ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Credenciado pelo Decreto de 06/07/2000 - D.O.U. nº 130 de 07/07/2000

Alex Coelho

UTILIZAÇÃO DO ALGORITMO DE DIJKSTRA PARA RESOLVER O PROBLEMA DO CAMINHO MÍNIMO EM MAPAS CONSTRUÍDOS COM O FORMATO SCALABLE VECTOR GRAPHICS

Palmas

2004

ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Credenciado pelo Decreto de 06/07/2000 - D.O.U. nº 130 de 07/07/2000

Alex Coelho

UTILIZAÇÃO DO ALGORITMO DE DIJKSTRA PARA RESOLVER O PROBLEMA DO CAMINHO MÍNIMO EM MAPAS CONSTRUÍDOS COM O FORMATO SCALABLE VECTOR GRAPHICS

"Trabalho de Conclusão de Curso apresentado como requisito parcial de disciplina Prática de Sistemas de Informação II do curso de Sistemas de Informação, orientado pelo Prof. M.Sc. Fabiano Fagundes"

Palmas

2004

ALEX COELHO

UTILIZAÇÃO DO ALGORITMO DE DIJKSTRA PARA RESOLVER O PROBLEMA DO CAMINHO MÍNIMO EM MAPAS CONSTRUÍDOS COM O FORMATO SCALABLE VECTOR GRAPHICS

"Trabalho de Conclusão de Curso apresentado como requisito parcial de disciplina Prática de Sistemas de Informação II do curso de Sistemas de Informação, orientado pelo Prof. M.Sc. Fabiano Fagundes"

BANCA EXAMINADORA

_____

Prof. M.Sc Fabiano Fagundes
Centro Universitário Luterano de Palmas

_____

Prof. Jackson Gomes de Souza
Centro Universitário Luterano de Palmas

_____

Prof. Ricardo Marx Costa Soares de Jesus
Centro Universitário Luterano de Palmas

Palmas

2004

AGRADECIMENTOS

Primeiramente gostaria de agradecer a todos os professores do curso de Sistemas de Informação do CEULP/ULBRA, dois em especial. Principalmente a meu orientador e amigo, Fabiano, que além de professor de disciplinas no decorrer do curso, se mostrou também um professor na forma de encarar a vida com seus conselhos, e ao professor Jackson, que nos momentos mais complicados do desenvolvimento de meu trabalho me auxiliou, sendo uma pessoa que se tornou essencial para a conclusão deste, sempre prestativo e atencioso, procurando me auxiliar no que fosse preciso. Não poderia esquecer de agradecer a todos meus amigos, sem citar nomes para não ser injusto com todos aqueles que sempre foram pessoas marcantes e presentes na minha vida. Agradecer a minha namorada Fernanda que me mostrou o verdadeiro sentido de gostar de alguém e por todos os momentos que esteve do meu lado, além de me ajudar a superar todas as dificuldades que passamos nestes últimos anos. Estaria sendo injusto se deixasse de mencionar meus colegas de curso principalmente os branquinhos (André, Carlos, Edeilson, Jorge, Leandro, Lucas e Jânio, que não é branquinho, mas sempre colaborou comigo) que em momentos cruciais do curso se mostraram mais que colegas e se tornaram verdadeiros amigos, lembrando que ninguém melhor que eles sabe o que representa este momento e o que passamos para estar aqui. E, por último, agradecer às pessoas que mais fizeram por mim, não só na minha graduação, mas em toda minha vida: à minha família (mãe, pai, Adriane, Chesy, André, Lanny e sobrinhos) gostaria de dizer meu muito obrigado por me entenderem e até por muitas vezes me aturarem, lembrando sempre que este momento é uma vitória não só minha, mas de todos nós. Obrigado a todos.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

API – *Aplication Programing Interface*

DAG - *Directed Acyclic Graph*

DOM – *Document Object Model*

HTML – *HyperText Markup Language*

RDF - *Resouce Description Framework*

SGML – *Standard General Markup Language*

SOAP - *Simple Object Aplication Protocol*

SVG – *Scalable Vector Graphics*

XML – *eXtensible Markup Language*

W3C - *World Wide Web Consortium*

RESUMO

Diante do avanço tecnológico e popularização do computador, surge a possibilidade da utilização deste como forma de auxílio às mais diversas atividades e serviços, dentre as quais a localização geográfica. O formato texto do SVG, baseado em XML, possibilita que sua apresentação, além de mais veloz e compacta, permite a manipulação e a alteração dinâmica das imagens construídas neste formato e o registro das interações sofridas. Este tipo de imagem aliada à estrutura de dados representada através de um grafo possibilita a utilização de algoritmos de processamento do menor caminho. Neste trabalho é implementado um protótipo que auxilia a localização do menor caminho entre determinados pontos representados por um grafo em imagens SVG, utilizando o algoritmo de *Dijkstra* implementado com a linguagem de *script* Javascript e a API DOM.

Palavras chave: Grafos, Algoritmo de *Dijkstra*, XML, SVG.

ABSTRACT

Ahead the technologic progress and computer popularization, appears the possibility of utilization as a way of assistence to the most diverse activities and services, like geographic localization. The text format of SVG, based in XML, makes possible its apresentation, beyond more ligth and compact, allow the manipulation, and update of the images constructed in this format, and the interaction register made. This image with the data estructure represented by a graph makes possible the utilization of processing algoritms of short path. In this work is implemented a prototype that assists the localization of the short path between certain points represented by one graph in SVG images, using, for this, Javascript languages and DOM API

Keywords: Grafs, Dijkstra's Algoritm, XML, SVG.

1 INTRODUÇÃO

Com a popularização dos computadores, nisto se incluem computadores portáteis e celulares, novas possibilidades passaram a ser exploradas por fabricantes e empresas especializadas na disponibilização de serviços e *softwares*. Desde simples serviços, como a aquisição de jogos para celulares, a mapas complexos de perímetros urbanos são oferecidos aos usuários destes dispositivos. Na construção destes mapas podem ser utilizadas imagens construídas com o formato SVG (*Scalable Vector Graphics*).

Este trabalho consiste na utilização do SVG para a representação de um mapa sobre o qual se determinará o menor caminho entre dois pontos, cálculo este que será feito através da representação dos pontos do mapa em um grafo sobre o qual será aplicado o algoritmo de menor caminho de Dijkstra (1959).

Em conjunto com o formato SVG é comum a utilização de linguagens *scripts* para a construção e alteração, de forma dinâmica, dos elementos presentes nas imagens. Assim, deve-se mencionar a utilização da linguagem Javascript no trabalho para a inserção e alteração na imagem, bem como a realização do processamento do algoritmo de Dijkstra e o registro e consulta dos dados obtidos da interação dos usuários com a imagem em arquivos XML (*eXtensible Markup Language*) utilizando para isto a API (*Aplication Programing Interface*) DOM (*Document Object Model*).

Na seção “Revisão de Literatura” são apresentadas as tecnologias utilizadas no trabalho destacando-se a descrição de alguns dos elementos SVG, além de alguns aspectos da linguagem Javascript como a utilização da API DOM. Outro ponto a ser mencionado é a

forma de armazenamento em arquivos XML. Estas formas de armazenamento são descritas na “Revisão de Literatura”.

A seção “Implementação” neste trabalho, descreve as funções criadas durante a implementação do protótipo, além de expor algumas dificuldades encontradas no processo. O trabalho se encerra com algumas considerações na Conclusão, a apresentação das referências bibliográficas utilizadas no trabalho, além da seção de anexos com o código das funções e da imagem SVG.

2 REVISÃO DE LITERATURA

Para a realização deste trabalho foi necessário o entendimento dos algoritmos que auxiliam na busca pelo menor caminho entre pontos modelados através de um grafo. Um destes algoritmos foi implementado utilizando a linguagem *script* Javascript e DOM (*Document Object Model*) sobre mapas construídos no formato SVG (*Scalable Vector Graphics*), além de utilizar arquivos XML (*eXtensible Markup Language*) para o armazenamento dos dados. As próximas seções apresentarão os conceitos levantados neste estudo.

2.1 Grafos

Conceitualmente, "um grafo consiste em um conjunto de nós - ou vértices - e um conjunto de arcos - ou arestas -" (TENENBAUM et. al., 1995), sendo uma das formas de representar estruturas de dados computacionalmente. A teoria dos grafos, como é mais conhecida, tem importância ímpar na representação de problemas como fluxo de rede, roteamento e caminho mínimo (RABUSKE, 1995).

Os grafos podem ser representados como G=(V,E), onde V representa o conjunto de vértices e E o conjunto de arestas (*edges*) (PREISS, 2004) conforme demonstrado graficamente na Figura 1.

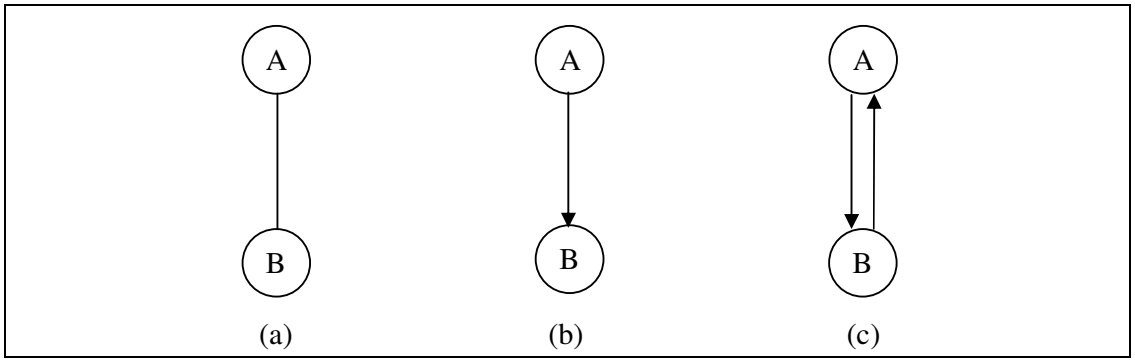

Figura 1 – Exemplos de grafos.

Algumas definições devem ser apresentadas de maneira a deixar clara a idéia de um grafo. A Figura 1(a) demonstra uma seqüência {A, B}, a qual consiste de um conjunto de arestas (A, B), no caso, os vértices existentes nas extremidades das arestas consistem no par de vértices de um grafo.

A representação ordenada destas arestas contidas no grafo forma um grafo orientado ou dígrafo, conforme apresentado na Figura 1(b) e 1(c). Uma definição que deve ser levada em consideração é a de que um grafo não necessariamente precisa ser uma árvore, porém uma árvore é um grafo (TENENBAUM et al, 1995).

Outra definição importante para um grafo é a incidência de um vértice sobre um arco. Por exemplo, para que um vértice *a* incida em uma aresta *x*, este deve ser um dos dois vértices do par ordenado que constituem a aresta *x*. Tomando como exemplo a Figura 1(b), têm-se dois vértices (A e B) que incidem sobre uma aresta. Esta incidência mede o grau de entrada e grau de saída de um vértice. O grau de entrada de um vértice consiste no número de arestas que tem um vértice *n* como cabeça, e o grau de saída no número de arestas que tem um vértice *m* como terminação (TENENBAUM et al, 1995).

"Um caminho em um dígrafo é uma seqüência não vazia de vértices" (PREISS, 2000). Os caminhos podem ser classificados como (TENENBAUM et al, 1995):

- cíclicos: o nó inicial é o mesmo final;
- laço: é um ciclo de tamanho igual a um ou seja, caminho de um nó para si mesmo;

- acíclicos: ao contrário dos cíclicos, são aqueles que não contêm um ciclo; estes, quando orientados, também são chamados DAG (*Directed Acyclic Graph*).

Grafos orientados denotam sentido ou ordem de um caminho – ou percurso - em um grafo, conforme pode ser visto na Figura 2 (a). A existência de ciclos em um grafo consiste em um problema que pode vir a afetar uma busca. No caso, um ciclo consiste na repetição da ocorrência de um vértice num caminho, podendo assim gerar como conseqüência disto um *loop* infinito (RABUSKE, 1995).

Grafos não-orientados consistem em vértices conectados por arestas não direcionadas, ou seja, uma aresta que não possui direção, o que pode ser percebido na Figura 2 (b).

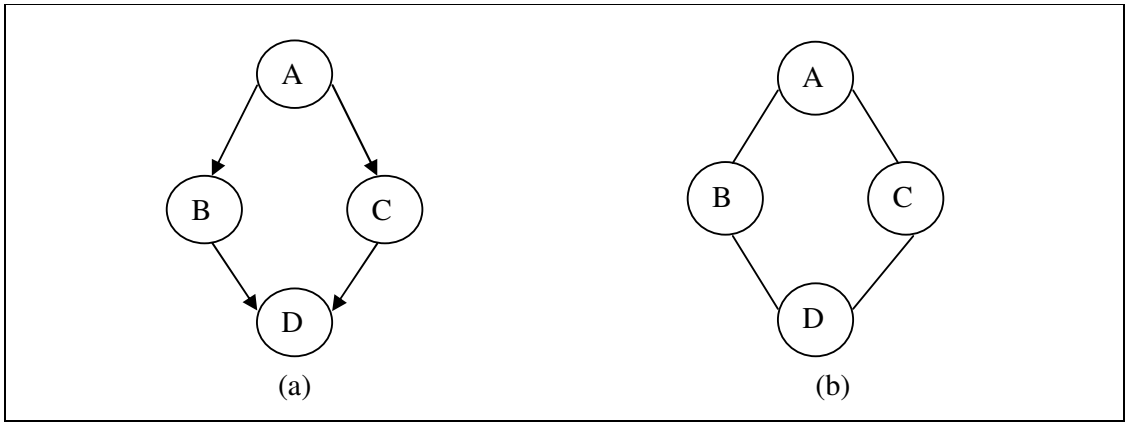

Figura 2 – Grafo orientado (a) e não-orientado (b).

