SISTEMA DE ROTEIRIZAÇÃO PARA TRANSPORTE PÚBLICO RODOVIÁRIO

Diego Carvalho Domingos
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-093 – Itajubá-MG
diego_cdomingos@yahoo.com.br

Rodrigo Figueiredo Zaiden
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-093 – Itajubá-MG
rodrigoffzz@gmail.com

Vinícius José Silveira de Souza
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-093 – Itajubá-MG
viniciusjssouza@gmail.com

Edmilson Marmo Moreira
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-093 – Itajubá-MG
edmarmo@unifei.edu.br

Otávio Augusto Salgado Carpinteiro
Universidade Federal de Itajubá
Av. BPS, 1303 – Pinheirinho – Cep. 37.500-093 – Itajubá-MG
otavio@unifei.edu.br

RESUMO

Este artigo apresenta um sistema capaz de realizar o planejamento de uma viagem de ônibus intermunicipal ou interestadual, exibindo ao usuário as melhores alternativas quanto aos trajetos que podem ser utilizados, considerando critérios como tempo e custo da viagem. O sistema foi desenvolvido sob a plataforma Java Enterprise Edition em conjunto com tecnologias recentes, como o Ajax, Google Maps e bancos de dados NoSQL.

PALAVRAS CHAVE. Roteirização. Transporte público. Otimização. Logística.

ABSTRACT

This paper introduces a software system which performs the planning of travels by intermunicipal and interstate buses, showing to its passengers the alternatives about the routes that can be taken, considering criteria such as the total time and total cost spent on the travel. The system was built using the Java Enterprise Edition platform and recent technologies, such as Ajax, Google Maps and NoSQL database.

KEYWORDS. Routing. Public Transportation. Optimization. Logistics.

1. Introdução

Realizar o planejamento de uma viagem é fundamental para reduzir tempo e custos necessários. Com o GPS (*Global Position System*) ou Sistema de Posicionamento Global, os proprietários de automóveis particulares podem realizar esta tarefa de maneira simples, rápida e confiável. Até mesmo pessoas que não possuem o aparelho podem atingir este objetivo através de consultas *on-line*. Existem muitos *sites* para este fim, entre eles Google Maps®, MapLink®, Guia 4 Rodas®, Via Fácil®, entre outros. Porém, para aqueles que necessitam de transporte público rodoviário, esta tarefa se torna bem mais complexa.

Atualmente, alguns *sites* fornecem este serviço para linhas municipais, mas somente para poucas capitais, como é o caso do Google Transit®. Outros fornecem apenas a informação se há uma linha que faz o trajeto entre duas cidades, como é o caso do *site* da ANTT (Agência Nacional de Transportes Terrestres) e da ARTESP (Companhia Reguladora de Serviços Públicos Delegados de Transporte do Estado de São Paulo), sendo que este último é restrito ao estado de São Paulo. Devido às dificuldades encontradas para se obter informações a respeito das linhas de transporte público rodoviário, foi desenvolvido um sistema que fornece um serviço de roteirização utilizando trajetos de ônibus intermunicipais e interestaduais, através do qual o usuário poderá obter diversas informações que o ajudarão a planejar sua viagem utilizando a Internet.

Os conceitos fundamentais para realização deste trabalho estão ligados à teoria dos grafos, em especial aos chamados algoritmos de caminho mínimo. Além disso, para a construção do sistema, foram utilizadas tecnologias para a visualização de rotas, criação de páginas dinâmicas, *caching* de informações e gerenciamento de base de dados.

Este artigo está estruturado da seguinte forma: a próxima seção faz uma breve explanação das principais tecnologias utilizadas no desenvolvimento deste trabalho. A seção 3 faz uma breve discussão do problema do caminho mínimo e do algoritmo de Dijkstra, contextualizando-os no problema cujo objetivo é o foco deste trabalho de pesquisa. A seção 4 apresenta o sistema desenvolvido e, finalmente, a seção 5 encerra este artigo com os resultados alcançados.

