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
ASSOCIAÇÃO EDUCACIONAL LUTERANA DO BRASIL
José da Silva Filho

CRIAÇÃO DE UMA FERRAMENTA INFORMATIVA SOBRE TEORIA DOS GRAFOS

Palmas – TO
2017

José da Silva Filho

CRIAÇÃO DE UMA FERRAMENTA INFORMATIVA SOBRE TEORIA DOS 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 Fernando Luiz de Oliveira.

Palmas – TO

2017

José da Silva Filho

CRIAÇÃO DE UMA FERRAMENTA INFORMATIVA SOBRE TEORIA DOS 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 Fernando Luiz de Oliveira.

Aprovado em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. M.e Fernando Luiz de Oliveira

Orientador

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. M.e Fabiano Fagundes

Centro Universitário Luterano de Palmas – CEULP

_____

Profa. M.e Parcilene Fernandes de Brito

Centro Universitário Luterano de Palmas – CEULP

Palmas – TO

2017

RESUMO

A teoria dos grafos é um conhecimento antigo utilizado inicialmente por matemáticos para representar e resolver problemas do mundo real. Com o surgimento da computação, a teoria dos grafos ganhou novas utilidades, tais como: no desenvolvimento de jogos eletrônicos; representação de diagramas de UML, entre outras. Assim como na matemática, a computação busca representar e resolver problemas do mundo real. Dessa forma, durante a formação acadêmica de profissionais da computação são ministradas aulas para o aprendizado de teoria dos grafos. Nesse trabalho são abordados conceitos básicos da teoria dos grafos, como definição, elementos e tipos de grafos. Esses conceitos foram utilizados no decorrer do desenvolvimento de uma aplicação web com funcionalidades para desenhar grafos, informar o tipo do grafo desenhado e apresentar as informações que permitirão a inferência do tipo do grafo. Esta aplicação pode ser utilizada nas aulas de estrutura de dados dos cursos de Ciência da Computação e Sistemas de Informação do CEULP/ULBRA para auxiliar o professor no ensino da teoria de grafos.

Palavras-chave: Estruturas de Dados, Teoria dos Grafos.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CEULP/ULBRA - Centro Universitário Luterano de Palmas

IDE – Ambiente de Desenvolvimento Integrado

SVG - Gráficos Vetoriais Escaláveis

HTML - Linguagem de Marcação de Hipertexto

UML - Linguagem Unificada de Modelagem

SUMÁRIO

1 INTRODUÇÃO

Grafos são utilizados na computação para representar problemas do mundo real, mas não se trata apenas de representá-los e sim também de responder questões acerca desses problemas. Desta forma a teoria dos grafos tem grande importância na computação. Em cursos da área da computação a teoria dos grafos geralmente é ensinada na disciplina de estruturas de dados. Porém, grafos não estão presentes somente nas aulas de estruturas de dados, disciplinas como redes de computadores, linguagens formais, desenvolvimento de jogos, compiladores, entre outras, utilizam grafos de alguma forma.

Um grafo é representado por meio de um desenho. Muitas ferramentas podem ser usadas para desenhar grafos como, por exemplo, o Visio da Microsoft e o Draw.oi da JGraph. Embora essas ferramentas sirvam para desenhar grafos, elas não têm o foco em grafos e sim em diagramas de uma forma geral. Uma ferramenta bastante conhecida e que tem seu foco exclusivamente para grafos é o Graphviz da AT&T. Para desenhar no Graphviz é necessário inserir um código em uma linguagem própria da ferramenta. O Graphviz, o Visio e o Draw.io, não mostram informações teóricas sobre o grafo desenhado.

A terminologia utilizada na teoria dos grafos diz respeito aos elementos, às características e as propriedades presentes em desenhos de grafos, ou seja, a terminologia utilizada na teoria dos grafos permite o aprendizado sobre os diferentes tipos de grafos. Os conhecimentos sobre os tipos de grafos e seus elementos, características e propriedades se fazem necessários para o aprofundamento em assuntos mais avançados em teoria dos grafos como, por exemplo: os tipos de problemas que podem ser representados por determinado tipo de grafo; e os diversos algoritmos para responder perguntas acerca de problemas representados por grafos. Assim, este trabalho teve a finalidade de criar uma aplicação web que trabalhe exclusivamente com grafos de forma informativa.

O objetivo da aplicação que foi desenvolvida nesse trabalho é informar o usuário sobre os diferentes tipos de grafos, de forma que o usuário possa desenhar os grafos e sejam mostradas informações teóricas referentes ao grafo desenhado, como: o tipo do grafo desenhado; e os elementos, as características e as propriedades no desenho que o classificam de acordo com tal tipo de grafo.

A plataforma web foi escolhida para que a aplicação desenvolvida nesse trabalho seja de fácil acesso, possibilitando que seu uso seja feito em vários tipos de sistemas operacionais,

não sendo necessária a instalação de qualquer outro recurso computacional para o seu funcionamento. As tecnologias web que foram usadas no desenvolvimento da aplicação são HTML e JavaScript. Todo o processamento será feito no lado do cliente, assim sendo pode ser disponibilizado o *download* da aplicação para que seja usada de forma off-line no browser.

Portanto, esse trabalho tem como objetivo geral criar uma aplicação web informativa sobre teoria dos grafos para auxiliar os professores nas aulas de estruturas de dados. Nesse sentido objetiva-se especificamente criar funcionalidades na aplicação para:

- desenhar grafos;
- inferir e mostrar para o usuário em qual tipo de grafo se classifica o desenho feito por ele na aplicação;
- mostrar para o usuário as informações que permitiram a classificação do grafo de acordo com seu tipo.

Este trabalho está organizado da seguinte forma: na seção 2 é apresentado o referencial teórico com os principais conceitos para o desenvolvimento do trabalho. Na seção 3 são apresentados os materiais e os métodos utilizados para o desenvolvimento e o planejamento da execução do trabalho. A seção 4 apresenta o processo de desenvolvimento da aplicação e os resultados obtidos. Na seção 5 são apresentadas as considerações finais, e por fim as referências bibliográficas utilizadas.

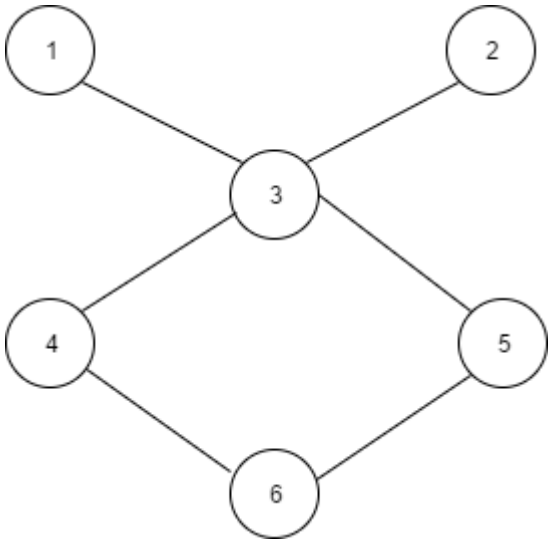
2 REFERENCIAL TEÓRICO

Nessa seção serão abordados os conceitos teóricos sobre grafos.

2.1 GRAFOS

Segundo Dovicchi (2007, p. 75), "grafos são estruturas matemáticas usadas para representar ideias ou modelos, por intermédio de uma ilustração, gráfico ou esquema". Na matemática grafos representam relacionamento entre dois ou mais conjuntos, esses conjuntos podem ser grandezas ou valores. Na computação grafos são usados para representar estruturas de redes de computadores, organização de dados entre outras aplicações. Na figura 1 é apresentada uma ilustração de um grafo.

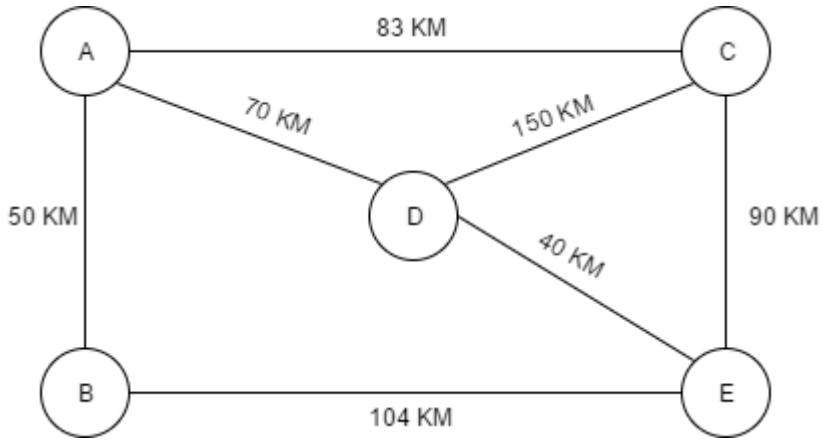
Figura 1 - Ilustração de um grafo



Na figura 1 é mostrado um grafo com valores numéricos. A estrutura de um grafo é composta por nós (vértices) e arcos (arestas), onde os nós podem se ligar a outro por arcos, na imagem cada círculo representa um nó, as linhas são os arcos e cada nó representa um valor numérico.

Para uma exemplificação prática, considere 5 cidades (A, B, C, D, E), que são ligadas por rodovias. As ligações entre as cidades são representadas por meio de um grafo mostrado na figura 2.

Figura 2 - Exemplo de grafo de ligação entre cidades



Na figura 2 o grafo representa todas as informações descritas de forma gráfica. Cada vértice representa uma cidade, as arestas representam as rodovias com seus respectivos tamanhos em KM. Dessa forma o grafo da Figura 2 é uma representação gráfica que facilita o entendimento das rotas entre as cidades, utilizando as informações descritas no grafo é possível responder questões como por exemplo qual o menor caminho entre duas cidades? Muitos dos problemas da computação buscam resolver esse tipo de questão por meio de aplicações, para isso muitas vezes um problema do mundo real é abstraído para a forma de um grafo que em seguida é transformado em código computacional para que o computador possa responder as questões referente ao problema.

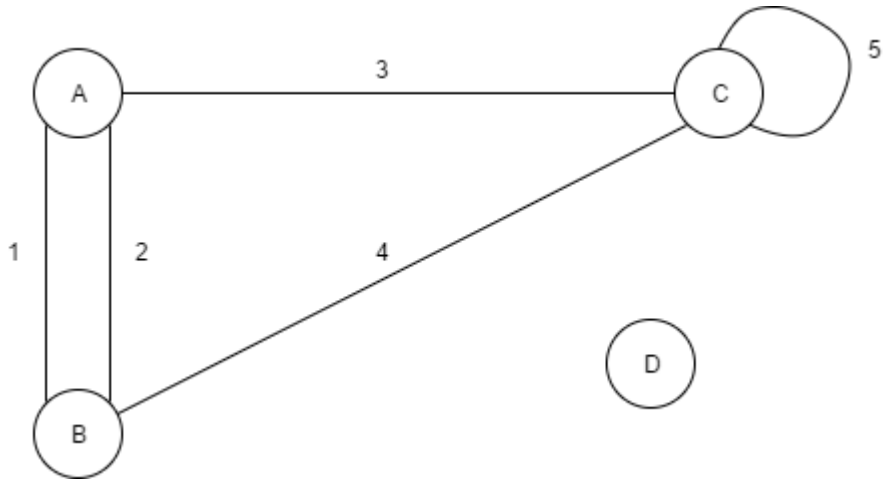
2.2 ELEMENTOS, CARACTERÍSTICAS E PROPRIEDADES DOS GRAFOS

Segundo Dovicchi (2007, p. 77), “A terminologia utilizada na teoria dos grafos compreende as definições dos elementos, características e propriedades dos grafos”. Os elementos de um grafo são:

- Laços: são arestas que as extremidades estão ligadas ao mesmo vértice.
- Vértices adjacentes: quando dois vértices estão ligados pela mesma aresta.
- Arestas paralelas: São duas arestas com as mesmas extremidades, ou seja, estão ligadas aos mesmos vértices.
- Vértice isolado: É um vértice que não é adjacente a nenhum outro vértice.
- Grau de um vértice: número de arestas que se ligam a um vértice.

A figura 3 apresenta exemplificação dos elementos de um grafo.

Figura 3 - Exemplificação dos elementos de um grafo



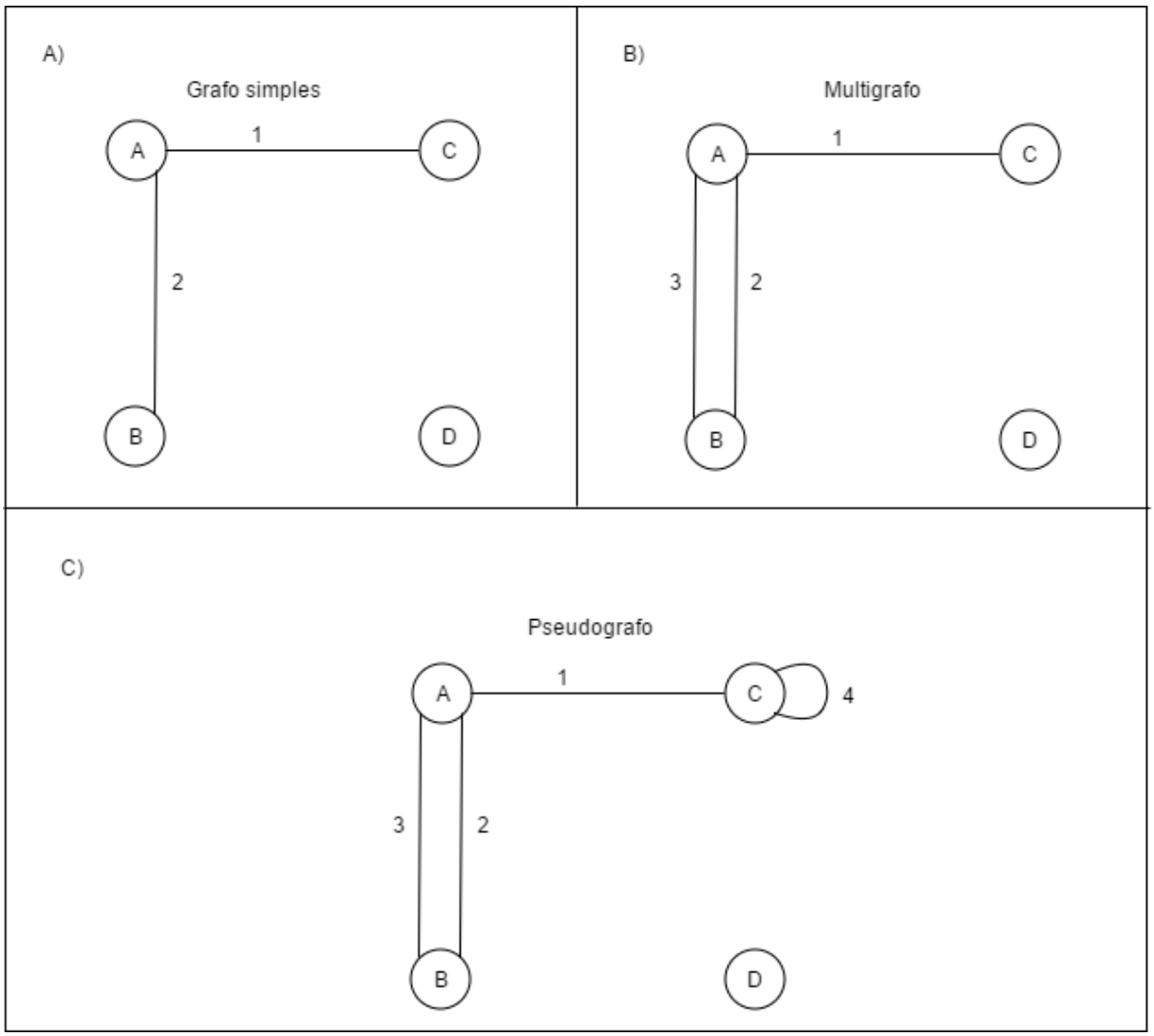
A figura 3 ilustra os seguintes exemplos: As arestas 1 e 2 são paralelas porque incidem nos mesmos vértices; o vértice D é um vértice isolado visto que não se liga a nenhum outro; o vértice C tem um laço, que se caracteriza por ser a origem e o destino de uma aresta; o vértice A é adjacente aos vértices C e B, o vértice C é adjacente aos vértices A e B e a si mesmo, o vértice B é adjacente aos vértices A e C; e os vértices A, B e C tem grau 3, o vértice D tem grau 0.