Em grafos não-orientados não podem existir laços (PREISS, 2000). Como o trabalho consiste na manipulação de grafos orientados, não serão apresentados estudos mais aprofundados sobre os grafos não-orientados, mas lembrando que a este tipo de grafo também são aplicados algoritmos de menor caminho.

O primeiro problema representado pela teoria dos grafos é atribuído a Leonhard Euler em 1730 (GUIMARÃES, 2004). No caso, o grafo foi aplicado para solucionar o problema das pontes da cidade de *Könisberg* na Prússia. A cidade possuía um rio e neste existiam duas ilhas e sete pontes que as interligavam à cidade. O problema consistia em

encontrar um caminho que percorresse todas as pontes apenas uma vez e retornasse a seu ponto de origem.

Euler utilizou grafos para resolver o problema considerando que as partes de terra eram vértices e as pontes arestas. Um determinado circuito que percorre cada aresta de um grafo exatamente uma vez é chamado de circuito euleriano e um grafo que possui tal circuito é chamado de grafo euleriano. Euler provou que o problema citado era impossível de ser resolvido através do argumento de que um circuito qualquer deve chegar à um vértice e sair dele o mesmo número de vezes. Logo, para que exista um circuito euleriano no grafo, deve haver um número par de arestas com extremidade neste vértice. Como não existia em determinados pontos do grafo esta condição, concluiu-se que não era possível encontrar um circuito euleriano sendo o problema impossível de ser resolvido (GUIMARÃES, 2004). A Figura 3 apresenta a representação do grafo criado por *Euler*.

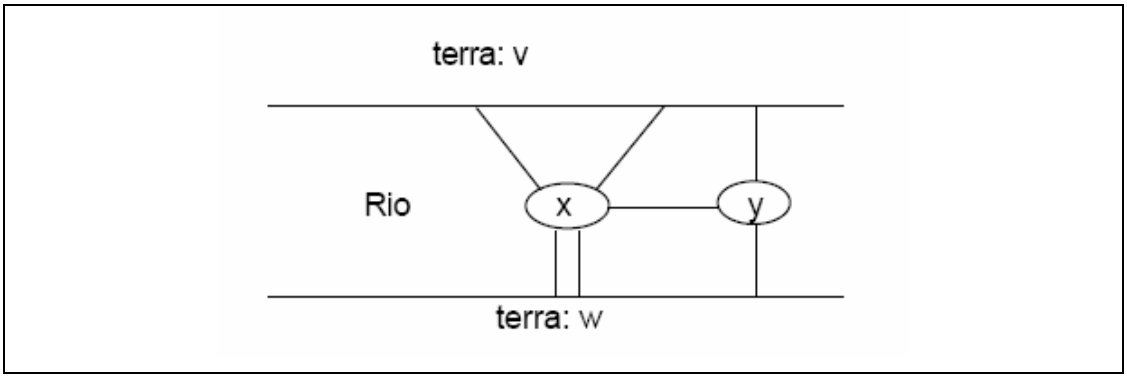

Figura 3 – Problema da Ponte de *Köninsberg*.

Segundo Tenenbaum et al (1995), em um grafo um vértice é considerado adjacente a um outro vértice se entre estes existir a incidência de uma aresta, sendo um vértice denominado sucessor e o outro predecessor. A relação em um conjunto qualquer consiste em "uma seqüência qualquer de pares ordenados de elementos" (TENENBAUM et al, 1995).

A Figura 4 demonstra um grafo e sua relação em um conjunto A, em que A={3,5,6,8,10,17}, e sua relação é R={<3,10>,<5,6>,<5,8>,<6,17>,<8,17>,<10,17>}. Neste relacionamento um número pode ser associado a cada aresta. Este elemento do grafo é denominado peso. Este tipo de grafo é chamado de grafo ponderado ou rede

(TENENBAUM et al, 1995). A Figura 4 apresenta o grafo aplicado sobre o conjunto citado anteriormente.

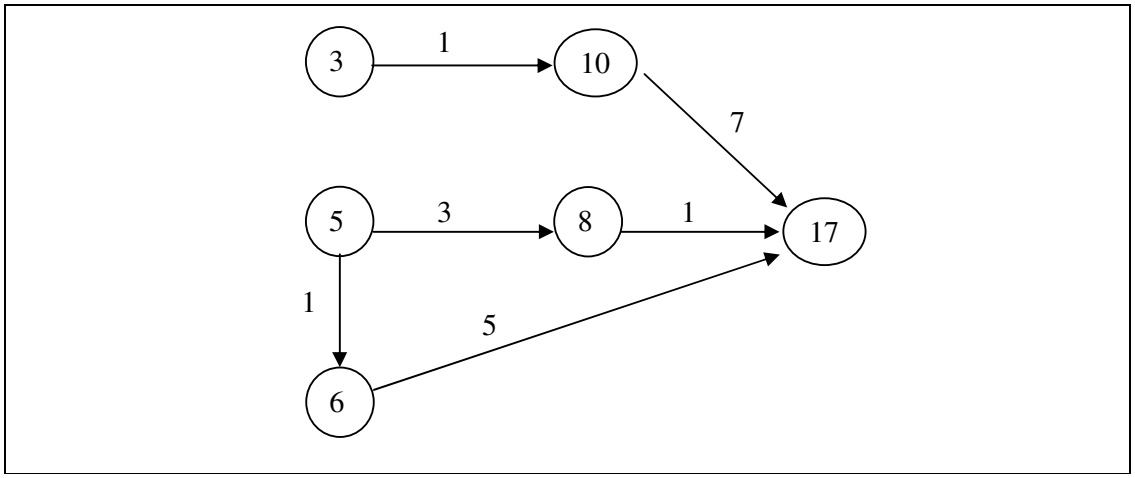

Figura 4 – Relações, peso e grafos ponderados (TENENBAUM et al, 1995).

No grafo exemplificado na Figura 4, para o valor do peso das arestas, assumiu-se o resto obtido da divisão dos valores atribuídos ao sucessor e o predecessor de cada par ordenado. Por exemplo, na aresta que sofre a incidência dos vértices 3 e 10, o valor obtido como peso foi 1 (TENENBAUM et al, 1995).

As informações em grafos estão vinculadas a arestas e aos vértices e um grafo deste tipo é chamado de grafo rotulado (PREISS, 2000). Os vértices são os elementos de um grafo a serem percorridos, que de alguma maneira devem possuir os vértices adjacentes a ele e estes por sua vez também serão percorridos, aplicando-se a mesma regra. Assim tem se a idéia de percurso em um grafo e este pode ser percorrido tanto em sua profundidade quanto em sua largura. Isto, computacionalmente, é feito por meio de algoritmos voltados a trabalhar os percursos em grafos.

Com a utilização destes algoritmos, surgem inúmeras possibilidades, dentre estas se pode citar a de se encontrar o menor caminho entre vértices de um grafo. Estes algoritmos

são utilizados para realizar o processamento e cálculo deste percurso, logo, estes são mais conhecidos como algoritmos de menor caminho, tema da próxima seção.

2.2 Algoritmo de menor caminho

Os algoritmos de menor caminho - ou caminho mínimo - são aplicados sobre grafos ponderados, em que se deseja achar o menor caminho entre dois nós. O menor caminho consiste na soma dos pesos das arestas de um grafo a partir de um determinado vértice (GUIMARÃES, 2004).

Considerando um grafo G, que é composto por um conjunto de vértices G={A, B, C, D, E, F} interligados por arestas com seus respectivos pesos, no qual se deseja obter o menor caminho entre dois vértices A e E, tomando como início o vértice A, é obtido como resultado o caminho denotado no grafo da Figura 5.

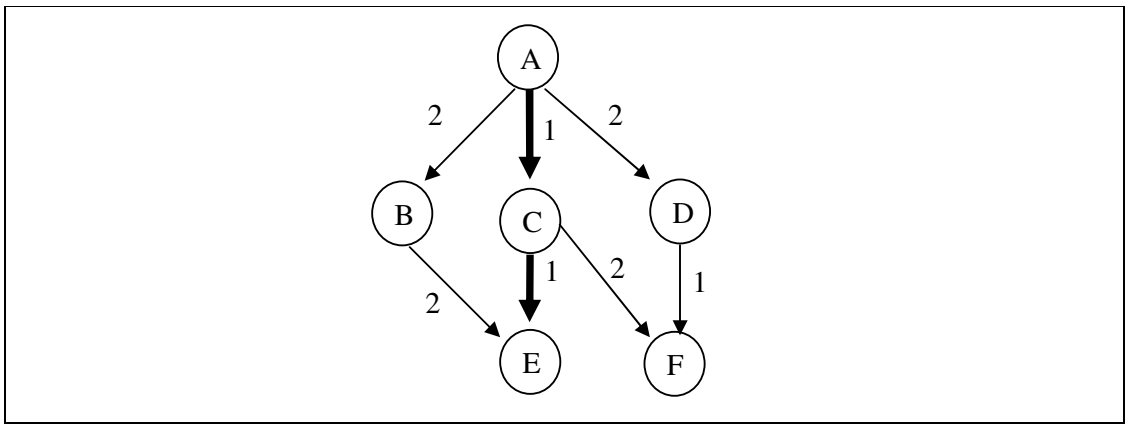

Figura 5 – Menor caminho entre dois vértices de um grafo.

Conforme pode ser constatado na Figura 5, o menor caminho entre o vértice A e o vértice E é o caminho A-C-E, tendo como resultado da soma dos pesos o valor 2.

Os algoritmos de menor caminho são comumente utilizados para efetuar processamento sobre problemas como (GUIMARÃES, 2004):

- encontrar o menor caminho entre cidades;

- solução para labirintos;
- eliminação de código morto no desenvolvimento de programas;

Para a representação e implementação dos grafos, podem ser utilizadas técnicas como (PRIESS, 2000):

- listas de adjacências: são listas encadeadas, sendo uma para cada vértice e o espaço total necessário para a representação destas listas consiste na quantidade de arcos do grafo;
- matrizes de adjacências: é o método de representação mais simples que consiste em uma matriz da quantidade de vértices *n* multiplicado por esta mesma quantidade no caso, G=(V,E) onde a quantidade de espaço total é n^2, e os valores expressos nessa matriz consistem em 0 (zero) e 1 (um) , no qual o valor 1 (um) representa a existência da adjacência entre dois vértices e caso contrário o valor será 0 (zero).

A utilização de listas de adjacências fornece uma maneira mais compacta de representar grafos chamados esparsos, que são aqueles em que o número de arestas é consideravelmente menor que o número de vértices ao quadrado, ou seja, enquanto existirem poucas ligações entre os vértices, já que, com a utilização de listas, a velocidade do processamento está diretamente ligada ao número de arestas presentes no grafo, pois o cálculo para a velocidade de processamento é (V*E).

Já a utilização de matrizes de adjacência pode ser mais indicada quando o grafo é denso, ou seja, quando o número de arestas do grafo for próximo de n^2, possibilitando com maior velocidade, saber se existem arestas conectadas a dois vértices quaisquer, pois a quantidade de arestas não afeta na velocidade do processamento de matrizes, já que, a fórmula para calcular a velocidade do processamento de matrizes é $|n^2|$ (CORMEN et al., 2001). A Figura 6 demonstra a representação de um mesmo grafo utilizando para isto lista de adjacência e matriz de adjacência.

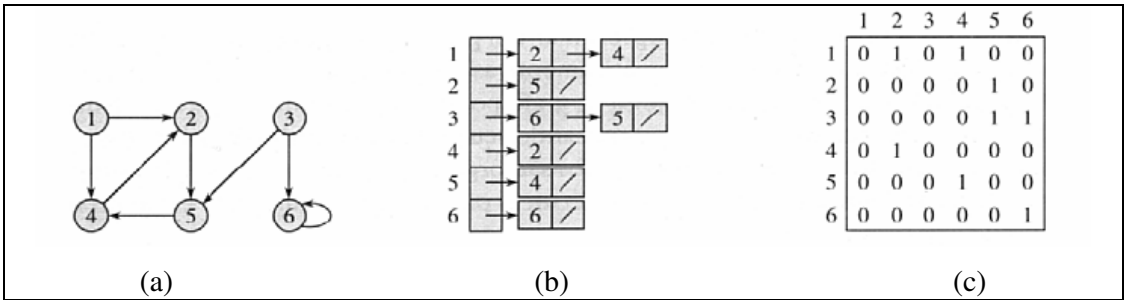

Figura 6 – Grafo (a) representado através de lista (b) e matriz de adjacência (c) (CORMEN et al, 2001).

Para grafos que apresentam uma realidade diferente, em que os pesos das arestas que o compõem podem ser maiores que um, solucionam-se estes problemas trabalhando-se com as informações contidas nos vértices e arestas aliadas a possibilidades como a utilização de listas ou matrizes de adjacência.

A representação de listas de adjacência, conforme apresentado na Figura 6, consiste em uma lista com a quantidade de vértices do grafo. Para cada posição *u* desta lista existe uma ligação para todos os vértices que são adjacentes a *u* no grafo.

A representação de matriz de adjacência consiste em uma matriz cujas dimensões seja a quantidade de vértices em ambas as dimensões da matriz, em que, caso exista adjacência entre determinados vértices, estes recebem o valor 1 (um), indicando esta adjacência, caso contrário recebem o valor 0 (zero). Quanto ao seu armazenamento, matrizes de adjacência levam uma grande vantagem, pois são representadas somente com um *bit* de entrada, caso não existam rótulos nas arestas.

Outra versão de matriz de adjacência é a matriz de peso, que consiste em uma matriz similar à matriz de adjacência, sendo que, ao invés de se armazenar o valor 1 (um), para indicar a existência de adjacência entre os vértices, são utilizados os valores referentes aos pesos das arestas que compõem a adjacência entre os vértices sucessor e predecessor.