2. Tecnologias utilizadas

Uma aplicação *web* envolve, basicamente, máquinas conectadas em uma rede, que trocam informações entre si. Devido a essa característica, é necessário que sejam estabelecidas regras sobre como será a comunicação, ou seja, o envio e o recebimento dessas informações. O protocolo utilizado neste caso é o *Hypertext Transfer Protocol* (HTTP), que é baseado no paradigma *request-response* (TROELSEN, 2007).

Devido à natureza *stateless* do protocolo HTTP, o ambiente *web* apresenta uma séria de dificuldades ao desenvolvedor. Os *frameworks* que implementam o padrão MVC (*Model-View-Controller*) surgiram com o intuito de facilitar o desenvolvimento *web* e forçar o programador a utilizar as boas práticas. O principal objetivo deste padrão de projeto é separar claramente a aplicação em três componentes lógicos: *Model*, responsável pelo encapsulamento de acesso a dados e serviços; *View*, responsável pela apresentação dos dados; *Controller*, que realiza a integração entre os dois componentes anteriores (MCARTHUR, 2008).

Diversas são as ferramentas disponíveis atualmente para a construção de sistemas *web*. Há um bom tempo no mercado, a plataforma J2EE (*Java Entrerprise Edition*) é construída sobre a linguagem de programação Java e a máquina virtual Java (JVM), que promovem portabilidade, segurança, desempenho e produtividade. É capaz de fornecer um conjunto completo de tecnologias para desenvolvimento *web*. Atualmente na versão 6, apresenta um grande progresso em se tratando de facilidade e agilidade de projeto (MICROSYSTEMS, 2009).

Um dos grandes avanços no desenvolvimento *web* dos últimos anos foi a

introdução das chamadas assíncronas ao servidor, técnica conhecida por Ajax (*Asynchronous JavaScript and XML*). Antes da adoção desta técnica, a interação do usuário com páginas *web* era realizada unicamente por meio de requisições que atualizavam todo o estado da página. Isso também introduz um tráfego maior na rede, uma vez que imagens, arquivos CSS e outros tipos de conteúdos estáticos são carregados diversas vezes. A utilização de Ajax fornece ao usuário uma experiência agradável durante a navegação e diminui o tráfego na rede, pois apenas os dados necessários são trocados e os elementos da página são atualizados por código cliente Javascript (BIBEAULT; KATZ, 2008). Juntamente com a adoção do Ajax, surgiram diversas bibliotecas que visavam facilitar a utilização da técnica, além de simplificar a manipulação de documentos HTML, tratamento de eventos e animações, entre outras facilidades. O jQuery é um exemplo de biblioteca Javascript que fornece todas esta funcionalidades (JQUERY, 2006).

Atualmente, na internet, as soluções de mapeamento são um ingrediente natural, sendo usadas para consultar localizações em geral, para obter instruções de direção e fazer inúmeras outras tarefas. A maioria das informações tem uma localização que pode ser exibida em um mapa. A Google Maps API é uma interface JavaScript para manipulação e exibição de mapas utilizando o Google Maps (SVENNERBERG, 2010).

A quantidade de dados gerados, armazenados e processados atingiu escalas inéditas com a *Web* 2.0. Soluções diversas, não baseadas em bancos de dados relacionais, estão sendo implementadas e, ao conjunto desses bancos de dados não-relacionais, deu-se o nome de NoSQL (*Not Only SQL*) (DIANA; GEROSA, 2010). Nesse contexto surgem, dentre outros tipos, os bancos de dados em grafos. Embora escalabilidade não seja um requisito para a aplicação desenvolvida, utilizar um banco de dados que armazena os dados na forma de um grafo traz vantagens, principalmente no que diz respeito à rapidez no acesso, tendo em vista que os dados de cidades e linhas entre elas formam implicitamente um grafo. Além disso, é possível desenvolver a aplicação de uma maneira mais intuitiva, já com a noção de grafo, ao invés das tradicionais tabelas do modelo relacional. Há uma implementação madura nesse segmento, o Neo4j. Ele foi desenvolvido em Java e possui uma API de utilização simples, o que facilita a integração com a aplicação desenvolvida (INFOQ, 2010).

