UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

UM FRAMEWORK PARA ALGORITMOS BASEADOS NA TEORIA DOS GRAFOS

MAICON RAFAEL ZATELLI

BLUMENAU
2010

2010/2-20

MAICON RAFAEL ZATELLI

UM FRAMEWORK PARA ALGORITMOS BASEADOS NA 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
2010

2010/2-20

UM FRAMEWORK PARA ALGORITMOS BASEADOS NA TEORIA DOS GRAFOS

Por

MAICON RAFAEL ZATELLI

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. Antônio Carlos Tavares, Esp. – FURB
Membro:	Prof. Roberto Heinzle, Ms. – FURB

Blumenau, 08 de dezembro de 2010

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, pelo apoio de sempre.

À minha irmã, Gabriele Andressa Zatelli, pela ajuda para resolver alguns problemas com formatações.

Ao meu orientador, Paulo César Rodacki Gomes, por ter auxiliado na conclusão deste trabalho.

Ao professor José Roque Voltolini da Silva, pelo esclarecimento de diversas dúvidas durante o trabalho.

A coisa mais bela que podemos experimentar é o mistério. Essa é a fonte de toda a arte e ciências verdadeiras.

Albert Einstein

RESUMO

Este trabalho apresenta o desenvolvimento do FGA, um *framework* de algoritmos baseados na teoria dos grafos implementado na linguagem Java. O FGA disponibiliza uma série de verificações de propriedades de grafos, bem como um subconjunto de funções capazes de gerar grafos com base em certas características tais como grafos completos, bipartidos, regulares entre outros. Além disso, é disponibilizado um conjunto de classes para a execução e manipulação de resultados obtidos com algoritmos clássicos da teoria dos grafos. Tais algoritmos são úteis para modelar e resolver diversos problemas práticos e teóricos. Por fim, uma aplicação de exemplo é construída aproveitando todos os recursos oferecidos pelo FGA. Como item adicional do trabalho, o *framework* Java também foi portado para linguagem Objective-C.

Palavras-chave: Algoritmos. Estuturas de dados. *Framework*. Teoria dos grafos.

ABSTRACT

This work presents the development of FGA, a Java framework for algorithms based on graph theory concepts. The FGA provides features for inspections of graph properties and a set of functions for automatic graph construction based on certain characteristics such as complete graphs, bipartite graphs, regular graphs and others. The FGA also provides several classes for execution of classic graph theory algorithms which can be used to model and solve both practical and theoretical problems. Finally, an example application is developed using all the resources offered by the FGA. As an additional feature, the framework also has been ported from Java to Objective-C.

Key-words: Algorithms. Data structures. Framework. Graph theory.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

BFS – *Breadth First Search*

CSV – *Comma-separated Values*

DFS – *Depth First Search*

DOM – *Document Object Model*

FGA – *Framework for Graph Algorithms*

GB – *GBytes*

GHz – *GigaHertz*

HTML – *HiperText Markup Language*

RGB – *Red Green Blue*

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

W3C – *World Wide Web Consortium*

SUMÁRIO

1 INTRODUÇÃO

Grafo é um tipo de estrutura de dados utilizada para representar relacionamentos entre pares de objetos. Basicamente é formado por um conjunto de vértices e uma coleção de conexões entre pares de vértices, chamadas de arestas (GOODRICH; TAMASSIA, 2007, p. 508). Ao longo do tempo a teoria dos grafos evoluiu muito e diversos problemas práticos e teóricos podem ser reduzidos a um modelo de grafos, que, com o auxílio de teoremas e algoritmos, podem levar a conclusões e soluções acerca destas questões. Segundo Skiena e Revilla (2003, p. 237), a chave para solucionar problemas consiste em identificar o grafo fundamental oculto àquela situação e utilizar algoritmos clássicos para resolver a questão resultante.

Porém, há uma enorme gama de propriedades e algoritmos de grafos a serem estudados e implementados. Muitas implementações são simples, outras são mais complexas e podem demandar horas de pesquisa, especificação e desenvolvimento. Em termos de desenvolvimento de aplicações isso pode desviar o objetivo do desenvolvedor, que não é de realizar tais implementações, mas sim, por meio de um conhecimento prévio, apenas utilizar chamadas de funções de bibliotecas já prontas. Sendo assim, alguém que eventualmente sabe a aplicabilidade de determinado algoritmo envolvendo grafos pode não querer implementá-lo, mas sim apenas modelar o grafo, chamar a execução do algoritmo e trabalhar com a resposta fornecida. Hoje isto é possível graças a orientação a objetos, que tem como uma de suas características a reusabilidade de código, permitindo a integração de classes desenvolvidas por terceiros em uma aplicação.

Nesse momento aparece um outro desafio: a criação de casos de teste utilizando instâncias de grafos contendo muitos vértices e arestas, ou um tipo específico com base em algumas restrições. Muitas vezes é necessário gerar tais grafos a fim de testar mais profundamente a aplicação e avaliar seu desempenho e bom funcionamento.

Diante da falta de uma ferramenta que contemple todos estes requisitos, surge a ideia de desenvolver um *framework*[1] de grafos, que permita gerar grafos mediante critérios, editar os existentes e que seja capaz de extrair diversas propriedades importantes e executar algoritmos clássicos. Além disso, faz-se necessário ter uma boa documentação, bem como uma aplicação exemplo para demonstrar a utilização dos recursos oferecidos pelo *framework*.

[1] Sauvé (2007) define *framework* como sendo um conjunto de classes e interfaces que provê soluções para uma família de problemas semelhantes.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é construir um *framework* para auxiliar no desenvolvimento de softwares baseados em teoria dos grafos.

Os objetivos específicos do trabalho são:

a) disponibilizar um conjunto de algoritmos clássicos de grafos, entre os quais citam-se: Dijkstra, Floyd-Warshall, Bellman-Ford, Prim, Kruskal, Ford-Fulkerson, Hopcroft-Tarjan, ordenação topológica, busca em largura e busca em profundidade;
b) permitir extrair algumas importantes propriedades do grafo, tais como: completo, regular, trivial, nulo, bipartido, bipartido completo, conexo ou desconexo, fortemente conexo, denso ou esparso e simples ou multigrafo;
c) permitir gerar instâncias de grafos com base em restrições, como grafos completos com `n` vértices, grafos regulares de `n` vértices e grau `k`, grafos densos, esparsos, conexos ou desconexos e simples ou multigrafos;
d) persistir os grafos;
e) documentar o *framework*;
f) disponibilizar uma aplicação exemplo para demonstrar a utilização das funções do *framework*.

1.2 ESTRUTURA DO TRABALHO

O trabalho está estruturado em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica necessária para o entendimento do trabalho. Nele são expostos os principais conceitos relacionados a teoria dos grafos, as propriedades de grafos e também os algoritmos clássicos utilizados na teoria dos grafos para resolver diversos problemas práticos e teóricos. Além disso, neste capítulo, na seção de trabalhos correlatos, são mencionadas ferramentas existentes no mercado e desenvolvidas em trabalhos de conclusão de curso com funcionalidades semelhantes às do *framework*. No terceiro capítulo é mostrado o desenvolvimento do sistema. Nesse capítulo pode ser encontrado os requisitos e especificação do trabalho, como também a descrição dos algoritmos implementados e os resultados obtidos.

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

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são explanados os principais conceitos relacionados à estrutura básica de um grafo. Na sequência são expostas as principais propriedades de grafos, com exemplos em figuras para melhor compreensão. Logo em seguida são detalhados os principais algoritmos clássicos de grafos, bem como suas aplicações. Por fim, em trabalhos correlatos, são mencionadas ferramentas existentes no mercado e desenvolvidas em trabalhos de conclusão de curso com funcionalidades semelhantes às do FGA.

2.1 CONCEITOS BÁSICOS DA TEORIA DOS GRAFOS

Um grafo pode ser definido como um par ordenado $G = (V, E)$, sendo V um conjunto finito e não vazio e E uma relação binária sobre V. Os elementos que pertencem a V são chamados de vértices ou nós, enquanto que os pares não-ordenados de E são denominados de arestas ou arcos (RABUSKE, 1992, p. 5).

Um grafo dirigido $G = (V, E)$ é definido por um conjunto não vazio V de vértices e um conjunto E de arestas, onde cada aresta $e \in E$ é definida por um par ordenado de vértices (u, v), sendo ambos u e $v \in V$. O vértice u é o vértice origem, e v é o vértice destino da aresta e (RABUSKE, 1992, p. 7). Um exemplo de grafo dirigido e outro de um grafo não dirigido podem ser vistos na Figura 1.

A representação gráfica padrão de um grafo é um desenho formado por pontos, que representam os vértices, e por retas ou arcos conectando os pontos, que são as arestas (GROSS; YELLEN, 2006, p. 2).

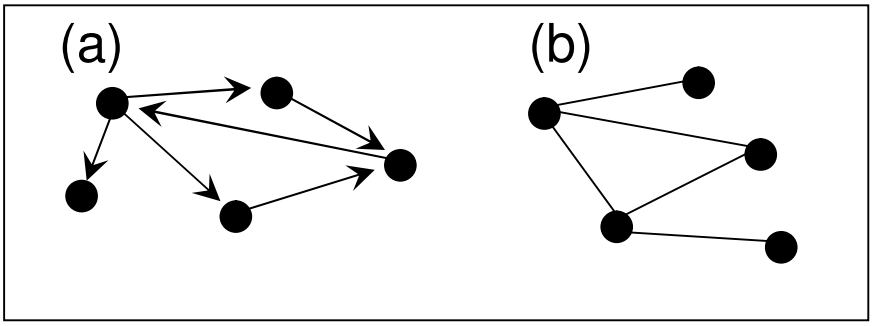
Vértices que pertencem a uma mesma aresta são denominados vizinhos ou adjacentes. Uma aresta que conecta somente um vértice é chamada de laço e duas arestas diferentes que possuem os mesmos vértices, tanto de origem como de destino, são chamadas de paralelas (FEOFILOFF; KOHAYAKAWA; WAKABAYASHI, 2009).

Uma aresta também pode possuir atributos. Um atributo pode ser um valor numérico que, para Rabuske (1992, p. 12) é chamado de peso, distância ou custo. Tais atributos representam alguma quantidade física em problemas envolvendo transportes, redes de

transmissão de dados, entre outros.

Outra definição importante é o grau de um vértice, que segundo Feofiloff, Kohayakawa e Wakabayashi (2009), é o número de arestas que incidem em um vértice. Scheinerman (2003, p. 423) classifica um vértice como isolado se o seu grau é igual a zero. Para Rabuske (1992, p. 10), um vértice pode ser chamado de pendente se o seu grau for um.

Em grafos dirigidos o grau de um vértice pode ser classificado em grau de saída, que representa a quantidade de arestas que partem de um vértice, e grau de entrada, que é a quantidade de arestas que atingem o vértice (RABUSKE, 1992, p. 32).

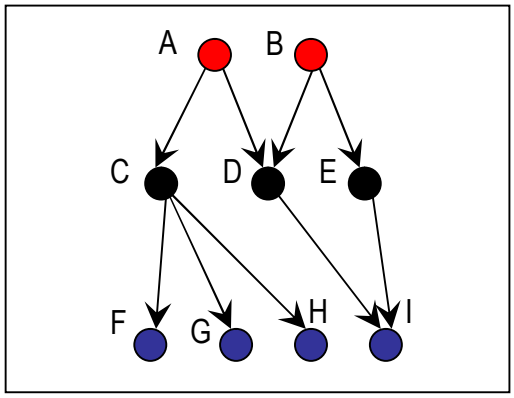


Fonte: adaptado de Gross e Yellen (2006, p. 2).
Figura 1 – Exemplo de um grafo dirigido (a) e um grafo não dirigido (b)

Um caminho é uma sequência alternada de vértices e arestas sendo que o vértice final de uma aresta é o vértice inicial da aresta seguinte no caminho. Um caminho inicia em um vértice e pode terminar tanto no mesmo vértice de partida como em outro qualquer. Quando os vértices de início e fim são os mesmos o caminho é chamado de ciclo ou de caminho fechado. Um grafo que não possui ciclo e é conexo, pode ser chamado de árvore e um conjunto de árvores é uma floresta (GOODRICH; TAMASSIA, 2004, p. 296).

Outra informação importante a respeito de grafos dirigidos é a base e a antibase (Figura 2). Base é o conjunto de vértices que não podem ser alcançados partindo-se de qualquer outro vértice do grafo, ou seja, os vértices de grau de entrada igual a zero. Já, a antibase é o conjunto de vértices que não alcançam nenhum outro vértice, isto é, os vértices de grau de saída igual a zero (RABUSKE, 1992, p. 32).

Estes dados têm utilidade em problemas organizacionais, onde é possível encontrar, por exemplo, as pessoas que teriam alguma autoridade sobre outras pessoas, e também encontrar as pessoas que estão submetidas a outras, conforme ilustrado na Figura 2 (RABUSKE, 1992, p. 33).

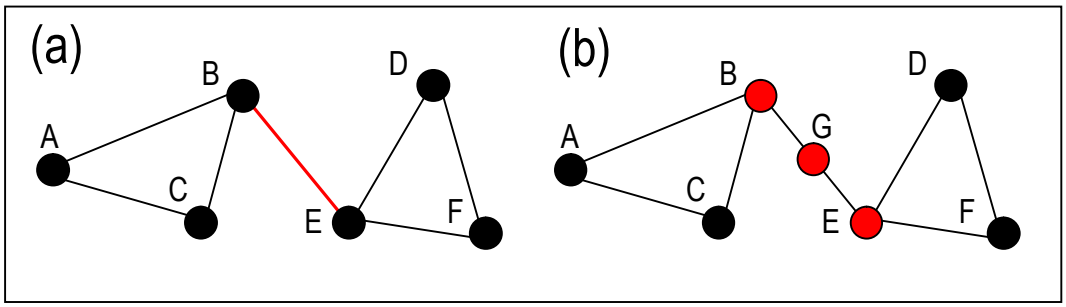


Fonte: adaptado da definição de Rabuske (1992, p. 32).

Figura 2 – Exemplo de base (A e B) e antibase (F, G, H e I)

Pontes e nós de articulação (Figura 3) também fornecem outras propriedades interessantes. Uma ponte é uma aresta, que, no caso de ser removida torna o grafo desconexo, e, por sua vez, um nó de articulação é um vértice, que, caso removido do grafo também torna o grafo desconexo (CORMEN et al., 2001, p. 558).

Estas informações têm aplicações úteis em, por exemplo, redes de distribuição elétrica, onde os postes podem representar os nós de articulação e os cabos de energia podem representar as pontes (CORMEN et al., 2001, p. 558).



Fonte: adaptado de Cormen et al. (2001, p. 559).

Figura 3 – Exemplo de uma ponte (a) e três nós de articulação (b)

2.2 PROPRIEDADES DOS GRAFOS

Existem diversas propriedades ou classificações de grafos. Saber com que tipo de grafo se está trabalhando pode ser fundamental para escolher o algoritmo certo e chegar na resposta de um problema de maneira eficiente e não errônea (SKIENA; REVILLA, 2003, p. 237).

Dentre as propriedades principais de um grafo citam-se:

a) grafo trivial: segundo Gross e Yellen (2006, p. 3), um grafo é dito trivial (Figura 4 (a)) se contém apenas um vértice e nenhuma aresta;

b) grafo nulo: Rabuske (1992, p. 9) afirma que para um grafo ser nulo (Figura 4 (b)) é necessário que o conjunto de arestas seja vazio;

c) densidade: para Lipschutz e Lipson (2004, p. 204), um grafo G com m vértices e n arestas é dito denso (Figura 4 (c)) quando m é maior ou igual a n^2, caso contrário é chamado de esparso (Figura 4 (d));

d) grafo bipartido: segundo Kocay e Kreher (2005, p. 47), um grafo G é dito bipartido (Figura 4 (e)) se pode ser dividido em dois conjuntos X e Y tal que cada aresta tem uma ponta em X e um fim em Y e/ou vice-versa;

e) grafo bipartido completo: para Scheinerman (2003, p. 433), um grafo bipartido completo (Figura 4 (f)) $K_{n,m}$ é um grafo cujos vértices podem ser particionados em dois conjuntos X e Y, onde $|X| = n$ e $|Y| = m$, de modo que para todo vértice u pertencente a X e para todo v pertencente a Y, uv é uma aresta. Nenhuma aresta tem ambas as extremidades em X ou em Y;

f) conexidade: de acordo com Rabuske (1992, p. 21), um grafo é dito conexo (Figura 4 (h)) se for possível visitar qualquer vértice, partindo de um outro e passando por arestas. Caso não seja possível, então é considerado desconexo (Figura 4 (g));

g) grafo regular: Kocay e Kreher (2005, p. 9) definem um grafo pertencente a esta classe como tendo todos os vértices com o mesmo grau. Um grafo regular de grau igual a dois é mostrado na Figura 4 (i);

h) grafo completo: para Alsuwaiyel (2003, p. 106) um grafo não-dirigido é completo se há exatamente uma aresta ligando cada par de vértices. Em outras palavras, Rabuske (1992, p. 7) afirma que um grafo completo é um grafo simples em que cada par distinto de vértices é adjacente. A Figura 4 (j) mostra dois grafos completos, sendo um formado por apenas um vértice e outro formado por quatro vértices;

i) grafo fortemente conexo: Cormen et al. (2001, p. 1082) afirma que um grafo é fortemente conexo quando é composto por apenas uma componente fortemente conexa, ou seja, se cada dois vértices do grafo são alcançáveis entre si. Tal classificação aplica-se somente a grafos dirigidos. A Figura 4 (l) mostra um grafo fortemente conexo, enquanto que a Figura 4 (k) mostra um grafo não fortemente conexo;

j) grafo simples ou multigrafo: de acordo com Lau (2007, p. 4), um grafo simples (Figura 4 (m)) é o que não possui laços ou arestas paralelas, caso contrário é

multigrafo (Figura 4 (n) e Figura 4 (o));

k) grafo cordal: segundo Gross e Yellen (2006, p. 437), um grafo é dito cordal (Figura 4 (p)) se possui número de vértices maior ou igual a quatro e se para cada subciclo de quatro ou mais vértices há uma corda, ou seja, uma aresta que não pertence ao ciclo e liga dois vértices não adjacentes;

l) grafo ciclo: segundo Braun (2009, p. 20), um grafo ciclo é formado por um único vértice com um laço, ou um grafo simples e conexo, sendo que todos os vértices e arestas existentes formam um único circuito fechado. A Figura 4 (q) mostra dois exemplos de grafos ciclos, sendo um formado por apenas um vértice, enquanto que o outro é formado por cinco vértices;

m) grafo acíclico dirigido: Goodrich e Tamassia (2004, p. 327) definem um grafo acíclico dirigido (Figura 4 (r)) como sendo um digrafo que não possui ciclos;

n) grafo planar: Rabuske (1992, p. 129) afirma que para um grafo ser planar (Figura 4 (s)) é necessário que tenha uma forma de ser representado geometricamente num plano, de tal modo que nenhuma aresta do grafo cruze com outra, caso contrário o grafo é dito não planar (Figura 4 (t)).

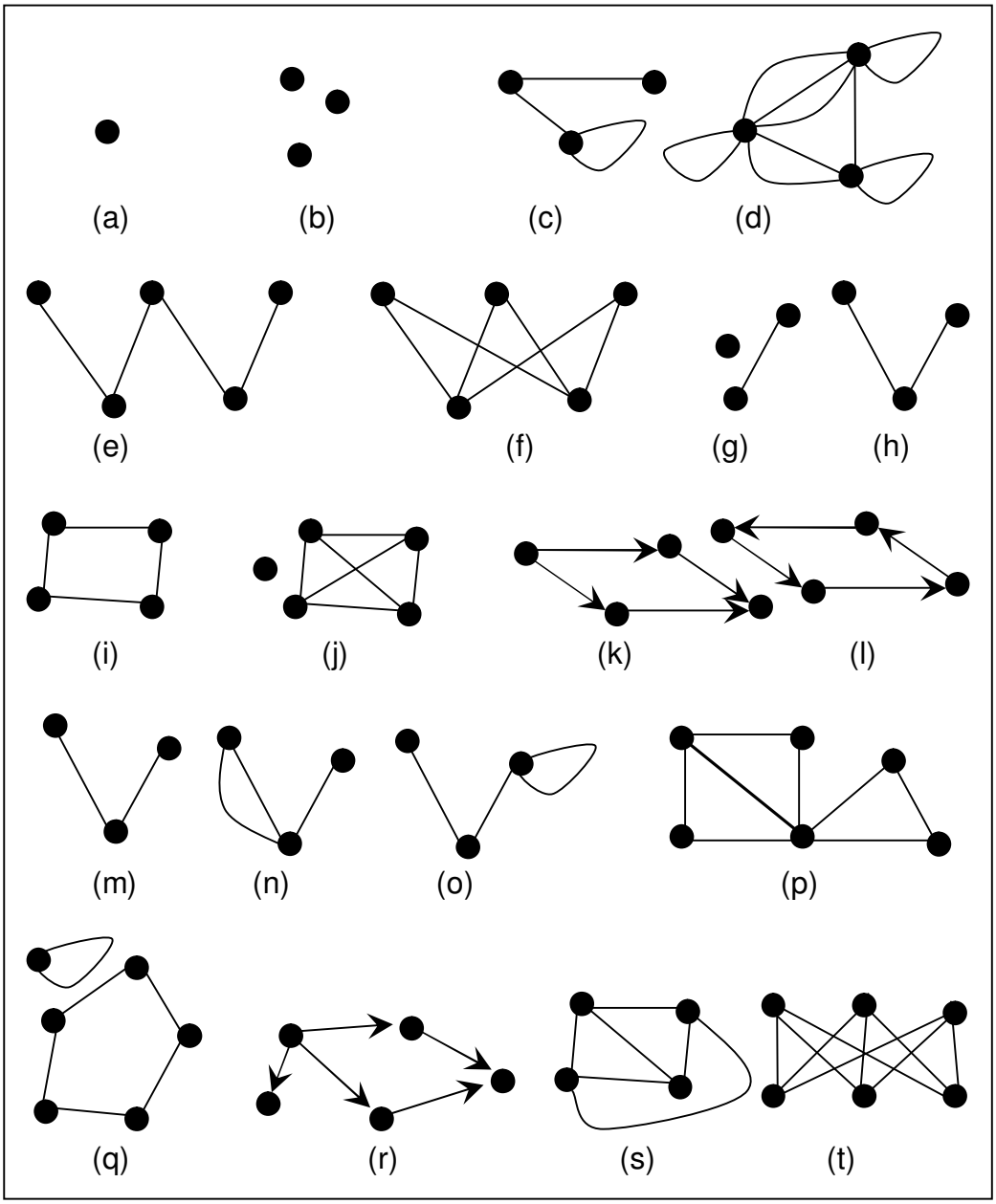


Figura 4 – Exemplos de grafos

2.3 ALGORITMOS CLÁSSICOS DE GRAFOS

É necessário saber como cada um dos algoritmos funciona, em que tipos de grafo se aplicam e como o grafo deve ser modelado. O uso de um algoritmo no lugar de outro pode comprometer tanto o desempenho do sistema como chegar a resultados errados, ou ainda trazer informações desnecessárias (SKIENA; REVILLA, 2003, p. 237).

