CDU 519.17

RUDINI MENEZES SAMPAIO
TURMA DE 1998

ESTUDO E IMPLEMENTAÇÃO DE ALGORITMOS DE ROTEAMENTO

TRABALHO DE GRADUAÇÃO

ORIENTADORES:
CELSO DE RENNA E SOUSA – IEC - ITA
HORÁCIO HIDEKI YANASSE – LAC - INPE

DIVISÃO DE CIÊNCIA DA COMPUTAÇÃO

SÃO JOSÉ DOS CAMPOS
CENTRO TÉCNICO AEROESPACIAL
INSTITUTO TECNOLÓGICO DE AERONÚTICA
1998

ESTUDO E IMPLEMENTAÇÃO DE ALGORITMOS DE ROTEAMENTO

Esta publicação foi aceita como Relatório Final de Trabalho de Graduação.

São José dos Campos , novembro de 1998.

Rudini Menezes Sampaio
Aluno

Prof. Dr. Horácio Hideki Yanasse
Orientador e Chefe do Laboratório Associado de Computação e Matemática Aplicada (LAC - INPE)

Prof. Dr. Celso de Renna e Sousa
Orientador e Chefe da Divisão de Ciência da Computação (IEC - ITA)

AGRADECIMENTOS

Ao prof. Horácio Hideki Yanasse pela objetiva e eficiente orientação, pelos seus valiosos conselhos e pela sua paciência e motivação.

Ao prof. Celso de Renna e Sousa pelo seu crédito e confiança.

Dedico esta obra "AO DIVINO CORAÇÃO TRANSBORDANTE DE FIRMEZA".

Sta. Teresa de Lisieux

"O MUNDO DOS FATOS NÃO CONDUZ NENHUM CAMINHO PARA O MUNDO DOS VALORES, PORQUE ESTES VÊM DE OUTRA REGIÃO"

Albert Einstein

RESUMO

Nesse documento apresenta-se um estudo de alguns dos principais problemas de roteamento em grafos, como Menor Caminho, Minimum Spanning Tree (Árvore de Custo Mínimo), Carteiro Chinês e Caixeiro Viajante, bem como o desenvolvimento e a implementação em um Software gráfico e prático de algoritmos que os solucionem em tempo hábil.

É discutido o projeto de desenvolvimento de bibliotecas dinâmicas (DLLs) para Visual C++ 5.0 que implementam classes de Vetores, Matrizes e Conjuntos, bem como uma classe de objetos do tipo Grafo, onde estão implementados os algoritmos de roteamento. Descreve-se também o desenvolvimento de um Aplicativo que se utiliza dessas DLLs e permite a visualização e manipulação de grafos sobre arquivos bitmap e a execução dos algoritmos desenvolvidos sobre diversos contextos, possibilitando o seu uso em casos de Fotos de Satélites, GPS, Mapas e outros.

ÍNDICE

1. INTRODUÇÃO

Este trabalho de graduação faz parte de um projeto em andamento no LAC (Laboratório Associado de Computação e Matemática Aplicada) do INPE (Instituto Nacional de Pesquisas Espaciais), que tem por objetivo desenvolver e implementar algoritmos em redes aplicadas a Sistemas Geográficos de Informação.

Este relatório está dividido em 4 (quatro) partes ou capítulos. Na 1° parte faz-se uma análise minuciosa dos principais problemas de roteamento estudados, acompanhada das explicações e escolhas sobre os algoritmos implementados. Na 2° parte apresenta-se os resultados das aplicações práticas do estudo realizado e de exemplos que mostram a utilidade do Software desenvolvido. Na 3° parte é apresentado o Manual do Usuário para o Software **Grafos** acompanhado de exemplos, aplicações e explicações sobre a utilização do produto final deste trabalho. Na 4° parte é apresentado o Manual do Programador, onde são explicados com mais detalhes a estrutura de dados utilizada para grafos, e como funcionam e são utilizados os principais métodos implementados.

Para a realização deste trabalho, fez-se inicialmente um estudo sobre os diversos problemas conhecidos de roteamento em grafos. Estudou-se minuciosamente as particularidades de cada um desses problemas, procurando obter possíveis soluções e otimizações.

Conhecido o problema e com um algoritmo otimizado e satisfatório para a resolução do mesmo, implementou-se o algoritmo e realizou-se vários testes de desempenho para avaliar o efeito das otimizações, visando melhorá-las.

Foi desenvolvido também um software em linguagem C++ para a visualização de grafos, manipulação de suas características e execução dos algoritmos desenvolvidos, utilizando o Microsoft Visual C++ 5.0.

O software desenvolvido contém a parte relativa a visualização de grafos (acompanhada de mapas ou figuras), a parte relativa a sua manipulação, e a parte relativa a execução dos algoritmos, deixando espaço reservado no código do programa para outros algoritmos que venham a ser implementados futuramente em trabalhos posteriores. Já está em funcionamento os algoritmos para solução dos problemas de menor caminho, árvore de cobrimento mínimo, carteiro chinês e caixeiro viajante. Para a solução desses, foi necessário a implementação de diversas rotinas que em conjunto os resolvem.

Para realizar as implementações, foi necessário um estudo aprofundado de técnicas de programação e Orientação a Objetos, e para a resolução teórica dos problemas, foi necessário um aprendizado paralelo de teoria da computação e técnicas de otimização de algoritmos para entendimento, familiarização e domínio do assunto.

Em suma, o trabalho desenvolvido neste projeto consistiu do estudo dos principais problemas de roteamento em grafos, da implementação dos algoritmos, de testes de verificação e desempenho e da confecção de manuais explicativos e bem documentados.

Para a implementação dos algoritmos, utilizou-se a linguagem C++ auxiliada pelas ferramentas MFC (Microsoft Foundation Class) através do software Microsoft Developer Studio (Visual C++ 5.0) utilizado também para digitação, compilação e depuração dos programas.

Foram desenvolvidos 2 (dois) projetos para este trabalho: uma DLL (Dynamic Link Library) chamada GrafAlgorit que é a implementação dos algoritmos de roteamento e diversas classes auxiliares como CSet (Conjuntos), CVetor, CMatriz e CGrafo, e o projeto do aplicativo chamado Grafos que utiliza os métodos da DLL GrafAlgorit e possibilita a visualização na tela da execução dos algoritmos em Grafos sobre mapas ou figuras. Tais projetos unidos formam o produto final deste trabalho, o **Software Grafos**.

Durante os testes realizados, verificou-se a exatidão do algoritmo utilizando-se vários exemplos que abordam todas as peculiaridades dos possíveis exemplares do problema, e observou-se o seu desempenho quanto à eficácia e tempo de execução, com o objetivo de tornar o código mais eficiente introduzindo melhorias.

A seguir são discutidos aspectos teóricos sobre os algoritmos de roteamento estudados que se dividem em métodos exatos e heurísticos. Posteriormente, são fornecidos detalhes sobre a implementação do software desenvolvido, incluindo-se as estruturas de dados para se armazenar grafos, distâncias, nós e arcos.

É descrito também como o software possibilita criar novos grafos, como estes podem ser modificados pela inclusão ou eliminação de nós ou arcos, como executar os algoritmos implementados e manter um bom nível de eficiência e visualização.

2. PLATAFORMA

Como já dito, o programa foi escrito em linguagem C++ e compilado pelo software Microsoft Visual C++ 5.0.

O software foi desenvolvido, executado e testado em um computador Pentium 233MHz MMX com 64MB de RAM, mas poderá ser utilizado em qualquer computador com Sistema Operacional Windows 95, 98 ou NT.

Os dados de eficiência e velocidade das implementações dos algoritmos descritos durante este trabalho são tomados em relação ao tipo de computador citado. Recomenda-se o uso do Software desenvolvido em computadores rápidos, pois os tempos envolvidos em alguns problemas desse tipo podem vir a ser significativos.

3. NOTAÇÃO E DEFINIÇÃO DE TERMOS

Um GRAFO é uma entidade G(N,A) consistindo de um conjunto finito N de nós e um conjunto finito A de arcos os quais conectam nós. Um ARCO conectando nós i e j pertencentes a N, será denotado (i,j).

Um grafo é não-orientado quando os arcos não possuem uma orientação (direção) específica. Um arco orientado (i,j) vai de i para j.

Quaisquer dois nós conectados por um arco ou quaisquer dois arcos conectados por um nó são chamados ADJACENTES. O GRAU de um nó em um grafo não orientado é o número de arcos incidentes nele.

Um CAMINHO em um grafo não orientado é uma seqüência de arcos, onde o nó final de um arco é o nó inicial do próximo (seqüência de nós adjacentes). Em um grafo orientado, os caminhos também são orientados, com arcos adjacentes simultaneamente chegando e saindo dos nós. Um caminho é SIMPLES quando cada arco aparece apenas uma vez na seqüência, e é ELEMENTAR quando cada nó aparece apenas uma vez. Um CICLO é um caminho onde o nó inicial é igual ao final.

Um ciclo é chamado EULERIANO quando passa exatamente uma vez sobre todos os arcos de um grafo, obviamente pela própria definição de ciclo, começando e terminando no mesmo nó. Um caminho ligando dois nós é chamado EULERIANO quando passa exatamente uma vez sobre todos os nós de um grafo, onde os nós terminais são diferentes.

Pelo TEOREMA de EULER, um grafo G conexo possui (a) um Ciclo Euleriano se e somente se G contém exatamente ZERO nós de Grau Ímpar, e (b) um Caminho Euleriano se e somente se G contém exatamente DOIS (2) nós de Grau Ímpar e estes são seus nós terminais.

Um nó i é CONEXO a um nó j se existe um caminho ligando de i a j. Um grafo não orientado é CONEXO quando existe um caminho ligando todos os pares de nós. Um grafo orientado é CONEXO quando é conexo o grafo resultante eliminando-se a direção de seus arcos, ou seja, seus arcos orientados se tornam não orientados.

Um grafo é FORTEMENTE Conexo quando sempre existe um caminho ligando todos os pares de nós, isto é, existe um caminho ligando i e j, $\forall$ i,j$\in$N, i$\neq$j. Note que grafos conexos não-orientados são sempre fortemente conexos.

Um SUB-GRAFO G'(N',A') de um grafo G(N,A) é um grafo tal que N'$\subset$N e A'$\subset$A. Uma ÁRVORE ou TREE de um grafo conexo não orientado é um subgrafo conexo que não contém ciclos. Portanto, uma árvore conexa com **t** nós tem exatamente **t-1** arcos. Uma SPANNING TREE de um grafo G(N,A) é uma árvore que contém todos os nós do conjunto N dos nós de G.

O TAMANHO ou Comprimento de um arco é um valor numérico associado a cada arco (i,j) do grafo e será denotado por l(i,j). O TAMANHO ou Comprimento de um caminho S entre dois nós de G é dado por $L(S)=\Sigma\, l(i,j)$, onde (i,j)$\in$S.

O LABEL (Rótulo) de um nó é um valor alfanumérico ou nome associado a um nó, por exemplo, o nome de uma cidade, com o objetivo de caracterizar o nó de forma apropriada.

4. PROBLEMAS DE ROTEAMENTO EM GRAFOS

4.1. MENOR CAMINHO (Shortest Path)

A questão mais comum que preocupa um motorista que deve dirigir de uma localidade em uma cidade até outra, é a escolha da rota. Entre as alternativas das rotas disponíveis, uma é escolhida tomando por base um critério como distância da viagem, tempo médio de percurso, segurança e confiabilidade da rota e outros mais.

Nas seções em que trataremos do problema de menor caminho, serão estudados e implementados eficientes métodos para determinação de caminhos que minimizam tempos de percurso (ou distância, ou custo) entre quaisquer dois pontos específicos ou entre conjuntos de pares de pontos em um grafo que representa uma rede de transporte.

Além de ser óbvio a aplicabilidade dessas técnicas para o problema de motoristas, correios ou serviços de emergência, existe uma motivação muito mais importante para se começar a estudá-los: a determinação dos menores caminhos aparece constante e consistentemente como um subproblema de problemas em grafos mais complexos. Portanto, os algoritmos de menores caminhos são usados como partes nas soluções desses problemas mais complexos.

Um grande número de algoritmos tem sido propostos durante os anos para a resolução de problemas de menores caminhos. Entretanto, todos podem ser vistos como variações dos mesmos temas e portanto os dois algoritmos que serão descritos nesse trabalho ilustrarão esses temas. O primeiro deles, o algoritmo de Dijkstra, resolve o problema de menor caminho entre um dado nó e os outros demais, enquanto que o segundo, o algoritmo de Floyd, resolve problemas de menor caminho entre todos os pares de nós.

Ambos os algoritmos podem ser usados em grafos orientados, não-orientados ou mistos, e ambos admitem que todos os arcos possuem comprimentos de arcos não negativos (nulo é possível). Enquanto essa suposição é razoável para o contexto de sistemas de serviços urbanos, onde os arcos representam normalmente distâncias ou tempos de percurso, poderá ser possível aplicações onde os arcos representam custos e portanto poderiam ser negativos.

Existem algoritmos para resolução de problemas de menores caminhos em grafos que contenham arcos negativos, e estes podem ser vistos relativamente como simples extensões dos dois algoritmos que descreveremos a seguir.

A preferência neste problema sobre grafos com arcos não negativos se dá pelo fato de serem estes os observados na maioria dos problemas combinatoriais práticos envolvendo grafos.

4.1.1. MENOR CAMINHO ENTRE UM DADO NÓ E OS OUTROS NÓS (Algoritmo de Dijkstra)

A necessidade mais comum em problemas em grafos é o de se precisar dos menores caminhos entre um nó fonte dado e os outros nós de um grafo. Podemos citar o exemplo de uma empresa de ônibus interurbanos que faz diversos trajetos, tais como Fortaleza - Belém, Fortaleza - Porto Alegre, Fortaleza - São Paulo e Fortaleza - Salvador. Como podemos observar, o nó fonte neste caso é a cidade de Fortaleza.

O algoritmo consiste basicamente em fazer uma visita por todos os nós do grafo, iniciando no nó fonte e encontrando sucessivamente o nó mais próximo, o segundo mais próximo, o terceiro mais próximo e assim por diante, um por vez, até que todos os nós do grafo tenham sido visitados.

Esse procedimento é realizado utilizando dois vetores como saída com elementos **D**[j] e **Cam**[j] para cada nó j, onde **D**[j] consiste do comprimento do menor caminho entre o nó fonte e o nó j, e **Cam**[j] consiste do nó predecessor imediato para ir a j no menor caminho do nó fonte ao nó j. Se **D**[j]=∞ ou **Cam**[j]=**0**, então não existe caminho entre o nó fonte e o nó j.

Seja **C**[i,j] a matriz das distâncias entre todos os nós do grafo e **min**(A,B) uma função cujo resultado é o menor valor entre A e B. Seja **n** e **Fonte** respectivamente o número de nós do grafo e o nó fonte. Sejam **V** e **S** respectivamente o conjunto de todos os nós e o conjunto dos nós já visitados. Com isso, podemos mostrar os passos para a solução exata desse problema:

PASSO 1: Inicialização. Fazemos **V**={1,2,...,n}, **S**={Fonte}, **D**[j]=**C**[Fonte,j] e **Cam**[j]=**Fonte**, para cada nó j do grafo. Faça **Cam**[Fonte]=0 e seja **i=0** um contador utilizado em passo 2.

PASSO 2: Repita **n-1** vezes:
Escolha um nó **w** em **V-S** tal que **D**[w] é mínimo
Acrescente **w** a **S**
Para cada nó **v** em **V-S** faça
D[v]=**min**(**D**[v] , **D**[w]+**C**[w,v])
Se **D**[v]=**D**[w]+**C**[w,v] então **Cam**[v]=w

PASSO 3: **i=i+1**
Se **D**[i]= ∞ e i≠**Fonte** então **Cam**[i]=0
Se i=**n**, Pare.
Se i<n, vá para PASSO 2.

No passo 1, fazemos a inicialização das variáveis. No passo2, procura-se pelo nó não visitado mais próximo à nó fonte (seja A esse nó), e atualiza-se os vetores D e Cam observando se existem caminhos melhores aos já obtidos dos outros nós até o nó fonte que passem pelo nó A. No passo 3, se algum elemento do vetor D for "infinito", o elemento do vetor Cam de mesmo índice será nulo, pois nenhum caminho para o nó fonte foi encontrado até o momento. Nesse passo, também verifica-se se todos os nós já foram visitados. Caso não, o algoritmo retorna ao passo2.

Como exemplo da execução desses passos, podemos observar a figura 1 abaixo. Seja 1 o nó fonte. Então, inicializamos com **Fonte**=1, **V**={1,2,3,4,5} e **S**={1}. Os vetores **D** e **Cam** possuem os seguintes valores: **D**=[0,10,∞,30,100] e **Cam**=[0,1,1,1,1], no passo 1.

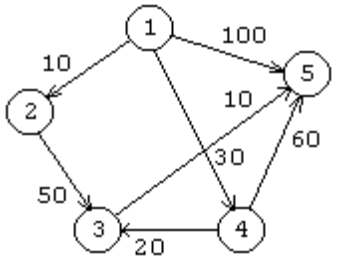


Figura 1: **Grafo para exemplificar o algoritmo de Dijkstra**

No passo 2, repetimos 4 vezes as mesmas operações.

Na primeira vez, **w**=2, **S**={1,2}, **D**=[0,10,60,30,100] e **Cam**=[0,1,2,1,1].
Na segunda vez, **w**=4, **S**={1,2,4}, **D**=[0,10,50,30,90] e **Cam**=[0,1,4,1,4].
Na terceira vez, **w**=3, **S**={1,2,3,4} **D**=[0,10,50,30,60] e **Cam**=[0,1,4,1,3].
Na quarta vez, **w**=5, **S**={1,2,3,4,5}, **D**=[0,10,50,30,60] e **Cam**=[0,1,4,1,3].

O vetor **D** representa o comprimento dos menores caminhos do nó fonte aos demais, enquanto **Cam** representa o nó predecessor imediato no caminho do nó fonte aos outros nós.

Por exemplo, para ir de 1 a 5 temos em **D** que o comprimento mínimo do percurso eqüivale a 60 unidades. Rastreando o caminho de 1 a 5 no vetor **Cam**, temos o nó anterior ao 5 é o 3. De 1 a 3, temos em **Cam** que o nó anterior ao 3 é o 4. De 1 a 4, temos que o anterior ao 4 é o 1, ou seja, consiste de um caminho direto.

Logo, fazendo o retrocesso temos 5-3-4-1. Portanto o menor caminho de 1 a 5 é o caminho 1-4-3-5.

Esse algoritmo possui várias outras denominações e/ou variações como algoritmo OSPF (Open Shortest Path First) de redes de computadores e Menor Caminho em Busca Ordenada.

4.1.2. MENOR CAMINHO ENTRE TODOS OS PARES DE NÓS (Algoritmo de Floyd)

É muito comum o caso de ser necessário o menor caminho entre todos os pares de nós de um grafo. Como exemplo podemos citar a preparação de tabelas indicando distâncias entre todas as cidades em mapas rodoviários de estados ou regiões, ou obter o menor caminho que parta de um nó dado, passe por alguns nós intermediários dados em ordem de prioridade e chegue em um nó final. Possui muitas aplicações, como entrega de encomendas em hora marcada por empresas distribuidoras (de refrigerantes ou etc..).

Uma solução óbvia é repetir o algoritmo de Dijkstra, descrito na seção anterior, sucessivamente para todos os nós do grafo. Outra solução, mais elegante, eficiente e fácil de implementar é conhecida como *algoritmo de Floyd*.

O algoritmo de Floyd é simples de descrever mas sua lógica não é fácil de se entender. A idéia geral desse algoritmo é atualizar a matriz de menores distâncias **n** vezes (onde **n** é o número de nós do grafo) procurando na **K**-ésima interação por melhores distâncias entre pares de nós que passem pelo vértice **K**.

Inicialmente, enumera-se os nós do grafo G(N,A) com inteiros positivos 1,2,...**n**. Então, duas matrizes $D^{(0)}$ e $P^{(0)}$, que após as **n** iterações virão a ser respectivamente a de distâncias e a de marcação do caminho, são inicializadas da forma:

$$d_0(i,j) = \begin{cases} l(i,j) \text{ , se } (i,j) \text{ existe} \\ 0 \text{ , se } i=j \\ \infty \text{, se } (i,j) \text{ não existe} \end{cases}$$

$$p_0(i,j) = \begin{cases} i \text{ , se } i \neq j \\ 0 \text{ , se } i=j \end{cases}$$

Os passos do algoritmo são descritos a seguir:

PASSO 1: Faça k=1;

PASSO 2: Para i e j variando de 1 a n faça
$d_k(i,j)=\text{Min}[\, d_{k-1}(i,j) \text{ , } d_{k-1}(i,k)+d_{k-1}(k,j)]$

PASSO 3: Para i e j variando de 1 a n faça
$$p_k(i,j) = \begin{cases} p_{k-1}(k,j) \text{ , se } d_k(i,j) \neq d_{k-1}(i,j) \\ p_{k-1}(i,j) \text{ , caso contrário} \end{cases}$$

PASSO 4: Se k=n, PARE
Se k<n, faça k=k+1 e vá para PASSO 2

No passo 1, inicializamos k em 1. No passo 2 e no passo 3, atualizamos os elementos d[A,B] da matriz D e os elementos p[A,B] da matriz P, observando se o comprimento do caminho de A a B é menor que o caminho de A a k mais o de k a B. Caso não seja, d[A,B] recebe d[A,k]+d[k,B] e p[A,B] recebe p[k,B].

