UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

DESENVOLVIMENTO DE UM PROTÓTIPO PARA O AUXÍLIO DE DISTRIBUIÇÃO DE CARGAS

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

EVANDRO SESTREM

BLUMENAU, JUNHO/2001.

2001/1-32

DESENVOLVIMENTO DE UM PROTÓTIPO PARA O AUXÍLIO DE DISTRIBUIÇÃO DE CARGAS

EVANDRO SESTREM

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Maurício Capobianco Lopes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Maurício Capobianco Lopes

_____

Prof. Paulo César Rodacki Gomes

_____

Prof. Roberto Heinzle

“É melhor tentar e falhar

que preocupar-se e ver a vida passar.

É melhor tentar ainda em vão,

que sentar-se fazendo nada até o final.

Eu prefiro na chuva caminhar,

que em dias tristes em casa me esconder.

Prefiro ser feliz, embora louco,

que em conformidade viver.”

Martin Luther King

AGRADECIMENTOS

À Deus, por todas as oportunidades que eu tive durante a minha vida.

Aos meus pais, José e Rosemarie, por terem feito de mim o que sou e pelo exemplo que são.

Ao meu orientador, Mauricio Capobianco Lopes, pela sua paciência, dedicação e sabedoria.

Aos meus amigos, que me ajudaram no desenvolvimento deste trabalho e que, sempre que possível, tentaram fazer com que eu não pensasse apenas no TCC.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta o desenvolvimento e a implementação de um protótipo para o auxílio de distribuição de cargas. Este protótipo possibilita utilizar entregas agendadas para atender cargas que necessitam ser entregues. Para isto foram estudados conceitos e algoritmos da teoria dos grafos, sendo utilizado o algoritmo de *Dijkstra* em conjunto com listas de *Fibonacci*. O uso das listas de *Fibonacci* tornou a implementação do algoritmo muito eficiente. Para a especificação do sistema foi utilizada a linguagem de modelagem UML através da ferramenta case *Rational Rose* e para a implementação utilizou-se ASP e Delphi.

ABSTRACT

This work presents the development of a software prototype to help in the distribution of loadings. This prototype allows the use of scheduled delivery to attend loadings that need to be delivered. In order to this a study of the concepts and graphs theory algorithm was done where the Dijkstra algorithm and Fibonacci heaps were used together. The use of Fibonacci heaps turned the implementation of the main algorithm to be much more efficient. The tool used for the especification in UML was Rational Rose and the language programming was ASP and Delphi.

1 INTRODUÇÃO

A logística empresarial estuda como a administração pode prover melhor nível de rentabilidade nos serviços de distribuição aos clientes e consumidores, através de planejamento, organização e controle efetivo para as atividades de movimentação e armazenagem que visam facilitar o fluxo de produtos (Ballou, 1993).

No meio empresarial nunca se falou tanto em logística como agora. Muitos fatores explicam essa tendência. De um lado, a maior preocupação com os custos nas empresas; de outro, como decorrência da maior competição pelo mercado consumidor, a necessidade de garantir prazos de distribuição e oferecer um melhor nível de serviços de forma geral (Alvarenga, 1994).

O custo de transporte representa a maior parcela dos custos logísticos na maioria das empresas. Pode variar entre 4% e 25% do faturamento bruto de uma empresa superando, em muitos casos, o lucro operacional. Em 1998, o custo total de transporte nos EUA foi de R$ 529 bilhões, representando 59% de todos os custos logísticos e 6,2% do PIB. No Brasil, estima-se que esses custos estão na ordem de R$ 60 bilhões. (Nazário, 2000)

Segundo Nazário (2000), as principais funções do transporte na Logística estão ligadas principalmente às dimensões de tempo e lugar. Desde os primórdios, o transporte de mercadorias tem sido utilizado para disponibilizar, em tempo hábil, produtos onde existe demanda potencial. Mesmo com o avanço de tecnologias que permitem a troca de informações em tempo real, o transporte continua sendo fundamental para que o objetivo logístico seja alcançado, que é o produto certo, na quantidade certa, na hora certa, no lugar certo ao menor custo possível.

Segundo Ballou (1993), as pequenas cargas representam uma oportunidade para os gerentes de tráfego reduzirem o dispêndio total de transportes. Pequenos carregamentos consolidados em cargas maiores podem gerar substanciais reduções de custos. Assim, o desenvolvimento de um sistema que auxiliasse as empresas de transporte a definirem as melhores rotas de transporte, para um aproveitamento melhor das cargas, permitiria que as mesmas reduzissem seus custos e aumentassem sua eficiência.

Problemas de logística, como os citados acima, encontram fundamentos para sua solução, na teoria dos grafos. Ao contrário de muitos ramos da matemática, nascidos de especulações puramente teóricas, a teoria dos grafos tem sua origem no confronto de problemas práticos relacionados a diversas especialidades e na constatação da existência de estruturas e propriedades comuns, dentre os conceitos relacionados a esses problemas (Boaventura, 1979). Dentre as técnicas existentes para a solução de problema algorítmico em grafos, a busca ocupa lugar de destaque, pelo grande número de problemas que podem ser resolvidos através da sua utilização (Szwarcfiter, 1984).

De um modo geral, a área de algoritmos em grafos tem como interesse principal a resolução de problemas, tendo em mente uma preocupação computacional, ou seja, o objetivo principal é encontrar algoritmos, eficientes se possível, para resolver um dado problema em grafos (Pereira, 1999).

Ao lado da teoria dos fluxos em redes, o problema geral de caminhos em grafos é de importância primordial na teoria dos grafos e uma das questões fundamentais em problemas de logística. O Problema do Menor Caminho é o mais importante problema relacionado com a busca de caminhos em grafos, em vista de sua ligação direta com uma realidade encontrada a todo momento (Boaventura, 1979). Pode-se citar, como exemplo, o problema do caixeiro-viajante, que consiste na determinação da rota de menor custo para uma pessoa que deseja visitar diversas cidades e retornar ao ponto de partida passando apenas uma vez em cada cidade.

Assim, neste trabalho a teoria dos grafos será utilizada para o desenvolvimento de um sistema para atender a situação de melhor aproveitamento de rota para caminhões de uma transportadora que tenham viagens agendadas, mas que não estão com a sua capacidade de transporte completa. Um exemplo é quando um caminhão faz uma entrega e tem que regressar ao ponto de partida. Geralmente esta viagem é feita com o caminhão vazio gerando um prejuízo para a transportadora. Assim, será desenvolvido um sistema para auxiliar a transportadora reduzir este tipo de problema.

1.1 OBJETIVOS

O objetivo deste trabalho é implementar um protótipo que auxilie a distribuição de cargas.

Como objetivos específicos destacam-se:

a) a construção de um site onde o transportador possa cadastrar suas cargas e possibilidades de entregas;
b) o uso da teoria dos grafos para determinar o melhor caminho para as entregas.

1.2 ESTRUTURA

O presente trabalho está subdividido em capítulos que serão explicitados a seguir.

O primeiro capítulo apresenta a contextualização e justificativa para o desenvolvimento da proposta do trabalho.

O segundo capítulo aborda grafos falando de seus conceitos e características. Este capítulo também fala sobre algoritmos de menor caminho.

O terceiro capítulo explica listas de *Fibonacci*, que são utilizadas para implementar filas de prioridade.

O quarto capítulo trata sobre o desenvolvimento do trabalho, mostrando os diagramas de classe, casos de uso e diagramas de seqüência. Este capítulo também explica a implementação do protótipo.

O quinto capítulo apresenta as considerações finais, abrangendo as conclusões do desenvolvimento deste trabalho, as dificuldades encontradas e as sugestões para próximos trabalhos.

2 GRAFOS

2.1 HISTÓRICO

De um modo geral, a área de algoritmos em grafos pode ser caracterizada como aquela cujo interesse principal é resolver problemas de uma forma computacional. O objetivo principal é encontrar algoritmos para resolver um problema em grafos, de forma eficiente se possível.

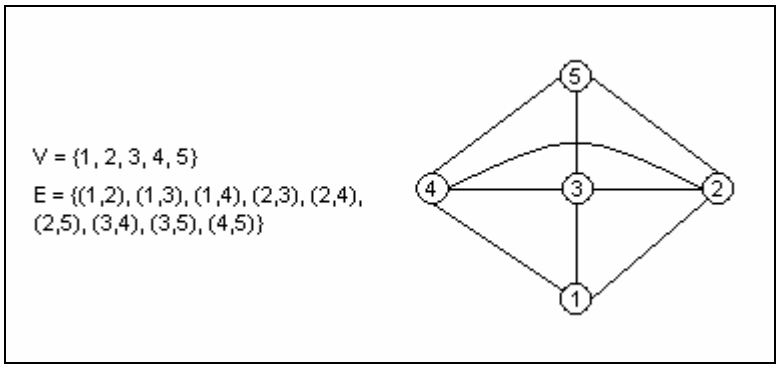
A teoria dos grafos tem contribuído para a análise de uma ampla variedade de problemas combinatoriais e pode ser utilizado para resolver problemas de análise de caminho crítico, sistema de comunicação, estudo de transmissão de informações, escolha de uma rota ótima, entre outros.

2.2 DEFINIÇÃO DE GRAFO

Segundo Szwarcfiter (1984), um *grafo* G(*V*, *E*) é definido como um conjunto finito não-vazio *V* e um conjunto *E* de pares não-ordenados de elementos distintos de *V*. Os elementos de *V* são os *vértices* e os de *E* são as *arestas* de G, respectivamente. Cada aresta $e \in E$ será denotada pelo par de vértices $e = (v, w)$ que a forma. Nesse caso, os vértices *v* e *w* são os extremos (ou extremidades) da aresta *e*, sendo denominados adjacentes. A aresta *e* é incidente a ambos *v* e *w*.

Um grafo pode ser visualizado através de uma *representação geométrica*, na qual seus vértices correspondem a pontos distintos do plano em posições arbitrárias, enquanto que a cada aresta (*v*, *w*) é associada uma linha arbitrária unindo os pontos correspondentes a *v* e *w*. Um exemplo de grafo pode ser visto na figura 1.

Figura 1 - Grafo e sua representação geométrica



Fonte: Szwarcfiter (1984)

Do ponto de vista geométrico, um grafo pode ser descrito, em um espaço euclidiano de *n* dimensões, como sendo um conjunto *V* de pontos e conjunto *A* de curvas contínuas que não se interceptam, satisfazendo as seguintes condições (Furtado, 1973):

a) toda curva fechada de *A* contém exatamente um ponto de *V*;
b) toda curva aberta de *A* contém exatamente dois pontos de *V*;
c) as curvas de *A* não tem pontos em comum, a não ser pontos de *V*.

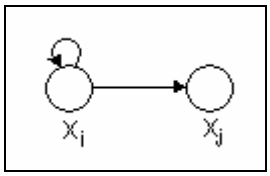
Segundo Sedgewick (1988), o número de arestas de um grafo pode variar de 0 até ½ $v(v - 1)$, onde *v* é o número de vértices. Grafos com todas as arestas são chamados de grafos completos. Grafos com alguns vértices (até $v \log v$) são denominados de esparsos. Grafos com quase todas as arestas são chamados de densos. Existem ainda os grafos direcionados e os grafos valorados, descritos nas próximas seções.

2.3 GRAFOS DIRECIONADOS

Um grafo direcionado (dígrafo) D(*V*, *E*) é um conjunto finito não vazio *V* (os vértices), e um conjunto *E* (as arestas) de pares ordenados de vértices distintos. Portanto, num dígrafo cada aresta (*v*, *w*) possui uma única direção de *v* para *w*. Dessa forma, (*v*, *w*) é divergente de *v* e convergente a *w* (Szwarcfiter, 1984).

Cada arco é representado por uma seta cujo sentido corresponde à orientação do par ordenado. Um exemplo de grafo direcionado pode ser visto na figura 2.

Figura 2 - Grafo orientado

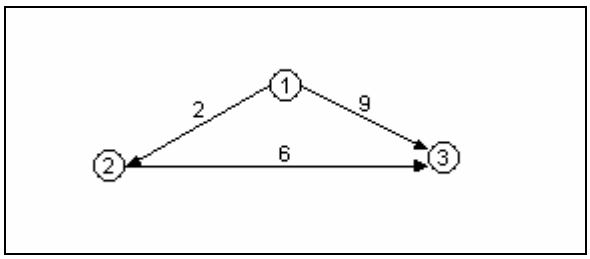


Em um grafo orientado, um arco de forma (x_i, x_i) é possível e chama-se laço. Porém, não é possível definir um sentido desse arco. Dessa forma, um vértice não pode possuir dois laços de sentidos opostos (Boaventura, 1979).

2.4 GRAFO VALORADO

Um grafo, no qual um número w_{ij} está associado a cada aresta, é denominado de grafo valorado e o número w_{ij} é chamado o custo da aresta. Em redes de comunicação ou transportes estes custos representam alguma quantidade física, tal como distância, eficiência, capacidade da aresta correspondente, etc (Rabuske, 1992). Um exemplo de grafo valorado pode ser visto na figura 3.

Figura 3 - Exemplo de grafo valorado



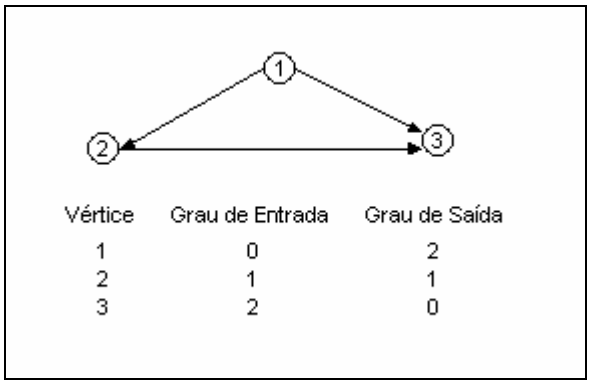
2.5 RELAÇÕES DE ADJACÊNCIA

Dois vértices *v* e *w* são ditos adjacentes em um grafo G se existe em G a aresta {*v*, *w*}. A relação de incidência é definida entre um vértice e uma linha: uma linha é incidente a um vértice se o vértice é uma de suas extremidades.

Em grafos dirigidos, um arco (*v*, *w*) é dito exteriormente incidente a *v* e interiormente incidente a *w* (Furtado, 1973).

