**Relatório do Problema 4: “Software para empresa de segurança”**

**Valmir Vinícius de Almeida Santos**

Engenharia de Computação – Universidade Estadual de Feira de Santana  
Feira de Santana, 14 de maio de 2016

vvalmeida96@gmail.com

**1. Introdução**

O transporte de altas quantias monetárias é uma operação logística que possui elevado fator de risco associado, qualquer tipo de falha nesse processo pode resultar na subtração de dezenas e centenas de milhares de reais. Por isso, a segurança e a agilidade devem ser parâmetros fundamentais para guiar a execução dessa atividade. Esses e outros fatores fazem com que as empresas disponibilizadoras desse tipo de serviço invistam, constantemente, em pesquisa, buscando desenvolver técnicas que potencializem os níveis de confiabilidade ofertado.

Esse relatório visa detalhar o processo de modelagem e concepção do "*ForteSeguro*", um software responsável por encontrar a rota mais rápida para carros fortes que saem do estacionamento de uma empresa, passam por um ponto de coleta e vão em direção a um banco, onde o valor recolhido será depositado. O executável implementado pode ser executado em navegadores *web* e possui, ainda, uma interface gráfica, na qual o usuário realiza todas as interações, como o cadastrado e remoção de pontos de coleta e caminhos, além do cálculo da trajetória mínima.

A aplicação foi desenvolvida por meio da linguagem de programação Java, seguindo os fundamentos do paradigma de programação orientado a objetos e aplicando os preceitos do padrão de projeto *MVC* (*Model-View-Controller*. Para representar computacionalmente elementos intrínsecos ao domínio do problema, a estrutura de dados conhecida como grafo ponderado foi empregada. O cálculo de menores caminhos é realizado através de uma versão modificada da técnica proposta por *Dijkstra.* Por fim, para promover a usabilidade do software, a interface gráfica foi desenvolvida por meio de uma *GUI Builder,* e uma versão em *applet* do executável foi gerada.

**2. Fundamentação teórica**

**2.1. Estrutura de dados**

**2.1.1. Grafos**

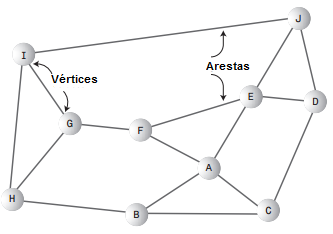
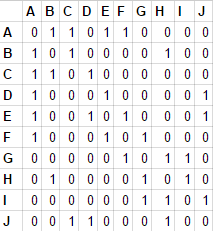
No escopo das ciências da computação, um grafo é um mecanismo empregado na estruturação de informações em memória computacional, ou seja, é uma de estrutura de dados. Através dessa estrutura, é possível representar conexões entre pares de entidades que, de alguma forma, relacionam-se. Uma gama de situações cotidianas podem ser representadas em grafos, as mais comuns na literatura envolvem rotas de trânsito numa cidade e linhas aéreas, por exemplo (LAFORE, 2004).

Um grafo consiste, basicamente, de dois conjuntos finitos: um de vértices e outro de arestas. As arestas são elementos que associam dois vértices e, dessa forma, estabelecem uma interligação entre eles. Caso a direção desse relacionamento seja levada em consideração, define-se a aresta como dirigida, caso contrário, ela é denominada de não dirigida. Os vértices que compõem as extremidades de uma mesma aresta são conhecidos como adjacentes (LAFORE, 2004).

Existem diversas abordagens para a modelagem computacional de um grafo, essas técnicas estão baseadas no emprego de estruturas de dados auxiliares, como vetores e listas encadeadas. Nesse cenário, a forma de representação mais difundida é conhecida como “matriz de adjacência” e consiste, basicamente, em um vetor bidimensional no qual estão os elementos indicam se há uma aresta entre dois vértices. Além desse, um outro tipo de caracterização é conhecido como “lista de adjacências”, que é uma lista encadeada de listas encadeadas, sendo cada lista individual responsável por armazenar a quais vértices um dado vértice é adjacente (LAFORE, 2004).

Na figura 1 há a representação de um grafo e da sua respectiva matriz de adjacências, na qual os vértices são usados como cabeçalho, tanto para as linhas quanto para as colunas e a existência de uma aresta entre dois vértices é representada com um algarismo 1, enquanto a ausência é indicada por 0.

