UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA VISUAL PARA CRIAÇÃO E EXECUÇÃO DE ALGORITMOS APLICADOS SOBRE TEORIA DOS GRAFOS

SUSAN BRAUN

BLUMENAU
2009

2009/2-22

SUSAN BRAUN

FERRAMENTA VISUAL PARA CRIAÇÃO E EXECUÇÃO DE ALGORITMOS APLICADOS SOBRE TEORIA DOS GRAFOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Paulo César Rodacki Gomes, Dr. - Orientador

BLUMENAU
2009

2009/2-22

FERRAMENTA VISUAL PARA CRIAÇÃO E EXECUÇÃO DE ALGORITMOS APLICADOS SOBRE TEORIA DOS GRAFOS

Por

SUSAN BRAUN

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

Presidente:	Prof. Paulo César Rodacki Gomes, Dr .– Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Dr. – FURB
Membro:	Prof. Dalton Solano dos Reis, Ms. – FURB

Blumenau, 16 de dezembro de 2009.

Dedico este trabalho a minha família, aos amigos e especialmente aqueles que de alguma forma contribuíram para realização deste.

AGRADECIMENTOS

A Deus, por estar sempre presente em minha vida.

Aos meus pais, por sempre incentivarem meus estudos.

Ao meu noivo, Paulo Rosa, pela paciência e incentivo nas horas de desânimo, pela ajuda na realização deste e por estar sempre presente em minha vida.

Aos meus amigos, pelas ajudas, em especial à Fernanda Gums por pacientemente me auxiliar nas dúvidas encontradas e nos assuntos não compreendidos no decorrer deste curso.

Ao meu orientador, Paulo César Rodacki Gomes, pelas dicas, pelo auxílio e por ter acreditado na conclusão deste trabalho.

Grande homem é aquele que não perdeu o coração de criança.

J. Wu

RESUMO

Este trabalho apresenta o desenvolvimento de uma ferramenta para criação de algoritmos e a visualização da execução dos mesmos em grafos. Para a implementação dos algoritmos a ferramenta disponibiliza uma interface que permite ao usuário criar os algoritmos utilizando a linguagem Java, bem como compilar e visualizar os resultados da compilação. A ferramenta disponibiliza para a execução dos algoritmos uma interface visual que permite ao usuário montar um grafo e visualizar a execução do algoritmo e a coloração dos vértices. Para o desenvolvimento do trabalho foram utilizados a ferramenta Ant, a biblioteca Java Reflection e as bibliotecas Java Open Graphics Library (JOGL) e Document Object Model (DOM).

Palavras-chave: Grafos. Algoritmos.

ABSTRACT

This work describes an algorithm creating tool based on graphs structures and graph visualization. The tool has a interface that makes possible for the use to build, compile and executes algorithms in Java language. The tool has also a visual interface to build the graph and to visual inspection of algorithm execution and node colory. The tool was build with Java Reflection, Java Open Graphics Library (JOGL) and Document Object Model (DOM)..

Key-words: Graph. Algorithms.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ANT – *Another Neat Tool*

BFS – *Breadth First Search*

DFS – *Depth First Search*

DOM – *Document Object Model*

EVG - Editor Visual de Grafos

IDE – *Integrated Development Environment*

IUP – Interface com Usuário Portátil

JOGL – Java Open Graphics Library

JVM – Java Virtual Machine

LED – Linguagem de Especificação de Diálogos

OpenGL – *Open Graphics Library*

UC – Caso de Uso

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

W3C – *World Wide Web Consortium*

SUMÁRIO

1 INTRODUÇÃO

Atualmente dentro da área de tecnologia existe um amplo campo de problemas que requerem a construção de sistemas complexos, devido às combinações de seus componentes. Estes problemas abrangem processos industriais, análise de caminho crítico, tática e logística, sistemas de comunicação, estudo de transmissão de informações, escolha de uma rota ótima, fluxos em redes, redes elétricas, genética, economia, jogos, física, química, tecnologia de computadores, entre outros (SOUZA; VICENTE, 2008, p. 02).

Estes problemas podem ser facilmente resolvidos ao serem modelados em forma de grafos e então sobre eles utilizados teoremas e algoritmos da teoria dos grafos para obter conclusões e soluções satisfatórias.

Um grafo pode ser definido de forma simples por um conjunto de pontos, denominados vértices, interligados por retas, denominadas arestas. Já um algoritmo sobre um grafo pode ser definido como um conjunto de instruções e ações a serem realizadas sobre o grafo que permitam obter os resultados e soluções do problema encontrado.

Segundo Rezende (2006, p. 50), o desenvolvimento de algoritmos que contribuem para a resolução de problemas, como os acima citados, tem motivado muitos pesquisadores da matemática aplicada, da educação matemática, da engenharia e da informática a produzirem pesquisas nesta área.

A utilização na prática dos conceitos e técnicas da teoria dos grafos contribui para melhor compreensão do assunto. Entretanto é importante que os esforços concentrados durante os estudos práticos estejam focados nos algoritmos, evitando que a atenção seja desviada para forma como nodos e arestas serão implementados e os dados armazenados.

A partir dos fatos citados, o presente trabalho busca desenvolver uma ferramenta que permita ao usuário desenvolver algoritmos e visualizar sua execução. Desta forma a ferramenta disponibiliza uma interface para criação de algoritmos e outra para criação de grafos. A primeira permite ao usuário desenvolver seus próprios algoritmos a partir de uma estrutura de grafo disponível. Já a segunda permite ao usuário criar um grafo, selecionar qual algoritmo a ser executado e, por fim, visualizar sua execução.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta que permita a criação dinâmica de grafos e algoritmos e, que a partir do uso de reflexão computacional permita visualizar a execução e os resultados obtidos do algoritmo implementado, sobre o grafo criado.

Os objetivos específicos do trabalho são:

a) disponibilizar um editor para criação e compilação de algoritmos em Java;
b) disponibilizar uma interface gráfica para criação e visualização de grafos e de suas propriedades;
c) visualizar a execução e os resultados dos algoritmos implementados, sobre o grafo criado.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está organizado em três capítulos: fundamentação teórica, desenvolvimento e conclusões.

A fundamentação teórica apresenta os principais conceitos sobre grafos e alguns trabalhos correlatos, para que o leitor possa melhor entender o funcionamento da ferramenta.

Os principais requisitos, a especificação através de diagramas da *Unified Modeling Language* (UML) e detalhes do desenvolvimento da ferramenta, são apresentados no desenvolvimento.

Por fim, nas conclusões são discutidos os resultados obtidos, a usabilidade da ferramenta, bem como sugestões de implementações futuras.

2 FUNDAMENTAÇÃO TEÓRICA

A Teoria dos Grafos é um assunto amplamente estudado em cursos relacionados à computação e matemática. Seu conteúdo é de grande importância e utilizado na solução de muitos problemas nas mais diversas áreas, tais como logística, genética, redes e jogos.

Este capítulo apresenta os tópicos relevantes ao conceito de grafos, descrevendo sua definição, seus atributos, as principais famílias, as formas de representação e a importância de algoritmos em grafos. Em seguida são apresentados os trabalhos correlatos a ferramenta desenvolvida.

2.1 TEORIA DOS GRAFOS

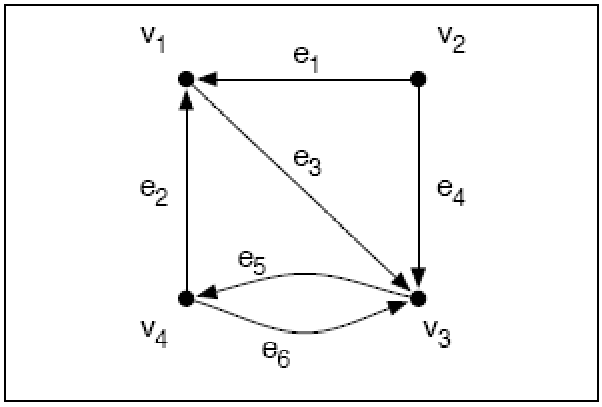
A Teoria dos Grafos é definida como uma área da matemática discreta que possibilita que problemas do mundo real sejam modelados como conjuntos finitos contendo objetos e relacionamentos. A partir destes conjuntos e relacionamentos é possível analisar situações e aplicar soluções (SANGIORGI, 2006, p. 04).

Para Gross e Yellen (2006, p. 02) um grafo `G = (V, E)` é composto por dois conjuntos `V` e `E`, sendo que os elementos de `V` são denominados vértices ou nodos e os elementos de `E` denominados arestas. Cada aresta possui um ou dois vértices associados a si.

2.1.1 Dígrafos

Segundo Diestel (2005, p. 31), em alguns casos o conceito de grafos não é exatamente adequado ao problema encontrado. Ao aplicar o uso de grafos sobre um fluxo de tráfico, alguns problemas como rodovias com apenas um sentido de direção são encontrados. Neste caso são necessários grafos com arestas em somente um sentido. Gomes (2007, p. 11) explica que neste caso são utilizados grafos dirigidos, nomeados de dígrafos. Em dígrafos as arestas apresentam um sentido que vai do vértice de origem para o vértice de destino, ou seja, possuem um único sentido. Para a representação gráfica de um dígrafo (Figura 1) são comumente utilizadas arestas em forma de flecha, que partem do vértice de origem e apontam

para o vértice de destino.

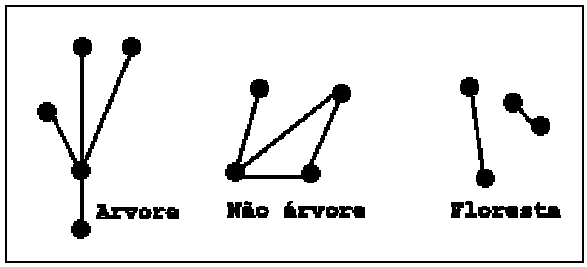


Fonte: Gomes (2007, p. 12).

Figura 1 – Exemplo de dígrafo

2.1.2 Árvores

Gross e Yellen (2006, p. 91) afirmam que árvores são importantes estruturas dentro da área da teoria dos grafos. Uma árvore é um grafo conexo que não possui ciclos. Ligado à definição de grafos esta a definição de floresta, sendo esta um grafo sem ciclos, porém não conexo. Assim em uma árvore existe somente um caminho possível entre dois vértices quaisquer. Já em uma floresta pode não existir este caminho, devido a não conexidade. A Figura 2 apresenta um exemplo de grafo árvore, seguido de um grafo não árvore e uma floresta.



Fonte: adaptado de Gross e Yellen (2006, p. 11).

Figura 2 - Exemplo de árvore, não árvore e floresta

2.1.3 Atributos de um grafo

Existem problemas importantes que necessitam vértices com atributos para que possam ser solucionados. Um exemplo é o peso de vértice que pode representar o custo de produção em uma fábrica (GROSS; YELLEN, 2006, p. 41).

Segundo Hackbarth (2008, p. 13), outro atributo importante é o custo de uma aresta, que nada mais é que um valor numérico associado a ela. Este atributo é muito utilizado em algoritmos que necessitam descobrir o menor caminho entre dois vértices.

Gross e Yellen (2006, p. 41) definem um terceiro atributo, a cor do vértice, que permite melhor visualizar os vértices. Observa-se que o atributo cor, não deve ser confundido com o problema da coloração de vértices.

Por fim, Gross e Yellen (2006, p. 42) apresentam um quarto atributo comumente utilizado, a descrição do vértice, que pode representar apenas nome ou um estado do vértice.

2.1.4 Parâmetros quantitativos

Gomes (2007, p. 13), afirma que normalmente são utilizados parâmetros quantitativos para auxiliar na resolução de um problema modelado pelo grafo. Dentre os parâmetros mais utilizados estão:

a) ordem: representa a quantidade de vértices em um grafo. É denotada pelo símbolo $|V|$ ou pela letra n;
b) tamanho: representa a quantidade de arestas existentes num grafo. É representado por $|E|$ ou pela letra m;
c) densidade: representa a relação entre a ordem e o tamanho. Um grafo é considerando denso quando possui muitas arestas para uma quantidade de vértices, ou é considerado esparso quando possui poucas arestas para uma determinada quantidade de vértices;
d) grau do vértice: representa a quantidade de arestas ligadas a um vértice. Em um grafo define-se o grau do vértice como a quantidade de arestas incidentes ao vértice, com cada laço sendo contado duas vezes. Representa-se o grau de um vértice por $grau(v)$. Em um dígrafo não se utiliza o conceito de grau do vértice, mas sim os conceitos de grau de entrada e grau de saída. O grau de entrada de um vértice representa a quantidade de arestas dirigidas que chegam ao vértice e é denotado pelo símbolo $grau_e(v)$. Já o grau de saída representa a quantidade de arestas dirigidas que saem do vértice e é denotado pelo símbolo $grau_s(v)$.

Na Figura 3 são apresentados exemplos dos parâmetros quantitativos. Ambos os grafos apresentados são de ordem cinco. Em relação ao tamanho o grafo A possui tamanho igual a

quatro enquanto que o grafo B possui tamanho igual a dezessete. O grafo A é um grafo esparso e o grafo B um grafo denso. Por fim o vértice A do grafo A possui grau igual a dois, enquanto que o vértice A do grafo B possui grau igual a oito.

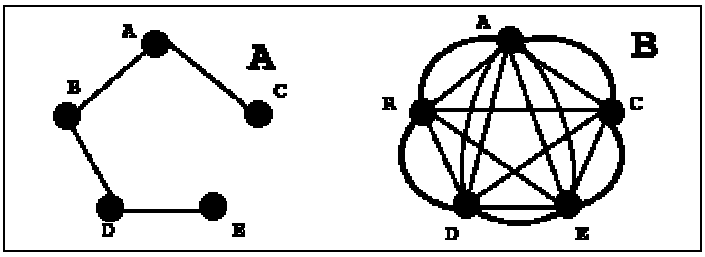
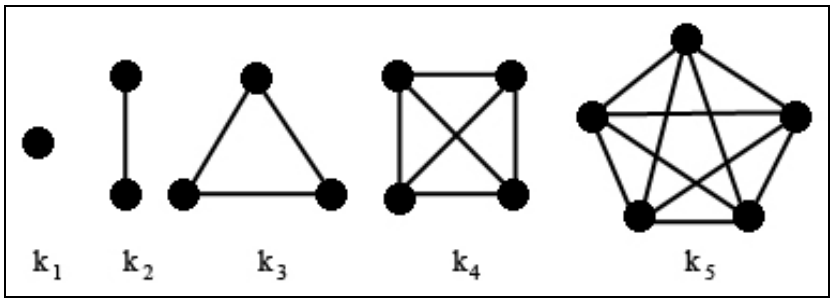


Figura 3- Grafo esparso e grafo denso

2.1.5 Famílias de grafos

Gross e Yellen (2006, p.10) destacam que comumente as principais famílias de grafos estão presentes nos problemas modelados, sendo elas o grafo completo, o grafo bipartido, o grafo regular e o grafo ciclo. Lopes (1999, p. 295) destaca também a família dos grafos conexos.

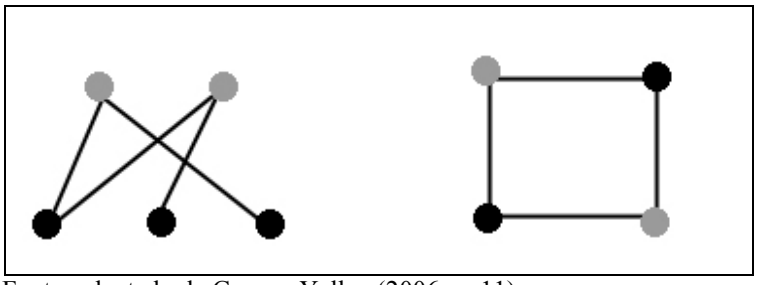
Um grafo completo (Figura 4) é um grafo simples que possui todos os pares de vértices ligados por uma aresta. Qualquer grafo completo com n vértices é denotado por K_n.



Fonte: adaptado de Gross e Yellen (2006, p. 11).

Figura 4 - Os cinco primeiros grafos completos

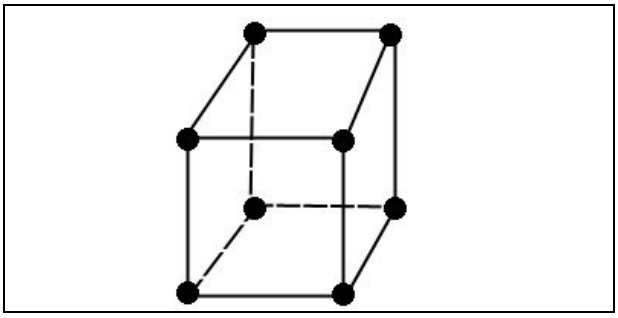
Um grafo bipartido (Figura 5) é um grafo G onde o conjunto de vértices V pode ser dividido em dois subconjuntos U e W, tal que cada aresta de G possui um ponto final em U e um ponto final em W.



Fonte: adaptado de Gross e Yellen (2006, p. 11).

Figura 5 - Dois grafos bipartidos

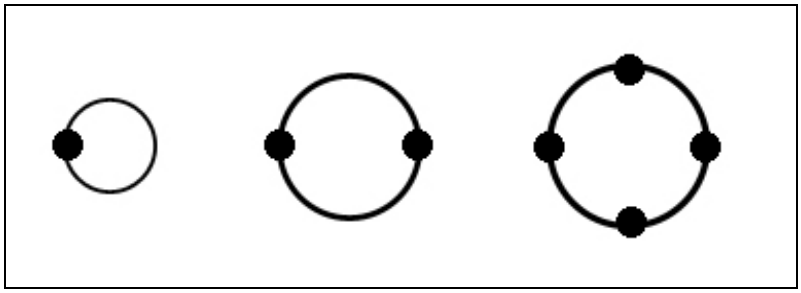
Um grafo regular (Figura 6) é um grafo em que todos os vértices possuem o mesmo grau.



Fonte: adaptado de Gross e Yellen (2006, p. 12).

Figura 6 - Um cubo é um grafo regular

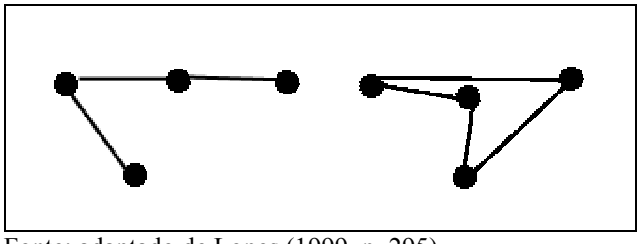
Um grafo ciclo (Figura 7) é um único vértice ligado à uma aresta, formando um laço, ou um grafo simples conexo, onde todos os vértices e arestas formam um círculo.