Diante disso, abaixo são detalhados os principais algoritmos de teoria dos grafos e suas

aplicações.

2.3.1 Busca em profundidade

O algoritmo de busca em profundidade, ou *Depth First Search* (DFS), é definido como uma sequência de visitações iniciando em um vértice qualquer `p`. O método consiste em escolher algum vértice `q`, ainda não visitado, adjacente a `p` e colocar em uma pilha. Depois de examinar `p`, o vértice do topo da pilha é removido e processado da mesma forma. Este procedimento é feito até que a pilha seja vazia, e portanto o algoritmo se encerra (LAU, 2007, p. 39). Um pseudocódigo do algoritmo de busca em profundidade pode ser visto no Quadro 1.

```
dfs(G)
    para cada u em V
        visitado[u] = false

    para cada u em V
        se visitado[u] = false
            push(S, u)
            enquanto S não é vazia
                u = pop(S)
                se visitado[u] = false
                    visitado[u] = true
                    para cada v em Adjacente(u)
                        se visitado[v] = false
                            push(S, v)
```

Fonte: adaptado de Gross e Yellen (2006, p. 132).

Quadro 1 – Algoritmo busca em profundidade

2.3.2 Busca em largura

A diferença entre o algoritmo da busca em profundidade e busca em largura, ou *Breadth First Search* (BFS), está na estrutura de dados auxiliar utilizada. Enquanto que a primeira armazena os vértices em uma pilha, a segunda os armazena em uma fila. Basicamente na busca em largura é escolhido um vértice inicial `p`. O método então visita todos os vértices adjacentes a `p`, ainda não visitados, e os adiciona em uma fila. Quando todos os vizinhos são visitados então um vértice novo é removido da fila e processado da mesma maneira que `p`. O algoritmo encerra quando a fila ficar vazia (LAU, 2007, p. 43). Um pseudocódigo do algoritmo de busca em largura pode ser visto no Quadro 2.

```
bfs(G)
    para cada u em V
        visitado[u] = false

    para cada u em V
        se visitado[u] = false
            insere(Q, u)
            enquanto Q não é vazia
                u = retira(Q)
                para cada v em Adjacente(u)
                    se visitado[v] = false
                        insere(Q, v)
                        visitado[u] = true
```

Fonte: adaptado de Gross e Yellen (2006, p. 134).

Quadro 2 – Algoritmo busca em largura

2.3.3 Ordenação topológica

Uma ordenação topológica é uma ordenação de todos os vértices de um grafo acíclico dirigido, sendo que, se o grafo `G` contém a aresta `(u,v)` então `u` aparece antes de `v`. Já, se o grafo não contém ciclo então esta ordenação é possível. De forma simplificada, a ordenação topológica de um grafo pode ser imaginada como uma linha horizontal onde todas as arestas orientadas vão da esquerda para a direita (CORMEN et al., 2001, p. 494). Um pseudocódigo do algoritmo de ordenação topológica pode ser visto no Quadro 3.

O pseudocódigo inicia com uma lista vazia `Q` e com todos os vértices marcados como não visitados. A seguir, para cada vértice ainda não visitado, é iniciada uma busca em profundidade e, ao encerrar cada vértice, o mesmo é adicionado no início da lista `Q`. O pseudocódigo encerra retornando a lista (CORMEN et al., 2001, p. 550).

```
topologica(G)
    Q = {}
    para cada u em V
        visitado[u] = false

    para cada u em V
        se visitado[u] = false
            dfs(u)
    retorna Q

dfs(u)
    visitado[u] = true
    para cada v em Adjacente(u)
        se visitado[v] = false
            dfs(v)

    insere(Q,u)
```

Fonte: adaptado de Cormen et al. (2001, p. 550).

Quadro 3 – Algoritmo ordenação topológica

2.3.4 Algoritmo de Dijkstra

Segundo Goodrich e Tamassia (2004, p. 372), o algoritmo de Dijkstra é utilizado para encontrar o caminho de custo mínimo entre dois vértices em grafos com apenas pesos positivos. O caminho de custo mínimo é um caminho que, considerando os pesos das arestas como distâncias, o somatório dos pesos das arestas envolvidas no trajeto é o menor possível.

Kocay e Kreher (2005, p. 35) classificam o algoritmo de Dijkstra como um algoritmo guloso, pois a cada iteração o vértice seguinte a ser escolhido é o mais próximo. Um pseudocódigo do algoritmo de Dijkstra pode ser visto no Quadro 4.

```
dijkstra(G, s, d)
    para cada u em V
        visitado[u] = false
        distancia[u] = INF

    visitado[s] = true
    distancia[s] = 0

    insere(Q, {s, distancia[s]})
    enquanto Q não é vazia
        {u, dc} = retira(Q)
        para cada v em V
            se visitado[v] = false
                novaDistancia = dc + custo[u, v]
                se (novaDistancia < distancia[v])
                    distancia[v] = novaDistancia
                insere(Q, {v, distancia[v]})

        visitado[u] = true

    retorna distancia[d]
```

Fonte: adaptado de Gross e Yellen (2006, p. 148).

Quadro 4 – Algoritmo de Dijkstra

2.3.5 Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é utilizado em problemas onde é necessário encontrar o caminho de custo mínimo entre todos os pares de vértices de um grafo. Este algoritmo também pode ser executado sobre um grafo que contém arestas com pesos negativos, mas que não há ciclos negativos (CORMEN et al., 2001, p. 629).

O algoritmo inicialmente tem uma matriz com custos infinitos entre todos os pares de vértices. Para cada dois vértices `i` e `j`, em cada etapa do algoritmo é gerada uma nova matriz com os custos mínimos entre ambos, passando por alguns vértices intermediários, e o

resultado estará na última matriz. Também é possível controlar uma outra matriz, chamada matriz de roteamento, que é responsável por manter o caminho mínimo entre cada par de vértice (CORMEN et al., 2001, p. 629). Um pseudocódigo do algoritmo de Floyd-Warshall pode ser visto no Quadro 5.

No pseudocódigo é mostrada a inicialização do algoritmo, atribuindo para cada par de vértices `(u, v)`, o custo de `u` até `v`, caso o vértice `u` é adjacente ao vértice `v` ou infinito caso não seja. A seguir para cada vértice intermediário `k` é verificado para cada par de vértices `(u, v)` se a soma da distância de `u` até `k` e de `k` até `v` é menor que a distância de `u` até `v`. Em caso verdadeiro a nova distância de `u` até `v` é atualizada (CORMEN et al., 2001, p. 635).

```
floyd(G)
    para cada u em V
        para cada v em V
            se u é adjacente a v
                distancia[u,v] = custo[u,v]
            senão
                distancia[u,v] = INF

    para cada k em V
        para cada u em V
            para cada v em V
                se (distancia[u,v] > (distancia[u,k] + distancia[k,v]))
                    distancia[u,v] = distancia[u,k] + distancia[k,v]
```

Fonte: adaptado de Cormen et al. (2001, p. 635).

Quadro 5 – Algoritmo de Floyd-Warshall

2.3.6 Algoritmo de Bellman-Ford

Este algoritmo surge para resolver o problema de encontrar um caminho de custo mínimo de um vértice de origem a todos os demais. Um diferencial do algoritmo é que o grafo pode conter arestas com peso negativo. Entretanto, o algoritmo não consegue chegar a um resultado quando o grafo originar um ciclo de custo negativo (CORMEN et al., 2001, p. 588). Para que o algoritmo execute de forma correta é necessário que o grafo seja também dirigido, caso contrário, a passagem por uma aresta inúmeras vezes pode criar o ciclo de custo negativo (GOODRICH; TAMASSIA, 2007, p. 534). Um pseudocódigo do algoritmo de Bellman-Ford pode ser visto no Quadro 6.

O pseudocódigo inicia atribuindo infinito para a distância até cada vértice `u`. Em seguida atribui zero como distância até o vértice de origem `s` e para cada vértice `i` é percorrida a lista de arestas do grafo. Para cada aresta `a` é verificado se a soma da distância do vértice `a.u` e do peso da aresta é menor que a distância atual para o vértice `a.v`. Em caso

verdadeiro a nova distância é atribuída como distância mínima até o vértice `a.v` (CORMEN et al., 2001, p. 588).

```
Ford(G, s)
    para cada u em V
        distancia[u] = INF

    distancia[s] = 0

    para i = 1 até tamanho(G)
        para cada a em A
            se distancia[a.v] > (distancia[a.u] + a.peso)
                distancia[a.v] = (distancia[a.u] + a.peso)
```

Fonte: adaptado de Cormen et al. (2001, p. 588).

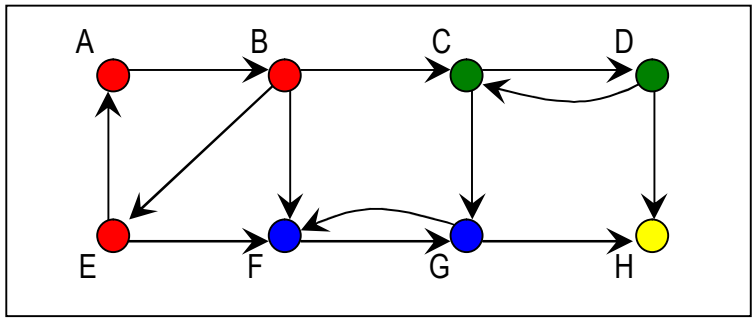
Quadro 6 – Algoritmo de Bellman-Ford

2.3.7 Algoritmo de Hopcroft-Tarjan

A algoritmo de Hopcroft-Tarjan foi o primeiro a reconhecer a importância algorítmica da busca em profundidade. O núcleo do algoritmo contém como base uma busca em profundidade modificada que é utilizada para encontrar as componentes fortemente conexas de um grafo (ALSUWAIYEL, 2003, p. 273).

Uma componente fortemente conexa em um grafo é um subconjunto de vértices do grafo onde para cada vértice `u` e para cada vértice `v`, existe um caminho que vai de `u` até `v` e outro que vai de `v` até `u`, ou seja, todos os vértices de uma componente fortemente conexa são alcançáveis entre si (CORMEN et al., 2001, p. 490).

Um exemplo de um grafo e suas componentes fortemente conexas é exibido na Figura 5. Os vértices de uma mesma cor pertencem a mesma componente fortemente conexa.



Fonte: adaptado de Cormen et al. (2001, p. 553).

Figura 5 – Exemplo de um grafo e suas componentes fortemente conexas

Um pseudocódigo do algoritmo de Hopcroft-Tarjan pode ser visto no Quadro 7.

```
tarjan(G)
    idx = 0
    para cada u em V
        num[u] = INF
        low[u] = INF

    para cada u em V
        se num[u] = INF
            dfs(u)

dfs(u)
    num[u] = idx
    low[u] = idx
    idx = idx + 1

    push(S, u)

    para cada v em Adjacente(u)
        se num[v] = INF
            dfs(v)
            se low[v] < low[u]
                low[u] = low[v]
        senao
            se v esta em S
                se num[v] < low[u]
                    low[u] = num[v]

    se low[u] = num[u]
        faca
            v = pop(S)

            insere(L, v)
        enquanto u diferente de v
        insere(Componentes, L)
```

Fonte: adaptado de Eppstein (2001).

Quadro 7 – Algoritmo de Hopcroft-Tarjan

2.3.8 Algoritmo de Prim

O algoritmo de Prim é utilizado para encontrar uma árvore geradora de custo mínimo em um grafo `G` (LAU, 2007, p. 75). A árvore geradora de um grafo `G` é um subgrafo de `G` formado pelo menor número de arestas que mantém o subgrafo ainda conexo. Já, a árvore geradora de custo mínimo é a árvore geradora formada pelas arestas que, somando seus pesos, dará o menor custo total.

Inicialmente é escolhido um vértice `j` que constrói parcialmente uma árvore `T`. Une-se a `T` a aresta do grafo cujo peso é mínimo entre todas as demais arestas com um vértice pertencente a `T` e outro vértice não pertencente a `T`. Este processo é repetido enquanto existam vértices que possam ser inseridos em `T` (LAU, 2007, p. 75). A execução do algoritmo de Prim

é restrita a grafos conexos (GOODRICH; TAMASSIA, 2004, p. 368). Um pseudocódigo do algoritmo de Prim pode ser visto no Quadro 8.

No pseudocódigo é mostrada a inicialização do algoritmo, atribuindo nulo para o predecessor de um vértice, infinito para a distância de um vértice `u` até a árvore `T`, e falso para o vetor `naArvore`, responsável por controlar se um vértice está ou não contido na árvore `T`. A seguir, um vértice arbitrário `u` é adicionado a uma fila de prioridades `Q` com o custo zero (GROSS; YELLEN, 2006, p. 146).

Enquanto a fila `Q` não estiver vazia, o primeiro vértice `u` é retirado, adicionado na árvore `T` e para cada vértice `v`, adjacente a `u`, é verificado se o custo de `u` até `v` é menor que o custo para adicionar `v` na árvore. Em caso positivo é definido `u` como predecessor de `v` e atualizada a distância de `v` até `T`. Ao final do algoritmo, a árvore geradora de custo mínino pode ser recuperada com os valores contidos em `predecessor` e `distancia` (GROSS; YELLEN, 2006, p. 146).

```
prim(G)
    para cada u em V
        predecessor[u] = null
        distancia[u] = INF
        naArvore[u] = false

    u = um vertice qualquer
    insere(Q, {u, 0})
    enquanto Q não é vazia
        {u, dc} = retira(Q)

        se naArvore[u] = false
            naArvore[u] = true

            para cada v em Adjacente(u)
                se naArvore[v] = false E custo[u, v] < distancia[v]
                    distancia[v] = custo[u, v]
                    predecessor[v] = u
                    insere(Q, {v, distancia[v]})
```

Fonte: adaptado de Gross e Yellen (2006, p. 146).

Quadro 8 – Algoritmo de Prim

2.3.9 Algoritmo de Kruskal

O algoritmo de Kruskal é utilizado para gerar árvores geradoras de custo mínimo em grafos desconexos (KOCAY; KREHER, 2005, p. 76). Diferentemente do algoritmo de Prim, que inicia em algum vértice, o algoritmo de Kruskal inicia a partir de alguma aresta. Inicialmente cada componente de um grafo é composta por apenas um vértice. Em cada

iteração a aresta de menor peso conecta duas componentes distintas. O algoritmo termina quando não é mais possível ligar nenhuma componente. O resultado é uma floresta, que é um conjunto de uma ou mais árvores (KOCAY; KREHER, 2005, p. 76). Um pseudocódigo do algoritmo de Kruskal pode ser visto no Quadro 9.

```
kruskal(G)
    para cada u em V
        predecessor[u] = u
        tamanho[u] = 1

    insere(Q, A)
    ordena Q em ordem crescente pelo custo

    k = 0
    enquanto k < tamanho(G) E Q não é vazia
        a = retira(Q)
        se find(a.u) diferente de find(a.v)
            union(a.u, a.v)
            k = k + 1

find(u)
    retorna raiz da arvore que contem u

union(u,v)
    se tamanho[u] < tamanho[v]
        predecessor[u] = v
        tamanho[v] = tamanho[v] + tamanho[u]
    senao
        predecessor[v] = u
        tamanho[u] = tamanho[u] + tamanho[v]
```

Fonte: adaptado de Cormen et al. (2001, p. 569).

Quadro 9 – Algoritmo de Kruskal

2.3.10 Algoritmo de Ford-Fulkerson

A proposta do algoritmo de Ford-Fulkerson é determinar o fluxo máximo em uma rede de fluxo utilizando um método guloso (GOODRICH; TAMASSIA, 2004, p. 389). Para entender como este algoritmo funciona é necessário entender o conceito de rede de fluxo. Uma rede de fluxo é um grafo em que o valor de uma aresta é chamado de capacidade. A capacidade é o valor máximo de um produto que pode passar pela aresta (GOODRICH; TAMASSIA, 2004, p. 382).

A ideia principal é aumentar o valor de um fluxo em estágios, onde para cada etapa uma quantidade é colocada ao longo de um caminho entre a origem e o destino. O algoritmo encerra quando não existe mais um caminho para utilizar um fluxo. O resultado final é o valor máximo de fluxo de produto que pode atravessar a rede (GOODRICH; TAMASSIA, 2004, p.

389).

Além de encontrar o fluxo máximo em grafos, o algoritmo de Ford-Fulkerson pode ser aplicado juntamente com o algoritmo de Dijkstra, sendo assim possível encontrar o fluxo máximo de custo mínimo. O algoritmo de Dijkstra entra no lugar do algoritmo de busca em profundidade para gerar os caminhos onde passará o fluxo de dados, portanto é obrigatório que o grafo utilizado para a execução do algoritmo não forme nenhum ciclo com custo negativo (FLEMING; CRUTCHFIELD, 2006).

Outro problema que pode ser resolvido a partir do algoritmo de Ford-Fulkerson é o emparelhamento máximo em grafos bipartidos. Um emparelhamento é um subconjunto de arestas de um grafo que são mutuamente não adjacentes. Para a modelagem do problema é necessário adicionar dois novos vértices ao grafo. Um vértice para ser a fonte, que será ligada a todos os demais de um dos conjuntos do grafo bipartido, e outro para ser o sorvedouro, que será ligado a todos os vértices do outro conjunto (GROSS; YELLEN, 2006, p. 427). Sendo a capacidade de todas as arestas do grafo sempre igual a um, após a execução do algoritmo de Ford-Fulkerson o resultado será o emparelhamento máximo possível para o grafo bipartido dado (GROSS; YELLEN, 2006, p. 427).

Um pseudocódigo do algoritmo de Ford-Fulkerson pode ser visto no Quadro 10. O algoritmo recebe como parâmetros um grafo `G`, um vértice de origem `s` e um vértice de destino `d`. Inicialmente é atribuído zero como fluxo inicial entre todos os pares de vértices. A partir daí, enquanto existir um caminho `C`, entre a origem e o dentino, é obtida a capacidade da aresta de menor capacidade no caminho. Por fim, a capacidade é somada ao fluxo de cada aresta `a` que pertence ao caminho (CORMEN et al., 2001, p. 658).

```
fulkerson(G, s, d)
    para cada vértice u em V
        para cada vértice v em V
            fluxo[u, v] = 0

    enquanto existir caminho C entre s e d
        fatual = capacidade da aresta de menor capacidade no caminho C
        para cada aresta a em C
            fluxo[u, v] = fluxo[u, v] + fatual
            fluxo[v, u] = fluxo[v, u] – fatual
```

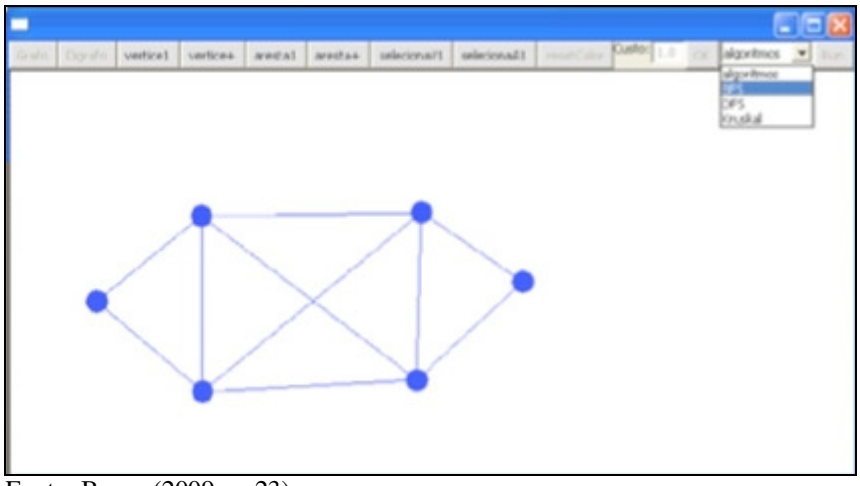
Fonte: adaptado de Cormen et al. (2001, p. 658).

Quadro 10 – Algoritmo de Ford-Fulkerson

2.4 TRABALHOS CORRELATOS

Nesta seção são descritos três trabalhos que apresentam algumas das funcionalidades explanadas ou relacionadas aos principais objetivos deste trabalho: "Ferramenta para representação gráfica do funcionamento de algoritmos aplicados em grafos" (HACKBART, 2008), "JgraphT" (JGRAPHT TEAM, 2005) e "Ferramenta visual para criação e execução de algoritmos aplicados sobre teoria dos grafos" (BRAUN, 2008).

O primeiro deles descrito em Hackbarth (2008) é uma ferramenta para representação gráfica do funcionamento de algoritmos de grafos, que implementa os algoritmos de busca em largura, profundidade e Dijkstra. Neste trabalho há preocupação com as questões didáticas, pois o objetivo da ferramenta é fazer com que o usuário compreenda rapidamente o funcionamento dos algoritmos disponibilizados. A Figura 6 exibe a interface gráfica da ferramenta.



Fonte: Braun (2009, p. 23).

