UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FURB GRAPHS: UMA APLICAÇÃO PARA TEORIA DOS GRAFOS

ANDERSON DE BORBA

BLUMENAU
2014

2014/2-01

ANDERSON DE BORBA

FURB GRAPHS: UMA APLICAÇÃO PARA TEORIA DOS GRAFOS

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2014

2014/2-01

FURB GRAPHS: UMA APLICAÇÃO PARA TEORIA DOS GRAFOS

Por

ANDERSON DE BORBA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, Mestre – Orientador, FURB
Membro:	Prof. Joyce Martins, Mestre – FURB
Membro:	Prof. Roberto Heinzle, Doutor – FURB

Blumenau, 12 de dezembro de 2014

Dedico este trabalho a todos que de algum forma contribuíram para a realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, por todo o todo o apoio, educação, confiança. Em especial aos meus pais, Gilberto e Roseli, pela orientação.

A minha namorada, Cintia Aline Siqueira, por todo o amor, apoio e compreensão.

Aos meus amigos e colegas de cursos por todo o suporte, companheirismo e ideias trocadas no decorrer do curso.

Ao meu orientador Aurélio Faustino Hoppe, pela confiança e auxílio prestado para a conclusão deste trabalho e pelos cinco anos como *coach* da Maratona de Programação. Agradeço-o também pelo incrível dom de transmitir calma mesmo nos momentos mais turbulentos.

Um raciocínio lógico leva você de A a B. A imaginação leva você a qualquer lugar que você quiser.

Albert Einstein

RESUMO

Este trabalho apresenta a continuação do desenvolvimento de um *framework* para a área da teoria dos grafos com o adendo de fornecer uma interface visual e interativa para manipulação e criação do grafo, tornando-o assim, uma aplicação para teoria dos grafos. A aplicação foi implementada na linguagem Java e possui o teste de propriedades tais como: número cromático, hipercubo e isomorfismo a outro grafo. A aplicação também foi complementada com os algoritmos ciclo hamiltoniano e caminho euleriano. Por fim, é disponibilizado uma aplicação visual e interativa feita para manipulação do grafo, podendo realizar o teste de propriedades e executar algoritmos. Os resultados obtidos demonstram que a aplicação cumpre o seu objetivo com a corretude para as propriedades, algoritmos e disponibilização da interface visual, porém pode ainda ser complementado para disponibilizar mais testes de propriedades e algoritmos, além de melhorar a usabilidade da interface visual.

Palavras-chave: Teoria dos grafos. Algoritmos. Estrutura de dados.

ABSTRACT

This works presents the further development of a framework for the area of graph theory with the addendum to provide a visual and interactive interface for manipulating and creating the graph, thus making it an application to graph theory. The application has been implemented in Java and has the properties of test such as chromatic number, hypercube and graph isomorphism. The application was also supplemented with algorithms Hamiltonian cycle and Euler path. Finally, it is also made available a visual and interactive application made to the graph manipulation, may also hold the properties of test and execute algorithms. The results show that the application complies with the aim of correctness for the properties, algorithms and availability of the visual interface, but can still be supplemented to provide more testing properties and algorithms, and improve the usability of the visual interface.

Keywords: Graph theory. Algorithms. Data structure.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

API – *Application Programming Interface*

JSON – JavaScript *Object Notation*

UML – *Unified Modeling Language*

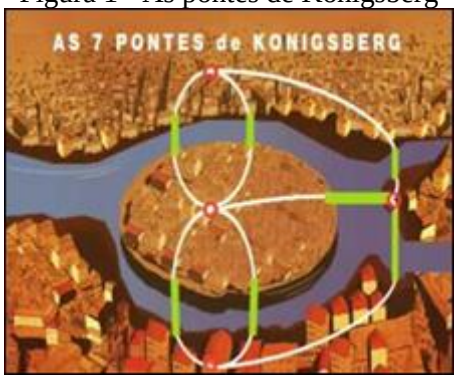
XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A teoria dos grafos vem desde muito tempo sendo um instrumento para resolver problemas do dia a dia através da modelagem de problemas do cotidiano para grafos. A mais antiga menção ao assunto ocorreu no trabalho de Euler, no ano de 1736 (RABUSKE, 1992, p. 3). O trabalho de Euler consistia em resolver uma dúvida do cotidiano da época que mais tarde ficou conhecido como "problema das pontes de Königsberg" (BOAVENTURA NETTO, 1996, p. 1). O problema é baseado na cidade de Königsberg que é cortada pelo Rio Prególia, onde há duas ilhas que juntas formavam uma cidade que continha sete pontes (Figura 1). O problema consistia em saber se havia possibilidade de atravessar todas as pontes da cidade sem repetir nenhuma. Na Figura 1, as linhas verdes representam as pontes (arestas) e os círculos vermelhos representam um ponto da cidade (vértice).

Figura 1 - As pontes de Königsberg



Fonte: adaptado de Rabuske (1992, p. 2).

Tal dúvida assombrava os moradores da época. Euler conseguiu provar através da modelagem do problema para grafos que não é possível realizar tal caminho. Caminho que ficou conhecido posteriormente como "caminho euleriano".

Atualmente teoria dos grafos é ensinada nos cursos de graduação em Ciência da Computação, sendo um assunto bastante complexo para ser lecionado devido à sua essência algorítmica. Segundo Santos e Costa (2006), assimilar as propriedades e entender os nuances da execução dos algoritmos são as principais dificuldades enfrentadas pelos alunos.

Diante deste contexto, percebe-se a importância de se ter uma aplicação voltada para o ensino de algoritmos em grafos. Para melhor compreensão e assimilação do aluno de grafos, propõe-se neste trabalho a extensão do *framework* para grafos desenvolvido por Zatelli (2010), adicionando uma interface gráfica na qual o aluno poderá visualizar e verificar propriedades por si mesmo, além de executar algoritmos relacionados à teoria dos grafos, caracterizando assim, uma aplicação para aprendizagem de grafos.

1.1 OBJETIVOS

Este trabalho tem como objetivo dar continuação ao trabalho de conclusão de curso de Zatelli (2010), produzindo novas funcionalidades para a aplicação de grafos proposto anteriormente.

Os objetivos específicos do trabalho são:

a) disponibilizar a verificação de mais propriedades, tais como o número cromático, hipercubo e isomorfo a outro grafo;
b) incluir mais algoritmos clássicos de grafos, tais como ciclo hamiltoniano e caminho euleriano;
c) disponibilizar uma interface gráfica com recursos visuais e interativos para a manipulação do grafo.

1.2 ESTRUTURA

Este trabalho está subdividido em quatro capítulos. Esse primeiro capítulo apresenta a justificativa para o desenvolvimento do trabalho, assim como seus objetivos e sua estrutura.

O segundo capítulo contém a fundamentação teórica, explicando conceitos gerais sobre a teoria dos grafos esclarecendo tópicos importantes que devem ser considerados para o desenvolvimento da aplicação.

O terceiro capítulo trata o desenvolvimento da aplicação, onde são listados seus requisitos, bem como sua especificação através dos diagramas de caso de uso, de atividade e de classes. Também é descrita a implementação, técnicas e ferramentas utilizadas, operacionalidade e, por fim, são apresentados e discutidos os resultados obtidos.

Por fim, no quarto capítulo são descritas as conclusões encontradas e as extensões sugeridas para trabalhos futuros.

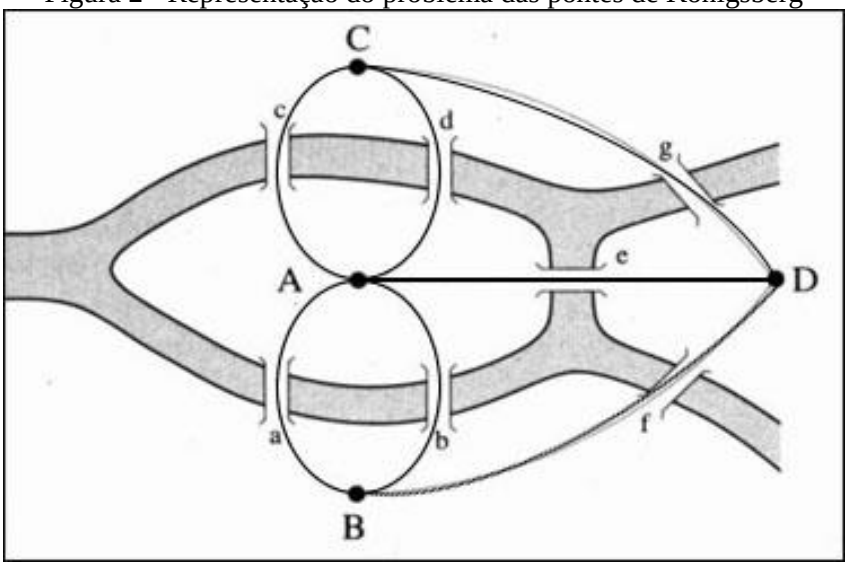
2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 apresenta o surgimento da área da teoria dos grafos. A seção 2.1.1 apresenta os conceitos básicos sobre a teoria dos grafos. Na seção 2.1.2 é apresentada algumas das propriedades dos grafos. A seção 2.1.3 apresenta alguns algoritmos de grafos. Na seção 2.2 é detalhado o trabalho ao qual será dada continuação, primeiramente desenvolvido por Zatelli (2010). Por fim, na seção 2.3, são apresentados os trabalhos correlatos, assim como uma comparação entre eles.

2.1 TEORIA DOS GRAFOS

A menção mais antiga sobre a teoria dos grafos ocorreu no ano de 1736 e não passava de uma especulação matemática (RABUSKE, 1992, p. 1). Na época, o matemático Leonhard Paul Euler resolveu o "problema das pontes de Königsberg" através uma modelagem para grafos (Figura 2).

Figura 2 - Representação do problema das pontes de Königsberg



Fonte: adaptado de Szwarcfiter (1984, p. 18).

No problema estudado por Euler, ele demonstrou que, dada a disposição existente das pontes, era impossível percorrer todas as pontes passando uma e só uma vez em cada ponte. Mais tarde, esse tipo de situação foi categorizado como "caminho euleriano".

Mais de um século depois, ocorreu outra menção importante sobre a teoria dos grafos. Boaventura Netto (1996, p. 1) conta alguns estudos importante da história da teoria dos grafos. Segundo o autor, em 1847, Kirchhoff iniciou os estudos sobre árvores - um tipo de grafo - quando estudava problemas de circuitos elétricos. Em 1857, Cayley relacionou teoria dos grafos com o estudo de hidrocarbonetos alifáticos, em química orgânica. Em 1859,

Hamilton estudava problemas de caminho enquanto Jordan (dez anos depois, 1869), formalizava a teoria das árvores.

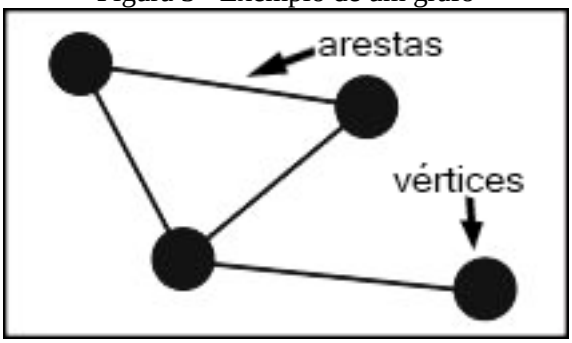
No século seguinte, o estudo sobre a teoria dos grafos se desenvolveu mais rápido ainda com o interesse de muitos pesquisadores (BOAVENTURA NETTO, 1996, p. 2). Nessa época, em 1962, surgiu a teoria dos fluxos em redes - originalmente proposto por Ford e Fulkerson - uma área de grande importância. Rabuske (1992, p. 4) ainda cita o problema das quatro cores, provado por Appel e Haken em 1977. Tal problema era provar que qualquer mapa ou superfície possa ser colorido utilizando somente quatro cores, de forma que nunca dois países vizinhos fiquem com a mesma cor.

A partir destes estudos a área de teoria dos grafos começou a ser utilizada para os mais diversos fins, tais como planejamento eficiente de roteamento de pacotes na Internet, definição de melhor rota para distribuições de correspondências, reconhecimento de padrões, sendo úteis também em áreas como química (comparação da topologia da estrutura de orgânicos), biologia (cálculo da distância máxima entre as células de um vírus ofensivo ao corpo humano) e geografia (caminho mínimo entre cidades).

2.1.1 Conceitos básicos da teoria dos grafos

Segundo Kocay e Kreher (2005, p. 1), um grafo $G(V, E)$ pode ser definido como um conjunto não vazio de vértices $V(G)$ e um conjunto de arestas $E(G)$ onde cada aresta é um par $\{u, v\}$ de vértices $u, v \in V(G)$. Os elementos que pertencem a V são representados por pontos, enquanto que os elementos de E são representado por linhas ou arcos (RABUSKE, 1992, p. 5). Um exemplo de grafo pode ser visualizado na Figura 3.

Figura 3 - Exemplo de um grafo



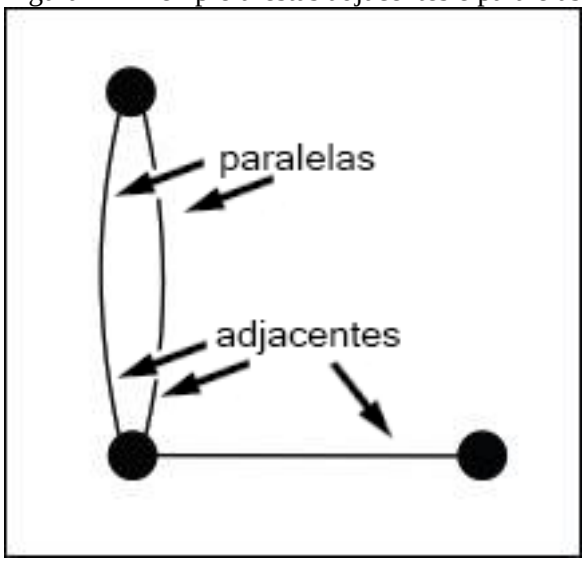
Fonte: adaptado de Gross e Yellen (2006, p. 3).

Grafos podem ser basicamente separados em grafos dirigidos ou não dirigidos. Um grafo é dirigido (ou digrafo) se suas arestas possuem orientações, caso contrário, é dito não dirigido (RABUSKE, 1992, p. 7). Grafos dirigidos são utilizados quando a relação entre dois vértices não é simétrica, ou seja, o vértice A implica no vértice B porém o vértice B não

implica para o vértice A. Um exemplo disto é o fluxo de carros em uma rodovia de mão única (SZWARCFITER, 1984, p. 3).

Duas arestas que incidem sobre o mesmo vértice são chamadas de adjacentes ou vizinhas. Arestas que possuem tanto o mesmo vértice origem e o mesmo vértice de destino são chamadas de paralelas (Figura 4) (RABUSKE, 1992, p. 6).

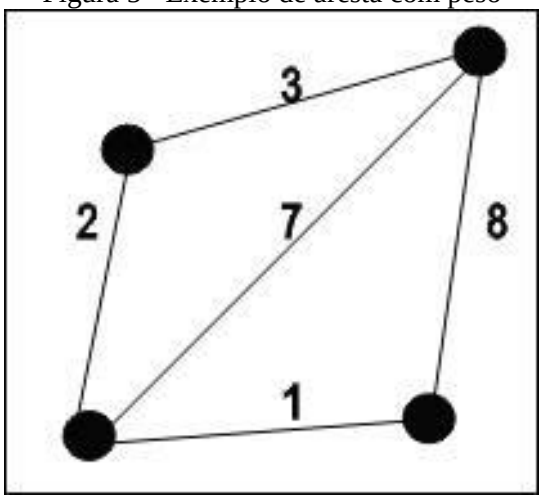
Figura 4 - Exemplo arestas adjacentes e paralelas



Fonte: adaptado de Gross e Yellen (2006, p. 4).

Uma definição importante é o grau de um vértice, que segundo Rabuske (1992, p. 8), é o número de arestas que incidem em um vértice. Uma aresta também pode possuir valores ou atributos. Um atributo pode ser um valor numérico que, para Rabuske (1992, p. 12) é chamado de peso, distância ou custo (Figura 5).

Figura 5 - Exemplo de aresta com peso



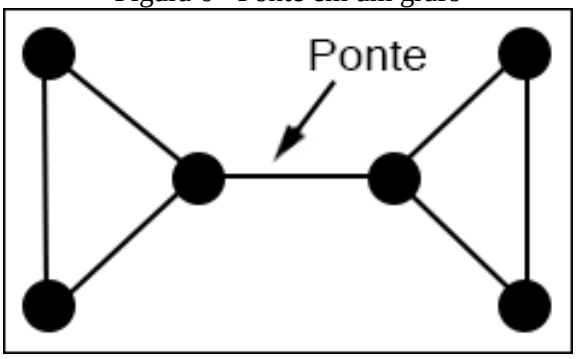
Fonte: adaptado de Gross e Yellen (2006, p. 20).

As arestas valoradas servem para indicar o custo de navegação entre vértices adjacentes. Em redes de comunicação ou transporte, estes custos representam alguma

quantidade física, tal como distância, eficiência, carga, capacidade da aresta correspondente, entre outros (RABUSKE, 1992, p. 12).

Uma ponte é uma aresta que no caso de ser removida torna o grafo desconexo (Figura 6) (CORMEN et al., 2001, p. 558).

Figura 6 - Ponte em um grafo



Fonte: adaptado de Rabuske (1991, p. 46).

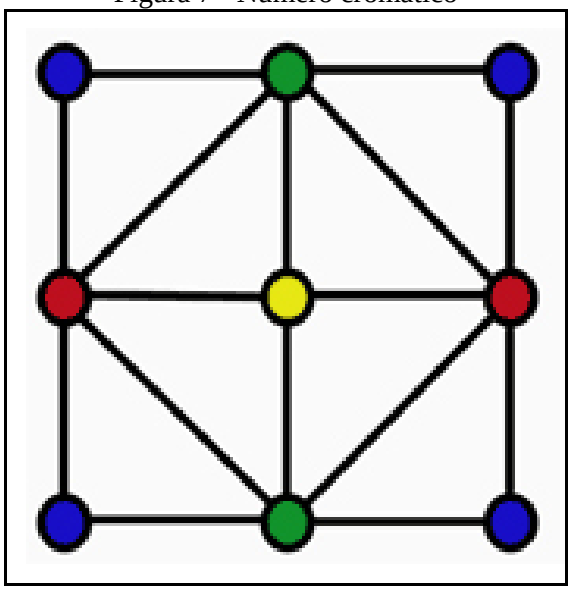
2.1.2 Propriedades dos grafos

Existem diversas propriedades que podem ser extraídas a partir da representação de um grafo. Nas próximas seções serão apresentados as propriedades propostas para este trabalho.

2.1.2.1 Número cromático

Gross e Yellen (2006, p. 326) definem o número cromático de um grafo como sendo o menor número de cores necessárias para colorir todos os vértices de um grafo, de modo que todos os vértices adjacentes tenham cores diferentes. A Figura 7 exibe um grafo com o número cromático quatro, pois são necessárias quatro cores para que não haja vértices adjacentes com as mesmas cores (azul, verde, vermelho e amarelo).

Figura 7 - Número cromático



Fonte: adaptado de Gross e Yellen (2006, p. 67).

A resolução do problema do número cromático permite a resolução do problema de particionamento. Rabuske (1992, p. 143) observa que ao separar as cores de um grafo, as cores podem serem agrupadas em grupo de vértices com cores A, outro de cor B e assim sucessivamente.

O particionamento é aplicável em muitos problemas práticos, tais como redução de estados de máquinas sequencias, problema de horários, sinalização de trânsito, transporte de bens, justificando assim o grande interesse no estudo dos números cromáticos.

O ato de pintar os vértices de um grafo de modo que dois vértices adjacentes nunca tenham cores iguais chama-se de coloração do grafo (RABUSKE, 1992, p. 144). Uma coloração de interesse é aquela que usa o menor número possível de cores, pois caso contrário, seria simplesmente definir o número cromático de um grafo como sendo o tamanho do grafo. Um grafo *G* que exige `k` cores para pintar os vértices dos grafos e não menos, é chamado de grafo k-cromático e o número `k` representa o número cromático do grafo.

Encontrar o valor k-cromático ótimo de um grafo é um problema NP-completo (CORMEN et al., 2001, p. 804). Portanto, encontrar a solução ótima é uma tarefa difícil. Um dos métodos conhecidos para encontrar a solução ótima é o método da força bruta, na qual busca-se todas as combinações possíveis de cores entre os vértices, porém, mostra-se insatisfatório computacionalmente para um grafo com muitos vértices. Entretanto, existem algoritmos baseados em heurísticas que executam em tempo polinomial e que trazem uma solução aproximada (RABUSKE, 1992, p. 146). O Quadro 1 define um algoritmo baseado em heurística com princípios gulosos para a determinação do número cromático de um grafo.

Quadro 1 - Algoritmo para número cromático

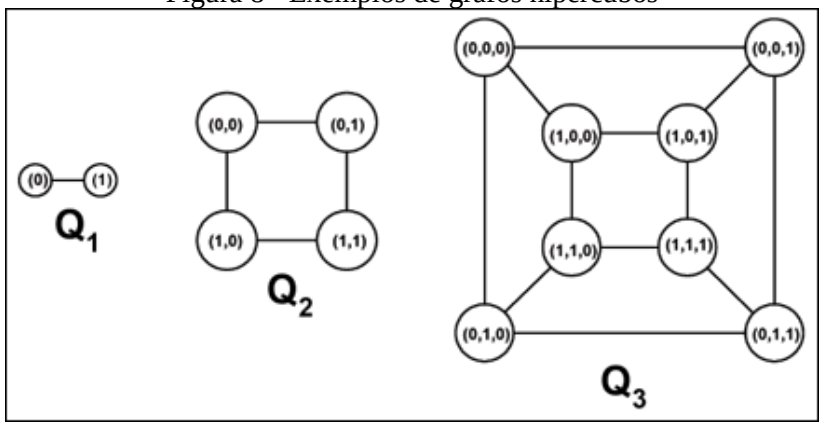
```
1. enquanto existirem vértices não coloridos:
1.1 escolha um vértice não colorido qualquer;
2.2 atribua a menor cor possível não utilizada ao vértice.
```

A ideia do algoritmo é verificar a coloração atual dos vértices adjacentes ao vértice que deseja-se atribuir uma cor. O algoritmo obtém a primeira cor disponível para uso que seja diferente dos vértices adjacentes.

2.1.2.2 Hipercubo

Hipercubo ou grafo cubo é definido por Prestes (2012) como “grafos bipartidos cujos vértices são k-tuplas de 0's e 1's, onde os vértices adjacentes diferem em exatamente uma coordenada” (Figura 8).

Figura 8 - Exemplos de grafos hipercubos



Fonte: Prestes (2012).

Na Figura 8 são exibidos três grafos hipercubos distintos (Q1, Q2 e Q3). Grafos hipercubos são conhecidos por Qx sendo x a quantidade de k-tuplas. O valor de x também corresponde a uma potência de dois, sendo que o valor de x é o que determina a quantidade de ligações entre os vértices.

O Quadro 2 apresenta um algoritmo para detecção de grafos hipercubos. O algoritmo baseia-se em verificar se o grafo apresenta as características que o tornam um grafo hipercubo.

Quadro 2 - Algoritmo para detecção de hipercubo

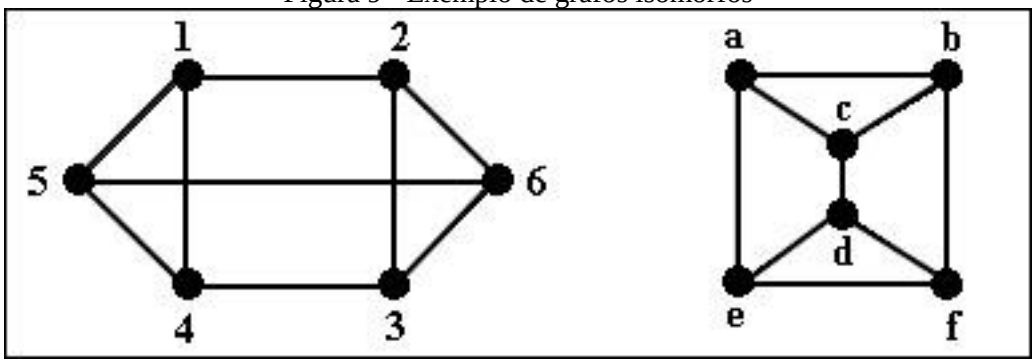
```
1. verifica se o tamanho do vértice é uma potência de dois;
2. verifica se todos os vértices possuem o grau correspondente ao valor
   originado da potência de dois;
3. verifica a ausência de laços e arestas paralelas.
```

Ranka e Sahni (2011) explicam que um grafo que apresenta a propriedade hipercubo é importante por diversas razões. Grafos hipercubos tornaram-se importante para os modelos arquiteturais paralelos, sendo que vários problemas de processamento de sinais, visão computacional e processamento de imagens podem ser resolvidos eficientemente através de algoritmos paralelos em hipercubo.