**Figura 1. Representação de um grafo e da sua matriz de adjacência**

**FONTE: (DEITEL, p. 564, 2004)**

**2.1.2. Grafos ponderados**

Além das suas extremidades e da direção, no caso dos grafos dirigidos, é possível atribuir outro atributos às arestas de um grafo: o peso. Essa propriedade representa um dado quantitativo associado a conexão realizada pela aresta. Em um dado grafo, caso as arestas representem, por exemplo, uma estrada entre duas cidades, o peso pode ser utilizado para caracterizar o tempo necessário para transitar entre elas ou, ainda, o preço a ser pago em um pedágio existente no caminho (LAFORE, 2004).

Grafos que possuem peso associado a cada uma das suas arestas são denominados de ponderados ou valorados. A representação desse tipo de grafo ainda pode ser feita com matriz ou lista de adjacências, entretanto, no caso da matriz, ao invés dos algarismos 0 ou 1, deve ser indicado o peso entre da aresta que liga os vértices (LAFORE, 2004).

**2.2. *Collections Framework***

*Collections* - coleções - são tipos de objetos que possuem o objetivo de armazenar e manipular conjuntos de dados e informações em tempo de execução. Geralmente, esses dados são, na verdade, outros tipos de objetos, que podem ou não ser também coleções. A linguagem de programação Java fornece uma arquitetura unificada, composta por classes e interfaces, para representar e manipular coleções, conhecida como *Collections Framework* (GUPTA, 2005).

Um dos principais recursos fornecidos por esse *framework* são as listas: tipo de coleção que permite elementos duplicados. Na interface *List* estão definidos métodos como *add* (adiciona um elemento na coleção), *get* (permite a obtenção de um elemento inserido), *iterator* (disponibiliza um iterador para a coleção) e *remove* (realiza a remoção de um elemento inserido). A implementação mais conhecida e utilizada de *List* é o *ArrayList,* que trabalha internamente com um *Array* e, dessa forma, possui maior eficiência para pesquisa, quando comparado com seu concorrente, o *LinkedList*, que, por sua vez, é mais ágil na inserção e remoção de itens (GUPTA, 2005).

Outra interface disponibilizada pelo *Collections* é conhecida como *Queue,* que, tipicamente, mas não necessariamente, agrupa elementos segundo a definição da estrutura de dados fila. Uma das implementações mais utilizadas da *Queue* é a *PriorityQueue,* que funciona como uma fila de prioridade. Contudo, para inserir objetos na *PriorityQueue,* é fundamental que esses implementem a interface *Comparable*, que permite a definição dos parâmetros de prioridade. É válido ressaltar, ainda, a existência de implementações de *Queue* que seguem as definições de uma outra estrutura de dados: a pilha (GUPTA, 2005).

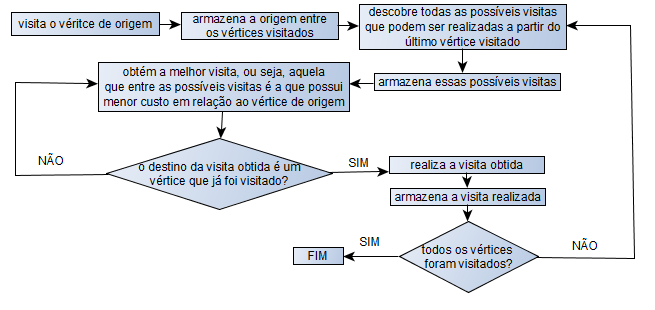
Além disso, existem outros dois tipos de coleções disponibilizados pelo *Collections,* a *Set* e a *Maps*, enquanto a primeira é um tipo de coleção que não permite itens duplicados, a segunda associa a cada item uma chave, de forma a potencializar funcionalidades de busca (GUPTA, 2005).

**2.3. Algoritmo de *Dijkstra***

Dado um grafo G, ponderado e conexo, um problema muito frequente associado a esse tipo de estrutura é o de encontrar o menor caminho entre dois vértices quaisquer de G. Esse caminho é, na verdade, aquele que proporciona a mínima soma total dos pesos das arestas percorridas. Na literatura são propostas diversas formas de se alcançar uma solução para esse tipo de cenário, uma das mais conhecidas é através do algoritmo de *Dijkstra,* desenvolvido em 1956 pelo holandês *Edsger Dijkstra* (LAFORE, 2004).