Seja D(*V*, *E*) um dígrafo e um vértice *v* ϵ *V*. O *grau de entrada* de *v* é o número de arestas convergentes a *v*. O *grau de saída* de *v* é o número de arestas divergentes de *v*. Uma *fonte* é um vértice com grau de entrada nulo, enquanto que um *sumidouro* (ou *poço*) é um com grau de saída nulo (Szwarcfiter, 1984). Na figura 4, o vértice 1 é uma fonte e o vértice 3 é um sumidouro.

Figura 4 - Exemplo de grau

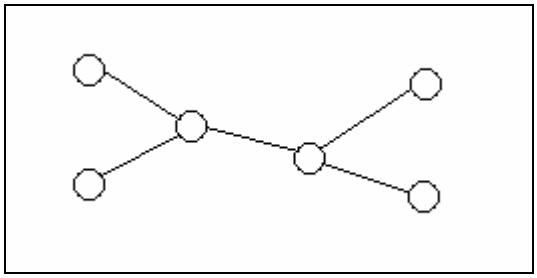


2.6 ÁRVORES

Um grafo que não possui ciclos é chamado acíclico. Denomina-se *árvore* a um grafo T(*V*, *E*) que seja acíclico e conexo. Se um vértice *v* da árvore T possui Grau <= 1 então *v* é uma *folha* (Szwarcfiter, 1984).

Um grafo G é uma árvore se e somente se existir um único caminho entre cada par de vértices de G. Segundo Rabuske (1992), um caminho é qualquer seqüência de arestas onde o vértice final de uma aresta é o vértice inicial da próxima. Um exemplo de árvore pode ser visto na figura 5.

Figura 5 - Árvore



2.7 REPRESENTAÇÃO DE GRAFOS

Segundo Szwarcfiter (1984), a representação de grafos adequadas ao uso em computador tem como objetivo fornecer estruturas que correspondam univocamente a um grafo dado e possam ser armazenadas sem dificuldade, em um computador.

Dentre as várias representações para computador, as mais importantes são as *representações matriciais* e as *por listas*.

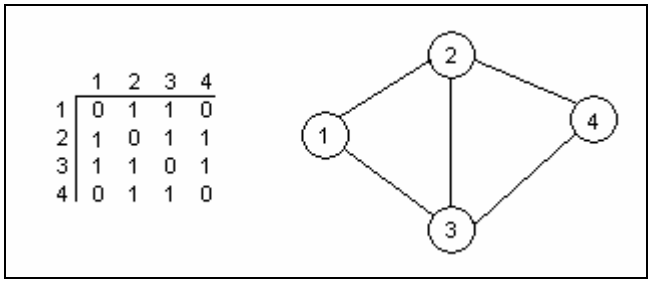
Na representação matricial é utilizada a matriz de adjacências. Em um grafo G(*V*, *E*) a *matriz de adjacências* R = (r_{ij}) é uma matriz *n* x *n* tal que:

r_{ij} = 1 <==> (v_i,v_j) ϵ *E*

r_{ij} = 0 caso contrário.

Ou seja, r_{ij} = 1 quando os vértices v_i,v_j forem adjacentes e r_{ij} = 0, caso contrário. Uma matriz de adjacências caracteriza univocamente um grafo. Um exemplo de matriz de adjacências pode ser visto na figura 6.

Figura 6 - Matriz de adjacências

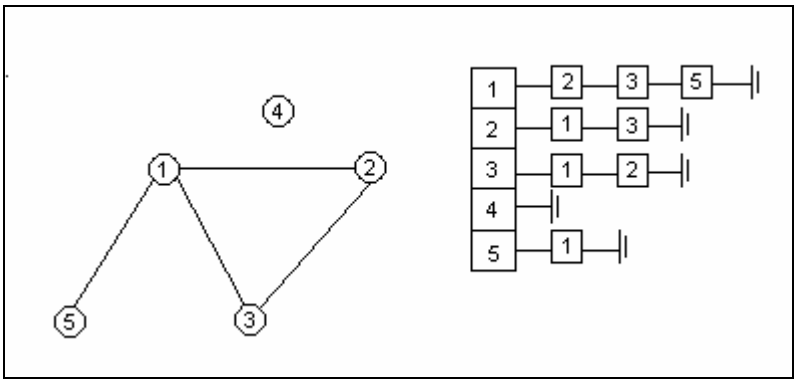


A principal desvantagem desta representação é o espaço necessário para o armazenamento da matriz. Em qualquer caso existem n^2 informações binárias, o que significa um espaço $0(n^2)$. Segundo Sedgewick (1988), a matriz de adjacências é recomendada apenas para grafos densos.

Uma das principais representações de grafos por listas é a *estrutura de adjacências*. Seja G(*V*, *E*) um grafo. A estrutura de adjacências A de G é um conjunto de n listas A(*v*), uma para cada *v* ϵ *V*. Cada lista A(*v*) é chamada de *lista de adjacências* do vértice *v*, e possui os vértices *w* adjacentes a *v* em G. Ao adicionar uma conexão de um vértice *v* a um vértice *w*, o

vértice *v* é adicionado na lista adjacente de *w* e o vértice *w* é adicionado na lista adjacente de *v*. Um exemplo de estrutura de adjacências pode ser visto na figura 7.

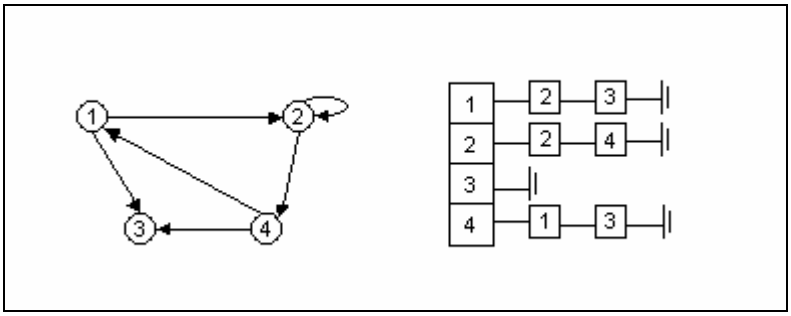
Figura 7 - Estrutura de adjacências



A estrutura de adjacências é a representação mais utilizada nas implementações em algoritmos de grafos, pois o espaço utilizado por uma estrutura de adjacências é 0 (*n* + *m*), ou seja, linear com o tamanho de *G*, onde *n* é o número de arestas e *v* é o número de vértices

Segundo Sedgewick (1988), grafos dirigidos e valorados são representados com estruturas semelhantes. Para grafos dirigidos a única alteração é que cada conexão é representada apenas uma vez: uma conexão de um vértice *v* para um vértice *w* é representada por um valor *true* em [*v, w*] na matriz de adjacências e na representação por lista de adjacências o *w* está na lista de adjacências de *v*. A figura 8 é um exemplo de uma estrutura de adjacências para um grafo dirigido.

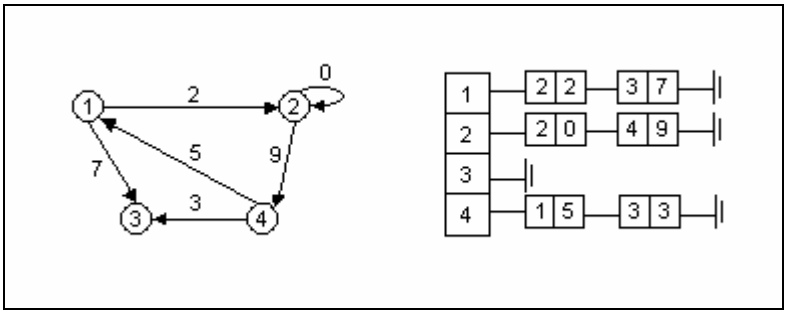
Figura 8 - Estrutura de adjacências para um grafo dirigido



Para grafos valorados a única alteração em uma matriz de adjacências é que se utiliza o valor ao invés de valores booleanos (utiliza-se um valor inválido para representar *false*); em uma estrutura de adjacências é incluído um campo para armazenar o valor em cada item da

lista de adjacências. O espaço utilizado por esta estrutura de adjacência é $n + 2m$. A figura 9 é um exemplo de uma estrutura de adjacência para um grafo dirigido valorado contendo o campo adicional para armazenar o valor de cada aresta.

Figura 9 - Estrutura de adjacências para um grafo dirigido e valorado



2.8 BUSCA EM GRAFOS

Segundo Szwarcfiter (1984), a busca é uma das técnicas mais importantes para a solução de problemas algorítmicos em grafos, devido ao grande número de problemas que a busca pode resolver. Essa importância cresce ainda mais, quando se procura algoritmos considerados eficientes.

A busca procura resolver o problema de explorar um grafo, ou seja, como caminhar pelos seus vértices e arestas.

Quando o grafo é uma árvore, a busca é mais simples. Uma das formas de fazer essa busca é denominado de *preordem*, seguindo os seguintes passos:

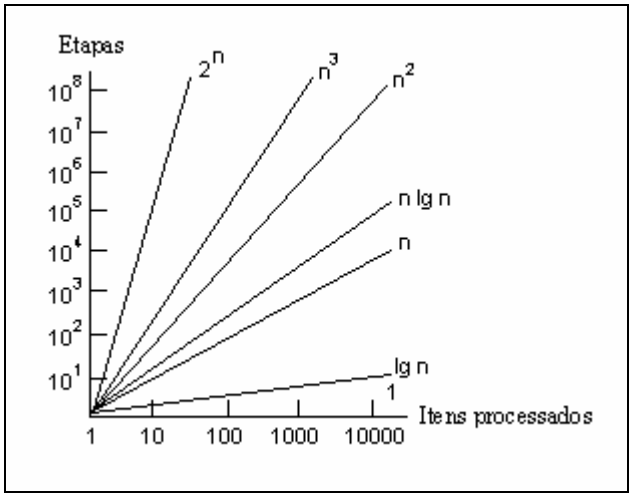
a) Se a árvore for vazia, não há nada a fazer. Caso contrário:
b) Visite a raiz da árvore.
c) Caminhe pela subárvore mais à esquerda da raiz,
d) após pela 2ª mais à esquerda,
e) após pela 3ª mais à esquerda,
f) e assim por diante.

Uma outra forma de se caminhar em árvores enraizadas é conhecido como *ordem de nível*. Neste processo, o caminhamento é nível a nível, da esquerda para direita.

Quando o grafo não é uma árvore, a busca é mais difícil, pois não existem, de forma absoluta, os conceitos de esquerda, direita e nível. A principal dificuldade em relação aos algoritmos de busca diz respeito à complexidade do problema estudado.

Segundo Lafore (1999), a notação *Big O* é utilizada para dizer quão eficiente é um algoritmo. A figura 10, mostra a relação entre o crescimento do número de etapas que um algoritmo executa e o número de itens que ele processa.

Figura 10 - Big O



Fonte: baseado em Kruse (1994).

A tabela 1 mostra a quantidade de passos executados em relação ao número de itens (*n*) processados.

Tabela 1 - Big O

N	**1**	**lg *n***	***N***	***n* lg *n***	n^2	n^3	2^n
1	1	0,00	1	0	1	1	2
10	1	3,32	10	33	100	1.000	1024
100	1	6,64	100	66	10.000	1.000.000	1.268×10^{30}
1000	1	9,97	1000	997	1.000.000	10^9	1.072×10^{301}

Fonte: baseado em Kruse (1994).

2.9 ALGORITMO BÁSICO

Seja G um grafo conexo em que todos os seus vértices se encontram desmarcados. No passo inicial, marca-se um vértice escolhido ao acaso. No passo geral, seleciona-se algum vértice *v* que esteja marcado e seja incidente a alguma aresta (*v*, *w*) ainda não selecionada. A aresta (*v*, *w*) passa então a ser selecionada e o vértice *w* é marcado, se ele já não estiver. O processo termina quando todas as arestas de G tiverem sido selecionadas. Esse tipo de caminhamento é chamado de *busca* no grafo G. Este algoritmo está descrito no quadro 1.

Quadro 1 - Algoritmo: busca geral

```
 dados grafo G(V, E)
 escolher e marcar um vértice inicial
 enquanto existir algum vértice v marcado e incidente a uma aresta (v, w)
não explorada, efetuar
    escolher o vértice v e explorar a aresta (v, w)
    se w é não marcado então marcar w
```

2.10 BUSCA EM PROFUNDIDADE

Segundo Szwarcfiter (1984), a busca *em profundidade* escolhe para explorar o vértice *mais recentemente* alcançado na busca dentre todos os vértices marcados e incidentes a alguma aresta ainda não explorada. A busca em profundidade possui complexidade O (*n* + *m*). O algoritmo de busca em profundidade, segundo Heinzle (2001), está descrito no quadro 2.

Quadro 2 - Algoritmo de busca em profundidade

```
crie duas pilhas abertos e fechados
inicialize a pilha abertos = [nó início]
enquanto abertos não estiver vazia faça
   remova o nó do topo de abertos e chame-o de x
   se x é conclusivo termine com sucesso
   senão busque todos os filhos de x
      dispense os filhos de x que já estão em abertos ou fechados
      coloque sobre a pilha abertos os filhos
   remanescentes de x (na ordem em que foram buscados??)
   coloque x sobre a pilha fechados
fim enquanto
```

2.11 BUSCA EM PROFUNDIDADE – DÍGRAFOS

A partir de um vértice s = v_1 denominado *raiz*, constróem-se caminhos Q = v_1, ..., v_k. Se existe algum vértice w divergente de v_k tal que w nunca pertenceu a algum caminho Q, então w é incluído em Q, o qual se torna v_1, ..., v_k, w e o processo é repetido. Caso contrário, se não existe w nessas condições, o vértice v_k é retirado do caminho, transformando-o em Q = v_1, ..., v_{k-1}, e o processo é repetido. O término da busca se dá quando o vértice v_1 é retirado de Q. Esse processo é denominado *busca em profundidade de raiz v_1* (Szwarcfiter, 1984). Este algoritmo está descrito no quadro 3.

Os vértices z não alcançáveis de v não serão incluídos em Q. Assim, apenas os vértices e arestas alcançáveis da raiz são incluídos na lista.