O comprimento do menor caminho entre os nós i e j é dado pelo elemento $d_n(i,j)$ da matriz $D^{(n)}$, enquanto a matriz de marcação $P^{(n)}$ torna possível conhecer a rota do menor caminho a ser seguido. Observando o grafo da figura 2 e aplicando o algoritmo de Floyd, temos:

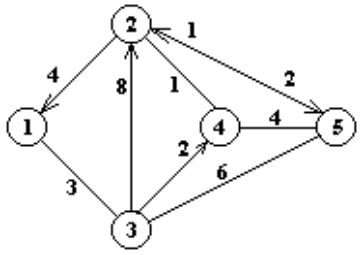


Figura 2: **Grafo para exemplificar o algoritmo de Floyd**

$D^{(0)}=$

0	∞	3	∞	∞
4	0	∞	1	2
3	8	0	2	6
∞	1	∞	0	4
∞	1	6	4	0

$P^{(0)}=$

0	1	1	1	1
2	0	2	2	2
3	3	0	3	3
4	4	4	0	4
5	5	5	5	0

Após k variar de 0 a 5, obtemos:

$D^{(5)}=$

0	6	3	5	8
4	0	7	1	2
3	3	0	2	5
5	1	8	0	3
5	1	6	2	0

$P^{(5)}=$

0	4	1	3	2
2	0	1	2	2
3	4	0	3	2
2	4	1	0	2
2	5	5	2	0

As matrizes $D^{(5)}$ e $P^{(5)}$ indicam, respectivamente, que o menor caminho entre o nó 1 e o nó 5, por exemplo, vale 8 unidades, e que o próprio é o caminho 1-3-4-2-5.

Verifica-se esses resultados pelo elemento $d_5(1,5)=8$ e pelos elementos:

Nó final=5 ↑
$p_5(1,5)=2$ ↑
$p_5(1,2)=4$ ↑ $\Rightarrow$ 1 , 3 , 4 , 2 , 5
$p_5(1,4)=3$ ↑
$p_5(1,3)=1$ ↑

4.2. ÁRVORE DE COBRIMENTO MÍNIMO (Minimum Spanning Tree)

O Problema da Spanning Tree mínima é encontrar uma entre todas as possíveis spannings trees (definido na seção 3) de um grafo G(N,A) com soma total de comprimentos de arcos mínima. Se o número de nós do conjunto N é n, todas as spannings trees de G conterão n-1 arcos.

Este problema aparece, por exemplo, no seguinte contexto: é dado um mapa de n cidades rurais com uma matriz listando as distâncias euclidianas entre todos os pares possíveis de cidades e deseja-se obter o menor comprimento de rodovias necessárias para unir todas elas. Outro contexto importante seria para auxiliar na decisão de onde se localizar postos de emergência ou delegacias de polícia em uma cidade, por exemplo.

Além de aplicações como estas, a solução desse problema é de grande utilidade para a solução de problemas combinatoriais em grafos mais complexos, tais como o do caixeiro viajante (Traveling Salesman Problem). É portanto de enorme importância o seu estudo, solução e melhorias.

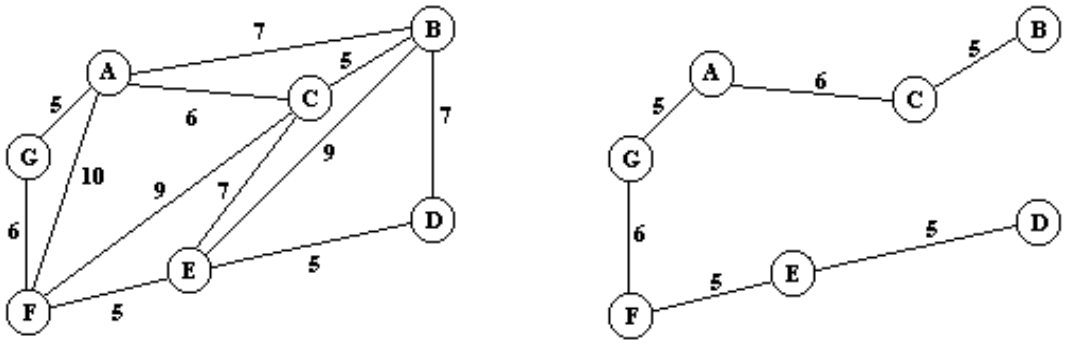
Uma solução para esse problema pode ser derivada, por exemplo, de uma idéia básica: escolhendo um nó arbitrário inicialmente, visitar todos os nós do grafo escolhendo como próximo a ser visitado o nó mais "***perto***" de um dos vértices já visitados. **n**>1 é o número de elementos do conjunto N de nós do grafo G. Utilizamos então os seguintes passos:

PASSO 1: Inicie a construção da Spanning Tree mínima, escolhendo um nó arbitrário, digamos i. Conecte i ao nó "mais próximo" dele, digamos j.

PASSO 2: Se todos os nós do grafo foram conectados, PARE. Se existem nós isolados, vá ao PASSO 3.

PASSO 3: Encontre o arco de menor tamanho que ligue um dos nós já conectados a um dos nós ainda não conectados. Introduza o arco na Spanning Tree mínima que está sendo construída, tornando já conectado um nó ainda não conectado. Volte ao PASSO 2.

Como exemplo de execução desses passos, vejamos a figura 3 e posteriores explicações, onde a figura 3(a) representa o grafo G e a figura 3(b) representa sua spanning tree mínima:



***(a)* Grafo G** **(b) Spanning Tree mínima de G**

Figura 3: O grafo G e a spanning tree mínima para exemplificar algoritmo

Seguindo os passos do algoritmo e escolhendo o nó A como inicial, temos:

ARCOS INTRODUZIDOS	NÓS JÁ CONECTADOS	NÓS AINDA NÃO CONECTADOS
-	A	B - C - D - E - F - G
(A,G)	A - G	B - C - D - E - F
(A,C)	A - C - G	B - D - E - F
(C,B)	A - B - C - G	D - E - F
(G,F)	A - B - C - F - G	D - E
(F,E)	A - B - C - E - F - G	D
(E,D)	A - B - C - D - E - F - G	-

Temos portanto que a spanning tree mínima de G, que está sendo gerada, consiste dos arcos da coluna "*Arcos Introduzidos*" da tabela acima e dos nós do conjunto N do grafo G(N,A), mostrado na figura 3.

4.3. PROBLEMA DO CARTEIRO CHINÊS (Chinese Postman Problem - CPP)

Considere o caso de um carteiro responsável pela correspondência em uma área mostrada pelo grafo abaixo:

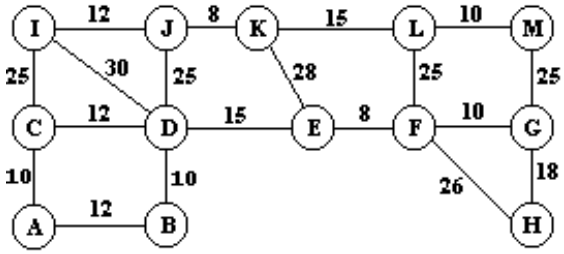


Figura 4: **Área da cidade de responsabilidade do carteiro**

O carteiro deverá sempre iniciar sua rota de entrega no nó A onde está localizado o Correio, deverá passar por todas as ruas (arcos) de sua área e retornar ao nó A.

A questão mais natural a se perguntar é: a fim de minimizar a distância total que o carteiro percorre, como deverá ser a sua rota de forma que ele passe por todas as ruas ao menos uma vez ?

Essa questão é conhecida como o Chinese Postman Problem, nome derivado do fato de ter sido no jornal Chinese Mathematics em 1952 a primeira vez em que esse problema foi discutido.

A história desse problema é muito interessante. No século XVIII, os moradores da cidade russa de Königsberg (hoje Kaliningrad, cidade onde o filósofo prussiano *Immanuel Kant* nasceu (em 1724) e passou toda a sua vida) queriam realizar um desfile que pudesse passar pelas sete (7) pontes sobre o rio Prevel apenas uma vez e deram o problema para o matemático suíço *Leonhard Euler* resolver.

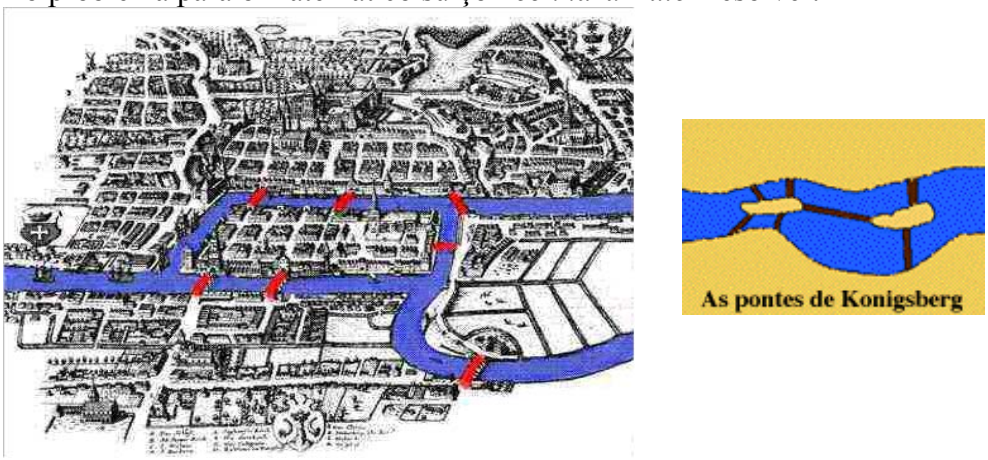


Figura 5: **As sete pontes de Königsberg**

Euler provou em 1736 que não existe solução para esse problema. Ele também obteve alguns resultados gerais que possibilitaram a motivação para as soluções do problema do carteiro chinês. Um deles bastante importante, nos diz que o número de nós de grau ímpar de um grafo não-orientado G é sempre par, visto que a soma dos graus de todos os nós em G é um número par pois cada arco é incidente a dois nós.

Outro resultado bastante importante, o teorema de Euler, já citado na seção 3, nos diz que para o carteiro efetuar um "Ciclo Euleriano", devem existir ZERO nós de grau ímpar no grafo. Em outras palavras, se quisermos realizar um ciclo euleriano em um grafo qualquer, este deverá ser modificado de modo a tornar de grau par todos os seus nós de grau ímpar, sem mudanças em sua estrutura.

Seja um grafo G(N,A) e $\mathbf{M} \subset N$ o subconjunto dos nós de grau ímpar. Então como dissemos anteriormente, a cardinalidade (número de elementos) de **M** é par e chamaremos esse valor de **m**. Realizando **m**/2 ligações dois a dois entre os elementos de **M**, aumentamos em uma unidade o grau desses nós, tornando-os de grau par.

Por exemplo, para o grafo da figura 4, N={A,B,C,D,E,F,G,H,I,J,K,L,M}, **M**={C,D,E,G,I,J,K,L} e **m**=8 (um número par).

Definimos a DISTÂNCIA de um par de nós, de um grafo conexo não-orientado, não necessariamente interligados por um arco, como sendo o COMPRIMENTO do menor CAMINHO entre os nós desse par. No grafo da figura 4, por exemplo, a DISTÂNCIA do par [E-L] eqüivale a 33 unidades.

Com essas observações podemos enumerar os passos para a solução do problema do carteiro chinês:

PASSO 1: Seja **M** o conjunto de todos os nós de grau ímpar de G(N,A). Digamos que existam **m** deles.

PASSO 2: Encontre a combinação de pares (pairwise matching) de nós de **M** cuja soma de sua DISTÂNCIAS é mínima.

PASSO 3: Encontre os arcos dos menores caminhos entre os dois nós que compõem cada um dos **m**/2 pares obtidos no passo 2.

PASSO 4: Para cada um dos pares obtidos em passo 2, adicione ao grafo G(N,A) os arcos obtidos no passo 3. O grafo $G^1(N,A^1)$ obtido não contém nenhum nó de grau ímpar.

PASSO 5 : Encontre um Ciclo Euleriano sobre $G^1(N,A^1)$. Esse ciclo obtido é a solução ótima para o problema do carteiro chinês no grafo original G(N,A). O comprimento do ciclo ótimo é igual a soma total dos arcos de G mais a soma de todos os arcos obtidos no passo 3 referentes ao Matching (Combinação de pares).

Resumidamente devemos executar no passo 2 um algoritmo conhecido como Matching Problem, no passo 3 o algoritmo de Floyd para os pares obtidos e no passo 5 o algoritmo de obtenção do Ciclo euleriano.

Serão descritos nas próximas seções os algoritmos do Matching Problem e do Ciclo Euleriano.

Observe a execução desses passos para o grafo abaixo.

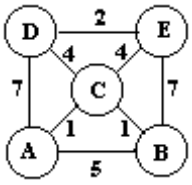


Figura 6: **Exemplo de execução do algoritmo do Carteiro Chinês**

No passo 1, para esse grafo temos N={A,B,C,D,E} e **M**={A,B,D,E}, logo **m**=4.

No passo 2, temos 3 combinações: (AB,DE) - (AD,BE) - (AE,BD). Dessas a que possui menor soma é a primeira com valor de AB=2 mais DE=2 igual a 4.

No passo 3, encontramos os arcos dos menores caminhos para a combinação escolhida (AB,DE). Para AB os arcos são (A,C) e (B,C), e para DE o único arco é o próprio (D,E). Logo os arcos encontrados são (A,C), (C,B) e (D,E).

No passo 4, obtemos o grafo G^1 adicionando a G os arcos anteriormente determinados. Se estes já existirem em G, como é o caso, eles serão arcos repetidos. Isso quer dizer que o carteiro irá passar duas vezes por eles.

Em passo 5, encontramos o ciclo euleriano. Para isso precisamos de um nó inicial. Suponhamos que seja o nó A. Assim, o carteiro faz, por exemplo, o percurso **ABCACBECDEDA**. Portanto, ele percorre todos os arcos ao menos uma vez e volta ao nó inicial. Esse é o caminho ótimo do carteiro chinês.

4.3.1. MATCHING PROBLEM

Como foi visto no exemplo anterior, o "matching problem" é a busca para se obter o conjunto de pares de nós de grau ímpar cuja soma dos valores dos menores caminhos em cada par seja mínima.

Para se obter esse conjunto de pares em uma abordagem teórica, e sendo **m** o número de nós de grau ímpar do grafo, então existem :

$$\prod_{i=1}^{m/2} (2i-1)$$ possibilidades distintas de combinações.

Portanto, para **m**=4, **m**=10 e **m**=20, existem respectivamente 3 , 945 e 655.000.000 de combinações. Com isso vemos que um algoritmo que tente resolver esse problema analisando todas as possibilidades de uma forma exaustiva não será muito adequado devido a sua lentidão à medida que **m** cresce.

Implementamos esse complexo algoritmo e realizamos substanciais melhorias, que serão citadas mais adiante.

Citaremos um exemplo simples e logo depois os passos para a sua solução. Sejam A,B,C,D,E,F os nós de grau ímpar (**m**=6). As notações (A,B), [A,B,C,D] e →, significam respectivamente, par A B, a execução do algoritmo sobre o conjunto de nós {A,B,C,D} e a saída referente a execução do algoritmo. Seja a denominação "cabeça" como sendo o primeiro elemento escolhido do conjunto para a procura das combinações. Seja a denominação "vice" como sendo o segundo para fechar o par com o nó cabeça. Assim sendo para o conjunto {A,B,C,D,E,F}, o nó cabeça é o A e o nó vice varia de B a F. Assim sendo:

	CABEÇA , VICE	NOVA EXECUÇÃO
[A,B,C,D,E,F] →	**(A , B)**	[C,D,E,F]
	(A , C)	[B,D,E,F]
	(A , D)	[B,C,E,F]
	(A , E)	[B,C,D,F]
	(A , F)	[B,C,D,E]

Para cada um dos conjuntos entre colchetes na coluna "Nova Execução" o mesmo procedimento é aplicado. Temos por exemplo para **[C,D,E,F]**:

[C,D,E,F] →	(C , D) (E , F)
	(C , E) (D , F)
	(C , F) (D , E)

Mostraremos agora o algoritmo desenvolvido, que realiza o procedimento mostrado no exemplo anterior. Seja SOMAPAR uma função que tem como entrada **TOTCONJ** (o conjunto total de nós de grau ímpar) e **num** (número de elementos de **TOTCONJ**), e tem como saída MINOW (matriz dos pares escolhidos) e VALOR (soma das distâncias dos pares em MINOW). Assim sendo:

NOME DA FUNÇÃO	**ENTRADA**	**SAÍDA**
SOMAPAR	TOTCONJ: conjunto	MINOW: matriz **n**x**2**
	NUM: inteiro	VALOR: inteiro

Representaremos por **VALOR.SOMAPAR(TOTCONJ , num)** a soma dos pares para o conjunto **TOTCONJ** com **num** elementos. Seja também a matriz D **n**x**n** como sendo a matriz dos valores dos menores caminhos entre os nós do grafo G, por exemplo D[A,E]=5 e D[A,B]=2 para o grafo ilustrado na figura 6.

Com essas observações podemos enumerar os passos para execução de SOMAPAR.

PASSO 1: Escolhe-se o **nó cabeça** como sendo o elemento de **TOTCONJ** na posição 1, na primeira posição. **MAX** recebe o maior valor inteiro possível.

PASSO 2: Se **num**≠2 então Vá para PASSO 3. Senão, **nó vice** recebe o elemento de TOTCONJ na posição 2 e então **VALOR=D[cabeça , vice]**. PARE.

PASSO 3: Para **nó vice** variando do elemento da posição 2 até a posição **num** faça: { **L=D[cabeça , vice]+VALOR.SOMAPAR(TOTCONJ-{cabeça , vice} , num-2);** Se **L<MAX** então **MAX=L** }. Note que na recursividade executamos para um outro conjunto que é igual ao **TOTCONJ** anterior menos os elementos **cabeça** e **vice**, reduzindo assim o seu número de elementos para **num-2**.

A matriz **nx2 MINOW.SOMAPAR(TOTCONJ , num)** representa os pares escolhidos pelo algoritmo do Matching Problem. Observe que o algoritmo mostrado é recursivo.

No passo 1, escolhemos o nó Cabeça do conjunto passado como parâmetro do algoritmo. No passo 2, verificamos se o número de elementos desse conjunto é igual a 2. Caso seja, o nó Vice é o outro elemento do conjunto (diferente do nó Cabeça) e o valor da função é a distância entre os nós cabeça e vice, dada por **D[cabeça, vice]**. Caso contrário, vai para o passo 3.

No passo 3, o número de elementos do conjunto é maior que 2. Fazemos então o nó vice variar entre todos os elementos do conjunto que são diferentes do nó Cabeça. O valor da função será o menor valor obtido da variação entre todos os nós Vice da sentença: **D[cabeça, vice]** mais o valor da recursividade para o conjunto resultante da eliminação dos nós cabeça e vice.

Um exemplo para ilustrar a resposta desse algoritmo pode ser visto na figura 7 a seguir obtida da execução do aplicativo desenvolvido, para um grafo com 18 nós e todos eles com grau ímpar. Os arcos resultantes do algoritmo do matching problem estão duplicados.

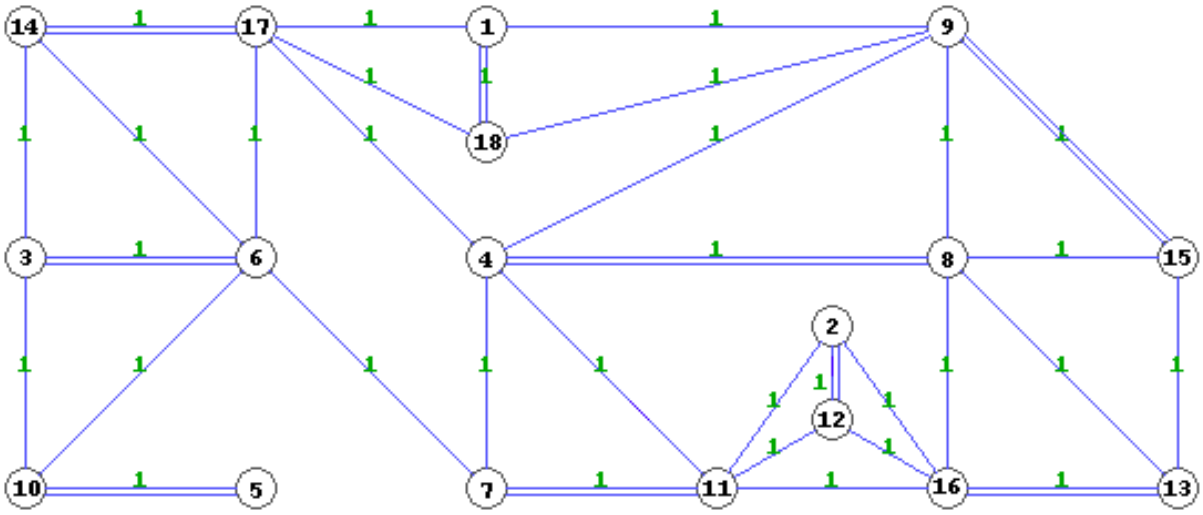


Figura 7: **Grafo com 18 nós (todos de grau ímpar) sobre o qual foi executado o algoritmo de resolução do Matching Problem.**

4.3.2. OTIMIZAÇÕES DO MATCHING PROBLEM

Para o algoritmo descrito anteriormente, conseguiu-se algumas melhorias bastante significativas, que reduziram o tempo de processamento drasticamente. Citaremos essas reduções de tempo adiante.

Várias dessas melhorias foram feitas apenas em âmbito de programação, não sendo portanto necessário descrevê-las. A principal delas realizada em nível teórico e combinacional, foi o fato de eliminarmos logo várias combinações de pares cuja soma das distâncias era muito alta. Com isso fazemos com que o algoritmo não processe indevidamente procurando por combinações que obviamente não fazem parte da solução ótima.

Isso foi feito introduzindo-se um novo parâmetro na função SOMAPAR descrita anteriormente que restringe um valor máximo para o qual a soma não pode exceder, ou seja o valor de L. Por exemplo, se temos 20 nós de grau ímpar e escolhemos o nó **cabeça** como sendo o A, então o método procurará o vice (19 opções), restando 18 nós de grau ímpar. De maneira recursiva um novo nó cabeça é escolhido e mais 17 opções para nó vice serão procuradas, e assim por diante. Temos então a uma *explosão combinatorial* de 19*17*15*13*11*9*7*5*3*1. Como *L=D[cabeça , vice]+ VALOR.SOMAPAR(**TOTCONJ** - {**cabeça** , **vice**} , **num** - 2)*, se restringíssemos o valor de L para um valor máximo calculado durante o processamento, podemos evitar que L execute a recursividade para o conjunto *TOTCONJ-{cabeça , vice}*.