2.1.2.3 Isomorfismo

Rabuske (1992, p. 9) define que dois grafos são isomorfos quando é possível fazer coincidir, respectivamente, vértices de suas arestas gráficas, preservando as adjacências das arestas. Isto é, suas estruturas são equivalentes. A Figura 9 exibe um exemplo de dois grafos isomorfos porém postos de maneiras diferentes visualmente.

Figura 9 - Exemplo de grafos isomorfos



Fonte: adaptado de Rabuske (1992, p. 9).

Isomorfismo de grafos é um do problemas em abertos da computação do qual não se sabe se é pertencente à classe de problemas P ou NP (GAREY; JOHNSON, 1983). Uma forma de implementar um algoritmo para detecção de grafos é *backtracking* no qual os dois grafos caminham juntos até encontrar a solução. O Quadro 3 exibe o pseudocódigo para detecção de dois grafos isomorfos. A ideia do algoritmo é utilizar a técnica de *backtracking* para verificar todas as possibilidades possíveis de combinações dos vértices e arestas entre os dois grafos.

Quadro 3 - Algoritmo para isomorfismo

```
1. verifica se o tamanho e o grau de todos os vértices entre os grafos A
   e B são iguais;
2. escolhe como vértice inicial do grafo A, o vértice com o maior grau,
   para cada vértice com o mesmo grau em B:
2.1 cria uma associação entre o vértice e a aresta;
2.2 determina um caminho que seja viável entre os dois vértices do grafo
    A e B;
2.3.1 se for viável:
2.3.1.1 continua a operação de backtracing com as novas relações
2.3.2 se não for viável:
2.3.2.1 desfaz a relação e realiza o backtracking para encontrar a
        próxima relação.
```

Isomorfismo de grafos aplica-se para reconhecimento de padrões, como por exemplo o padrão biométrico baseado em impressões digitais (pois podem ser criados grafos a partir das minúcias do dedo), reconhecimento de imagens e química (através da comparação das estruturas moleculares).

2.1.3 Principais algoritmos da teoria dos grafos

A essência da teoria dos grafos consiste em saber modelar e aplicar corretamente os algoritmos disponíveis para o grafo (SKIENA; REVILLA, 2003, p. 237). Nas próximas seções são apresentados os algoritmos propostos para este trabalho.

2.1.3.1 Ciclo hamiltoniano

Segundo Boaventura Netto (1996, p. 26), um ciclo hamiltoniano ocorre quando um ciclo passa por todos os vértices somente uma vez. Nem todo grafo conexo possui um ciclo hamiltoniano e é um grande problema da computação identificar a condição necessária e suficiente para que um grafo conexo *G* possua tal ciclo. Esta questão foi proposta inicialmente pelo matemático Sir William Rowan Hamilton em 1859 (RABUSKE, 1992, p. 46).

Um problema bastante conhecido baseado em ciclos hamiltonianos é o problema do caixeiro-viajante. Este problema consiste em determinar a rota de menor custo de um viajante que parte de uma cidade inicial, visita diversas outras cidades passando uma só vez em cada cidade e retorna a cidade do ponto de partida (BOAVENTURA NETTO, 1996, p. 190).

Identificar um ciclo hamiltoniano é um problema NP-completo implicando que não há algoritmo conhecido até o momento que resolva este problema em tempo polinomial (CORMEN *et al*., 2001, p. 792). Boaventura Netto (1996, p. 191) também observa que se um grafo não for muito grande e se não tiver muitas arestas, é possível enumerar todos os circuitos hamiltonianos para depois achar o valor de cada um deles, porém, no caso geral, mantém-se inviável enumerar todas as possibilidades.

Um algoritmo que retorna um ciclo hamiltoniano (caso exista) de um grafo é o algoritmo de Robert e Flores (BOAVENTURA NETTO, 1996, p. 192) (Quadro 4).

Quadro 4 - Algoritmo de Robert e Flores

```
1. escolha do vértice inicial;
2. determina um caminho (se possível) que leva até o próximo vértice não
   visitado;
2.1. se um vértice viável é encontrado, continua a busca a partir deste
     vértice;
2.2. se nenhum vértice viável é encontrado, realiza a operação de
     retrocesso (backtracking) e reinicia a busca a partir de outro
     vértice;
3. encerra a busca quando todos os caminhos forem explorados.
```

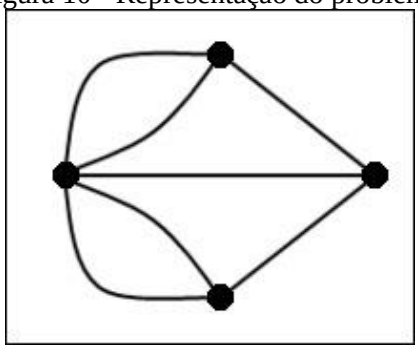
Este algoritmo cria uma árvore de busca com todas as possibilidades para encontrar o ciclo hamiltoniano. Embora não seja eficiente, o algoritmo retorna a solução ótima.

2.1.3.2 Caminho euleriano

De forma análoga ao ciclo hamiltoniano apresentado na seção 2.1.3.1, Boaventura Netto (1996, p. 26) define um caminho como sendo euleriano quando um circuito passa por todas as arestas uma e só uma vez.

Conforme mencionado na seção 2.1.1, o problema das pontes de Könisberg é a primeira aparição de caminhos eulerianos da literatura. A Figura 10 apresenta o problema em uma modelagem para grafos.

Figura 10 - Representação do problema



Na Figura 10, os vértices representam as ilhas e as arestas representam as pontes. Euler provou que não é possível encontrar um caminho que passe por todas as arestas somente uma vez e volte ao vértice inicial.

Caminhos eulerianos ainda podem ser divididos em dois grupos: caminhos fechados e caminhos abertos. Caminhos fechados são aqueles no qual é formado um ciclo entre os vértices ao final do caminhamento pois a última aresta visitada vai de encontro ao vértice inicial. Tal caminho é conhecido por caminho de Euler e um grafo que possui tal caminho fechado é conhecido como grafo de Euler ou grafo euleriano (RABUSKE, 1992, p. 43). Caminhos abertos são aqueles em que a última aresta visitada não vai de encontro ao vértice inicial, também conhecido por grafos semi-euleriano.

Euler provou que um grafo conexo *G* é um grafo de Euler se e somente se todos os seus vértices possuem grau par (RABUSKE, 1992, p. 43). Com tal teorema é possível observar que o grafo modelado para as pontes de Könisberg não pode ser um grafo euleriano pois todos os graus são ímpares. Já um grafo é semi-euleriano se e somente se existirem somente dois vértices de grau ímpar (RABUSKE, 1992, p. 44).

Rabuske (1992, p. 45) cita Fleury como um algoritmo bastante fácil para traçar um caminho euleriano (Quadro 5).

Quadro 5 - Algoritmo de Fleury

```
1. escolha do vértice inicial;
1.1. se todos os graus forem de grau par, inicia por qualquer um;
1.2. se o grafo conter dois vértices com grau ímpar, inicia por um deles;
1.3. se não atingir os requisitos, não há caminho euleriano;
2. atravessa todas as arestas de maneira arbitrária com uma única
   condição;
2.1. somente atravessa pontes caso não exista opção;
3. remove a aresta visitada;
4. se o vértice ficou isolado, remove o vértice isolado.
```

Este algoritmo baseia-se em caminhar pelo grafo respeitando as condições impostas. O resultado do algoritmo é uma lista de vértices os quais representam o caminho percorrido que retorna o caminho euleriano. O caminho será fechado caso o primeiro e o último vértice da lista sejam o mesmo e será um caminho aberto caso sejam dois vértices diferentes.

2.2 UM FRAMEWORK PARA ALGORITMOS BASEADOS NA TEORIA DOS GRAFOS

O trabalho desenvolvido por Zatelli (2010) é o FGA, um *framework* de algoritmos baseados na teoria dos grafos implementado na linguagem Java. Ao qual, permite persistir e salvar grafos em um formato próprio *eXtensible Markup Language* (XML).

O FGA disponibiliza várias verificações de propriedades de grafos. As propriedades disponíveis para consulta são: verificação para grafo completo, regular, trivial, nulo, bipartido, bipartido completo, conexo ou desconexo, fortemente conexo, denso ou esparso e simples ou multigrafo. Também é disponibilizado um subconjunto de funções capazes de gerar grafos com base em certas características. Os geradores de grafos disponíveis são: geradores para grafos completos, regulares, conexos ou desconexos, densos ou esparsos e simples ou multigrafos.

Por último, o FGA disponibiliza implementação para um conjunto de algoritmos clássicos. Os algoritmos disponibilizados são: Dijkstra, Floyd-Warshall, Bellman-Ford, Prim, Kruskal, Ford-Fulkerson, Hopcroft-Tarjan, ordenação topológica, busca em largura e busca em profundidade.

Entre os resultados apresentados, os algoritmos implementados pelo FGA mostram um bom desempenho. Os testes foram feitos com grafos de 10, 30, 60, 100, 200 e 500 vértices.

Uma limitação do trabalho é a falta de uma aplicação visual e interativa para manipulação e visualização do grafo. Essa aplicação seria de utilidade didática, demonstrando um grafo e suas propriedades. O trabalho citado também é facilmente extensível, através da adição de verificação de propriedades e implementação de algoritmos.

2.3 TRABALHOS CORRELATOS

A seguir estão relacionados três trabalhos com características semelhantes aos objetivos deste trabalho: a seção 2.3.1 detalha a “Ferramenta para representação gráfica do funcionamento de algoritmos aplicados em grafos” (HACKBARTH, 2008), a seção 2.3.2 descreve “Grafos” (VILLALOBOS, 2006), e por fim, a seção 2.3.3 descreve a ferramenta JGraph (JGRAPHT TEAM, 2005).

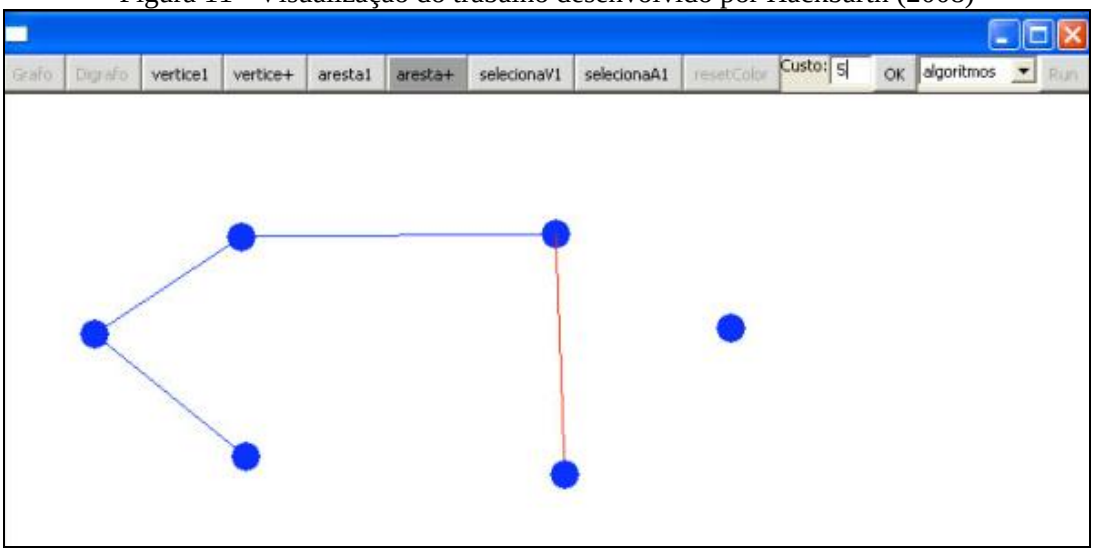
2.3.1 Ferramenta para representação gráfica do funcionamento de algoritmos aplicados em grafos

O trabalho desenvolvido por Hackbarth (2008) é uma ferramenta para representação gráfica do funcionamento de algoritmos de grafos. Os algoritmos implementados são: busca em largura, busca em profundidade e Kruskal.

A ferramenta foi desenvolvida utilizando linguagem C++, a biblioteca OpenGL para a parte visual e o *framework* GraphObj para a parte lógica. GraphObj é uma biblioteca desenvolvida pelo Departamento de Sistemas e Computação (DSC) da Universidade Regional de Blumenau (FURB) e conta com implementações em Java e C++.

O foco do trabalho era disponibilizar uma ferramenta que permita ao usuário compreender rapidamente o funcionamento dos algoritmos (Figura 11).

Figura 11 - Visualização do trabalho desenvolvido por Hackbarth (2008)



Fonte: Hackbarth (2008).

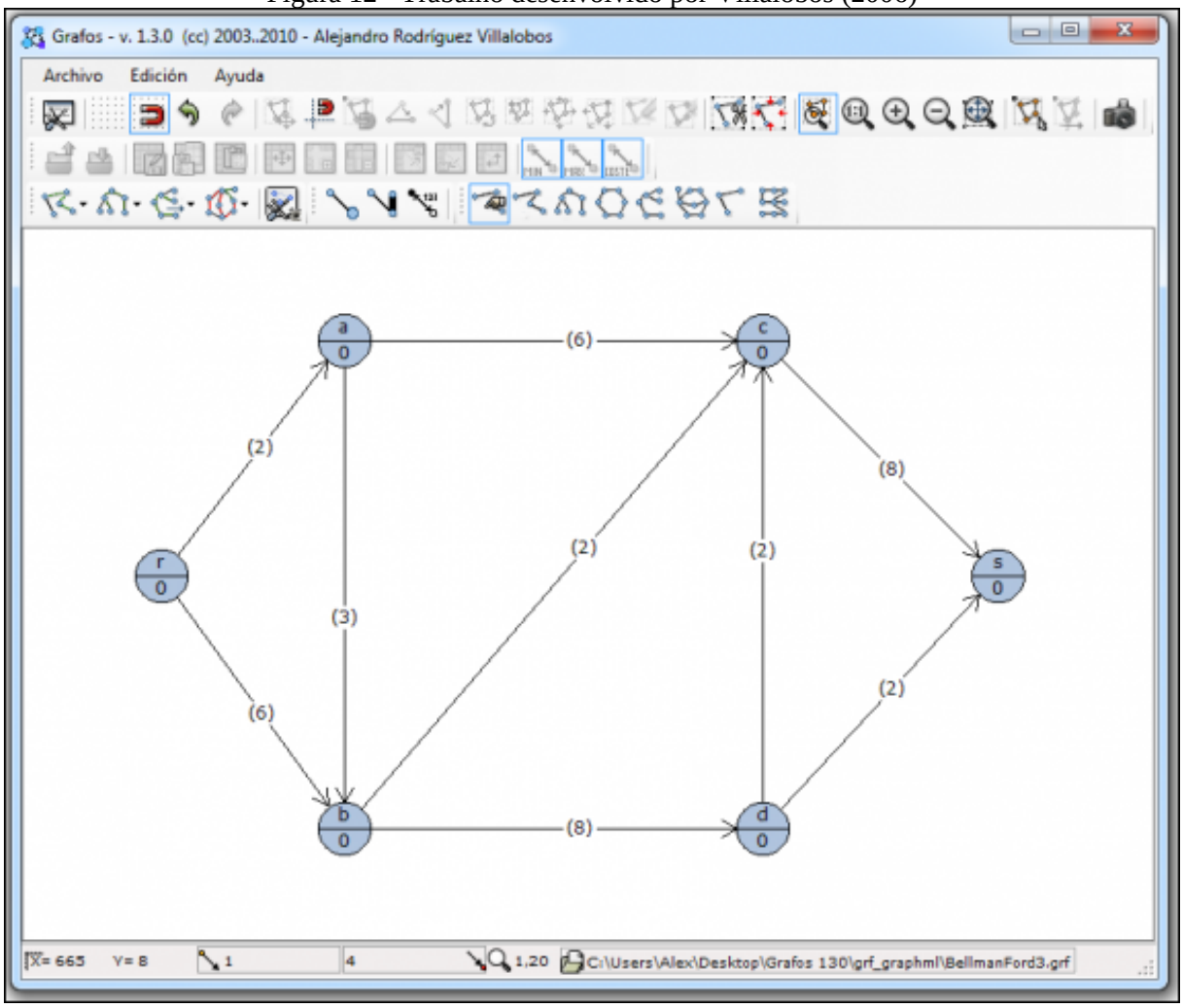
O usuário também pode visualizar a execução dos algoritmos implementados assim como o seu resultado. Porém, os algoritmos são disponibilizados somente para execução via interface gráfica, não dispondo da disponibilização de uma *Application Programming Interface* (API).

O resultado atingido foi uma interface simples que permite desenhar um grafo interativamente com o uso do *mouse*. A usabilidade da ferramenta é prejudicada pela falta de recursos pois a interface não é por si só autoexplicativa (não é clara a diferença entre as opções *vertice1* e *vertice+*) e falta de recursos básicos como selecionar vértices. O trabalho permite adicionar custo para as arestas, porém não exibe um indicativo visual do custo da aresta, o que pode dificultar o usuário da aplicação a visualizar e compreender o resultado da ação do algoritmo. O trabalho também não permite salvar e carregar grafos a partir de arquivos.

2.3.2 Grafos

O trabalho desenvolvido por Villalobos (2006) procura ser um ambiente útil para o ensino e aprendizagem a teoria dos grafos. O software foi desenvolvido em VisualBasic e possui uma boa interface e implementação para os algoritmos básicos relacionados a grafos (Figura 12). Os algoritmos implementados são: caminho mínimo/máximo, árvore mínima/máxima, fluxo máximo, ciclo euleriano e caixeiro viajante.

Figura 12 - Trabalho desenvolvido por Villalobos (2006)



Fonte: Villalobos (2006).

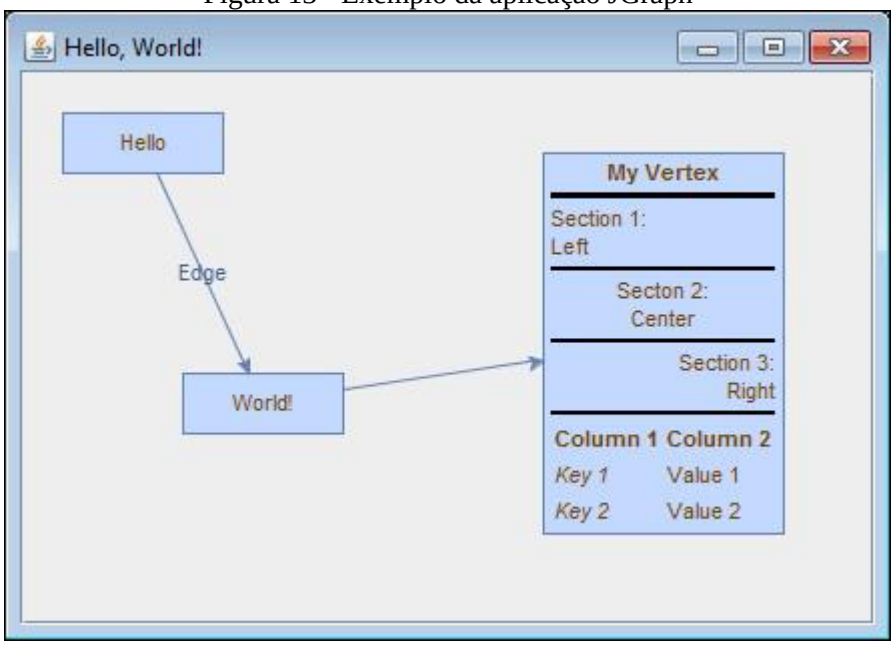
A Figura 12 exibe visualmente o resultado do desenho de um grafo com vértices e arestas nomeadas e com custos. O trabalho permite tanto a criação de grafos dirigidos e não dirigidos. Essa criação pode ser feita interativamente através da utilização do *mouse* ou através do preenchimento de uma matriz que contém a relação de nó-aresta-nó.

O trabalho de Villalobos (2006) é uma ferramenta de aprendizagem inicial de grafos pois permite editar, visualizar e executar algoritmos. Apesar da interface possuir muitos botões, a usabilidade da ferramenta não é tão boa. Por exemplo, para adicionar uma aresta entre dois vértices foi necessário ler o manual de instruções para entender como executar tal tarefa. A grande maioria dos botões da interface tem o propósito de organizar o grafo visualmente, são recursos como: alinhar todos os vértices na horizontal, vertical, dispor os vértices em formato oval, retangular, entre outras. A ferramenta porém não possui nenhuma verificação de propriedade de grafos e também não é disponibilizada na forma de API. A aplicação permite salvar e carregar grafos a partir de arquivos.

2.3.3 JGraph

O trabalho desenvolvido pela JGraph Team (2005) é um *framework* de código aberto para o desenvolvimento de aplicações utilizando grafos em Java, contendo uma API para criação e análise de grafos, além de uma API focada para visualizar e editar o grafo (Figura 13).

Figura 13 - Exemplo da aplicação JGraph



Fonte: JGraph Team (2012).

A Figura 13 mostra uma aplicação feita a partir da API disponibilizada para montar um grafo. Toda a visualização dos vértices e arestas é estática pois dependem do código do programador para suportar movimentação.

Alguns dos algoritmos implementados são: Bron-Kerbosch, Bellman-Ford, número cromático, Dijkstra, ciclo Euleriano, ciclo Hamiltoniano, Floyd-Warshall, Hopcroft-Karp e ordenação topológica. A ferramenta também possui geradores de grafos aleatórios para grafos nulos, completos, bipartidos completos, hipercubo, ciclo, estrela e roda.

A visualização do grafo é um ponto de destaque, sendo que a ferramenta pode ser adaptada para ser utilizada para os mais diversos tipos, como tais: criação de diagramas *Unified Modeling Language* (UML), criação de fluxos de *Workflow*, visualização de circuitos eletrônicos, entre outros. Como contra partida, a visualização/manipulação do grafo depende inteiramente da programação do usuário desenvolvedor da ferramenta, sendo disponibilizada somente uma API que pode ser utilizada para construir a aplicação visual.

2.3.4 Comparação entre os trabalhos correlatos

O Quadro 6 apresenta de forma comparativa algumas características dos trabalhos apresentados nesta seção.

Quadro 6 - Características dos trabalhos relacionados

características / trabalhos relacionados	Hackbarth (2008)	Villalobos (2006)	JGraph Team (2005)
foco didático	sim	sim	não
linguagem	Java	VisualBasic	Java/web
é framework	não	não	sim
possui aplicação visual nativa	sim	sim	não
aplicação visual interativa	sim	sim	sim
usabilidade da aplicação visual interativa	fraca	média	não se aplica
quantidade de algoritmos implementados	3	6	10+
possui testes para propriedades	não	não	sim
disponibiliza API	não	não	sim
exibe resultado visual da execução de um algoritmo	sim	sim	não se aplica
permite criação de novos algoritmos	não	não	sim
permite salvar e carregar grafos criados nativamente	não	sim	não

A partir do Quadro 6 percebe-se que a aplicação JGraph (JGRAPH TEAM, 2005) é a mais rica em algoritmos, enquanto os trabalhos de Hackbarth (2008) e Villalobos (2006) focam somente nos algoritmos clássicos da teoria dos grafos. Os testes de propriedades de grafos também se aplicam somente para o traablho JGraph porém fornecendo suporte somente para os testes de propriedades mais básicos. Por ser disponibilizado somente em forma de API, a aplicação JGraph não possui nativamente suporte para desenho de grafos de forma interativa, isto fica a cargo do usuário da API, assim como a funcionalidade para salvar e carregar grafos devem ser programadas pelo usuário da API. Enquanto isso, a aplicação de Hackbarth (2008) e Villalobos (2006) permite a criação de um grafo interativamente através do uso de uma interface gráfica. A aplicação de Villalobos (2006) possui uma solução mais completa para desenhar um grafo, enquanto a aplicação de Hackbarth (2008) fornece somente o básico para executar os algoritmos propostos pelo trabalho. Por fim, somente a aplicação JGraph disponibilizada em forma de API, permitindo extensões e adições de novos algoritmos e propriedades.