A técnica de *Dijkstra* fornece, em relação a um vértice de origem, o caminho de menor custo para cada um dos outros vértices de um grafo. Por conta da sua abordagem, que se baseia em definir as melhores decisões em tempo execução, o algoritmo de *Dijkstra* é definido como “guloso”. Esses e outros aspectos intrínsecos à sua execução estão evidenciados no fluxograma da figura 2, que apresenta os passos primordiais para a aplicação da técnica de *Dijkstra* (LAFORE, 2004).

**Figura 2. Representação em fluxograma do algoritmo de *Dijkstra.***



**FONTE: Próprio Autor**

Por não haver uma modelagem fixa de implementação, muitas especificidades do algoritmo podem variar de acordo com o código fonte construído, como as estruturas de dados empregadas e o custo de execução, que possui valor médio de O([a+v]log n), onde a é o número de arestas e v é o número de vértices do grafo no qual o algoritmo é aplicado. Contudo, é importante ressaltar que o comportamento do algoritmo é bem definido apenas para situações em que todas as arestas possuem peso não negativo, podendo ser dirigidas ou não dirigidas (LAFORE, 2004).

**2.4. *Applets***

*Applets* são pequenos softwares responsáveis, em geral, por realizar uma atividade específica dentro do contexto de outro programa. O maior campo de aplicação desse tipo de ferramenta é em aplicações desenvolvidas especificamente para *Web.* Ao carregar uma página *Web* que possui esse tipo de executável associado, o navegador carrega o *applet* e realiza a sua execução (DEITEL, 2005).

A linguagem de programação Java fornece a superclasse *Applet,* que deve ser herdada pela classe na qual será iniciada a execução da aplicação*.* Entretanto, essa classe pai não oferece suporte aos componentes da biblioteca *Swing.* Dessa forma, em projetos que utilizem esses recursos, a classe herdada deve ser a *JApplet,* que estende *Applet* e disponibiliza essa funcionalidade. (DEITEL, 2005).

Cinco métodos dessa classe merecem destaque, pois eles definem o ciclo de vida do *applet.* Esses métodos e os seus respectivos propósitos estão listados na tabela 1 (DEITEL, 2005).

**Tabela 1. Métodos que definem o ciclo de vida de um *applet* e seus propósitos.**

|  |  |
| --- | --- |
| **Método** | **Propósito** |
| **public void init()** | Chamado pelo navegador ou *applet view* para informar ao *applet* que ele foi carregado no sistema. |
| **public void start()** | Chamado pelo navegador ou applet view para informar ao applet que ele precisa iniciar sua execução. |
| **public void paint(Graphics g)** | Chamado pelo navegador ou applet view após os métodos init() e start() para pintar a tela do applet. É chamado também quando a tela precisa ser repintada. |
| **public void stop()** | Chamado pelo navegador ou *applet view* para informar ao *applet* que  ele precisa parar sua execução. |
| **public void destroy()** | Chamado pelo navegador ou *applet view* para informar ao *applet* que  ele foi removido da memória e precisa desalocar todos os recursos alocados. |

FONTE: (DEITEL, p. 719, 2005)

Com exceção do método paint(), as implementações dos métodos supracitados na superclasse são vazias, ou seja, não possuem linhas de código. Portanto, caso uma classe filha deseje executar alguma ação relevante durante a chamada de algum deles, ela precisa sobrescrever o método específico (DEITEL, 2005).

A forma geração de um arquivo .HTML que permite que o *applet* seja executado no navegador varia em relação ao ambiente de desenvolvimento integrado utilizado. Entretanto, no caso específico do *Netbeans,* o ato de estender uma classe que herda de *Applet* faz com que o arquivo seja gerado automaticamente e, dessa forma, torna-se possível utilizar a aplicação em *browsers*.

***2.6. GUI Builder***

Diversos ambientes de desenvolvimento integrados oferecem ferramentas para facilitar a construção de interface gráficas. Com esses utilitários, conhecidos como *GUI Builder,* é possível substituir grande parte da codificação pelas simples ações de selecionar, arrastar e ajustar os componentes desejados em tela. Em muitos *IDE’s,* o *GUI Builder* é instalado automaticamente. Entretanto, em outros, é necessário fazer um download extra dessa ferramenta (DEITEL, 2005).

