Forte Seguro

Esdras E. S. Santos[1], Daniel F. Campos[2]

Universidade Estadual de Feira de Santana(UEFS)

[1]EsdrasSenaSantos@gmail.com, [2]dfc152@gmail.com

Resumo: Este artigo descreve o processo de desenvolvimento de um software de logística para uma empresa de segurança. Ao longo deste artigo será explicado de forma detalhada as ferramentas utilizadas para a resolução do problema e de que maneira essas ferramentas foram utilizadas, bem como a implementação do algoritmo de Dijkstra. Ao final do artigo serão apresentados todas as possíveis melhorias, limitações e conclusões.

1. Introdução

A renomada empresa de segurança chamada Forte Seguro solicitou a ajuda dos alunos de MI de programação na solução de um problema de grande relevância em seu negócio. A Forte Seguro é responsável por recolher dinheiro de pequenas, médias e grandes empresas e levá-lo ao banco. Todos os carros fortes da empresa saem de um ponto em comum, que corresponde ao estacionamento da Forte Seguro e recolhem o dinheiro de somente um estabelecimento por vez para levá-lo ao banco. Cada carro forte só toma conhecimento do ponto que vai passar para coletar o dinheiro uma hora antes da coleta, por motivos de segurança. O problema é que existem vários caminhos diferentes entre o estacionamento e o ponto de coleta, e o ponto de coleta e o banco, com tempos de duração diferentes, porém conhecidos. É muito difícil descobrir o caminho mais curto de forma manual em menos de 60 minutos. Por isso é necessário o desenvolvimento de um *software* que resolva essa questão.

2. Fundamentação teórica

Segundo [Lafore 2003] o algoritmo de Dijkstra é baseado em uma matriz de adjacências representando um grafo. Ele não acha somente o caminho mais curto de um vértice especificado para outro, mas também o caminho mais curto do vértice especificado para todos os outros vértices. O algoritmo de Dijkstra trabalha com grafos ponderados, segundo Lafore, se vértices podem representar cidades em um grafo, os pesos em um grafo ponderado podem

representar distâncias entre eles.

Garfos matematicamente falando é um conjunto, não vazio, de vértices e arestas, sendo vértices os “pontos” do grafo e aresta as ligações entre eles. Na computação grafos podem ser representados de duas formas, sendo elas com a utilização de matriz ou a utilização de listas encadeadas. Cada uma delas tendo suas vantagens e desvantagens, como mostrado por [Lafore 2003].

[Eckstein 1998] define o java swing como um conjunto de componentes gráficos customizáveis que podem ter seu look-and-feel editados em tempo de execução. Havendo algumas classes pré-feitas como container, que são objetos gráficos que guardam outros objetos gráficos, como painéis e barras de menu e objetos gráficos de interação com o usuário campos de texto, botões ,dentre outros.

Para [Eric Freeman 2007], no padrão de projeto Facade, criamos uma classe que simplifica e unifica um conjunto de classes mais complexas pertencentes a algum subsistema. Diferentemente de muitos outros padrões, o padrão Facade é razoavelmente simples; ele não usa abstrações sofisticadas cuja compreensão exige um enorme esforço mental. Mais isso não o torna menos poderoso: o padrão Facade permite evitar a vinculação rıgida entre clientes e subsistema.

O padrão Observer para [Eric Freeman 2007] define uma dependência de um para-muitos entre objetos para que quando um objeto mude de estado todos os seus dependentes sejam avisados e atualizados automaticamente.

[Eric Freeman 2007] diz que o Strategy define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. A estratégia deixa o algoritmo varie independentemente dos clientes que o utilizam.

Segundo [Eric Freeman 2007], o padrão Modelo-Visualização- Controlador(MVC) é um padrão composto que utiliza originalmente os padrões Observer para manter seus observadores atualizados sem manter vínculos diretos com eles, o Strategy para o controlador e o Composite para implementar a interface de usuário. Todos os padrões funcionam conjuntamente para desvincular os três elementos do modelo MVC, mantendo a estrutura transparente e flexível. [Vincent Massol 2004] afirma que um teste unitário examina o comportamento de um distinta unidade de trabalho dentro de uma aplicação java, a “distinta unidade de trabalho” é frequentemente(mas não sempre) um único método.