Apresentados os conceitos de listas e matrizes de adjacência, abre-se espaço para a utilização de algoritmos específicos que se utilizam destas técnicas, que se propõem a resolver o problema de encontrar o menor caminho entre vértices de um grafo. Existem algoritmos que se prestam a solucionar os mais variados tipos de problemas relacionados a

caminhos, mas este estudo está voltado para o problema de caminhos mais curtos de origem única.

Dentre estes, pode-se citar o algoritmo de Bellman-Ford (1957) que se propõe a resolver o problema de caminho mais curto de única origem de forma mais abrangente que outros, pois permite a utilização de pesos tanto negativos quanto positivos nas arestas que compõem o grafo (CORMEN et al., 2001), além de também permitir encontrar a distância de um nó para todos os outros é vice-versa (de todos para todos), porém este tem seu tempo de execução prejudicado.

Por exemplo, no caso do algoritmo de Dijkstra que será apresentado, o algoritmo têm complexidade $O(V^2)$, que consiste no número de vértices do grafo elevado ao quadrado, enquanto o algoritmo de *Bellman-Ford* tem complexidade O(VE), ou seja, a quantidade de vértices multiplicado pela quantidade de arestas presentes no grafo. Pode-se citar como exemplo um grafo contendo 5 (cinco) vértices e 8 (oito) arestas, no qual, o tempo de processamento para Dijkstra seria $5^2= 25$, e para *Bellman-Ford* 5*8=40, porém este exemplo serve meramente uma forma de exemplificar o processamentos dos algoritmos já que, a notação O não retorna o tempo exato.

Como este trabalho consiste em encontrar a menor distância entre pontos de um mapa, sendo que estas distâncias nunca serão negativas, são utilizados grafos ponderados, cíclicos e de valores não negativos para as arestas, nos quais foi aplicado o algoritmo de Dijkstra que se mostrou mais aconselhável para o problema, levando-se em consideração a vantagem no tempo de processamento conforme mencionado. Assim, este será citado na seção seguinte.

2.2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra (1959) é também conhecido como um “algoritmo guloso” por sempre escolher o vértice mais leve ou mais próximo ao vértice a que faz adjacência, além de ser um algoritmo que percorre todos os vértices de um grafo, a fim de se conhecer à distância do vértice fonte a todos os outros vértices do grafo. O algoritmo objetiva

resolver o problema do menor percurso entre os vértices de um grafo, com origem única, em grafos ponderados no qual os pesos das arestas são não-negativos (PREISS, 2000).

O algoritmo percorre todos os vértices do grafo a partir de um determinado vértice inicial ou fonte. “A característica essencial do algoritmo de *Dijskstra* é a ordem na qual os caminhos são determinados: Os caminhos são encontrados na ordem dos comprimentos ponderados, começando pelo mais curto e prosseguindo até o mais longo” (PREISS, 2000).

O algoritmo de *Dijskstra* mantém três informações essenciais na estrutura de cada vértice de um grafo (PREISS, 2000):

- um *flag* determinando se o caminho mais curto do vértice fonte até um vértice qualquer é conhecido;
- o comprimento do caminho mais curto do vértice fonte ao vértice destino. Ao ser iniciado o algoritmo, todos os comprimentos dos vértices do grafo possuem um valor desconhecido que é alterado durante a execução do algoritmo;
- o predecessor do vértice destino no caminho mais curto do vértice fonte a seu destino, sendo que ao início o predecessor de um vértice destino não é conhecido.

O algoritmo de Dijkstra utiliza a técnica de relaxamento de arestas que consiste em verificar se existem possibilidades para melhorar um caminho mais curto até um determinado vértice. Esse procedimento é aplicado |V| vezes, no qual V consiste no conjunto de vértices do grafo e, no final, tem-se o caminho mínimo da fonte a cada vértice do grafo (PREISS, 2000). A Figura 7 apresenta o pseudocódigo do algoritmo de Dijkstra que resolve o problema de caminho mais curto entre vértices.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5     do u ← EXTRACT-MIN(Q)
6        S ← S ∪ {u}
7        for cada vértice v ∈ Adj[u]
8           do RELAX(u, v, w)
```

Figura 7 – Pseudocódigo do algoritmo de *Dijsktra* (CORMEN et al, 2000).

No início do pseudocódigo da Figura 7 tem-se a definição do algoritmo de Dijkstra que recebe os parâmetros: `G,` que consiste no grafo com `n` vértices que podem ser representados como lista, matriz de adjacência ou mesmo matriz de peso; o segundo parâmetro consiste em uma função `w` que tem como objetivo retornar o peso das arestas de um determinado vértice a outro e o terceiro parâmetro é o vértice fonte da pesquisa. Na primeira linha da Figura 7 o método `INITIALIZE-SINGLE-SOURCE` inicializa as informações referentes a cada vértice do grafo a serem preenchidas durante o processamento do algoritmo como pode ser constatado na Figura 8.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1 for cada vértice v ∈ V[G]
2      do d[v] ← ∞
3      π[v] ← NIL
```

Figura 8 – Inicialização do algoritmo de Dijkstra (CORMEN et al, 2000).

O funcionamento deste método consiste em fazer com que sejam iniciados o vetor que indique a distância do vértice naquele índice ao vértice fonte e o vetor que indique se o vértice em determinado índice foi visitado. No caso, estes vetores em cada posição do índice indicam os vértices do grafo, que recebem um valor extremamente grande ou infinito para o vetor de distância a fim de que quando ocorra o teste para verificar as distâncias de um vértice esta seja maior que qualquer outra presente no grafo e algum valor que determine que o vértice a que o índice se refere ainda não foi visitado no vetor que representa os vértices visitados.

Na Figura 7, segunda linha, o vértice fonte `s` recebe zero como distância até ele mesmo. Na terceira linha a fila de prioridade mínima Q, sendo que, alguma prioridade deve ser definida para a fila e a ordenação dos vértices de acordo com esta prioridade de cada um na fila recebe uma fila de vértices adjacentes a ser percorrida. O laço é iniciado na quarta linha sendo que é realizado enquanto existirem vértices adjacentes na fila Q. Na quinta linha um vértice `u` é extraído de Q. Na primeira passagem do *loop*, o vértice `u` será o vértice fonte (COMEM, 2000).

Um outro laço é iniciado na sétima linha do algoritmo, sendo que este é executado enquanto existirem vértices adjacentes ao vértice `u`. Na oitava linha ocorre o processo de

"relaxamento de arestas". Neste é realizado uma verificação se o vértice já foi visitado e se a distância de um vértice `u` mais o peso da aresta entre este e um adjacente `v` é maior que a distância existente para o vértice `v`; caso isto seja verdade a distância existente para um vértice `v` até o vértice fonte recebe a soma entre a distância existente para um vértice `u` e o peso da aresta entre ambos. Ainda no processo de "relaxamento de aresta" o vértice `v` recebe como seu predecessor o vértice `u`. Isto ocorre até que não existam mais adjacentes ao vértice retirado da fila. O código referente ao "relaxamento de arestas" é apresentado na Figura 9.

```
RELAX (u, v, w)
1 if d[v] > d[u] + w(u,v)
2      then d[v] ← d[u] + w(u,v)
3           π[v] ← u
```

Figura 9 – Técnica de relaxamento de arestas (CORMEN et al, 2000).

A Figura 10 apresenta o percurso em um grafo ponderado cíclico com aresta de peso positivo, no qual se utiliza o algoritmo de Dijkstra para descobrir o menor caminho a um vértice destino.

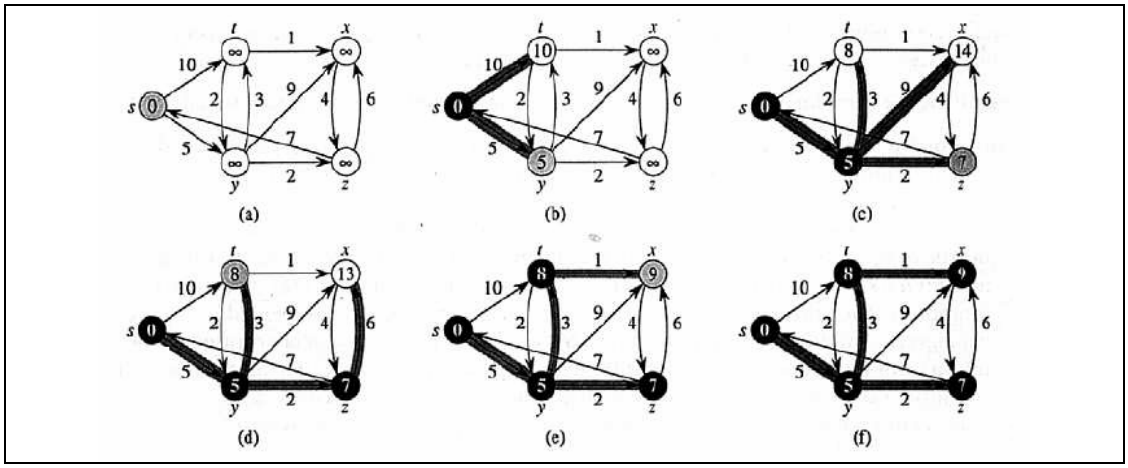

Figura 10 – Percorrendo o algoritmo de *Dijkstra* (CORMEN et al, 2000).

Na Figura 10 (a) a origem é o vértice s. Depois de todo o processo de inicialização, as estimativas de caminho mais curto ou distância até o vértice fonte são mostradas dentro de cada vértice. No momento inicial, todos os vértices possuem um valor infinito dele ao vértice fonte. No momento da execução da primeira iteração, linhas 4 a 8 presentes na Figura 7, os vértices adjacentes ao vértice fonte recebem o valor do peso de suas arestas, assumindo-se como vértice u, o de menor valor entre os vértices adjacentes; no caso, o

vértice que assume o valor 5 na Figura 10 (b). Isto ocorre até que todos os vértices do grafo sejam percorridos - Figura 10 (f) - tendo como resultado uma estimativa da distância de cada vértice ao vértice fonte.

Para a implementação do algoritmo de Dijkstra neste trabalho foi adotada a linguagem de *script* Javascript, o XML para o armazenamento dos dados, o SVG como elemento de interface, e o DOM para manipulação destes arquivos e que é tema da próxima seção.

2.3 DOM (*Document Object Model*)

"O DOM é um modelo que disponibiliza uma interface de programação de aplicativos (API), independentemente de linguagem ou plataforma, para documentos HTML (*HyperText Markup Laguange*) e XML" (W3C, 2004c). Criado em agosto de 1997 e especificado pela *World Wide Web Consortium*, segue a estrutura de árvores para manipulação e consulta dos chamados "documentos" como são mais conhecidos no DOM. Os documentos manipulados utilizando DOM apresentam uma hierarquia de objetos, mais conhecidos como nós, sendo que estes nós podem ou não conter filhos.

O objetivo do DOM foi o de padronizar uma API para ser utilizada em diversos ambientes, aplicativos e linguagens de programação. Com a utilização do DOM o documento pode ser processado e o resultado do processamento pode ser incorporado ao estado anterior do mesmo documento de forma dinâmica.

A especificação DOM consiste em um núcleo (*core*), que fornece objetos de baixo nível para representar documentos estruturados (McGRATH, 1999). O DOM trabalha com a idéia de níveis ou camadas, sendo que, a cada alteração que ocorra no DOM novas camadas ou níveis são acrescentados à API. Hoje a API DOM encontra-se em seu terceiro nível com completo suporte a arquivos XML 1.0 (W3C, 2004c).

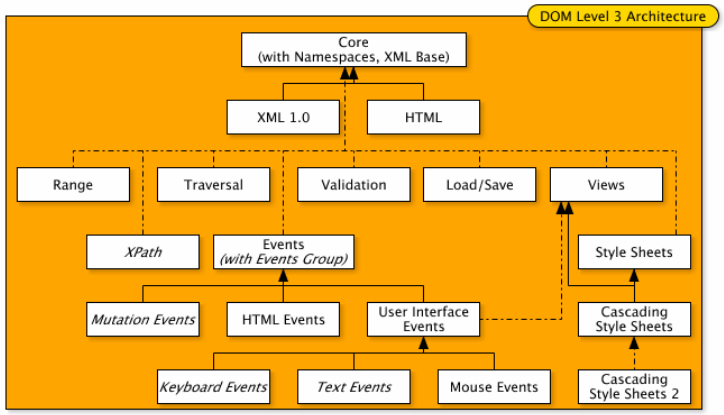

Figura 11 – Arquitetura da API DOM em seu terceiro nível (W3C, 2004c).

Conforme apresentado na Figura 11, documentos do tipo XML 1.0 e HTML formam a raiz de manipulações ou consultas utilizando DOM. Os principais objetivos da API DOM são (McGRATH, 1999):

- fornecer objetos e interfaces para representar documentos HTML e XML sem perda de informações;
- independência de plataforma e linguagem;
- disponibilizar funcionalidades que permitam a criação de objetos ou documentos HTML e XML;
- possibilitar a escrita com objetos em documentos que sejam estruturalmente idênticos aos lidos pela API;
- fornecer base consistente e extensível para que sejam adicionadas novas camadas ou níveis futuramente.

As operações sobre os documentos XML e HTML são realizadas através da utilização de métodos e atributos. A Tabela 1 apresenta alguns métodos utilizados em consultas e manipulação de conteúdo de documentos XML e HTML.

Tabela 1 – Principais métodos utilizados na manipulação de documentos XML e HTML.

Assinatura	Função
`createElement(element)`	`Cria um elemento na estrutura.`
`setAttribute(attribute)`	`Atribui valor a um atributo.`
`getAttribute(attribute)`	`Retorna o valor de um atributo.`

load(file)	Carrega a estrutura do documento
createTextNode(“Text”)	Cria um texto a ser inserido em um nó.
createAttribute(attribute)	Cria um atributo em um nó.
getElementByTagName(tag)	Retorna o nó especificado bem como todos os seus filhos.