As características e propriedades de um grafo definem o tipo do grafo como simples, completo, conexo, desconexo entre outros tipos.

2.2.1 Grafo simples, multigrafo e pseudografo

O grafo simples é um grafo sem laços e sem arestas paralelas, já o pseudografo é um grafo que possui laços e arestas paralelas e o multigrafo é um grafo que possui arestas paralelas sem laços. A figura 4 apresenta exemplos de grafo simples, pseudografo e multigrafo.

Figura 4 - Exemplo de grafo simples, multigrafo e pseudografo



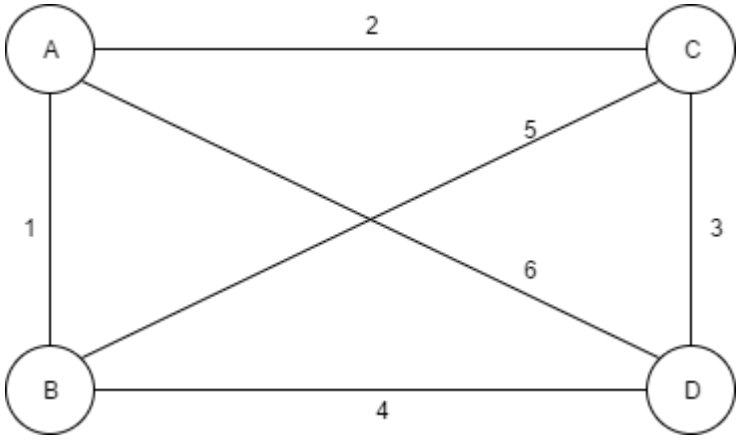
A figura 4 apresenta 3 tipos de grafos simples, multigrafo e pseudografo. No retângulo A) é mostrado um grafo simples sem laços e sem arestas paralelas, apenas um vértice isolado e vértices adjacentes. No retângulo B) o grafo simples ganhou mais uma aresta que liga o vértice A ao vértice B, passando a ter duas arestas paralelas as arestas 2 e 3, assim sendo o grafo deixa de ser simples e passa a ser um multigrafo. No retângulo C) o multigrafo ganha um laço no vértice C, assim sendo o grafo deixa de ser um multigrafo e passa a ser um pseudografo.

2.2.2 Grafo completo

O grafo completo é um grafo simples que tem a característica de que todo vértice é adjacente aos outros vértices. No grafo completo qualquer vértice possui arestas que se ligam a cada um dos demais vértices, existindo assim uma ligação por uma aresta entre quaisquer dois vértices no grafo. O total de arestas em um grafo completo é definido pela formula $\frac{n\,(n-1)}{2}$,

onde n é o total de vértices do grafo (GOLDBARG; GOLDBARG, 2012). A figura 5 mostra um exemplo de grafo completo.

Figura 5 - Exemplo de grafo completo

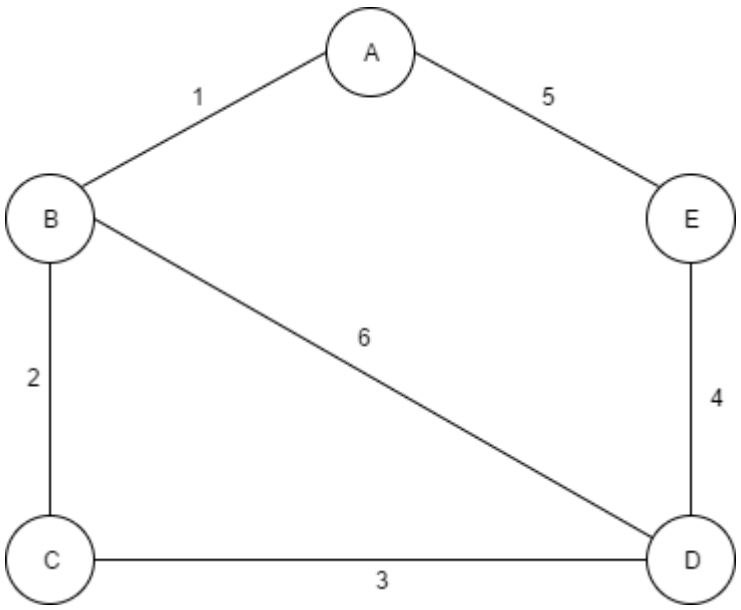


Na figura 5 todos os vértices são ligados entre si: o vértice A possui arestas que se ligam aos vértices B, C e D; o vértice B possui arestas que se ligam aos vértices A, C e D; o vértice C tem arestas que se ligam aos vértices A, B e D; e o vértice D tem arestas que se ligam aos vértices A, B e C.

2.2.3 Grafo conexo

Segundo Dovicchi (2007, p. 78), um grafo conexo é um grafo em que existe um caminho para quaisquer dois vértices, por meio de uma ou mais arestas. A figura 6 apresenta um exemplo de grafo conexo.

Figura 6 - Exemplo de grafo conexo

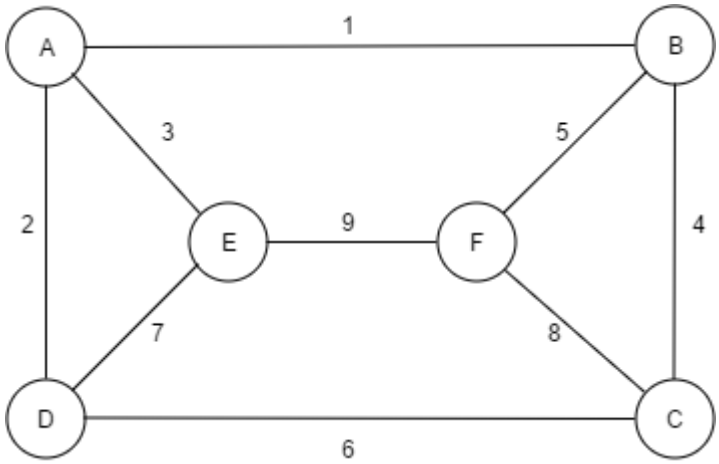


A figura 6 mostra um grafo conexo onde existe um caminho para qualquer par de vértices, como por exemplo: do vértice A ao vértice B existem dois caminhos, o primeiro pela aresta 1, o segundo é pelas arestas 5, 4, 3 e 2; e do vértice B ao vértice D existem três caminhos, o primeiro pelas arestas 2 e 3, o segundo pelas arestas 1, 5 e 4, o terceiro pela aresta 6.

2.2.4 Grafo regular

De acordo com Ascencio e Araújo (2010), um grafo é chamado regular quando todos os seus vértices possuem o mesmo grau. Todo grafo completo como já visto na figura 5 é um grafo regular, mas nem todo grafo regular é um grafo completo, a figura 7 apresenta um exemplo de grafo regular.

Figura 7 - Exemplo de grafo regular

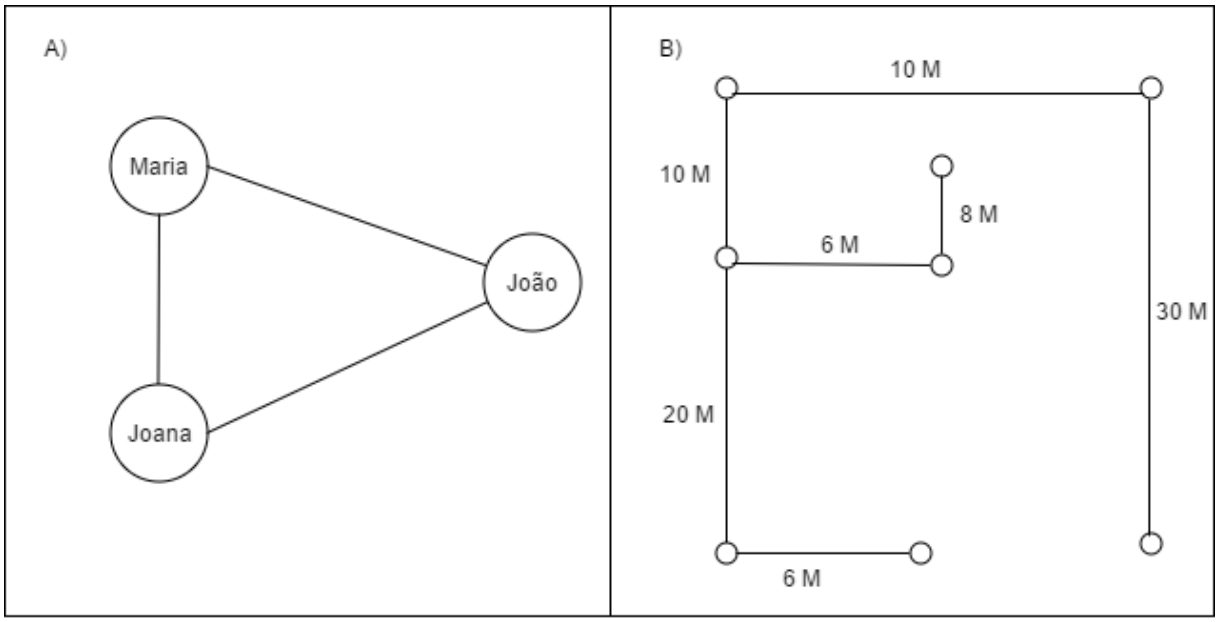


A figura 7 mostra um grafo regular com todos os vértices com grau 3. Todos os vértices do grafo da figura 7 tem 3 arestas ligadas a eles: o vértice A está ligado pelas arestas 1, 2 e 3; o vértice B está ligado pelas arestas 1, 4 e 5; o vértice C está ligado pelas arestas 4, 6 e 5; o vértice D está ligado pelas arestas 2, 6 e 7; o vértice E está ligado pelas arestas 3, 7 e 9; e o vértice F está ligado pelas arestas 5, 8 e 9.

2.2.5 Grafo rotulado

De acordo com Goldbarg e Goldbarg (2012), um grafo rotulado possui suas arestas ou vértices rotulados ou identificados com informações numéricas ou alfanuméricas. A figura 8 apresenta um exemplo de grafo rotulado.

Figura 8 - Exemplo de grafo rotulado

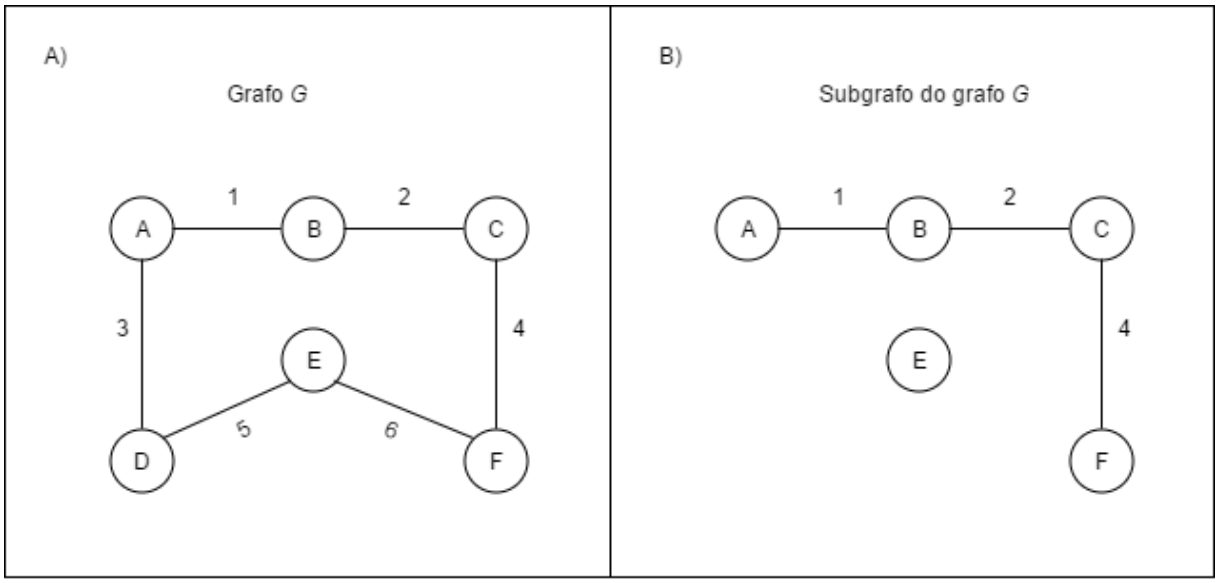


A figura 8 mostra dois grafos rotulados nos retângulos A) e B). No retângulo A) é mostrado um grafo rotulado com os vértices identificados por nomes de pessoas. No retângulo B) é mostrado um grafo rotulado que representa um esboço da planta baixa de um galpão em forma de grafo, as arestas contêm informações do comprimento das paredes em metros.

2.2.6 Subgrafo

Segundo Stein, Drysdale e Bogart (2013), um grafo *H* é *subgrafo* de um grafo *G* se o conjunto de vértices de *H* é um subconjunto do conjunto de vértices de *G* e o conjunto de arcos de *H* é um subconjunto do conjunto de arcos de *G*, ou seja, o grafo *H* está contido no grafo *G*. A figura 9 apresenta um exemplo de subgrafo.