Para [Horstmann 2010] uma thread é uma unidade de programa que é executada independente das outras partes do programa. Ou seja é executado de forma “paralela” e independente um script. Para entendermos isso devemos entender que todo programa quando em execução tem um thread principal, podendo ser criados outros. Os threads de um mesmo programa

compartilham o mesmo espaço de memória, ou seja podem acessar ou modificar os mesmo objetos em tempos de execução.

3. Metodologia

O problema foi resolvido utilizando o padrão arquitetural MVC, que nos permite desvincular as operações lógicas da aplicação da interface disponível para o cliente, através de uma ponte. As operações lógicas no MVC são organizadas no pacote chamado de Model. Para esse problema o pacote Model foi desenvolvido utilizando um grafo ponderado, os tempos de duração de um ponto a outro foram inseridos no grafo como pesos das ligações em uma matriz de adjacências, dessa forma a diagonal principal da matriz se mantém nula, assim para adicionar mais um vértice a matriz basta somente adicionar mais um espaço a matriz, esse processo é feito criando uma nova matriz com um espaço a mais e copiando os dados da anterior para a nova e posteriormente salvando-a no lugar da antiga. A remoção funciona de forma similar, criando uma matriz com um espaço a menos e depois copiando todos os dados, que não envolvam o vértice removido, da matriz anterior e salvando esta nova matriz no lugar da antiga. Uma nova ligação é adicionada indicando dois vértices ligados e peso entre essa ligação e adicionando esse peso em sua respectiva posição na matriz. Zerando esse peso entre os vértices a ligação é removida.

O algoritmo de Dijkstra é crucial para encontrar o caminho que mais curto, a sua implementação foi feita recebendo um matriz de adjacências, um ponto de partida e um ponto de chegada, assim ele calculava o menor caminho do ponto inicial para o ponto final.

Para a implementação do Controller foi utilizado o padrão de projeto Facade ao invés do Strategy, pois tínhamos apenas o grafo como estrutura principal de nossa Model. Seria mais simples a implementação do Facade, dessa forma nós encapsulamos todos os métodos de baixo nível do grafo e o View conheceria apenas a interface do Controller, sendo assim se o View fosse alterado não precisamos a Model e apenas pequenas ou nulas mudanças seriam feitas no Controller, porém se a Model fosse alterada precisamos ajustar o conteúdo dos métodos presentes no Controller, mas o View continuaria inalterado, Assim seguindo as regras do padrão arquitetural MVC Ainda no Controller foi implementado o Observer para observar as interações com o View.

O View foi implementado utilizando a biblioteca de interfaces gráficas Swing, essa biblioteca foi escolhida por ser considerada mais simples do que a outra alternativa o JavaFX. A interface foi desenvolvida para ser navegada utilizando abas, onde ao clicar em cada uma das abas é possível chegar em uma das telas e cada botão das telas pode chamar um método do controller que corresponde a ação que ele vai realizar.

O algoritmo tem três telas principais, Figura de 1 a 3. No algoritmo foi utilizado threads para

mostrar mensagens para informações do usuário se a operação foi realizada com sucesso ou houve algum problema .Para os próximos parágrafos os campos de texto são contados de cima para baixo, da direita para esquerda.

Figura 1. Tela de Adição ou remoção de vértices.

Na Figura 1 é possível realizar a adição vértices, digitando no primeiro campo o número de vértices que deseja adicionar e selecionando o tipo dos vértices na caixa de seleção ao lado e por fim apertando botão adicionar. Para remover um vértice é necessário digitar o index do vértice no 2 campo de texto e apertando o botão de remover, não sendo possível remover o vértice contendo o estacionamento. Para trocar dois vértices de posição, representando a modificação do endereço de um ponto de coleta, banco ou estacionamento, digita o index do ponto que deseja remover no terceiro campo de texto e o novo index dele no quarto campo de texto. E por fim para importar o arquivo digita o caminho do no quinto campo de texto e apertar o botão importar, caso arquivo não seja encontrado aparece uma mensagem de erro na tela informando.

