grafo no formato SVG.

Foi realizada uma declaração da classe ***Viz*** da biblioteca externa que possui a função de renderização do SVG a partir do código em DOT. O retorno, no caso a imagem, é atribuída a uma área especifica da pagina HTML indicada pelo *id* “imagem”.

Figura 31: Função de renderização do Grafo.

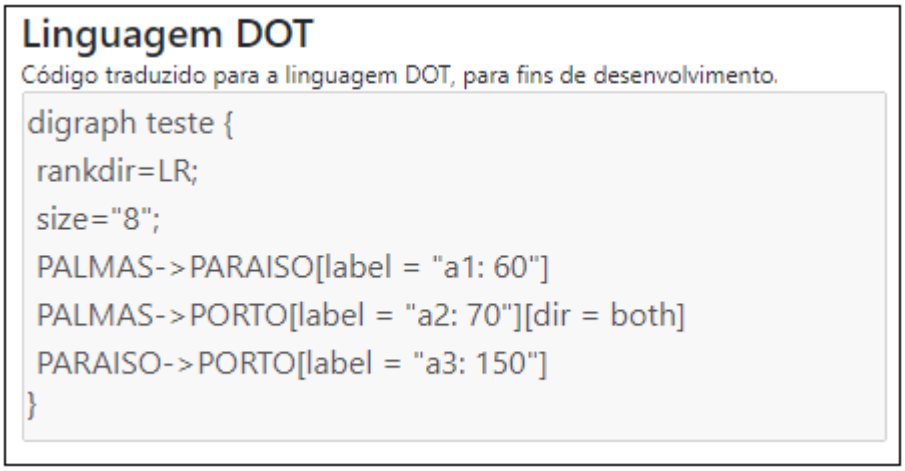
```
var viz = new Viz();

viz.renderSVGElement(htmlGraphG['value'][0])
.then(function(element) {
    var elm = document.getElementById('imagem');
    if(elm.childElementCount > 0){
        elm.innerHTML='';
    }
    desenhar();
    elm.appendChild(element);

})
.catch(error => {
    viz = new Viz();
    console.error(error);
});
```

A função de renderização apresentada na imagem 31 utiliza uma operação de composição através da função ***promise*** utilizada para processamento assíncrono e tem como retorno um dos dois métodos: ***then*** para processamento bem sucedido e ***catch*** para processamento mal sucedido.

4.6. APRESENTAÇÃO DOS RESULTADOS

Partido de um resultado bem sucedido da tradução e da geração da imagem do grafo é utilizado ***jquery*** para atribuir as informações retornadas pelo compilador em seus devidos locais na página. Em um primeiro momento, o código traduzido para DOT também é apresentado apenas para fins de comparação, não sendo necessário para sua utilização.

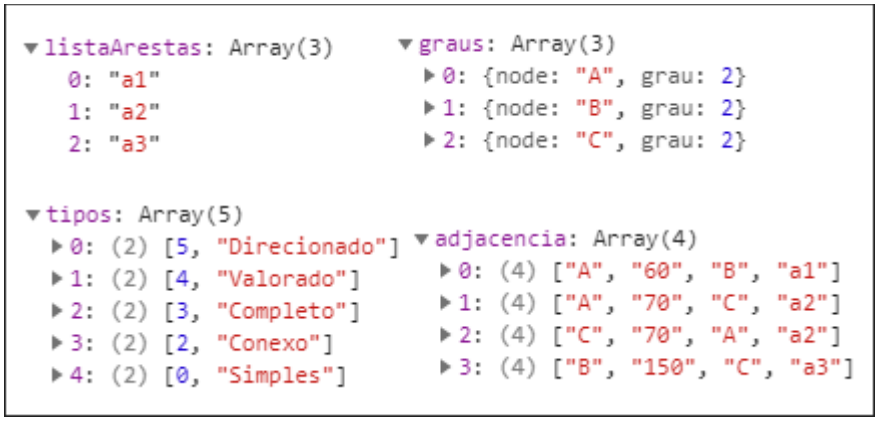
Figura 32: Tela de apresentação: código traduzido.



Com base na ultima verificação realizada são listados ao usuário os tipos que definem o grafo obtido. Os tipos de grafos identificados pelo GraphLive são: simples, multigrafo, conexo, completo, valorado ou não valorado e direcionado ou não direcionado.