Figura 6 – Interface gráfica da ferramenta

O segundo trabalho correlato é uma biblioteca escrita na linguagem Java, a JgraphT (JGRAPHT TEAM, 2005). A biblioteca dispõe de alguns algoritmos de grafos, tais como Bellman-Ford, busca em largura e profundidade, Dijkstra, Floyd-Warshall, ordenação topológica e Edmonds-Karp. Também mantém classes para notificar, por meio de eventos, alterações realizadas nos grafos, vértices atingidos em determinado algoritmo, arestas visitadas, entre outros. Além disso, para diversos tipos de grafos há um gerador aleatório (grafos nulos, completos, bipartidos completos, hiper cubo, ciclo, estrela e roda). Outras características são a forte integração com a biblioteca Jgraph, utilizada para representação gráfica dos modelos de grafos criados, e a exportação de grafos no formato *Comma-separated Values* (CSV), que pode ser aberto no *Microsoft Visio*. O Quadro 11 mostra um trecho de

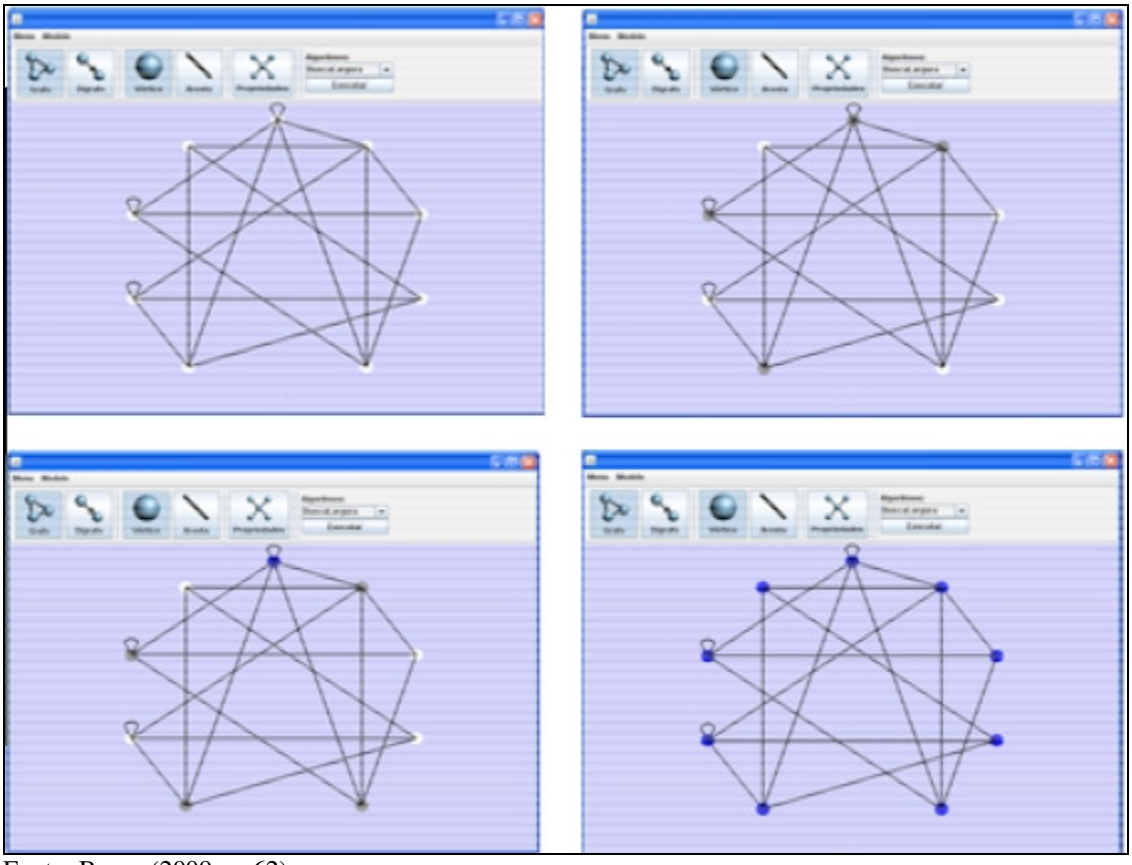
código feito com o auxílio da JgraphT que evidencia a integração com a biblioteca Jgraph.

```
    public void init(  ) {
        // create a JGraphT graph
        ListenableGraph g=new ListenableDirectedGraph(DefaultEdge.class);

        // create a visualization using JGraph, via an adapter
        m_jgAdapter = new JGraphModelAdapter( g );

        JGraph jgraph = new JGraph( m_jgAdapter );

        adjustDisplaySettings( jgraph );
        getContentPane(  ).add( jgraph );
        resize( DEFAULT_SIZE );

        // add some sample data (graph manipulated via JGraphT)
        g.addVertex( "v1" );
        g.addVertex( "v2" );
        g.addVertex( "v3" );
        g.addVertex( "v4" );

        g.addEdge( "v1", "v2" );
        g.addEdge( "v2", "v3" );
        g.addEdge( "v3", "v1" );
        g.addEdge( "v4", "v3" );

        // position vertices nicely within JGraph component
        positionVertexAt( "v1", 130, 40 );
        positionVertexAt( "v2", 60, 200 );
        positionVertexAt( "v3", 310, 230 );
        positionVertexAt( "v4", 380, 70 );
    }
```

Fonte: JgraphT Team (2005).

Quadro 11 – Exemplo de integração da biblioteca `JgraphT` e `Jgraph`

O terceiro trabalho, segundo Braun (2009), é uma aplicação onde é possível desenvolver algoritmos, desenhar um grafo e acompanhar a execução do algoritmo sobre o grafo. Basicamente a aplicação foi dividida em dois módulos, um para a criação e compilação de algoritmos de grafos na linguagem Java versão 5 e outro para a criação de grafos, onde é também permitida a seleção de um algoritmo existente e o acompanhamento de sua execução em um modo visual. Outra funcionalidade é a exportação do grafo no formato graphml, o que permite que outros sistemas possam abrir o mesmo grafo. Um exemplo da interface gráfica da ferramenta pode ser visto na Figura 7.



Fonte: Braun (2009, p. 62).

Figura 7 – Interface gráfica da ferramenta

O Quadro 12 apresenta uma comparação entre os três trabalhos correlatos.

	JgraphT (2005)	HACKBART (2008)	BRAUN (2009)
Permite criar grafos	Sim	Sim	Sim
Permite estender vértices, arestas e grafos	Sim	Não	Não
Permite gerar grafos	Sim	Não	Não
Permite persistir grafos	Sim	Não	Sim
Permite verificar propriedades	Sim	Não	Sim
Disponibiliza algoritmos	Sim	Sim	Não
Permite criar novos algoritmos	Sim	Não	Sim
Permite acompanhar execução de algoritmos	Sim	Sim	Sim

Quadro 12 – Comparação entre trabalhos correlatos

3 DESENVOLVIMENTO

Neste capítulo é abordado o desenvolvimento do FGA. Primeiramente são apresentados os requisitos do FGA. Na sequência é mostrada a especificação, que dá uma maior formalização sobre a ferramenta. Logo em seguida é apresentada a implementação, onde são comentadas as técnicas e ferramentas utilizadas, o desenvolvimento de cada etapa e a operacionalidade da ferramenta. Por fim são discutidos os testes e resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O sistema desenvolvido atende aos seguintes requisitos:

a) disponibilizar funções para criação e edição de grafos com as seguintes operações: adicionar e remover vértices e arestas, atribuir pesos e capacidades para arestas (Requisito Funcional - RF);
b) disponibilizar funções para gerar grafos completos, regulares, conexos ou desconexos, densos ou esparsos e simples ou multigrafos (RF);
c) permitir extrair propriedades tais como o grafo ser completo, regular, trivial, nulo, bipartido, bipartido completo, conexo ou desconexo, fortemente conexo, denso ou esparso e simples ou multigrafo (RF);
d) disponibilizar um subconjunto de algoritmos clássicos, sendo: Dijkstra, Floyd-Warshall, Bellman-Ford, Prim, Kruskal, Ford-Fulkerson, Hopcroft-Tarjan, ordenação topológica, busca em largura e busca em profundidade (RF);
e) permitir persistir e carregar grafos (RF);
f) ser implementado utilizando o ambiente Fedora Eclipse 3.4.1 e a linguagem Java versão 6 (Requisito Não-Funcional - RNF);
g) conter documentação detalhada sobre os recursos oferecidos pelo FGA (RNF).

3.2 ESPECIFICAÇÃO

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

3.2.1 Casos de uso

O FGA é constituído de seis casos de uso (Figura 8). Os três primeiros mostram as diferentes formas de poder instanciar um grafo: criando manualmente, gerando um grafo com base em algumas restrições ou mesmo carregando um grafo a partir de um arquivo. Já o caso de uso, designado `Salvar grafo em arquivo`, detalha como é feito o procedimento para persistir um grafo criado.

Por fim, os dois últimos casos de uso, designados por `Executar algoritmos` e `Testar propriedades`, detalham as formas de trabalhar com o grafo.

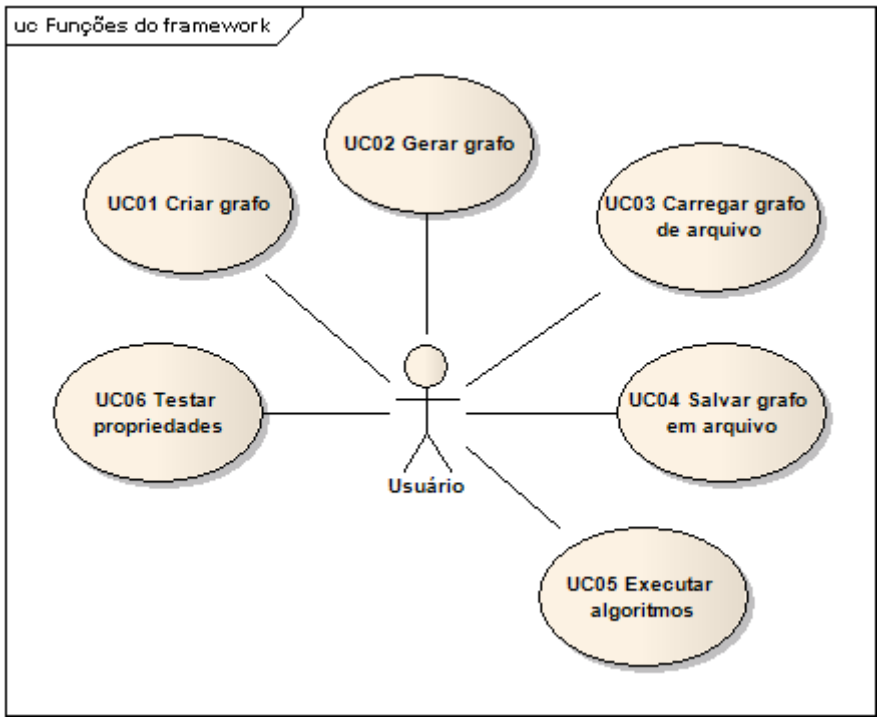


Figura 8 – Diagrama de casos de uso

O primeiro caso de uso (Quadro 13), chamado de `Criar grafo`, descreve o processo de criar e editar um grafo. Ele possui um cenário principal, dois alternativos, permitindo ao

usuário poder adicionar mais vértices ou arestas, e dois cenários de exceção, os quais informam se há duplicidade de identificadores de vértices ou arestas no grafo.

Cenário principal	1 Usuário solicita criação de um tipo de grafo 2 FGA retorna um grafo do tipo escolhido 3 Usuário cria um vértice 4 Usuário informa um identificador para o vértice 5 Usuário adiciona vértice 6 Usuário cria uma aresta 7 Usuário informa um identificador para a aresta 8 Usuário informa o vértice de origem e o vértice de destino 9 Usuário adiciona aresta
Fluxo alternativo 1	No passo 5, caso o usuário deseja infomar outro vértice: 5.1 Retorna ao passo 3
Fluxo alternativo 2	No passo 9, caso o usuário deseja infomar outra aresta: 9.1 Retorna ao passo 6
Exceção 1	No passo 5, caso o identificador do vértice já existir no grafo: 5.1 FGA retorna exceção avisando da duplicidade
Exceção 2	No passo 9, caso o identificador da aresta já existir no grafo: 9.1 FGA retorna exceção avisando da duplicidade

Quadro 13 – Caso de uso `Criar grafo`

O segundo caso de uso (Quadro 14), designado `Gerar grafo`, descreve o procedimento para gerar um grafo com base em restrições. Ele possui, além do cenário principal, um cenário alternativo, necessário para caso haja mais parâmetros para a geração do grafo, e um cenário de exceção, para informar sobre a impossibilidade de gerar um grafo com as especificações dadas pelo usuário.

Cenário principal	1 Usuário solicita a geração de um grafo de um certo tipo 2 FGA solicita a quantidade de vértices 3 Usuário informa a quantidade de vértices 4 FGA retorna o grafo gerado
Fluxo alternativo 1	No passo 3, caso o tipo de grafo solicitado necessite de mais parâmetros: 3.1 FGA solicita outros parâmetros 3.2 Usuário informa outros parâmetros
Exceção 1	No passo 3, caso seja impossível gerar um grafo com as especificações do usuário: 3.1 FGA retorna exceção avisando da impossibilidade

Quadro 14 – Caso de uso `Gerar grafo`

O terceiro caso de uso (Quadro 15), chamado de `Carregar grafo de arquivo`, descreve o processo de recuperação de um grafo persistido em um arquivo. Ele possui um cenário principal e outro cenário de exceção, que informa caso um arquivo não exista.

Cenário principal	1 Usuário informa arquivo onde está o grafo 2 FGA retorna o grafo contido no arquivo
Exceção 1	No passo 1, caso o arquivo não exista: 1.1 FGA retorna exceção avisando da inexistência do arquivo

Quadro 15 – Caso de uso Carregar grafo de arquivo

O quarto caso de uso (Quadro 16), designado Salvar grafo em arquivo, descreve o processo para salvar um grafo em um arquivo. Ele possui, além do cenário principal, um cenário de exceção, que informa ao usuário se o caminho passado como parâmetro é inválido.

Cenário principal	1 Usuário informa grafo a ser salvo 2 Usuário informa arquivo onde será salvo o grafo 3 FGA salva o grafo
Exceção 1	No passo 2, caso o endereço for inválido: 2.1 FGA retorna exceção avisando da invalidez

Quadro 16 – Caso de uso Salvar grafo em arquivo

O quinto caso de uso (Quadro 17), chamado de Executar algoritmos, descreve o procedimento para executar um algoritmo sobre um grafo. Ele possui, além do cenário principal, um cenário alternativo, para o caso de haver mais parâmetros necessários para a execução do algoritmo, e um cenário de exceção, para informar ao usuário se o grafo dado como parâmetro é inválido para o algoritmo escolhido.

Cenário principal	1 Usuário solicita a execução de um algoritmo 2 FGA solicita um grafo como parâmetro 3 Usuário informa o grafo para o algoritmo 4 FGA executa o algoritmo solicitado 5 Usuário solicita obter o resultado 6 FGA retorna o resultado da execução do algoritmo
Fluxo alternativo 1	No passo 3, caso o algoritmo necessite de mais parâmetros: 3.1 FGA solicita outros parâmetros 3.2 Usuário informa outros parâmetros
Exceção 1	No passo 3, caso o grafo informado for inválido para a execução do algoritmo: 3.1 FGA retorna exceção avisando da invalidez do grafo

Quadro 17 – Caso de uso Executar algoritmos

O sexto caso de uso (Quadro 18), designado Testar propriedades, descreve o processo para testar alguma propriedade de um grafo. Ele possui somente um cenário principal, já que não há como ocorrer exceção ou mesmo haver um fluxo alternativo.

Cenário principal	1 Usuário solicita o teste de uma propriedade 2 FGA solicita um grafo 3 Usuário informa o grafo 4 FGA retorna o resultado do teste

Quadro 18 – Caso de uso Testar propriedades

3.2.2 Diagrama de classes

O diagrama de classes apresenta uma visão de como as classes estão estruturadas e relacionadas. Inicialmente é mostrado um diagrama de pacotes (Figura 9), que exibe uma visão macro sobre como estão dispostas as classes do FGA e da aplicação de teste, sendo esta última não tendo suas classes detalhadas posteriormente.

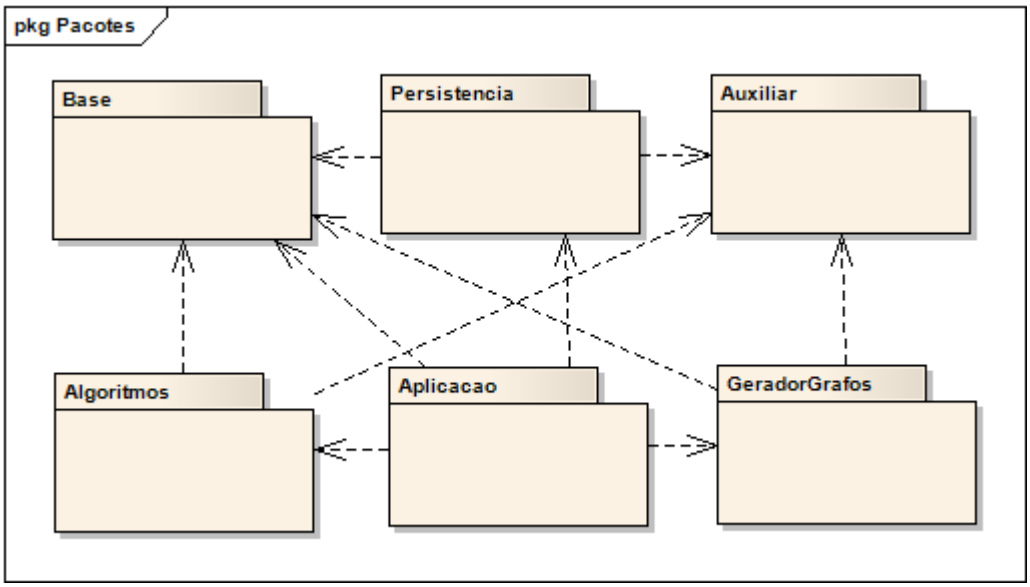


Figura 9 – Diagrama de pacotes

O primeiro pacote apresentado é o que contém as classes que compõe a estrutura principal de um grafo. A Figura 10 mostra uma visão geral de como estão relacionadas as classes do pacote `Base`, enquanto que a Figura 11 exibe uma visão detalhada dos atributos e métodos de cada uma das classes.

Neste pacote estão inclusas as classes que representam a estrutura de um grafo, contendo a classe abstrata `Grafo`, que implementa os métodos e atributos comuns a todos os grafos, e como classes específicas para grafos dirigidos e não dirigidos as classes `GrafoDirigido` e `GrafoNaoDirigido`, respectivamente.

Além destas, estão inclusas as classes que trabalham com as arestas. Há uma classe abstrata chamada `Aresta`, a qual implementa os métodos e atributos comuns a todo tipo de aresta, e também há suas extensões, a classe `ArestaDirigida`, que implementa os métodos pertencentes somente a arestas dirigidas, e a classe `ArestaNaoDirigida`, que contém a implementação específica de uma aresta não dirigida.

Por fim, existe uma classe `Vertice`, que é responsável pela implementação dos vértices do grafo. Não há classes específicas para um vértice dirigido e um não dirigido, já

que não há o conceito de direção de um vértice.

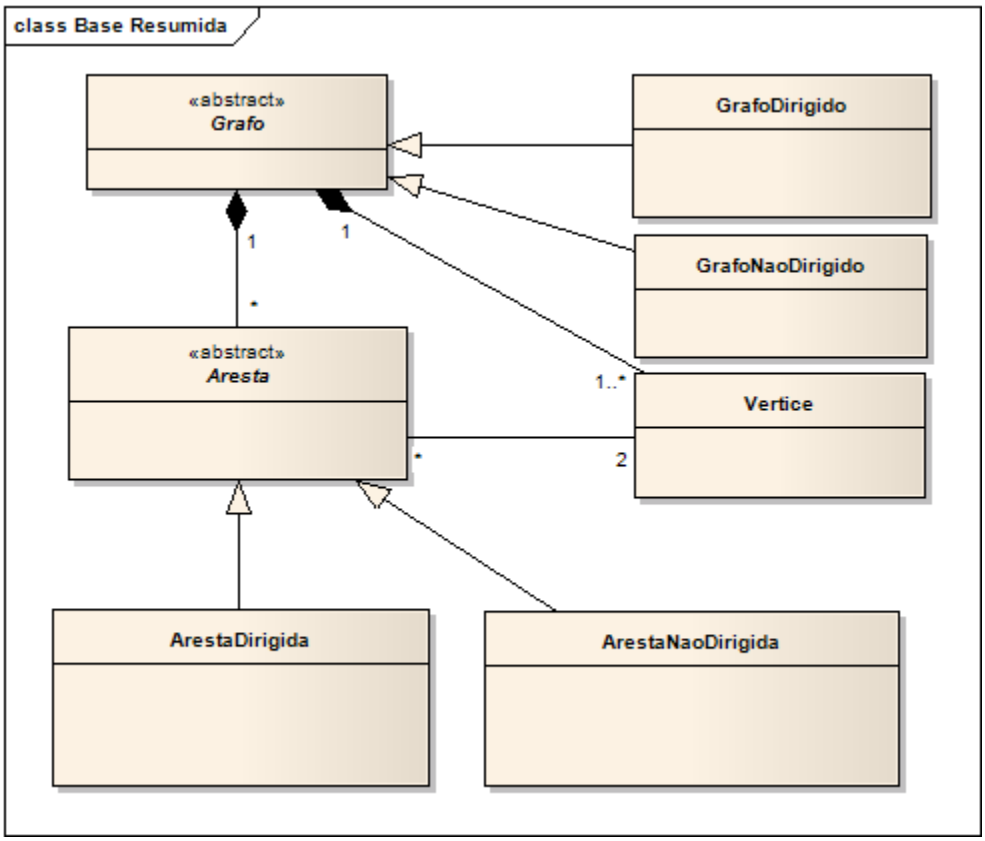


Figura 10 – Diagrama de classes resumido do pacote `Base`

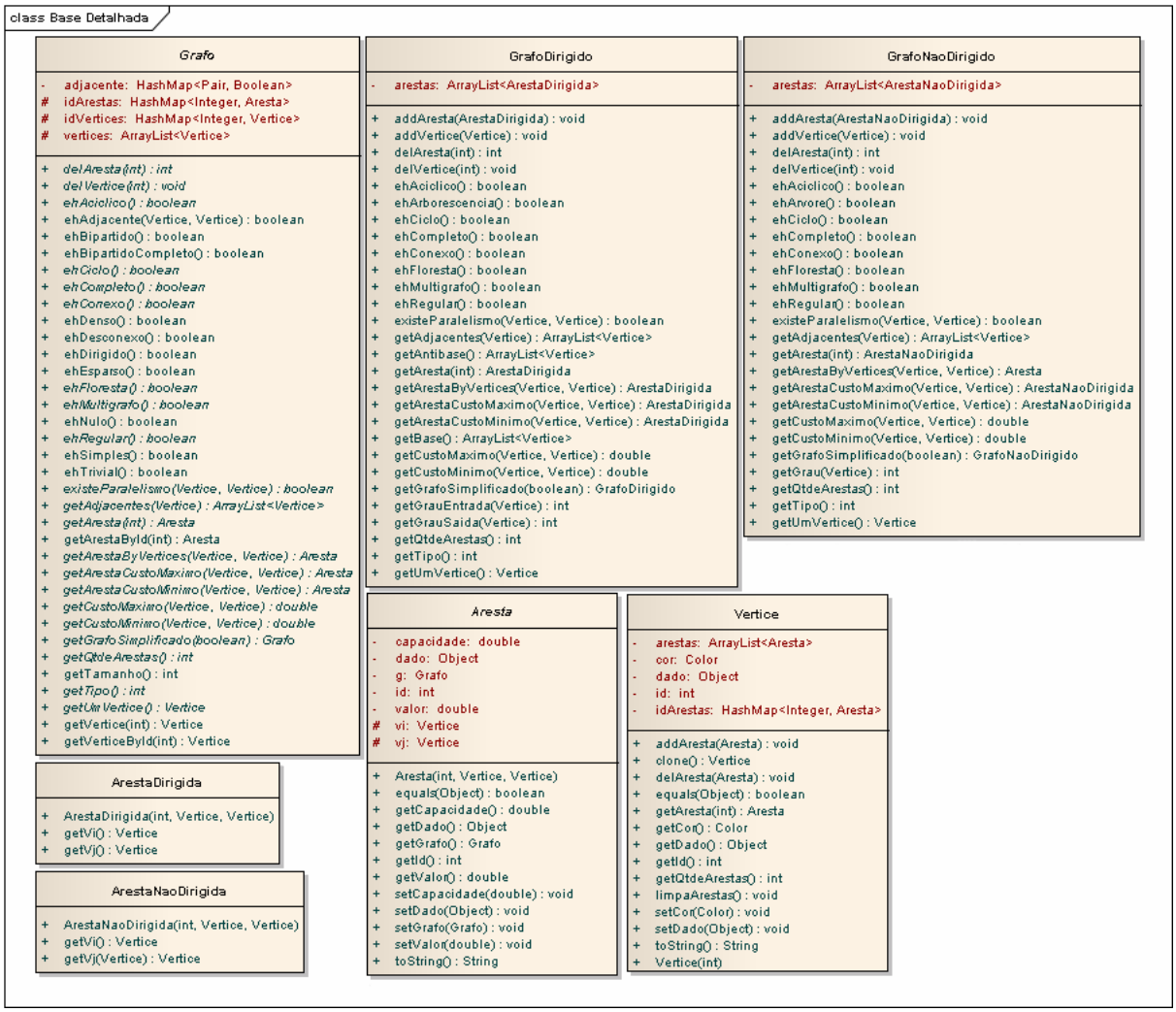


Figura 11 – Diagrama de classes detalhado do pacote `Base`

O segundo pacote apresentado contém as classes responsáveis pela execução de algoritmos e seus respectivos resultados. A organização de classes de todos os algoritmos e resultados foi modelada de forma semelhante, portando é apresentado na Figura 12, o diagrama de classes detalhado do algoritmo de busca em profundidade, enquanto que na Figura 13 são mostrados os demais algoritmos de forma resumida.

