UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FURB GRAPHS: UMA FERRAMENTA DE APOIO AO APRENDIZADO PARA A DISCIPLINA DE TEORIA DOS GRAFOS

LUIZ HENRIQUE BERNARDES

BLUMENAU
2016

LUIZ HENRIQUE BERNARDES

FURB GRAPHS: UMA FERRAMENTA DE APOIO AO APRENDIZADO PARA A DISCIPLINA DE TEORIA DOS GRAFOS

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

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2016

FURB GRAPHS: UMA FERRAMENTA DE APOIO AO APRENDIZADO PARA A DISCIPLINA DE TEORIA DOS GRAFOS

Por

LUIZ HENRIQUE BERNARDES

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

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

Blumenau, 04 de julho de 2016

Dedico este trabalho à minha família e amigos, que sempre acreditaram em min e que contribuíram para a realização deste.

AGRADECIMENTOS

Agradeço a minha família pelo apoio e compreensão em todos os momentos durante o desenvolvimento deste trabalho. Em especial aos meus pais Marcelino Bernardes e Alcineia Medeiros que sempre me incentivaram a estudar e a me dedicar aos meus objetivos.

A todos os meus amigos que me incentivaram e estiveram comigo durante todo o processo de criação deste trabalho. Em especial ao Gibran Peruzzolo Godoi que foi o primeiro a ler este trabalho e que me ajudou nas revisões. Ao Renato Ramos que foi o primeiro a testar e dar a sua opinião neste trabalho. Ao Paulo César Schmidt que me auxiliou com características importantes compartilhando seu conhecimento.

Ao meu orientador, Aurélio Faustino Hoppe, por ter me incentivado a continuar nos momentos mais difíceis, pela disposição e atenção com as caraterísticas mais importantes, por ter posto confiança nas minhas capacidades, tornando possível a conclusão deste trabalho.

A todos os professores do curso de Ciência da Computação, que durante toda esta jornada contribuíram com o aprendizado que auxiliou na realização deste trabalho.

A introdução de abstrações adequadas é a nossa única ajuda mental para reduzir o apelo à enumeração, para organizar e dominar a complexidade.

Edsger Dijkstra

RESUMO

Este trabalho descreve o desenvolvimento e adaptação da aplicação FURB Graphs desenvolvida por Borba (2014), tendo como objetivo torná-la uma ferramenta de apoio ao ensino para a disciplina de teoria dos grafos. O objetivo principal desta ferramenta é permitir que o usuário possa compreender os algoritmos de busca em largura, busca em profundidade e Dijkstra de forma visual e interativa, acompanhando passo a passo a execução dos mesmos, contando com informações para que o processo se torne mais prático e menos abstrato durante a navegação. Através de pesquisas com trabalhos correlatos, foram identificados meios de transmitir o funcionamento dos algoritmos de forma mais adequada, utilizando legendas com cores relacionadas aos estados de vértices e arestas, informações e instruções sobre os algoritmos executados. Os resultados obtidos através de experimentos e testes de usabilidade demonstram que a ferramenta é capaz de fornecer um ambiente onde o usuário possa aprender sobre os algoritmos e suas características, facilitando o processo de compreensão de temas importantes na disciplina de teoria dos grafos.

Palavras-chave: Ambiente de aprendizado. Busca e caminhamento em grafos. Busca em largura. Busca em profundidade. Algoritmo de Dijkstra.

ABSTRACT

This work describes the development and adaptation of FURB Graphs application developed by Borba (2014), with the objective to make it a teaching tool for the discipline of graph theory. The main purpose of this tool is to allow the user to understand the algorithms of breadth first search, deep first search and Dijkstra in a visual and interactive way, following step by step the execution of the same, with information so that the process becomes more practical and less abstract while the navigation. Through researches with related works, were identified ways to transmit the operation of the algorithms more appropriately, using captions with colors related to the states of vertices and edges, information and instructions about the algorithms executed. Results from usability experiments and tests demonstrate that the tool is able to provide an environment where the user can learn about the algorithms and their characteristics, facilitating the process of understanding of important issues in graph theory.

Key-words: Environment of learning. Search and traversal in graphs. Breadth first search. Deep first search. Dijkstra algorithm.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API – Application Programming Interface

BFS – Breadth First Search

DFS – Deep First Search

HTML – Hyper Text Markup Language

JSON – JavaScript Object Notation

UML – Unified Modeling Language

XML – eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A teoria dos grafos se apresenta como uma ferramenta simples, acessível e poderosa para a construção de modelos e resolução de problemas relacionados a arranjos de objetos discretos (RABUSKE, 1992, p.1). Ainda segundo Rabuske (1992, p.1), ter conhecimento de grafos se torna importante e tem impacto sobre diversas áreas, pois, eles são utilizados para modelagem de diversos problemas, tais como: processos industriais, análise de caminho crítico, tática e logística (campo militar), sistemas de comunicação, estudo de transmissão de informações, escolha de uma rota ótima, fluxos em redes, redes elétricas (engenharia elétrica e civil, arquitetura, computação), genética, psicologia, economia, estrutura social, jogos, física, química, tecnologia de computador, antropologia, linguística, etc. Juntando assim, áreas de diferentes aplicações e objetivos, onde todas podem ser beneficiadas a fim de solucionar seus problemas com o auxílio da teoria dos grafos.

Visto o impacto que se tem nas diversas áreas abrangentes do tema, se torna possível visualizar a importância e necessidade de se criar metodologias e ferramentas práticas, que auxiliem na compreensão dos algoritmos e estratégias aplicadas à teoria dos grafos. Segundo Soares et. al. (2004, p. 1) atualmente o problema está na dificuldade dos estudantes em compreender os algoritmos utilizados na disciplina de teoria dos grafos, seja devido à complexidade ou até mesmo ao nível de abstração dos problemas. Essa confirmação representa um dos problemas no estudo dos algoritmos de teoria dos grafos, onde a necessidade de abstração se dá em um nível elevado, fazendo com que o aluno necessite de uma certa familiaridade com os cenários complexos dos problemas abordados no estudo destes algoritmos.

Ainda segundo Soares et al. (2004, p. 1) o ensino de teoria dos grafos em cursos de Ciência da Computação e Sistemas de Informação normalmente tem como objetivo o estudo e implementação de algoritmos em grafos. Soares et al. (2004, p. 2) complementam que grande parte dos alunos sentem dificuldades na implementação desses algoritmos, devido a necessidade de suporte e a diferença de níveis de abstração entre as definições teóricas dos algoritmos e representações computacionais necessárias para implementá-los.

Soares et al. (2004, p. 2) também afirmam que, ao estudarem as técnicas envolvidas na implementação de algoritmos em grafos, os alunos normalmente criam mentalmente ou através de desenhos, visualizações da execução desses algoritmos na tentativa de entender seus passos. Logo, a visualização gráfica e a interação passo a passo proporcionada através de uma ferramenta computacional, torna o ato de estudar, implementar e testar estes algoritmos

em uma experiência prática e dinâmica, diminuindo o nível de abstração e complexidade e melhorando o nível de interpretação dos algoritmos.

Diante deste contexto e visando auxiliar o ensino dos conceitos e algoritmos de Teoria dos Grafos, este trabalho apresenta uma extensão para o trabalho criado por Zatelli (2010), que por sua vez recebeu uma extensão de Borba (2014), tendo como objetivo tornar o *framework* de Borba (2014) uma ferramenta que recrie um ambiente de ensino com o auxílio de *logs* das instruções que estão sendo processadas, assim como, apresentando a relação de cada instrução executada com as linhas dos algoritmos em questão.

1.1 OBJETIVOS

O objetivo deste trabalho é adaptar a aplicação FURB Graphs (BORBA, 2014) tornando-a uma ferramenta de apoio ao ensino para a disciplina de teoria dos grafos, a partir da implementação de métodos para visualização e acompanhamento gráfico dos estados e atributos dos algoritmos disponibilizados.

Os objetivos específicos do trabalho são:

a) adaptar a ferramenta de Borba (2014) para demonstrar o funcionamento passo a passo dos algoritmos de busca em largura (BFS), busca em profundidade (DFS) e o algoritmo de Dijkstra para caminhamento através da interface gráfica;
b) apresentar informações e instruções quanto aos algoritmos executados.

1.2 ESTRUTURA

Este trabalho está subdividido em quatro capítulos. O primeiro capítulo apresenta a justificativa para o desenvolvimento do trabalho, assim como seus objetivos e sua estrutura. O segundo capítulo contém a fundamentação teórica, explicando conceitos gerais sobre a teoria dos grafos e esclarecendo tópicos importantes que devem ser considerados para o desenvolvimento da aplicação.

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

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

2 FUNDAMENTAÇÃO TEÓRICA

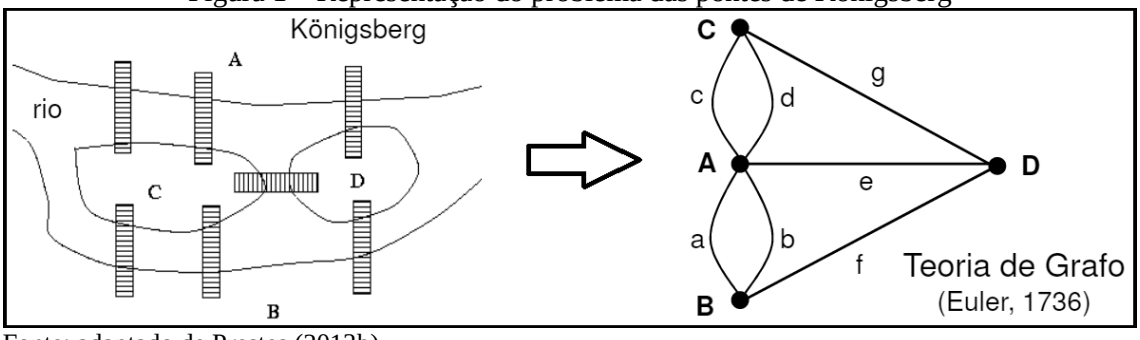
Este capítulo está organizado em seis seções. A seção 2.1 aborda o surgimento da teoria dos grafos. A seção 2.1.1 apresenta os conceitos básicos sobre grafos. Na seção 2.2 são apresentados os algoritmos de busca em grafos, com seus respectivos itens 2.2.1 especificando a busca em profundidade e 2.2.2 a busca em largura. A seção 2.3 descreve o algoritmo de Dijkstra. Já na seção 2.4 são descritos os estudos feitos no trabalho de Borba (2014). Por fim, na seção 2.5 são apresentados três trabalhos correlatos, assim como, na seção 2.6 a comparação entre eles.

2.1 TEORIA DOS GRAFOS

A teoria dos grafos estuda objetos combinatórios – os grafos – que são um bom modelo para muitos problemas em vários ramos da matemática, da informática, da engenharia e da indústria (FEOFILOFF; KOHAYAKAWA; WAKABAYASHI; 2011, p. 5). A teoria dos grafos busca estudar estes modelos matemáticos visando solucionar uma gama de problemas. Segundo Rabuske (1992, p.2), a menção mais antiga na área de teoria dos grafos ocorreu no trabalho de Leonhard Paul Euler, no ano de 1736, através do problema das pontes de Königsberg onde existiam sete pontes e havia uma dúvida quanto à possibilidade de percorrer todas elas passando apenas uma vez em cada ponte.

Transferindo este problema de Euler para o papel, pode-se obter um desenho onde nele se é necessário percorrer suas linhas utilizando um lápis com intuito de iniciar e terminar no mesmo ponto sem que se remova o lápis do papel e sem passar duas vezes sobre a mesma linha, conforme demonstrado na Figura 1 que apresenta a cidade de Königsberg e suas 7 pontes, onde hoje é a atual cidade de Kaliningrado.

Figura 1 – Representação do problema das pontes de Königsberg



Fonte: adaptado de Prestes (2013b).

Este problema foi solucionado por Euler, em 1736, dando origem ao que chamamos grafos eulerianos ou “caminhos eulerianos”. Segundo Prestes (2011a, p. 15) o teorema de Euler define que, um grafo conexo G = (V, A) é euleriano, se e somente se, os graus de todos

os nós de G são pares. Partindo deste princípio é possível analisar que não era possível caminhar por todas as pontes de Königsberg e retornar ao ponto inicial sem repetir nenhuma ponte, pois neste modelo de grafo existiam nós ou vértices com grau ímpar, o que entra em conflito com o teorema.

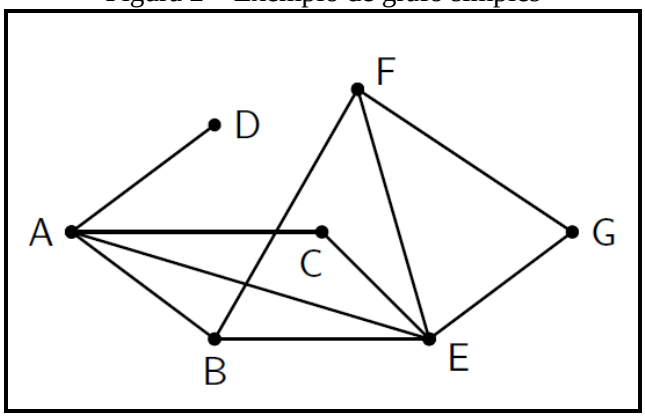
A teoria está também intimamente relacionada com muitos ramos da matemática, incluindo teoria de grupos, teoria de matrizes, análise numérica, probabilidade, topologia e obviamente análise combinatória. O fato é que a teoria dos grafos serve como um modelo matemático para qualquer sistema, envolvendo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

Sendo assim, a teoria dos grafos é uma área abrangente que atua em diversas outras áreas baseando-se na ideia de pontos interligados por linhas. Segundo Rabuske (1992, p. 2) a teoria dos grafos combina estes ingredientes básicos em um rico sortimento de formas e dota estas formas com propriedades flexíveis, fazendo assim, com que esta teoria seja uma ferramenta útil para o estudo de vários tipos de sistemas.

2.1.1 Conceitos fundamentais de um grafo

Como em toda a matemática, a teoria dos grafos está repleta de nomenclaturas e termos técnicos (HOLANDA, 2011, p. 2). Considerando estes termos, a definição de um grafo é descrita por Costa (2011, p. 19) como: um grafo (finito) G é formado por um par (V(G), A(G)) onde V(G) é um conjunto finito não vazio e A(G) uma família de pares não ordenados de elementos, não necessariamente distintos, de V(G). Logo, pode-se afirmar que um grafo é composto de um conjunto V(G) de vértices e outro conjunto de pares de vértices V(A) que constituem as arestas, podendo este conjunto ser vazio ou conter pares de vértices. A Figura 2 apresenta um exemplo de grafo simples.

Figura 2 – Exemplo de grafo simples



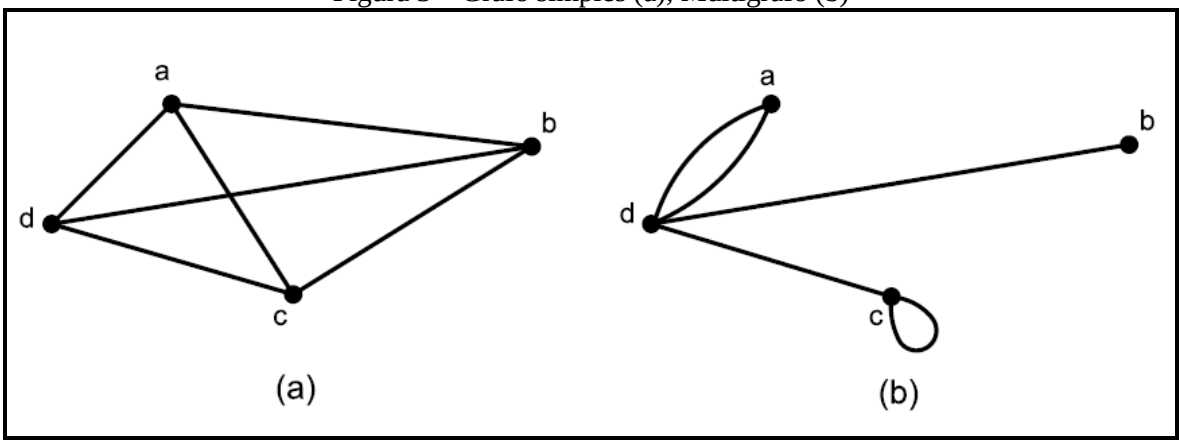
Fonte: adaptado de Holanda (2011).

No exemplo da Figura 2, podemos coletar as seguintes informações do grafo: o conjunto V(G) é constituído por {A, B, C, D, E, F, G} que contempla todo o conjunto finito

de vértices do grafo, o conjunto de arestas A(G) por sua vez é composto por {(A, B), (A, D), (A, C), (A, E), (B, E), (B, F), (C, E), (E, F), (E, G), (F, G)} que constitui o conjunto de pares de vértices deste grafo. Analisando ainda o grafo da Figura 2, Holanda (2011, p. 2) afirma que o grafo é conexo, pois, é possível ir de um vértice a qualquer outro passando por algumas de suas arestas. Ou seja, todo grafo que possua todos os seus vértices do conjunto V(G) interligados de alguma forma por arestas do conjunto A(G) é considerado um grafo conexo.

Um grafo simples, segundo Costa (2011, p. 19) se dá por G que é formado por um par (V(G), A(G)) onde V(G) é um conjunto não vazio e A(G) um conjunto de pares distintos não ordenados de elementos distintos de V(G). Logo, para que um grafo seja simples, este precisa ser composto de vértices V(G) e arestas A(G) com ligações simples, ou seja, onde os pares do conjunto A(G) sejam distintos. A Figura 3 expõe a diferença entre um grafo simples e um multigrafo.

Figura 3 – Grafo simples (a), Multigrafo (b)



Fonte: adaptado de Costa (2011).

Um multigrafo por sua vez é definido por Costa (2011, p. 10) como um grafo (ou digrafo) que contém pelo menos um laço ou um conjunto de arestas paralelas. Um laço é um par de vértices do conjunto de arestas A(G) que é composto por dois vértices V(G) não distintos, ou seja, uma aresta que inicia e termina no mesmo vértice. Já a aresta paralela significa que ao menos dois pares de vértices V(G) do conjunto de arestas A(G) não são distintos, sendo assim, o conjunto de arestas possui dois ou mais pares iguais, ou então duas ou mais ligações entre os mesmos vértices.

2.2 ALGORITMOS DE BUSCA EM GRAFOS

Segundo Mariani (2015), realizar uma busca ou explorar um grafo, significa obter um processo sistemático de como caminhar por seus vértices e arestas. As buscas são utilizadas para solucionar diversos problemas, tais como: encontrar um caminho qualquer entre dois

computadores em uma rede ou até mesmo encontrar o caminho mínimo entre roteadores de internet e provedor. Existem dois algoritmos clássicos referentes a este tema, o de busca em largura (Quadro 1) – em inglês *Breadth First Search* (BFS) – e o de busca em profundidade (Quadro 2) – em inglês *Depth First Search* (DFS) - que são utilizados para visitação de vértices de um grafo, dado um vértice inicial.

2.2.1 Algoritmo BFS

O algoritmo BFS é um algoritmo que parte de um vértice inicial, visita os adjacentes e consequentemente os vértices adjacentes dos seus vizinhos inexplorados, até que encontre o destino ou enquanto for possível atingir os vértices dentro do grafo. O Quadro 1 demonstra o pseudocódigo do algoritmo BFS.

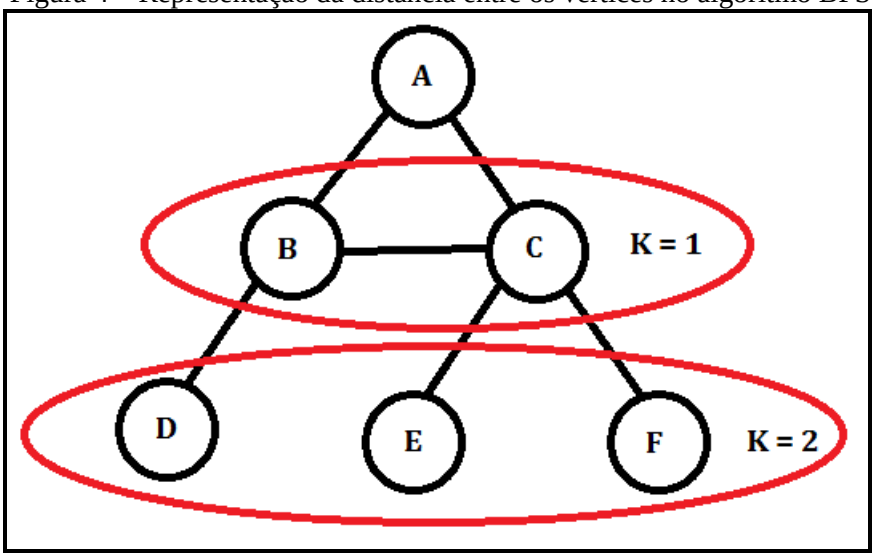
Quadro 1 – Algoritmo da busca em largura (BFS)

```
01. BUSCA-EM-LARGURA(G, s)
02.     > INICIALIZAÇÃO
03.     para cada u ∈ V[G] – {s} faça
04.         cor[u] <- branco
05.         d[u] <- ∞
06.         π[u] <- NIL
07.     cor[s] <- cinza
08.     d[u] <- 0
09.     π[u] <- NIL
10.     Q <- 0
11.     ENQUEUE(Q, s)
12.         enquanto Q ≠ 0 faça
13.             u <- DEQUEUE(Q)
14.             para cada v ∈ Adj[u] faça
15.                 se cor[v] == branco então
16.                     cor[v] <- cinza
17.                     d[v] <- d[u] + 1
18.                     π[v] <- u
19.                     ENQUEUE(Q, v)
20.             cor[u] <- preto
```

Fonte: adaptado de Miyazawa (2012).

Dado um grafo G = (V, E) e um vértice s, chamado de fonte, a busca em largura sistematicamente explora as arestas de G de maneira a visitar todos os vértices alcançáveis a partir de s (KAESTNER, 2013, p. 14). Ainda segundo Kaestner (2013, p. 15) esta busca é dita em largura porque ela expande a fronteira entre vértices conhecidos e desconhecidos de uma forma uniforme ao longo da fronteira. Logo, o algoritmo descobre todos os vértices com distância k de s antes de descobrir qualquer vértice de distância k + 1. A representação desta expansão pode ser identificada na Figura 4.

Figura 4 – Representação da distância entre os vértices no algoritmo BFS

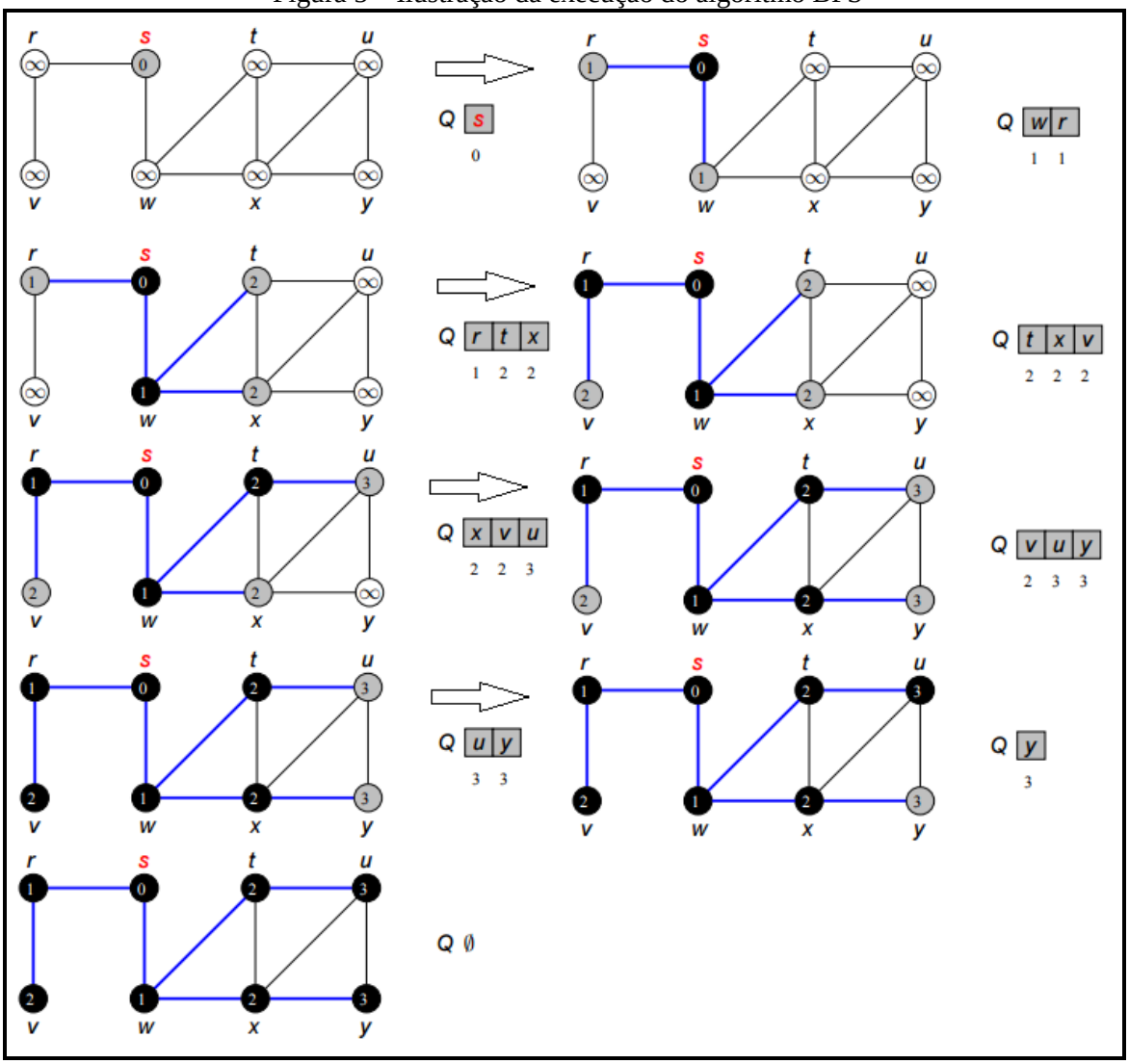


Fonte: adaptado de Kaestner (2013).

Observando a Figura 4, pode-se identificar que o k representa os níveis da árvore que o algoritmo BFS constrói durante sua execução. Sendo assim, uma representação textual da árvore deste grafo seria <a <b <d>, c <e, f>>>, dada esta representação pode-se verificar que a ligação entre os vértices B e C não aparece na árvore e estes dois vértices estão em ramos diferentes da mesma. Isto ocorre devido ao processo expansivo do algoritmo, que ao atingir os vértices B e C através do vértice A, considera os dois como já visitados e estes por sua vez só visitam os que não são conhecidos.

Uma execução deste algoritmo sob um grafo pode ser visualizada passo a passo na Figura 5, onde é possível acompanhar desde os estados dos vértices, com as cores: branco, que indica um vértice não conhecido, cinza que se trata de um vértice conhecido e preto que representa um vértice que já foi totalmente explorado. Quando um vértice é visitado pela primeira vez, sua cor é modificada de branco para cinza, quando todos os vértices adjacentes a um vértice cinza são visitados, este se torna preto. Na Figura 5 também é possível acompanhar o vetor de roteamento representado por Q, onde se adiciona um vértice quando este se torna conhecido e o remove do vetor quando este já se encontra explorado.

Figura 5 – Ilustração da execução do algoritmo BFS



Fonte: adaptado de Miyazama (2012).

Seguindo estes conceitos e analisando a Figura 5, pode-se avaliar que o algoritmo BFS efetua a busca no grafo de forma expansiva, onde todos os vértices mais próximos são adicionados ao vetor de roteamento por ordem de visita se o vértice em questão ainda não foi visitado, seguindo este mesmo procedimento para cada vértice do vetor até que sejam atingidos todos os vértices conexos.

2.2.2 Algoritmo DFS

O algoritmo DFS realiza uma busca no grafo partindo de um vértice inicial, seguindo para um dos adjacentes e consequentemente para o próximo vértice adjacente de seu vizinho, procurando seguir sempre um caminho até esgotar todas as opções. Logo, quando não existem mais vértices inexplorados no ramo percorrido é executado o processo de *backtracking*, que seria o processo de retornar até o último vértice que ainda possui adjacentes inexplorados para que então se repita o procedimento, até que todo o grafo seja percorrido.

Caso o grafo seja desconexo, quando todos os caminhos conexos para se percorrer estiverem finalizados, o algoritmo escolherá um outro vértice ainda não visitado desconexo e então reiniciará o processo até que todo o grafo seja explorado. O Quadro 2 demonstra um pseudocódigo deste algoritmo.

