**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO**

# FERRAMENTA PARA REPRESENTAÇÃO GRÁFICA DO FUNCIONAMENTO DE ALGORITMOS APLICADOS EM GRAFOS

**RODRIGO HACKBARTH**

**BLUMENAU**
**2008**

**2008/1-32**

**RODRIGO HACKBARTH**

# FERRAMENTA PARA REPRESENTAÇÃO GRÁFICA DO FUNCIONAMENTO DE ALGORITMOS APLICADOS EM GRAFOS

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

Prof. Paulo Cesar Rodacki Gomes - Orientador

**BLUMENAU**
**2008**

**2008/1-32**

# FERRAMENTA PARA REPRESENTAÇÃO GRÁFICA DO FUNCIONAMENTO DE ALGORITMOS APLICADOS EM GRAFOS

Por

**RODRIGO HACKBARTH**

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

| | |
|---|---|
| Presidente: | Prof. Paulo Cesar Rodacki Gomes, Dr. – Orientador, FURB |
| Membro: | Prof. Antônio Carlos Tavares, MSc. – FURB |
| Membro: | Prof. Dalton Solano dos Reis, MSc. – FURB |

Blumenau, 10 de julho de 2008

## AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, amigos e namorada pelo apoio e cobrança quando necessários e por acreditarem na conclusão deste trabalho.

Ao Germano Fronza pela ajuda durante uma dificuldade encontrada no desenvolvimento, apresentando-me o padrão *Command*.

Ao meu orientador, Paulo Cesar Rodacki Gomes, por ter me ajudado com idéias, fontes de pesquisa e com a biblioteca GraphObj.

# RESUMO

Este trabalho tem por objetivo o desenvolvimento de uma ferramenta que permita ao usuário a criação de um grafo de forma interativa. O usuário deve poder visualizar a execução de algoritmos encontrados na teoria dos grafos e seus resultados. A ferramenta é desenvolvida com orientação a objetos e especificada utilizando a Unified Modeling Language (UML). É implementada fazendo uso do padrão de projetos conhecido como Model-View-Controller (MVC) em conjunto com o padrão *Command*. A criação de sua interface utiliza as bibliotecas IUP, em conjunto com a linguagem de especificação de diálogos (LED), e OpenGL. Sua camada lógica utiliza a biblioteca GraphObj.

Palavras-chave: Teoria dos grafos. Padrão de projetos. IUP. LED. GraphObj.

# ABSTRACT

The present work aims the development of a tool that allows the user to create a graph in an interactive way. The user should be able to visualize the execution of the algorithms found in the graph theory and its results. The tool is developed using objects orientation and it's specified using the Unified Modeling Language (UML). It's implemented making use of the design pattern known as Model-View-Controller (MVC) along with the pattern Command. The creation of its interface uses the libraries IUP, along with the dialog specification language (LED), and OpenGL. Its logical layer uses the library GraphObj.

Key-words: Graph theory. Design patterns. IUP. LED. GraphObj.

# LISTA DE ILUSTRAÇÕES


Figura 1 – Grafo não dirigido e sua lista de adjacência ..... 12
Figura 2 – Exemplo de um dígrafo e sua lista de adjacência ..... 13
Figura 3 – Árvore de busca em largura ..... 14
Quadro1 - Pseudocódigo do algoritmo BFS ..... 14
Figura 4 – Árvore de busca em profundidade ..... 15
Quadro 2 - Pseudocódigo do algoritmo DFS ..... 16
Quadro 3 – Pseudocódigo do algoritmo de Kruskal ..... 17
Figura 5 – Árvore de custo mínimo gerada pelo algoritmo de Kruskal ..... 18
Figura 6 – Arquitetura do padrão MVC ..... 19
Figura 7 - Diagrama de classes da biblioteca GraphObj ..... 21
Quadro 4 - Exemplo de código em LED ..... 23
Figura 8 - Diálogo feito em linguagem LED ..... 23
Figura 9 – Aplicação desenvolvida utilizando a JGraph ..... 26
Figura 10 – Diagrama de casos de uso da ferramenta ..... 28
Figura 11 – Diagrama de seqüência para criação de vértices ..... 29
Figura 12 – Diagrama de seqüência para criação de arestas ..... 30
Figura 13 – Diagrama de seqüência para execução do algoritmo BFS ..... 31
Figura 14 – Diagrama de classes da biblioteca GraphObj ..... 32
Figura 15 – Diagrama de classes da ferramenta ..... 34
Figura 16 – Diagrama de classes da integração entre GraphObj e a ferramenta ..... 36
Quadro 5 – Método `removeVertexInterf` ..... 37
Quadro 6 – Classe abstrata `MyListener` ..... 37
Quadro 7 – Classe `MyListenerImpl` ..... 38
Quadro 8 – Método `createVertex` ..... 39
Quadro 9 – Método `createVertexInterf` ..... 39
Quadro 10 – Método `createEdge` da classe `Graph` ..... 40
Quadro 11- Construtor da classe `UndirectedEdge` ..... 40
Quadro 12 – Método `createEdge` da classe `Digraph` ..... 40
Quadro 13 – Construtor da classe `DirectedEdge` ..... 41
Quadro 14 – Método `createEdgeInterf` ..... 41

Quadro 15 – Construtor da classe `EdgeInterf` ..... 41
Quadro 15 – Construtor da classe `AlgBFS` ..... 42
Quadro 16 – Método `run` da classe `AlgBFS` ..... 43
Quadro 17 – Método `BFSVisit` ..... 44
Quadro 18 – Método `execute` implementado na classe `MyListenerImpl` ..... 45
Quadro 19 – Método `DFS` da classe `AlgDFS` ..... 46
Quadro 20 – Método `DFSVisit` ..... 46
Quadro 21 – Método `run` da classe `AlgMSTKruskal` ..... 47
Quadro 22 – Método `setKruskalTreeColor` ..... 48
Figura 17 – Tela inicial do programa ..... 49
Figura 18 – Função de criação de vértices ..... 50
Figura 19 – Função de criação de arestas e definição de novo custo ..... 51
Figura 20 – Aresta selecionada ..... 52
Figura 21 – Aresta excluída ..... 52
Figura 22 – Seleção de um vértice ..... 53
Figura 23 – Vértice excluído ..... 53
Figura 24 – Lista de algoritmos expandida ..... 54
Figura 25 – Resultado da busca BFS ..... 55
Figura 26 – Resultado da busca DFS ..... 55
Figura 27 – Resultado da execução do algoritmo de Kruskal ..... 56
Quadro 23 - Diferenças entre os trabalhos correlatos e a ferramenta desenvolvida ..... 57

# SUMÁRIO

**1 INTRODUÇÃO ..... 10**
1.1 OBJETIVOS DO TRABALHO ..... 10
1.2 ESTRUTURA DO TRABALHO ..... 11
**2 FUNDAMENTAÇÃO TEÓRICA ..... 12**
2.1 GRAFOS ..... 12
2.1.1 Breadth First Search ..... 14
2.1.2 Depth First Search ..... 15
2.1.3 Kruskal ..... 16
2.2 PADRÕES DE PROJETO ..... 18
2.2.1 Command ..... 19
2.2.2 MVC ..... 19
2.3 GRAPHOBJ ..... 20
2.4 IUP/LED ..... 22
2.4.1 LED ..... 22
2.5 TRABALHOS CORRELATOS ..... 24
2.5.1 BGL ..... 24
2.5.2 VGTL ..... 24
2.5.3 GTL ..... 25
2.5.4 JGRAPH ..... 25
**3 DESENVOLVIMENTO ..... 27**
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO ..... 27
3.2 ESPECIFICAÇÃO ..... 27
3.2.1 Diagrama de casos de uso ..... 28
3.2.2 Diagrama de seqüência ..... 29
3.2.3 Diagrama de classes ..... 32
3.3 IMPLEMENTAÇÃO ..... 37
3.3.1 Técnicas e ferramentas utilizadas ..... 37
3.3.2 Desenvolvimento da ferramenta ..... 38
3.3.2.1 Desenvolvimento do grafo ..... 38
3.3.2.2 Desenvolvimento dos algoritmos ..... 41
3.3.3 Operacionalidade da implementação ..... 48

3.4 RESULTADOS E DISCUSSÃO ..... 56
**4 CONCLUSÕES ..... 59**
4.1 EXTENSÕES ..... 59
**REFERÊNCIAS BIBLIOGRÁFICAS ..... 61**

# 1 INTRODUÇÃO

Há muitos anos, problemas encontrados no cotidiano vêm sendo resolvidos através de sua modelagem para a forma de grafos. Ao serem transformados em grafos, estes mostram-se mais fáceis de ser entendidos e, portanto, resolvidos. Com o surgimento da informática, as etapas para a resolução destes problemas foram modeladas em algoritmos.

Algoritmos aplicados em grafos são formas detalhadas, passo a passo, da resolução de determinadas situações. Eles são estudados até hoje, sendo que um algoritmo pode ser usado para resolver diversos problemas, dependendo da semelhança que possuem com o problema que originou o mesmo. Quando estão sendo estudados ou visados para alguma situação específica, faz-se necessária a realização de testes de mesa verificando seus resultados, a fim de compreender melhor o funcionamento do algoritmo ou garantir a melhor escolha para resolver os problemas apresentados.

Este trabalho consiste na criação de uma interface gráfica, através da qual o usuário pode criar seu grafo, verificar o resultado dos algoritmos nele aplicados, assim como o caminho percorrido no grafo. Para a camada lógica do programa foi utilizada a biblioteca GraphObj, a qual foi desenvolvida no Departamento de Sistemas e Computação (DSC) da Universidade Regional de Blumenau (FURB) pelo professor Paulo Cesar Rodacki Gomes. Esta possui um conjunto de vários algoritmos voltados a grafos nela implementados, os quais são indispensáveis para o funcionamento da ferramenta.

Por fim, foi utilizado o padrão de projeto para softwares orientados a objeto (*design patterns*) conhecido como *Model-View-Controller* (MVC). Este padrão define a divisão da arquitetura do programa. O *model* contém a parte lógica, onde está a biblioteca GraphObj. A *view* consiste na interface da ferramenta, enquanto o *controller* fica encarregado de ligar a interface com a camada lógica realizando as mudanças necessárias.

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é implementar uma ferramenta para representar graficamente estruturas de grafos. Os objetivos específicos são:

a) criar uma interface gráfica para a representação dos grafos;
b) permitir ao usuário a criação interativa de seu próprio grafo;
c) visualizar graficamente o funcionamento de algoritmos aplicados a grafos.

## 1.2 ESTRUTURA DO TRABALHO

O capítulo 1 apresenta uma introdução aos assuntos abordados neste trabalho definindo os objetivos a serem alcançados e a estrutura deste relatório.

No capítulo 2 encontra-se a fundamentação teórica utilizada para a realização deste trabalho. Primeiramente é descrita uma visão geral da teoria dos grafos. Em seguida é realizada uma explicação sobre padrões de projetos com ênfase nos modelos MVC e *Command*. Por fim são discutidos alguns aspectos da biblioteca GraphObj e apresentados os trabalhos correlatos.

O capítulo 3 descreve, a princípio, os requisitos que a ferramenta deve obedecer. A seguir explica a especificação da ferramenta através do uso de diagramas de caso de uso, seqüência e de classes. Aborda o desenvolvimento da aplicação apresentando trechos de seu código fonte e apresenta a forma com que o usuário pode interagir com a ferramenta. Por fim, os resultados obtidos são apresentados e confrontados com os trabalhos correlatos.

Para finalizar, o capítulo 4 apresenta as conclusões obtidas com o desenvolvimento da ferramenta, além de sugerir complementos a serem realizados para dar continuidade ao presente trabalho.

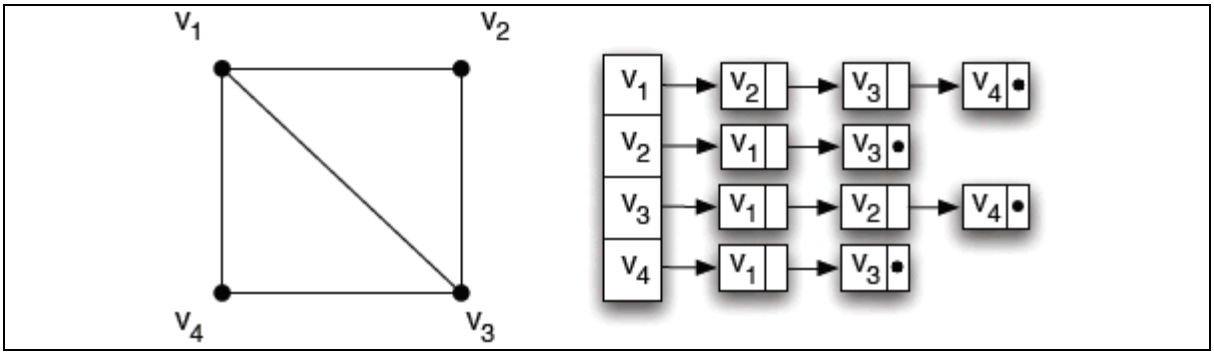
# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo explica as teorias e ferramentas utilizadas no desenvolvimento deste trabalho. A seção 2.1 explica a teoria dos grafos, certos conceitos encontrados na teoria e os algoritmos *Breadth First Search* (BFS), *Depth First Search* (DFS), e Kruskal. A seção 2.2 aborda padrões de projeto explicando o modelo MVC e o padrão *Command*. A seção 2.3 discute alguns aspectos da biblioteca GraphObj. A seção 2.4 explica o sistema de interface IUP/LED e a seção 2.5 comenta sobre os trabalhos correlatos encontrados.

## 2.1 GRAFOS

Segundo Gomes (2007, p. 12), um grafo é definido por um conjunto não vazio de vértices e um conjunto de arestas, onde cada aresta é definida por um par não-ordenado de vértices pertencentes ao conjunto de vértices do grafo. No entanto, para fins didáticos, os grafos são representados graficamente por uma estrutura composta de pontos e linhas, os quais representam os vértices e as arestas respectivamente.

Uma forma de armazenamento das estruturas de grafos é através do uso de listas de adjacência. A estrutura de listas de adjacência define uma lista para cada vértice do grafo de modo que, a lista de cada vértice `vi` armazena os vértices `vj` que possuem aresta com `vi`. A figura 1 apresenta a estrutura de um grafo e sua lista de adjacência.

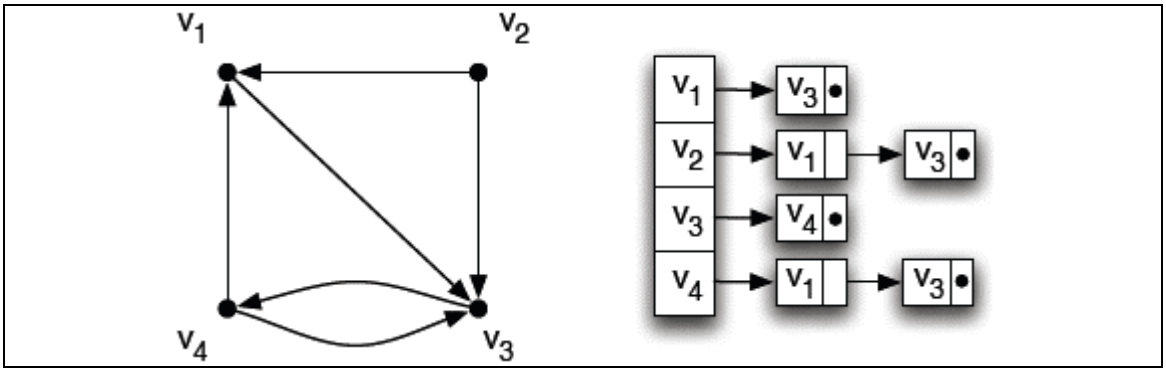



Fonte: Gomes (2007, p. 30).

Figura 1 – Grafo não dirigido e sua lista de adjacência

Além do grafo, existe o conceito de dígrafo. A diferença entre um grafo e um dígrafo está no fato de que as arestas de um grafo são simplesmente formadas por dois vértices, enquanto no dígrafo as arestas possuem um vértice de origem e um vértice de destino,

resultando em uma aresta dirigida, ou seja, com um sentido. Isto tem influência direta na aplicação de certos algoritmos. Na execução de algoritmos em grafos, as arestas são percorridas e os vértices visitados de modo que, quando as arestas forem dirigidas, o algoritmo terá que obedecer aos sentidos apontados pelas mesmas para percorrer o grafo. Quando representados em listas de adjacência, cada aresta `(vi, vj)` em um dígrafo resulta apenas na adição do vértice `vj` na lista de `vi`. No entanto em um grafo, cada aresta `(vi,vj)` causa a adição de `vj` na lista de `vi` e vice-versa. A representação de um dígrafo é apresentada na figura 2 juntamente com sua lista de adjacência.




Fonte: Gomes (2007, p. 30).

Figura 2 – Exemplo de um dígrafo e sua lista de adjacência

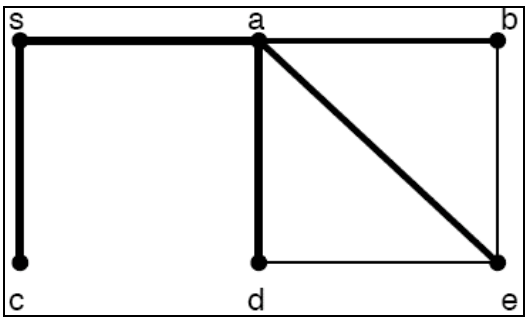
A definição de árvore também é importante na teoria dos grafos. Uma árvore ainda é um grafo, no entanto esse precisa possuir certas características. Uma árvore é um grafo que possui todos seus vértices ligados por arestas e não possui ciclos. Sendo assim, qualquer árvore composta por "n" vértices terá exatamente "n-1" arestas.

Outro conceito de grande influência na teoria dos grafos é o do custo associado às arestas. O custo nada mais é que um valor associado a cada aresta do grafo. No entanto, mesmo que sua definição seja tão simples, o custo é imprescindível para a resolução de certos problemas como, por exemplo, problemas que necessitem da descoberta do menor caminho existente entre dois vértices de um grafo. Um exemplo de problema de custo mínimo é explicado por Chartrand (1985, p. 79-80), o qual envolve um conjunto de cidades entre as quais deseja-se criar um sistema de estradas para conectar todas elas. No entanto, por causa da atual situação econômica, deve-se analisar qual a forma mais barata de construção, independente de quão inconveniente isso seja para os passageiros. Neste caso, faz-se necessária a atribuição de custos às arestas, representando quanto custará a construção de cada caminho.

Dentre os algoritmos aplicados a grafos encontram-se o BFS, DFS e Kruskal, os quais são explicados nas seções a seguir.

### 2.1.1 Breadth First Search

O algoritmo BFS, também conhecido como busca em largura, é uma busca que começa em um vértice inicial e, a cada iteração, visita todos os vértices a uma mesma distância do vértice inicial em quantidade de arestas percorridas. Através do seu modo de caminhamento no grafo, este algoritmo encontra o menor caminho, em arestas percorridas, entre os vértices do grafo. A seqüência de arestas percorridas gera uma estrutura de árvore que pode ser armazenada ao longo processamento do algoritmo, chamada de árvore de busca em largura. Um exemplo dessa árvore é apresentado na figura 3.




Fonte: Gomes (2007, p. 37).

Figura 3 – Árvore de busca em largura

O algoritmo BFS é apresentado no Quadro 1.

```
BFS(G,s)
1  for cada vértice u∈V[G]−{s}
2     do cor[u] ← BRANCO
3        d[u] ← ∞
4        α[u] ← NIL
5  cor[s] ← CINZA
6  d[s] ← 0
7  α[s] ← NIL
8  Q ← ∅
9  ENQUEUE(Q,s)
10 while Q ≠ ∅
11    do u ← DEQUEUE(Q)
12       for cada v ← Adj[u]
13          do if cor[v] = BRANCO
14             then cor[v] ← CINZA
15                  d[v] ← d[u]+1
16                  π[v] ← u π[v] ← u
17                  ENQUEUE(Q,v)
18       cor[u] ← PRETO
```

Fonte: Cormen et al. (2002, p. 423).