Para todo algoritmo que o FGA disponibiliza, existe uma classe que executa o algoritmo e outra que apenas mantém o resultado obtido durante a execução do mesmo. Optou-se por modelar desta forma porque o tipo de resultado depende muito do tipo de algoritmo, sendo então conveniente que cada algoritmo seja associado a uma classe específica de algoritmo. A Figura 12 mostra a classe `AlgoritmoBuscaProfundidade`, que executa o algoritmo de busca em profundidade, e a classe `AlgoritmoBuscaProfundidadeResultado`, que contém o resultado da execução. Neste caso, o resultado consiste na árvode de busca e por valores indicando a ordem de visitação dos vértices, feita durante a execução do algoritmo.

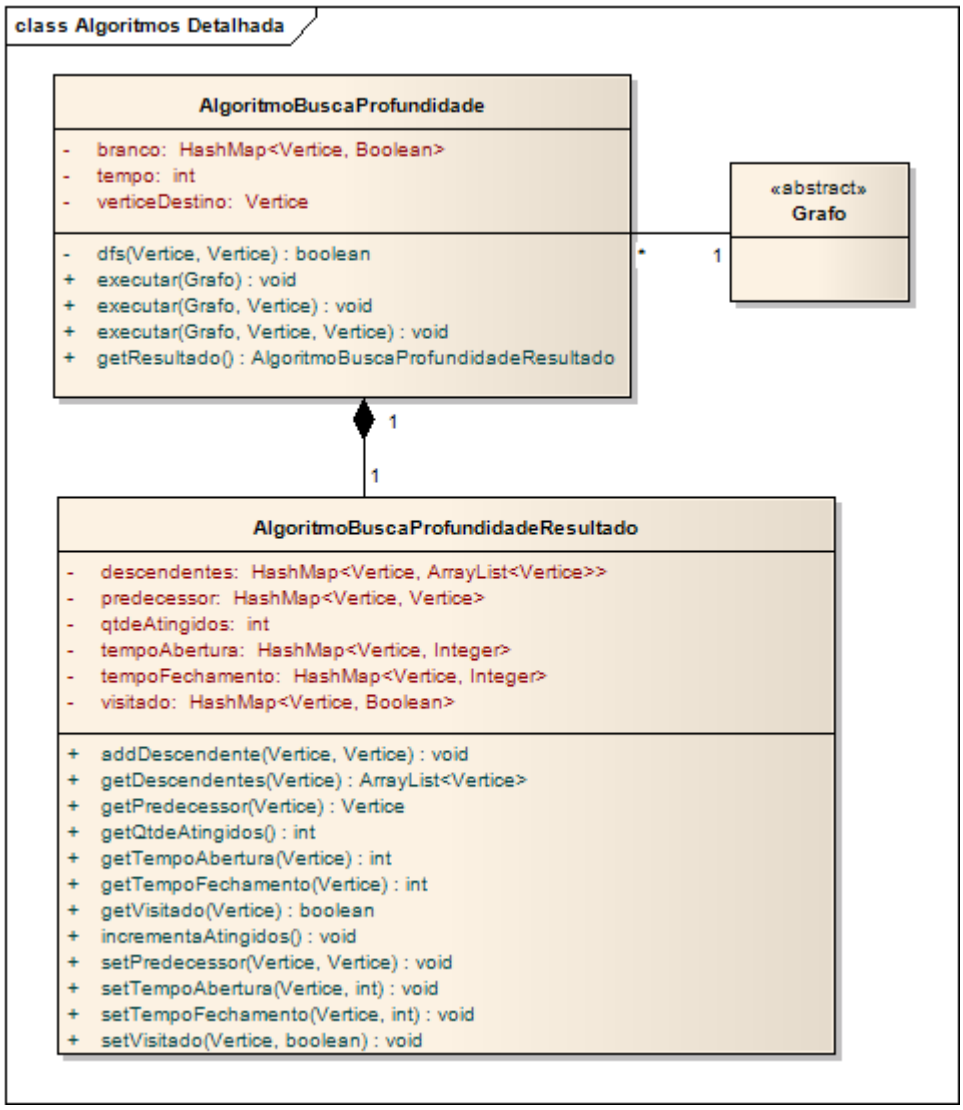


Figura 12 – Diagrama de classes detalhado do pacote `Algoritmos`

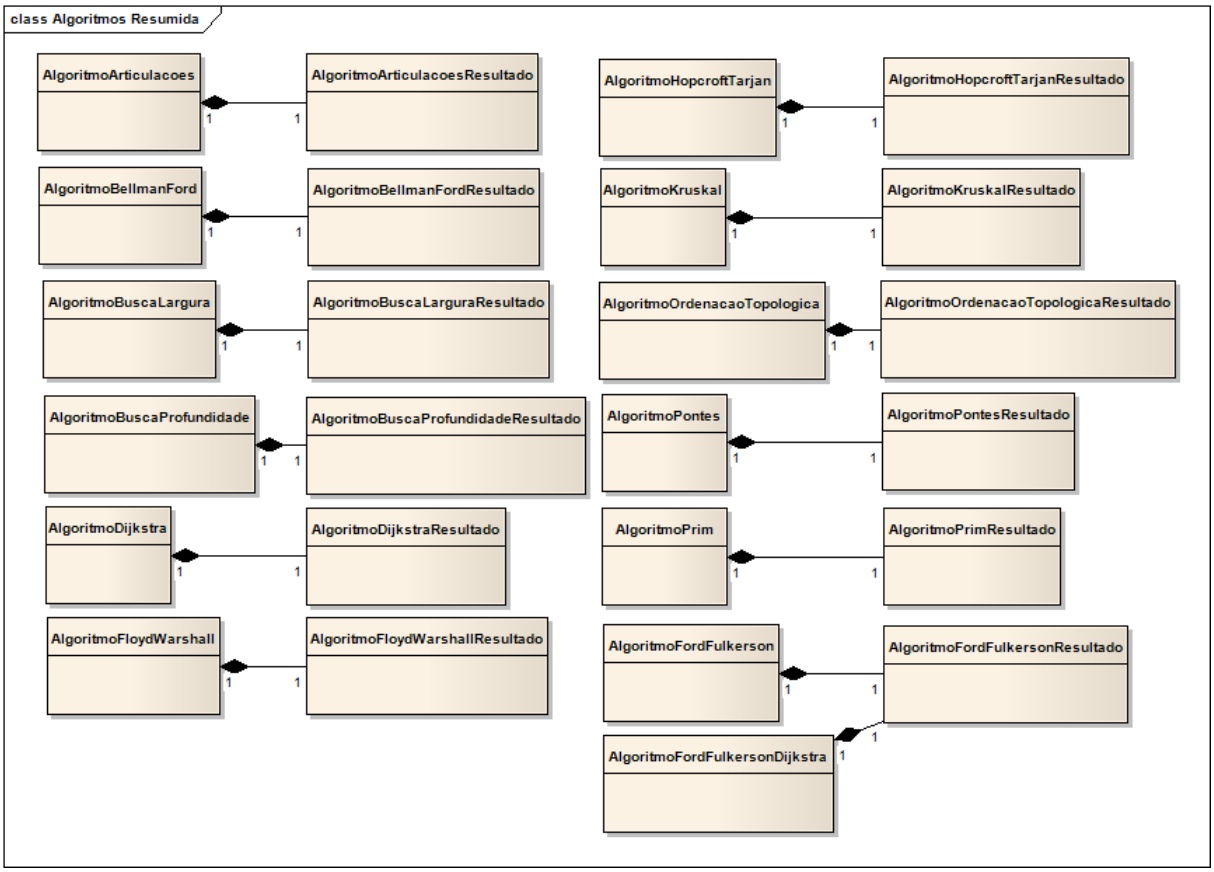


Figura 13 – Diagrama de classes resumido do pacote `Algoritmos`

O terceiro pacote apresentado é o que contém a classe `Persistencia`, responsável pela gravação e recuperação dos grafos armazenados em arquivo.

A Figura 14 exibe a classe `Persistencia` e os métodos disponibilizados.

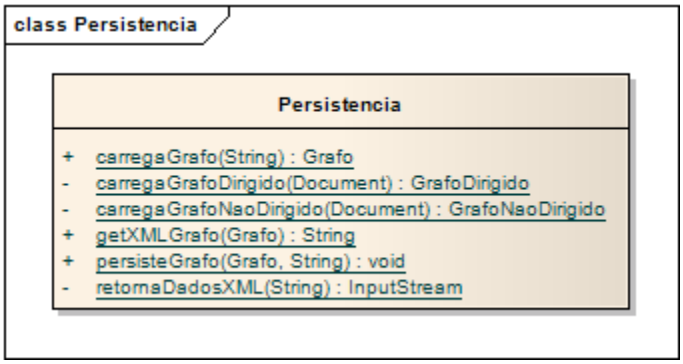


Figura 14 – Diagrama de classes do pacote `Persistencia`

O quarto pacote apresentado é o que contém a classe `GeradorGrafos`, responsável pela geração de grafos com base em certas restrições. Para cada tipo possível de grafo a ser gerado são disponibilizados dois métodos: um para gerar um grafo dirigido e outro para gerar um grafo não dirigido.

A Figura 15 exibe a classe `GeradorGrafos` com todos os tipos de grafos disponibilizados no FGA.

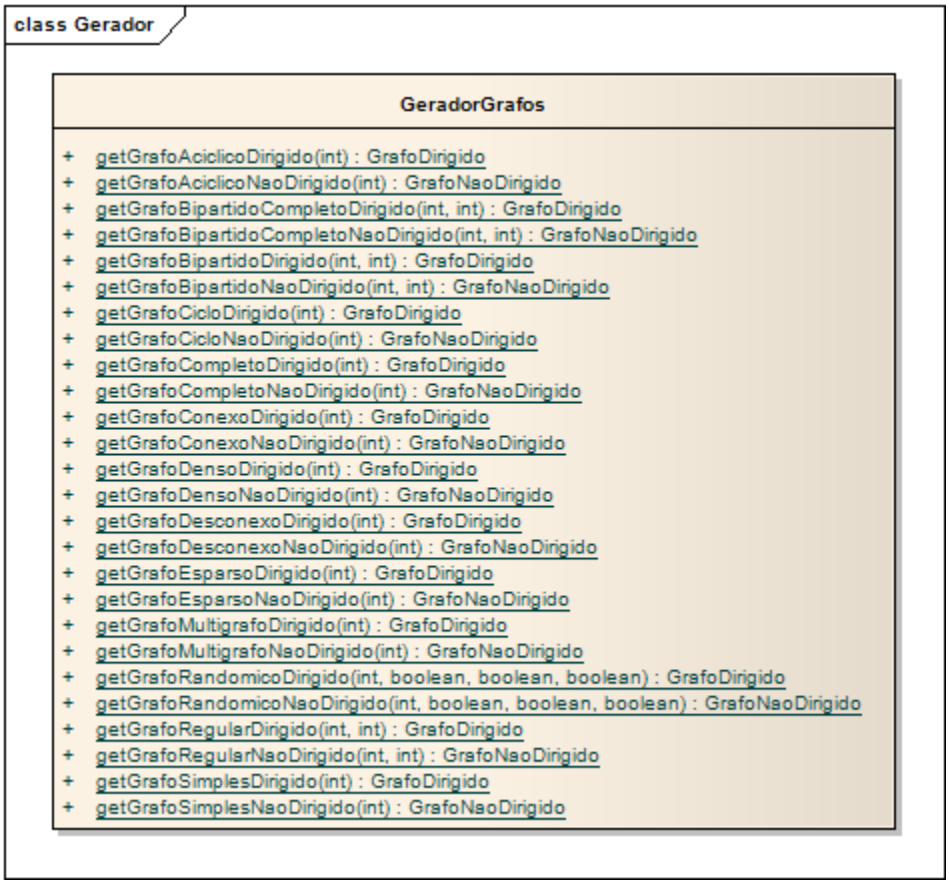


Figura 15 – Diagrama de classes do pacote `GeradorGrafos`

O quinto pacote apresentado é o que contém um conjunto de classes auxiliares para o funcionamento do FGA. A classe `Comandos` é responsável por manter um conjunto de métodos utilizados pelo FGA para operações diversas. O método `salvar` é utilizado para gravar um texto em um arquivo especificado. Este método é chamado pela classe de `Persistencia` para gravar o grafo em um arquivo. Já, os métodos `getDoubleNumeroAleatorio` e `getIntNumeroAleatorio` são utilizados pela classe `GeradorGrafos` para construir um grafo randômico com as especificações do usuário.

A classe `Constante` contém um conjunto de valores extremos para serem utilizados nos algoritmos de grafos. Tais valores são usados para representar o infinito positivo e infinito negativo.

A classe `Pair` é utilizada como uma estrutura de dados para representar um par de objetos. Tal recurso é utilizado principalmente para representar a conexão entre dois vértices.

Por fim, a classe `PairPriority` é utilizada como a estrutura `Pair`, porém implementa a interface `Comparable` e é genérica. Tal mudança foi feita para que seja possível inserir um par de objetos em uma fila de prioridades. A fila de prioridades é uma estrutura de dados utilizada em algoritmos como Dijkstra, Prim e Kruskal. A Figura 16 apresenta as classes do pacote `Auxiliar`.

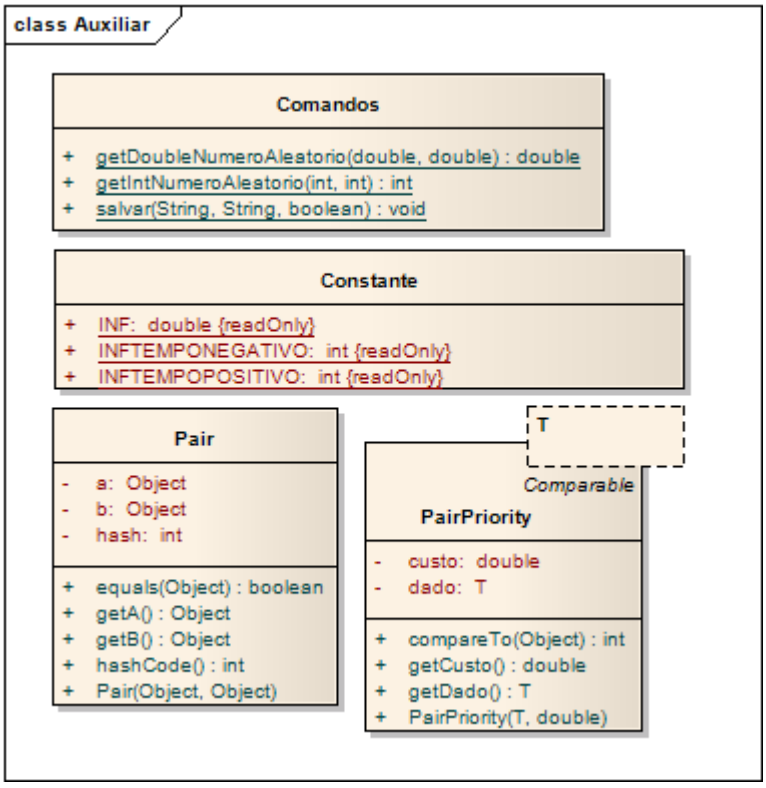


Figura 16 – Diagrama de classes do pacote `Auxiliar`

3.2.3 Diagrama de atividades

O modo de utilizar o FGA é representado pelo diagrama de atividades da Figura 17. Nele são mostradas, de forma resumida, diversas operações que podem ser feitas utilizando o FGA.

O diagrama inicia com a instanciação de um grafo. Esta instanciação pode ser feita de três formas diferentes: criando o grafo manualmente, ou seja, adicionando vértices e arestas, gerando o grafo de forma automática, informando algumas restrições, e por fim carregando um grafo previamente definido em um arquivo XML.

Depois de ter o grafo instanciado são mostradas as operações possíveis a serem feitas. Dentre elas está a persistência do grafo em um arquivo, a execução de algum algoritmo e a verificação de alguma propriedade.

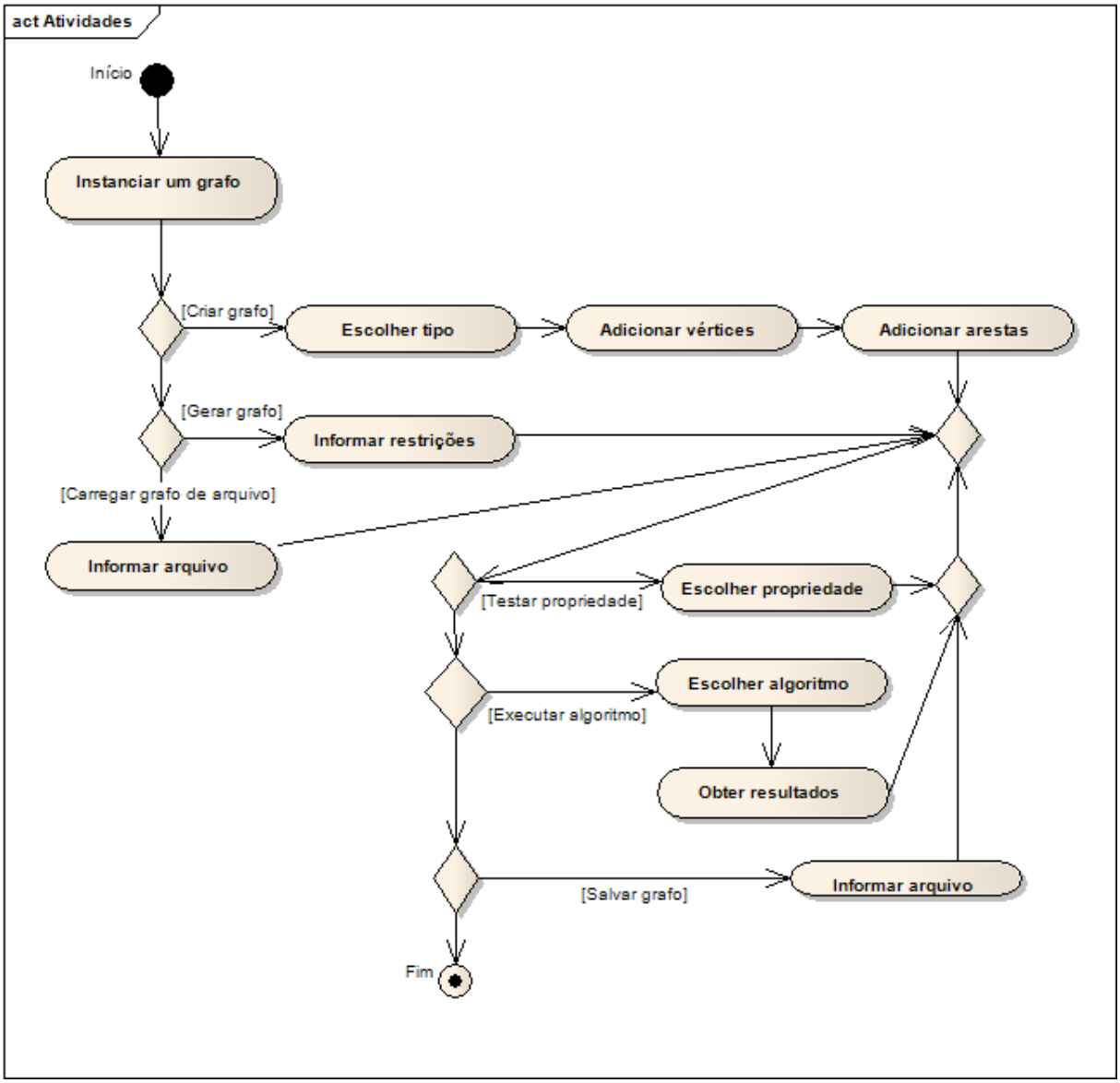


Figura 17 – Diagrama de atividades

3.2.4 Diagrama de sequência

O diagrama de sequência mostra a sequência de interações entre os diversos componentes do FGA em um cenário específico. Para demonstrar a troca de mensagens existentes no FGA, desde a instanciação de um grafo até a execução de alguma operação sobre ele, foram criados três diagramas de sequência.

O primeiro diagrama (Figura 18) mostra a sequência de mensagens trocadas para que seja possível salvar um grafo. Em linhas gerais, inicialmente é instanciado um grafo e por fim o mesmo é salvo em um arquivo.