Fonte: adaptado de Gross e Yellen (2006, p. 13).

Figura 7 – Grafos ciclos

Um grafo conexo (Figura 8) é um grafo que contém ao menos um vértice a partir do qual se pode alcançar qualquer outro vértice. Se a partir de qualquer vértice do grafo for possível alcançar qualquer outro vértice o grafo é denominado fortemente conexo (Figura 8).



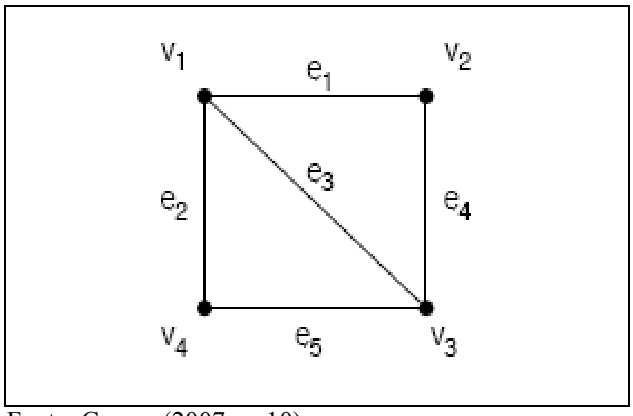
Fonte: adaptado de Lopes (1999, p. 295).

Figura 8 – Exemplo de grafo conexo e grafo fortemente conexo

2.1.6 Formas de representação

Existem várias maneiras de representar um grafo. Dentre as principais destacam-se a representação gráfica, a matriz de adjacência e a lista de adjacência.

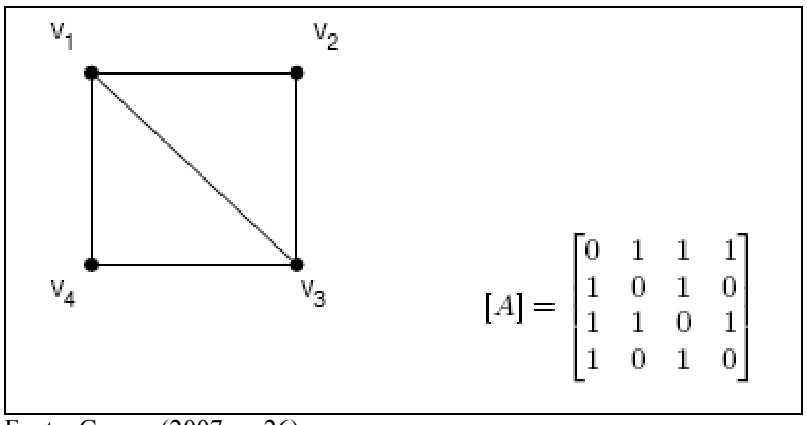
Gomes (2007, p. 10) explica que para fins didáticos, geralmente grafos são representados graficamente por desenhos de pontos interligados por linhas, sendo os pontos representantes dos vértices e as linhas das arestas (Figura 9).



Fonte: Gomes (2007, p. 10).

Figura 9 - Representação gráfica de um grafo

Outra forma de representar um grafo é através da matriz de adjacência. Gross e Yellen (2006, p. 76) explicam que uma matriz adjacência de um grafo `G`, denominado `AG`, possui linhas e colunas, ambas indexadas em ordem idêntica de V_G, tal que o valor é a quantidade de arestas entre `u` e `v`, se `u` for diferente de `v` ou o valor é a quantidade de arestas que partem e chegam em `v`, se `v` for igual a `u`. A Figura 10 apresenta um grafo juntamente com sua matriz de adjacência.

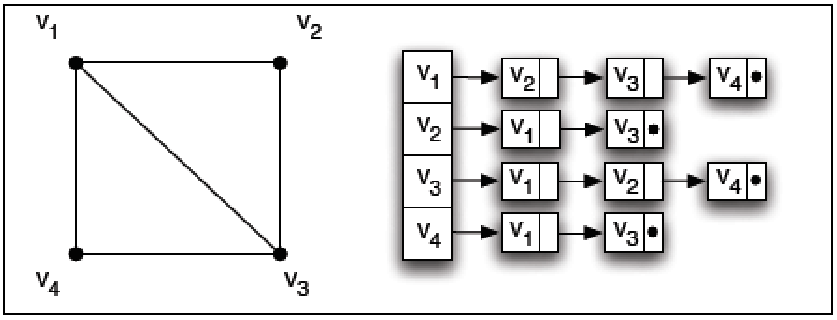


Fonte: Gomes (2007, p. 26).

Figura 10 - Matriz de adjacência de um grafo

A terceira forma de representação é a lista de adjacência. Gomes (2007, p. 27) define a lista de adjacência de um grafo `G = (V, E)` como um arranjo de `n` listas de adjacência,

representas por `Adj[v]`, sendo uma para cada vértice `v` do grafo. Cada uma destas listas é composta por referências aos vértices adjacentes a `v`, representando individualmente as arestas dos grafos, conforme mostrado esquematicamente na Figura 11.



Fonte: Gomes (2007, p. 28).

Figura 11 - Lista de adjacência de um grafo

2.1.7 Algoritmos em grafos

Em alguns casos é possível encontrar a solução de um problema apenas visualizando o seu grafo, entretanto em problemas maiores e mais complexos faz-se necessário o uso de um algoritmo para obter os resultados desejados.

Um algoritmo é a representação formal da solução para um problema sob a forma de um enunciado preciso com uma seqüência finita de passos (LOPES, 1999, p. 18).

Segundo Gomes (2007, p. 15), dentre os principais algoritmos da teoria dos grafos destacam-se os de busca. Estes algoritmos constituem métodos para promover a pesquisa em grafos, servindo assim como base para construção de algoritmos mais especializados como os algoritmos de Dijkstra, para cálculo de caminhos de custo mínimo e o de Prim, para determinação da árvore geradora de custo mínimo. Existem basicamente dois métodos de busca, sendo eles o de busca em largura onde um vértice inicial é definido e a busca se alastra uniformemente pelo grafo e a busca em profundidade, que pesquisa o grafo de uma forma recursiva, procurando sempre entrar mais profundamente na estrutura do grafo.

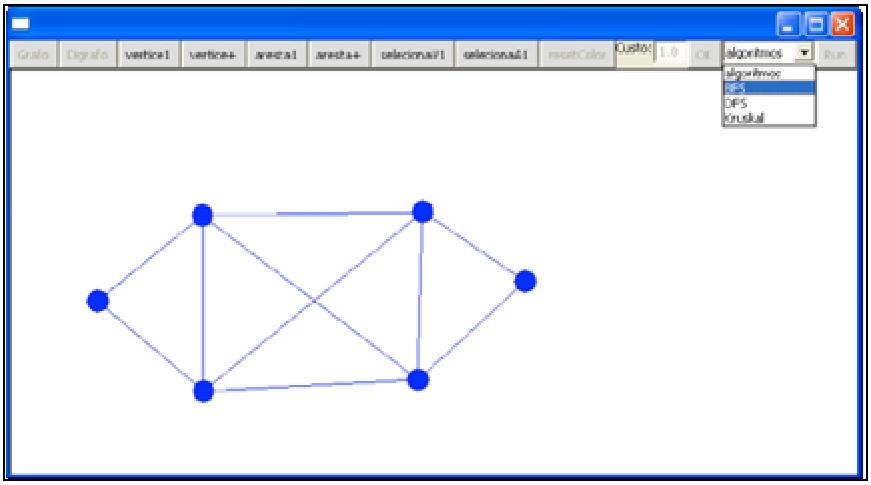
2.2 TRABALHOS CORRELATOS

A seguir estão relacionados três trabalhos com características semelhante aos

principais objetivos deste trabalho: “Ferramenta para representação gráfica do funcionamento de algoritmos aplicados em grafos” (HACKBART, 2008), “Grafos” (VILLALOBOS, 2006) e “RoxGT” (SANGIORGI, 2006).

Hackbarth (2008, p. 10) define uma ferramenta para representar graficamente estruturas de grafos (Figura 12). A ferramenta consiste na criação de uma interface gráfica, através da qual o usuário pode criar seu grafo, verificar o resultado dos algoritmos nele aplicados, assim como o caminho percorrido no grafo. Estão disponíveis para simulação os algoritmos *Breadth First Search* (BFS), *Depth First Search* (DFS) e Kruskal.

A ferramenta é implementada na linguagem C++, utilizando a biblioteca *Open Graphics Library* (OpenGL) e a Interface com Usuário Portátil (IUP) em conjunto com a Linguagem de Especificação de Diálogos (LED). Para a camada lógica do programa foi utilizada a biblioteca GraphObj, a qual foi desenvolvida no Departamento de Sistemas e Computação (DSC) da Universidade Regional de Blumenau (FURB) pelo professor Paulo Cesar Rodacki Gomes.



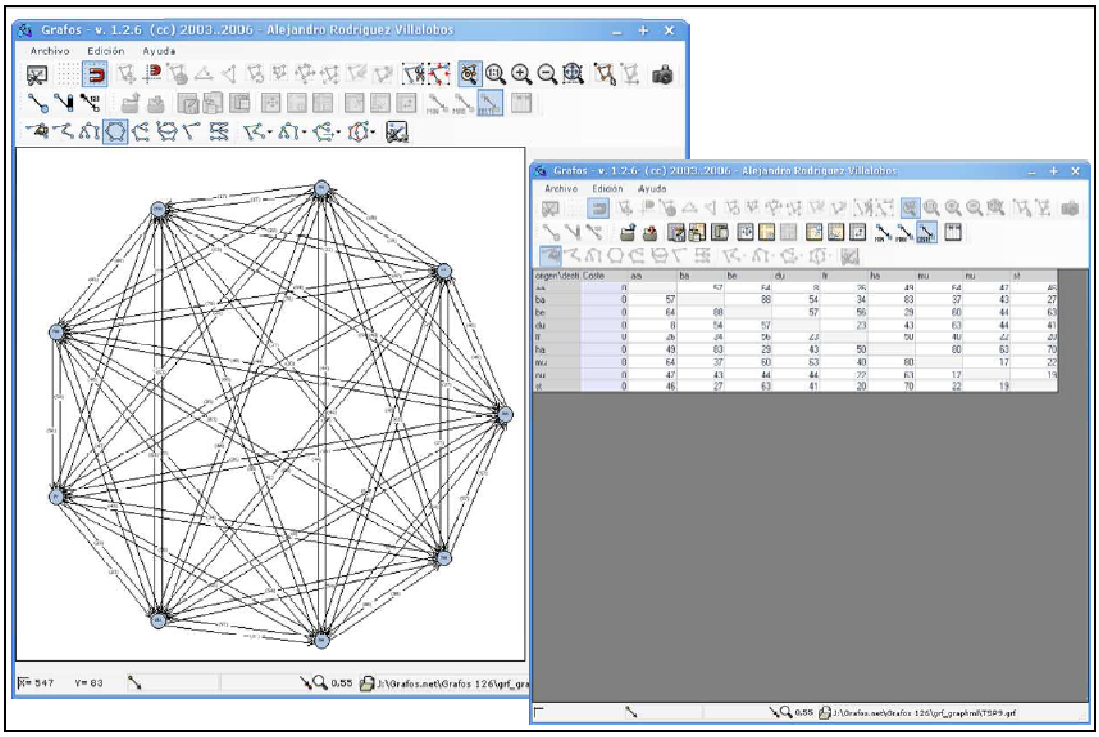
Fonte: Hackbarth (2008, p. 54).

Figura 12 – Interface da ferramenta

De acordo com Villalobos (2006, p. 01), Grafos é um software para construção, edição e análise de grafos que tem por principal objetivo o uso no ensino e aprendizagem da disciplina de Teoria dos Grafos e outras disciplinas relacionadas como a logística e desenho de redes. O software foi desenvolvido na linguagem Visual Basic e atualmente esta na versão 1.2.9.

O software Grafos permite a construção tanto de grafos dirigidos como não dirigidos, podendo seus vértices e nodos ter valores associados. Grafos podem ser trabalhados de forma gráfica, podendo arrastar e soltar vértices e arestas e na forma tabular, utilizando uma matriz que indica quais nodos relacionam-se (Figura 13).

Dentre os algoritmos disponibilizados estão os algoritmos de Dijkstra, Bellman-Ford, Floyd-Warshall, Kruskal, Prim, Ford-Fulkerson e caixeiro viajante.

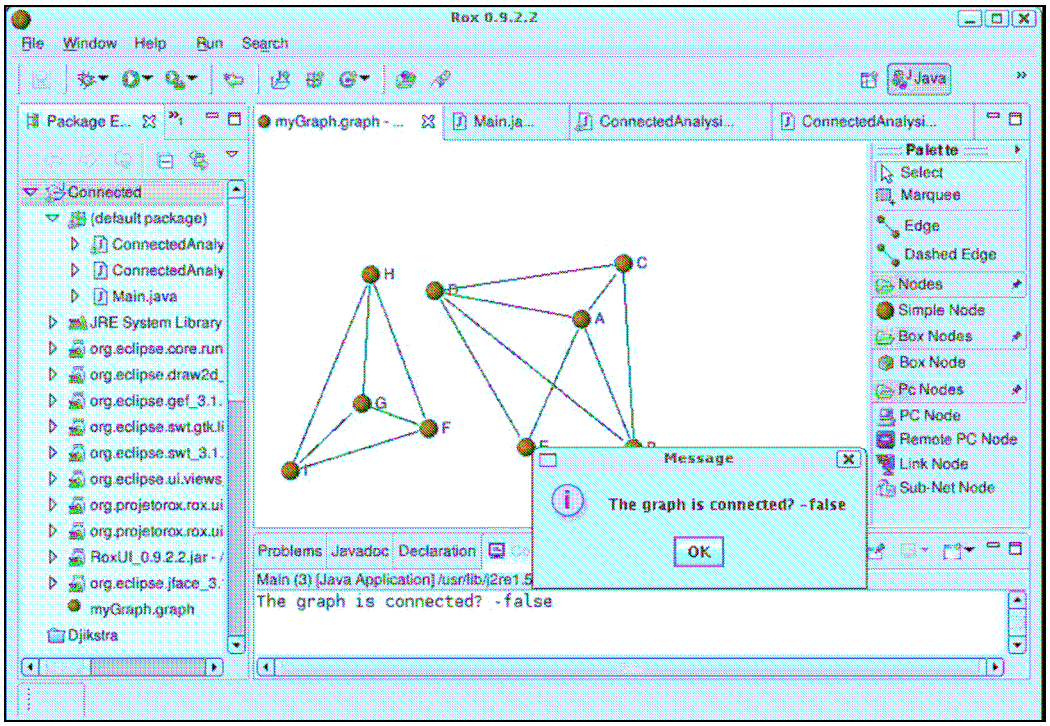


Fonte: Villalobos (2006, p. 02).

Figura 13 - Interface do sistema Grafos exibindo os dados de forma gráfica e tabular

O RoxGT (Figura 14) é um *framework* para aplicações baseadas em Teoria dos Grafos. O mesmo está implementado em forma de *plug-in* para a *Integrated Development Environment* (IDE) Eclipse.

Segundo Sangiorgi (2006, p. 9), a arquitetura do *framework* é definida em três partes distintas, sendo um editor de grafos que permite a edição visual da estrutura do grafo, a biblioteca Graph, que contém a estrutura básica, a camada de execução de algoritmos e as classes a serem herdadas pela aplicação cliente e o *framework* RoxGT que encapsula estas duas funcionalidades e se comunica com a plataforma Eclipse.



Fonte: Sangiorgi (2006, p. 37).

Figura 14 - Interface do *framework* RoxGT

Para melhor compreensão das funcionalidades de cada ferramenta o Quadro 1 apresenta uma comparação entre os trabalhos correlatos.

	HACKBART 2008	VILLALOBOS 2006	SANGIORGI 2006
permite criar grafos visualmente	Sim	Sim	Sim
disponibiliza algoritmos a serem executados	Sim	Sim	Não
permite criar novos algoritmos	Não	Não	Sim
permite visualizar as propriedades do grafo	Não	Sim	Sim
permite salvar o grafo criado	Não	Sim	Não

Quadro 1 - Análise comparativa entre os trabalhos correlatos

3 DESENVOLVIMENTO DA FERRAMENTA

Neste capítulo são abordados os principais pontos no desenvolvimento do presente trabalho, denominado Editor Visual de Grafos (EVG). Primeiramente são apresentados os requisitos da ferramenta e a sua especificação. Na sequência, a implementação, onde são comentadas as técnicas e ferramentas utilizadas, o desenvolvimento de cada módulo e operacionalidade da ferramenta. Por fim são discutidos os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta deve permitir ao usuário visualizar a execução de seus algoritmos sobre um grafo, além de permitir a criação de grafos e visualização de suas propriedades. Para isto a ferramenta deve disponibilizar uma interface onde o usuário redija e compile seus algoritmos e outra interface onde o usuário crie seus grafos, selecione um algoritmo e visualize sua execução.

Na sequência são apresentados respectivamente os requisitos funcionais (Quadro 2) e não funcionais (Quadro 3) atendidos pela ferramenta.