Seja esse valor máximo chamado TOP, agora parte integrante dos parâmetros de SOMAPAR. Calcularemos L apenas quando *D[cabeça , vice]<TOP* e passamos para a recursividade um valor TOP igual *TOP-D[cabeça , vice]*, diminuindo-o cada vez mais. À cada rejeição em calcular L, diminuímos em uma unidade um fator do produto da explosão combinatorial, e à medida que o algoritmo vai executando, o número de rejeições vai aumentando também com a obtenção de soma de distâncias de pares mais em conta, mais próximas do valor ótimo.

Implementando tais melhorias, conseguimos os desempenhos mostrados na tabela abaixo na execução do algoritmo com relação ao tempo. A precisão do algoritmo não foi modificada, apenas reduzimos o número de procuras desnecessárias. Observe as drásticas reduções, para os grafos indicados na tabela, colocados como exemplo juntamente com o software. (Tempo em horas, minutos, segundos e milissegundos)

Tabela 1: Diminuições nos tempos devido as melhorias do Matching Problem

Nome do Arquivo	N° de nós de grau Ímpar	COM MELHORIAS	PURO
MATC2. GRF	6	0	0:0:0:05
MATC5. GRF	10	0:0:0:06	0:0:0:06
MATC8. GRF	14	0:0:0:09	0:0:3:57
MATC9. GRF	16	0:0:0:11	0:0:55:48
MATC10. GRF	18	0:0:0:88	0:16:46:95
MATC7. GRF	20	0:0:2:64	5:47:47:89
MATC7A. GRF	20	0:0:3:13	5:47:47:89
MATC6. GRF	20	0:01:20:19	5:47:47:89

4.3.3. OBTENÇÃO DO CICLO EULERIANO

A obtenção do Ciclo Euleriano depende da execução do algoritmo do "matching", pois o grafo não pode conter nós de grau ímpar. A execução desse algoritmo será sobre o grafo G^1 resultante da adição em G dos arcos obtidos pelo algoritmo do "matching".

A solução desse problema é teoricamente simples, mas seus pequenos detalhes aumentam incrivelmente a sua complexidade. Tentaremos mostrar nessa seção a solução total para este problema, incluindo seus pequenos detalhes.

Em primeiro lugar, é necessário para este problema um dado externo (do usuário ou de um banco de dados): O nó de partida; que em nossas analogias ilustramos como sendo o Correio. Em segundo lugar, quando falarmos em adjacência será em relação ao grafo G^1. Com essas observações, podemos agora enumerar em termos gerais os passos do algoritmo.

PASSO 1: Pergunte ao usuário qual é o nó de partida. Chamemo-lo de NóPart.

PASSO 2: Se não existem nós adjacentes a NóPart , então PARE. O Ciclo euleriano já foi descrito. Caso contrário, vá para PASSO 3.

PASSO 3: Dos nós adjacentes a NóPart escolha um cujo arco que o ligue ao nó de partida não seja uma Ponte (BRIDGE, ou seja, a sua eliminação não torna o grafo desconexo) e vá para ele. Elimine de G^1 o arco entre esses nós, ou seja, o nó de partida NóPart e o novo nó escolhido. Faça NóPart ser esse novo nó escolhido. Vá para PASSO 2.

O problema em si está no conceito de arco Ponte ou Bridge. Um arco é considerado Ponte em um grafo quando a sua eliminação deste torna o grafo desconexo. A definição de Conectividade já foi citada na seção 3, mas deveremos nos aprofundar mais um pouco nesses conceitos.

Para sabermos se um arco é ponte ou não, deveremos eliminá-lo do grafo e verificar se o grafo resultante dessa eliminação é ou não conexo. Caso seja, o arco não é ponte.

A denominação Ponte ou Bridge é derivada de um exemplo cotidiano de nossa sociedade. Se uma ilha é ligada ao continente por uma ponte e um desastre natural a destrói, então os moradores ficarão ilhados, separados do continente, sem *conexão* alguma com o restante da sociedade.

4.3.4. IDENTIFICAÇÃO DE PONTES

Nos passos acima para resolução do problema de obtenção do ciclo euleriano, surge-nos a seguinte pergunta no passo 3: ***Como averiguar se o arco escolhido é ou não Ponte.***

Para essa questão, temos também alguns passos a seguir.

PASSO 1: O grafo auxiliar G^2 recebe o grafo G^1.

PASSO 2: De G^2 eliminamos o arco que se quer verificar se é ou não ponte. G^2 recebe o grafo resultante dessa eliminação.

PASSO 3: Se G^2 for *conexo*, então o arco não é ponte. Caso contrário, o arco é ponte.

Os passos descritos executam bem a *Identificação de Pontes*, mas outro problema a se resolver e que está omisso nos passos acima, é como descobrir se um grafo é ou não conexo. Este problema é ligeiramente mais complicado, visto que teremos que fazer uma pesquisa geral por todo o grafo, e não mais apenas em um arco.

Outro fator que complica um pouco mais esse problema é o fato de podermos tratar tanto com grafos orientados como com grafos não-orientados. E para dar um pouco mais de complexidade ao nosso problema, podemos querer também saber se, além de ser conexo, o grafo é fortemente conexo. Tais definições podem ser encontradas na seção 3.

Se um grafo é não-orientado, que é o nosso caso para o problema do carteiro chinês, então a solução é trivial. Basta observar que se o grafo for desconexo, então para um nó A qualquer do grafo existirá um nó B que não se comunica com ele, ou seja, não existe um caminho que os ligue.

Portanto, executando o algoritmo de Dijkstra ou o de Floyd, ambos para obtenção dos menores caminhos, para um nó qualquer (podemos escolher o nó 1 por exemplo) de um grafo não-orientado, se existir em algum dos vetores de saída desses algoritmos um valor que seja *infinito* (na prática um número muito grande), então o grafo será desconexo. No entanto, aplicar o algoritmo de Dijkstra ou de Floyd para cada arco que se deseja saber se é ponte ou não, tornará excessivamente lenta a obtenção do ciclo euleriano. Outras rotinas foram desenvolvidas para resolver esse problema. Uma delas, por exemplo, denominada *Busca*, mantém um vetor que diz com 0's ou 1's se existem caminhos de um dado nó para os outros. Executando de modo recursivo, esta rotina rapidamente abarca todos os nós do grafo e, caso exista algum valor 0 no vetor citado, então o grafo é desconexo.

Com isso, resolvemos o problema de identificar a conectividade de um grafo não-orientado, e por conseqüência o problema de identificação de pontes para a resolução do problema de obtenção de ciclo euleriano, que por sua vez, é parte integrante da resolução do problema do carteiro chinês.

Como exemplo podemos observar o grafo da figura 8, para identificação de pontes:

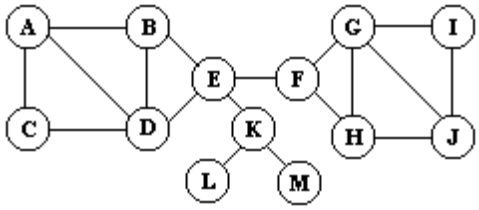


Figura 8: **Grafo G para identificação de pontes**

Observe que o arco (E,F) é uma ponte ou *bridge*, pois retirando-o do grafo acima, o grafo resultante mostrado abaixo será desconexo, pois não há como acessar o nó H a partir do nó A, por exemplo. Dividiu-se o grafo em dois.

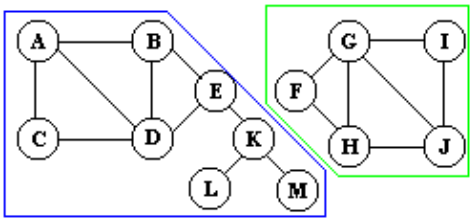


Figura 9: **Grafo resultante da eliminação de (E,F) de G**

Observe que o arco (E,K) é uma ponte também, pois retirando-o de G, resulta no grafo mostrado abaixo e, apesar de ser possível acessar H a partir de A, não é possível acessar o nó M a partir do nó B, por exemplo. Aqui também o grafo foi dividido em dois.

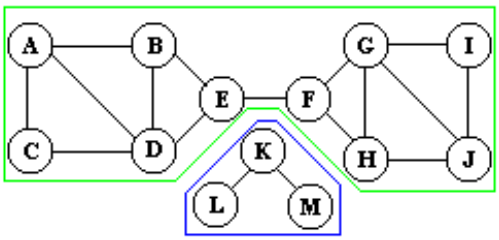


Figura 10: **Grafo resultante da eliminação de (E,K) de G**

4.3.5. CONECTIVIDADE

Entretanto, faremos agora uma abordagem mais geral sobre o problema de se identificar se um grafo é conexo ou não, se é fortemente conexo ou não.

Podemos considerar esse problema fazendo a seguinte abordagem: Seja N o conjunto de todos os nós e **n** o número de nós do grafo, logo N={1,2,...**n**}. seja S o conjunto que no final do algoritmo conterá os nós isolados e consideramos que inicialmente o único nó não pertencente a S é o nó 1, ou seja, S = N – {1}.

Façamos um passeio por todos os nós. Se o nó em que estivermos pertencer a S e se existir alguma ligação entre ele e um *nó não isolado*, então ele não é isolado, deixando de pertencer a S. Por outro lado, se o nó em que estivermos não pertencer a S e se existir alguma ligação entre ele e um nó pertencente a S, então este último não é isolado, deixando de pertencer a S. Se no final, S (o conjunto de nós isolados) não tiver mais elemento algum, então o grafo é conexo.

Colocando essas observações em linguagem computacional, teremos os seguintes passos:

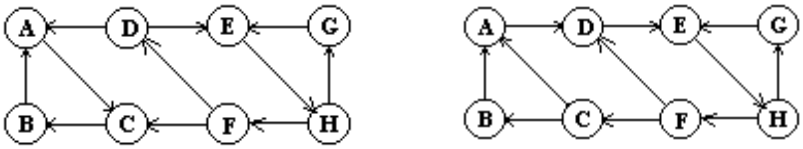
PASSO 1: Seja **n** o número de nós do grafo G(N,A). Seja N o conjunto de todos os nós do grafo e S o conjunto dos nós isolados e consideramos que o nó 1 é o único não isolado inicialmente. Logo N={1,2,...**n**} e S={2,3,...**n**}. Sejam **i** e **j**, valores inteiros menores que **n**. Seja D **n**x**n** a matriz dos valores dos menores caminhos entre todos os nós.

PASSO 2: Para **i** variando de **1** a **n** faça (*Passeio pelos nós*)
Se **i**∈**S** então (*Se i pertence a S*)
Para **j** variando de **1** a **n** faça
Se **j**∈(**N**-**S**) então (*mas j não é isolado*)
Se D[**i**,**j**] ≠ ∞ então (*e existe um caminho*)
S←**S**-{**i**} (*Então i não é isolado*)

Se **i**∈(**N**-**S**) então (*Se i não é isolado*)
Para **j** variando de **1** a **n** faça
Se **j**∈**S** então (*mas j pertence a S*)
Se D[**i**,**j**] ≠ ∞ então (*e existe um caminho*)
S←**S**-{**j**} (*então j não é isolado*)

PASSO 3: Se **S** for Vazio, então (*Se não há nós isolados*)
O grafo é conexo. (*O grafo é conexo*)
Se **S** não for Vazio, então (*Mas, se há nós isolados*)
O grafo é desconexo. (*O grafo é desconexo*)

Como exemplo, de grafos orientados simplesmente conexos e fortemente conexos, ilustramos abaixo os grafos da figura 11 (a) e (b) respectivamente.



***(a)* Simplesmente Conexo** **(b) Fortemente Conexo**

***Figura 11:* Exemplificações de conceitos de Conectividade**

Se todos os arcos do grafo (a) fossem não-orientados, então seria possível acessar todos os nós de qualquer um nó. Isso nos indica que este grafo é conexo, mas sem fazer esta operação, não é possível acessar o nó D a partir do nó C por exemplo. Logo este grafo é Simplesmente Conexo.

Executando o algoritmo de Floyd para menores caminhos, observaremos que na matriz dos comprimentos dos menores caminhos existem valores *infinitos* (∞). Com isso, outra forma de se verificar se um dado grafo é ou não simplesmente conexo, é modificando o grafo de forma a retirar a sua orientação.

Por outro lado, no grafo (b), distinto de (a) apesar de muito parecido, é possível acessar todos os nós a partir de um nó qualquer dado. Por exemplo, para acessarmos o nó C a partir do nó B, devemos descrever o seguinte caminho **B-A-D-E-H-F-C**. Isso indica que este grafo orientado é fortemente conexo.

Vários problemas mais complexos utilizam tais abordagens e algoritmos de identificação de conectividade para as suas soluções, mostrando-nos a importância de sua resolução.

4.4. PROBLEMA DO CAIXEIRO VIAJANTE (Traveling Salesman Problem - TSP)

Quando entregas ou visitas devem ser feitas para um número específico de pontos, o problema de roteamento que deve ser solucionado se torna um de cobrimento de nós. De todos os problemas de cobrimento de nós, o mais conhecido e fundamental é o Problema do Caixeiro Viajante (TSP, devido as iniciais de seu nome em inglês), cujo objetivo é o seguinte: *encontre a rota de menor distância que inicie em um dado nó de um grafo, visite todos os membros de um conjunto específico de nós do grafo uma única vez e retorne ao nó inicial.*

Está especificado que cada nó deve ser visitado uma única vez, mas implicitamente admite-se que isso seja possível.

Podemos estar interessados em um tipo particular de TSP onde *o grafo é* ***completamente conectado***, ou seja, é possível ir diretamente de qualquer nó para outro qualquer nó do grafo, sem passar por outro no meio do percurso. Admite-se ainda que o grafo *satisfaz a* ***desigualdade triangular***, ou seja, $d(i,j)<d(i,k)+d(k,j)$, para quaisquer 3 nós i, j e k, e que *a matriz de distâncias é simétrica.*

Resumindo, essa versão do TSP inclui completa conectividade no grafo, desigualdade triangular e simetria da matriz de distâncias. Chamaremos essa versão do TSP (Traveling Salesman Problem) por **TSP1**. Existem **(n-1)!/2** diferentes soluções para este problema (divide-se por 2 pois cada ciclo pode ser seguido em dois sentidos). Por exemplo, para grafos com apenas 10 e 20 nós, temos respectivamente 1,814,400 e $1.2*10^{18}$ soluções possíveis. Eventualmente, apenas num pequeno número dentre essas possibilidades fornecem o valor mínimo desejado e encontrar uma dessas soluções pode ser uma tarefa demasiadamente complexa.

Vários algoritmos exatos têm sido desenvolvidos ao longo dos anos, mas nem um deles é eficiente em termos de sua complexidade computacional. De fato, o problema de decisão do Caixeiro Viajante é considerado pelos matemáticos e cientistas da computação como sendo parte de uma classe de equivalência chamada **NP-Completo** de problemas difíceis onde não se conhecem algoritmos que fornecem respostas em tempo polinomial.

O Problema de Obtenção da Rota Mínima do Caixeiro Viajante é um problema de Otimização **NP-Hard**, sendo justificável o uso de uma heurística para sua resolução, visto que em uma boa parte dos casos práticos o número de nós do grafo é geralmente elevado.

Como o **TSP1** faz parte do conjunto dos problemas **NP-Completos** juntamente com o **TSP** geral, não podemos considerar que um seja mais fácil que o outro.

Isso também quer dizer que se existe um algoritmo eficiente e exato que resolva o TSP1, também deve existir um algoritmo eficiente e exato que resolva o TSP geral.

Também é verdade que a intuição nos ajuda a construir algoritmos heurísticos para o TSP1, não necessariamente exatos mas que nos fornecem resultados muito bons. Vários algoritmos heurísticos foram sugeridos na literatura, inclusive alguns que fornecem *manualmente* boas soluções.

A ênfase em se procurar por soluções heurísticas boas é também justificada pelo fato de na prática alguns parâmetros do problema, tais como tempos de percurso entre pontos serem voláteis (de acordo com o horário por exemplo), tornando o conceito de uma solução ótima apenas uma procura exigente por uma "excelente solução" que se modifica devido a tais volatilidades.

Apresentaremos a seguir um algoritmo heurístico para o TSP1, desenvolvido por *Christofides* em 1976. Este algoritmo consiste de 3 passos que são, na verdade, a aplicação de algoritmos já desenvolvidos nesse trabalho.

Algoritmo Heurístico para o TSP1 (Para grafos com completa conectividade, com a propriedade de desigualdade triangular e simetria da matriz de distâncias):

PASSO 1: Encontre a Spanning Tree mínima que passe por todos os **n** pontos. Chame essa árvore de **T**.

PASSO 2: Seja $\mathbf{n_0}$ o número de nós de grau ímpar dos **n** nós de **T** ($\mathbf{n_0}$ é sempre um número par). Encontre o comprimento mínimo da combinação de pares (*matching pairwise*) desses $\mathbf{n_0}$ nós, utilizando o algoritmo de matching; com os arcos do grafo dado inicialmente o qual respeita as condições do TSP1. Seja **M** o grafo (conexo ou não) dos arcos da solução ótima do matching. Seja **H** um grafo consistindo da união entre **M** e **T**, ou seja, **H=M∪T**. Note que se um ou mais arcos existem tanto em T como em M, então ele aparecerá duas vezes em H.

PASSO 3: O grafo **H** é Euleriano pois não contém nós de grau ímpar. Desenhe um Ciclo Euleriano sobre **H**, iniciando e terminando no nó especificado para ser o nó inicial do TSP1. Esse Ciclo Euleriano é a solução aproximada do problema do Caixeiro Viajante.

Esses três (3) passos básicos produzem um Ciclo que tem garantia de não ser maior que 50% do Ciclo Ótimo, conforme o teorema abaixo. Apesar de não ser uma limitante muito bom, esse é o melhor desempenho, para o pior caso, obtido para heurísticas conhecidas até hoje para o TSP1.

Teorema: Seja **H** o ciclo da solução aproximada obtida pelo algoritmo heurístico anteriormente citado e seja **TST** (Traveling Salesman Tour) o ciclo da solução ótima. Portanto, Se L(**H**) e L(**TST**) são respectivamente os comprimentos associados aos ciclos **H** e **TST**, então **L(H)<3*L(TST)/2.**

A demonstração desse teorema é trivial. Sejam **T** e **M** conforme definidos durante a exposição dos passos do algoritmo. Sendo **TST** um ciclo que passa por todos os **n** pontos e visita cada um exatamente uma vez, é óbvio que, se removemos qualquer um de seus arcos, geraremos um spanning tree $\mathbf{T^1}$ que não é exatamente a mínima com **n** nós e **n-1** arcos. Portanto, L($\mathbf{T^1}$)<L(**TST**). Como **T**, por definição é a mínima, então L(**T**)<L($\mathbf{T^1}$).

Logo L(**T**)<L(**TST**).

Do mesmo modo, identificando sobre **TST** os $\mathbf{n_0}$ pontos de grau ímpar com relação a **T** e executando o algoritmo do matching utilizando apenas os arcos contidos em **TST**, obteremos um conjunto de arcos que denotaremos por $\mathbf{M^1}$. Portanto, L($\mathbf{M^1}$)<L(**TST**)/2, visto que em **TST** esses arcos aparecem duplicados.

Como **M** é o conjunto de arcos obtido do matching sobre os $\mathbf{n_0}$ pontos, mas não restrito apenas aos arcos contidos em **TST** como é o caso de $\mathbf{M^1}$, então L(**M**)<L($\mathbf{M^1}$). Logo L(**M**)<L(**TST**)/2.

Como L(**H**)=L(**T**)+L(**M**), então L(**H**)<3*L(**TST**)/2.

CQD

Entretanto, apesar do algoritmo heurístico citado ter um limitante para o erro máximo de 50%, a experiência prática obtida no decorrer deste trabalho mostrou que em aplicações envolvendo grafos com mais de 50 nós, o algoritmo quase nunca fornece soluções com valor pior do que é 10% do ótimo.

Além do mais, é possível melhorar os resultados obtidos através de perturbações que serão descritos a seguir juntamente com alguns exemplos da execução dos passos do TSP1, procurando mostrar inclusive o efeito das melhorias.

4.4.1. MELHORIAS NA HEURÍSTICA PARA O TSP

Suponhamos que a rota apresentada na figura 12 seja a solução obtida pelo algoritmo de Christofides para o TSP1. Observe como o nó A abaixo aparece 2 (duas) vezes na rota gerada. É possível se aproveitar da desigualdade triangular e eliminá-lo em uma de suas aparições (X-A-C ou Z-A-Y), gerando 2 (duas) novas rotas possíveis (XC eliminando XA e AC, ou ZY eliminando ZA e AY) que serão obviamente menores que o da figura. Basta-nos apenas escolher a menor dessas 2 (duas) novas rotas (no caso, ZY).

Outra característica interessante de ser observada é: os arcos AC e BD fazem parte da solução, mas o custo total da rota seria minimizado se fossem trocados pelos arcos AB e CD. É possível fazermos o seguinte: se AC+BD>AB+CD, então trocar AC e BD por AB e CD. A essa melhoria chamamos OPT **2**, pois a troca é de 2 em 2 arcos. À melhoria feita de 3 em 3 arcos chamamos OPT **3** e assim por diante.

Caso fossem feitas todas as trocas de OPT **2** a OPT **n-1**, obteríamos a solução ótima para o problema.

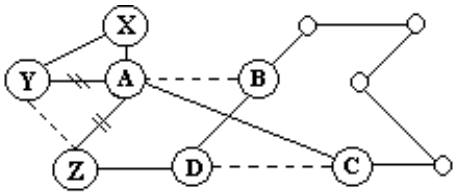


Figura 12: **Grafo para exemplificar otimizações no TSP**

Com essas observações, incluímos mais três passos ao algoritmo de Christofides para resolução do TSP1.

PASSO 4: Observe os nós de H que são visitados mais de uma vez no ciclo euleriano e melhore o caminho do caixeiro viajante aproveitando-se da desigualdade triangular. O ciclo resultante desse passo conterá apenas nós visitados uma única vez.

PASSO 5: Compare dois a dois os arcos do ciclo gerado no passo 4. Sejam os dois arcos observados, digamos, (A,C) e (B,D), conforme o exemplo da figura 11. Verifique se **d(A,C)+d(B,D)>d(A,B)+d(C,D)**. Caso seja, troque os arcos (A,C) e (B,D) por (A,B) e (C,D) no percurso do caixeiro viajante.