Quadro 3 - Algoritmo de busca em profundidade em dígrafos

```
dados dígrafo D(V,E)
procedimento P(v)
   marcar v
   colocar v na pilha Q
   para w ∈ A(v) efetuar
      visitar (v,w)
      se w não marcado então P(w)
   retirar v de Q
desmarcar todos os vértices
definir uma pilha Q
definir uma raiz s ∈ V
P(s)
```

Segundo Szwarcfiter (1984), a busca em profundidade para dígrafos possui complexidade O (*n* + *m*).

2.12 BUSCA EM LARGURA

Segundo Szwarcfiter (1984) a busca em largura utiliza o seguinte critério de escolha de vértice marcado: “dentre todos os vértices marcados e incidentes a alguma aresta ainda não explorada, escolher aquele *menos recentemente* alcançado na busca”.

A busca em largura utiliza uma fila, assim como a busca em profundidade é implementada como auxílio de uma pilha. O algoritmo de busca em largura, segundo Heinzle (2001), está descrito no quadro 4.

Quadro 4 - Algoritmo de busca em largura

```
crie duas filas abertos e fechados
inicialize a fila abertos = [nó início]
enquanto abertos não estiver vazia faça
   remova um nó da fila abertos e chame-o de x
   se x é conclusivo termine com sucesso
   senão busque todos os filhos de x
      dispense os filhos de x que já estão em abertos ou fechados
      coloque na fila abertos os filhos remanescentes de x
   coloque x na fila fechados
fim enquanto
```

2.13 PROBLEMA DO CAMINHO MAIS CURTO

O caminho crítico é o mais importante problema relacionado com a busca de caminhos em grafos (Boaventura, 1979). Dentre os algoritmos utilizados para resolver esse problema, destacam-se os algoritmos de *Bellman-Ford* e *Dijkstra*.

Os algoritmos descritos a seguir utilizam uma técnica conhecida como otimização. Cada vértice $v \in V$ possui um atributo distância $d[v]$, que é a distância do menor caminho do vértice origem s à v e um atributo precedente $\pi[v]$, que é o vértice precedente à v no menor caminho. Estes atributos são inicializados no algoritmo descrito no quadro 5.

Quadro 5 - Algoritmo de Inicialização

```
INITIALIZE-SINGLE-SOURCE (G,s)
for each vertex v ∈ V[G] do
     d[v] ← ∞
     π[v] ← NIL
d[s] ← 0
```

Depois da inicialização, $\pi[v]$ é NIL para todo $v \in V$, $d[v] = 0$ para $v = s$, e $d[v] = \infty$ para $v \in V - \{s\}$.

O processo de otimizar uma aresta (*u*, *v*) consiste em verificar se o menor caminho a *v* pode ser otimizado passando pelo vértice *u*. Em uma otimização, os valores *d*[*v*] e π[*v*] devem ser atualizados. O algoritmo descrito no quadro 6 executa uma otimização na aresta (*u*, *v*).

Quadro 6 - Algoritmo de Otimização

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

2.14 ALGORITMO DE BELLMAN-FORD

Segundo Cormen (1989), o algoritmo de *Bellman-Ford* soluciona o problema de encontrar o menor caminho de um vértice para todos os outros vértices de um grafo valorado G = (*V*, *E*), sendo que os valores podem ser negativos. O algoritmo de *Bellman-Ford* retorna um valor booleano indicando se existe um ciclo negativo que pode ser alcançado pelo vértice de origem. Se este ciclo existir, o algoritmo indica que não existe solução para o problema. Se o ciclo não existe, o algoritmo gera os menores caminhos e suas distâncias.

O algoritmo de *Bellman-Ford* utiliza a técnica de otimização para diminuir a distância estimada de um vértice *v* ao vértice de origem *s*. O algoritmo descrito no quadro 7 retorna t*rue* se o grafo não contém ciclos negativos que podem ser alcançados pelo vértice origem.

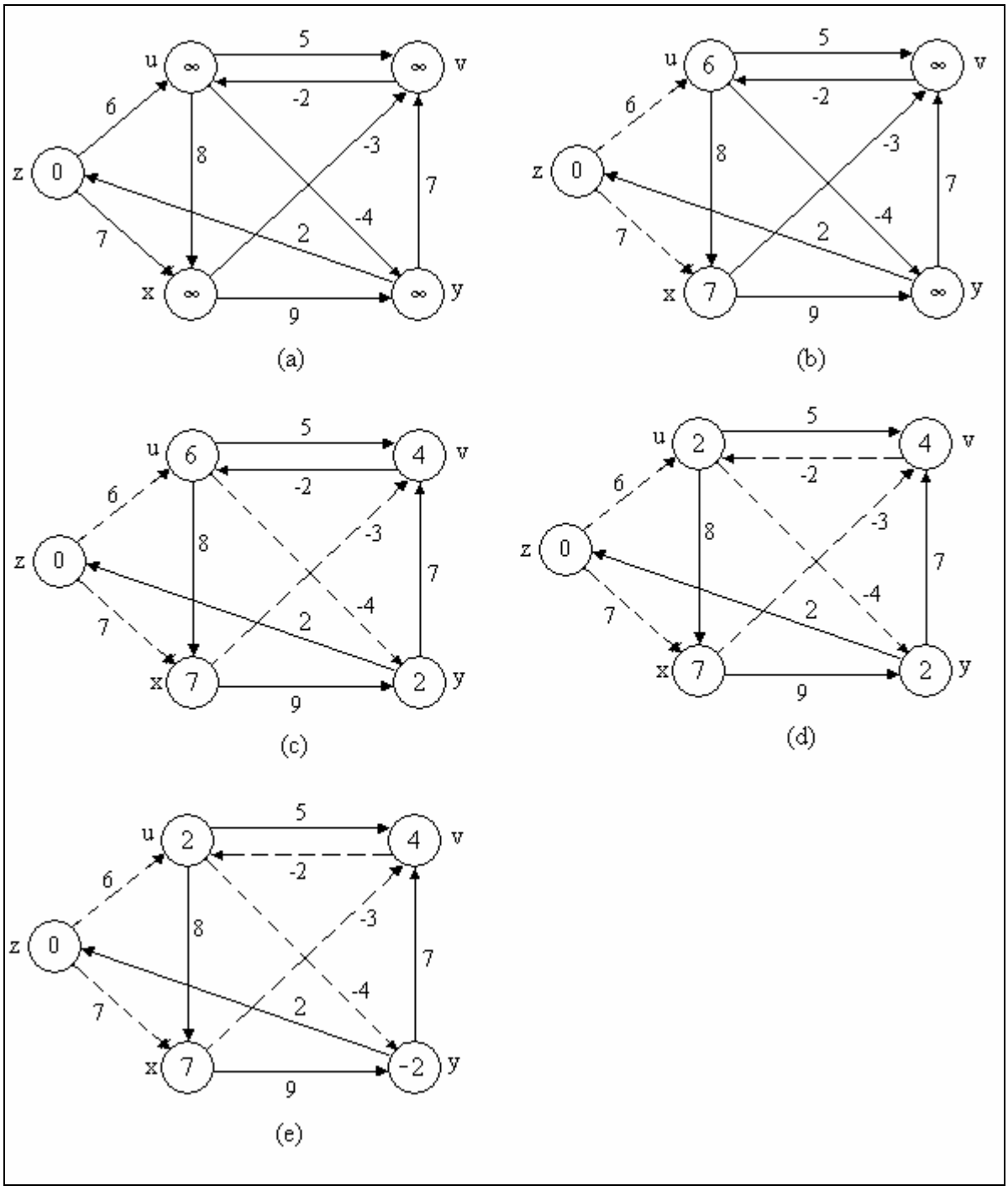
Quadro 7 - Algoritmo de Bellman-Ford

```
BELLMAN-FORD (G,w,s)
1. Initialize-Single-Source(G,s)
2. for i ← 1 to |V[G]| - 1 do
3.      for each edge (u,v) ∈ E[G] do
4.           Relax(u,v,w)
5. for each edge (u,v) ∈ E[G] do
6.      if d[v] > d[u] + w(u,v) then
7.           return false
8. return true
```

A figura 11 mostra um exemplo da execução do algoritmo *Bellman-Ford*. Após a execução da inicialização, o algoritmo executa |*V*| - 1 passos nas arestas do grafo. Cada passo

é uma iteração do loop das linhas 2-4 e consiste em otimizar cada aresta do grafo uma vez. A figura 11 (b) – (e) mostra o estado do algoritmo após cada um dos 4 passos nas arestas. Depois de executar $|V|$ - 1 passos, as linhas 5-8 verificam se existe um ciclo negativo e retornam o valor booleano apropriado.

Figura 11 - Exemplo do algoritmo Bellman-Ford



Fonte: Baseado em Cormen (1989).

Segundo Cormen (1989), o tempo de execução do algoritmo de *Bellman-Ford* é O(*VE*).

2.15 ALGORITMO DE DIJKSTRA

Segundo Cormen (1989), o algoritmo de *Dijkstra* soluciona o problema de encontrar o menor caminho entre um vértice e todos os outros vértices de um grafo valorado G = (*V*, *E*), sendo que todos os valores do grafo devem ser não-negativos.

O algoritmo de *Dijkstra* mantém uma lista *S* de vértices que já tiveram determinados o menor caminho e o custo ao vértice de origem *s*. Assim, para cada vértice $v \in S$, o valor de *d*[*v*] é a menor distância entre *v* e *s*. O algoritmo repetidamente seleciona um vértice *v* com o menor caminho estimado que não pertence à *S*, insere *v* em *S*, e otimiza as arestas de *v*. A implementação do quadro 8 mantém uma fila de prioridades *Q* que contém os vértices *V* – *S*, indexados pelo atributo *d* de cada vértice e assume que o grafo G é representado por listas de adjacências.

Quadro 8 - Algoritmo de Dijkstra

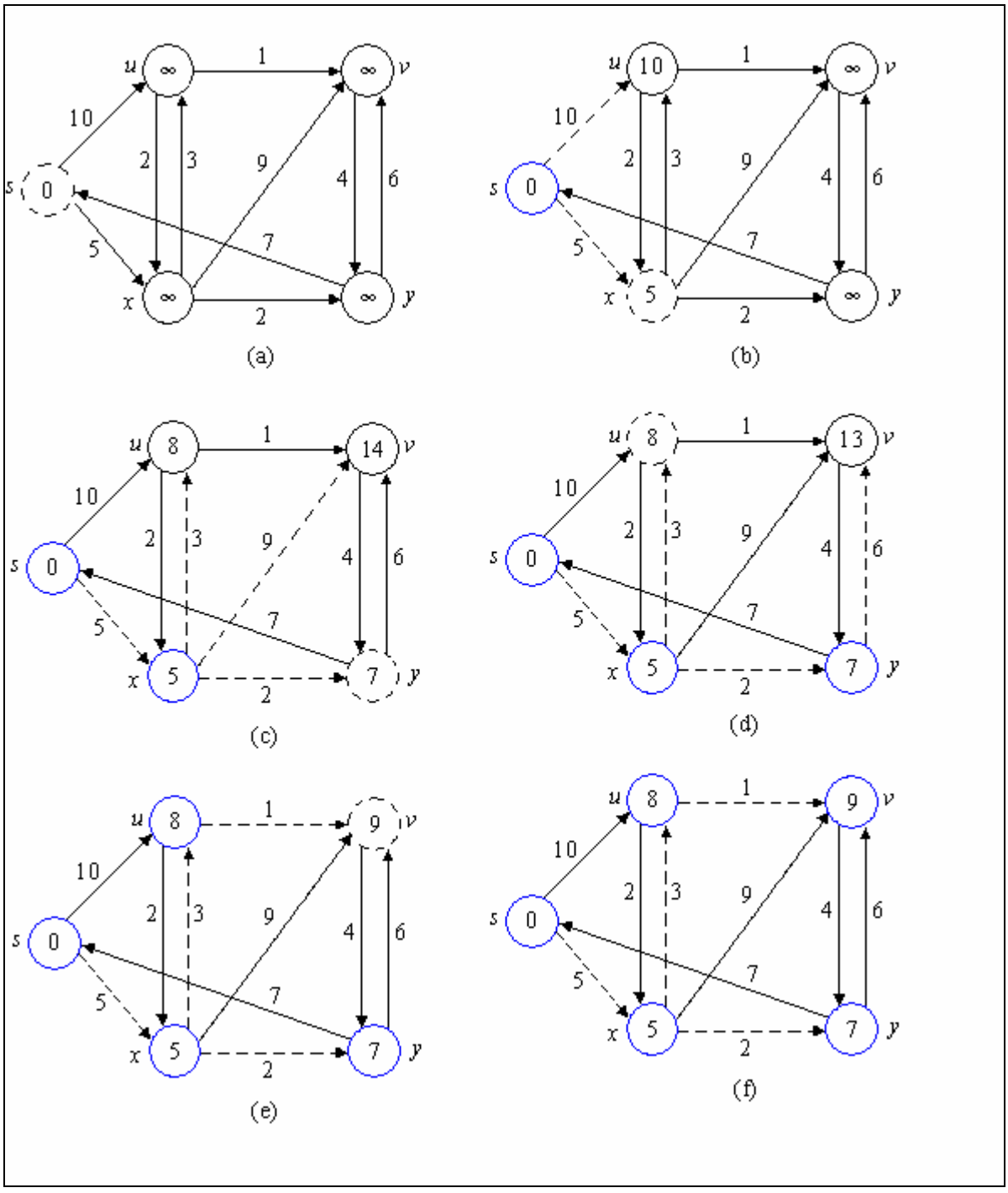
```
Dijkstra (G,w,s)
1. Initialize-Single-Source(G,s)
2. S ← {}
3. Q ← V[G]
4. while Q <> {} do
5.      u ← ExtractMin(Q)
6.      S ← S U {u}
7.      for each vertex v ∈ Adj[u] do
8.           Relax(u,v,w)
```

A linha 1 inicializa os valores distância e precedente de cada vértice. O vértice *s* possui distância 0 e os demais possuem ∞. A linha 2 inicializa a lista *S* como vazia. A linha 3 inicializa a fila de prioridades *Q* com todos os vértices de *G*. Cada execução das linhas 4-8 extrai um vértice *u* de *Q* e insere na lista *S* (na primeira execução, *u* = *s*). As linhas 7-8 otimizam cada aresta (*u*, *v*) adjacente a *u*. Os vértices nunca são inseridos na lista *Q* depois da linha 3 e cada vértice é extraído de *Q* e inserido em *S* uma vez, dessa forma o loop das linhas 4-8 é executado uma vez para cada vértice do grafo.