Requisitos Funcionais (RF)	**Caso de Uso (UC)**
RF01: a ferramenta deve permitir ao usuário criar e salvar novos algoritmos, bem como editar algoritmos já existentes	UC02.01, UC02.02, UC02.03, UC02.04
RF02: a ferramenta deve disponibilizar a linguagem Java para o desenvolvimento dos algoritmos	UC02.01, UC02.02
RF03: a ferramenta deve compilar os algoritmos criados	UC02.05
RF04: a ferramenta deve exibir os erros de compilação encontrados	UC02.05
RF05: a ferramenta deverá disponibilizar uma interface gráfica para criação de novos grafos	UC03.01 à UC03.07
RF06: a ferramenta deve exibir as propriedades dos grafos criados	UC03.08
RF07: a ferramenta deve executar os algoritmos criados	UC03.09
RF08: a ferramenta deve exibir a execução do algoritmo sobre o grafo através da coloração dos vértices e arestas determinadas nos algoritmos	UC03.09
RF09: a ferramenta deve exibir em forma de texto o resultado do algoritmo criado	UC03.09
RF10: a ferramenta deve permitir salvar e abrir os grafos criados	UC03.10, UC03.13
RF11: a ferramenta deve disponibilizar uma função para gerar grafos automaticamente	UC03.11, UC03.12

Quadro 2 - Requisitos funcionais

Requisitos Não Funcionais (RNF)
RNF01: a ferramenta deve ser implementada na linguagem Java SE 6.0
RNF02: a ferramenta deve utilizar a biblioteca *Java Open Graphics Library* (JOGL) para interface gráfica
RNF03: a ferramenta deve utilizar a ferramenta *Another Neat Tool* (ANT) para compilar e gerar o *bytecode* dos algoritmos criados
RNF04: a ferramenta deve utilizar a biblioteca Java Reflection para instanciar os algoritmos criados e compilados em tempo de execução

Quadro 3 - Requisitos não funcionais

3.2 ESPECIFICAÇÃO

Esta seção descreve a especificação do EVG, através do uso dos conceitos de orientação a objetos e da *Unified Modeling Language* (UML). São apresentados os diagramas de caso de uso, de classes e de sequência. A modelagem dos diagramas foi feita utilizando a ferramenta *Enterprise Architect*.

3.2.1 Diagramas de casos de uso

A seguir são apresentados os diagramas de três pacotes: `Seleção do projeto`, `Compilação` e `Criação do grafo`, com seus respectivos casos de uso, que representam as funcionalidades da ferramenta.

O primeiro pacote, designado `Seleção de projeto` (Figura 15), apresenta os casos de uso (UC) de criação e abertura de um projeto. A execução de ao menos um destes casos de uso é obrigatória para que se seja possível acessar as demais funcionalidades da ferramenta.

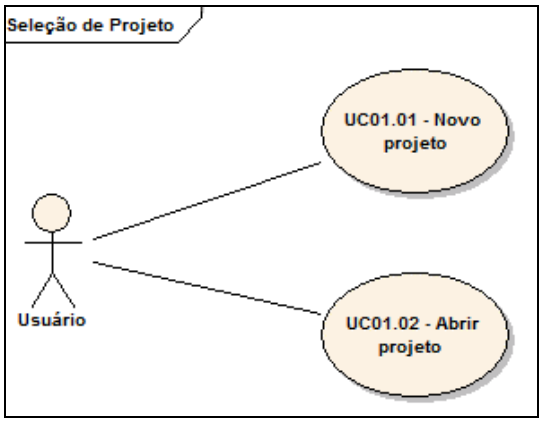


Figura 15 - Pacote: `Seleção do projeto`

No Quadro 4 são apresentados os cenários de cada caso de uso sendo que, cada caso de uso possui um cenário principal e cenários de exceção.

CASO DE USO	CENÁRIOS
UC01.01 – Novo Projeto	**Criar novo projeto (Principal)** 01 - O usuário seleciona o item "Novo projeto" no menu principal. 02 - O EVG solicita o nome e o caminho para o novo projeto. 03 - O usuário preenche os dados e confirma. 04 - O EVG cria um novo projeto. **Caminho já existente (Exceção)** No passo 03, caso o usuário informe um nome de projeto e caminho já existentes é apresentada uma mensagem solicitando um novo nome ou caminho para o projeto. **Dados inválidos (Exceção)** No passo 03, caso o usuário não preencha todos os dados é apresentada uma mensagem solicitando o preenchimento dos dados.
UC01.02 – Abrir projeto	**Abrir projeto (Principal)** 01 - O usuário seleciona o item "Abrir projeto" no menu principal. 02 - O EVG apresenta tela para escolha de projetos. 03 - O usuário seleciona o diretório do projeto e confirma. 04 - O EVG carrega os dados do projeto selecionado. **Projeto inválido (Exceção)** No passo 03, caso o diretório selecionado não contenha um projeto, o sistema apresenta mensagem informando a inexistência do projeto.

Quadro 4 - Descrição dos casos de uso do pacote: `Seleção do projeto`

Na Figura 16, é apresentado o pacote `Compilação`, que relaciona os casos de uso referente às funcionalidades de criação e compilação dos algoritmos executados sobre os grafos criados.

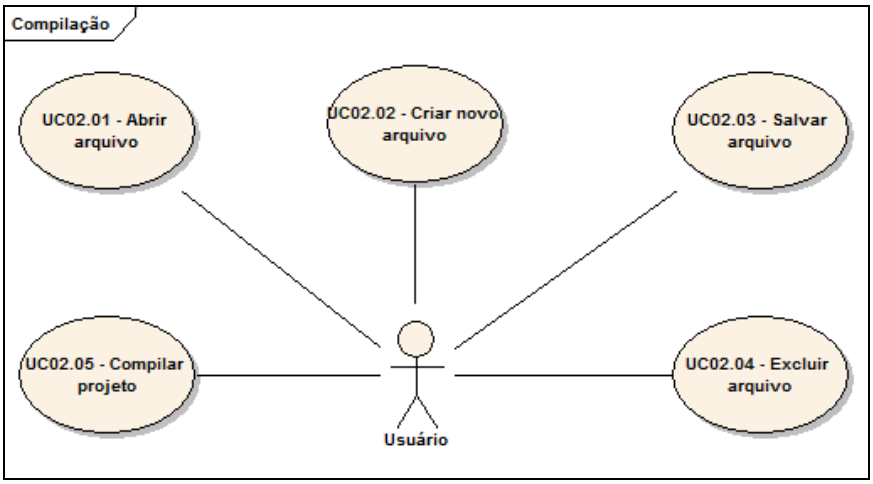


Figura 16 - Pacote: `Compilação`

O Quadro 5 apresenta os cenários de cada caso de uso do pacote: `Compilação`.

CASO DE USO	CENÁRIOS
UC02.01 – Abrir arquivo	**Abrir arquivo (Principal)** 01 - O usuário solicita abertura de um arquivo. 02 - O EVG apresenta os arquivos relacionados ao projeto. 03 - O usuário seleciona o arquivo desejado. 04 - O EVG disponibiliza o conteúdo do arquivo selecionado para edição.
UC02.02 – Cria novo arquivo	**Criar novo arquivo (Principal)** 01 - O usuário solicita ao sistema criar um novo arquivo. 02 - O EVG disponibiliza um arquivo vazio para edição.
UC02.03 – Salvar arquivo	**Salvar arquivo (Principal)** 01 - O usuário solicita ao sistema salvar o arquivo. 02 - O EVG solicita o nome do arquivo a ser salvo. 03 - O usuário informa o nome do arquivo. 04 - O EVG salva o conteúdo do arquivo. **Salvar arquivo existente (Alternativo)** No passo 02, caso o arquivo a ser salvo seja um arquivo já existente aberto pelo usuário o sistema executa o passo 04. **Arquivo já existente (Exceção)** No passo 03 caso o usuário informe o nome de um arquivo já existente sistema apresenta mensagem informando que o arquivo já existe.
UC02.04 – Excluir arquivo	**Excluir arquivo (Principal)** 01 - O usuário seleciona um arquivo. 02 - O usuário solicita exclusão do arquivo. 03 - O EVG exclui o arquivo.
UC02.05 – Compilar projeto	**Compilar projeto (Principal)** 01 - O usuário solicita a compilação do projeto. 02 - O EVG compila todos os algoritmos relacionados ao projeto. 03 - O EVG apresenta mensagem de compilação finalizada. **Falha na compilação (Exceção)** No passo 03, caso existam erros de compilação sistema apresenta uma mensagem informando o erro encontrado, a linha de código e a classe onde erro ocorre.

Quadro 5 - Descrição dos casos de uso do pacote: `Compilação`

A representação as funcionalidades presentes na interface de criação de grafos e execução dos algoritmos, é apresentado no pacote: `Criação de grafos` (Figura 17).

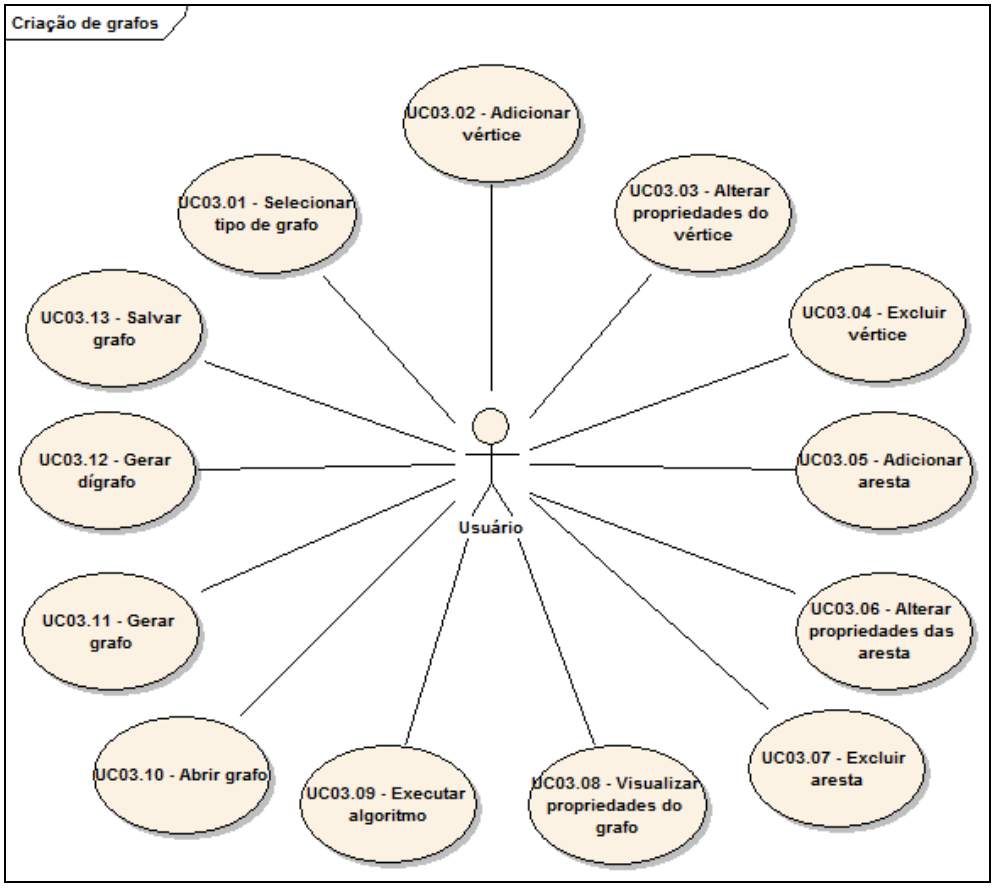


Figura 17 - Pacote: `Criação de grafos`

No Quadro 6 são apresentados os cenários de cada caso de uso presentes no pacote `Criação de grafos`.

CASO DE USO	CENÁRIOS
UC03.01 – Selecionar tipo de grafo	**Selecionar tipo de grafo (Principal)** 01 - O usuário seleciona o tipo de grafo a ser trabalhado. 02 - O EVG disponibiliza os componentes (vértices e arestas) correspondentes ao tipo de grafo selecionado. **Limpar dados (Alternativo)** No passo 02, caso já exista um grafo na ferramenta o mesmo é apagado.
UC03.02 – Adicionar vértice	**Adicionar vértice (Principal)** 01 - O usuário seleciona a inserção de um vértice. 02 - O usuário seleciona o ponto de inserção (coordenas X e Y) na tela. 03 - O EVG cria um novo vértice e adiciona ao grafo. 04 - O EVG apresenta o vértice na tela.
UC03.03 – Alterar propriedades do vértice	**Alterar propriedades do vértice (Principal)** 01 - O usuário solicita a alteração do vértice selecionado. 02 - O EVG apresenta tela com os dados do vértice. 03 - O usuário altera os dados desejados e confirma. 04 - O EVG aplica as alterações sobre o vértice.
UC03.04 – Excluir vértice	**Excluir vértice (Principal)** 01 - O usuário solicita a exclusão de um vértice selecionado. 02 - O EVG exclui as arestas associadas ao vértice. 03 - O EVG exclui o vértice selecionado
UC03.05 – Adicionar aresta	**Adicionar aresta (Principal)** 01 - O usuário seleciona a inserção de uma aresta. 02 - O usuário seleciona os dois vértices ao qual a aresta deve estar associada. 03 - O EVG cria uma nova aresta e adiciona ao grafo. 04 - O EVG apresenta a aresta na tela.
UC03.06 – Alterar propriedades da aresta	**Alterar propriedades da aresta (Principal)** 01 - O usuário solicita a alteração da aresta selecionada. 02 - O EVG apresenta tela com os dados da aresta. 03 - O usuário altera os dados desejados e confirma. 04 - O EVG aplica as alterações sobre a aresta.
UC03.07 – Excluir aresta	**Excluir aresta (Principal)** 01 - O usuário solicita a exclusão de uma aresta selecionada. 03 - O EVG exclui a aresta selecionada.

UC03.08 - Visualizar propriedades do grafo	**Visualizar propriedades do grafo (Principal)** 01 - O usuário solicita a visualização das propriedades do grafo. 02 - O EVG apresenta uma tela contendo os valores das principais propriedades do vértice.
UC03.09 – Executar algoritmo	**Executar algoritmo (Principal)** 01 - O usuário seleciona um algoritmo a ser executado e confirma. 02 - O EVG inicia a executa do algoritmo. 03 - O EVG apresenta a saída do algoritmo. **Excluir arquivo (Alternativo)** No passo 02, caso o algoritmo necessite vértices como parâmetros de entrada: 2.1 - O EVG apresenta uma tela exibindo os vértices do grafo e solicitando a seleção dos que servirão como entrada. 2.2 - O usuário seleciona os vértices desejados e confirma.
UC03.10 – Abrir grafo	**Abrir grafo (Principal)** 01 - O usuário solicita a abertura de grafo salvo. 02 - O EVG apresenta tela para busca do grafo. 03 - O EVG seleciona um arquivo contendo os dados do grafo e confirma. 04 - O EVG apresenta o grafo na tela.
UC03.11 – Gerar grafo	**Gerar grafo (Principal)** 01 - O usuário solicita a geração de um grafo. 02 - O EVG solicita a quantidade de vértices. 03 - O usuário informa a quantidade de vértices desejados. 04 - O EVG gera um grafo e apresenta na tela.
UC03.12 – Gerar dígrafo	**Gerar dígrafo (Principal)** 01 - O usuário solicita a geração de um dígrafo. 02 - O EVG solicita a quantidade de vértices. 03 - O usuário informa a quantidade de vértices desejados. 04 - O EVG gera um dígrafo e apresenta na tela.
UC03.13 – Salvar grafo	**Salvar grafo (Principal)** 01 - O usuário solicita salvar o grafo apresentado na tela. 02 - O EVG solicita o nome e o formato do arquivo. 03 - O usuário informa os dados e confirma. 04 - O EVG salva os dados do grafo.

	Arquivo existente (Exceção) No passo 03 caso usuário informe um nome de arquivo já existente, sistema apresenta mensagem solicitando a permissão de sobrescrita do arquivo. **Dados incompletos (Exceção)** No passo 03, caso o usuário não preencha todos os dados o sistema apresenta uma mensagem solicitando o preenchimento de todos os dados.

Quadro 6 - Cenários do caso de uso da criação de grafos

3.2.2 Diagramas de classe

Esta seção apresenta os diagramas de classe das três funcionalidades do EVG e o diagrama de classe da estrutura de grafos. Para melhor visualização foram omitidos os atributos e métodos das classes. No apêndice A são exibidas as classes detalhadas.

Na Figura 18 é mostrado o diagrama de classe da estrutura de grafos. Estas classes são responsáveis pelo armazenamento da estrutura e das informações dos grafos criados. As classes `GrafoAbstract`, `VerticeAbstract` e `ArestaAbstract` compõe a estrutura genérica do grafo e as classes `Vertice`, `Aresta` e `ArestaDirigida` são responsáveis pelas propriedades particulares de um grafo ou um dígrafo. As classes `GrafoAtributosAdicionais`, `ArestaAtributosAdicionais` e `VerticeAtributosAdicionais` permitem armazenar informações adicionais em relação ao grafo e seus vértices e arestas. As classes `Adicionais`, `DadosInterface` e `AlgoritmoInterface` são utilizadas como classes auxiliares na comunicação entre o grafo e a interface de criação de grafos.

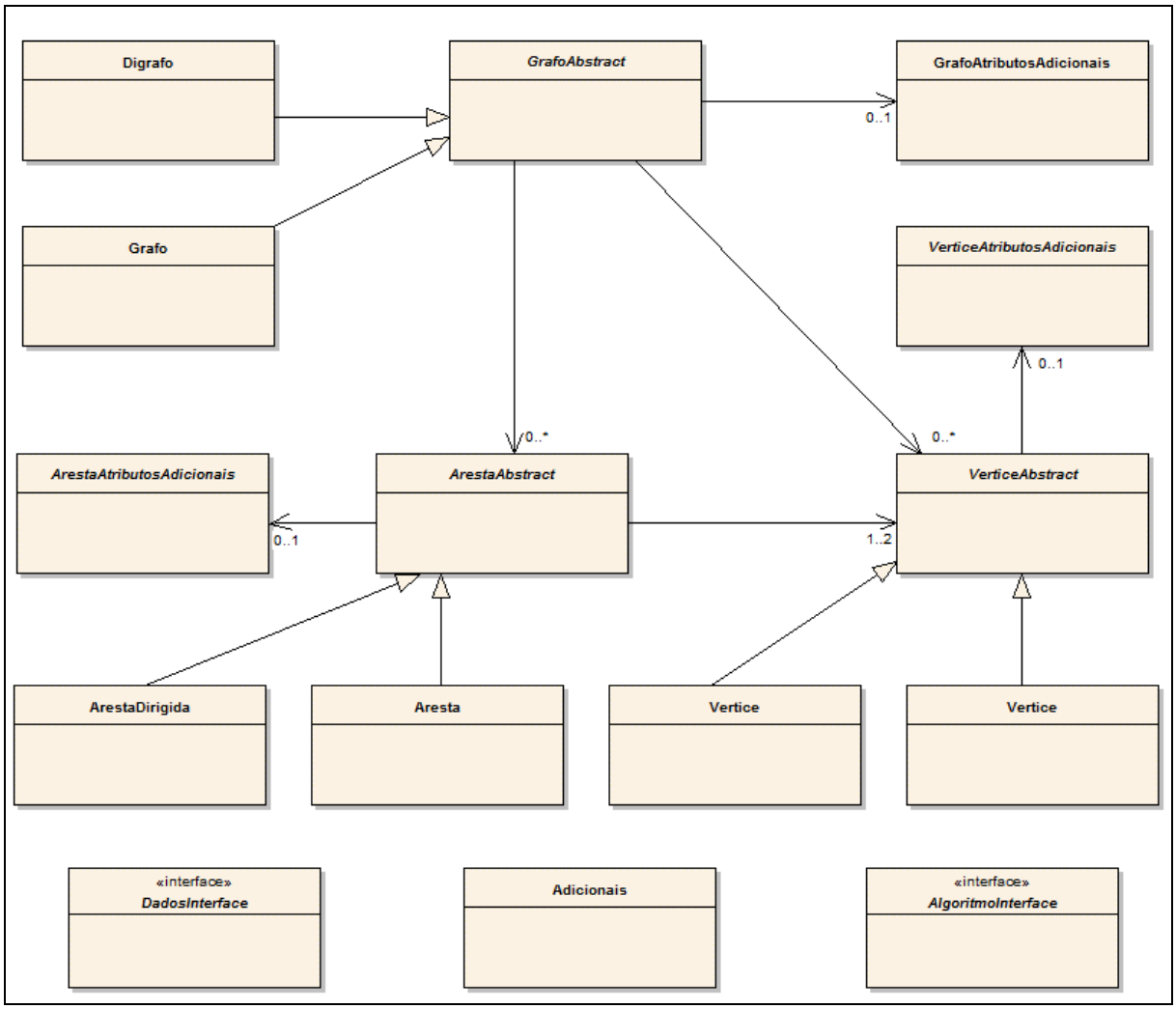


Figura 18 - Diagrama de classe da estrutura de grafos

A Figura 19 exibe o diagrama de classes da interface inicial do sistema. A classe `Principal` é responsável pela interação com o usuário. As classes `CaminhoNovoProjeto` e `Arquivo` são responsáveis pela criação de novos projetos. A classe `PropriedadesProjeto` é responsável por buscar os dados do último projeto aberto e a classe `BatInterface` por carregar a interface de criação de grafos.