Além dos métodos pode-se citar a utilização de atributos que tornam a manipulação de documentos HTML e XML mais simples conforme é apresentado na Tabela 2, que contém os principais atributos utilizados no tratamento de documentos.

Tabela 2 – Principais atributos utilizados na manipulação de documentos XML e HTML

Assinatura	Função
Length	Retorna o tamanho do nó.
Name	Retorna o nome do nó.
Value	Retorna o valor do nó.

A Figura 12 demonstra a utilização de alguns destes atributos e métodos. O exemplo apresentado tem por objetivo carregar a estrutura de um documento e obter os elementos de um nó dentro da estrutura do documento. Na primeira linha tem-se a declaração do objeto `xmlDoc` do tipo `ActivexObject` e passando como parâmetro a API DOM da Microsoft (MICROSOFT, 2004), que conterá a estrutura hierárquica XML. Na segunda linha, através do atributo async do objeto, identificam-se as transações referentes ao objeto como assíncronas. Na terceira linha, através do método `load`, o documento XML passado como parâmetro é carregado para o objeto `xmlDoc`. Na quarta linha uma variável x recebe os elementos contidos em um nó do documento XML.

```
1 var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
2 xmlDoc.async = false;
3 xmlDoc.load(strFileXML);
4 var x = xmlDoc.getElementsByTagName("aresta");
```

Figura 12 – Manipulação de documentos XML utilizando API DOM.

Segundo Martin et al (2001), “o papel do DOM é ocultar o documento básico é permitir que operemos em uma árvore de nós”. No trabalho foi utilizada a API DOM, sendo aplicada sobre documentos XML e SVG, utilizando a linguagem *script* Javascript apresentada na próxima seção.

2.4 Javascript

A Javascript é uma linguagem *script*, interpretada, no qual seu processamento pode ser realizado tanto no lado cliente – ou usuário – quanto no lado servidor da aplicação. Javascript deriva do C e C++ com estilo Java, porém são linguagens completamente distintas (GOODMAN, 2001).

A linguagem foi utilizada pela primeira vez no início dos anos 90 na versão 2.0 do *browser* Nestcape, sendo uma linguagem dinâmica orientada a objetos para criar tanto roteiros dinâmicos quanto promover a interação no lado cliente dos *browsers*. As principais características de Javascript são (RITCHEY, 2000):

- orientada a eventos: pode reagir às solicitações dos usuários;
- orientada a objetos: pode implementar seus próprios objetos, além de interagir com objetos do navegador.

A criação de objetos na Javascript a torna uma linguagem muito robusta. A Javascript suporta operações conhecidas em outras linguagens como: funções, procedimentos e estruturas de dados através das *custom functions*. Estas funções, como são mais conhecidas, são geralmente pequenos blocos de código reutilizáveis que podem ou não retornar valores (GOODMAN, 2001). A sintaxe para definição de uma função é apresentada na Figura 13.

```
1 function name_function( [arg1]...[argN]){
2       ...
3       código da função;
4       ...
5 }
```

Figura 13 – Sintaxe da declaração de uma função Javascript.

A utilização de argumentos em funções escritas na linguagem Javascript é optativa ao desenvolvedor, sendo que são comumente utilizados na criação de objetos. A criação de objetos em Javascript consiste em funções cujos atributos são declarados com a utilização da palavra reservada `this`, conforme pode ser constatado na Figura 14.

```
1 function aresta(origem, destino, peso){
```

```
2   this.origem = origem;
3   this.destino = destino;
4   this.peso = peso;
5 }
6 var a = aresta("Araguaína", "Palmas", "393");
```

Figura 14 – Declaração de um objeto em Javascript.

Na Javascript estruturas de dados são definidas utilizando-se de objetos, sendo que, com a utilização de outros componentes, como a API DOM, torna possível a manipulação de objetos pertencentes a *browsers*, além de possibilitar a manipulação de documentos texto ou mesmo arquivos XML.

Outro ponto a ser mencionado em Javascript, é a utilização de vetores, também chamados de *arrays*. Estes *arrays* são utilizados para armazenar vários valores em uma única variável, sendo referenciados com a utilização de um índice. A criação de vetores em Javascript se dá por meio da utilização da palavra reservada *new*, seguida da função `Array()`, como é demonstrado na Figura 15.

```
1 var nome = new Array(); //declarando as arrays
2 nome[0] = "Alex"; //atribuindo os valores
```

Figura 15 – Criação de vetores em Javascript.

Um aspecto a ser mencionado em Javascript, é a criação de vetores multidimensionais, também conhecidos como matrizes. No caso, para a criação de matrizes em Javascript é necessária a criação de novos vetores para cada posição – índice – do vetor, sendo que uma matriz é nada mais que um vetor que aponta para um novo vetor, conforme é apresentado na Figura 16.

```
1 var matriz = Mult_Array(cont_Vertice,cont_Vertice);
2 function Mult_Array(linhas,colunas){
3      var a = new Array(linhas);
4      for (var i=0; i < linhas; i++)
5      {
6            a[i] = new Array(colunas);
7      }
8      return a;
9 }
```

Figura 16 – Criação de matrizes em Javascript.

Na Figura 16, a primeira linha consiste na criação de uma variável que irá receber o resultado da função `Mult_Array()`, que tem como parâmetros a quantidade de linhas e

colunas que a matriz deve possuir. Na terceira linha é criado um vetor do tamanho de `linhas`, que na sexta linha recebe um novo vetor para cada índice.

Estas matrizes podem conter tanto valores quanto variáveis, além de estruturas como as demonstradas na Figura 14, sendo que estas podem ser utilizadas para a manipulação de valores obtidos de documentos XML sendo tema da próxima seção.

2.5 XML (*eXtensible Markup Language*)

XML é uma linguagem de marcação padronizada pelo consórcio W3C (*World Wide Web Consortium*), que combina as potencialidades e a extensibilidade de sua linguagem–mãe, a SGML (*Standard Generalized Markup Language*), padrão para armazenamento e intercâmbio de informações e dados em todo o mundo, que foi adotado em 1986 (KIRK & PITTS-MOULTIS, 2000).

A XML também fornece suporte para definir outras linguagens ou protocolos. Dentre estas podem ser citados o RDF (*Resouce Description Framework*), SOAP (*Simple Object Access Protocol*) e o SVG (*Scalable Vector Graphics*) (W3C, 2004a), que consiste em imagens vetoriais e que será apresentada no decorrer deste trabalho. O XML é também comumente utilizado como uma ferramenta de intercâmbio de informações entre diferentes aplicações. Porém o XML também apresenta características de estruturação de dados, sendo tema da próxima seção.

2.5.1 Armazenamento em estruturas hierárquicas XML

O XML tem como maior finalidade descrever informações contidas em sua estrutura de forma hierárquica. A estrutura do XML permite que sejam definidos documentos de forma a agilizar o processo de consulta e exibição de informações (SIEDLER & DE SOUZA, 2002).

Os dados geralmente são armazenados em banco de dados relacionais - linhas e colunas. Já o armazenamento em documentos XML, podem assumir diversos modelos,

permitindo a criação de vários elementos do mesmo tipo, além de vários atributos para cada elemento (GOMES et al., 2002).

Os dados podem ser armazenados de duas maneiras em documentos XML: a primeira é considerando que o conteúdo existente entre as *tags* de início e fim representam os valores a serem assumidos, conforme é demonstrado na Figura 17.

```
<?xml version="1.0"?>
<grafo>
	<vertice>
		<id>1</id>
		<x>130</x>
		<y>255</y>
	</vertice>
	<aresta>
		<origem>B</origem>
		<destino>C</destino>
		<peso>10</peso>
	</aresta >
</grafo>
```

Figura 17 – Conteúdo armazenado entre *tags*.

Outra maneira de armazenar valores em documentos XML seria assumir que os valores podem ser associados a atributos. Desta maneira cada *tag* possui seus atributos que representam as colunas a serem armazenadas, como é apresentado na Figura 18. Ambas as formas de armazenamento são interessantes já que podem manter várias estruturas diferentes em um único arquivo, conforme visualizado nas Figuras 17 e 18, ao se manter *tags* que representam diversos tipos.

```
<?xml version="1.0"?>
<grafo>
   <vertice id="1" x="130" y="130"/>
   <aresta id="12" origem="1" destino="2" peso="5" x1="30" y1="30"/>
   <vertice id="2" x="205" y="164"/>
   <vertice id="3" x="155" y="222"/>
   <vertice id="4" x="199" y="110"/>
   <aresta id="34" origem="1" destino="2" peso="10" x1="93" y1="09"/>
</grafo>
```

Figura 18 – Conteúdo armazenado como atributo.

No trabalho foi utilizado o armazenamento dos dados como atributos de uma estrutura. O armazenamento em documentos XML foi adotado neste trabalho, sendo que os dados eram utilizados para interação com o mapa criado no formato SVG que é o próximo tema a ser considerado.

2.5.2 SVG (*Scalable Vector Graphics*)

SVG é uma linguagem baseada em XML para a criação de imagens bidimensionais, ou seja, uma imagem vetorial baseada em texto, em que a possibilidade de programação, controle, interação com o usuário e conexão a banco de dados a tornam diferenciada, pois estas possibilidades não estão presentes em formatos como JPEG, GIF (PEARLMAN & HOUSE, 2003).

Os elementos básicos da geometria são representados no SVG pelas *tags* (EISENBERG, 2003) *rect, line, polygon, circle, ellipse*. Dentre estes serão abordados somente os elementos *line* e *ellipse*, que serão apresentados nas seções seguintes. Outro elemento a ser mencionado no trabalho e que é muito comum na criação de imagens SVG, é o elemento *path* que torna possível criar as mais diversas formas, dando mais possibilidades ao SVG. Para estudos mais detalhados sobre os outros elementos que compõem a estrutura das imagens SVG sugere-se a leitura da documentação disponibilizada durante a realização da disciplina de Prática de Sistemas de Informação I (COELHO & FAGUNDES, 2004). Os elementos *ellipse*, *line* e *path* são apresentados nas próximas subseções.

2.5.2.1 Ellipse

O elemento *ellipse* é simplesmente um círculo onde o raio no sentido de `x` e o raio `y` no sentido se diferem. Caso os raios x e y sejam iguais o elemento *ellipse* pode ser considerado um círculo. No SVG pode-se representar um círculo utilizando tanto o elemento *circle* quanto o elemento *ellipse*, sendo os raios `x` e `y` trabalhados de forma igual para ambos os elementos. A Tabela 3 apresenta os principais atributos referentes a manipulação do elemento *ellipse* (WATT, 2002).

Tabela 3 – Principais atributos do elemento *ellipse* (PEARLMAN & HOUSE, 2003).

Descrição	Tipo do Atributo	Função
`cx`	`SVGLength`	`Define a posição da coordenada x do ponto central da ellipse.`
`cy`	`SVGLength`	`Define a posição da coordenada y do ponto central da ellipse.`
`rx`	`SVGLength`	`Define o raio (horizontal) da linha`

		central de x da *ellipse*.
ry	SVGLength	Define o raio (vertical) da linha central de y da *ellipse*.
id	ID	Identificador do elemento
style	String	Estilos CSS.

No elemento necessita-se que os atributos `cx` e `cy` sejam definidos, marcando assim o ponto central da *ellipse*, sendo aplicado o raio sobre este ponto. O atributo *style* e *id* são atributos aplicáveis a qualquer elemento no SVG. A utilização dos atributos é apresentada na Figura 19, na qual o elemento *ellipse* é demonstrado.

```
1 <svg width="500" height="500">
2      <ellipse cx="206" cy="171.5" rx="84" ry="31.5"
3       style="fill:rgb(192,192,255);stroke:rgb(0,0,0);stroke-
4      width:1"/>
5       <ellipse cx="226" cy="278.5" rx="52" ry="52"
6      style="fill:rgb(192,0,0);stroke:rgb(0,0,0);stroke-
7      width:1"/>
8 </svg>
```

Figura 19 – Utilização do elemento *ellipse*.

Na linha primeira da Figura 19 é definida a área da imagem a ser apresentada dentro da tag inicial `<svg>`. Nas linhas 2 (dois) a 7 (sete) são criadas duas formas elípticas na imagem, no qual os atributos `cx` e `cy` definem o ponto central de cada uma das formas. Conforme mencionado os atributos `rx` e `ry` definem o raio a partir do ponto central definido, em que raios iguais formam um círculo conforme apresentado na Figura 20, que é o resultado do código mencionado na Figura 19.

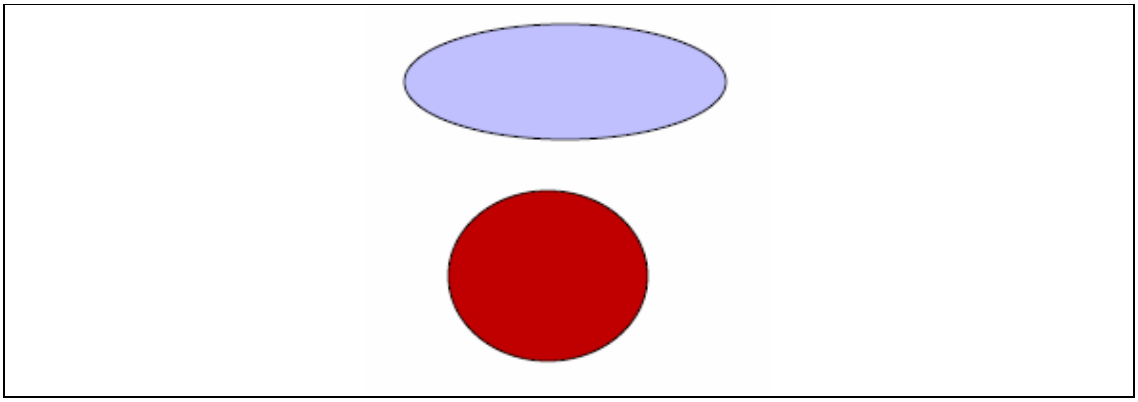

Figura 20 – Resultado do processamento do código da Figura 19.