Para apresentar estas informações, o compilador retorna um conjunto de *arrays* conforme apresentado na figura a seguir.

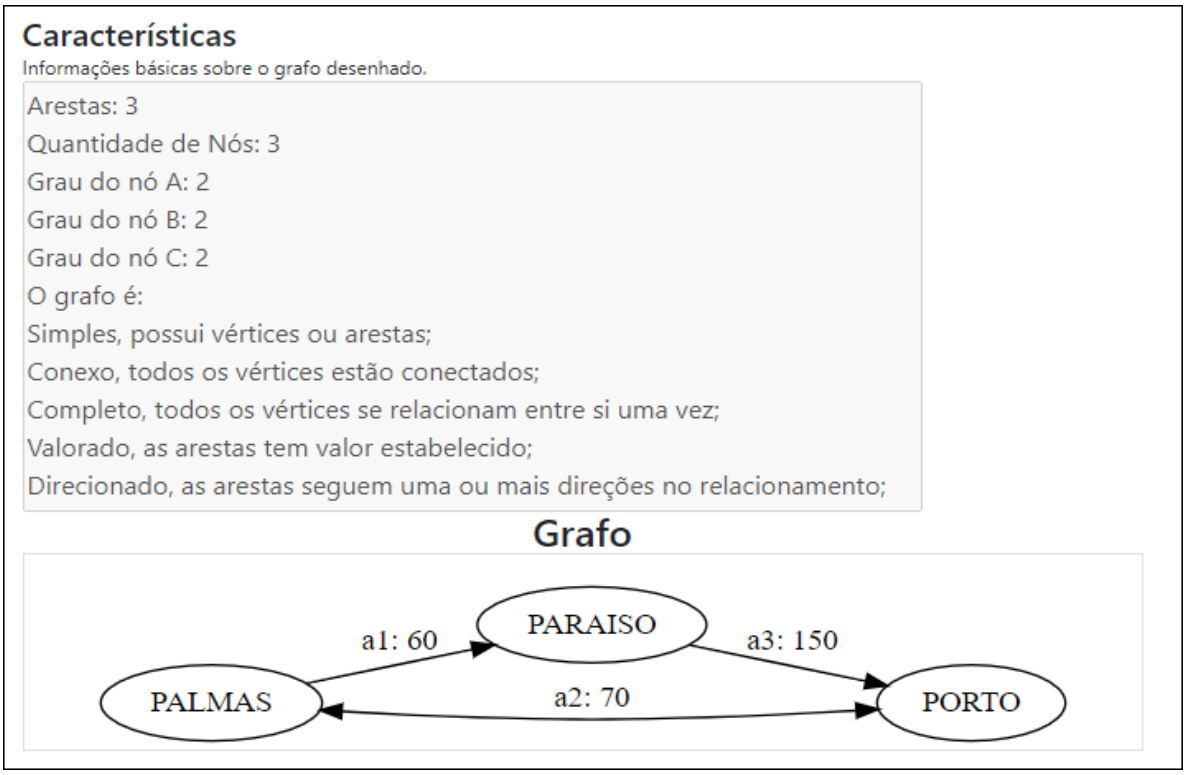
Figura 33: Tela de apresentação: características e grafo gerado.



A figura 33 apresenta os principais conjuntos de informações enviadas pelo compilador: a lista de nomes das arestas, os graus de cada nó, as características do grafo e a lista de relacionamentos, no formato vértice A, valor da aresta, vértice B e nome da aresta.

Com as informações do *array* ***listaArestas***, ***graus*** e ***tipos*** é possível apresentar as informações básicas do grafo e são incluídas no HTML via funções *jquery*. Já para a apresentação do tipo de grafo, um condicional *switch* foi utilizado para dar uma resposta de texto com base no primeiro indice de cada objeto do *array*.