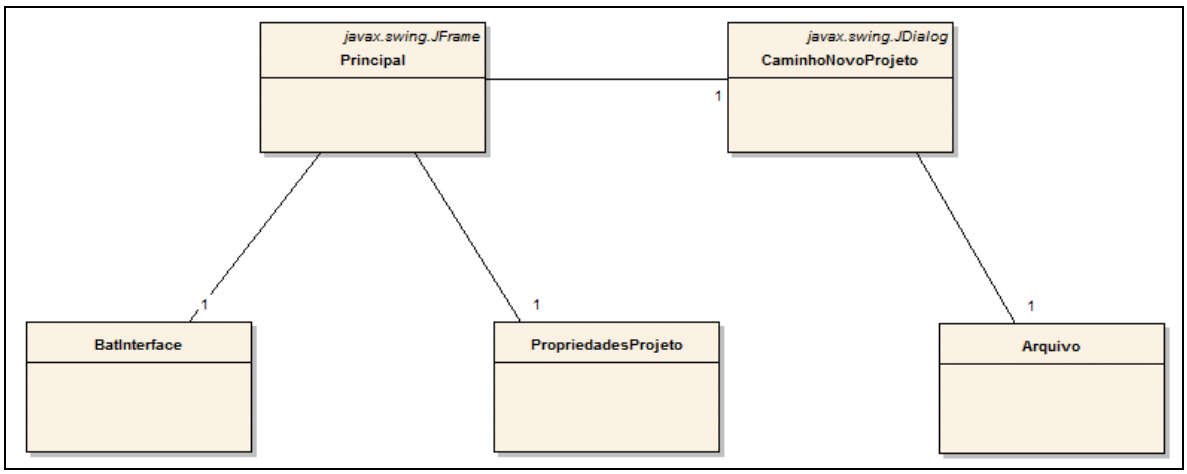


Figura 19 - Diagrama de classes da interface inicial do sistema

O diagrama de classes da interface de compilação pode ser observado na Figura 20. A classe `Compilador` é responsável pela interação com o usuário e por abrir, criar, salvar e excluir arquivos. A classe `Filtros` é responsável por filtrar os tipos de arquivos permitidos. As classes `JanelaInterna` e `Arquivo` são responsáveis por exibir o conteúdo dos arquivos. As classes `ConstrutorBuildXML`, `BuildLoggerAdapter`, juntamente com a classe `Compilador` são responsáveis pela compilação dos arquivos.

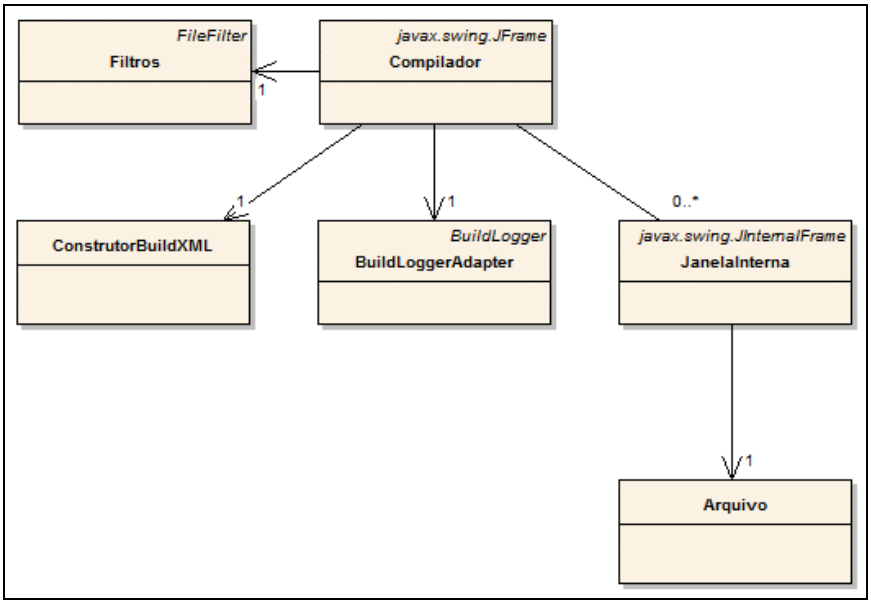


Figura 20 - Diagrama de classes da interface de compilação

Pode-se observar na Figura 21 o diagrama de classes da interface de criação de grafos. As classes `Main`, `CarregarJar`, `ListaArquivos` e `Propriedades` são responsáveis por carregar os dados iniciais da aplicação, tais como os algoritmos criados e caminho do projeto. A classe `Interface` é responsável pela interação com o usuário. As classes `Filtros`, `XML`, `ArestaXML`, `VerticeXML`, `GrafoXML` são responsáveis por abrir e salvar os grafos criados. As classes `PropriedadesArestas`, `PropriedadesVertices`, `PropriedadesGrafos` são responsáveis por exibir as propriedades e atributos do grafo e de seus vértices e arestas. As classes `ListenerMouse` e `GrafoPopup` são responsáveis por captar as coordenadas do mouse e ação executada. As classes `GLRenderizador` e `GLJanela2D` são responsáveis pelo desenho do grafo. As classes `VerticeParametro` e `SaidaAlgoritmo` juntamente com a classe `Interface` são responsáveis pela execução dos algoritmos.

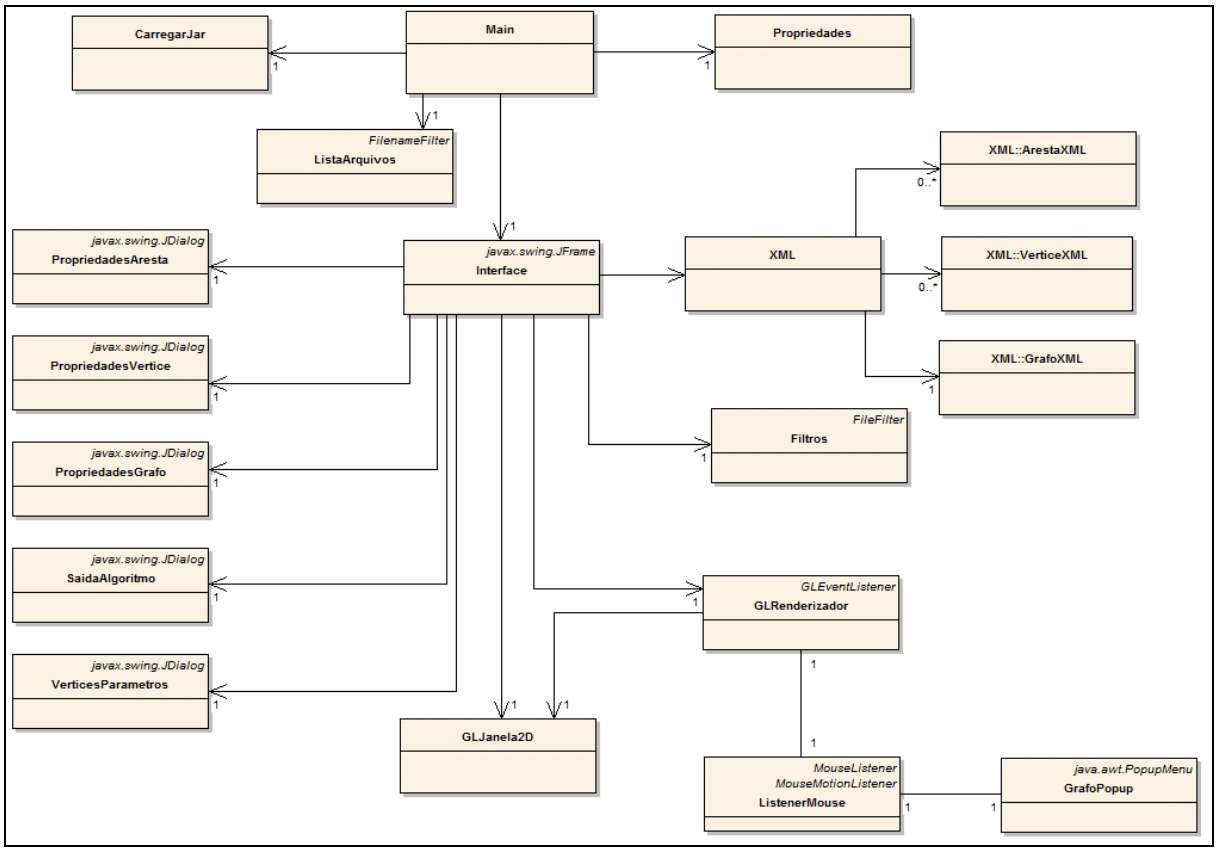


Figura 21 - Diagrama de classes da interface de criação de grafos

3.2.3 Diagramas de atividades

A interação do usuário com o EVG é apresentada através do diagrama de atividades (Figura 22), exibindo os passos executados desde a criação de um projeto até a execução de um algoritmo.

O diagrama esta dividido em três partes módulos principais sendo eles `Interface inicial`, a `Interface de Compilação` e a `Interface de criação de grafos`. Na `Interface inicial` são relacionadas as atividades de criação de novos projetos e abertura de projetos existentes. Na `Interface de Compilação` estão as atividades relacionadas à criação dos algoritmos do projeto. Por fim, na `Interface de criação de grafos` estão as atividades de criação de grafos e execução de algoritmos.

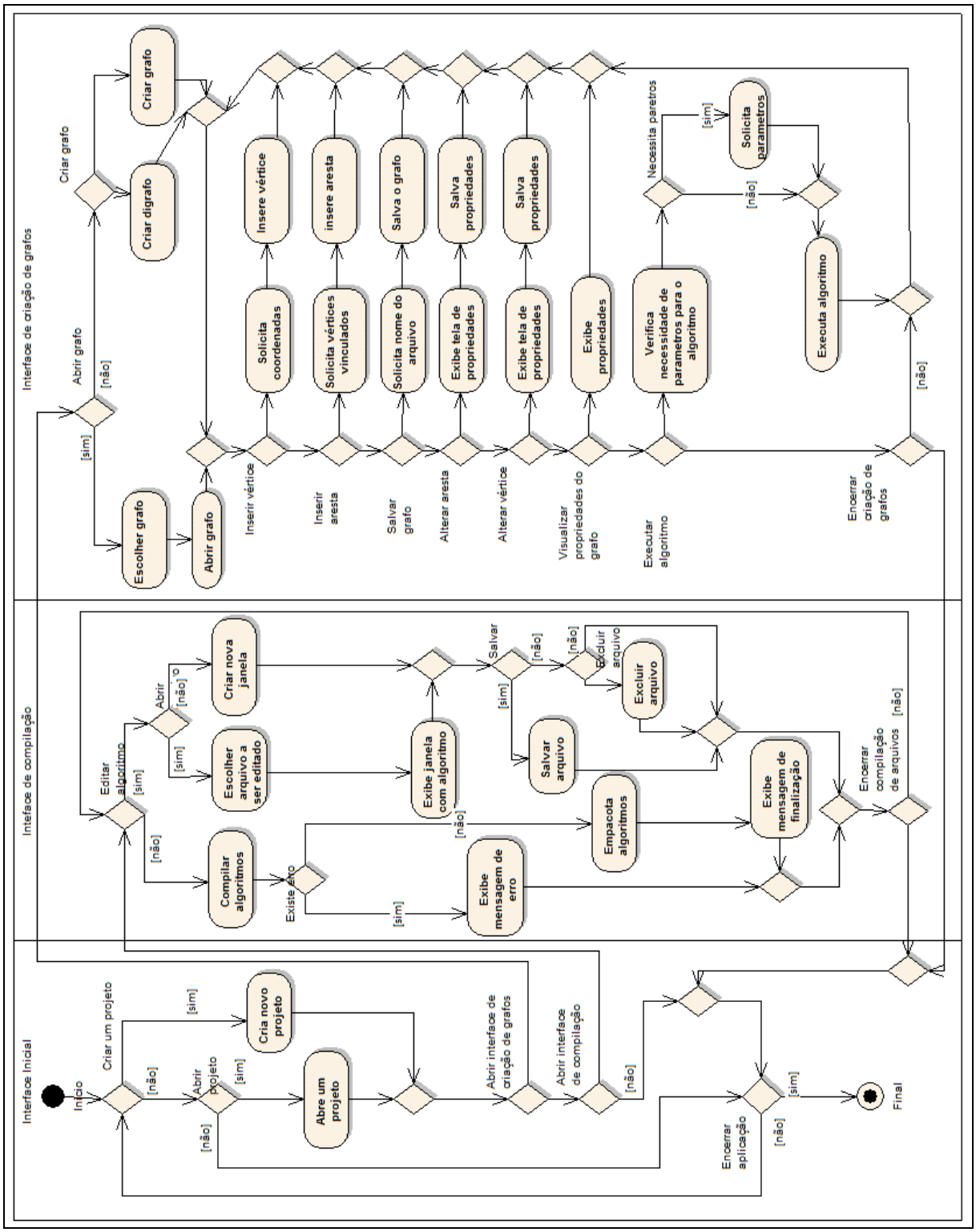


Figura 22 - Diagrama de atividades

3.3 IMPLEMENTAÇÃO

São apresentadas nesta seção as técnicas e ferramentas utilizadas na implementação do EVG, a descrição do desenvolvimento do trabalho e por fim a operacionalidade da ferramenta.

3.3.1 Técnicas e ferramentas utilizadas

A implementação do EVG foi realizada utilizando a linguagem de programação Java, permitindo a portabilidade da ferramenta para diversos sistemas operacionais. Foram também utilizadas as bibliotecas Java Open Graphics Library (JOGL) para construção da interface gráfica, *Another Neat Tool* (ANT) para compilação em tempo de execução e *Document Object Model* (DOM) para criação dos arquivos *eXtensible Markup Language* (XML).

Foi utilizada a tecnologia de Reflexão Computacional para instanciar os algoritmos criados em tempo de execução e o padrão Graphml para salvar os grafos criados.

Para a codificação da ferramenta foi utilizado o ambiente de programação NetBeans 6.7.1.

3.3.1.1 ANT

Inicialmente o ANT era parte do projeto Tomcat e era utilizado somente para compilar e construir o projeto. Esta ferramenta foi desenvolvida por James Duncan Davidson e foi doada à empresa Apache Software Foundation (APACHE ANT, 2008).

Mais tarde observou-se que o ANT poderia resolver diversos problemas encontrados em outras ferramentas de compilação. Assim, o ANT foi transformado em um projeto próprio e oficialmente lançado em julho de dois mil (APACHE ANT, 2008).

Segundo Bell (2005, p. 30), o ANT destaca-se de outras ferramentas de compilação por seu arquivo de informações não ser um formato proprietário e por ser desenvolvido em Java, tornando assim o projeto multiplaforma.

O ANT utiliza um arquivo no formato XML para armazenar todas as informações necessárias para a construção de um projeto. O arquivo é divido em tarefas, denominadas

tasks, que podem entre diversas tarefas copiar, excluir, compilar e executar um projeto ou de seus arquivos.

Um exemplo do arquivo de informações pode ser visto no Quadro 7.

```
01 <?xml version="1.0"?>
02 <project name="Hello" defaul="compile">
03    <target name="clean" description="remove itermediate files">
04       <delete dir="classes"/>
05    </target>
06
07    <target name="compile" description="compile the Java source code to
08 class files">
09       <mkdir dir="classes"/>
10          <javac srcdir="."destdir="classes"/>
11    </target>
12
13    <target name="jar" depends="compile" description="create a Jar file
14 for the application">
15       <jar destfile="hello.jar">
16          <fileset dir="classes" include="**/*.java"/>
17       </jar>
18    </target>
19 </project>
20
21
```

Fonte: Bell (2005, p. 248-249).

Quadro 7 - Exemplo de arquivo de informações do ANT

3.3.1.2 DOM

A *Document Object Model* (DOM) é uma especificação da *World Wide Web Consortium* (W3C) independente de plataforma e linguagem, onde pode-se alterar e editar a estrutura de um documento (WORLD WIDE WEB CONSORTIUM, 2005).

A biblioteca DOM do pacote org.w3c.dom da Sun Microsystem fornece interfaces de acesso a documentos DOM. Esta biblioteca permite que programas possam acessar e atualizar dinamicamente o conteúdo e a estrutura de documentos de uma maneira padrão, além de se poder trabalhar com cada um destes elementos separadamente (SUN MICROSYSTEM, 2008).