Quadro1 - Pseudocódigo do algoritmo BFS

O algoritmo recebe o grafo no qual o caminhamento será realizado e um vértice raiz, a partir do qual a busca será iniciada. Nas linhas 1 a 4 todos os vértices são inicializados com cor branca, distância até o vértice inicial igual a "∞" e vértice predecessor igual a `NIL`. As linhas 5 a 7 inicializam o vértice raiz com cor cinza, distância do vértice inicial equivalente a 0, pois esse é o vértice inicial, e predecessor `NIL`. Nas linhas 8 e 9, a fila `Q`, que servirá para gerenciar o conjunto de vértices de cor cinza, é inicializada contendo apenas o vértice raiz. A seguir, enquanto houver vértices na fila `Q`, é removido o primeiro vértice da fila e associado à variável `u`. Então para cada vértice adjacente a `u`, se sua cor for branca, o vértice terá sua cor alterada para cinza, distância igual à distância do vértice inicial até o vértice `u` + 1 e a variável que guarda seu predecessor recebe `u`. O vértice é então adicionado à fila `Q`. Quando todos os vértices adjacentes a `u` tiverem sido examinados, `u` receberá a cor preta (CORMEN et al., 2002, p. 423-424).

### 2.1.2 Depth First Search

A busca DFS, conhecida como busca em profundidade, assim como a BFS, percorre todos os vértices do grafo, no entanto o faz de forma diferente. Segundo Gomes (2007, p. 38), este algoritmo percorre o grafo de forma recursiva sendo que, ao invés de visitar todos a uma mesma distância, ao atingir um vértice no grafo procura seu vizinho, depois o vizinho de seu vizinho e assim por diante. Quando não consegue ir adiante, ela volta gradativamente aos passos anteriores procurando uma nova opção de caminho. Este processo de voltar atrás é conhecido como *backtracking*. A partir desta busca também é gerada uma árvore, como acontece na BFS, no entanto não é possível gerar o caminho mínimo. A figura 4 apresenta um exemplo de árvore de busca gerada pela pelo algoritmo DFS.

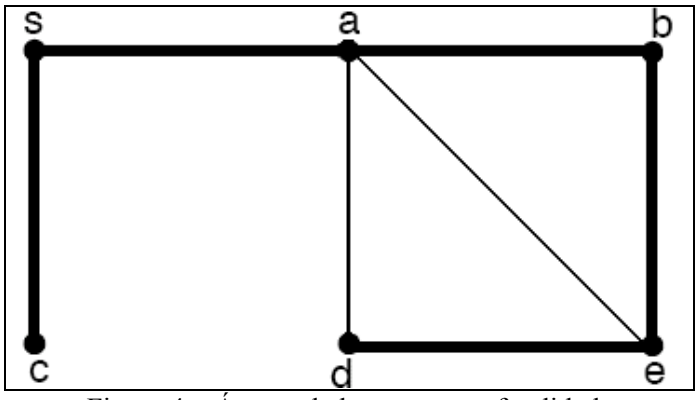



Figura 4 – Árvore de busca em profundidade

O quadro 2 mostra o pseudocódigo do algoritmo DFS.

```
DFS(G)
1 for cada vértice u ← V[G]
2   do cor[u] ← BRANCO
3      π[u] ← NIL
4 tempo ← 0
5 for cada vértice u ∈ V[G]
6   do if cor[u] = BRANCO
7      then DFS – VISIT(u)
```

```
DFS – VISIT(u)
1 cor[u] ← CINZA
2 tempo ← tempo + 1
3 d[u] ← tempo
4 for cada v ∈ Adj[u]
5   do if cor[u] = BRANCO
6      then π[v] ← u
7           DFS – VISIT(v)
8 cor[u] ← PRETO
9 f[u] ← tempo ← tempo + 1
```

Fonte: Cormen et al. (2002, p. 430).

Quadro 2 - Pseudocódigo do algoritmo DFS

Inicialmente o algoritmo recebe, por parâmetro, o grafo no qual a busca deverá ser executada. Nas linhas 1 a 3 é realizada a inicialização dos vértices com cor branca e predecessor nulo. A variável `tempo` é então inicializada com 0. As linhas 5 a 7 fazem para cada vértice do grafo, caso sua cor seja branca, a chamada do método `DFS-VISIT()` passando o vértice atual `u` por parâmetro. O método `DFS-VISIT(u)` inicia alterando a cor do vértice recebido por parâmetro para cinza, a variável `tempo` é incrementada e a distância entre esse vértice e o vértice de origem recebe o valor de `tempo`. Nas linhas 4 a 7, para cada vértice `v` adjacente ao vértice atual `u`, caso `v` seja branco, o vértice `u` torna-se seu predecessor e o método `DFS-VISIT()` é chamado recursivamente passando `v` por parâmetro. Por fim, nas linhas 8 e 9, o vértice `u` recebe a cor preta e o tempo de término é incrementado e registrado em `f[u]` (CORMEN et al., 2002, p. 431).

### 2.1.3 Kruskal

Segundo Cormen et al. (2002, p. 450-452), o algoritmo de Kruskal serve para encontrar a árvore de amplitude mínima, também chamada de árvore geradora de custo mínimo. É considerado um algoritmo guloso, pois a cada passo de sua execução uma aresta com o menor custo possível é incluída à árvore. Esse algoritmo procura sempre adicionar a aresta de menor custo sem importar-se em manter a conexidade com uma única árvore. Dessa forma, várias árvores surgem no grafo durante a execução do algoritmo e irão unir-se formando uma única árvore ao final da execução. O algoritmo de Kruskal é composto dos seguintes métodos:

a) `MAKE-SET(v)`, responsável pela criação de conjuntos, os quais possuem um

identificador e contém um vértice que é passado por parâmetro;

b) `FIND-SET(v)`, retorna o identificador do conjunto que contém o vértice passado ao método por parâmetro;

c) `UNION(u,v)`, realiza a união dos conjuntos que contém os vértices `u` e `v`, atribuindo um identificador ao novo conjunto.

O pseudocódigo desse algoritmo é apresentado no quadro 3.

```
MST – KRUSKAL(G, w)
1  A ← ∅
2  for cada vértice v∈V[G]
3    do MAKE – SET(v)
4  ordenar as arestas de E por peso w não decrescente
5  for cada aresta (u,v)∈E , em ordem de peso não decrescente
6    do if FIND – SET(u) ≠ FIND – SET(v)
7      then A ← A ∪ {(u,v)}
8           UNION(u,v)
9  Return A
```

Fonte: Cormen et al. (2002, p. 450 - 452).

Quadro 3 – Pseudocódigo do algoritmo de Kruskal

As linhas 1 a 3 inicializam o conjunto `A`, conjunto de arestas da árvore de amplitude mínima, como vazio e criam `|V|` árvores, cada uma contendo 1 vértice. Na linha 4 as arestas do grafo são ordenadas por peso de forma não decrescente. Nas linhas 5 a 8, para cada aresta do grafo, os vértices que a formam são verificados para descobrir se fazem parte da mesma árvore. Caso façam parte, a aresta não pode ser adicionada à floresta (conjunto de árvores), pois apareceria um ciclo. Caso não pertençam à mesma árvore, a aresta é adicionada ao conjunto `A` e os vértices nas duas árvores diferentes são intercalados. Por fim o conjunto de arestas `A` é retornado na linha 9.

A figura 5 apresenta um exemplo de árvore de custo mínimo gerada pela execução do algoritmo de Krukal sobre um grafo. As arestas em destaque fazem parte da árvore encontrada.

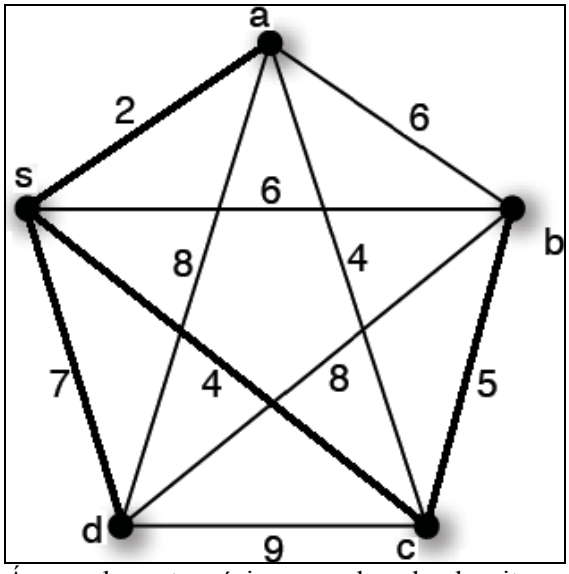


Figura 5 – Árvore de custo mínimo gerada pelo algoritmo de Kruskal

## 2.2 PADRÕES DE PROJETO

De acordo com Gamma et al. (1994, p. 3-4), um padrão de projeto ajuda o programador a escolher alternativas de projeto para desenvolver um sistema reusável e pode melhorar o processo de documentação e manutenção de sistemas já existentes. Um padrão possui quatro elementos principais:

a) o nome do padrão, que pode ser usado para descrever um problema, suas soluções e conseqüências em poucas palavras;
b) o problema, para explicar o problema e seu contexto indicando quando um padrão deve ser usado;
c) a solução, a qual descreve os elementos que geram o projeto, suas relações, responsabilidades e colaborações;
d) as conseqüências, que são os resultados da aplicação do padrão a um projeto e servem para melhor entender e avaliar um projeto.

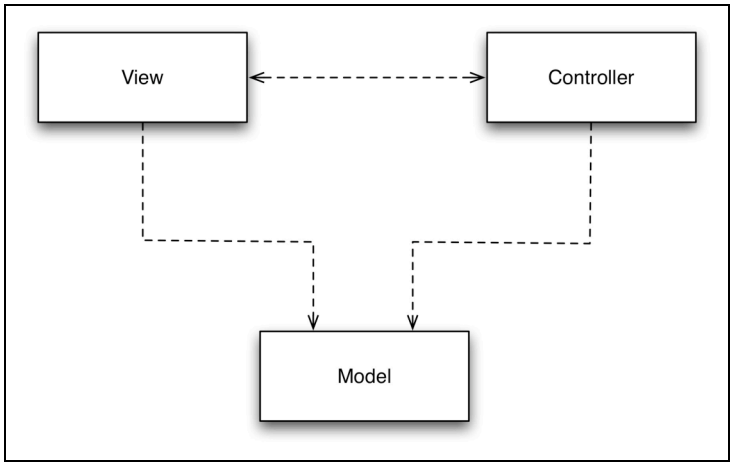
Dentre os padrões existentes encontram-se o padrão *Command* e o MVC que são explicados mais detalhadamente nas seções a seguir.

### 2.2.1 Command

Para tratar mensagens de determinados eventos em um sistema, uma abordagem encontrada entre os padrões de projetos é a utilização do padrão *Command*. Esse padrão é utilizado quando certas mensagens devem ser passadas a objetos, sem saber que operações são pedidas na mensagem ou até mesmo, quem receberá essa mensagem. O padrão *Command* sugere que uma classe deve ser definida para cada mensagem, sendo que cada uma dessas classes deve ter um método *execute*. O controlador deve então criar uma instância de *Command* correspondente à mensagem do evento e enviar uma mensagem *execute* para esta instância. Cada classe *Command* possui um único método *execute* com ações específicas para tratar a mensagem que deve receber. Desse modo, cada mensagem em um sistema pode ser tratada por um objeto *Command* separado (GAMMA et al., 1994, p. 233-235 ).

### 2.2.2 MVC

Segundo Gamma et al. (1994, p. 4), o MVC é um padrão de projeto que consiste de três tipos de objetos: o *Model*, que é o objeto de aplicação, a *View*, que é a tela de apresentação e o *Controller*, o qual define como a interface reage aos comandos do usuário. Em outras palavras, o *Model* consiste da camada lógica do programa, a *View* é a interface com o usuário e o *Controller* é encarregado de fazer a ligação entre estas duas camadas, realizando as modificações que mostrarem-se necessárias. A figura 6 apresenta a arquitetura do padrão MVC.




Fonte: Fowler (2003, p. 330).

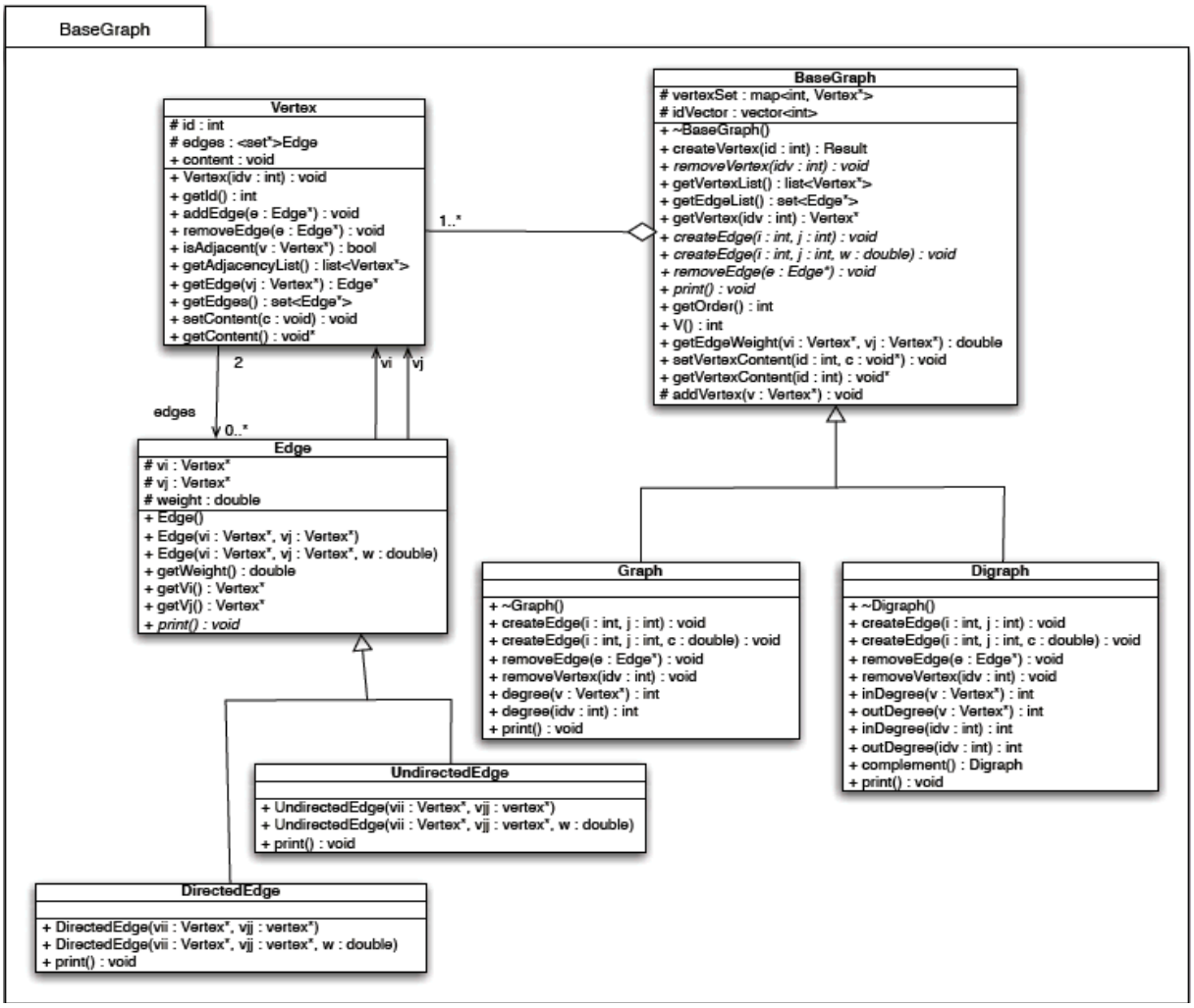
Figura 6 – Arquitetura do padrão MVC

Com relação ao acesso dessas camadas entre si, Pree (1994, p. 70-71) diz que a *View* e o *Controller* podem acessar o *Model*, podendo se informar sobre os dados guardados no *Model* e até mesmo modificá-los. Já o *Model* não tem acesso a nenhuma das outras duas camadas, pois esse não precisa se preocupar com o modo como será mostrado. Se o usuário realizar alguma ação pelo teclado ou pelo *mouse*, então o *Controller* mandará uma mensagem ao *Model* ou eventualmente direto para a *View*. O *Model* muda então os dados necessários encontrados nele e informa essa mudança às outras camadas. Este é o modelo conhecido como MVC, no entanto variações desse modelo foram surgindo com o passar dos tempos para adaptarem-se a cada aplicação sendo que certas aplicações, mesmo que desenvolvidas seguindo o modelo MVC, acabam por unir os componentes da *View* e do *Controller*.

## 2.3 GRAPHOBJ

Sobre a biblioteca GraphObj, Gomes (2007) diz que "Trata-se de uma biblioteca de programação para grafos implementada em duas versões, uma em linguagem Java e outra em Linguagem C++. A principal finalidade deste *framework* é didática [...]".

Esta biblioteca possui uma estrutura de classes que permite a criação de grafos e dígrafos, além de permitir a associação de custo às arestas. Utiliza-se das estruturas de dados disponibilizadas pela STL para organizar a estrutura interna do grafo, além de trazer consigo um conjunto de algoritmos voltados à aplicação em grafos. Este *framework* permite a utilização de seus métodos pelo programador para o desenvolvimento de uma ferramenta que crie grafos e aplique algoritmos sobre os mesmos. No entanto, por estar ainda em desenvolvimento, qualquer funcionalidade não encontrada na biblioteca que se faça necessária ao programador deve ser implementada pelo mesmo. A figura 7 mostra o modelo de classes da biblioteca disponibilizado pelo professor Paulo Cesar Rodacki Gomes.




Fonte: Gomes (2008).

Figura 7 - Diagrama de classes da biblioteca GraphObj

Conforme apresentado na figura 7, a biblioteca possui uma classe abstrata chamada BaseGraph que possui atributos e métodos comuns às classes Graph e Digraph, as quais permitem ao usuário a criação de grafos e dígrafos respectivamente. Assim como no caso do grafo, a classe Edge para criação de arestas também é abstrata, com características comuns às classes de arestas dirigidas, DirectedEdge, e não dirigidas, UndirectedEdge. Além disso, pode-se perceber que há uma relação de agregação, onde um grafo deve agregar pelo menos um vértice, e uma associação entre vértice e aresta, na qual um vértice não precisa necessariamente ter arestas associadas a ele, no entanto uma aresta é obrigatoriamente formada por dois vértices.

## 2.4 IUP/LED

IUP/LED é um sistema portátil de interface com o usuário. Foi projetado com o objetivo ser portátil e fácil de usar. Suas principais características são que ele permite a composição de diálogos através da descrição de um layout abstrato, tem uma linguagem de especificação de diálogos (LED) compilada em tempo de execução da aplicação, separa o sistema de interface do sistema gráfico, permite tanto *look-and-feel* nativo quanto *look-and-feel* fixo e é expansível.

Um diálogo, que pode ser composto pela IUP/LED, é uma área retangular da tela que contém outros elementos de interface e os gerencia na interação entre a aplicação e o usuário. Esse sistema permite a criação e manipulação de elementos da interface através de um pequeno conjunto de funções. A interação é feita principalmente através de duas funções IUP que servem para estabelecer e consultar atributos.

Cada elemento da interface tem associado a si um atributo cujo valor é usado para identificar o elemento. Assim a aplicação pode consultar este valor e utilizá-lo para a execução de funções do sistema de interface nativo.

Para a IUP, uma aplicação é formada por um conjunto de diálogos, sendo que cada diálogo é formado por componentes de interface que interagem com o usuário. Sendo assim, o desenvolvimento de uma aplicação consiste apenas em definir os diálogos da aplicação, o que pode ser feito utilizando-se LED, e implementar suas rotinas (LEVY, 1993, p. 7, 33-34).