A integração destas tecnologias no sistema proposto pode ser visualizada na figura 1. O usuário inicia o uso do sistema através de um navegador *web*, fornecendo os dados da viagem que pretende realizar. O navegador realiza diversas requisições HTTP ao servidor, onde o processamento de busca pela melhor rota é feito. Para a execução do algoritmo, o servidor acessa o seu banco de dados, que contém as informações das linhas de ônibus. Ao final do processamento, o servidor retorna uma resposta HTTP contendo a melhor rota codificada no formato JSON, além de poder retornar algum conteúdo HTML para exibição. O cliente (navegador) utiliza código Javascript para exibir as informações ao usuário. A rota é exibida utilizando o GoogleMaps, através de uma nova requisição HTTP aos serviços do Google.

3. Caminho de custo mínimo

Um mapa rodoviário pode ser modelado como um grafo orientado, onde cada vértice representa uma rodoviária ou parada de ônibus em uma cidade, e as arestas são as linhas que ligam estas localidades. A informação de "custo" contida em cada aresta pode ser qualquer um dos parâmetros analisados, como o preço, a distância ou o tempo de viagem. De posse de tal modelo, o problema resume-se em encontrar o caminho entre duas localidades que minimize um destes parâmetros.

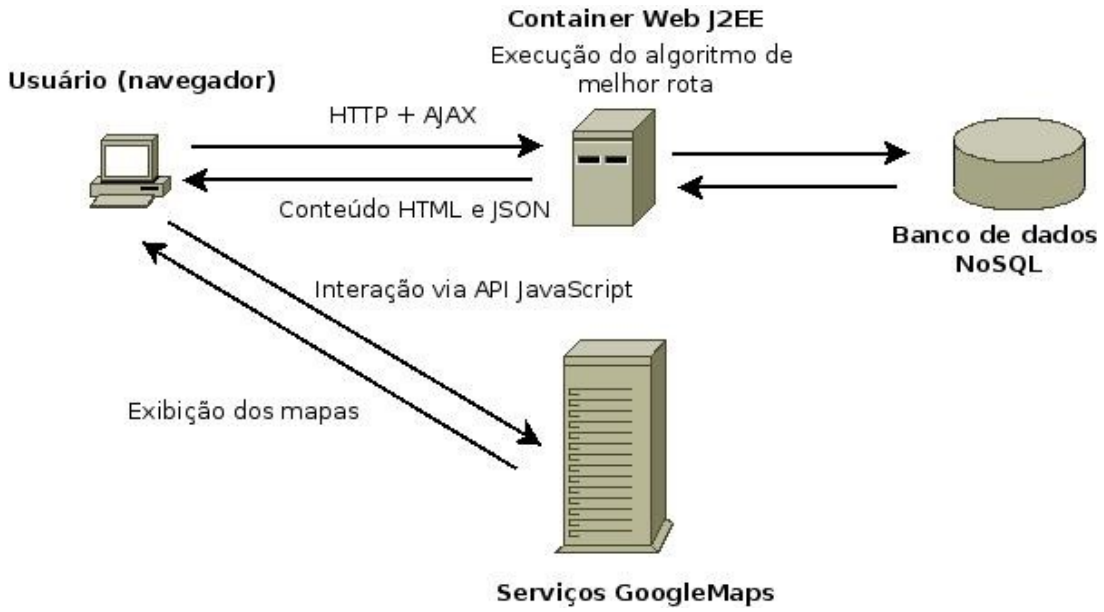

Figura 1 – Interação entre as tecnologias utilizadas no sistema

O algoritmo de Dijkstra (DIJKSTRA, 1959) é uma solução eficiente para o problema de se encontrar o caminho mínimo entre um vértice e os demais. Para cada vértice *v*, o algoritmo mantém a informação da menor distância d_v entre *v* e o vértice de partida. Inicialmente, é atribuído um valor infinito à d_v para todos os vértices *S*, exceto o vértice de partida, cujo d_v é zero. Em cada iteração, o algoritmo seleciona um vértice visitado, que ainda não pertence ao conjunto de vértices finalizados e que possui a menor distância ao ponto de partida até o momento (que inicialmente é o próprio vértice de origem). Uma fila de prioridades é utilizada para armazenar e selecionar esse vértice. A estrutura de dados escolhida para a construção da fila altera a complexidade algorítmica do problema. Selecionado o vértice *v*, este passa a pertencer ao conjunto de vértices finalizados *S*, uma vez que seu caminho mínimo à origem já foi encontrado. Em seguida, todos os seus vizinhos são visitados, armazenando em d_v a menor distância contabilizada de *v* à origem. O caminho seguido também é atualizado em cada iteração, armazenando o vértice precedente de *v* em $prec_v$, de modo que se possa obter o caminho mínimo ao final do processo. O algoritmo termina quando não há nenhuma atualização, ou seja, todos os vértices foram visitados. Caso se queira conhecer o caminho apenas entre dois vértices, o algoritmo pode ser encerrado assim que o destino for marcado como finalizado (ATALLAH; BLANTON, 2010).