**3. Metodologia**

Antes de iniciar a resolução propriamente dita do problema, foi fundamental entender o cenário proposto. Esse entendimento aconteceu através da leitura crítica do texto descritivo, a partir do qual também tornou-se possível delimitar quais aspectos teóricos deveriam ser estudados e compreendidos, além dos requisitos solicitados.

Ademais, as reuniões tutoriais durante o processo de desenvolvimento foram fundamentais, pois nelas foram discutidas dúvidas que surgiam apenas em tempo de implementação. Pelo fato do processo de programação ter ocorrido em dupla, existiram também discussões internas sobre alguns detalhes específicos e, principalmente, sobre a modelagem do solução que estava sendo consolidada.

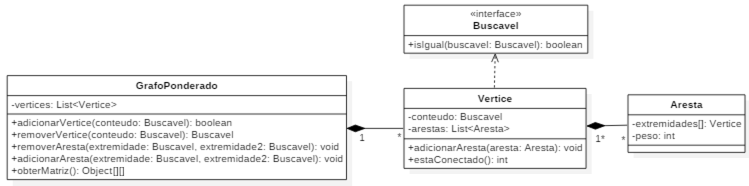
O código fonte da aplicação foi criado no sistema operacional *Windows 8*, por meio do versão 8.1 do ambiente de desenvolvimento integrado *Netbeans.* Além disso, por conta do projeto ter sido desenvolvido em dupla, foi fundamental utilizar o *GitHub*, uma plataforma para controle e gerenciamento de versões.

**3.1. Modelagem da solução**

**3.1.1. Estruturação do grafo ponderado**

Por conta de sua estrutura flexível e da grande potencialidade de simular situações do mundo real, a estrutura de dados empregada para representar pontos e caminhos no contexto do software implementado foi o grafo ponderado. Três classes estão envolvidas na arquitetura dessa estrutura: “Vertice”, que representa o vértice de um grafo, “Aresta”, representando uma aresta e “GrafoPonderado”, que, por sua vez, representa o grafo propriamente dito.

Além disso, é importante ressaltar, ainda, a existência da interface “Buscavel”, que deve ser obrigatoriamente implementada pelas classes cujas instância serão armazenadas no grafo. Esse aspecto é fundamental, pois essa interface permite que qualquer um dos objetos anteriormente armazenados sejam identificados em uma possível busca. Essa e as outras classes envolvidas na estruturação do grafo ponderado estão evidenciadas no diagrama de classes da figura 3. Buscando-se potencializar a clareza do diagrama, apenas os métodos mais relevantes foram representados.

**Figura 3. Diagrama das classes que formam a arquitetura do grafo ponderado**

**FONTE: Próprio Autor**

Como pode ser observado, a interface *List*, que faz parte do *Collections Framework*, foi utilizada tanto para alocação de arestas em vértices, quanto para alocação de vértices no grafo. Vale ressaltar que a implementação de *List* empregada, efetivamente, foi a *ArrayList.*

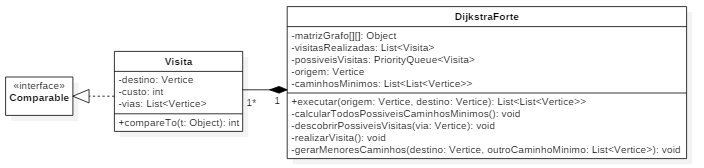
**3.1.2. Estruturação do algoritmo *DijkstraForte***

O algoritmo de *Dijkstra* originalmente realiza a aferição de todos os menores caminhos partindo de um vértice de origem em direção a todos os outros vértices de um grafo ponderado, desconsiderando os possíveis caminhos que possuam o mesmo custo mínimo. Entretanto, foi um requisito do problema que todas as possíveis rotas mínimas fossem disponibilizadas ao usuário, nos casos em que houvesse mais de uma. Dessa forma, foi necessário adaptar a técnica original de *Dijkstra,* o que culminou na criação do algoritmo DijkstraForte.