PASSO 6: Compararemos três a três arcos do ciclo gerado no passo 5. Sejam os arcos considerados, digamos, (A,B), (G,H) e (R,S). Das seguintes somas, observamos qual é a menor **d(A,B)+d(G,H)+d(R,S), d(A,G)+d(B,R)+d(H,S)** ou **d(A,R)+d(H,B)+d(G,S)**. A configuração dos arcos tomada inicialmente será trocada pela combinação de arcos cuja soma for a menor.

Com isso, conseguimos uma excelente melhoria para o percurso do caixeiro viajante, conforme podemos observar nos exemplos a seguir.

Como exemplo da resolução do problema do caixeiro viajante seguindo-se os passos do algoritmo de Christofides para o TSP1, consideremos o seguinte problema:

A partir do nó 1, fazer entregas nos nove pontos restantes realizando o mínimo percurso possível, e sabendo que existem caminhos diretos entre todos os pontos segundo a seguinte matriz de distâncias:

Tabela 2: Matriz das distâncias para o problema do Caixeiro Viajante (grafo da figura 13)

	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
1	0	25	43	57	43	61	29	41	48	71
2	25	0	29	34	43	68	49	66	72	91
3	43	29	0	52	72	96	72	81	89	114
4	57	34	52	0	45	71	71	95	99	108
5	43	43	72	45	0	27	36	65	65	65
6	61	68	96	71	27	0	40	66	62	46
7	29	49	72	71	36	40	0	31	31	43
8	41	66	81	95	65	66	31	0	11	46
9	48	72	89	99	65	62	31	11	0	36
10	71	91	114	108	65	46	43	46	36	0

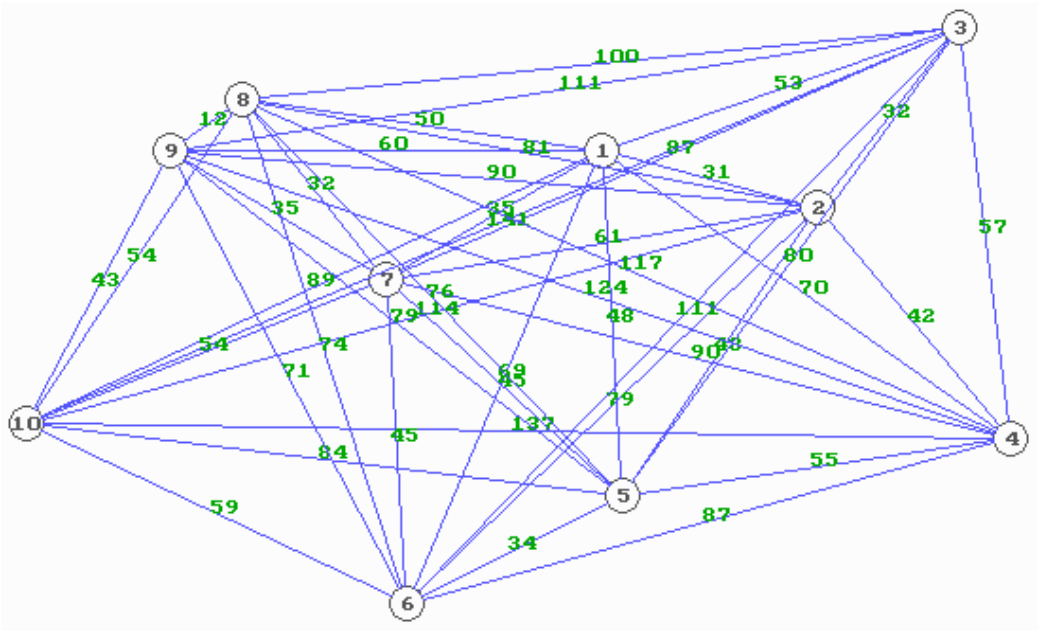


Figura 13: **Distribuição de pontos para problema do caixeiro viajante**

Executando o passo 1 do algoritmo, obtemos a spanning tree mínima **T** apresentada na figura 14. Executando o passo 2, identificamos os 6 pontos de grau ímpar indicados por um asterisco (*) no grafo e os arcos resultantes do matching em **M** indicados em pontilhado. Logo **H=T∪M** é a união dos arcos cheios com os pontilhados.

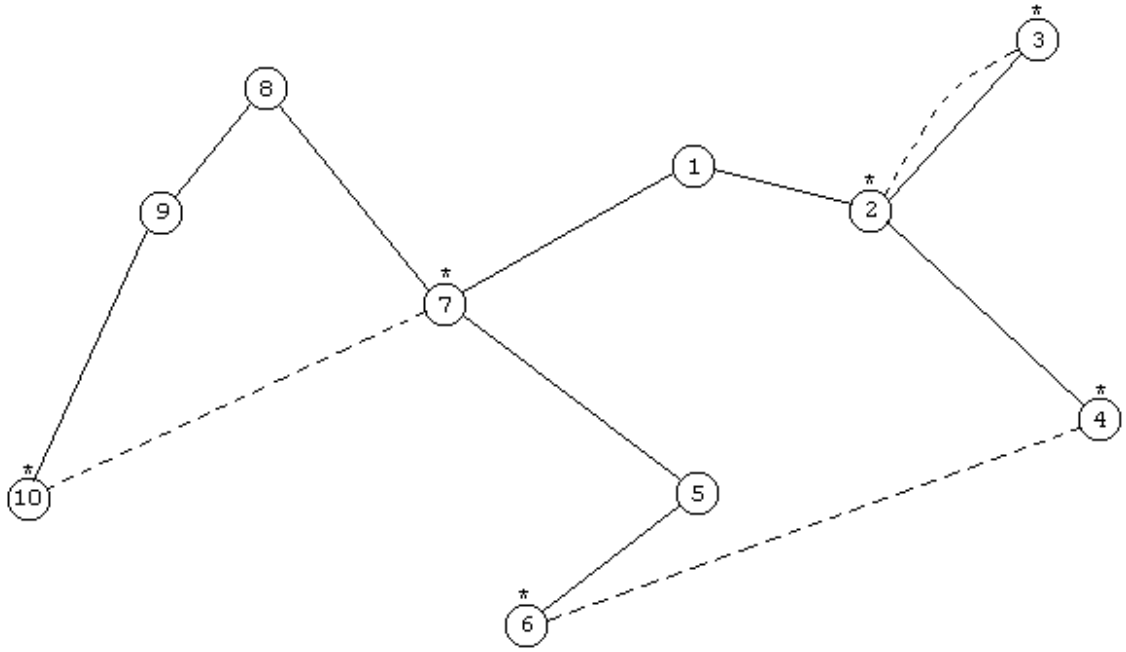


Figura 14: **Spanning Tree mínima (em arcos cheios), nós de grau ímpar (identificados com *) e os arcos obtidos do Matching (em tracejado).**

No passo 3, desenhamos o ciclo euleriano sobre **H**.

No passo 4, verificamos que os nós 2 e 7 aparecem duas vezes em H. As opções para eliminar uma das ocorrências de 7, por exemplo, é eliminar os arcos (8,7) e (7,1) adicionando o arco por (8,1) ou eliminar os arcos (10,7) e (7,5) adicionando o arco (10,5). Observando a figura, vemos que a melhor opção é a primeira.

Efetuando o passo 5, obtemos a solução ótima do problema mostrada na figura 15. Seria desnecessário executar o passo 6 neste caso.

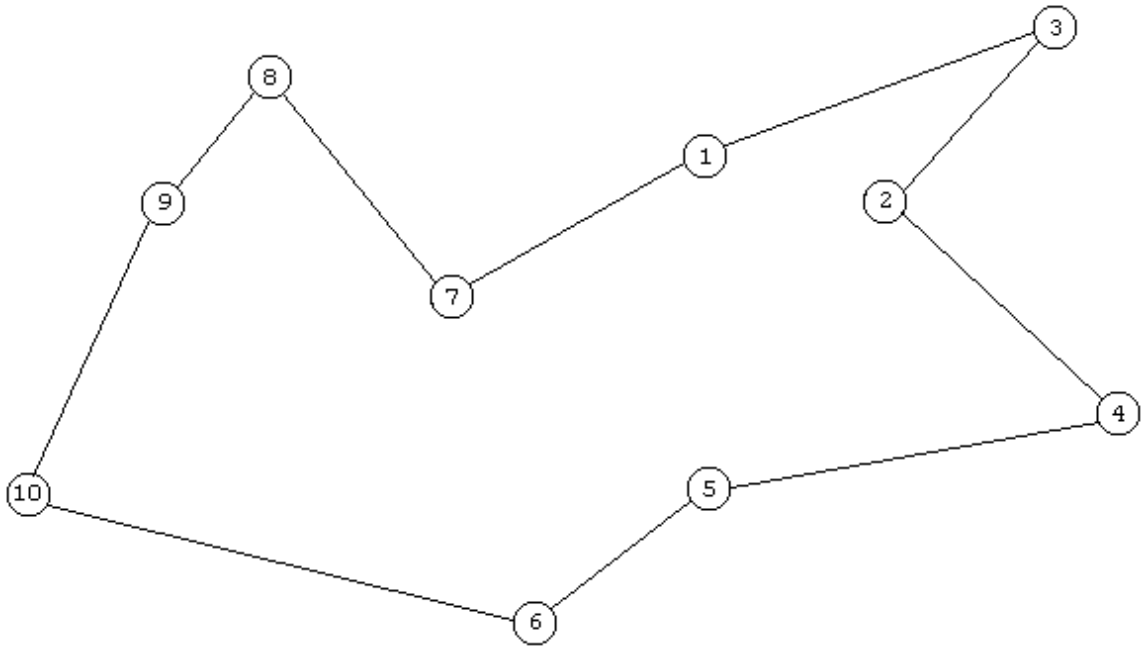


Figura 15: **Solução ótima do problema do caixeiro viajante conseguida no passo 5 do algoritmo desenvolvido para o TSP1.**

Portanto, como pudemos observar, o algoritmo heurístico desenvolvido para o TSP1 conseguiu realizar uma excelente aproximação para a solução ótima (na verdade, neste exemplo, obteve a ótima). A heurística desenvolvida resolve muito bem problemas com mais de 50 nós. Em casos práticos, obteve-se soluções com no máximo 10% de erro em relação ao ótimo.

5. APRESENTAÇÃO DOS RESULTADOS

5.1. UMA BREVE INTRODUÇÃO SOBRE O SOFTWARE

O Software Grafos desenvolvido durante este trabalho consta de dois módulos: o projeto de GrafAlgorit.DLL e o projeto do aplicativo Grafos.EXE.

O projeto da DLL consta de 1796 linhas de código CPP e 364 linhas de código H (Header file) totalizando 2160 linhas de código. O projeto do Aplicativo consta de 4509 linhas de código CPP e 1291 linhas de código H, totalizando 5800 linhas de código. Unindo o projeto da DLL ao do Aplicativo, somamos em torno de 8000 linhas de código.

Achamos importante o desenvolvimento deste trabalho justamente pelo fato de mesclar a Teoria e a Prática, bem como a sua importância em um projeto em desenvolvimento no INPE. Utilizou-se conhecimentos e conceitos obtidos durante o curso universitário de disciplinas como Estrutura de Dados, Estruturas Discretas para Computação, Linguagens de Programação, Engenharia de Software, Sistemas Operacionais, Teoria da Computação e Computação Gráfica, ministradas no curso de Engenharia de Computação do ITA.

FUNÇÕES OFERECIDAS PELO APLICATIVO:

VISUALIZAÇÃO:

O aplicativo permite visualizar um grafo direcionado ou não com tela de fundo como um mapa ou figura escolhido pelo usuário, e com desenho na tela dos nós, arcos, custos e direção dos arcos. Existem várias opções de mudança de cores, larguras dos traços do desenho e outras Características do Grafo.

MANIPULAÇÃO:

O aplicativo permite modificar o grafo pela inclusão ou exclusão de Nós e Arcos, pela mudança da posição de Nós, pela translação do grafo inteiro ou pela mudança na escala tanto horizontal como vertical e pela mudança dos "Labels" dos Nós.

EXECUÇÃO DOS ALGORITMOS:

CONECTIVIDADE:

Pela opção "VER CARACTERÍSTICAS DO GRAFO" em "EXIBIR" é informado se o grafo é fortemente conexo, simplesmente conexo ou desconexo.

OUTROS ALGORITMOS:

Pela opção "ALGORITMOS" podemos executar:

MENOR CAMINHO DE DIJKSTRA:

Possibilita visualizar tanto escrito como graficamente o comprimento e o percurso do menor caminho entre dois nós dados: o nó Fonte e o nó Destino.

MENOR CAMINHO DE FLOYD:

Possibilita visualizar as matrizes do comprimento e do percurso do menor caminho geradas pelo algoritmo de Floyd.

Possibilita também visualizar tanto escrito como graficamente o *percurso mínimo necessário para se chegar de um Nó a outro passando por Nós intermediários dados em ordem de prioridade.* Por exemplo: 23 45 12 30, parte do nó 23 e chega a nó 30 passando antes pelos nós 45 e 12, nessa ordem.

MINIMUM SPANNING TREE (ÁRVORE DE CUSTO MÍNIMO):

Possibilita visualizar graficamente o processo de visita a todos os nós do grafo, gerando no final a ***árvore*** de custo mínimo.

CARTEIRO CHINÊS:

Possibilita visualizar tanto escrito como graficamente o processo do percurso por todos os ***arcos*** do grafo, retornando ao nó inicial, que é pedido no início da execução.

CAIXEIRO VIAJANTE:

Possibilita visualizar tanto escrito como graficamente o processo do percurso por todos os ***nós*** do grafo, retornando ao nó inicial. Na execução desse algoritmo omite-se o desenho dos arcos do Grafo para facilitar a visualização, como pode ser visto na figura 17 abaixo.

OBSERVAÇÃO:

As figuras mostradas a seguir foram extraídas da tela do microcomputador obtida da execução do aplicativo desenvolvido neste trabalho. Todas elas estão em modo de ***"Vista Aérea"*** permitindo visualizar o grafo completo eliminando-se as barras de rolagem. Fez-se isso para ser possível a impressão destas figuras neste relatório. A perda de nitidez é devido a este modo de ***"ZOOM"*** criado para facilitar a visualização global do grafo. Voltando-se ao modo Normal, voltam as barras de rolagem e a sua nitidez característica.

5.2. EXEMPLOS

5.2.1. HEURÍSTICA DO CAIXEIRO VIAJANTE

Para exemplificar a utilização do aplicativo, inicialmente montou-se um grafo semelhante aos grafos das figuras 13, 14 e 15, e executamos o algoritmo do Caixeiro Viajante. O resultado dessa execução pode ser visto em vermelho na figura 16 abaixo.

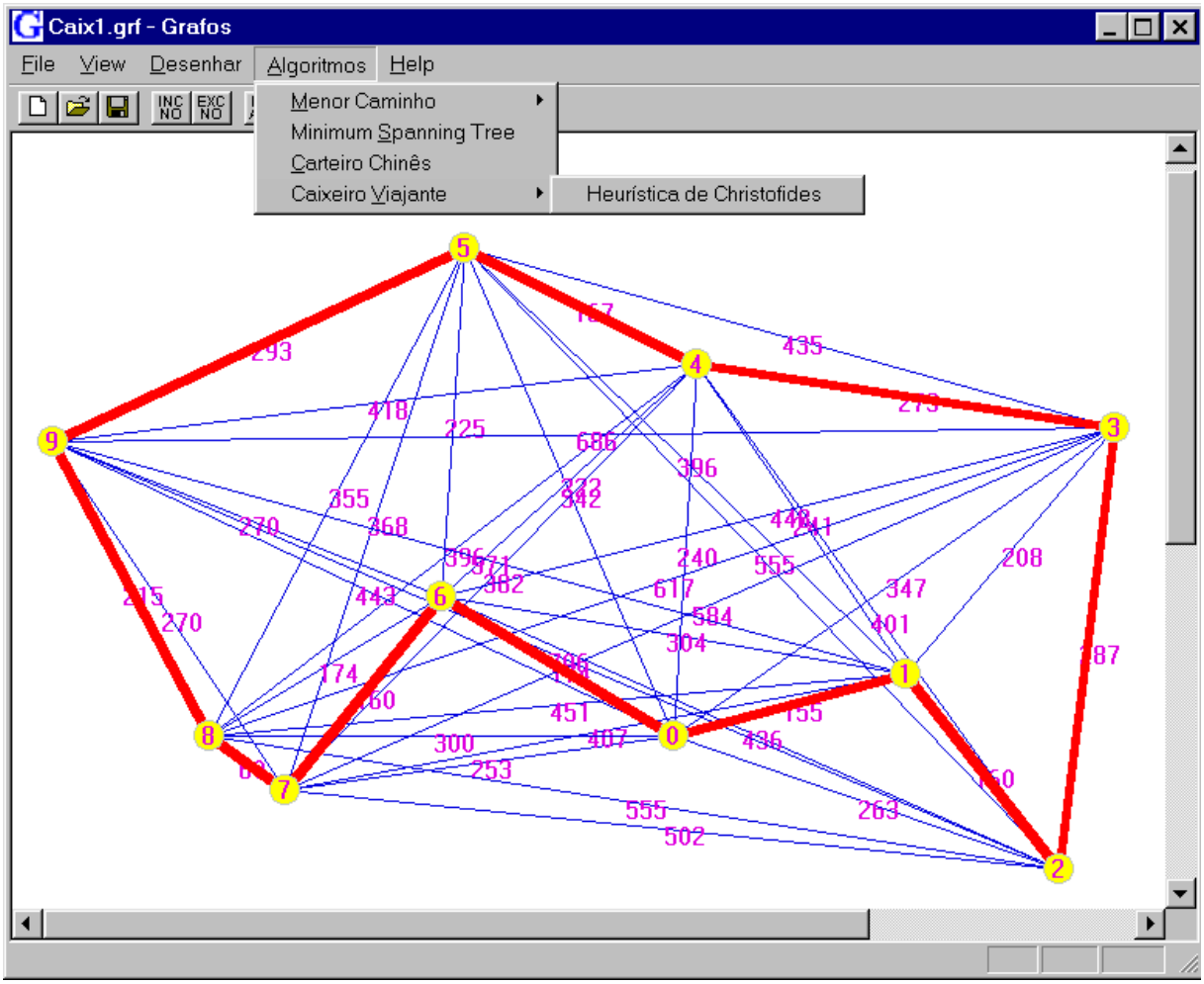


Figura 16: **Execução da Heurística do Caixeiro Viajante no Aplicativo.**

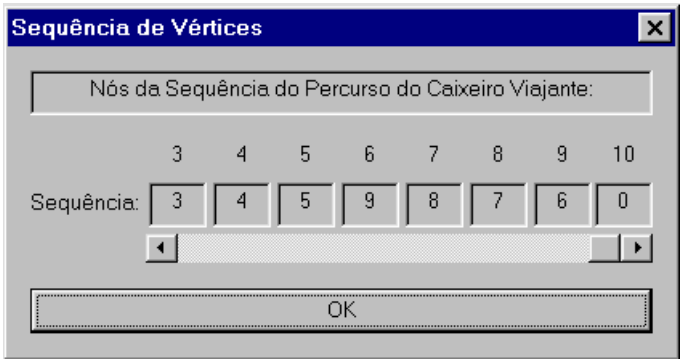


Figura 17: **Seqüência de nós do percurso do caixeiro viajante da figura 16.**

5.2.2. JOGADAS DO CAVALO NO TABULEIRO DE XADREZ

Com o objetivo de exemplificar a funcionalidade do Software, montamos um grafo referente a todas as possíveis jogadas do Cavalo no tabuleiro de xadrez. A partir desse grafo podemos exemplificar a aplicação de alguns algoritmos desenvolvidos. Esse grafo possui 64 nós (de 0 a 63) visto que um tabuleiro de xadrez é uma matriz **8x8**. Na figura 18 a seguir, observamos o tabuleiro de xadrez com as possíveis jogadas do cavalo em cor Azul.

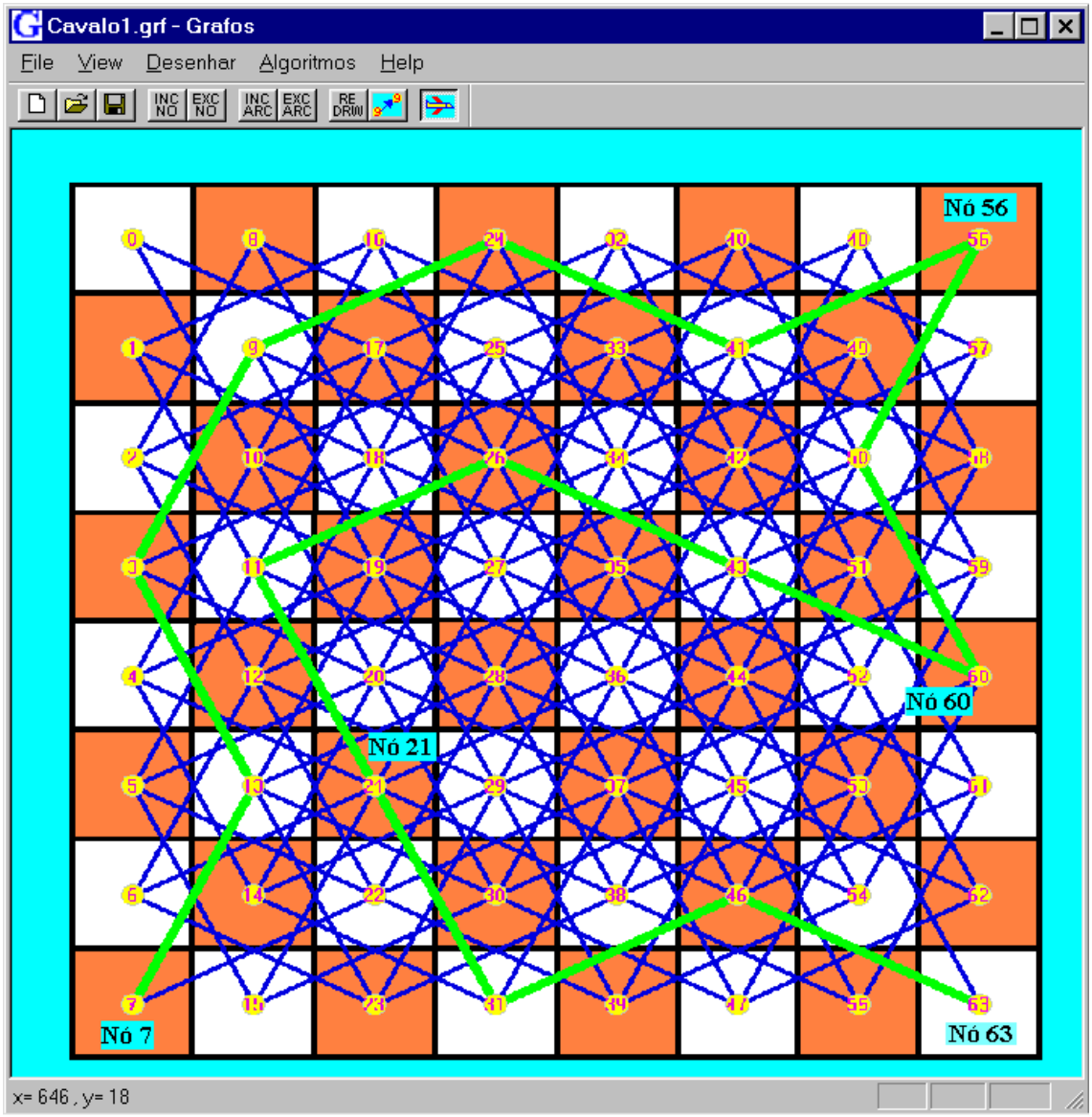


Figura 18: **Exemplo da execução do Algoritmo de Floyd iniciando no Nó 7, terminando no Nó 63 e passando pelos Nós intermediários 56, 60 e 21 dados em ordem de prioridade, para o caso dos movimentos do Cavalo no Tabuleiro de Xadrez.**

O grafo da figura 18, representa todos os movimentos possíveis do cavalo no tabuleiro de xadrez. Executando o algoritmo de Floyd para este grafo para a seqüência 7 56 60 21 63, obtemos o percurso em **verde** na figura 18, as matrizes **D** e **Cam** (ver seção 4.1.2) ilustradas na figura 19 a seguir e a seqüência de nós desse percurso na figura 20 abaixo.