3.1. Otimização do algoritmo de Dijkstra

O tempo de execução do algoritmo de Dijkstra depende fortemente da implementação da fila de prioridades *Q*. Uma escolha trivial é a utilização de um vetor, levando a uma complexidade de $O(|V|^2)$. Pode-se também utilizar um *heap* binário, obtendo o tempo de $O((|V|+|E|)\log|V|)$ (DASGRUPTA; PAPADIMITRIOU; VAZIRANI, 2009).

Existe ainda uma implementação mais eficiente, que utiliza uma estrutura de dados conhecida por *heap* de Fibonacci. Esta estrutura tem a vantagem de que as operações que não envolvem remoção de elementos podem ser executadas em um tempo amortizado de $O(1)$. Quando é utilizado uma fila de prioridades implementada com este *heap*, o algoritmo de Dijkstra fornece uma complexidade de $O(|V|log|V|+|E|)$ (DASGRUPTA; PAPADIMITRIOU; VAZIRANI, 2009). O *heap* de Fibonacci é uma coleção de árvores binomiais ordenadas como *heaps* mínimos. Apesar de sua eficiência, o *heap* de Fibonacci é

uma estrutura difícil de ser implementada na prática (CORMEN et al., 2008).

A decisão de qual implementação utilizar depende do tipo do grafo em questão. Neste projeto, o número de arestas (linhas de ônibus) é consideravelmente maior do que o número de vértices (cidades). Assim, tem-se um tipo de grafo denso e multi-arestas. De acordo com a complexidade de cada implementação, a melhor escolha é o *heap* de Fibonacci, devido a sua menor dependência ao número de arestas.

3.2. Roteamento com dependência temporal

O algoritmo de Dijkstra, na maneira como foi elaborado, não é capaz de resolver o problema de roteamento devido à dependência temporal do grafo. Na visão do algoritmo, cada vértice não é representado apenas por uma localidade, mas sim pela tupla `(localidade,horário)` (figura 2). Se, por exemplo, se está em Campinas às 14h00min, as possibilidades de linhas (arestas) para se tomar são diferentes do que estar no mesmo local às 16h00min. Essa representação causa um aumento considerável no número de vértices, já que, considerando que um dia possui 1440 minutos e que pode existir uma linha com horário de partida em qualquer minuto do dia, tem-se 1440 vezes o número de localidades O número de arestas também sofre este aumento proporcional, aumentando ainda mais a complexidade.

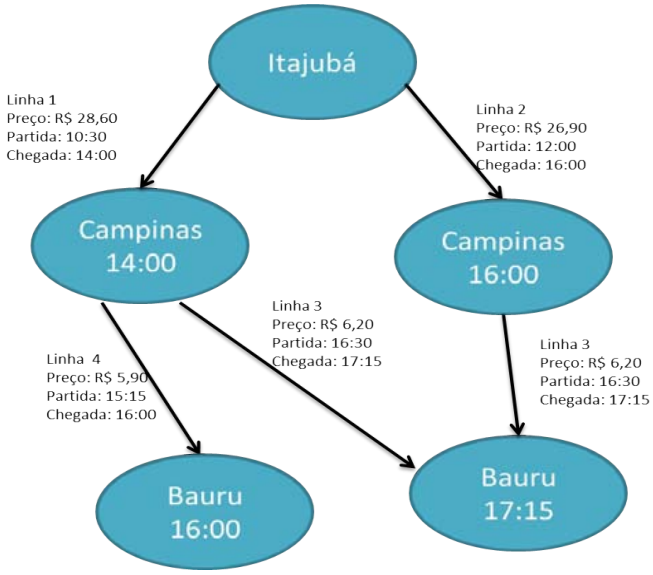

Figura 2 – O grafo visto pelo algoritmo durante a execução

Para este tipo de problema existe uma variante do algoritmo de Dijkstra conhecida por Caminho Mínimo com Dependência Temporal. Esta variante adiciona duas modificações na versão independente do tempo (PAJOR, 2009):