### 2.4.1 LED

LED é uma linguagem de expressões que serve para especificar diálogos. Esta linguagem é compilada em tempo de execução, garantindo independência de diálogos, prototipação rápida e customização para diferentes tipos de usuários e plataformas. A independência de diálogos é alcançada pela utilização da linguagem LED, que foi projetada com este objetivo. A prototipação rápida é alcançada pelo fato de ser compilada em tempo de execução sem a necessidade de funções específicas da aplicação. Apenas precisa ser compilada através da chamada de uma função IUP localizada no programa principal. Por fim, a customização é realizada pelo usuário programando em linguagem LED, sendo possível modificar a especificação dos diálogos ou traduzir a interface para outro idioma.

Essa linguagem permite a construção de um modelo abstrato de interface, cujos diálogos são definidos por um *layout* abstrato e seus elementos, definidos pela sua funcionalidade sem a necessidade de especificar sua aparência. Dessa forma, o programador não precisa se preocupar com o sistema de interface sob o qual o programa irá executar (LEVY, 1993, p. 34-35).

Um exemplo de código em linguagem LED é apresentado no quadro 4.

```
1 confirm=dialog[TITLE="Attention"](body)
2 body=vbox(fill(),prompt,fill(),buttons)
3 prompt=hbox(fill(),warning,fill())
4 buttons=hbox(fill(),replace,fill(),cancel,fill())
5 warning=label("File already exists!")
6 replace=button("Replace",do_replace)
7 cancel=button("Cancel",do_cancel)
```

Fonte: Levy (1993, p. 36-37).

Quadro 4 - Exemplo de código em LED

No quadro 4 é possível visualizar alguns componentes da linguagem LED. Na linha 1 é criado o diálogo com um título e um corpo, no entanto esse corpo é definido nas linhas seguintes. Na linha 2, 3 e 4 são apresentados os componentes `hbox` e `vbox` que consistem de uma caixa horizontal e uma vertical respectivamente, dentro das quais outros componentes são adicionados. Na linha seguinte é criado apenas um texto que será apresentado ao usuário. Por fim são criados botões sendo que, o primeiro parâmetro equivale ao texto contido no botão, enquanto o segundo é a ação que será realizada ao clicar nos mesmos. A figura 8 apresenta o resultado gerado pelo código do quadro 4.

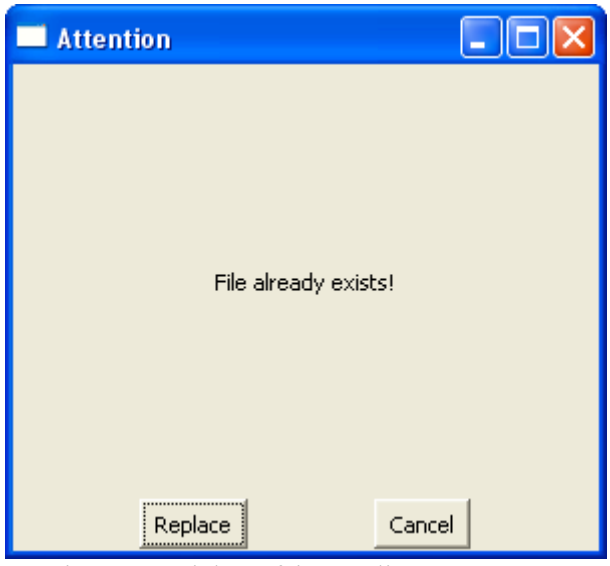



Figura 8 - Diálogo feito em linguagem LED

## 2.5 TRABALHOS CORRELATOS

Dentre os trabalhos encontrados que desempenham papéis semelhantes ao proposto neste trabalho, são descritos de maneira geral as bibliotecas *Boost Graph Library* (BGL), *Viena Graph Template Library* (VGTL), *Graph Template Library*(GTL) e JGraph.

### 2.5.1 BGL

A BGL é uma sub-biblioteca, voltada à aplicação em grafos, encontrada na biblioteca *Boost*. *Boost* traz consigo um conjunto de várias sub-bibliotecas, todas implementadas em C++, e tem como um de seus objetivos o fornecimento de implementações de referência para que suas bibliotecas estejam preparadas para uma eventual padronização. Segundo Siek, Lee e Lumsdaine (2008), parte da BGL é uma interface genérica que permite o acesso à estrutura do grafo, enquanto esconde os detalhes de implementação. Além disso, qualquer biblioteca de grafos que implemente esta interface irá tornar-se operável com os algoritmos genéricos da BGL e com outros que também utilizem esta interface. Esta biblioteca fornece como estruturas de dados a lista de adjacência, matriz de adjacência e uma lista de arestas. Também traz consigo os algoritmos de caminho mínimo de Dijkstra e Bell-Ford, os algoritmos para verificação de componentes conexos, ordenação, transposição, coloração e árvore geradora de custo mínimo.

### 2.5.2 VGTL

Segundo a Silicon Graphics (2006), a *Standard Template Library* (STL) é uma biblioteca genérica desenvolvida em C++, que possui um conjunto de classes, algoritmos e iteradores.

De acordo com Schichl (2003), a VGTL é uma biblioteca de grafos genérica, com uma estrutura genérica de programação. Utiliza-se dos *containers* da STL como mapas e vetores para organizar a estrutura interna do grafo. Nela foi implementado um conjunto de algoritmos

genéricos para a análise e trabalho com grafos. Similar aos iteradores STL, que são usados para manipular dados em *containers* independente da implementação dos mesmos, para o caminhamento em grafos é utilizado um iterador. Este iterador é a generalização de um ponteiro que faz referência aos dados armazenados por um vértice no grafo e pode ser dividido em dois tipos: recursivo e iterativo. O recursivo é um ponteiro para vértices, o qual pode ser movido pelo grafo simplesmente mudando o vértice para o qual aponta. O iterativo pode mover-se pelo grafo sem a necessidade de ser guiado, sendo que o iterador por si só procura o próximo vértice a percorrer. Esta biblioteca possui estruturas de dados que permitem a criação de grafos e dígrafos, sendo que os dígrafos dividem-se em dígrafo padrão e dígrafo acíclico.

### 2.5.3 GTL

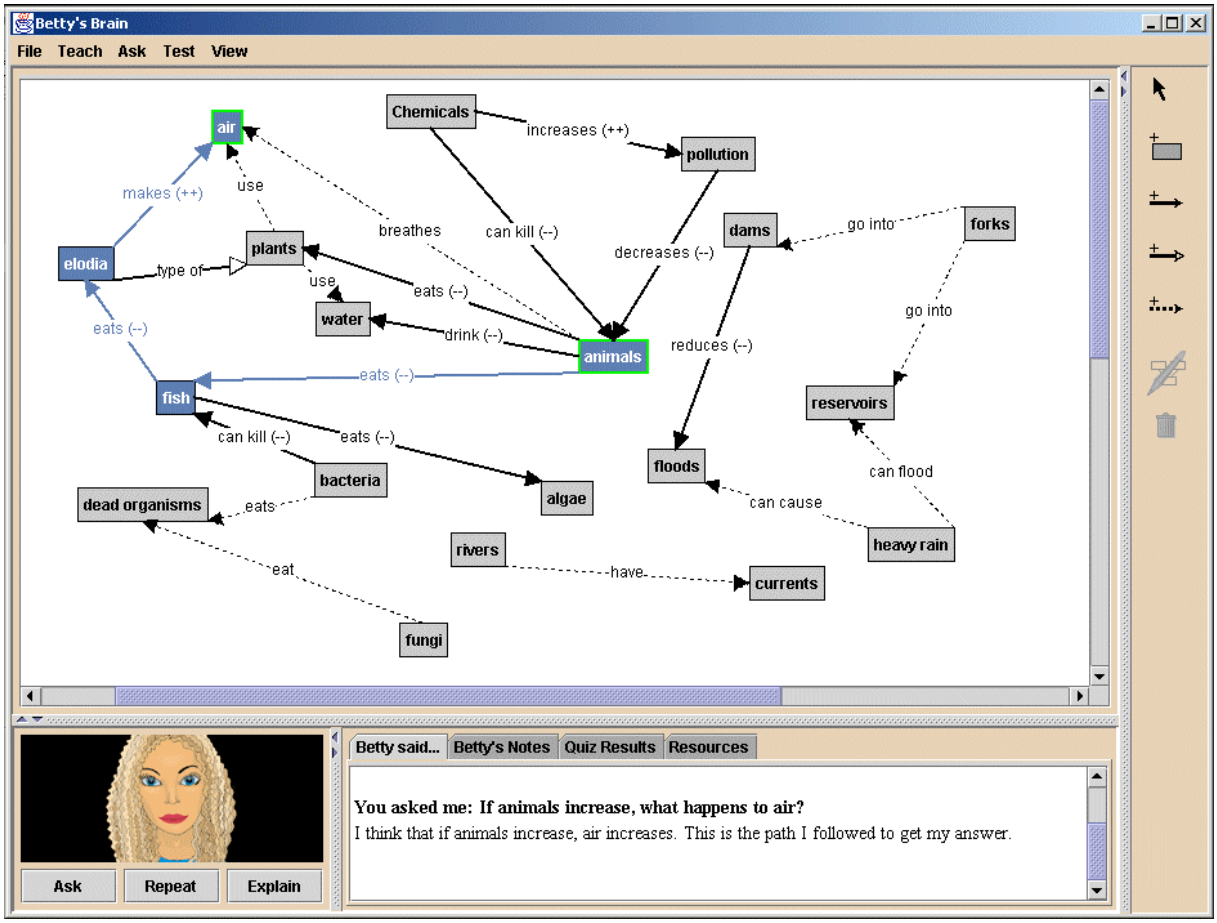
A GTL é de certa forma, uma extensão da STL para grafos e seus algoritmos. É independente de plataforma e desenvolvida em C++. Essa biblioteca possui classes de vértices, arestas e algoritmos, as quais são necessárias para o trabalho com grafos. Dentre as classes de algoritmos da biblioteca encontram-se classes para execução de BFS e DFS, encontrar a árvore geradora de custo mínimo, testar se um grafo é ou não planar, além de algoritmos de fluxo máximo. Outros algoritmos, a serem adicionados à biblioteca, ainda estão em fase de desenvolvimento (RAITNER, 2008).

### 2.5.4 JGRAPH

Segundo Benson (2007, p. 7), JGraph é uma biblioteca para visualização de grafos com código aberto, desenvolvida em linguagem Java. Ela possui algumas funcionalidades voltadas ao desenho de grafos para aplicações de cliente ou servidor. JGraph possui uma simples *Application Programing Interface* (API) que permite visualizar, interagir, gerar *layout* automaticamente e realizar a análise de grafos. Através de sua API, esta biblioteca fornece meios para configurar como um grafo ou uma rede serão mostrados e meios para associar um contexto aos elementos apresentados. Esta biblioteca é fortemente voltada à

visualização, sendo possível a criação até mesmo de diagramas UML, redes sociais e financeiras e circuitos eletrônicos. A figura 9 apresenta um exemplo de diagrama feito com a utilização do JGraph.

A ferramenta apresentada na figura 9, chamada “Betty’s Brain”, serve para o ensino do agente inteligente “Betty”. Esse agente é ensinado utilizando-se de fluxogramas construídos com o auxilio da biblioteca JGraph. Cada nodo representa um objeto enquanto que as conexões entre eles são ações, as quais têm influência sobre os objetos. A contribuição da biblioteca JGraph com a aplicação é, no entanto, apenas com a interface gráfica que permite a construção do fluxograma. A programação lógica da ferramenta deve ser realizada pelo programador.




Fonte: Davis (2005).

Figura 9 – Aplicação desenvolvida utilizando a JGraph

# 3 DESENVOLVIMENTO

Neste capítulo são abordados, a princípio, os requisitos da ferramenta e sua especificação. Além disso, é explicada a implementação da mesma detalhando as ferramentas utilizadas em seu desenvolvimento e como interage com o usuário. Por fim são apresentados os resultados obtidos.

## 3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta permite ao usuário a criação de vértices e arestas utilizando-se apenas do *mouse*. Dessa forma, o mesmo pode criar seu próprio grafo ou dígrafo na forma que preferir visualizá-lo.

Após a construção do grafo o usuário pode escolher, em uma lista a ele disponível, um algoritmo a ser aplicado sobre o grafo construído. Enquanto o algoritmo estiver sendo executado, sua execução é mostrada ao usuário através da coloração das arestas e vértices do grafo. Por fim é apresentado o resultado final obtido pelo algoritmo.

O desenvolvimento da ferramenta foi realizado utilizando a linguagem C++ e sua arquitetura foi dividida, obedecendo ao estabelecido pelo padrão de projetos MVC. A interface foi implementada fazendo uso das bibliotecas IUP e OpenGL, enquanto a camada lógica do programa foi desenvolvida usando a biblioteca GraphObj.

## 3.2 ESPECIFICAÇÃO

A seguir é descrita a especificação da ferramenta. Para especificar seu funcionamento são utilizados diagramas de casos de uso e de seqüência. Por fim é apresentada a modelagem da ferramenta através de diagramas de classes e as principais classes são explicadas.

### 3.2.1 Diagrama de casos de uso

Esta seção apresenta os seis casos de uso da ferramenta, os quais são apresentados no diagrama da figura 10. O caso de uso UC01 serve para que o usuário defina o tipo de grafo que irá criar. Como explicado na fundamentação teórica, um grafo pode ser não dirigido, conhecido simplesmente como grafo, ou dirigido, conhecido como dígrafo. Esse caso de uso é importante porque a definição do tipo de grafo a ser criado define como são criadas as arestas do grafo, como é criada sua lista de adjacência e como certos algoritmos atuam sobre o grafo.

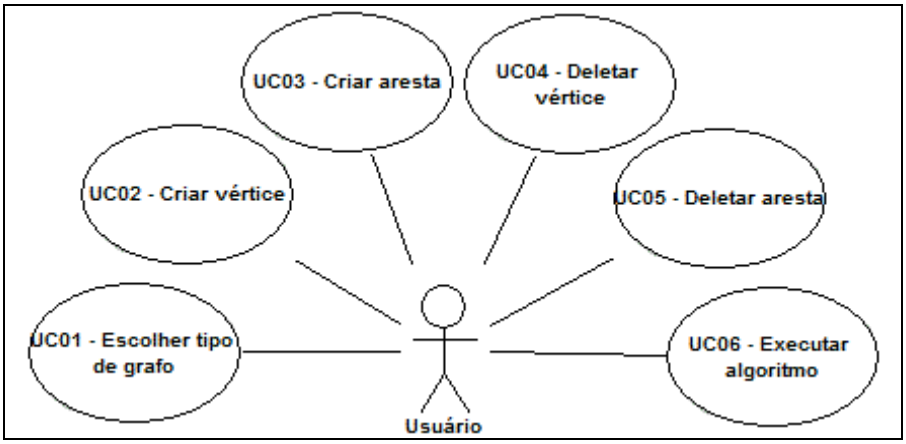



Figura 10 – Diagrama de casos de uso da ferramenta

O caso de uso UC02 serve para a criação dos vértices no grafo. Para começar a criar os vértices, o usuário já deve ter escolhido o tipo de grafo que deseja criar. Neste caso de uso o usuário pode escolher criar um ou mais vértices. Se desejar criar um único vértice, o usuário deve escolher uma posição onde criá-lo. O sistema deve então encarregar-se de criar o vértice e desenhá-lo para a visualização pelo usuário. No entanto, caso o usuário opte por criar mais de um vértice, escolhe várias posições onde o sistema deve criá-los.

O caso de uso UC03 tem como objetivo a criação de arestas no grafo. Para que uma aresta seja criada deve haver ao menos dois vértices criados no grafo. O usuário deve optar por criar uma ou mais arestas assim como é feito com a criação dos vértices. Deve então selecionar os dois vértices entre os quais deseja que a aresta seja criada e o sistema se encarrega de criá-la e desenhá-la. Assim como acontece com a criação dos vértices, se o usuário optar por criar apenas uma aresta, essa é criada e, caso deseje criar outra aresta, o usuário deve solicitar novamente a criação de arestas. Caso solicite a criação de mais arestas, o usuário continua selecionando vértices para a criação das arestas até que solicite outra opção.

Os casos de uso UC04 e UC05 servem para excluir os vértices e arestas do grafo

respectivamente. Para fazê-lo, logicamente, deve haver algum vértice ou aresta criada no grafo. O usuário deve selecionar o componente que deseja excluir e, então, solicitar sua exclusão. O sistema deve realizar a exclusão do componente selecionado.

O caso de uso UC06 é encarregado da execução dos algoritmos no grafo. Para isso um grafo já deve ter sido criado. O usuário deve selecionar o algoritmo que deseja executar. Caso opte por rodar o algoritmo BFS ou DFS, após solicitar a execução do algoritmo o usuário deve selecionar dois vértices que são os vértices, inicial e final da busca respectivamente. O sistema executa o algoritmo sobre o grafo apresentando ao usuário, passo a passo, o caminhamento executado pelo algoritmo. Por fim o sistema apresenta o resultado final da execução. A única diferença caso o usuário opte pelo algoritmo de Kruskal é que não é necessária a seleção de nenhum vértice para a execução do algoritmo sendo que ao solicitar sua execução, o caminhamento começa imediatamente a ser executado e apresentado ao usuário assim como seu resultado final.

### 3.2.2 Diagrama de seqüência

Esta seção apresenta o diagrama de seqüência da ferramenta, que explica a seqüência a ser seguida pelo usuário para o uso da mesma e a seqüência de ações do próprio sistema. O diagrama apresentado na figura 11 apresenta a seqüência para a criação de um vértice no grafo.

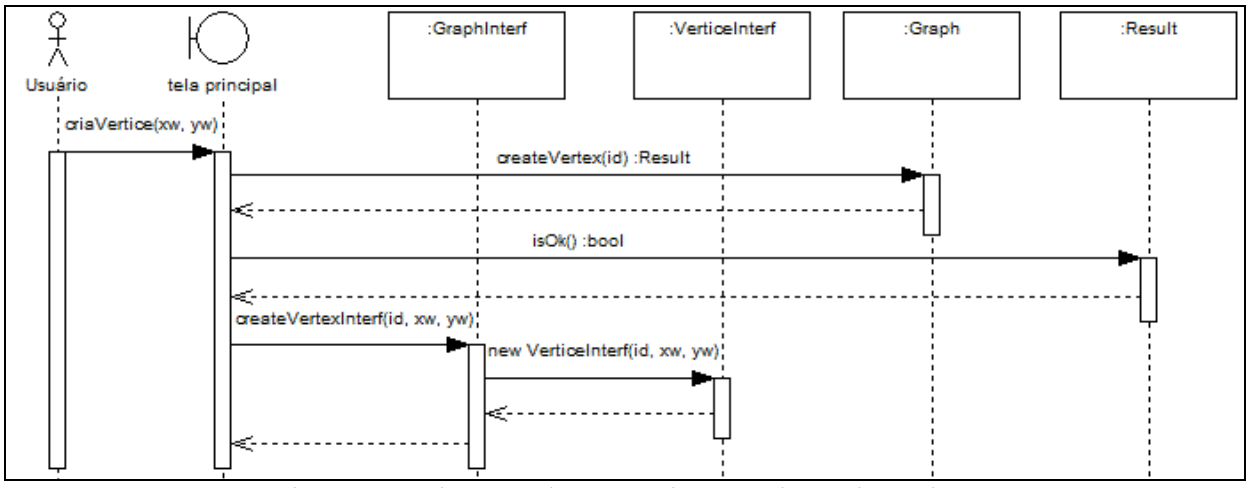



Figura 11 – Diagrama de seqüência para criação de vértices

O diagrama da figura 11 apresenta a seqüência de chamadas de métodos realizada pelo sistema para que o usuário crie vértices no grafo. Caso o usuário opte por criar um dígrafo, a

seqüência executada é a mesma, sendo que a única diferença é que o método `createVertex` é chamado pela classe `Digraph` ao invés de pela classe `Graph`.

Como explicado nos casos de uso, após a criação de pelo menos dois vértices, o usuário pode criar arestas. O diagrama de seqüência da criação de uma aresta é apresentado na figura 12.