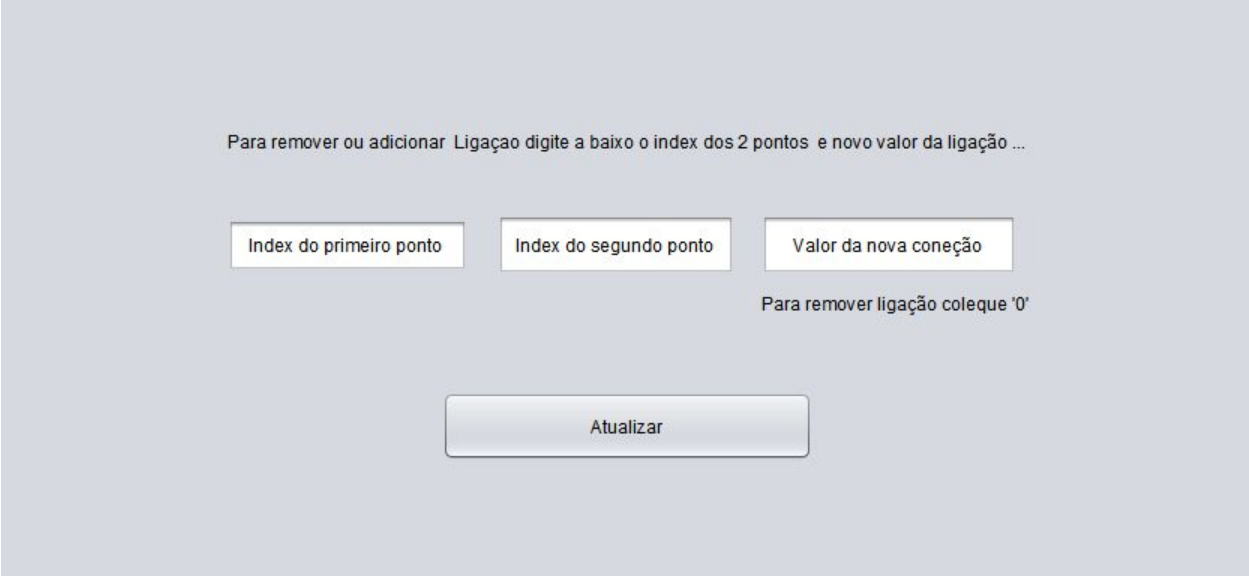

Figura 2. Tela de modificação das ligações (arestas).

Na Figura 2 temos a tela de modificação de aresta, sendo ela possível adicionar ou remover ligações , para adicionar ligações digitando os index dos vértices que serão ligados no primeiro e segundo campo e no terceiro digitando o valor da ligação, para remover deve ser colocado 0 no campo de valor para remover a ligação, e por fim apertando o botão atualizar.

Por fim na última tela, Figura 3, temos a tela na qual é possível realizar o dijkstra. Nesta tela é realizado o cálculo do menor caminho entre os pontos do grafo, para isso ser realizado é necessário digitar no primeiro campo o index de um vértice de início, sendo possível digitar ‘ ‘ para que esse vértice seja o estacionamento mas também podendo ser usado qualquer vértice, no segundo campo o index de um ponto intermediário, sendo normalmente o ponto de coleta, e por fim no terceiro campo o index do ponto final, sendo normalmente o index do banco.