Figura 9 - Exemplo de subgrafo

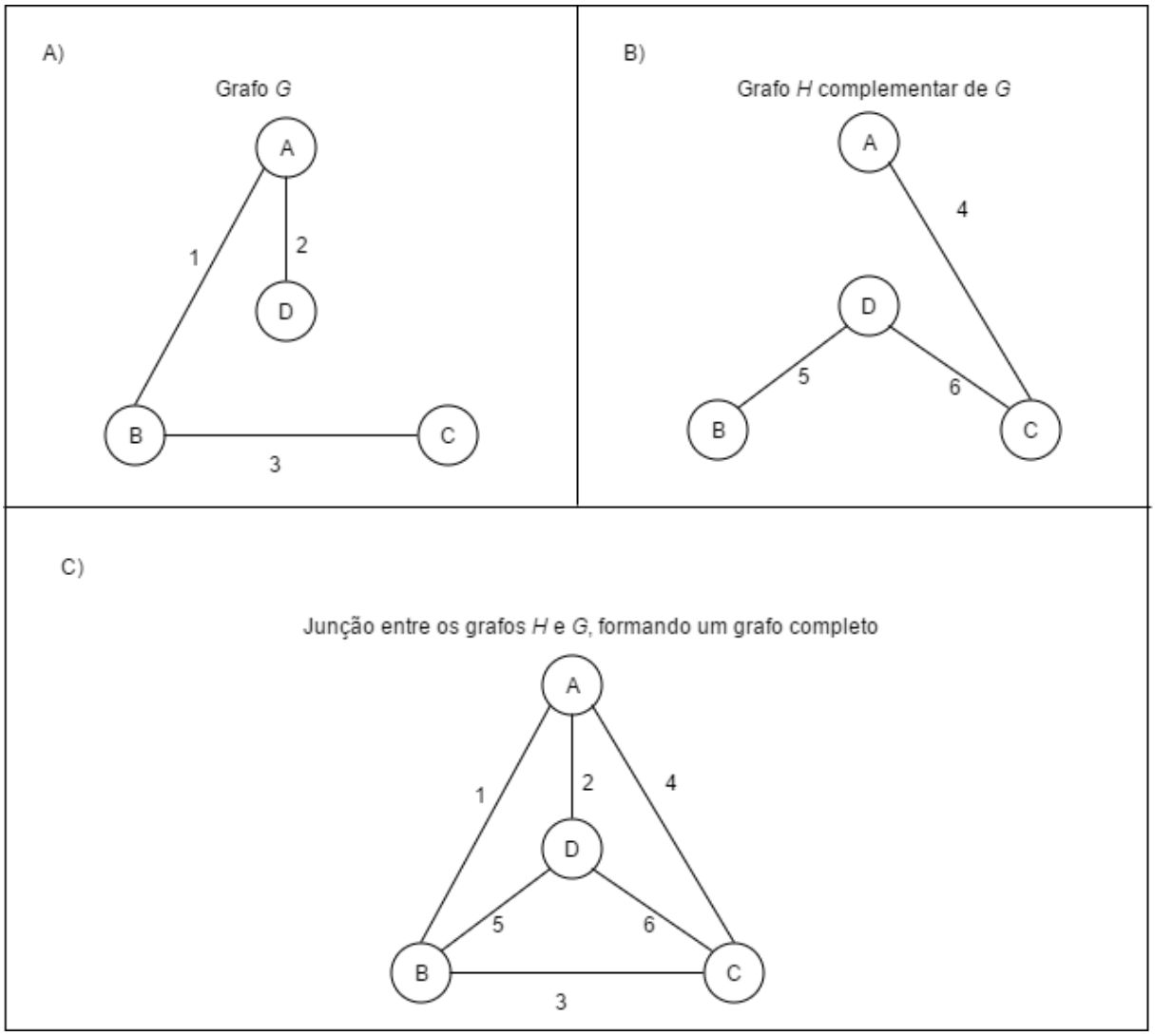


Na figura 9 são mostrados dois grafos nos retângulos A) e B). Todos os vértices e arestas do grafo do retângulo B) estão presentes no grafo do retângulo A), portanto, o grafo do retângulo B) é subgrafo do grafo do retângulo A).

2.2.7 Grafo complementar

Considere dois grafos *H* e *G*, para que *H* seja complementar de *G*, ele deve possuir algumas características relacionadas a *G*. O grafo *H* para ser complementar de *G* deve possuir o mesmo conjunto de vértices de *G* e não deve ter as ligações que já existem em *G*, ele deve ter apenas as ligações que faltam em *G*, ou seja, apenas as arestas são complementadas de tal forma que a junção de *H* e *G* formam um grafo completo como já visto na figura 5 (CAVALCANTE; SILVA, 2009). A figura 10 mostra um exemplo de grafo complementar.

Figura 10 - Exemplo de grafo complementar



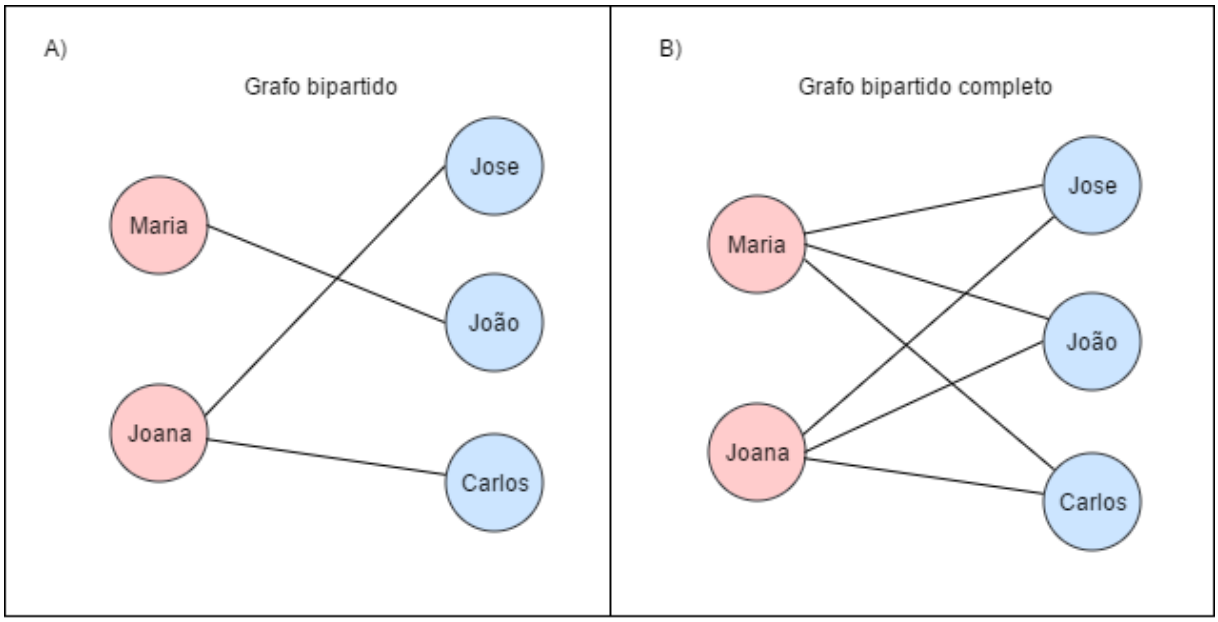
A figura 10 mostra três grafos nos retângulos A), B) e C). O grafo *H* do retângulo B) é um grafo complementar do grafo *G* do retângulo A), o grafo *H* tem as arestas 4, 5 e 6 que fazem

as ligações que não existem em *G*. O grafo completo do retângulo C) é formado pela junção dos grafos *H* e *G*, ele possui os vértices A, B, C e D que estão presentes tanto em *H* e *G*, e todas as ligações formadas pelas arestas de *H* e *G*.

2.2.8 Grafo bipartido

Um grafo bipartido é um grafo em que os vértices podem ser divididos em dois conjuntos e nenhuma aresta do grafo se liga a dois vértices do mesmo conjunto, ou seja, cada aresta liga vértices de conjuntos diferentes. Quando um grafo bipartido possui os vértices de um conjunto ligados a todos os vértices do outro conjunto, ele é chamado de grafo bipartido completo (ASCENCIO; ARAÚJO, 2010). A figura 11 mostra um exemplo de grafo bipartido e grafo bipartido completo.

Figura 11 - Exemplo de grafo bipartido e grafo bipartido completo

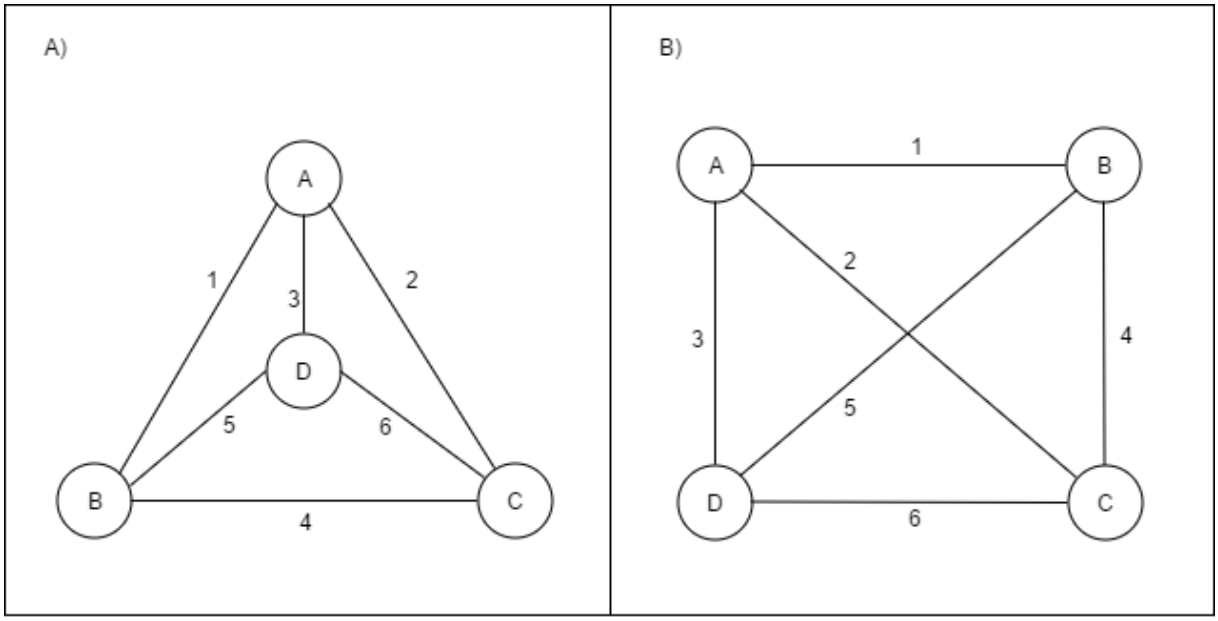


Na figura 11 são mostrados dois grafos nos retângulos A) e B). No retângulo A) é mostrado um grafo bipartido com dois conjuntos de vértices que são homens e mulheres, o conjunto de homens tem vértices na cor azul rotulados com nomes masculinos, e o conjunto das mulheres tem vértices na cor rosa rotulados com nomes femininos. No grafo do retângulo B) foram acrescentadas ligações ao grafo do retângulo A) para que todos os vértices do conjunto mulheres tenham ligações com todos os vértices do conjunto homens e vice-versa, dessa forma o grafo passou a ser um grafo bipartido completo.

2.2.9 Grafo isomorfo

Um grafo é isomorfo quando ele pode ser representado graficamente de duas ou mais formas diferentes, contendo os mesmos vértices, as mesmas arestas e as mesmas ligações entre os vértices, mas os rótulos dos vértices e arestas podem ser diferentes. Portanto dois grafos são isomorfos quando os dois são representados graficamente de formas diferentes, e os dois tem os mesmos vértices, as mesmas arestas e as mesmas ligações entre os vértices (DOVICCHI, 2007). A figura 12 mostra um exemplo de grafos isomorfos.

Figura 12 - Exemplo de grafos isomorfos



A figura 12 mostra dois grafos isomorfos nos retângulos A) e B). Os dois gráficos da figura possuem os mesmos vértices (A, B, C e D), as mesmas arestas (1, 2, 3, 4, 5, e 6) e as mesmas ligações entre os vértices: os vértices A e B são ligados pela aresta 1; os vértices A e C são ligados pela aresta 2; os vértices A e D são ligados pela aresta 3; os vértices B e C são ligados pela aresta 4; os vértices B e D são ligados pela aresta 5; e os vértices C e D são ligados pela aresta 6.

Nos dois grafos da figura 12 os rótulos são os mesmos, mas os rótulos de cada grafo poderiam ser diferentes, nesse caso é necessária uma função de mapeamento dos vértices e arestas para indicar quais vértices e quais arestas se correspondem nos dois grafos. Para exemplificar no caso de dois grafos com rótulos diferentes, considere que o grafo do retângulo A) é formado pelo conjunto de vértices {A1, B1, C1, D1} e pelo conjunto de arestas {1A, 2A, 3A, 4A, 5A, 6A}, e o grafo do retângulo B) é formado pelo conjunto de vértices {A2, B2, C3, D4} e pelo conjunto de arestas {1B, 2B, 3B, 4B, 5B, 6B}, para associar os vértices e arestas dos dois grafos foram criadas as funções:

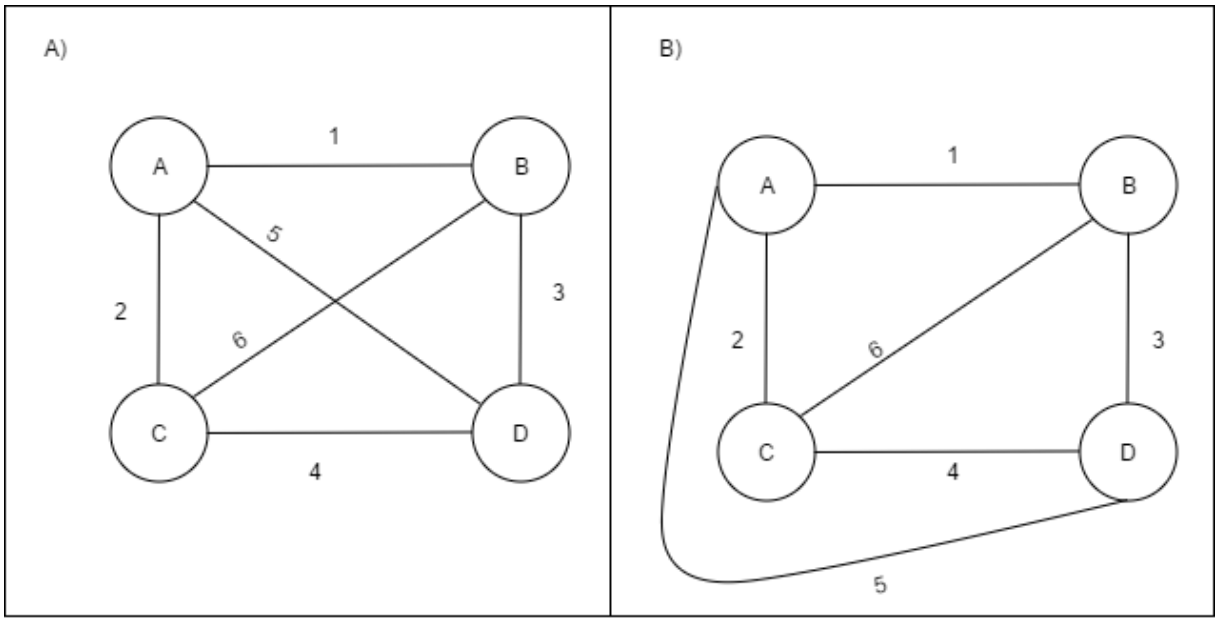
- $f1:$ A1 → A2; B1 → B2; C1 → C2; D1 → D2
- $f2:$ 1A → 1B; 2A → 2B; 3A → 3B; 4A → 4B; 5A → 5B; 6A → 6B

A função $f1$ faz as seguintes associações entre os vértices dos dois grafos, os vértices A1, B1, C1 e D1 do retângulo A) correspondem respectivamente aos vértices A2, B2, C2 e D2 do retângulo B). A função $f2$ faz as seguintes associações entre as arestas dos dois grafos, as arestas 1A, 2A, 3A,4A, 5A e 6A do retângulo A) correspondem respectivamente as arestas 1B, 2B, 3B,4B, 5B e 6B do retângulo B). Utilizando as funções $f1$ e $f2$ é possível verificar se os dois grafos possuem as mesmas ligações entre os vértices, portanto é possível verificar se dois grafos com rótulos diferentes são isomorfos.

2.2.10 Grafo planar

Um grafo é dito planar se não tiver intersecções entre suas arestas, ou seja, as arestas não podem se cruzar. Se um grafo tiver intersecções entre suas arestas ele ainda pode ser um grafo planar, se ele puder ser redesenhado de outra forma que contenha os mesmos vértices, as mesmas arestas e as mesmas ligações entre os vértices, mas sem intersecções entre suas arestas. O matemático polonês Kazimierz Kuratowski estabeleceu um teorema que determina quando um grafo não é planar, o teorema diz que um grafo não é planar se e somente se ele contiver um subgrafo homeomorfo a um grafo completo de cinco vértices ou a um grafo bipartido completo com conjuntos de três vértices. Dois grafos são homeomorfos se eles puderem ser obtidos do mesmo grafo por uma sequência de subdivisões elementares, uma subdivisão elementar é dividir uma aresta em duas com a adição de um vértice de tal forma que esse vértice fique adjacente aos dois vértices ligados pela aresta que foi dívida (ROSEN, 2009; SIMÕES-PEREIRA, 2013). A figura 13 mostra um exemplo de grafo planar.