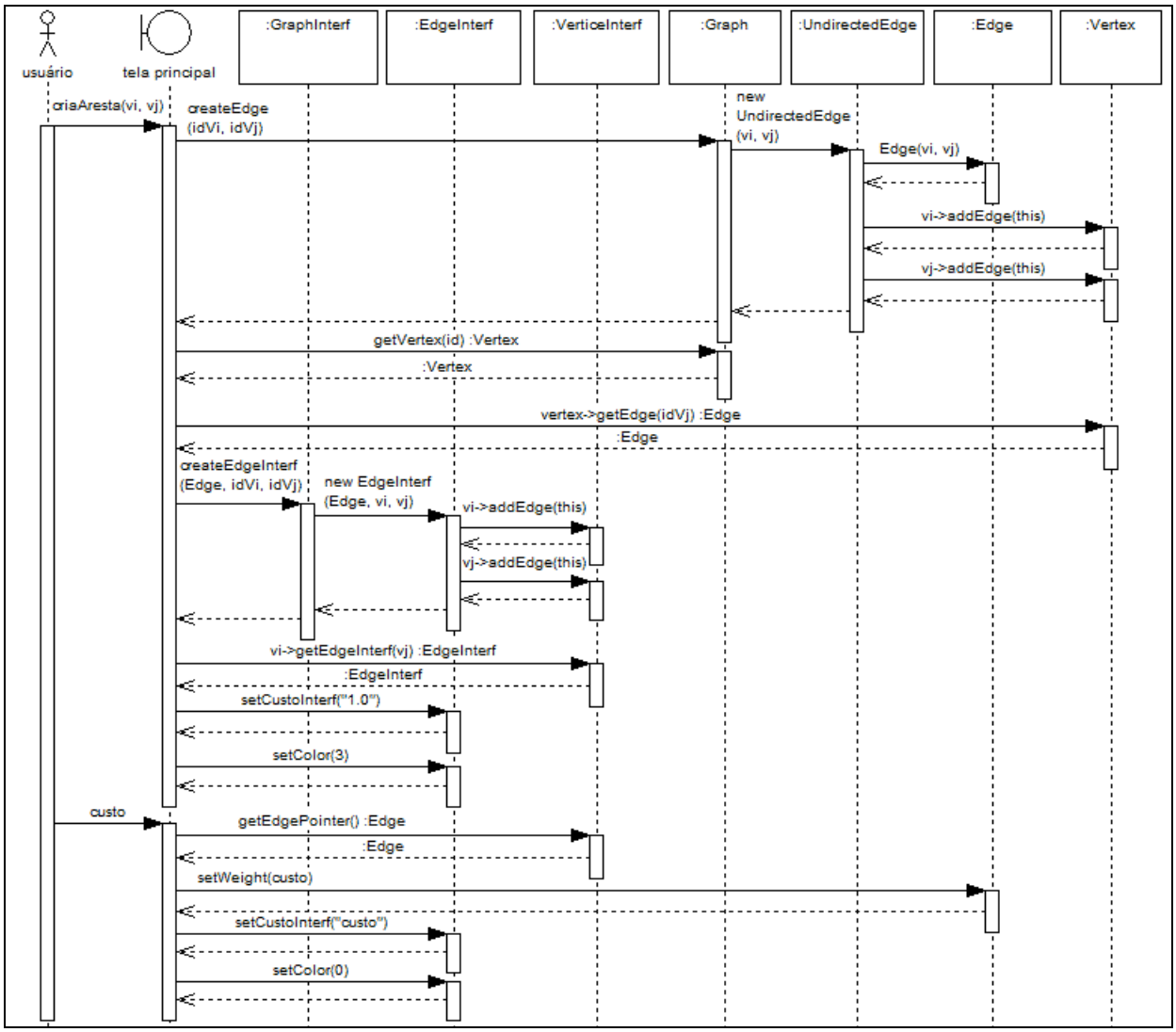



Figura 12 – Diagrama de seqüência para criação de arestas

Assim como na criação dos vértices, caso o usuário opte inicialmente por criar um dígrafo, o método `createEdge` é chamado na classe `Digraph` que por sua vez, instancia um objeto da classe `DirectedEdge`. A aresta é então adicionada apenas ao vértice `vi` chamando o método `addEdge` desse vértice. Desse modo o caminhamento no grafo respeita as direções das arestas sendo que, quando um algoritmo for executado, só encontrará essa aresta quando estiver sobre o vértice `vi`.

Como explicado nos casos de uso, após a criação do grafo o usuário pode optar por executar um algoritmo sobre o grafo. A figura 13 apresenta o diagrama de seqüência para a

execução do algoritmo BFS.

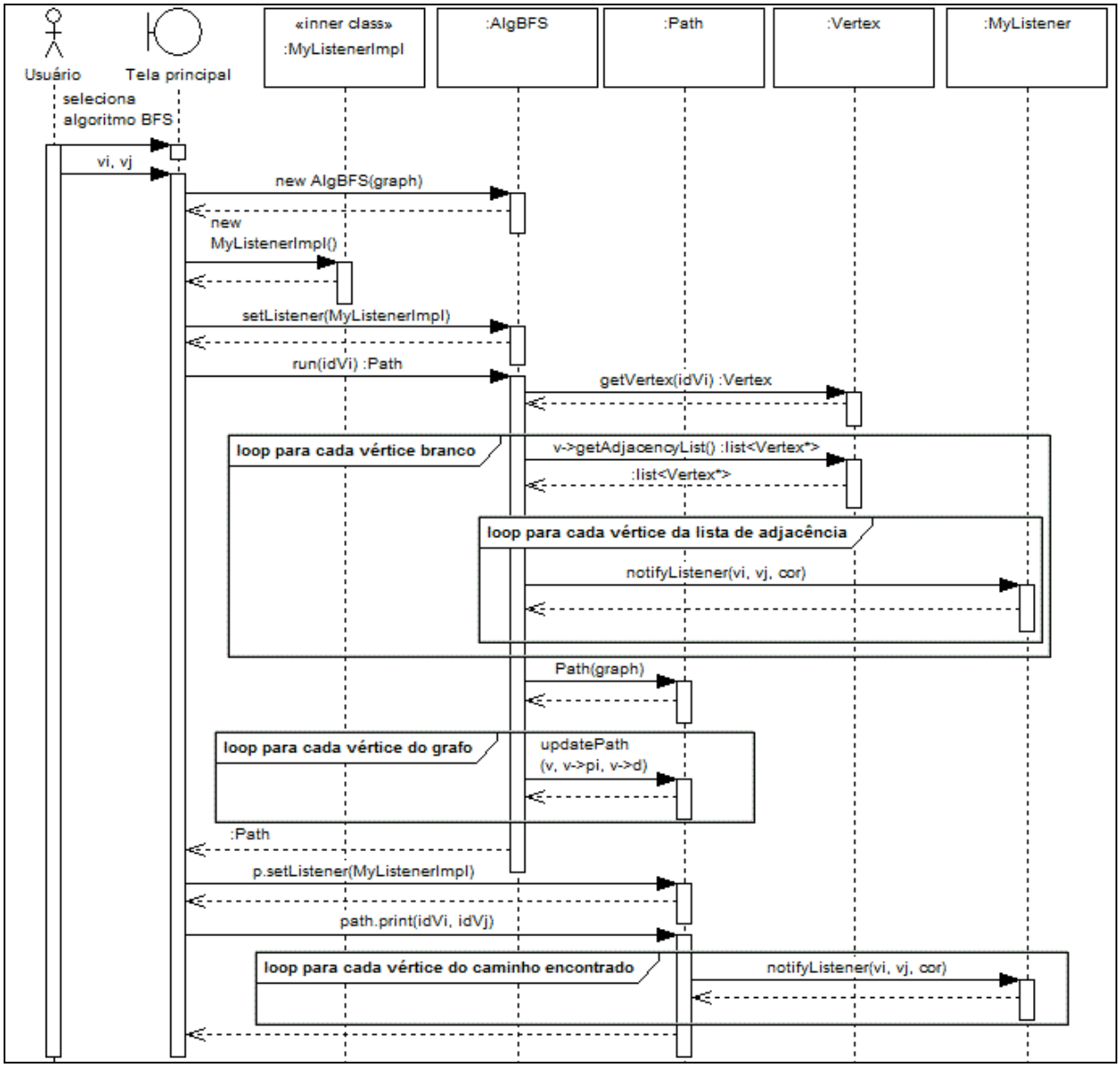



Figura 13 – Diagrama de seqüência para execução do algoritmo BFS

Esse diagrama apresenta a seqüência para executar o algoritmo BFS disponibilizado pela ferramenta. A classe `MyListener` apresentada na figura 13 é uma classe abstrata, enquanto a classe `MyListenerImpl` tem relação de herança com essa classe e é implementada dentro de outro arquivo fonte. Estas classes servem para que a camada lógica da ferramenta comunique-se com a interface, notificando-a quando alguma mudança na interface deve ser realizada. Esta implementação é sugerida pelo padrão *Command* sendo que dessa forma, o utilizador da classe `MyListener` não precisa conhecer sua implementação concreta, fazendo com que haja um fraco acoplamento entre o objeto que faz a notificação e o notificado.

### 3.2.3 Diagrama de classes

Esta seção apresenta os diagramas de classe da ferramenta que apresentam as classes da ferramenta e como elas estão relacionadas. Para melhor visualizar tais relações e explicá-las são apresentados três diagramas. O primeiro, na figura 14, apresenta as classes da biblioteca GraphObj.

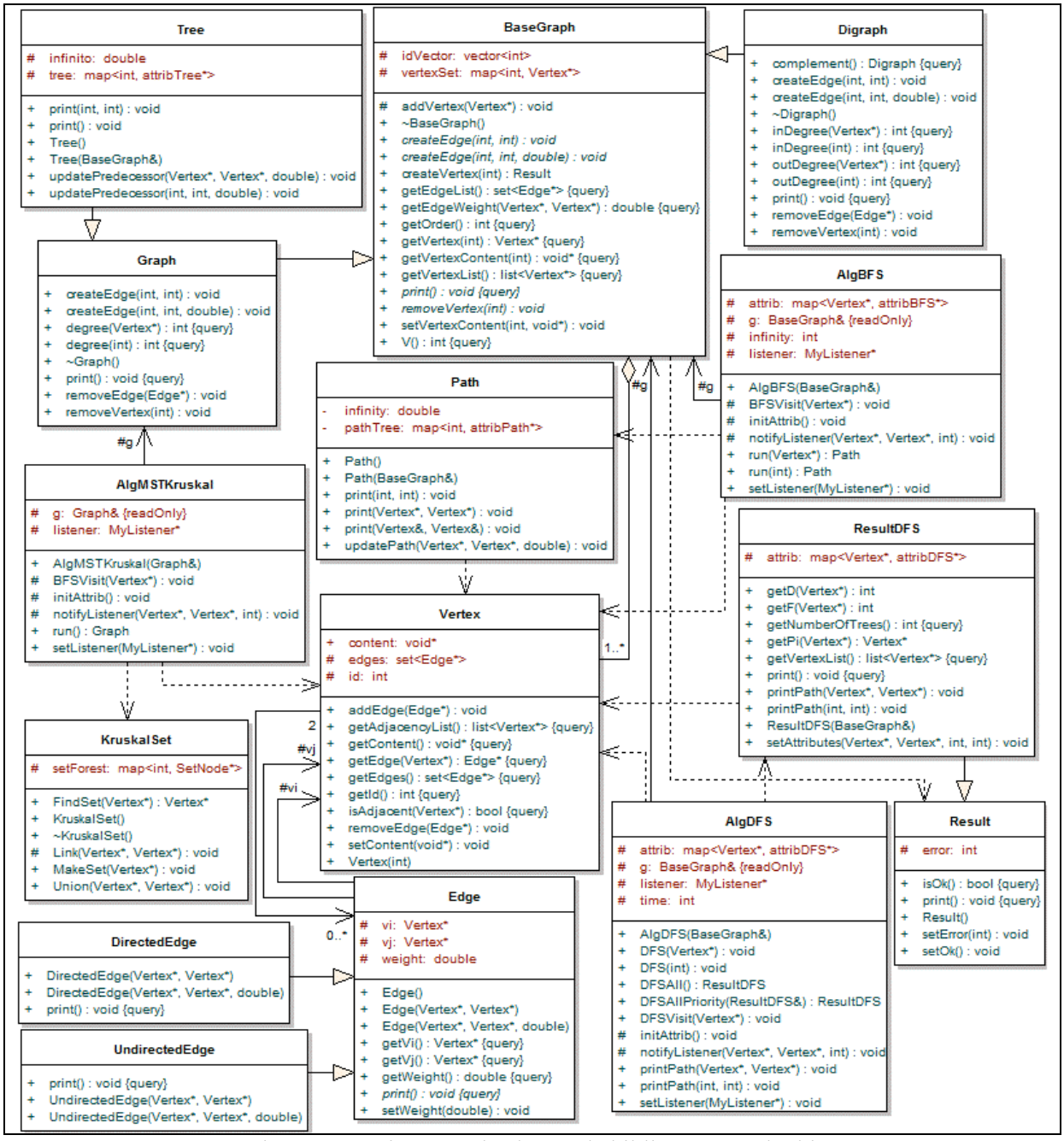



Figura 14 – Diagrama de classes da biblioteca GraphObj

O diagrama apresentado na figura 14 mostra como as classes da biblioteca GraphObj estão relacionadas. Como apresentado na fundamentação teórica, as classes responsáveis pela

criação do grafo na camada lógica são `BaseGraph`, `Graph`, `Digraph`, `Vertex`, `Edge`, `DirectedEdge` e `UndirectedEdge`. `BaseGraph` é uma classe abstrata da qual generalizam as classes `Graph` e `Digraph`. Essa classe ainda agrega um ou mais objetos da classe `Vertex` que, por sua vez, está associada à classe `Edge` de modo que um objeto da classe `Edge` necessita de dois vértices para ser criado, enquanto os objetos da classe `Vertex` podem possuir várias arestas associadas a eles ou nenhuma. As classes `DirectedEdge` e `UndirectedEdge` são generalizadas a partir da classe `Edge`. Essas são responsáveis pela criação de arestas dirigidas e não dirigidas. Para cada aresta criada, dirigida ou não dirigida, um objeto de `Edge` é criado sendo que essas generalizações definem apenas se a aresta é adicionada à lista de adjacência de ambos os vértices que a compõe, ou apenas à lista do vértice de origem. A classe `BaseGraph` possui ainda a relação de dependência com `Result`, porque o método `createVertex()` retorna um objeto de `Result` para que se possa descobrir se o vértice foi criado com sucesso ou se houve erro durante essa operação.

Após a criação do grafo é possível executar algoritmos sobre o mesmo. As classes responsáveis pelos algoritmos BFS, DFS e Kruskal são `AlgBFS`, `AlgDFS` e `AlgMSTKruskal` respectivamente. Para a execução do algoritmo BFS é necessário, primeiramente, passar por parâmetro na instanciação do objeto `AlgBFS` um grafo ou um dígrafo que será guardado em uma variável e utilizado apenas para leitura. Por isso essa classe possui uma associação com a classe `Basegraph`. Essa classe possui também dependência com `Path` e `Vertex`. A primeira dependência existe porque `AlgBFS` necessita de `Path` para realizar o método `run()` sendo que será guardado em um objeto de `Path`, o resultado do caminhamento realizado pelo algoritmo. A segunda existe porque há na classe `AlgBFS` uma estrutura que guarda um objeto da classe `Vertex` para controle de vértices predecessores e sucessores na busca. As classes `Path`, `AlgDFS`, `ResultDFS` e `AlgMSTKruskal` possuem essa mesma relação com `Vertex` pelo mesmo motivo. O método responsável pelo caminhamento do algoritmo BFS no grafo é o `BFSVisit()`.

Para a execução do algoritmo DFS existem os métodos `DFS()`, passando por parâmetro um objeto de `Vertex` ou um inteiro que é a identificação do vértice. Esse método é responsável por chamar os métodos `initAttrib()`, para inicializar a estrutura que guarda os vértices visitados na busca, e o método `DFSVisit()`, que recebe um vértice por parâmetro e é responsável pelo caminhamento do algoritmo no grafo. Os métodos `DFSAll()` e `DFSAllPriority()`, após o caminhamento do algoritmo no grafo, passam o resultado do caminhamento para a classe `ResultDFS` e retornam um objeto dessa classe. `ResultDFS` possui

métodos que permitem descobrir os tempos de abertura (`getD()`) e fechamento (`getF()`) dos vértices, os quais foram determinados durante a execução do algoritmo. Ambas as classes `AlgDFS` e `ResultDFS` podem apresentar o caminho realizado pelo algoritmo.

Ao instanciar-se um objeto da classe `AlgMSTKruskal` passa-se por parâmetro um objeto da classe `Graph`, sendo que esse algoritmo só é realizado em grafos e não em dígrafos. O método `run()` é responsável pela execução desse algoritmo em conjunto com a classe `KruskalSet`. Como explicado na fundamentação teórica, esse algoritmo define inicialmente que cada vértice do grafo é uma árvore. Em seguida vai verificando as arestas do grafo em ordem crescente de custo e conectando diferentes árvores. Para que não conecte vértices de uma mesma árvore, o que causaria a criação de um ciclo, antes de juntar as árvores é verificado se os vértices em questão fazem parte do mesmo conjunto. Para realizar essa verificação, criar e unir árvores no grafo, é utilizada a classe `KruskalSet`. Durante a execução desse algoritmo é criado um grafo, o qual guarda a árvore resultante do algoritmo, que é retornado pelo método `run()`.

A figura 15 apresenta um diagrama de classes da ferramenta, no entanto não apresenta ainda as relações com a biblioteca GraphObj, de forma a apresentar melhor as classes criadas para a ferramenta e explicá-las permitindo melhor visualização.

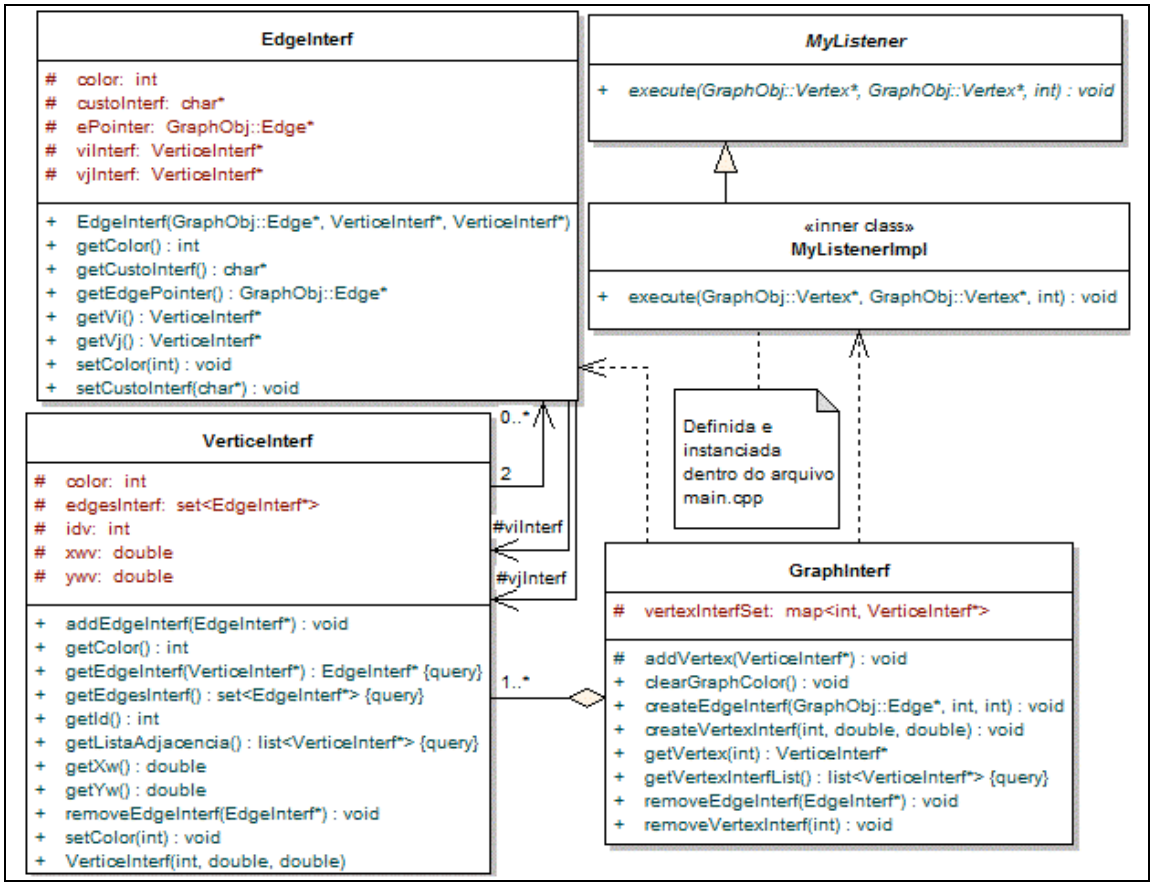



Figura 15 – Diagrama de classes da ferramenta

O diagrama da figura 15 apresenta cinco classes criadas para a ferramenta. As classes `VertexInterf` e `EdgeInterf` são utilizadas para guardar as informações que os vértices e as arestas têm com relação à interface. Sendo assim cada vértice criado gera um objeto da classe `VertexInterf` que guarda seu identificador, a cor em que o vértice será desenhado, a sua posição na interface, e uma lista de objetos da classe `EdgeInterf`. Seu construtor obriga que sejam passados por parâmetro o identificador e a posição do vértice a ser criado. O restante dos métodos serve para definir e retornar os atributos dessa classe, bem como adicionar e remover arestas da lista de adjacência do vértice ou retornar essa lista. A classe `EdgeInterf` por sua vez, possui atributos para guardar seu custo, cor, os objetos de `VerticeInterf` que a compõe e um ponteiro para um objeto da classe `Edge` na biblioteca GraphObj. Seus métodos existem apenas para definir e retornar seus atributos. Assim como acontece na biblioteca GraphObj, cada aresta definida necessita de dois objetos da classe `VerticeInterf`, no entanto os vértices não precisam estar ligados a nenhuma aresta.