A figura 12 é um exemplo de execução do algoritmo *Dijkstra*. Neste exemplo, o vértice origem é o vértice *s*. A distância do menor caminho estimada de cada vértice é o valor que está dentro dele. As arestas pontilhadas indicam vértices precedentes, por exemplo: se a

aresta (*u*, *v*) está pontilhada, então $\pi[v] = u$. Os vértices azuis são os vértices que estão na lista *S* e o vértice pontilhado foi o escolhido da linha 5 do algoritmo.

Figura 12 - Exemplo do algoritmo de Dijkstra



Fonte: Baseado em Cormen (1989).

2.15.1 DESEMPENHO DOS ALGORITMOS

Segundo Cormen (1989), considerando a fila de prioridades *Q* implementada como um vetor linear, cada operação ExtrairMinimo da fila de prioridades leva O(*V*), e como esta operação é executada |*V*| vezes, o tempo total de ExtrairMinimo é $O(V)^2$. Como cada vértice $v \in V$ é inserido na lista *S* uma vez, cada aresta da lista de adjacências de *v* é examinado no *loop* das linhas 4-8 uma vez durante a execução do algoritmo. Como o número total de arestas em todas as listas de adjacências é |*E*|, existem |*E*| iterações neste loop, cada iteração levando O(1). Desta forma, o tempo de execução total é $O(V^2 + E) = O(V^2)$.

Se a fila de prioridades *Q* for implementada como uma lista binária, cada operação ExtrairMinimo leva O(log *V*). Como visto, existem |*V*| operações ExtrairMinimo. O tempo para construir a lista binária é O(*V*). A atribuição $d[v] \leftarrow d[u] + w(u, v)$ em Relax é executada pela operação da lista binária DiminuirChave(Q, v, $d[u] + w(u, v)$), que leva tempo O(log *V*) e é executado |*V*| vezes. Desta forma, o tempo de execução é do algoritmo é de O((*V* + *E*) log *V*).

Se a fila de prioridades for implementada como uma lista de *Fibonacci*, o tempo de execução é de O(*V* log *V* + *E*), pois neste tipo de estrutura o custo de cada uma das |*V*| operações de ExtrairMinimo é de O (log *V*), e cada uma das |*E*| operações de DiminuirChave leva O(1).

Como o grafo utilizado para a resolução do problema não possui valores negativos, o algoritmo utilizado será *Dijkstra* utilizando listas de *Fibonacci*.

3 Listas de Fibonacci

Segundo Boyer (1997), listas normalmente são implementadas utilizando árvores binárias. Um dos primeiros usos para uma lista foi usá-la como uma fila de prioridade. Filas de prioridades são usadas para manter uma lista dinâmica de tarefas de diferentes prioridades. Em 1984, Michael Fredman e Robert Tarjan descreveram um novo modo de implementar listas: a lista de *Fibonacci*. A lista de *Fibonacci* é rápida principalmente nas operações de **DiminuirChave** e **Inserir**. A lista de *Fibonacci* executa estas operações utilizando tempo constante, enquanto que listas binárias executam estas operações utilizando um tempo logarítmico. Entretanto, a lista de *Fibonacci* utiliza mais memória que uma lista binária, pois necessita de informações adicionais de cada elemento.

A tabela 2 mostra as operações suportadas pela lista de *Fibonacci*.

Tabela 2 - Funções da lista de Fibonacci

Função	**Descrição**
Inserir	Insere um novo nodo na lista, o novo nodo deve conter um valor chave.
ExtrairMinimo	Retorno o nodo com o menor valor depois de extrai-lo da lista.
DiminuirChave	Associa um novo e menor valor chave a um nodo.
União	Cria uma nova lista unindo duas listas recebidas como parâmetros de entrada
Mínimo	Retorna uma referência ao nodo contendo o valor chave de menor valor
Deletar	Deleta um nodo da lista.

3.1 ESTRUTURA DA LISTA DE FIBONACCI

A lista de *Fibonacci* armazena todos os nodos em uma coleção de listas circulares duplamente encadeadas.

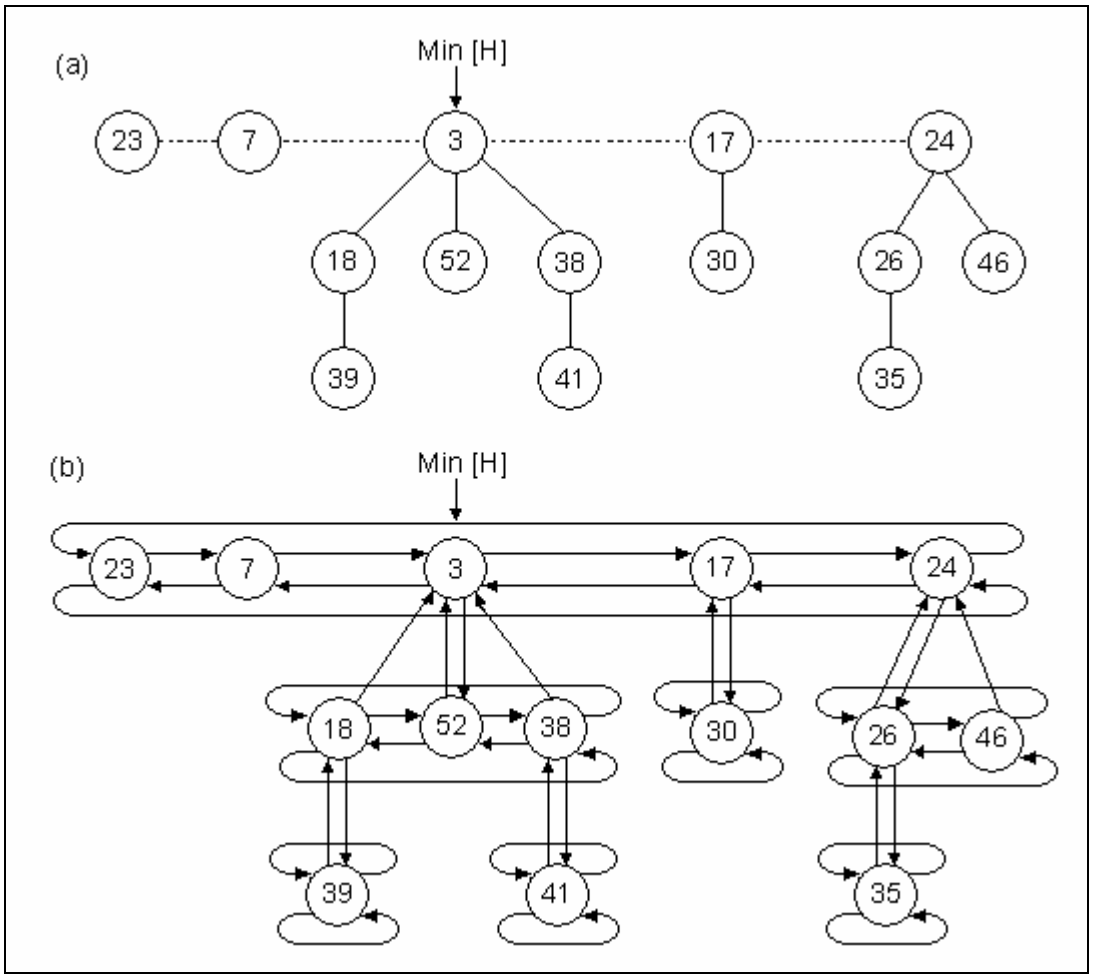
Uma lista de *Fibonacci H* é acessada através de um ponteiro *min*[*H*] para a raiz da árvore contendo um valor mínimo. Este nodo é chamado de nodo mínimo da lista de *Fibonacci*. Se uma lista de *Fibonacci H* é vazia, então *min*[*H*] = NIL.

As raízes das árvores de uma lista de *Fibonacci* são ligadas usando os ponteiros *left* e *right*, em uma lista circular duplamente encadeada chamada de **lista de raízes**. O ponteiro *min*[*H*] aponta para o nodo da lista de raízes que possui o menor valor. A lista de *Fibonacci* possui ainda a informação de quantos nodos ela possui.

Cada nodo *x* possui um campo *key*[*x*], que é um valor utilizado pela aplicação que está usando a lista de *Fibonacci*. Além dessa informação, o nodo possui um ponteiro *parent*[*x*] para o seu pai e um ponteiro *child*[*x*] para um dos seus filhos. Os filhos de *x* estão ligados em uma lista circular duplamente encadeada, que é chamada de *lista de filhos* de x. Cada filho *y* em uma lista de filhos possui ponteiros *left*[*y*] e *right*[*y*] que aponta para os seus irmão da esquerda e direita, respectivamente. Se o nodo *y* é filho único, então *left*[*y*] = *right*[*y*] = *y*. O número de filhos na lista de filhos (grau) do nodo *x* é armazenado em *degree*[*x*]. O valor booleano *mark*[*x*] indica se o nodo *x* perdeu um filho desde a última vez que *x* foi transformado em um filho de outro nodo. O uso deste campo será visto mais adiante. Inicialmente, os nodo são criados desmarcados (*mark*[*x*] = falso), e este valor permanece inalterado até que o nodo é transformado em filho de um outro nodo.

Um exemplo de lista de *Fibonacci* pode ser visto na figura 13 (a). A figura 13 (b) é uma representação mais completa mostrando os ponteiros *parent* (setas para cima), *child* (setas para baixo) e *left* e *right* (setas para os lados).

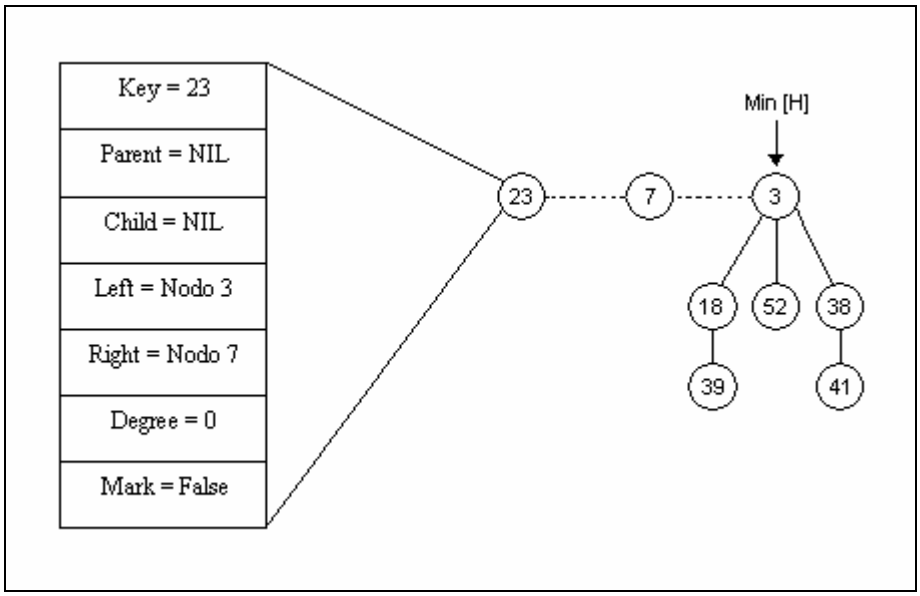
Figura 13 - Exemplo de uma lista de Fibonacci



Fonte: Cormen (1989).

Um exemplo da estrutura de um nodo da lista de *Fibonacci* pode ser visto na figura 14.

Figura 14 – Exemplo da estrutura de um nodo da lista de Fibonacci



3.2 INSERIR UM NODO

O algoritmo do quadro 9 insere um nodo *x* em uma lista de *Fibonacci*, assumindo que o nodo já está alocado e que o campo *key*[*x*] já foi informado.

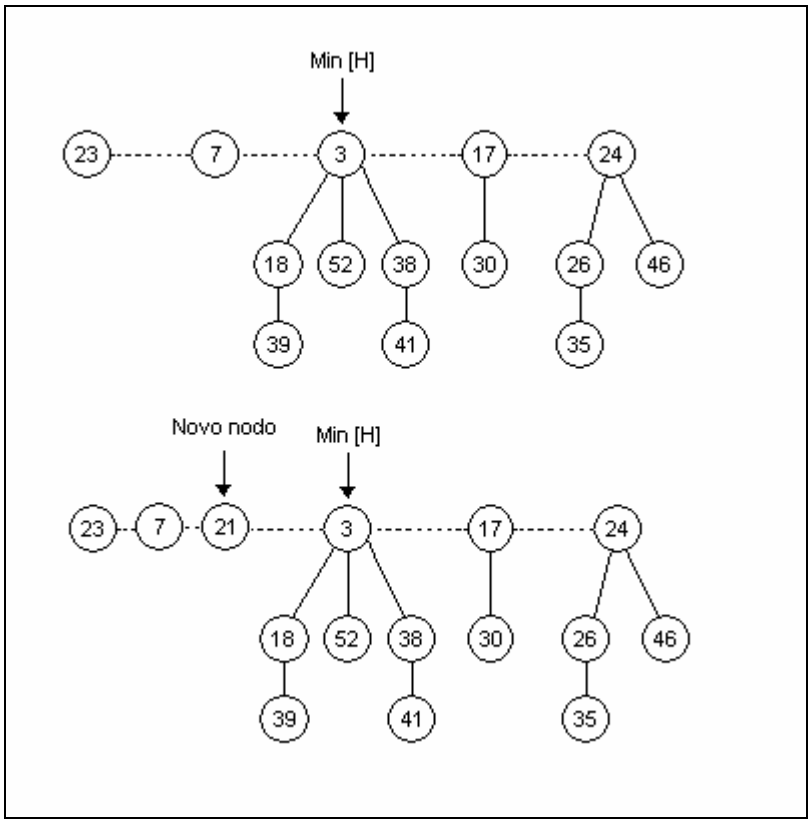
Quadro 9 - Algoritmo de inserção de um nodo na lista de Fibonacci

```
Inserir (H, x)
1. Degree[x] := 0
2. p[x] := nil
3. child[x] := nil
4. left[x] := x
5. right[x] := x
6. mark[x] := false
7. concatenate the root list containing x with root list H
8. if min[H] = nil or key[x] < key[min[H]] then
9.    min[H] := x
10. n[H] := n[H] + 1
```

As linhas 1 à 6 inicializam os campos do nodo *x*. A linha 7 adiciona *x* à lista de raízes de H, o valor *right*[*x*] passa a ser o nodo mínimo, o valor *left*[*x*] recebe o valor atual de *left* do nodo mínimo e o atributo *left* do nodo mínimo passa a ser *x*. As linhas 8 e 9 atualizam, se

necessário, o ponteiro para o nodo mínimo. A linha 10 incrementa o valor de *n[H]*. A figura 15 mostra um nodo com o valor de chave 21 inserido na lista de *Fibonacci* da figura 13.