3 DESENVOLVIMENTO

Neste capítulo é abordado o desenvolvimento da aplicação FURB Graphs. Primeiramente são apresentados os requisitos da aplicação, seguido da sua especificação. Após, é apresentada a implementação, onde são comentadas as técnicas e ferramentas utilizadas durante o desenvolvimento da aplicação e, posteriormente, a operacionalidade da aplicação. Por fim, são discutidos os testes e resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A aplicação desenvolvida atende aos seguintes requisitos:

a) disponibilizar funções para criação e edição visual de grafos com as seguintes operações: adicionar e remover vértices e arestas, atribuir pesos e identificadores para os vértices e arestas (Requisito Funcional - RF);
b) permitir extrair as seguintes propriedades do grafo: número cromático, hipercubo e isomorfo a outro grafo (RF);
c) disponibilizar a implementação dos seguintes algoritmos: hamiltoniano e euleriano (RF);
d) ser implementado utilizando o ambiente Eclipse Luna e a linguagem Java 7 (Requisito Não-Funcional - RNF).

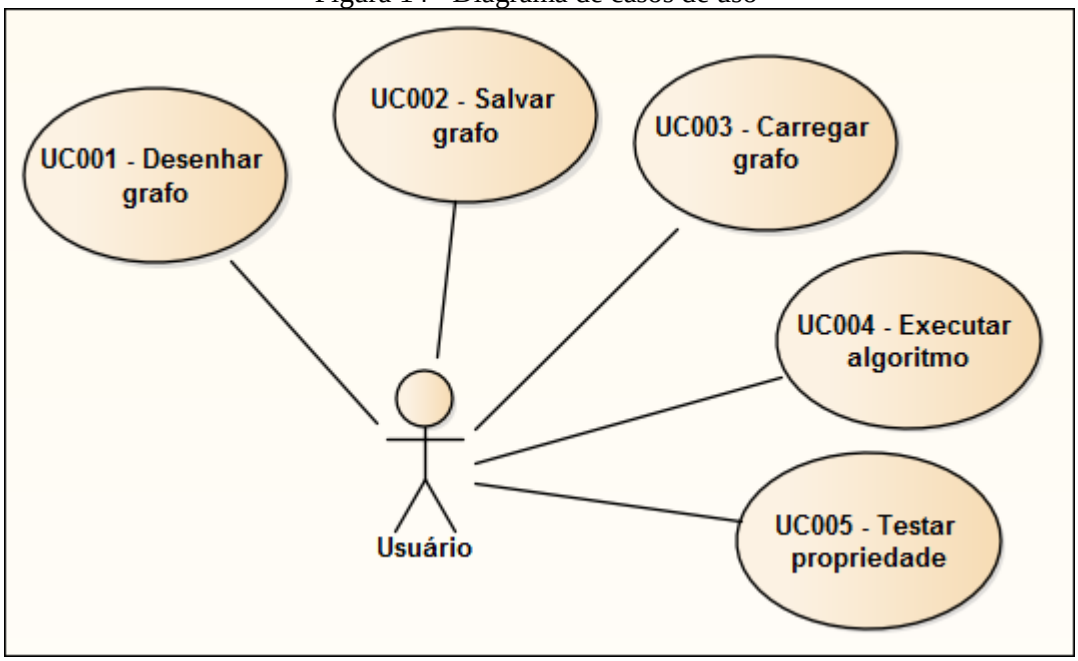
3.2 ESPECIFICAÇÃO

Esta seção apresenta a especificação do FURB Graphs através do uso dos conceitos de orientação a objetos e da UML, utilizando a ferramenta Enterprise Architect (EA). Neste trabalho foram elaborados os diagramas de casos de uso, de classes e de atividades, sendo descritos nas próximas seções.

3.2.1 Diagramas de casos de uso

A Figura 14 exibe o diagrama de casos de uso com as ações disponibilizadas pela aplicação para manipulação do grafo.

Figura 14 - Diagrama de casos de uso



O caso de uso `UC001 - Desenhar grafo` descreve a interação entre o usuário e a funcionalidade que permite o desenho de grafos. Detalhes sobre este caso de uso estão descritos no Quadro 7.

Quadro 7 - Caso de uso `UC001 - Desenhar grafo`

Número	001
Caso de Uso	Desenhar grafo
Descrição	Este caso de uso permite o desenho de um grafo com vértices, arestas e pesos.
Ator	Usuário
Cenário principal	1. O usuário abre a aplicação. 2. O usuário utiliza o mouse para adicionar um vértice na tela. 3. O usuário informa um identificador para o vértice. 4. O usuário informa um peso para o vértice. 5. O usuário seleciona os vértices que serão ligados e utiliza o mouse para criar uma aresta. 6. O usuário informa um identificador para a aresta. 7. O usuário informa um peso para a aresta.
Fluxo alternativo	No passo 2, caso o usuário deseja criar outro vértice: 2.1 Retorna ao passo 2. No passo 2, caso o usuário não deseja criar um vértice na tela: 2.2 Encerra o caso de uso. No passo 5, caso o usuário deseja criar outra aresta: 5.1 Retorna ao passo 5. No passo 5, caso o usuário não deseja conectar duas arestas: 5.2 Encerra o caso de uso.

O caso de uso `UC002 - Salvar grafo` descreve a interação entre o usuário e a funcionalidade que permite salvar grafos em um arquivo. Detalhes sobre este caso de uso estão descritos no Quadro 8.

Quadro 8 - Caso de uso `UC002 - Salvar grafo`

Número	002
Caso de Uso	Salvar grafo
Descrição	Este caso de uso permite salvar um grafo em um arquivo.
Ator	Usuário
Cenário principal	1. Usuário informa arquivo onde será salvo o grafo.
Fluxo alternativo	No passo 1, caso ocorra algum erro de I/O: 1.1 É exibido erro com a mensagem do erro.

O caso de uso `UC003 - Carregar grafo` descreve a interação entre o usuário e a funcionalidade que permite carregar grafos a partir de um arquivo. Detalhes sobre este caso de uso estão descritos no Quadro 9.

Quadro 9 - Caso de uso `UC003 - Carregar grafo`

Número	003
Caso de Uso	Carregar grafo
Descrição	Este caso de uso permite carregar um grafo a partir de um arquivo.
Ator	Usuário
Cenário principal	1. Usuário informa origem do arquivo do grafo.
Fluxo alternativo	No passo 1, caso ocorra algum erro de I/O: 1.1 É exibido erro com a mensagem do erro.

O caso de uso `UC004 - Executar algoritmo` descreve a interação entre o usuário e a funcionalidade que permite executar um algoritmo de grafos. Detalhes sobre este caso de uso estão descritos no Quadro 10.

Quadro 10 - Caso de uso `UC004 - Executar algoritmo`

Número	004
Caso de Uso	Executar algoritmo
Descrição	Este caso de uso permite executar um algoritmo de grafo sobre o grafo desenhado.
Ator	Usuário
Pré-condições	UC001 - Desenhar grafo
Cenário principal	1. Usuário solicita a execução de um algoritmo. 2. A aplicação pede ao usuário para que informe os dados de entrada do algoritmo (vértices iniciais para execução, caso necessário para o algoritmo). 3. A aplicação executa o algoritmo. 4. A aplicação exibe o resultado do algoritmo em uma tela modal.

Por fim, o caso de uso `UC005 - Testar propriedade` descreve a interação entre o usuário e a funcionalidade que permite testar uma propriedade de um grafo. Detalhes sobre este caso de uso estão descritos no Quadro 11.

Quadro 11 - Caso de uso `UC005 - Testar propriedade`

Número	005
Caso de Uso	Testar propriedade
Descrição	Este caso de uso permite testar uma propriedade sobre o grafo desenhado.
Ator	Usuário
Pré-condições	UC001 - Desenhar grafo
Cenário principal	1. Usuário solicita o teste de uma propriedade. 2. A aplicação exibe o resultado do teste da propriedade em uma tela modal.

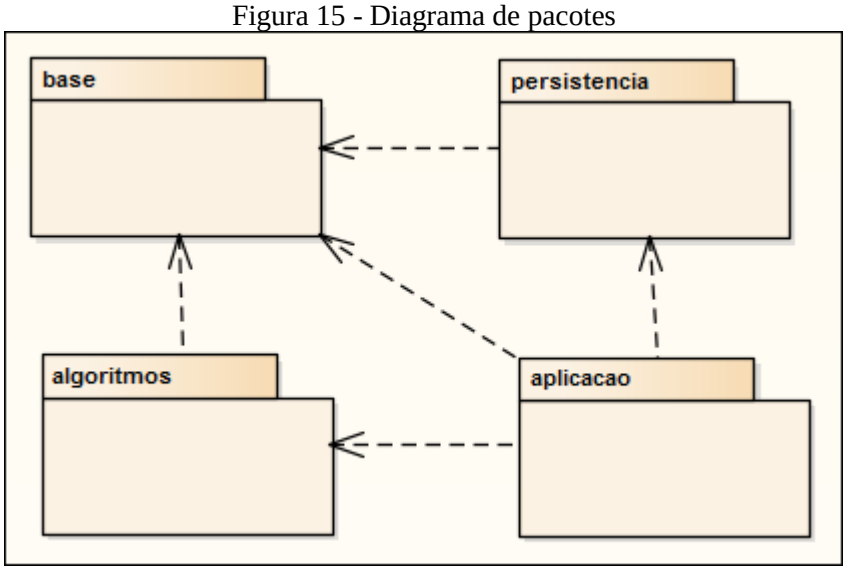
3.2.2 Diagrama de classes

Nesta seção são descritas as classes e as estruturas que compõem o funcionamento da aplicação. A estrutura de pacotes e classes segue a ideia inicialmente proposta por Zatelli (2010), somente adicionando as classes e pacotes necessários para o desenvolvimento do trabalho.

Inicialmente é apresentada a estrutura dos pacotes do trabalho, para depois detalhar o que faz cada pacote e por fim exibir os pontos de extensão no trabalho. Os pacotes desenvolvidos por este trabalho são: base, algoritmos, aplicação e persistência.

A Figura 15 apresenta a visão macro de como os pacotes estão disposto na aplicação.

Figura 15 - Diagrama de pacotes

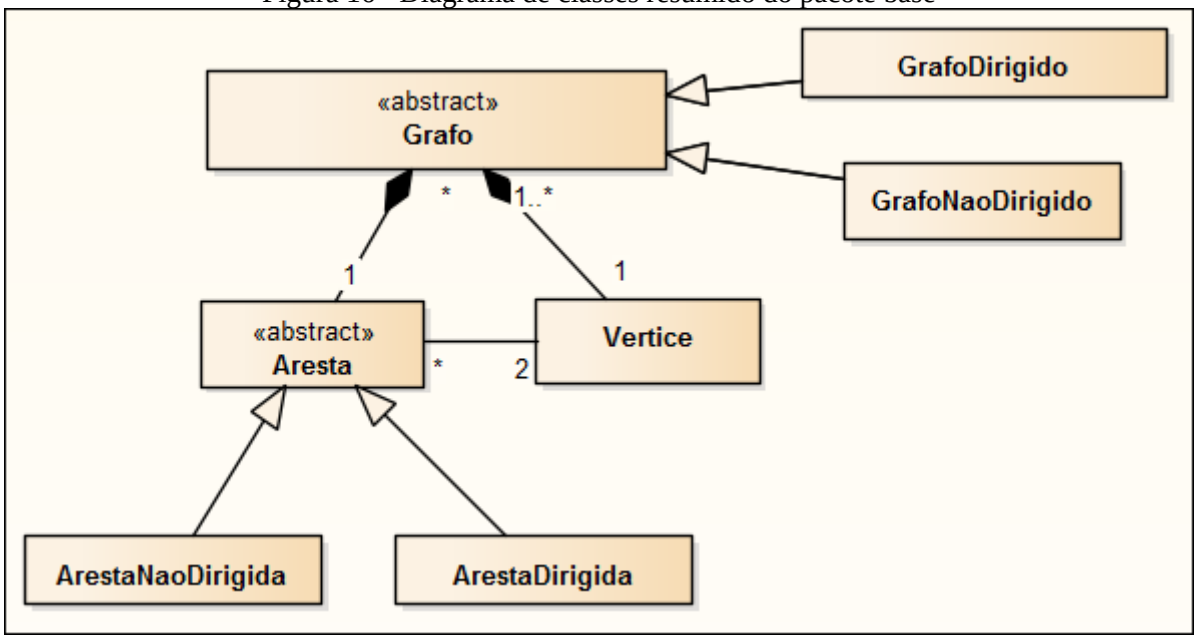


A Figura 15 exibe os relacionamentos entre os pacotes, enquanto as subseções seguintes apresentam a diagramação por classes de cada pacote.

3.2.2.1 Pacote `base`

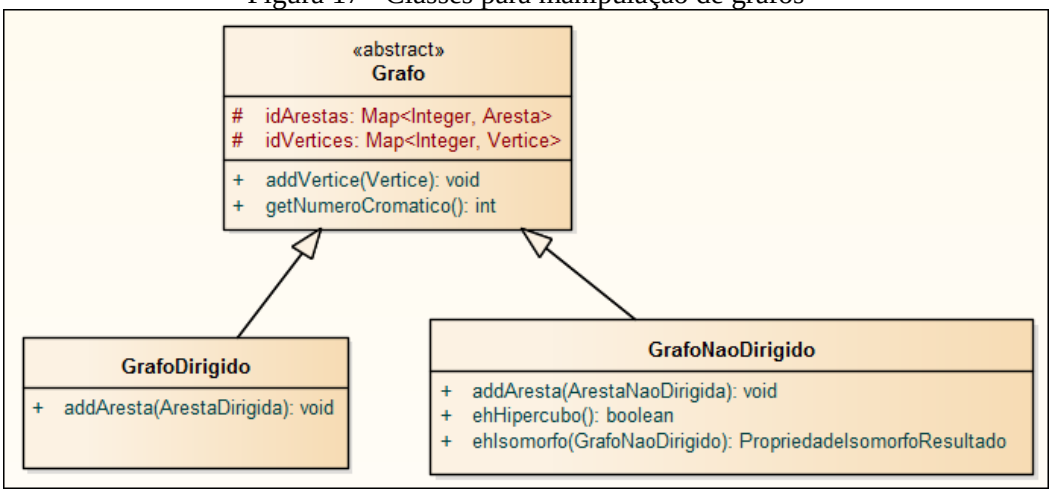
O pacote `base` é o pacote que contém as classes que compõem um grafo em sua essência, contendo as classes que representam um grafo, um vértice e uma aresta (Figura 16).

Figura 16 - Diagrama de classes resumido do pacote base



Neste pacote estão as classes que representam a estrutura de um grafo. A classe `Grafo` contém os métodos comuns a todos os grafos enquanto as classes `GrafoDirigido` e `GrafoNaoDirigido` possui os particularidades e métodos de especialização do seu tipo. São nestas classes que estão incluídas as verificações de propriedades propostas para este trabalho. Estas verificações ocorrem através de métodos correspondentes ao tipo de grafo que será testado (Figura 17).

Figura 17 - Classes para manipulação de grafos



Na Figura 17 são exibidos de forma resumida os métodos das classes de manipulação do grafo. Na classe `Grafo` tem-se o método `getNumeroCromatico` que retorna o número cromático de um grafo dirigido ou de um grafo não dirigido. Na classe `GrafoNaoDirigido` existem os métodos `ehHipercubo` e `ehIsomorfo` que retornam, respectivamente, se é um grafo é hipercubo ou isomorfo a outro grafo que fazem parte do objetivo deste trabalho.

Dentro do pacote `base`, há ainda as classes destinadas para trabalharem com arestas. A classe abstrata `Aresta` define os métodos em comum entre a classe `ArestaDirigida` (utilizada somente por grafos dirigidos) e a classe `ArestaNaoDirigida` (utilizada somente por grafos não dirigidos).

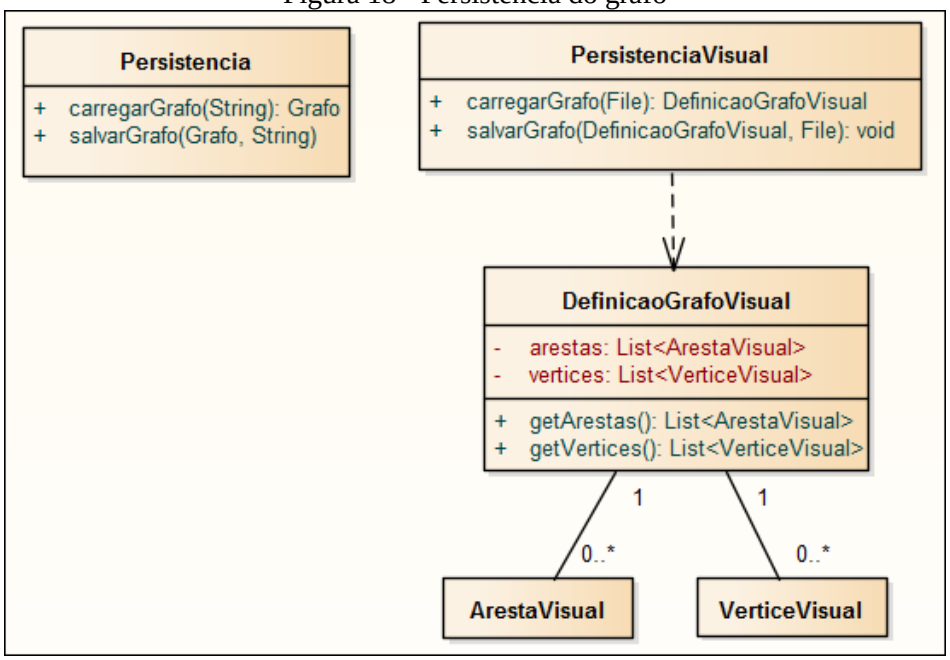
Por último, existe uma classe destinada a trabalhar com vértices, chamada `Vertice`.

3.2.2.2 Pacote `persistencia`

Neste pacote estão as classes responsáveis pela persistência em arquivo de um grafo. O trabalho desenvolvido por Zatelli (2010) possui a persistência de uma instância de uma classe `GrafoDirigido` ou `GrafoNaoDirigido` no formato XML. Porém, esta classe de persistência não pode ser utilizada porque ela não dispõe de recursos para salvar o grafo visual que está sendo exibido para o usuário da aplicação. Por este motivo, foi criada a classe `PersistenciaVisual` (Figura 18) que é capaz de ler e carregar um grafo no formato de arquivo Json.

A Figura 18 exibe as relações entre a classe de persistência visual e suas definições.

Figura 18 - Persistência do grafo



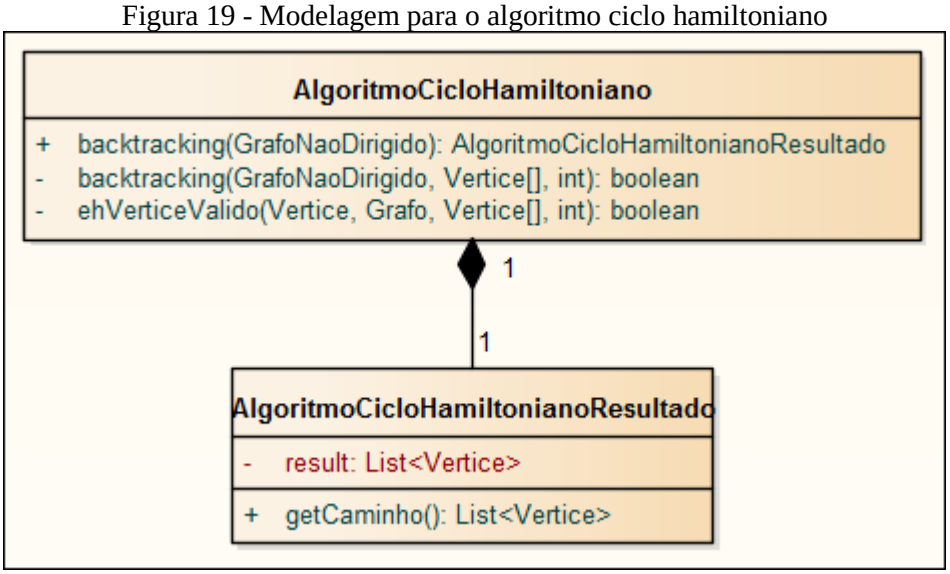
A Figura 18 exibe os modelos para persistência dos grafos. A classe `Persistencia` é a classe responsável por persistir o grafo lógico, disponibilizado pela API. A classe `PersistenciaVisual` é a classe responsável por persistir o grafo disponibilizado pela aplicação visual e interativa. Esta classe além de salvar o grafo lógico, salva também informações visuais, tais como cor do vértice, posição na tela e tamanho do diâmetro do vértice.

3.2.2.3 Pacote `algoritmos`

Este pacote tem por objetivo agrupar todos os algoritmos disponibilizados pelo FURB Graphs. A organização da classe dos algoritmos e resultados propostos seguem o modelo definido por Zatelli (2010) onde cada algoritmo da aplicação, existe uma classe que executa o algoritmo e uma classe que armazena o resultado do algoritmo obtido após a sua execução.

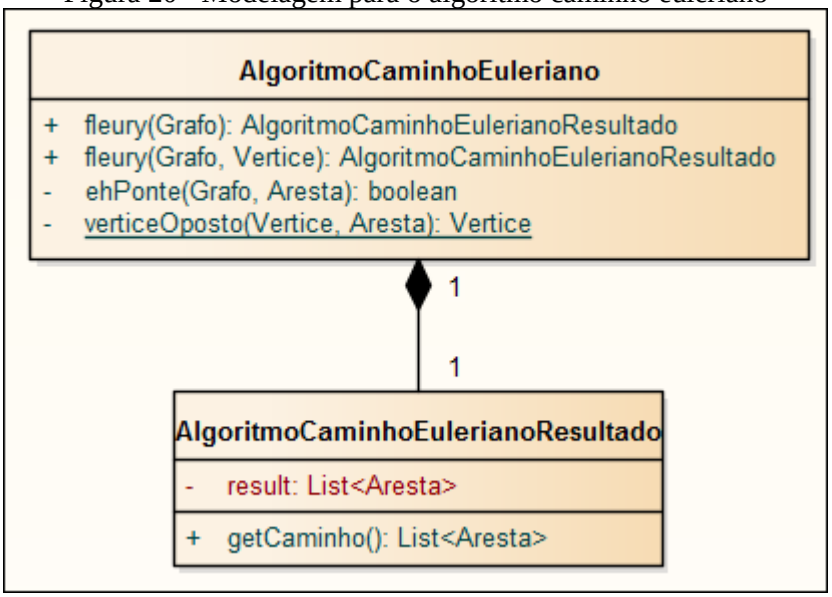
A Figura 19 exibe a modelagem das classes para o algoritmo de ciclo hamiltoniano.

Figura 19 - Modelagem para o algoritmo ciclo hamiltoniano



A Figura 19 exibe o diagrama do algoritmo de ciclo hamiltoniano. Após a execução do algoritmo, é retornada a classe `AlgoritmoCicloHamiltonianoResultado` que possui uma lista com os vértices que representam o ciclo hamiltoniano. Caso não exista ciclo hamiltoniano, é retornado `null`. A classe ainda possui dois métodos auxiliares para a execução do algoritmo: o primeiro método é o método privado `backtracking`, método recursivo responsável por caminhar pelo grafo, e o segundo método, `ehVerticeValido`, verifica se um vértice é válido para o ciclo hamiltoniano encontrado durante a execução do algoritmo. A Figura 20 exibe a modelagem das classes para o algoritmo de caminho euleriano.