A classe que, efetivamente, implementa o algoritmo desenvolvido é denominada de “*DijkstraForte*” e possui a maioria dos seus métodos privados, pois correspondem a ações internas que não precisam estar disponíveis. Ademais, a classe “*Visita”* foi criada para representar a visita a um vértice durante a execução do algoritmo. A estruturação do DijkstraForte pode ser observada na figura 4, por meio de um diagrama das classes envolvidas no algoritmo.

Um fato que deve ser destacado é a presença de uma lista de objetos “Vertice” como atributo de “Visita”, representando as vias para se chegar a um destino. Esse é o aperfeiçoamento que permite o “*DijkstraForte”* considerar vários caminhos com o mesmo custo para um mesmo destino. Ademais, ao fim da sua execução, o resultado disponibilizado por “*DijkstraForte”* é uma lista contendo uma ou mais listas de instâncias de “Vertice”, sendo que cada uma delas contém os vértices que formam um dos caminhos mínimos.

**Figura 4. Diagrama das classes envolvidas na estruturação do *“DijkstraForte”***



**FONTE: Próprio Autor**

**3.1.3. Estruturação da camada *Model***

A única entidade que faz parte do “*Model”* é o ponto. O ponto representa um local físico no qual o carro forte irá, obrigatoriamente, passar durante a sua rota. Portanto, uma instância de “Ponto” pode se referir ao estacionamento de uma empresa, a um ponto de coleta ou ao banco. Percebeu-se que fazer distinção entre esses locais não seria necessário, afinal todos possuem um único atributo que é de interesse: o seu nome.

Na figura 5 há a representação da classe “Ponto”, é importante notar que ela implementa precisa implementar a interface “Buscavel”, pois será alocada e, posteriormente, buscada no grafo ponderado.

**Figura 5. Representação da classe “Ponto”**



**FONTE: Próprio Autor**

**3.1.4. Estruturação da camada *Controller***

A camada de negócios, proposta pelo padrão MVC, é materializada no contexto do software implementado pela classe “ForteSeguroController”, nela há os métodos responsáveis pelo cadastro e remoção de novos pontos e caminhos. Entretanto, o método mais significativo dessa camada é o responsável por utilizar o DijkstraFortee realizar o cálculo dos caminhos mínimos.

Tendo em mente que o requisito inicial do problema foi descrito: “[..]sair do estacionamento de uma empresa, passar em um ponto de coleta e entregar a quantia coletada em um banco.”, fica fácil perceber que uma única aplicação do algoritmo DijkstraForte não resolve o problema. A abordagem utilizada para solucionar esse cenário está representada na figura 6.

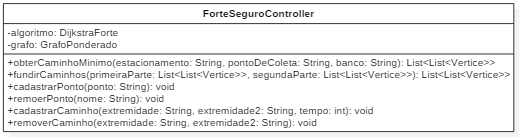
**Figura 6. Aplicação no problema proposto do algoritmo “DijkstraForte”.**



**FONTE: Próprio Autor**

A execução dos passos acima é feita por meio dos métodos “obterCaminhoMinimo” e “fundirCaminhos”. Esse e outros métodos estão evidenciados na figura 7, onde há uma representação da classe “ForteSeguroController”.

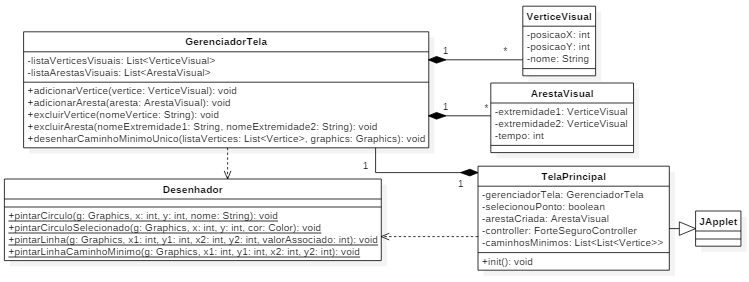
**Figura 7. Representação da classe “*ForteSeguroController*”**



**FONTE: Próprio Autor**

**3.1.5. Estruturação da camada *View***

Grande parte da interface gráfica foi desenvolvida com base na ferramenta *GUI Builder*, disponibilizada pelo *Netbeans.* Os principais componentes utilizados foram painéis, *ComboBox,* caixas de texto e janelas de mensagem. Contudo, a representação visual do grafo foi codificada manualmente e culminou criação das classes “VerticeVisual”, “ArestaVisual” e “Desenhador”. As duas primeiras classes modelam a representação gráfica de um vértice e de uma aresta, respectivamente. Já a classe “Desenhador” não pode ser instanciada e serve apenas para disponibilizar métodos estáticos que desenham os vértices e arestas em tela. Na figura 8 estão evidenciadas essas e outras classes que formam a camada “*View”* da aplicação.