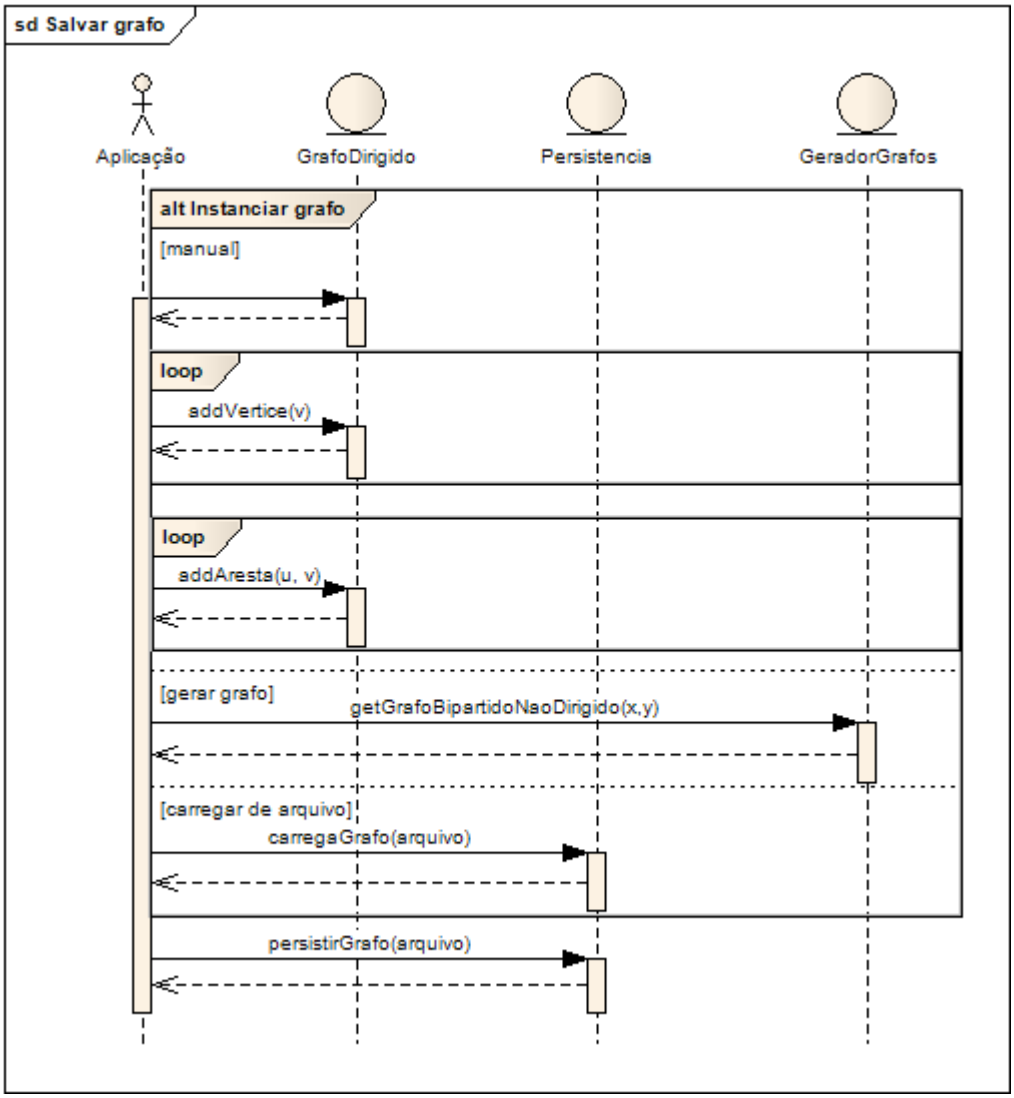


Figura 18 – Diagrama de sequência da operação salvar grafo

O segundo diagrama de sequência (Figura 19) apresenta a operação de executar algum algoritmo do FGA. Para exemplificar é mostrada a execução do algoritmo de Dijkstra e do algoritmo de Prim no mesmo diagrama. Nota-se que, também para a execução do algoritmo, é necessário primeiramente instanciar um grafo.

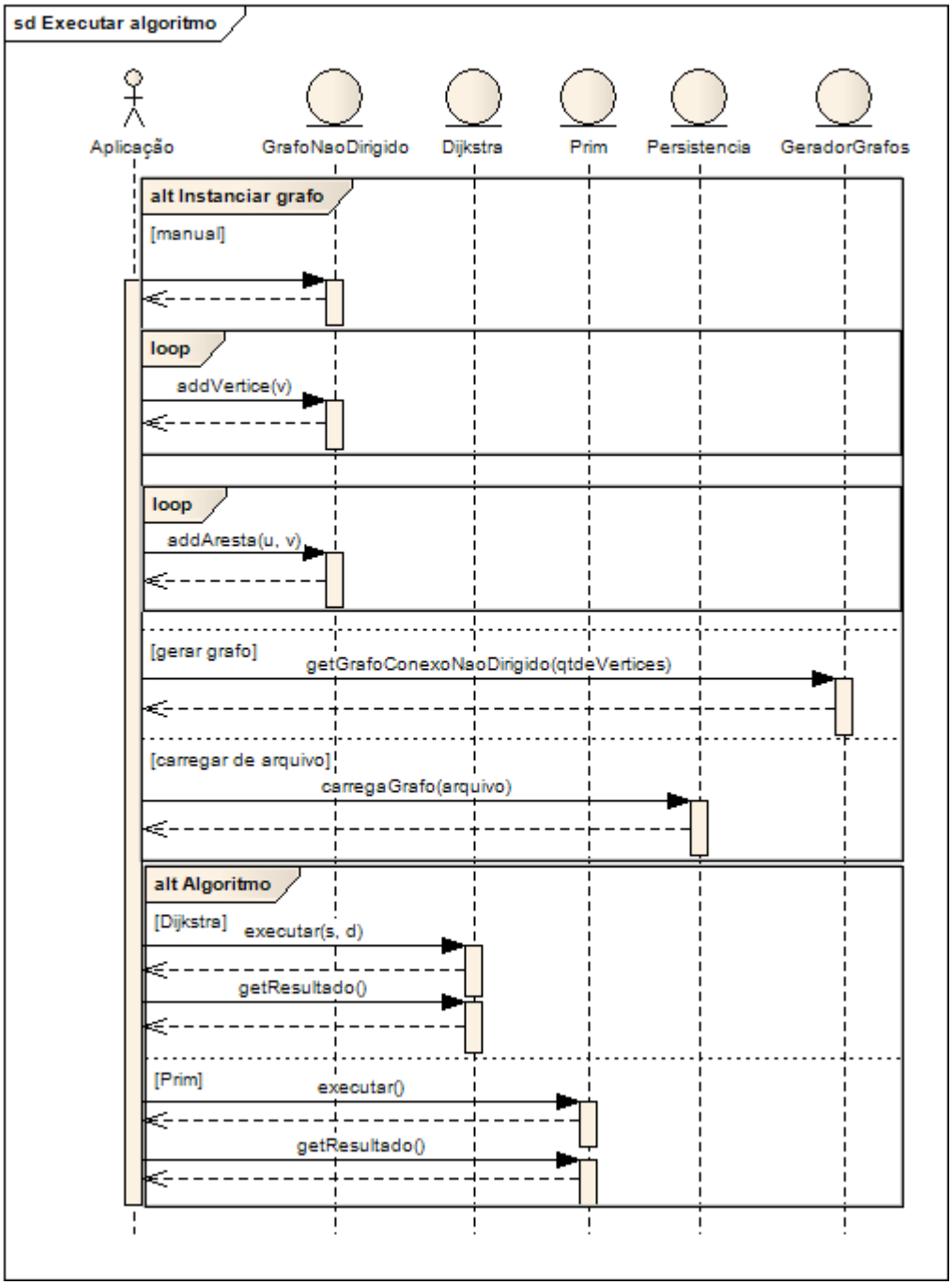


Figura 19 – Diagrama de sequência da operação executar algoritmo

O terceiro diagrama de seqüência (Figura 20) apresenta a operação de testar alguma propriedade de um grafo. Como o teste de qualquer propriedade de um grafo é feito da mesma forma, têm-se como exemplos o teste de conexidade, a verificação se o grafo é acíclico e o teste se o grafo é simples. Neste novo cenário também é possível observar que o grafo precisa ser instanciado.

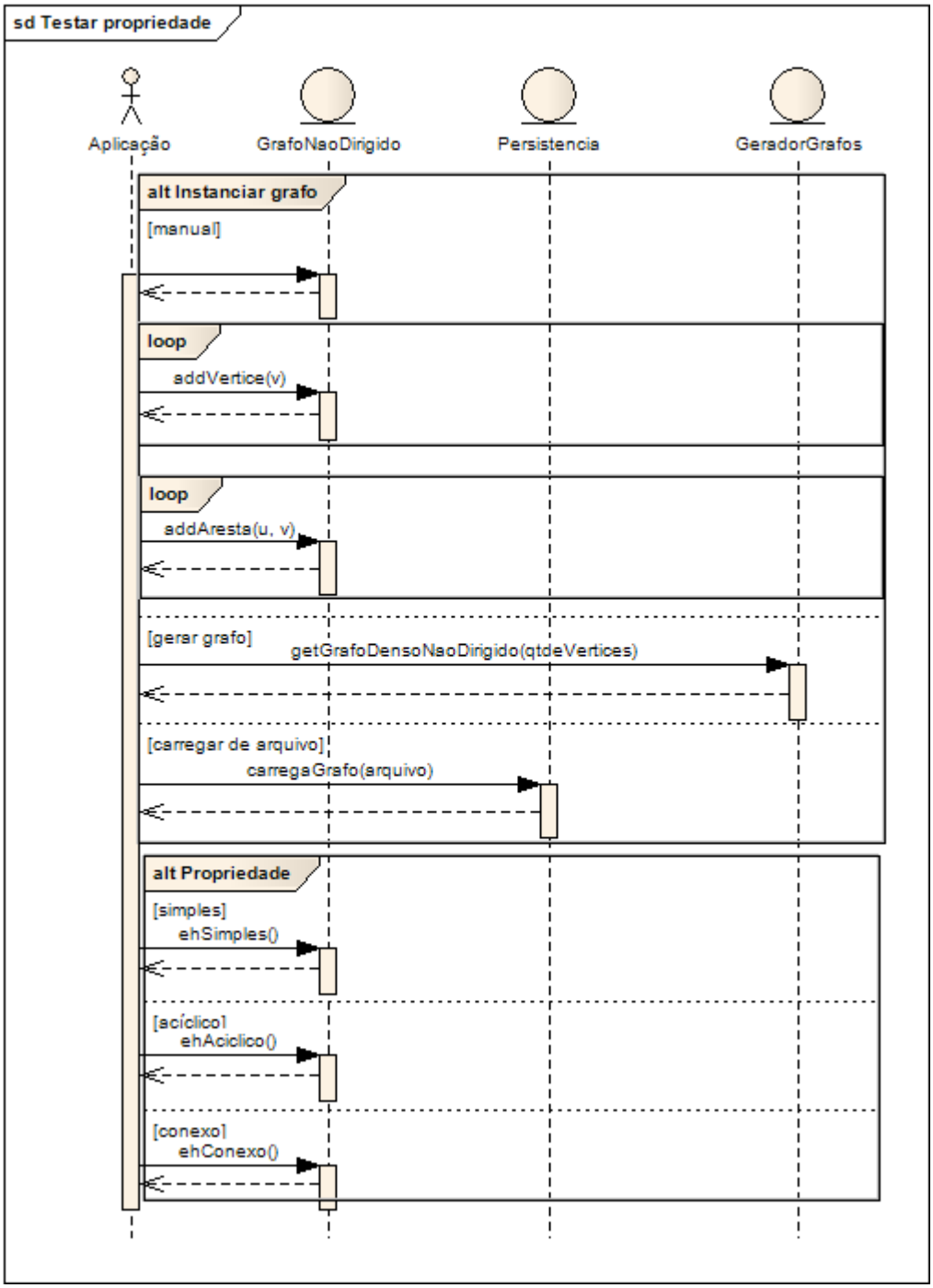


Figura 20 – Diagrama de sequência da operação testar propriedade

3.3 IMPLEMENTAÇÃO

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

3.3.1 Técnicas e ferramentas utilizadas

A implementação do FGA e aplicação de teste foram feitas utilizando a linguagem de programação Java versão 6, a qual permite a portabilidade do FGA para diversas plataformas que suportam a linguagem. Durante o desenvolvimento também foi utilizada a biblioteca *Document Object Model* (DOM) para criação e leitura de arquivos *eXtensible Markup Language* (XML).

Para a codificação do FGA, bem como para a aplicação de teste, foi utilizado o ambiente de programação Fedora Eclipse 3.4.1. Por fim, para gerar a documentação do FGA, foi utilizado o Javadoc.

3.3.1.1 DOM

O *Document Object Model* (DOM) é um padrão desenvolvido pela *World Wide Web Consortium* (W3C) e define uma forma padrão para criar, acessar e manipular documentos XML. O DOM define os objetos e propriedades de todos os elementos do documento, e os métodos para acessá-los. Para trabalhar com os arquivos XML o DOM representa os documentos como uma estrutura baseada em árvores (WORLD WIDE WEB CONSORTIUM, 2010). O uso da biblioteca DOM, no FGA, permite disponibilização de funções para persistir e carregar grafos de arquivos.

3.3.1.2 Javadoc

O Javadoc é uma ferramenta que analisa as declarações e os comentários de documentação em um conjunto de arquivos-fonte e produz um conjunto de páginas *HyperText Markup Language* (HTML), descrevendo as classes, interfaces, construtores, métodos e campos (ORACLE TEAM, 2005).

O Quadro 19 mostra um exemplo de código com comentários para o Javadoc.

```
/**
 * Carrega um grafo a partir de um arquivo
 * @param arquivo Arquivo onde está o grafo
 * @return Grafo
 * @throws ParserConfigurationException
 * @throws FileNotFoundException
 * @throws SAXException
 * @throws IOException
 */
public static Grafo carregaGrafo(String arquivo) throws
ParserConfigurationException, FileNotFoundException, SAXException,
IOException {...}
```

Quadro 19 – Exemplo de código com comentários para o Javadoc

3.3.2 Desenvolvimento do FGA

O desenvolvimento do trabalho foi dividido em cinco etapas principais, a implementação da estrutura do grafo, o desenvolvimento da persistência, a implementação dos geradores de grafos, o desenvolvimento dos algoritmos e por fim a implementação da aplicação de exemplo. A seguir é descrita a implementação de cada uma das etapas.

3.3.2.1 Desenvolvimento da estrutura do grafo

A estrutura básica do grafo é responsável por manter o conjunto de vértices e os identificadores das arestas existentes. A partir desta estrutura básica é possível que a mesma seja estendida tanto para um grafo dirigido como para um grafo não dirigido.

Como boa parte dos métodos e atributos de um grafo dirigido e um grafo não dirigido são os mesmos, estes dados acabam ficando como responsabilidade da classe `Grafo`, deixando as implementações específicas para as classes `GrafoDirigido` e `GrafoNaoDirigido`. Parte do código da classe `Grafo` é exibida no Quadro 20.

```
public abstract class Grafo {
    protected ArrayList<Vertice> vertices;
    private HashMap<Pair, Boolean> adjacente;
    protected HashMap<Integer, Vertice> idVertices;
    protected HashMap<Integer, Aresta> idArestas;

    /**
     * Retorna se existe uma conexão u -> v
     * @param u Vértice de origem
     * @param v Vértice de destino
     * @return true ou false
     */
    public boolean ehAdjacente(Vertice u, Vertice v) {
        Pair par = new Pair(u, v);
        return adjacente.get(par) != null;
    }

    /**
     * Retorna se o grafo é conexo.<br>
     * Um grafo é conexo se for possível visitar qualquer vértice,
partindo de um outro e passando por arestas.
     * @return true ou false
     */
    Public abstract Boolean ehConexo();
```

Quadro 20 – Classe `Grafo`

Os métodos abstratos existentes na classe `Grafo` são métodos que necessitam de implementações específicas para grafos dirigidos e não dirigidos. Tais implementações estão nas classes `GrafoDirigido` e `GrafoNaoDirigido` que estendem a classe `Grafo`.

Para um melhor desempenho na busca da existência de conexões entre dois vértices foi criado o atributo `adjacente`, o qual é utilizado para verificar se existe alguma conexão entre um vértice de origem e outro de destino. Esta informação pode ser obtida chamando o método `ehAdjacente` passando como argumentos os vértices de origem e destino.

Outra característica importante desta classe é a forma de obter vértices e arestas de uma forma rápida. Para que seja possível tal desempenho foi necessária a criação de um identificador único para cada vértice e para cada aresta. Este valor é um número inteiro e é feito um mapeamento com as classes `Vertice` e `Aresta`.

A classe `GrafoDirigido` implementa os métodos específicos para os grafos dirigidos. Parte do seu código é apresentado no Quadro 21. Já a classe `GrafoNaoDirigido` é responsável pelos métodos com implementação específica para grafos não dirigidos, sendo que um trecho do seu código é exposto no Quadro 22.

O conjunto de arestas do grafo também é mantido nas classes `GrafoDirigido` e `GrafoNaoDirigido`, já que as arestas podem ser dirigidas ou não dirigidas.

```
public class GrafoDirigido extends Grafo {
    private ArrayList<ArestaDirigida> arestas;
    /**
     * Retorna se o grafo é conexo.<br>
     * Um grafo é conexo se for possível visitar qualquer vértice,
partindo de um outro e passando por arestas.<br>
     * Em grafos dirigidos é feito o teste se o grafo é fortemente conexo,
ou seja, possui apenas uma componente<br>
     * fortemente conexa.
     * @return true ou false
     */
    public boolean ehConexo() {
        AlgoritmoBuscaLargura alg = new AlgoritmoBuscaLargura();
        alg.executar(this, getUmVertice());

        AlgoritmoBuscaLarguraResultado res = alg.getResultado();

        int tamanhoGrafo = getTamanho();
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = getVertice(i);
            if (res.getVisitado(v) != true) return false;
        }

        //executa no grafo transposto
        Grafo g = getTransposto();

        alg = new AlgoritmoBuscaLargura();
        alg.executar(g, g.getUmVertice());

        res = alg.getResultado();
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = g.getVertice(i);
            if (res.getVisitado(v) != true) return false;
        }
        return true;
    }
```

Quadro 21 – Classe `GrafoDirigido`

```
public class GrafoNaoDirigido extends Grafo {
    private ArrayList<ArestaNaoDirigida> arestas;
    /**
     * Retorna se o grafo é conexo.<br>
     * Um grafo é conexo se for possível visitar qualquer vértice,
partindo de um outro e passando por arestas.
     * @return true ou false
     */
    public boolean ehConexo() {
        AlgoritmoBuscaLargura alg = new AlgoritmoBuscaLargura();
        alg.executar(this, getUmVertice());

        AlgoritmoBuscaLarguraResultado res = alg.getResultado();

        int tamanhoGrafo = getTamanho();
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = getVertice(i);
            if (res.getVisitado(v) != true) return false;
        }
        return true;
    }
```

Quadro 22 – Classe `GrafoNaoDirigido`

Todas as verificações das propriedades do grafo têm seus métodos chamados a partir da classe `Grafo`. Algumas propriedades necessitam de cálculos diferenciados para grafos dirigidos e não dirigidos, portanto tais métodos são abstratos na classe `Grafo`. O Quadro 21 e o Quadro 22 exemplificam a propriedade que diz respeito a conexidade de um grafo, enquanto que o Quadro 23 mostra exemplos de métodos que testam outras propriedades.

```
    /**
     * Retorna se o grafo é trivial.<br>
     * Um grafo é trivial se contém apenas um vértice e nenhuma aresta.
     * @return true ou false
     */
    public boolean ehTrivial() {
        return getTamanho() == 1 && getQtdeArestas() == 0;
    }

    /**
     * Retorna se o grafo é nulo.<br>
     * Um grafo é nulo se o conjunto de arestas é vazio.
     * @return true ou false
     */
    public boolean ehNulo() {
        return getQtdeArestas() == 0;
    }

    /**
     * Retorna se o grafo é denso.<br>
     * Um grafo é denso se o total de arestas é maior ou igual ao quadrado
da quantidade de vértices.
     * @return true ou false
     */
    public boolean ehDenso() {
        int m = getQtdeArestas();
        int n = getTamanho();

        return m >= (n * n);
    }

    /**
     * Retorna se o grafo é esparso.<br>
     * Um grafo é esparso se o total de arestas é menor que o quadrado da
quantidade de vértices.
     * @return true ou false
     */
    public boolean ehEsparso() {
        return !ehDenso();
    }

    /**
     * Retorna se o grafo é simples.<br>
     * Um grafo é simples se não possui arestas paralelas e laços.
     * @return true ou false
     */
    public boolean ehSimples() {
        return !ehMultigrafo();
    }
```

Quadro 23 – Métodos que testam propriedades de grafos

Um grafo é constituído por vértices e arestas. Da mesma forma que as classes

relacionadas com as características do grafo, são implementadas as classes para trabalhar com as arestas.

A classe abstrata `Aresta` disponibiliza os métodos comuns para as arestas dirigidas e não dirigidas. Uma parte do seu código é exibida no Quadro 24.

```
public abstract class Aresta {
    private int id;
    private Object dado;
    private double valor;
    private double capacidade;
    private Vertice vi;
    private Vertice vj;
    private Grafo g;

    /**
     * Retorna o dado da aresta
     * @return dado
     */
    public Object getDado() {
        return dado;
    }

    /**
     * Retorna o valor/custo da aresta
     * @return valor
     */
    public double getValor() {
        return valor;
    }
```

Quadro 24 – Classe `Aresta`

Os métodos `getVi` e `getVj`, que são utilizados para retornar os vértices que formam uma determinada aresta, são escritos nas classes `ArestaDirigida` e `ArestaNaoDirigida`. Um trecho da classe `ArestaDirigida` é exibido no Quadro 25, enquanto que parte do código da classe `ArestaNaoDirigida` é mostrado no Quadro 26.

```
public class ArestaDirigida extends Aresta {
    /**
     * Retorna o vértice de origem da aresta
     * @return vi
     */
    public Vertice getVi() {
        return vi;
    }

    /**
     * Retorna o vértice de destino da aresta
     * @return vj
     */
    public Vertice getVj() {
        return vj;
    }
```

Quadro 25 – Classe `ArestaDirigida`

```
public class ArestaNaoDirigida extends Aresta {
    /**
     * Retorna o vértice de origem da aresta
     * @return vi
     */
    public Vertice getVi() {
        return vi;
    }

    /**
     * Retorna o vértice de destino da aresta
     * @param u Um vértice
     * @return vj
     */
    public Vertice getVj(Vertice u) {
        if (vi.equals(u))
            return vj;
        return vi;
    }
```

Quadro 26 – Classe `ArestaNaoDirigida`

Por fim, a classe `Vertice` disponibiliza os métodos para trabalhar com os vértices do grafo. Um dos atributos da classe é a lista de arestas em que o vértice é o ponto de partida. A partir deste atributo é possível descobrir os vértices adjacentes sem que seja necessário percorrer todo o grafo. Um trecho do seu código é exposto no Quadro 27.

```
public abstract class Vértice {
    private int id;
    private Object dado;
    private Color cor;
    private ArrayList<Aresta> arestas;
    private HashMap<Integer, Aresta> idArestas;

    /**
     * Adiciona uma aresta ao vértice
     * @param a Aresta
     */
    protected void addAresta(Aresta a) {
        idArestas.put(a.getId(), a);
        arestas.add(a);
    }

    /**
     * Retorna o grau do vértice
     * @return grau
     */
    public int getGrau() {
        return arestas.size();
    }
```

Quadro 27 – Classe `Vertice`

3.3.2.2 Implementação da persistência de grafos

Para a gravação e recuperação de grafos foi optado pelo formato de arquivo XML, pois é um formato aberto e de implementação relativamente simples.