Figura 13 - Exemplo de grafo planar



A figura 13 mostra dois grafos nos retângulos A) e B), ambos são o mesmo grafo desenhados de maneira diferente. O grafo do retângulo A) tem uma intersecção entre as arestas 5 e 6, mas apesar dele ter uma intersecção entre suas arestas, ele é um grafo planar porque ele pode ser desenhado de outra forma, ou seja, de uma forma que não tenham intersecções entre suas arestas, como mostra o retângulo B), onde a aresta 5 é mudada de posição para que não haja intersecção com a aresta 6.

2.2.11 Grafo direcionado

Segundo Goldbarg e Goldbarg (2012), um grafo é dito direcionado (dígrafo) quando as ligações entre os vértices têm um sentido, nesse caso as arestas têm uma seta para indicar o sentido das ligações e recebem o nome de arcos. Os arcos dos dígrafos representam relações unidirecional entre os vértices, ou seja, relações mútuas como a amizade entre duas pessoas não podem ser representadas por um arco ligando dois vértices, são necessários dois arcos em sentidos contrários ligando os dois vértices. A figura 14 apresenta um exemplo de grafo direcionado.

Figura 14 - Exemplo de grafo direcionado

Na figura 14 é mostrado um grafo direcionado mostrando quem é mais velho entre irmãos. Os vértices representam os irmãos Ana, Maria, João e José, as ligações entre os vértices mostram que: Ana é mais velha que João; João é mais velho que Maria; e Maria é mais velha que José.

A informações presentes nesse referencial teórico a respeito das propriedades, características e elementos presentes nos diversos tipos de grafos serão usadas para a criação de procedimentos na aplicação que verificam a existência de determinadas propriedades, características e elementos em um desenho de um grafo. Com os resultados dos procedimentos será possível determinar qual o tipo do grafo desenhado na aplicação.

3 MATERIAIS E MÉTODOS

Nessa seção são apresentados os materiais que serão usados e os procedimentos para o desenvolvimento do trabalho.

3.1 MATERIAIS

Para o entendimento e criação do referencial teórico foram realizadas pesquisas bibliográficas em livros, artigos, trabalhos de conclusão de curso, conteúdos on-line entre outros. Já para o desenvolvimento da aplicação foi utilizada a linguagem de marcação de hipertexto HTML e a linguagem de programação JavaScript. O elemento gráfico SVG do HTML foi utilizado como tela de desenho, para facilitar sua manipulação foi utilizado o framework SVG.js desenvolvido por Wout Fierens e licenciado nos termos da licença MIT. Com o elemento SVG é possível desenhar gráficos vetoriais que não perdem qualidade e nitidez ao serem redimensionados. Com JavaScript e HTML juntos é possível criar aplicações web interativas, dessa forma o usuário pode interagir com a aplicação.

A codificação foi feita utilizando a IDE Atom criada pela comunidade do GitHub. Atom é uma IDE *open source* com foco no desenvolvimento web, com suporte para várias tecnologias como HTML e JavaScript, possui diversos recursos que facilitam a codificação como por exemplo: auto complemento inteligente de código; interface personalizável; abertura de vários painéis permitindo a visualização na tela de vários arquivos de código ao mesmo tempo; e um painel que mostra a estrutura do projeto para facilitar a navegação nos arquivos e pastas do projeto.

3.2 PROCEDIMENTOS

Para o desenvolvimento desse trabalho foram realizadas duas etapas: desenvolvimento do módulo para desenhar grafos; e desenvolvimento do módulo para analisar os grafos desenhados. Esse projeto trabalha com os grafos simples, completo, conexo, planar, rotulado, complementar, bipartido, regular, direcionado, subgrafo, pseudografo e multigrafo. Durante todo o desenvolvimento do projeto foram realizadas reuniões com o orientador do projeto, professor M.e Fernando Luiz de Oliveira, para discussões a respeito do trabalho.

A primeira etapa foi o desenvolvimento do módulo para desenhar grafos. Para que a aplicação permita desenhar grafos foi criada uma tela de desenho usando o elemento SVG do HTML. O mouse e o teclado permitem que o usuário interaja com a aplicação, criando desenhos de grafos, alguns exemplos dessas interações para desenhar grafos são: ao realizar um duplo clique na tela de desenho é desenhado um vértice em forma de circunferência, o centro do

vértice é a posição do mouse no momento do duplo clique; e os rótulos dos vértices podem ser alterados dando um duplo clique em cima do vértice e em seguida alterando o conteúdo do rotulo digitando no teclado.

A segunda etapa foi o desenvolvimento do módulo para analisar os grafos desenhados. Para que a aplicação possa fazer a análise dos grafos desenhados foi implementada a estrutura de dados de um grafo em código JavaScript. Toda tela de desenho tem um grafo executando em código JavaScript, os dois módulos funcionam em conjunto, assim quando o usuário faz alguma alteração no grafo da tela de desenho essa alteração é refletida no grafo em código JavaScript. A análise do grafo desenhado não é feita no desenho em si, mas no grafo em código JavaScript que é o espelho do desenho do grafo em forma de código. O resultado da análise é mostrado em forma de texto na tela da aplicação, os resultados consistem em informações relacionadas ao grafo desenhado informando qual o tipo do grafo e as informações técnicas que permitiram tal inferência em relação ao tipo do grafo. Durante o desenvolvimento do módulo para analisar os grafos desenhados foram realizadas verificações manuais para conferir se os resultados da análise estão corretos.

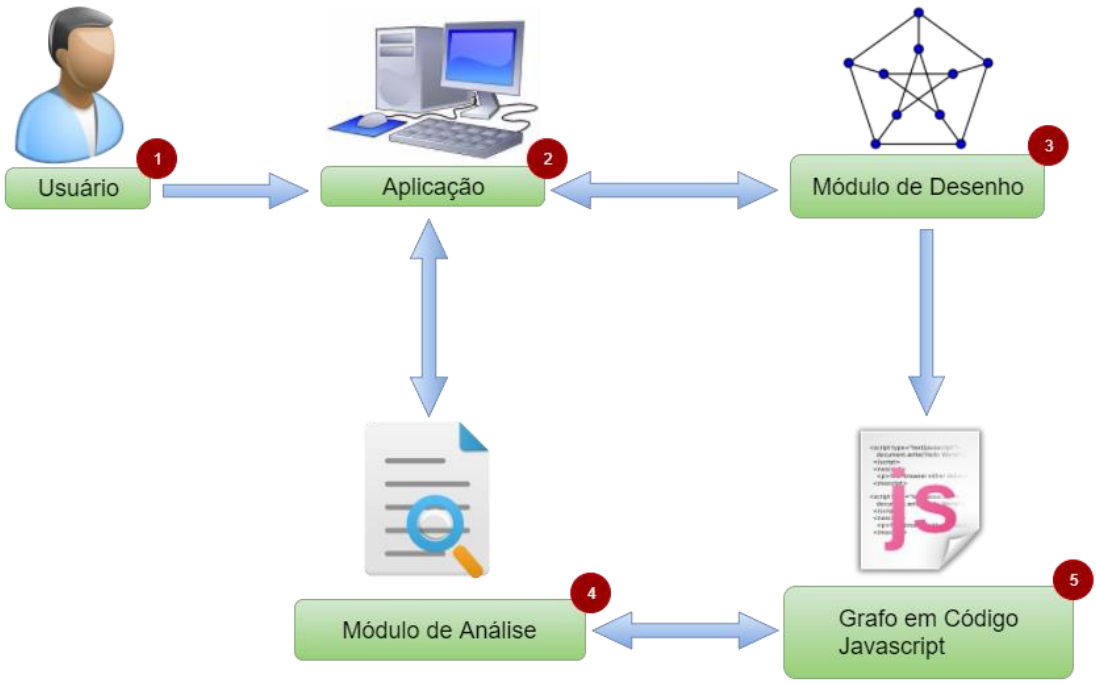
4 RESULTADOS E DISCUSSÃO

Nessa seção são apresentados os resultados obtidos com o desenvolvimento desse trabalho, bem como as soluções encontradas para cumprir com os objetivos do mesmo. Inicialmente será apresentada a arquitetura da aplicação, em seguida os dois módulos da aplicação na seguinte ordem: módulo de desenho e módulo de análise.

4.1 ARQUITETURA

Para um melhor entendimento da aplicação a seguir, figura 15, é apresentada a arquitetura da mesma.

Figura 15 - Arquitetura da aplicação

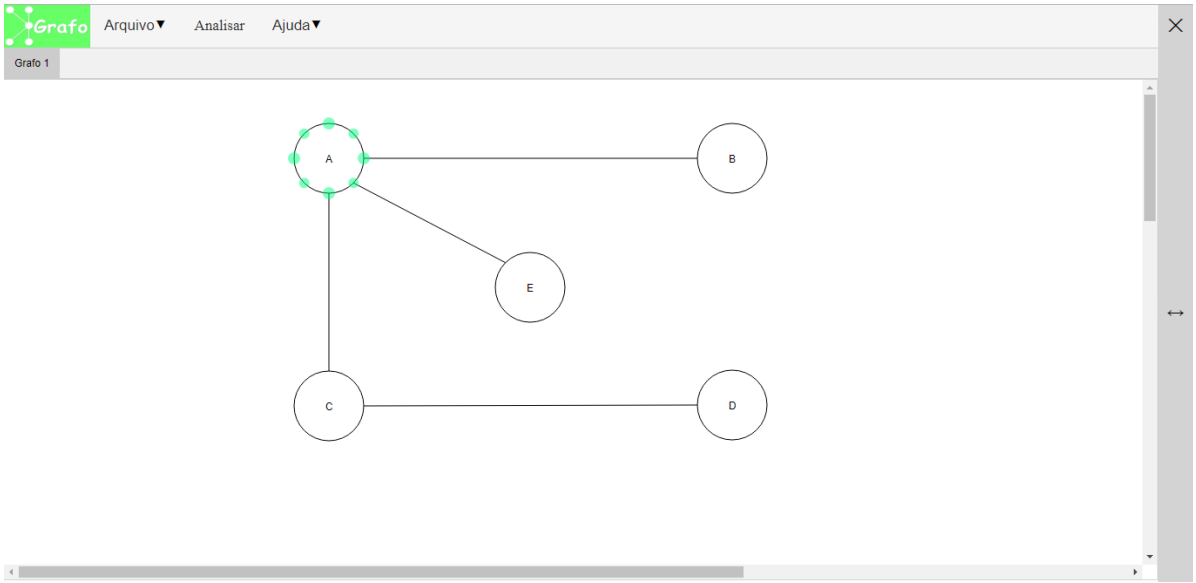


Na figura 15, o Usuário (1) interage com a Aplicação (2), que possui funcionalidades para desenhar grafos implementadas no Módulo de Desenho (3). Esta aplicação também tem funcionalidades para analisar o grafo desenhado implementadas no Módulo de Análise (4). A cada alteração realizada pelo usuário no desenho do grafo, o Módulo de Desenho reflete essas alterações do grafo em código Javascript (5) que, por sua vez, é objeto de análise do Módulo de Análise. As seções seguintes apresentam cada um destes módulos.

4.2 MÓDULO DE DESENHO

No módulo de desenho de grafos, foi criada uma área para desenho na tela da aplicação. A área para desenho é um elemento SVG do HTML. Para facilitar a manipulação do SVG foi utilizado o framework SVG.js. A interatividade do usuário com a área de desenho consiste em eventos de mouse e teclado, de forma que a aplicação fica ‘escutando’ eventos do mouse e teclado e a partir desses eventos acionados pelo usuário é que são criados, movimentados e excluídos os vértices e gerenciados as ligações entre os vértices. A figura 16, a seguir, apresenta a tela da aplicação com um grafo desenhado.

Figura 16 - Tela da aplicação com um grafo desenhado



Na figura 16 é mostrado um grafo desenhado na área de desenho da aplicação. No desenho, os vértices são representados em forma de circunferência, e as arestas são representadas por linhas que ligam essas circunferências. O vértice com o rótulo A tem pontos verdes. Esses pontos verdes são mostrados quando o ponteiro do mouse é colocado sobre o vértice, e desaparecem quando o ponteiro do mouse sair de cima do vértice. No caso, esses pontos verdes são utilizados para fazer as ligações entre os vértices. Assim, ao pressionar e segurar o botão esquerdo do mouse em cima de um desses pontos verdes e arrastar o mouse com o botão pressionado até outro ponto verde, que pode ser até mesmo no próprio vértice ou em outro vértice qualquer, e soltar o botão do mouse quando o cursor estiver em cima desse outro ponto verde é desenhada uma aresta ligando os vértices dos dois pontos verdes ou criando um laço se caso os dois pontos verdes pertencerem ao mesmo vértice.

Para que o usuário desenhe um vértice ele deve dá um duplo clique com o botão esquerdo do mouse em qualquer lugar da área de desenho que não tenha nada desenhado. O vértice é desenhado com o seu centro sendo o ponto onde foi realizado o duplo clique.

O usuário pode mudar a posição do vértice na área de desenho. Para isso deve-se pressionar e segurar o botão esquerdo do mouse em cima do vértice que se deseja mover. Então deve-se arrastar o mouse com o botão esquerdo pressionado até a nova posição escolhida. O centro do vértice é o ponto onde o mouse parou.

Outra funcionalidade permite nomear / renomear o rótulo do vértice. Para isto, basta dar um duplo clique com o botão esquerdo do mouse em cima do vértice que se deseja alterar o rótulo. Após o duplo clique, o rótulo se tornara editável, assim o usuário pode usar o teclado para alterar o conteúdo do rótulo.

Da mesma forma que é possível criar um vértice, é possível exclui-lo. Para isso, o usuário deve dar um clique com o botão direito do mouse em cima do vértice que ele deseja excluir e, em seguida, é aberta um menu de opções, no qual o usuário deve clicar na opção “Excluir”.

Cada tela de desenho tem uma estrutura de dados de um grafo em código Javascript. Essa estrutura tem uma lista de todos os vértices e uma lista de todas as arestas. Cada vértice e cada aresta é representado por um objeto Javascript. A figura 17 mostra a estrutura do grafo em código Javascript.

Figura 17 - Estrutura de dados do grafo em código Javascript

```
1  function Vertice(id){
2    this.id = id;
3    this.rotulo = "";
4    this.verticesAdjacentes = {};
5  }
6
7  function Aresta(id, idVertice1, idVertice2){
8    this.id = id;
9    this.rotulo = "";
10   this.extremidades = [idVertice1, idVertice2];
11 }
12
13 function Grafo(eGrafoDirigido){
14   this.eGrafoDirigido = eGrafoDirigido;
15   this.vertices = {};
16   this.arestas = {};
17 }
18
19 var grafo = new Grafo(false);
```

Na figura 17 é mostrado o código referente as funções da estrutura dos vértices, das arestas e do grafo. Nas linhas de 1 a 5 da Figura 17 é mostrada a estrutura de um objeto vértice. Esse objeto tem 3 atributos: na linha 2, o atributo *id* que armazena a identificação do vértice na estrutura do grafo; na linha 3, o atributo *rotulo* que armazena o nome do rótulo do vértice; e na linha 4, o atributo *verticesAdjacentes* que armazena uma lista com todos os vértices adjacentes. Neste caso, cada elemento da lista representa um vértice adjacente que armazena todas as arestas que fazem as ligações entre o vértice e o vértice adjacente.

Nas linhas de 7 a 11 é mostrada a estrutura de um objeto aresta. Esse objeto tem 3 atributos: na linha 8, o atributo *id* que armazena a identificação da aresta na estrutura do grafo; na linha 9, o atributo *rotulo* que armazena o nome do rótulo da aresta; e na linha 10, o atributo *extremidades* que armazena os ids dos vértices que são ligados pela aresta.