**Figura 8. Diagrama das classes que formam a “*View”.***

**FONTE: Próprio Autor**

A classe que molda a tela única da aplicação e pela qual a execução é iniciadafoi denominada de “*TelaPrincipal*”, nela estão presentes os métodos que lidam com os eventos gerados por cliques e outras interações do usuário com a *GUI.* Outro aspecto importante dessa classe é que ela herda diretamente da classe *JApplet,* fornecida pela biblioteca *Swing* para a criação de *applets.*

**3.2. Desenvolvimento de testes unitários.**

Os principais módulos do programa foram aferidos por meio da construção de testes unitários, com base no *framework* *JUnit.* Esses testes foram direcionados para verificar a funcionalidade da classe que implementa o algoritmo “*DijkstraForte*” e das classes “*GrafoPonderado*” e “*ForteSeguroController*”. Os casos de uso considerados foram aqueles em que a execução ocorre dentro da normalidade e, além disso, situações excepcionais, como o lançamento de exceções.

Vale salientar que, especificamente para o teste do algoritmo “*DijkstraForte”,* os casos foram disponibilizados pelas duplas da sessão tutorial e abrangem desde a obtenção caminhos mínimos triviais, até aqueles em que são gerados duas ou mais rotas com um mesmo custo para um mesmo destino.

**4. Resultados**

**4.1. *Users Stories* implementadas**

No texto do problema foram informadas as *Users Stories* que, obrigatoriamente deveriam ser desenvolvidas. Na tabela 2 estão listadas essas *Users Stories* e o percentual de cada uma delas que foi, efetivamente, implementado.

**Tabela 2. *UsersStories* implementadas.**

|  |  |
| --- | --- |
| ***User Story*** | **Percentual de Implementação** |
| 1 – Cadastrar pontos e caminhos | 100% |
| 2 – Remover pontos e caminhos | 100% |
| 3 – Calcular caminho mais rápido | 100% |
| 4- Exibir visualmente os pontos e caminhos cadastrados | 100% |

    FONTE: Próprio Autor

**4.2. Criação de testes unitários**

Com o objetivo de potencializar os níveis de confiabilidade e robustez do sistema foram desenvolvidos testes para cada uma das unidades individuais da aplicação. Esses testes e o percentual de funcionamento obtido através deles estão evidenciados na tabela 3.

**Tabela 3. Testes unitários realizados e percentual de funcionamento obtido.**

|  |  |
| --- | --- |
| **Teste Realizado** | **Percentual de Funcionamento** |
| 1 – ForteSeguroControllerTest | 100% |
| 2 – DijkstraForteTest | 100% |
| 3 – GrafoPonderadoTest | 100% |