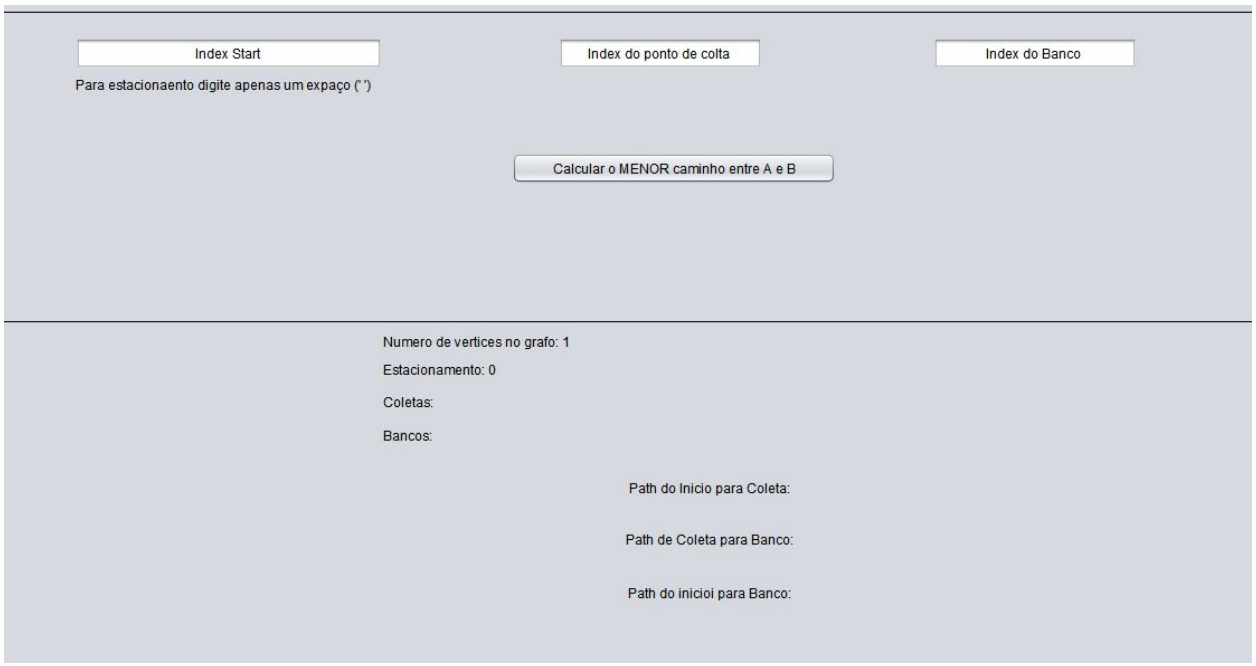

Figura 3. Tela do Dijkstra.

Sendo desenvolvido utilizando as IDEs NetBeans, na versão 10.0, e CoolBeans, na versão 2019.06, no sistema operacional Windows 10.

4. Resultados e Discussões

O *software* foi projetado para ter a capacidade de resolver o problema do caminho mínimo e apresentar para o usuário qual os pontos que ele tem que percorrer para chegar ao seu destino da forma mais rápida em questões de tempo gasto, o *software* também tem a capacidade de importar o arquivo contendo uma matriz de adjacência padronizada. O *software* não tem a capacidade de representar o grafo de forma gráfica, a concorrência e o paralelismo não são tratados no *software* com o uso de Threads. Os testes foram feitos utilizando o Framework JUnit 4.12 que nos permite realizar testes unitários em nossa aplicação, dessa forma foram testados cada método das classe criadas para garantir que não iriam ocorrer erros em tempo execução.

5. Conclusão

Ao final do processo de desenvolvimento o *software* foi alcançado o resultado esperado, informando ao usuário o caminho que demorasse o menos tempo e atualizando o caminho toda vez que um novo vértice fosse adicionado ou removido. O *software* se limita a mostrar o caminho de forma textual sem a apresentação de um forma gráfica para tal.

Futuramente pode ser implementado Threads para resolver os problemas de concorrência/paralelismo. Também pode ser implementado a visualização gráfica do grafo para uma melhor compreensão do usuário.

Referências

Eckstein (1998). Java Swing. O'Reilly, 1st edition.

Eric Freeman, E. F. (2007). Use a Cabeça! Padrões de Projetos. Alta Books, 2nd edition.

Horstmann, C. (2010). Big Java. Wiley, 4th edition.

Lafore, R. (2003). Data Structures and Algorithms in Java. Pearson Education, 2nd edition.

Vincent Massol, T. H. (2004). JUnit in Action. Manning, 1nd edition.

Feofiloff, Paulo (2019). Estrutura de dados para grafos. Disponível em:

ime.usp.br. acesso em 23 de set de 2019.