Figura 20 - Modelagem para o algoritmo caminho euleriano

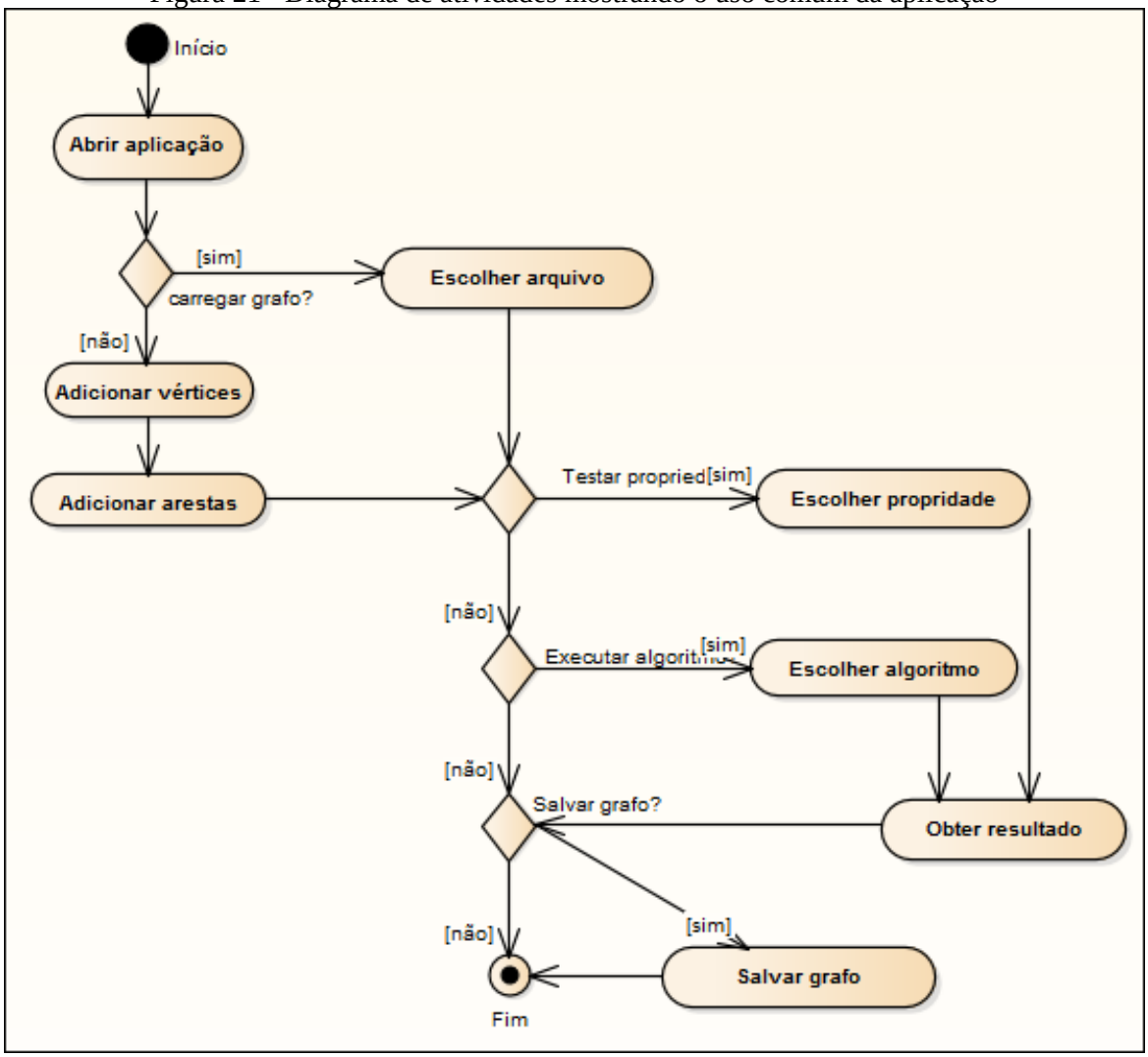


A Figura 20 exibe o diagrama do algoritmo de caminho euleriano. Após a execução do algoritmo, é retornada a classe `AlgoritmoCaminhoEulerianoResultado` que possui uma lista com as arestas que representam o caminho euleriano. Caso não exista ciclo euleriano, é retornado `null`. A classe `AlgoritmoCaminhoEuleriano` possui ainda dois métodos auxiliares e privados utilizados somente para a execução da classe: `ehPonte`, usado para verificar se uma aresta é uma ponte no grafo, e `verticeOposto`, que obtém o vértice oposto ao que está sendo verificado.

3.2.3 Diagrama de atividades

O diagrama de atividades representado na Figura 21 mostra o caminho normal para uso da aplicação.

Figura 21 - Diagrama de atividades mostrando o uso comum da aplicação



A aplicação inicia com o usuário escolhendo entre carregar um grafo pronto ou desenhar um novo grafo. Neste caso, é necessário adicionar vértices e arestas. Após isso o usuário escolhe qual propriedade verificar ou qual algoritmo executar e obtém o resultado. Por último, o usuário pode salvar o grafo para uso posterior. Este é o caminho comum da aplicação, porém é possível o usuário decidir por caminhos alternativos como por exemplo, carregar o grafo e fazer alterações nele, ou não executar algoritmos nem verificar propriedades, somente fazendo uso do desenho do grafo.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação do FURB Graphs, a descrição do desenvolvimento do trabalho e a operacionalidade da aplicação.

3.3.1 Técnicas e ferramentas utilizadas

Para codificação do trabalho, foi utilizado o ambiente de desenvolvimento Eclipse Luna na linguagem de programação Java 7, notável por sua portabilidade para as diversas plataformas suportadas.

A estrutura do software desenvolvido possui dois módulos principais: interface e algoritmos. Para o módulo de interface, foi utilizada a biblioteca gráfica *Swing*, nativa do JDK Java. Durante o desenvolvimento, foi utilizado também o formato de dados *JavaScript Object Notation* (JSON) para criação e leitura da serialização do formato visual dos grafos. Para esta parte, foi feito uso da biblioteca Gson (desenvolvida pela Google) para armazenar em tal formato.

3.3.2 Desenvolvimento do FURB Graphs

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da aplicação. A seção 3.3.2.1 apresenta o desenvolvimento dos testes das propriedades, enquanto a seção 3.3.2.2 apresenta o desenvolvimento dos algoritmos, por fim, na seção 3.3.2.3 é apresentado o desenvolvimento da aplicação visual assim como sua operacionalidade.

3.3.2.1 Propriedades

As propriedades exploradas por este trabalho serão apresentadas nas seções seguintes. A seção 3.3.2.1.1 apresenta a propriedade de verificação se um grafo é hipercubo, a seção 3.3.2.1.2 verifica o número cromático de um grafo e por fim, a seção 3.3.2.1.3 verifica o isomorfismo entre grafos.

3.3.2.1.1 Hipercubo

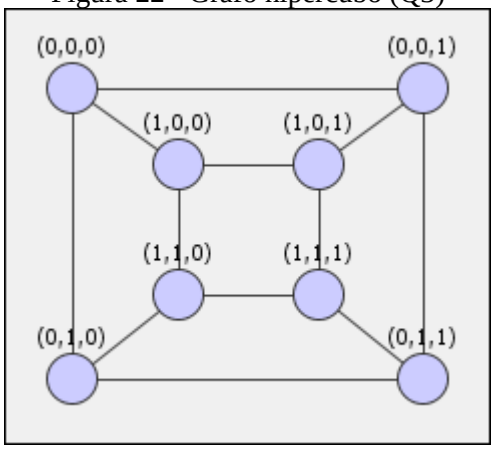
A implementação para testar se um determinado grafo é hipercubo baseia-se no conceito de exaurir todas as possibilidades que invalidam um grafo de ser hipercubo, conforme explicado na seção 2.1.2.2. Existem três condições que identificam um grafo como sendo hipercubo, se uma delas não for verdadeira, é retornado `false`, caso todas sejam verdadeiras, é retornado `true` (Quadro 12).

Quadro 12 - Propriedade hipercubo

```
01. /**
02.  * Retorna <code>true</code> se um grafo é hipercubo
03.  * e <code>false</code> caso contrário.
04.  *
05.  * @return <code>true</code> se um grafo é hipercubo
06.  *          e <code>false</code> caso contrário
07.  */
08. public boolean ehHipercubo() {
09.     int tamanho = getTamanho(), n;
10.     for (n = 0;; n++) {
11.         double pow = Math.pow(2, n);
12.         if (pow == tamanho) {
13.             break;
14.         }
15.         if (pow > tamanho) { // não é uma potência de n.
16.             return false;
17.         }
18.     }
19.
20.     for (int i = 0; i < tamanho; i++) {
21.         Vertice vertice = getVertice(i);
22.         if (vertice.getQtdeArestas() != n) {
23.             return false;
24.         }
25.
26.         // verifica a ausência de arestas parelas e laços nos vértices
27.         Set<Integer> set = new HashSet<Integer>();
28.         for (int j = 0; j < vertice.getQtdeArestas(); j++) {
29.             Aresta aresta = vertice.getAresta(j);
30.             Vertice verticeOposto =
31.               (aresta.getVi().getId() == vertice.getId()) ?
32.                aresta.getVj() : aresta.getVi();
33.
34.             if (verticeOposto.getId() == vertice.getId()) {
35.                 return false;
36.             }
37.             if (set.contains(aresta.getVj().getId())) {
38.                 return false;
39.             }
40.         }
41.     }
42.     return true;
43. }
```

Entre as linhas 9 e 18 é verificado se a quantidade de vértices do grafo é uma potência de dois, requisito básico para um grafo hipercubo. Caso não seja uma potência de dois, é retornado `false`. Após isso, entre as linhas 20 e 41 são feitas duas verificações para os vértices que compõem o grafo. Primeiramente, o `if` da linha 22 verifica se o grau de cada vértice é `n`, sendo `n` o valor da potência de dois identificado previamente. Caso não seja, é retornado `false`. Por último, o código entre as linhas 27 e 40 garante que não existem arestas paralelas e nem laços para as arestas de um vértice. Se todos os casos citados acima forem verdadeiros, a linha 42 retorna `true` indicando que o grafo possui a propriedade hipercubo. A Figura 22 exibe o grafo hipercubo conhecido por `Q3`.

Figura 22 - Grafo hipercubo (Q3)



Ao aplicar o algoritmo para o grafo descrito na Figura 22 ele identificará que o valor para a variável `n` do algoritmo proposto é 3 (dando origem ao nome do grafo de `Q3`) em decorrência da quantidade de vértices (8 vértices). Após a identificação do valor de `n` o algoritmo deverá verificar se todos os vértices possuem tal grau e também deverá verificar a ausência de laços e arestas paralelas para cada vértice, tais propriedades podem ser visualmente verificadas a partir da Figura 22.

3.3.2.1.2 Número cromático

A solução proposta para identificar o número cromático de um grafo é uma solução baseado em algoritmos gulosos que roda em tempo polinomial, porém, sem a garantia de trazer a solução ótima. Algoritmos gulosos são um tipo de algoritmo onde eles escolhem uma solução local ótima, a fim de montar a solução global com a partir das soluções locais (ROCHA; DORINI, 2014). Feofiloff, Kohayakawa e Wakabayashi (2014) ainda citam que algoritmos gulosos são “míopes” pois tomam as decisões baseados nas informações disponíveis correntemente, sem levar em consideração as consequências dessas decisões para o futuro.

A solução proposta para encontrar o número cromático utiliza-se deste artifício guloso para encontrar a solução global, não ótima, porém aproximada, conforme explicado na seção 2.1.2.1. A ideia por trás do algoritmo é montar a solução global através de cada decisão local, no caso, cada decisão local é definir a cor de um vértice no momento da sua iteração (Quadro 13). A cada iteração é verificado qual é a primeira cor disponível para uso para o vértice em questão. A decisão é tomada com base na melhor cor para ser utilizada localmente, observando que a melhor cor que pode ser utilizada localmente não é necessariamente a melhor cor para a solução global.

Quadro 13 - Propriedade número cromático

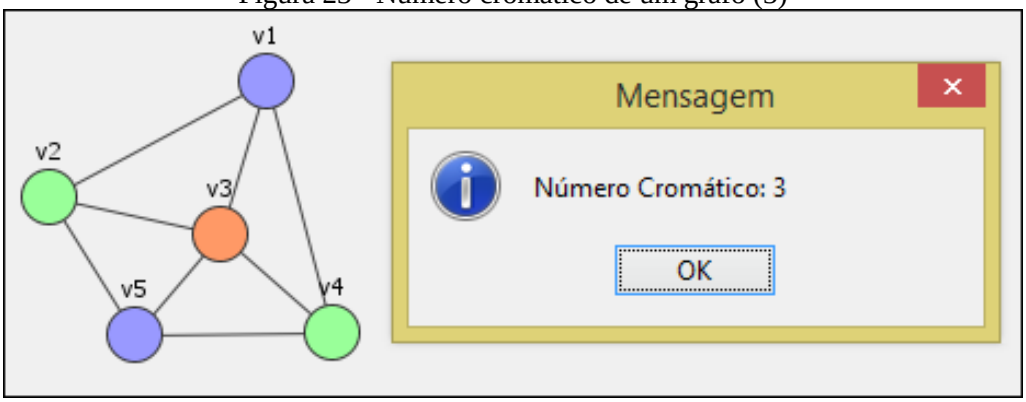
```
01. /**
02.  * Retorna um mapa com uma cor associada para cada vértice.
03.  * <p>
04.  * O algoritmo utilizado é baseado em uma solução gulosa que não
05.  * traz o resultado ótimo, porém resultados aproximados.
06.  *
07.  * @return um mapa com uma cor associada para cada vértice
08.  */
09. public Map<Vertice, Integer> getNumeroCromatico() {
10.     if (getTamanho() == 0) {
11.         return Collections.emptyMap();
12.     }
13.
14.     Map<Vertice, Integer> cores = new LinkedHashMap<>();
15.     cores.put(getVertice(0), 0); // atribui a cor inicial
16.
17.     boolean[] coresUtilizadas = new boolean[getTamanho()];
18.
19.     for (int u = 1; u < getTamanho(); u++) {
20.         // encontra as cores dos vértices adjacentes já coloridos
21.         List<Vertice> adjacentes = getAdjacentes(getVertice(u));
22.         for (Vertice adjacente : adjacentes) {
23.             if (cores.containsKey(adjacente)) {
24.                 coresUtilizadas[cores.get(adjacente)] = true;
25.             }
26.         }
27.
28.         // encontra a primeira cor disponível
29.         int cor;
30.         for (cor = 0; cor < getTamanho(); cor++) {
31.             if (!coresUtilizadas[cor]) {
32.                 break;
33.             }
34.         }
35.         cores.put(getVertice(u), cor); // atribui a cor encontrada
36.
37.         Arrays.fill(coresUtilizadas, false);
38.     }
39.     return cores;
40. }
```

As cores são abstraídas como um inteiro de 0 até o tamanho do grafo (caso for necessário utilizar todas as cores possíveis).

Inicialmente, a linha 10 verifica se o grafo é vazio, caso for, é retornado um mapa vazio. Após isso, a linha 15 inicia a solução definindo o vértice do índice zero como possuindo a cor 0. O mapa chamado de cores é responsável por armazenar a relação entre

vértice do mapa e o inteiro que representa uma cor. Seguindo, a linha 17 instancia o vetor de `boolean` que é responsável por identificar cada cor utilizada a cada iteração, sendo a linha 37 responsável por reinicializar o mapa para o seu estado inicial (com todas as cores com o valor `false` - não utilizada). A estrutura de controle `for` entre as linhas 19 e 38 representa cada iteração gulosa do algoritmo. A cada iteração é verificado quais são as cores utilizadas pelos vértices adjacentes ao vértice em questão, essas cores são marcadas como utilizadas na linha 24. Após esta verificação, a estrutura de controle `for` da linha 30 tem por finalidade identificar qual é a primeira cor disponível para uso, fazendo uso da verificação do vetor de cores utilizadas. Ao encontrar a primeira cor (ou não encontrar, neste caso recebe o valor máximo da cor disponível), é feita a associação no mapa de cores com a cor encontrada. A Figura 23 exibe o número cromático de um grafo.

Figura 23 - Número cromático de um grafo (3)



A execução do algoritmo precisa de um vértice inicial qualquer para iniciar o algoritmo. Para a Figura 23 será utilizado o vértice `v1` como vértice inicial. Inicialmente é definida a cor para o vértice inicial, no caso, para o vértice `v1` será atribuída a cor zero (neste exemplo, a cor zero equivale a cor roxa). Em seguida, é tomado um outro vértice qualquer do grafo para ser definida a cor. Neste exemplo, a ordem dos vértices navegados equivale a sua nomeação, portanto será visitado os vértices por sequência numérica (em ordem: `v1`, `v2`, `v3`, `v4` e `v5`). O próximo vértice `v2` deverá verificar todas as cores utilizadas pelos vértices vizinhos. Como é o segundo vértice ainda, um vértice possui uma cor definida (a cor zero) e os outros dois vértices ainda não possuem cores definidas, portanto para este vértice, o procedimento guloso obtém a próxima cor disponível: cor um (representado neste exemplo pela cor verde). O próximo vértice visitado é o vértice `v3`. Ao realizar o procedimento guloso, será verificado que as cores zero e um já foram utilizada, e que os vértices `v4` e `v5` não possuem cores definidas. Portanto, é utilizada a cor dois (representado pela cor laranja). O vértice `v4` iniciará o mesmo procedimento e identificará que as cores zero e dois já foram

utilizadas e que existe um vértice adjacente sem cor, portanto ele obtém a próxima cor disponível para ele: a cor um (verde). Por fim, todos os vértices adjacentes de `v5` já possuem cores definidas (cores um e dois) e é obtido a primeira cor disponível (cor zero).

3.3.2.1.3 Isomorfismo entre grafos

Isomorfismo de grafos é um dos poucos problemas em aberto da computação que não se sabe se é pertencente a classe de problemas P ou NP (GAREY; JOHNSON, 1983). Para este trabalho, adotou-se o algoritmo que traz a solução ótima, porém o algoritmo não executa em tempo polinomial.

A ideia da solução proposta é verificar todas as combinações possíveis entre os dois grafos e, caso possível, retornar a relação de arestas e vértices que equivalem entre ambos. Para implementação, foi feito uso de *backtracking* e navegação em profundidade, conforme explicado na seção 2.1.2.3.

Antes de realizar o *backtracking* são realizadas algumas verificações que já invalidam os grafos de serem isomorfos (Quadro 14).

Quadro 14 - Pré-verificações de isomorfismo

```
01. // a) compara o tamanho
02. if (g1.getTamanho() != g2.getTamanho()) {
03.     return null;
04. }
05.
06. // b) compara as arestas
07. if (g1.getQtdeArestas() != g2.getQtdeArestas()) {
08.     return null;
09. }
10.
11. // c) compara o grau de cada aresta
12. Vertice[] grau1 = new Vertice[g1.getTamanho()];
13. for (int i = 0; i < grau1.length; i++) {
14.     grau1[i] = g1.getVertice(i);
15. }
16. Arrays.sort(grau1, ARESTAS_COMPARATOR);
17.
18. Vertice[] grau2 = new Vertice[g2.getTamanho()];
19. for (int i = 0; i < grau2.length; i++) {
20.     grau2[i] = g2.getVertice(i);
21. }
22. Arrays.sort(grau2, ARESTAS_COMPARATOR);
23.
24. for (int i = 0; i < grau1.length; i++) {
25.     if (grau1[i].getQtdeArestas() != grau2[i].getQtdeArestas()) {
26.         return null;
27.     }
28 }
```

No Quadro 14 é validado se a quantidade de vértices entre os dois grafos é a mesma (linha 2), se a quantidade de arestas é a mesma (linha 7) e por fim, se o grau de cada vértice é o mesmo entre os grafos (linha 25).

Após os procedimentos básicos, é iniciado o *backtracking* a partir dos vértices com os maiores grau (Quadro 15).

Quadro 15 - Início do *backtracking* para isomorfismo

```
01. // d) backtracking
02. // para todos que tem o mesmo grau, verifica isomorfismo
03. Vertice exemplo = grau1[0];
04. for (int i = 0;
05.          exemplo.getQtdeArestas() == grau2[i].getQtdeArestas(); i++) {
06.     if (backtracking(g1, g2, exemplo, grau2[i],
07.              new ArrayList<Vertice>(), new ArrayList<Aresta>())) {
08.         return new PropriedadeIsomorfismoResultado(
09.                                              relacao_a, relacao_v);
10.     }
11. }
12. return null;
```

No Quadro 15 é verificado para cada vértice com o mesmo grau se a operação de *backtracking* é bem sucedida. Em caso positivo, é retornando o resultado com as relações entre vértices e arestas entre os dois grafos.

A operação de *backtracking* é o núcleo da operação de isomorfismo. A operação é dividida em três fases: verificação se é válido continuar a busca, continuação da busca em profundidade recursiva a fim de encontrar o grafo correspondente e por último, realização do *backtracking*, caso necessário. A primeira parte é apresentada no Quadro 16.

Quadro 16 - Primeira parte do *backtracking*

```
01. private boolean backtracking(GrafoNaoDirigido g1,GrafoNaoDirigido g2,
02.           Vertice v1, Vertice v2, List<Vertice> verticesAdicionados,
03.                                   List<Aresta> arestasAdicionadas) {
04.     // se já foi visitado, a relação deve ser a mesma
05.     if (relacao_v.containsKey(v1)) {
06.         return relacao_v.get(v1) == v2;
07.     }
08.     if (relacao_v.containsValue(v2)) {
09.         return false; // já existe relação para tal vértice
10.     }
11.     if (v1.getQtdeArestas() != v2.getQtdeArestas()) {
12.         return false;
13.     }
```

Existem duas estruturas auxiliares, `relacao_v` que é um mapa que armazena as relações entre o vértice do grafo 1 com o grafo 2 e `relacao_a` que de maneira similar, armazena as relações de arestas entre os dois grafos. O código do Quadro 16 é responsável por verificar se os vértices que serão comparados são candidatos a serem isomorfos. As três condições são: caso já tenha sido visitado é necessário que os vértices relacionados sejam os mesmos (linha 5), caso o vértice candidato já tenha sido adicionado em outra relação retorna como sendo candidato inválido (linha 8) e por fim, a quantidade de arestas deve ser a mesma,

caso não seja, retorna `false` (linha 11). A segunda parte apresenta a busca em profundidade (Quadro 17).

Quadro 17 - Segunda parte do *backtracking*

```
01. // adicionamos ambos na mesma relação
02. relacao_v.put(v1, v2);
03. verticesAdicionados.add(v1);
04.
05. List<Vertice> _verticesAdicionados_1 = new ArrayList<>();
06. List<Aresta> _arestasAdicionadas_1 = new ArrayList<>();
07.
08. // comparamos todas as arestas agora que ainda não foram visitadas
09. out: for (int i = 0; i < v1.getQtdeArestas(); i++) {
10.     Aresta aresta1 = v1.getAresta(i);
11.     if (relacao_a.containsKey(aresta1)) {
12.         continue; // aresta já visitada, então continua
13.     }
14.     // procura por todas as arestas não visitadas
15.     for (int j = 0; j < v2.getQtdeArestas(); j++) {
16.         Aresta aresta2 = v2.getAresta(j);
17.         if (relacao_a.containsValue(aresta2))
18.             continue; // aresta já visitada, então continua
19.         // se ambos possuem o mesmo grau, continua a busca
20.         Vertice v1_j = otherside(aresta1, v1);
21.         Vertice v2_j = otherside(aresta2, v2);
22.         if (v1_j.getQtdeArestas() != v2_j.getQtdeArestas())
23.             continue;
24.         List<Vertice> _verticesAdicionados_2 = new ArrayList<>();
25.         List<Aresta> _arestasAdicionadas_2 = new ArrayList<>();
26.
27.         relacao_a.put(aresta1, aresta2);
28.         arestasAdicionadas.add(aresta1);
29.
30.         if (backtracking(g1, g2, v1_j, v2_j,
31.                 _verticesAdicionados_2, _arestasAdicionadas_2)) {
32.             _verticesAdicionados_1.addAll(_verticesAdicionados_2);
33.             _arestasAdicionadas_1.addAll(_arestasAdicionadas_2);
34.             continue out;
35.         }
36.         // desfaz o backtracking
37.         for (Vertice vertice : _verticesAdicionados_2) {
38.             relacao_v.remove(vertice);
39.         }
40.         for (Aresta aresta : _arestasAdicionadas_2) {
41.             relacao_a.remove(aresta);
42.         }
43.     }
```

Primeiramente é adicionada uma relação entre os vértices que serão testados indo pelo caminho otimista, caso comprove-se que não é um vértice válido, o mesmo é removido posteriormente. Esta parte é responsável por dar continuidade a navegação de cada aresta do grafo, a mesma navegação feita no primeiro grafo, será relacionada ao segundo grafo. Esta parte possui quatro variáveis locais importantes: _verticesAdicionados_1 e _arestasAdicionadas_1 são responsáveis por armazenar todos os vértices e arestas do caminho percorrido pelo vértice atual, sua utilidade é poder desfazer as operações feitas que caminham entre os dois grafos, as outras duas variáveis são _verticesAdicionados_2 e _arestasAdicionadas_2 que seguem a mesma ideia das variáveis anterior, porém seu contexto é mais restrito, sendo que é utilizada para desfazer as operações de comparações entre a navegação de duas arestas. A terceira parte é responsável por verificar se uma solução foi encontrada (Quadro 18).