Figura 15 - Exemplo de inserção na lista de Fibonacci



Fonte: Baseado em Cormen (1989).

3.3 EXTRAINDO O NODO MÍNIMO

O processo de extrair o nodo mínimo é o mais complicado das operações descritas neste trabalho. O algoritmo de extrair o nodo mínimo pode ser visto no quadro 10.

Quadro 10 - Algoritmo ExtrairMinimo

```
ExtrairMinimo(H)
1. z := min[H]
2. if z <> nil then
3.    for each child x of z do
4.       add x to the root list of H
```

```
5.       p[x] := nil
6.    remove z from the root list of H
7.    if z = right[z] then
8.       min[H] := nil
9.    else min[H] := right[z]
10.        Consolidate(H)
11.   n[H] := n[H] – 1
12.return z
```

A linha 1 salva um ponteiro *z* com o nodo mínimo para retornar no final. Se *z* = NIL, então a lista de *Fibonacci* já está vazia. Se a lista não está vazia, o nodo *z* é deletado de *H* inserindo todos os filhos de *z* na lista de raízes de *H* (linhas 3-5). O nodo *z* é removido da lista de raízes na linha 6. Se *z* = *right[z]* (linha 7), então *z* era o único nodo da lista de raízes e não possui filhos, o valor nil é atribuído ao nodo mínimo (linha 8); senão um outro nodo é atribuído ao nodo mínimo.

O próximo passo, no qual é diminuído o número de árvores na lista de *Fibonacci*, é consolidar a lista de raízes de *H*, que é feito por *Consolidate(H)*. Este algoritmo pode ser visto no quadro 11. Consolidar a lista de raízes consiste em executar os seguintes passos até que todo nodo da lista de raízes possua um valor em *degree* diferente dos demais nodos:

a) achar 2 nodos *x* e *y* na lista de raízes que possuam o mesmo grau (*degree*), onde *key*[*x*] <= *key*[*y*];
b) ligar *y* à *x*: remover *y* da lista de raízes, e tornar *y* um filho de *x*. Esta operação está descrita no quadro 12. O campo *degree*[*x*] é incrementado, e atribui-se *false* para o campo *mark*[*y*].

Quadro 11 - Algoritmo Consolidate

```
Consolidate(H)
1. for i := 0 to D(n[H]) do
2.    A[i] := nil
3. for each node w in the root list of H do
4.    x := w
5.    d := degree[x]
```

```
6.    while A[d] <> nil do
7.       y := A[d]
8.       if key[x] > key[y] then
9.          exchange x <--> y
10.      Ligar(H, y, x)
11.      A[d] := nil
12.      d := d + 1
13.   A[d] := x
14.min[H] := nil
15.for i := 0 to D(n[H]) do
16.   if A[i] <> nil then
17.      add A[i] to the root list of H
18.      if min[H] = nil or key[A[i]] < key[min[H]] then
19.         min[H] := A[i]
```

Quadro 12 - Algoritmo Ligar

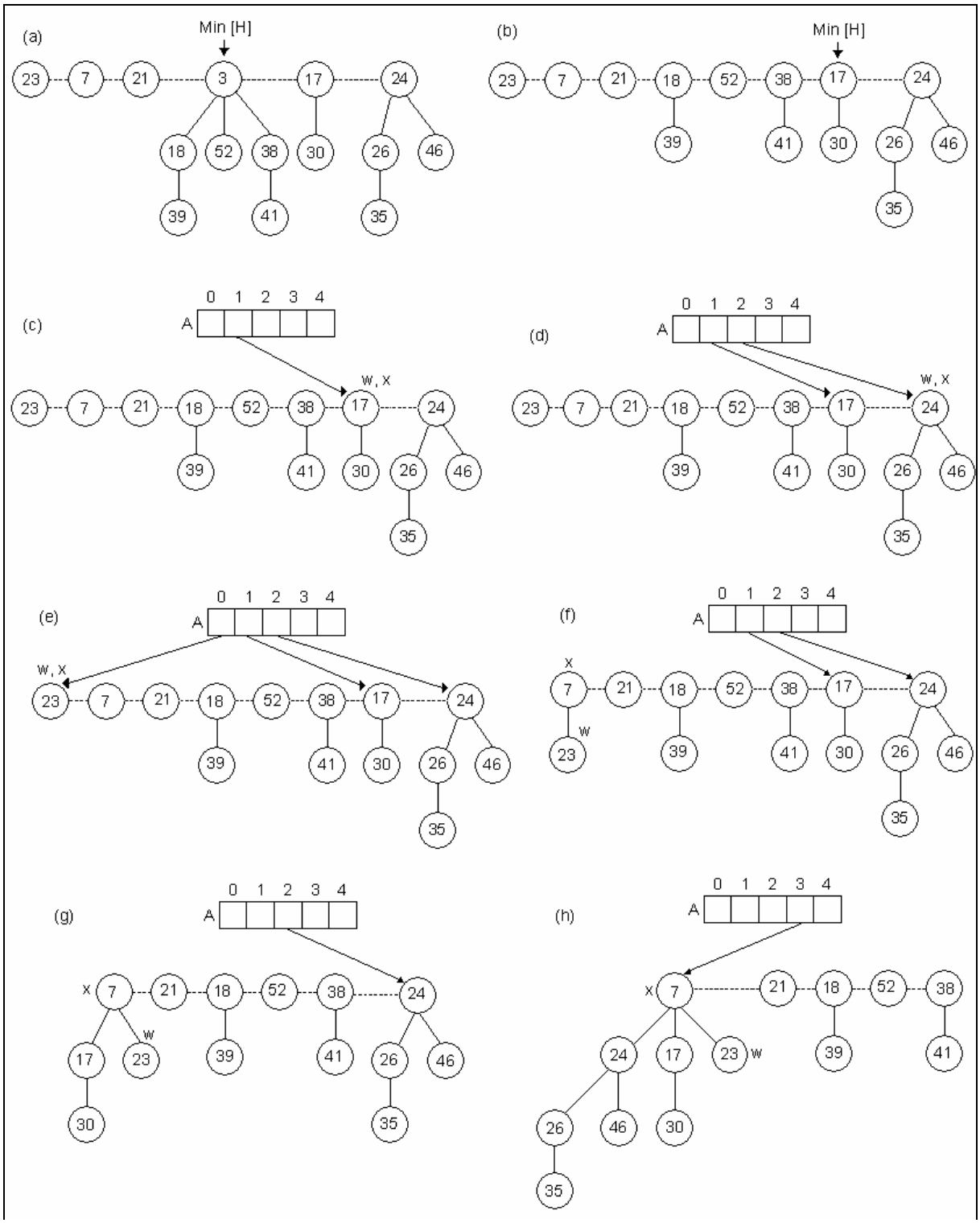
```
Ligar (H, y, x)
1. remove y from the root list of H
2. make y a child of x, incrementing degree[x]
3. mark[y] := false
```

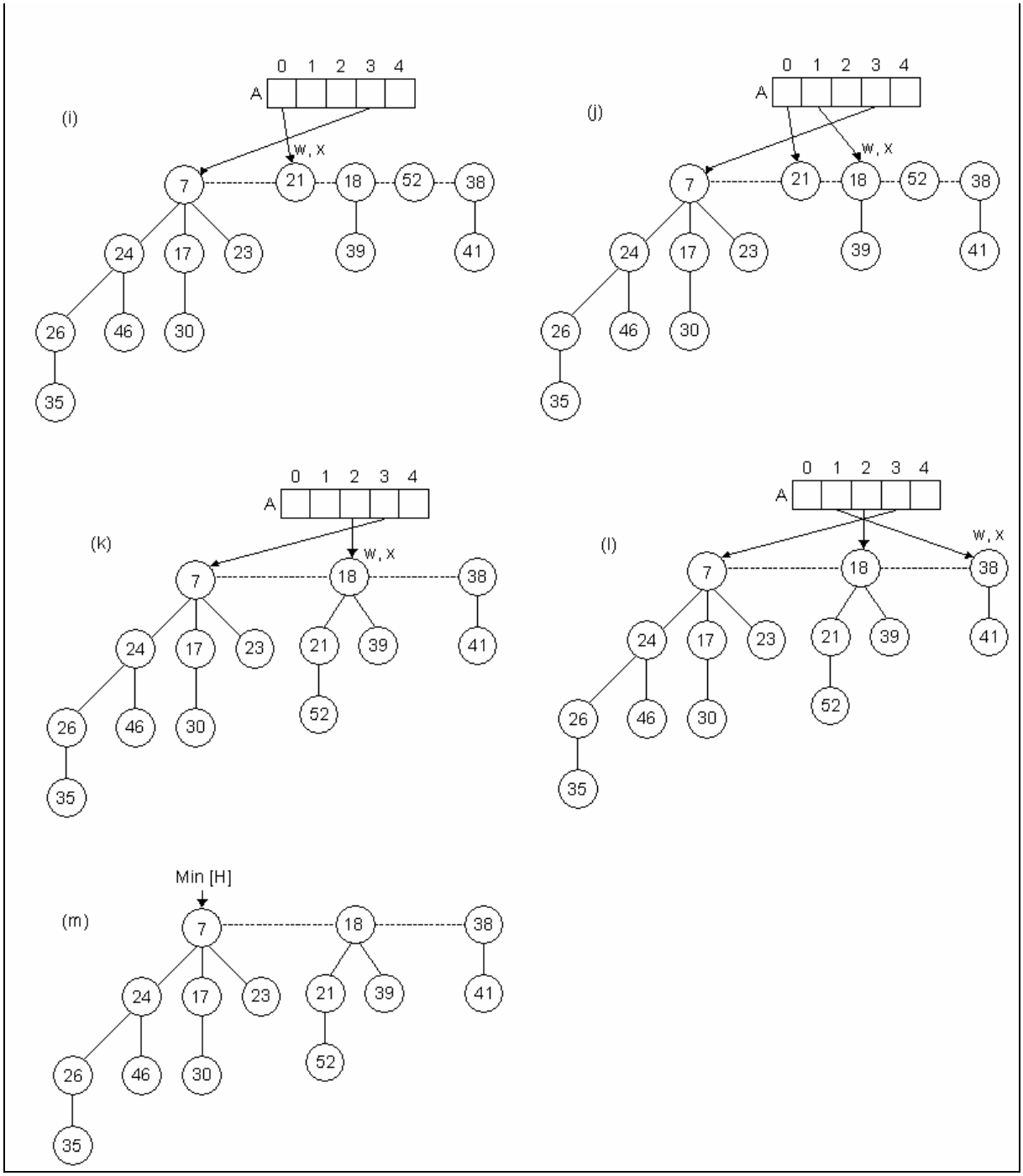
O algoritmo Consolidate usa um vetor auxiliar A[0..D(n(*H*))], onde D(n(*H*)), segundo Cormen (1989), é lg *n*.

As linhas 1-2 inicializam o vetor A. Nas linhas 3-13 são examinados todos os nodos da lista de raízes. O loop das linhas 6-12 liga os elementos com o mesmo grau, garantindo dessa forma, que todos os nodos da lista de raízes possuam graus diferentes. A linha 14 elimina a lista de raízes, que é reconstruída pelas linhas 15-19.

Um exemplo da operação ExtrairMínimo é mostrado na figura 16.

Figura 16 - Exemplo da operação ExtrairMínimo na lista de Fibonacci





Fonte: Cormen (1989).

A figura 16 (a) é a lista de *Fibonacci* original; (b) mostra a lista após o elemento mínimo *z* ser removido da lista de raízes e os seus filhos terem sido adicionados à lista de raízes; (c) – (e) mostram a lista após as 3 primeiras iterações do loop for das linhas 3-13 de Consolidate e os valores de *w* e *x* ao final de cada iteração. A lista de raízes é processada começando pelo nodo mínimo e seguindo os ponteiros *right* de cada nodo. Em (f) é mostrado

a lista após a primeira iteração do loop while. O nodo com chave 23 foi ligado ao nodo com o nodo com chave 7. Em (g), o nodo com chave 17 é ligado ao nodo com chave 7. Como não há nodo apontado por A[3], no final do loop for A[3] aponta para o nodo com chave 7. (i) – (l) mostra a lista após cada uma das 4 iterações do loop while; (m) mostra a lista de *Fibonacci* após a reconstrução da lista de raízes a partir do vetor A e a determinação do novo ponteiro ao nodo de valor mínimo.

3.4 DIMINUINDO UMA CHAVE

O Quadro 13 possui o algoritmo de DiminuirChave. Este algoritmo utiliza os algoritmos Cut e CascadingCut que estão descritos nos quadros 14 e 15, respectivamente.

Quadro 13 - Algoritmo DiminuirChave

```
DiminuirChave(H, x, y)
1.    if k > key[x] then
2.       error “nova chave é maior que a chave atual”
3.    key[x] := k
4.    y := p[x]
5.    if y <> nil and key[x] < key[y] then
6.       Cut(H, x, y)
7.       CascadingCut(H, y)
8.    if key[x] < key[min[H]] then
9.       min[H] := x
```

As linhas 1 à 3 garantem que a nova chave não é maior que a chave já existente e atribui a nova chave à *x*. Se *x* é uma raiz ou se *key*[*x*] >= *key*[*y*], onde *y* é pai de *x*, então não são necessárias mudanças na lista, conforme teste nas linhas 4-5.

Caso contrário, algumas alterações devem ser feitas. Inicialmente, “corta-se” (linha 6) a ligação entre *x* e o seu pai *y*, fazendo *x* uma raiz. Este algoritmo está descrito no quadro 14.