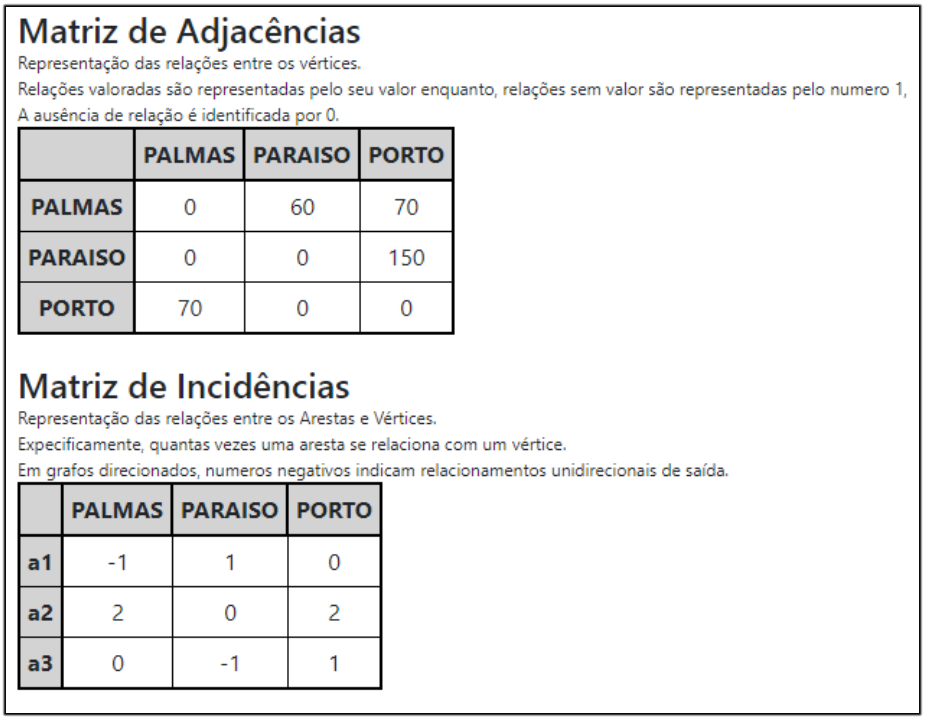
Figura 34: Tela de apresentação: características e grafo gerado.



Como evidenciado pela figura 34, as informações são apresentadas em texto plano, com uma breve explicação seguida da imagem do grafo que é apresentada para visualização do usuário.

Por fim, um conjunto de funções *JavaScript* é responsável por percorrer os *arrays* de retorno do compilador, gerando as matrizes de adjacência e de incidência, percorrendo o *array* ***adjacencia*** e relacionando-o com os outros *arrays*.

Figura 35: Tela de apresentação: Matrizes de Adjacências e Incidência.



Matriz de Adjacências

Representação das relações entre os vértices.
Relações valoradas são representadas pelo seu valor enquanto, relações sem valor são representadas pelo numero 1,
A ausência de relação é identificada por 0.

	PALMAS	PARAISO	PORTO
PALMAS	0	60	70
PARAISO	0	0	150
PORTO	70	0	0

Matriz de Incidências

Representação das relações entre os Arestas e Vértices.
Expecificamente, quantas vezes uma aresta se relaciona com um vértice.
Em grafos direcionados, numeros negativos indicam relacionamentos unidirecionais de saída.

	PALMAS	PARAISO	PORTO
a1	-1	1	0
a2	2	0	2
a3	0	-1	1

Para a Matriz de Adjacência, que relaciona os vértices entre si, o algoritmo valida a presença de relação e de valores nestas relações. Para relacionamentos valorados, o valor do relacionamento é apresentado na matriz, já para relacionamentos não valorados, é apresentada a quantidade de relações declaradas, 1 para relacionamento simples ou n para relacionamento com n arestas paralelas.

Relacionamentos valorados com arestas paralelas existem, mas não são conceitos abordados na literatura quando se trata de matrizes de adjacência, que são estruturas de dados de apenas duas dimensões, podendo conter apenas um relacionamento entre suas dimensões. Já a ausência de relacionamento é representada pelo numeral zero.

Já a Matriz de Incidência relaciona os vértices com as arestas. Em caso de grafos não direcionados, são apresentadas quantas vezes uma aresta se relaciona com um vértice, já para grafos direcionados, os vértices ligados apenas por arestas de “saída” são indicados por um numero negativo.

As duas matrizes são geradas dinamicamente: de adjacência pelo quadrado da quantidade de vértices e a de incidências pela multiplicação entre a quantidade de vértices pela quantidade de arestas. Por tanto, quanto maior o grafo, maiores serão as matrizes.