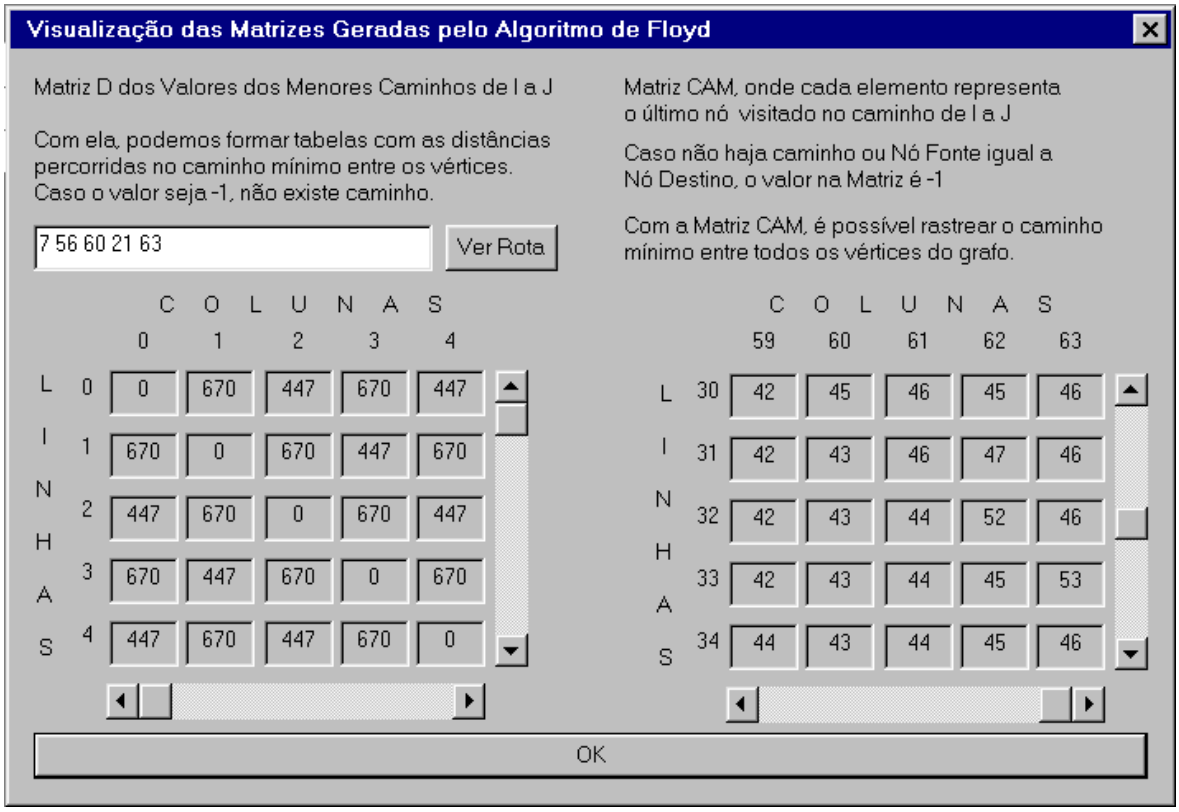


Figura 19: **Matrizes D e Cam obtidas da execução do algoritmo de Floyd para o grafo da figura 18. Note a opção de "VER ROTA"**

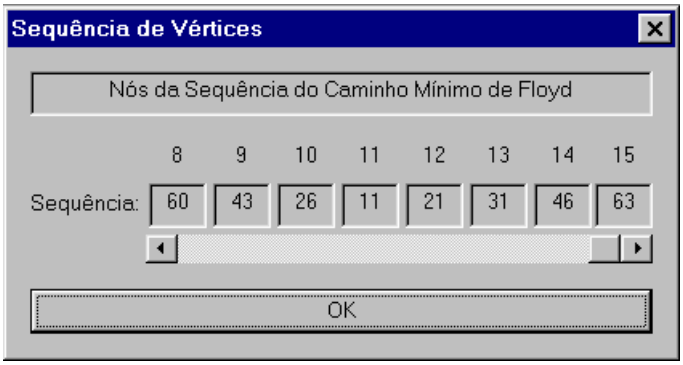
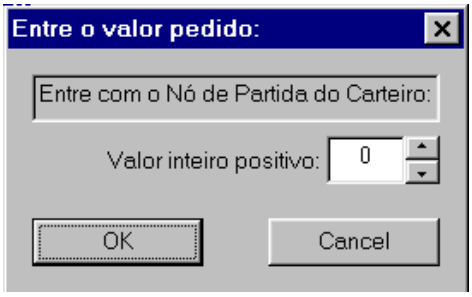
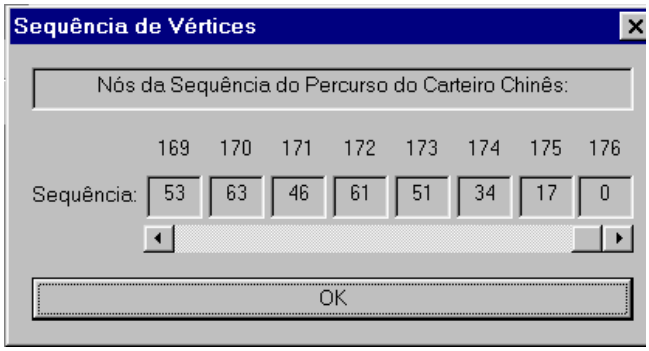


Figura 20: **Seqüência de 15 nós do percurso do caminho mínimo obtido da execução do algoritmo de Floyd para os nós 7, 56, 60, 21 e 63, ilustrado na figura 18.**

Executando o algoritmo do Carteiro Chinês para o mesmo grafo e iniciando no nó 0, obtemos um total de 176 nós no percurso, conforme ilustrado na figura 21 abaixo, mas infelizmente não foi possível mostrar a visualização da execução desse algoritmo de forma impressa devido a dinâmica desse algoritmo.



***(a)* Pedido do nó de partida para o algoritmo do carteiro chinês**



***(b)* Seqüência de 176 vértices do percurso do Carteiro Chinês**

***Figura 21:* Caixas de diálogo da execução do Algoritmo do Carteiro Chinês**

Utilizando agora as funções de manipulação do Aplicativo (botão **EXC NO**), eliminamos alguns Nós do Grafo da figura 18 e executamos o algoritmo do Caixeiro Viajante, obtendo a seqüência e o percurso ilustrados nas figura 22 e 23 abaixo.

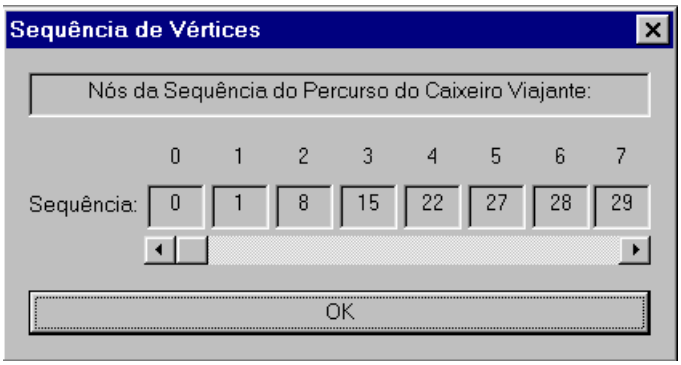


***Figura 22:* Seqüência de vértices relativa ao percurso do Caixeiro Viajante para uma modificação do grafo da figura 18 (veja a figura 23 a seguir)**

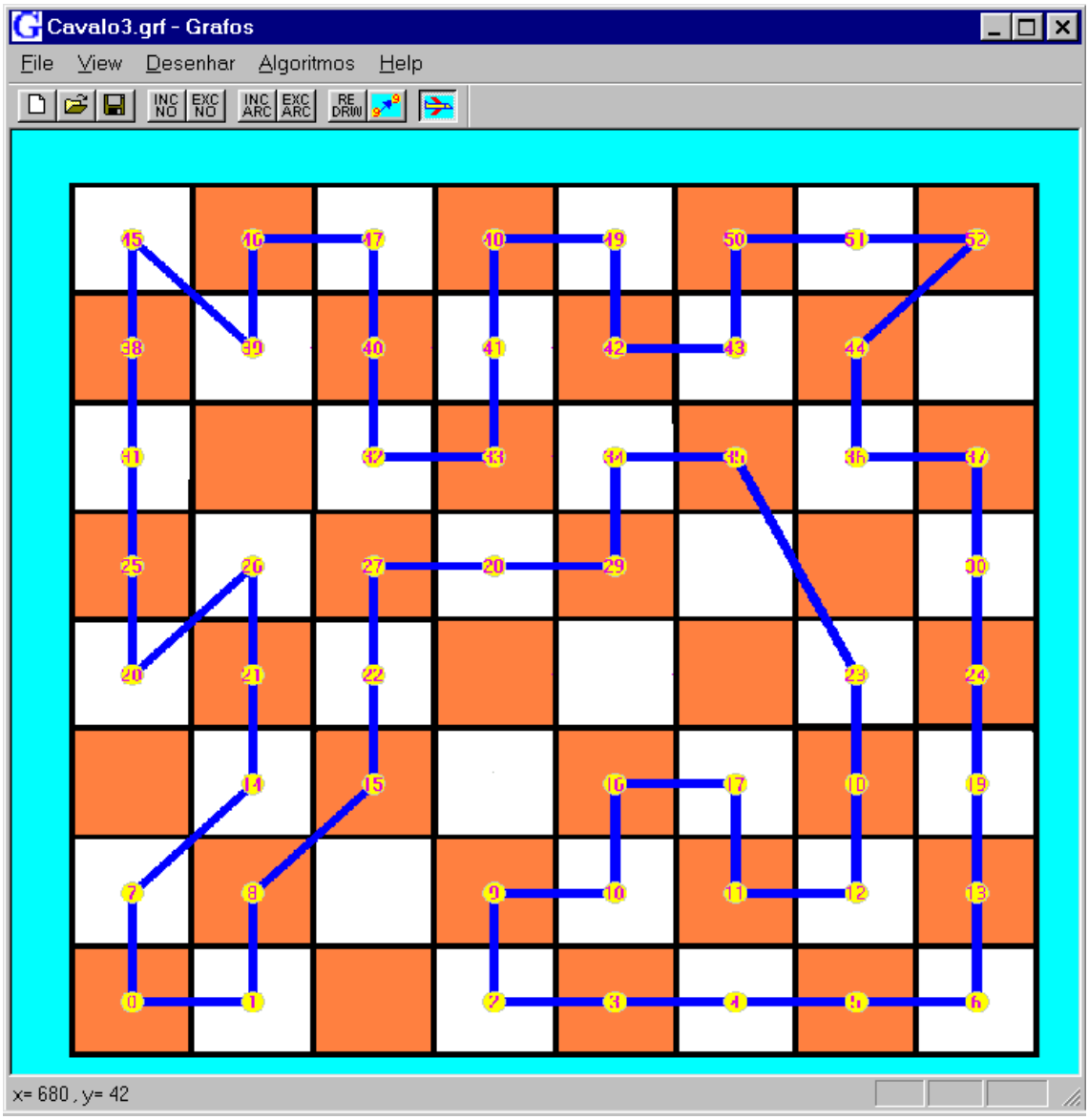


Figura 23: **Exemplo da execução do algoritmo do Caixeiro Viajante para o grafo da figura 18 modificado pela exclusão de alguns Nós.**

Esse exemplo ilustra bem como a heurística de Christofides para o Caixeiro Viajante pode fornecer bons resultados. Neste caso, um grafo com 53 nós, o percurso obtido não chega a ser 1% pior que o percurso ótimo. Para obtenção do percurso ótimo, precisaríamos de um esforço computacional intenso devido a explosão combinatorial ser de **52 ! / 2** possibilidades.

Note também que para a execução desse algoritmo, utilizamos a distância euclidiana entre as casas do tabuleiro, e consideramos também a total conectividade do grafo.

5.2.3. REPRESENTAÇÃO DAS 7 PONTES DE KÖNIGSBERG

O grafo da figura 24 é uma representação das 7 (sete) pontes de Königsberg, considerada historicamente como sendo um dos primeiros casos que levou a formulação do Problema do Carteiro Chinês.

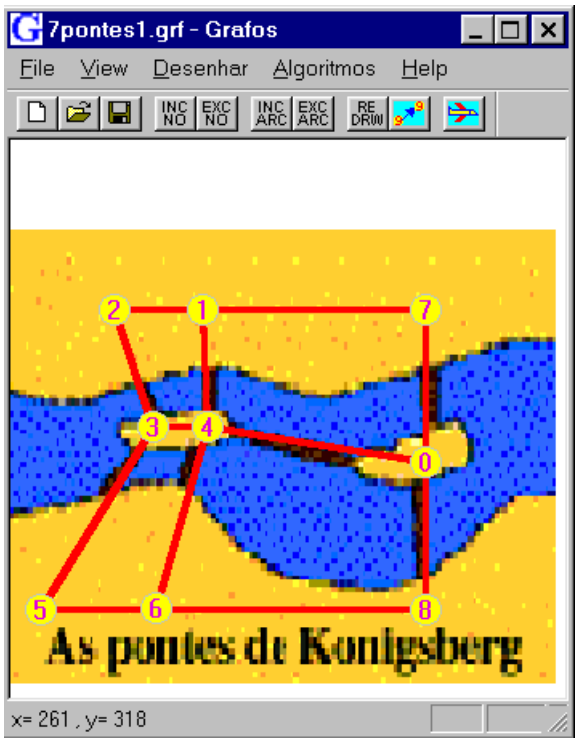


Figura 24: **Grafo que representa as 7 (sete) Pontes de Königsberg**

Para este grafo, atribuímos altíssimos custos aos arcos referentes às pontes sobre o rio Prevel, pois pretendemos executar o Algoritmo do Carteiro Chinês e verificar que não existe um circuito (ciclo) Euleriano que passe por todas as pontes apenas uma vez. Sendo os custos nas pontes muito elevado, o algoritmo tentará evitar ao máximo que elas sejam repetidas no percurso.

Observando mais uma vez a figura 24, clicando no Menu " ALGORITMOS " e depois em " CARTEIRO CHINÊS " o aplicativo perguntará pelo Nó inicial do percurso, suponhamos o nó 0 (ZERO).

Desta forma, obtemos o grafo da figura 25 abaixo, onde os arcos duplicados foram os arcos resultantes do algoritmo de MATCHING, parte integrante do problema do Carteiro Chinês.

Observe também, que apesar dos altos custos colocados para as pontes, os arcos (0,7) e (4,6) (que são pontes), tiveram que ser repetidos no ciclo euleriano.

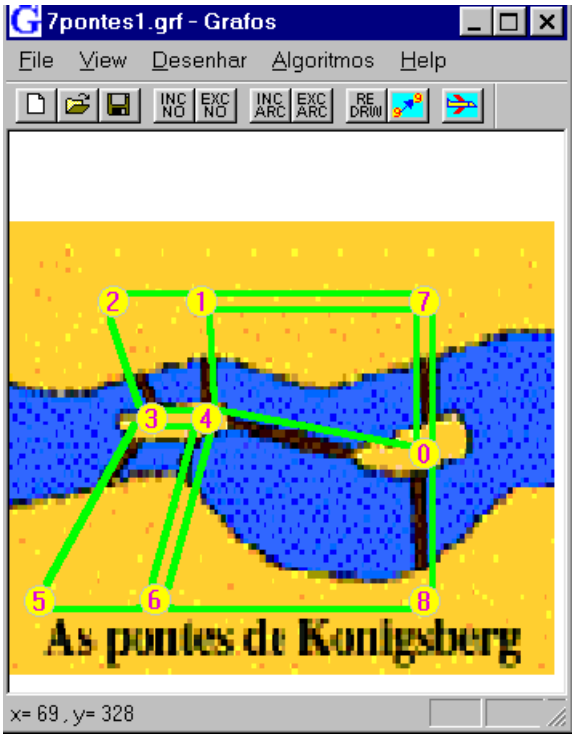


Figura 25: **Execução do Algoritmo do Carteiro Chinês para o Grafo da Figura 24**

Como já dissemos, não foi possível apresentar de maneira impressa a dinâmica deste algoritmo mostrando passo a passo o percurso do Carteiro Chinês. O percurso obtido foi: **0-4-1-2-3-4-3-5-6-4-6-8-0-7-1-7-0**. Muitos outros percursos podem existir, como por exemplo: **0-4-3-2-1-4-6-5-3-4-6-8-0-7-1-7-0**. O algoritmo de Obtenção do Ciclo Euleriano têm preferência pelos nós de menor índice, por isso no 1° percurso obtido, do nó **0** vai-se para o **4**, e do nó **4** vai-se para o **1**, ao invés do **3** como no 2° percurso citado anteriormente.

5.2.4. TRECHO DO MAPA DE WASHINGTON

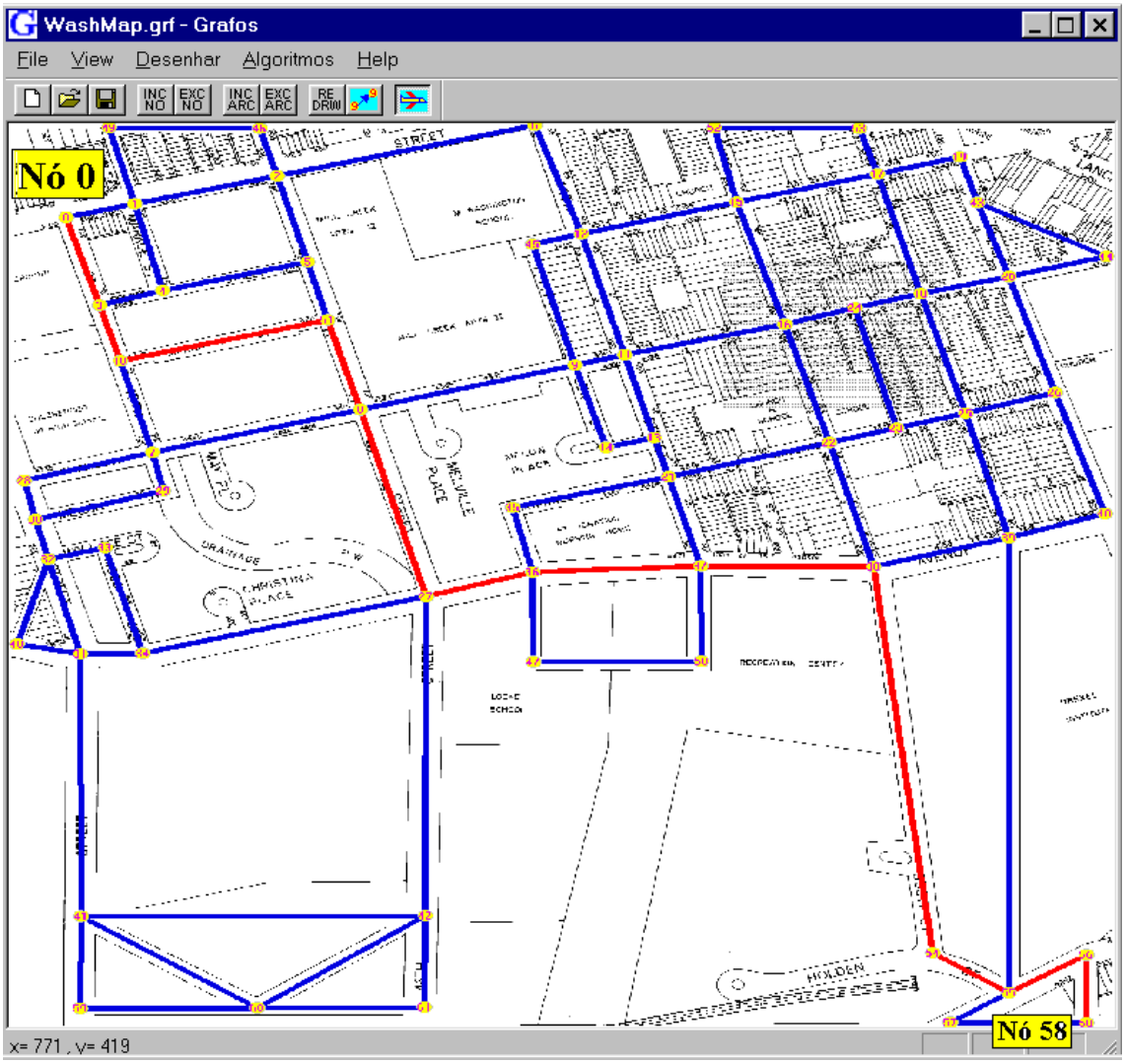
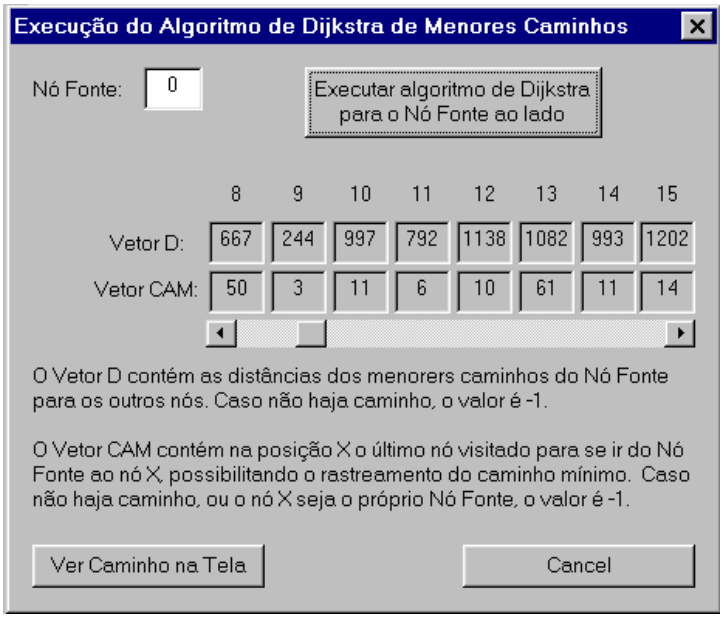
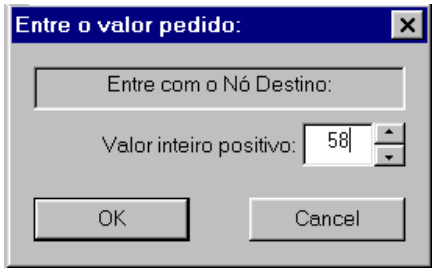