- Deve ser fornecido o tempo de partida τ como entrada adicional.
- Para calcular o peso de cada aresta, deve-se considerar o tempo (horário) corrente no vértice atual v. Sendo $e = (v, w)$ a aresta em que deseja-se conhecer o peso, então a função f_e da aresta e é calculada fornecendo o tempo τ mais o tempo de viagem $dist_v$ até v. Assim, o peso $w(e)$ é dado por $f(\tau + dist_v)$.

Para o caso da busca pelo menor tempo de viagem, o peso das arestas é o próprio tempo, sendo que a função *f* não precisa realizar nenhum tipo de conversão.

Para a busca pelo menor custo monetário, caso seja utilizado apenas o custo da viagem em si como o custo da aresta, o passageiro pode ser obrigado a esperar por muito

tempo em uma estação apenas para pegar a linha mais barata. Para evitar este caso, a função *f* também deve levar em conta o tempo em que o usuário está parado em uma estação esperando para a próxima viagem. Dessa forma, o algoritmo consegue eliminar longas esperas em troca de alguns centavos. Assim sendo, primeiro obtém-se uma função *p* que deve converter a variável de entrada, que é uma informação temporal (ou seja, a duração completa da viagem, incluindo os tempos de espera), em uma informação monetária, que é o custo utilizado no grafo. Essa conversão é possível por que o custo monetário de uma viagem é proporcional ao seu tempo, ou seja, quanto maior é o tempo de viagem, maior é o seu custo. Essa relação é ilustrada pelo gráfico na figura 3. Através da coleta de dados reais, foi obtida a função *p* que representa uma linha de tendência dos custos das viagens. A função relaciona o tempo total de viagem t_e até a aresta *e* com o preço *p*:

$$p(t_e) = 9 \times 10^{-8} t_e^3 - 1 \times 10^{-4} t_e^2 + 0{,}1744 t_e + 1{,}0771.$$

A partir dessa equação, o custo de cada aresta para a busca pelo menor custo monetário é dada pela heurística:

$$f(e) = k \times w(e) + p(t_e), \text{onde:}$$

K é um fator que relaciona o quanto o preço da viagem *w(e)* é mais importante que a espera $p(t_e)$.

A corretude do algoritmo de Caminho Mínimo com Dependência Temporal é comprovada desde que a função f possua a propriedade FIFO (*first in, first out*). O problema em questão apresenta esta propriedade, já que quanto mais cedo o passageiro chegar a uma localidade, mais cedo ele vai sair dela (PAJOR, 2009).