Quadro 18 - Terceira parte do *backtracking*

```
01. // se chegar aqui, então não encontrou caminho para válido
02. // para a aresta, desfaz e continua
03. for (Vertice vertice : _verticesAdicionados_1) {
04.     relacao_v.remove(vertice);
05. }
06. for (Aresta aresta : _arestasAdicionadas_1) {
07.     relacao_a.remove(aresta);
08. }
09. return false;
```

O algoritmo do Quadro 18 verifica se todas as arestas possuem relações preenchidas. Em caso positivo, retorna true (linha 9).

3.3.2.2 Algoritmos

Os algoritmos implementados por este trabalho são apresentadas nas seções seguintes. A seção 3.3.2.2.1 apresenta a implementação de um algoritmo para verificação de um caminho euleriano, enquanto a seção 3.3.2.2.2 apresenta a implementação de verificação de um ciclo hamiltoniano.

3.3.2.2.1 Caminho Euleriano

O algoritmo implementado para encontrar um caminho euleriano é o algoritmo simples de Fleury, conforme explicado na seção 2.1.3.2. A implementação é dividida em três fases simples: início da detecção se é um grafo com um caminho euleriano, busca do caminho, validação se encontrou o caminho. A primeira parte é opcional caso seja utilizado o método

que passe um vértice inicial para ser percorrido, porém, neste caso o vértice fornecido deve ter a garantia que é válido perante ao algoritmo.

A primeira parte consiste em encontrar um vértice inicial para iniciar a busca. O vértice inicial deve possuir um grau ímpar para execução correta do algoritmo de Fleury (Quadro 19).

Quadro 19 - Identificação do vértice inicial

```
01. // escolha do vértice inicial
02. int quantidadeVerticesImpares = 0;
03. for (int i = g.getTamanho() - 1; i >= 0; i--) {
04.     Vertice vertice = g.getVertice(i);
05.     if (vertice.getQtdeArestas() % 2 == 1) {
06.         quantidadeVerticesImpares++;
07.     }
08. }
09. if (quantidadeVerticesImpares != 0
10.       && quantidadeVerticesImpares != 2) {
11.     return null;
12. }
```

O Quadro 19 apresenta a escolha do vértice inicial e também apresenta uma propriedade do algoritmo de Fleury. A quantidade de vértices ímpares no grafo deve ser zero ou dois para existir um caminho válido. Se for ímpar, deve-se começar por um eles.

Após a identificação do vértice, é executado o `while` principal que busca o caminho euleriano (Quadro 20).

Quadro 20 - Busca do caminho euleriano

```
01. Vertice verticeAtual = verticeInicial;
02. while (g.getQtdeArestas() != 0) {
03.     Aresta aresta = null;
04.
05.     if (verticeAtual.getQtdeArestas() == 1) {
06.         aresta = verticeAtual.getAresta(0);
07.     } else {
08.         for (int i = 0; i < verticeAtual.getQtdeArestas(); i++) {
09.             Aresta outraAresta = verticeAtual.getAresta(i);
10.             if (!ehPonte(g.clone(), outraAresta)) {
11.                 aresta = outraAresta;
12.                 break;
13.             }
14.         }
15.         if (aresta == null) {
16.             break;
18.         }
19.     }
20.
21.     result.add(aresta);
22.
23.     g.delAresta(aresta.getId());
24.
25.     if (aresta.getVi().getQtdeArestas() == 0) {
26.         g.delVertice(aresta.getVi().getId());
27.     }
28.     if (aresta.getVj().getQtdeArestas() == 0) {
29.         g.delVertice(aresta.getVj().getId());
30.     }
31.
32.     verticeAtual = verticeOposto(verticeAtual, aresta);
33. }
```

Seguindo com o algoritmo de Fleury, é escolhida uma aresta qualquer do vértice atual, desde que este aresta não seja uma ponte, pois só são visitadas em últimos casos. Após escolher uma aresta, é feita a remoção desta aresta do grafo e é feito o mesmo procedimento para o vértice destino ao removido. Por último, vem a parte que verifica se todas as arestas foram percorridas (Quadro 21).

Quadro 21 - Verificação das arestas e resultado

```
01. if (g.getQtdeArestas() > 0 || g.getTamanho() > 0) {
02.     return null;
03. }
04. return new AlgoritmoCaminhoEulerianoResultado(result);
```

O Quadro 21 mostra que caso todas as arestas tenham sido percorridas, retorna o resultado com uma lista das sequências em que as arestas foram visitadas, caso contrário, retorna `null`.

3.3.2.2.2 Ciclo Hamiltoniano

Ciclo hamiltoniano é um problema NP-completo. Portanto, não se conhece ainda algoritmos capazes de resolver este problema em tempo polinomial. Para este trabalho, foi adotado a estratégia de *backtracking* onde são tentadas todas as combinações possíveis para encontrar o resultado, conforme explicado na seção 2.1.3.1. Embora o algoritmo seja bastante lento e por muitas vezes inviável, o resultado obtido é o resultado ótimo.

O algoritmo produzido é relativamente simples e dividido em duas etapas: início da função de *backtracking* e realização do *backtracking.* Para iniciar o algoritmo, é escolhido um vértice qualquer (Quadro 22).

Quadro 22 - Início da função de *backtracking*

```
01. public AlgoritmoCicloHamiltonianoResultado
02.                                     backtracking(GrafoNaoDirigido g) {
03.     if (g.getTamanho() == 0) {
04.         return null;
05.     }
06.
07.     Vertice[] caminho = new Vertice[g.getTamanho()];
08.
09.     caminho[0] = g.getVertice(0);
10.     if (!backtracking(g, caminho, 1)) {
11.         return null;
12.     }
13.     return
14.       new AlgoritmoCicloHamiltonianoResultado(Arrays.asList(caminho));
15. }
```

É iniciado o vetor caminho. Este vetor é a estrutura dinâmica que é preenchida com o caminho do ciclo hamiltoniano. Ao final é aplicada a função de *backtracking*. Caso seja encontrada uma solução, retorna a solução, caso contrário, retorna `null`. A função de *backtracking* é apresentada na Quadro 23.

Quadro 23 - *Backtracking* para ciclo hamiltoniano

```
01. private boolean backtracking(GrafoNaoDirigido g,
02.                              Vertice[] caminho, int i) {
03.     if (i == g.getTamanho()) { // encontrou todos os vértices
04.         // verifica se existe aresta entre o último e o primeiro.
05.         // isto para fechar o ciclo hamiltoniano
06.         return g.ehAdjacente(caminho[i - 1], caminho[0]);
07.     }
08.
09.     for (int v = 1; v < g.getTamanho(); v++) {
10.         Vertice vertice = g.getVertice(v);
11.         if (ehVerticeValido(vertice, g, caminho, i)) {
12.             caminho[i] = vertice;
13.
14.             if (backtracking(g, caminho, i + 1)) {
15.                 return true;
16.             }
17.             caminho[i] = null;
18.         }
19.     }
20.
21.     // caso nenhum vértice seja passível de entrar no ciclo
22.     return false;
23. }
```

Inicialmente é verificado se todos os vértices já foram visitados (linha 3). Caso sim, retorna se o último e o primeiro vértices são adjacentes. O comando `for` principal da função é apresentado na linha 9. Nele é iterado por cada vértice a fim de verificar se é um caminho válido e se existe um caminho a partir dele. Para cada vértice, é verificado se é um vértice válido (Quadro 24).

Quadro 24 - Verificação ser um vértice é válido

```
01. private static boolean ehVerticeValido(Vertice vertice, Grafo grafo,
02.                                        Vertice[] caminho, int i) {
03.     if (!grafo.ehAdjacente(vertice, caminho[i - 1])) {
04.         return false;
05.     }
06.     // verifica se o vértice já não está incluso
07.     for (int j = 0; j < caminho.length; j++) {
08.         if (caminho[j] == vertice) {
09.             return false;
10.         }
11.     }
12.     return true;
13. }
```

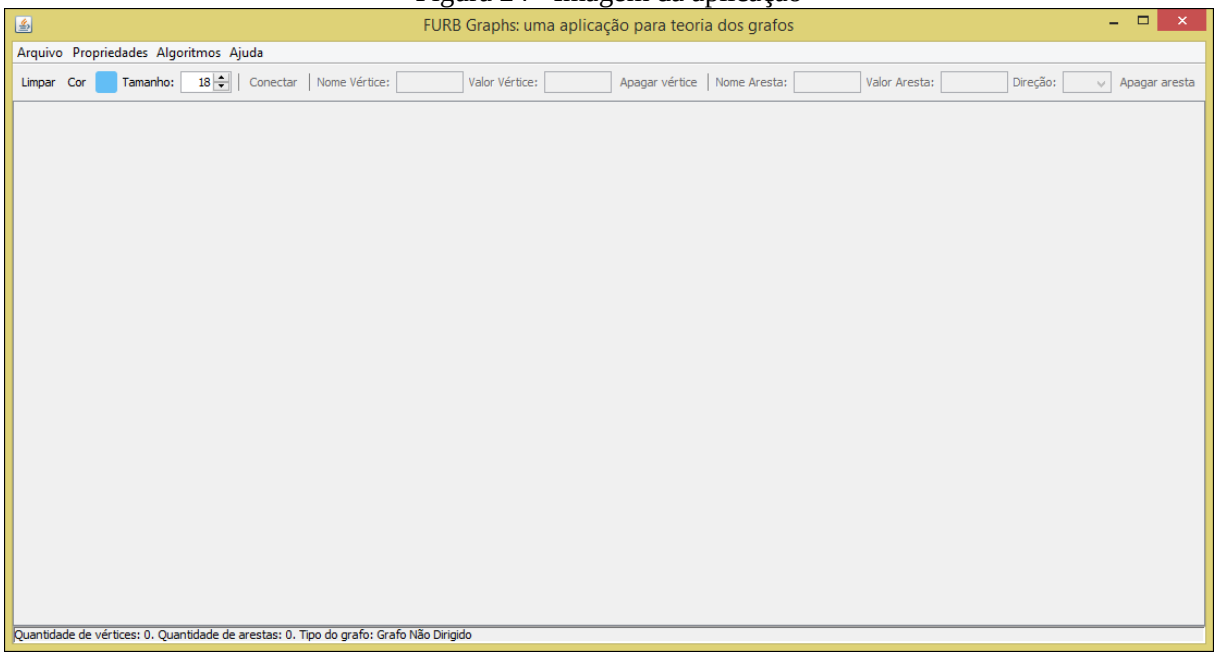
Um vértice válido consiste em vértice adjacente ao vértice anterior (linha 3) e que ainda não tenha sido visitado (linha 8). Caso o vértice seja válido, continua o *backtracking* a

partir dos vértices não visitados, caso contrário, limpa a referência ao vértice atual e verifica os demais.

3.3.3 Operacionalidade da implementação

O desenvolvimento da aplicação visual foi feito visando fornecer para o usuário uma tela auto explicativa que permita utilizar os recursos das aplicação FURB Graphs. A aplicação permite desenhar um grafo por vez e é constituída por quatro menus, uma barra para manipulação dos vértices e arestas, uma área interativa para desenho do grafo e uma barra de status (Figura 24).

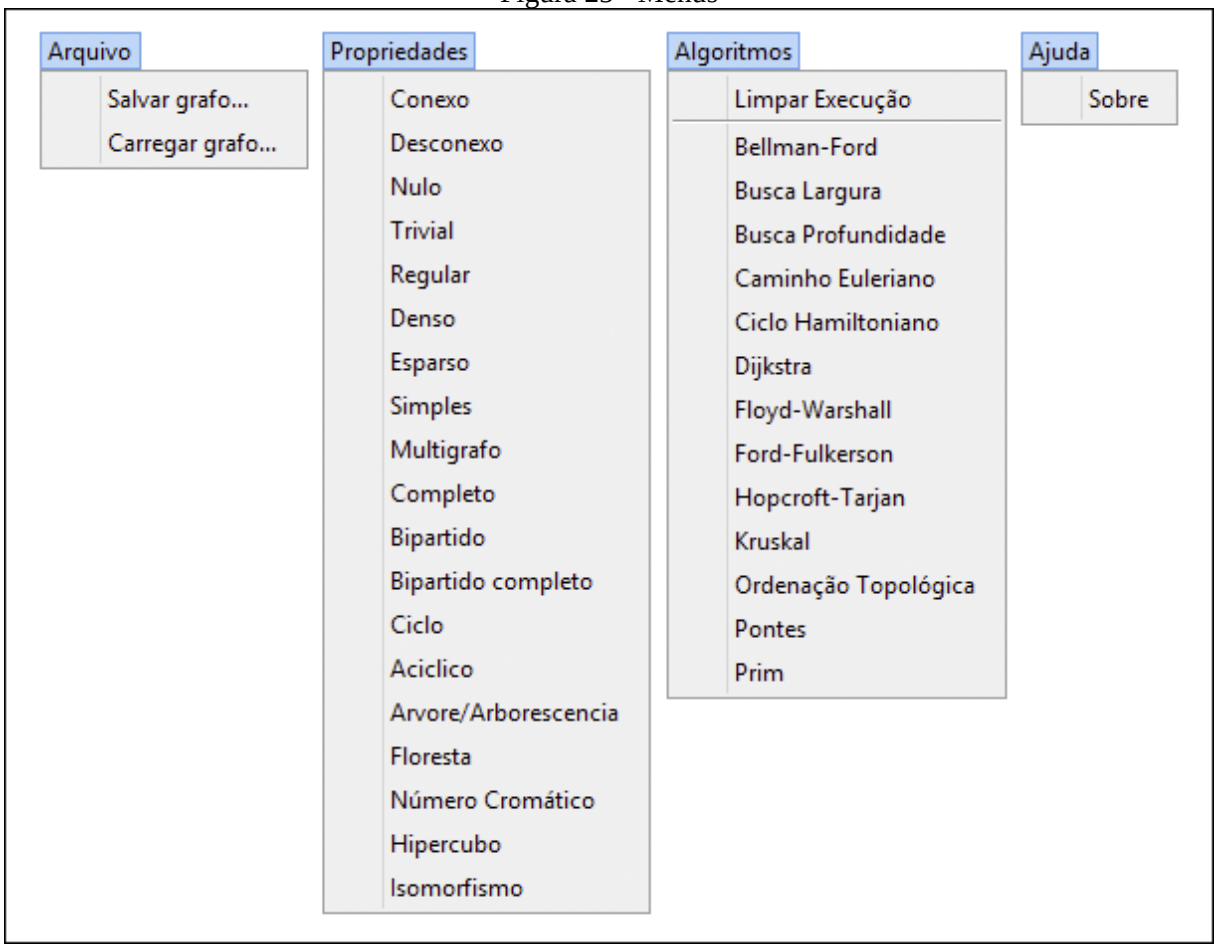
Figura 24 - Imagem da aplicação



A seguir são apresentadas características do programa junto com um estudo de caso no qual é desenhado um grafo, e a partir desse grafo é extraído o número cromático e verificado se há caminho euleriano possível.

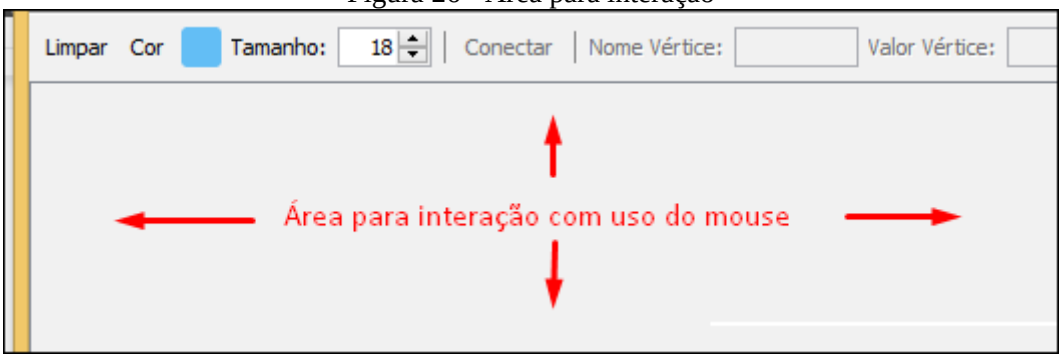
Os quatro itens que compõem o menu são: `Arquivo`, `Propriedades`, `Algoritmos` e `Ajuda`. O item `Arquivo` é responsável por salvar e carregar um grafo através de uma tela de seleção de arquivo fornecida pelo sistema operacional. O item `Propriedades` disponibiliza as propriedades disponíveis para teste. O item `Algoritmos` disponibiliza os algoritmos disponíveis para execução. O item `Ajuda` exibe uma opção chamada de `Sobre` que exibe os desenvolvedores da aplicação. Os itens dos quatro menus são apresentados na Figura 25.

Figura 25 - Menus



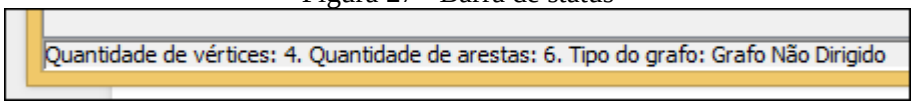
Dentro da área disponível para desenho, há duas partes principais: a área de manipulação dos vértices e arestas e a área para manipulação interativa através do uso do mouse (Figura 26).

Figura 26 - Área para interação



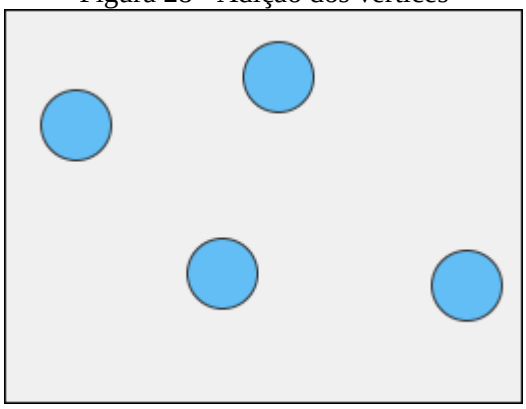
A área indicada pelas setas da Figura 26 exibe a área na qual podem ser adicionados os vértices do grafo através do clique com o botão direito do mouse. Nesta área também pode ser utilizado o recurso de *drag-and-drop* para selecionar vários vértices. Por último, existe a barra de *status* que fornece informações sobre o grafo atual (Figura 27).

Figura 27 - Barra de status



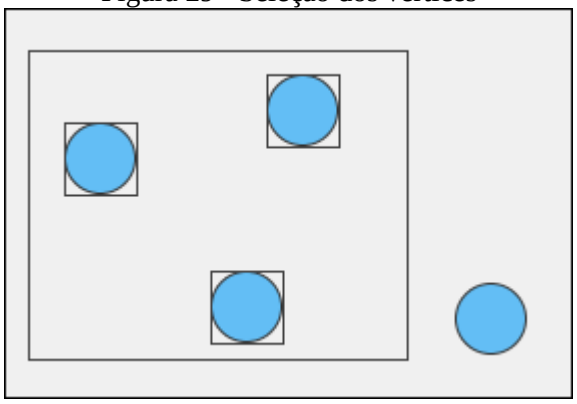
Para exemplificação, é criado um grafo com quatro vértices nomeados e quatro arestas nomeadas. Para adicionar os vértices, é utilizado o clique com o botão esquerdo do mouse na área de interação. Um vértice é adicionado sempre que não há nenhum outro vértice selecionado por questões de usabilidade (evitar adicionar vértices não desejados) (Figura 28).

Figura 28 - Adição dos vértices



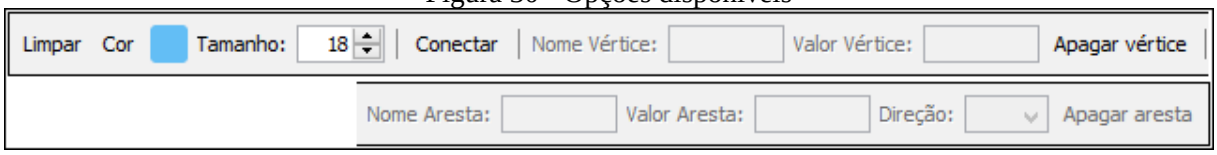
Para conectar os vértices, deve-se primeiro selecionar os vértices que serão conectados. A seleção pode ser feita de dois modos: (a) pressionando a tecla SHIFT e selecionando os vértices desejados ou (b) mantendo pressionado o botão esquerdo do mouse para criar um retângulo de seleção na área de interação da tela, todos os vértices dentro desse retângulo serão selecionados ao soltar o botão do mouse. A Figura 29 mostra os três vértices selecionados que estão dentro da área de seleção.

Figura 29 - Seleção dos vértices



Ao selecionar os vértices, algumas opções ficam disponíveis a partir do menu de manipulação do grafo (Figura 30).

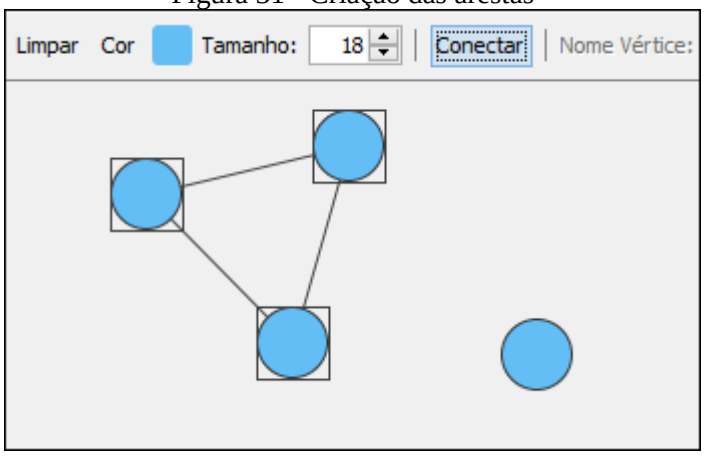
Figura 30 - Opções disponíveis



A Figura 30 mostra as opções disponíveis após a seleção dos vértices. As opções disponíveis são somente as opções que alteram o conteúdo do grafo a partir do contexto selecionado. Por exemplo, a opção `Apagar aresta` fica desabilitada pois não há aresta para ser removida.

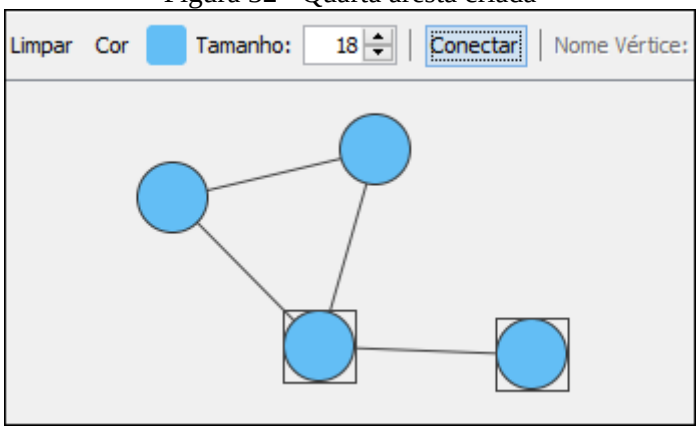
Ao clicar na opção `Conectar` são criadas arestas entre todas as combinações possíveis entre os vértices selecionados. Em outras palavras, é feito o sub grafo completo entre os vértices selecionados. A Figura 31 mostra as conexões entre os vértices criadas a partir da opção `Conectar`.