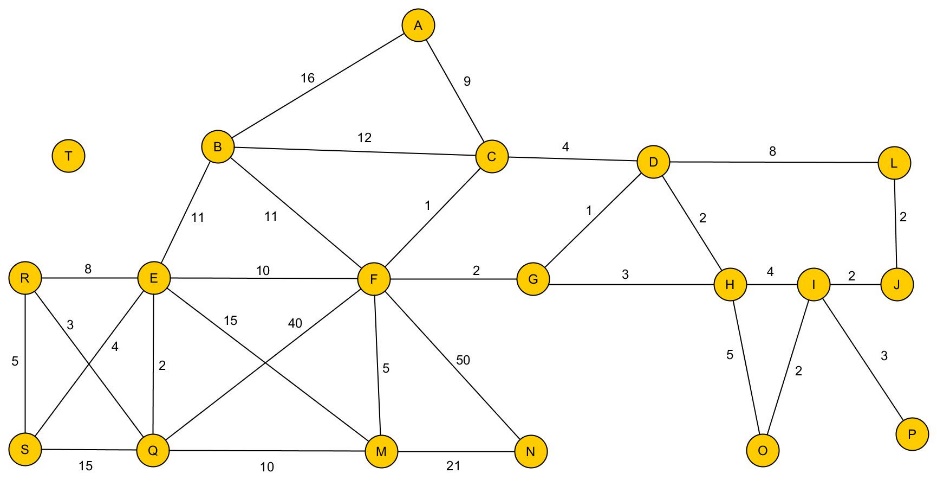
    FONTE: Próprio Autor

A análise desses resultados evidencia que o funcionamento de cada uma das principais funcionalidades do software ocorre dentro dos parâmetros esperados.

**4.3. Análise de resultados fornecidos por “*DijkstraForte”***

Diversas configurações e possíveis caminhos foram testados durante a criação dos testes unitários para a classe “*DijkstraForte”*. Na figura 9 evidencia-se um dos grafos testados e na tabela 4 a comparação entre os caminhos esperados e os que foram, efetivamente, obtidos.

**Figura 9. Grafo utilizado como entrada para “*DijkstraForte”***



**FONTE: Próprio Autor**

**Tabela 4. Comparação entre os caminhos mínimos esperados e obtidos.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Origem** | **Destino** | **Caminho(s) Esperado(s)** | **Caminho(s) Obtido(s)** | **Custo** |
| **A** | **B** | **A-B** | **A-B** | **16** |
| **C** | **Q** | **C-F-E-Q** | **C-F-E-Q** | **13** |
| **A** | **L** | **A-C-D-L / A-C-F-G-D-L** | **A-C-D-L / A-C-F-G-D-L** | **21** |
| **G** | **H** | **G-H / G-D-H** | **G-H / G-D-H** | **3** |
| **P** | **Q** | **P-I-H-D-G-F-E-Q / P-I-H-G-F-E-Q** | **P-I-H-D-G-F-E-Q / P-I-H-G-F-E-Q** | **24** |
| **C** | **B** | **C-B / C-F-B** | **C-B / C-F-B** | **12** |
| **R** | **J** | **R-Q-E-F-G-H-I-J / R-Q-E-F-G-D-H-I-J** | **R-Q-E-F-G-H-I-J / R-Q-E-F-G-D-H-I-J** | **26** |
| **S** | **N** | **S-E-Q-M-N** | **S-E-Q-M-N** | **37** |
| **J** | **B** | **J-I-H-G-F-B ou J-I-H-D-G-F-B** | **J-I-H-G-F-B ou J-I-H-D-G-F-B** | **22** |

FONTE: Próprio Autor

De acordo com os casos evidenciados percebe-se que para o grafo de entrada, todos os caminhos mínimos de saída obtidos são, exatamente, os esperados.

**4.4. Telas do sistema**

A primeira tela exibida ao iniciar o *applet* em um navegador está representada na figura 10.

**Figura 10. Tela exibida ao inicializar o *applet*.**

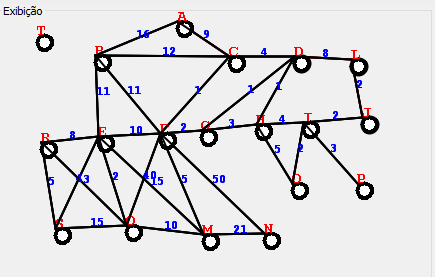


**FONTE: Próprio Autor**

Para cadastrar um novo ponto o usuário precisa clicar em qualquer região do painel “Exibição” e informar o nome do ponto na caixa de diálogo que será exibida. Já o cadastro de um novo caminho é realizado através do clique nos pontos que representam as extremidades desejadas e da entrada do tempo de percurso na caixa de diálogo exibida. Para realizar remoções, o usuário deve alternar entre as opções disponíveis na *ComboBox*, localizada no painel esquerdo da tela e preencher corretamente as informações na caixa de texto localizadas abaixo dela.

Na figura 11 há uma representação de como pontos e caminhos cadastrados são exibidos em tela.