A estrutura do formato do grafo no arquivo requer que seja informado se o tipo grafo que está sendo armazenado é dirigido ou não dirigido. O campo responsável por este controle é o `Tipo`. Se nele contiver o valor `1` então o grafo é dirigido, enquanto que, se contiver o valor `2` será um grafo não dirigido. Informado o tipo do grafo segue-se a lista de vértices e arestas.

Para os vértices estão disponíveis um campo obrigatório para informar o identificador, chamado `Id`, outro campo chamado `Cor` para a cor do vértice, a qual está no formado *Red Green Blue* (RGB), e por fim um campo chamado `Dado`, responsável por armazenar um objeto qualquer que pode estar vinculado ao vértice.

Para as arestas é necessário informar um identificador, representado pelo campo `Id`, um identificador de um vértice de origem para o campo `Vi` e outro identificador de um vértice de destino para o campo `Vj`. Além destes campos estão disponíveis um campo chamado `Valor`, do tipo numérico com ponto flutuante, que representa o custo ou peso de uma aresta, outro campo chamado `Capacidade`, também do tipo numérico com ponto flutuante, responsável pelo fluxo de dados que pode ser transmitido pela aresta e por fim, um campo chamado `Dado`, que pode conter um objeto qualquer para estar vinculado na aresta.

Um exemplo de arquivo que demonstra uma configuração de um grafo utilizando este formato pode ser observado no Quadro 28.

```
<Grafo>
	<Tipo>1</Tipo>
	<Vertices>
		<Vertice>
			<Id>1</Id>
			<Cor><R>255</R><G>0</G><B>0</B></Cor>
			<Dado>A</Dado>
		</Vertice>
		<Vertice>
			<Id>2</Id>
			<Cor><R>0</R><G>255</G><B>0</B></Cor>
			<Dado>B</Dado>
		</Vertice>
		<Vertice>
			<Id>3</Id>
			<Cor><R>0</R><G>0</G><B>255</B></Cor>
			<Dado>C</Dado>
		</Vertice>
	</Vertices>
	<Arestas>
		<Aresta>
			<Id>1</Id>
			<Vi>1</Vi>
			<Vj>2</Vj>
			<Valor>1442.44</Valor>
			<Capacidade>6.3</Capacidade>
			<Dado>Trilha 1</Dado>
		</Aresta>
		<Aresta>
			<Id>2</Id>
			<Vi>1</Vi>
			<Vj>3</Vj>
			<Valor>13.45</Valor>
			<Capacidade>5.0</Capacidade>
			<Dado>Trilha 2</Dado>
		</Aresta>
	</Arestas>
</Grafo>
```

Quadro 28 – Exemplo de um arquivo com um grafo

O FGA utiliza a biblioteca DOM para trabalhar com XML, sendo que a interação com este formato de arquivo ocorre somente por meio da classe `Persistencia`. Através do método `carregaGrafo` é possível passar um endereço de arquivo como parâmetro e obter como retorno o grafo nele persistido. Neste método é feito o teste do tipo de grafo e é chamada uma função auxiliar `carregaGrafoDirigido` para o caso de um grafo dirigido ou a função `carregaGrafoNaoDirigido` para um grafo não dirigido.

O método `carregaGrafo` tem seu código exposto no Quadro 29, enquanto que um trecho do método `carregaGrafoDirigido` está no Quadro 30 e, um trecho do método `carregaGrafoNaoDirigido` no Quadro 31.

```
/**
 * Carrega um grafo a partir de um arquivo
 * @param arquivo Arquivo onde está o grafo
 * @return Grafo
 * @throws ParserConfigurationException
 * @throws FileNotFoundException
 * @throws SAXException
 * @throws IOException
 */
public static Grafo carregaGrafo(String arquivo) throws
ParserConfigurationException, FileNotFoundException, SAXException,
IOException {
    DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();
    DocumentBuilder construtor = fabrica.newDocumentBuilder();
    Document documento = construtor.parse(retornaDadosXML(arquivo));

    Element elementoGrafo =
      (Element) documento.getElementsByTagName("Grafo").item(0);
    Node nodeTipo =
      elementoGrafo.getElementsByTagName("Tipo").item(0).getFirstChild();

    if (nodeTipo.getNodeValue().equals("1"))
        return carregaGrafoDirigido(documento);
    else
        return carregaGrafoNaoDirigido(documento);
}
```

Quadro 29 – Método `carregaGrafo`

```
private static GrafoDirigido carregaGrafoDirigido(Document documento) {
    GrafoDirigido g = new GrafoDirigido();
    Element elementoGrafo = (Element)
      documento.getElementsByTagName("Grafo").item(0);

    //Vertices
    Element elementoVertices = (Element)
      elementoGrafo.getElementsByTagName("Vertices").item(0);
    NodeList nodelistVertices =
      elementoVertices.getElementsByTagName("Vertice");

    for (int i = 0; i < nodelistVertices.getLength(); i++) {
        {...}

        Vertice v = new
          Vertice(Integer.parseInt(nodeId.getNodeValue()));
        v.setCor(new Color(Integer.valueOf(nodeR.getNodeValue()),
                           Integer.valueOf(nodeG.getNodeValue()),
                           Integer.valueOf(nodeB.getNodeValue())));
        v.setDado(nodeDado.getNodeValue());
        g.addVertice(v);
    }

    //Arestas
    Element elementoArestas = (Element)
      elementoGrafo.getElementsByTagName("Arestas").item(0);
    NodeList nodelistArestas =
      elementoArestas.getElementsByTagName("Aresta");

    for (int i = 0; i < nodelistArestas.getLength(); i++) {
        {...}

        Vertice u =
          g.getVerticeById(Integer.valueOf(nodeVi.getNodeValue()));
        Vertice v =
          g.getVerticeById(Integer.valueOf(nodeVj.getNodeValue()));

        ArestaDirigida a = new
          ArestaDirigida(Integer.valueOf(nodeId.getNodeValue()), u, v);
        a.setValor(Double.valueOf(nodeValor.getNodeValue()));
        a.setCapacidade(Double.valueOf(nodeCapacidade.getNodeValue()));
        a.setDado(nodeDado.getNodeValue());
        g.addAresta(a);
    }
    return g;
}
```

Quadro 30 – Método `carregaGrafoDirigido`

```
private static GrafoNaoDirigido carregaGrafoNaoDirigido(Document
documento) {
    GrafoNaoDirigido g = new GrafoNaoDirigido();
    Element elementoGrafo = (Element)
      documento.getElementsByTagName("Grafo").item(0);

    //Vertices
    Element elementoVertices = (Element)
      elementoGrafo.getElementsByTagName("Vertices").item(0);
    NodeList nodelistVertices =
      elementoVertices.getElementsByTagName("Vertice");

    for (int i = 0; i < nodelistVertices.getLength(); i++) {
        {...}

        Vertice v = new
          Vertice(Integer.parseInt(nodeId.getNodeValue()));
        v.setCor(new Color(Integer.valueOf(nodeR.getNodeValue()),
                           Integer.valueOf(nodeG.getNodeValue()),
                           Integer.valueOf(nodeB.getNodeValue())));
        v.setDado(nodeDado.getNodeValue());
        g.addVertice(v);
    }

    //Arestas
    Element elementoArestas = (Element)
      elementoGrafo.getElementsByTagName("Arestas").item(0);
    NodeList nodelistArestas =
      elementoArestas.getElementsByTagName("Aresta");

    for (int i = 0; i < nodelistArestas.getLength(); i++) {
        {...}

        Vertice u =
          g.getVerticeById(Integer.valueOf(nodeVi.getNodeValue()));
        Vertice v =
          g.getVerticeById(Integer.valueOf(nodeVj.getNodeValue()));

        ArestaNaoDirigida a = new
          ArestaNaoDirigida(Integer.valueOf(nodeId.getNodeValue()), u, v);
        a.setValor(Double.valueOf(nodeValor.getNodeValue()));
        a.setCapacidade(Double.valueOf(nodeCapacidade.getNodeValue()));
        a.setDado(nodeDado.getNodeValue());
        g.addAresta(a);
    }
    return g;
}
```

Quadro 31 – Método `carregaGrafoNaoDirigido`

O FGA também disponibiliza os métodos `geraXMLGrafo, que` é responsável por converter um objeto da classe `Grafo` em formato XML, e `persisteGrafo` que faz a gravação do XML em um arquivo informado.

No Quadro 32 está mostrado o código do método `persisteGrafo`, enquanto que no Quadro 33 é exibida parte do código do método `geraXMLGrafo`.

```
/**
 * Persiste uma grafo em arquivo
 * @param g Grafo
 * @param arquivo Arquivo de destino
 * @throws IOException
 */
public static void persisteGrafo(Grafo g, String arquivo) throws
IOException {
    String texto = getXMLGrafo(g);
    Comandos.salvar(arquivo, texto, false);
}
```

Quadro 32 – Método `persisteGrafo`

```
/**
 * Retorna uma String com o grafo convertido em XML
 * @param g Grafo
 * @return String XML que representa o grafo
 */
public static String getXMLGrafo(Grafo g) {
    DocumentBuilderFactory fabrica = DocumentBuilderFactory.newInstance();
    DocumentBuilder construtor = null;
    try {
        construtor = fabrica.newDocumentBuilder();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    }
    Document documento = construtor.newDocument();

    Element elementoGrafo = documento.createElement("Grafo");
    documento.appendChild(elementoGrafo);

    Element elementoTipo = documento.createElement("Tipo");
    Text nodeTipo = documento.createTextNode(String.valueOf(g.getTipo()));
    elementoTipo.appendChild(nodeTipo);
    elementoGrafo.appendChild(elementoTipo);

    //Percorre vertices e arestas
    {...}

    //Gera xml
    try {
        trans = transfac.newTransformer();
    } catch (TransformerConfigurationException e) {}
    trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
    trans.setOutputProperty(OutputKeys.INDENT, "yes");
    StringWriter sw = new StringWriter();
    StreamResult result = new StreamResult(sw);
    DOMSource source = new DOMSource(documento);
    try {
        trans.transform(source, result);
    } catch (TransformerException e) {}
    return sw.toString();
}
```

Quadro 33 – Método `geraXMLGrafo`

3.3.2.3 Desenvolvimento dos algoritmos de grafos

Para cada algoritmo desenvolvido existem duas classes que disponibilizam métodos para serem trabalhados. Uma classe é específica para a execução do algoritmo, enquanto a outra é o resultado da sua execução. A classe de resultado armazena todos os dados importantes obtidos durante a execução do algoritmo escolhido.

Toda classe de execução de algum algoritmo possui dois métodos principais sendo eles o `executa`, ao qual cabe a tarefa de executar o algoritmo sobre o grafo passado como parâmetro, e `getResultado`, responsável por retornar o resultado da execução do algoritmo.

Uma mesma classe de algoritmo pode conter mais de um método `executa`, ou seja, é permitido que haja uma maneira de executar o algoritmo utilizando parâmetros diferentes. Um exemplo para o caso mencionado é o algoritmo de busca em profundidade. Nele é possível chamar a execução da busca apenas passando um grafo como informação, sendo que para todo vértice não visitado uma nova execução do algoritmo é feita, ou também é válido informar qual é o vértice de origem para que seja feita a busca. Neste caso o algoritmo pára quando não há mais nenhum vértice possível de ser atingido a partir do vértice de origem.

No Quadro 34 é apresentada parte do código da classe `AlgoritmoBuscaProfundidade`, que contém a implementação dos métodos `executa` e `getResultado` para o algoritmo de busca em profundidade. Já, no Quadro 35 é mostrado um trecho do código fonte da classe `AlgoritmoBuscaProfundidadeResultado`, que disponibiliza métodos para acessar as conclusões obtidas pelo algoritmo.

```
public class AlgoritmoBuscaProfundidade {
    private AlgoritmoBuscaProfundidadeResultado resultado;

    /**
     * Retorna o resultado do algoritmo
     * @return AlgoritmoBuscaProfundidadeResultado
     */
    public AlgoritmoBuscaProfundidadeResultado getResultado() {
        return resultado;
    }

    private void dfs(Vertice pred, Vertice u) {
        branco.put(u, false);
        resultado.setTempoAbertura(u, tempo);
        resultado.addDescendente(pred, u);
        resultado.setPredecessor(u, pred);
        resultado.setVisitado(u, true);
        resultado.incrementaAtingidos();
        tempo++;

        ArrayList<Vertice> adjU = g.getAdjacentes(u);
        for (Vertice v: adjU) {
            if (branco.get(v))
                dfs(u, v)
        }
        resultado.setTempoFechamento(u, tempo);
        tempo++;
    }

    /**
     * Método que executa o algoritmo sobre um grafo
     * @param g Grafo
     */
    public void executar(Grafo g) {
        {...}
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = g.getVertice(i);
            if (branco.get(v))
                dfs(null, v);
        }
    }

    /**
     * Método que executa o algoritmo sobre um grafo a partir de um
vértice de origem
     * @param g Grafo
     * @param s Vértice de origem
     */
    public void executar(Grafo g, Vertice s) {
        {...}
        dfs(null, v);
    }
```

Quadro 34 – Classe `AlgoritmoBuscaProfundidade`

À medida que o algoritmo de busca em profundidade é executado há uma sequência de interações com a classe `AlgoritmoBuscaProfundidadeResultado`, que é notificada com os eventos ocorridos durante a execução. No exemplo da classe `AlgoritmoBuscaProfundidade` estas interações podem ser vistas no método `dfs`, que invoca os métodos `setTempoAbertura`,

`addDescendente`, `setPredecessor`, `setVisitado` e `setTempoFechamento` da classe resultado. É desta forma que, ao término da execução, o resultado final é construído.

```
public class AlgoritmoBuscaProfundidadeResultado {
    private HashMap<Vertice, Integer> tempoAbertura;
    private HashMap<Vertice, Integer> tempoFechamento;
    private HashMap<Vertice, Vertice> predecessor;
    private HashMap<Vertice, ArrayList<Vertice>> descendentes;
    private HashMap<Vertice, Boolean> visitado;

    /**
     * Atribui um tempo de abertura para o vértice
     * @param v Vértice
     * @param t Tempo de abertura
     */
    public void setTempoAbertura(Vertice v, int t) {
        tempoAbertura.put(v, t);
    }

    /**
     * Retorna o tempo de abertura do vértice
     * @param v Vértice
     * @return Tempo de abertura
     */
    public int getTempoAbertura(Vertice v) {
        return tempoAbertura.get(v);
    }

    /**
     * Atribui um predecessor para um vértice
     * @param v Vértice
     * @param pai Vértice pai
     */
    public void setPredecessor(Vertice v, Vertice pai) {
        predecessor.put(v, pai);
    }

    /**
     * Retorna o predecessor de um vértice
     * @param v Vértice
     * @return Predecessor do vértice
     */
    public Vertice getPredecessor(Vertice v) {
        return predecessor.get(v);
    }
```

Quadro 35 – Classe `AlgoritmoBuscaProfundidadeResultado`

Para o usuário do FGA apenas os métodos da classe ficam disponíveis, sendo que a partir deles é possível construir outras estruturas de dados implícitas nos atributos da classe `AlgoritmoBuscaProfundidadeResultado`. Como exemplo é possível citar uma árvore construída através das leituras dos predecessores de cada vértice. Esta informação pode ser obtida invocando o método `getPredecessor`.

Outro exemplo de algoritmo implementado é o algoritmo de Prim. Parte da classe `AlgoritmoPrim` é mostrada no Quadro 36 enquanto que um trecho da classe `AlgoritmoPrimResultado` é mostrado no Quadro 37.

Para a implementação do algoritmo foi necessária a utilização de uma fila de prioridades, que é responsável por armazenar, em ordem, as distâncias dos vértices para que sejam adicionados na árvore geradora de custo mínimo. Além disso, é utilizada uma `HashMap` para identificar se um vértice já está presente na árvore.

Na classe dos resultados foram utilizadas uma `HashMap` para armazenar os predecessores de cada vértice, outra para a lista de vértices descendentes e outra, cujo objetivo é armazenar o custo até cada vértice. Por fim, uma lista foi usada para manter as arestas que fazem parte da árvore. Todos os demais detalhes da implementação do algoritmo foram feitos conforme descrito em pseudocódigo no Quadro 8.

```
public class AlgoritmoPrim {
    private HashMap<Vertice, Boolean> naArvore;
    private PriorityQueue<PairPriority<Vertice>> fila;

    public void executar(GrafoNaoDirigido g) {
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = g.getVertice(i);
            naArvore.put(v, false);
            resultado.setCusto(v, Constante.INF);
            resultado.setPredecessor(v, null);
        }
        Vertice x = g.getUmVertice();
        fila.offer(new PairPriority<Vertice>(x, 0));
        while (fila.size() > 0) {
            PairPriority<Vertice> p = fila.peek();
            fila.poll();
            Vertice u = p.getDado();
            if (naArvore.get(u)) continue;

            naArvore.put(u, true);
            resultado.addCustoTotal(p.getCusto());
            int qtdeArestas = u.getQtdeArestas();
            for (int i = 0; i < qtdeArestas; i++) {
                ArestaNaoDirigida a = (ArestaNaoDirigida) u.getAresta(i);
                Vertice v = a.getVj(u);
                if (naArvore.get(v) == false && a.getValor() <
resultado.getCusto(v)) {
                    resultado.setCusto(v, a.getValor());
                    resultado.setPredecessor(v, u);
                    fila.offer(new PairPriority<Vertice>(v,
a.getValor()));
                }
            }
        }
        for (int i = 0; i < tamanhoGrafo; i++) {
            Vertice v = g.getVertice(i);
            Vertice pred = resultado.getPredecessor(v);
            if (pred != null) {
                resultado.addDescendente(pred, v);
                resultado.addAresta(g.getArestaCustoMinimo(pred, v));
            }
        }
    }
}
```

Quadro 36 – Classe `AlgoritmoPrim`

```
public class AlgoritmoPrimResultado {
    private HashMap<Vertice, Vertice> predecessor;
    private HashMap<Vertice, ArrayList<Vertice>> descendentes;
    private HashMap<Vertice, Double> custo;
    private ArrayList<Aresta> arestas;
    private double custoTotal = 0;

    /**
     * Atribui um predecessor para um vértice
     * @param v Vértice
     * @param pai Vértice pai
     */
    public void setPredecessor(Vertice v, Vertice pai) {
        predecessor.put(v, pai);
    }

    /**
     * Retorna o predecessor de um vértice
     * @param v Vértice
     * @return Predecessor do vértice
     */
    public Vertice getPredecessor(Vertice v) {
        return predecessor.get(v);
    }
```

Quadro 37 – Classe `AlgoritmoPrimResultado`

3.3.2.4 Desenvolvimento dos geradores de grafos

Outra funcionalidade do FGA é a capacidade de gerar grafos com diversas propriedades especiais. A parte do FGA que fica responsável por isso está na classe `GeradorGrafos`. Nela estão, para cada tipo de grafo que pode ser gerado, um método para gerar um grafo dirigido e outro para gerar um grafo não dirigido. No Quadro 38 é exibido o método `getGrafoCompletoDirigido`, para construir um grafo completo dirigido, enquanto que no Quadro 39 é exposto o método `getGrafoCompletoNaoDirigido` que constrói um grafo completo não dirigido. Nota-se que, pelas características particulares do grafo dirigido e do não dirigido, o algoritmo que os gera é sempre diferente.

```
/**
 * Gera um grafo dirigido completo.<br>
 * Um grafo é completo se ele é simples e existe uma aresta ligando cada
par de vértices distintos.
 * @param qtdeVertices Quantidade de vértices
 * @return GrafoDirigido completo
 */
public static GrafoDirigido getGrafoCompletoDirigido(int qtdeVertices) {
    GrafoDirigido g = new GrafoDirigido();

    for (int i = 1; i <= qtdeVertices; i++) {
        Vertice v = new Vertice (i);
        v.setDado(i);
        g.addVertice(v);
    }

    int cArestas = 1;
    for (int i = 0; i < qtdeVertices; i++) {
        Vertice u = g.getVertice(i);
        for (int j = 0; j < qtdeVertices; j++) {
            if (i == j) continue;

            Vertice v = g.getVertice(j);

            ArestaDirigida a = new ArestaDirigida(cArestas, u, v);
            a.setValor(Comandos.getDoubleNumeroAleatorio(0.0, 100.0));
            a.setDado(cArestas);
            a.setCapacidade(Comandos.getDoubleNumeroAleatorio(0.0,
100.0));

            g.addAresta(a);
            cArestas++;
        }
    }
    return g;
}
```

Quadro 38 – Método `getGrafoCompletoDirigido`

```
/**
 * Gera um grafo não dirigido completo.<br>
 * Um grafo é completo se ele é simples e existe uma aresta ligando cada
par de vértices distintos.
 * @param qtdeVertices Quantidade de vértices
 * @return GrafoNaoDirigido completo
 */
public static GrafoNaoDirigido getGrafoCompletoNaoDirigido(int
qtdeVertices) {
    GrafoNaoDirigido g = new GrafoNaoDirigido();

    for (int i = 1; i <= qtdeVertices; i++) {
        Vertice v = new Vertice (i);
        v.setDado(i);
        g.addVertice(v);
    }

    int cArestas = 1;
    for (int i = 0; i < qtdeVertices; i++) {
        Vertice u = g.getVertice(i);
        for (int j = i + 1; j < qtdeVertices; j++) {
            Vertice v = g.getVertice(j);

            ArestaNaoDirigida a = new ArestaNaoDirigida(cArestas, u, v);
            a.setValor(Comandos.getDoubleNumeroAleatorio(0.0, 100.0));
            a.setDado(cArestas);
            a.setCapacidade(Comandos.getDoubleNumeroAleatorio(0.0,
100.0));

            g.addAresta(a);
            cArestas++;
        }
    }
    return g;
}
```

Quadro 39 – Método `getGrafoCompletoNaoDirigido`

Dependendo do tipo de grafo, dirigido ou não dirigido, e da propriedade que precisa ser atingida, muitas configurações de parâmetros de entrada para os geradores podem ser inválidas.