Por fim, nas linhas de 13 a 17 da figura 17 é mostrada a estrutura de um objeto grafo. Esse objeto tem 3 atributos: na linha 14, o atributo *eGrafoDirigido* que armazena o valor *true* caso o grafo seja um grafo dirigido e *false* caso o grafo seja um grafo não dirigido; na linha 15, o atributo *vertices* que armazena todos os objetos dos vértices do grafo; e na linha 16, o atributo *arestas* que armazena todos os objetos das arestas do grafo.

Na linha 19 é mostrada a instanciação de um objeto grafo. Essa instanciação é realizada na inicialização de uma nova área de desenho criada pelo usuário. Todas as alterações no desenho do grafo dessa nova área de desenho serão refletidas no objeto grafo que foi instanciado no momento de sua inicialização.

Quando o usuário realiza uma ação de adicionar ou excluir vértice ou aresta no desenho do grafo, a ação também acontece no objeto grafo em código Javascript. Na figura 18 são mostradas as funções para adicionar e excluir vértice e aresta no objeto grafo em código Javascript.

Figura 18 - funções de adição e exclusão de vértices e arestas

```
var adicionarVertice = function(grafo, idVertice){
  grafo.vertices[idVertice] = new Vertice(idVertice);
}

var adicionarAresta = function(grafo, idAresta, idVertice1, idVertice2){
  grafo.arestas[idAresta] = new Aresta(idAresta, idVertice1, idVertice2);
  if(grafo.vertices[idVertice1].verticesAdjacentes[idVertice2] == undefined){
    grafo.vertices[idVertice1].verticesAdjacentes[idVertice2] = [];
  }
  grafo.vertices[idVertice1].verticesAdjacentes[idVertice2].push(idAresta);
  if(grafo.eGrafoDirigido == false){
    if(grafo.vertices[idVertice2].verticesAdjacentes[idVertice1] == undefined){
      grafo.vertices[idVertice2].verticesAdjacentes[idVertice1] = [];
    }
    grafo.vertices[idVertice2].verticesAdjacentes[idVertice1].push(idAresta);
  }
}

var excluirVertice = function(grafo, idVertice){
    for(var verticeAdjacente in grafo.vertices[idVertice].verticesAdjacentes){
      for(var aresta in grafo.vertices[idVertice].verticesAdjacentes[verticeAdjacente]){
        excluirAresta(grafo, grafo.vertices[idVertice].verticesAdjacentes[verticeAdjacente][aresta]);
      }
    }
    delete grafo.vertices[idVertice];
}

var excluirAresta = function(grafo, idAresta){
  idVertice1 = grafo.arestas[idAresta].extremidades[0];
  idVertice2 = grafo.arestas[idAresta].extremidades[1];
  for(var aresta in grafo.vertices[idVertice1].verticesAdjacentes[idVertice2]){
    if(grafo.vertices[idVertice1].verticesAdjacentes[idVertice2][aresta] == idAresta){
      if(grafo.vertices[idVertice1].verticesAdjacentes[idVertice2].length<2){
        delete grafo.vertices[idVertice1].verticesAdjacentes[idVertice2];
        break;
      }else{
        grafo.vertices[idVertice1].verticesAdjacentes[idVertice2].splice(aresta,1);
        break;
      }
    }
  }
  if(grafo.eGrafoDirigido == false){
    for(var aresta in grafo.vertices[idVertice2].verticesAdjacentes[idVertice1]){
      if(grafo.vertices[idVertice2].verticesAdjacentes[idVertice1][aresta] == idAresta){
        if(grafo.vertices[idVertice2].verticesAdjacentes[idVertice1].length<2){
          delete grafo.vertices[idVertice2].verticesAdjacentes[idVertice1];
          break;
        }else{
          grafo.vertices[idVertice2].verticesAdjacentes[idVertice1].splice(aresta,1);
          break;
        }
      }
    }
  }
  delete grafo.arestas[idAresta];
}
```

Na figura 18 são mostradas quatros funções: uma para adicionar vértice, outra para adicionar arestas, outra para excluir vértices e outra para excluir arestas. A seguir, o código será explicado:

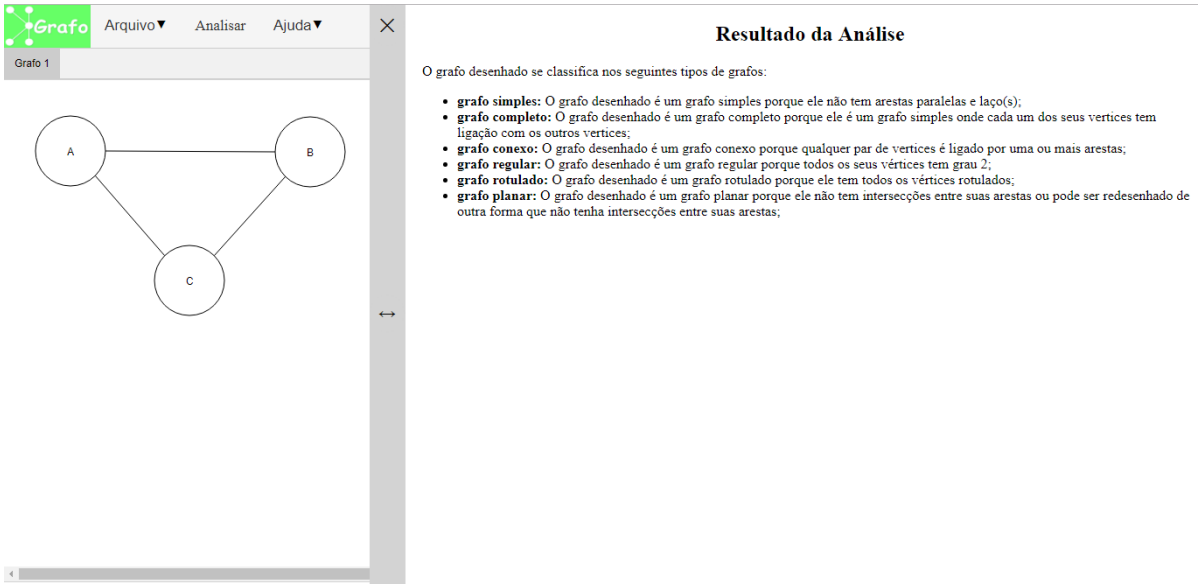
- Nas linhas de 1 a 3 é mostrada a função para adicionar vértice. Essa função recebe o grafo em que será adicionado o vértice e o *id* do novo vértice que será adicionado. Na linha 2 é instanciado um novo objeto vértice e esse novo objeto é adicionado ao grafo recebido como parâmetro.
- Nas linhas de 5 a 17 é mostrada a função para adicionar aresta. Essa função recebe como parâmetros o grafo em que será adicionada a aresta, o *id* da nova aresta que será adicionada e os *ids* dos vértices ligados pela nova aresta. Na linha 6 é instanciado um novo objeto aresta e esse novo objeto aresta é adicionado ao grafo recebido como parâmetro. Nas linhas de 7 a 10 o primeiro vértice passado como parâmetro adiciona o segundo vértice passado como parâmetro a sua lista de vértices adjacentes, nas linhas de 11 a 16 tem um condicional que verifica se o grafo é um grafo não dirigido, caso ele seja um grafo não dirigido o segundo vértice passado como parâmetro adiciona o primeiro vértice passado como parâmetro a sua lista de vértices adjacentes, esse condicional é necessário porque em grafos não dirigidos as ligações são simétricas.
- Nas linhas de 19 a 23 é mostrada a função para excluir vértices. Essa função recebe como parâmetros o grafo em que será excluído o vértice e o *id* do vértice que será excluído. Da linha 20 até a linha 24 tem um laço de repetição que percorre todos os vértices adjacentes ao vértice que será excluído, a cada interação é executado outro laço de repetição da linha 21 até a linha 23, onde percorre todas as arestas que ligam o referido vértice adjacente e o vértice que será excluído, a cada interação e feita uma chamada a função que excluir arestas para que a referida aresta seja excluída. Na linha 25 o vértice passado como parâmetro é excluído da lista de vértices do grafo passado como parâmetro. Portanto para excluir um vértice também é necessário excluir todas as arestas ligadas a ele.
- Nas linhas de 28 a 56 é mostrada a função para excluir arestas. Essa função recebe como parâmetro o grafo em que será excluída a aresta e o *id* da aresta que será excluída. Nas linhas 29 e 30 são obtidos os ids dos vértices das extremidades da aresta que será excluída. Da linha 31 até a linha 41 tem um laço de repetição que percorre todas as arestas contidas em um dos elementos da lista de vértices adjacentes do vértice obtido na linha 29, o elemento da lista que conténs as arestas é o elemento que representa o vértice adjacente obtido na linha 30, a cada interação é executado um condicional da linha 32 até a linha 40,

que verifica se a aresta que está sendo percorrida é a aresta que será excluída, caso seja, essa aresta é excluída da lista de arestas contidas no vértice adjacente. Da linha 42 até a linha 54 tem um condicional que verifica se o grafo recebido como parâmetro é um grafo não dirigido, caso ele seja um grafo não dirigido é executado um laço de repetição semelhante ao da linha 31 até a linha 41, com a diferença de que as arestas percorridas são de um elemento da lista de vértices adjacentes do vértice obtido na linha 30 e o elemento da lista de vértices adjacentes é o elemento que representa o vértice adjacente obtido na linha 29, esse condicional é necessário porque as ligações entre os vértices de um grafo não dirigido são simétricas. Na linha 55 a aresta é excluída da lista de arestas do grafo. Portanto para excluir uma aresta do grafo é necessário exclui-la da lista de arestas e também na lista de vértices adjacentes dos vértices que são ligados pela aresta.

4.3 MÓDULO DE ANÁLISE

O resultado da análise do grafo desenhado é mostrado em um painel retrátil que fica no lado direito da tela da aplicação. A figura 19 mostra o painel com o resultado da análise.

Figura 19 - Painel com o resultado da análise



Na figura 19 é mostrado o resultado da análise do grafo desenhado na tela de desenho. O resultado da análise é uma lista de tópicos com os tipos de grafos que o desenho se classifica, em cada tópico também é mostrado porque o grafo se classifica em determinado tipo. O módulo para analisar os grafos desenhados faz a análise em uma estrutura de dados de um grafo em código Javascript mostrada na figura 17.

4.3.1 Análise de existência de laços e arestas paralelas

Dois elementos de grafos que são importantes para análise de alguns tipos de grafo são laços e arestas paralelas. Antes de criar as funções que verificam se um grafo se classifica em determinado tipo foram criadas as funções que verificam a existência de laços e arestas paralelas, a figura 20 mostra as funções que verificam a existência de laços e arestas paralelas.

Figura 20 - Funções que verificam a existência de laços e arestas paralelas

```
1  var possuiLacos = function(grafo){
2    for(var aresta in grafo.arestas){
3      if(grafo.arestas[aresta].extremidades[0] == grafo.arestas[aresta].extremidades[1]){
4        return true;
5      }
6    }
7    return false;
8  }
9
10 var possuiArestasParalelas = function(grafo){
11     for(var vertice in this.vertices){
12       for(var verticeAdjacente in grafo.vertices[vertice].verticesAdjacentes){
13         if(grafo.vertices[vertice].verticesAdjacentes[verticeAdjacente].length>1){
14           return true;
15         }
16       }
17     }
18     return false;
19 }
```

Na figura 20 são mostradas duas funções que recebem um grafo como parâmetro, uma para verificar a existência de laços no grafo e outra para verificar a existência de arestas paralelas no grafo. Da linha 1 até a linha 8 é mostrada a função que verifica a existência de laços no grafo passado como parâmetro. A função possui um laço de repetição da linha 2 até a linha 6 que percorre todas as arestas do grafo. A cada iteração tem um condicional que verifica se as duas extremidades da aresta são iguais, ou seja, é verificado se as extremidades armazenam o mesmo id de um vértice, caso as extremidades sejam iguais é retornado o valor *true*. Na linha 7 é retornado o valor *false* caso nenhuma aresta possua as suas extremidades iguais.

Nas linhas de 10 até 19 é mostrada a função que verifica a existência de arestas paralelas no grafo passado como parâmetro. A função possui um laço de repetição da linha 11 até a linha 17 que percorre todos os vértices do grafo, a cada interação é executado um outro laço de repetição da linha 12 até a linha 16 que percorre todos os vértices adjacentes do vértice que está sendo percorrido pelo laço de repetição superior. A cada iteração do laço da linha 12 até a linha 16 é executado um condicional da linha 13 até a linha 15 que verifica se existe mais de uma

aresta fazendo a ligação entre o vértice e o seu vértice adjacente que está sendo percorrido, ou seja, é verificado se o tamanho do vetor que armazena as arestas que fazem a ligação entre o vértice e o seu vértice adjacente é maior que 1. Caso seja maior que 1 é porque existe mais de uma aresta já que cada elemento do vetor armazena uma aresta, caso exista mais de uma aresta fazendo a ligação entre o vértice e o seu vértice adjacente é retornado o valor *true*. Na linha 18 é retornado *false* caso nenhum vértice possua mais de uma aresta fazendo a ligação entre ele e um de seus vértices adjacentes.

4.3.2 Análise de grafo vazio

Para verificar se um grafo é um grafo vazio basta verificar se ele não possui nenhum vértice. A figura 21 mostra a função que analisa, se um grafo é um grafo vazio.

Figura 21 - Função que analisa, se um grafo é um grafo vazio

```
1  var eUmGrafoVazio = function(grafo){
2    var totalDeVertices = Object.keys(grafo.vertices).length;
3    if(totalDeVertices == 0){
4      return true;
5    }else{
6      return false;
7    }
8  }
```

Na figura 21 é mostrada uma função que recebe um grafo como parâmetro, para verificar se esse grafo é um grafo vazio. Na linha 2 é atribuído a uma variável o total de vértices do grafo passado como parâmetro. Da linha 3 até a linha 7 tem um condicional que verifica se o total de vértices é igual a zero, ou seja, é verificado se o grafo não possui vértices, caso o total de vértices seja igual a zero é retornado o valor *true*, caso contrário é retornado o valor *false*.

4.3.3 Análise de grafo nulo

Para verificar se um grafo é um grafo nulo basta verificar se ele não possui nenhuma aresta. A figura 22 mostra a função que analisa, se um grafo é um grafo nulo.

Figura 22 - Função que analisa, se um grafo é um grafo nulo

```
1  var eUmGrafoNulo = function(grafo){
2    var totalDeArestas = Object.keys(grafo.arestas).length;
3    if(totalDeArestas == 0){
4      return true;
5    }else{
6      return false;
7    }
8  }
```

Na figura 22 é mostrada a função que analisa se um grafo é um grafo nulo. Essa função recebe como parâmetro o grafo que será analisado. Na linha 2 é atribuído a uma variável o total de arestas do grafo passado como parâmetro. Da linha 3 até a linha 7 tem um condicional que verifica se o total de arestas é igual a zero, ou seja, é verificado se o grafo não possui arestas, caso o total de arestas seja igual a zero é retornado o valor *true*, caso contrário é retornado o valor *false*.

4.3.4 Análise de grafo simples

Para verificar se um grafo é um grafo simples basta verificar se ele não possui arestas paralelas e nem laços. A figura 23 mostra a função que analisa, se um grafo é um grafo simples.

Figura 23 - Função que analisa, se um grafo é um grafo simples

```
1  var eUmGrafoSimples = function(grafo){
2    if(possuiArestasParalelas(grafo) == false && possuiLacos(grafo) == false){
3      return true;
4    }else{
5      return false;
6    }
7  }
```

Na figura 23 é mostrada a função que analisa se um grafo é um grafo simples, essa função recebe como parâmetro o grafo que será analisado. Da linha 2 até a linha 6 é mostrado um condicional que verifica se o grafo passado como parâmetro não possui laços e nem arestas paralelas, utilizando as funções mostradas na figura 20, caso o grafo não possua arestas paralelas e também não possua laços é retornado o valor *true*, caso contrário é retornado o valor *false*.