Algumas propriedades presentes no código da Figura 19 também são aplicadas a outros elementos no SVG como, por exemplo, algumas presentes no atribuo *style*, no qual

são definidas largura, cor e opacidade da linha ou preenchimento, o que pode ser constatado no elemento *line*, apresentado a seguir.

2.5.2.2 Line

O elemento *line* descreve um segmento de linha reta que começa em um determinado ponto e termina em outro (PEARLMAN & HOUSE, 2003). Mais detalhadamente, o elemento *line* consiste em dois pontos em específico (`x1`,`y1`) e (`x2`,`y2`) com uma linha ligando-os.

Ao elemento *line* podem ser acrescidos alguns estilos como: *stroke*, *stroke-with* e *stroke-opacity*, sendo que a primeira propriedade consiste na cor da linha, a segunda na largura da linha e a terceira a opacidade aplicada sobre esta. A Tabela 4 apresenta alguns dos atributos do elemento *line*.

Tabela 4 – Principais atributos do elemento *line* (CAGLE, 2002).

Descrição	Tipo do Atributo	Função
x1	SVGLength	Determina o inicio da linha no eixo x.
y1	SVGLength	Determina o inicio da linha no eixo y.
x2	SVGLength	Determina o fim da linha no eixo x.
y2	SVGLength	Determina o fim da linha no eixo x.

O elemento *line* é o mais simples de todos os elementos básicos do SVG já que consiste apenas na definição dos pontos inicial e final. A Figura 21 demonstra o código para a confecção de uma linha simples utilizando o elemento *line* bem como a utilização de seus atributos (CAGLE, 2002).

```
1 <svg width="500" height="500">
2      <line id="simple_line" x1="50" y1="50" x2="320" y2="240"
3              stroke=”rgb(255,0,0)"/>
4      <line id="attr_line" x1="100" y1="50" x2="320" y2="240"
5              style="stroke-width:4;fill:rgb(0,0,0);
6            stroke:rgb(0,0,0)"/>
7 </svg>
```

Figura 21 – Utilização do elemento *line*.

Na Figura 21 as *tags* contidas nas linhas 1 e 7 consistem na declaração e finalização da imagem SVG. As linhas 2 e 3 fazem referência a uma simples linha que vai do ponto

(`x1,y1`) ao ponto (`x2,y2`) de cor vermelha, linha esta denominada *simple_line* através da utilização do atributo identificador do elemento.

Nas linhas 4, 5 e 6 é apresentada a declaração de outra linha em que se utilizam os atributos `stroke-width` e `stroke` dentro do atributo `style` para definir propriedades da linha como a largura por exemplo. O resultado do código presente na Figura 21 pode ser visualizado na Figura 22.

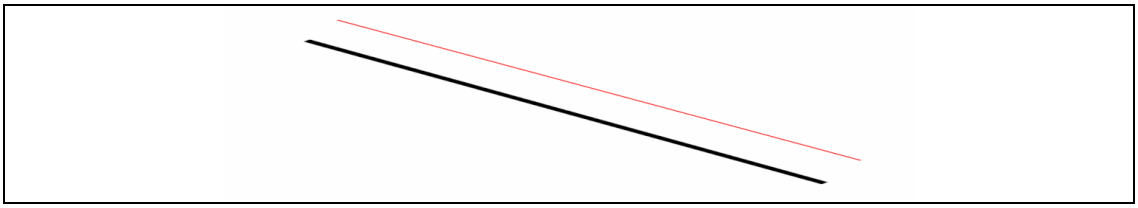

Figura 22 – Resultado do processamento do código da Figura 18.

O elemento *line* é uma primitiva, muito útil para descrição e criação de desenhos simples, mas não é recomendado para a criação de formas ou trajeto, recomendando-se a utilização do elemento *polyline* e ou *path* (CAGLE, 2002), que é o próximo elemento a ser apresentado.

2.5.2.3 Path

O elemento *path* é o elemento mais versátil do SVG (WATT, 2001), permitindo a criação de formas mais complexas. O *path* trabalha com a idéia de pontos que se interligam por linhas, na qual se define um ponto inicial e a seguir se definem os pontos subseqüentes do ponto de origem, desenhando-se uma linha entre cada um dos pontos (PEARLMAN & HOUSE, 2003).

O elemento *path* chama a atenção por sua sintaxe, que é relacionada às propriedades ou comandos como o "`moveTo`", "`lineTo`" e "`closePath`", sendo utilizado para definir caminhos e formas a serem determinados (NEUMANN & WINTER, 2001). Estes comandos devem estar declarados no atributo `d`, sendo um atributo obrigatório na estrutura do elemento *path*, pois define o desenho a ser criado. Porém o elemento *path* não se prende apenas às propriedades apresentadas, sendo que existem diversas propriedades

que contribuem para a elaboração das mais diversas formas, sendo que três comandos mais utilizados são apresentados na Tabela 5.

Tabela 5 – Propriedades do elemento *path.*

Descrição	Comando	Função
M	moveTo	Move o ponto para a origem.
L	lineTo	Desenha a linha e entre os pontos.
Z	closePath	Força o fechamento dos pontos.

O *path* é mais abrangente que outros elementos do SVG, conseguindo-se desenhar qualquer forma desejada por meio de uma serie de linhas, arcos e curvas conectadas. Todo elemento *path* deve começar com a declaração do comando de `moveTo`, descrito pela letra "M", que define o ponto inicial das coordenadas `x` e `y` da forma a ser desenhada (EISENBERG, 2002).

O comando `moveTo` sempre é seguido de um ou mais comandos `lineTo`, representado pela letra `L` seguido de suas coordenadas `x` e `y`. O comando `lineTo` é o responsável por desenhar a linha entre o último ponto desenhado e o novo ponto - coordenadas `x` e `y` - que compõem o comando `lineTo` no elemento *path*. A utilização dos comandos por ser observada na Figura 23.

```
1 <svg width="500" height="500">
2    <path d="M10 10, L100 10"/>
3    <path d="M10 20, L100 20, L100 50"/>
4    <path d="M40 60, L10 60, L40 42, M60 60, L90 60, L60 42 "/>
5 </svg>
```

Figura 23 – Utilização do elemento *path* (EISENBERG, 2002).

A Figura 23 apresenta a declaração de três elementos *path*, em que o comando `moveTo` define o ponto de origem de cada elemento, linhas 2, 3 e 4. Na seqüência é utilizado o comando `lineTo` que desenha linhas entre as coordenadas subseqüentes. Por exemplo, na segunda linha o elemento *path* contém a declaração de um comando `moveTo` nas coordenadas (10,10), logo em seguida e declarado um comando `lineTo` nas coordenadas (100,10), com isto é desenhada uma linha ligando as coordenadas dos comandos. A linha 4 apresenta uma característica muito comum no elemento *path*, que é a declaração de um novo ponto inicial com a utilização do comando `moveTo`, em meio às

declarações de desenho `lineTo`. O resultado do código apresentado na Figura 23 está presente na Figura 24.

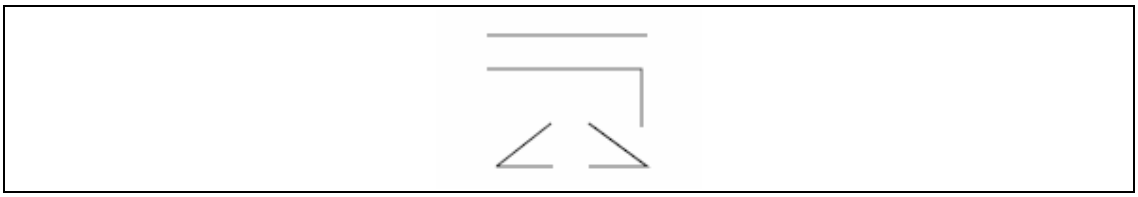

Figura 24 – Resultado do processamento do código da Figura 23 (EISENBERG, 2002).

Uma das possibilidades apresentadas pelo elemento *path* é a utilização do comando `closePath` por meio da letra `Z` que desenha a linha entre o ponto inicial e o ultimo ponto declarado no elemento `lineTo`. Uma forma de se deixar isto mais claro, por exemplo, é a construção de um triângulo em que podem ser desenhadas duas linhas e, com a utilização do elemento `closePath`, realizar a junção entre estas (EISENBERG, 2002).

É comum a utilização dos comandos representados por letras minúsculas. Estes nada mais são que os mesmos comandos representados pelas letras maiúsculas, sendo que representam a posição relativa dos pontos, além de existirem comandos mais complexos relacionados ao elemento *path* como *ellipses*, arcos, curvas bézianas e alguns outros (W3C, 2004b).Todos os elementos SVG podem ser manipulados através da utilização de uma linguagem *script* e DOM, sendo isto reflexo da herança do XML.

3 MATERIAL E MÉTODOS

Na realização deste trabalho foram utilizados recursos como hardware e software, além de produções bibliográficas que juntamente com orientações tornaram a realização deste trabalho possível.

3.1 Local e Período

O trabalho foi desenvolvido no período de Agosto a Dezembro de 2004, no Laboratório de Multimídia e Hipermídia (LabMídia) no Centro Universitário Luterano de Palmas – CEULP/ULBRA.

3.2 Materiais

Na realização do trabalho foram utilizados *softwares* disponíveis na Internet, além de *hardware* e *software* licenciados disponibilizados pelo CEULP/ULBRA.

Softwares:

Jasc WebDraw versão 1.1, para criação das imagens SVG;
Internet *Explorer* versão 6.0, para a visualização das imagens SVG;
Adobe SVG *plugin* versão 3.0, plugin SVG para o Internet Explorer 6.0;
Editor Bloco de Notas do *Windows* versão 2000, utilizado na construção do código Javascript e depuração da imagem SVG.

Hardware:

Pentium III, 750 Mhz com 128 MB de memória RAM;

3.3 Metodologia

O material foi utilizado visando obter a maior quantidade de informações relevantes à conclusão deste trabalho, em que se mostraram necessárias pesquisas sobre grafos, algoritmos de menor caminho e em específico o algoritmo de *Dijkstra*.

Foi essencial no desenvolvimento e na implementação do protótipo, o uso de *softwares* para a criação das imagens SVG, e de todo o código Javascript, no qual foram utilizados tanto os conceitos adquiridos com os estudos realizados, bem como todo o aparato de *hardware* e *software* citado. Para a realização do trabalho foram assumidas etapas para o desenvolvimento como, a construção do mapa no formato SVG, criação das funções para a interação com o usuário e a adaptação do algoritmo de Dijkstra. Os resultados obtidos são apresentados na próxima seção.

4 RESULTADOS E DISCUSSÃO

Todo o material bibliográfico apresentado foi utilizado como base para a implementação do protótipo do sistema, sendo que os temas mais relevantes encontrados durante o desenvolvimento são apresentados a seguir na seção implementação.

4.1 Implementação

A implementação do protótipo se dividiu em três partes, sendo elas: construção e manipulação do mapa no formato SVG, armazenamento de dados em arquivos XML e a implementação do algoritmo de *Dijkstra*.

A primeira parte consistiu no desenvolvimento de um mapa com o formato SVG que recebesse a interação do usuário, respondendo a esta dinamicamente com a utilização de Javascript e DOM. Na criação do mapa foi utilizado o elemento `path`, já que somente este elemento possibilitaria a criação da forma que se necessitava, em que foram utilizados os comandos `lineTo` e `moveTo` obtendo como resultado a imagem na Figura 25.

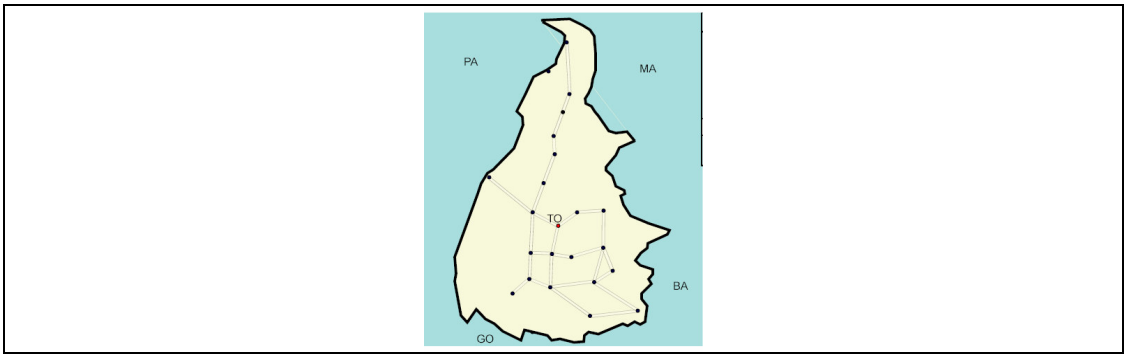

Figura 25 – Mapa dinâmico em SVG e Javascript e DOM.

O protótipo do sistema desenvolvido divide-se em dois módulos: um administrativo, em que um administrador cadastra as cidades (vértices) e as estradas (arestas) no mapa, e outro em que os usuários escolhem os pontos de origem e destino, executando-se o algoritmo sobre os dados fornecidos pelo usuário.