Não é possível, por exemplo, gerar um grafo regular não dirigido com uma quantidade de vértices ímpar e exigir que o mesmo tenha todos seus vértices com grau também ímpar. Se uma entrada for dada da maneira mencionada o FGA retorna uma exceção, garantindo que não fique tentando gerar um grafo inválido.

Outro exemplo pode ser verificado quando se tenta gerar um grafo desconexo de apenas um vértice. O retorno para esta entrada também é de uma configuração inválida. No Quadro 40 é mostrado o exemplo do tratamento feito ao gerar um grafo regular não dirigido, bem como o tratamento feito ao gerar um grafo desconexo dirigido, nos métodos `getGrafoRegularNaoDirigido` e `getGrafoDesconexoDirigido`, respectivamente.

```
/**
 * Gera um grafo não dirigido regular.<br>
 * Um grafo é regular se todos os vértices possuem o mesmo grau.
 * @param qtdeVertices Quantidade de vértices
 * @param grau Grau dos vértices
 * @return GrafoNaoDirigido regular
 * @throws Exception caso a quantidade de vértices e o grau seja ímpar
 */
public static GrafoNaoDirigido getGrafoRegularNaoDirigido(int
qtdeVertices, int grau) throws Exception {

    if (qtdeVertices % 2 != 0 && grau % 2 != 0)
        throw new Exception("Nao e possivel gerar um grafo dessa
configuracao");

/**
 * Gera um grafo dirigido desconexo.<br>
 * Um grafo é desconexo se não for possível visitar qualquer vértice,
partindo de um outro e passando por arestas.
 * @param qtdeVertices Quantidade de vértices
 * @return GrafoDirigido desconexo
 * @throws Exception caso a quantidade de vértices seja 1
 */
public static GrafoDirigido getGrafoDesconexoDirigido(int qtdeVertices)
throws Exception {
    if (qtdeVertices == 1)
        throw new Exception("Nao e possivel gerar um grafo dessa
configuracao");
```

Quadro 40 – Métodos `getGrafoRegularNaoDirigido` e `getGrafoDesconexoDirigido`

3.3.2.5 Desenvolvimento da aplicação de exemplo

Como aplicação de teste, foi desenvolvida uma ferramenta simples que utiliza todos os recursos que o FGA oferece. A aplicação é constituída por três menus, sendo um para a execução de algoritmos de grafos, outro para o teste de propriedades de grafos e por fim, um menu que contém as funções de gerar grafos, criar, salvar e carregar um grafo.

Além disto, está disponível uma interface para editar o grafo, ou seja, adicionar e remover vértices e arestas. Com isto é possível testar também os métodos básicos da estrutura do grafo, sendo eles `addAresta`, `addVertice`, `delAresta` e `delVertice`.

A Figura 21 mostra a interface gráfica da aplicação de teste, com o resultado da execução do algoritmo de Hopcroft-Tarjan.

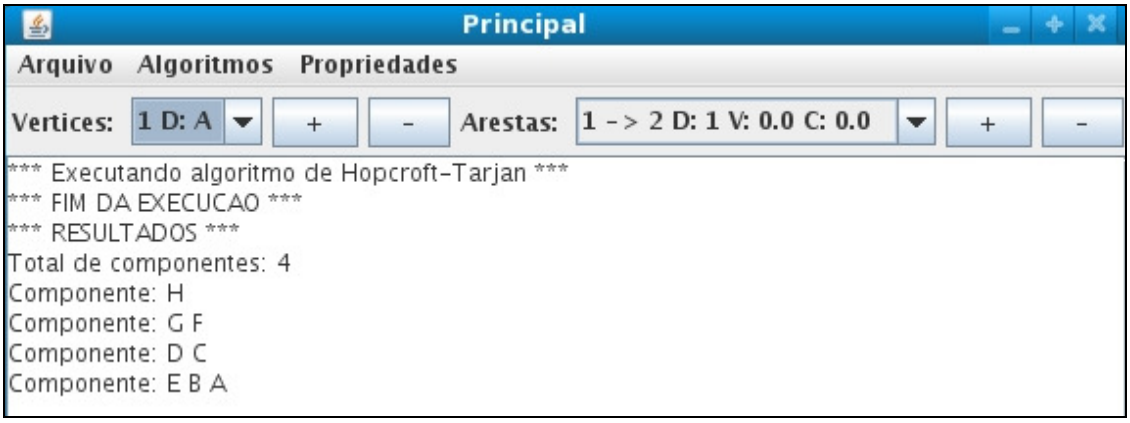


Figura 21 – Interface gráfica da aplicação de exemplo

No Quadro 41 é mostrada uma parte do código que chama a execução do algoritmo de ordenação topológica. O modo de obter o resultado da execução do algoritmo também está exemplificado no mesmo quadro.

```
public void executaOrdenacaoTopologica() {
    if (g == null) {
        JOptionPane.showMessageDialog(null, "Nenhum grafo criado...");
        return;
    }
    if (!g.ehDirigido() || !g.ehAciclico()) {
        JOptionPane.showMessageDialog(null, "O grafo precisa ser dirigido
e acíclico...");
        return;
    }

    imprimeln("*** Executando algoritmo de Ordenacao Topologica ***");
    AlgoritmoOrdenacaoTopologica alg = new AlgoritmoOrdenacaoTopologica();
    try {
        alg.executar((GrafoDirigido)g);
    } catch (Exception e) {
        imprimeln(e.getMessage());
        return;
    }

    imprimeln("*** FIM DA EXECUCAO ***");
    imprimeln("*** RESULTADOS ***");

    AlgoritmoOrdenacaoTopologicaResultado res = alg.getResultado();

    int tamanhoLista = res.getTamanho();
    Vertice v = res.getVertice(0);
    String seq = v.getDado() + "[" + res.getTempoAbertura(v) + "/" +
res.getTempoFechamento(v) + "]";
    for (int i = 1; i < tamanhoLista; i++) {
        v = res.getVertice(i);
        seq = v.getDado() + "[" + res.getTempoAbertura(v) + "/" +
res.getTempoFechamento(v) + "]" + " -> " + seq;
    }
    imprimeln(seq);
}
```

Quadro 41 – Método que exemplifica uso do algoritmo de ordenação topológica

O Quadro 42 exibe um trecho do código que chama os métodos que trabalham com a

persistência do grafo, ou seja, o método `persisteGrafo` e o `carregaGrafo` da classe `Persistencia`.

```
private void salvaGrafo() {
    String arquivo = javax.swing.JOptionPane.showInputDialog("Informe o
endereco do arquivo do grafo para salvar...");

    try {
        Persistencia.persisteGrafo(g, arquivo);
    } catch (IOException e) {}
}

private void carregaGrafo() {
    String arquivo = javax.swing.JOptionPane.showInputDialog("Informe o
endereco do arquivo do grafo...");
    File file = new File(arquivo);

    if (! file.exists()) {
        JOptionPane.showMessageDialog(null, "Arquivo inexistente");
        return;
    }

    cboArestas.removeAllItems();
    cboVertices.removeAllItems();
    g = null;

    try {
        g = Persistencia.carregaGrafo(arquivo);
        carregaVertices();
        carregaArestas();
    } catch (Exception e) {}
}
```

Quadro 42 – Métodos que exemplificam uso da persistência de grafos

Um exemplo da chamada de uma função para gerar um grafo bipartido, bem como testar se o grafo é bipartido pode ser visto no Quadro 43.

```
private void gerarGrafoDirigidoBipartido() {
    String strQtdeX = javax.swing.JOptionPane.showInputDialog("Informe a
quantidade de vertices do conjunto X...");
    if (strQtdeX == null || strQtdeX.equals("")) {
        JOptionPane.showMessageDialog(null, "Quantidade inválida...");
        return;
    }

    String strQtdeY = javax.swing.JOptionPane.showInputDialog("Informe a
quantidade de vertices do conjunto Y...");
    if (strQtdeY == null || strQtdeY.equals("")) {
        JOptionPane.showMessageDialog(null, "Quantidade inválida...");
        return;
    }

    g =
GeradorGrafos.getGrafoBipartidoDirigido(Integer.parseInt(strQtdeX),
Integer.parseInt(strQtdeY));
    carregaVertices();
    carregaArestas();
}

public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(null, "Bipartido: " + g.ehBipartido());
}
```

Quadro 43 – Métodos que exemplificam uso da geração de grafos e teste de propriedades

3.3.3 Operacionalidade da implementação

Esta seção apresenta a operacionalidade do FGA através da execução de um estudo de caso. Para a demonstração do FGA foi construída uma aplicação simples que desenha um grafo na tela, mostra algumas propriedades e a execução dos algoritmos de busca em profundidade e de Kruskal.

Inicialmente é necessário estender a classe `Vertice` para que seja possível criar novos métodos e atributos. Portanto é apresentada a classe `VerticeGeometrico` (Quadro 44), que é uma extensão da classe `Vertice`, e contém como atributos o raio de um círculo e a posição `x` e `y`. Esta classe é utilizada para desenhar os vértices do grafo na tela.

```
import base.Vertice;
public class VerticeGeometrico extends Vertice {
    private int x;
    private int y;
    private int raio;

    public VerticeGeometrico(int id) {
        super(id);
    }

    public VerticeGeometrico(int id, int x, int y, int raio) {
        super(id);
        this.x = x;
        this.y = y;
        this.raio = raio;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }
```

Quadro 44 – Classe `VerticeGeometrico`

Terminada a classe `VerticeGeometrico`, é necessário instanciar um grafo, conforme ilustrado no Quadro 45.

```
GrafoNaoDirigido grafo = new GrafoNaoDirigido();
```

Quadro 45 – Instanciação de um grafo não dirigido

O próximo passo é instanciar alguns vértices e arestas. O Quadro 46 mostra um exemplo da instanciação de vértices geométricos e arestas não dirigidas.

Para os vértices são informados um identificador, uma posição `x` e `y` e um raio. Além disso, é definida uma cor específica para cada vértice.

Para as arestas são informados um identificador, um vértice de origem e um vértice de destino. Outro atributo dado para a aresta é um peso, no caso acessado pelo método `setValor`. Tal atributo é necessário para que mais tarde possa ser executado o algoritmo de Kruskal e descobrir a árvore ou floresta geradora de custo mínimo.

```
VerticeGeometrico v1 = new VerticeGeometrico(1, 10, 3, 20);
VerticeGeometrico v2 = new VerticeGeometrico(2, 100, 150, 30);
VerticeGeometrico v3 = new VerticeGeometrico(3, 58, 300, 50);
VerticeGeometrico v4 = new VerticeGeometrico(4, 300, 200, 40);
VerticeGeometrico v5 = new VerticeGeometrico(5, 400, 100, 20);
VerticeGeometrico v6 = new VerticeGeometrico(6, 450, 200, 30);
VerticeGeometrico v7 = new VerticeGeometrico(7, 30, 500, 20);
VerticeGeometrico v8 = new VerticeGeometrico(8, 400, 520, 40);

v1.setCor(Color.green);
v2.setCor(Color.yellow);
v3.setCor(Color.orange);
v4.setCor(Color.red);
v5.setCor(Color.black);
v6.setCor(Color.blue);
v7.setCor(Color.gray);
v8.setCor(Color.cyan);

ArestaNaoDirigida a1 = new ArestaNaoDirigida(1, v1, v2);
ArestaNaoDirigida a2 = new ArestaNaoDirigida(2, v2, v3);
ArestaNaoDirigida a3 = new ArestaNaoDirigida(3, v2, v4);
ArestaNaoDirigida a4 = new ArestaNaoDirigida(4, v4, v1);
ArestaNaoDirigida a5 = new ArestaNaoDirigida(5, v5, v6);
ArestaNaoDirigida a6 = new ArestaNaoDirigida(6, v3, v4);
ArestaNaoDirigida a7 = new ArestaNaoDirigida(7, v3, v7);
ArestaNaoDirigida a8 = new ArestaNaoDirigida(8, v4, v7);
ArestaNaoDirigida a9 = new ArestaNaoDirigida(9, v7, v8);
ArestaNaoDirigida a10 = new ArestaNaoDirigida(10, v8, v4);

a1.setValor(100);
a2.setValor(5);
a3.setValor(40);
a4.setValor(50);
a5.setValor(14);
a6.setValor(3);
a7.setValor(15);
a8.setValor(7);
a9.setValor(2);
a10.setValor(1);
```

Quadro 46 – Instanciação de vértices e arestas

Para que os vértices e arestas componham o grafo, eles precisam ser explicitamente adicionados ao grafo. O Quadro 47 mostra um exemplo de como adicionar vértices e arestas em um grafo.

```
grafo.addVertice(v1);
grafo.addVertice(v2);
grafo.addVertice(v3);
grafo.addVertice(v4);
grafo.addVertice(v5);
grafo.addVertice(v6);
grafo.addVertice(v7);
grafo.addVertice(v8);

grafo.addAresta(a1);
grafo.addAresta(a2);
grafo.addAresta(a3);
grafo.addAresta(a4);
grafo.addAresta(a5);
grafo.addAresta(a6);
grafo.addAresta(a7);
grafo.addAresta(a8);
grafo.addAresta(a9);
grafo.addAresta(a10);
```

Quadro 47 – Adicionar vértices e arestas em um grafo

Com o grafo criado, contendo vértices e arestas, é possível executar algoritmos e verificar propriedades. O Quadro 48 mostra como executar e obter o resultado dos algoritmos de busca em profundidade e Kruskal.

```
AlgoritmoBuscaProfundidade algDFS = new AlgoritmoBuscaProfundidade();
algDFS.executar(grafo);
AlgoritmoBuscaProfundidadeResultado resDFS = algDFS.getResultado();

AlgoritmoKruskal algKruskal = new AlgoritmoKruskal();
algKruskal.executar(grafo);
AlgoritmoKruskalResultado resKruskal = algKruskal.getResultado();
```

Quadro 48 – Execução algoritmos de busca em profundidade e Kruskal

Terminada a execução dos dois algoritmos e de posse dos resultados obtidos pode-se desenhar o grafo na tela e destacar as soluções obtidas. O Quadro 49 mostra o desenho das arestas. Inicialmente são desenhadas todas as arestas do grafo e, em seguida, são pintadas em verde apenas as arestas que fazem parte da floresta de custo mínimo encontrada pelo algoritmo de Kruskal. Por fim, a última iteração desenha na tela o conteúdo do atributo `valor` de cada aresta.

```
for (int i = 0; i < grafo.getQtdeArestas(); i++) {
    ArestaNaoDirigida a = grafo.getAresta(i);

    VerticeGeometrico u = (VerticeGeometrico) a.getVi();
    VerticeGeometrico v = (VerticeGeometrico) a.getVj(u);

    g.setColor(Color.black);
    g.drawLine(u.getX() + u.getRaio() / 2, u.getY() + u.getRaio() / 2,
   v.getX() + v.getRaio() / 2, v.getY() + v.getRaio() / 2);
}

for (int i = 0; i < resKruskal.getArestas().size(); i++) {
    ArestaNaoDirigida a = (ArestaNaoDirigida)
      resKruskal.getArestas().get(i);

    VerticeGeometrico u = (VerticeGeometrico) a.getVi();
    VerticeGeometrico v = (VerticeGeometrico) a.getVj(u);

    g.setColor(Color.green);
    g.drawLine(u.getX() + u.getRaio() / 2, u.getY() + u.getRaio() / 2,
   v.getX() + v.getRaio() / 2, v.getY() + v.getRaio() / 2);
}

for (int i = 0; i < grafo.getQtdeArestas(); i++) {
    ArestaNaoDirigida a = grafo.getAresta(i);

    VerticeGeometrico u = (VerticeGeometrico) a.getVi();
    VerticeGeometrico v = (VerticeGeometrico) a.getVj(u);

    g.setColor(Color.white);
    g.fillRect((u.getX() + v.getX()) / 2, (u.getY() + v.getY()) / 2,
                String.valueOf(a.getValor()).length() * 7, 10);

    g.setColor(Color.blue);
    g.drawString(""+a.getValor(), (u.getX() + v.getX()) / 2,
                (u.getY() + v.getY()) / 2 + 10);
}
```

Quadro 49 – Desenho de arestas

O Quadro 50 mostra o desenho dos vértices do grafo. Junto ao vértice são destacados o atributo `id` e o tempo de abertura e fechamento encontrados pela execução do algoritmo de busca em profundidade.

```
for (int i = 0; i < grafo.getTamanho(); i++) {
    VerticeGeometrico v = (VerticeGeometrico) grafo.getVertice(i);

    g.setColor(v.getCor());
    g.fillOval(v.getX(), v.getY(), v.getRaio(), v.getRaio());

    g.setColor(Color.black);
    g.drawString("{Id = " + v.getId() + "}
                 [" + resDFS.getTempoAbertura(v) + "/" +
                 resDFS.getTempoFechamento(v) + "]",
                 v.getX() + v.getRaio(),
                 v.getY() + v.getRaio() / 2);
}
```

Quadro 50 – Desenho de vértices

As propriedades que são mostradas na tela são desenhadas como texto, que aparece

como `true`, caso a propriedade esteja satisfeita ou `false`, caso a propriedade não seja satisfeita. O Quadro 51 apresenta como é trabalhado com as propriedades de um grafo.

```
g.setColor(Color.black);
g.drawString("Conexo: " + grafo.ehConexo(), 480, 250);
g.drawString("Simples: " + grafo.ehSimples(), 480, 270);
g.drawString("Acíclico: " + grafo.ehAciclico(), 480, 290);
g.drawString("Denso: " + grafo.ehDenso(), 480, 310);
g.drawString("Bipartido: " + grafo.ehBipartido(), 480, 330);
g.drawString("Completo: " + grafo.ehCompleto(), 480, 350);
g.drawString("Regular: " + grafo.ehRegular(), 480, 370);
g.drawString("Ciclo: " + grafo.ehCiclo(), 480, 390);
g.drawString("Nulo: " + grafo.ehNulo(), 480, 410);
g.drawString("Trivial: " + grafo.ehTrivial(), 480, 430);
g.drawString("Árvore: " + grafo.ehArvore(), 480, 450);
g.drawString("Floresta: " + grafo.ehFloresta(), 480, 470);
```

Quadro 51 – Teste de propriedades

Por fim, a Figura 22 apresenta o resultado final da implementação. Nela é possível ver as arestas atingidas pelo algoritmo de Kruskal, bem como os tempos de abertura e fechamento dos vértices atingidos pelo algoritmo de busca em profundidade.

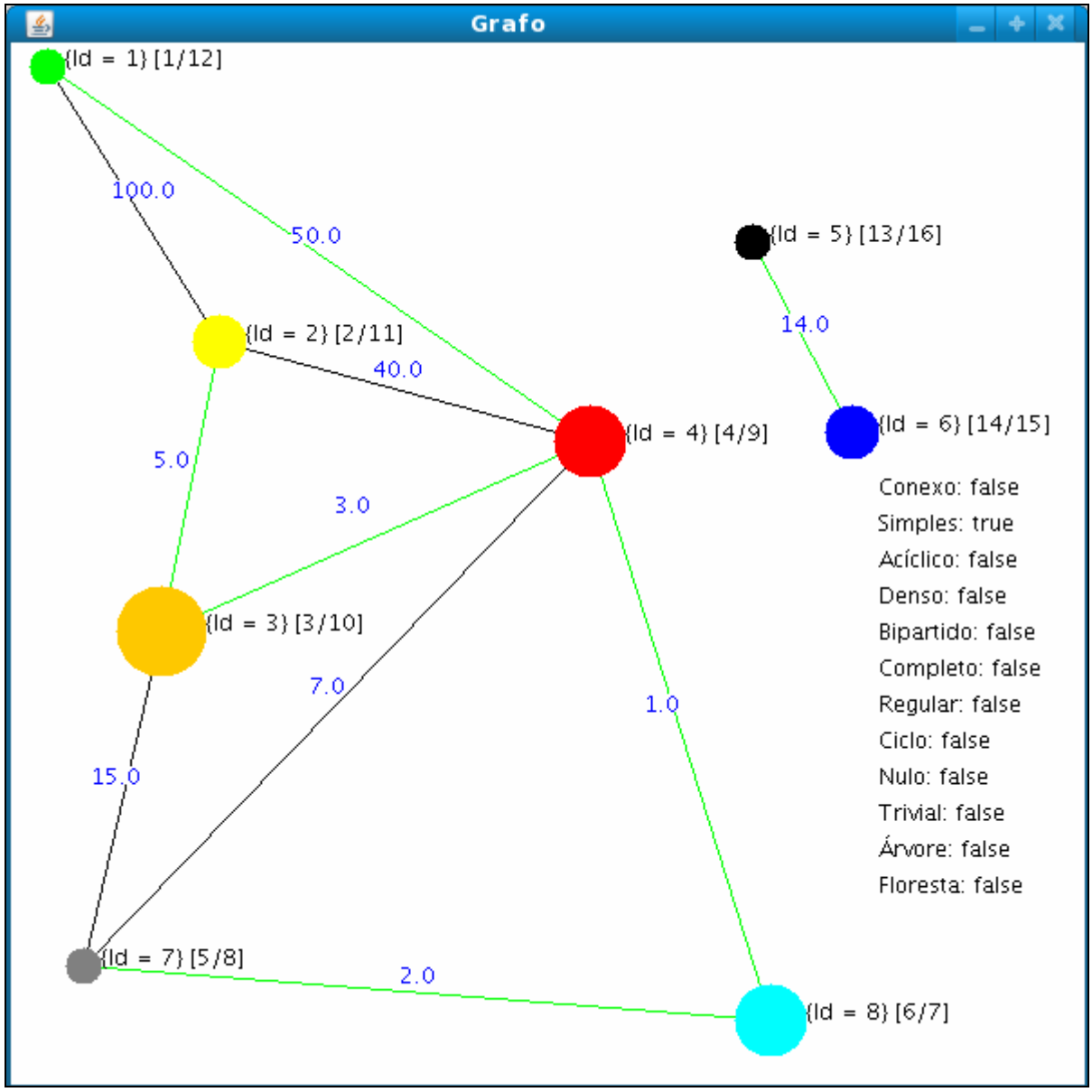


Figura 22 – Visualização do grafo

3.4 RESULTADOS E DISCUSSÃO

Nesta seção são apresentadas as comparações realizadas entre diversos algoritmos implementados no FGA e também uma breve discussão sobre os trabalhos correlatos.