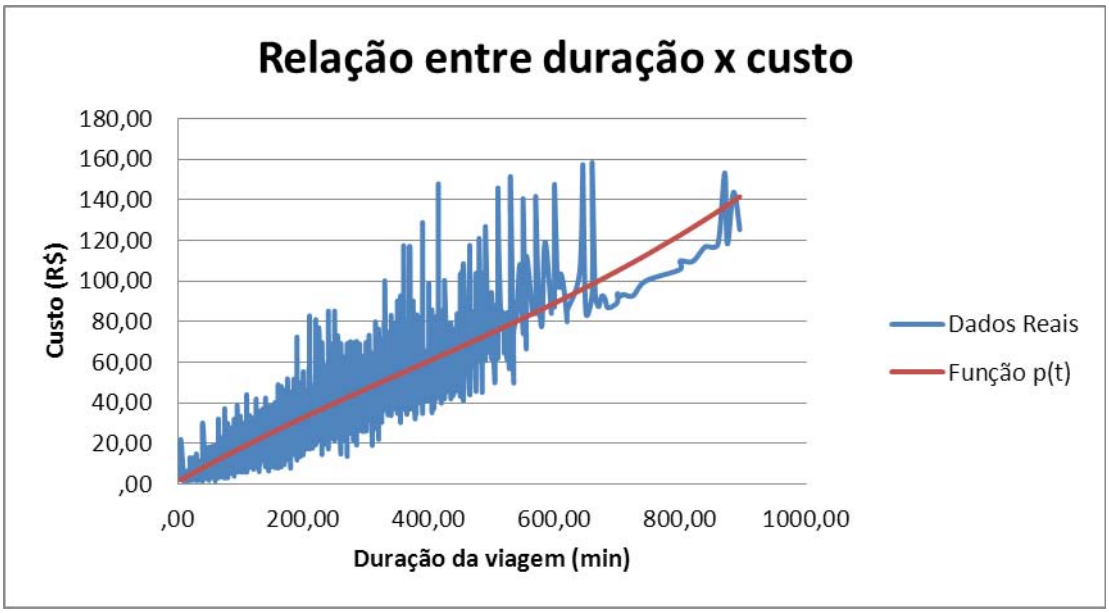

Figura 3 – Gráfico mostrando a relação entre duração e custo da viagem

4. Arquitetura do sistema

A fim de facilitar o desenvolvimento e a manutenção, além de aumentar a modularidade, o sistema de *software* foi projetado em três camadas distintas, cada qual com sua devida responsabilidade:

- Camada *web* (visualização): composta pelos arquivos HTML, Javascript, CSS, imagens, JSP, *servlets* e classes *controller*.
- Camada de serviços (*model*): composta pelas classes que implementam o algoritmo de busca, por objetos do domínio modelado e por classes de serviços específicos.
- Camada de dados: composta pelas classes que acessam os dados na base e pela própria base de dados.

Na camada *web*, do lado servidor, são utilizados os componentes J2EE: páginas JSP e *servlets*, com a aplicação do padrão MVC. No lado cliente, é utilizada a linguagem Javascript em conjunto com o *framework* jQuery para manipulação dos elementos das páginas e tratamento das chamadas Ajax, e a API do GoogleMaps para exibição das rotas encontradas.

Na camada de serviços, é modelado o sistema rodoviário na forma de um grafo contendo linhas de ônibus e localidades. Também é implementada a adaptação do algoritmo de Dijkstra para o problema proposto.

A camada de dados possui objetos gerenciadores dos dados para fornecer uma interface comum de acesso. A base é um banco de dados em grafos (*graph database*), capaz de manter um grafo em disco para rápido acesso e caminhamento.

A funcionalidade base do sistema é a busca por rotas mais rápidas ou de menor custo, ligando duas localidades. O diagrama de sequência da figura 4 ilustra uma busca realizada pela arquitetura proposta.