5 CONSIDERAÇÕES FINAIS

Neste trabalho foi desenvolvido um sistema WEB interativo para criação de grafos, o GraphLive, projetado para ser utilizado como ferramenta de auxilio ao ensino de teoria dos grafos. Para alcançar este objetivo foi utilizada uma ferramenta para reconhecimento de linguagem, o ANTLR4, para criação de uma linguagem própria utilizada pelo usuário para a descrição de grafos.

Com base no código de entrada, realizada por um compilador desenvolvido para este trabalho, é gerada a análise sobre o grafo descrito, referente às suas características: número de vértices, arestas, graus, e quanto a seus tipos: simples, multigrafo, conexo, completo, valorado ou não valorado e direcionado ou não direcionado, com uma breve explicação sobre o tipo indicado. Também são geradas matrizes de adjacência e incidência como informação complementar sobre os relacionamentos do grafo obtido.

A imagem do grafo é gerada a partir da biblioteca Graphviz, uma ferramenta própria para gerar grafos de alto nível, que utiliza a linguagem DOT para descrição de grafos, mas não gera informações sobre os mesmos. Parte deste trabalho foi a tradução de uma linguagem simples de descrição de grafos para a linguagem DOT, bem mais verboso, complexo e pouco recomendado para usuários iniciantes em programação.

Este sistema foi criado de forma a ser prontamente instalado em um servidor WEB para ser facilmente acessado de qualquer localidade por professores e alunos. A aplicação foi testada com diversos tipos de grafos, com suas características devidamente validadas.

O GraphLive possui um grande potencial de escalabilidade recomendados como trabalhos futuros, por exemplo a configuração de mensagens de erro de descrição do grafo, uma funcionalidade possível de ser alcançada utilizando o ANTLR4, mas que não foi implementada neste projeto. Outos projetos podem envolver a análise de tipos expecíficos de grafos como árvores, buscas e pesquisas em grafos. É possível também que este projeto possa ainda servir como uma base para o desenvolvimento de um sistema integrado completo para o auxílio ao ensino de estruturas de dados em geral.

REFERÊNCIAS

BONDY, John Adrian et al. **Graph theory with applications.** London: Macmillan, 1976. 270 p.

BRASIL, Ministério da Educação**. Diretrizes Curriculares Nacionais para os cursos de graduação em Computação - DCN.** Disponível em: <http://portal.mec.gov.br/escola-de-gestores-da-educacao-basica/323-secretarias-112877938/orgaos-vinculados-82187207/12991-diretrizes-curriculares-cursos-de-graduacao>. Acessado em 10 nov. 2020.

COSTA, Polyanna Possani da. **Teoria dos grafos e suas aplicações.** 2011. 77 p. Dissertação - (mestrado) - Universidade Estadual Paulista, Instituto de Geociências e Ciências Exatas, 2011. Disponível em: <http://hdl.handle.net/11449/94358>.

DOVICCHI, João. Grafos e Árvores. In: 2007. **Estrutura de Dados**. Florianópolis: Ufsc, 2007. Cap. 4. p. 7-91. Disponível em: <http://www.inf.ufsc.br/~joao.dovicchi/pos-ed/ebook/e-book_estrut_dados_dovicchi.pdf.> Acesso em: 18 ago. 2019.

FEOFILOFF, Paulo; KOHAYAKAWA, Yoshiharu; WAKABAYASHI, Yoshiko. **Uma introdução sucinta à teoria dos grafos.** 2011. Disponível em: <http://www.ime.usp.br/~pf/teoriadosgrafos/texto/TeoriaDosGrafos.pdf>.

GARCIA, Islene C.; REZENDE, PJ de; CALHEIROS, Felipe C. Astral: um ambiente para ensino de estruturas de dados através de animações de algoritmos. **Revista Brasileira de Informática na Educação**, v. 1, n. 1, p. 71-80, 1997.

MADEIRA, Maicon Francisco et al. ODIN - Ambiente Web de Apoio ao Ensino de Estruturas de Dados Lista Encadeada. **Congresso Sul Brasileiro de Computação**, Criciúma, Santa Catarina, nov. 2012. Disponível em: <http://periodicos.unesc.net/sulcomp/article/view/800. Acesso em: 15 ago. 2019.>

SILVA FILHO, J. **Criação de uma Ferramenta Informativa sobre Teoria dos Grafos**. 2017. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação). Centro Universitário Luterano de Palmas, Palmas, Tocantins, 2017. Disponível em: <http://ulbra-to.br/bibliotecadigital/publico/home/documento/338>.