Tanto o cadastro dos vértices e das arestas quanto o algoritmo utilizado para localizar o menor caminho foram implementados utilizando Javascript e DOM. Para a construção dos vértices na imagem SVG foi utilizado o elemento `ellipse`, em que cada elemento possui um identificador que, com a saída fornecida pelo algoritmo, é utilizado para apresentar o menor caminho ao usuário. A função que cria os vértices no mapa é demonstrada na Figura 26.

```
1 function Create_Vertice(evt){
2      svgdoc=evt.getTarget().getOwnerDocument();
3      node=svgdoc.createElement("ellipse");
4      node.setAttribute("id",cont_Vertice);
5      node.setAttribute("cx",Coord_X(evt));
6      node.setAttribute("cy",Coord_Y(evt));
7      node.setAttribute("rx","6");
8      node.setAttribute("ry","6");
9      node.setAttribute("style","fill:white");
10     node.addEventListener("click",Return_Id, false);
11     out=svgdoc.getElementById("mapa");
12     out.appendChild(node);
13     window.parent.parent.addVertice(cont_Vertice,Coord_X(evt),Coord_Y(evt));
14     cont_Vertice++;
15 }
```

Figura 26 – Criação dos vértices dinamicamente.

Na primeira linha da Figura 26 tem-se a assinatura da função que cria um vértice tendo como parâmetro a variável `evt`, que representa o documento que acionou o evento. Na segunda linha este documento é acessado de maneira a se ter todo o documento contido na variável `svgdoc`. Na terceira linha é criado um elemento do tipo `ellipse` e da quarta linha a nona têm-se os atributos definidos. Na décima linha é adicionado um evento ao elemento que está sendo criado e que é utilizado para a definição do `id`, que irá compor uma aresta que se origina ou mesmo termina no vértice em questão. Na décima-primeira linha a variável `out` recebe o elemento `mapa`, contido no documento SVG, e a esta variável é acrescentado o elemento `ellipse` criado, através da função `appendChild()` na décima-segunda linha. A décima-terceira linha consiste no armazenamento dos dados no documento XML, no qual a imagem acessa uma função em um *frame* pai, a qual será

explicada no decorrer deste trabalho. A variável `cont_Vertice`, na décima-quarta linha, é utilizada para determinar o `id` de cada vértice, em que a cada novo vértice criado a variável deve ser incrementada. A execução desta função é demonstrada na Figura 27.

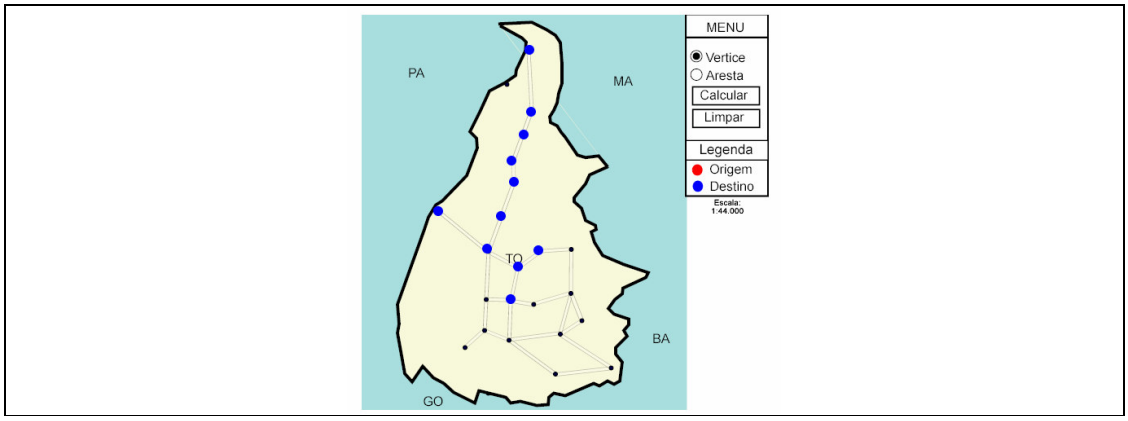

Figura 27 – Criação de vértices na imagem.

Assim como foi realizado com os vértices, foi necessária a implementação de uma função que criasse as arestas de forma dinâmica, sendo que a instanciação da aresta dependia diretamente da existência de vértices, isto realizado através da ocorrência do evento `onclick` sobre o vértice que chama a função atribuída ao vértice em sua criação dinâmica. Esta função tem como objetivo concatenar os `id's` do vértice de origem com o do destino, tendo-se assim o elemento identificador da aresta definido. A função que realiza esta operação é demonstrada na Figura 28.

```
1 function Return_Id(evt){
2      svgdoc = evt.getTarget;
3      var value = svgdoc.getAttribute("id");
4      if (id_aresta == ""){
5            origem = value;
6      }
7      else{
8            destino = value
9            peso = prompt("Digite o valor para o peso:","");
10     }
11     id_aresta = id_aresta+""+value;
12 }
```

Figura 28 – Concatenação dos `id's` dos vértices obtendo-se o `id` da aresta.

Na Figura 28, outro aspecto que chama a atenção é a utilização da função *prompt* do Javascript para obter o valor a ser associado à aresta como o peso. A função que cria as arestas se assemelha à que cria os vértices, o que pode se constatado na Figura 29.

```
1 function Create_Aresta(evt){
2      svgdoc=evt.getTarget().getOwnerDocument();
3      node=svgdoc.createElement("line");
4      node.setAttribute("id",id_aresta);
5      node.setAttribute("x1",X_1);
6      node.setAttribute("y1",Y_1);
7      node.setAttribute("x2",X_2);
8      node.setAttribute("y2",Y_2);
9      node.setAttribute("style","stroke:rgb(0,0,0);stroke-width:2");
10     out=svgdoc.getElementById("mapa");
11     out.appendChild(node);
12     window.parent.parent.addAresta(id_aresta,origem,destino,
13      peso,X_1,Y_1,X_2,Y_2);
14 }
```

Figura 29 – Criação das arestas dinamicamente.

O resultado obtido da execução desta função na imagem SVG é uma linha ligando os vértices escolhidos, sendo registrados como origem e destino do vértice como pode ser visualizado na Figura 30, que demonstra a imagem alterada.

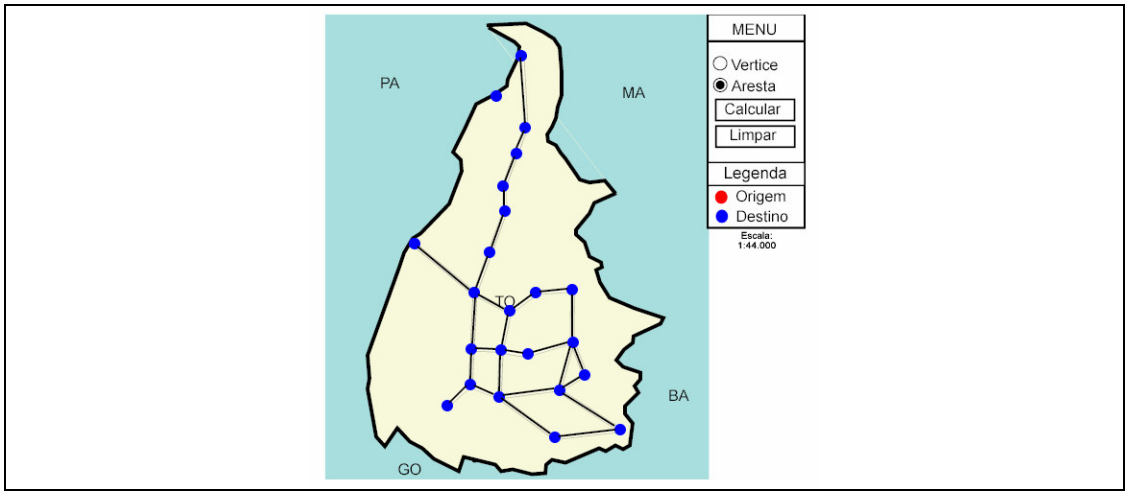

Figura 30 – Criação da aresta na imagem SVG.

Com isto, foi necessário distinguir quando são criados vértices e quando são criadas as arestas dinamicamente no mapa, sendo assim necessária a implementação de uma função que distinguisse as opções utilizando elementos presentes na imagem. Assim, foram criados dois elementos do tipo `ellipse`, que representam a criação de vértices ou de arestas, apresentados como `radiobuttons`, que variam entre elementos visíveis e invisíveis, sendo que o elemento que estiver visível no momento da interação do usuário com o mapa será criado dinamicamente. O código que efetua esta verificação está presente na Figura 31.

```
1 function Option(evt){
2      svgdoc = evt.getTarget().getOwnerDocument();
3      map = svgdoc.getElementById('mapa');
4      objet = svgdoc.getElementById('vertice');
5      objet1 = svgdoc.getElementById('aresta');
6
7      if (objet.getStyle.getPropertyValue('visibility') == 'visible'){
8            Create_Vertice(evt);
9      }
10     if (objet1.getStyle.getPropertyValue('visibility') == 'visible'){
11           Values_Edges(evt);
12           if ((X_2 != 0) && (id_aresta != "")){
13                 Create_Aresta(evt);
14                 X_1=0;Y_1=0;X_2=0;Y_2=0;
15                 id_aresta = "";
16                 origem = "";
17                 destino = "";
18           }
19     }
20 }
```

Figura 31 – Escolha do elemento a ser inserido no mapa.

A partir da sétima linha da Figura 31 até a décima-nona tem-se a verificação para a criação de vértices ou de arestas. Na sétima linha é verificado se o elemento que representa a criação dos vértices esta visível; se verdadeiro, executa-se a função `Create_Vertice()`. Na décima linha é realizada verificação semelhante, sendo que para a criação de arestas.

Feito isto, a segunda etapa, como mencionado, consistiu no armazenamento dos dados referentes aos vértices e arestas criados, em um documento XML, tornando estas informações disponíveis ao usuário. Foi necessário tornar os dados representados na imagem SVG persistentes, sendo apresentados na forma de um grafo. Para isto foi utilizada a API DOM da Microsoft para o Internet Explorer 6.0, além de objetos do próprio *browser*, que auxiliam em abrir, ler e salvar os dados no documento XML existente. A função criada para o armazenamento dos dados referentes aos vértices no arquivo XML é apresentada na Figura 32.

```
1 function addVertice(id,x,y){
2      var docXML = abrirDoc("grafo.xml");
3      if(docXML.parseError.errorCode != 0){
4            document.write("Erro: ");
5            document.write(docXML.parseError);
6            if (docXML.parseError == -2146697210){
7                  document.write("Criar arquivo");
8            }
9            return false;
10     }
11     else{
12           vertice = docXML.createElement("vertice");
13           vertice.setAttribute("id",id);
```

```
14            vertice.setAttribute("x",x);
15            vertice.setAttribute("y",y);
16            docXML.documentElement.appendChild(vertice);
17            SalvaXML("grafo.xml", docXML);
18            return true;
19      }
20 }
```

Figura 32 – Função que registra os dados dos vértices no documento XML.

A função que adiciona os dados ao documento XML (Figura 32), em sua primeira linha apresenta a assinatura da função, no qual são repassados como parâmetros os dados que se deseja armazenar. Na segunda linha a variável `docXML` recebe o retorno da função `abrirDoc()`, que abre o arquivo XML passado como parâmetro para a manipulação, e que é exposta função na Figura 33.

```
1 function abrirDoc(strFile){
2       var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
3       xmlDoc.async = false;
4       xmlDoc.load(strFile);
5       return xmlDoc;
6 }
```

Figura 33 – Função que abre o documento XML para manipulação.

Como mencionado, nesta função é utilizada a API DOM através de um objeto *ActivexObject* para a manipulação do arquivo XML que é carregado na quarta linha e retornado na quinta linha. Seguindo na Figura 32, as linhas de 3 a 10 consistem no tratamento de erro caso o arquivo não exista. Existindo o arquivo, na décima-segunda linha é criado um nó, em que são atribuídos valores a seus atributos e é anexada a hierarquia do objeto aberto, logo após invocada a função `SalvaXML()`, sendo passados como parâmetros o nome do arquivo a ser salvo e a estrutura hierárquica construída, como pode ser observado na Figura 34.

```
1 function SalvaXML(caminho, docXML){
2       try {
3             var fso = new ActiveXObject("Scripting.FileSystemObject");
4             var file = fso.createTextFile(dir+caminho, true);
5             file.writeLine(docXML.xml);
6             file.close();
7       } catch (ex) {
8             alert(ex.message);
9       }
10 }
```

Figura 34 – Função que salva o objeto no documento XML.

A Figura 34 na terceira linha apresenta a utilização dos objetos `Scripting.FileSystemObject`, que fazem parte da API `File System Object` do *Internet Explorer*, para trabalhar com pastas e arquivos, sendo utilizado na quarta linha para a criação de um documento texto no caminho especificado e atribuído o valor *true* à possibilidade (opção) de sobrescrita do arquivo em questão. Na quinta linha da função toda a hierarquia montada na função `addVertice()`, da Figura 32, é escrita em formato XML no arquivo, e na sexta linha o documento é fechado. As demais linhas consistem no tratamento de erros da função.

Com isto, caso todas as operações apresentadas na função `addVertice()`, da Figura 32 ocorram sem problemas, é retornado *true*, indicando o sucesso da operação. Assim como a função demonstrada na Figura 32, para o registro dos dados referentes aos vértices, foi criada uma função semelhante que registrava os dados referentes às arestas.

Seguindo a representação do mundo real em que as estradas possuem dois sentidos, eram armazenadas no arquivo XML as arestas tanto no sentido obtido da interação do usuário na imagem, quanto no sentido oposto, tendo se assim ambas as direções para um mesmo trajeto. Para este registro são utilizadas operações semelhantes às apresentadas para o registro dos dados referentes aos vértices.

Porém, no decorrer da implementação alguns problemas foram encontrados, dentre estes pode-se citar a falta de suporte do *plugin* para SVG, disponibilizado pela Adobe Inc., aos objetos *Activex* para a manipulação de documentos XML. Ao tentar ler ou manipular um arquivo XML, a partir de uma função Javascript que se encontrava dentro do escopo da imagem SVG, que utilizava o *ActivexObject*, era emitido um alerta de erro relatando que o *ActivexObject* era um objeto indefinido.