Figura 26: **Exemplo da Execução do Algoritmo de Dijkstra entre os Nós 0 e 58 sobre o trecho de um mapa da cidade de Washington.**

Na figura 26 é possível observar o resultado da execução do algoritmo de Dijkstra para o grafo sobre o trecho do Mapa de Washington, onde o nó 0 (ZERO) é o nó Fonte e o nó 58 o nó Destino. As figuras 27 (a) e (b) ilustram como foi obtido o percurso em vermelho da figura 26 acima, juntamente com a visualização dos vetores **D** e **Cam** (ver seção 4.1.1)



(a): **Vetores D e Cam obtidos da execução do algoritmo de Dijkstra para o nó fonte 0.**



(b): **Clicando em "VER CAMINHO NA TELA" em (a) é pedido o nó destino do percurso.**

Figura 27: **Caixas de diálogo da execução do algoritmo de Dijkstra para o grafo da figura 26.**

Como pudemos observar o algoritmo de Dijkstra fornece o menor caminho entre dois nós dados, o nó fonte e o nó destino. Pode-se precisar em alguns casos do segundo menor caminho ou de menores caminhos com algum tipo de restrição, tais como evitar passar por certos nós ou certos arcos.

Como exemplo, podemos citar o caso em que o menor caminho fornecido entre dois pontos da cidade passe por uma "*favela*" ou por uma rodovia de alta velocidade, como a via Dutra, por exemplo. O usuário talvez não deseje percorrer o trajeto fornecido pelo algoritmo.

Para resolver problemas desse tipo, desenvolvemos diversas rotinas que possibilitam a modificação dos parâmetros do grafo. A seguir ilustraremos a execução no aplicativo das opções de reposicionar e excluir vértices, pois são bons exemplos que possibilitam modificar as respostas do algoritmo de Dijkstra.

Na figura 28 a seguir, retiramos a opção de "VISTA AÉREA" e mudamos o nó 8 de posição. Para fazer isso basta ir no Menu do Software em "FILE", "EDIT" e "REPOSICIONAR NÓS". Clica-se com o mouse sobre um nó arrastando-o para a nova posição desejada.

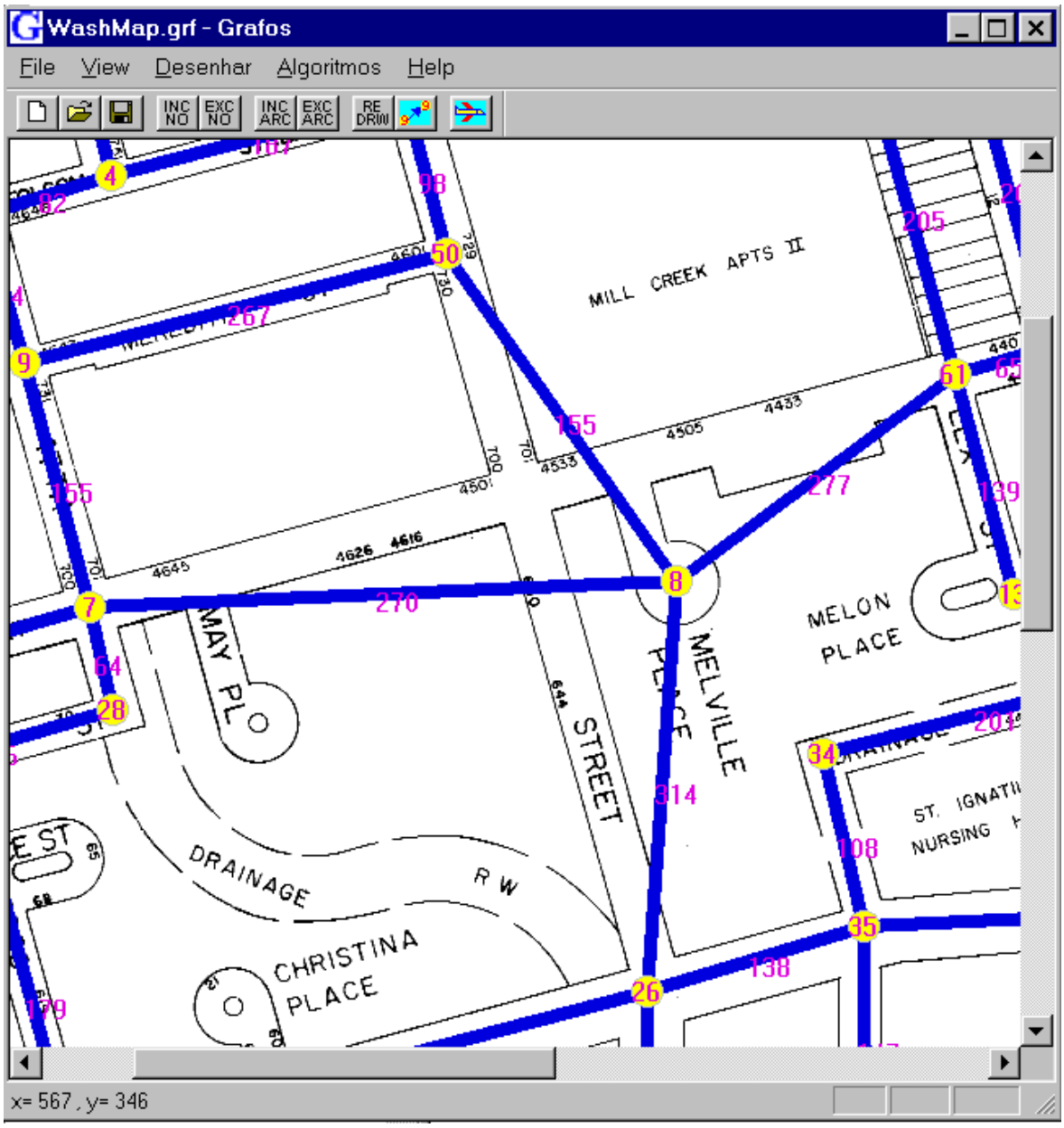


Figura 28: **Exemplo de reposicionamento de nós (no caso, o nó 8) do grafo sobre um trecho do mapa da cidade de Washington. Observe que esta ilustração não está em modo de "VISTA AÉREA".**

Suponhamos que estão ocorrendo OBRAS no nó 8 visto na figura anterior, tornando-o inacessível. Excluindo o nó 8 do grafo pela opção "EXC NO" na Barra de Ferramentas, e executando o algoritmo novamente, obtemos o caminho apresentado na figura 29 abaixo, como sendo a melhor alternativa do percurso entre os nós 0 (ZERO) e 58, quando não se pode passar pelo nó 8.

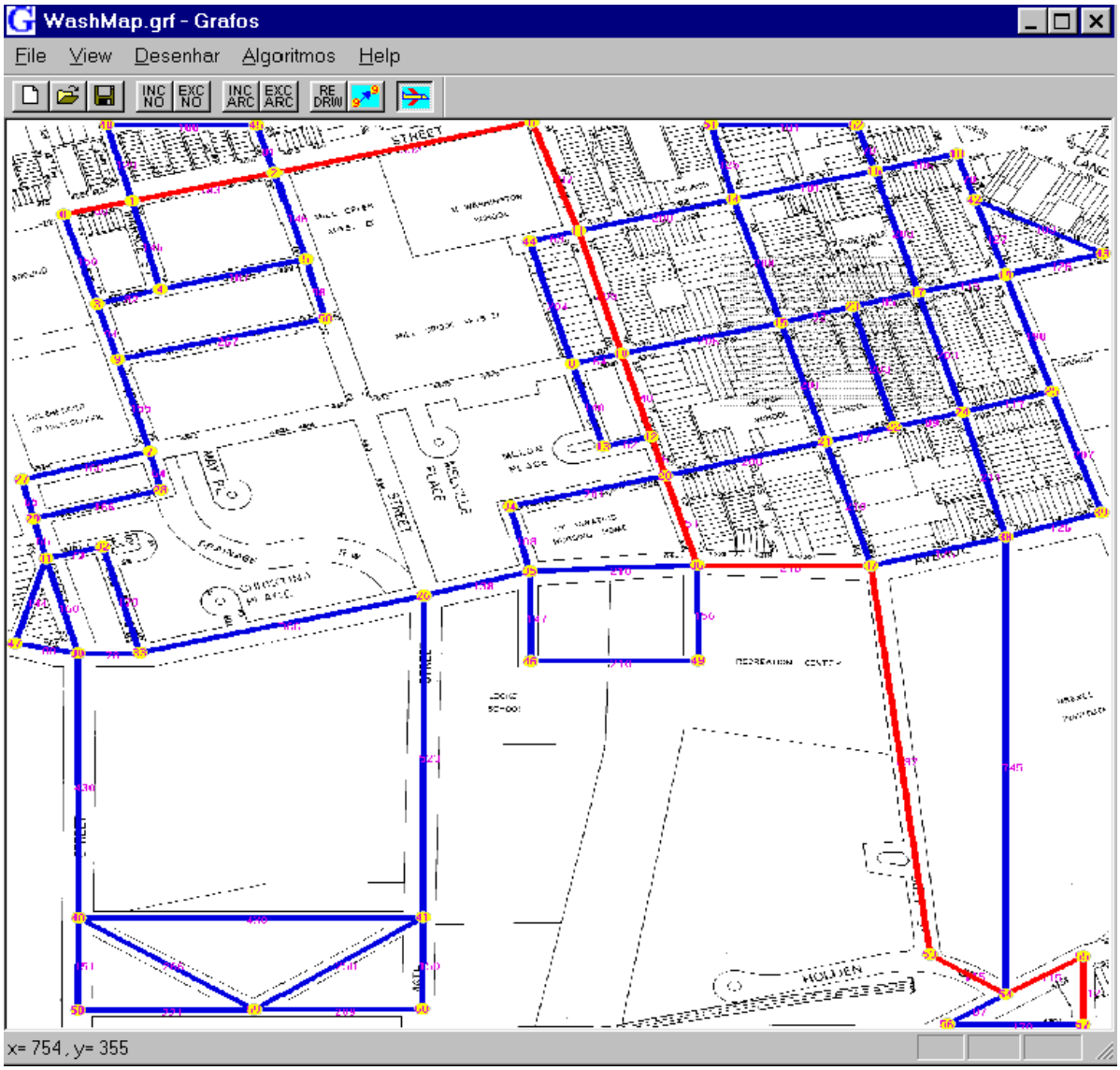


Figura 29: **Exemplo da execução do Algoritmo de Dijkstra entre os nós 0 e 58, para o grafo da figura 26 modificado pela exclusão do nó 8.**

Voltando ao grafo original da figura 26 e executando o Algoritmo de Obtenção da Minimum Spanning Tree (Árvore de Custo Mínimo), teremos a disposição ilustrada em vermelho na figura 30 como sendo a árvore do grafo que contém todos os nós e cuja soma dos arcos é mínima.

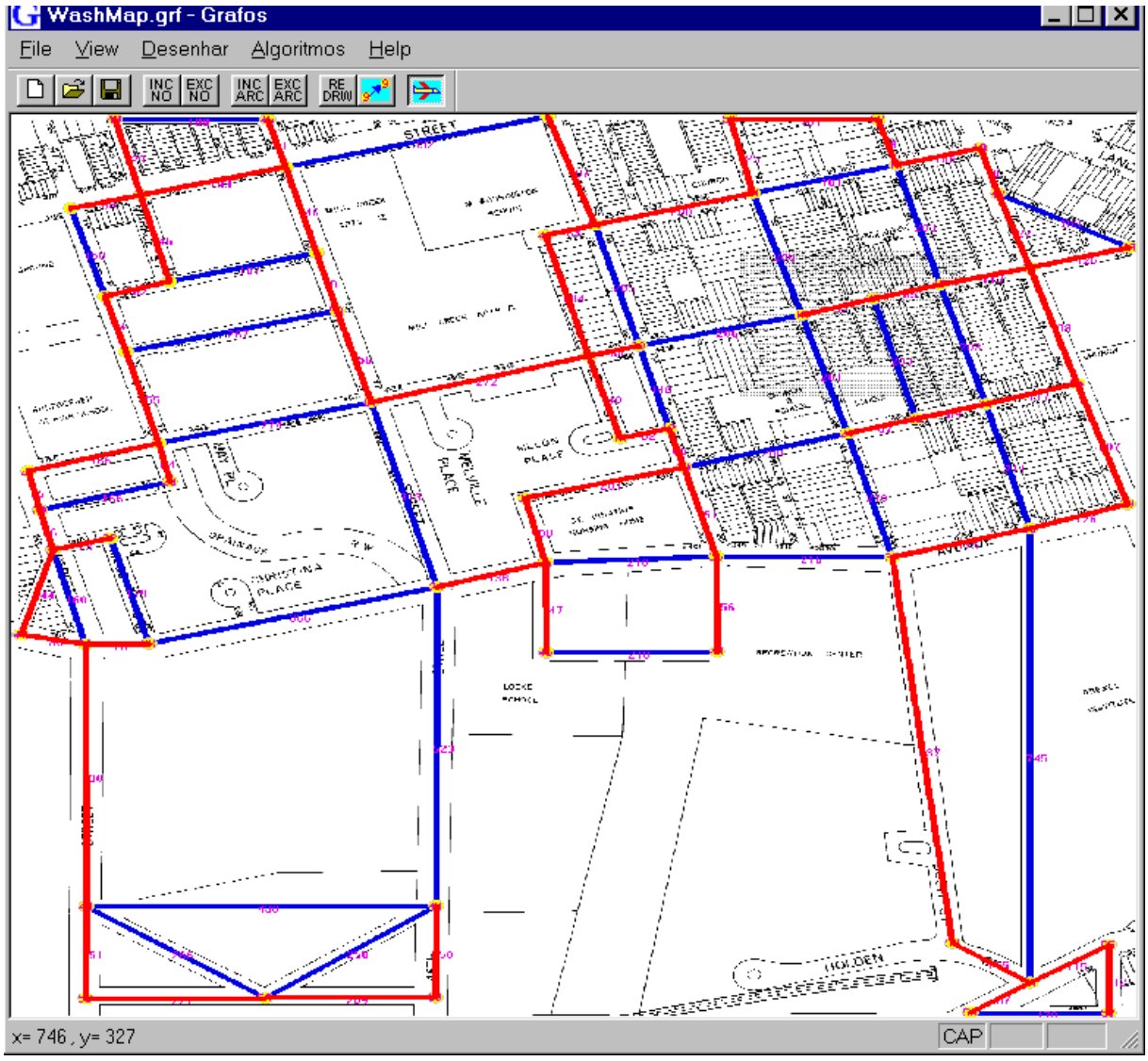


Figura 30: **Exemplo da Execução do Algoritmo de Obtenção da Minimum Spanning Tree (Árvore de Custo Mínimo) para o grafo da figura 26.**

5.2.5. FOTO AÉREA DE WASHINGTON

Mostraremos uma interessante aplicação ilustrada na figura 31: Suponhamos que um professor primário americano more no **Nó 3**, ministre aulas pela manhã em um colégio situado no **Nó 31** e à tarde em outro colégio situado no **Nó 27**. À noite, ele vai estudar na biblioteca de sua universidade situada no **Nó 23**, onde faz doutorado. Às 8:30 ele deve buscar o seu filho e sua filha que estudam respectivamente em colégios situados nos **Nós 39 e 10**; e às 10:00 ele deve buscar sua esposa que trabalha em um Shopping Center situado no **Nó 41**, retornando depois com toda a sua família para casa (**Nó 3**).

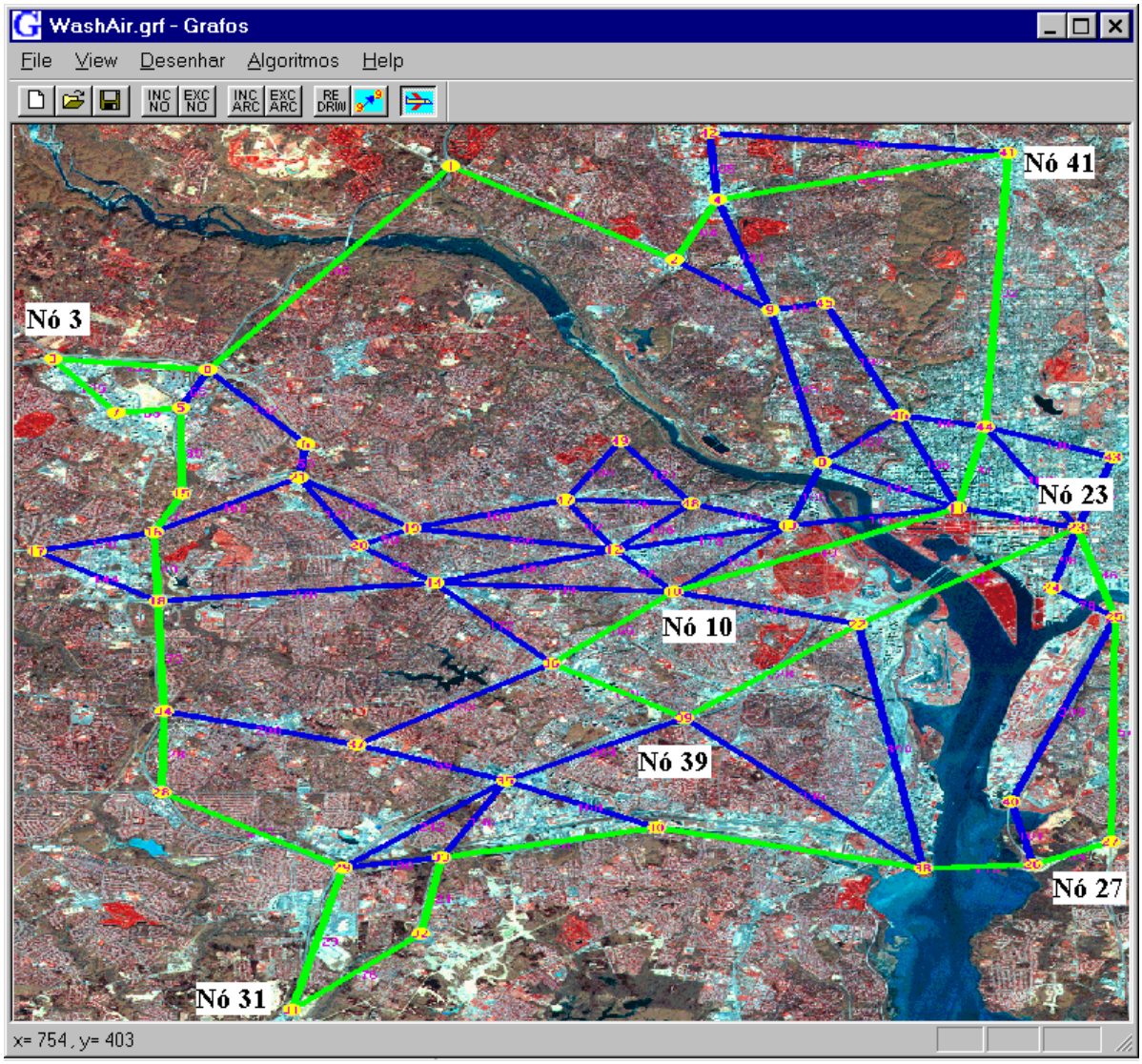


Figura 31: **Exemplo da Execução do Algoritmo de Floyd para o grafo obtido da Foto Aérea da cidade de Washington, para o percurso pelos Nós 3, 31, 27, 23, 39, 10, 41 e 3 novamente**

Na figura 31, é possível visualizar em verde o percurso do professor, que deve sair do nó 3, passar pelos nós 31, 27, 23, 39, 10 e 41 nessa ordem e depois retornar ao nó 3 inicial.

Pelo contexto mostrado, podemos ver que a presença nesses nós possui uma prioridade, os nós devem ser percorridos nessa ordem. Poderia haver o caso de uma empresa de refrigerantes que deve fazer entregas em diversos locais da cidade, de forma a minimizar o quanto ela anda, sem haver ordem de prioridade nos locais de entrega. Esta é outra importante aplicação onde o Algoritmo do Caixeiro Viajante desenvolvido pode ser aplicado.