O atributo *mark* é utilizado para identificar a seguinte situação:

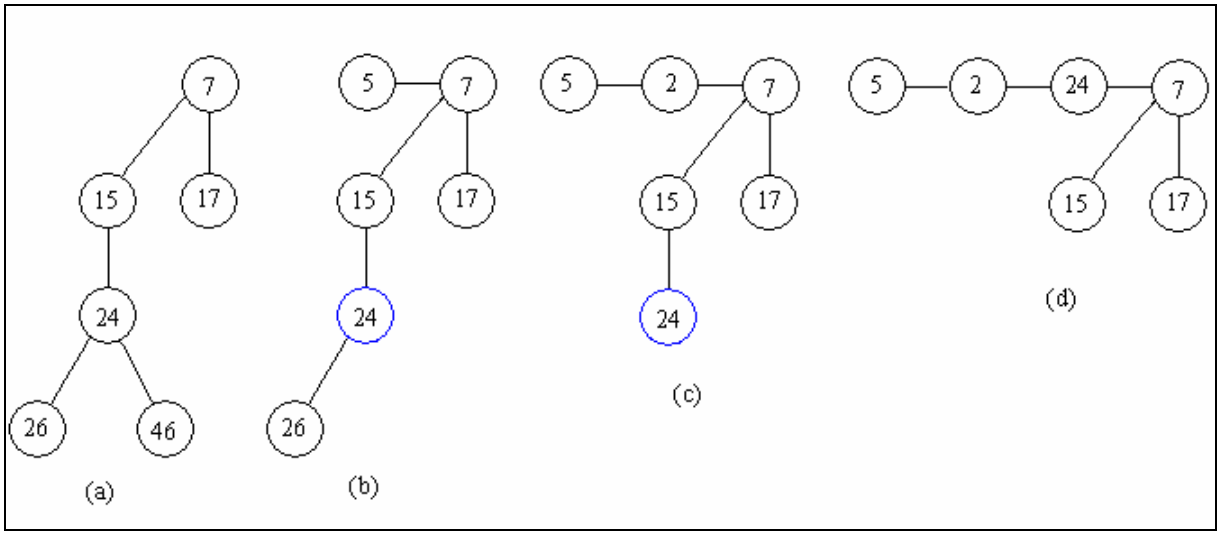
a) *x* era um nodo na lista de raízes;

b) *x* foi ligado a um nodo;

c) 2 filhos de *x* foram removidos por “cortes”.

Assim que o segundo filho de *x* é removido, *x* é “cortado” (elimina-se a ligação entre *x* e seu pai) de seu pai, adicionando *x* na lista de raízes. Para verificar se *x* é o segundo filho cortado do seu pai *y* desde que *y* foi ligado a um outro nodo, a linha 7 executa uma operação cascading-cut, descrito no quadro 15, em *y*. Se *y* é uma raiz, esta operação não faz nada. Se *y* está desmarcado (*mark*[*y*] = *false*), então *y* é marcado. Se *y* estiver marcado, *y* é “cortado” e cascading-cut se chama recursivamente para o pai de *y*. Por último, as linhas 8-9 atualizam o ponteiro ao nodo mínimo, se necessário. A figura 17 mostra um exemplo deste processo.

Figura 17 - Exemplo do Algoritmo DiminuirChave



Fonte: Baseado em Cormen (1989).

Neste exemplo, a situação inicial da lista é mostrada na figura 17 (a). Na figura 17 (b), o nodo com chave 46 é diminuído para 5, este nodo é inserido na raiz e o seu pai (nodo 24) é marcado (em azul). Na figura 17 (c), o nodo com chave 26 é diminuído para 2 e é inserido na lista raiz. Como o seu pai já está marcado (nodo 24) ele também é inserido na raiz, figura 17 (d).

Quadro 14 - Algoritmo Cut

```
Cut(H, x, y)
1. remove x from the child list of y, decrementing degree[y]
2. add x to the root list of H
3. p[x] := nil
4. mark[x] := false
```

Quadro 15 - Algoritmo CascadingCut

```
CascadingCut(H, y)
1. x := p[y]
2. if z <> nil then
3.    if mark[y] = false then
4.       mark[y] := true
5.    else Cut(H, y, z)
6.       CascadingCut(H, z)
```

3.5 DELETANDO UM NODO

O algoritmo para deletar um nodo da lista de *Fibonacci* pode ser visto no quadro 16. Este algoritmo assume que não existe nodo com o valor de chave $-\infty$.

Quadro 16 - Algoritmo Deletar

```
Deletar(H, x)
1. DiminuirChave(H, x, -∞)
2. ExtrairMinimo(H)
```

Para deletar um nodo simplesmente atribui-se a esse nodo o valor $-\infty$ e depois extrai-se da lista o nodo com o valor mínimo.

4 Desenvolvimento do Protótipo

Durante a pesquisa para o desenvolvimento do protótipo foram encontradas no mercado 2 ferramentas que podem ser consideradas correlatas.

A primeira ferramenta é chamada de *MaxLog*, que segundo Mesquita (2001), possui um investimento mínimo de 30 milhões de dólares e permite que embarcadores de cargas de baixo valor agregado (como soja, feijão e aço, por exemplo) entrem com os dados da entrega - tipo de carga, onde retirar e onde entregar, qual e o prazo máximo - para que as empresas de transporte ofereçam seus serviços.

A segunda ferramenta, segundo Yuri (2001), chama-se *Multiponto* e faz o planejamento on-line das melhores rotas para transportadoras com muitos pontos de paradas.

Não foi possível, entretanto, utilizar estas ferramentas, pois as mesmas são proprietárias e não estão disponíveis para uso de pessoas não autorizadas.

Neste capítulo será apresentado o desenvolvimento do protótipo deste trabalho.

4.1 ESPECIFICAÇÃO

O protótipo a ser desenvolvido neste trabalho visa atender a seguinte situação: uma transportadora possui caminhões com viagens agendadas. Ao retornar ao ponto de origem, constantemente os caminhões voltam vazios.

Deste modo, é possível, através do protótipo, fazer o cadastro de cargas que necessitam ser entregues, e das entregas, ou seja, das viagens de retorno disponíveis. Quando uma carga é cadastrada, o protótipo verifica se existe alguma entrega agendada que possa atender também a esta carga. Da mesma forma, quando uma entrega é cadastrada, verifica-se se existem cargas que podem ser atendidas por esta entrega. É possível ainda, no cadastro de uma entrega, informar se esta entrega pode sofrer um desvio, ou seja, desviar da rota de menor caminho entre a origem e destino da entrega para atender uma carga e qual será o valor máximo do desvio, de modo a não onerar demais a entrega.

Na modelagem do protótipo, cada possível ponto de parada é representado por um vértice e cada caminho entre os vértices e representado por uma aresta.

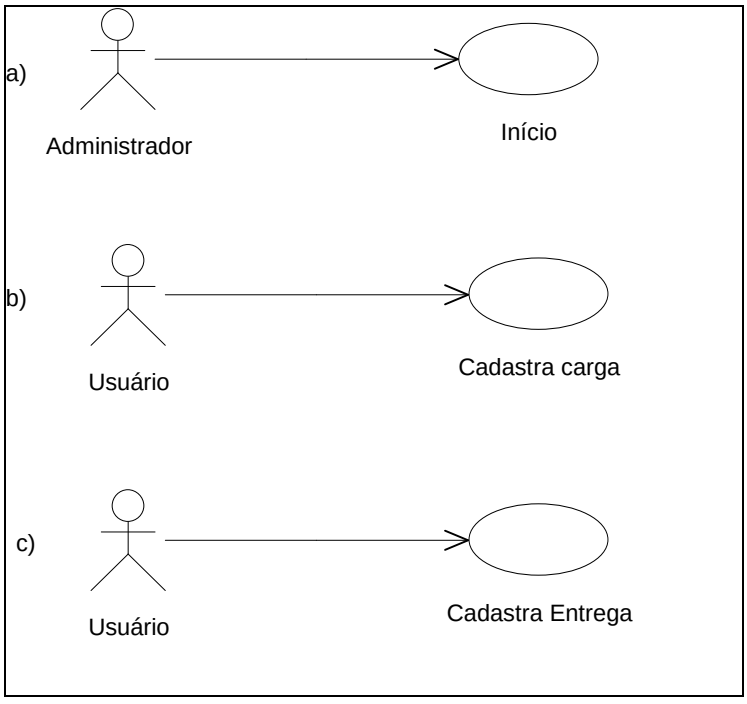
Para fazer a especificação deste protótipo foi utilizada uma metodologia orientada a objetos, a *Unified Modeling Language* (UML), usando como ferramenta o *Rational Rose*.

4.1.1 CASOS DE USO

O protótipo possui 3 casos de usos principais (existem casos de usos secundários, como cadastro de proprietário e de caminhão):

a) início: responsável pelo cálculo dos menores caminhos entre os vértices e pela carga das demais informações utilizadas pelo protótipo. Este caso de uso pode ser visto na figura 18 (a);
b) cadastra carga: responsável por fazer o cadastro da carga e verificar se existe uma entrega que atenda essa carga. Este caso de uso pode ser visto na figura 18 (b);
c) cadastra entrega: responsável por fazer o cadastro da carga e verificar se existem cargas que podem ser atendidas por esta entrega. Este caso de uso pode ser visto na figura 18 (c).

Figura 18 - Casos de uso



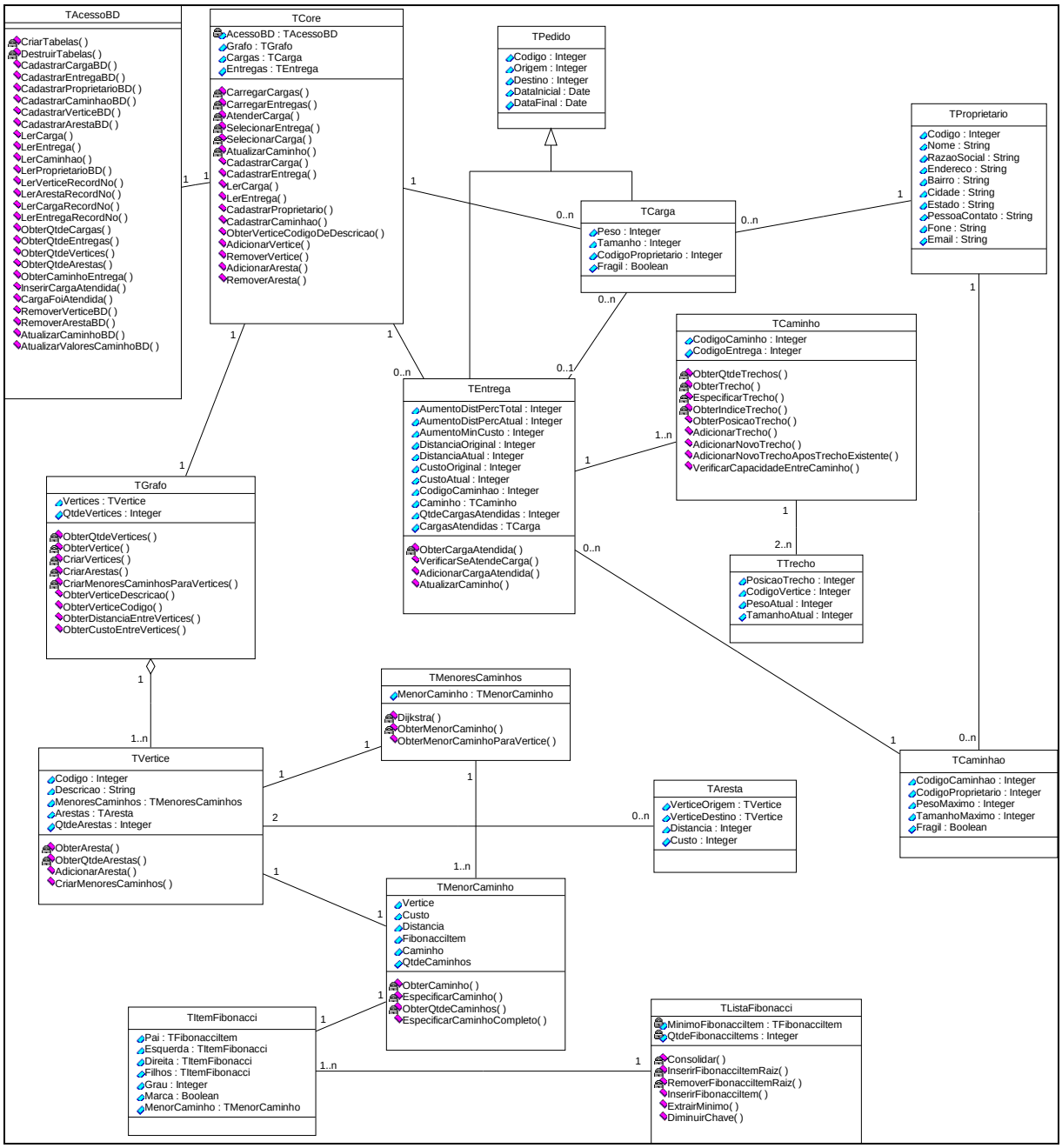
4.1.2 DIAGRAMA DE CLASSES

As classes utilizadas no protótipo são:

a) TCore: classe principal do protótipo que contém as cargas e entregas disponíveis e o grafo, além de fazer os cadastros de: carga, entrega, proprietário, caminhão, vértices e arestas;
b) TAcessoBD: responsável por fazer todo o acesso à base de dados do protótipo. Desta forma, nenhuma outra classe faz acesso a base de dados;
c) TPedido: contém informações comuns a uma carga e uma entrega;
d) TEntrega: especialização da classe TPedido. Contém as informações relativas à uma entrega. Uma entrega sempre está associada ao caminhão que fará essa entrega;
e) TCarga: especialização da classe TPedido. Esta classe contém as informações relativas a uma carga que deve ser entregue;
f) TCaminhao: representa um caminhão que fará uma entrega;
g) TProprietario: representa o proprietário de um caminhão ou de uma carga;
h) TGrafo: representa um grafo, tendo como atributo os vértices deste grafo;
i) TVertice: representa um vértice do grafo;
j) TAresta: representa uma aresta entre dois vértices;
k) TMenoresCaminhos: representa os menores caminhos entre um vértice e os demais vértices do grafo. Esta classe é responsável por fazer o cálculo do menor caminho entre este vértice e os demais vértices do grafo;
l) TMenorCaminho: representa o menor caminho entre dois vértices;
m) TCaminho: representa o caminho completo de uma entrega;
n) TTrecho: representa os trechos do caminho de uma entrega;
o) TListaFibonacci: representa uma lista de *Fibonacci*. Esta classe é utilizada no cálculo do menor caminho entre os vértices;
p) TItemFibonacci: representa um nodo na lista de *Fibonacci*.