Figura 31 - Criação das arestas



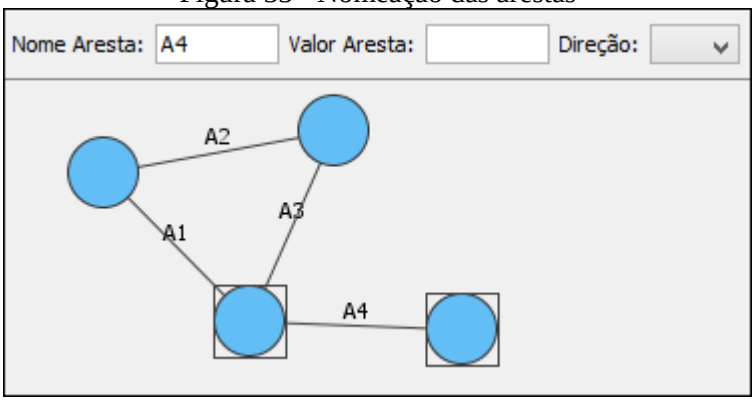
Com uma das maneiras demonstradas, é feita a ligação entre o vértice isolado com um vértice da componente conexa (Figura 32).

Figura 32 - Quarta aresta criada



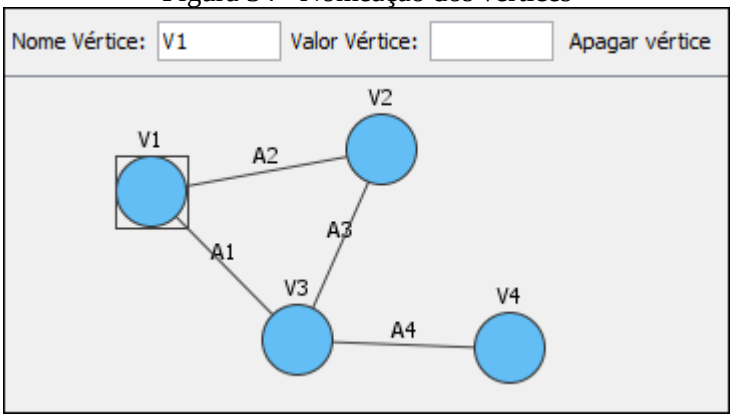
Ao selecionar somente duas arestas, é habilitada a opção que permite dar um nome, um valor e uma direção para as arestas (Figura 33).

Figura 33 - Nomeação das arestas



Ao selecionar somente um vértice, é habilitada a opção que permite dar um nome e um valor para o vértice selecionado (Figura 34).

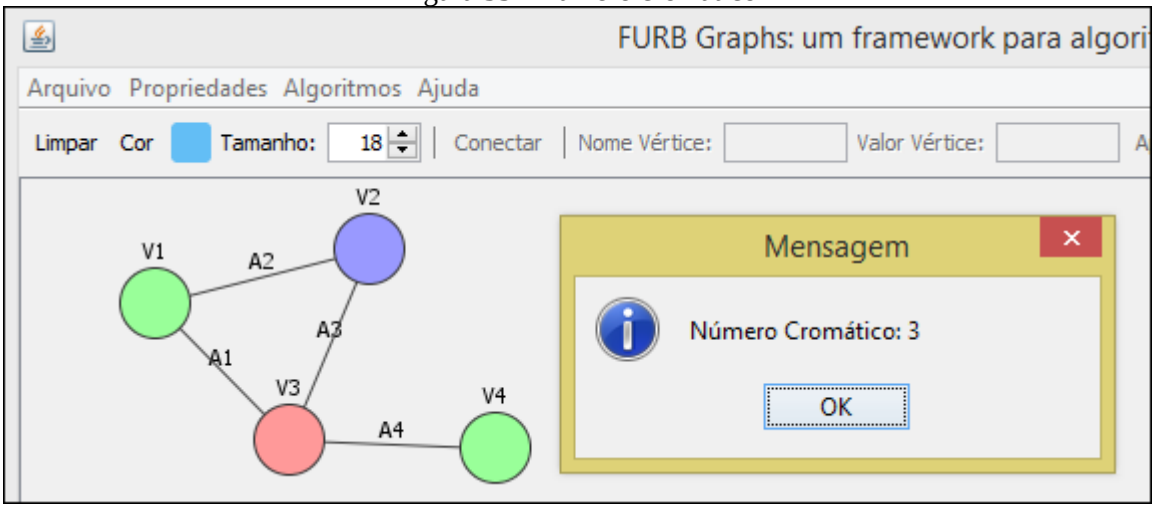
Figura 34 - Nomeação dos vértices



Com o grafo pronto, é possível testar as propriedades (Figura 35) e executar os algoritmos (A **Erro! Autoreferência de indicador não válida.** exibe o resultado da execução do caminho euleriano com o *feedback* visual do caminho. É exibida também uma modal com o caminho das arestas que caracteriza o caminho euleriano.

Figura 36).

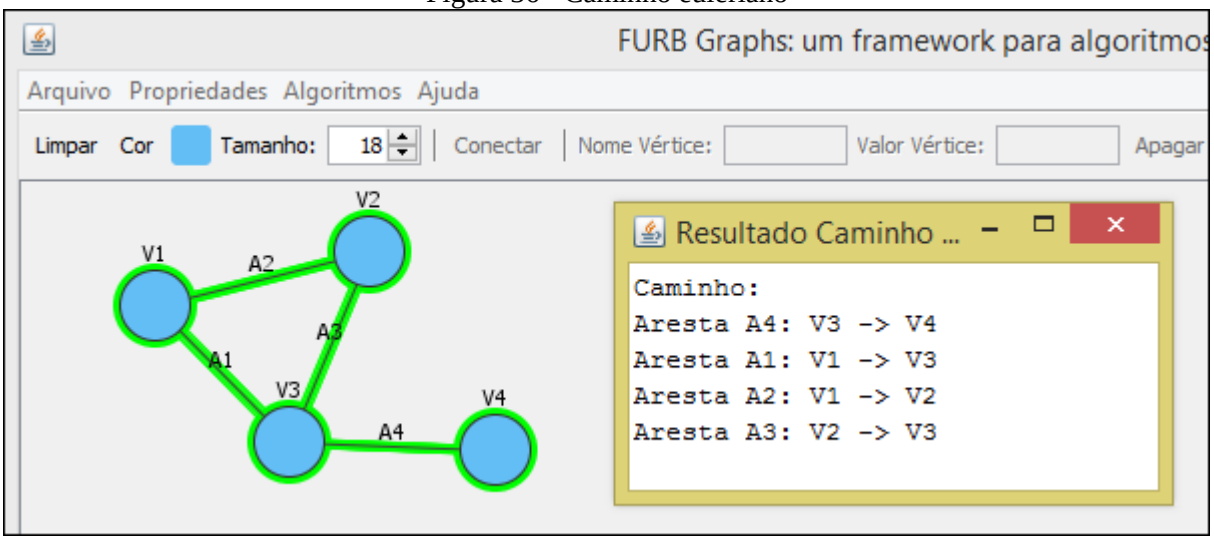
Figura 35 - Número cromático



A Figura 35 exibe o número cromático do grafo, observando que as cores encontradas pelo algoritmo são sugeridas e apresentadas na interface.

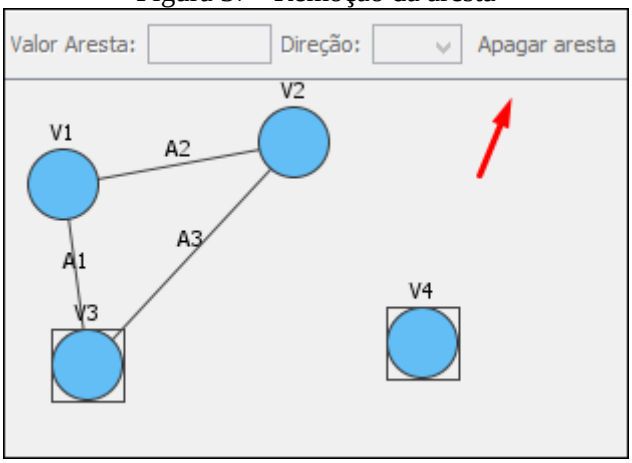
A **Erro! Autoreferência de indicador não válida.** exibe o resultado da execução do caminho euleriano com o *feedback* visual do caminho. É exibida também uma modal com o caminho das arestas que caracteriza o caminho euleriano.

Figura 36 - Caminho euleriano



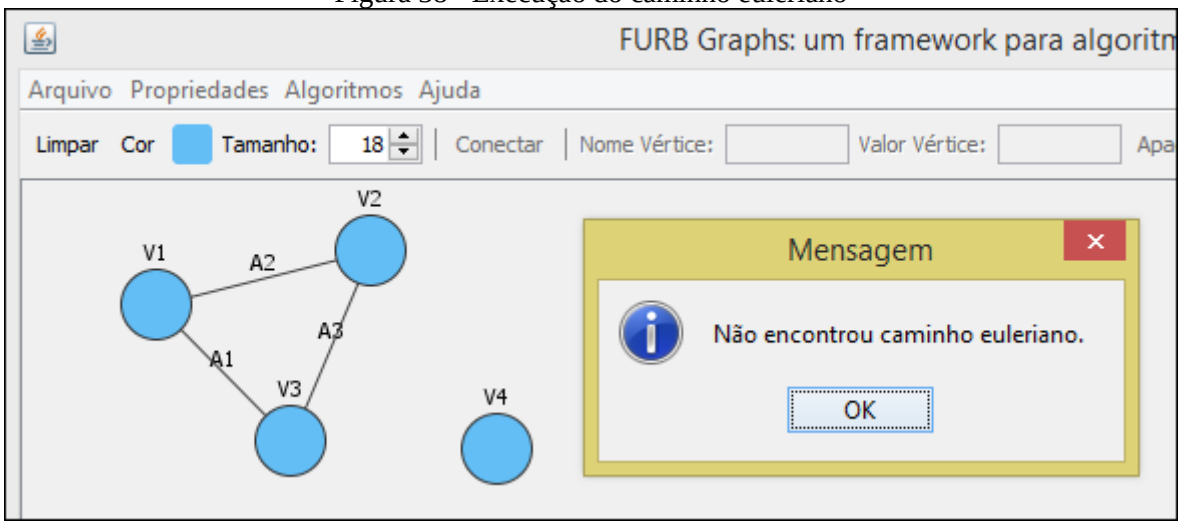
Para exemplificar o que ocorre quando não existe caminho euleriano, é removida a aresta A4 através da opção `Remover aresta` (pré-requisito: estar com os dois vértices que ligam a aresta selecionado) (Figura 37).

Figura 37 - Remoção da aresta



Após executar o algoritmo novamente, o resultado é uma modal indicando a falta da propriedade que define o caminho euleriano (Figura 38).

Figura 38 - Execução do caminho euleriano



Por fim, o grafo desenhado é salvo utilizando o menu `Arquivo` > `Salvar Grafo`. O arquivo no formato Json gerado pode ser utilizado para carregar a mesma instância do grafo no momento em que foi salvo.

3.4 RESULTADOS E DISCUSSÃO

Para validar a implementação, foram feitos testes de corretude para os algoritmos e propriedades propostos neste trabalho. Também foram feitos testes não funcionais para observar como a interface gráfica se comporta com grafos densos. A seção 3.4.1 descreve os testes para as propriedades implementadas, enquanto a seção 3.4.2 demonstra os testes para os algoritmos implementos, já na seção 3.4.3 é apresentado os testes não-funcionais feito sobre a aplicação gráfica. Por fim, na seção 3.4.4 é mostrada a comparação com os trabalhos correlatos.

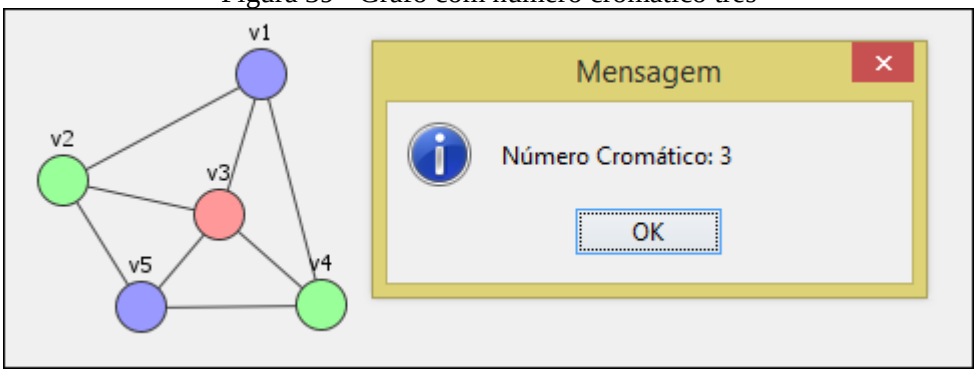
3.4.1 Testes das propriedades

Nas próximas seções são apresentados os testes feitos para validar as propriedades propostas e implementadas neste trabalho, são elas: número cromático, isomorfismo e hipercurbo.

3.4.1.1 Testes para número cromático

Para verificar a corretude do algoritmo que determina o número cromático do grafo foram feitos 3 testes. O primeiro teste foi realizado utilizando um grafo simples com 5 vértices e 8 arestas, conforme mostra a Figura 39.

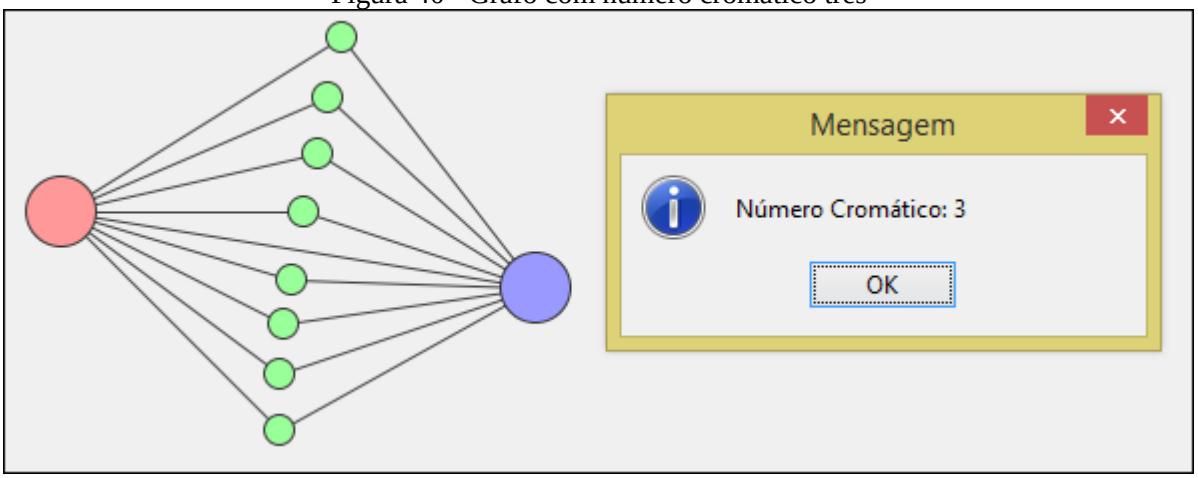
Figura 39 - Grafo com número cromático três



A partir da Figura 39 pode-se observar que o número cromático apontado pelo algoritmo de coloração está correto (3 cores), pois com duas cores não seria possível colorir o grafo sem que houvesse vértices adjacentes com a mesma cor. Ou seja, todo e qualquer grafo que contém um subgrafo `K3` (triângulo) necessita de no mínimo 3 cores.

No segundo teste utilizou-se um grafo com 10 vértices e 17 arestas (Figura 40). O objetivo deste teste foi criar um grafo onde a maioria dos vértices possuíssem a mesma cor, independentemente da disposição dos vértices.

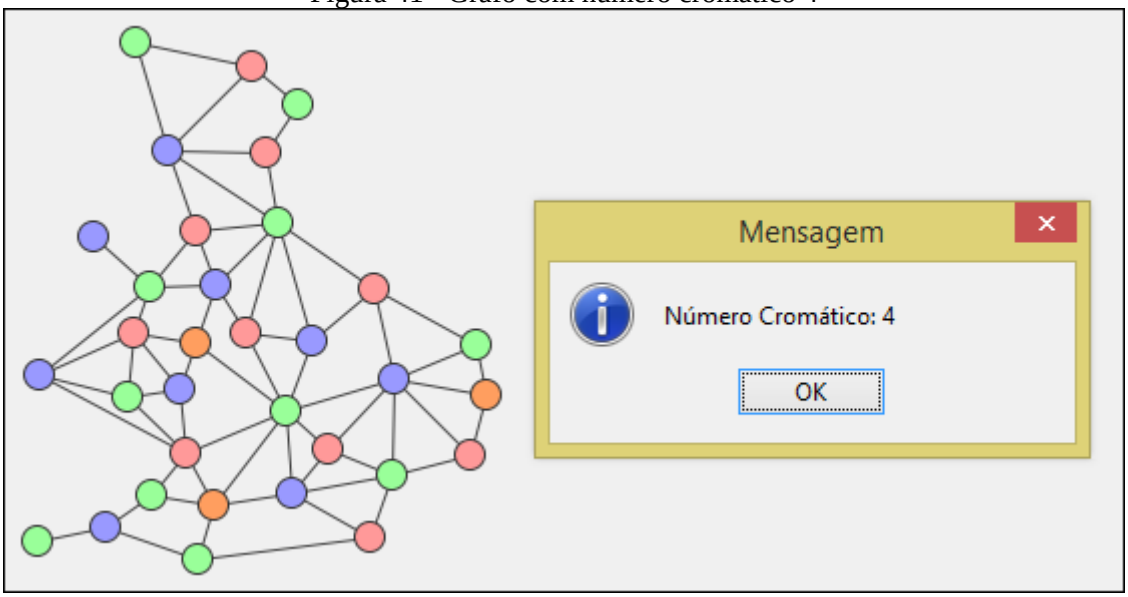
Figura 40 - Grafo com número cromático três



A Figura 40 demonstra que o número cromático do grafo proposto é 3. Está correto, pois este grafo também contém um triângulo (`K3`).

No terceiro teste utilizou-se um grafo relativamente grande, sendo composto de 33 vértices e 66 arestas (Figura 41).

Figura 41 - Grafo com número cromático 4

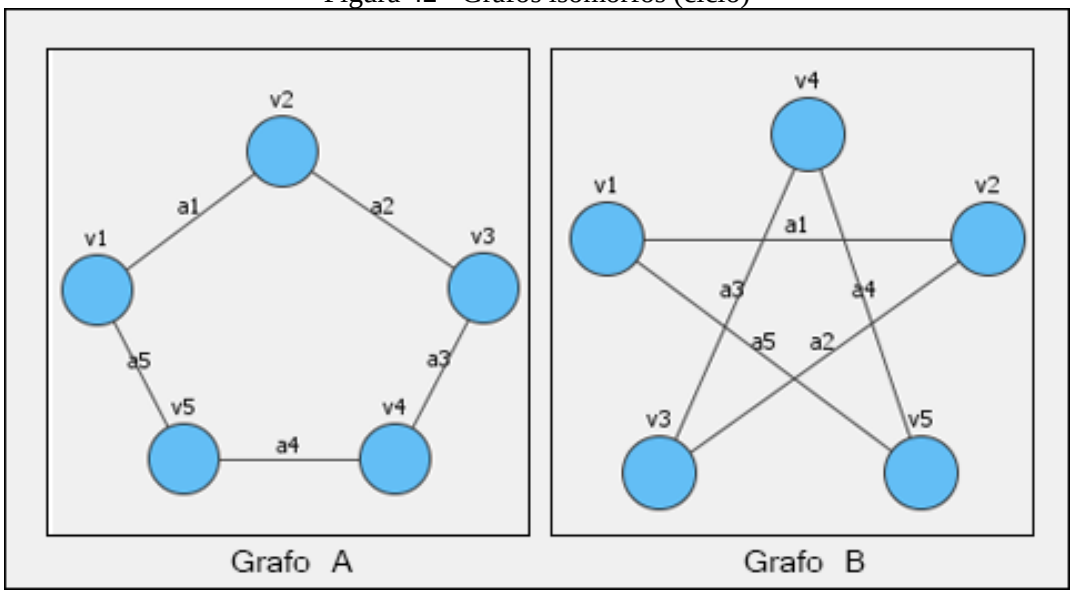


A Figura 41 mostra que o resultado obtido neste teste foi 4. O resultado exibido é o resultado ótimo pois existe um subgrafo K_4 dentro do grafo. O resultado também não pode ser maior do que 4 pois o Teorema das Quatro Cores diz que todo grafo planar admite 4-coloração dos vértices (FEOFILOFF; KOHAYAKAWA; WAKABAYASHI, 2013). A partir dos resultados obtidos, pode-se concluir que o algoritmo desenvolvido apresenta um bom índice de corretude para os mais diversos tipos de grafos.

3.4.1.2 Testes para a propriedade de isomorfismo

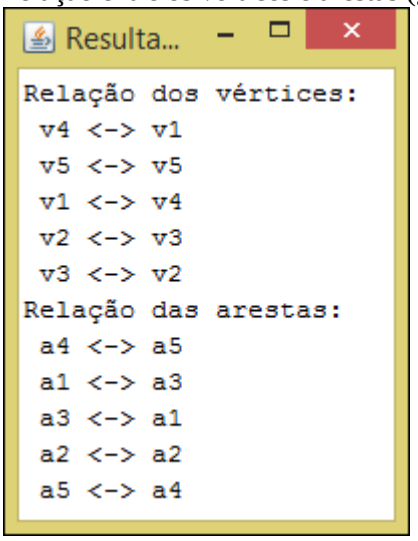
Para validar a implementação da propriedade isomorfismo foram utilizados seis grafos, sendo testados em pares. O primeiro par de grafos testa o isomorfismo simples entre um círculo. O objetivo do teste é mostrar que é possível encontrar o resultado independentemente da escolha do vértice inicial, pois neste caso, qualquer vértice inicial escolhido encontrará o grafo isomórfico. A Figura 42 exibe um exemplo de dois grafos isomorfos, enquanto a Figura 43 exibe a relação dos vértices e arestas entre os dois grafos.

Figura 42 - Grafos isomorfos (ciclo)



A Figura 43 exibe a relação *de-para* dos vértices e arestas do grafo A para os vértices e arestas do grafo B.

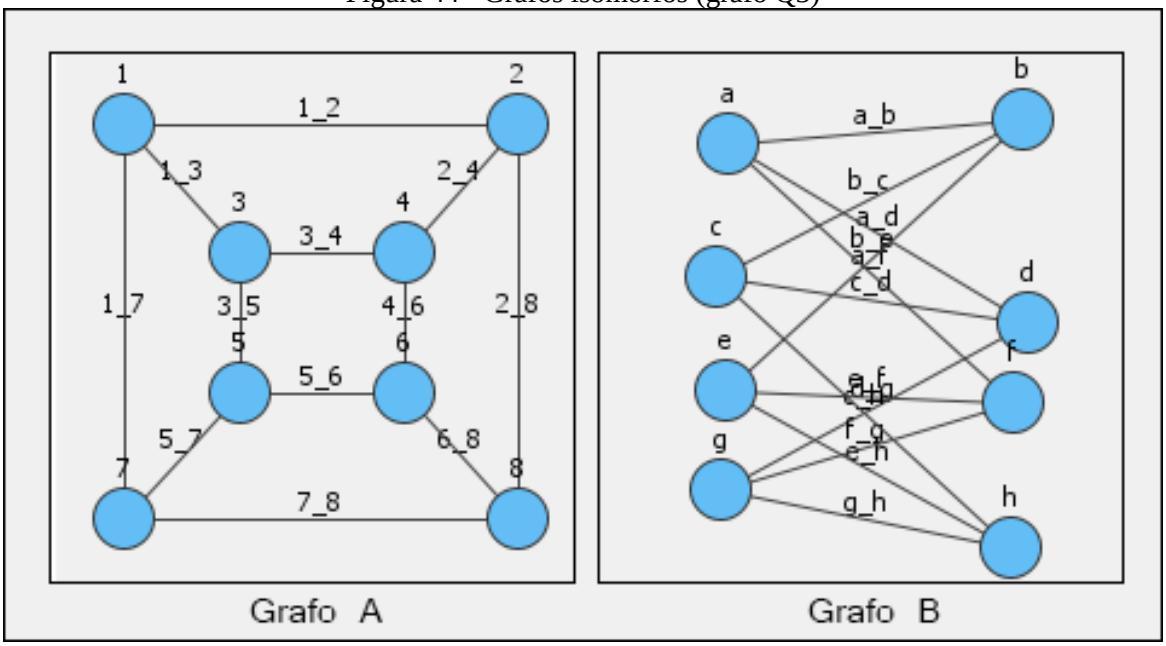
Figura 43 - Relação entre os vértices e arestas (grafo ciclo)



A Figura 43 exibe a relação das arestas entre os dois grafos. Percebe-se que independentemente das rotulações existentes (nome dos vértices relacionados), o isomorfismo foi encontrado.