Para esse mesmo grafo, poderemos executar também o algoritmo do Carteiro Chinês para um veículo do Correio (**Nó 0)** que passe pelos arcos do grafo da figura 31 entregando correspondências (ou fazendo a limpeza das ruas, para o caso do serviço de Limpeza Pública) e retornando de volta ao nó inicial.

Executando esse algoritmo, obteremos um percurso passando por **99 Nós** e repetindo os arcos **(2,4)**, **(4,41)**, **(5,15)**, **(8,13)**, **(8,46)**, **(14,20)**, **(15,16)**, **(16,18)**, **(18,34), (26,38)**, **(30,38)** e **(35,37)**, conforme ilustrado na figura 32 abaixo.

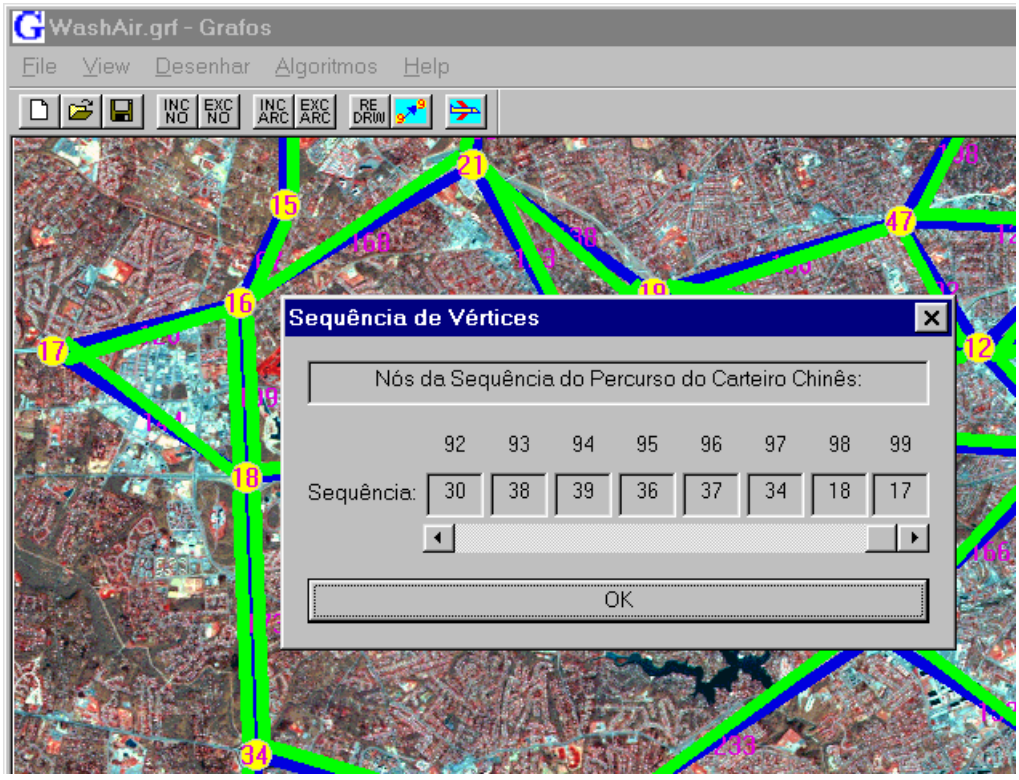


Figura 32: **Seqüência de 99 nós do percurso do Carteiro Chinês para o grafo da figura 31. Note na ilustração alguns dos arcos que foram percorridos duas vezes, como os arcos (16,18) e (18,34).**

Executando o Algoritmo de Obtenção da Minimum Spanning Tree para este grafo sobre a cidade de Washington, obtemos a Árvore de Custo Mínimo mostrada na figura 33 a seguir. Esta árvore, além de ter importância por fornecer subsídios à tomada de decisões onde situar ou construir Postos de Emergência, Corpo dos Bombeiros ou Delegacias de Polícia, também tem grande utilidade para outros algoritmos, como por exemplo, na Heurística de Christofides para o Caixeiro Viajante.

Existem métodos especialmente desenvolvidos para solucionar problemas de localização (citadas anteriormente) que poderiam ser implementadas posteriormente em outros trabalhos de graduação, dando continuidade a este trabalho. Com relação a isso, é sugerido no fim deste relatório algumas outras implementações para uma possível continuação deste projeto. Mas a árvore de custo mínimo já nos dá uma boa indicação de uma solução para este problema.

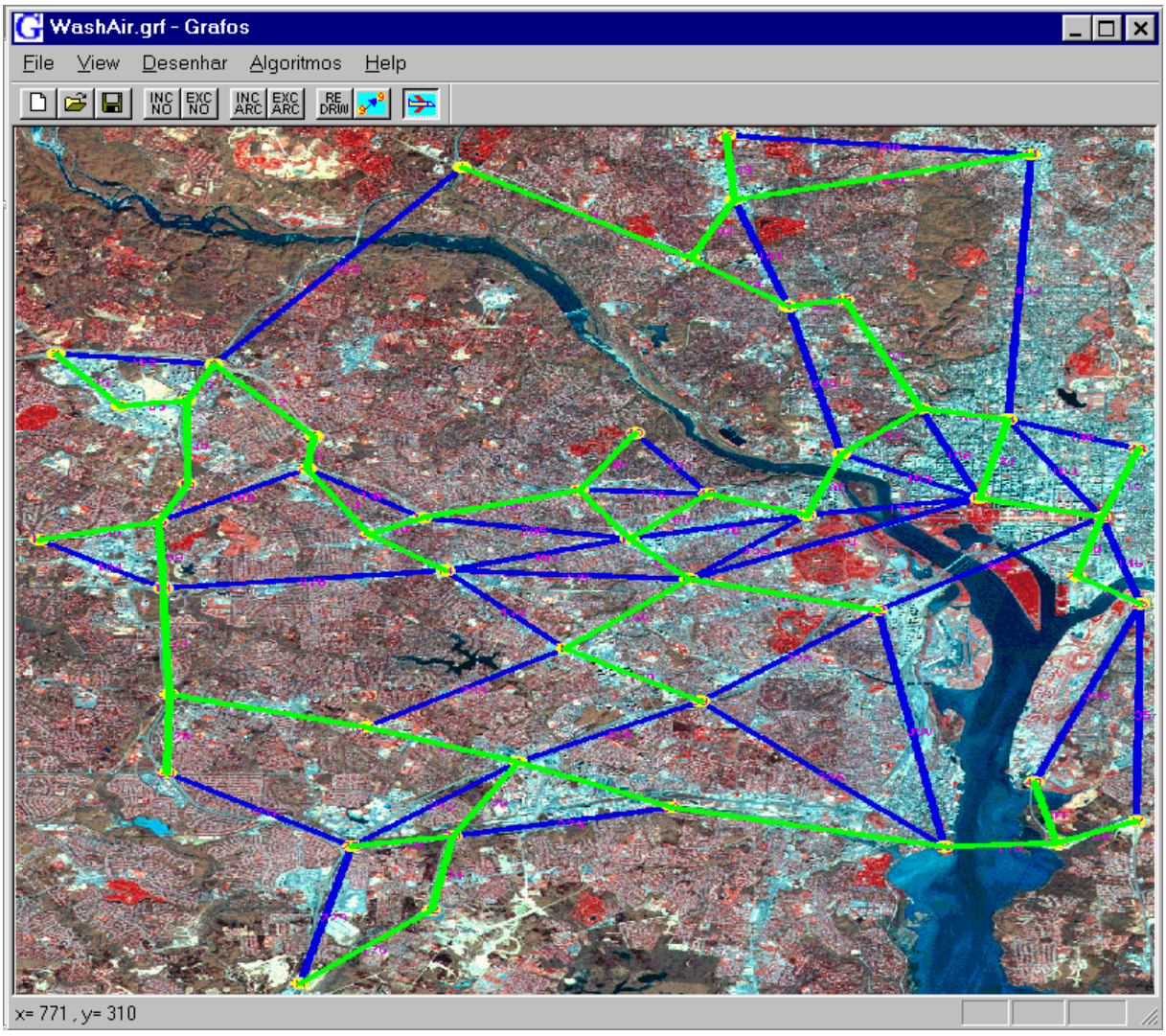


Figura 33: Exemplo da Execução do Algoritmo para Obtenção da Minimum Spanning Tree (Árvore de Custo Mínimo) sobre o grafo obtido da Foto Aérea da cidade de Washington.

5.2.6 TRECHO DO MAPA DE SÃO JOSÉ DOS CAMPOS

Nos exemplos anteriores, utilizamos exemplos hipotéticos que, por falta de dados reais, talvez não tenham reproduzido a realidade com a devida perfeição. Para estes exemplos, utilizamos grafos não orientados, ou seja, nenhum dos arcos dos exemplos anteriores possuíam direção ou orientação.

Em vários casos reais, as ruas de uma cidade possuem orientação: Existem ruas em mão única e outras em mão dupla. Os custos dos arcos em mão dupla não são necessariamente iguais para os dois sentidos. Talvez em uma rodovia, ocorra um congestionamento em um sentido, e no outro o fluxo esteja normal. Se, ao invés de distâncias, os custos nos arcos representarem o fluxo de uma rua em determinado sentido, então está claro que os custos serão diferentes para cada sentido.

Com o objetivo de mostrar um exemplo com dados mais reais sendo executado pelo aplicativo, digitalizamos um trecho do Mapa da Cidade de São José dos Campos (CENTRO) e construímos um grafo sobre ele, colocando os sentidos reais das ruas.

Este grafo é mostrado na figura 34, sendo possível observar alguns nós característicos, como o CTA (0), CURSO CASD (Centro Acadêmico Santos Dumont, 4), CENTER VALE SHOPPING (42), RODOVIÁRIA NOVA (55), INPE (66), BANHADO (17 ao 38), UNIVAP (20), VIA DUTRA INDO P/ SP (60), VIA DUTRA VINDO DE SP (76), VIA DUTRA INDO P/ RJ (75), VIA DUTRA VINDO DO RJ (65), entre outros.

Para este grafo executamos o Algoritmo de Floyd por exemplo para o seguinte contexto: Um turista vindo de carro do Rio de Janeiro (**Nó 65**) com destino a São Paulo (**Nó 60**) deseja conhecer a cidade de São José dos Campos no seu trajeto. Para isto, ele pretende passar na RODOVIÁRIA (**Nó 55**), onde pretende conseguir algumas informações e depois visitar um dos pontos turísticos da cidade, a ORLA DO BANHADO (que vai do **Nó 17** ao **Nó 38**), retornando posteriormente ao seu trajeto inicial.

Executamos o algoritmo para este caso, mas em duas épocas diferentes: a atual (novembro de 1998) (**figura 34**) e na época da construção do ANEL VIÁRIO (concluído em outubro de 1998) que obstruía o arco entre os Nós 31 e 32, fato ainda na lembrança dos motoristas da cidade, devido a volta pelo Nó 26 que deviam dar para chegar a VIA DUTRA (**figura 35**).

Os arcos em verde representam o percurso do turista. Devido ao fato de estar em modo de ZOOM a visualização em folha impressa está pior do que a que aparece na tela do microcomputador. Para ser possível visualizar a diferença entre esses percursos com mais detalhes, adicionou-se as figuras 36 e 37.

Figura 34: **Exemplo da Execução do Algoritmo de Floyd para o grafo obtido do Mapa de São José, com percurso 65-55-17-38-60, na época atual (11/98)**

Figura 35: **Idem para a Época da Construção do ANEL VIÁRIO (09/98), que obstruía o arco entre os Nós 31 e 32, obrigando a passar pelo Nó 26.**

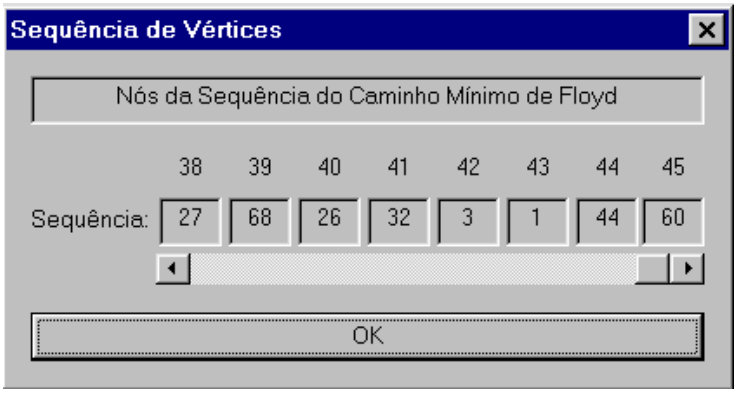


Figura 36: **Seqüência dos 45 nós do percurso obtido pelo algoritmo de Floyd na figura 35, considerando obstruído o arco (31,32).**

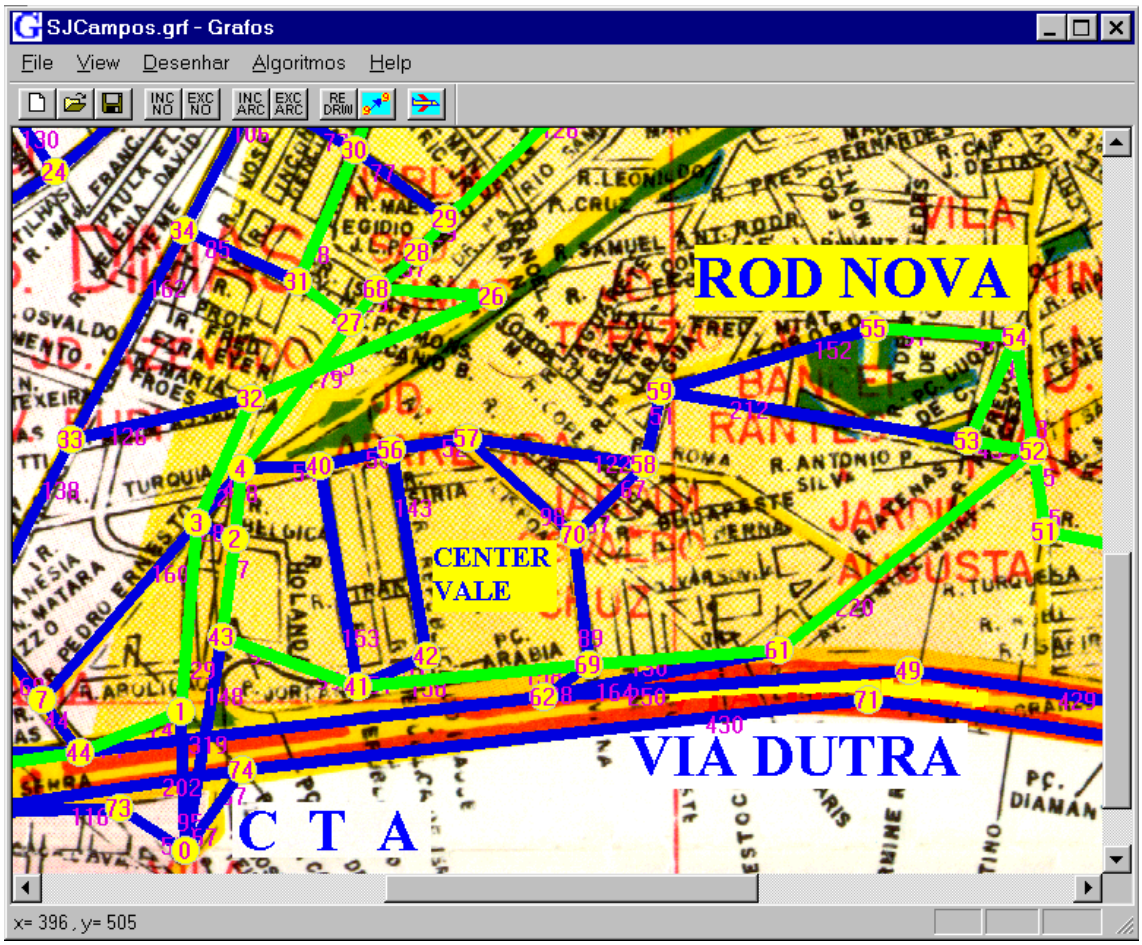


Figura 37: **Detalhes da modificação no percurso mínimo devido a eliminação do arco (31,32). Assim, é necessário fazer a volta 31-27-68-26-32 (observe mais detalhadamente a figura 36). Observe que esta ilustração não está em modo de "VISTA AÉREA".**

Em verde estão os arcos percorridos pelo turista e em azul os outros arcos do grafo. Como podemos observar, no 1° exemplo o arco entre os nós 31 e 32 é percorrido para se chegar à Via Dutra, mas no 2° exemplo é necessário se fazer uma volta passando pelos nós 31 – 27 – 68 – 26 – 32.

Sobre este grafo, vários outros contextos poderiam ser analisados:

- Um aluno de pós-graduação do INPE (nó 66) deseja chegar ao CTA (nó 0), por exemplo, em um horário que já esteja fechado o portão interno. Ele pode desejar dirigir ou não pela Via Dutra. Podemos executar o algoritmo de Floyd para o 1° caso e depois eliminar os arcos (47,48) , (61,62) e (69,62) que seriam as ligações com a Dutra para o 2° caso.

- Um hóspede do Modena Hotel ou um aluno do colégio Anglo (nó 28), deseja ir para São Paulo (nó 60) na época atual ou na época da construção do Anel Viário (basta eliminar o arco (31,32)).

- Um ônibus deseja ir da Rodoviária Velha (nó 17) à Rodoviária Nova (nó 55), passando antes na UNIVAP (nó 20) e na UNESP (nó 30) para buscar alguns alunos.

- Um aluno do ITA deve dar aulas no Curso CASD (nó 4) e na volta para o CTA (nó 0) deseja jantar no Center Vale Shopping (nó 42).

6. MANUAL DO USUÁRIO

6.1. INTRODUÇÃO

Para o uso desse Aplicativo, padronizou-se 3 (três) tipos (ou estilos) de arquivos: TIPO1, TIPO2 e TIPO3, com terminação **GRF**.

Em geral, cada arquivo GRF contém na 1° linha o nome do arquivo Bitmap de fundo, e na 2° linha o tipo do arquivo. A partir da 4° linha, cada linha indicará as coordenadas X e Y dos nós, bem como os nós aos quais ele se liga seguido do custo do respectivo arco, na forma de Lista de Adjacências.

No caso do TIPO1, não é necessário explicitar as coordenadas X e Y dos nós. Eles são dispostos ao redor de uma circunferência que maximize a tela, igualmente espaçados entre si. No caso do TIPO2, nada pode ser omitido. No caso do TIPO3, não é necessário explicitar os custos dos arcos que saem do nó da linha do arquivo em questão. Esses valores são colocados como sendo a distância euclidiana entre os nós, sendo necessário explicitar apenas os nós aos quais ele se liga (sem indicar o respectivo custo do arco). Para grafos desse tipo, os custos dos arcos são imutáveis. Algumas facilidades explicadas posteriormente foram introduzidas para capacitar a mudança de um tipo de grafo em outro

Outra facilidade do software é a introdução de escalas tanto na vertical quanto na horizontal. Essa facilidade é interessante para o caso de grafos postos sobre Mapas com determinadas Escalas. Assim é possível representar os comprimentos reais dos arcos para tais grafos.

Várias outras opções foram colocadas no Software de modo a melhorar a execução e a visualização do Aplicativo, como possibilidade da mudança das cores, das larguras das linhas, da fonte do texto, do tempo de atraso em caso de visualização passo a passo de algoritmos sendo executados, e assim por diante. Tais opções serão mais bem detalhadas a seguir.

6.2. OPERAÇÕES COM ARQUIVOS

SAVE AS TIPO 2

Grava um arquivo GRAFO de qualquer tipo, modificando-o para TIPO2. Introduz, se necessário, as coordenadas X e Y do nó (caso seja do TIPO1), bem como os custos dos arcos que saem de cada nó (caso seja do TIPO3).

SAVE WITH ESCALE

Grava um arquivo GRAFO considerando as coordenadas atuais X e Y na tela para uma dada escala. Caso a escala esteja em 2, por exemplo, as coordenadas dos nós estarão duas vezes maior do que o normal. Estas coordenadas são gravadas no arquivo e a escala é modificada para 1 (um) tanto na vertical quanto na horizontal.

6.3. OPÇÕES GERAIS DE EXECUÇÃO

VISUALIZAR GRAFO: Possibilita Visualizar os Nós do Grafo na Tela.

VER POSIÇÃO DO MOUSE: Possibilita Visualizar as Coordenadas X e Y do Mouse na Barra de Status.

ALGORITMO COM ATRASO: Possibilita Executar os Algoritmos de Roteamento sobre Grafos em Modo Passo a Passo, indicando um Valor em Milissegundos para o Tempo de Atraso (Delay) na visualização.

EDIT EM MODO GRÁFICO: Possibilita Utilizar as Funções de Manipulação de Grafos (INCLUIR NÓ, INCLUIR ARCO, ...) utilizando o mouse. No caso de INCLUIR ARCO, por exemplo, possibilita clicar sobre um nó e arrastar o mouse até outro nó, caracterizando um nó direcionado.

INC E EXC ARC NOS 2 SENTIDOS: No exemplo citado no parágrafo anterior, possibilita realizar a inclusão e exclusão de arcos de um grafo nos 2 (dois) sentidos, não se importando com a orientação dada manualmente.

REDESENHAR GRAFO SEMPRE AO EXECUTAR UM ALGORITMO GRÁFICO: Possibilita redesenhar o grafo sempre ao executar um algoritmo. É uma importante opção, pois mudanças feitas no desenho devido a execuções anteriores de alguns algoritmos serão apagadas, possibilitando a execução do algoritmo que se deseja sobre o desenho original do grafo.