O uso desta biblioteca no EVG permite que os grafos criados possam ser salvos e reutilizados posteriormente, além de serem utilizados por outras aplicações que utilizem o mesmo formato de arquivo.

3.3.1.3 Graphml

Graphml é um formato de arquivo para grafos. Este formato não utiliza uma sintaxe personalizada, sendo baseado em XML, uma extensão flexível que permite que o mesmo possa ser utilizado por diversas aplicações que geram, salvam ou processam grafos (GRAPHML TEAM, 2007).

Este formato inclui suporte para grafos não dirigidos, dígrafos e grafos mistos[1], além de seus vértices e arestas. Além destes Graphml disponibiliza um mecanismo para definição de atributos adicionais, tais com cor e peso, para vértices e arestas (GRAPHML TEAM, 2007).

Todas as informações do grafo estão contidas no elemento `graphml`, sendo formado por definições de grafos (elemento `graph`), novos atributos (elemento `key`), vértices (elemento `node`), arestas (elemento `edge`) e hiperarestas (elemento `hyperedege`) (SENA, p. 71, 2006).

No Quadro 8 é exibido um exemplo de grafo salvo no formato Graphml. Nas linhas 6 e 9 estão definidos os atributos para um vértice e uma aresta, definidos pelo elemento `key` e identificado pelo valor de `id`. Para definir estes atributos, é necessário informar os tipos de elementos que podem possuí-lo (`for`), um nome (`attr. name`) e um tipo (`attr. type`).

```
01 <?xml version="1.0" encoding="UTF-8"?>
02  <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
03        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04        xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
05          http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
06    <key id="d0" for="node" attr.name="color" attr.type="string">
07      <default>yellow</default>
08    </key>
09    <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
10    <graph id="G" edgedefault="undirected">
11      <node id="n0">
12        <data key="d0">green</data>
13      </node>
14      <node id="n1"/>
15      <node id="n2"/>
16      <edge id="e0" source="n0" target="n2">
17        <data key="d1">1.0</data>
18      </edge>
19      <edge id="e1" source="n0" target="n1"/>
20    </graph>
21  </graphml>
```

Fonte: Graphml Team (2007).

Quadro 8 - Exemplo de formato Graphml

A linha 10 define os dados de um grafo, identificado pelo elemento `graph`. Neste é

[1] Grafos mistos são grafos que apresentam simultaneamente arestas dirigidas e arestas não dirigidas.

possível informar alguns parâmetros do grafo, por exemplo, o tipo das arestas (parâmetro `edgedefault`), se direcionadas ou não.

A partir da linha 11 estão os dados dos vértices (`node`) que são referenciados pelo identificador (`id`), juntamente com parâmetros e atributos adicionais. A partir da linha 16, são listados os dados das arestas (`edge`), definidas através de um identificador (`id`), o nó fonte (`source`) e o nó destino (`target`).

3.3.1.4 JOGL

A OpenGL é uma biblioteca de rotinas gráficas bidimensional (2D) e tridimensional (3D) extremamente portável e rápida, utilizada para o desenvolvimento de aplicações de Computação Gráfica, tais como jogos e sistemas de visualização com um alto grau de realismo. A maior vantagem na sua utilização é a rapidez, uma vez que incorpora vários algoritmos otimizados, incluindo o desenho de primitivas gráficas, o mapeamento de texturas e outros efeitos especiais (COHEN; MANSSOUR, 2006, p. 18).

Cohen e Massour (2006, p.18), afirmam que a especificação OpenGL é gerenciada por um consórcio independente, criado em 1992, o Architecture Review Board (ARB), sendo este constituído por várias empresas líderes da área como 3Dlab, Apple Computer, NVIDEA, SGI e SUN. Este grupo é responsável pela aprovação de novas funcionalidades, versões e extensões da OpenGL. A versão atual desta especificação é 2.0.

Atualmente, a OpenGL é reconhecida e aceita como uma biblioteca padrão para o desenvolvimento de aplicações gráficas 3D interativas e que geram imagens em tempo real.

Segundo Cohen e Mansour (2006, p. 364), Java Open Graphics Library (JOGL) é uma implementação de referência que permite o desenvolvimento de programas Java com OpenGL e atualmente prove acesso à versão OpenGL2.0. A biblioteca OpenGL é acessada através de chamadas *Java Native Interface*[2] (JNI) permitindo o mapeamento direto entre as funções disponíveis na biblioteca desenvolvida em C e os métodos disponíveis na biblioteca JOGL (DAVISON, 2007, p. 378).

[2] Segundo Liang(1999, p. 3) JNI é uma funcionalidade do Java que permite incorporar ao código fonte, códigos nativos escritos em outras linguagens, como se fossem escritos em Java.

3.3.1.5 JAVA REFLECTION

No domínio de Ciências da Computação, reflexão computacional remete a dois conceitos: introspecção e redirecionamento. O primeiro denota a capacidade de um sistema computacional examinar sua própria estrutura, estado e representação. Estes três fatores são denominados meta-informação e representam toda e qualquer informação contida e manipulável por um sistema computacional que seja referente a si próprio. O segundo denota a capacidade de auto-modificação de um sistema computacional (SENRA, 2001, p. 02). Assim, reflexão computacional é a capacidade de um sistema computacional refletir sobre suas próprias ações e ajustar-se conforme a variedade de condições a que é submetido.

Perez e Zancanela (2003, p. 100) explicam que a arquitetura de um sistema reflexivo é divido em dois níveis: meta-nível e nível base. O meta-nível é responsável por gerar uma auto-representação do sistema de forma a gerar o conhecimento sobre sua estrutura e seu comportamento. Já o nível base contém os componentes responsáveis pelas funcionalidades básicas do sistema.

A biblioteca `Java Reflection` possibilita visualizar a estrutura de um objeto. É possível, dentro dos limites de segurança do Java, visualizar os construtores, métodos e campos de uma classe, além de alterar valores de campos, invocar métodos dinamicamente e instanciar novos objetos a partir de classes nunca vistas pela aplicação (NIEMEYER; KNUDSEN, 2005, p. 180).

3.3.2 Desenvolvimento do EVG

O desenvolvimento do EVG foi dividido em três módulos principais, o desenvolvimento da estrutura do grafo, o desenvolvimento da interface de compilação e o desenvolvimento da interface de criação de grafos. A seguir é descrita a implementação de cada um dos módulos.

3.3.2.1 Desenvolvimento da estrutura do grafo

A estrutura do grafo é responsável pelo correto armazenamento dos vértices, arestas,

grafos e dígrafos criados. Esta é a estrutura que esta disponível ao usuário para a criação dos algoritmos que o mesmo desenvolverá. Esta seção apresenta as principais classes da estrutura do grafo.

Um dígrafo é um grafo dirigido, desta forma todos os atributos e propriedades existentes em um grafo existem em um dígrafo. Embora parte das propriedades de um dígrafo sejam calculas de forma diferente das de um grafo, a estrutura de armazenamento pode ser a mesma, o que permite que determinados algoritmos possam ser aplicados em grafos e dígrafos sem que nenhum tratamento ou distinção necessite ser feita. A classe abstrata `GrafoAbstract` é responsável por esta estrutura, parte de seu código é exibida no Quadro 9.

```
01  public abstract class GrafoAbstract {
02
03      private List<ArestaAbstract> listaArestas;
04      private List<VerticeAbstract> listaVertices;
05      private GrafoAtributosAdicionais atributos;
06
07      /**
08       * Retorna se o grafo é nulo
09       * @return se o grafo é nulo
10       */
11      public boolean ehNulo(){
12          if(getListaVertices().size() == 0)
13              return true;
14          return false;
15      }
16
17      public abstract boolean ehCompleto();
```

Quadro 9 - Classe `GrafoAbstract`

Os métodos abstratos representam as propriedades do grafo que necessitam de um cálculo específico e devem ser implementados em classes específicas.

A classe `Grafo` é responsável por implementar os cálculos específicos de um grafo. Um trecho de seu código é apresentado no Quadro 10. Já a classe `Digrafo` é responsável por implementar os cálculos específicos de um dígrafo, sendo um trecho de código pode ser visto no Quadro 11.

```
01  public class Grafo extends GrafoAbstract {
02
03      /**
04       * Retorna se o grafo é completo
05       * @return se o grafo é completo
06       */
07      @Override
08      public boolean ehCompleto() {
09          for (int i = 0; i < getListaVertices().size(); i++) {
10              VerticeAbstract vAUm = getListaVertices().get(i);
11              for (int j = 0; j < getListaVertices().size(); j++) {
12                  if(i != j){
13                      VerticeAbstract vADois  = getListaVertices().get(j);
14                       if (!(isAdjacente(vAUm, vADois) ||
15                             isAdjacente(vADois, vAUm))) {
16                          return false;
17                       }
18                  }
19              l }
20          }
21          return true;
22      }
```

Quadro 10 - Classe `Grafo`

```
01  public class Digrafo extends GrafoAbstract {
02
03      /**
04       * Retorna se o dígrafo é completo
05       * @return se o dígrafo é completo
06       */
07      @Override
08      public boolean ehCompleto() {
09
10          int size = getListaVertices().size();
11          for (VerticeAbstract vAbs : getListaVertices()) {
12              Vertice v = (Vertice) vAbs;
13              if (v.getVerticesSaida().size() < size - 1) {
14                  return false;
15              }
16              for (VerticeAbstract vAbsAdj : getListaVertices()) {
17                  if (v != vAbsAdj) {
18                     if (!v.getVerticesSaida().contains((Vertice)
19  vAbsAdj)) {
20                          return false;
21                     }
22                  }
23              }
24          }
25          return true;
26      }
27
```

Quadro 11 - Classe `Digrafo`

Para que se possa construir um grafo, é necessária a presença de vértices e arestas. De forma semelhante à implementação do grafo é feita a implementação dos vértices e das arestas.

A classe abstrata `VerticeAbstract` implementa os métodos comuns de um vértice em

um grafo e um vértice em um dígrafo. Um trecho de seu código é exposto no Quadro 12.

```
01 public abstract class VerticeAbstract {
02
03     protected Object id;
04     protected Float valor;
05     protected List<ArestaAbstract> listaArestas;
06     protected VerticeAtributosAdicionais atributosAdicionais;
07     protected Color;
08     protected Color corInterna;
09     private double x;
10     private double y;
11
12     /**
13      * Retorna o identificador do vértice
14      * @return identifcador
15      */
16     public Object getId() {
17         return id;
18     }
19
20     /**
21      * Retorna o valor do vértice
22      * @return valor
23      */
24     public Float getValor() {
25         return valor;
26     }
```

Quadro 12 - Classe `VerticeAbstract`

A classe `Vertice` do pacote grafo não necessita de nenhum cálculo especial e somente implementa a classe `VerticeAbstract`, enquanto que a classe `Vertice` do pacote `digrafo` disponibiliza os métodos para busca dos vértices aos quais as arestas de entrada e de saída estão ligadas, bem como os métodos para busca das arestas de entrada e de saída. O Quadro 13 contém um trecho do código da classe `Vertice`.

```
01 public class Vertice extends VerticeAbstract{
02
03     /**
04      * Retorna os vértices as quais as arestas de saída estao ligadas
05      * @return lista de vértices
06      */
07     public ArrayList<Vertice> getVerticesSaida(){
08         ArrayList<Vertice> lista = new ArrayList<Vertice>();
09         for(ArestaAbstract aAbs: super.getListaArestas()){
10             if(((ArestaDirigida) aAbs).getVerticeIncio() == this){
11                 lista.add(((ArestaDirigida) aAbs).getVerticeFim());
12             }
13         }
14        return lista;
15     }
16
17     /**
18      * Retorna as arestas de saída
19      * @return lista de arestas
20      */
21     public ArrayList<ArestaDirigida> getArestasSaida(){
22         ArrayList<ArestaDirigida> lista = new
23 ArrayList<ArestaDirigida>();
24         for(ArestaAbstract aAbs: super.getListaArestas()){
25             if(((ArestaDirigida) aAbs).getVerticeIncio() == this){
26                 lista.add(((ArestaDirigida) aAbs));
27             }
28         }
29        return lista;
30     }
```

Quadro 13 - Classe `Vertice`

Por fim, a classe `ArestaAbstract` implementa os métodos comuns a uma aresta e uma aresta dirigida. Um trecho de seu código é mostrado noQuadro 14.

```
01  public abstract class ArestaAbstract {
02
03      protected ArestaAtributosAdicionais atributosAdicionais;
04      protected VerticeAbstract verticeUm;
05      protected VerticeAbstract verticeDois;
06      protected Object id;
07      protected Float valor;
08      protected Color cor;
09
10      /**
11       * Cria uma nova aresta
12       * @param verticeInicio
13       * @param verticeDois
14       */
15      public ArestaAbstract(VerticeAbstract verticeUm,
16                            VerticeAbstract verticeDois) throws Exception
17  {
18          if (verticeUm == null || verticeDois == null) {
19              throw new Exception("Uma aresta não pode conter"+
20                                  " vértices sem valor");
21          }
22          this.verticeUm = verticeUm;
23          this.verticeDois = verticeDois;
24      }
25
26      /**
27       * Retorna o vértice inverso ao informado
28       * @param vertice
29       * @return vértice inverso
30       */
31      public VerticeAbstract getVerticeInverso(VerticeAbstract vertice) {
32          if (vertice == null) {
33              return null;
34          } else if (vertice.equals(getVerticeUm())) {
35              return getVerticeDois();
36          } else if (vertice.equals(getVerticeDois())) {
37              return getVerticeUm();
38          } else {
39              return null;
40          }
41      }
42
```

Quadro 14 - Classe `ArestaAbstract`

As classes `Aresta` e `ArestaDirigida` implementam a classe `ArestaAbstract`. Os métodos disponíveis na classe `ArestaAbstract` são os necessários para classe `Aresta`, desta forma a mesma não necessita sobrescrever nenhum método e nem implementar novos métodos. Já a classe `ArestaDirigida` disponibiliza também os métodos para busca do vértice inicial e final, visto que o conhecimento dos mesmos é de alta importância em um dígrafo. O Quadro 15 exibe um trecho da classe `ArestaDirigida`.

```
01  public class ArestaDirigida extends ArestaAbstract {
02
03      /**
04       * Construtor da classe
05       * @param inicio vértice de saída da aresta
06       * @param fim vértice de chegada da aresta
07       */
08      public ArestaDirigida(Vertice inicio, Vertice fim) throws Exception
09  {
10          super(inicio, fim);
11      }
12
13      /**
14       * Retorna o vértice de saída da aresta
15       * @return vértice
16       */
17      public Vertice getVerticeInicio() {
18          return (Vertice) getVerticeUm();
19      }
20
21      /**
22       * Retorna o vértice de chegada da aresta
23       * @return vértice
24       */
25      public Vertice getVerticeFim() {
26          return (Vertice) getVerticeDois();
27      }
```

Quadro 15 - Classe `ArestaDirigida`

3.3.2.2 Desenvolvimento da interface de compilação

Para que o usuário possa criar seus próprios algoritmos sem se preocupar em como o mesmo deve ser tratado posteriormente, é disponibilizada uma interface que permite compilar, exibir os erros existentes, gerar o arquivo `.class`, bem como compactar todos os algoritmos em um arquivo `.jar` utilizado pela Java Virtual Machine (JVM).

A compilação dos algoritmos é feita quando solicitada pelo usuário. Caso não existam erros nos algoritmos, implicitamente é feita a construção do arquivo `.jar`. O código fonte deste procedimento é mostrado no Quadro 16.

```
01  try {
02      ConstrutorBuildXML builXml = new
03  ConstrutorBuildXML(diretorioProjeto);
04      builXml.salvarBuildXML();
05      File buildFile = new File(diretorioProjeto.getAbsolutePath()+
06                                "\\build.xml");
07      Project p = new Project();
08      p.setUserProperty("ant.file", buildFile.getAbsolutePath());
09      p.init();
10      ProjectHelper helper = ProjectHelper.getProjectHelper();
11      p.addReference("ant.projectHelper", helper);
12      helper.parse(p, buildFile);
13      p.addBuildListener(new BuildLoggerAdapter(statusJTA));
14      p.executeTarget(p.getDefaultTarget());
15      statusJTA.setText("Projeto compilado com sucesso.");
16  } catch (Exception e) {
17          statusJTA.setText("Foram encontrados erros durante "+
18         "a compilação : \n" +e.getMessage() + "\n" +
19  statusJTA.getText());
20     }
```

Quadro 16 - Compilação do algoritmo

A classe `ConstrutorBuildXML` é responsável por criar o arquivo `build.xml`. Este arquivo contém todas as informações necessárias para que os algoritmos possam ser compilados e o arquivo `.jar` criado. No Quadro 17 é apresentado o arquivo `build.xml` criado.

```
01  <project name="grafos" default="empacotar">
02       <property name="sourcedir" value="${basedir}/src"/>
03       <property name="targetdir" value="${basedir}/bin"/>
04       <property name="librarydir" value="${basedir}/lib"/>
05           <path id="libraries">
06               <fileset dir="${librarydir}">
07                   <include name="*.jar"/>
08               </fileset>
09           </path>
10       <target name="limpar">
11            <delete dir="${targetdir}"/>
12           <mkdir dir="${targetdir}"/>
13       </target>
14       <target name="compilar" depends="limpar, copiar">
15           <javac srcdir="${sourcedir}"
16               destdir="${targetdir}"
17               classpathref="libraries" fork="yes"  />
18       </target>
19       <target name="copiar">
20            <copy todir="${targetdir}">
21               <fileset dir="${sourcedir}">
22                   <exclude name="**/*.java"/>
23               </fileset>
24           </copy>
25       </target>
26       <target name="empacotar" depends="compilar">
27             <jar destfile="${targetdir}/algoritmos.jar"
28  basedir="${targetdir}" />
29       </target>
30  </project>
```

Quadro 17 - Arquivo `build.xml` criado pela ferramenta

As classes `Project` e `ProjectHelper` pertencem a biblioteca do projeto ANT. A classe `Project` executa os passos descritos no arquivo `build.xml`. A classe `ProjectHelper` é responsável por retornar o resultado de cada passo que é executado, permitindo a exibição dos erros, quando os mesmos existirem. A classe `BuildLoggerAdapter` implementa a classe `BuildLogger` também pertencente a biblioteca ANT, e permite que sejam identificados os erros. O Quadro 18 apresenta o código responsável pela identificação de erros.

```
01  Public class BuildLoggerAdapter implements BuildLogger {
02
03      private JTextArea output;
04
05      public BuildLoggerAdapter(JTextArea output) {
06          this.output = output;
07      }
08
09      public final void messageLogged(final BuildEvent event) {
10
11          if (event.getPriority() == Project.MSG_WARN ||
12              event.getPriority() == Project.MSG_ERR) {
13              output.setText(output.getText() + "\n" +
14  event.getMessage());
15          }
16      }
```

Quadro 18 - Identificação de possíveis erros

3.3.2.3 Desenvolvimento da interface de criação de grafos

A interface de criação de grafos é responsável por armazenar a estrutura do grafo criado, desenhar o grafo na tela e executar os algoritmos solicitados, para isto são necessárias classes que controlem a interação com o usuário, que tratem o desenho do grafo e que controle a execução dos algoritmos. A seguir são descritas as principais classes desta interface.