O problema foi solucionado com a utilização de *frameset*, em que todas as funções para manipulação do documento XML ficam contidas na página principal de todos os *frames*, sendo que as imagens SVG tinham acesso às funções com a utilização de hierarquia do *browser* conforme pode ser averiguado na Figura 35.

```
1 window.parent.parent.addVertice(cont_Vertice,Coord_X(evt),Coord_Y(evt));
```

Figura 35 – Acesso a funções no *frame* pai.

A imagem SVG para acessar uma função em um frame diferente era considerada pelo *browser* como filho do *frame* em que ela está inserida; Com isto, para acessar as funções, foi necessário subir dois níveis na hierarquia para a utilização da função. Porém, no decorrer dos trabalhos ficou constatado que esta forma de acesso aos documentos XML não era eficiente para consulta aos dados.

Se fossem utilizadas estas funções, os dados consultados no *frame* pai seriam passados na forma de *string* para a imagem e na imagem, deveria ser utilizada a função `parseXML()`, para transformar a *string* em um objeto manipulável com a utilização da API DOM. A consulta aos arquivos XML foi realizada diretamente pela imagem SVG, sendo que a *string* a ser convertida em um documento manipulável pela API DOM pode ser obtida pela própria imagem. No caso, as funções em *frames* foram utilizadas somente para registro dos dados.

Assim, para a consulta do arquivo XML foi utilizada a função `getUrl()` e a função `parseXML()` na própria imagem. A primeira é responsável por ler o documento XML e transformá-lo em *string* e a segunda é responsável por fazer a conversão da *string* obtida com a função `getUrl()`em um `document`, sendo assim um objeto manipulável com a utilização da API DOM. O código utilizado para consulta ao documento XML é apresentado na Figura 36.

```
1 function FileLoad(){
2      getURL(filename, abrirDoc);
3      function abrirDoc(data) {
4             var string = '';
5             if(data.success) {
6                    string = data.content;
7                    var document = parseXML(string,document);
8                    file = document.childNodes.item(0);
9                    return file;
10            }
11            else {
12                   return;
13            }
14     }
15 }
```

Figura 36 – Utilização das funções `getUrl()` e `parseXML()`.

Assim, com o retorno de um objeto manipulável pela própria imagem foi possível consultar os dados que estão cadastrados no documento XML com a utilização da API DOM dentro do código das imagens SVG, que consulta os atributos referentes a cada elemento presente no documento, como demonstrado na Figura 37.

```
1 function Load_Grafo(evt){
2      xml = abrirDoc(data);
3      x = xml.getElementsByTagName("vertice");
4       svgdoc=evt.getTarget().getOwnerDocument();
5      for(var i=0; i < x.length; i++){
6             var vertice_x = x.item(i).getAttribute("x");
7             var vertice_y = x.item(i).getAttribute("y");
8             var vertice_id = x.item(i).getAttribute("id");
9             node=svgdoc.createElement("ellipse");
10            node.setAttribute("id",vertice_id);
11            node.setAttribute("cx",vertice_x);
12            node.setAttribute("cy",vertice_y);
13            node.setAttribute("rx","6");
14            node.setAttribute("ry","6");
15            node.setAttribute("style","fill:black");
16            node.addEventListener("click",getSource, false);
17            out=svgdoc.getElementById("mapa");
18            out.appendChild(node);
19            cont_Vertice++;
20     }
21      ...
22 }
```

Figura 37 – Consulta ao documento XML e criação de elementos SVG com os valores.

A função apresentada na Figura 37 consiste em ler os dados presentes no arquivo XML e com isto, utilizar estes valores para a criação dinâmica dos elementos que devem compor a imagem. No caso são acessados os valores referentes aos vértices e, logo após, são criados os elementos SVG.

Na mesma função `Load_Grafo(evt)`, apresentada na Figura 37, é criada a matriz de adjacência que representa o grafo utilizado para o processamento do algoritmo de *Dijkstra*, sendo que a matriz é instanciada após todos os vértices lidos e anexados a imagem SVG, já que a matriz deve ter em ambas as dimensões o tamanho referente à quantidade de vértices presentes na imagem. Feito isto foi necessário inicializar a matriz de adjacência, utilizando-se uma estrutura na criação da matriz e presente na Figura 39 que indica a existência ou não da adjacência a um determinado vértice e, além disto, determinar o peso – distância - para a adjacência deste vértice. A criação da matriz de adjacência e inicialização desta estão presentes no código descrito na Figura 38.

```
1 function Load_Grafo(evt){
2       ...
3       var matriz = Mult_Array(cont_Vertice,cont_Vertice);
4       Inicializa_Matriz();
5       ...
6 }
```

Figura 38 – Criação e inicialização da matriz de adjacência.

Conforme mencionado, foi adotada uma estrutura que armazenasse o peso e a adjacência de um vértice a outro na matriz. Para isto foi utilizada uma estrutura (ou objeto) Javascript que guardasse estes valores para cada vértice, representando uma aresta na matriz. A estrutura do objeto é apresentada na Figura 39.

```
1 function aresta(){
2     this.peso;
3     this.adj;
4 }
```

Figura 39 – Criação da estrutura utilizada na matriz.

Assim, a função `Inicializa_Matriz()`, presente na Figura 40 é responsável por inicializar os valores presentes em cada posição da matriz, sendo que, para indicar a adjacência era utilizado o valor 0 (zero) e, para o peso um valor infinito, no qual sempre deveria ser maior que qualquer outra ocorrência, caso esta viesse a acontecer.

```
1 function Inicializa_Matriz(){
2     for (var i=0; i < cont_Vertice; i++){
3           for (var j=0; j < cont_Vertice; j++){
4                 matriz[i][j].peso = 999999;
5                 matriz[i][j].adj = 0;
6                 }
7     }
8 }
```

Figura 40 – Função que inicializava a matriz de adjacência.

Vários destes valores presentes na matriz de adjacência, que registram a ocorrência tanto para a adjacência quanto para o peso, são alterados ainda durante o processamento da função `Load_Grafo(evt)`, já que, após a inicialização da matriz, ocorre a leitura das ocorrências de arestas registradas no documento XML, que contêm as adjacências e pesos, sendo que esta leitura ocorre de forma semelhante à utilizada para os vértices (apresentada na Figura 37). Para a inserção destes valores na matriz criou-se uma função de forma a facilitar esta inserção cujo código é apresentado na Figura 41.

```
1 function inseri_grafo(origem, destino, peso) {
2     matriz[origem][destino].peso = peso;
```

```
3       matriz[origem][destino].adj = 1
4 }
```

Figura 41 – Função responsável por inserir os valores na matriz de adjacência.

Depois de lidos os valores presentes no arquivo XML, criados os elementos que passaram a compor a imagem SVG com os valores mencionados e, criada, inicializada e preenchida a matriz de adjacência para o processamento do algoritmo, iniciou-se a terceira etapa que consistiu na implementação do algoritmo de *Dijkstra*.

Na função que realiza o processamento de Dijkstra foi necessária a utilização da matriz de adjacência, além de criados três vetores auxiliares do tamanho igual ao número de vértices presentes no grafo para a representação das informações vitais para cada um destes vértices, citadas na seção 2 (Revisão de Literatura) que são: se o vértice já foi visitado; o menor peso até o vértice fonte; o vértice predecessor ao vértice que o índice referencia, sendo denominados: `percorrido`; `distância`; `caminho`. Assim como no algoritmo demonstrado na seção Revisão de Literatura, foi criada uma função para a inicialização dos vetores de vértices percorridos e de distância conforme demonstrado na Figura 42.

```
function Inicializer(){
       for(var i = 0; i < cont_Vertice;i++){
              percorrido[i] = 0;
              distancia[i] = 99999;
       }
}
```

Figura 42 – Função que inicializa os vetores de vértices percorridos e distância.

Assim como aconteceu na inicialização da matriz de adjacência, foi utilizado o valor 0 (zero) para indicar que o vértice referenciado pelo índice ainda não havia sido percorrido pelo algoritmo e um valor extremamente grande para indicar que o caminho deste vértice até o vértice fonte ainda não é conhecido, de forma que este valor sempre fosse maior que qualquer outro possível no protótipo. Ao final deste processamento todos os vértices e arestas foram criados dinamicamente na imagem, com os valores retirados do arquivo XML como é visualizado na Figura 43.

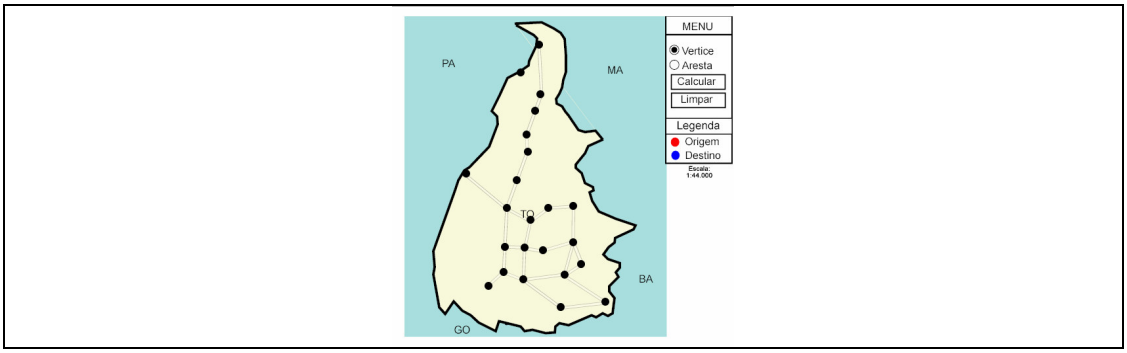

Figura 43 – Imagem SVG após carregar todos os valores registrados no documento XML.

Somente os vértices eram apresentados ao usuário e as arestas eram criadas de forma invisível, porém fazendo parte da imagem. As arestas são apresentadas somente se estas fizerem parte do menor caminho processado pelo algoritmo de *Dijkstra* que consiste na terceira etapa do trabalho.

Feito isto, o algoritmo de Dijkstra consistiu na utilização da estrutura do algoritmo apresentada na Revisão de Literatura, sendo que foram feitas adaptações como inserir uma regra de para se o vértice procurado fosse encontrado antes do final do processamento total do grafo, evitando se processamento desnecessário. Nestas adaptações mostrou-se necessária a utilização de variáveis auxiliares, de forma que estas, na maioria da vezes, continham valores utilizados nas iterações que o algoritmo realiza, o que pode ser verificado na Figura 44, que contém a implementação utilizada no trabalho.

```
1 function dijkstra(source,destin){
2       ...
3      percorrido[source] = 1;
4      distancia[source] = 0;
5       ...
6       while (corrente != destin){
7             ...
8             menor_distancia = 99999;
9             distancia_momento = parseInt(distancia[corrente]);
10
11            for(i = 0 ; i < cont_Vertice; i++){
12
13                  if(percorrido[i] == 0){
14                        aux = parseInt(matriz[corrente][i].peso);
15                        result = eval(distancia_momento+aux);
16
17                        if(result < distancia[i]){
18                              distancia[i] = result;
19                              caminho[i] = corrente;
20                        }
21
22                        if(distancia[i] < menor_distancia){
23                              menor_distancia = distancia[i];
```

```
24                                  next = i;
25                              }
26                      }
27              }
28              ...
29              corrente = next;
30              percorrido[corrente] = 1;
31              ...
32      }
33 }
```

Figura 44 – Função que realiza o processamento do algoritmo de Djikstra.

A Figura 44 em sua assinatura tem passadas duas variáveis como parâmetro, sendo que ambas são preenchidas com valores retirados da interação do usuário com a imagem. O primeiro parâmetro consiste no vértice fonte da consulta e o segundo o destino final, conforme é apresentado na Figura 45 que demonstra a escolha de origem e destino no mapa no formato SVG.

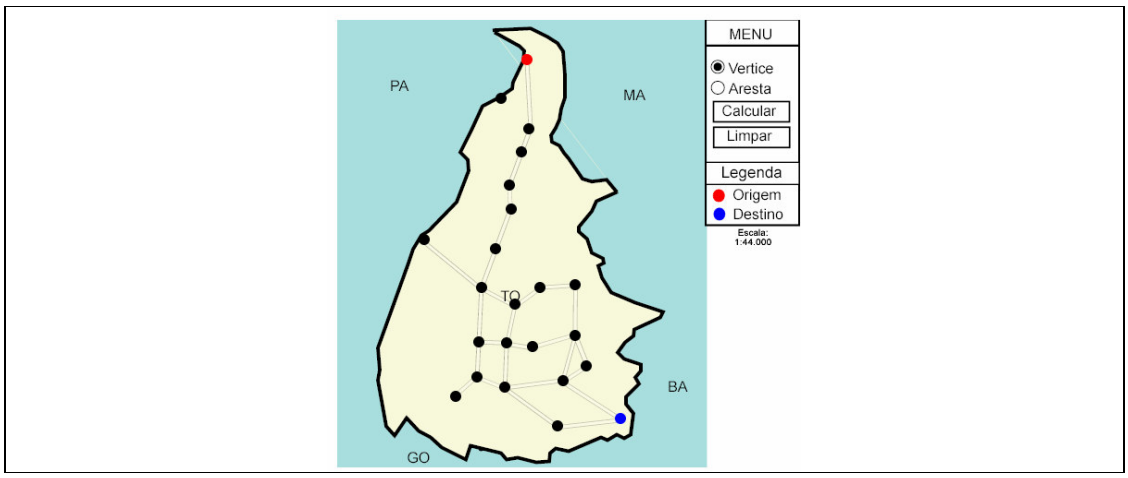

Figura 45 - Escolha de origem e destino na imagem SVG.