Inicialmente são mostrados os resultados das comparações feitas com os algoritmos de grafos. As comparações foram efetuadas subdividindo os algoritmos em grupos. Cada grupo de algoritmos é utilizado para resolver o mesmo problema proposto. Todos os testes foram feitos em um camputador com processador Intel com velocidade de 2.0 *GigaHertz* (GHz) e 2

GBytes (GB) de memória.

A seguir são mostradas as semelhanças, diferenças, vantagens e desvantagens encontradas no presente trabalho em relação aos trabalhos correlatos. Por fim é descrita, de forma breve, a implementação de parte do FGA na linguagem Objective-C.

3.4.1 Comparação entre algoritmos de fluxo

Utilizar o algoritmo certo para o problema em questão é fundamental para não comprometer o desempenho da aplicação. Um exemplo é mostrado na Tabela 1, onde a execução do algoritmo de Ford-Fulkerson é utilizada para encontrar o fluxo máximo em uma rede de fluxo.

Se houver a necessidade de escolher o fluxo máximo de custo mínimo, o tempo de execução do algoritmo é muito maior. Para um problema onde o tempo de processamento é crítico e não há necessidade de encontrar o custo mínimo do fluxo, não é viável a utilização do algoritmo de Dijkstra para distribuir o fluxo pelas rotas. Para este caso convém manter o algoritmo de busca em profundidade.

Tabela 1 – Comparação entre o algoritmo de Dijkstra e busca em profundidade

Vértices	**Busca em profundidade**			**Dijkstra**		
	Fluxo	**Custo**	**Tempo(ms)**	**Fluxo**	**Custo**	**Tempo(ms)**
10	63,78	8464,72	2	63,78	8330,53	7
30	248,64	58934,58	20	248,64	33229,07	62
60	1811,04	294781,04	69	1811,04	185502,11	622
100	1179,41	331843,56	114	1179,41	149274,08	915
200	2451,92	597813,16	245	2451,92	252038,08	19595
500	6594,15	1874917,23	2007	6594,15	685813,41	767539

3.4.2 Comparação entre algoritmos de busca

A Tabela 2 mostra a comparação feita entre os algoritmos de busca em largura e busca em profundidade. Como elemento temporal de comparação foi adotado o tempo de abertura do vértice procurado, ou seja, quando o vértice de destino foi atingido. Observa-se que ambas as buscas tiveram médias semelhantes, o que demonstra que muito do desempenho dos dois algoritmos deve-se a localização do vértice que está sendo procurado. Caso o vértice esteja num nível mais profundo, então a busca em profundidade leva vantagem, enquanto que se o

vértice estiver em níveis iniciais, a busca em largura mostra-se muito eficiente.

Para os testes foram gerados dois grafos aleatórios: um grafo dirigido simples e conexo e outro grafo não dirigido simples e conexo.

Tabela 2 – Comparação entre algoritmos de busca

Vértices	**Tempo de abertura Teste 1**		**Tempo de abertura Teste 2**	
	BFS	**DFS**	**BFS**	**DFS**
10	7	10	13	6
30	35	13	5	13
60	103	9	37	43
100	161	96	25	45
200	239	128	129	70
500	617	145	127	348

3.4.3 Comparação entre algoritmos de geração de árvore de custo mínimo

A Tabela 3 mostra a comparação entre os algoritmos de Prim e de Kruskal, ambos resolvendo o problema de árvore geradora de custo mínimo. O grafo utilizado para todos os testes é não dirigido simples e conexo, já que o algoritmo de Prim não se aplica em casos de grafos desconexos ou dirigidos.

Observa-se que a implementação do algoritmo de Prim no FGA é relativamente mais rápido que o de Kruskal. Boa parte do custo temporal do algoritmo de Kruskal deve-se a estruturas auxiliares para manter as árvores geradores de custo mínimo.

Tabela 3 – Comparação entre algoritmos de geração de árvore de custo mínimo

Vértices	**Prim**		**Kruskal**	
	Custo	**Tempo(ms)**	**Custo**	**Tempo(ms)**
10	231,15	2	231.15	2
30	330,80	3	330,80	6
60	140,31	8	140,31	52
100	113,87	15	113,87	77
200	384,80	27	384,80	134
500	133,67	105	133,67	1019

3.4.4 Comparação entre algoritmos de menor caminho

Os algoritmos de busca de menor caminho em um grafo utilizados na comparação foram os algoritmos de Dijkstra, Bellman-Ford e Floyd-Warshall. Como o propósito de cada

algoritmo é diferente foram realizados diversos tipos de teste, sendo que o grafo utilizado para todos os testes foi um grafo dirigido simples.

O primeiro teste realizado foi o de efetuar apenas uma consulta, partindo de um vértice de origem qualquer até outro vértice de destino. A Tabela 4 como resultado o algoritmo de Dijkstra sempre com o melhor desempenho. Nota-se também que o algoritmo de Bellman-Ford executou mais rápido no teste com 100 vértices do que com 60 vértices. Isso se deve ao fato do algoritmo de Bellman-Ford depender do número de arestas do grafo. Evidentemente, pelo tempo de execução, o grafo com 60 vértices possuía mais arestas que o grafo com 100 vértices.

Tabela 4 – Comparação entre algoritmos de menor caminho em uma consulta

Vértices	**Dijkstra**		**Bellman-Ford**		**Floyd-Warshall**	
	Custo	**Tempo(ms)**	**Custo**	**Tempo(ms)**	**Custo**	**Tempo(ms)**
10	35,17	3	35,17	2	35,17	11
30	56,46	9	56,46	15	56,46	53
60	17,25	42	17,25	69	17,25	137
100	58,68	46	58,68	54	58,68	446
200	17,44	80	17,44	94	17,44	3775
500	4,49	1133	4,49	4531	4,49	77766

O segundo teste feito foi com um número maior de consultas, todas partindo de um mesmo vértice inicial. A Tabela 5 mostra o resultado das execuções dos algoritmos.

É possível notar que o algoritmo de Bellman-Ford é mais eficiente, com as novas circunstâncias de avaliação. Já o algoritmo de Dijkstra mostra-se perdendo desempenho à medida que o número de consultas aumenta. Por fim, o algoritmo de Floyd-Warshall manteve-se com tempo de execução praticamente igual nos dois testes.

Tabela 5 – Comparação entre os algoritmos com várias consultas tendo mesma origem

Vértices	**Consultas**	**Dijkstra Tempo(ms)**	**Bellman-Ford Tempo(ms)**	**Floyd-Warshall Tempo(ms)**
10	5	4	1	6
30	15	107	40	56
60	30	201	47	171
100	50	631	65	439
200	100	6653	233	3882
500	250	158305	2318	75219

O terceiro teste efetuado leva em consideração o mesmo número de consultas anterior, porém o vértice de partida não é o mesmo.

Com esta nova configuração quem se destacou foi o algoritmo de Floyd-Warshall, que manteve o mesmo tempo de execução para os três tipos de teste efetuados. Para este novo

problema os algoritmos de Dijkstra e Bellman-Ford mostraram-se pouco eficientes. A Tabela 6 mostra o resultado comparando os três algoritmos.

Tabela 6 – Comparação entre os algoritmos com várias consultas tendo origens diferentes

Vértices	Consultas	Dijkstra Tempo(ms)	Bellman-Ford Tempo(ms)	Floyd-Warshall Tempo(ms)
10	5	8	11	13
30	15	79	77	57
60	30	214	326	151
100	50	1014	1657	436
200	100	13870	36652	4352
500	250	546846	1908240	85741

3.4.5 Comparação entre trabalhos correlatos

Em relação aos trabalhos correlatos, verifica-se que o presente trabalho possui algumas funcionalidades semelhantes aos mesmos, como a disponibilidade de diversos algoritmos, persistência e geração de grafos. Como diferenciais tem-se o maior número de algoritmos implementados, a flexibilidade para a geração de grafos aleatórios e a extração de propriedades.

Quanto aos trabalhos de Braun (2009) e de Hackbarth (2008), observa-se que não são disponibilizadas opções para gerar instâncias aleatórias de grafos com finalidade de realizar testes. Já a JgraphT Team (2005) disponibiliza geração de tipos como grafos roda, estrela, ciclo e hiper cubo, enquanto que o FGA possui geração de grafos regulares, conexos ou desconexos, bipartidos e bipartidos completos, esparsos ou densos e simples ou multigrafos.

Outro ponto a ser destacado é o da checagem de propriedades. Dos três trabalhos apresentados, o de Braun (2009) permite extrair algumas propriedades, como o grafo ser completo, bipartido, cíclico, nulo, simples e regular. A JgraphT Team (2005) também testa propriedades e disponibiliza classes para verificar grafos biconectados, acíclicos ou ciclos, conexos ou desconexos.

Na persistência nota-se a preocupação em relação a outras aplicações. No trabalho de Braun (2009) e na biblioteca da Jgrapht Team (2005) é disponibilizada a exportação de grafos em formatos que podem ser abertos por outros softwares, enquanto que no FGA é disponibilizada uma função para persistir grafos em um formato XML, porém em uma estrutura de arquivo própria.

O Quadro 52 apresenta uma comparação entre os três trabalhos correlatos e o

framework desenvolvido.

	JgraphT (2005)	HACKBART (2008)	BRAUN (2009)	FGA (2010)
Permite criar grafos	Sim	Sim	Sim	Sim
Permite estender vértices, arestas e grafos	Sim	Não	Não	Sim
Permite gerar grafos	Sim	Não	Não	Sim
Permite persistir grafos	Sim	Não	Sim	Sim
Permite verificar propriedades	Sim	Não	Sim	Sim
Disponibiliza algoritmos	Sim	Sim	Não	Sim
Permite criar novos algoritmos	Sim	Não	Sim	Sim
Permite acompanhar execução de algoritmos	Sim	Sim	Sim	Não

Quadro 52 – Comparação entre trabalhos correlatos e o FGA

3.4.6 O *framework* em Objective-C

Parte do FGA foi reescrita na linguagem Objective-C. Nesta versão estão disponíveis todas as classes responsáveis pela estrutura do grafo, com a verificação das mesmas propriedades desenvolvidas em Java. Além disso, todos os algoritmos também tiveram uma versão implementada em Objective-C.

Cada classe é composta por dois arquivos: um arquivo para a interface (`.h`) e outro arquivo para a implementação (`.m`).

Como exemplo é mostrada parte da implementação do algoritmo de Prim. O Quadro 53 detalha o conteúdo do arquivo `AlgoritmoPrim.h`, enquanto que o Quadro 54 mostra um trecho do arquivo `AlgoritmoPrim.m`.

```
#ifndef _ALGORITMOPRIM_
#define _ALGORITMOPRIM_
#import <Foundation/Foundation.h>
#import <Foundation/NSArray.h>
#import "AlgoritmoPrimResultado.h"
#import "GrafoNaoDirigido.h"
#import "PriorityQueue.h"

@interface AlgoritmoPrim: NSObject {
@private
    AlgoritmoPrimResultado* resultado;
    NSMapTable* naArvore;
    PriorityQueue* fila;
}
-(AlgoritmoPrimResultado*) getResultado;
-(void) executar: (GrafoNaoDirigido*) g;
@end

#endif
```

Quadro 53 – Arquivo `AlgoritmoPrim.h`

```
#import "AlgoritmoPrim.h"
#import "Constante.h"

@implementation AlgoritmoPrim
-(AlgoritmoPrimResultado*) getResultado {
    return resultado;
}

-(void) executar: (GrafoNaoDirigido*) g {
    int tamanhoGrafo = [g getTamanho];
    resultado = [[AlgoritmoPrimResultado alloc] init];
    naArvore = [NSMapTable mapTableWithStrongToStrongObjects];
    fila = [[PriorityQueue alloc] init: tamanhoGrafo];

    NSMapTable* pares = [NSMapTable mapTableWithStrongToStrongObjects];

    int i;
    for (i = 0; i < tamanhoGrafo; i++) {
        Vertice* v = [g getVertice: i];

        [naArvore setObject:[NSNumber numberWithInt: 0] forKey: v];
        [resultado setCusto: v andCusto: INF];
        [resultado setPredecessor: v andPai: nil];

        PairPriority* par = [[PairPriority alloc] init: v andCusto: 0.0];
        [pares setObject: par forKey: v];
    }

    Vertice* x = [g getUmVertice];

    PairPriority* par = [pares objectForKey: x];
    [par setCusto: 0.0];
    [fila push: par];

    while ([fila count] > 0) {
        PairPriority* p = [fila pop];

        Vertice* u = (Vertice*) [p getDado];
        {...}
```

Quadro 54 – Arquivo `AlgoritmoPrim.m`

Para armazenar os resultados obtidos foi utilizada a mesma idéia da versão desenvolvida em Java, onde para cada algoritmo existe uma classe correspondente responsável por manter o resultado. Como exemplo é apresentada a classe `AlgoritmoPrimResultado` e seus respectivos arquivos `AlgoritmoPrimResultado.h` (Quadro 55) e `AlgoritmoPrimResultado.m` (Quadro 56).

```
#ifndef _ALGORITMOPRIMRESULTADO_
#define _ALGORITMOPRIMRESULTADO_
#import <Foundation/Foundation.h>
#import <Foundation/NSArray.h>
#import "Vertice.h"
@interface AlgoritmoPrimResultado: NSObject {
@private
    NSMapTable* predecessor;
    NSMapTable* descendentes;
    NSMapTable* custo;
    NSMutableArray* arestas;
    double custoTotal;
}
-(void) setPredecessor: (Vertice*) v andPai: (Vertice*) pai;
-(Vertice*) getPredecessor: (Vertice*) v;
-(void) addCustoTotal: (double) valor;
-(double) getCustoTotal;
-(void) addDescendente: (Vertice*) v andFilho: (Vertice*) filho;
-(NSMutableArray*) getDescendentes: (Vertice*) v;
-(void) addAresta: (Aresta*) a;
-(NSMutableArray*) getArestas;
-(void) setCusto: (Vertice*) v andCusto: (double) pCusto;
-(double) getCusto: (Vertice*) v;
@end
#endif
```

Quadro 55 – Arquivo `AlgoritmoPrimResultado.h`

```
#import "AlgoritmoPrimResultado.h"
#import "Constante.h"

@implementation AlgoritmoPrimResultado
-(id) init {
    self = [super init];

    predecessor = [NSMapTable mapTableWithStrongToStrongObjects];
    descendentes = [NSMapTable mapTableWithStrongToStrongObjects];
    custo = [NSMapTable mapTableWithStrongToStrongObjects];
    arestas = [[NSMutableArray alloc] init];
    custoTotal = 0.0;

    return self;
}

-(void) setPredecessor: (Vertice*) v andPai: (Vertice*) pai {
    [predecessor setObject: pai forKey: v];
}

-(Vertice*) getPredecessor: (Vertice*) v {
    return [predecessor objectForKey: v];
}
{...}
```

Quadro 56 – Arquivo `AlgoritmoPrimResultado.m`

De forma similar a versão implementada em Java, a chamada dos métodos para criação dos grafos e execução de algoritmos é explanada no Quadro 57. Para o exemplo é utilizada a execução do algoritmo de Prim.

```
    GrafoNaoDirigido* g = [[GrafoNaoDirigido alloc] init];

    Vertice* v1 = [[Vertice alloc] init: 1];
    Vertice* v2 = [[Vertice alloc] init: 2];
    Vertice* v3 = [[Vertice alloc] init: 3];
    Vertice* v4 = [[Vertice alloc] init: 4];

    [v1 setDado: @"1"];
    [v2 setDado: @"2"];
    [v3 setDado: @"3"];
    [v4 setDado: @"4"];

    [g addVertice: v1];
    [g addVertice: v2];
    [g addVertice: v3];
    [g addVertice: v4];

    ArestaNaoDirigida* a1 = [[ArestaNaoDirigida alloc] init: 1 andVi: v1
andVj: v2];
    [a1 setValor: 10923.0];
    ArestaNaoDirigida* a2 = [[ArestaNaoDirigida alloc] init: 2 andVi: v1
andVj: v3];
    [a2 setValor: 1235.0];
    ArestaNaoDirigida* a3 = [[ArestaNaoDirigida alloc] init: 3 andVi: v2
andVj: v3];
    [a3 setValor: 1200.0];
    ArestaNaoDirigida* a4 = [[ArestaNaoDirigida alloc] init: 4 andVi: v1
andVj: v4];
    [a4 setValor: 1000.0];

    [g addAresta: a1];
    [g addAresta: a2];
    [g addAresta: a3];
    [g addAresta: a4];

    AlgoritmoPrim* alg = [[AlgoritmoPrim alloc] init];
    [alg executar: g];
    AlgoritmoPrimResultado* res = [alg getResultado];

    printf("Resultado: %lf\n", [res getCustoTotal]);
```

Quadro 57 – Execução do algoritmo de Prim

4 CONCLUSÕES

Os resultados obtidos com o desenvolvimento do FGA foram considerados satisfatório, pois os requisitos propostos foram cumpridos, e, além disso, o FGA conta com uma série de outros recursos não mencionados inicialmente na proposta. O objetivo final do trabalho foi atingido, que é a disponibilização de um *framework* de algoritmos de grafos que contivesse desde algoritmos clássicos, funções de geração de grafos com base em restrições e por fim, verificação de propriedades.

Com a utilização do FGA foi possível comparar o desempenho dos algoritmos implementados. Foram feitas diversas baterias de testes com diferentes tipos de problemas a fim de verificar qual é a melhor situação para utilizar cada algoritmo.

Com o recurso de gerar grafos aleatórios foi possível criar instâncias de grafos com mais vértices e arestas. As instâncias geradas puderam ser facilmente verificadas se estão corretas utilizando o recurso de testar propriedades dos grafos.

Além disso, a persistência permite ao usuário salvar o grafo em que está trabalhando. Tal recurso facilita o transporte do grafo de um lugar para outro. Por ser um *framework*, é possível que o grafo possa ser persistido em outros formatos, já que para isso basta implementar a função de persistência para acessar os atributos de vértices e arestas. A biblioteca DOM apresentou grande importância na persistência do grafo, sendo que a partir dela é feito todo o controle de geração e recuperação de dados no modelo XML.

Como principal limitação do trabalho tem-se o fato de não haver uma maneira de ver a representação de um grafo na forma gráfica. No entanto, a aplicação de teste mostra de maneira simples o conjunto de vértices e arestas, tendo disponíveis funções para adicionar e remover os mesmos.

Outra limitação encontrada é o baixo desempenho da linguagem Java. Para instâncias grandes de grafos há uma demora demasiada tanto na geração de grafos como na execução de algum algoritmo sobre o grafo em questão. Cabe uma investigação mais profunda sobre este problema, podendo ser feita a comparação entre as implementações em Java e em Objective-C.

O grande consumo de memória também foi outro fator crítico enfrentado principalmente nos algoritmos de geração de grafos. Um exemplo é na geração de grafos densos de mais de 1000 vértices. Para que ele seja considerado denso é necessário que ele contivesse no mínimo 1000000 arestas. Tanto para arestas como vértices há atributos em

questão, portanto este grafo pode consumir mais de 100Mb de memória.

Por fim, está aberto um horizonte para trabalhos futuros, desde implementação de novos algoritmos, verificação de outras propriedades, além de gerar um novo módulo para visualização gráfica dos grafos criados.

4.1 EXTENSÕES

Como extensão para este trabalho sugere-se verificar outras propriedades de grafos, tais como o grafo cordal, hipercubo, perfeito, cactos, planar, isomorfo a outro grafo, entre outras. Além disso, disponibilizar funções para que sejam gerados estes tipos de grafos.

Sugere-se também a implementação de recursos que permitam ao usuário trabalhar com o grafo em uma forma visual.

Outra sugestão é a possibilidade de exportar o grafo para outros formatos, facilitando assim a integração com outras aplicações.

Por fim, sugere-se a implementação de outros algoritmos clássicos de grafos, tais como emparelhamento perfeito, clique máximo, ciclo hamiltoniano, ciclo euleriano, relabel-to-front, Boruvka, entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

ALSUWAIYEL, Muhammad H. **Algorithms**: design techniques and analysis. Singapore: World Scientific Publishing, 2003.

BRAUN, Susan. **Ferramenta visual para criação e execução de algoritmos aplicados sobre teoria dos grafos**. 2009. 74 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

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

EPPSTEIN, David. **Strong connectivity**. [California], 2001. Disponível em: <http://www.ics.uci.edu/~eppstein/161/960220.html>. Acesso em: 30 ago. 2010.

FEOFILOFF, Paulo; KOHAYAKAWA, Yoshiharu; WAKABAYASHI, Yoshiko. **Teoria dos grafos**: uma introdução sucinta. [São Paulo], 2009. Disponível em: <http://www.ime.usp.br/~pf/teoriadosgrafos/>. Acesso em: 27 fev. 2010.

FLEMING, Kermin; CRUTCHFIELD, Chris. **Minimum cost maximum flow, minimum cost circulation, cost/capacity scaling**. [Cambridge], 2006. Disponível em: <http://courses.csail.mit.edu/6.854/06/scribe/s12-minCostFlowAlg.pdf>. Acesso em: 10 ago. 2010.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estrutura de dados e algoritmos em Java**. 4. ed. Porto Alegre: Bookman, 2007.

_____. **Projeto de algoritmos**: fundamentos, análise e exemplos da Internet. Tradução Bernardo Copstein. Porto Alegre: Bookman, 2004.

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

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

JGRAPHT TEAM. **JgraphT**: a free Java graph library. [S.l.], 2005. Disponível em: <http://www.jgrapht.org/>. Acesso em: 13 mar. 2010.

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

LAU, Hang T. **A Java library of graph algorithms and optimization**. Boca Raton: Chapman & Hall/CR, 2007.

LIPSCHUTZ, Seymour; LIPSON, Marc. **Matemática discreta**. 2. ed. Porto Alegre: Bookman, 2004.

ORACLE TEAM. **JDK 5.0 Javadoc technology**. [S.l.], [2005?]. Disponível em: <http://download.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html>. Acesso em: 10 ago. 2010.

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

SAUVÉ, Jacques P. **O que é um framework?** [Campina Grande], [2007?]. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>. Acesso em: 13 mar. 2010.

SCHEINERMAN, Edward R. **Matemática discreta**: uma introdução. Tradução Alfredo Alves de Farias. São Paulo: Pioneira, 2003.

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

WORLD WIDE WEB CONSORTIUM. **XML DOM introduction**. [S.l.], [2010?]. Disponível em: <http://www.w3schools.com/dom/dom_intro.asp>. Acesso em: 10 ago. 2010.