A classe `Interface` controla a interação com o usuário, nela estão disponíveis os botões para inserção de vértices e arestas, visualização das propriedades, solicitação da execução de um algoritmo, abertura e gravação dos grafos.

Ao ser instanciada, a classe `Interface` carrega a lista dos algoritmos disponíveis para execução e adiciona os mesmos ao *classpath* da aplicação. O Quadro 19 mostra a rotina para carregar todos os algoritmos disponíveis e o Quadro 20 a rotina para adicionar os algoritmos ao *classpath*.

```
01  File atual =
02        new
03  File(caminhoProjeto.getAbsolutePath()+"\\bin\\algoritmos.jar");
04  CarregarJar.adicionaAoClassPath(atual.toURI().toURL());
05  for (Class c : CarregarJar.getClasses(atual)) {
06     if (c.newInstance() instanceof AlgoritmoInterface) {
07         algoritmosJCB.addItem(new ItemComboClasse(c));
08     }
09  }
```

Quadro 19 - Rotina para carregar os algoritmos disponíveis

```
01  private static final Class[] parameters = new Class[]{URL.class};
02
03   public static void adicionaAoClassPath(URL u) throws IOException {
04       URLClassLoader sysloader =
05            (URLClassLoader) ClassLoader.getSystemClassLoader();
06       Class sysclass = URLClassLoader.class;
07       try {
08            Method metodo =
09                 sysclass.getDeclaredMethod("addURL", parameters);
10            metodo.setAccessible(true);
11            metodo.invoke(sysloader, new Object[]{u});
12       } catch (Throwable t) {
13            throw new IOException("Erro: " + u.getFile() +
14                                  " não pode ser carregado.");
15       }
16  }
17
```

Quadro 20 - Carregamento dos algoritmos no `classpath`

Ao solicitar a criação de um novo grafo, bem como solicitar a inserção de vértices ou arestas a classe `Interface` interage com a classe `GLRenderizador` informando a mesma a ação a ser executada.

A classe `GLRenderizador` é responsável pelo armazenamento dos vértices e arestas que são inseridos, bem como o desenho dos mesmos na tela. Esta classe estende da classe `GLEventListener` da biblioteca JOGL. Um trecho de seu código, exibindo a função de desenho do grafo, é mostrado no Quadro 21.

```
01  @Override
02  public void display(GLAutoDrawable drawable) {
03       GL gl = drawable.getGL();
04       this.pegaMatrizesGL(gl);
05       this.glAutoDrawable = drawable;
06       gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
07  gl.glMatrixMode(GL.GL_MODELVIEW);
08  gl.glLoadIdentity();
09  glu = new GLU();
10  glu.gluOrtho2D(dimensoes.getXMin(),
11                 dimensoes.getXMax(),
12                 dimensoes.getYMin(),
13                 dimensoes.getYMax());
14  if (getModelo() != null) {
15       for(VerticeAbstract vertex : getModelo().getListaVertices()) {
16            Color c = vertex.getCor() !=null ?
17                      vertex.getCor():Color.WHITE;
18            gl.glColor3f(c.getRed() / 255.0f,
19                         c.getGreen() / 255.0f,
20                         c.getBlue() / 255.0f);
21            desenhaVertice(gl, vertex.getX(), vertex.getY());
22       }
23       for(ArestaAbstract aresta : getModelo().getListaArestas()) {
24          desenhaAresta(gl,
25                        aresta.getVerticeUm(),
26                        aresta.getVerticeDois(),
27                        aresta.getCor());
28       }
29       if (getVerticeArestaAberta() != null &&
30           listenerMouse.getCoordenadaMouse() != null) {
31            desenhaArestaAberta(gl);
32       }
33  }
34  }
```

Quadro 21 - Desenho do modelo

Ao solicitar a execução de um algoritmo a classe `Interface` através de reflexão cria uma nova instância do item selecionado, após verifica se existe uma classe de informações do algoritmo, caso exista, cria também uma instância desta, e exibe as informações contidas nela para o usuário e por fim inicia a execução do algoritmo. Esta rotina é apresentada no Quadro 22.

```
    try {
01      ItemComboClasse item =
02                      (ItemComboClasse)algoritmosJCB.getSelectedItem();
03      Class c = Class.forName(item.getClasse().getName());
04      AlgoritmoInterface alg = (AlgoritmoInterface) c.newInstance();
05      ArrayList<VerticeAbstract> arrayVertice =
06                                  new ArrayList<VerticeAbstract>();
07      try {
08          c = Class.forName(c.getName() + "Dados");
09          if (c != null) {
10              DadosInterface dados = (DadosInterface) c.newInstance();
11              if (dados != null && dados.getVerticesMinimos() > 0) {
12                  VerticesParametros vp = new VerticesParametros(null,
13  true);
14                  vp.setVisible(renderizador.getModelo(),
15                                true,
16                                dados.getMensagemVertices(),
17                                dados.getVerticesMinimos());
18                  arrayVertice =  vp.getVerticesSelecionados();
19              }
20          }
21      } catch (Exception e) {
22          e.printStackTrace();
23      }
24      alg.executa(renderizador.getModelo(), arrayVertice);
25      executando = false;
26  } catch (Exception ex) {
27      ex.printStackTrace();
28  }
```

Quadro 22 - Rotina de execução do algoritmo

Ao solicitar que um grafo seja aberto ou salvo a classe `Interface` interagem com a classe `XML`, que utiliza as classes da biblioteca DOM para ler os arquivos criados, como apresentado no Quadro 23.

```
01  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
02  DocumentBuilder docBuilder = dbf.newDocumentBuilder();
03  Document doc = docBuilder.parse(file);
04
05  Element grafoTag = doc.getDocumentElement();
06  Element grafoTipo =
07
08  (Element)grafoTag.getElementsByTagName("graph").item(0);
09  if ("directed".equalsIgnoreCase(grafoTipo.getAttribute("edgedefault")))
10      grafo = new Digrafo();
11  else
12      grafo = new Grafo();
13
14  NodeList nodeList = grafoTag.getElementsByTagName("node");
15  int j = 0;
16  for (int i = 0; i < nodeList.getLength(); i++) {
17      Element nodeTag = (Element) nodeList.item(i);
18      VerticeAbstract v;
19      if (grafo instanceof Digrafo)
20          v = new Vertice();
21      else
22          v = new br.com.implementacao.grafo.Vertice();
23      v.setId(nodeTag.getAttribute("id"));
24
25      double angle = (float) (((double) j) / 57.29577957795135);
26      v.setX(1 + (200 * (float) Math.sin((double) angle)));
27      v.setY(1 + (200 * (float) Math.cos((double) angle)));
28      j = j + (360 / nodeList.getLength());
29
30      grafo.addVertice(v);
31      controleVertices.put((String) v.getId(), v);
32  }
33
34  NodeList edgeList = grafoTag.getElementsByTagName("edge");
35  for (int i = 0; i < nodeList.getLength(); i++) {
36      Element nodeTag = (Element) edgeList.item(i);
37      ArestaAbstract a;
38      String um = nodeTag.getAttribute("source");
39      String dois = nodeTag.getAttribute("target");
40      if (grafo instanceof Digrafo) {
41          a = new ArestaDirigida((Vertice) controleVertices.get(um),
42                  (Vertice) controleVertices.get(dois));
43      } else {
44          a = new Aresta((br.com.implementacao.grafo.Vertice)
45                  controleVertices.get(um),
46                  (br.com.implementacao.grafo.Vertice)
47                  controleVertices.get(dois));
48      }
49      grafo.addAresta(a);
50  }
```

Quadro 23 - Rotina para abrir um grafo salvo

3.3.3 Operacionalidade da implementação

Esta seção apresenta a operacionalidade do EVG através da execução de um estudo de

caso. Para demonstração da ferramenta será criado e executado um algoritmo simples de busca em largura. O apêndice B apresenta o algoritmo utilizado neste estudo de caso.

Ao iniciar o EVG é exibida ao usuário a tela apresentada na Figura 23. Esta tela apresenta três áreas distintas, sendo elas o menu superior, a opção de acesso ao compilador e a opção de acesso aos grafos.

Figura 23 - Tela inicial da ferramenta

Para que o usuário possa iniciar o desenvolvimento de algoritmos e a criação de grafos é necessário a criação de um novo projeto ou a abertura de um projeto existente. Todos os algoritmos criados estão sempre relacionados ao projeto aberto, não sendo possível utilizar algoritmos criados em outros projetos. A Figura 24 mostra o processo de criação de um novo projeto.

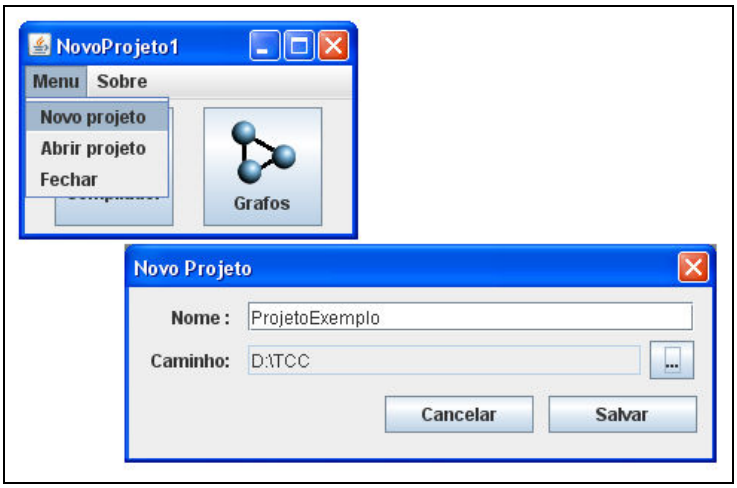


Figura 24 - Criação de um projeto

Após criado o projeto o usuário pode optar por criar novos algoritmos ou novos grafos. Ao optar por criar novos algoritmos a interface para o compilador é exibida(Figura 25). Esta interface possui as funcionalidades básicas para edição e compilação dos algoritmos.

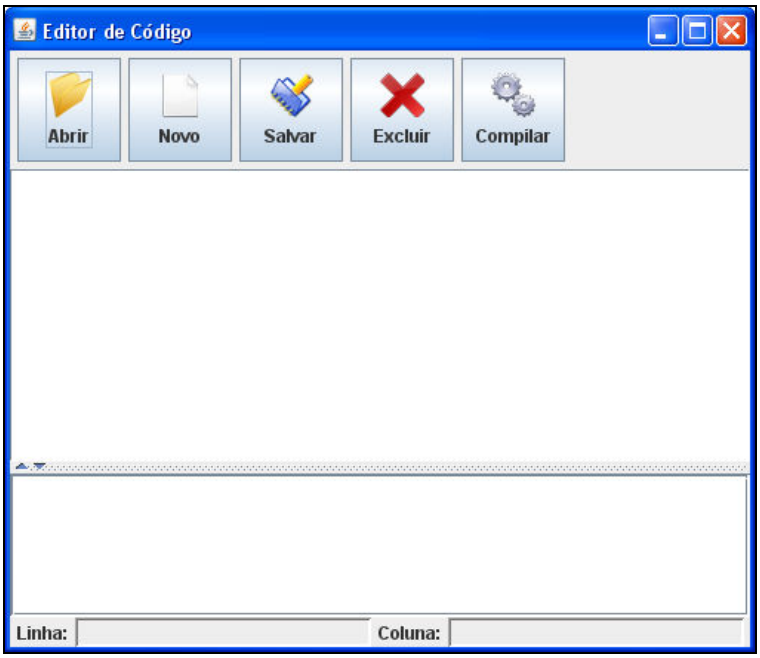


Figura 25 - Interface para criação e compilação de algoritmos

Ao optar por criar um novo arquivo uma nova janela em branco é aberta permitindo ao usuário digitar o algoritmo. Para que um algoritmo seja executado pela interface de grafos o mesmo deve implementar a classe `AlgoritmoInterface`. Caso o algoritmo necessite de vértices de entrada para sua execução uma nova classe com o mesmo nome da classe que implementa o algoritmo seguido da palavra Dados deve ser criada e deve implementar a classe `DadosInterface`.

Após o código fonte digitado e o arquivo salvo o usuário deve selecionar a opção de compilação para que os algoritmos estejam disponíveis para execução no grafo. Caso existam erros os mesmos serão apresentados, caso contrário uma mensagem de compilação finalizada será apresentada (Figura 26).

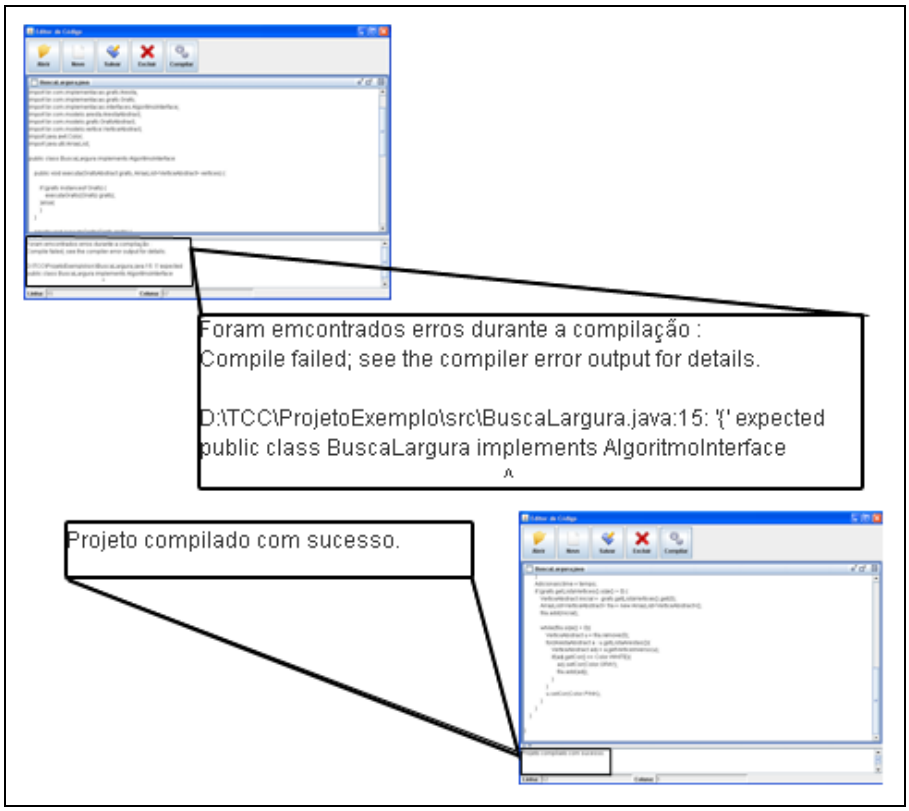


Figura 26 - Mensagens de compilação

Ao optar por criar novos grafos a tela exibida na Figura 27 é apresentada. A partir desta tela o usuário pode optar por criar grafos ou dígrafos, salvar, visualizar as principais propriedades do modelo criado, alterar os atributos de um vértice ou uma aresta e executar um algoritmo.

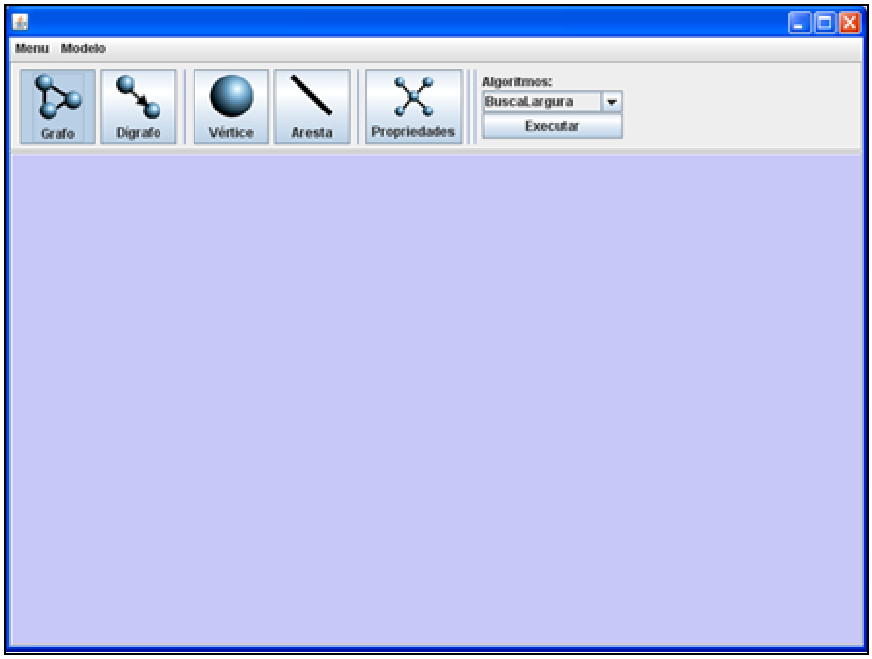


Figura 27 - Tela inicial do editor de grafo

A criação de um novo grafo pode ser feita de duas maneiras. A primeira é através da

seleção do botão que contém o item desejado, um vértice ou aresta, e a escolha de sua posição na tela, através de um clique do mouse no local desejado. A segunda é através da seleção do menu denominado Modelo seguido pela seleção do tipo de grafo, após a ferramenta solicita a quantidade de vértices desejados e gera um grafo aleatório. É possível também alterar a posição de um vértice na tela através do clique do botão direito do mouse sobre o vértice e na lista exibida a seleção da opção Mover, assim o vértice será movido junto com o mouse até que um clique de mouse seja realizado definindo a posição do vértice.