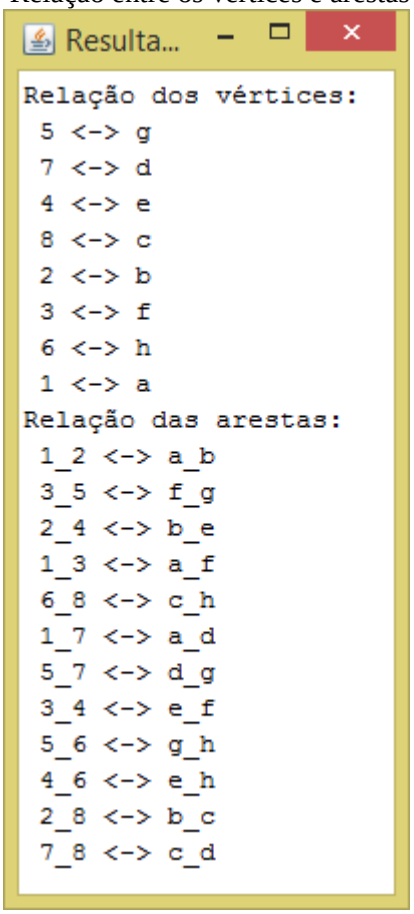
No segundo teste, objetivou-se verificar a existência de isomorfismo em grafos de difícil visualização, como por exemplo, em grafos Q3. A Figura 44 apresenta dois grafos isomorfos.

Figura 44 - Grafos isomorfos (grafo Q3)



O grafo A possui seus vértices nomeados com números e suas arestas representam a relação entre os dois vértices, o grafo B segue a mesma ideia, porém utiliza letras no lugar de números. A relação entre os vértices e arestas entre o grafo A e grafo B são exibidos na Figura 45.

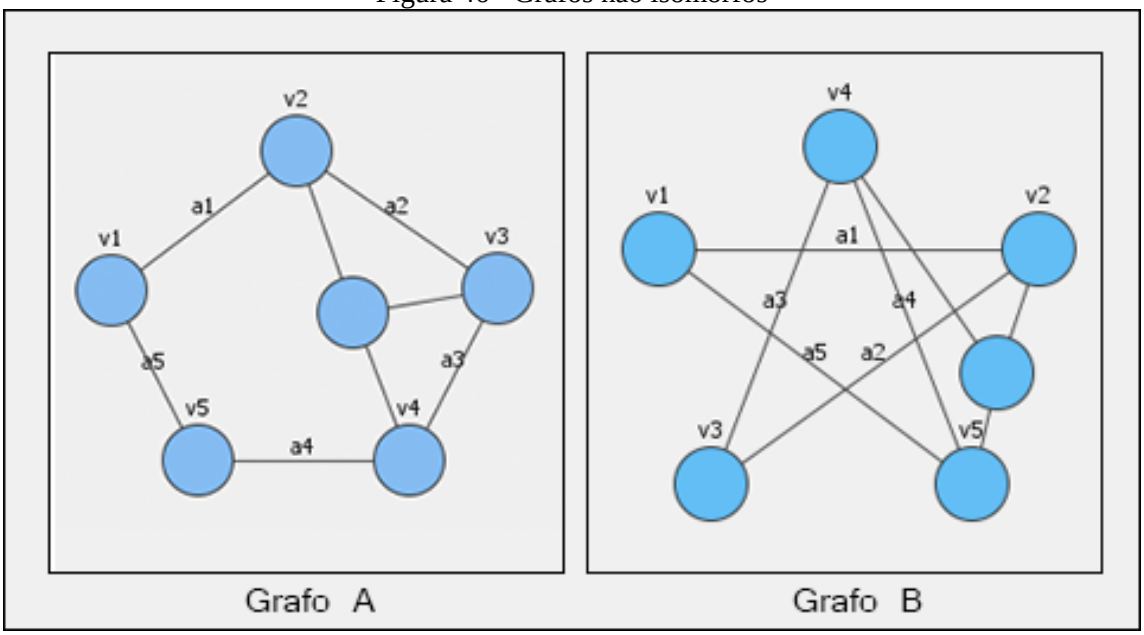
Figura 45 - Relação entre os vértices e arestas (grafo Q3)



A Figura 45 exibe a relação dos vértices/arestas do grafo A (lado esquerdo) que são equivalentes aos vértices/arestas do grafo B (lado direito) mostrando que o algoritmo encontra a relação de arestas e vértices de grafos isomórficos mesmo de grafos que são difíceis de ver visualmente.

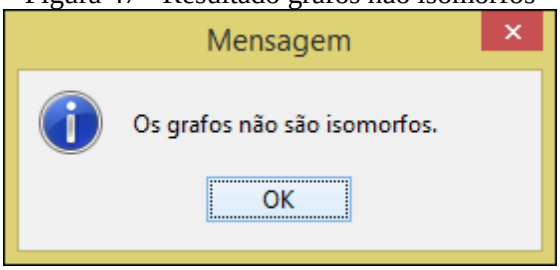
Por fim, foi feito o teste sobre dois grafos que não são isomorfos (Figura 46).

Figura 46 - Grafos não isomorfos



Os dois grafos da Figura 46 não são isomorfos porque o vértice incluído no grafo A é adjacente a três vértices em sequência (`v2`, `v3` e `v4`), enquanto o grafo B não incluiu o vértice da mesma forma (`v2`, `v4` e `v5`). A Figura 47 exibe o resultado para o teste de dois grafos que não são isomorfos.

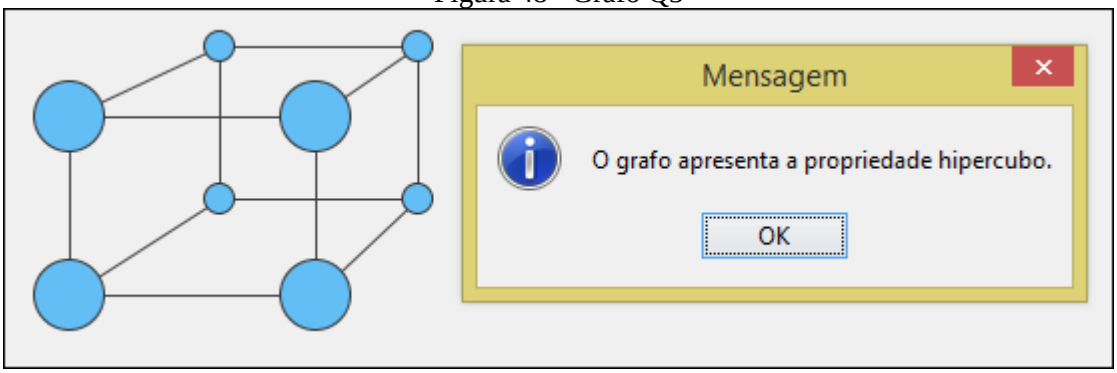
Figura 47 - Resultado grafos não isomorfos



3.4.1.3 Testes para a propriedade hipercubo

Para testar a propriedade hipercubo foram feitos testes para validar o valor retornado para os grafos Q3, Q4 e um grafo inválido. O primeiro teste foi aplicado em um grafo Q3 (Figura 48).

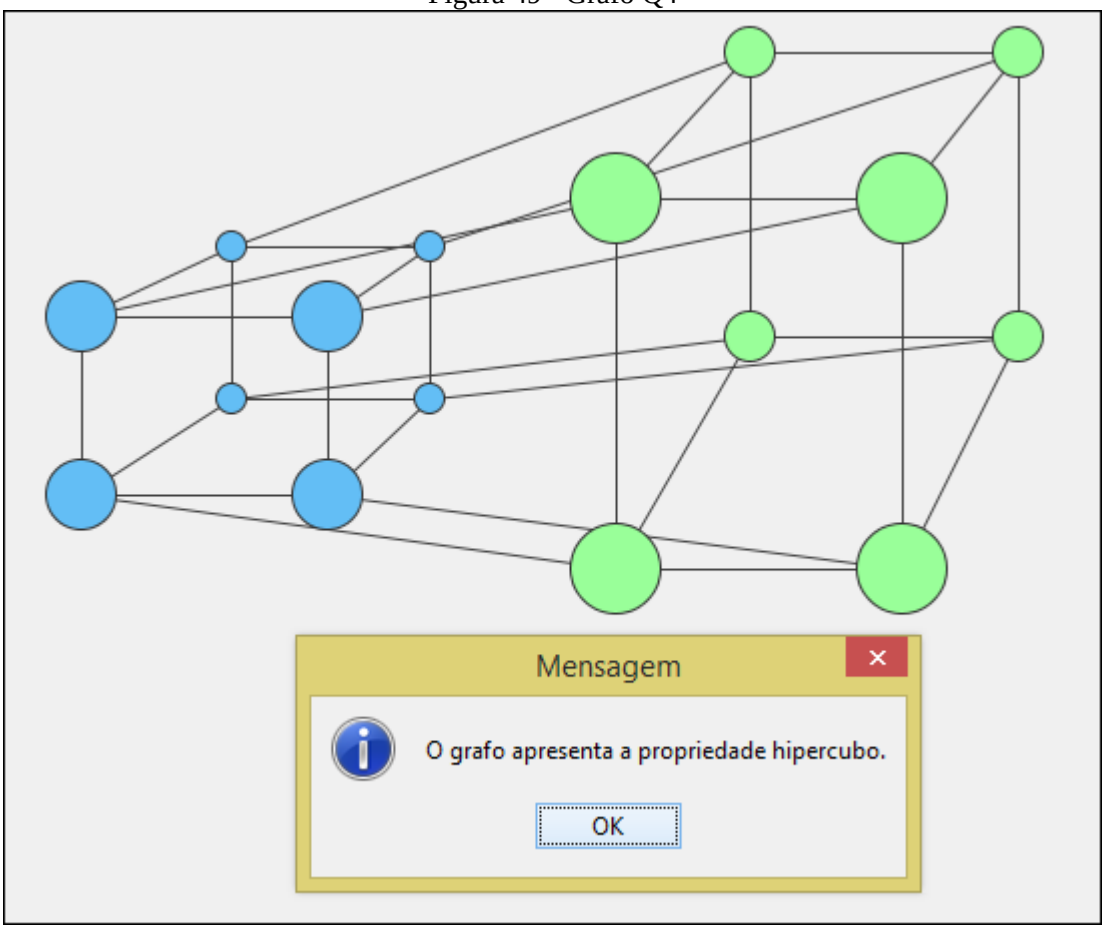
Figura 48 - Grafo Q3



A Figura 48 aponta que o grafo em questão apresenta a propriedade de hipercubo, pois todos os vértices possuem grau 3.

O segundo teste é uma evolução do teste apresentado na Figura 48. Neste teste é apresentado o grafo Q4 (Figura 49).

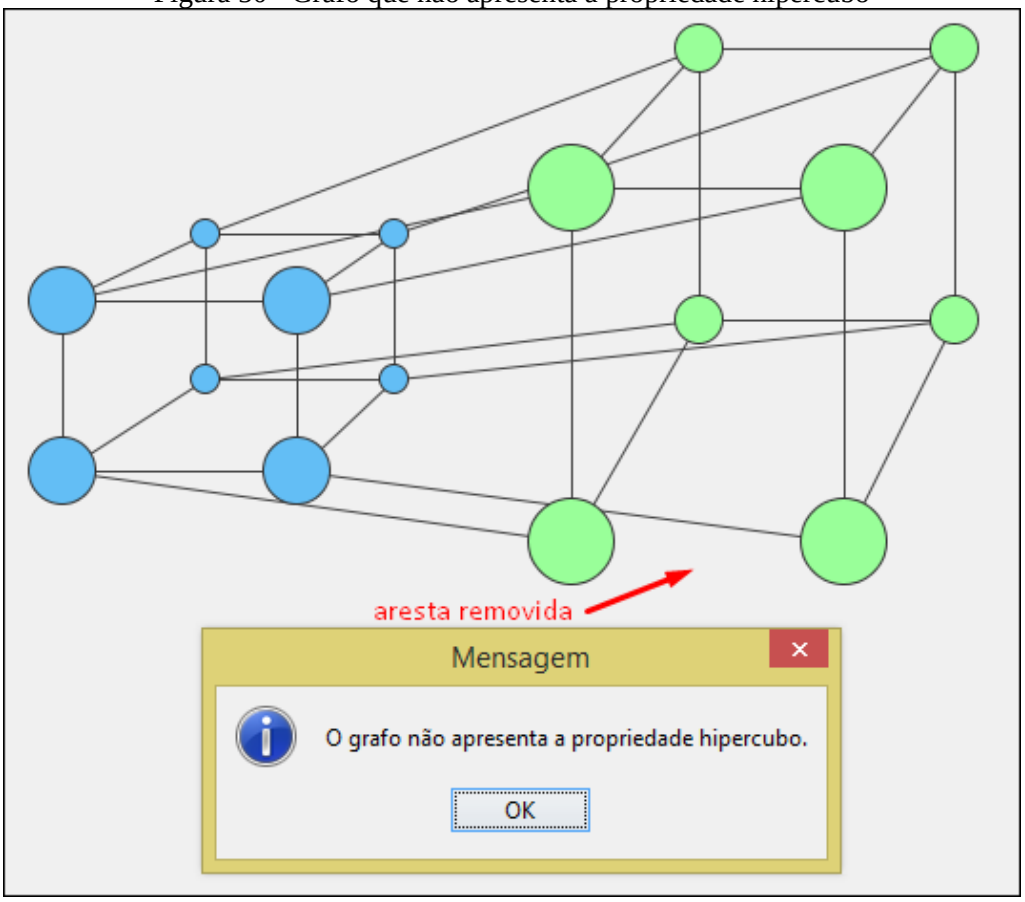
Figura 49 - Grafo Q4



A Figura 49 apresenta o teste para verificar a existência da propriedade hipercubo no grafo Q4. O grafo Q4 é composto por dois grafos Q3 postos lado a lado, conectados pelos vértices equivalentes. Este grafo também apresenta a propriedade de hipercubismo.

No terceiro teste foi removida uma aresta do grafo Q4 (Figura 49), tornando-o desta forma, um grafo que não apresenta a propriedade hipercubo (Figura 50).

Figura 50 - Grafo que não apresenta a propriedade hipercubo



A Figura 50 mostra que o grafo proposto não possui a propriedade hipercubo. Ao executar o algoritmo descrito na seção 3.3.2.1.1, o algoritmo começará tentando identificar qual é a potência de 2 que representa o tamanho do grafo, como este grafo possui 16 vértices, a potência de dois que o define é 4. Ao continuar o algoritmo, será verificado se todos os graus dos vértices possuem o valor 4, neste ponto o algoritmo irá identificar que nem todos os vértices possuem tal grau (existem dois vértices com o grau 3), retornando assim, o resultado `false` para a execução do algoritmo.

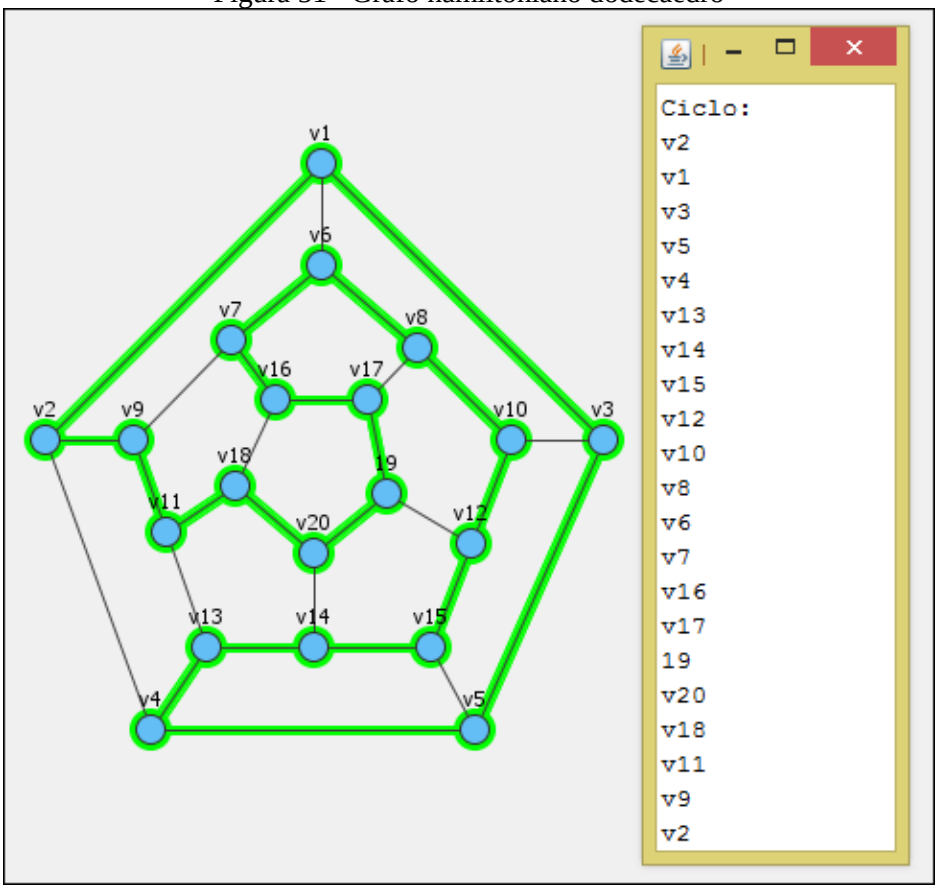
3.4.2 Testes dos algoritmos

Nas próximas seções são apresentados os testes realizados para validar os algoritmos propostos e implementados neste trabalho, são eles: ciclo hamiltoniano e caminho euleriano.

3.4.2.1 Testes para ciclo hamiltoniano

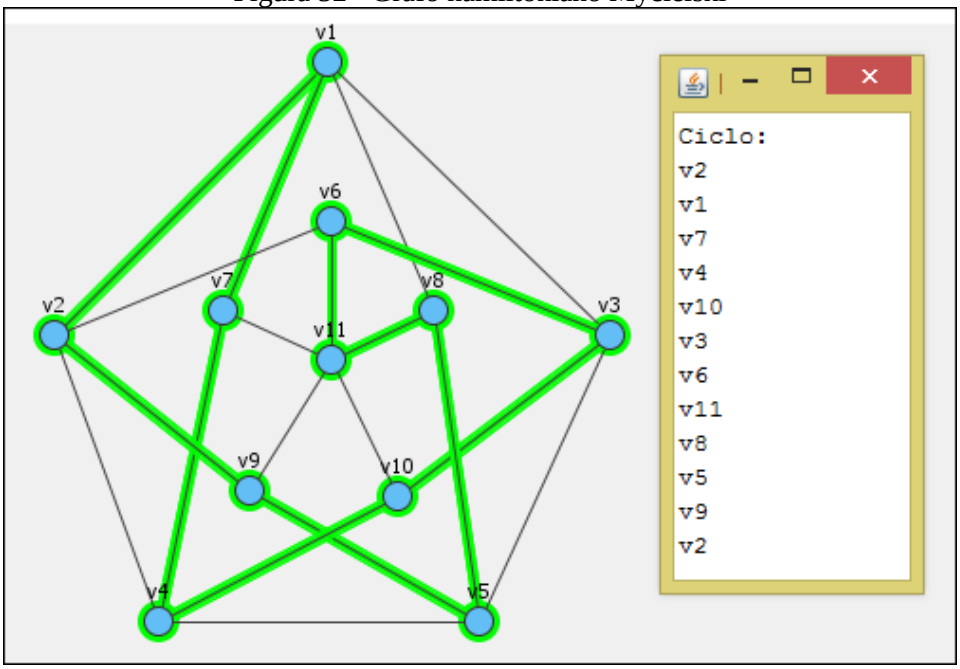
Os testes para verificar a existência de ciclos hamiltonianos foram feitos utilizando dois grafos hamiltonianos clássicos, são eles: o sólido platônico dodecaedro e grafo de Mycielski. Por fim, foi feito um teste em um grafo que não é hamiltoniano. O primeiro teste apresenta o sólido platônico dodecaedro (Figura 51).

Figura 51 - Grafo hamiltoniano dodecaedro



A Figura 51 mostra o ciclo hamiltoniano encontrado a partir do grafo sólido platônico dodecaedro, juntamente com a lista de vértices que formam o ciclo. O próximo teste foi feito utilizando o grafo de Mycielski (Figura 52).

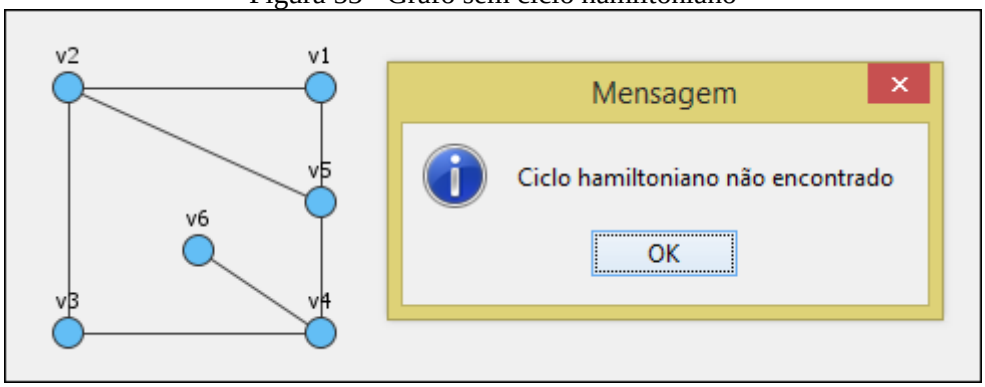
Figura 52 - Grafo hamiltoniano Mycielski



A partir da Figura 52 é possível verificar a existência de um ciclo hamiltoniano no grafo de Mycielski.

O último teste apresenta um grafo que não tem um ciclo hamiltoniano (Figura 53).

Figura 53 - Grafo sem ciclo hamiltoniano

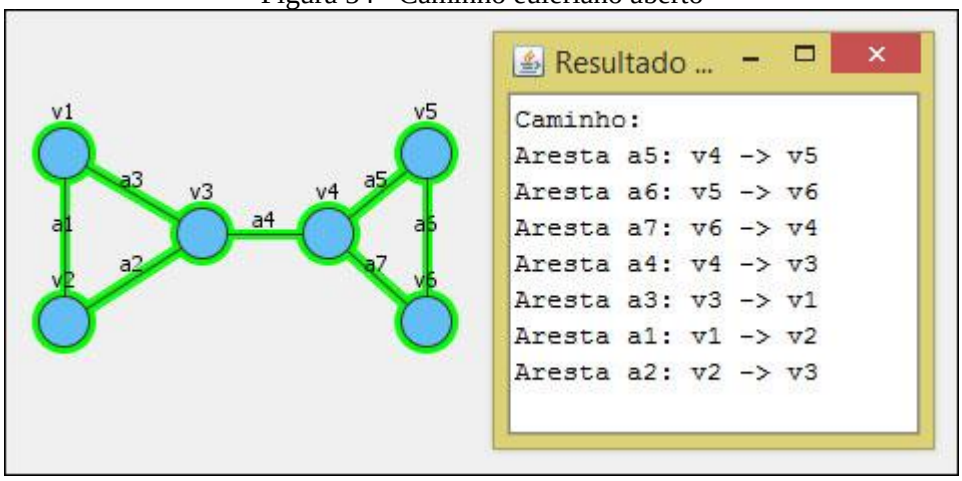


A Figura 53 mostra que o grafo testado não possui um ciclo hamiltoniano, pois o vértice `v6` está isolado. Ou seja, para entrar e depois sair dele, é necessário passar 2 vezes pelo vértice `v4`, circunstância que quebra o conceito de ciclo hamiltoniano (um vértice só pode ser visitado uma única vez).