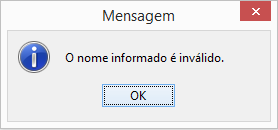
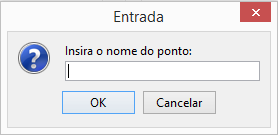
**Figura 11. Exibição de pontos e caminhos cadastrados.**



**FONTE: Próprio Autor**

Além das janelas para a inserção de dados cadastrais, são exibidas também janelas contendo mensagens de erro, caso a execução do programa fuja da normalidade. Exemplo de ambas pode ser visualizado na figura 12.

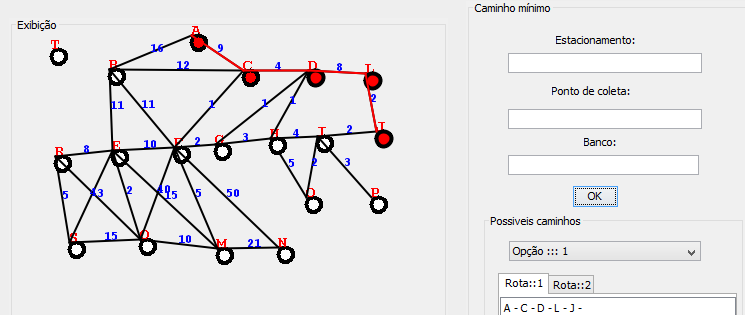
**Figura 12. Exemplos de janelas exibidas em tempo de execução.**

****

**FONTE: Próprio Autor**

Outro painel existente na tela está localizado na extremidade direita e nele são gerenciados os processos para cálculo de menor caminho. O usuário deve preencher as caixas de texto com as informações solicitadas e confirmar. Em tela a primeira rota mínima disponível será destacada em vermelho, caso existam mais caminhos disponíveis, eles devem ser selecionados através da *ComboBox* intitulada “Possiveis Caminhos”. Além disso, nesse mesmo painel, é disponibilizada uma representação textual do caminho mínimo, para sanar possíveis ambiguidades geradas pela representação gráfica. Na figura 13 a representação de uma rota mínima é evidenciada.

**Figura 13. Representação gráfica de um caminho mínimo.**

****

**FONTE: Próprio Autor**

**5. Conclusão**

Não é difícil abrir as páginas policias de jornais, blogs e meios de comunicação em geral, e se deparar com manchetes que noticiam roubo de cargas durante o processo de transporte. Muitos desses casos culminam, ainda, em latrocínio, tendo como consequência a perda de vidas humanas, algo que não pode ser recuperado. O grau de risco associado a essa atividade cresce de maneira diretamente proporcional com o valor do insumo transportado, sendo especialmente alto para o translado de dinheiro vivo, como acontece nos carros fortes.

Dessa forma, a produção e o aperfeiçoamento das técnicas que visam incrementar os níveis de confiabilidade desse serviço devem ser estimulados e potencializados. O software, cuja implementação foi descrita e discutida nesse relatório, é uma dessas ferramentas indispensáveis para as empresas que se propõem a realizar a condução de elevadas quantias monetárias, afinal, se o tempo necessário para o transporte for decrementado, o processo torna-se mais ágil e menos vulnerável a ação de criminosos.

Ademais, pode-se afirmar que a solução alcançada para o problema é extremamente favorável, pois todos os requisitos foram atendidos e a robustez de cada um dos módulos da aplicação foi fomentada pelo desenvolvimento de testes unitários. A adoção da divisão em camadas, sugerida pelo padrão MVC, possibilitou a geração de um código fonte com grande capacidade de reutilização e atendedor das boas práticas de programação.

Contudo, uma possibilidade de atualização do programa deve ser focada no aperfeiçoamento da interface gráfica, aumentando, assim, o grau de usabilidade da aplicação. Por fim, é possível concluir que processo de implementação foi fundamental para consolidar os conceitos teóricos relacionados a grafos, algoritmos de caminhos mínimos, interfaces gráficas e, até mesmo, serviu como uma pequena introdução para o campo de desenvolvimento *web.*

**6. Referências**

DEITEL, H.M.; DEITEL, P.J. **Java – Como Programar.** 6 ed. São Paulo: Pearson Prentice Hall, 2005.

LAFORE, R. **Estruturas de dados & Algoritmos em Java.** Rio de Janeiro: Editora Ciência Moderna Ltda., 2004.

GUPTA, M. **OCP Java SE 7 - Certification Guide.** New York: Manning Publications, 2005.