Na terceira e quarta linha da Figura 44, os vetores de “percorridos” e “distância” no índice que representam o vértice fonte são preenchidos com os valores 0 (zero) e 1 (um) respectivamente, denotando que o vértice já foi visitado e que sua distância até o vértice fonte é 0 (zero). Na sexta linha acontece o início da primeira iteração do algoritmo, que é realizada enquanto o vértice `corrente` for diferente do destino final passado como parâmetro e que como mencionado evita processamento desnecessário, visto que já se conhece o menor caminho. Na oitava linha é atribuída à variável auxiliar `menor_distância` um valor extremamente elevado, sendo que a variável é utilizada no decorrer do algoritmo para a comparação dos valores presentes no vetor de distância. Na nona linha a variável `distância_momento` recebe o valor referente ao vértice que o

índice indica. Nesta mesma linha pode-se mencionar a utilização da função `parseInt()` que força a variável ser interpretada como um inteiro.

Isto foi utilizado, visto que quando era necessário efetuar a soma entre duas variáveis ou entre variáveis e vetores, como será descrito mais a frente no trabalho, a Javascript ao invés de somar os valores presentes nas variáveis e vetores, realizava a concatenação dos valores, devendo-se mencionar também a utilização da função `eval()`, para a realização da soma dos valores.

Como mencionado, foi utilizada a matriz de adjacência para indicar a existência de adjacência entre os vértices, assim, cada linha do vértice era percorrida, sendo que se o vértice ainda não tivesse sido percorrido a rotina seria iniciada a realização das tarefas mencionadas é implementada na Figura 44, da décima-primeira até a décima-terceira linha. Na décima-quarta e décima-quinta linhas ocorrem os processos antes mencionados de transformação dos valores das variáveis em inteiro e a soma. Utilizando para isto a função `eval()`.

Da décima-sétima à vigésima linha é realizada uma operação condicional que verifica se o valor presente na variável `result` - soma do valor da variável `distancia_momento` e `aux` - é menor que o valor existente no vetor `distância`. Se verdadeiro, o vetor `distância` recebe o valor da variável `result` e o vetor `caminho` recebe o valor da posição `corrente`.

Outra operação condicional é efetuada da vigésima-segunda linha até a vigésima-quinta linha, sendo que é verificado se o valor atribuído ao vetor de distância no índice relacionado é menor que o valor contido na variável `menor_distancia`. Caso seja verdadeiro, `menor_distancia` assume o valor do vetor de `distancia` e a variável `next` que representa o próximo vértice a ser percorrido, recebe o valor do índice. Assim, finalizando a implementação do algoritmo de Dijkstra, na vigésima-nona linha a variável `corrente` que representa o vértice a ser percorrido na próxima iteração do algoritmo, recebe o valor atribuído à variável `next`. E, na trigésima linha, o vetor que informa os vértices “percorridos”, recebe o valor 1 (um) na posição indicada pela variável `corrente`, informando que o vértice está sendo percorrido.

Depois de processado todo o grafo no caso representado pela matriz de adjacência, era necessário mostrar ao usuário o menor caminho encontrado, no caso os valores armazenados no vetor "`caminho`". Para isto foi utilizado, ainda na função `Dijkstra` apresentada na Figura 44, um *loop* que é executado enquanto o vértice visitado no momento for diferente ao do vértice fonte. A Figura 46 apresenta a codificação que realiza a operação de imprimir o menor caminho no mapa.

```
1 function dijkstra(source,destin){
2     ...
3     var vertice = destin;
4     while(vertice != source){
5          id_aresta = (caminho[vertice]+""+vertice);
6          objet1 = init_doc.getElementById(id_aresta);
7          objet1.getStyle().setProperty('visibility', 'visible');
8          objet1.setAttribute("style","stroke:rgb(255,0,0);");
9          vertice = caminho[vertice];
10    }
11    ...
12 }
```

Figura 46 – Operação que imprime na imagem o menor caminho.

Conforme citado, as arestas do grafo criado sobre o mapa eram mantidas na imagem de forma transparente ao usuário. Com isto, para demonstrá-las ao usuário foi necessário realizar somente a alteração do atributo que dá visibilidade a estas arestas que eram obtidas da concatenação dos resultados presentes no vetor `caminho`. O resultado do processamento do algoritmo pode ser visualizado na Figura 47.

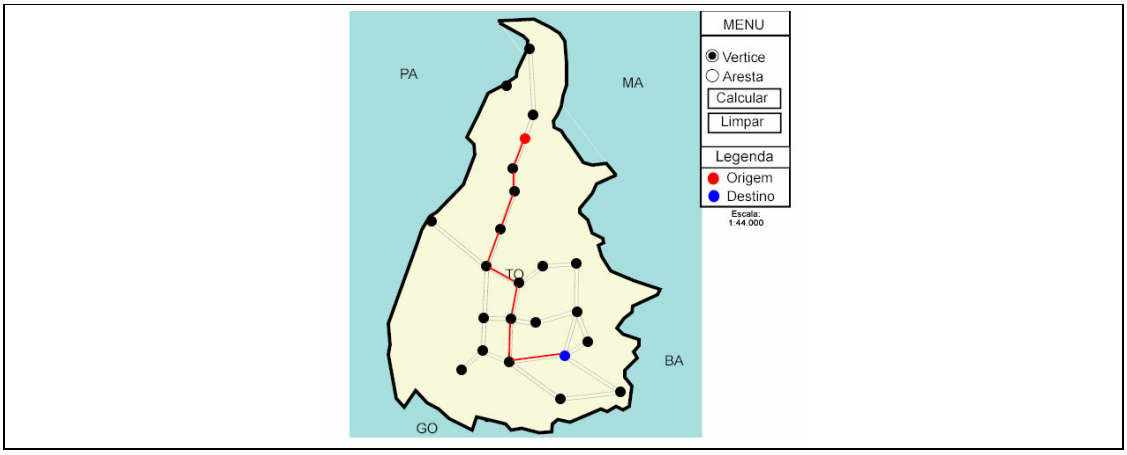

Figura 47 – Resultado do processamento do algoritmo de *Dijkstra*.

Juntas, as etapas desenvolvidas formam o protótipo do sistema conforme demonstrado na Figura 47, sendo que outras partes do código do sistema podem ser analisadas na seção de anexos deste trabalho em que o mesmo é disponibilizado.

5 CONCLUSÕES

Pode-se notar que a rápida evolução da Internet fez com que novas tecnologias fossem desenvolvidas. Com isto novas possibilidades, como as utilizações de imagens SVG para o processamento de informação foram incorporadas ao desenvolvimento de aplicações.

No trabalho se obteve como resultado o protótipo de um sistema que auxilie na localização do menor caminho, utilizando a teoria dos grafos e o algoritmo de Dijkstra, implementados utilizando a linguagem Javascript além da API DOM sobre imagens SVG.

O protótipo apresenta potencialidades que podem ser exploradas, já que, com a devida manipulação das funções criadas, tem-se um núcleo (*core*), composto por funções e variáveis responsáveis por registrar, alterar, consultar processar os grafos tanto em arquivos SVG quanto XML. Uma proposta de continuidade é o desenvolvimento de uma ferramenta na qual possam ser aplicadas quaisquer imagens SVG, sendo incorporadas ao sistema com a utilização da API DOM e Javascript.

Outra proposta de continuidade a ser mencionada é a disponibilização deste serviço para dispositivos móveis (*palm's* e celulares), já que o formato SVG possibilita sua utilização, além das imagens serem menores já que consistem em formato texto. Isto acaba se tornando desejável nestes dispositivos, levando-se em consideração que sua memória é relativamente pequena.

No decorrer do trabalho foram encontradas diversas dificuldades, dentre estas pode se citar a manipulação e registro dos dados em arquivos XML por meio das imagens SVG,

sendo que o *plugin* para a visualização das imagens não dava suporte para a utilização de objetos *ActiveX*. Além disto pode-se mencionar a utilização da linguagem Javascript, que em determinados momentos do desenvolvimento do sistema se mostrava um obstáculo a mais como na criação da matriz e no processamento do algoritmo de Dijkstra em que a linguagem trabalhava com os valores na forma de *string*, sendo que ao invés de soma os valores estes eram concatenados comprometendo o processamento do algoritmo. Em contra-partida o Javascript permitia a manipulação dos arquivos XML, sendo que esta também foi aplicada para a manipulação sobre a imagem SVG, sem que se tivesse que adotar uma linguagem para acessar os dados como ASP, JSP ou PHP e outra para manipulação.

Dentre alguns aspectos interessantes que foram considerados durante a implementação, pode se citar a constatação de que a utilização de grafos não orientados no sistema seria muito interessante, já que na descrição do problema no mundo real as estradas não possuem somente um sentido, sendo utilizadas para ambas as direções, - mão dupla - no caso como foi utilizado grafo orientado foi necessário assumir que os valores registrados para as arestas consistiam nos dados para ambos os sentidos do percurso.

Além disto, deve-se citar o material levantado durante o processo de pesquisa, além das referências apresentadas, tornando-se uma fonte de pesquisa visto o crescente interesse das pessoas e do mercado em geral em serviços deste porte e das imagens no formato SVG viabilizando a aplicação destas em ferramentas comerciais .

6 REFERÊNCIAS BIBLIOGRÁFICAS

(BASTOS, 2003) BASTOS, Eduardo; **Introdução a Grafos**. Disponível em <http://bdi.ucpel.tche.br/home/Members/eduardob/content/academico/Grafos.pdf> Acessado em 15 de agosto de 2004.

(CAGLE, 2002) CAGLE, Kurt; **SVG Programming: The Graphical Web**, New York: Apress, 2002.

(COELHO & FAGUNDES, 2004) COELHO, Alex, FAGUNDES, Fabiano; Protótipo de um Sistema de Auxílio à Localização no Ceulp/Ulbra utilizando Imagens Vetoriais Scalable Vector Graphics. **Relatório da disciplina de Prática de Sistemas de Informação I. Curso de Graduação em Sistemas de Informação**. Centro Universitário Luterano de Palmas, 2004.

(COMEN et al, 2002) COMEN, Thomas H., LEISERSON, Charles E., RIVEST, Ronald L.; **Algoritmos:** Teoria e Prática, tradução da 2ª Edição Americana Vanderberg D. de Souza – Rio de Janeiro: Campus, 2002.

(EISENBERG, 2003) EISENBERG, David J.; **SVG Essentials. Producing Scalable Vector Graphics with XML**, New York: Publish by O'Reilly & Associates, 2003.

(GOMES et al., 2002) GOMES, Adriano, VIEIRA, Leandro, ANTUNES, Marcel; **Integração XML e Banco de Dados**. Disponível em <http://www.xml.com.br/index.cfm?fuseaction=VeTexto&CD_Texto=156. Acessado em 17 de outubro de 2004.

(GOODMAN, 2001) GOODMAN, Danny; **JavaScript Bible, Gold Edition**, New York: Hungry Minds Inc., 2001.

(GUIMARÃES, 1998) Guimarães, José de O.; **Teoria dos Grafos**. Disponível em <http://www.dc.ufscar.br/~jose/> Acessado em 31 de agosto de 2004.

(KIRK & PITTS-MOULTIS, 2000) KIRK, C., PITTS-MOULTIS, N.; **XML – Black Book,** São Paulo: Makron Books, 2000.

(MARIANI, 2004) MARIANI, Antonio C.; **Conceito Básicos da Teoria de Grafos.** Disponível em <http://www.inf.ufsc.br/grafos/definicoes/definicao.html> Acessado em 22 de agosto de 2004.

(McGRATH, 1999) McGRATH, Sean; **XML Aplicações Práticas – Como Desenvolver Aplicações de Comércio Eletrônico**, Tradução de Vitor Hugo da Paixão Alves – Rio de Janeiro: Campus, 1999.

(PEARLMAN & HOUSE, 2003) PEARLMAN, Ellen., HOUSE, Lorien; **SVG for Web Developers**, New Jersey: Pearson Education, 2003.

(PREISS, 2000) PREISS, Bruno R.; **Estruturas de Dados e Algoritmos – Padrões de Projetos Orientados a Objetos com Java**, Rio de Janeiro: Campus, 2000.

(RABUSKE, 1995) RABUSKE, Renato A.; **Inteligência Artificial**, Florianópolis: Editora da UFSC, 1995.

(RITCHEY, 2000) RITCHEY, Tim.; **Programando Java & Javascript para Netscape 2.0**, São Paulo: Quark do Brasil Ltda, 2000.

(SIEDLER & DE SOUZA, 2002). SIEDLER, Marcelo S., DE SOUZA, Fernando F.; "Integração de Dados na Web Utilizando o Tamino XML Server", **Seminário de**

Computação FESURV/SENAC e Feira de Informática 2002: Anais do Seminário de Computação FESURV/SENAC e Feira de Informática, Rio Verde, 2002.

(TENENBAUM et al, 1995) TENENBAUM, Aaron M., AUGENSTEIN, Moshe J., LANGSAM, Yedidyah; **Estruturas de Dados usando C**, tradução: Teresa Cristina Félix de Souza, São Paulo: Makron Books, 1995.

(WATT, 2002) WATT, Andrew H.; **Designing SVG Web Graphics**, Indianapolis: Riders Publishing, 2002.

(W3C, 2004a) W3C – *World Wide Web Consortium;* **eXtensible Markup Language (XML)**. Disponível em <http://www.w3.org/XML/>. Acessado em 16 de outubro de 2004.

(W3C, 2004b) W3C – *World Wide Web Consortium;* **Scalable Vector Graphics (SVG)**. Disponível em <http://www.w3.org/SVG/>. Acessado em 08 de setembro de 2004.

(W3C, 2004c) W3C – *World Wide Web Consortium;* **Document Object Model (DOM)**. Disponível em <http://www.w3.org/DOM/>. Acessado em 14 de outubro de 2004.

(W3SCHOOL, 1998) W3 SHOOL – *World Wide Web School;* **Document Object Model (DOM)**. Disponível em <http://www.w3school.com/dom>. Acessado em 16 de setembro de 2004.