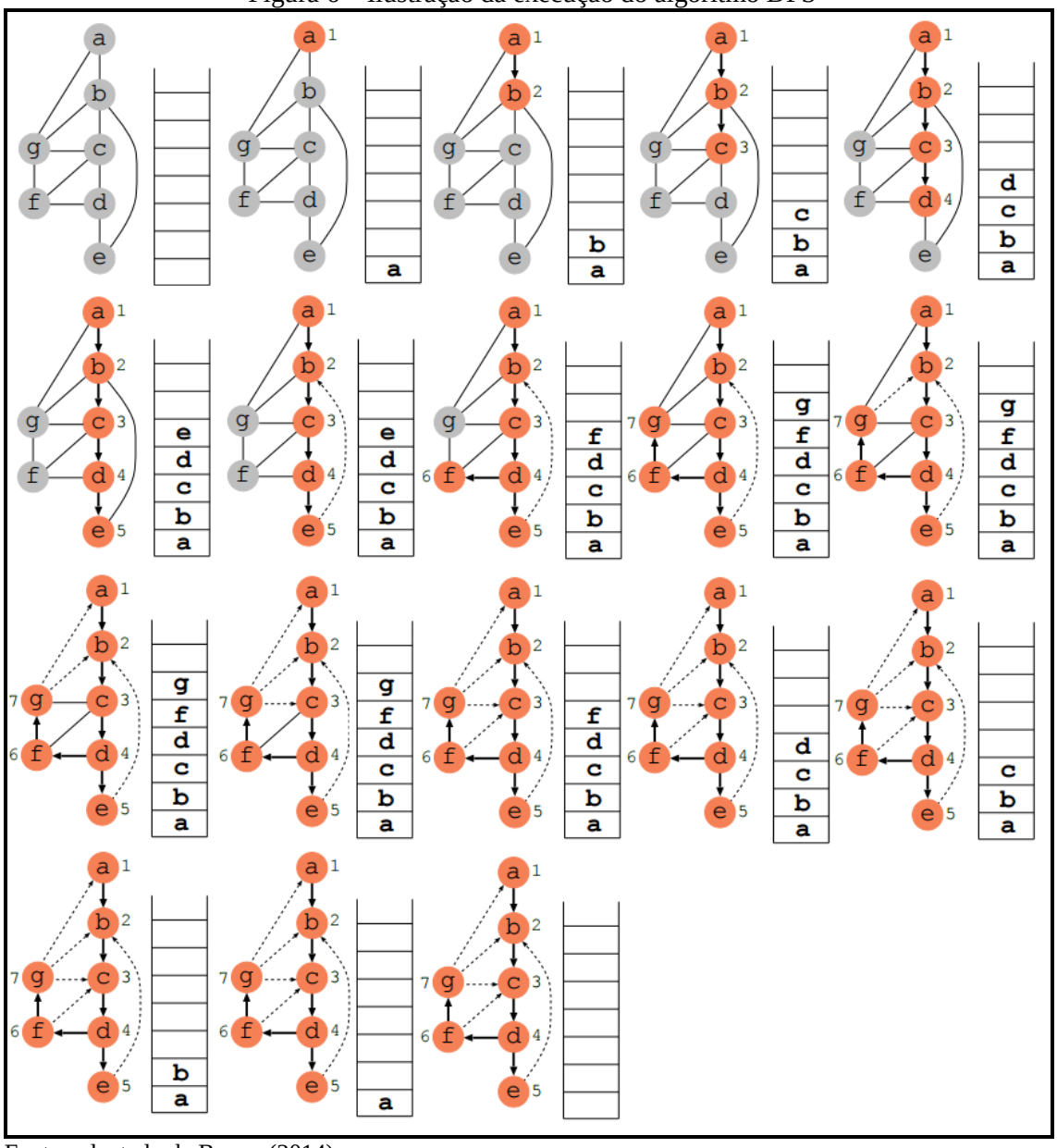
Quadro 2 – Algoritmo de busca em profundidade (DFS)

```
01. DFS(G)
02.     para cada vértice u ← V[G]
03.         cor[u] ← BRANCO
04.     tempo ← 0
05.     para cada vértice u ∈ V[G]
06.         se cor[u] = BRANCO
07.             DFS-VISIT(u)
08. DFS-VISIT(u)
09.     cor[u] ← CINZA
10.     tempo ← tempo + 1
11.     d[u] ← tempo
12.     para cada vértice v ∈ Adj(u)
13.         se cor [v] = BRANCO
14.             DFS-VISIT(v)
15.     cor[u] ← PRETO
16.     f[u] ← tempo ← (tempo + 1)
```

Fonte: adaptado de Lobo (2012a).

O algoritmo DFS conta com um marcador de tempo de abertura e fechamento para cada vértice, que indica o tempo em que este vértice foi visitado e o tempo em que foi explorado de fato. O algoritmo também sinaliza com cores para indicar os estados dos vértices, tais como: branco para vértices não conhecidos, cinza para vértices visitados e preto para os vértices já explorados. Sempre que um vértice é visitado este recebe um tempo de abertura, que indica a quantos vértices de distância ele já processou até ali desde a origem. Assim como quando este vértice é totalmente explorado recebe um tempo de fechamento. A Figura 6 apresenta a execução passo a passo do DFS, demonstrando os estados dos vértices e a sua pilha de execução.

Figura 6 – Ilustração da execução do algoritmo DFS



Fonte: adaptado de Bueno (2014).

A partir da Figura 6 pode-se observar que a estrutura de dados utilizada pelo algoritmo DFS é uma pilha. Apesar dos algoritmos BFS e DFS terem a mesma funcionalidade de atingirem todos os vértices partindo de uma origem, ambos algoritmos possuem algumas características que os diferenciam. Uma delas está nas estruturas de dados auxiliares das duas estratégias, onde o BFS utiliza uma fila e o DFS uma pilha. Pode-se citar que no DFS o vértice inicial é escolhido pelo algoritmo, quando no BFS este normalmente é escolhido pelo usuário. O algoritmo DFS também se diferencia por efetuar buscas em grafos desconexos, visitando todos os vértices, quando o BFS atinge somente os vértices do grafo conexo a partir da origem.

2.3 ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra que foi publicado em 1959 por Edsger Dijkstra, é um algoritmo para caminhamento em grafos que soluciona o problema de caminho mínimo. O pseudocódigo do algoritmo pode ser visto na Quadro 3.

Quadro 3 – Algoritmo de Dijkstra

```
01. INICIALIZAÇÃO(G,s)
02.     para cada vertex v ∈ V[G]
03.         faça d[v] ← ∞
04.             ∏[v] ← NIL
05.     d[s] ← 0;
06. RELAXAMENTO(u, v, w)
07.     if d[v] > d[u] + w(u,v)
08.         then d[v] ← d[u] + w(u,v)
09.             ∏[v] ← u
10. DIJKSTRA(G, w, s)
11.     INICIALIZAÇÃO(G,s)
12.     S ← ∅
13.     Q ← V[G]
14.     enquanto Q ≠ ∅
15.        faça u ← EXTRACT-MIN(Q)
16.            S ← S ∪ [u]
17.            para cada vertex v ∈ Adj[u]
18.                faça RELAXAMENTO(u,v,w)
```

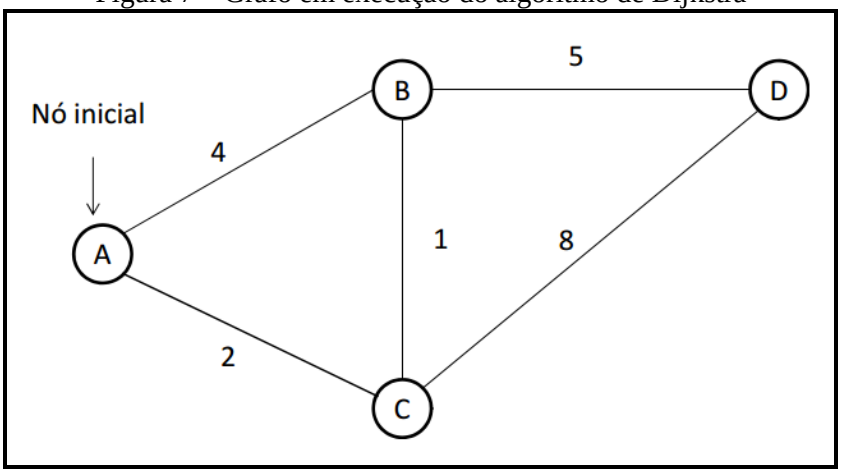
Fonte: adaptado de Lobo (2012b).

Segundo Rabuske (1992, p. 65), o algoritmo de Dijkstra tem a seguinte definição:

> Seja um grafo G(V, E) e uma função distância L que associe cada aresta (v, w) a um número real negativo L(v, w) e também um vértice fixo v0 em V, chamado fonte. O problema consiste em se determinar os caminhos de v0 para cada vértice v de G, de tal forma que a somatória das distâncias das arestas envolvidas em cada caminho seja mínima. Isto é equivalente a determinar um caminho v0, v1,....,vk tal que a somatória das distâncias das arestas envolvidas em cada caminho seja mínimo (RABUSKE, 1992, p. 65).

Sendo assim, a execução do algoritmo inicialmente atribui a todos os pares de vértices uma distância infinita, exceto para o vértice inicial. Logo, todos os vértices são marcados como não visitados e o vértice inicial é marcado como atual. A Figura 7 apresenta um exemplo de grafo onde pode-se visualizar a seleção do vértice A como ponto inicial do processamento.

Figura 7 – Grafo em execução do algoritmo de Dijkstra



Fonte: adaptado de Castro (2016).

Para o vértice atual são calculadas as distâncias entre ele e seus vértices vizinhos não visitados, logo, é realizada uma comparação entre as distâncias dos diferentes caminhos para se atingir um mesmo vértice, se esta distância total calculada entre o vértice atual e o destino for menor que a distância que foi encontrada anteriormente, então o vértice atual passa a ser o pai, ou então o caminho para aquele vértice destino e a distância entre os dois é atualizada como o caminho mínimo até este vértice. A Figura 8 apresenta quatro quadros com a execução passo a passo do Dijkstra.

Figura 8 – Execução do algoritmo de Dijkstra

Vértices	Passo 1	Passo 2	Passo 3	Passo 4
A	0, A	*	*	*
B	4, A			
C	2, A			
D	∞			

Vértices	Passo 1	Passo 2	Passo 3	Passo 4
A	0, A	*	*	*
B	4, A	3, C		
C	2, A	2, C	*	*
D	∞	10, C		

Vértices	Passo 1	Passo 2	Passo 3	Passo 4
A	0, A	*	*	*
B	4, A	3, C	3, C	*
C	2, A	2, C	*	*
D	∞	10, C	8, B	

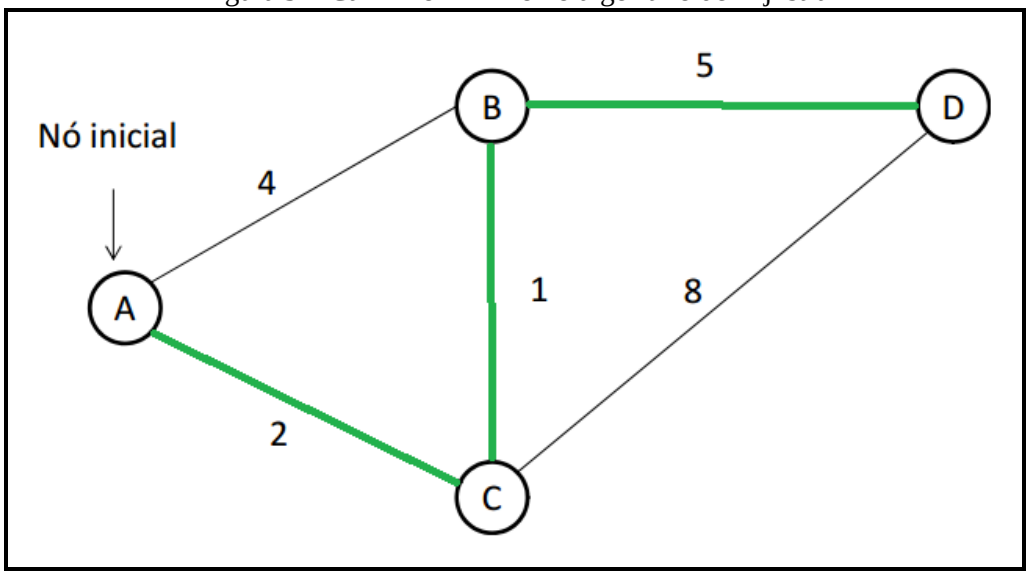
Vértices	Passo 1	Passo 2	Passo 3	Passo 4
A	0, A	*	*	*
B	4, A	3, C	3, C	*
C	2, A	2, C	*	*
D	∞	10, C	8, B	8, B

Fonte: adaptado de Castro (2016).

Ainda na Figura 8, no passo 1, são atribuídas as distâncias iniciais conhecidas a partir do vértice A para seus vértices vizinhos. No quadro ao lado, no passo 2, pode-se verificar que o processamento continua a partir do vizinho com o menor custo, no caso, o vértice C com custo de 2. No passo 2 também é possível identificar que o caminho (A, B) com custo 4 foi alterado para (C, B) de custo 3 e o vértice D foi visitado pela primeira vez através do vértice C. No passo 3 é selecionado o vértice de menor custo novamente, que neste caso é o vértice B

e é descoberto um caminho mais curto para o vértice D, alterando o caminho (C, D) de custo 10 para o caminho (B, D) de custo 8. Por fim, no passo 4 se verifica que o vértice D é o último a ser processado pois através deste vértice não é possível encontrar nenhum outro caminho menor que os demais já encontrados. A Figura 9 apresenta o resultado final do caminho mínimo entre o vértice de origem A para todos os demais vértices do grafo.

Figura 9 – Caminho mínimo no algoritmo de Dijkstra



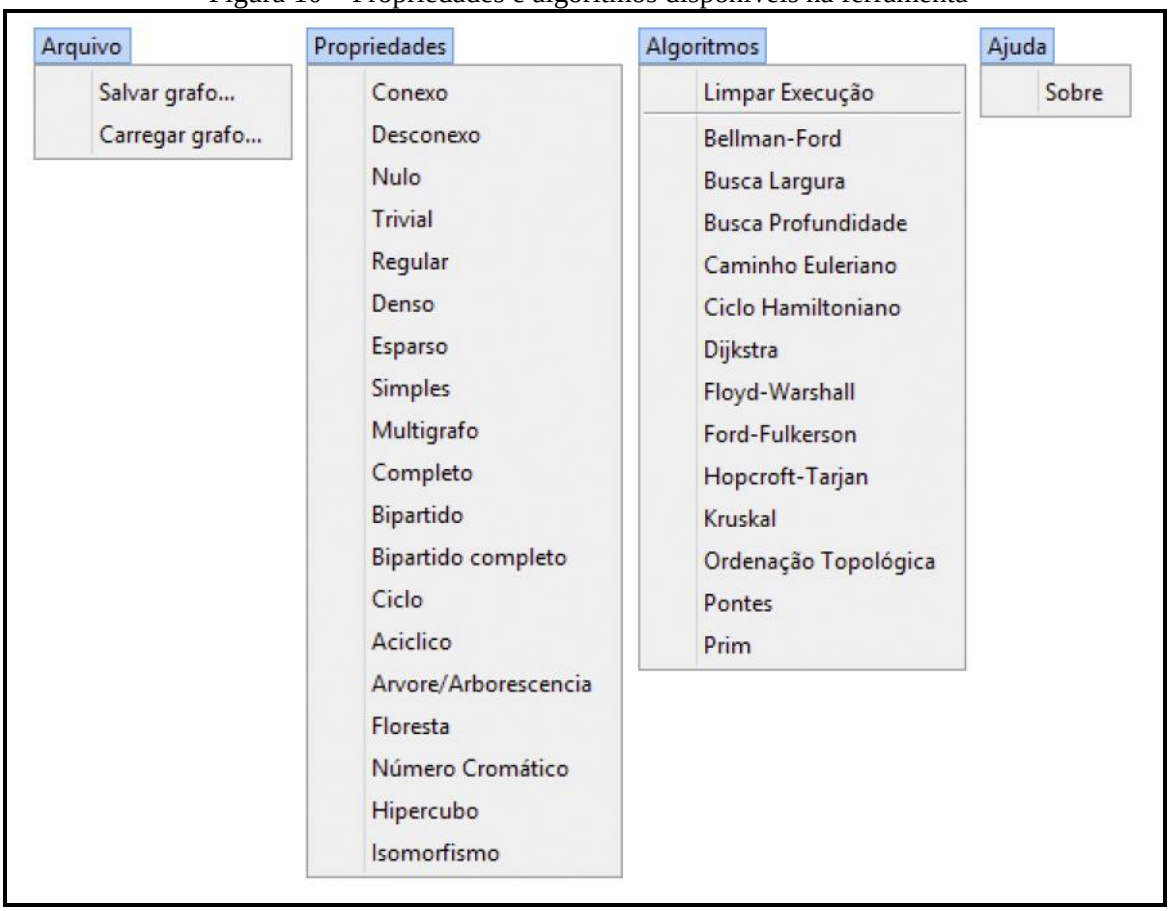
Fonte: adaptado de Castro (2016).

Desta forma, o algoritmo de Dijkstra calcula o caminho mínimo entre o vértice inicial e cada um dos demais vértices atingíveis de um grafo. Caso o grafo em questão seja desconexo, o algoritmo não atingirá todos os vértices, encontrando assim só o caminho mínimo do vértice origem para todos os demais conexos.

2.4 FURB GRAPHS: UMA APLICAÇÃO PARA TEORIA DOS GRAFOS

O trabalho de Borba (2014) propôs novas funcionalidades e melhorias para a ferramenta FURB Graphs que teve seu início no trabalho de Zatteli (2010). A ferramenta foi desenvolvida em Java e foram adicionados alguns testes de propriedades, tais como: número cromático, hipercubo e isomorfismo a outro grafo. A Figura 10 apresenta todas as propriedades e algoritmos que a ferramenta FURB Graphs suporta.

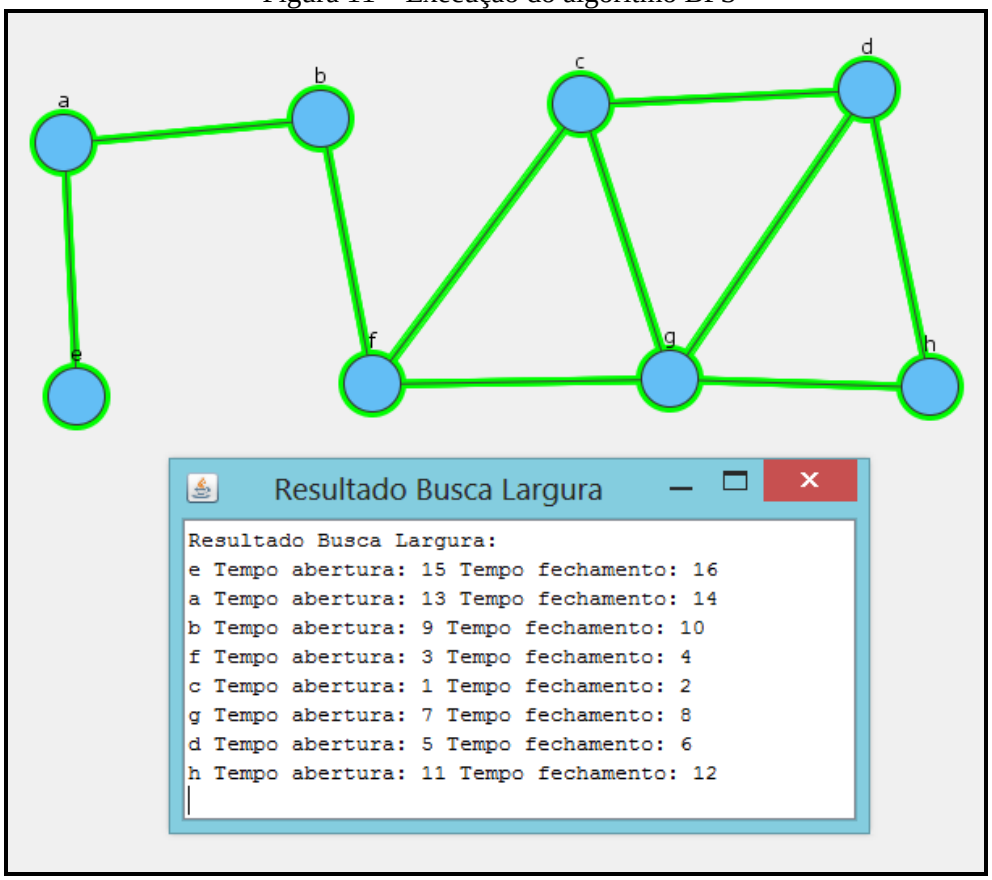
Figura 10 – Propriedades e algoritmos disponíveis na ferramenta



Fonte: Borba (2014).

Levando em consideração todas as funcionalidades da ferramenta FURB Graphs apresentadas na Figura 10, pode-se identificar que os algoritmos mais tradicionais e relevantes de teoria dos grafos já estão implementados. Porém, o foco da ferramenta era de executar de fato os algoritmos selecionados sobre os grafos desenhados pelo usuário, sem o intuito de demonstrar como ocorreram os processos ou então os estados e características dos vértices. Logo, os algoritmos apenas são executados e então retornam um resultado final, tal como pode ser visto ao se executar o algoritmo BFS na Figura 11.

Figura 11 – Execução do algoritmo BFS



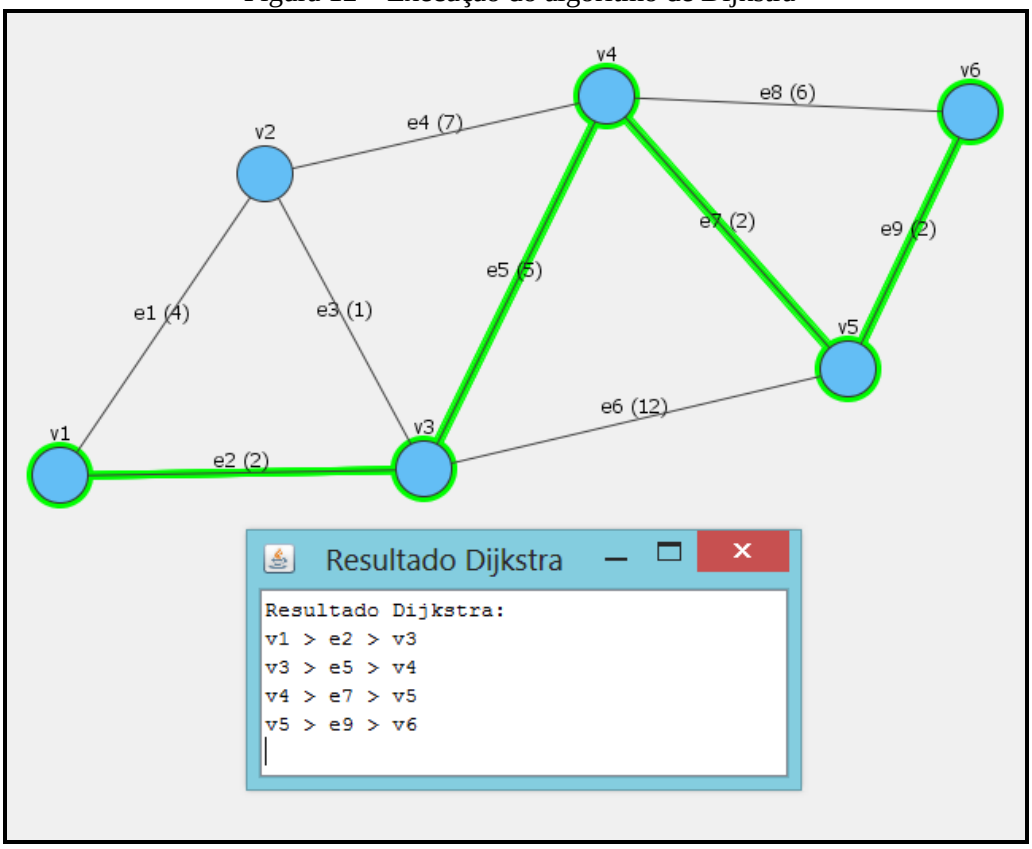
Fonte: adaptado de Borba (2014).

Avaliando o retorno da execução do algoritmo BFS na ferramenta de Borba (2014), pode-se identificar que a representação gráfica final do resultado é sucinta e direta, demonstrando o resultado simplificado em uma caixa de texto e colorindo todos os vértices e arestas do grafo em questão. Esta representação por sua vez pode trazer uma interpretação errônea do algoritmo, pois, se faz entender de que todos os caminhos do grafo foram de fato utilizados na execução do algoritmo BFS. Entretanto, nesta representação gráfica não se consegue identificar a árvore que foi gerada sobre esta execução, que é uma característica importante da BFS.

Outro algoritmo clássico que pode ser citado como exemplo é o algoritmo de Dijkstra para caminhamento em grafos, que também possui um comportamento similar na ferramenta de Borba (2014). Ao executar o algoritmo é requisitado ao usuário para que selecione um vértice inicial e um vértice final, após feita a escolha da origem e destino a ferramenta apresenta uma caixa de texto contendo o resultado descrito do caminho mínimo encontrado entre o vértice inicial e final. Assim, como em sua representação gráfica são coloridos na tela todos os vértices e arestas que fazem parte deste caminho mínimo.

O caminho que é apresentado contempla apenas o caminho menor entre estes dois vértices e não o proposto pelo algoritmo, que deve ser o caminho mínimo da origem para cada vértice do grafo. Na Figura 12 pode-se visualizar tal como é representada a execução deste algoritmo.

Figura 12 – Execução do algoritmo de Dijkstra



Fonte: adaptado de Borba (2014).

Analisando a Figura 12, identifica-se que o algoritmo de Dijkstra em questão não calculou o caminho mínimo entre o vértice v1 e v2, pois, estes não se encontravam no trajeto mínimo entre v1 e v6 que são os vértices de origem e destino da execução deste grafo. Nesta representação gráfica tende-se a compreender que o algoritmo calcula o caminho mínimo entre dois pontos escolhidos, quando o objetivo de fato é calcular o custo mínimo entre o vértice de origem para todos os demais vértices atingíveis do grafo.

2.5 TRABALHOS CORRELATOS

Ferramentas que utilizam metodologias de ensino-aprendizagem para teoria dos grafos são amplamente aperfeiçoadas por pesquisadores, que procuram melhorar os meios de abstrair o algoritmo e transmitir o funcionamento dos grafos e suas aplicações. É possível citar três trabalhos relacionados a esta temática. A ferramenta A-Graph de Lozada (2014), permite a criação de grafos de forma visual, assim como permite a execução de diversos algoritmos,

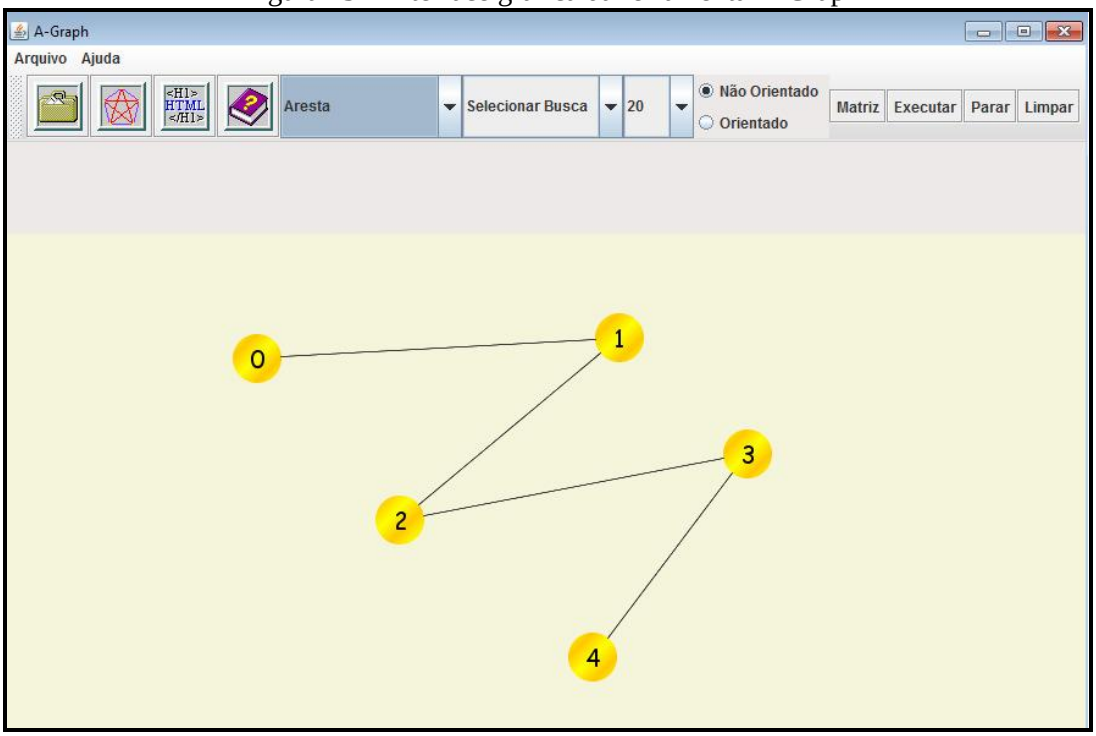
como a busca em largura e a busca em profundidade. O trabalho de Sangiorgi (2004) apresenta a ferramenta Rox que também facilita o entendimento dos grafos, seus algoritmos e aplicações, permitindo por exemplo a modelagem de uma topologia de rede com base no grafo criado. No trabalho de Santos e Costa (2007) a ferramenta TBC – Grafos/Web tem o foco no detalhamento, passo a passo de algoritmos de busca, caminhamento em grafos e arvore geradora mínima.