4.3.5 Análise de multigrafo

Para verificar se um grafo é um multigrafo basta verificar se ele possui arestas paralelas e não possui laços. A figura 24 mostra a função que analisa, se um grafo é um multigrafo.

Figura 24 - Função que analisa, se um grafo é um multigrafo

```
1  var eUmMultigrafo = function(grafo){
2    if(possuiArestasParalelas(grafo) == true && possuiLacos(grafo) == false){
3      return true;
4    }else{
5      return false;
6    }
7  }
```

Na figura 24 é mostrada a função que analisa se um grafo é um multigrafo, essa função recebe como parâmetro o grafo que será analisado. Da linha 2 até a linha 6 é mostrado um condicional que verifica se o grafo passado como parâmetro possui arestas paralelas e não possui laços, utilizando as funções mostradas na figura 20, caso o grafo possua arestas paralelas e não possua laços é retornado o valor *true*, caso contrário é retornado o valor *false*.

4.3.6 Análise de pseudografo

Para verificar se um grafo é um pseudografo basta verificar se ele possui laços. A figura 25 mostra a função que analisa, se um grafo é um pseudografo.

Figura 25 - Função que analisa, se um grafo é um pseudografo

```
1  var eUmPseudografo = function(grafo){
2    if(possuiLacos(grafo) == true){
3      return true;
4    }else{
5      return false;
6    }
7  }
```

Na figura 25 é mostrada a função que analisa se um grafo é um pseudografo, essa função recebe como parâmetro o grafo que será analisado. Da linha 2 até a linha 6 é mostrado um condicional que verifica se o grafo passado como parâmetro possui laços, utilizando a função que verifica se um grafo possui laços mostrada na figura 20, caso o grafo possua laços é retornado o valor *true*, caso contrário é retornado o valor *false*.

4.3.7 Análise de grafo rotulado

Para verificar se um grafo é um grafo rotulado basta verificar se ele possui todos os seus vértices e/ou todas suas arestas com o rótulo preenchido. A figura 26 mostra as funções utilizadas para analisar se um grafo é um grafo rotulado.

Figura 26 - Funções para analisar, se um grafo é um grafo rotulado

```
var todasAsArestasPossuemRotulo = function(grafo){
  var totalDeArestas = Object.keys(grafo.arestas).length;
  if(totalDeArestas == 0){
    return false;
  }else{
    for(var aresta in grafo.arestas){
      if(grafo.arestas[aresta].rotulo == "" ){
        return false;
      }
    }
    return true;
  }
}

var todosOsVerticesPossuemRotulo = function(grafo){
  var totalDeVertices = Object.keys(grafo.vertices).length;
  if(totalDeVertices == 0){
    return false;
  }else{
    for(var vertice in grafo.vertices){
      if(grafo.vertices[vertice].rotulo == ""){
        return false;
      }
    }
    return true;
  }
}

var eUmGrafoRotulado = function (grafo){
  var verticesRotulados = todosOsVerticesPossuemRotulo(grafo);
  var arestasRotuladas = todasAsArestasPossuemRotulo(grafo);
  if(verticesRotulados == true && arestasRotuladas == true){
    return 3;
  }else if(verticesRotulados == true){
    return 2;
  }else if(arestasRotuladas == true){
    return 1
  }else{
    return 0;
  }
}
```

Na figura 26 são mostradas três funções que recebem um grafo como parâmetro. A figura 26 tem uma função para verificar se os rótulos de todas as arestas de um grafo estão preenchidos, outra para verificar se os rótulos de todos os vértices de um grafo estão preenchidos e outra para verificar se um grafo é um grafo rotulado.

Nas linhas de 1 a 13 da figura 26 é mostrada a função que verifica se os rótulos de todas as arestas de um grafo estão preenchidos. Na linha 2 é atribuído a uma variável o total de arestas. Da linha 3 até a linha 12 tem um condicional que verifica se o total de arestas é igual a zero, se não há arestas então não existe arestas com os rótulos preenchimentos, então caso o total de arestas seja igual a zero é retornado o valor *false*. Caso o total de arestas não seja zero é executado um laço de repetição da linha 6 até a linha 10, que percorre todas as arestas. A cada iteração é executado um condicional da linha 7 até a linha 9, que verifica se o rotulo da aresta que está sendo percorrida tem o rotulo não preenchido, caso o rotulo da aresta não esteja preenchido é retornado o valor false. Na linha 11 é retornado o valor *true* se não for encontrada uma aresta com o rotulo não preenchido.

A função mostrada nas linhas de 15 a 27 da figura 26, verifica se todos os vértices possuem o rótulo preenchido. A função que verifica se todos os vértices possuem seu rótulo preenchido segue o mesmo padrão da função que verifica se todas as arestas possuem seu rótulo preenchido, com as diferenças que: em vez de armazenar em uma variável o total de arestas é armazenado o total de vértices; em vez de verificar se o total de arestas é igual a zero é verificado se o total de vértices é igual a zero; e em vez de percorrer todas as arestas para verificar se alguma possui o rotulo não preenchido é percorrido todos os vértices para verificar se algum possui o rotulo não preenchido.

Nas linhas de 29 até 41 da figura 26 é mostrada a função que analisa se um grafo é um grafo rotulado. A análise é feita no grafo recebido como parâmetro. Na linha 30 é atribuída a uma variável o resultado da função que verifica se todos os vértices possuem seu rótulo preenchido e na linha 31 é atribuída a uma variável o resultado da função que verifica se todas as arestas possuem seu rótulo preenchido. Nas linhas de 32 a 40 é mostrado um condicional que verifica se todos os vértices e arestas possuem seu rótulo preenchido. Caso todos os vértices e arestas possuam seus rótulos preenchidos é retornado o número 3, caso contrário é executado outro condicional na linha 34 que verifica se todos os vértices possuem seu rótulo preenchido. Caso todos os vértices possuam seus rótulos preenchidos é retornado o número 2, caso contrário é executado um condicional na 36 que verifica se todas as arestas possuem seu rotulo preenchido. Caso todas as arestas possuam seus rótulos preenchidos é retornado o número 1, caso contrário é retornado o número 0. Devido a função retornar 4 valores diferentes, a análise

é mais detalhada no sentido de não apenas informar se o grafo é rotulado ou não, mas também de informar se o grafo é rotulado nos vértices e arestas ou apenas nos vértices ou apenas nas arestas.

4.3.8 Análise de grafo regular

Para verificar se um grafo é um grafo regular basta percorrer todos os vértices e verificar se todos possuem o mesmo grau. A figura 27 mostra as funções utilizadas para analisar se um grafo é um grafo regular.

Figura 27 - Funções para analisar, se um grafo é um grafo regular

```
1  var qualOGrauDoVertice = function(vertice){
2    var grau = 0;
3    for(var verticeAdjacente in vertice.verticesAdjacentes){
4      grau += vertice.verticesAdjacentes[verticeAdjacente].length;
5    }
6    return grau;
7  }
8
9  var eUmGrafoRegular = function(grafo){
10   var grauDoVerticeAnterior = -1;
11   for(var vertice in grafo.vertices){
12     var grauDoVertice = qualOGrauDoVertice(grafo.vertices[vertice]);
13     if(grauDoVerticeAnterior != -1 && grauDoVerticeAnterior != grauDoVertice){
14       return -1;
15     }
16     grauDoVerticeAnterior = grauDoVertice;
17   }
18   return grauDoVerticeAnterior;
19 }
```

Na figura 27 são mostradas duas funções, uma que retorna o grafo de um vértice passado como parâmetro e outra que analisa se um grafo passado como parâmetro é um grafo regular. Da linha 1 até a linha 7 é mostrada a função que retorna o grau de um vértice, na linha 2 é declarada uma variável que armazena o grau do vértice que será retornado, inicialmente a variável recebe o valor zero, já que zero é o grau mínimo de um vértice (um vértice isolado). Da linha 3 até a linha 5 tem um laço de repetição que percorre todos os vértices adjacentes do vértice recebido como parâmetro, a cada interação na linha 4 é somado ao valor da variável que armazena o grau do vértice, a quantidade de arestas que ligam o vértice recebido como

parâmetro ao seu vértice adjacente que está sendo percorrido pelo laço de repetição. Na linha 6 é retornado o grau do vértice.

Da linha 9 até a linha 19 da figura 27 é mostrada a função que analisa se um grafo recebido como parâmetro é um grafo regular. Na linha 10 é declarada uma variável que armazenara o valor do grau do vértice percorrido anteriormente pelo laço de repetição. Inicialmente a variável recebe o valor -1, pois atribuir inicialmente o valor -1 a variável é uma forma de verificar durante a execução do laço de repetição qual é a primeira interação do laço de repetição, já que não existe vértice com o grau menos um. Da linha 11 até a linha 17 tem um laço de repetição que percorre todos os vértices do grafo passado como parâmetro. A cada iteração na linha 12 é atribuída a uma variável o valor do grau do vértice que está sendo percorrido. Da linha 13 até a linha 15 tem um condicional que retorna o número menos um, caso não seja a primeira iteração do laço de repetição e também caso o grau do vértice percorrido anteriormente seja diferente do grau do vértice que está sendo percorrido. Na linha 16 a variável que armazena o valor do grau do vértice percorrido anteriormente recebe o valor do grau do vértice que está sendo percorrido. Se a função retornar o número menos um é porque o grafo não é um grafo regular, se ela retornar qualquer outro número diferente de menos um é porque o grafo é um grafo regular e o número retornado é o valor do grau de todos os vértices do grafo.

4.3.9 Análise de grafo completo

Para analisar se um grafo é um grafo completo basta verificar se ele é um grafo simples e em seguida calcular o total de arestas utilizando a formula que define o total de arestas de um grafo completo e em seguida verificar se o resultado do cálculo é igual ao o número de arestas do grafo em análise. A figura 28 mostra a função que analisa se um grafo é um grafo completo.

Figura 28 - Função que analisa, se um grafo é um grafo completo

```
1  var eUmGrafoCompleto = function(grafo){
2    if(eUmGrafoSimples(grafo) == true){
3      var totalDeVertices = Object.keys(grafo.vertices).length;
4      var totalDeArestas = Object.keys(grafo.arestas).length;
5      var numeroDeArestasGrafoCompleto = (totalDeVertices*(totalDeVertices - 1))/2;
6      if(numeroDeArestasGrafoCompleto == totalDeArestas){
7        return true;
8      }else{
9        return false;
10     }
11   }else{
12     return false;
13   }
14 }
```

Na figura 28 é mostrada a função que analisa se um grafo é um grafo completo. Essa função recebe como parâmetro o grafo que será analisado. Nas linhas de 2 a 13 tem um condicional que verifica se o grafo passado como parâmetro é um grafo simples, caso o grafo não seja um grafo simples é retornado o valor *false*. Caso o grafo seja um grafo simples, nas linhas 3 e 4 são atribuídos a variáveis o total de vértices e o total de arestas do grafo em análise. Na linha 5 é atribuído a uma variável o total de arestas de um grafo completo que possua o mesmo total de vértices do grafo em análise. Nas linhas de 6 a 10 tem um condicional que retorna o valor *true*, caso o total de arestas de um grafo completo com o mesmo número de vértices do grafo em análise seja igual ao total de arestas do grafo em análise, caso contrário o condicional retorna o valor *false*.

4.3.10 Análise de grafo conexo

Para analisar se um grafo é um grafo conexo basta verificar se a partir de um vértice qualquer do grafo existe pelo menos um caminho que o liga a cada um dos outros vértices do grafo. A figura 29 mostra as funções utilizadas para analisar, se um grafo é um grafo conexo.

Figura 29 - Funções para analisar, se um grafo é um grafo conexo

```
1  var osVerticesSaoConectados = function(grafo, vertice1, vertice2, verticesVerificados){
2    var retorno = false;
3    if(vertice1.verticesAdjacentes[vertice2.id] == undefined){
4      verticesVerificados[vertice1.id] = true;
5      for(var verticeAdjacente in vertice1.verticesAdjacentes){
6        if(verticesVerificados[verticeAdjacente] != true){
7          retorno = osVerticesSaoConectados(grafo, grafo.vertices[verticeAdjacente], vertice2, verticesVerificados);
8        }
9      }
10     return retorno;
11   }else{
12     return true;
13   }
14 }
15
16 var eUmGrafoConexo = function(grafo){
17   var idsVertices = Object.keys(grafo.vertices);
18   var vertice1 = grafo.vertices[idsVertices[0]];
19   for(var i=1; i<idsVertices.length; i++){
20     var vertice2 = grafo.vertices[idsVertices[i]];
21     var verticesVerificados = {};
22     var saoVerticesConectados = osVerticesSaoConectados(grafo, vertice1, vertice2, verticesVerificados);
23     if(saoVerticesConectados == false){
24       return false;
25     }
26   }
27   return true;
28 }
```

Na figura 29 são mostradas duas funções, uma que verifica se existe um caminho que conecta dois vértices quaisquer do grafo e outra que analisa se um grafo é um grafo conexo. Nas linhas de 1 até 14 é mostrada a função recursiva que verifica se existe um caminho ligando dois vértices quaisquer, a função recebe como parâmetro um grafo, dois vértices quaisquer desse grafo e uma lista de vértices. Na linha 2 é declarada uma variável de retorno que inicialmente recebe o valor *false*. Da linha 3 até a linha 13 tem um condicional que retorna o valor *true* caso os dois vértices passados como parâmetro sejam adjacentes, caso contrário, na linha 4 o primeiro vértice passado como parâmetro é adicionado à lista de vértices. Da linha 5 até a linha 9 tem um laço de repetição que percorre a lista de vértices adjacentes do primeiro vértice passado como parâmetro, a cada iteração tem um condicional da linha 6 até a linha 8 que verifica se o vértice adjacente que está sendo percorrido está na lista de vértices. Caso não esteja na lista de vértices na linha 7 é atribuída a variável de retorno a chamada recursiva da função, essa nova chamada da função recebe os mesmos parâmetros com exceção do primeiro vértice passado como parâmetro que é trocado pelo vértice adjacente que está sendo percorrido. Na linha 10 é retornado o valor armazenado na variável de retorno depois da execução do laço de repetição. A função percorre recursivamente o grafo utilizando as ligações dos vértices adjacentes do primeiro vértice passado como parâmetro e testa se o vértice é adjacente ao segundo vértice passado como parâmetro caso seja é porque existe um caminho entre os dois vértices passados na chamada inicial da função. O condicional que verifica se o vértice está na

lista de vértices é necessário para que a função não fique chamado ela mesmo infinitamente, ou seja, o vértice que já foi testado a adjacência não será testado de novo.

Nas linhas de 16 a 28 da figura 29 é mostrada a função que analisa se um grafo passado como parâmetro é um grafo conexo. Na linha 17 é atribuída a uma variável uma lista de todos os ids dos vértices do grafo. Na linha 18 é atribuída a uma variável o vértice que será verificado se existe um caminho que liga ele a cada um dos outros vértices do grafo. Da linha 19 a 26 tem um laço de repetição que percorre todos os outros vértices. A cada interação na linha 21 é criada uma lista vazia de vértices. Na linha 22 é atribuída a uma variável o retorno da função que verifica se dois vértices são ligados por um caminho, passando como parâmetros o grafo, o vértice que está sendo verificado se existe um caminho entre ele e os outros vértices, o vértice que está sendo percorrido pelo laço de repetição e a lista vazia de vértices. Nas linhas de 23 a 25 tem um condicional que retorna o valor *false* caso não exista um caminho entre os dois vértices. Na linha 27 é retornado o valor *true* se nenhuma vez durante a execução do laço de repetição a função que verifica se dois vértices são ligados por um caminho retornar o valor *false*.