A classe `GraphInterf` possui como atributo apenas uma lista que guarda ponteiros para objetos do tipo `VerticeInterf`. Os métodos `createVertexInterf()` e `createEdgeInterf()` são utilizados para a criação dos objetos `VertexInterf` e `EdgeInterf` no grafo. Possui os métodos `addVertex()`, `getVertex()` e `getVertexInterfList()` de modo que, o primeiro é responsável pela adição de vértices à lista de vértices do grafo, o segundo retorna um vértice do grafo a partir de um identificador do vértice que é passado por parâmetro  e o terceiro, retorna a lista completa de vértices do grafo. Os métodos `removeEdgeInterf()` e `removeVertexInterf()` servem para a exclusão de arestas e vértices do grafo respectivamente. Enfim o método `clearGraphColor()` é utilizado para retornar as cores dos componentes do grafo ao seu valor inicial. Assim como na GraphObj, essa classe agrega um ou mais vértices. No entanto essa classe possui uma relação de dependência com a classe `EdgeInterf` pois possui métodos para criar e remover arestas.

Por fim encontra-se no diagrama as classes `MyListener` e `MyListenerImpl`. Como já comentado anteriormente, essas classes servem para a comunicação entre a camada lógica e a interface. Com o uso dessas classes é possível notificar a interface sobre mudanças que devem ser realizadas durante a execução de um algoritmo. A classe `MyListener` é uma classe abstrata com relação de herança para com a classe `MyListenerImpl` sendo que a primeira, possui a definição do método `execute()`, enquanto a segunda implementa esse método. Essa forma de desenvolvimento é sugerida pelo padrão *Command*, permitindo que haja comunicação entre a camada lógica e a interface com fraco acoplamento, de modo a não

prejudicar o modelo MVC.

A figura 16 apresenta o diagrama completo da aplicação, apresentando as relações existentes entre as classes da ferramenta e da biblioteca GraphObj.

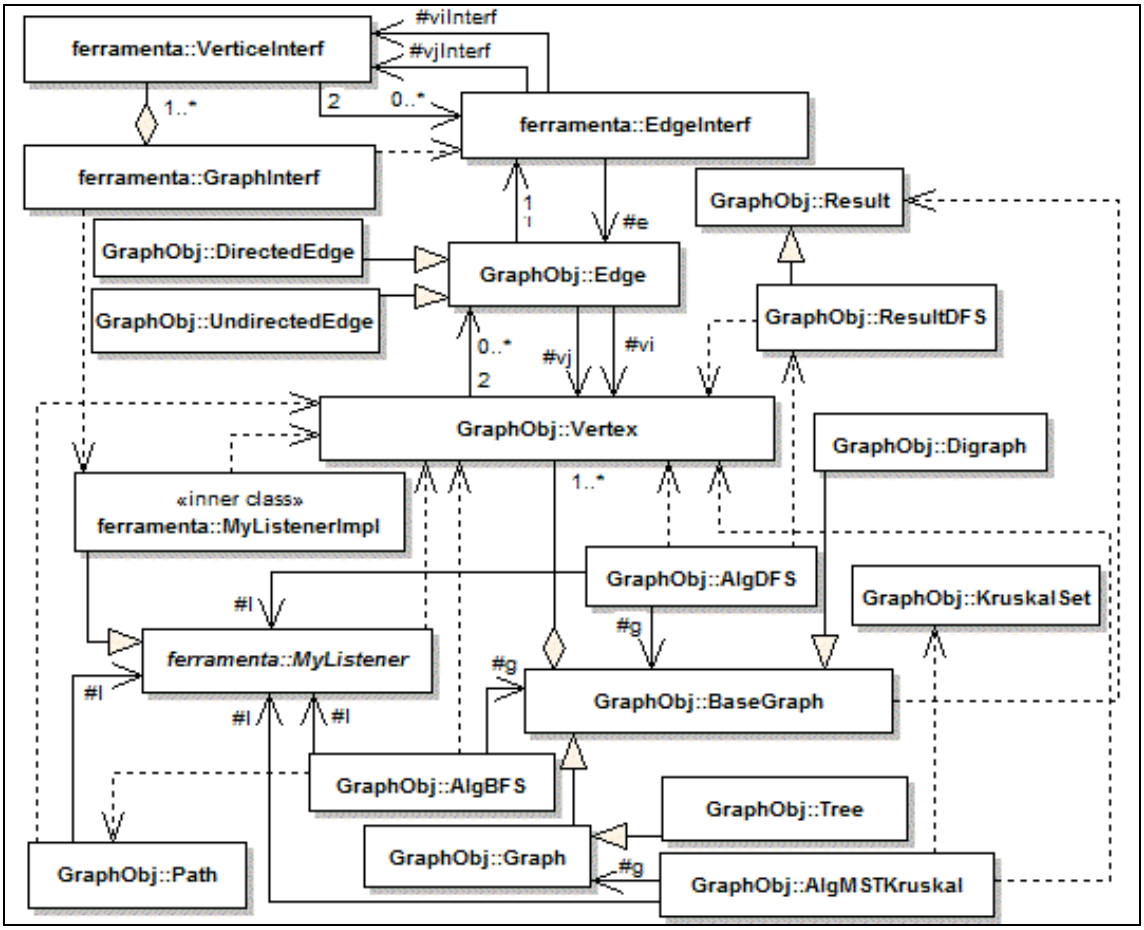



Figura 16 – Diagrama de classes da integração entre GraphObj e a ferramenta

O diagrama da figura 16 apresenta a integração das classes da biblioteca GraphObj com as classes da ferramenta. Como explicado anteriormente, cada objeto da classe `EdgeInterf` possui um ponteiro para um objeto da classe `Edge` da biblioteca, de modo que surge o relacionamento de associação entre essas classes. A classe interna `MyListenerImpl` possui dependência com `Vertex` porque o método `execute()`, nela implementado, recebe objetos de `Vertex` por parâmetro. Por fim, a classe abstrata `MyListener`, além da dependência com `Vertex` pelo mesmo motivo que a classe `MyListenerImpl`, possui associação com `Path`, `AlgMSTKruskal`, `AlgBFS` e `AlgDFS`. Essas associações existem porque essas classes possuem um atributo da classe `MyListener`, o qual utilizam afim de notificar a interface para que seu caminhamento no grafo seja apresentado ao usuário.

## 3.3 IMPLEMENTAÇÃO

Esta seção apresenta primeiramente as técnicas e ferramentas utilizadas na programação da aplicação. Em seguida é mostrado o desenvolvimento da ferramenta em nível de código. Por fim, é apresentado o modo como o usuário pode interagir com a ferramenta.

### 3.3.1 Técnicas e ferramentas utilizadas

Para a implementação da ferramenta foi utilizado o ambiente de programação Microsoft Visual Studio 2005. Dentre as linguagens de programação que se pode utilizar nesse ambiente, foi utilizada a linguagem C++. O quadro 5 apresenta um trecho de código implementado em C++, o qual é responsável por remover um vértice do grafo que é apresentado na interface e suas arestas adjacentes.

```
void GraphInterf::removeVertexInterf(int idv)
{
  set<EdgeInterf*> edgeInterf_set = vertexInterfSet[idv]->getEdgesInterf();
  set<EdgeInterf*>::iterator i;
  EdgeInterf* edgeInterf;
  for (i = edgeInterf_set.begin(); i != edgeInterf_set.end(); i++) {
    edgeInterf = (*i);
    removeEdgeInterf(edgeInterf);
  }
  VerticeInterf* v = vertexInterfSet[idv];
  delete v;
  vertexInterfSet.erase(idv);
}
```

Quadro 5 – Método `removeVertexInterf`

Além da linguagem C++ no ambiente Microsoft Visual Studio 2005, foram utilizados os padrões de projeto conhecidos como MVC e *Command*. Como explicado anteriormente, para a utilização do padrão *Command* foram criadas as classes `MyListener` e `MyListenerImpl`. A primeira é uma classe abstrata que serve apenas de interface, sendo que possui apenas a definição de um método, sem sua implementação. Essa classe é apresentada no quadro 6.

```
class MyListener{
public:
      virtual void execute(GraphObj::Vertex *vi, GraphObj::Vertex *vj, int
color) = 0;
};
```

Quadro 6 – Classe abstrata `MyListener`

A classe `MyListenerImpl`, por sua vez, possui a implementação do método definido

pela classe abstrata com a qual possui relação de herança e é apresentada no quadro 7.

```
class MyListenerImpl: public MyListener {
      void execute(GraphObj::Vertex *vi, GraphObj::Vertex *vj, int color) {
            if(vi == vj){
                  VerticeInterf *v = graphInterf.getVertex(vi->getId());
                  v->setColor(color);
            }else{
                  VerticeInterf *v1 = graphInterf.getVertex(vi->getId());
                  VerticeInterf *v2 = graphInterf.getVertex(vj->getId());
                  EdgeInterf *e = v1->getEdgeInterf(v2);
                  v2->setColor(color);
                  e->setColor(color);
            }
            display();
            Sleep(1000);
      }
};
```

Quadro 7 – Classe `MyListenerImpl`

A classe apresentada no quadro 7 atualiza as cores do grafo quando o método `execute` é chamado através da classe `MyListener`. As chamadas para esse método ocorrem durante a execução dos algoritmos.

### 3.3.2 Desenvolvimento da ferramenta

O desenvolvimento da ferramenta pode ser dividido basicamente em construção do grafo e execução dos algoritmos. No entanto, para a construção do grafo é necessário criar vértices e arestas. O código fonte da ferramenta que mostra como essas funções foram implementadas são apresentados a seguir.

#### 3.3.2.1 Desenvolvimento do grafo

Para o desenvolvimento do grafo são criados vértices e arestas. Ao solicitar-se a criação de um vértice, é chamado no arquivo “main.cpp” o método `criaVertice`. Esse método chama, primeiramente, o método `createVertex` encontrado na classe `BaseGraph`, o qual é apresentado no quadro 8.

```
Result BaseGraph::createVertex(int id)
{
  Result result;
  // verifica se o id já existe
  bool existingId = binary_search(idVector.begin(), idVector.end(), id);
  if (existingId) {
    result.setError(GRAPHOBJ_EXISTING_ID_ERROR);
    return result;
  }
  else {
    Vertex* v = new Vertex(id);
    addVertex(v);
    idVector.push_back(id);
    sort(idVector.begin(), idVector.end());
    result.setOk();
    return result;
  }
}
```

Quadro 8 – Método `createVertex`

O código apresentado no quadro 8 é responsável pela criação do vértice no grafo da biblioteca GraphObj. Esse método recebe o identificador do vértice por parâmetro e verifica, em uma lista de identificadores de vértices que se encontra na classe `Basegraph`, se já existe algum vértice com o mesmo identificador. Caso exista, o vértice não pode ser criado e um objeto de `Result` com uma mensagem de erro é retornado. Se não existir, é então instanciado um objeto da classe `Vertex` que recebe em seu construtor, o identificador do vértice por parâmetro. Em seguida o vértice é adicionado à lista de vértices existente na classe Basegraph e seu identificador, adicionado à lista de identificadores dos vértices do grafo. Por fim a lista de identificadores é ordenada e um objeto de `Result`, com uma mensagem informando que a operação foi realizada com sucesso, é retornado.

Em seguida, no arquivo "main.cpp", verifica-se a partir do objeto retornado se o vértice foi criado com sucesso. Caso tenha sido criado, é então chamado o método `createVertexInterf` da classe `GraphInterf`, passando por parâmetro o identificador do vértice e sua posição na tela. Esse método é apresentado no quadro 9.

```
void GraphInterf::createVertexInterf(int id, double xw, double yw){
      VerticeInterf* v = new VerticeInterf(id, xw, yw);
    addVertex(v);
}
```

Quadro 9 – Método `createVertexInterf`

O código apresentado no quadro 9 simplesmente cria um novo objeto da classe `VerticeInterf`, o qual recebe por parâmetro um identificador e sua posição na tela. Esses valores passados por parâmetro são definidos como atributos do objeto de `VerticeInterf`. Por fim, esse objeto é adicionado à lista de vértices encontrada em `GraphInterf`.

Após a criação de pelo menos dois vértices no grafo, pode-se criar uma aresta entre

eles. Para isso são escolhidos pelo usuário dois vértices entre os quais a aresta deve ser criada e, então, é chamado o método `criaAresta` no arquivo "main.cpp". Esse método verifica se o usuário optou inicialmente pela criação de um grafo ou um dígrafo. Caso tenha optado por criar um grafo, o método `createEdge` da classe `Graph` será chamado. Esse método é apresentado no quadro 10.

```
void Graph::createEdge(int i, int j)
{
      Vertex* vi = vertexSet[i];
      Vertex* vj = vertexSet[j];

      new UndirectedEdge(vi, vj);
}
```

Quadro 10 – Método `createEdge` da classe `Graph`

O método apresentado no quadro 10 recupera os vértices cujos identificadores equivalem aos valores passados por parâmetro. Em seguida cria um objeto da classe `UndirectedEdge`, passando os dois vértices ao seu construtor. Esse método cria um objeto de `Edge` passando por parâmetro os dois vértices escolhidos pelo usuário, então adiciona a aresta à lista de adjacência de ambos os vértices, como é possível ver no quadro 11.

```
UndirectedEdge::UndirectedEdge(Vertex *vii, Vertex *vjj) : Edge(vii, vjj)
{
  if ((vii!=0)&&(vjj!=0)) {
    vii->addEdge(this);
    vjj->addEdge(this);
  }
}
```

Quadro 11- Construtor da classe `UndirectedEdge`

A diferença, caso o usuário opte por criar um dígrafo, é que nesse caso o método `createEdge` da classe `Digraph` é chamado. Nesse método é instanciado um objeto da classe `DirectedEdge` que, por sua vez, também cria um objeto de `Edge`. No entanto a aresta é adicionada apenas à lista de adjacência do vértice inicial. O método `createEdge` de `Digraph` e o construtor da classe `DirectedEdge` são apresentados nos quadros 12 e 13 respectivamente.

```
void Digraph::createEdge(int i, int j)
{
  // verifica se o id já existe
  bool existingIdI = binary_search(idVector.begin(), idVector.end(), i);
  bool existingIdJ = binary_search(idVector.begin(), idVector.end(), j);

  if (existingIdI && existingIdJ) {
    Vertex *vi = vertexSet[i];
    Vertex *vj = vertexSet[j];
    new DirectedEdge(vi, vj);
  }
}
```

Quadro 12 – Método `createEdge` da classe `Digraph`

```
DirectedEdge::DirectedEdge(Vertex *vii,
                           Vertex *vjj): Edge(vii, vjj)
{
  if (vii != 0)
    vii->addEdge(this);
}
```

Quadro 13 – Construtor da classe `DirectedEdge`

Após a construção da aresta dentro da biblioteca GraphObj, é chamado o método `createEdgeInterf` da classe `GraphInterf`. Esse método recebe, por parâmetro, os identificadores dos vértices que compõem a aresta e um ponteiro para o objeto `Edge` criado na GraphObj. O quadro 14 apresenta o código fonte de `createEdgeInterf`.

```
void GraphInterf::createEdgeInterf(GraphObj::Edge* e, int i, int j)
{
      VerticeInterf* vi = vertexInterfSet[i];
      VerticeInterf* vj = vertexInterfSet[j];
      new EdgeInterf(e, vi, vj);
}
```

Quadro 14 – Método `createEdgeInterf`

O código no quadro 14 serve para recuperar os vértices que compõem a aresta a partir de seus identificadores e, então, instanciar um novo objeto de `EdgeInterf`. Ao instanciá-lo, é necessária a passagem do ponteiro recebido por `createEdgeInterf` e dos vértices encontrados a partir dos identificadores. O construtor de `EdgeInterf` por sua vez, define os parâmetros a ele passados como atributos do objeto e adiciona o objeto `EdgeInterf` à lista de adjacência de ambos os vértices. O construtor dessa classe é apresentado no quadro 15.

```
EdgeInterf::EdgeInterf(GraphObj::Edge *e, VerticeInterf *vi, VerticeInterf
*vj):ePointer(e), viInterf(vi), vjInterf(vj){
      vi->addEdgeInterf(this);
      vj->addEdgeInterf(this);
      color = 0;
}
```

Quadro 15 – Construtor da classe `EdgeInterf`

Após a criação de `EdgeInterf`, essa aresta é selecionada para que o usuário possa definir seu custo. Agora, com o grafo criado, é possível executar algoritmos sobre o mesmo.

A seção a seguir apresenta a implementação dos algoritmos que se encontram na biblioteca GraphObj, como os métodos que implementam esses algoritmos são chamados e como os mesmos tem influência na interface.

#### 3.3.2.2 Desenvolvimento dos algoritmos

Após a criação do grafo, o usuário pode optar por executar um algoritmo sobre o

mesmo. Para fazê-lo, o usuário deve selecionar um algoritmo em uma lista e solicitar sua execução. Caso selecione pelo algoritmo BFS, o método `algBFS` é chamado no arquivo "main.cpp". O usuário deverá optar por dois vértices para que após a execução do algoritmo, seja apresentado o caminho encontrado entre os mesmos. É instanciado um objeto da classe `AlgBFS`, encontrada na biblioteca GraphObj, e passado por parâmetro o grafo construído, seja esse um grafo ou um dígrafo. Esse construtor é responsável apenas por inicializações que devem ser feitas antes da execução do algoritmo e é apresentado no quadro 15.

```
AlgBFS::AlgBFS(const BaseGraph& gg):g(gg)
{
      listener = NULL;
      infinity = numeric_limits<int>::infinity();

  list<Vertex*> lvert = g.getVertexList();
      list<Vertex*>::iterator i;
      Vertex* v;

      for(i = lvert.begin(); i != lvert.end(); i++){
            v = (*i);
            attribBFS *a = new attribBFS();
            a->pi     = 0;
            a->color  = WHITE;
            a->d      = infinity;
            attrib[v] = a;
      }
}
```

Quadro 15 – Construtor da classe `AlgBFS`

Essa classe possui um atributo `listener`, da classe `MyListener`, o qual é inicializado como nulo. Em seguida atribui à variável `infinity` o maior valor possível para uma variável de valor inteiro. Por fim, para cada vértice do grafo, é instanciado um `attribBFS`. Esse `attribBFS` é uma estrutura definida dentro da classe `AlgBFS`, que possui armazenadas as variáveis `pi`, responsável por armazenar um ponteiro para o vértice que precede o atual na busca, `color`, que define a cor atual do vértice, e `d`, a qual guarda a distância entre o vértice inicial e o vértice atual. Os atributos `pi`, `color` e `d` da estrutura, são inicializados com 0, branco e `infinity` respectivamente. Cada uma dessas estruturas é relacionada a um vértice do grafo.