2.5.1 A-Graph

Conforme Lozada (2014, p. 2), o A-Graph foi completamente implementado na linguagem de programação Java utilizando os recursos de computação gráfica bidimensional suportada pela API Java2D. A aplicação roda em modo *stand-alone*, o que significa que é necessário somente o executável para que a mesma funcione, ficando assim sem dependências de terceiros.

O aplicativo A-Graph (LOZADA, 2014) oferece uma interface de interação com a qual os estudantes podem desenhar grafos, com ou sem direção, definindo livremente a sua geometria e consequentemente a topologia subjacente. Permite a execução dos algoritmos de busca em grafos, de maneira didática, inclusive implementando a heurística de escolha livre do vértice inicial a partir do qual a busca se iniciará (LOZADA, 2014, p. 6). A interface gráfica da ferramenta permite o desenho livre dos grafos, conforme mostra a Figura 13.

Figura 13 – Interface gráfica da ferramenta A-Graph



Fonte: Lozada (2014).

A ferramenta foi elaborada com o propósito de ser utilizada como suporte ao ensino-aprendizado da disciplina de teoria dos grafos. Ela tem como foco a execução dos algoritmos de busca em largura (BFS) e busca em profundidade (DFS), assim como também exibe as matrizes de incidência e de adjacência. Na Figura 14 pode-se observar a funcionalidade de exibição das matrizes de incidência e de adjacência.

Figura 14 – Matriz de incidência e matriz de adjacência

	v0	v1	v2	v3	v4
v0	0	1	0	0	0
v1	1	0	1	0	0
v2	0	1	0	1	0
v3	0	0	1	0	1
v4	0	0	0	1	0

	0⇔1	3⇔4	2⇔3	1⇔2
v0	1	0	0	0
v1	1	0	0	1
v2	0	0	1	1
v3	0	1	1	0
v4	0	1	0	0

Fonte: Lozada (2014).

A ferramenta também permite a substituição dos vértices por imagens diversas, tais como roteadores, onde é possível a modelagem de problemas em outras áreas, como a área de redes. Sendo assim, a ferramenta atinge o seu propósito de executar os algoritmos BFS e DFS, permitir a criação dos grafos e dígrafos de forma livre e apresentar as matrizes de incidência e de adjacência, tornando-se válida para auxiliar no ensino-aprendizagem de grafos.

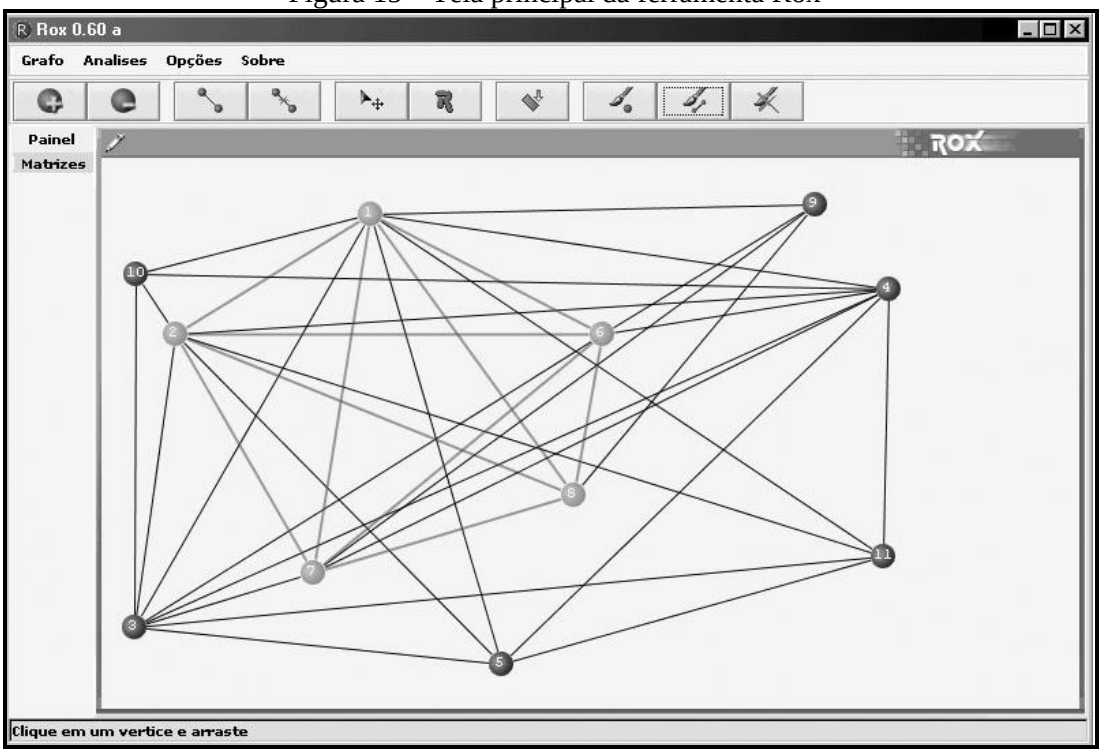
2.5.2 Rox

A ferramenta Rox (SANGIORGI, 2004) se apresenta como um meio de comunicação entre o usuário e a representação em um grafo, com intuito de diminuir a abstração dos algoritmos de teoria dos grafos.

> O Rox, possibilita a construção e manipulação de grafos simples, bem como execução de algoritmos feitos para serem aplicados nos mesmos. Foi construída para ser utilizada tanto por professores que desejem apresentar a disciplina de teoria dos grafos e que queiram, porventura, fazê-lo de uma forma lúdica, quanto por alunos que desejem visualizar uma aplicação prática dos problemas expostos em sala e construir algoritmos para resolução, ou simples análise, dos mesmos (SANGIORGI, 2004, p. 1).

O Rox foi construído em Java 2 SDK Standard Edition. Conforme Sangiorgi (2004) os menus principais permitem a "criação e remoção de vértices e arestas, movimentação de vértices, geração de um grafo aleatório, recarregamento e execução do último algoritmo, e ainda coloração manual de vértices e arestas", conforme pode ser visualizado na Figura 15.

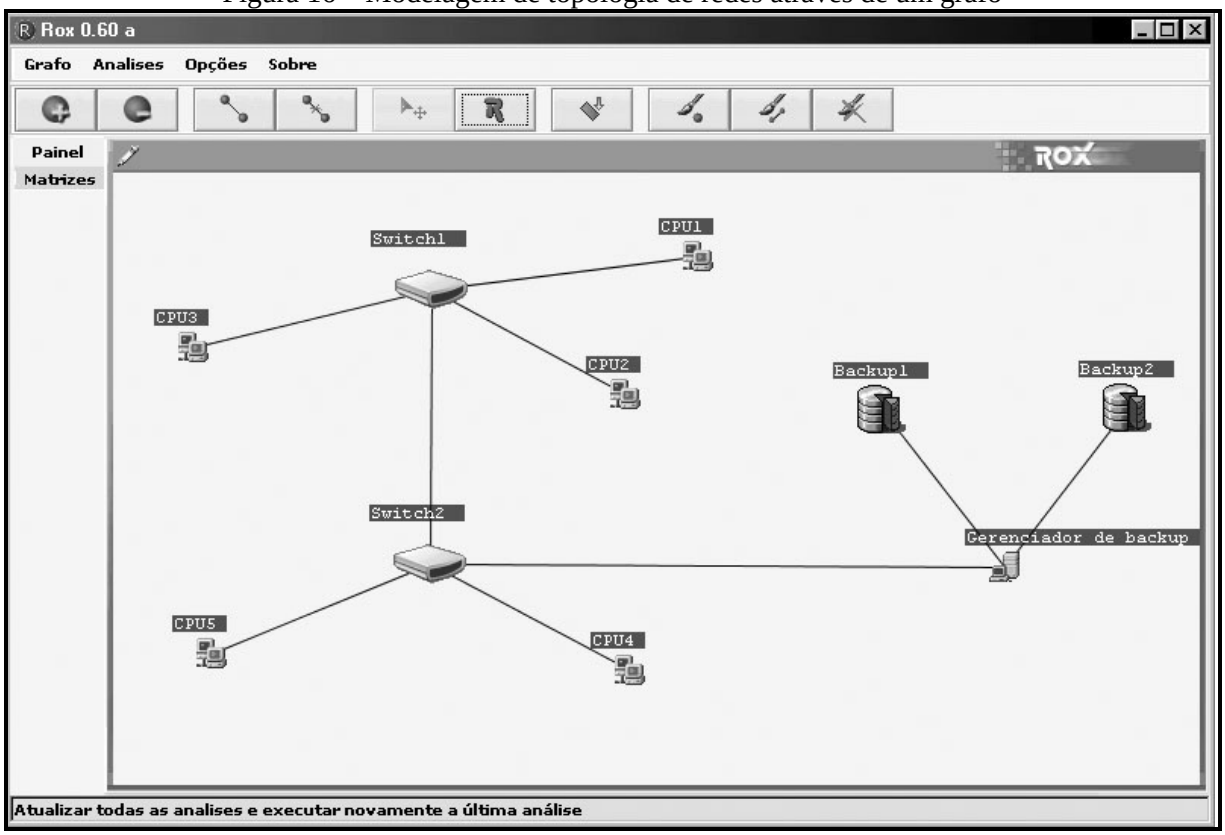
Figura 15 – Tela principal da ferramenta Rox



Fonte: Sangiorgi (2004).

A ferramenta apresenta diversas outras funcionalidades, tais como: análise de grafos bipartidos e ciclos Hamiltonianos. Também existe a possibilidade de trocar as imagens dos vértices de modo a possibilitar um melhor entendimento de uma determinada situação. O Rox possibilita que o usuário modele situações tais como topologias de redes de computadores (SANGIORGI, 2004, p. 4). Esta representação de modelagem de problemas de outras áreas como a de redes pode ser visualizada na Figura 16.

Figura 16 – Modelagem de topologia de redes através de um grafo



Fonte: Sangiorgi (2004).

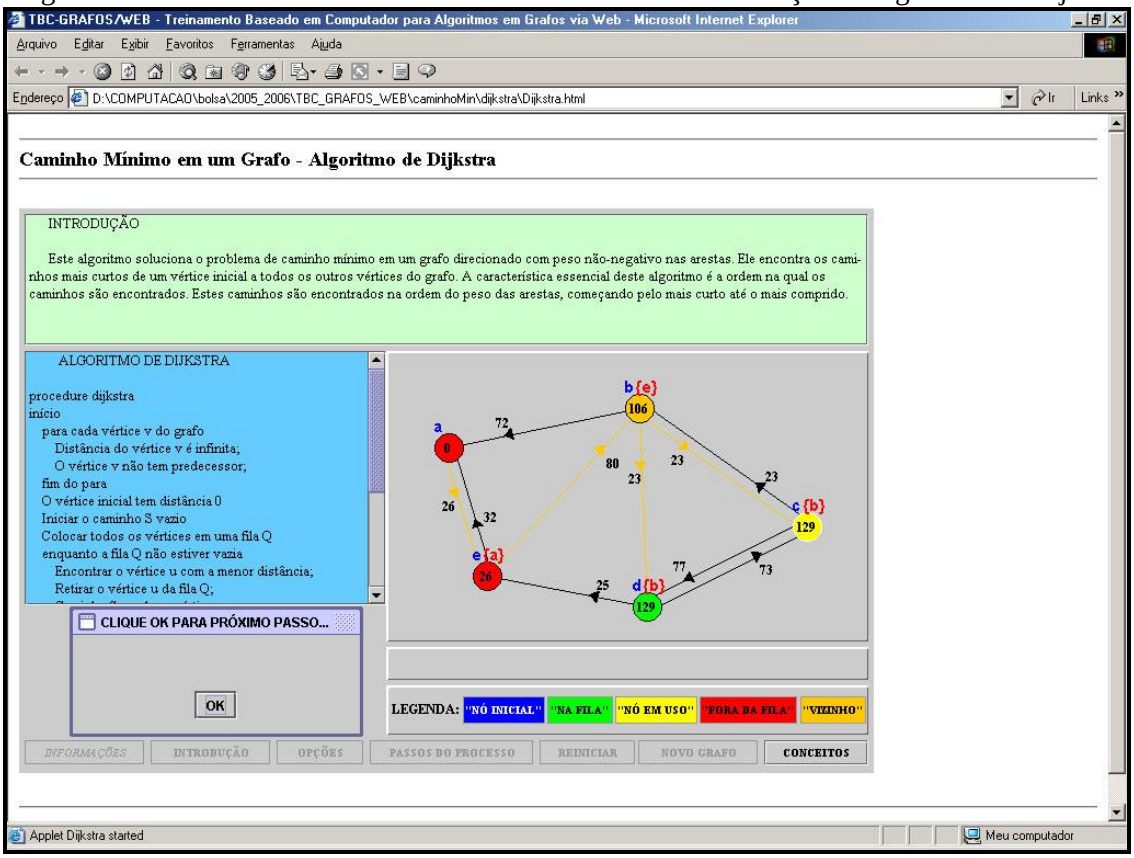
Uma das funcionalidades mais interessante do Rox (SANGIORGI, 2004) é a possibilidade que o usuário tem de criar seus próprios algoritmos, dentro de um padrão que a ferramenta compreenda, compila e executa. Desta forma, pode-se avaliar o entendimento que o estudante teve dos algoritmos e testá-los em execução na ferramenta, assim como aplicar ajustes nos algoritmos e fazer alterações nas estratégias, melhorando assim a relação da representação gráfica com o algoritmo executado.

2.5.3 TBC – Grafos/Web

O trabalho de Santos e Costa (2007), apresenta a ferramenta TBC – Grafos/Web, que foi desenvolvida na plataforma web e tem o objetivo de ensinar os algoritmos utilizados na disciplina de teoria dos grafos. Segundo Santos e Costa (2007, p. 2) a ferramenta "analisa tópicos básicos de programação, englobando conteúdo teórico sintético, representando um repositório didático com visualização básica passo a passo. Isso facilita o entendimento das informações apresentadas e economiza o tempo gasto em transcrever explicações de forma tradicional (quadro-negro/caderno), visando maior resolução de exercícios de aplicação e fixação". Na Figura 17 é possível visualizar a interface da ferramenta em execução do algoritmo de Dijkstra. Pode-se perceber que a ferramenta tem o objetivo de ser bem instrutiva,

quanto a fornecer desde informações sobre o funcionamento do determinado algoritmo, descrição detalhada do código fonte, até legendas para o acompanhamento das etapas, processos e status dos vértices, assim como um botão para controlar os passos de execução do algoritmo.

Figura 17 – Interface da ferramenta TBC – Grafos/Web em execução do algoritmo de Dijkstra



Fonte: Santos e Costa (2007).

A ferramenta foi desenvolvida totalmente em Java, utilizando a IDE NetBeans. A ferramenta funciona sobre um *applet*, que é executado quando se seleciona alguma das funcionalidades da ferramenta. Segundo Santos e Costa (2007) "o TBC-Grafos/Web visa abordar alguns dos mais significativos algoritmos em grafos, relacionados a busca em grafos (algoritmos em profundidade e em largura), árvore geradora mínima (algoritmos de Kruskal e de Prim) e caminho mínimo entre vértices (algoritmos de Dijkstra e Bellman-Ford)". Além do usuário ter como selecionar estes algoritmos para execução e aprendizado, ele também pode criar o seu próprio grafo dentro da ferramenta e acompanhar os processos passo a passo.

Tendo estes temas como foco, o TBC-Grafos/Web atingiu o objetivo de trazer um ambiente de auxílio a disciplina de teoria dos grafos, permitindo a interação do usuário com o grafo, explanando os detalhes de cada algoritmo através de textos informativos, detalhamento do código fonte e controle de status de vértices e de etapas do algoritmo.

2.5.4 Comparativo entre as características dos trabalhos correlatos

A partir das informações obtidas com os trabalhos descritos nas seções anteriores, foi elaborado o Quadro 4 com as principais características.

Quadro 4 – Características dos trabalhos correlatos

Características / trabalhos correlatos	Lozada (2014)	Santos e Costa (2007)	Sangiorgi (2004)
permite a criação de grafos livremente	Sim	Sim	Sim
suporta dígrafos	Sim	Sim	Sim
permite trocar as imagens dos vértices	Sim	Não	Sim
executa busca em grafos (BFS, DFS)	Sim	Sim	Não
executa árvore geradora mínima (Kruskal e Prim)	Não	Sim	Não
executa caminho mínimo entre vértices (Dijkstra e Bellman-Ford)	Sim	Sim	Não
analisa ciclos Hamiltonianos	Não	Não	Sim
analisa grafos bipartidos	Não	Não	Sim
apresenta informativos referente aos algoritmos	Não	Sim	Não
apresenta material teórico quanto aos algoritmos	Não	Sim	Não
apresenta recursos de apoio ao aprendizado dos algoritmos (legendas, código fonte)	Sim	Sim	Sim
permite carregar um código fonte de terceiro para utilizar na execução	Não	Não	Sim
possui plataforma de ampla disponibilidade (web)	Não	Sim	Não

A partir do Quadro 4 avalia-se que a ferramenta TBC-Grafos/Web (SANTOS; COSTA, 2007) é a mais completa, pois apresenta um ambiente mais didático e dinâmico. Nela são disponibilizadas várias informações que realmente auxiliam no aprendizado. Já o trabalho Rox (SANGIORGI, 2004) possui diferencial por permitir a execução do código fonte do usuário, o que ajuda na validação do código construído além de auxiliar na compreensão dos processos. Porém, o mesmo não executa os algoritmos mais complexos como os de busca e de caminho mínimo em grafos. O A-Graph (LOZADA, 2014) apresenta diversos algoritmos relevantes. Porém, em comparação aos outros dois trabalhos não apresenta um diferencial em relação ao ensino-aprendizagem para teoria dos grafos.

3 DESENVOLVIMENTO DA FERRAMENTA

Neste capítulo é abordado o desenvolvimento da ferramenta FURB Graphs voltada ao ensino-aprendizagem de teoria dos grafos. Primeiramente são apresentados os requisitos da aplicação seguido da sua especificação, em seguida, é apresentada a implementação, onde são comentadas técnicas e ferramentas utilizadas durante o desenvolvimento da aplicação e, posteriormente, a operacionalidade da aplicação. Por fim, são discutidos os testes e resultados obtidos.

3.1 REQUISITOS

A ferramenta desenvolvida atende aos seguintes requisitos:

a) permitir que o usuário visualize os atributos de vértices e das arestas de um grafo através da interface gráfica adaptada do trabalho de Borba (2014) (Requisito Funcional – RF);
b) permitir que o usuário controle o passo a passo (avance e retroceda) da execução dos algoritmos BFS, DFS e Dijkstra (RF);
c) ser implementado utilizando linguagem de programação Java (Requisito não funcional – RNF);
d) deve se utilizar o ambiente de desenvolvimento Eclipse (RNF);
e) deve se utilizar a interface gráfica fornecida no trabalho de Borba (2014) (RNF).

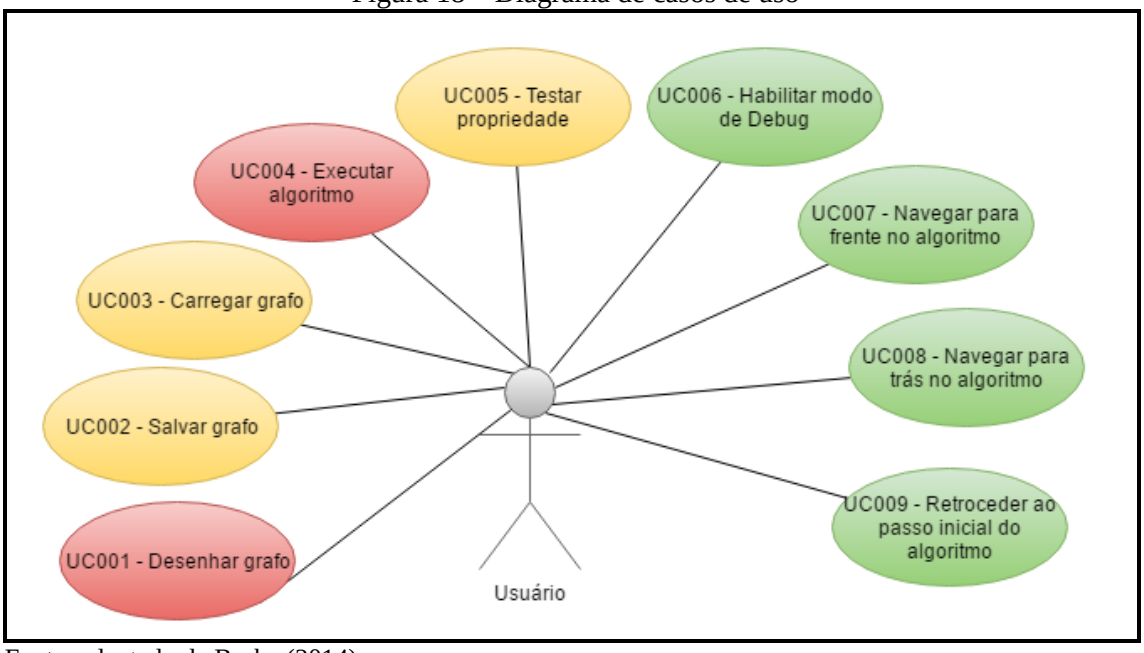
3.2 ESPECIFICAÇÃO

Esta seção apresenta a especificação do FURB Graphs através do uso dos conceitos de orientação a objetos e da *Unified Modeling Language* (UML), utilizando a plataforma online Draw.io. Neste trabalho foram elaborados os diagramas de caso de uso, de classes e de atividades, sendo descritos nas próximas seções.

3.2.1 Diagramas de caso de uso

A Figura 18 exibe o diagrama de casos de uso com as ações fundamentais disponibilizadas pela aplicação para manipulação do grafo.

Figura 18 – Diagrama de casos de uso



Fonte: adaptado de Borba (2014).

Na Figura 18 pode-se verificar que existem diversos casos de uso representados na cor amarela, tal como `UC002 - Salvar grafo`, `UC003 - Carregar grafo` e `UC005 - Testar propriedade`. Tais casos de uso pertencem somente ao trabalho de Borba (2014) e não foram alterados, portanto, não são abordados neste trabalho. Os casos de uso que estão representados em vermelho, `UC001 - Desenhar grafo` e `UC004 - Executar algoritmo` sofreram correções e melhorias rápidas de desempenho, porém, suas funcionalidades chaves pertencem ao trabalho de Borba (2014), logo, estes não são abordados neste trabalho.

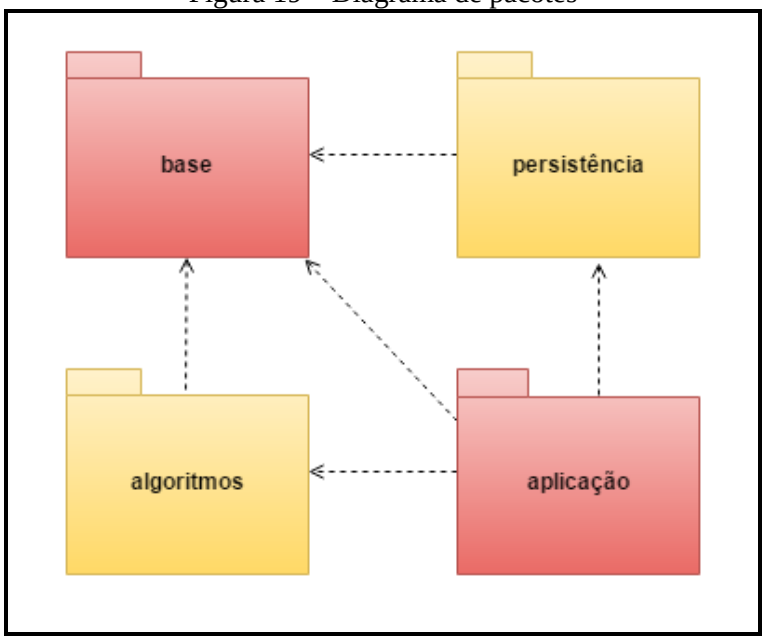
O detalhamento dos casos de uso da ferramenta pode ser consultado no Apêndice A. Nele estão descritas as pré-condições, os cenários e os fluxos alternativos de cada caso de uso.

3.2.2 Diagrama de classes

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

Este trabalho utilizou a mesma estrutura de pacotes do trabalho de Borba (2014) somente aplicando a elas, melhorias e novas funcionalidades. Os pacotes que constituem a aplicação são: `base`, `persistência`, `algoritmos` e `aplicação`. A Figura 19 apresenta a visão macro de como os pacotes estão dispostos na aplicação.

Figura 19 – Diagrama de pacotes



Fonte: adaptado de Borba (2014).

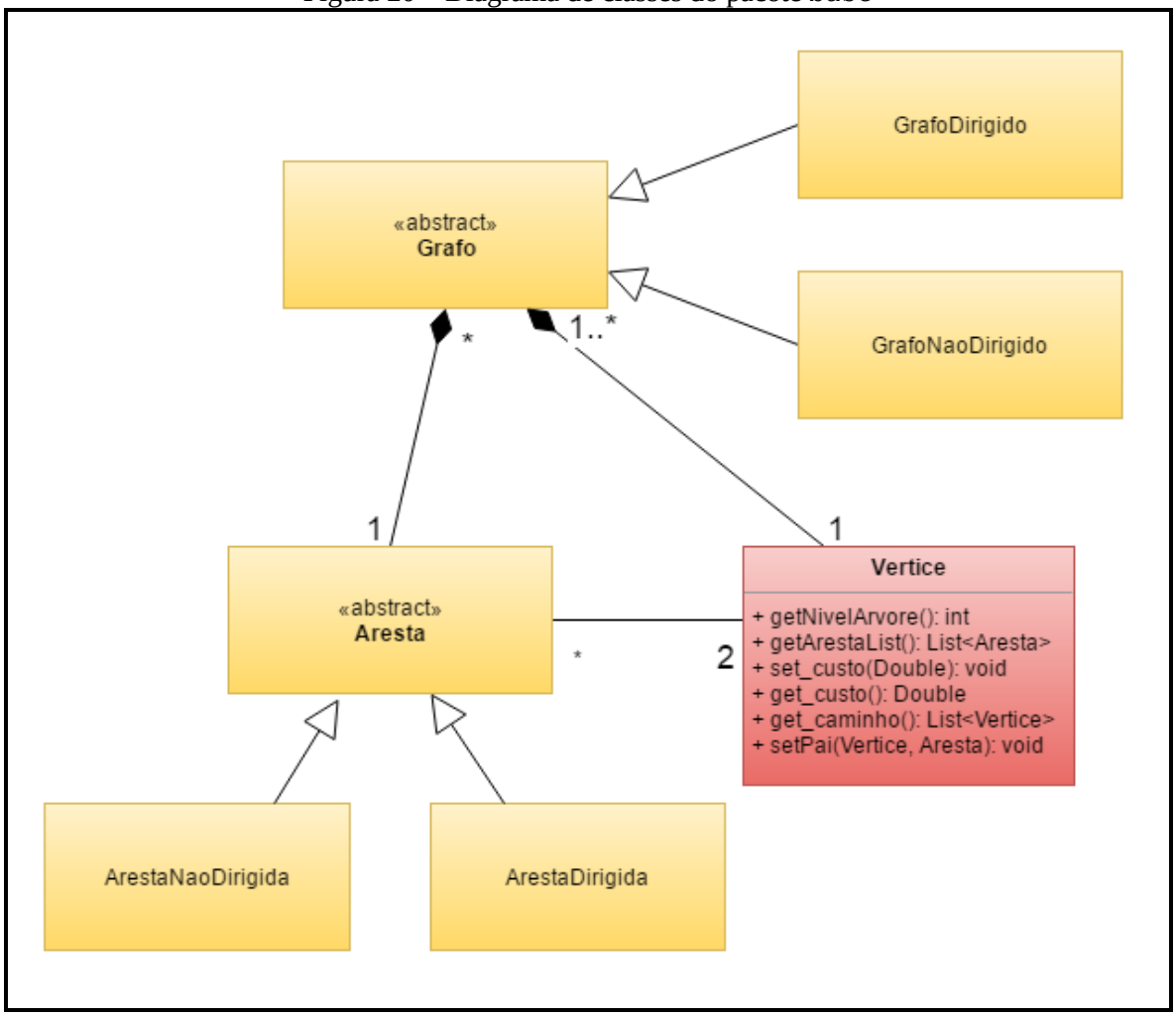
No diagrama de pacotes da Figura 19, pode-se observar os pacotes `algoritmos` e `persistência` que estão representados na cor amarela referem-se somente ao trabalho de Borba (2014) e, que não são abordados neste trabalho. Já os pacotes `base` e `aplicação` representados pela cor vermelha são os que sofreram alterações neste trabalho.