Para salvar um grafo são disponibilizadas duas extensões: Graphml e XML. O formato XML guarda todos os atributos dos vértices e arestas, já o formato Graphml é um formato bastante utilizado por aplicações que trabalham com grafos. Um trecho dos arquivos gerados são mostrados nos Quadro 24 e Quadro 25. A opção para salvar um grafo esta disponível através da seleção da opção salvar no menu denominado Menu. Da mesma forma para abrir um grafo salvo deve-se acessar a opção abrir do menu.

```
<grafo>
  <digrafo>false</digrafo>
  <vertice>
    <controle>0</controle>
    <id class="int">6</id>
    <cor>
      <red>0</red>
      <green>255</green>
      <blue>0</blue>
      <alpha>255</alpha>
    </cor>
    <x>-141.1167544232408</x>
    <y>129.85054993428045</y>
  </vertice>
  <vertice>
    <controle>1</controle>
    <id class="int">7</id>
    <cor>
      <red>0</red>
      <green>255</green>
      <blue>0</blue>
      <alpha>255</alpha>
    </cor>
    <x>118.78172854330343</x>
    <y>59.53846176025961</y>
  </vertice>
  <aresta>
    <verticeUm>0</verticeUm>
    <verticeDois>1</verticeDois>
    <id class="int">1</id>
    <cor>
      <red>0</red>
      <green>255</green>
      <blue>0</blue>
      <alpha>255</alpha>
    </cor>
  </aresta>
</grafo>
```

Quadro 24 - Grafo salvo no formato XML

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
 http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<graph id="G" edgedefault="undirected">
    <node id="1"/>
    <node id="2"/>
    <node id="3"/>
    <edge source="1" target="3"/>
    <edge source="2" target="3"/>
    <edge source="3" target="1"/>
  </graph>
</graphml>
```

Quadro 25 - Grafo salvo no formato Grapml

A visualização das principais propriedades do modelo criado é feita acessando a opção propriedades (Figura 28).

Figura 28 - Grafo e dígrafo criados na ferramenta

Para visualizar e alterar os atributos de um vértice ou uma aresta deve-se clicar com o botão direito do mouse sobre o vértice ou aresta desejada e então selecionar a opção Propriedades assim uma tela é exibida permitindo a edição (Figura 29). Para a exclusão de um vértice ou aresta deve-se selecionar a opção Excluir do menu exibido através do uso do botão direito do mouse.

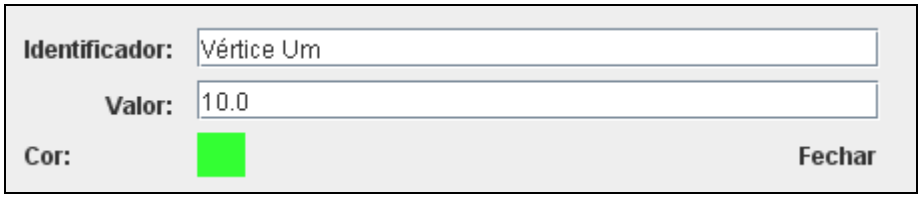


Figura 29 - Edição das propriedades

Para visualização do algoritmo criado, o mesmo deve ser selecionado na lista de algoritmos disponíveis. Caso o algoritmo necessite de vértices como parâmetro para sua execução uma tela solicitando os mesmos é exibida (Figura 30). Nesta tela o usuário deve selecionar cada vértice que será utilizado pelo algoritmo clicando sobre a descrição do mesmo

e então sobre o botão selecionar. A seleção dos vértices utilizados pode também ser feita mantendo a tecla `Ctrl` pressionada e clicando sobre a descrição dos vértices que serão utilizados e por fim sobre o botão selecionar. Os vértices são passados para o algoritmo na mesma ordem que são selecionados. Um exemplo da execução do algoritmo é apresentado na Figura 31. A execução do algoritmo ocorre através da coloração do vértice feita via programação no algoritmo criado pelo próprio usuário. Por fim, a ferramenta disponibiliza uma janela para saída do algoritmo em forma de texto (Figura 32). Esta tela recebe os dados que o usuário deseja imprimir durante a execução do algoritmo ou ao final do mesmo.

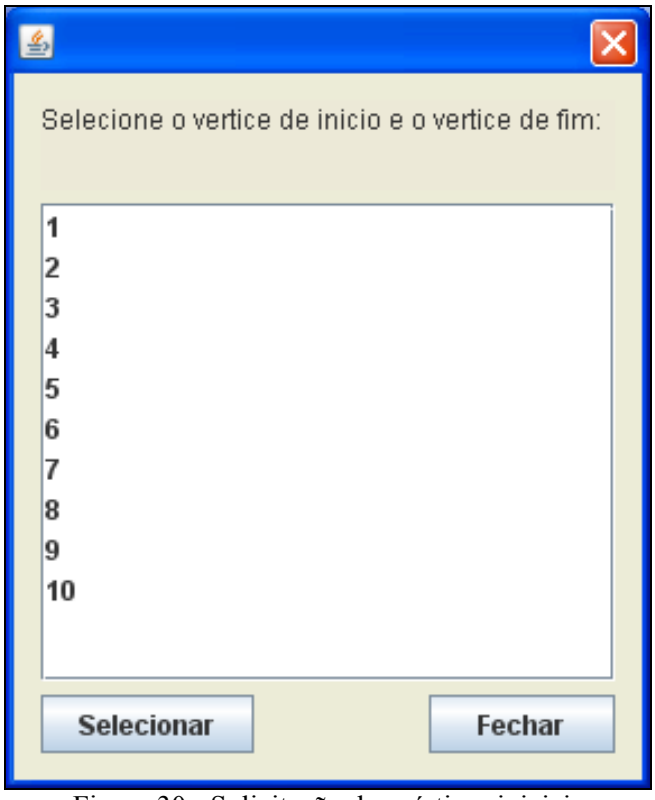


Figura 30 - Solicitação dos vértices iniciais

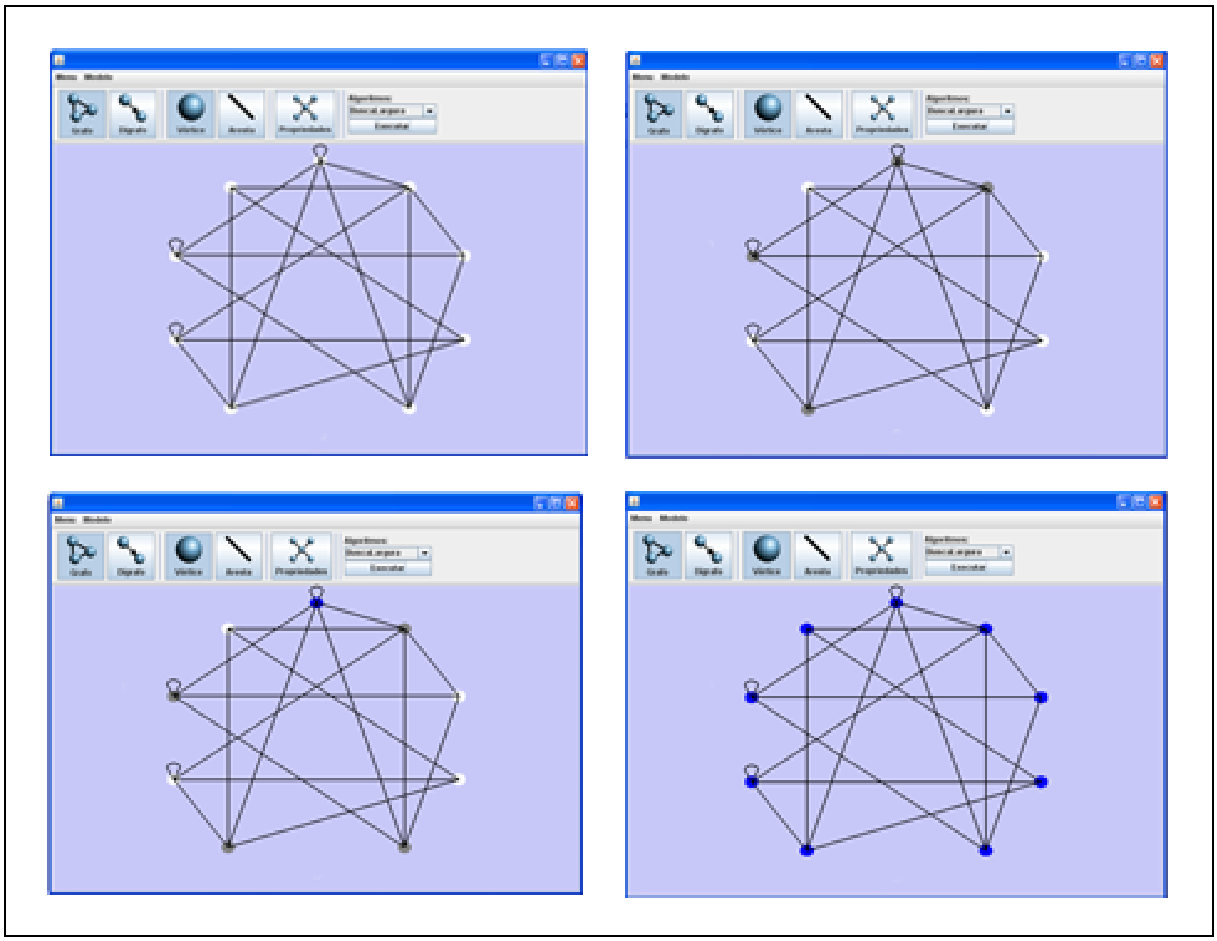


Figura 31 - Execução do algoritmo

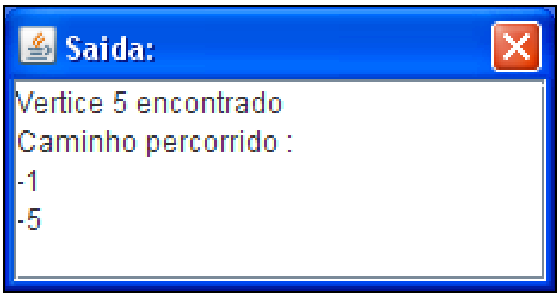


Figura 32 - Saída do algoritmo

3.4 RESULTADOS E DISCUSSÃO

Os resultados obtidos ao final deste trabalho foram satisfatórios. Os objetivos do trabalho foram alcançados e o desenvolvimento do mesmo permitiu aumentar o conhecimento em relação às técnicas e ferramentas utilizadas.

No Quadro 26 é apresentado um comparativo entre os trabalhos correlatos e o EVG.

	HACKBART 2008	VILLALOBOS 2006	SANGIORGI 2006	EVG 2009
Criação visual de grafos	Sim	Sim	Sim	Sim
Algoritmos disponíveis p/ execução	Sim	Sim	Não	Não
Criação de novos algoritmos	Não	Não	Sim	Sim
Visualizar propriedades do grafo	Não	Sim	Sim	Sim
Permite salvar o grafo criado	Não	Sim	Não	Sim
Multiplataforma	Não	Não	Sim	Sim
Coloração de vértices e arestas	Não	Sim	Não	Sim

Quadro 26 - Comparativo das ferramentas correlatas

Em relação aos trabalhos de Hackbarth (2008) e Villalobos (2006) o EVG destaca-se por permitir que o usuário desenvolva seu próprio algoritmo não ficando limitado aos algoritmos disponibilizados pela aplicação. Outro fato importante em relação ao trabalho de Hackbarth (2008) e também de Sangiorgi (2006) é a possibilidade de salvar os grafos criados no formato XML e GRAPHML, sendo que o primeiro permite que o grafo possa ser utilizado posteriormente sem perder nenhum atributo, além de ser de fácil leitura para outras aplicações que venham a utilizá-lo e o segundo é um padrão utilizado por diversas aplicações. Ainda em relação aos trabalhos de Hackbarth (2008) e Sangiorgi (2006), o EVG disponibiliza funcionalidades para coloração de vértices e arestas, pontos importantes na resolução de determinados problemas e não disponíveis nos trabalhos citados. Por fim, a ferramenta destaca-se também por ser desenvolvida em Java, podendo assim ser executada em diversos sistemas operacionais. Assim, pode-se afirmar que o EVG engloba as principais funcionalidades para o estudo de grafos em uma única ferramenta.

4 CONCLUSÕES

Os resultados obtidos com o desenvolvimento deste trabalho foram satisfatórios. Os requisitos propostos foram todos cumpridos, alcançando assim o objetivo principal do trabalho, o desenvolvimento de uma ferramenta para criação e execução de algoritmos em grafos.

O EVG permite criar grafos e dígrafos, podendo alterar suas propriedades, a disposição dos vértices, salvar os modelos criados, além de criar, editar e executar algoritmos. Dentre as principais vantagens do EVG está a possibilidade de auxílio no ensino de grafos, visto que a mesma permite a visualização dos algoritmos sendo executados sobre o grafo, permitindo uma melhor compreensão dos mesmos. Outra vantagem do EVG é a possibilidade de uso dos algoritmos em outras aplicações visto que a estrutura utilizada para armazenamento dos grafos não esta vinculada somente a aplicação desenvolvida, bastando apenas importar a biblioteca de grafos desenvolvida para o projeto que irá utilizá-la.

No entanto existem limitações. É somente possível a criação de um grafo por vez, o que não permite ao EVG executar algoritmos como a verificação de grafos isomorfos.

O uso das bibliotecas ANT e JOGL foi de grande importância durante o desenvolvimento do EVG. O JOGL disponibiliza rotinas que permitem maior facilidade no desenho da parte gráfica, enquanto que o ANT se responsabiliza por grande parte do processo compilação. O uso destas bibliotecas facilitou o desenvolvimento de algumas rotinas, permitindo maior dedicação ao desenvolvimento da estrutura dos grafos e algoritmos.

Uma dificuldade encontrada durante o desenvolvimento do EVG foi a forma de carregar os algoritmos criados para memória do computador. Inicialmente ao serem compilados pela ferramenta, os algoritmos já eram carregados, porém verificou-se que após a segunda compilação não era possível carregar os mesmos algoritmos para memória e nem os remover, limitação da linguagem Java, que segundo informações no site oficial, não há uma previsão certa para correção. Assim, foi necessário, a cada solicitação da abertura da interface gráfica, abrir a mesma como uma nova aplicação Java, que assim passa a ocupar outro espaço de memória, ou seja, a cada abertura um novo espaço de memória é utilizado podendo os algoritmos alterados serem novamente carregados.

4.1 EXTENSÕES

Como extensão para este trabalho sugere-se a possibilidade de criação de mais de um grafo para que algoritmos que trabalhem com mais modelos de grafos, como a verificação de isomorfismo, possam ser criados.

Sugere-se também a implementação de um recurso que permita ao usuário pausar a execução do algoritmo e a implementação de um recurso que permita ao usuário retroceder na execução do algoritmo.

Por fim, sugere-se a possibilidade de execução passo a passo de forma que o usuário possa definir quando o próximo passo será executado, bem como visualizar no código fonte a linha que esta sendo executada.

REFERÊNCIAS BIBLIOGRÁFICAS

APACHE ANT. **Apache Ant 1.7.0**: manual. [S.l.], 2008. Documento eletrônico disponibilizado com o Ambiente Apache Ant 1.7.0.

BELL, AlexisT. **Ant Java notes**: an accelerated intro guide to the Java Ant Build Tool. [S.l.]: Virtualbookworm.com Publishing, 2005.

COHEN, Marcelo; MANSSOUR, Isabel H. **OpenGL**: uma abordagem prática e intuitiva. São Paulo: Novatec, 2006.

DAVISON, Andrew. **Pro Java 6 3D game development**: Java 3D, JOGL, Jinput and Joal apis. [S.l.], 2007.

DIESTEL, Reinhard. **Graph theory**. 3th ed. New York: Springer-Verlag, 2005.

GRAPHML TEAM. **The GraphML file format**. [S.l.]: 2007. Disponível em: <http://graphml.graphdrawing.org/>. Acesso em: 04 out. 2009.

GOMES, Paulo C. R. **Grafos**: conceitos, algoritmos e aplicações. Blumenau, 2007. Trabalho não publicado.

GROSS, Jonathan; YELLEN, Jay. **Graph theory and its applications.** 2th ed. New York: CRC Press, 2006.

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

LIANG, Sheng. **The Java™ NativeInterface**: programmer's guide and specification. [S.l.]: Addison-Wesley, 1999. Disponível em: <http://java.sun.com/docs/books/jni/download/jni.pdf>. Acesso em: 13 set. 2009.

LOPES, Arthur V. **Estruturas de dados para construção de softwares**. 1. ed. Canoas: ULBRA, 1999.

MANSSOUR, Isabel H. **Introdução à OpenGL**. [S.l.], 2005. Disponível em: <http://www.inf.pucrs.br/~manssour/OpenGL/Tutorial.html>. Acesso em: 13 set. 2009.

NIEMEYER, Patrick; KNUDSEN, Jonathan. **Learning Java**. 3th ed. Beijing: O'Reilly, 2005.

PEREZ, Anderson L. F.; ZANCANELA, Luiz C. Reflexão computacional: a nova dimensão da programação orientada a objetos. In: ESCOLA REGIONAL DE INFORMÁTICA, 11., 2003, Lages. **Anais…** Lages; SBC, 2003. p. 99-117. Disponível em: <http://inf.uniplac.net/eventos/eri2003/anais/capitulo_4.pdf >. Acesso em: 16 mar. 2009

REZENDE, Oscar L. T. **A teoria dos grafos na solução de "quebra-cabeças"**. Revista Capixaba da Ciência e Tecnologia, Vitória, n. 2, p. 50-53, 1. sem. 2006. Disponível em <http://recitec.cefetes.br/artigo/documentos/Artigo%208.pdf >. Acesso em: 13 set. 2009.

SANGIORGI, Ugo B. **RoxGT**: um framework de código aberto para o ensino, modelagem e análise de grafos. 2006. 53 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Faculdade Ruy Barbosa, Salvador. Disponível em: <http://www.roxgt.org/documentacao_old/Monografia-UgoBragaSangiorgi.pdf >. Acesso em: 13 mar. 2009.

SENA, Demóstenes S. **Agraphs**: definição, implementação e suas ferramentas. [S.l.], 2006. Disponível em: <http://www.ppgsc.ufrn.br/html/Producao/Dissertacoes/DemostenesSantosDeSena.pdf>. Acesso em: 02 nov. 2009.

SENRA, Rodrigo D. A. **Programação reflexiva sobre o protocolo de meta-objetos guaraná**. 2001. 164f. Dissertação (Mestrado em Ciências da Computação) – Curso de Pós-Graduação em Ciência da Computação, Universidade Estadual de Campinas, Campinas.

SOUZA, Talita B.; VICENTE, Amarildo de. **Grafos e a localização do centro de emergência**. [S.l.], 2008. Disponível em: <http://projetos.unioeste.br/cursos/cascavel/matematica/xxisam/PDFs/09.pdf>. Acesso em: 12 set. 2009.

SUN MICROSYSTEM. Package org.w3c.dom. In: SUN MICROSYSTEM. Java Platform, standard edition 6: API specification. [S.l.], 2008. Dispoinível em: <http://java.sun.com/javase/6/docs/api/org/w3c/dom/package-summary.html#package_description>. Acesso em: 04 out. 2009.

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: 14 mar. 2009.

WORLD WIDE WEB CONSORTIUM. **Document Object Model (DOM)**. [S.l.], 2005. Disponível em: <http://www.w3.org/DOM/#what >. Acesso em: 04 out. 2009.

APÊNDICE A – Detalhamento das classes do sistema

Na Figura 33, Figura 34, Figura 35, Figura 36 e Figura 37 são detalhadas as classes do sistema. Para melhor visualização foram ocultados os relacionamentos entre as classes. Os relacionamentos são demonstrados na Figura 18, na Figura 19, na Figura 20 e na Figura 21.

A Figura 33 exibe as classes da estrutura do grafo.

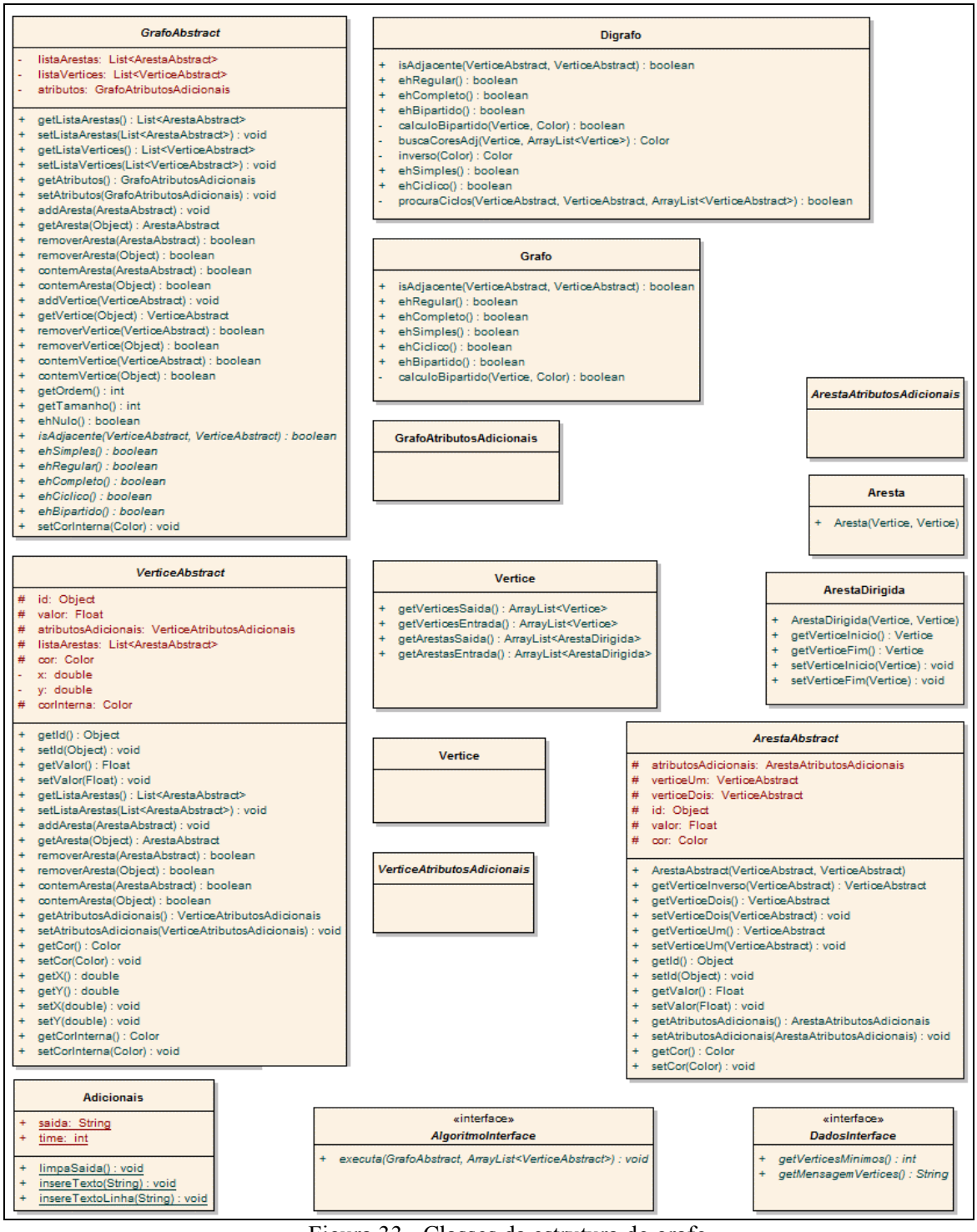


Figura 33 - Classes da estrutura do grafo

A Figura 34 exibe as classes da interface inicial do sistema.

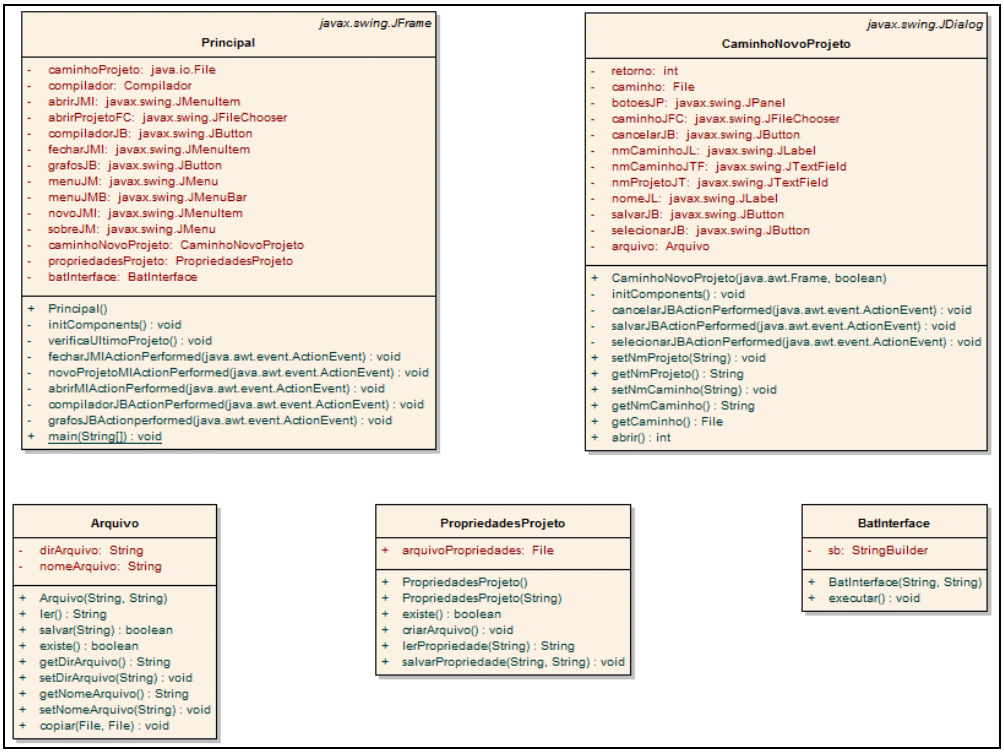


Figura 34 - Classes da interface inicial do sistema

A Figura 35 exibe as classes da interface de compilação do sistema.

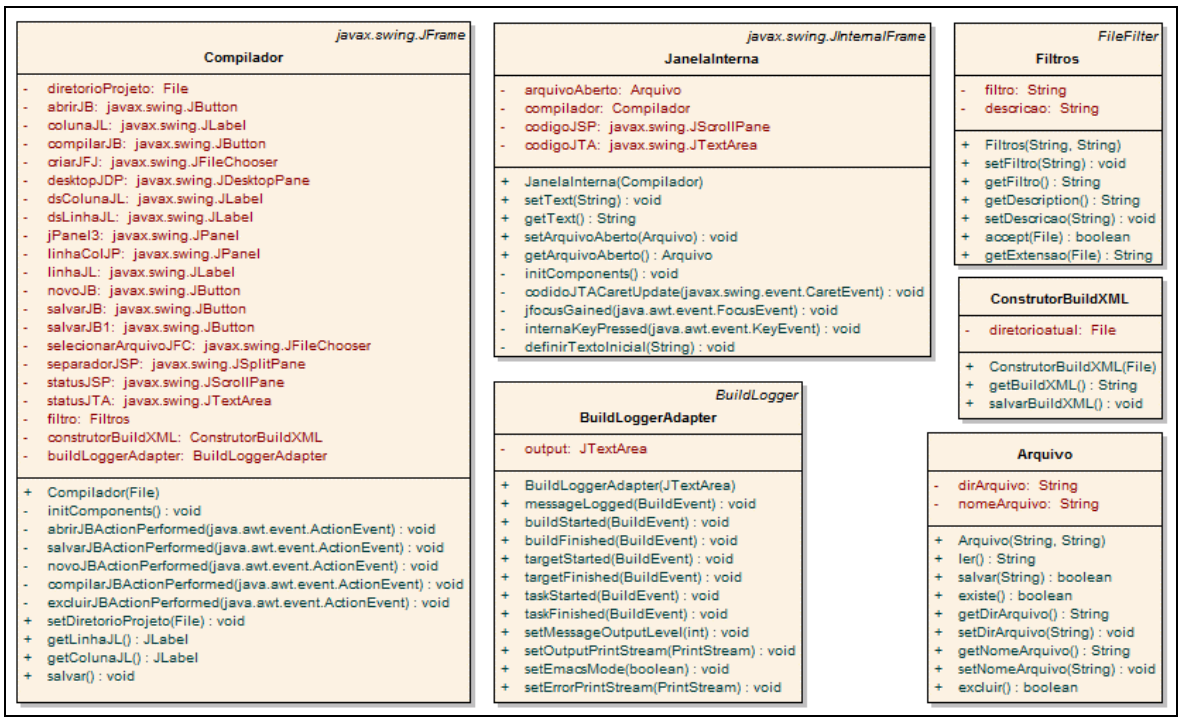


Figura 35 - Classes da interface de compilação do sistema

A Figura 36 e a Figura 37 exibem as classes da interface de criação de grafos.

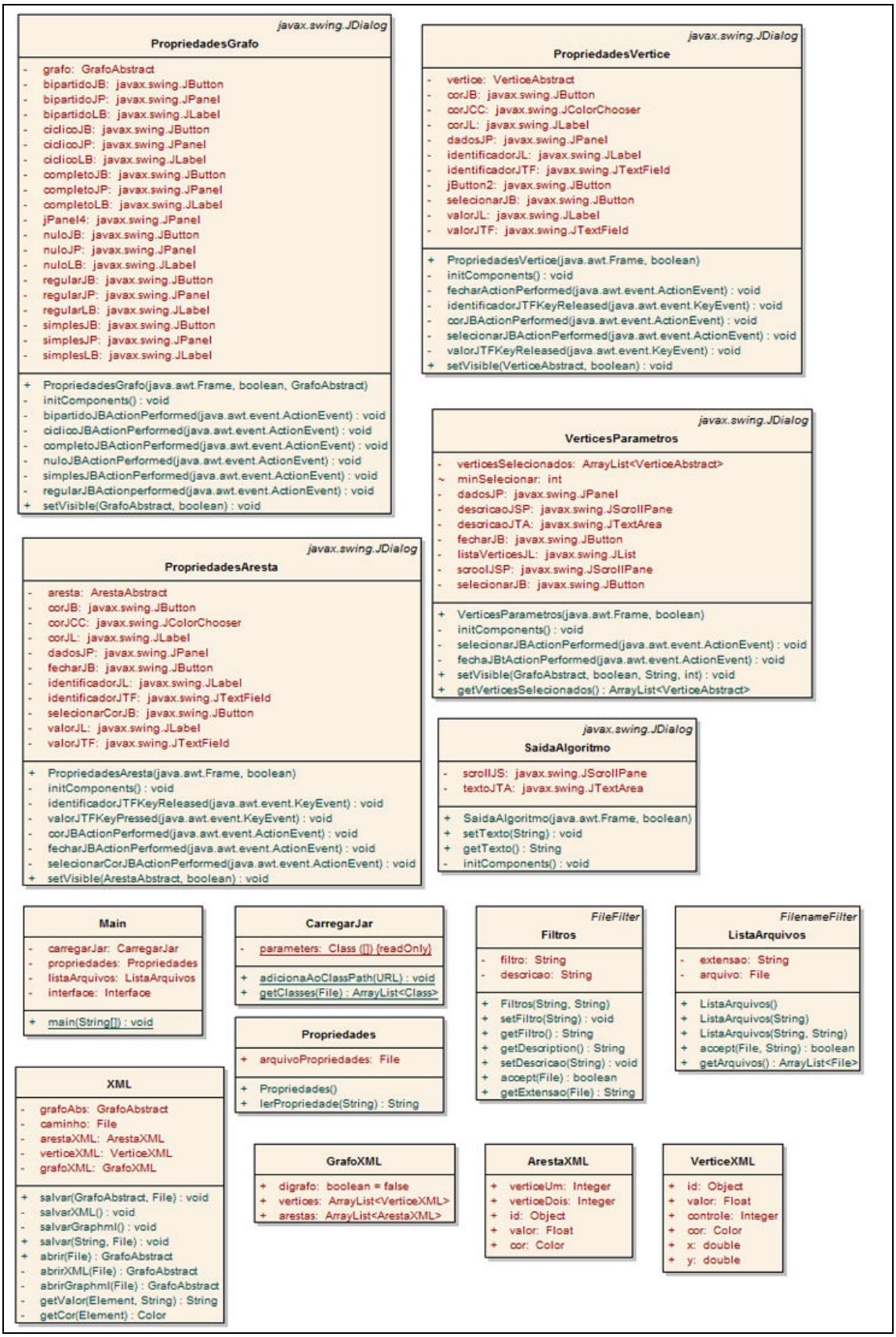


Figura 36 - Classes da interface de criação de grafos

Figura 37 - Classes da interface de criação de grafos

APÊNDICE B – Algoritmos utilizados na operacionalidade da ferramenta

No Quadro 27 é apresentado o código fonte da classe `BuscaLarguraDados`.