Ao terminar as inicializações retorna-se ao método `algBFS` que, então, cria uma instância de `MyListenerImpl` e a atribui a uma variável da classe `MyListener`. É chamado a seguir o método `setListener` encontrado na classe `AlgBFS` passando a variável de `MyListener` por parâmetro. Em seguida chama-se o método `run` de `AlgBFS` apresentado no quadro 16.

```
Path AlgBFS::run(int ids)
{
      Vertex* v = g.getVertex(ids);
      initAttrib();
      BFSVisit(v);

      Path c(g);

      map<Vertex*, attribBFS*>::iterator i;

      for(i = attrib.begin(); i != attrib.end(); i++) {
    Vertex* v = i->first;
    c.updatePath(v, attrib[v]->pi, attrib[v]->d);
  }
      return c;
}
```

Quadro 16 – Método `run` da classe `AlgBFS`

O método apresentado no quadro 16 recebe o identificador de um vértice por parâmetro a partir do qual recupera o vértice correspondente ao mesmo. Em seguida chama o método `initAttrib`, responsável por inicializar as estruturas `AttribBFS` de cada vértice. É chamado o método `BFSVisit`, que se encarrega de realizar o caminhamento pelo grafo. O grafo é então passado a uma variável da classe `Path` que serve para guardar o caminho realizado pelo algoritmo no grafo. Para obter-se o caminho completo realizado pelo algoritmo é chamado, para cada vértice, o método `updatePath` da classe `Path` passando por parâmetro um vértice, seu predecessor e a distância entre o vértice inicial e o atual. O método `BFSVisit` recebe um vértice para começar a busca e é apresentado no quadro 17.

```
void AlgBFS::BFSVisit(Vertex* s)
{
      deque<Vertex*> q;
      deque<Vertex*>::iterator i;
      list<Vertex*> adjU;
      list<Vertex*>::iterator j;

  attrib[s]->d = 0;
      attrib[s]->color = GRAY;
      notifyListener(s, s, 1);
      q.push_back(s);
      while(!q.empty()) {
    i = q.begin();
    Vertex* u = (*i);
    q.pop_front();
    adjU = u->getAdjacencyList();
    for (j = adjU.begin(); j != adjU.end(); j++) {
      Vertex* v = (*j);
      if (attrib[v]->color == WHITE) {
        q.push_back(v);
        attrib[v]->color  = GRAY;
        attrib[v]->pi     = u;
        attrib[v]->d      = attrib[u]->d + 1;
        notifyListener(u, v, 1);
      }
      }
      attrib[u]->color = BLACK;
        notifyListener(u, u, 2);
      }
}
```

Quadro 17 – Método `BFSVisit`

O método no quadro 17 executa o algoritmo BFS sobre o grafo. São criadas inicialmente uma fila e uma lista de vértices. O vértice inicial tem definida sua distância `d`, distância entre o vértice inicial e o atual, como zero e sua cor como cinza. É então chamado o método `notifyListener` responsável por chamar o método execute da classe `MyListener`, de modo que seja alterada a cor do vértice na interface. O vértice é armazenado na fila e então, enquanto a fila não estiver vazia, o vértice inicial da mesma é atribuído à variável `u` e retirado da fila. A seguir, cada vértice branco na lista de adjacência de `u` é inserido na fila, recebe cor cinza, `u` é definido como seu predecessor e a distância `d` é definida como a distância entre `u` e o vértice inicial acrescida de 1. O método `notifyListener` é novamente chamado e então, quando todos os vértice na lista de adjacência de `u` forem percorridos, `u` recebe preto como cor e `notifyListener` é novamente chamado. A implementação do método `execute`, responsável pela atualização das cores de vértices e arestas na interface, recebe por parâmetro dois objetos da classe `Vertex` e um número inteiro que define sua cor. Caso os vértices passados sejam o mesmo, é recuperado o vértice da interface correspondente ao vértice passado e sua cor é alterada. Caso sejam diferentes, a aresta existente entre eles e o segundo vértice passado por parâmetro têm suas cores alteradas. Esse método encontra-se no

quadro 18.

```
class MyListenerImpl: public MyListener {
      void execute(GraphObj::Vertex *vi, GraphObj::Vertex *vj, int color) {
            if(vi == vj){
                  VerticeInterf *v = graphInterf.getVertex(vi->getId());
                  v->setColor(color);
            }else{
                  VerticeInterf *v1 = graphInterf.getVertex(vi->getId());
                  VerticeInterf *v2 = graphInterf.getVertex(vj->getId());
                  EdgeInterf *e = v1->getEdgeInterf(v2);
                  v2->setColor(color);
                  e->setColor(color);
            }
            display();
            Sleep(1000);
      }
};
```

Quadro 18 – Método `execute` implementado na classe `MyListenerImpl`

Ao término do método `run`, é retornado o caminho realizado pelo algoritmo em um objeto da classe `Path`. É chamado por fim o método `print` existente na classe `Path`, o qual imprime o caminho entre os vértices selecionados pelo usuário e notifica a interface para que o caminho seja apresentado, em vermelho, ao usuário.

Além do BFS, o usuário pode optar por executar o algoritmo DFS. Caso o faça, o método `algDFS` no arquivo "main.cpp" é chamado. O usuário escolhe dois vértices para que seja apresentado o caminho entre os mesmos após a execução do algoritmo. Esse método age de forma semelhante ao método `algBFS`, de modo que inicialmente é passado, ao construtor da classe `AlgDFS`, o grafo ou dígrafo no qual a busca é executada. Essa classe possui uma estrutura chamada `AttribDFS`, a qual possui uma variável de referência para o vértice antecessor na busca, uma cor e as variáveis `d`, que representa o tempo lógico de abertura do vértice, e `f`, que guarda o tempo lógico de fechamento do vértice. Os tempos de abertura e fechamento equivalem aos momentos em que o vértice se torna cinza e preto respectivamente. Essas variáveis recebem o valor da variável `time` que é atualizada durante a execução do algoritmo. O método `algBFS` define a variável da classe `AlgBFS` que notifica a interface para que as mudanças necessárias sejam realizadas e chama o método `DFS` passando por parâmetro o identificador do vértice inicial da busca. Após a execução do algoritmo é chamado o método `printPath` encontrado na classe `AlgDFS` para a apresentação do caminho encontrado entre os vértices escolhidos pelo usuário. O método `DFS` encontra-se no quadro 19.

```
void AlgDFS::DFS(int ids)
{
      Vertex* v = g.getVertex(ids);
      initAttrib();
      DFSVisit(v);
}
```

Quadro 19 – Método `DFS` da classe `AlgDFS`

O código no quadro 19 recupera o vértice equivalente ao identificador passado por parâmetro e chama os métodos `initAttrib` e `DFSVisit`. O primeiro é responsável por inicializar cada vértice com o antecessor e os tempos de abertura e fechamento iguais a zero e cor branca. À variável `time` também é atribuído o valor zero. O segundo é responsável pela execução do algoritmo e é apresentado no quadro 20.

```
void AlgDFS::DFSVisit(Vertex* v)
{
      attrib[v]->color = GRAY;
      notifyListener(v, v, 1);
      time++;
      attrib[v]->d = time;

  list<Vertex*> adjV = v->getAdjacencyList();
  list<Vertex*>::iterator i;
  Vertex* u;
  for (i = adjV.begin(); i != adjV.end(); i++) {
      u = (*i);
      if (attrib[u]->color == WHITE) {
            attrib[u]->pi = v;
            notifyListener(v, u, 1);
            DFSVisit(u);
      }
  }
  attrib[v]->color = BLACK;
  notifyListener(v, v, 2);
  time++;
  attrib[v]->f = time;
}
```

Quadro 20 – Método `DFSVisit`

O método `DFSVisit` inicia atribuindo cor cinza ao vértice inicial, notificando a interface sobre essa mudança e incrementando a variável `time` para atribuir seu valor ao tempo de abertura do vértice inicial. Então, para cada vértice branco adjacente ao vértice atual, o vértice atual é definido como seu predecessor, a interface é notificada para apresentar em cores diferentes o vértice visitado e o método `DFSVisit` é chamado passando o vértice adjacente como parâmetro. Quando não possuir mais vértices adjacentes a serem visitados, o vértice atual recebe cor negra e a interface é notificada para redesenhar o vértice. Por fim, a variável `time` é novamente incrementada e seu valor, atribuído ao tempo de fechamento do vértice.

O usuário pode ainda optar por executar o algoritmo de Kruskal sobre o grafo. Ao fazê-lo, o método `algKruskal` é chamado e verifica, a princípio, se foi criado um grafo ou um

dígrafo. Kruskal só pode ser executado sobre um grafo não dirigido de modo que se for um dígrafo, o algoritmo não será executado. Se for um grafo, um objeto da classe `AlgMSTKruskal` é instanciado recebendo o grafo por parâmetro e a variável `listener` encontrada nessa classe é inicializada. É então chamado o método `run`, responsável por executar o algoritmo sobre o grafo, o qual é apresentado no quadro 21.

```
Graph AlgMSTKruskal::run() {
  Graph msTree;
  KruskalSet kruskalSet;

  list<Vertex*>lvert = g.getVertexList();
  list<Vertex*>::const_iterator i;
  Vertex* u;
  Vertex* v;
  for (i = lvert.begin(); i!=lvert.end(); i++) {
    v = *i;
    kruskalSet.MakeSet(v);
    msTree.createVertex(v->getId());
  }

  set<Edge*>edges_set = g.getEdgeList();
  set<Edge*>::const_iterator j;
  vector<Edge*>edgeSortedList;

  for (j = edges_set.begin(); j != edges_set.end(); j++) {
    edgeSortedList.push_back(*j);
  }

  sort(edgeSortedList.begin(), edgeSortedList.end(),
MinEdgeWeightOrdenator());

  vector<Edge*>::const_iterator k;
  Edge* edge;
  for (k = edgeSortedList.begin(); k != edgeSortedList.end(); k++) {
    edge = (*k);
    u = edge->getVi();
      if (k == edgeSortedList.begin()){
            notifyListener(u, u, 1);
      }
    v = edge->getVj();
      notifyListener(u, v, 1);
    if (kruskalSet.FindSet(u) != kruskalSet.FindSet(v)) {
      msTree.createEdge(u->getId(),v->getId(),edge->getWeight());
      kruskalSet.Union(u, v);
    }
  }
  return msTree;
}
```

Quadro 21 – Método `run` da classe `AlgMSTKruskal`

No código apresentado no quadro 21, inicialmente é criado, para cada vértice, um conjunto contendo apenas o vértice em questão. Cada vértice é criado no grafo da variável `msTree`. A seguir é criada uma lista de arestas, que recebe as arestas do grafo passado por parâmetro no construtor da classe, e a mesma é ordenada em ordem crescente de custo. Cada aresta nessa lista é verificada sendo que, caso os vértices que a compõem não façam parte do

mesmo conjunto, a aresta verificada é criada no grafo da variável `msTree` e os conjuntos dos vértices que a compõem são unidos em um só. A interface é notificada a cada aresta verificada. Por fim, o grafo de `msTree` é retornado.

Após o retorno de `msTree`, é chamado o método `setKruskalTreeColor` que recebe o grafo retornado por parâmetro. Esse método é responsável por mudar a cor da árvore resultante do algoritmo de Kruskal. Para isso, cada vértice e aresta da árvore são verificados e têm sua cor mudada para vermelho. O grafo é então redesenhado permitindo ao usuário sua visualização da árvore. O quadro 22 apresenta o código fonte de `setKruskalTreeColor`.

```
void setKruskalTreeColor(const Graph& arvore){
      VerticeInterf *vi;
      VerticeInterf *vj;
      EdgeInterf *e;
      list<Vertex*> adjVi;

      list<Vertex*> lvert = arvore.getVertexList();
      list<Vertex*>::iterator i;
      for(i = lvert.begin(); i != lvert.end(); i++){
            vi = graphInterf.getVertex((*i)->getId());
            adjVi = (*i)->getAdjacencyList();
            list<Vertex*>::iterator j;
            for(j = adjVi.begin(); j != adjVi.end(); j++){
                  vj = graphInterf.getVertex((*j)->getId());
                  vi->setColor(3);
                  vj->setColor(3);
                  e = vi->getEdgeInterf(vj);
                  e->setColor(3);
            }//for
      }//for
      display();
}
```

Quadro 22 – Método `setKruskalTreeColor`

### 3.3.3 Operacionalidade da implementação

Ao executar o programa, o usuário depara-se com a tela apresentada na figura 17. Nesse momento, a única opção permitida ao usuário é a escolha entre a criação de um grafo ou um dígrafo. Como é possível notar na figura 17, todas as outras opções encontram-se desabilitadas.

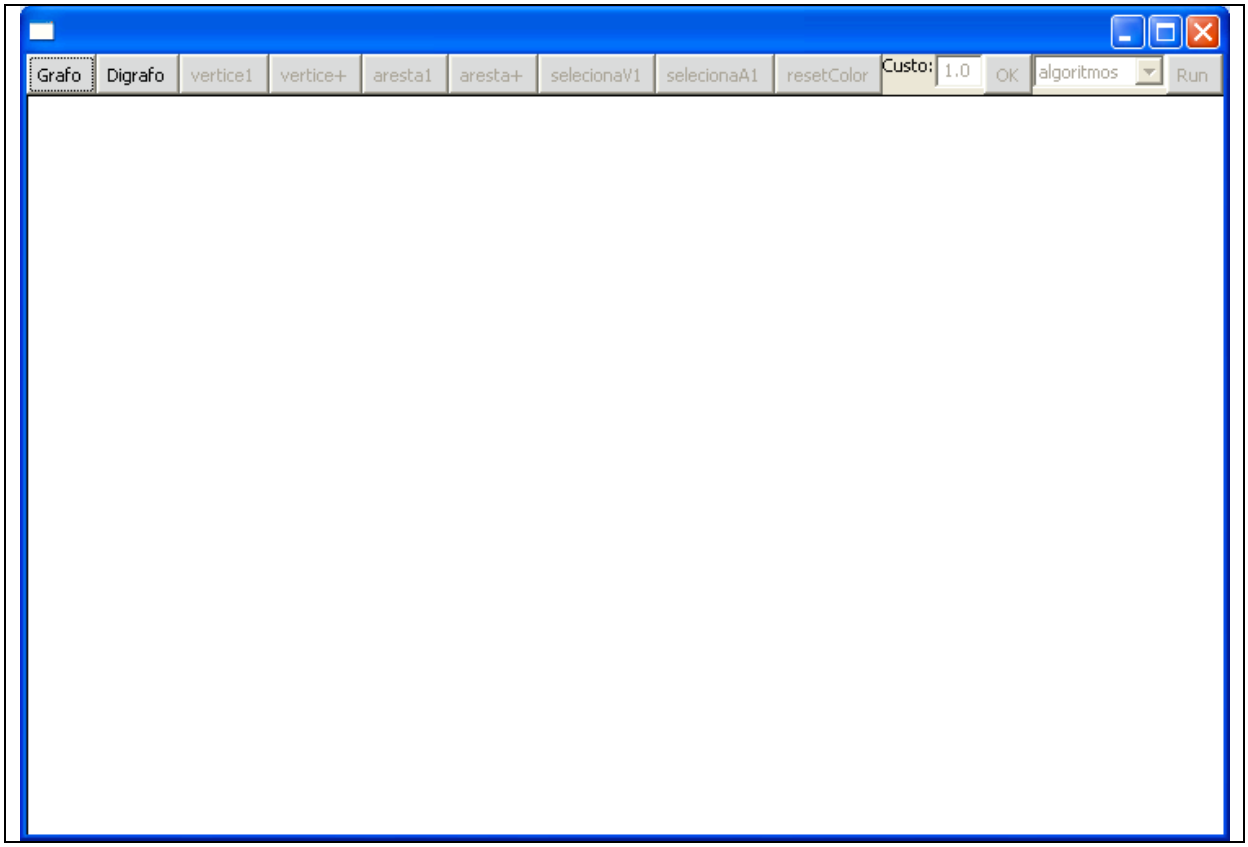



Figura 17 – Tela inicial do programa

Ao selecionar o tipo de grafo que deseja construir, os botões para criação e seleção de vértices e arestas são habilitados, enquanto os botões para seleção do tipo de grafo são desabilitados. Nesse momento o usuário pode apenas criar vértices porque mesmo que selecione as outras opções, as mesmas não podem ser executadas pelo fato de não haver nada criado. Portanto o usuário deve pressionar o botão “vertice1” ou “vértice+”. O botão “vértice1” permite a criação de apenas um único vértice, enquanto que o botão “vértice+” permite que o usuário crie quantos vértices desejar, até que selecione outra opção. O usuário deve então selecionar, com o *mouse*, o local onde o vértice deve ser desenhado. A criação de vértices é apresentada na figura 18.

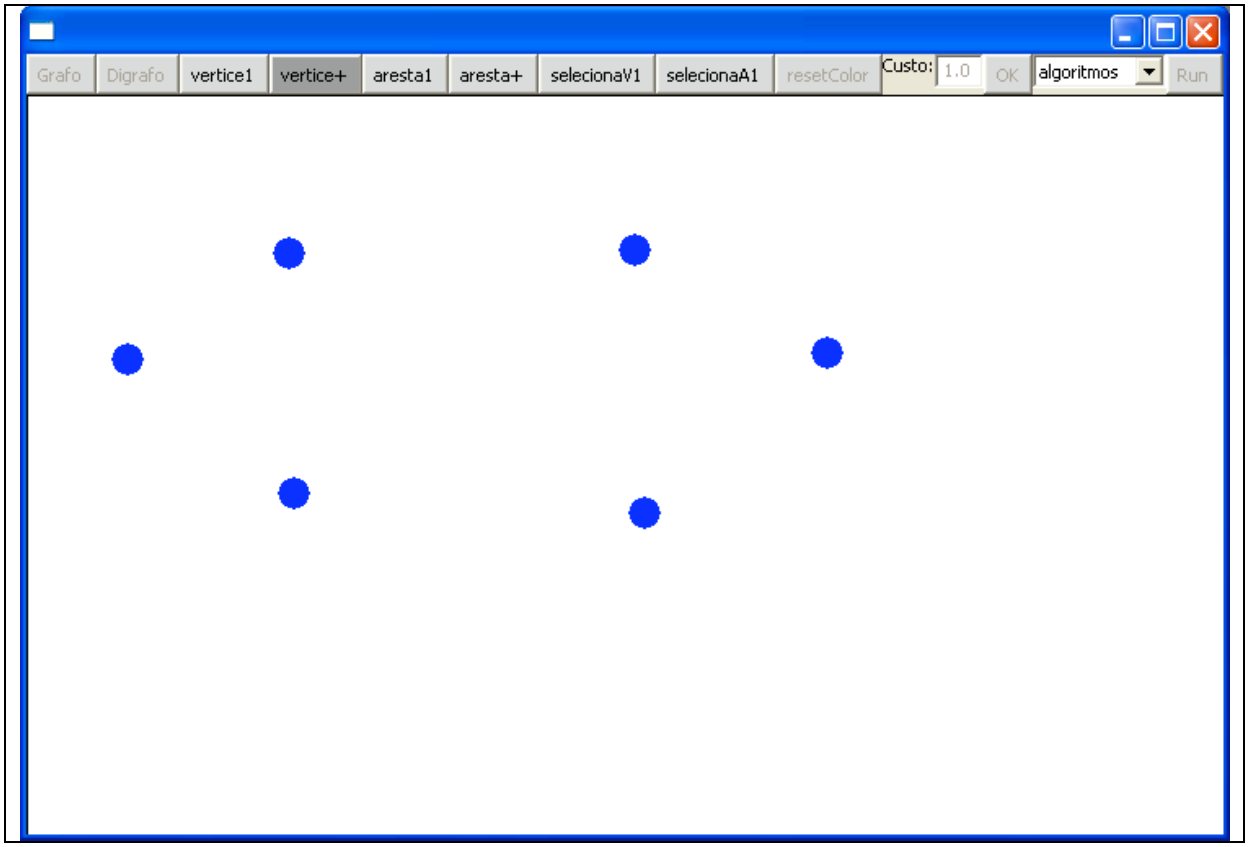



Figura 18 – Função de criação de vértices

Após a criação dos vértices, o usuário pode optar por criar arestas. Do mesmo modo que acontece com os vértices, o usuário pode escolher criar apenas uma aresta, pressionando o botão ”aresta1”, ou quantas arestas desejar até que selecione outra opção, pressionando o botão “aresta+”. Ao criar uma aresta, a mesma encontra-se selecionada, permitindo ao usuário a atribuição de um custo à aresta. Para isso, no momento em que uma aresta encontra-se selecionada, a área de texto para inserção do custo é habilitada. Ao modificar-se o valor encontrado nesse campo de texto, o botão “OK” é habilitado, permitindo ao usuário redefinir o custo da aresta. O custo inicial das arestas é igual a um. A figura 19 apresenta a criação de arestas no grafo juntamente com a definição de um custo para uma delas.