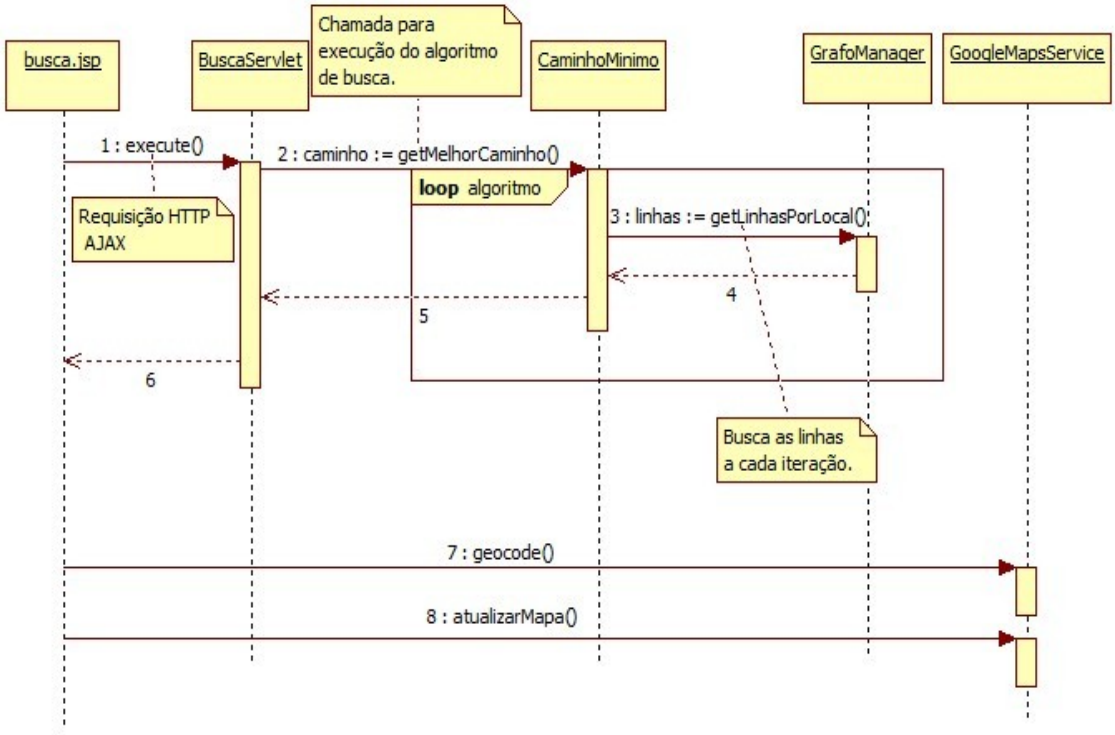

Figura 4 - Diagrama de sequência de uma busca por rotas no sistema

A página de busca `busca.jsp` inicia o processo, disponibilizando ao usuário um formulário para entrada dos dados e filtros para a pesquisa da rota. Os filtros são o local de origem, o local de destino, data da viagem e o parâmetro a ser minimizado: tempo ou custo. Nesta página em especial, a utilização de páginas *JavaServerPages* é uma melhor opção, devido às dificuldades encontradas atualmente na interação dos componentes *JavaServerFaces* com a API do GoogleMaps e código Javascript.

Devido ao fato de toda a API do GoogleMaps ser baseada em código Javascript e chamadas Ajax, a utilização deste tipo de requisição (assíncrona) fornece uma usabilidade melhor ao usuário, uma vez que apenas o mapa será atualizado de acordo com as opções fornecidas, e não toda a página, como ocorre na requisição comum do HTTP. Para facilitar a implementação do Ajax, normalmente utiliza-se uma biblioteca Javascript que encapsule as incompatibilidades com os diversos tipos de navegadores, como o jQuery.

A requisição Ajax invoca um *servlet*, que recebe os dados necessários para a busca da rota via parâmetros GET ou POST HTTP. O *servlet* invoca o serviço de busca responsável pela execução do algoritmo de custos mínimos. A execução do algoritmo requer um caminhamento sobre as diversas localidades (os vértices no grafo). A fim de se minimizar o acesso ao disco, uma camada de *cache* foi utilizada para armazenar as linhas mais acessadas. A política de *cache* mais eficiente é a de itens mais utilizados, onde as cidades mais visitadas são armazenadas na memória principal.

O acesso ao banco é feito por um objeto gerenciador que isola a implementação do banco, podendo ser utilizado qualquer tipo de banco de dados no *backend*. Para o banco de dados, a solução escolhida é a utilização de armazenamento NoSQL através de um banco de dados em grafos. Java possui uma implementação madura e bem documentada, o Neo4j, que é livre para propósitos não comerciais.

Ao final do algoritmo, o *servlet* recebe uma estrutura de dados com as rotas encontradas pela busca. Como o cliente requisitou o serviço utilizando Ajax, uma resposta apropriada deve ser criada pelo *servlet*. Normalmente, são utilizados dois tipos de retorno: um documento XML ou um documento JSON. Os documentos JSON são processados com maior facilidade pelo código Javascript (o próprio JSON é uma forma de estrutura Javascript). Assim, o *servlet* deve codificar a estrutura de dados recebida em forma de um documento JSON para a resposta. De posse da resposta, o código cliente (Javascript) atualiza o mapa, traçando as rotas encontradas.