O Diagrama de Classes está demonstrado na figura 19.

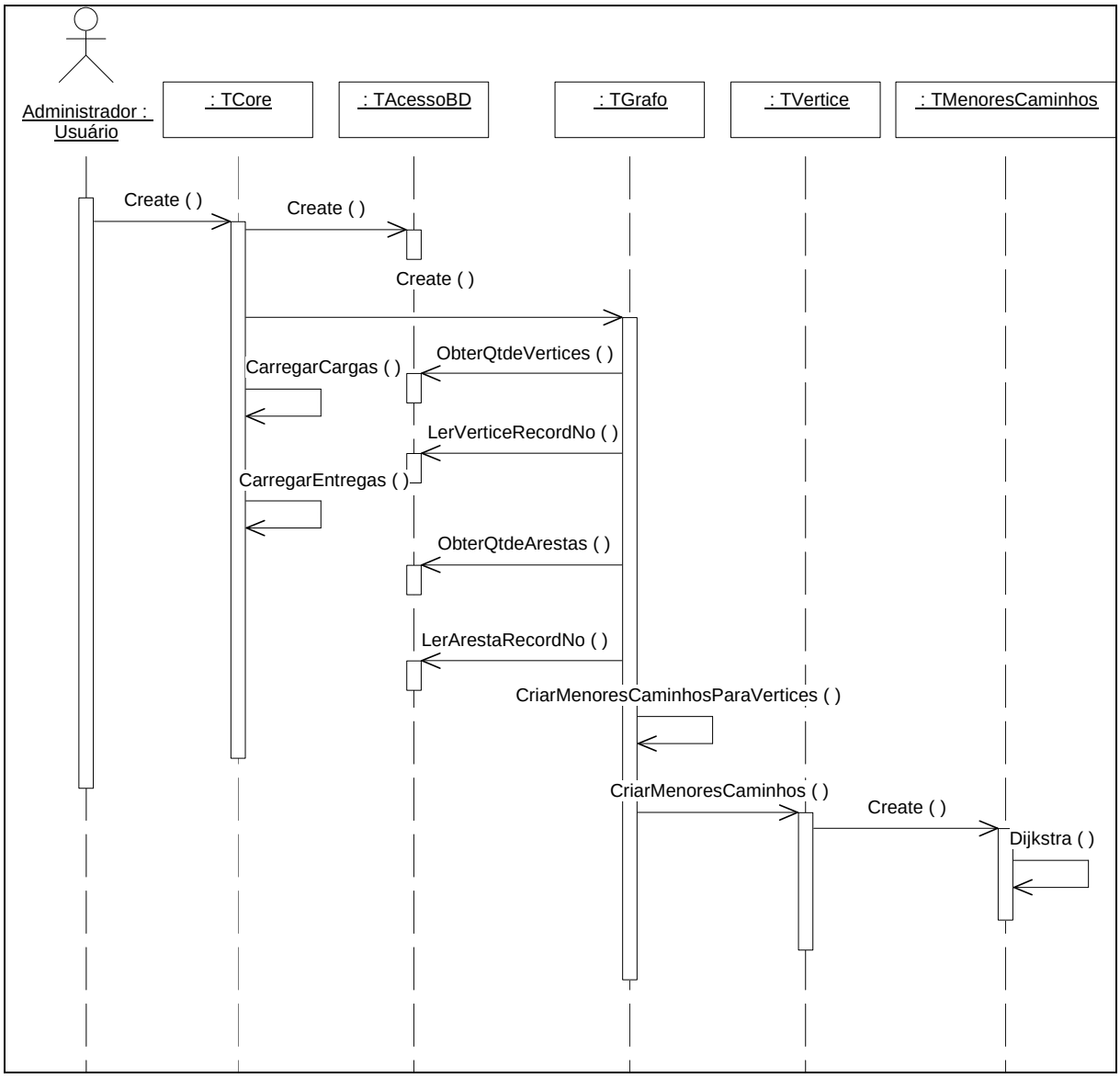
Figura 19 - Diagrama de Classes



4.1.3 DIAGRAMAS DE SEQUÊNCIA

O diagrama de seqüência “Início” está representado na figura 20.

Figura 20 - Diagrama de seqüência "Início"

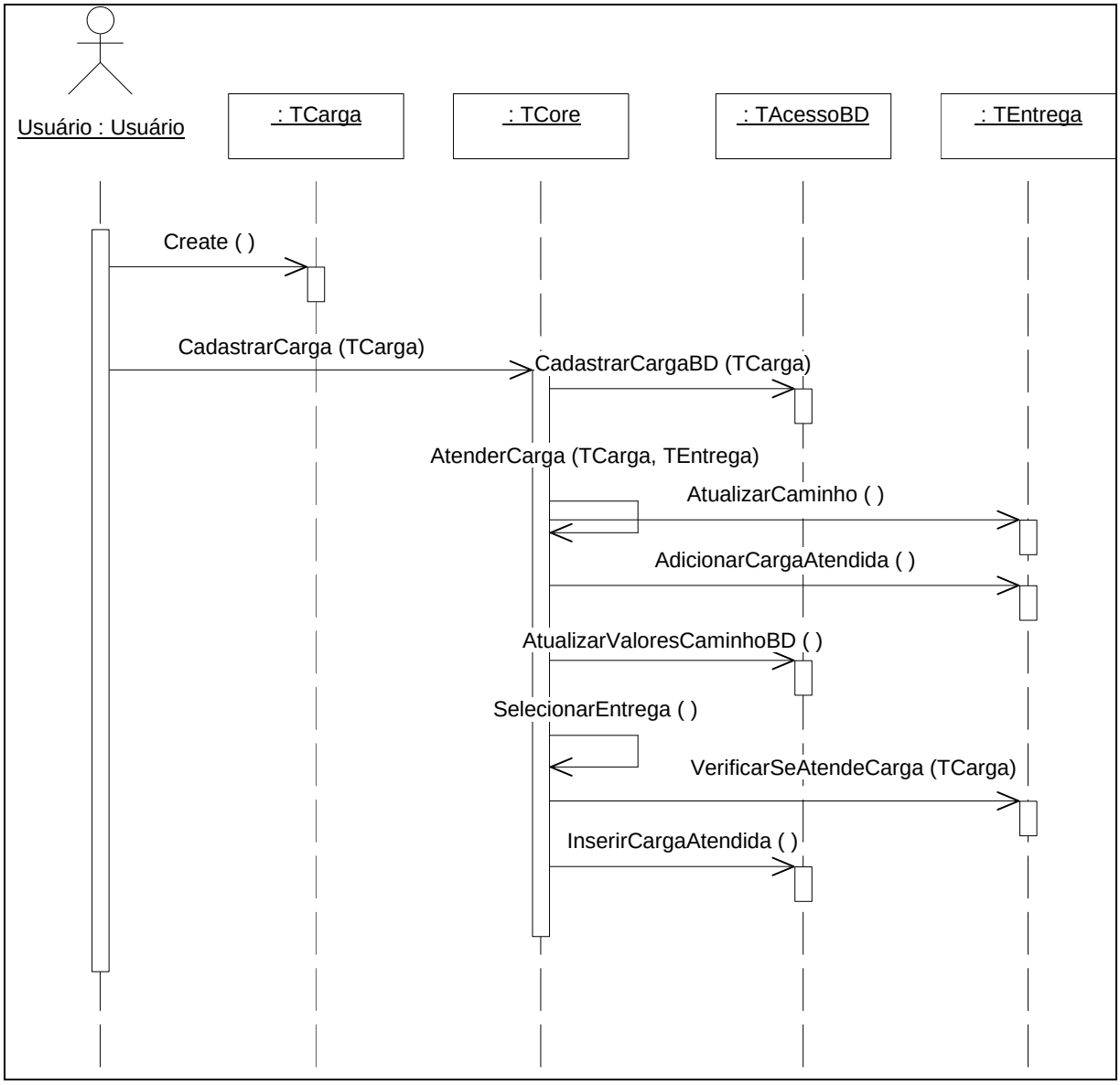


Este diagrama de seqüência é executado sempre que o protótipo é inicializado e quando algum vértice ou aresta é incluído no grafo. É instanciado um objeto da classe *TCore*. Este objeto instancia um objeto *TAcessoBD* para fazer o acesso à base de dados para carregar o grafo, as cargas não atendidas, as entregas disponíveis e um objeto *TGrafo* para representar o grafo. Este objeto cria os objetos vértices e arestas que compõem o grafo e executa o seu método *CriarMenoresCaminhosParaVértices*, que apenas chama o método *CriarMenoresCaminhos* de cada vértice criado. Este método, por sua vez, cria um objeto *TMenoresCaminhos* para este vértice, que executa o método *Dijkstra*, que é a implementação

do algoritmo de *Dijkstra* em conjunto com as listas de *Fibonacci*, para calcular o menor caminho entre este vértice e todos os outros vértices do grafo. O código do algoritmo de *Dijkstra* pode ser visto no Anexo I.

O diagrama de seqüência “Cadastra carga” está demonstrado na figura 21.

Figura 21 - Diagrama de seqüência "Cadastra carga"

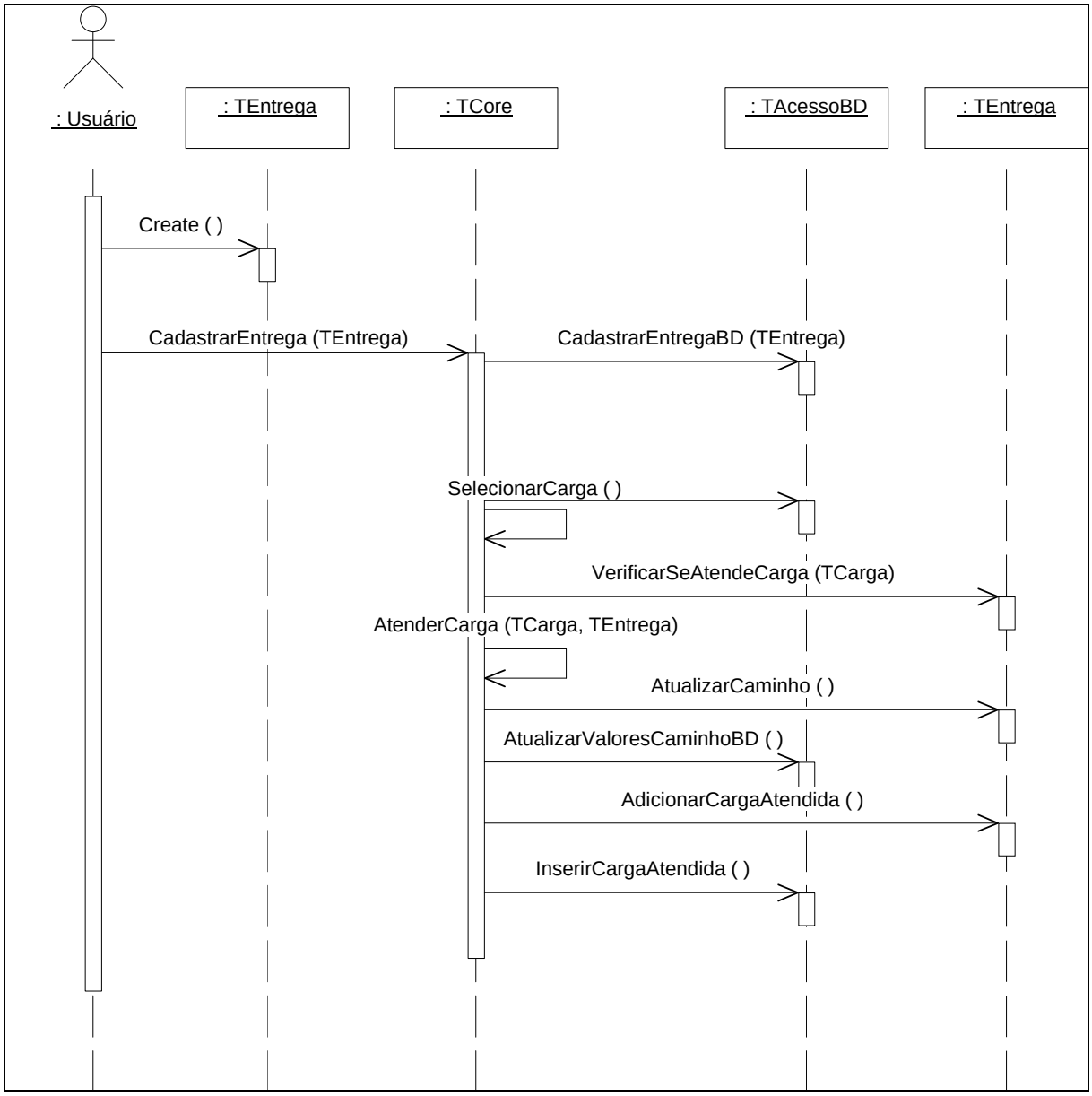


Este diagrama de seqüência instancia um objeto da classe *TCarga* que é carregado com os dados da carga a ser cadastrada e é passado como parâmetro para o método *CadastrarCarga* do objeto *TCore* criado na inicialização do protótipo.

Ao cadastrar a carga, verifica-se na lista de entregas disponíveis se uma entrega pode atender esta carga. Se não houver nenhuma entrega disponível, esta carga será adicionada na lista de cargas não atendidas aguardando que uma nova entrega possa atendê-la.

O diagrama de seqüência “Cadastra entrega” está representado na figura 22.

Figura 22 - Diagrama de seqüência "Cadastra entrega"



Este diagrama de seqüência instancia um objeto da classe *TEntrega* que é carregado com os dados da entrega a ser cadastrada e é passado como parâmetro para o método *CadastrarEntrega* do objeto *TCore* criado na inicialização do protótipo.