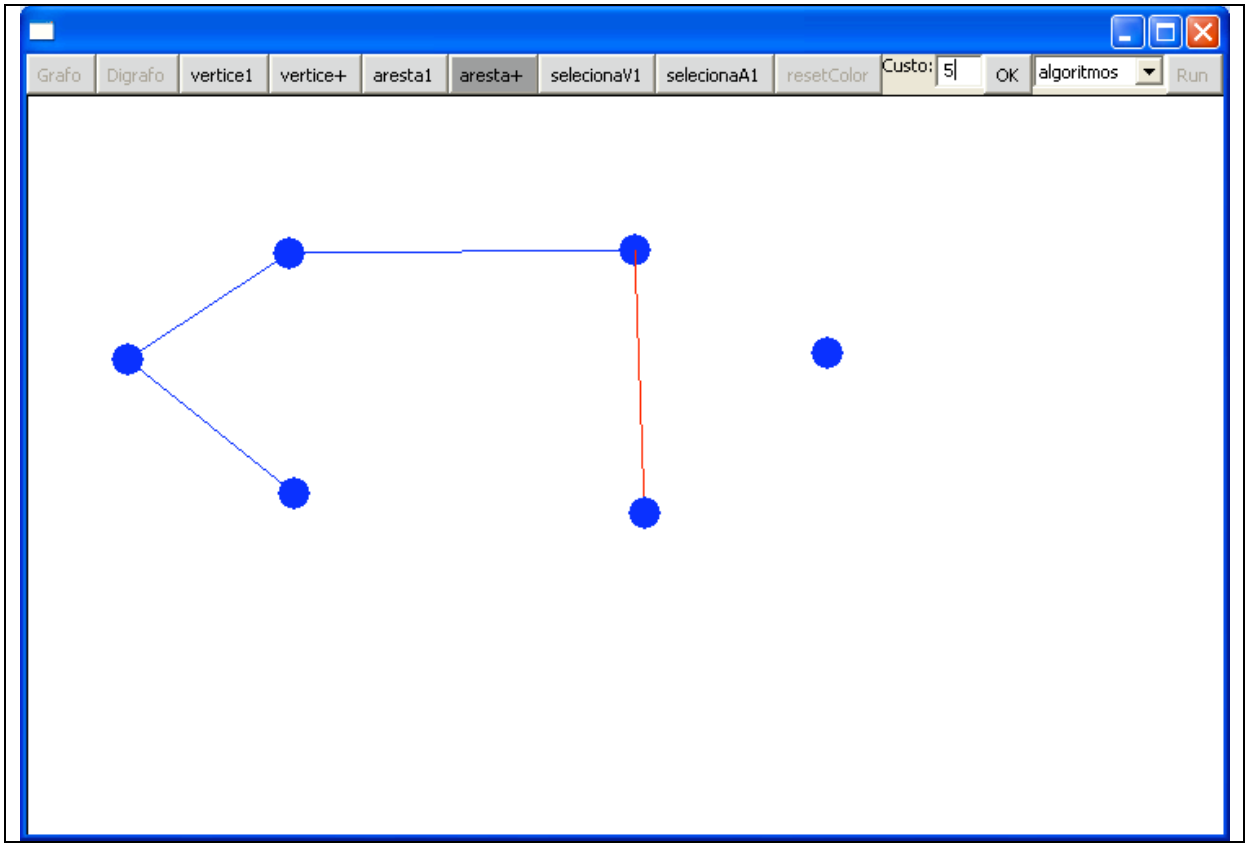



Figura 19 – Função de criação de arestas e definição de novo custo

Quando uma aresta está selecionada, é possível removê-la do grafo. O mesmo pode ser feito ao selecionar-se um vértice, no entanto, a remoção da aresta apenas remove a aresta selecionada do grafo enquanto que, ao remover um vértice, o mesmo é removido juntamente com as arestas conectadas a ele. Para remover uma aresta ou um vértice do grafo, o usuário deve pressionar o botão “Delete” do teclado quando algum deles estiver selecionado. As figuras 20 e 21 demonstram a seleção e remoção de uma aresta respectivamente, enquanto que as figuras 22 e 23 apresentam a seleção e remoção de um vértice.

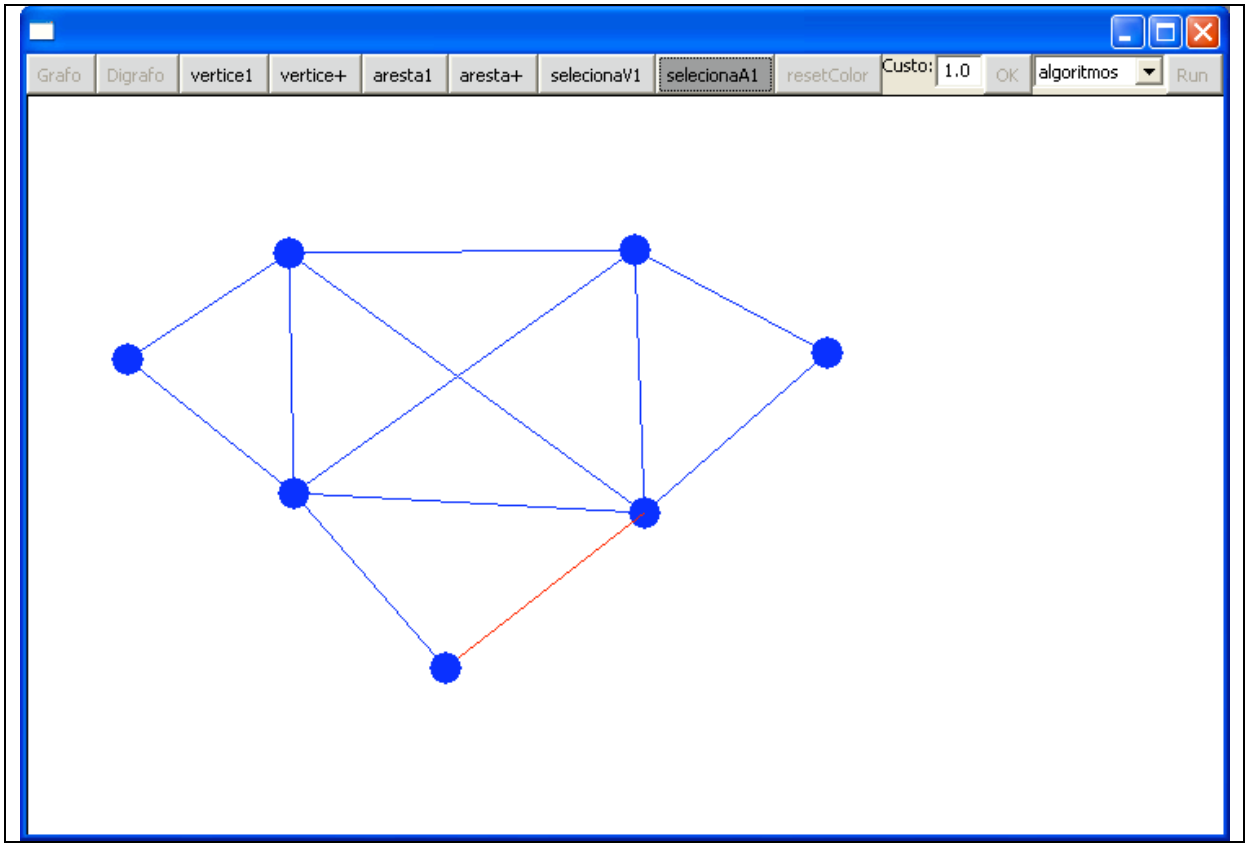



Figura 20 – Aresta selecionada

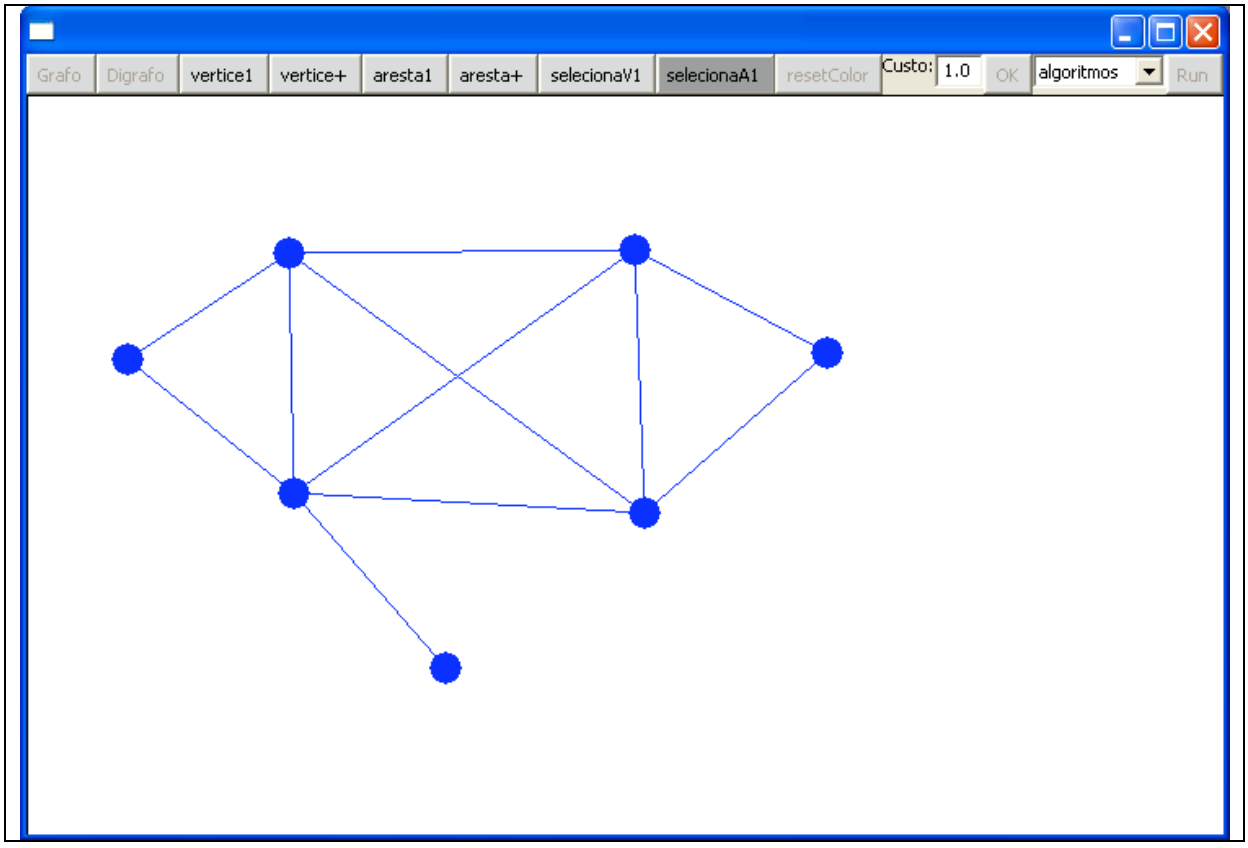



Figura 21 – Aresta excluída

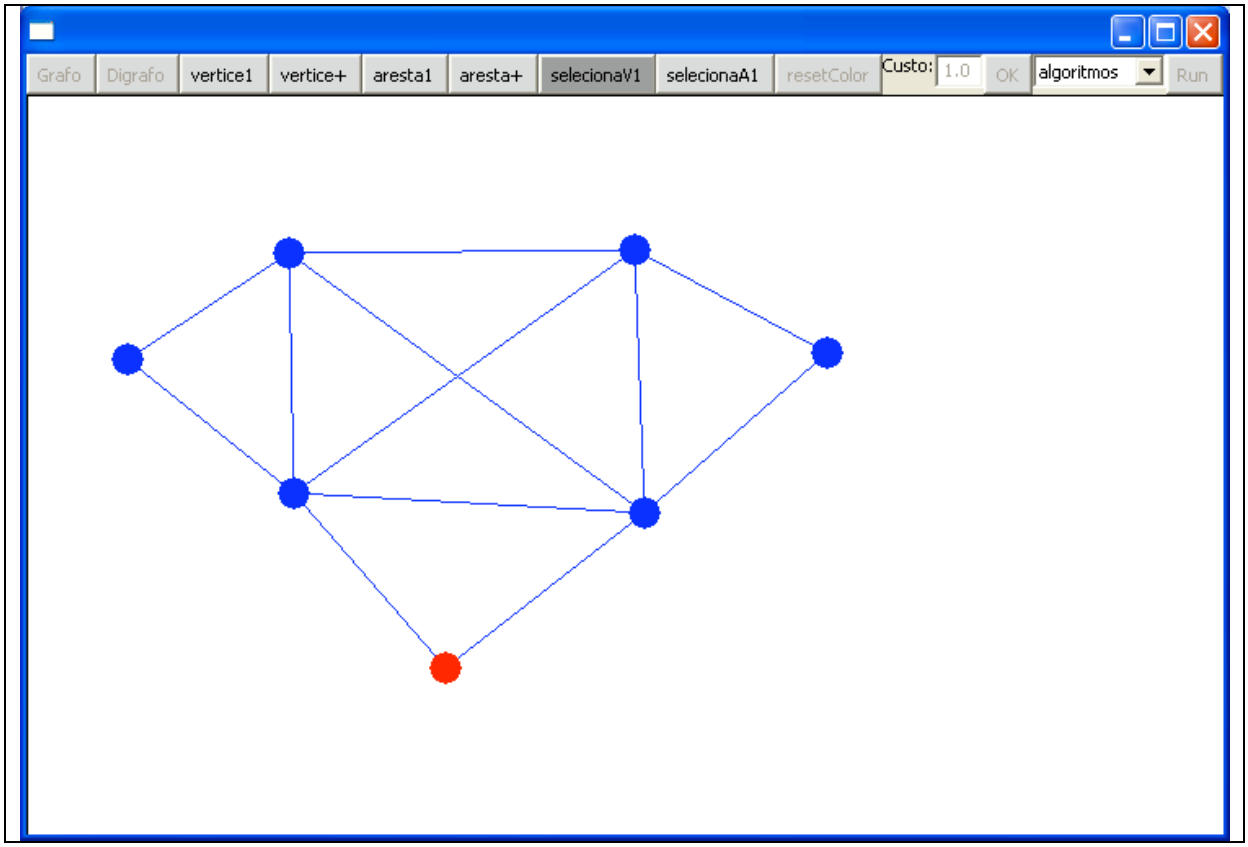



Figura 22 – Seleção de um vértice

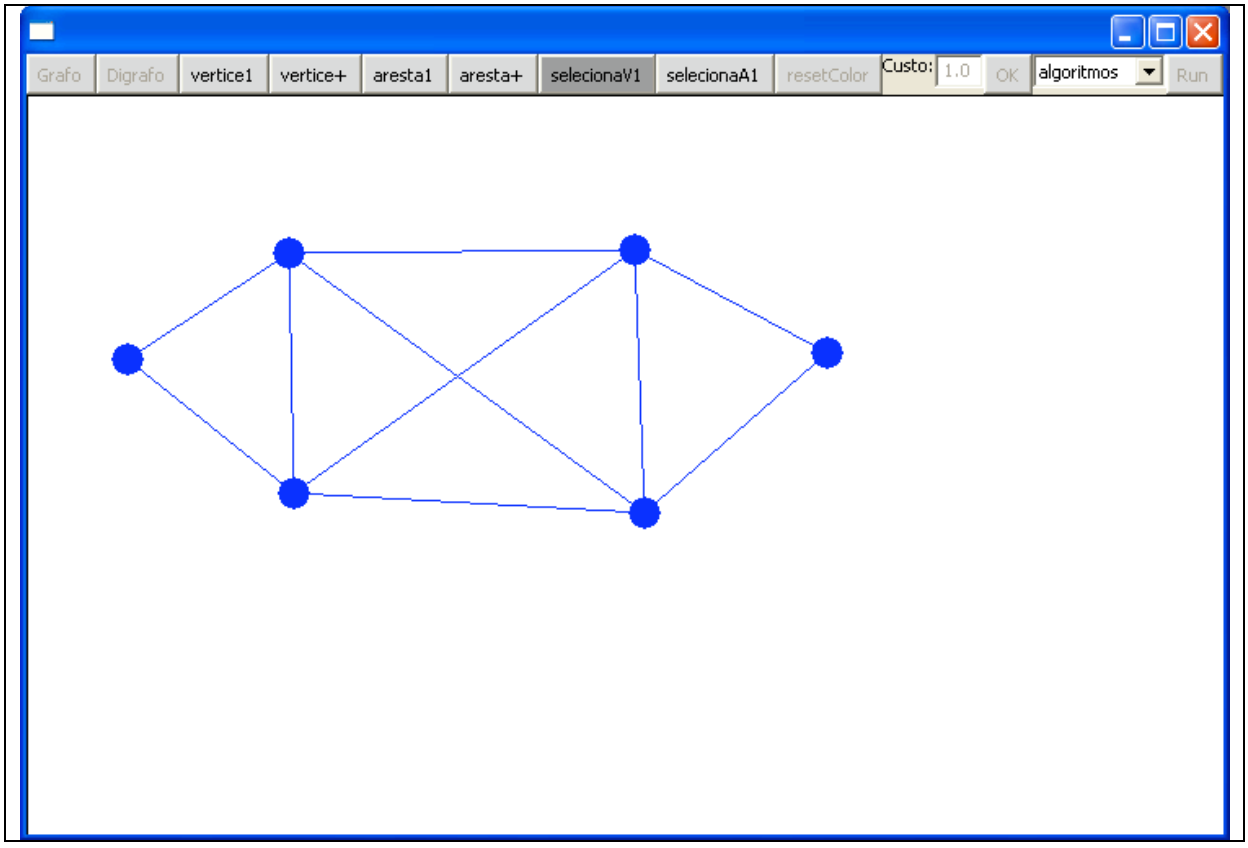



Figura 23 – Vértice excluído

Após a construção do grafo desejado, o usuário pode selecionar, em uma lista, um

algoritmo a ser executado sobre o grafo. A lista expandida é apresentada na figura 24. Após a escolha do algoritmo o usuário deve pressionar o botão “Run” encontrado ao lado da lista, o qual é habilitado após a seleção do algoritmo.