4.3.11 Geração de grafo complementar

Para gerar um grafo complementar de um grafo basta verificar se o grafo é um grafo simples, em seguida criar um novo grafo e adicionar a ele todos os vértices existentes no grafo simples, em seguida deve-se adicionar ao novo grafo apenas as arestas que faltam para o grafo simples se tornar um grafo completo. A figura 30 mostra a função que cria o grafo complementar de um grafo simples.

Figura 30 - Função que cria o grafo complementar de um grafo simples

```
1  var criarGrafoComplementar = function(grafo){
2    if(eUmGrafoSimples(grafo) == true){
3      var grafoComplementar = new Grafo(grafo.eGrafoDirigido);
4      var prefixoIDAresta = "aresta";
5      var sufixoIDAresta = 1;
6      for(var idVertice in grafo.vertices){
7        adicionarVertice(grafoComplementar, idVertice);
8      }
9      var idsVertices = Object.keys(grafo.vertices);
10     for(var i = 0 ; i<idsVertices.length; i++){
11       for(var j = i+1 ; j<idsVertices.length; j++){
12         if(grafo.vertices[idsVertices[i]].verticesAdjacentes[idsVertices[j]] == undefined){
13           var idAresta = prefixoIDAresta+sufixoIDAresta;
14           sufixoIDAresta++;
15           adicionarAresta(grafoComplementar, idAresta, idsVertices[i], idsVertices[j]);
16         }
17       }
18     }
19     return grafoComplementar;
20   }else{
21     return null;
22   }
23 }
```

Na figura 30 é mostrada uma função que cria um grafo complementar de um grafo simples, essa função recebe como parâmetro o grafo que será gerado o seu grafo complementar. Da linha 2 até a linha 22 tem um condicional que verifica se o grafo passado como parâmetro é um grafo simples, caso não seja um grafo simples é retornado o valor *null*. Caso o grafo seja um grafo simples na linha 3 é criado e atribuído a uma variável um novo grafo, da linha 6 até a linha 8 tem um laço de repetição que percorre todos os vértices do grafo passado como parâmetro, a cada iteração o vértice que está sendo percorrido é adicionado ao novo grafo, ou seja o novo grafo terá os mesmos vértices do grafo passado como parâmetro. Da linha 10 até a linha 18 tem dois laços de repetição aninhados, com os dois laços aninhados é possível acessar todas as combinações de par de vértices do grafo passado como parâmetro. Assim sendo dentro do laço interno tem um condicional da linha 12 até a linha 16 que verifica se os vértices que compõe o par de vértices são adjacentes, caso não sejam adjacentes é adicionada uma nova aresta ao novo grafo ligando os dois vértices, ou seja são adicionadas ao novo grafo apenas as arestas que faltam no grafo passado como parâmetro para se tornar um grafo completo. Na linha 19 é retornado o novo grafo que após a execução dos dois laços de repetição aninhados se tornou o grafo complementar do grafo passado como parâmetro.

4.3.12 Geração de subgrafos

Para gerar todos os subgrafos de um grafo basta gerar todas as possibilidades de combinações entre vértices e arestas presentes no grafo, sendo que em cada combinação não pode haver vértices ou arestas repetidas. A figura 31 mostra a função que cria todos os subgrafos de um grafo qualquer.

Figura 31 - Funções para gerar todos os subgrafos de um grafo qualquer

```
var gerarSubgrafo = function(grafo, possibilidade){
  var subgrafo = new Grafo(grafo.eGrafoDirigido);
  for(var i = 0; i < possibilidade.vertices.length; i++){
    adicionarVertice(subgrafo, possibilidade.vertices[i]);
  }
  for(var i = 0; i < possibilidade.arestas.length; i++){
    var aresta = grafo.arestas[possibilidade.arestas[i]];
    adicionarAresta(subgrafo, aresta.id, aresta.extremidades[0], aresta.extremidades[1]);
  }
  return subgrafo;
}

var criarSubgrafos = function(grafo){
  var possibilidadesSemArestas = [];
  var subgrafos = [];
  var idsVertices = Object.keys(grafo.vertices);
  var possibilidadeSemArestas = {vertices: [], arestas: []};
  possibilidadesSemArestas.push(possibilidadeSemArestas);
  for(var i = 0; i < idsVertices.length; i++){
    var inicioK = possibilidadesSemArestas.length;
    possibilidadeSemArestas = {vertices: [], arestas: []};
    possibilidadeSemArestas.vertices.push(idsVertices[i]);
    possibilidadesSemArestas.push(possibilidadeSemArestas);
    for(var j = i+1; j < idsVertices.length; j++){
      var fimK = possibilidadesSemArestas.length;
      for(var k = inicioK; k < fimK; k++){
        possibilidadeSemArestas = {vertices: possibilidadesSemArestas[k].vertices.concat(), arestas: possibilidadesSemArestas[k].arestas.concat()};
        possibilidadeSemArestas.vertices.push(idsVertices[j]);
        possibilidadesSemArestas.push(possibilidadeSemArestas);
      }
    }
  }
  possibilidadesSemArestas = quickSort(false, possibilidadesSemArestas);
  for(var i = 0; i < possibilidadesSemArestas.length; i++){
    var subgrafoSemArestas = gerarSubgrafo(grafo, possibilidadesSemArestas[i]);
    subgrafos.push(subgrafoSemArestas);
    var possibilidadesComArestas = possibilidadesDeSubligacoesDosVertices(grafo, possibilidadesSemArestas[i]);
    possibilidadesComArestas = quickSort(true, possibilidadesComArestas);
    for(var j = 0; j < possibilidadesComArestas.length; j++){
      var subgrafoComArestas = gerarSubgrafo(grafo, possibilidadesComArestas[j]);
      subgrafos.push(subgrafoComArestas);
    }
  }
  return subgrafos;
}
```

Na figura 31 são mostradas duas funções, uma que gera um grafo a partir de uma lista de vértices e arestas e outra que gera todos os subgrafos de um grafo qualquer. Nas linhas de 1 até 11 é mostrada a função que gera um grafo a partir de uma lista de vértices e arestas. A função recebe como parâmetro um grafo e a lista contendo os vértices e arestas, a função retornar o grafo gerado, o grafo que é gerado é um subgrafo do grafo passado como parâmetro, ou seja os vértices e arestas contidos na lista estão presentes no grafo passado como parâmetro. Na linha 2 é declarada uma variável que recebe uma instanciação de um novo grafo. Da linha 3 até a linha 5 tem um laço de repetição que percorre todos os vértices da lista, a cada iteração na linha 4 é adicionado ao novo grafo o vértice que está sendo percorrido. Da linha 6 até a linha 9 tem um laço de repetição que percorre todas as arestas da lista, a cada iteração na linha 8 é adicionada ao novo grafo a aresta que está sendo percorrida. Na linha 10 o novo grafo é retornado.

Nas linhas de 13 a 45 da figura 31 é mostrada a função que gera todos os subgrafos de um grafo passado como parâmetro. Na linha 14 é declara uma variável para armazenar todas as possibilidades de combinações entre os vértices, inicialmente essa variável recebe um vetor vazio. Na linha 15 é declarada uma variável para armazenar todos os subgrafos do grafo passado como parâmetro, inicialmente essa variável recebe um vetor vazio. Nas linhas 17 e 18 é criada e adicionada a lista de possibilidades a primeira possibilidade que está presente em qualquer grafo que é uma possibilidade sem vértices e sem arestas, ou seja um grafo vazio. Da linha 19 a 32 tem três laços de repetição aninhados, com os três laços aninhados é possível acessar todas as possibilidades de combinações entre os vértices do grafo passado como parâmetro, assim sendo durante as iterações dos laços de repetição as possibilidades de combinações são adicionadas a lista de possibilidades.

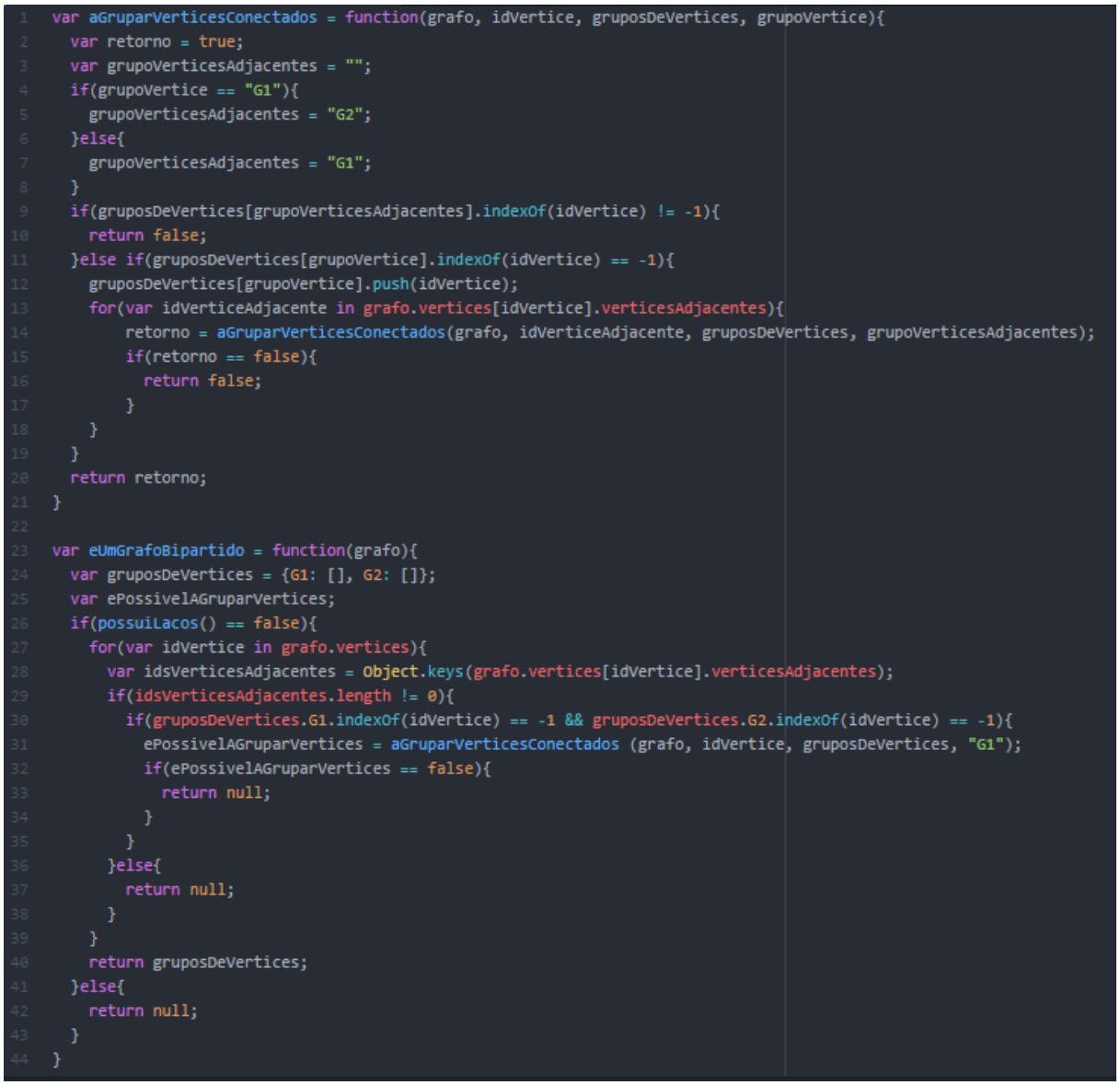
Depois de gerar todas as possibilidades de combinações entre os vértices, na linha 33 essas possibilidades são organizadas na lista em ordem crescente por quantidades de vértices utilizando um algoritmo *Quicksort*. Da linha 34 a 43 tem um laço de repetição que percorre todas as possibilidades de combinações entre os vértices, a cada iteração nas linhas 35 e 36 é realizada uma chamada a função que gera um grafo a partir de uma lista de vértices e arestas, passando como parâmetro a possibilidade que está sendo percorrida, o grafo retornado é adicionado a lista de subgrafos. Na linha 37 é feita uma chamada a uma função que possui três laços de repetição aninhados que tem o mesmo padrão dos três laços aninhados da linha 19 a 32 com a diferença de ao invés de acessa todas as possibilidades de combinações entre os vértices é acessada todas as possibilidades de combinações das ligações entre uma lista de vértices passada como parâmetro, ou sejam são geradas as possibilidades de combinações

incluído as arestas presentes no grafo. Da linha 39 a 42 tem um laço de repetição que percorre todas as possibilidades que inclui arestas, a cada iteração é gerado o grafo referente a possibilidade que está sendo percorrida e esse grafo é adicionado a lista de subgrafos. Na linha 44 é retornada a lista contendo todos os subgrafo gerados.

4.3.13 Análise de grafo bipartido

Para analisar se um grafo simples ou multigrafo é um grafo bipartido e necessário percorrer todos os vértices do grafo, a cada vértice percorrido é utilizada uma função recursiva que percorre os vértices que estão conexos a ele, utilizando as ligações dos vértices adjacentes. Durante a execução da função recursiva os vértices são agrupados em dois grupos, a cada iteração um vértice é agrupado e verificado se o vértice que está sendo agrupado já pertence a o outro grupo, caso ele já pertença a o outro grupo é provado que o grafo não é um grafo bipartido, caso contrário continua a execução até que todos os vértices estejam agrupados. A figura 32 mostra as funções utilizadas para analisar, se um grafo é um grafo bipartido.

Figura 32 - Funções para analisar, se um grafo é um grafo bipartido



```
1  var aGruparVerticesConectados = function(grafo, idVertice, gruposDeVertices, grupoVertice){
2    var retorno = true;
3    var grupoVerticesAdjacentes = "";
4    if(grupoVertice == "G1"){
5      grupoVerticesAdjacentes = "G2";
6    }else{
7      grupoVerticesAdjacentes = "G1";
8    }
9    if(gruposDeVertices[grupoVerticesAdjacentes].indexOf(idVertice) != -1){
10     return false;
11   }else if(gruposDeVertices[grupoVertice].indexOf(idVertice) == -1){
12     gruposDeVertices[grupoVertice].push(idVertice);
13     for(var idVerticeAdjacente in grafo.vertices[idVertice].verticesAdjacentes){
14         retorno = aGruparVerticesConectados(grafo, idVerticeAdjacente, gruposDeVertices, grupoVerticesAdjacentes);
15         if(retorno == false){
16           return false;
17         }
18     }
19   }
20   return retorno;
21 }
22
23 var eUmGrafoBipartido = function(grafo){
24   var gruposDeVertices = {G1: [], G2: []};
25   var ePossivelAGruparVertices;
26   if(possuiLacos() == false){
27     for(var idVertice in grafo.vertices){
28       var idsVerticesAdjacentes = Object.keys(grafo.vertices[idVertice].verticesAdjacentes);
29       if(idsVerticesAdjacentes.length != 0){
30         if(gruposDeVertices.G1.indexOf(idVertice) == -1 && gruposDeVertices.G2.indexOf(idVertice) == -1){
31           ePossivelAGruparVertices = aGruparVerticesConectados (grafo, idVertice, gruposDeVertices, "G1");
32           if(ePossivelAGruparVertices == false){
33             return null;
34           }
35         }
36       }else{
37         return null;
38       }
39     }
40     return gruposDeVertices;
41   }else{
42     return null;
43   }
44 }
```

Na figura 32 são mostradas duas funções, uma função recursiva que percorre e agrupa todos os vértices conexos um vértice qualquer do grafo e outra que analisa se um grafo é um grafo bipartido. Nas linhas de 1 até 21 é mostrada a função recursiva que percorre e agrupa todos os vértices conexos um vértice qualquer do grafo. A função recebe como parâmetro um grafo, um vértice qualquer desse grafo, uma lista dividida em dois grupos de vértices G1 e G2, e o nome do grupo em que o vértice recebido como parâmetro deve ser colocado. Na linha é declara uma variável de retorno que inicialmente recebe o valor *true*. Na linha 3 é declara uma variável que armazena o nome do grupo que os vértices adjacentes ao vértice recebido como parâmetro devem pertencer. Da linha 4 até a linha 8 tem um condicional que verifica se o nome do grupo do vértice recebido como parâmetro é G1, caso seja G1 a variável que armazena o nome do grupo dos vértices adjacentes recebe o valor G2, caso contraria a variável recebe o

valor G1. Da linha 9 até a linha 19 tem um condicional que retorna o valor *false* caso o vértice recebido como parâmetro já pertença ao grupo dos vértices adjacentes, caso contrário é verificado se o vértice recebido como parâmetro não pertence ao grupo recebido como parâmetro. Caso o vértice não pertença ao grupo na linha 12 ele é colocado dentro do grupo e é executado um laço de repetição da linha 13 até a linha 19 que percorre todos os vértices adjacentes ao vértice recebido como parâmetro. A cada iteração do laço de repetição na linha 14 é atribuída a variável de retorno a chamada recursiva da função, essa nova chamada da função recebe os mesmos parâmetros com exceção do vértice que foi passado como parâmetro que é trocado pelo vértice adjacente que está sendo percorrido e o nome do grupo do vértice recebido como parâmetro que é trocado pelo nome do grupo dos vértices adjacentes. Após a chamada recursiva da função é executado um condicional da linha 15 até a linha 17 que retorna o valor *false* caso o valor de retorno da chamada recursiva seja *false*. Na linha 20 é retornado o valor contido na variável de retorno.