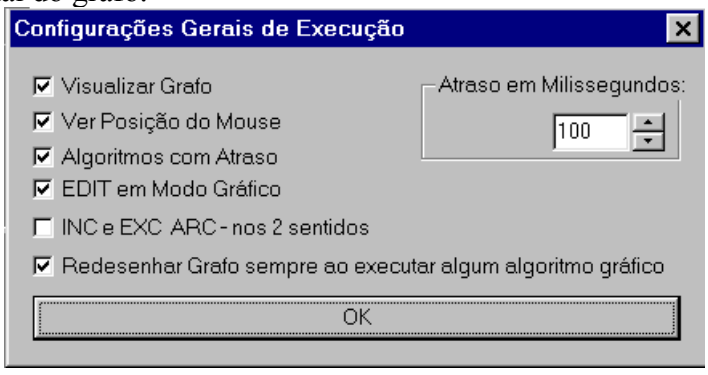


Figura 38: **Caixa de diálogo que permite alterar configurações gerais de execução do Software.**

6.4. OPÇÕES GERAIS DE VISUALIZAÇÃO

Possibilita escolher se deseja ou não visualizar os elementos abaixo:

- **VER ARCOS**
- **VER VALORES DOS ARCOS**
- **VER SETAS NOS ARCOS**
- **VER CÍRCULOS DOS NÓS**

Possibilita escolher se deseja que a escala dada para X ou Y, esteja sendo multiplicada ou dividida, ou seja, 2 pode significar 2 ou ½ (aumentando 2 vezes ou diminuindo 2 vezes)

- **ESCALA X DIVIDINDO, NÃO MULTIPLICANDO**
- **ESCALA Y DIVIDINDO, NÃO MULTIPLICANDO**

Possibilita modificar alguns atributos dos elementos a seguir:

- **ARCOS** **COR E LARGURA DO TRAÇO**
- **ESCALA EM X** **VALOR**
- **ESCALA EM Y** **VALOR**
- **NÓS (VÉRTICES)** **COR E LARGURA DO TRAÇO**
- **PREENCHIMENTO** **COR**
- **RAIO DO CÍRCULO** **COMPRIMENTO**
- **TEXTO** **COR**
- **TRAÇO DE EXECUÇÃO** **COR E LARGURA DO TRAÇO**

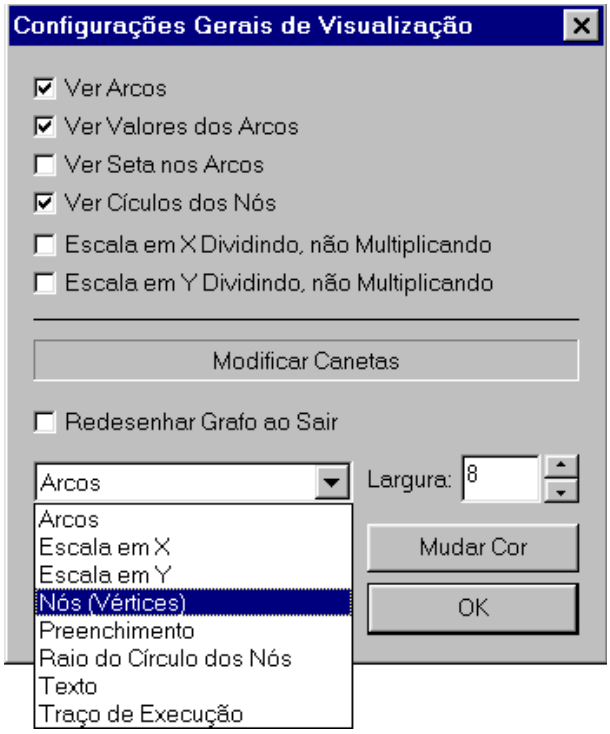


Figura 39: **Caixa de diálogo que permite alterar configurações gerais de visualização do Software.**

6.5. OPÇÕES GERAIS DE MANIPULAÇÃO

Suponhamos que está "ATIVADA" a opção geral de execução "EDIT EM MODO GRÁFICO". Assim, temos as seguintes opções de manipulação:

INCLUIR NÓ: Possibilita utilizar o Mouse e escolher a posição de um novo nó sobre um grafo na tela.

EXCLUIR NÓ: Possibilita utilizar o Mouse e escolher um nó a ser eliminado do grafo.

INCLUIR ARCO: Possibilita utilizar o Mouse, escolhendo dois nós com o objetivo de adicionar ou modificar o custo do arco entre eles.

EXCLUIR ARCO: Possibilita utilizar o Mouse, escolhendo dois nós com o objetivo de eliminar o arco entre eles. Uma mensagem de Erro é gerada se não existir um arco entre os nós escolhidos.

REPOSICIONAR NÓS: Possibilita utilizar o Mouse, escolhendo um nó a ser reposicionado, arrastando-o para a sua nova posição.

TRANSLADAR GRAFO: Possibilita utilizar o Mouse, escolhendo um nó e reposicionando-o. Todo o Grafo será transladado mantendo-se as proporções e os seus comprimentos de forma que o nó escolhido vá para a posição especificada.

MUDAR LABEL DE UM NÓ: Possibilita utilizar o Mouse, escolhendo um nó cujo Label (nome, rótulo) será modificado. Clicando sobre ele, uma caixa de diálogo perguntará pelo novo label do nó escolhido.

Figura 40: **Menu que permite realizar manipulações em grafos.**

Caso a opção "EDIT EM MODO GRÁFICO" não esteja ativada, tais manipulações serão feitas em caixas de diálogo.

6.6 ALGORITMOS

O Software possibilita a execução dos algoritmos estudados até então sobre os grafos carregados a partir dos arquivos. Cada algoritmo possui uma saída diferente, de forma que os seus resultados possam ser visualizados de uma forma adequada.

CONECTIVIDADE: Sua execução é realizada ao ver as características do grafo, no MENU VIEW, informando se o grafo é fortemente conexo, simplesmente conexo ou desconexo.

MENOR CAMINHO DE DIJKSTRA: Ao ser executado, o software pergunta pelo NÓ FONTE e espera pela acionamento do botão para dar início ao algoritmo, mostrando como saída os vetores dos comprimentos e dos percursos dos menores caminhos. Se se desejar visualizar algum menor caminho na tela, o algoritmo perguntará pelo NÓ DESTINO, mostrando, se existir, o menor caminho entre esses nós na tela.

MENOR CAMINHO DE FLOYD: Ao ser executado, o software mostra em uma janela as matrizes dos comprimentos e dos percursos dos menores caminhos. É permitido ao usuário entrar com uma lista de nós dados em seqüência de prioridade (separados por espaço) para que o algoritmo mostre a mínima rota a ser realizada para percorrê-los nessa ordem.

MINIMUM SPANNING TREE: Ao ser executado, o software já realiza o desenho da ÁRVORE DE CUSTO MÍNIMO na tela, sobre o desenho do grafo.

CARTEIRO CHINÊS: Ao ser executado, o software pergunta pelo NÓ INICIAL, de onde será iniciado e finalizado o ciclo euleriano, perfazendo depois o desenho na tela do percurso do Carteiro Chinês. No final da execução, é mostrado em uma caixa de diálogo a lista dos nós que fazem parte deste percurso.

CAIXEIRO VIAJANTE: Ao ser executado, o software já realiza o desenho do PERCURSO DO CAIXEIRO VIAJANTE na tela, sobre o desenho do grafo.

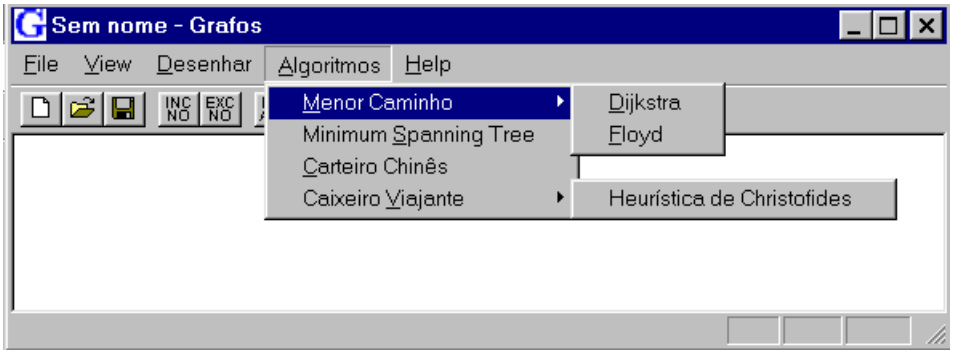


Figura 41: **Menu que permite executar os algoritmos de roteamento implementados.**

7. MANUAL DO PROGRAMADOR

- **ESTRUTURA DE DADOS**
- **ALOCAÇÃO DINÂMICA DE MEMÓRIA**
- **CLASSES DA DLL GRAFALGORIT**
- **INTERFACE DA DLL COM O SOFTWARE GRAFOS**

Para ser possível implementar os algoritmos citados, devemos obter formas para armazenar os dados referentes aos grafos, aos nós e aos arcos. No estudo de grafos, conhecemos duas formas principais de armazenar seus dados: Matriz de adjacências e Lista de adjacências.

A matriz de adjacências M consiste de uma matriz **n**x**n**, onde **n** é o número de nós do grafo, cujos elementos M[i,j] são os comprimentos dos arcos que vão do nó i ao nó j. A lista de adjacências L consiste de um vetor **n**-dimensional de ponteiros, cujos elementos L[j] apontam para células que indicam os nós que se ligam ao nó j. Como exemplo de matriz de adjacências temos a tabela 2 na página 33, e como exemplo de lista de adjacências vemos a seguir uma lista para o grafo da figura 1, da página 12.

Nó 1	3 arcos	→	2	10	→	4	30	→	5	100
Nó 2	1 arco	→	3	50						
Nó 3	1 arco	→	5	10						
Nó 4	2 arcos	→	3	20	→	5	20			
Nó 5	0 arcos									

Podemos observar portanto que a matriz de adjacências possui a grande vantagem de acessar facilmente as distâncias entre dois nós, e com isso diminuir o tempo de execução. Mas possui também a grande desvantagem de ocupar muito espaço na memória, **nxn,** para **n** nós.

Ao contrário da matriz de adjacências, a lista de adjacências possui a vantagem de ocupar relativamente pouca memória, pois cita apenas os arcos que saem de cada nó, mas possui a grande desvantagem de ser complicado o acesso das distâncias entre nós, acarretando num maior tempo de processamento por ser uma tarefa bastante freqüente.

Implementou-se essas duas estruturas de dados no software: Os arquivos armazenam os dados do grafo em formato de Lista de Adjacências, enquanto o programa armazena os dados do grafo em formato de Matriz de Adjacências (representando a matriz das distâncias entre dois nós **i** e **j** denominada por **C[i,j]**), visto que o limitante mais restritivo nesse trabalho é o tempo de processamento, e não a memória física.

Como os arquivos são escritos em formato de Lista de Adjacências, com o objetivo de ocupar pouca memória em disco, e o programa utiliza Matriz de Adjacências para suas execuções internas, com o objetivo de reduzir o tempo de processamento, é necessário fazer transformação de um formato em outro. Isto é feito nas funções REDRAW (transforma de Lista em Matriz) e SAVE AS TIPO 2 (transforma de Matriz em Lista).

Outra questão interessante a ser colocada é que, como o usuário pode incrementar o número de nós a vontade, a alocação de memória não pode ser estática. Esse problema foi solucionado no desenvolvimento da **DLL GrafAlgorit** com as classes *CMatrizDbl*, *CMatrizInt*, *CVetorDbl*, *CVetorInt* e *CPar*, que implementam matrizes e vetores de números inteiros e em ponto flutuante, com alocação dinâmica de memória. (*CPar* é uma classe que implementa matrizes com duas colunas, com o objetivo de representar arcos).

Desta forma, o número de nós do grafo pode ser qualquer, desde que não estoure a memória do computador. Vários métodos dessas classes possuem a mesma interface e foram implementados com o objetivo de alocar, desalocar ou de redimensionar a quantidade de memória utilizada por dado objeto dessas classes, como *DISPOSE*, *SETSIZE1* e *SETSIZE2*, *DELETE LINE*, *INSERT LINE*, *DELETE CELL* e *INSERT CELL*.

DISPOSE desaloca a memória destinada ao objeto. *SETSIZE1* redimensiona a quantidade de memória destinada a um objeto, colocando valores nulos nas células da matriz ou vetor. *SETSIZE2* possui a mesma função mas considera os valores que estão atualmente nas células, truncando-os ou pondo ZERO nas células a mais, caso a nova quantidade de memória seja menor ou maior que a atual. *DELETE LINE* e *INSERT LINE* são funções das classes *CMatrizInt* e *CMatrizDbl*, enquanto *DELETE CELL* e *INSERT CELL* são métodos das classes *CVetorInt* e *CVetorDbl*. São utilizados por exemplo ao incluir ou excluir nós.

Outra implementação necessária foi da Classe *CSet* destinada a representar Conjuntos, visto que várias implementações de algoritmos de roteamento se utilizam dos recursos de conjuntos, como interseção, união, diferença, pertence, não pertence, contém, está contido, e etc...Para esta classe também foi utilizada alocação dinâmica de memória, embora em menor escala que nas classes anteriores, visto que nessa aplicação não é necessária a utilização de conjuntos muito grandes, e ser possível o armazenamento de conjuntos de uma forma mais concisa do que em vetores e matrizes.

A implementação dessa classe é bastante interessante: Como em C++, cada número inteiro short é representado por 16 bits, então cada número inteiro pode ser considerado como sendo uma célula armazenadora de 16 números (cada bit indica a presença ou não de dado elemento). Pensando dessa forma, implementamos a classe *CSet* como sendo um vetor de inicialmente 20 números inteiros (podendo ser aumentado ou diminuído), sendo capaz de armazenar 20 * 16 (320) números inicialmente, ou seja, na 1° célula (de 0 a 15), na 2° célula (de 16 a 31), na 3°(de 32 a 47), até a 20° célula (de 304 a 319). Com algumas operações bit a bit fornecidas pela linguagem de programação, foi possível implementar as funções mais importantes de Conjuntos, tais como União e Interseção.

A principal classe da **DLL GrafAlgorit** é a classe *CGrafo* que implementa grafos, suas estruturas internas e os algoritmos de roteamento estudados durante este trabalho. Em cada objeto da classe *CGraf*o, existem armazenados o n° de nós do grafo (membro N inteiro), o n° de arcos (membro NumTotArcs inteiro), a Matriz de Adjacências (membro C, objeto da classe CMatrizDbl), o Conjunto dos nós de grau ímpar (o membro Impares, objeto da classe CSet) e 4 (quatro) variáveis que indicam se o grafo é orientado, se há ciclo ou caminho euleriano, e o tipo de conectividade do grafo (os membros mbOrientad, mbEulerTour e mbEulerPath, booleanos, e o membro miConexo inteiro).

Nesta classe existem também os métodos que implementam os algoritmos de roteamento, a saber:

- ***short Conexo();***
  ENTRADA: nenhuma
  SAÍDA: nenhuma
  RESPOSTA: Fortemente Conexo (2), Simplesmente Conexo (1) ou Desconexo (0).
- ***void Dijkstra(int NoF, CVetorDbl& D, CVetorInt& Cam);***
  ENTRADA: NoF (NÓ FONTE)
  SAÍDA: D (Vetor dos Custos (reais)) e Cam (Vetor dos Percursos (inteiros))
  RESPOSTA: nenhuma
- ***void Floyd(CMatrizDbl& D, CMatrizInt& Cam);***
  ENTRADA: nenhuma
  SAÍDA: D (Matriz dos Custos (reais)) e Cam (Matriz dos Percursos (inteiros))
  RESPOSTA: nenhuma
- ***BOOL Prim(CPar& Arvore, int& TotArcs);***
  ENTRADA: nenhuma
  SAÍDA: Arvore (Vetor dos Arcos da Minimum Spanning Tree)
  TotArcs (Número de Arcos da Minimum Spanning Tree)
  RESPOSTA: Árvore Conexa ou Desconexa
- ***void ChinesePostman(int NoInit, CVetorInt& PERC, int& TotArcs);***
  ENTRADA: NoInit (NÓ INICIAL DO CICLO EULERIANO)
  SAÍDA: PERC (Vetor do Percurso do Carteiro Chinês)
  TotArcs (Número de Arcos do Percurso do Carteiro Chinês)
  RESPOSTA: nenhuma
- ***void TravelSalesman(CVetorInt& Perc, int& TotArcs, CVetorInt& x, CVetorInt& y);***
  ENTRADA: Vetores X e Y (das coordenadas dos Nós do Grafo na Tela)
  SAÍDA: Perc (Vetor do Percurso do Caixeiro Viajante)
  TotArcs (Número de Arcos do Percurso do Caixeiro Viajante)
  RESPOSTA: nenhuma

8. TESTES

Durante a realização desse trabalho, além do estudo dos problemas e da implementação dos algoritmos, muitos testes foram realizados.

A grande maioria desses testes foram para simples verificação do funcionamento correto do algoritmo implementado. Os erros mais difíceis de se achar, foram encontrados depois de muitos testes com vários exemplos. Tais exemplos eram criados procurando fazer com que os mínimos detalhes de cada problema fossem postos à prova.

Com a detecção de um erro, procurava-se a falha no algoritmo, que poderiam ser simplesmente de programação, que são resolvidos com grande facilidade, ou grandes problemas de lógica. Em falhas desse tipo, procura-se uma nova abordagem para a resolução do problema combinatorial de roteamento.

Também com a ajuda de testes, verificou-se realmente se algumas melhorias introduzidas surtiam efeitos positivos. Em testes desse tipo, várias tentativas de melhorias foram ignoradas por não produzirem resultados satisfatórios. Tais testes também foram bastante importantes para quantificar o efeito dessas melhorias, como por exemplo no caso do "Matching Problem", observado na tabela 1.

Grande parte do período destinado ao trabalho desenvolvido foi gasto com a execução de testes de funcionamento, testes de verificação e testes de desempenho para a obtenção de resultados mais confiáveis e seguros, visando não se perder trabalho nem tempo pela correção de erros cometidos por falhas não detectadas.

9. CONCLUSÕES, COMENTÁRIOS E RECOMENDAÇÕES

Este trabalho de graduação, juntamente com o projeto em andamento no LAC (INPE) de desenvolvimento e implementação de algoritmos em redes aplicadas em Sistemas Geográficos de Informação, permitiu realizar uma coletânea de vários algoritmos de roteamento, que implementados serão utilizados para aplicação em casos reais com dados obtidos através de Imagem de Satélites, GPS e Mapas, como ilustrado nos exemplos, "linkados" ao software ARC-INFO do Environmental Systems Research Institute (ESRI) em uso no projeto.

Para futuros trabalhos, podemos citar vários outros itens interessantes e de fundamental importância a serem estudados e implementados, como por exemplo:

- Menor Caminho com Arcos de Custo Negativo
- Carteiro Chinês em Grafos Orientados
- Rural Postman Problem
- Outras Heurísticas para o Caixeiro Viajante (em diferentes contextos)
- Multiroute Node Covering (2 ou mais Caixeiros Viajantes em um Grafo)
- Carteiro Chinês introduzindo Capacidade da Mochila (Caminhão de Lixo, p.e.)
- Problemas de Localização de Facilidades (p.e, problema das **p-medianas**)

- Implementação de Bancos de Dados para os Grafos, para possibilitar, em um grafo cujos arcos representem os fluxos ou tempos de percurso em uma cidade por exemplo, a modificação dos custos dos arcos no decorrer das horas do dia, visto que no mesmo arco existem horas de congestionamento e horas de fluxo baixo.
- Implementação de classes que realizem visualização JPG ou outras mais compactas ainda, para reduzir o espaço em disco de armazenamento de Mapas ou Fotos de Satélite.

10. BIBLIOGRAFIA

•**Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.**, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

•**Boaventura Netto, P.O.**, *Grafos: Teoria, Modelos e Algoritmos*, Edgard Blücher, 1996.

•**Larson, R.C.; Odoni, A.R.**, *Urban Operations Research*, Prentice Hall, 1981.

•**Campello, R.E.; Maculan, N.**, *Algoritmos e heurísticas*, EDUFF, 1994.

•**Mokarzel, F.C.**, *Apostila do Curso de Estrutura de Dados (CES-20)*, Computação, ITA, 1990.

•**Pappas, C.H.; Murray, W.H.,** *Turbo C++ Completo e Total*, Editora McGraw-Hill Ltda & MAKRON Books do Brasil Editora Ltda., 1991.

•**Kruglinski, D.**J.**; Shepherd, G.,** *Programming Microsoft Visual C++*, 5° Edição, Microsoft Press, 1998.

•**Blaszczak, M.,** *Professional MFC with Visual C++*, Wrox Press Ltda, 1997.

•*Microsoft Visual C++ Programmer's Guides*, Microsoft Press, 1993.

11. APÊNDICE 1: SOBRE A IMPRESSÃO DOS PROGRAMAS

Como já dito anteriormente, os projetos desenvolvidos totalizaram em torno de 8000 linhas de código Visual C++ 5.0. É prática comum anexar ao relatório a impressão dos programas desenvolvidos durante o trabalho de graduação. Entretanto, isso ocasionaria em um acréscimo de mais de 130 páginas (letra em tamanho 7) a este relatório.

Assim, preferiu-se fornecer um CD-ROM com os códigos dos projetos e exemplos, facilitando a possível utilização destes futuramente.

12. APÊNDICE 2: SOBRE O SOFTWARE GRAFOS EM CD-ROM ANEXO

Os códigos dos programas desenvolvidos, compactados, cabem em apenas um disquete, por serem arquivos texto. Entretanto, fornecemos um CD-ROM devido ao fato da maioria dos exemplos mostrados se utilizarem de grandes arquivos de Figuras e Mapas, às vezes com mais de 10 MB. Assim, deixamos os códigos, o software, os exemplos, a apresentação do trabalho de graduação e este relatório, da seguinte forma:

Arquivo GRAFOS.EXE: Software Grafos
Arquivo TG.doc: este relatório
Arquivo TG.ppt: apresentação do trabalho de graduação
Diretório GRAFOS: códigos do projeto do Aplicativo já compilado.
Diretório GRAFALGORIT: códigos do projeto da DLL GrafAlgorit já compilado.
Diretório BITMAP: Mapas utilizados nos exemplos.
Arquivos GRF: exemplos.

Exemplo 5.2.1: **Caix1.grf** – Caix2.grf
Exemplo 5.2.2: Cavalo.grf – **Cavalo1.grf** – Cavalo2.grf – **Cavalo3.grf**
Exemplo 5.2.3: 7pontes.grf – **7pontes1.grf**
Exemplo 5.2.4: **WashMap.grf**
Exemplo 5.2.5: **WashAir.grf**
Exemplo 5.2.6: **SJCampos.grf**