As subseções seguintes apresentam a diagramação por classe que sofreu alteração nos pacotes `base` e `aplicação`.

3.2.2.1 Pacote `base`

O pacote `base` é o pacote que contém todas as classes que compõem um grafo em sua essência, contendo as classes que representam um grafo, um vértice e uma aresta. A Figura 20 representa a estrutura das classes do pacote `base`.

Figura 20 – Diagrama de classes do pacote `base`



Fonte: adaptado de Borba (2014).

Na Figura 20, pode-se verificar as classes do pacote `base` que estão representadas na cor amarela, não foram alteradas neste trabalho, mantendo a mesma estrutura do trabalho de Borba (2014), portanto, não serão abordadas na explicação deste trabalho. A classe `Vertice` do pacote `base` que é representada na cor vermelha, sofreu alterações. As alterações feitas na classe `Vertice` incluem métodos que são utilizados como estruturas auxiliares para a ferramenta. Os métodos que foram criados são: `getNivelArvore()`, `getArestaList()`, `set_custo()`, `get_custo()`, `get_caminho()` e `setPai()`.

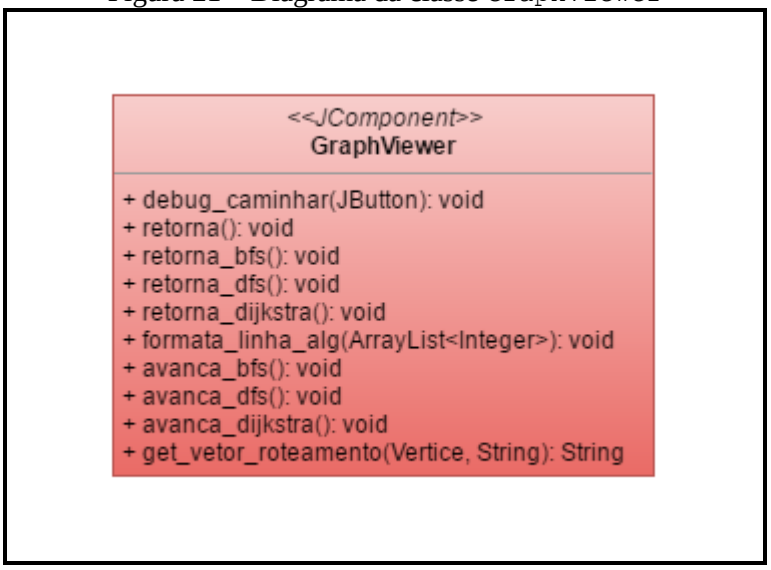
O método `getNivelArvore()` é responsável por indicar em qual nível o vértice atual se encontra na árvore de um grafo. Este método é utilizado para o algoritmo BFS. O método `getArestaList()` é utilizado por todos os algoritmos retornando a lista de arestas adjacentes ao vértice atual. O método `set_custo()` é utilizado para atribuir um custo total a um determinado vértice considerando o caminho até o mesmo. Este método é utilizado no algoritmo de Dijkstra para armazenar os custos até cada vértice. O método `get_custo()` é utilizado para obter o custo total de um vértice. O método `get_caminho()` é responsável por

retornar uma lista de vértices do caminho do vértice de origem até o vértice atual. Por fim, o método `setPai()` é um método chave para os métodos `getNivelArvore()` e `getCaminho()` pois é responsável por atribuir o vínculo de vértice pai com vértice filho, permitindo calcular os caminhos de um vértice ao outro.

3.2.2.2 Pacote `aplicação`

O pacote `aplicação` é o pacote principal da ferramenta, pois contém a classe onde se inicializa a aplicação, a `GraphViewer`. Esta classe é responsável por criar todo o menu, barras de ferramenta e interface gráfica em geral da ferramenta. A Figura 21 apresenta o diagrama de classes contendo todos os métodos que foram criados nesta classe.

Figura 21 – Diagrama da classe `GraphViewer`



Os métodos que foram criados na classe `GraphViewer`, para a efetividade deste trabalho, foram organizados em métodos de navegação e métodos auxiliares. Os métodos de navegação são aqueles que permitem ao usuário acompanhar a execução de um algoritmo sobre um grafo de forma visual e com informações detalhadas de cada passo do algoritmo. Estes métodos são: `retorna()` que permite retroceder ao primeiro passo do algoritmo, `retorna_bfs()`, `retorna_dfs()` que permitem voltar para o último passo executado dos algoritmos de BFS e DFS. Os métodos `avanca_bfs()`, `avanca_dfs()` e `avanca_dijkstra()` também fazem parte da navegação e permitem avançar um passo na execução de cada um dos três algoritmos.

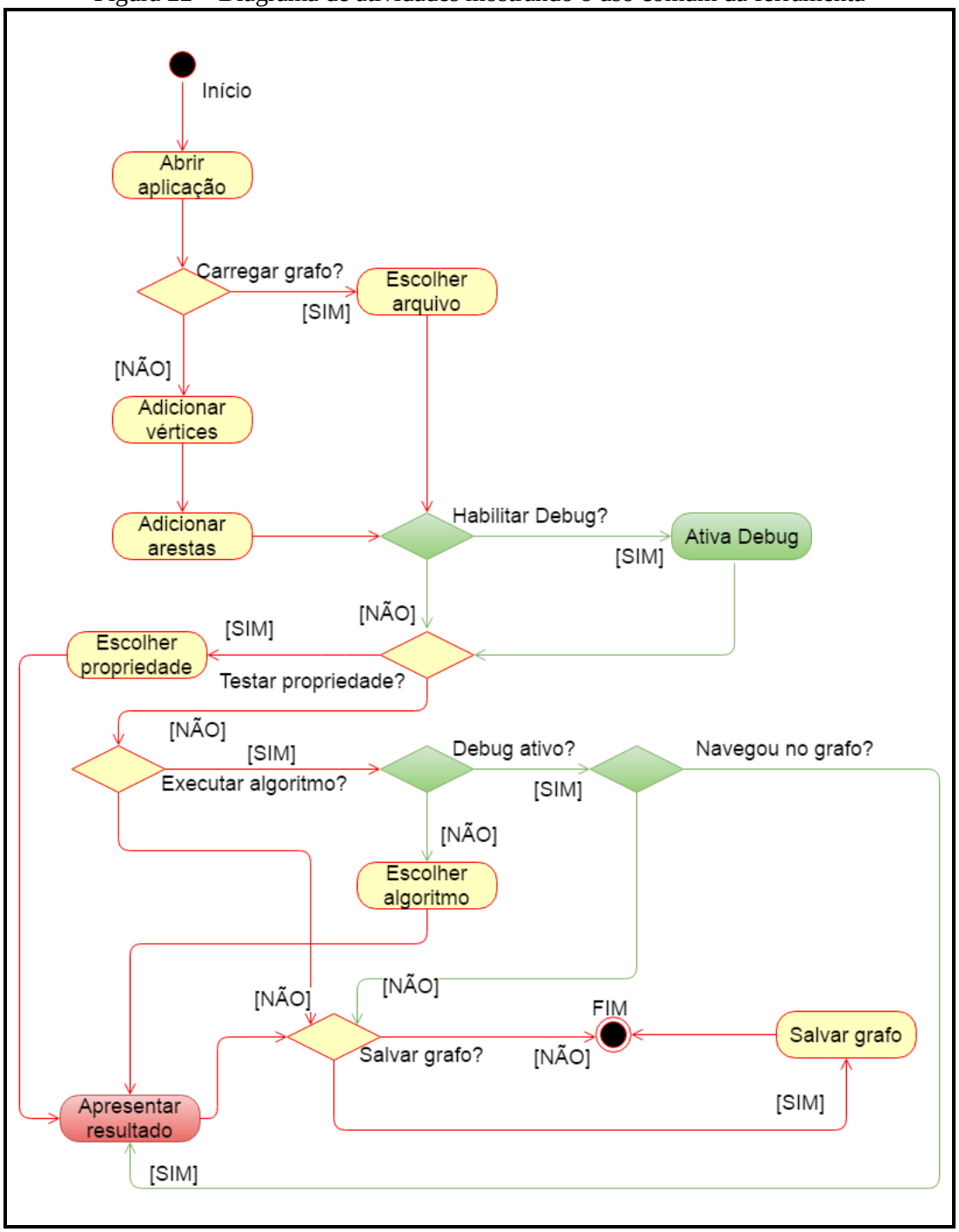
No grupo de métodos auxiliares estão o `debug_caminhar()`, `formata_linha_alg()` e `get_vetor_roteamento()`. Estes por sua vez possuem funções importantes, como controlar a chamada dos métodos de navegação, formatar as linhas do algoritmo para que se discrimine

cada passo executado e retornar informações de estruturas auxiliares no log, como o andamento do vetor de roteamento do algoritmo BFS.

3.2.3 Diagrama de atividades

O diagrama de atividades representado na Figura 22 demonstra o fluxo atual para o uso da ferramenta.

Figura 22 – Diagrama de atividades mostrando o uso comum da ferramenta



Fonte: adaptado de Borba (2014).

A partir do diagrama de atividades representado na Figura 22 pode-se verificar o fluxo mais comum de utilização da ferramenta FURB Graphs. É possível também verificar que neste diagrama foram discriminados em verde alguns processos, como verificação do modo de *debug*, ativação do *debug* e navegação no grafo, que fazem parte do processo das funcionalidades novas implementadas. Ainda dentro do mesmo fluxo, pode-se encontrar a atividade `apresentar resultado` que está na cor vermelha, o que indica que esta foi alterada em função da exposição de resultados em forma de *logs* detalhados, algoritmos e coloração do grafo. Já os demais itens do diagrama de atividades permanecem conforme o fluxo definido no trabalho de Borba (2014).

O usuário ao abrir a ferramenta pode inicialmente escolher entre carregar um grafo novo ou desenhá-lo. Ao realizar este procedimento, é possível habilitar o modo de *debug*, testar as propriedades do grafo, executar um algoritmo ou salvar um grafo. Caso o usuário opte por testar as propriedades, o resultado do teste selecionado será apresentado. Se o usuário executar um dos três algoritmos (BFS, DFS ou Dijkstra) com o modo de *debug* habilitado, será apresentada uma mensagem informando a possibilidade de navegar passo a passo no algoritmo. Caso o modo *debug* não esteja ativo, a execução do algoritmo segue normalmente e o resultado é apresentado para o usuário.

Este fluxo faz parte do funcionamento mais comum da ferramenta e o usuário pode ou não seguir esta ordem de execuções, permitindo ao usuário utilizar somente a funcionalidade de desenhar um grafo ou então carregar um grafo, aplicar alterações e salva-lo.

3.3 IMPLEMENTAÇÃO

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

3.3.1 Técnicas e ferramentas utilizadas

Para codificação deste trabalho, o ambiente de desenvolvimento utilizado foi o Eclipse Mars 1 com a linguagem de programação Java 7. A estrutura do software foi mantida com os seus dois módulos principais: interface e algoritmos.

O módulo interface utiliza a biblioteca gráfica *Swing*, nativa do JDK Java que é responsável por toda a criação de telas e menus da ferramenta. Para leitura e serialização dos grafos foi mantido também o formato de dados *JavaScript Object Notation* (JSON), armazenado utilizando a biblioteca Gson (desenvolvida pela Google) para lidar com tal

formato de dados. Para a apresentação do pseudocódigo dos algoritmos BFS, DFS e Dijkstra é utilizado a linguagem de marcação *Hyper Text Markup Language* (HTML) para que seja possível trabalhar de forma dinâmica com as cores das linhas do pseudocódigo.

3.3.2 Desenvolvimento do FURB Graphs

A seguir são apresentadas as descrições do desenvolvimento de novas funcionalidades, assim como, exibidos os principais trechos de código de cada rotina envolvida. A seção 3.3.2.1 apresenta a implementação do método `setPai()` da classe `Vertice` do pacote `base`, que é responsável pelo vínculo entre os vértices pai e filho do grafo e que permite a localização dos caminhos e cálculos de custo total. A seção 3.3.2.2 apresenta a implementação do método `formata_linha_alg()` responsável pela discriminação de linhas executadas do algoritmo no pseudocódigo. Na seção 3.3.2.3 é apresentado o método `get_vetor_roteamento()`. O método `getNivelArvore()` é apresentado na seção 3.3.2.4. O método `getCaminho()` é apresentado na seção 3.3.2.5. A seção 3.3.2.6 apresenta o desenvolvimento em sua totalidade do processo de navegação através do algoritmo BFS. A seção 3.3.2.7 demonstra como foi implementado todo o processo de navegação para o algoritmo DFS. Na seção 3.3.2.8 é apresentado o processo de navegação no algoritmo de Dijkstra. Na seção 3.3.2.9 é apresentada a implementação responsável por permitir retroceder os algoritmos de BFS, DFS e Dijkstra ao ponto inicial da sua representação gráfica no modo *debug*. Por fim, na seção 3.3.2.10 são apresentadas todas as alterações na interface gráfica que contribuem para o auxílio no ensino-aprendizagem da ferramenta FURB Graphs.

3.3.2.1 Atribuindo pais aos vértices

Atribuir a relação de vértice pai e aresta pai a um vértice facilita toda a navegação e permite buscar caminhos nos grafos. Para facilitar os procedimentos de navegação nos algoritmos de BFS, DFS e Dijkstra, foram criados dois atributos importantes na classe `Vértice`, o atributo `this.vertice_pai` e `this.aresta_pai`. Estes foram criados para que cada objeto do tipo `Vertice` criado tenha um vínculo com os seus demais, para que seja possível efetuar uma busca retroativa de filho para pai até a origem. Esta funcionalidade é exemplificada no Quadro 5.

Quadro 5 – Método responsável por atribuir vértices e arestas pai para um determinado vértice

```
01. public void setPai(Vertice v, Aresta a) {
02.     this.vertice_pai = v;
03.     this.aresta_pai = a;
04. }
```

3.3.2.2 Discriminando linhas do pseudocódigo

Para que se possa tirar proveito de forma adequada dos algoritmos BFS, DFS e Dijkstra, se tornou necessário elaborar um meio de relação entre a representação gráfica, informações do *log* para com o algoritmo em si. Foi então elaborado um método que permite discriminar cada linha de um pseudocódigo separadamente, colorindo-as para que se destaquem e se possa identificar quais linhas representam aquele processo ou aquele passo do algoritmo. O Quadro 6 representa o algoritmo responsável por esta funcionalidade.

Quadro 6 – Algoritmo que discrimina linhas do pseudocódigo

```
01. public void formata_linha_alg(ArrayList<Integer> linha_list) {
02.     String aux = "";
03.     int linha_aux = 0;
04.     for (String str : this.algoritmoStr.split("</p>")) {
05.         linha_aux++;
06.
07.         if (linha_list.contains(linha_aux)) {
08.             aux += "<p style='color: white;font-family: Courier
       New;background-color:6495ED;margin: 0;'><b>" + str + "</b></p>";
09.         } else {
10.             aux += "<p style='font-family: Courier New;margin:
                                                    0;'>" + str + "</p>";
11.         }
12.
13.     }
14.     this.txtAlg.setText(aux);
15. }
```

O algoritmo, na linha 01, primeiramente recebe uma lista de inteiros, que indicam quais linhas devem ser coloridas. Logo após, na linha 04, o texto do algoritmo é quebrado em linhas e iterado, para cada iteração destas linhas é verificado se a linha percorrida é uma das linhas informadas no construtor do método. Caso seja, na linha 08 o texto em HTML desta linha será alterado para que esta linha seja destacada no pseudocódigo, caso contrário o texto permanece normal.

3.3.2.3 Vetor de roteamento

Para a representação com detalhes do *log* do algoritmo BFS, se fez necessário desenvolver um método auxiliar que permita demonstrar no *log* o vetor final de roteamento contendo a árvore gerada do algoritmo BFS. O Quadro 7 representa este método.

Quadro 7 – Método que exibe o vetor de roteamento

```
01. public String get_vetor_roteamento(Vertice v_pai, String
                                                          roteamento) {
02.     if (roteamento.isEmpty()) {
03.         roteamento += "";
04.     }
05.     if (v_pai == null) {
06.         v_pai = this.vertice_origem;
07.         roteamento += v_pai.getDado();
08.     }
09.     if (!v_pai.getArestaList().isEmpty()) {
10.         roteamento += " <";
11.
12.         for (Aresta a : v_pai.getArestaList()) {
13.             Vertice v = (a.getVi().equals(v_pai) ? a.getVj() :
                                                          a.getVi());
14.             if (v.vertice_pai != null &&
                       this.algoritmoDesenho.verticesMarcados.contains(
                                                 findVerticeVisual(v))) {
15.                 if (v.vertice_pai.equals(v_pai)) {
16.                     roteamento += ", " + v.getDado();
17.                     roteamento += get_vetor_roteamento(v, "");
18.                 }
19.             }
20.         }
21.         roteamento += ">";
22.     }
23.
24.     return roteamento;
25. }
```

Este algoritmo é recursivo e busca através das relações de pai e filho dos vértices vinculadas no método `setPai()` até que não consiga mais alcançar outro vértice. Na linha 01, o construtor espera receber um objeto do tipo `Vértice` chamado `v_pai` e uma *String* chamada `roteamento`. Estes, inicialmente são enviados respectivamente como *Null* e ‘’, informando que ele deve partir da origem. Entre as linhas 02 a 10 são feitas formatações na *String* de roteamento. Na linha 12 as arestas adjacentes do vértice atual são iteradas, na linha 13 é efetuado uma verificação ternária que descobre o vértice de destino. Na linha 14 é verificado se o vértice de destino não é o vértice origem e se o mesmo consta na lista de vértices processados. Caso a condição seja verdadeira, é verificado se o pai do vértice destino é de fato o vértice atual do processamento. Se for, este vértice será enviado no parâmetro da invocação do método recursivamente. No retorno de cada invocação do método é retornada a *String* com o roteamento e esta por sua vez, forma uma árvore que é apresentada no *log* em formato textual.

3.3.2.4 Identificar o nível em uma árvore do BFS

Durante a execução do algoritmo BFS é construída uma árvore de execução que é representada graficamente por texto no grafo durante a navegação, sinalizada pelas arestas

que estão coloridas em laranja. Para se identificar o nível de um determinado vértice em uma árvore, é necessário utilizar os vínculos de pai e filho de vértices já definidos no método `setPai(`. O algoritmo responsável por esta funcionalidade está no Quadro 8.

Quadro 8 – Algoritmo que identifica o nível de um vértice em uma árvore

```
01. public int getNivelArvore() {
02.     int nivel_arvore = 0;
03.     Vertice v_aux = this;
04.     while (v_aux.vertice_pai != null) {
05.         nivel_arvore += 1;
06.         v_aux = v_aux.vertice_pai;
07.     }
08.     return nivel_arvore;
09. }
```

O método `getNivelArvore()` da classe `Vértice` utiliza o vértice atual como parâmetro para iniciar a busca retroativa do seu nível na árvore. Na linha 04 é iniciado um laço que deve permanecer enquanto o vértice em questão for diferente do vértice de origem. Na linha 05 é incrementado o nível da árvore e na linha 06 o objeto `Vértice` é atualizado com a referência do seu vértice pai para que possa dar continuidade ao laço. Ou seja, enquanto o vértice possuir pai, o algoritmo vai incrementar o seu nível da árvore, que é a distância entre o vértice origem para o vértice atual que invocou o método. Seguindo assim, em um processo retroativo do filho para o seu pai respectivamente até atingir o vértice de origem, que não possui pai e por isto acaba quebrando o laço e retornando o nível do vértice na árvore.

3.3.2.5 Caminhos em grafos

Para a navegação do algoritmo de Dijkstra foi necessário elaborar uma rotina auxiliar que permitisse retornar o caminho de um determinado vértice até o vértice de origem. O Quadro 9 apresenta o método `get_caminho()` da classe `Vértice` que é responsável por esta funcionalidade.

Quadro 9 – Algoritmo que retorna um caminho do vértice atual até a origem

```
01. public List<Vertice> get_caminho() {
02.     List<Vertice> temp_list = new ArrayList();
03.     Vertice temp_v = this.vertice_pai;
04.
05.     while (temp_v != null) {
06.         temp_list.add(temp_v);
07.         temp_v = temp_v.vertice_pai;
08.     }
09.
10.     return temp_list;
11. }
```

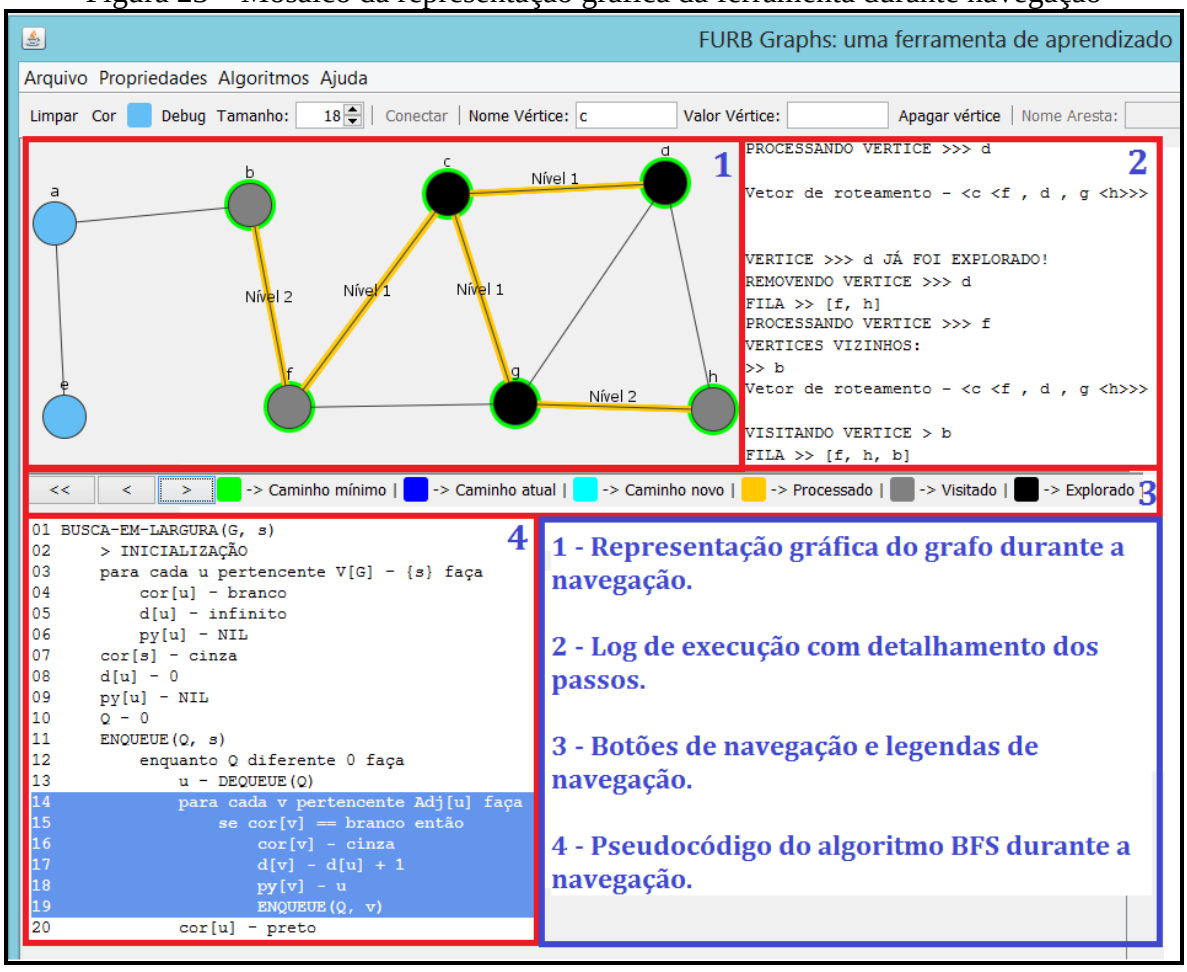
O algoritmo inicialmente tem como vértice inicial o vértice pai do vértice atual que invocou o método. Na linha 05 se inicia um laço que só finaliza quando o vértice não tiver mais um pai, ou seja, só finaliza quando o vértice encontrar a origem. Nas linhas 06 e 07

durante o laço, são armazenados em uma lista os vértices e o objeto anterior é substituído pelo seu pai e assim consecutivamente até quebrar o laço. Ao finalizar este processamento, tem-se uma lista ordenada com os vértices do caminho do vértice atual até a origem, sendo retornada pelo método invocado.

3.3.2.6 Navegação no algoritmo BFS

A navegação sobre a execução do algoritmo BFS leva em consideração todas as regras já explicadas na seção 2.2.1. Além das regras fundamentais do algoritmo, a navegação envolve a representação gráfica de cada passo do algoritmo avançado ou retrocedido, colorindo os vértices e arestas, a apresentação do *log* de execução do algoritmo com detalhes de atributos, estados e vetor de roteamento do grafo em que foi executado o algoritmo, além de discriminar no pseudocódigo as linhas que se referem ao passo executado no momento. A Figura 23 apresenta a coloração do grafo, *log* de execução e pseudocódigo da ferramenta durante a navegação deste algoritmo.

Figura 23 – Mosaico da representação gráfica da ferramenta durante navegação



O Quadro 10 apresenta parte do método `avanca_bfs()` responsável pela parte de navegação de passos para frente na representação do algoritmo BFS. Esta parte demonstrada

no Quadro 10 se refere ao bloco responsável pelo processamento principal das arestas e vértices no algoritmo BFS.

Quadro 10 – Navegação para frente no BFS

```
01. //Percorre as arestas do vertice principal
02. for (Aresta a : this.vertice_aux.getArestaList()) {
03.     //Encontra o destino
04.     Vertice v_destino = (a.getVi() == this.vertice_aux ?
                                                a.getVj() : a.getVi());
05.
06.     //Se o vertice destino não constar na lista de processamento
07.     if (!this.algoritmoDesenho.verticesMarcados.contains(
                                        findVerticeVisual(v_destino))) {
08.         //Adiciona o vertice a lista dos vertices processados
09.         this.vertice_proc_stack.push(v_destino);
10.         v_destino.setPai(this.vertice_aux, a);
11.     }
12. }
```

Analisando o Quadro 10, na linha 02, o algoritmo percorre todas as arestas adjacentes ao vértice processado. Durante a iteração das arestas deste vértice, na linha 04 é efetuado um teste dentro de uma operação ternária, que identifica qual o vértice destino da aresta. Tendo identificado o vértice de destino é efetuado um novo teste na linha 07, onde se verifica se o vértice de destino em questão, não está de fato na lista de vértices processados graficamente. Assim, identificado um vértice que seja destino e que não esteja marcado como já processado, ele é adicionado a uma pilha auxiliar de processamento que é utilizada para controlar os vértices já processados. Por fim, ao vértice de destino é atribuído como pai o vértice processado e sua respectiva aresta.

Outro bloco importante para a navegação do BFS está representado no Quadro 11. Ele é responsável pela representação da visita dos vértices adjacentes, processo que é explicado na definição do algoritmo BFS na seção 2.2.1. O Quadro 11 também efetua a geração de *log* da execução do algoritmo e a discriminação de linhas no pseudocódigo.