5. Conclusões e resultados alcançados

O sistema de *software* apresenta ao usuário, em sua página inicial (figura 5), o formulário de busca contendo a localidade de origem, o destino, a data da viagem e o parâmetro a ser minimizado (tempo de viagem ou custo). Ao final do processamento da requisição, é exibida no mapa a melhor rota encontrada pelo sistema, juntamente com uma tabela contento os detalhes de cada etapa ou baldeação necessária.

O sistema também apresenta ao usuário a opção de selecionar o horário em que se deseja iniciar a viagem. Assim, a busca pela melhor rota iniciará a partir da escolha feita pelo usuário no formulário. Outra opção disponível é a que permite ao usuário personalizar sua rota, obrigando o sistema a tomar, durante o algoritmo, determinadas linhas selecionadas por ele no mapa.

Para os testes do algoritmo, foram gerados alguns grafos aleatórios possuindo até 5.000 localidades e 500.000 linhas, que é um tamanho que pode ser atingido com o sistema em produção. A interação foi feita considerando apenas um usuário acessando o sistema (uma única *thread*). O algoritmo respondeu com um tempo de resposta médio de 3*s* para os maiores grafos e mostrou-se eficiente durante os testes realizados, fornecendo tempos de resposta aceitáveis para a interação com o usuário em uma aplicação *web*. O banco de dados Neo4j também justificou a sua escolha, disponibilizando um acesso rápido e fácil ao grafo armazenado. Para a visualização, o uso da API do GoogleMaps e a interação via Ajax possibilitaram a criação de páginas ágeis e intuitivas, facilitando a utilização da aplicação por parte do usuário.

Figura 5 – Aplicação desenvolvida em funcionamento

Embora a aplicação desenvolvida tenha atingido os objetivos estabelecidos no início do projeto, algumas melhorias ainda podem ser realizadas para aprimorar o serviço, tais como:

- Desenvolvimento de uma área de acesso para empresas, onde elas possam gerenciar a suas linhas cadastradas;
- Desenvolvimento de uma área onde o usuário possa pesquisar as linhas cadastradas;
- Fornecer ao usuário outras opções além das rotas ótimas, como, por exemplo, as *n* melhores rotas. Assim, caso a rota de menores custos não seja, por algum motivo, viável ao usuário, este terá outras opções a escolher.

Referências

Atallah, M. e Blanton, M. *Algorithms and Theory of Computation Handbook – General Concepts and Techniques*. CRC Press, Nova York, 2010.

Bibeault, B. e Katz, Y. *jQuery em ação*. Manning, São Paulo, 2008.

Cormem, T. H.; Leiserson, C. E.; Rivest, R. L. e Stein, C. *Algoritmos – Teoria e Prática*. Elsevier, Rio de Janeiro, 2008.

Dasgrupta, S.; Papadimitriou, C. e Vazirani, U. *Algoritmos.* McGraw-Hill, São Paulo, 2009.

Diana, M. D. e Gerosa, M. A. (2010), Nosql na web 2.0: Um estudo comparativo de bancos não-relacionais para armazenamento de dados na web 2.0. *IX Workshop de Teses e Dissertações em Banco de Dados*, 68-75.

Dijkstra, E. (1959), *A Note on Two Problems in Connection with Graphs.* Numerische Mathematik, 1, 269-271.

Infoq. *Graph Databases, NoSQL and Neo4j.* 2010 (http://www.infoq.com/articles/graph-nosql-neo4j, 4, 2011).

JQuery. *jQuery: The Write Less, Do More, JavaScript Library*. 2006 (http://jquery.com, 6, 2011).

Mcarthur, K. *Pro PHP - Patterns Frameworks, Testing and More,* Apress, Berkeley, 2008.

Microsystems, S. *The Java EE 6 Tutorial - Advanced Topics*. Sun Microsystems, Santa Clara, 2009.

Pajor, T. *Multi-modal Route Planning. Institut für Theoretische Informatik*, Universität Karlsruhe, 2009.

Svennerberg, G. *Beginning Google Maps API 3*. Apress, Suécia, 2010.

Troelsen, A. *Pro C# 2008 and the .NET 3.5 Plataform*. Apress, Nova York, 2007.