Nas linhas de 23 a 44 da figura 32 é mostrada a função que analisa se um grafo passado como parâmetro é um grafo bipartido. Na linha 24 é criada e atribuída a uma variável uma lista dividida em dois grupos G1 e G2 que armazenara os vértices agrupados, a lista é criada vazia. Da linha 26 a 43 tem um condicional que retorna o valor *null* caso o grafo recebido como parâmetro não seja um grafo simples ou um multigrafo, caso contrário é executado um laço de repetição da linha 27 até a linha 39 que percorre todos os vértices do grafo recebido como parâmetro. A cada iteração do laço de repetição tem um condicional da linha 29 até a linha 38 que retorna o valor *null* caso o vértice que está sendo percorrido não tenha vértices adjacentes, caso contrário é executado um condicional da linha 30 a 35 que verifica se o vértice que está sendo percorrido não pertence a nenhum grupo. Caso o vértice não pertença a nenhum grupo é realizada uma chamada a função da linha 1 a 21 que percorre e agrupa todos os vértices conexos a um vértice qualquer do grafo, essa chamada a função recebe como parâmetros o grafo que foi passado como parâmetro, o vértice que está sendo percorrido, a lista que armazena os vértices agrupados, e nome de um dos grupos. Nas linhas de 32 a 34 tem um condicional que verifica se o retorno da chamada da função que percorre e agrupa todos os vértices conexos a um vértice qualquer do grafo é igual ao valor *false*, caso seja é retornado o valor *null*. Se todos os vértices do grafo poderem ser agrupados na linha 40 é retornada a lista com os vértices agrupados.

As funções da figura 32 podem ser aproveitadas para analisar se um grafo é um grafo bipartido completo. Para verificar se um grafo simples é um grafo bipartido completo basta utilizar a lista retornada pela função que analisa se um grafo é um grafo bipartido para verificar

se todos os vértices de um grupo são adjacentes a todos os vértices do outro grupo. A figura 33 mostra a função utilizada para analisar, se um grafo é um grafo bipartido completo.

Figura 33 - Função que analisar, se um grafo é um grafo bipartido completo

```
1  var eUmGrafoBipartidoCompleto = function(grafo){
2    var gruposDeVertices = {};
3    if(eUmGrafoSimples(grafo)){
4      gruposDeVertices = eUmGrafoBipartido(grafo);
5      if(gruposDeVertices != null){
6        for(var i = 0 ; i < gruposDeVertices.G1.length; i++){
7          for(var j = 0 ; j < gruposDeVertices.G2.length; j++){
8            if(grafo.vertices[gruposDeVertices.G1[i]].verticesAdjacentes[gruposDeVertices.G2[j]] == undefined){
9              return null;
10           }
11         }
12       }
13       return gruposDeVertices;
14     }else{
15       return null;
16     }
17   }else{
18     return null;
19   }
20 }
```

Na figura 28 é mostrada a função que analisa se um grafo é um grafo bipartido completo, essa função recebe como parâmetro o grafo que será analisado. Nas linhas de 3 a 19 tem um condicional que verifica se o grafo passado como parâmetro é um grafo simples, caso o grafo não seja um grafo simples é retornado o valor *null*. Caso o grafo seja um grafo simples, na linha 4 é atribuído a uma variável o retorno da chamada da função que analisa se um grafo é um grafo bipartido. Da linha 5 até a linha 16 tem um condicional que verifica se o retorno obtido é diferente do valor *null*, caso o retorno seja igual ao valor *null* é retornado o valor *null*, caso contrário da linha 6 até a linha 12 são executados dois laços de repetição aninhados, com os dois laços de repetição aninhados e utilizando a lista com os vértices agrupados é possível acessar todas as combinações de pares de vértices formados por vértices de grupos diferentes. Assim sendo dentro do laço interno tem um condicional da linha 8 até a linha 10 que retorna o valor *null* caso os vértices que formam o par não sejam adjacentes. Se em todos os pares de vértices os vértices forem adjacentes na linha 13 é retornada a lista com os vértices agrupados.

4.3.14 Análise de grafo planar

Para analisar se um grafo é um grafo planar basta implementar uma função baseada no teorema de Kuratowski. A função deve gerar todos os subgrafos do grafo em que a análise é feita, cada subgrafo é verificado se ele é um grafo completo de cinco vértices ou um grafo bipartido completo com conjuntos de três vértices, caso ele seja a função deve retorna o valor *false* para indicar que o grafo não é planar, caso contrário deve-se excluir todas as subdivisões

elementares do subgrafo e fazer a verificação novamente. A figura 34 mostra as funções utilizadas para analisar, se um grafo é um grafo planar.

Figura 34 - Funções para analisar, se um grafo é um grafo planar

```
1  var excluirConfiguracoes = function(subgrafo){
2    var sufixoIDAresta = 1;
3    var prefixoIDAresta = "aresta";
4    var existeConfiguracao = true;
5    while(existeConfiguracao == true){
6      excluirTodosOsLacos(subgrafo);
7      excluirTodasAsArestasParalelas(subgrafo);
8      existeConfiguracao = false;
9      var idsVertices = Object.keys(subgrafo.vertices);
10     for(var i = 0; i < idsVertices.length; i++){
11       if(qualOGrauDoVertice(subgrafo.vertices[idsVertices[i]]) == 2){
12         var idsExtremidades = [];
13         for(var  idVerticeAdjacente in subgrafo.vertices[idsVertices[i]].verticesAdjacentes){
14           idsExtremidades.push(idVerticeAdjacente);
15         }
16         var idAresta = prefixoIDAresta+sufixoIDAresta;
17         sufixoIDAresta++;
18         adicionarAresta(subgrafo, idAresta, idsExtremidades[0], idsExtremidades[1]);
19         excluirVertice(subgrafo, idsVertices[i]);
20         existeConfiguracao = true;
21         break;
22       }
23     }
24   }
25 }
26
27 var eUmGrafoPlanar = function(grafo){
28   var subgrafos = criarSubgrafos(grafo);
29   for(var i = 0; i < subgrafos.length  ; i++){
30     var subgrafo = subgrafos[i];
31     if((eUmGrafoBipartidoCompleto(subgrafo) != null && eUmGrafoRegular(subgrafo) == 3) || (eUmGrafoCompleto(subgrafo) == true && eUmGrafoRegular(subgrafo) == 10)){
32       return false;
33     }else{
34       excluirConfiguracoes(subgrafo);
35       if((eUmGrafoBipartidoCompleto(subgrafo) != null && eUmGrafoRegular(subgrafo) == 3) || (eUmGrafoCompleto(subgrafo) == true && eUmGrafoRegular(subgrafo) == 10)){
36         return false;
37       }
38     }
39   }
40   return true;
41 }
```

Na figura 34 são mostradas duas funções, uma função que excluir todas as subdivisões elementares de um grafo e outra que analisa se um grafo é um grafo planar. Nas linhas de 1 até 25 é mostrada a função que excluir todas as subdivisões elementares de um grafo recebido como parâmetro. Na linha 4 é declarada uma variável que indica se o grafo tem subdivisões elementares ou não, inicialmente como não foi realizada nenhuma verificação ela recebe *true* indicando que o grafo tem subdivisões elementares. Da linha 5 até a linha 24 tem um laço de repetição que fica em *loop* até que a variável declarada na linha 4 indique que não existe subdivisões elementares no grafo. Nas linhas 6 e 7 são realizadas chamadas a duas funções para excluir todos os laços e arestas paralelas do grafo, isso é necessário porque a exclusão de uma subdivisão elementar pode criar um laço ou duas arestas paralelas. Na linha 8 a variável declarada na linha 4 recebe o valor *false* indicando que não existe mais subdivisões elementares. Da linha 10 até 23 tem um laço de repetição que percorre todos os vértices do grafo, a cada

iteração do laço de repetição tem um condicional da linha 11 até a linha 22 que verifica se o vértice que está sendo percorrido tem grau 2. Caso o vértice tenha grau dois na linha 18 e 19, ele é excluído e é realizada a fusão das duas arestas ligadas a ele, em seguida na linha 20 a variável da linha 4 recebe o valor *true* para indicar que ainda existe subdivisões elementares, na linha 21 é executado um comando para sair do laço de repetição que percorre todos os vértices. Cada vez que uma subdivisão elementar é excluída todos os vértices do grafo são verificados novamente até que não tenha mais nenhuma subdivisão elementar.

Nas linhas de 27 a 41 da figura 34 é mostrada a função que analisa se um grafo passado como parâmetro é um grafo planar. Na linha 28 é declarada uma variável e atribuída a ela todos os subgrafos do grafo passado como parâmetro. Da linha 29 a 39 tem um laço de repetição que percorre todos os subgrafos, a cada iteração tem um condicional da linha 31 a 38 que retorna o valor *false* caso o subgrafo que está sendo percorrido seja um grafo completo de cinco vértices ou um grafo bipartido completo com conjuntos de três vértices, caso contrário na linha 34 são excluídas todas as subdivisões elementares do subgrafo utilizando a função da linha 1 até a 25. Depois de serem excluídas todas as subdivisões elementares é executado um condicional da linha 35 a 37 que verifica novamente se o subgrafo é um grafo completo de cinco vértices ou um grafo bipartido completo com conjuntos de três vértices e retorna o valor *false* caso seja. Na linha 40 é retornado o valor *true* se nenhum subgrafo mesmo depois de excluir todas as suas subdivisões elementares for um grafo completo de cinco vértices ou um grafo bipartido completo com conjuntos de três vértices.

Essa seção apresentou os resultados obtidos no desenvolvimento da aplicação. Foram apresentados a arquitetura da aplicação, os módulos de desenho e análise. O módulo de desenho incluiu a apresentação da tela da aplicação e da estrutura de dados do grafo em código Javascript que é objeto de análise. No módulo de análise foram mostradas as funções criadas para analisar o objeto grafo em código Javascript.

A seção a seguir aborda as considerações finais deste trabalho e os possíveis trabalhos futuros.

5 CONSIDERAÇÕES FINAIS

Nesse trabalho foram desenvolvidos dois módulos de um sistema para desenhar grafos e mostrar informações teóricas sobre o grafo desenhado: o módulo de desenho de grafos e o módulo de análise de grafo desenhado. No módulo de desenho de grafos foi utilizado o *framework* SVG.js que facilita a manipulação do elemento SVG do HTML, um vértice é representado por um elemento CIRCLE e uma aresta por um elemento PATH, quando a aresta é direcionada sua seta é representada por um elemento POLYGON de três lados. Eventos em Javascript de teclado e mouse foram utilizados para que o usuário desenhe de forma interativa usando o mouse e o teclado. Para trabalhos futuros poderiam ser implementadas mais formas de representar um vértice na tela de desenho além de círculos, como, por exemplo, polígonos e imagens. Também poderiam ser implementadas funcionalidades que permitissem alterar as dimensões do vértice, a cor de preenchimento e da borda do vértice e o tamanho e estilo da fonte de texto do rótulo das aresta e vértices.

No módulo que analisa o grafo desenhado foram implementadas funções que permitem classificar e mostrar as informações que permitiram a classificação do grafo desenhado nos seguintes tipos de grafos: grafo vazio; grafo nulo; grafo simples; multigrafo; pseudografo; grafo rotulado; grafo regular; grafo completo; grafo conexo; grafo planar; e grafo bipartido. Também foram implementadas funções para gerar o grafo complementar e todos os subgrafos do grafo desenhado. Para trabalhos futuros poderia ser implementada a utilização, de forma informativa, no grafo desenhado algoritmos que buscam solucionar problemas representados por grafos, como por exemplo o algoritmo de Dijkstra para encontrar o menor caminho entre dois vértices selecionados pelo usuário no desenho do grafo.

A aplicação foi testada e está pronta para o uso de professores e alunos da disciplina de estruturas de dados. Durante os testes funcionais foi constatado que a aplicação classifica o grafo desenhado corretamente de acordo com os tipos de grafos, e as informações que permitiram a classificação estavam corretas.

A aplicação pode ser usada em trabalhos futuros para acrescentar novas funcionalidades e assim se tornar uma ferramenta mais completa, visto que a teoria dos grafos é um assunto vasto e esse trabalho abordou apenas conceitos básicos.

REFERÊNCIAS

GOLDBARG, Marco; GOLDBARG, Elizabeth. **Grafos:** conceitos, algoritmos e aplicações. Rio de Janeiro: Elsevier, 2012. 640 p.

SIMÕES-PEREIRA, J. M. S.. **Grafos e redes:** teoria e algoritmos básicos. Rio de Janeiro: Interciência, 2013. 354 p.

ROSEN, Kenneth H.. **Matemática discreta e suas aplicações.** 6. ed. Porto Alegre: Amgh, 2009. 982 p.

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de dados:** algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010. 448 p.

STEIN, Clifford; DRYSDALE, Robert L.; BOGART, Kenneth. **Matemática discreta para ciência da computação.** São Paulo: Pearson Education do Brasil, 2013. 416 p.

CAVALCANTE, Fabiana Nascimento Santos; SILVA, Severino Domingos da. **GRAFOS E SUAS APLICAÇÕES.** 2009. 63 f. TCC (Graduação) - Curso de Licenciando em Matemática, Centro Universitário Adventista de São Paulo, São Paulo, 2009. Disponível em: <http://www.pucrs.br/famat/viali/graduacao/producao/po_2/literatura/grafos/monografias/tcc1.pdf>. Acesso em: 31 maio 2017.

DOVICCHI, João. **Estrutura de Dados.** 2007. Disponível em: <http://www.inf.ufsc.br/~joao.dovicchi/pos-ed/ebook/e-book_estrut_dados_dovicchi.pdf>. Acesso em: 28 mar. 2017.

SVG.js. **SVG.js**. Disponível em: <http://svgjs.com/>. Acesso em: 25 out. 2017.