3.4.2.2 Testes para caminho euleriano

Para validação do caminho euleriano, são propostos três testes. O primeiro teste tem por objetivo testar um grafo com uma ponte (Figura 54).

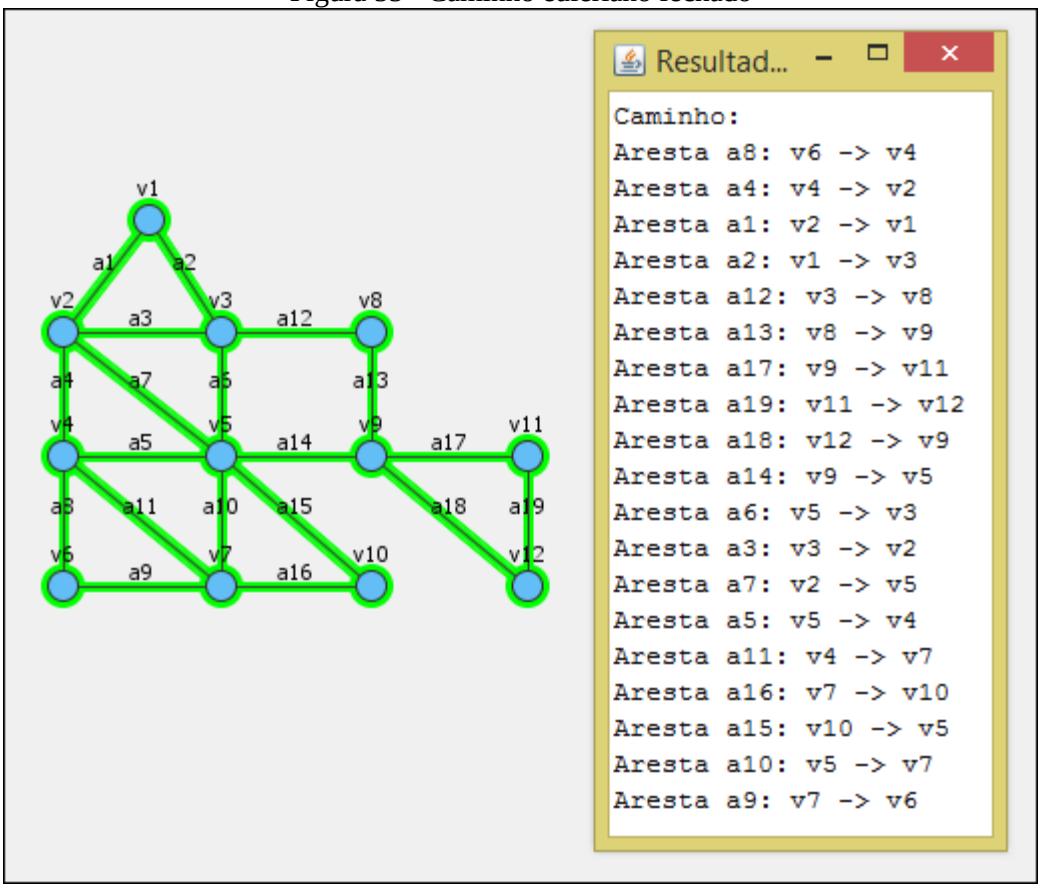
Figura 54 - Caminho euleriano aberto



A Figura 54 mostra o caminho euleriano aberto encontrado. Aberto pois o caminho inicia no vértice 4 e termina no vértice 3, a aresta `a4` que ligaria ambos os vértices, já foi utilizada, portanto não existe ciclo euleriano neste grafo. Outra propriedade que identifica o caminho euleriano aberto é que existem dois vértices de grau ímpar.

O segundo teste mostra um grafo com 12 vértices e 19 arestas que possui um caminho euleriano fechado (Figura 55).

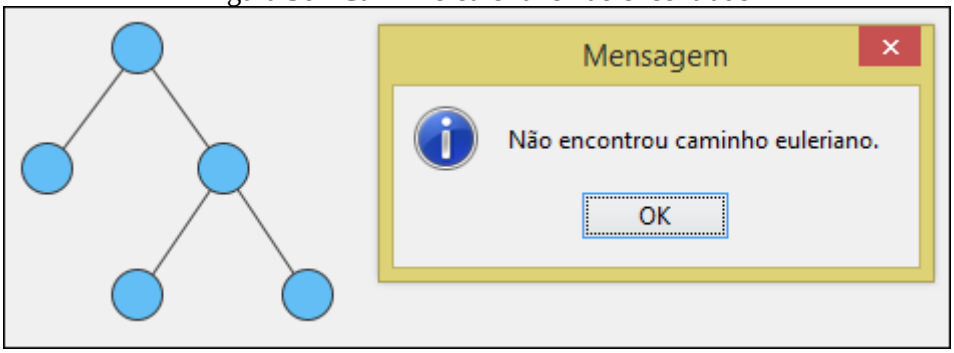
Figura 55 - Caminho euleriano fechado



A Figura 55 exibe um caminho euleriano fechado, pois o vértice inicial `v6` é também o vértice final. Outra característica que indica a existência de um caminho euleriano fechado é que o grau de todos os vértices deve ser par.

Por fim, aplicou-se o teste para verificar a existência de um caminho euleriano em uma árvore binária (Figura 56).

Figura 56 - Caminho euleriano não encontrado

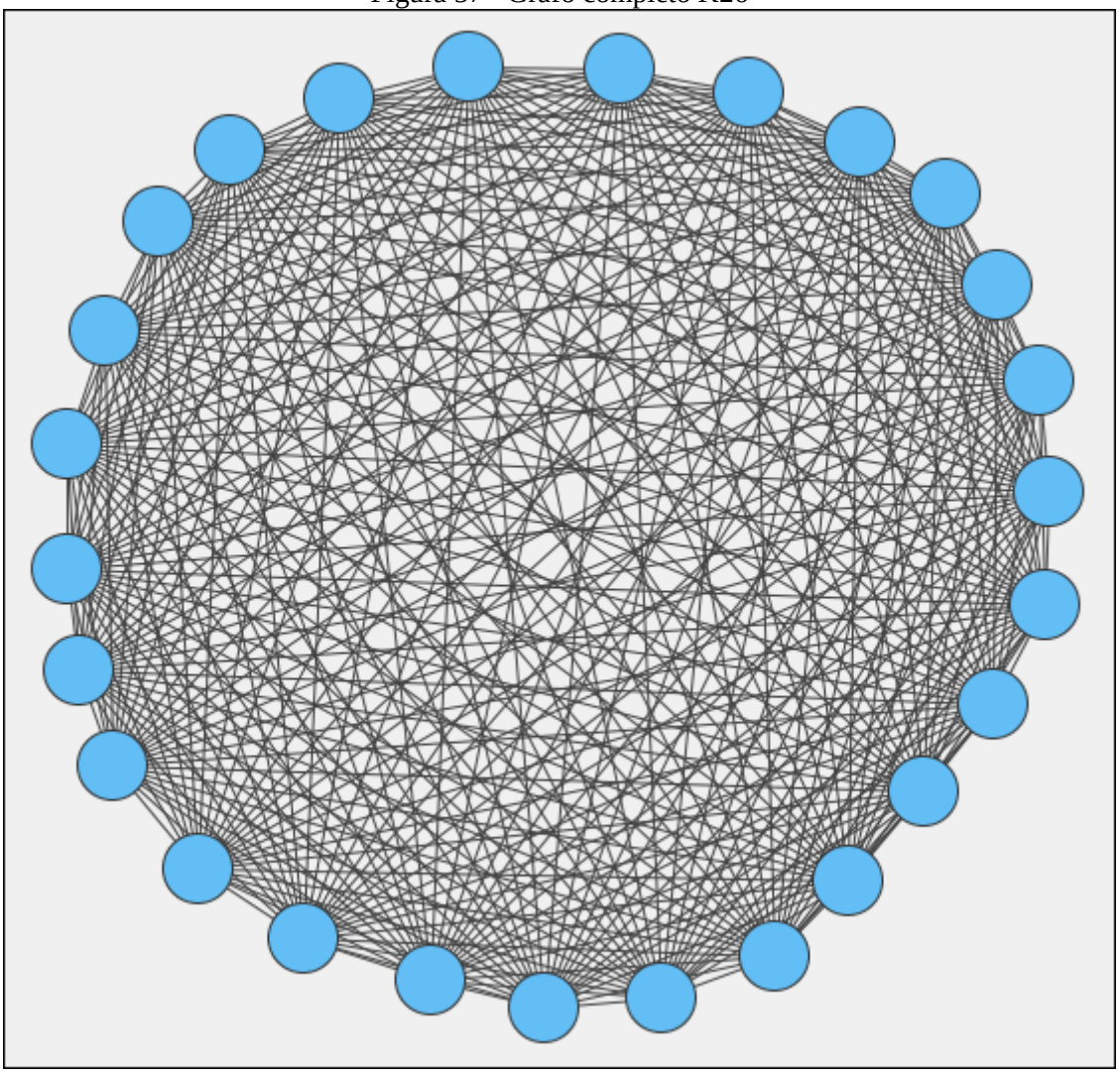


A partir do resultado exibido na Figura 56 pode-se perceber que uma árvore binária não possui um caminho euleriano, pois existem mais do que dois vértices com grau ímpar.

3.4.3 Testes da interface gráfica

Para avaliar como a aplicação gráfica se comportaria com grafos grandes, foram feitos dois testes. O primeiro teste exibe o grafo completo `K26` (Figura 57).

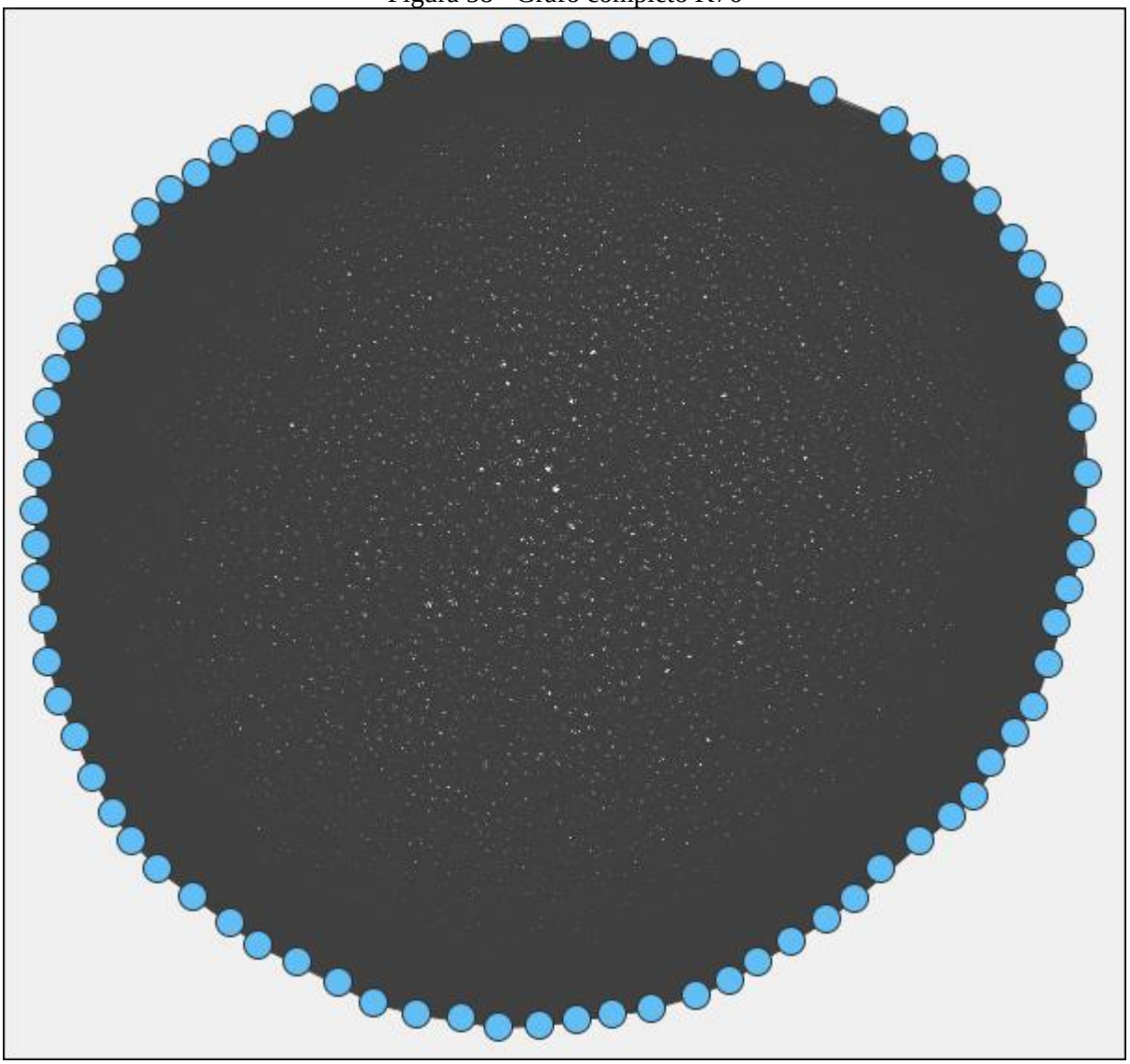
Figura 57 - Grafo completo K26



A Figura 57 exibe um grafo completo com 26 vértices e 365 arestas. Não houve lentidão ao selecionar e arrastar os vértices junto de suas arestas.

O segundo teste é similar ao primeiro, porém o teste se aplica ao grafo completo `K76` (Figura 58).

Figura 58 - Grafo completo K76



A Figura 58 exibe um grafo com 76 vértices e 3003 arestas. Também não foi percebido nenhum tipo de lentidão ao selecionar vários vértices e arrastá-los, porém, há claramente a falta de visibilidade de todas as arestas.

3.4.4 Comparação com os trabalhos correlatos

O Quadro 25 resume as características e resultados finais dos trabalhos correlatos comparados com os resultados obtidos por este projeto.

Quadro 25 - Comparativo dos trabalhos correlatos com este projeto

características / trabalhos relacionados	Hackbarth (2008)	Villalobos (2006)	JGraphT (2005)	Zatelli (2010)	FURB Graphs (2014)
foco didático	sim	sim	não	sim	sim
linguagem	Java	VisualBasic	Java/web	Java	Java
é framework	não	não	sim	sim	sim
possui aplicação visual nativa	sim	sim	não	não	sim
aplicação visual interativa	sim	sim	sim	não	sim
usabilidade da aplicação visual interativa	fraca	média	não se aplica	não se aplica	média
quantidade de algoritmos implementados	3	6	10+	10	10+
algoritmo ciclo hamiltoniano	não	sim	sim	não	sim
algoritmo caminho euleriano	não	sim	sim	não	sim
propriedade número cromático	não	não	sim	não	sim
propriedade isomorfismo	não	não	sim	não	sim
propriedade hipercubo	não	não	não	não	sim
disponibiliza API	não	não	sim	sim	sim
exibe resultado visual da execução de um algoritmo	sim	sim	não se aplica	não	sim
permite criação de novos algoritmos	não	não	sim	sim	sim
permite salvar e carregar grafos criados nativamente	não	sim	não	sim	sim

Em relação ao trabalho de Hackbarth (2008) o FURB Graphs destaca-se por ser uma solução mais completa e mais robusta, por exemplo, embora a solução de Hackbarth (2008) permite a criação de grafos de forma interativa, o mesmo é feito de forma simples e sem todos os recursos que a aplicação FURB Graphs permite. Outro ponto é a ausência de todos os algoritmos e testes propostos por este trabalho, além de não ser disponibilizado via API. Em relação ao trabalho de Villalobos (2006) o FURB Graphs destaca-se por possuir testes para as propriedades mais simples até as mais complexas tais como número cromático e hipercubo. O trabalho de Villalobos (2006) também não é disponibilizado em forma de API. A usabilidade da ferramenta de Villalobos (2006) e FURB Graphs são parecidas pois ambas permitem criar grafos de maneira interativa, porém ambas as ferramentas poderiam dispor de uma maneira para criar grafos de maneira mais rápida. A aplicação JGraphT (2005) assemelha-se muito ao

FURB Graphs nos pontos de testes para propriedades e execução de algoritmos clássicos da teoria dos grafos, porém, não é disponibilizado nenhuma interface nativa para manipulação do grafo, se o usuário, por exemplo, desejar visualizar um grafo afim de testar uma propriedade, o usuário terá que programar toda a interface gráfica desejada. Parte disso é em virtude da proposta do JGraphT (2005) ser utilizada para fins comercias, enquanto a aplicação FURB Graphs possui um foco didático. Por fim, o trabalho do Zatelli (2010) que deu origem a este trabalho apresenta semelhança na quantidade de algoritmos implementados pois este trabalhou herdou todos os seus algoritmos e propriedades. Este trabalho destaca-se ao trabalho de Zatelli (2010) por fornecer uma interface visual e interativa para manipulação e execução dos algoritmos, enquanto o trabalho de Zatelli (2010) concentra-se somente em fornecer um *framework* para desenvolvimento de grafos.

4 CONCLUSÕES

Este trabalho apresentou a continuação do desenvolvimento de um *framework* para a área de teoria dos grafos (ZATELLI, 2010), onde os objetivos principais foram o desenvolvimento de novas propriedades, algoritmos e uma interface visual e interativa para manipulação, criação e testes das propriedades e algoritmos. O desenvolvimento foi feito utilizando a linguagem Java, fazendo uso das bibliotecas Gson e Swing, responsáveis, respectivamente, por salvar/carregar um grafo e auxiliar na construção da interface gráfica.

Os resultados obtidos mostraram-se satisfatórios, mantendo um nível acima dos trabalhos correlatos devido à complexidade dos algoritmos implementados e pela construção da interface gráfica, que fornece os recursos para trabalhar com grafos simples e fornece um *feedback* para execução dos algoritmos e testes de propriedades muito bom. Com este trabalho, um aluno de computação que está estudando a matéria de grafos, conseguirá assimilar mais rapidamente o funcionamento das propriedades número cromático, hipercubo e isomorfismo, além de compreender e entender o que caracteriza um ciclo hamiltoniano e um caminho euleriano, pois, é possível reforçar os conceitos "brincando" com a manipulação do grafo com o uso interface gráfica enquanto testa as propriedades e algoritmos para os mais diversos tipos de grafos.

A maior dificuldade encontrada neste trabalho foi o desenvolvimento dos testes de propriedade e da implementação dos algoritmos, em especial, isomorfismo entre grafos. Implementações em grafos são complexas pois é necessário tomar cuidado com pensamento trivial afim de evitar problemas que não executem em tempo polinomial. É necessário muita pesquisa sobre cada algoritmo afim de saber como resolver o mesmo de maneira eficiente, caso possível. Também é necessário tomar cuidado com propriedades que são, de certa forma, fácil de compreender, porém difíceis de implementar. Por exemplo, entender o significado de um número cromático de um grafo é uma questão fácil de compreender, porém implementar a solução ótima é complicado. Outro caso percebido por este trabalho foi a implementação para o teste de isomorfismo. Apesar de ser um problema fácil de compreender e de imaginar uma solução (a solução é enumerar todas as possibilidades possíveis de combinações entre dois grafos), esta foi uma propriedade muito difícil de implementar. Apesar de ter gerado pouco código, o código produzido é enxuto e complexo e opera com vários níveis de recursão e manipulações de listas afim de manter a navegação sincronizada entre os dois grafos. Estes foram os motivos por ter sido deixado de fora deste trabalho as propriedades cordais e planar, além dos algoritmos clique máximo e emparelhamento perfeito máximo. Em especial, clique

máximo e grafos cordais foram deixado de fora pois o intuito deste trabalho era explorar a condição de utilizar de que grafos cordais podem resolver o problema do clique máximo (que é um problema NP-difícil) em tempo polinomial.

A principal vantagem do trabalho desenvolvido é a interface visual e interativa proposta que ajuda a assimilar e compreender algoritmos mais facilmente. Outro fator importante é a implementação das propriedades isomorfismo e hipercubo, pois carecem de exemplos de implementação na literatura. Entretanto, existem limitações do trabalho na interface visual sendo a mais perceptível que atualmente ela opera somente para grafos simples, portanto não é possível definir arestas paralelas e laços.

4.1 EXTENSÕES

Como extensão para trabalhos futuros, sugere-se:

a) realizar melhorias na interface visual para suportar arestas paralelas, laços e *bendpoints*;
b) realizar análise de usabilidade da interface visual e incluir funções como zoom e reposicionamento automático dos vértices;
c) complementar o *framework* com os algoritmos clique-máximo, emparelhamento perfeito máximo, relabel-to-front, Boruvka, entre outros;
d) complementar o *framework* com as propriedades para verificar grafos cordais.

REFERÊNCIAS

BOAVENTURA NETTO, Paulo O. **Teoria e modelos de grafos**. [S.I.]: Edgard Blucher, 1996.

CORMEN, Thomas H. et al. **Introduction to algorithms**. 2nd ed. Cambridge: MIT, 2001.

FEOFILOFF, Paulo; KOHAYAKAWA, Yoshiharu; WAKABAYASHI, Yoshiko. **Teoria dos grafos**: uma introdução sucinta. [São Paulo], 2013. Disponível em: <http://www.ime.usp.br/~pf/teoriadosgrafos/>. Acesso em: 30 mar. 2014.

_____. **Algoritmos gulosos**. [São Paulo], 2014. Disponível em: <http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html>. Acesso em: 11 nov. 2014.

GAREY, Michael R.; JOHNSON, David S. **Computers and Intractability**: a guide to the theory of NP-completeness. Ney York: W. H. Freeman, 1983.

GROSS, Jonathan L.; YELLEN, Jay. **Graph theory and its applications**. 2nd ed. Boca Raton: CRC, 2006.

HACKBARTH, Rodrigo. **Ferramenta para representação gráfica do funcionamento de algoritmos aplicados em grafos**. 2008. 61 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

JGRAPHT TEAM. **JGraphT**: a free Java graph library. [S.l.], 2005. Disponível em: <http://www.jgrapht.org/>. Acesso em: 30 mar. 2010.

KOCAY, William; KREHER, Donald L. **Graphs, algorithms, and optimization**. Boca Raton: Chapman & Hall/CRC, 2005.

PRESTES, Edson. **Teoria dos grafos**. [Rio Grande do Sul], 2012. Disponível em: <http://www.inf.ufrgs.br/~prestes/Courses/Graph%20Theory/GrafosA3.pdf>. Acesso em: 30 mar. 2014.

RABUSKE, Márcia A. **Introdução a teoria dos grafos**. Florianópolis: Ed. da UFSC, 1992.

RANKA, Sanjay; SAHNI, Sartaj. **Hypercube algorithms**: with applications to image processing and pattern recognition. New York: Springer-Verlag, 2011.

ROCHA, Anderson; DORINI, Leyza B. **Algoritmos gulosos**: definições e aplicações. Campinas, 2014. Disponível em: <http://www.ic.unicamp.br/~rocha/msc/complex/algoritmosGulososFinal.pdf>. Acesso em: 11 nov. 2014.

SANTOS, Rodrigo P. d.; COSTA, Heitor A. X. **Um software gráfico educacional para ensino de algorimos em grafos**. Minas Gerais, 2006. Disponível em: < http://www.cos.ufrj.br/~rps/pub/completos/2006/CIAWI.pdf >. Acesso em: 11 abr. 2014.

SKIENA, Steven S.; REVILLA, Miguel. **Programming challenges:** the programming contest training manual. New York: Springer-Verlag, 2003.

SZWARCFITER, Jayme L. **Grafos e algoritmos computacionais**. Rio de Janeiro: Campus, 1984.

VILLALOBOS, Alejandro R. Grafos: herramienta informática para el aprendizaje y resolución de problemas reales de teoría de grafos. In: CONGRESO DE INGENIERÍA DE ORGANIZACIÓN, 10., 2006, Valencia. **Anais…** Valencia: [s.n.], 2006. p. 1-10. Disponível em: <http://personales.upv.es/arodrigu/IDI/Grafos.pdf >. Acesso em: 30 mar. 2014.

ZATELLI, Maicon R. **Um framework para algoritmos baseados na teoria dos grafos**. 2010. 91 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.