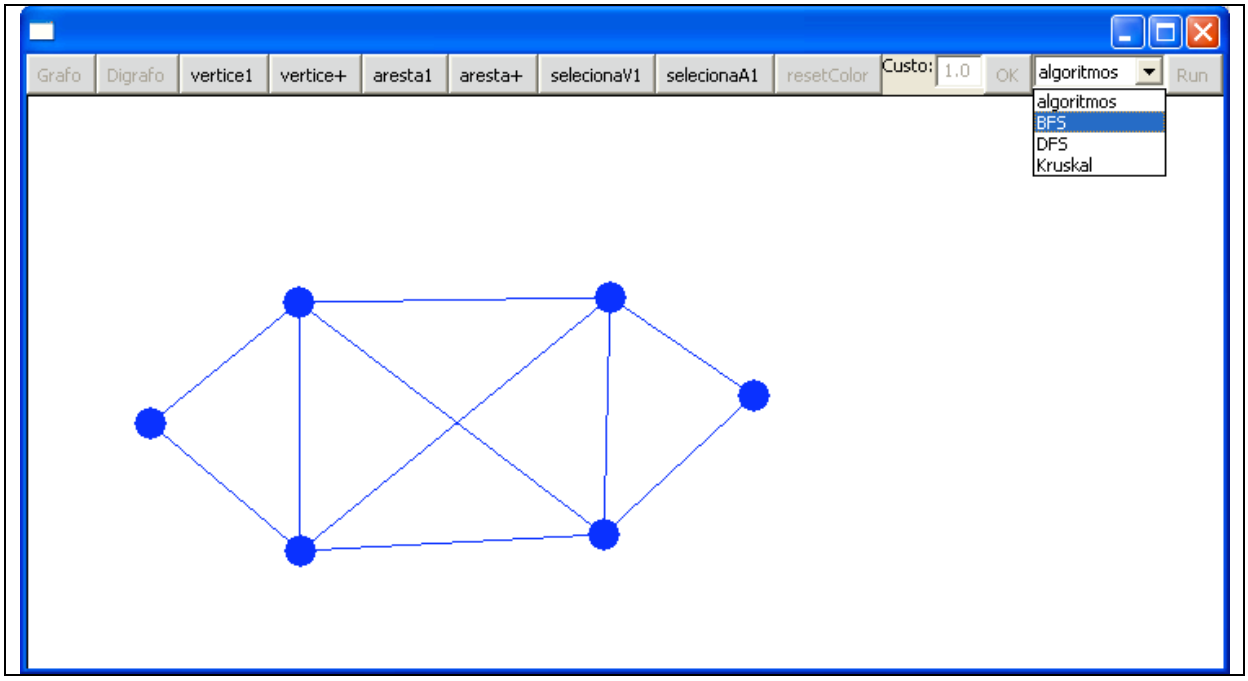



Figura 24 – Lista de algoritmos expandida

Como explicado anteriormente, caso selecione os algoritmos BFS ou DFS, antes que ocorra a execução do algoritmo o usuário deve escolher dois vértices entre os quais será encontrado o caminho. Caso escolha o algoritmo de Kruskal, o mesmo começará a ser executado imediatamente e, então, a árvore geradora de custo mínimo encontrada é apresentada no grafo.

Embora branco, cinza e preto sejam as cores definidas na teoria dos grafos para a representação do caminhamento de algoritmos sobre os mesmos, a fim de obter melhor visualização foi utilizada a cor azul para representar o momento em que vértices e arestas ainda não foram visitados, substituindo o branco. Cinza e preto são utilizados conforme o que está definido na teoria dos grafos. Por fim, o vermelho foi utilizado para apresentar o resultado final obtido pela execução dos algoritmos.

A figura 25 apresenta o caminho encontrado entre os vértices selecionados pelo usuário, após a aplicação da busca BFS sobre o grafo.

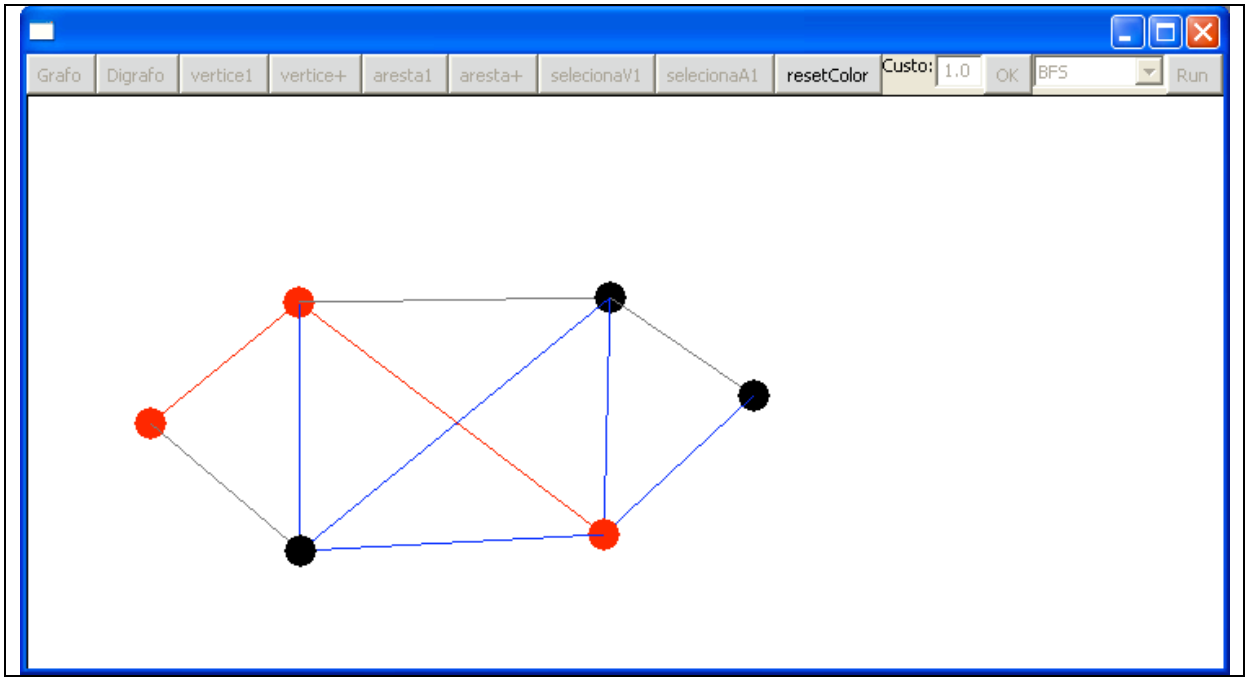



Figura 25 – Resultado da busca BFS

Após a execução de um algoritmo, a única opção que o usuário pode optar por fazer é pressionar o botão “resetColor”. Ao pressioná-lo, é chamado um método responsável por retornar o grafo à sua cor original. Em seguida o usuário pode optar por modificar o grafo, podendo adicionar ou remover vértices e arestas, ou executar, novamente, um algoritmo sobre o grafo. Assim como acontece com BFS, caso seja executado DFS ou Kruskal, após a apresentação de seu resultado ao usuário apenas o botão “resetColor” encontra-se habilitado. O resultado do algoritmo DFS é apresentado na figura 26.

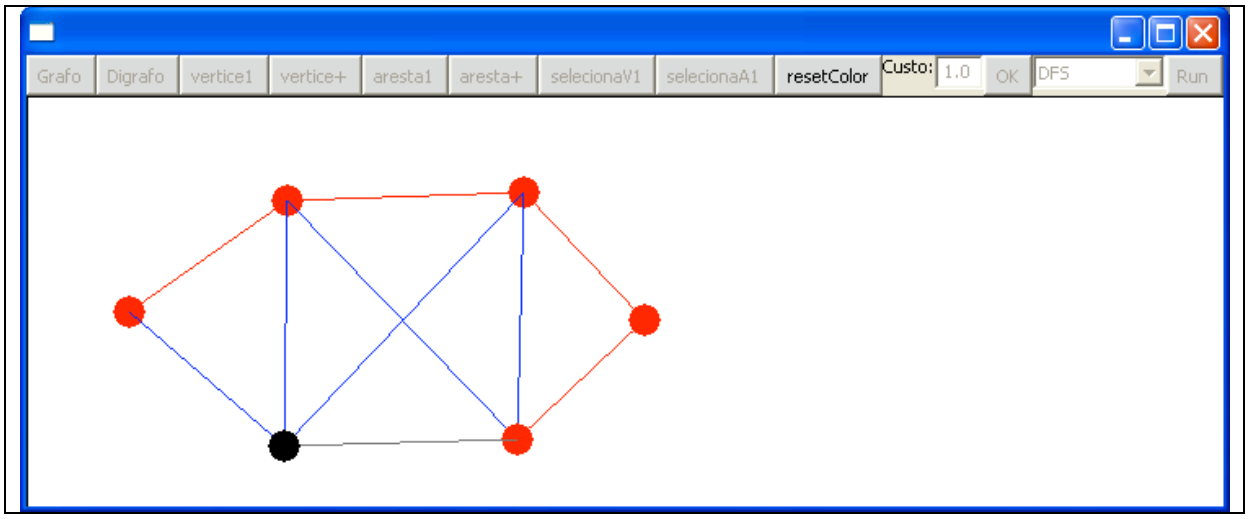



Figura 26 – Resultado da busca DFS

A diferença entre a execução dos algoritmos BFS e DFS no grafo é que BFS procura o menor caminho, em número de arestas, entre dois vértices no grafo, no entanto DFS é executado de modo diferente. Como apresentado na fundamentação teórica, DFS procura

sempre visitar um vértice vizinho, o vizinho do vizinho e assim por diante. Assim, os vértices escolhem as arestas que tentarão percorrer na ordem em que essas foram criadas no grafo, procurando percorrer todos os vértices do mesmo. Desse modo é encontrado apenas um caminho entre dois vértices no grafo, o qual muitas vezes não é o mais eficiente. Por fim é apresentado, na figura 27, o resultado da execução do algoritmo de Kruskal sobre o grafo. Como já explicado anteriormente, esse algoritmo tem a função de encontrar uma árvore de custo mínimo no grafo. Para isso é necessária a atribuição de custos às arestas do grafo.

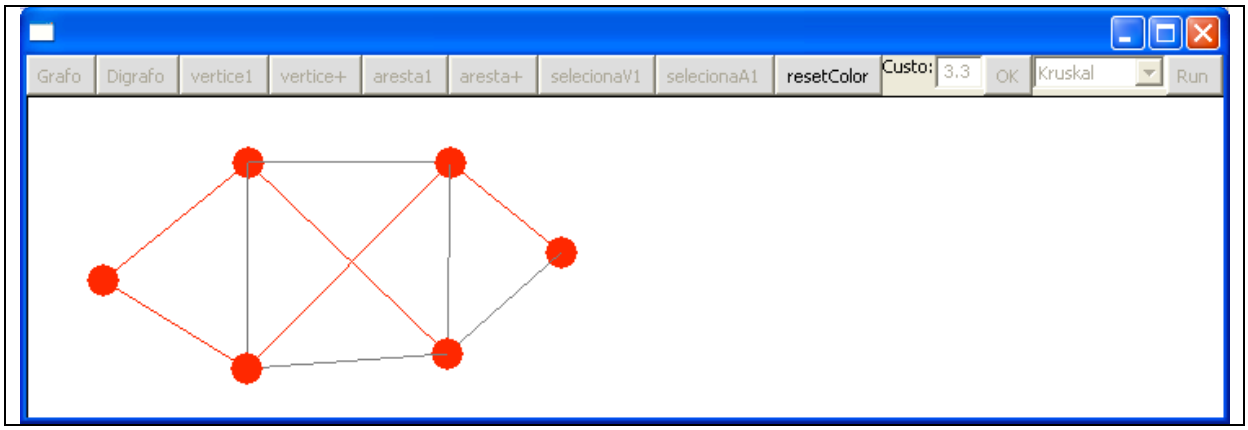



Figura 27 – Resultado da execução do algoritmo de Kruskal

## 3.4 RESULTADOS E DISCUSSÃO

Estabelecendo uma relação entre a ferramenta desenvolvida e os trabalhos correlatos descritos na fundamentação teórica, e comparando a biblioteca BGL com a ferramenta desenvolvida, podemos notar que a BGL, assim como a GraphObj, disponibiliza uma variedade de estruturas e algoritmos para o trabalho com grafos. O mesmo pode ser dito das bibliotecas VGTL e GTL. Pode-se notar ainda que todas essas bibliotecas utilizam-se dos *containers* STL, no entanto certas diferenças podem ser apontadas. Uma grande diferença está no fato de que a GraphObj, utilizada no desenvolvimento da ferramenta, até o momento não havia sido utilizada em uma aplicação. Além disso, outra diferença existente entre a ferramenta desenvolvida e as bibliotecas dos trabalhos correlatos, é que elas possuem as estruturas necessárias para a construção de grafos e execução de algoritmos, mas não possuem uma interface gráfica. Por outro lado, a ferramenta utiliza-se da GraphObj para sua camada lógica, além de permitir a interação do usuário com o grafo abstraindo as operações realizadas pela biblioteca.

A biblioteca JGraph, por outro lado, possui inúmeros recursos de interface. Ela fornece vários recursos de interface para a criação de grafos, fluxogramas e diagramas, ao contrário da ferramenta desenvolvida que permite apenas a construção de grafos. Apesar de possuir inúmeros recursos de interface, a JGraph não possui uma camada lógica implementada. Assim, a implementação de algoritmos que trabalhem com a interface fica sob responsabilidade do programador. Portanto, todos os trabalhos correlatos possuem, ou a estrutura necessária para a construção de grafos e aplicação de algoritmos sobre os mesmos, ou uma interface que permite apenas o desenho de grafos. A ferramenta desenvolvida, no entanto, realiza a integração das duas funcionalidades, permitindo ao usuário a criação de grafos e a visualização da aplicação de algoritmos sobre os mesmos. O quadro 23 apresenta, resumidamente, as diferenças existentes entre a ferramenta e os trabalhos correlatos.

| | Interface | Lógica | Linguagem |
|---|---|---|---|
| BGL | Não | Sim | C++ |
| VGTL | Não | Sim | C++ |
| GTL | Não | Sim | C++ |
| Jgraph | Sim | Não | Java |
| Ferramenta | Sim | Sim | C++ |

Quadro 23 - Diferenças entre os trabalhos correlatos e a ferramenta desenvolvida

Essa ferramenta permite ao usuário a criação de um grafo de forma interativa. O usuário pode criar os vértices selecionando as posições, na tela, onde quer que os mesmos sejam posicionados e criar arestas, apenas selecionando os vértices entre os quais deseja criá-la. Desse modo, o usuário consegue criar o grafo da forma que acreditar ser melhor visualizá-lo. Pode ainda executar um algoritmo sobre o mesmo, visualizando sua execução passo a passo seguida de seu resultado.

Seu desenvolvimento foi realizado utilizando o modelo MVC. Ao programar utilizando o modelo MVC, nota-se que o código encontra-se mais organizado, de modo a permitir ao programador um melhor entendimento do código, e facilitando a realização de alterações na programação. O uso de padrões de projetos é importante, pois trazem consigo um conjunto de soluções a problemas que muitas vezes podemos encontrar durante a programação de um aplicativo. Durante a programação dessa ferramenta foi encontrado um problema, em que não se sabia como informar a camada de interface da aplicação sobre as mudanças a serem realizadas no grafo durante o algoritmo. Surgiu a idéia de guardar as mudanças em uma estrutura e passá-las a interface, no entanto essa solução prejudicava o conceito do modelo MVC, já que a camada lógica conversaria diretamente com a camada de visualização. Para resolver esse problema foi utilizado o padrão Command, que utiliza uma

interface lógica para intermediar a comunicação entre as duas camadas. Pode-se então concluir que, ao utilizar padrões de projeto durante o desenvolvimento de uma aplicação, normalmente será utilizado mais que um padrão, mesmo porque certos padrões podem auxiliar uns aos outros.

# 4 CONCLUSÕES

Diferentes problemas encontrados no cotidiano podem ser modelados na forma de grafo, a fim de providenciar uma melhor visualização do problema para então encontrar sua solução. Alguns problemas podem ser resolvidos por algoritmos já conhecidos na teoria dos grafos. Mesmo conhecendo a teoria, muitas vezes é difícil visualizar o grafo e a solução do problema, às vezes pelo tamanho do grafo ser muito grande. Outro momento em que a visualização do grafo pode ser difícil é no aprendizado de sua teoria. Por esse motivo, mostrou-se interessante o desenvolvimento de uma ferramenta que permitisse ao usuário a construção de um grafo e a aplicação de algoritmos sobre o mesmo.

Essa ferramenta permite ao usuário a criação de um grafo de forma interativa. Assim, o usuário pode criá-lo da maneira que preferir visualizá-lo, e ainda verificar a execução de um algoritmo sobre o mesmo. Seu desenvolvimento foi realizado utilizando o modelo MVC. A utilização de padrões de projeto permite maior organização do código facilitando seu entendimento e alterações que venham a ser realizadas. Normalmente se utiliza mais de um padrão durante o desenvolvimento de um aplicativo, mesmo porque muitas vezes os padrões complementam uns aos outros. No caso da ferramenta desenvolvida, além do padrão MVC, foi utilizado o padrão Command, o qual permite que a comunicação entre as camadas de visualização e lógica seja realizada sem prejudicar o conceito do modelo MVC.

Foi utilizada a linguagem C++ para o desenvolvimento da ferramenta, possibilitando o uso da biblioteca GraphObj, a qual também foi desenvolvida em C++. Foram utilizados, para o desenho dos vértices e arestas do grafo, alguns métodos disponibilizados pela OpenGL. Por fim, foi utilizada a biblioteca IUP em conjunto com a linguagem LED, para a construção do diálogo, com seus botões, e da área para desenho do grafo de maneira simples.

## 4.1 EXTENSÕES

Algumas complementações podem ser feitas para dar continuidade a esse trabalho. São:

a) realizar melhorias na interface como, por exemplo, a apresentação dos custos sobre as arestas do grafo e o desenho de setas no dígrafo para mostrar o sentido

das arestas;

b) permitir ao usuário maior interação com o grafo criando, por exemplo, funções para movimentação dos vértices criados, zoom;
c) permitir ao usuário mudar as cores do grafo, assim como as cores utilizadas para a apresentação da execução e resultado dos algoritmos;
d) integrar outros algoritmos a serem aplicados sobre o grafo, os quais podem ser encontrados na teoria dos grafos;
e) incluir uma opção para gerar o reposicionamento automático do grafo;
f) apresentar os laços criados nos grafos na ferramenta;
g) permitir ao usuário salvar e carregar os grafos criados;
h) permitir que o usuário pare a execução do algoritmo quando desejar, para que o mesmo possa visualizar melhor a execução do algoritmo.

# REFERÊNCIAS BIBLIOGRÁFICAS


BENSON, D. **Jgraph and Jgraph Layout Pro**: user manual. Northampton, 2007. Disponível em: <http://www.jgraph.com/pub/jgraphmanual.pdf>. Acesso em: 21 set. 2007.

CHARTRAND, G. **Introductory graph theory.** New York: Dover, 1985.

CORMEN, T. H. et al. **Algoritmos**: teoria e prática. 2. ed. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2002.

DAVIS, J. M. **Java graph visualization and layout**. [S.l.], 2005. Disponível em: <http://jgraph.co.uk/images/bettysbrain.gif>. Acesso em: 8 jun. 2008.

FOWLER, M. **Patterns of enterprise application architecture.** Boston: Addison-Wesley, 2003.

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

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

_____. **Diagrama basegraph** [mensagem pessoal]. Mensagem recebida por <zeppelin@inf.furb.br> em 26 maio 2008.

LEVY, C. H. **IUP/LED**: uma ferramenta portátil de interface com usuário. 1993. 63 f. Dissertação (Mestrado em Informática) – Curso de Pós-Graduação em Ciência da Computação, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro. Disponível em: <http://www.tecgraf.puc-rio.br/iup/download/levy93.pdf>. Acesso em: 5 maio 2008.

PREE, W. **Design patterns for object-oriented software development.** Wokingham: Addison-Wesley, 1994.

RAITNER, M. **[GTL] The Graph Template Library.** Passau, 2008. Disponível em: <http://www.infosun.fim.uni-passau.de/GTL/index.html>. Acesso em: 8 jun. 2008.

SCHICHL, H. **VGTL (Vienna Graph Template Library)**: reference manual. Viena, 2003. Disponível em: <http://citeseer.ist.psu.edu/649000.html>. Acesso em: 18 set. 2007.

SIEK, J.; LEE, L.; LUMSDAINE, A. **The Boost Graph Library**. [S.l.], 2008. Disponível em: <http://www.boost.org/libs/graph/doc/index.html >. Acesso em: 12 maio 2008.

SILICON GRAPHICS. **Introduction to the Standard Template Library.** [S.l.], 2006. Disponível em: <http://www.sgi.com/tech/stl/stl_introduction.html>. Acesso em: 18 set. 2007.