Quadro 11 – Representação da visita de vértices adjacentes no BFS

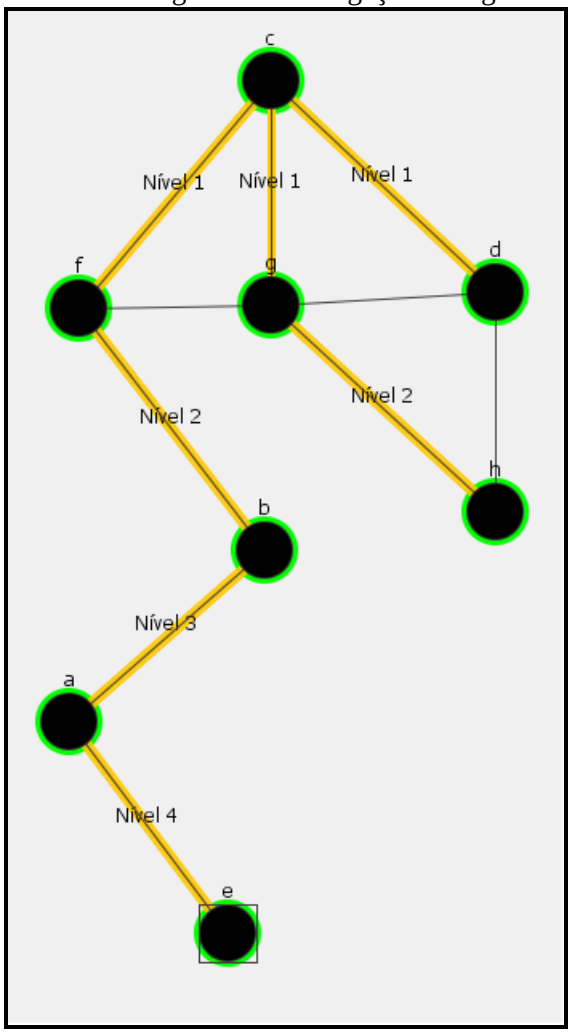
```
01. this.formata_linha_alg(new ArrayList<Integer>(
                                    Arrays.asList(14,15,16,17,18,19)));
02. this.algoritmoDesenho.verticesMarcados.add(findVerticeVisual(
                                                    this.vertice_aux));
03. this.algoritmoDesenho.coresVertices.put(findVerticeVisual(
                                        this.vertice_aux), Color.gray);
04. Vertice v_aux = this.vertice_proc_stack.pop();
05. this.txtLog.append("VISITANDO VERTICE > " + v_aux.getDado()+"\n");
06.
07. //Percorre a lista de destinos para colorir
08. if(!this.vertice_aux_list.contains(v_aux) &&
                            !this.vertice_black_list.contains(v_aux)) {
09.     //Colore de cinza
10.     this.algoritmoDesenho.verticesMarcados.add(findVerticeVisual(
                                                            v_aux));
11.     this.algoritmoDesenho.coresVertices.put(findVerticeVisual(
                                                    v_aux),Color.gray);
12.
13.     if (v_aux.aresta_pai != null) {
14.         this.algoritmoDesenho.arestasMarcadas.add(findArestaVisual(
                                                    v_aux.aresta_pai));
15.         this.algoritmoDesenho.coresArestas.put(findArestaVisual(
                                        v_aux.aresta_pai), Color.orange);
16.         findArestaVisual(v_aux.aresta_pai).name = "Nível " +
                                                v_aux.getNivelArvore();
17.     }
18.
19.     if (!this.fila_bfs_list.contains(v_aux.getDado())) {
20.         this.fila_bfs_list.add((String) v_aux.getDado());
21.     }
22.     this.txtLog.append("FILA >> " + this.fila_bfs_list + "\n");
23.
24.     this.vertice_aux_list.add(v_aux);
25. }
26.
27. this.vertice_proc_list.clear();
```

O bloco de código representado no Quadro 11 apresenta na linha 01 a discriminação de linhas executadas no pseudocódigo através do método `this.formata_linha_alg()` que é explicado na seção 3.3.2.2. Nas linhas 02 e 03 o vértice em processamento é adicionado na lista de vértices já processados e marcado como vértice visitado ao ser colorido de cinza. Na linha 04 o vértice que será visitado é desempilhado da estrutura da pilha de vértices processados. Feito isto, na linha 05 o vértice visitado é apresentado no *log* de execução. Na linha 08 são verificados se o vértice que está sendo visitado não consta na lista de vértices já visitados e nem na lista de vértices já explorados. Caso a condição seja verdadeira, nas linhas 10 e 11 o vértice visitado é adicionado à lista de vértices processados e então ele é colorido de cinza.

Na linha 13, é efetuado um teste que verifica se o vértice visitado possui uma aresta pai, pois caso o vértice não possua uma aresta pai ele é o vértice de origem. Caso o vértice visitado possua uma aresta pai, nas linhas 14 e 15 esta aresta é adicionada à lista de arestas

processadas e então é colorida de laranja. Ainda referente a árvore, a linha 16 se responsabiliza por demonstrar o nível do vértice visitado através daquela aresta de forma visual para que se compreenda a distância e os níveis da árvore gerada no algoritmo BFS. Através da Figura 24 é possível visualizar o algoritmo BFS finalizado e a sua árvore gerada.

Figura 24 – Árvore gerada na navegação do algoritmo BFS



Seguindo a mesma lógica de verificações, o algoritmo identifica quando os vértices já estão de fato explorados e então os colore de preto. A cada passo da navegação o algoritmo vai indicando no *log* as informações, detalhes e estados mais importantes dos vértices e arestas. No pseudocódigo são discriminadas as linhas referentes a execução do algoritmo a cada passo. Ao final do processamento se obtém uma árvore representando a execução do algoritmo BFS.

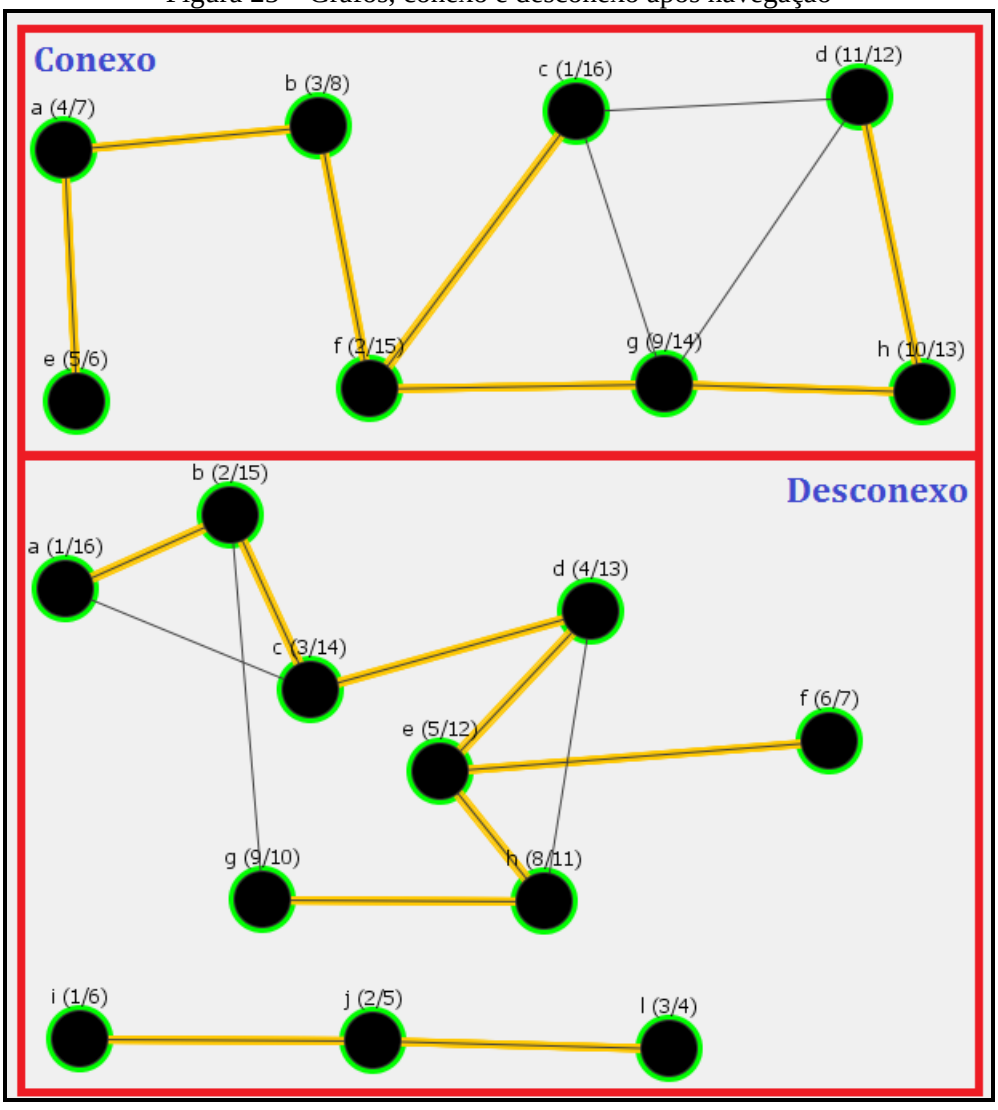
Outro ponto importante na navegação do algoritmo BFS é o processo de retroceder passos no algoritmo. Este que por sua vez representa o retorno de um passo, descolorindo as arestas e vértices que acabaram de ser processadas, retornando para o último vértice

processado para que se possa dar continuidade ao processamento através deste, ou então retornar mais passos.

3.3.2.7 Navegação no algoritmo DFS

A navegação sobre a execução do algoritmo DFS leva em consideração todas as regras já explicadas na seção 2.2.2. Além de todos as verificações e regras referentes ao algoritmo DFS, a navegação do mesmo contempla toda a parte de representação gráfica da execução passo a passo do algoritmo, avançando ou retrocedendo, considerando a coloração dos vértices e arestas envolvidos na execução, tanto em grafos conexos como desconexos. A Figura 25 apresenta um exemplo de grafo conexo e desconexo após a sua navegação finalizada.

Figura 25 – Grafos, conexo e desconexo após navegação



Também se inclui nas funcionalidades da navegação deste algoritmo a apresentação do tempo de abertura e fechamento de cada vértice quando este é visitado e ou explorado, assim

como são inclusas as informações detalhadas no *log* de execução e as linhas do pseudocódigo discriminadas conforme cada passo da execução. O Quadro 12 apresenta uma parte do método `avanca_dfs()`, esta parte se refere à parte inicial do algoritmo, onde se verifica os vértices vizinhos para se determinar a ordem em que irá visita-los.

Quadro 12 – Parte do algoritmo `avanca_dfs()` que identifica os vértices adjacentes

```
01. //Percorre as arestas do vertice principal
02. for (Aresta a : this.vertice_aux.getArestaList()) {
03.     //Encontra o destino
04.     Vertice v_destino = (a.getVi() == this.vertice_aux ?
                                                a.getVj() : a.getVi());
05.
06.     //Se o vertice destino não constar na lista de processamento
07.     if (!this.algoritmoDesenho.verticesMarcados.contains(
                                        findVerticeVisual(v_destino))) {
08.         //Adiciona na lista dos vertices que serão visitados
09.         this.vertice_aux_list.add(v_destino);
10.         v_destino.setPai(this.vertice_aux, a);
11.     }
12. }
13.
14. for (int x = this.vertice_aux_list.size(); x > 0; x--) {
15.     Vertice v_aux = this.vertice_aux_list.get(x - 1);
16.     if (this.vertice_aux_stack.contains(v_aux)) {
17.         this.vertice_aux_stack.remove(v_aux);
18.     }
19.     this.vertice_aux_stack.push(v_aux);
20. }
```

Nesta primeira parte do algoritmo, apresentada no Quadro 12, inicia-se o processamento na linha 02 percorrendo todas as arestas adjacentes ao vértice atual. Logo abaixo na linha 04 através de uma operação ternária é identificado qual é o vértice de destino da aresta que está sendo percorrida. Na linha 07 é feito uma verificação para identificar se o vértice de destino não está marcado, ou seja, se ele não consta na lista de vértices processados. Caso este não esteja na lista de vértices processados, nas linhas 09 e 10 o vértice será adicionado à lista de vértices que serão visitados e logo será atribuído a este vértice destino o seu vértice pai como o vértice atual de processamento e a sua respectiva aresta pai. Após identificar todos os vértices livres para serem visitados a partir do vértice atual, inicia-se um laço que percorre esta lista de vértices com intuito de reorganizá-los em uma pilha. A lista é percorrida de trás para frente e na linha 19 o vértice é adicionado a pilha para que esteja na ordem correta de visitação.

Feito isto, o algoritmo passa para a próxima fase, que é de descobrir estes vértices adjacentes, processar, colorir eles e lhes atribuir seus tempos de abertura e fechamento. Neste passo, o algoritmo inicia o processamento colorindo o vértice principal da execução, chamado `this.vertice_aux` de cinza, indicando que este já foi visitado, então trata de atribuir-lhe um

tempo, que pode ser definido dependendo do seu estado. Uma parte do algoritmo `avanca_dfs()` é apresentada no Quadro 13.

Quadro 13 – Atribuindo tempos de abertura e fechamento

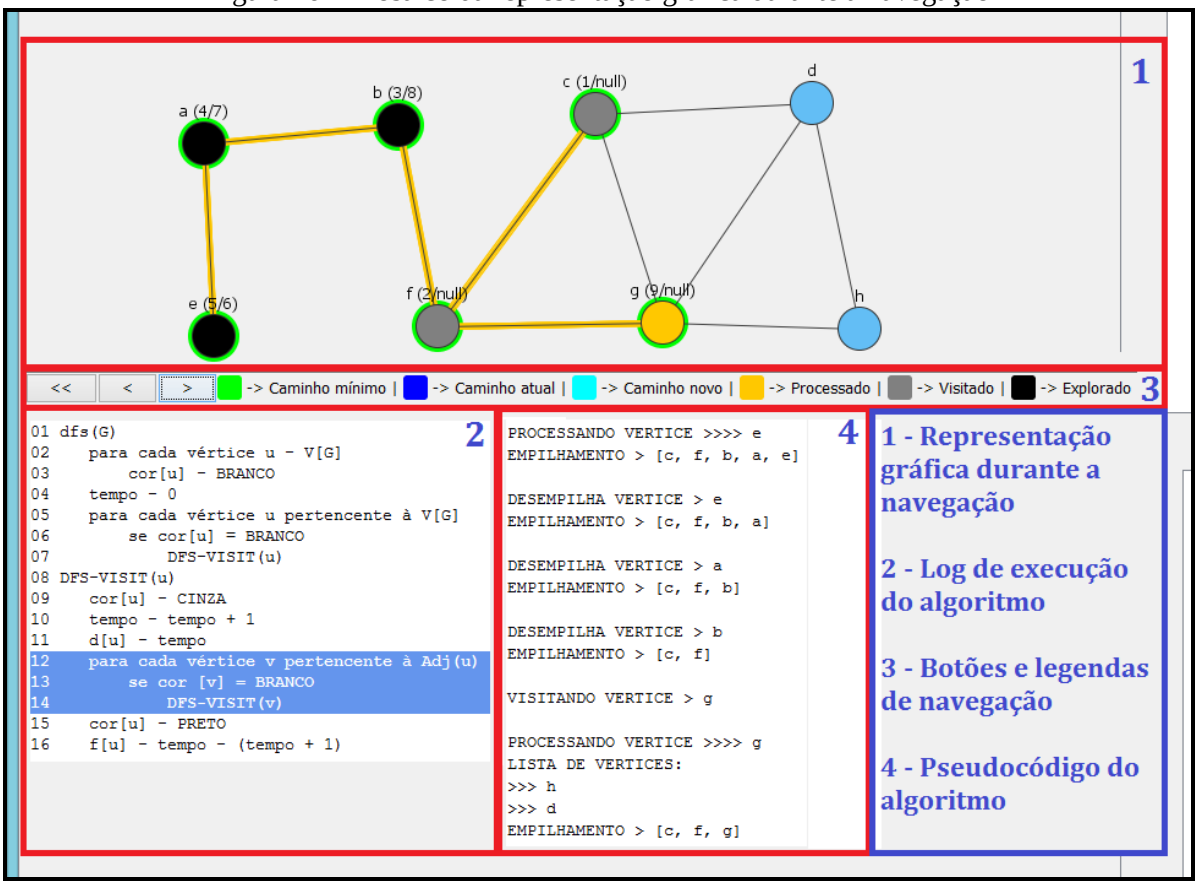
```
01. int count = 0;
02. for (Aresta a : this.vertice_aux.getArestaList()) {
03.     Vertice v_aux = (a.getVi() == this.vertice_aux ? a.getVj() :
                                                              a.getVi());
04.     if (this.algoritmoDesenho.verticesMarcados.contains(
                                          findVerticeVisual(v_aux))) {
05.         count += 1;
06.     }
07. }
08.
09. if (count == this.vertice_aux.getQtdeArestas()) {
10.     this.formata_linha_alg(new ArrayList<Integer>(Arrays.asList(15,
                                                                  16)));
11.     this.algoritmoDesenho.coresVertices.remove(findVerticeVisual(
                                                  this.vertice_aux));
12.     this.algoritmoDesenho.coresVertices.put(findVerticeVisual(
                                       this.vertice_aux), Color.black);
13.
14.     this.txtLog.append("DESEMPILHA VERTICE > " +
                                    this.vertice_aux.getDado() + "\n");
15.     this.empilhamento_dfs_stack.remove(this.vertice_aux.getDado());
16.
17.     String dados_pai = this.vertice_aux.temp_abertura + "/" +
                                     this.vertice_aux.temp_fechamento;
18.     findVerticeVisual(this.vertice_aux).value = dados_pai;
19.     this.vertice_destino = this.vertice_aux.vertice_pai;
20.     this.vertice_black_list.add(this.vertice_aux);
21.
22.     this.vertice_ordem_visita_stack.push(this.vertice_aux);
23.
24.     this.txtLog.append("EMPILHAMENTO > " +
                                   this.empilhamento_dfs_stack + "\n");
25. } else {
26.     this.formata_linha_alg(new ArrayList<Integer>(Arrays.asList(9,
                                                          10,11)));
27.     if (!this.vertice_ordem_visita_stack.contains(
                                                 this.vertice_aux)) {
28.         this.vertice_ordem_visita_stack.push(this.vertice_aux);
29      }
30.     this.txtLog.append("VISITANDO VERTICE > " + v_destino.getDado()
                                                           + "\n");
31.
32.     String dados_pai = this.vertice_aux.temp_abertura + "/null";
33.     findVerticeVisual(this.vertice_aux).value = dados_pai;
34.
35.     this.vertice_ordem_visita_stack.push(v_destino);
36. }
```

Após colorir o vértice atual de cinza, entre as linhas 02 a 07 a lista de arestas do vértice atual é percorrida. Onde, também é identificado o vértice de destino de cada aresta e se o mesmo consta na lista de vértices já processados. Cada vez que encontra um vértice já processado um contador é incrementado, que posteriormente é utilizado para controlar se todos os vértices adjacentes já foram visitados e ou explorados. Logo, na linha 09 é verificado

se todos os vértices adjacentes foram visitados, caso ainda exista algum, então segue-se para a linha 26, onde é efetuada a discriminação das linhas do pseudocódigo referente a visita de um vértice novo. Logo abaixo, na linha 30, é apresentado no *log* de execução o vértice que está sendo visitado. Nas linhas 32 e 33 se atribui o tempo de abertura ao vértice visitado.

Caso o vértice tenha sido explorado, segue-se para a linha 10, onde é efetuada a discriminação da linha do pseudocódigo referente a execução do algoritmo. Nas linhas 11 e 12 o vértice é colorido de preto, indicando que este não possui mais adjacentes a visitar. Na Figura 26 é possível identificar parte do processo de navegação do DFS.

Figura 26 – Mosaico da representação gráfica durante a navegação



Ainda conforme o algoritmo do Quadro 13, na linha 14 é apresentado no *log* a informação de que o vértice destino foi desempilhado e a linha 15 efetua de fato o processo de desempilhamento do vértice. Nas linhas 17 e 18 é atribuído o tempo de fechamento, de forma visual, junto ao vértice no grafo. Por fim, na linha 24 é apresentado no *log* de execução a informação de como se encontra o estado da pilha do algoritmo.

Este procedimento segue para todos os vértices do grafo, incluindo os que estão desconexos do vértice atual. O Quadro 14 demonstra a parte do método `avanca_dfs()` responsável por encontrar um novo vértice caso o grafo seja desconexo do subgrafo principal.

Quadro 14 – Identificando próximo vértice em um grafo desconexo

```
01. if (this.empilhamento_dfs_stack.isEmpty()) {
02.     for (Vertice v_aux : this.vertice_list) {
03.         if (!this.algoritmoDesenho.verticesMarcados.contains(
04.                                         findVerticeVisual(v_aux))) {
05.             this.vertice_destino = v_aux;
06.             this.vertice_aux = v_aux;
07.             break;
08.         }
09.     }
10. }
```

No trecho de código representado no Quadro 14, na linha 01 é verificado se a pilha de execução do DFS está vazia, caso esteja, é feito o percorrimento da lista que possui todos os vértices do grafo. Enquanto se percorre esta lista, são efetuados testes para verificar se estes vértices já foram processados ou não, caso não estejam nesta lista, então se trata de um vértice que está em um subgrafo desconexo, logo, nas linhas 05 e 06 são atualizadas as variáveis necessárias para se dar continuidade ao processamento a partir deste vértice e na linha 07 o laço é quebrado.

Outro método importante é o `retorna_dfs()`, ele é responsável por retroceder ao passo anterior do algoritmo sempre que o usuário utilizar o botão de voltar. Esta funcionalidade retrocede um passo de cada vez, ignorando em qual estado se encontrava o vértice ou aresta, voltando ao estado inicial do último passo que foi totalmente processado.

3.3.2.8 Navegação no algoritmo de Dijkstra

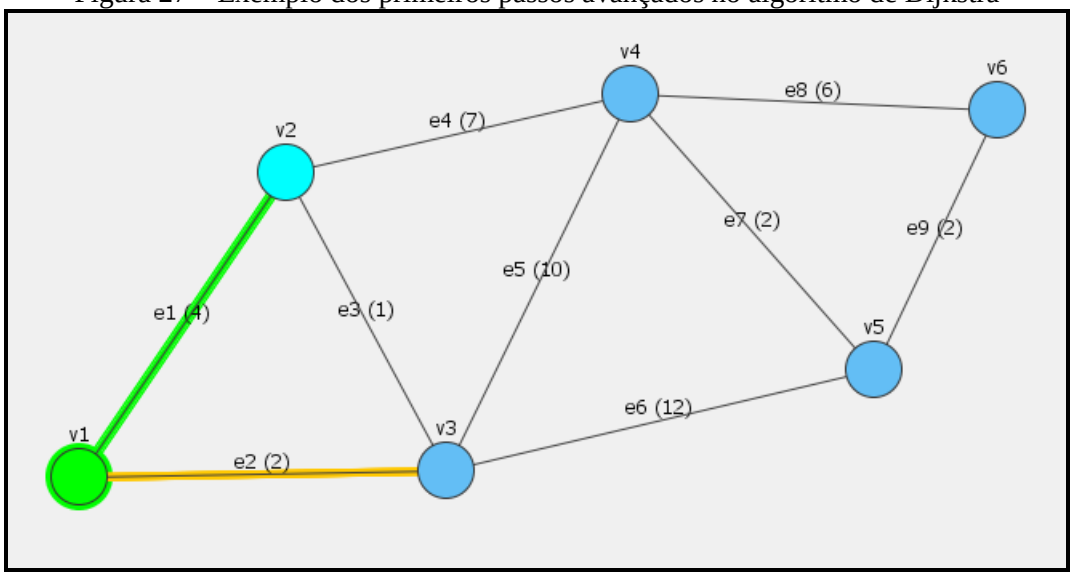
Na navegação do algoritmo de Dijkstra, segue-se todas as regras que foram detalhadas na definição do mesmo, na seção 2.3. O método responsável pela navegação sobre este algoritmo é o `avanca_dijkstra()`. Este método contempla toda a representação gráfica da execução dos passos do algoritmo, incluindo coloração dos vértices e arestas conforme seus estados e atributos, apresentação do *log* de execução do algoritmo e discriminação de linhas do pseudocódigo referentes a cada passo executado.

A representação da navegação do algoritmo de Dijkstra contém diversos detalhes, estados e atributos diferentes dos demais algoritmos. Durante todo o processo é possível verificar um processamento de um vértice ou aresta, a identificação de caminhos não explorados, uma comparação entre caminhos do grafo para se identificar um caminho mínimo e a própria discriminação do caminho mínimo.

O método de navegação do algoritmo de Dijkstra inicialmente assume o vértice atual como o vértice de origem para iniciar o seu processamento. Logo, o vértice atual é colorido de laranja, indicando que está em processamento. A partir deste vértice, são percorridos os seus

vértices adjacentes para identificar quais vértices ainda não possuem um pai. Para cada caminho adjacente encontrado a aresta é colorida de laranja. Para cada vértice que não possui pai as arestas são coloridas de azul claro. Para o vértice sem pai é atribuído o vértice atual como pai, sendo calculado e atribuído o custo para se atingir este vértice partindo da origem. A Figura 27 apresenta um exemplo durante os passos iniciais da navegação no algoritmo de Dijkstra.

Figura 27 – Exemplo dos primeiros passos avançados no algoritmo de Dijkstra



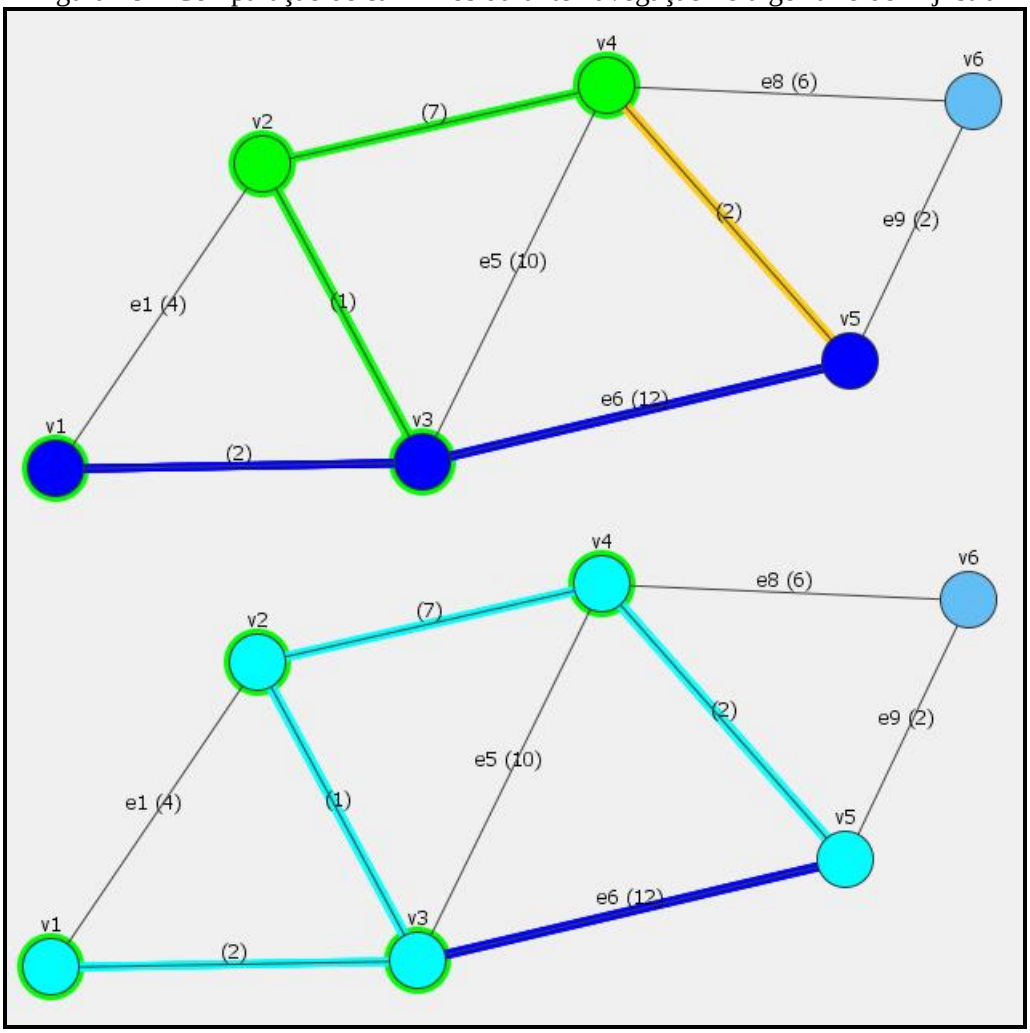
Quando se identifica que um vértice já possui pai, então segue-se para um passo onde se efetuam as verificações entre o caminho antigo, onde se encontra o pai atual do vértice para com o caminho novo. Parte do código que processa as comparações é apresentado no Quadro 15.