```
package algoritmos;

import br.com.implementacao.interfaces.DadosInterface;
import br.com.modelo.grafo.GrafoAbstract;
import br.com.modelo.vertice.VerticeAbstract;

public class BuscaLarguraDados implements DadosInterface {

    public int getVerticesMinimos(){
      return 2;
    }

    public String getMensagemVertices(){
      return "Selecione o vértice de início e o vértice de fim:";
   }
}
```

Quadro 27 - Código fonte da classe `BuscaLarguraDados`

No Quadro 28 é apresentado o código fonte da classe `BuscaLarguraVerticesAdicionais`.

```
package algoritmos;

import br.com.modelo.grafo.GrafoAbstract;
import br.com.modelo.vertice.VerticeAbstract;
import br.com.modelo.vertice.VerticeAtributosAdicionais;

public class BuscaLarguraVerticesAdicionais extends
VerticeAtributosAdicionais {

    private VerticeAbstract antecessor;
    public BuscaLarguraVerticesAdicionais(VerticeAbstract antecessor){
         this.antecessor = antecessor;
    }

    public VerticeAbstract getAntecessor(){
       return antecessor;
    }
}
```

Quadro 28 - Código fonte da classe `BuscaLarguraVerticesAdicionais`

No Quadro 29 é apresentado o código fonte da classe `BuscaLargura`.

```
package algoritmos;
import br.com.adicionais.Adicionais;
import br.com.implementacao.*;
import br.com.modelo.*;
import java.awt.Color;
import java.util.ArrayList;

public class BuscaLargura implements AlgoritmoInterface {

    public void executa(GrafoAbstract grafo,
                        ArrayList<VerticeAbstract> vertices) {
        if (grafo instanceof Grafo)
            executaGrafo((Grafo) grafo, vertices.get(0), vertices.get(1));
    }

    private void executaGrafo(Grafo grafo, VerticeAbstract inicio,
                                           VerticeAbstract fim ) {
        for (VerticeAbstract v : grafo.getListaVertices()) {
            v.setCor(Color.WHITE);
        }
        if (grafo.getListaVertices().size() > 0) {
            ArrayList<VerticeAbstract> fila =
                                          new ArrayList<VerticeAbstract>();
            inicio.setCor(Color.GRAY);
            fila.add(inicio);

            while(fila.size() > 0){
                VerticeAbstract u = fila.remove(0);
                for(ArestaAbstract a : u.getListaArestas()){
                    VerticeAbstract adj = a.getVerticeInverso(u);
                    if(adj == fim){
                        Adicionais.insereTexto("Vertice " +fim.getId()+
                                    " encontrado \nCaminho percorrido :");
      adj.setAtributosAdicionais(new BuscaLarguraVerticesAdicionais(u));
      adj.setCor(Color.GREEN);
      String caminhoStr = "";
      VerticeAbstract caminho = adj;
      while(caminho != null){
            caminhoStr = "\n-"+caminho.getId()+caminhoStr;
            if(caminho.getAtributosAdicionais() != null)
                  caminho = ((BuscaLarguraVerticesAdicionais)caminho.
                              getAtributosAdicionais()).getAntecessor();
            else
                  caminho  = null;
      }
      Adicionais.insereTexto(caminhoStr);
                        return;
                      }else if(adj.getCor() == Color.WHITE){
      adj.setAtributosAdicionais(new BuscaLarguraVerticesAdicionais(u));
                             adj.setCor(Color.GRAY);
                        fila.add(adj);
                    }
                }
                u.setCor(Color.BLUE);
            }
        }
        Adicionais.insereTexto("A busca a partir do vértice "+
                  inicio.getId()+  "não encontrou o vértice "+fim.getId());
      }
}
```

Quadro 29 - Código fonte da classe `BuscaLargura`