Quadro 15 – Comparações de caminhos para navegação no Dijkstra

```
01. if ((this.vertice_aux.get_custo() + a.getValor()) <
                                                  v_destino.get_custo()){
02.     this.log_cmp_novo_str += "> Sim\n";
03.     this.vertice_cmp_list = caminho_novo_list;
04.     this.vertice_cmp_list.add(this.vertice_aux);
05.     this.vertice_cmp_list.add(v_destino);
06.
07.     //Remove a aresta antiga pois encontrou uma aresta nova para o
                                                                  vertice
08.     this.aresta_list.remove(v_destino.aresta_pai);
09.     this.aresta_cmp_pai_novo_list.remove(v_destino.aresta_pai);
10.
11.     //Se não tem pai, adiciona o vertice atual como pai
12.     v_destino.setPai(this.vertice_aux, a);
13.
14.     //Seta o custo como Custo da aresta + Custo do pai
15.     v_destino.set_custo(a.getValor() +
                                       this.vertice_aux.get_custo());
16.
17.     if (!this.aresta_list.contains(a)) {
18.         this.aresta_list.add(a);
19.     }
20.     if (!this.aresta_cmp_pai_novo_list.contains(a)) {
21.         this.aresta_cmp_pai_novo_list.add(a);
22.     }
23.     this.aresta_cmp_pai_novo_list.add(v_destino.aresta_pai);
24.     //Percorre o caminho de vertices do pai novo até a origem
25.
26.     for (Vertice v : v_destino.get_caminho()) {
27.         if (!this.vertice_cmp_pai_novo_list.contains(v)) {
28.             this.vertice_cmp_pai_novo_list.add(v);
29.             if (!this.aresta_cmp_pai_novo_list.contains(
                                 v.aresta_pai)&& v.aresta_pai != null){
30.                 this.aresta_cmp_pai_novo_list.add(v.aresta_pai);
31.             }
32.         }
33.     }
```

Na linha 01 é feita a comparação entre os caminhos, onde se verifica qual dos caminhos possui o custo menor partindo da origem até o destino. Caso o caminho novo tenha um custo menor que o caminho antigo, segue-se para a linha 02, onde é apresentado no *log* da execução do algoritmo a informação referente a condição que foi testada, informando que o caminho novo é realmente o menor. Logo, a partir da linha 03 até a 09 são atualizadas as variáveis com o novo caminho encontrado. A linha 12 se encarrega de atribuir o novo pai do vértice destino e a sua nova aresta pai, assim como na linha 15 o novo custo é atribuído ao caminho da origem até o vértice destino. Entre as linhas 17 a 33 são atualizadas as listas de comparações que mais tarde serão utilizadas para colorir o grafo com suas devidas comparações. Uma representação gráfica de uma comparação durante a navegação no algoritmo de Dijkstra pode ser visualizada na Figura 28, onde o caminho `v1 > v3 > v5` em azul escuro é comparado com o caminho `v1 > v3 > v2 > v4 > v5` em azul claro.

Figura 28 – Comparação de caminhos durante navegação no algoritmo de Dijkstra



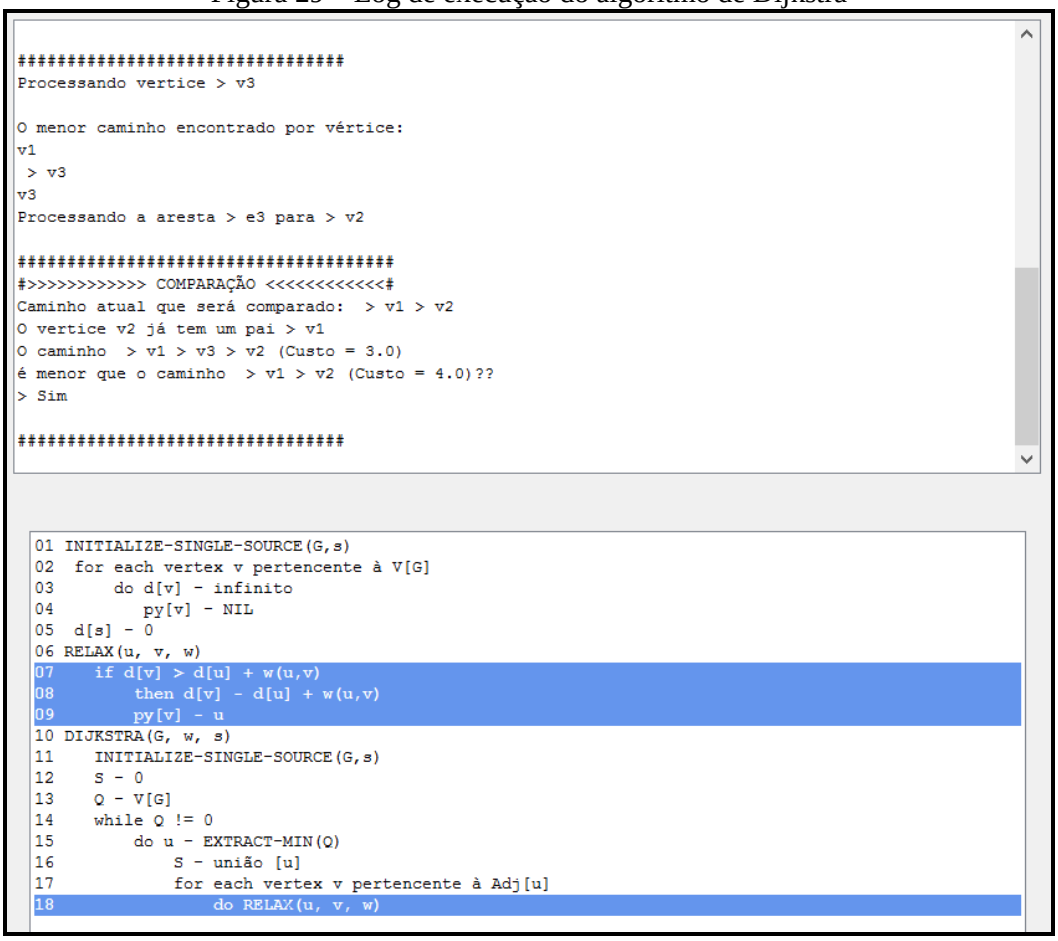
Caso o caminho novo encontrado não tenha um custo menor que o caminho antigo, segue-se para a linha 35 do Quadro 16, que apresenta a informação no *log* de execução informando que o caminho novo não possui custo menor que o antigo. Nas linhas 37 e 38 são extraídos o vértice e aresta pai para variáveis separadas. Já entre as linhas 42 e 45 são atribuídos os pais novos para que no intervalo das linhas 46 a 56 seja montada a lista para colorir as comparações. Logo após, nas linhas 60 e 63 são atribuídos de fato o vértice e aresta pai antigos para que o processo siga normalmente.

Quadro 16 – Comparações de caminhos no algoritmo de Dijkstra

```
34. } else {
35.     this.log_cmp_novo_str += "> Não\n";
36.
37.     Vertice pai_vertice_antigo = v_destino.vertice_pai;
38.     Aresta pai_aresta_antigo = v_destino.aresta_pai;
39.     Vertice pai_aux = (a.getVi().equals(v_destino) ? a.getVj() :
                                                              a.getVi());
40.
41.     //Se não tem pai, adiciona o vertice atual como pai
42.     v_destino.setPai(pai_aux, a);
43.
44.     //Seta o custo como Custo da aresta + Custo do pai
45.     v_destino.set_custo(a.getValor() + pai_aux.get_custo());
46.     this.aresta_cmp_pai_novo_list.add(v_destino.aresta_pai);
47.
48.     //Percorre o caminho de vertices do pai novo até a origem
49.     for (Vertice v : v_destino.get_caminho()) {
50.         if (!this.vertice_cmp_pai_novo_list.contains(v)) {
51.             this.vertice_cmp_pai_novo_list.add(v);
52.             if (!this.aresta_cmp_pai_novo_list.contains(
                            v.aresta_pai) && v.aresta_pai != null){
53.                 this.aresta_cmp_pai_novo_list.add(v.aresta_pai);
54.             }
55.         }
56.     }
57.     this.aresta_list.remove(v_destino.aresta_pai);
58.
59.     //Se não tem pai, adiciona o vertice atual como pai
60.     v_destino.setPai(pai_vertice_antigo, pai_aresta_antigo);
61.
62.    //Seta o custo como Custo da aresta + Custo do pai
63.     v_destino.set_custo(pai_aresta_antigo.getValor() +
                                      pai_vertice_antigo.get_custo());
64. }
```

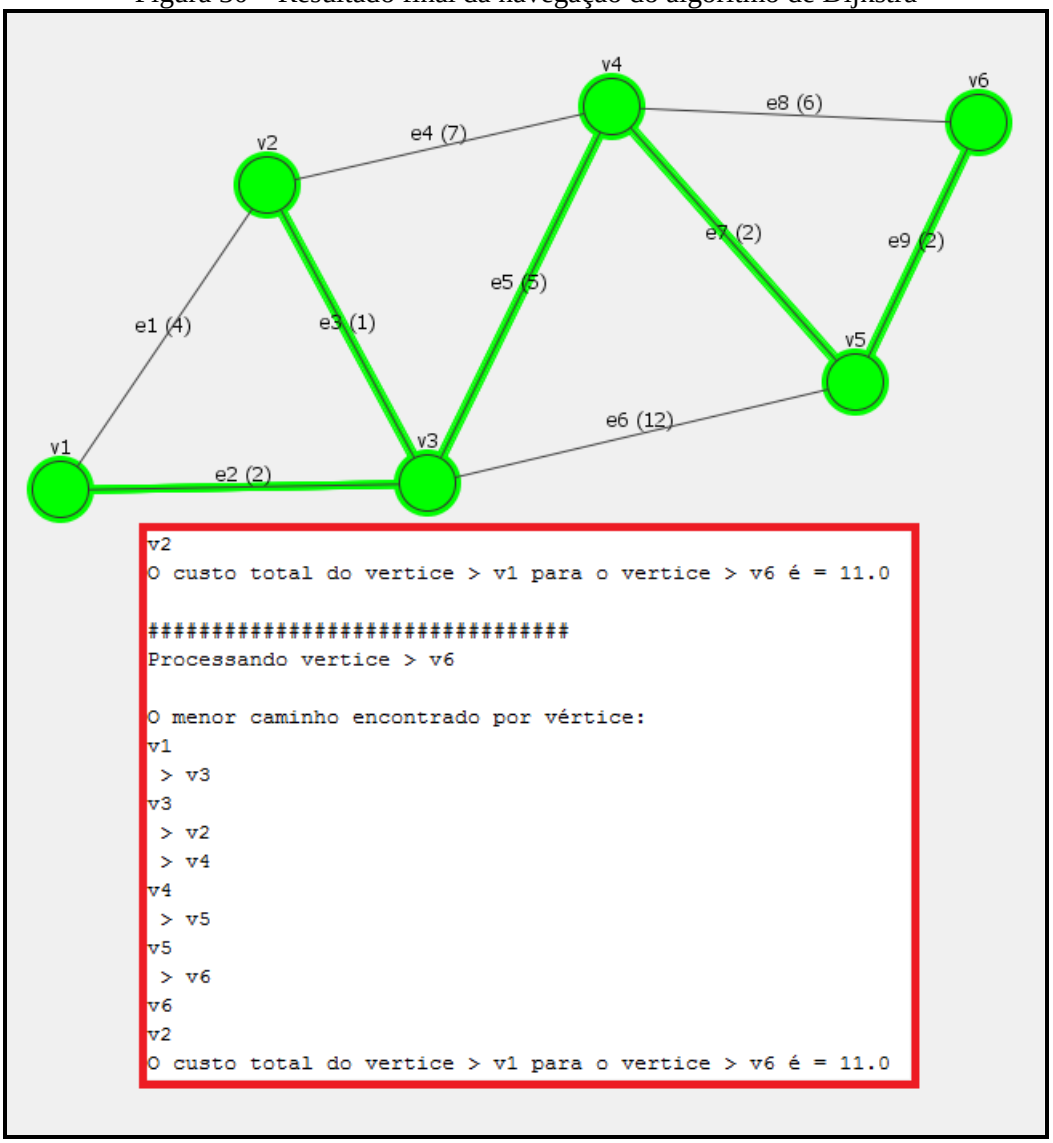
Ainda no Quadro 16, a cada passo do algoritmo o *log* de execução apresenta todo o ocorrido de forma detalhada para que seja possível acompanhar estas comparações. A Figura 29 exemplifica o resultado do *log* de execução e a relação com o pseudocódigo logo abaixo.

Figura 29 – Log de execução do algoritmo de Dijkstra



Assim, quando se tem as listas de comparações preenchidas, os próximos passos avançados colorem o caminho antigo de azul escuro e o novo de azul claro. Depois de cada comparação são coloridos de verde os vértices e arestas que até então representam o caminho mínimo do vértice origem até o vértice destino. Após destacar o caminho mínimo, o caminho que não foi escolhido é automaticamente descolorido, seguindo este processo a cada comparação até que se atinja o vértice de destino selecionado, restando na tela somente os caminhos em verde, que representam os caminhos mínimos entre a origem e cada um dos vértices. Por fim, é apresentado no *log* de execução do algoritmo, o resultado de custo mínimo entre o vértice de origem e de destino que foram inicialmente selecionados. Na Figura 30 é apresentado o resultado final da execução do algoritmo de Dijkstra, com o caminho mínimo da origem para todos os vértices e o custo mínimo da origem ao destino apresentado no log.

Figura 30 – Resultado final da navegação do algoritmo de Dijkstra



3.3.2.9 Retroceder todos os passos do algoritmo

Para que se tenha um aproveitamento melhor da ferramenta, sempre que o usuário estiver executando algum dos algoritmos como BFS, DFS ou Dijkstra, este poderá optar durante a navegação por retornar todos os passos já executados para o início, de tal forma que permita que ele inicie a navegação toda novamente. O método `retorna()` é responsável por limpar todas as variáveis e reiniciar todas as listas e processos que estavam em andamento, iniciando a execução desde o início do algoritmo. O Quadro 17 apresenta parte do método `retorna()`, responsável por limpar os vínculos de vértice pai, informações de tempo de abertura e fechamento e níveis do vetor de roteamento.

Quadro 17 – Algoritmo que retrocede todos os passos da navegação

```
01. for (Vertice v : this.vertice_list) {
02.     if (this.algoritmoDesenho.verticesMarcados.contains(
                                                  findVerticeVisual(v))) {
03.         this.algoritmoDesenho.verticesMarcados.remove(
                                                    findVerticeVisual(v));
04.         this.algoritmoDesenho.coresVertices.remove(
                                                    findVerticeVisual(v));
05.         if (v.aresta_pai != null) {
06.             this.algoritmoDesenho.arestasMarcadas.remove(
                                            findArestaVisual(v.aresta_pai));
07.             this.algoritmoDesenho.coresArestas.remove(
                                            findArestaVisual(v.aresta_pai));
08.             findArestaVisual(v.aresta_pai).name = "";
09.         }
10.         findVerticeVisual(v).value = "";
11.         v.setPai(null, null);
12.         v.set_custo(0.0);
13.         this.repaint();
14.     }
15. }
```

3.3.3 Operacionalidade da implementação

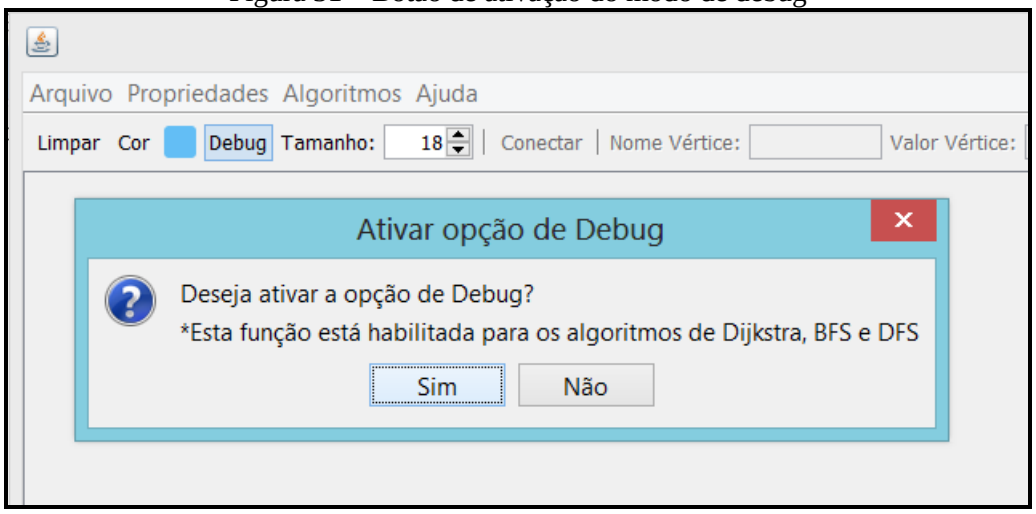
A aplicação visual do FURB Graphs foi desenvolvida por Borba (2014) com intuito de fornecer ao usuário uma tela autoexplicativa que permita a utilização dos seus recursos de forma intuitiva. A aplicação inicial foi construída com quadro menus, uma barra de manipulação de vértices e arestas, uma barra de status e uma área interativa destinada para que o usuário possa desenhar um grafo livremente.

Este trabalho não aborda estes itens listados acima, pois estes foram explicados no trabalho de Borba (2014). Neste trabalho foram implementadas novas funcionalidades e novas opções visuais que tornam a aplicação em uma ferramenta de apoio ao ensino. Neste trabalho, foram adicionados à interface gráfica um botão para ativação do modo de *debug*, dois campos de texto adicionais, um para apresentar os *logs* de execução para os algoritmos BFS, DFS e Dijkstra e um campo para apresentar o *log* com o pseudocódigo referente a cada um dos três algoritmos. Além destes foram adicionadas legendas como informativos a barra de status, para que o usuário possa identificar o significado de cada cor utilizada durante a representação gráfica da navegação em cada algoritmo.

O botão de ativação do modo de *debug* pode ser visualizado na barra superior, abaixo dos quatro menus. Inicialmente o usuário pode carregar um grafo na ferramenta ou então desenhar um novo antes de ativar o modo de *debug*. Quando o usuário utilizar o mouse para clicar no botão *debug*, este apresentará uma mensagem requisitando se o usuário deseja mesmo ativar este modo de execução, informando também que existe uma exceção, onde este

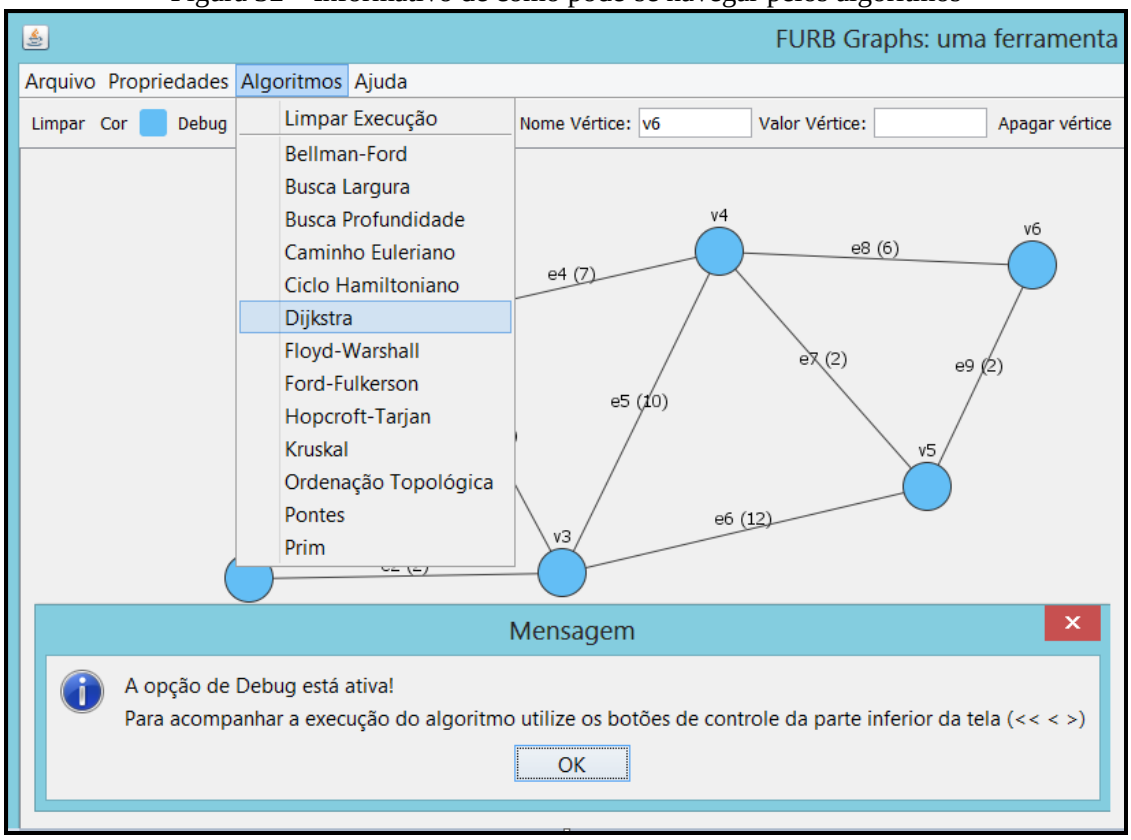
modo funcionará somente para os algoritmos BFS, DFS e Dijkstra, conforme mostra a Figura 31.

Figura 31 – Botão de ativação do modo de debug



Após a ativação do modo de *debug*, o usuário deve selecionar um dos três algoritmos citados para ser executado. Assim que executar um dos três algoritmos e selecionar os vértices requisitados para o devido algoritmo, será apresentado ao usuário uma advertência informando que o modo de debug está ativo, conforme apresenta a Figura 32.

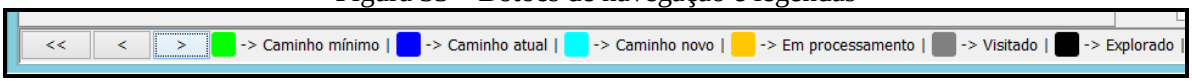
Figura 32 – Informativo de como pode-se navegar pelos algoritmos



Na mensagem apresentada na Figura 32 o usuário é informado de que pode utilizar os botões de controle, ou de navegação que se encontram na parte inferior da tela, representados

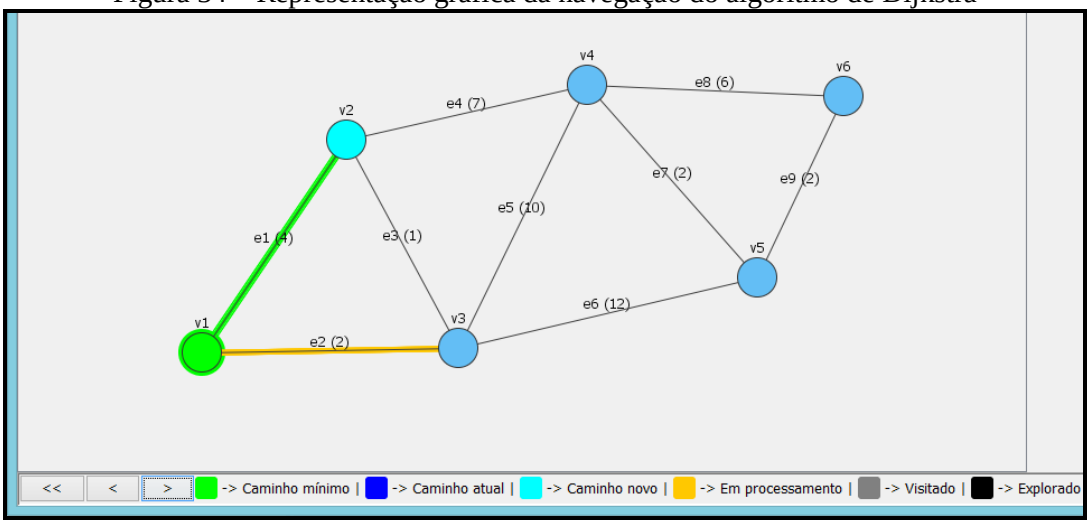
pelos símbolos “<<”, “<” e “>”. Na Figura 33 é possível verificar estes botões e, a direita, as legendas das cores utilizadas na representação da navegação.

Figura 33 – Botões de navegação e legendas



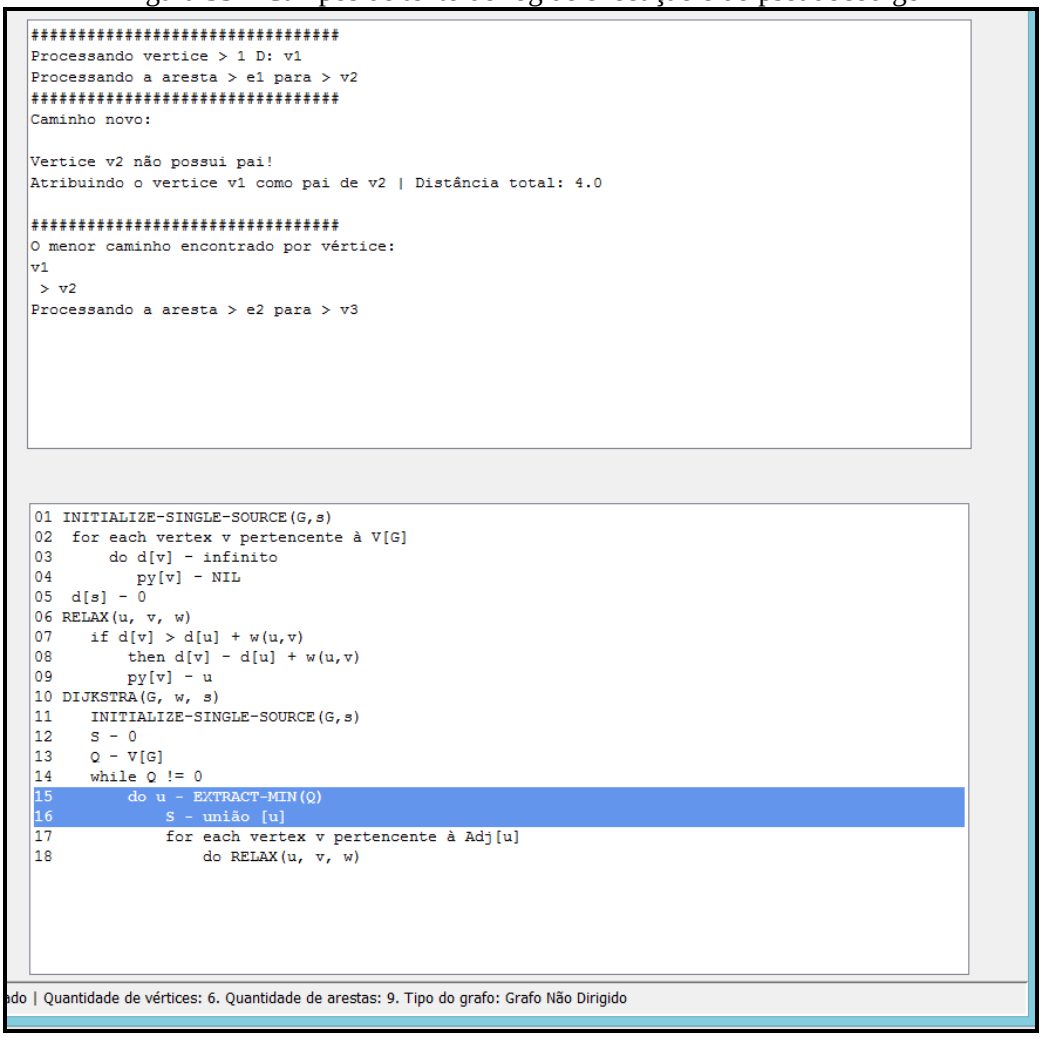
Ao utilizar o botão de navegação para frente, a representação gráfica será acionada, apresentando no grafo a cor referente ao passo executado do respectivo algoritmo. Uma representação gráfica dos primeiros passos de navegação do algoritmo de Dijkstra, conforme pode ser visualizada na Figura 34.

Figura 34 – Representação gráfica da navegação do algoritmo de Dijkstra



O campo de texto do *log* de execução apresenta o detalhamento do passo executado e o campo do pseudocódigo do algoritmo destaca a linha referente a atual execução. As informações apresentadas no *log* e no pseudocódigo durante a navegação do algoritmo de Dijkstra podem ser visualizadas na Figura 35.

Figura 35 – Campos de texto do log de execução e do pseudocódigo



O processo que apresenta as informações das execuções ocorre a cada passo do algoritmo, até que se atinja o objetivo. Se o botão de retornar for pressionado, o último vértice processado será descolorido e os vértices e arestas que foram processados através do mesmo, para que assim se possa reiniciar o processo visualmente, porém o *log* de execução não se altera, mantendo o histórico para que se possa acompanhar o avanço e retrocesso do algoritmo. Já a opção de retornar para o primeiro passo, faz com que todos os vértices e arestas do grafo sejam descoloridos e também faz com que o *log* de execução seja zerado, permitindo o reinicio da navegação no algoritmo.

3.4 RESULTADOS E DISCUSSÕES

Nesta seção são apresentados os experimentos realizados com a ferramenta FURB Graphs. A seção 3.4.1 apresenta o experimento de usabilidade da ferramenta, detalhado por algoritmo e por tarefa executada.

3.4.1 Experimento de usabilidade

O experimento de usabilidade foi realizado com 7 usuários, com perfis variados, para avaliar a aceitação e eficiência das funcionalidades implementadas.

3.4.1.1 Metodologia

O experimento ocorreu no mês de junho de 2016 por meio de testes com os usuários e com acompanhamento. Parte dos testes foram realizados *online*, através de vídeo conferências via Skype ou Hangouts e outra pequena parte foi realizada pessoalmente em computadores locais. Para efetuar os testes, foi disponibilizado para os usuários um executável da ferramenta, para que pudessem realizar os testes em seus próprios computadores. Os usuários acessaram formulários elaborados na ferramenta *online* Google Forms, onde responderam os questionários de perfil de usuário, de tarefas e de usabilidade da ferramenta. Estes questionários estão disponíveis no Apêndice B.

3.4.1.2 Aplicação do teste

Para cada usuário foi identificado o seu nível de familiaridade com a área de teoria dos grafos através do questionário de perfil que foi preenchido por cada usuário. Posteriormente, a ferramenta foi entregue ao usuário e as tarefas foram explicadas separadamente, uma a uma. A lista de tarefas é constituída de 13 questões, estas por sua vez contemplam execuções e análises de resultados obtidos na ferramenta, separado por algoritmo. Após a execução de cada tarefa, o usuário foi orientado a responder se foi possível executar o que foi solicitado e se haviam observações quanto ao procedimento executado.

O questionário de usabilidade da ferramenta foi elaborado com 21 questões fechadas e uma aberta. Todas as questões fechadas procuraram obter as impressões do usuário quanto a usabilidade e quanto ao nível de compreensão dos algoritmos em questão. Ao finalizar o questionário os usuários tiveram a oportunidade de deixar as suas sugestões, opiniões e críticas sobre a ferramenta na questão aberta. Os resultados deste experimento são apresentados na próxima seção.

Os questionários de perfil, lista de tarefas e questionário de usabilidade da ferramenta estão disponíveis no Apêndice B.

3.4.1.3 Análise e interpretação dos dados coletados

A primeira análise foi realizada sobre os dados coletados através do questionário de perfil do usuário. No Quadro 18 são apresentados os perfis dos usuários envolvidos no experimento.

Quadro 18 – Perfil dos usuários envolvidos no teste de usabilidade

Sexo	100 % masculino
Idade	57,1 % tem entre 20 e 25 anos 28,6 % tem entre 25 e 30 anos 14,3 % tem entre 30 e 35 anos
Atua na área de computação?	85,7 % sim 14,3 % não
Nível de escolaridade	14,3 % ensino médio completo 71,4 % ensino superior incompleto 14,3 % ensino superior completo
Compreende o que é um grafo?	71,4 % sim 28,6 % não
Já estudou alguma disciplina envolvendo grafos?	42,9 % sim 57,1 % não

Avaliando os resultados é possível identificar que o público de usuários é variado entre idades e níveis de escolaridade diferentes. É possível identificar também que a maioria dos usuários atua na área de computação. Quanto ao conhecimento específico na área de grafos, a maioria possui um conhecimento sobre o conceito de grafos, identificando o que é e do que se trata. Porém, muitos deles não chegaram a estudar teoria dos grafos ou algo relacionado. O que demonstra que o assunto é disseminado de forma natural e lógica entre os que atuam na área de computação.

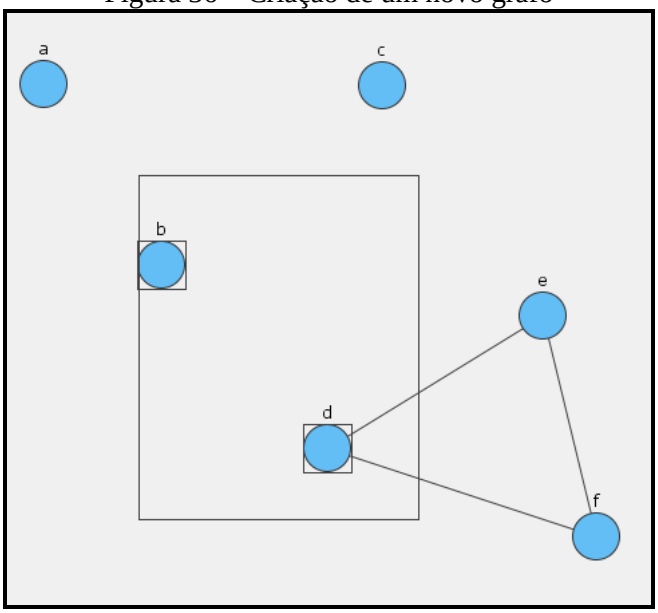
3.4.1.3.1 Análise dos resultados da lista de tarefas

Após realizar a análise de perfil dos usuários foram avaliados os resultados e iniciadas as instruções para aplicação da lista de tarefas. A lista de tarefas é dividida entre atividades iniciais, que são pré-requisitos para as demais tarefas e as que se referem a cada algoritmo separadamente.

Todos os usuários conseguiram abrir o executável da ferramenta em seus computadores. Nesta etapa já se identificou a diferença entre as versões do Java instalado em cada máquina, que foi desenvolvido na versão 7, porém um dos computadores tinha somente o Java 8 que abriu a ferramenta normalmente. Porém, a interface gráfica foi apresentada com dimensões diferentes do original, fato que não interferiu na usabilidade ou nos procedimentos. Além das versões diferentes do Java também foram utilizadas todas máquinas Windows em versões diferentes, tais como: 8, 8.1 e 10, onde todos funcionaram normalmente.

Após abrir o executável foi solicitado para que executassem as tarefas iniciais. A primeira constituía em criar ou carregar um novo grafo na ferramenta com no mínimo 6 vértices e 9 arestas, de forma que o grafo estivesse totalmente conectado, representando assim um grafo conexo. A maioria dos usuários optou por desenhar o próprio grafo e isto diversificou os testes. Outros em minoria decidiram carregar um grafo de teste já pronto. Visto que o foco é a execução e navegação dos algoritmos, estes passos iniciais poderiam ser ignorados e todos poderiam ter escolhido carregar um grafo já pronto. A criação de um novo grafo pode ser visualizada na Figura 36.

Figura 36 – Criação de um novo grafo



Os usuários que desenharam seu próprio grafo, adicionaram os devidos vértices e arestas e, posteriormente, atribuíram um nome para cada vértice e um valor para cada aresta. Depois deste passo, foi requisitado para que se ativasse o modo de *debug* da ferramenta, que poderia ser habilitado através do botão *debug* na barra de tarefas. Tendo este modo ativo, concluía-se as tarefas iniciais e os resultados podem ser visualizados no Quadro 19.

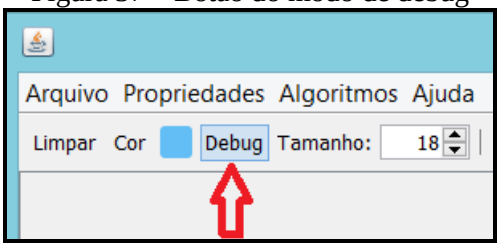
Quadro 19 – Respostas quanto as tarefas de criação do grafo e ativação do modo debug

Tarefas/Respostas	**Sim**	**Não**
Carregue ou desenhe um novo grafo com o mínimo de 6 vértices e 9 arestas, onde todos os vértices estejam interligados	100 %	
Se o grafo foi carregado, pule para a questão 4. Caso contrário, utilize o mouse clicando para criar vértices na tela, selecione cada vértice e atribua um nome a cada um	57,1%	42,9 %
Selecione cada par de vértice, clique com o botão direito e selecione conectar. Selecione cada par de vértice que possuir uma aresta e atribua um valor numérico aleatoriamente para cada	57,1%	42,9%
Utilize o mouse para clicar no botão “debug” no menu de ações da ferramenta e confirme a ativação do modo de debug	100 %	

A partir do Quadro 19, pode-se avaliar que todos conseguiram realizar as tarefas principais que seriam referentes a geração de um novo grafo e a ativação do modo de *debug*. Os 42,9% que responderam que não conseguiram criar o grafo, se referem aos usuários que carregaram grafos já prontos para dar continuidade aos testes.

Alguns usuários tiveram dificuldades na criação de um novo grafo, porém conseguiram resolver sozinhos. Outro ponto importante foi que alguns não conseguiram identificar facilmente a localização do botão de *debug* na ferramenta, sugerindo que o mesmo deveria ficar mais destacado na interface gráfica. A Figura 37 mostra a localização do botão de *debug* na ferramenta.

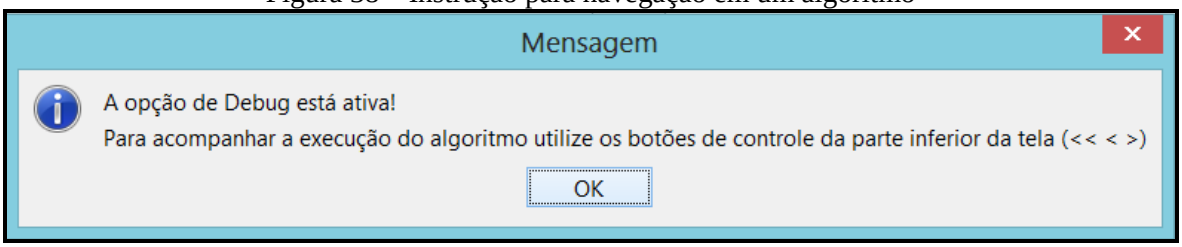
Figura 37 – Botão do modo de debug



Finalizadas as tarefas iniciais, os usuários estavam aptos para testar os procedimentos de navegação sob os algoritmos de BFS, DFS e Dijkstra que são solicitados nas demais tarefas. Para iniciar esta etapa, foi escolhido o algoritmo de busca em largura (BFS). Foi solicitado para que o usuário selecionasse no menu “Algoritmos” a opção “Busca Largura” e para que ele atendesse as instruções apresentadas na tela.

Ao executar um dos três algoritmos a ferramenta requisita para que o usuário selecione um vértice inicial no grafo. Se o modo de *debug* estiver ativo, a ferramenta apresenta uma informação instruindo o usuário deste modo e que nesta situação ele deve utilizar os botões de navegação, que ficam situados no canto inferior esquerdo da tela para acompanhar os algoritmos passo a passo. Conforme apresentado na Figura 38.

Figura 38 – Instrução para navegação em um algoritmo

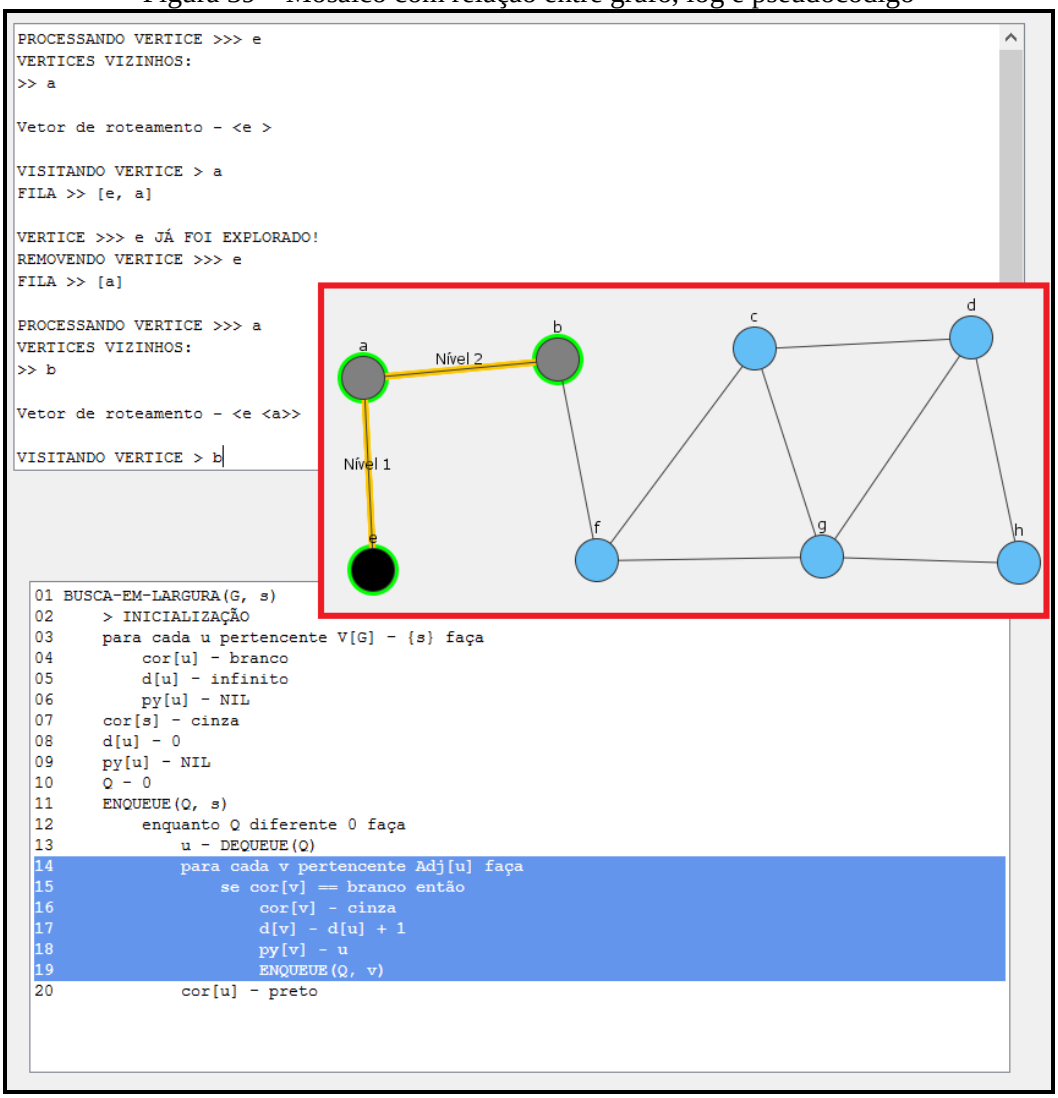


Após confirmar a mensagem, o pseudocódigo do algoritmo selecionado é apresentado no campo de texto situado no canto direito inferior. Neste momento os botões de navegação passam a interagir com o grafo e respectivamente com o *log* de execução e com o pseudocódigo. Na próxima tarefa, foi requisitado que o usuário executasse cada passo do algoritmo separadamente, levando em consideração e analisando a relação entre a coloração dos vértices e arestas com o *log* de execução e com a linha do pseudocódigo que foi destacada.

Alguns usuários informaram que o campo de texto destinado ao pseudocódigo do algoritmo fosse invertido de lugar com o campo do *log* de execução, informando que prestam mais atenção nas informações que estão na parte superior e que neste ponto que deveria se localizar o pseudocódigo do algoritmo. Após estes passos, em todos os algoritmos foi requisitado ao usuário que navegasse livremente utilizando os botões. Durante este processo, a maioria dos usuários conseguiram realizar ligações lógicas entre a coloração e o *log* de execução, informando que estava detalhado e informativo, permitindo que compreendessem o algoritmo somente com estas informações.

A relação destas representações com o pseudocódigo apresentado, ficou restrita aos usuários que atuam na área de computação e aos que já tinham algum conceito ou conhecimento sobre grafos. A Figura 39 apresenta estas relações em execução do algoritmo BFS ao avançar alguns passos iniciais.

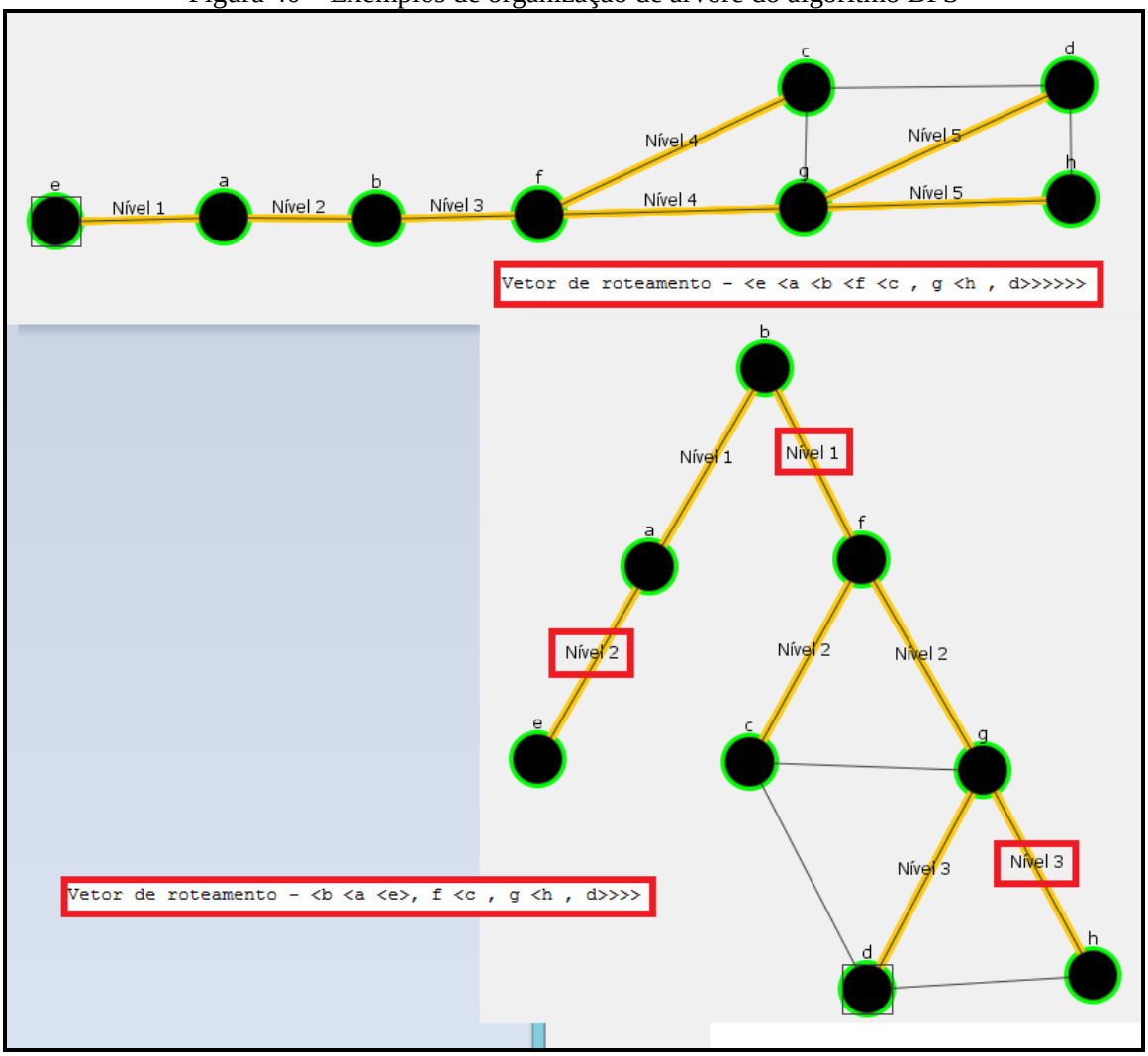
Figura 39 – Mosaico com relação entre grafo, log e pseudocódigo



Durante o processo de retrocesso dos passos deste algoritmo, em alguns casos foram encontradas situações inesperadas, onde dependendo do grafo que foi desenhado, ao retornar e avançar em passos específicos ocorriam problemas na coloração dos vértices, indicando o procedimento de forma indevida. Porém, nestas situações, ao retroceder todos os passos e iniciar novamente o processo ocorria normalmente. Este problema não foi corrigido, pois, ocorreu em casos muito específicos e foi identificado em menos da metade dos testes. Mas, mesmo nestes casos, os usuários puderam compreender completamente o algoritmo sem demais problemas.

Assim que os usuários finalizaram a navegação no algoritmo BFS, foi solicitado para que clicassem nos vértices e os arrastassem na tela, um por um, organizando-os da forma que mais fizesse sentido para eles, porém, levando em consideração o vetor de roteamento e as informações apresentadas nas arestas dos grafos. A Figura 40 apresenta dois exemplos de organizações do grafo realizadas pelos usuários.

Figura 40 – Exemplos de organização de árvore do algoritmo BFS



Estes mesmos usuários informaram que a relação ficou de fácil compreensão e que era possível visualizar uma aplicação depois do algoritmo executado e organizado desta forma. O Quadro 20 demonstra os resultados das tarefas relacionadas a navegação e análise do algoritmo BFS.

Quadro 20 – Respostas das tarefas de navegação do algoritmo BFS

Tarefas / Respostas	**Sim**	**Não**
Utilize o mouse no menu “Algoritmos” e selecione a opção Busca Largura. Siga as instruções apresentadas na tela	100%	
Utilize o mouse no botão de navegação para avançar cada passo do algoritmo. Analisando a coloração dos vértices e arestas em relação com o log de execução e o pseudocódigo em cada passo	100%	
Utilize o mouse nos botões de navegação para retroceder um passo ou para retroceder todos os passos. Após avance e retroceda os passos conforme desejar	100%	
Avance todos os passos do algoritmo e, utilizando o mouse para arrastar os vértices, organize-os se orientando através dos níveis indicados nas arestas, nível 1, nível 2.. seguindo a ordem	100%	

Desta forma, para os demais algoritmos, DFS e Dijkstra, foram requisitadas as mesmas tarefas iniciais, solicitando que o usuário primeiramente avançasse todos os passos dos algoritmos, avaliando o *log* de execução, legendas e o pseudocódigo. Para o DFS foi requisitado que o usuário se atentasse quanto ao empilhamento e desempilhamento durante a navegação. Já para o Dijkstra, foi requisitado que se atentasse as comparações que fossem apresentadas durante a navegação. Em geral, no algoritmo DFS não houveram situações inesperadas, onde todos os usuários conseguiram avançar e retroceder sem problemas, assim como conseguiram compreender seu funcionamento, muitas vezes informando que o pseudocódigo era desnecessário para tal compreensão do funcionamento.

Para o algoritmo de Dijkstra, nos primeiros testes foram encontrados alguns problemas nas comparações, onde o algoritmo considerava somente a última comparação feita a partir de do vértice pivô para se colorir na tela. Para solucionar este problema, foi reestruturada a rotina de comparações para armazenar todas as comparações que partissem do mesmo vértice processado. Este problema foi identificado e logo após corrigido, depois, nos demais testes não ocorreram situações inesperadas neste algoritmo.

O Quadro 21 estratifica as respostas quanto as tarefas de navegação dos algoritmos DFS e Dijkstra.

Quadro 21 – Respostas das tarefas de navegação nos algoritmos DFS e Dijkstra

Tarefas / Respostas	Sim	Não
Utilize o mouse no menu “Algoritmos” e selecione a opção Busca Profundidade. Siga as instruções apresentadas na tela	100%	
Utilize o mouse no botão de navegação para avançar cada passo do algoritmo. Analisando a coloração dos vértices e arestas em relação com o log de execução e o pseudocódigo em cada passo	100%	
Utilize o mouse nos botões de navegação para retroceder um passo ou para retroceder todos os passos. Após avance e retroceda os passos conforme desejar	100%	
Utilize o mouse no menu “Algoritmos” e selecione a opção Dijkstra. Siga as instruções apresentadas na tela	100%	
Utilize o mouse para avançar os passos do algoritmo através do botão de navegação indicado na tela, verifique as legendas, logs e o pseudocódigo a cada passo	100%	

Para o algoritmo de Dijkstra ainda foi afirmado, pela maioria dos usuários, que este foi o mais complexo para se compreender, que as legendas com cores diversificadas na ferramenta foram importantes especialmente neste algoritmo para a compreensão de todos os estados dos vértices. Para os usuários este algoritmo foi o mais fácil de se fazer uma relação com aplicações reais, tal como mapas, que foram citados nos testes.

Em geral, todas as tarefas foram executadas normalmente e durante o processo, a cada tarefa foram fornecidas informações sucintas sobre o algoritmo e sobre o passo avançado do mesmo. Desta forma os usuários ao executarem os primeiros passos com informações básicas foram capazes de compreender cada passo posterior do algoritmo, chegando ao ponto em que alguns usuários analisando o *log* e o código fonte, prediziam o que iria ocorrer no próximo passo, dando detalhes tais como: qual vértice será processado, qual vértice será empilhado ou desempilhado. Todos os usuários demonstraram interesse na ferramenta e nos grafos após a navegação nos algoritmos, informando que o fato de representar graficamente fazia com que a compreensão dos passos se tornasse intuitiva.

3.4.1.3.2 Análise quanto a usabilidade e funcionalidades

Foi aplicado um questionário com os usuários referente a avaliação a usabilidade e as funcionalidades da ferramenta assim quanto ao nível de compreensão dos algoritmos. Foram elaboradas ao total 21 questões para avaliar a ferramenta, onde 9 se referem a usabilidade e funcionalidades da ferramenta e 12 questões foram reservadas para a compreensão dos algoritmos. O Quadro 22 apresenta a resposta das questões elaboradas para avaliar a usabilidade e as funcionalidades da ferramenta.

Quadro 22 – Avaliação de usabilidade e das funcionalidades da ferramenta

Perguntas / Critérios de avaliação	**Concordo totalmente**	**Concordo parcialmente**	**Foi imparcial**	**Discordo totalmente**
1. De modo geral, você acredita que a ferramenta foi intuitiva e de fácil utilização?	71,4%	28,6%		
2. Para você, a interface gráfica e a disposição dos campos facilitaram a utilização da ferramenta?	28,6%	71,4%		
3. Você consegue se lembrar facilmente de como fazer as operações na ferramenta?	100%			
4. Você achou importante a utilização das legendas na ferramenta?	71,4%	14,3%	14,3%	
5. O modelo de navegação através dos botões foi prático e interativo?	57,1%	42,9%		
6. Você precisaria de ajuda para operar a ferramenta?	14,3%	28,6%		57,1%
7. A ferramenta em algum momento apresentou algum comportamento inesperado?	14,3%	42,9%	14,3%	28,6%
8. Seria adequado recomendar esta ferramenta para outras pessoas que estudam grafos?	100%			
9. A representação gráfica do passo a passo contribui para o interesse em grafos?	100%			

Através do Quadro 22 é possível analisar diversas informações relacionadas aos perfis dos usuários. A maioria dos usuários conseguiu utilizar a ferramenta de forma intuitiva e sem dificuldades, quanto aqueles com menos contato com a área de computação e com grafos precisaram de algumas instruções para começar a operar a ferramenta. A interface gráfica foi interpretada por cada usuário de uma forma diferente, alguns criticaram o fato do *log* de execução se encontrar acima do pseudocódigo, outros comentaram sobre manter a área de desenho do grafo no centro da tela e as demais informações nos cantos. Outras sugestões ainda foram sobre a posição das legendas que deveriam ficar à esquerda da tela e não na parte de baixo. Em outras situações sugeriram que se fosse adicionado um componente *split* que permitiria esconder os *logs*, pseudocódigo e legendas sempre que o usuário os arrastasse para fora da área de desenho. Desta forma a interface gráfica foi avaliada de diversos pontos de vista, ficando bem dividida nas avaliações dos usuários.

A ferramenta possui operações simples e bem instrutivas, conforme afirmou um usuário, por isso é fácil lembrar de como operá-la. Outro ponto importante e de diversas opiniões foi a utilização de legendas na ferramenta, onde para alguns foi algo que passou

despercebido afirmando que as cores e o *log* foram o suficiente. Para outros usuários as legendas foram identificadas na tela bem depois do início dos testes, casos em que informaram que dificultou pois precisavam das legendas para se orientar quanto aos estados dos vértices. Quanto a avaliação dos botões de navegação, foi constatado por todos que foi prático de se utilizar, porém, todos preferiam que houvessem atalhos nas setas do teclado para se navegar no algoritmo.

Referente a dificuldade de operar e compreender o que se fazer na ferramenta ficou definida pela falta de conhecimento e ou afinidade com a área de computação e com grafos, onde, aqueles que nunca estudaram grafos e ou não são da área de computação responderam que necessitariam de ajuda, no mínimo de explicações quanto ao significado e função de cada algoritmo. Já os usuários mais familiarizados responderam que se ficassem poucos minutos mexendo na ferramenta já conseguiriam opera-la normalmente.

Durante os testes de navegação foram encontrados alguns problemas nos algoritmos BFS e Dijkstra. O primeiro grupo de usuários a testar encontraram estes problemas e logo após foi aplicada uma correção nos erros mais críticos e nos que mais ocorriam, restando apenas os que ocorreram em situações muito específicas. Este fato explica a diversidade das respostas quanto comportamentos inesperados apresentados pela ferramenta. No mais, a ferramenta teve diversos elogios quanto a como seria mais fácil de se estudar através de algo visual. Outro ponto importante foi a citação dos usuários informando que tinham se interessado pelo assunto após estudar os algoritmos de uma forma dinâmica, onde os que não tinham familiaridade com a área compreenderam o funcionamento dos algoritmos e suas características sem ter de compreender completamente o pseudocódigo relacionado.

Relacionado ainda à compreensão dos algoritmos, foi elaborado o questionário que é apresentado no Quadro 23, que apresenta diversas questões gerais sobre como as funcionalidades novas da ferramenta ajudaram na compreensão de cada algoritmo. Inicialmente foi constatado que as cores utilizadas ficaram bem definidas para os usuários, principalmente alguns usuários que citaram que o verde é uma cor que demonstra um caminho certo, o azul chama a atenção nas verificações e o preto faz com que eles compreendam que aquele foi o último estado do vértice em questão. Desta forma permitindo a melhor compreensão e relação com os estados dos algoritmos.

O *log* de execução foi bem avaliado pela maioria dos usuários, pois, ele apresentou todas as informações relacionadas com a coloração representada no grafo a cada passo. Porém, inicialmente os usuários menos familiarizados se sentiram perdidos em alguns passos e aos poucos compreenderam a relação normalmente.

Quadro 23 – Avaliação de compreensão da navegação dos algoritmos

Perguntas / Critérios de avaliação	**Concordo totalmente**	**Concordo parcialmente**	**Foi imparcial**	**Discordo totalmente**
1. Na sua opinião, as cores utilizadas na representação gráfica contribuíram para a compreensão dos algoritmos?	85,7%	14,3%		
2. O log de execução foi claro e detalhado o suficiente para acompanhar os passos dos algoritmos?	71,4%	28,6%		
3. O pseudocódigo ajudou para a compreensão dos algoritmos?	28,6%	14,3%	57,1%	
4. Você teve dificuldades para interpretar cada passo avançado nos algoritmos?	14,3%			85,7%
5. A funcionalidade de retroceder um passo do algoritmo se fez útil para compreensão?	100%			
6. A funcionalidade de retroceder todos os passos do algoritmo se fez útil para compreensão?	100%			
7. A combinação das legendas, logs, pseudocódigo e representação gráfica juntos fizeram sentido para você?	85,7%	14,3%		
8. A compreensão geral dos algoritmos melhora depois de utilizar a ferramenta?	100%			
9. Para o estudo de grafos, seria útil utilizar esta ferramenta?	100%			
10. No algoritmo de busca em largura, você compreendeu o vetor de roteamento em relação com a árvore gerada?	71,4%	28,6%		
11. No algoritmo de busca em profundidade, o empilhamento e desempilhamento do algoritmo de busca em profundidade ficou visível e compreensível?	100%			
12. No algoritmo de Dijkstra, as comparações dos caminhos foram compreendidas?	42,9%	57,1%		

Ainda em relação a avaliação do Quadro 23, o pseudocódigo apresentado foi avaliado de diferentes formas, onde os menos familiarizados o desconsideraram e compreenderam através das cores, passos e detalhes do *log* de execução e os demais ficaram entre os que prestaram mais atenção ao próprio código fonte a cada passo e os que olharam uma vez a cada passo inicial e não o deram mais atenção. Quanto a dificuldade e ou demora de relacionar as representações e interpretar os passos avançados ficou restrito aos usuários que não eram da área de computação. As funcionalidades de retroceder passo a passo e de retroceder todos os

passos foi indicada por todos os usuários como ponto importante para se compreender mesmo o funcionamento do algoritmo, podendo ressaltar aqui que os que tinham mais dificuldades retornaram algumas vezes para retomar alguns passos e compreender o ocorrido. A relação de todos as informações na tela foi para todos bem instrutiva e fez sentido a todos em níveis diferentes, entre os que conheciam os algoritmos e os que não conheciam.

Em geral, os usuários informaram ao terminar os testes que compreendiam realmente qual era a função de cada algoritmo e aqueles que não os conheciam passaram a compreender sua essência. Ainda sobre a compreensão dos algoritmos, se fez importante a citação dos usuários questionando se esta ferramenta seria utilizada em aula e demonstrando interesse em estudar utilizando tal meio interativo.

Após as validações gerais de compreensão da ferramenta como um todo, foi questionado a cada usuário sobre características importantes dos algoritmos. Para o BFS foi questionado quanto a compreensão da árvore gerada ao navegar no algoritmo, em relação com a árvore representada no vetor de roteamento do *log*. A maioria dos usuários conseguiu relacionar facilmente e identificar por exemplo, pontos que marcavam quando um vértice faria parte do vetor de roteamento. Outros tiveram dificuldades devido aos problemas apresentados nos primeiros testes e pela falta de familiaridade. O DFS foi totalmente compreendido quanto a questão solicitada, esta que questionava ao usuário se ele compreendeu o processo de empilhamento e desempilhamento do algoritmo. Todos conseguiram apontar facilmente no *log* e no grafo quando que se estava empilhando um vértice e quando se estava desempilhando. Para o algoritmo de Dijkstra houveram mais problemas, pois, os erros identificados nos primeiros testes foram resolvidos somente antes dos últimos testes efetuados. Porém, a parte mais importante relacionada a comparação dos caminhos para encontrar o caminho mínimo foi compreendida e até explicada por alguns usuários. O que ajudou muitos usuários neste algoritmo foi a coloração, onde a cor que identificava um caminho novo e a cor do caminho atual foram assimiladas rapidamente, assim facilitava para identificar este processo importante do algoritmo.

3.4.2 Comparação com trabalhos relacionados

O Quadro 24 apresenta um comparativo entre a ferramenta desenvolvida e os trabalhos relacionados. Os dados utilizados para a comparação são baseados nos experimentos realizados.

Quadro 24 – Comparativo com os trabalhos relacionados

Características / trabalhos	**Lozada (2014)**	**Santos e Costa (2007)**	**Sangiorgi (2004)**	**Borba (2014)**	**Trabalho proposto**
permite a criação de grafos livremente	Sim	Sim	Sim	Sim	Sim
suporta dígrafos	Sim	Sim	Sim	Sim	Sim
permite trocar as imagens dos vértices	Sim	Não	Sim	Não	Não
executa busca em grafos (BFS, DFS)	Sim	Sim	Não	Sim	Sim
executa árvore geradora mínima (Kruskal e Prim)	Não	Sim	Não	Sim	Sim
executa caminho mínimo entre vértices (Dijkstra e Bellman-Ford)	Sim	Sim	Não	Sim	Sim
analisa ciclos Hamiltonianos	Não	Não	Sim	Sim	Sim
analisa grafos bipartidos	Não	Não	Sim	Sim	Sim
apresenta informativos referente aos algoritmos	Não	Sim	Não	Não	Sim
apresenta material teórico quanto aos algoritmos	Não	Sim	Não	Não	Sim
apresenta recursos de apoio ao aprendizado dos algoritmos (legendas, código fonte)	Sim	Sim	Sim	Não	Sim
permite carregar um código fonte de terceiro para utilizar na execução	Não	Não	Sim	Não	Não
possui plataforma de ampla disponibilidade (web)	Não	Sim	Não	Não	Não
permite o acompanhamento passo a passo dos algoritmos	Não	Sim	Não	Não	Sim
permite o acompanhamento do log de execução passo a passo dos algoritmos	Não	Não	Não	Não	Sim

Através do Quadro 24 é possível identificar que a ferramenta desenvolvida neste trabalho apresenta diversas características relacionadas ao ensino de grafos que se diferem da ferramenta de Borba (2014). Entre estas características pode se citar a apresentação de informações instrutivas referente aos algoritmos e as suas execuções, material teórico e recursos interativos para relacionamento de informações com o algoritmo, tais como: logs, legendas e código fonte.

Também é importante citar os ajustes que foram diferenciais do trabalho de Borba (2014) na execução dos algoritmos DFS e de Dijkstra. Onde o DFS considerava apenas grafos conexos para a sua execução, não considerando os grafos desconexos onde o algoritmo deve continuar o processamento mesmo nos vértices desconexos do vértice inicial. Tal ajuste foi adicionado a ferramenta para que o usuário pudesse compreender o algoritmo corretamente. No Dijkstra a ferramenta de Borba (2014) somente levava em consideração o cálculo de caminho mínimo entre o vértice de origem e destino, quando na definição do algoritmo o

Dijkstra calcula o caminho mínimo do vértice de origem para todos os demais vértices atingíveis do grafo. Esta funcionalidade também foi adicionada a ferramenta para obter o funcionamento adequado durante a navegação.

Entre os demais trabalhos correlatos, o que mais se enquadra entre as ferramentas de ensino é o trabalho de Santos e Costa (2007) que apresenta informações didáticas referente aos algoritmos, permite a navegação passo a passo e demonstra alguns estados dos vértices durante o processamento. O diferencial quanto a ferramenta desenvolvida está na questão de apresentar o *log* de execução detalhado por passo avançado do algoritmo, permitindo que entre a representação gráfica e o código fonte o usuário tenha um meio termo descritivo e detalhado.

O trabalho de Lozada (2014) se diferencia da ferramenta desenvolvida pois não possui foco de ensino e sim de execução dos algoritmos, onde não permite o detalhamento passo a passo de cada um, porém apresenta o código fonte relacionado a execução. Por fim, o trabalho de Sangiorgi (2004) apresenta uma característica única que permite o usuário criar o seu próprio código fonte ou alterar um já existente e então executá-lo, que se torna um ponto atrativo pois o usuário pode testar se o seu algoritmo funciona de fato. Porém, o diferencial desta ferramenta comparada ao trabalho de Sangiorgi (2004) está na capacidade dela de detalhar cada passo do algoritmo durante a navegação.

4 CONCLUSÕES

Este trabalho propôs a adaptação da aplicação FURB Graphs de Borba (2014) de tal forma que se tornasse uma ferramenta de apoio ao ensino para a disciplina de grafos. O foco deste trabalho foi de alterar as rotinas que foram desenvolvidas no trabalho de Borba (2014), que permitiam uma execução e resultados técnicos, com intuito de tornar possível o aprendizado sobre o funcionamento, lógica e uma melhor compreensão dos algoritmos BFS, DFS e Dijkstra.

Para o desenvolvimento desta ferramenta, foi mantido a implementação em Java no ambiente de desenvolvimento Eclipse Mars 1. Durante o desenvolvimento foram encontrados alguns problemas somente em relação a utilização da linguagem Java por falta de contato constante com a tecnologia.

Baseando-se nos testes e as respostas obtidas nos questionários, pode-se avaliar que os resultados alcançados foram satisfatórios. A ferramenta atingiu o objetivo de fazer com que o usuário consiga compreender o funcionamento dos algoritmos BFS, DFS e Dijkstra de tal forma que este possa fazer relações com possíveis aplicações reais e que possa explicar os passos do algoritmo. A utilização das cores simbolizando cada passo, estado e processamento do algoritmo facilitou para que o usuário compreendesse os passos avançados de forma visual e interativa, onde ele escolhe quando avançar ou retroceder cada passo, permitindo a compreensão de cada passo no seu próprio tempo, analisando as informações do *log* de execução, relacionando as cores e o pseudocódigo. Desta forma permitindo que usuários sem muito conhecimento na área de computação ou com nível de abstração diferenciado também possam compreender com mais facilidade os algoritmos.

Uma limitação foi encontrada durante o desenvolvimento. Esta refere-se ao procedimento de retroceder um passo de cada vez durante a navegação no algoritmo de Dijkstra. Para este processo inicialmente foi adotada uma estratégia complexa que não se mostrou eficaz na execução da funcionalidade proposta e acabou por consumir mais tempo do que o esperado, portanto este item não foi concluído. Desta forma a navegação através do algoritmo de Dijkstra ainda manteve o seu objetivo, pois foi possível avançar passos e retornar todos os passos de uma só vez, permitindo ainda que os usuários retornassem sempre que necessário para reiniciar a navegação, com a única diferença de não ser possível o retrocesso passo a passo.

É possível concluir que este trabalho se tornou mais uma alternativa de ferramenta auxiliar para ensino de teoria dos grafos, quanto as temáticas de busca e caminhamento em

grafos, abordando os algoritmos BFS, DFS e Dijkstra de forma visual e interativa. Por fim, este trabalho pode servir como base para futuros trabalhos e pesquisas em relação à ensino e melhoria da abstração de algoritmos complexos e fundamentais para computação.

4.1 EXTENSÕES

Para extensões deste trabalho propõem-se:

a) implementação da funcionalidade de retroceder passo a passo a navegação do algoritmo de Dijkstra;
b) implementação de atalhos para se navegar na execução dos algoritmos;
c) implementação da navegação completa no algoritmo de Prim;
d) implementação da navegação completa no algoritmo de Kruskal;
e) implementação da matriz de adjacência para apresentar na tela durante a navegação dos algoritmos;
f) implementação da matriz de incidência para apresentar na tela durante a navegação dos algoritmos;
g) alteração na interface gráfica para permitir a ocultação do *log* e pseudocódigo;
h) efetuar melhorias de desempenho nos algoritmos de navegação;
i) efetuar melhorias de usabilidade.

REFERÊNCIAS

BORBA, Anderson. **Furb Graphs**: uma aplicação para teoria dos grafos. 2014. 91 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

BUENO, Letícia R. **Algoritmos em grafos**: busca em profundidade. Santo André. 2014. Disponível em: < http://professor.ufabc.edu.br/~leticia.bueno/classes/teoriagrafos/materiais/dfs.pdf>. Acesso em: 23 set. 2015.

CASTRO, Carlos E. P. S. de. **Dijkstra**. 2016. Disponível em: <http://www.deinf.ufma.br/~portela/ed211_Dijkstra.pdf>. Acesso em: 25 jun. 2016.

COSTA, Polyana P. **Teoria de grafos e suas aplicações**. 2011. 77 f. Dissertação (Mestrado Profissional em Matemática Universitária) – Curso de Pós-Graduação em Profissional em Matemática, Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista, Rio Claro.

FEOFILOF, Paulo; KOHAYAKAWA, Yoshiharu; WAKABAYASHI, Yoshiko. **Uma introdução sucinta a teoria dos grafos**. São Paulo. 2011. Disponível em: < http://www.ime.usp.br/~pf/teoriadosgrafos/texto/TeoriaDosGrafos.pdf>. Acesso em: 16 set. 2015.

HOLANDA, Bruno. **Teoria dos grafos**. 2011. Disponível em: <http://www.obm.org.br/export/sites/default/semana_olimpica/docs/2011/Nivel1_grafos_bruno.pdf>. Acesso em: 24 jun. 2016.

KAESTNER, Celso A. A. **Grafos e algoritmos de busca**. Florianópolis. 2013. Disponível em: < http://www.dainf.ct.utfpr.edu.br/~kaestner/MatematicaDiscreta/Conteudo/Algoritmos/l13-graph-search.pdf>. Acesso em: 04 jun. 2016.

KLEINSCHMIDT, João H. **Comunicação e redes**: algoritmos em grafos. Santo André. 2011. Disponível em: < http://professor.ufabc.edu.br/~joao.kleinschmidt/aulas/comrede2011/aula_06_cr.pdf>. Acesso em: 21 set. 2015.

LOBO, Fernando**. Percursos e conectividade em grafos**: deep first search. 2012a. Disponível em: <http://www.fernandolobo.info/aed-II/teoricas/a19.print.pdf>. Acesso em: 25 jun. 2016.

_____. **Caminho mais curto a partir de um nó**: algoritmos de Dijkstra e Bellman-Ford. 2012b. Disponível em: <http://www.fernandolobo.info/aed-II/teoricas/a24e25.print.pdf>. Acesso em: 25 jun. 2016.

LOZADA, Luis A. P. **A-Graph**: uma ferramenta computacional de suporte para o ensino-aprendizado da disciplina teoria dos grafos e seus algoritmos. Santo André. 2014. Disponível em: <http://cbie2014.ufgd.edu.br/wp-content/uploads/2014/09/131483.pdf>. Acesso em: 29 ago. 2015.

MALTA, Gláucia H. S. **Grafos no ensino médio**: Uma inserção possível. Porto Alegre. 2008. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/14829/000668628.pdf>. Acesso em: 14 nov. 2015.

MARIANI, Antônio C. **Busca em grafos**. Florianópolis. 2015. Disponível em: <http://www.inf.ufsc.br/grafos/represen/busca.html>. Acesso em: 22 set. 2015.

MIYAZAWA, Flávio K. **Busca em grafos**. Campinas. 2012. Disponível em: < http://www.ic.unicamp.br/~fkm/disciplinas/mc448/2012s1/slides/aula17>. Acesso em: 23 set. 2015.

PRESTES, Edson. **Teoria dos grafos**. Porto Alegre. 2011a. Disponível em: < http://www.inf.ufrgs.br/~prestes/Courses/Graph%20Theory/GrafosA6.pdf>. Acesso em: 04 jun. 2016.

_____. **Teoria dos grafos**. Porto Alegre. 2013b. Disponível em: < http://www.inf.ufrgs.br/~prestes/Slides/aula1-Renan.pdf >. Acesso em: 04 jun. 2016.

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

SANGIORGI, U. B. **Rox uma ferramenta para o auxílio no aprendizado de teoria dos grafos**. Salvador. 2004. Disponível em: < http://www.uefs.br/erbase2004/documentos/weibase/Weibase2004Artigo006.pdf >. Acesso em: 29 ago. 2015.

SANTOS, Rodrigo P; COSTA, Heitor A. X. **TBC-GRAFOS/WEB – treinamento baseado em computador para algoritmos em grafos via web**. Rio de Janeiro. 2007. Disponível em: < http://www.cos.ufrj.br/~rps/pub/completos/2007/ICECE.pdf >. Acesso em: 29 ago. 2015.

SOARES, Tays C. A. et al. **Uma proposta metodológica para o aprendizado de algoritmos em grafos via animação não-intrusiva de algoritmos**. Belo Horizonte. 2004. Disponível em: < http://www.italost.com/academic/papers/pdf/ring_weimig2004.pdf >. Acesso em: 29 ago. 2015.

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

APÊNDICE A – Detalhamento sobre os casos de uso

A seguir é apresentado um detalhamento dos casos de uso, com descrição dos cenários. O caso de uso `UC006 – Habilitar o modo de Debug` descreve a interação entre o usuário e a funcionalidade que permite utilizar a ferramenta como meio de ensino para os algoritmos BFS, DFS e Dijkstra. Detalhes sobre este caso de uso estão descritos no Quadro 25.

Quadro 25 – Caso de uso `UC006 – Habilitar modo de Debug`

Número	006
Caso de uso	Habilitar modo de Debug
Descrição	Este caso de uso permite a utilização da ferramenta Furb Graphs como um meio de ensino para os algoritmos de Dijkstra, BFS e DFS.
Ator	Usuário
Cenário principal	1 – O usuário abre a aplicação. 2 – O usuário utiliza o mouse para clicar no botão Debug. 3 – O usuário utiliza o mouse para clicar em OK ou a tecla *enter* para confirmar a mensagem de ativação do modo de debug.

O caso de uso `UC007 – Navegar para frente no algoritmo` descreve a interação entre o usuário e a funcionalidade que permite avançar passos detalhados dos algoritmos de Dijkstra, BFS e DFS executados em um grafo. Detalhes sobre este caso de uso estão descritos no Quadro 26.

Quadro 26 – Caso de uso `UC007 – Navegar para frente no algoritmo`

Número	007
Caso de uso	Navegar para frente no algoritmo
Descrição	Este caso de uso permite a navegação passo a passo nos algoritmos de Dijkstra, BFS e DFS que forem executados em um grafo já desenhado.
Ator	Usuário

Pré-condições	`UC001 – Desenhar grafo`
	`UC006 – Habilitar o modo de Debug`
	`UC004 – Executar algoritmo`

Cenário principal	1 – O usuário utiliza o mouse para clicar no botão indicado pelo símbolo > na barra inferior no lado esquerdo da tela. 2 – A aplicação representa graficamente no grafo executado a coloração em vértices e arestas referente ao passo avançado no algoritmo. 3 – A aplicação apresenta o log detalhado do algoritmo em um campo de texto no lado superior direito. 4 – A aplicação discrimina a linha de código executada referente ao passo avançado do algoritmo no campo de texto no lado inferior direito.

O caso de uso `UC008 – Navegar para trás no algoritmo` descreve a interação entre o usuário e a funcionalidade que permite retroceder passos dos algoritmos de Dijkstra, BFS e DFS executados em um grafo. Detalhes sobre este caso de uso estão descritos no Quadro 27.

Quadro 27 – Caso de uso UC008 – Navegar para trás no algoritmo

Número	008
Caso de uso	Navegar para trás no algoritmo
Descrição	Este caso de uso permite a navegação retrocedendo passos nos algoritmos de BFS e DFS que forem executados em um grafo já desenhado.
Ator	Usuário
Pré-condições	UC001 – Desenhar grafo UC006 – Habilitar o modo de Debug UC004 – Executar algoritmo
Cenário principal	1 – O usuário utiliza o mouse para clicar no botão indicado pelo símbolo < na barra inferior no lado esquerdo da tela. 2 – A aplicação remove graficamente no grafo executado a coloração em vértices e arestas referente ao passo retrocedido no algoritmo. 3 – A aplicação retorna os estados e atributos dos vértices e arestas do grafo para o passo retrocedido.

Por fim o caso de uso UC009 – Retroceder ao passo inicial do algoritmo descreve a interação entre o usuário e a funcionalidade que permite retroceder todos os passos, voltando ao início dos algoritmos de Dijkstra, BFS e DFS executados em um grafo. Detalhes sobre este caso de uso estão descritos no Quadro 28.

Quadro 28 – Caso de uso UC009 – Retroceder ao passo inicial do algoritmo

Número	009
Caso de uso	Retroceder ao passo inicial do algoritmo
Descrição	Este caso de uso permite a navegação retrocedendo todos os passos, voltando ao início nos algoritmos de Dijkstra, BFS e DFS que forem executados em um grafo já desenhado.
Ator	Usuário
Pré-condições	UC001 – Desenhar grafo UC006 – Habilitar o modo de Debug UC004 – Executar algoritmo
Cenário principal	1 – O usuário utiliza o mouse para clicar no botão indicado pelo símbolo << na barra inferior no lado esquerdo da tela. 2 – A aplicação remove graficamente no grafo executado as colorações de todos os vértices e arestas. 3 – A aplicação reseta todas os estados e atributos dos vértices e arestas do grafo.

APÊNDICE B – Roteiro e questionário de perfil de usuário e usabilidade

Neste apêndice constam os questionários de perfil de usuário, questionário de tarefas que foram executadas pelos usuários e questionários de usabilidade e compreensão da ferramenta. No Quadro 29 pode-se visualizar o questionário de perfil. O Quadro 30 apresenta todas as tarefas que foram executadas e o Quadro 31 as avaliações de usabilidade e compreensão que foram respondidas pelos usuários.

Quadro 29 - Questionário de perfil de usuário

PERFIL DE USUÁRIO
Observação: as informações recebidas abaixo serão mantidas de forma confidencial.

Sexo: ( ) Masculino ( ) Feminino

Atua na área de computação?
( ) Sim
( ) Não - Qual área? _____

Idade:	
( ) Tenho menos de 5 anos	( ) Tenho entre 20 e 25 anos
( ) Tenho entre 5 e 10 anos	( ) Tenho entre 25 e 30 anos
( ) Tenho entre 10 e 15 anos	( ) Tenho entre 30 e 35 anos
( ) Tenho entre 15 e 20 anos	( ) Tenho mais de 35 anos

Nível de escolaridade
( ) Ensino fundamental incompleto
( ) Ensino fundamental completo – 1º grau
( ) Ensino médio incompleto
( ) Ensino médio completo – 2 grau
( ) Ensino superior incompleto
( ) Ensino superior completo
Observação:

Você compreende o que é um grafo?
( ) Sim
( ) Não

Você já estudou alguma disciplina que envolvesse teoria dos grafos?
( ) Sim
( ) Não

Quadro 30 - Questionário de tarefas a serem realizadas pelo usuário

1. Carregue ou desenhe um novo grafo com 6 vértices e 9 arestas onde todos os vértices estejam interligados.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

2. Se o grafo foi carregado, pule para a questão 4. Caso contrário, utilize o mouse clicando para criar vértices na tela, selecione cada vértice e atribua um nome a cada um.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

3. Selecione cada par de vértice, clique com o botão direito e selecione conectar. Selecione cada par de vértice que possuir uma aresta e atribua um valor numérico aleatoriamente para cada.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

4. Utilize o mouse para clicar no botão "debug" no menu de ações da ferramenta e confirme a ativação do modo de debug.

A tarefa foi executada? ( ) Sim ( ) Não
Observação:

5. Utilize o mouse no menu "Algoritmos" e selecione a opção Busca Largura. Siga as instruções apresentadas na tela.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

6. Utilize o mouse no botão de navegação para avançar cada passo do algoritmo. Analisando a coloração dos vértices e arestas em relação com o log de execução e o pseudocódigo em cada passo.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

7. Utilize o mouse nos botões de navegação para retroceder um passo ou para retroceder todos os passos. Após avance e retroceda os passos conforme desejar.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

8. Avance todos os passos do algoritmo e, utilizando o mouse para arrastar os vértices, organize-os se orientando através dos níveis indicados nas arestas, nível 1, nível 2.. seguindo a ordem.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

9. Utilize o mouse no menu "Algoritmos" e selecione a opção Busca Profundidade. Siga as instruções apresentadas na tela.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

10. Utilize o mouse no botão de navegação para avançar cada passo do algoritmo. Analisando a coloração dos vértices e arestas em relação com o log de execução e o pseudocódigo em cada passo.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

11. Utilize o mouse nos botões de navegação para retroceder um passo ou para retroceder todos os passos. Após avance e retroceda os passos conforme desejar.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

12. Utilize o mouse no menu "Algoritmos" e selecione a opção Dijkstra. Siga as instruções apresentadas na tela.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

13. Utilize o mouse para avançar os passos do algoritmo através do botão de navegação indicado na tela, verifique as legendas, logs e o pseudocódigo a cada passo.
A tarefa foi executada? ( ) Sim ( ) Não
Observação:

Quadro 31 - Questionário de usabilidade e compreensão aplicado ao usuário

Perguntas / Critérios de avaliação	**Concordo totalmente**	**Concordo parcialmente**	**Foi imparcial**	**Discordo totalmente**
De modo geral, você acredita que a ferramenta foi intuitiva e de fácil utilização?				
Para você, a interface gráfica e a disposição dos campos facilitaram a utilização da ferramenta?				
Você consegue se lembrar facilmente de como fazer as operações na ferramenta?				
Na sua opinião, as cores utilizadas na representação gráfica contribuíram para a compreensão dos algoritmos?				
Você achou importante a utilização das legendas na ferramenta?				
O log de execução foi claro e detalhado o suficiente para acompanhar os passos dos algoritmos?				
O pseudocódigo ajudou para a compreensão dos algoritmos?				
O modelo de navegação através dos botões foi prático e interativo?				
Você teve dificuldades para interpretar cada passo avançado nos algoritmos?				
A funcionalidade de retroceder um passo do algoritmo se fez útil para compreensão?				
A funcionalidade de retroceder todos os passos do algoritmo se fez útil para compreensão?				
A combinação das legendas, logs, pseudocódigo e representação gráfica juntos fizeram sentido para você?				
A compreensão geral dos algoritmos melhora depois de utilizar a ferramenta?				
Você precisaria de ajuda para operar a ferramenta?				
Para o estudo de grafos, seria útil utilizar esta ferramenta?				
A ferramenta em algum momento apresentou algum comportamento inesperado?				
Seria adequado recomendar esta ferramenta para outras pessoas que estudam grafos?				
No algoritmo de busca em largura, você compreendeu o vetor de roteamento em relação com a árvore gerada?				
No algoritmo de busca em profundidade, o empilhamento e desempilhamento do algoritmo de busca em profundidade ficou visível e compreensível?				
No algoritmo de Dijkstra, as comparações dos caminhos foram compreendidas?				
A representação gráfica do passo a passo contribui para o interesse em grafos?				

Sugestões?

_____

_____