Ao cadastrar a entrega, verifica-se na lista de cargas não atendidas se uma ou mais cargas podem ser atendidas por esta entrega. Se a entrega ainda tiver capacidade de entregar alguma carga, esta entrega será adicionada na lista de entregas disponíveis.

4.2 IMPLEMENTAÇÃO

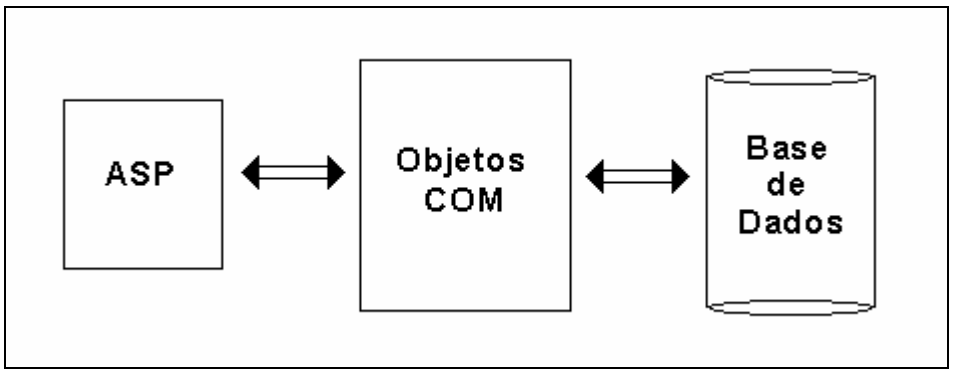
Para a implementação o protótipo foi dividido em 2 partes principais:

a) objetos COM: são responsáveis pelo cálculo de menor caminho, pela verificação se a carga ou entrega pode ser atendida e pelo acesso à base de dados. Esta parte do protótipo não interage com o usuário. Esta parte do protótipo foi desenvolvida utilizando a ferramenta *Borland Delphi 5.0*;

b) páginas HTML: são responsáveis pela interface com o usuário. Esta parte do protótipo não faz nenhum cálculo de menor caminho nem verificação se a carga ou entrega pode ser atendida, nem faz acesso à base de dados. Estas páginas utilizam ASP para construir e utilizar os objetos COM.

A interação entre os componentes do protótipo ocorre da seguinte forma: as páginas, através do uso de ASP, criam os objetos COM que fazem os cadastros necessários, executam os processos de cálculo e retornam os resultados. Cabe ressaltar que a camada em HTML do protótipo cuida apenas da interface com o usuário, sendo responsabilidade da camada em COM a execução dos processos e o acesso à base de dados. O fluxo macro do protótipo pode ser visto na figura 23.

Figura 23 - Fluxo macro do protótipo



A base de dados foi construída utilizando a ferramenta *Microsoft Access 2000*.

As classes que foram transformadas em objetos COM foram: TCore, TCarga, TPedido, TProprietario e TCaminhao.

4.3 Operacionalidade da Implementação

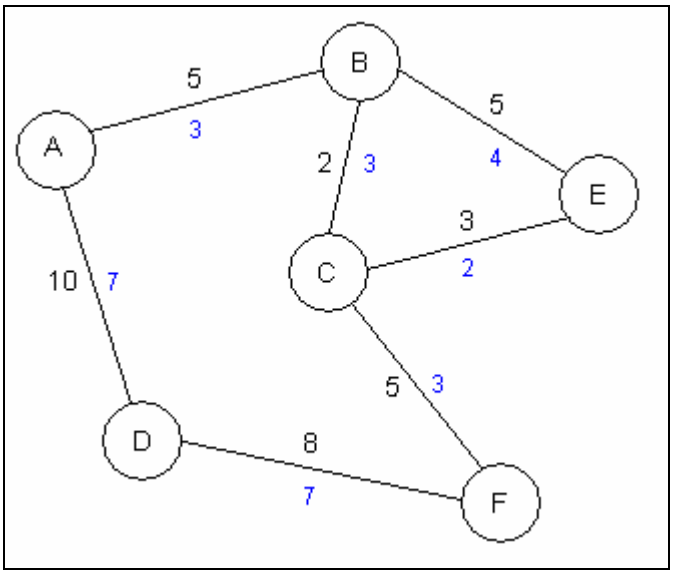
O grafo utilizado no cálculo do menor caminho entre os vértices possui 2 valores associados com cada aresta: a distância, que representa a distância entre os 2 vértices que são ligados por esta aresta e o custo financeiro, que representa o valor financeiro da aresta, ou seja, quanto a transportadora cobra para fazer o percurso entre os vértices ligados por esta aresta. O custo financeiro existe para atender a situação de arestas que possuem a mesma distância mas que podem ter custo financeiro diferentes, dependendo das condições da rodovia que a aresta representa. O cálculo de menor caminho sempre é feito baseado na distância da aresta. O custo financeiro, como será visto adiante, é utilizado para verificar se uma entrega pode ter sua rota alterada.

Uma limitação do protótipo é não fazer qualquer otimização da rota depois que esta é alterada. Quando uma rota é alterada para atender uma carga, após chegar ao destino da carga, volta-se ao ponto de origem do desvio da rota. Isto é feito pois ainda pode haver alguma carga pendente após o desvio.

Os passos descritos no caso de uso “Início” são executados apenas quando o protótipo é inicializado pelo administrador, assim, quando um usuário fazer algum cadastro de carga ou de entrega, os menores caminhos entre os vértices já estarão calculados.

Para um melhor entendimento sobre a execução do protótipo será demonstrada a execução dos casos de uso “Cadastra carga” e “Cadastra entrega”. Os casos de uso a seguir são executados utilizando como exemplo o grafo demonstrado na figura 24. As distâncias das arestas estão representadas em preto e os custos financeiros das arestas deste grafo estão representados em azul.

Figura 24 - Grafo utilizado nos casos de uso



As cargas já cadastradas estão descritas na tabela 3.

Tabela 3 - Cargas cadastradas

Código	**Origem**	**Destino**	**Data Inicial**	**Data Final**	**Peso (kg)**	**Tamanho (m^2)**	**Frágil**
59	A	E	10/07/2001	12/07/2001	3	2	Não
60	A	B	11/07/2001	12/07/2001	4	5	Não
61	C	E	11/07/2001	12/07/2001	3	2	Não

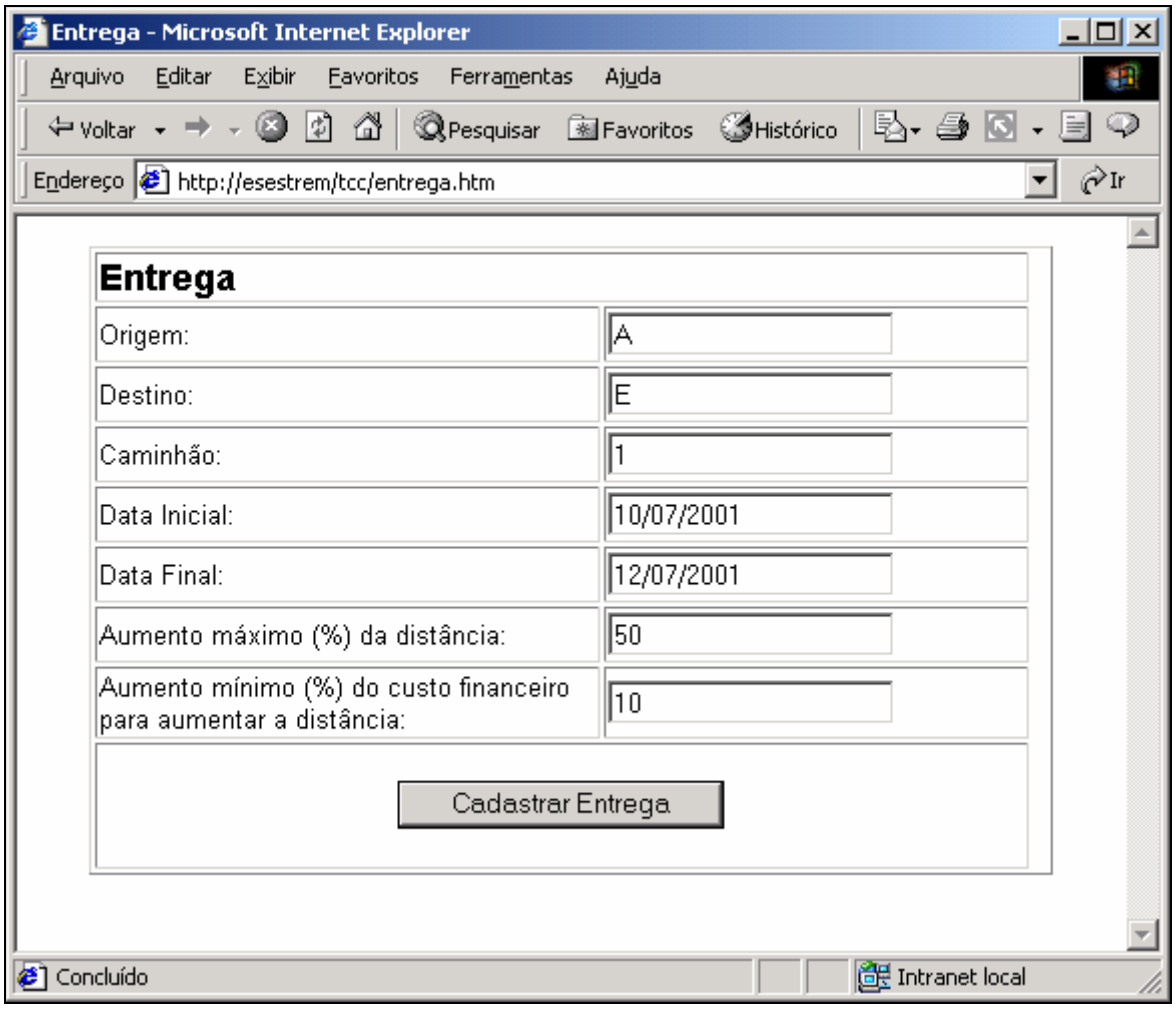
A tela inicial do protótipo pode ser visto na figura 25.

Figura 25 - Tela inicial do protótipo

4.3.1 CASO DE USO “CADASTRAR ENTREGA”

O caso de uso “Cadastrar entrega” ocorre quando o proprietário de um caminhão faz o cadastro de uma entrega que este caminhão irá efetuar acessando, através de um *browser*, a tela para informar dados sobre a entrega a ser efetuada. Esta tela bem como os dados da entrega que está sendo cadastrada podem ser vistas na figura 26.

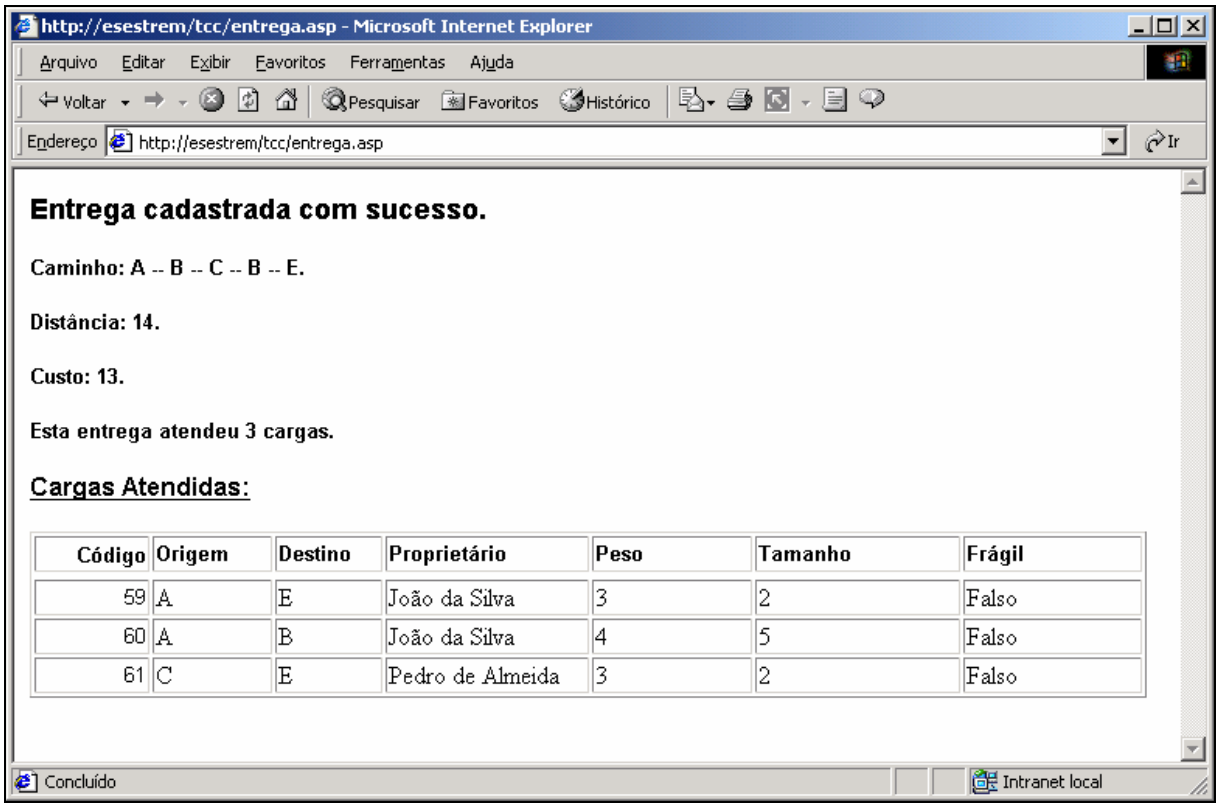
Figura 26 - Tela de cadastro de entrega do protótipo



Ao cadastrar uma entrega, o transportador pode informar em “Aumento máximo (%) da distância” qual a distância máxima que a rota da entrega pode aumentar para atender uma carga que não esteja na rota do menor caminho da entrega. Pode informar também, em “Aumento mínimo (%) do custo para aumentar a distância”, qual o valor mínimo que o custo financeiro da entrega deve aumentar para que a rota seja alterada para atender uma carga que não está no caminho original da entrega.

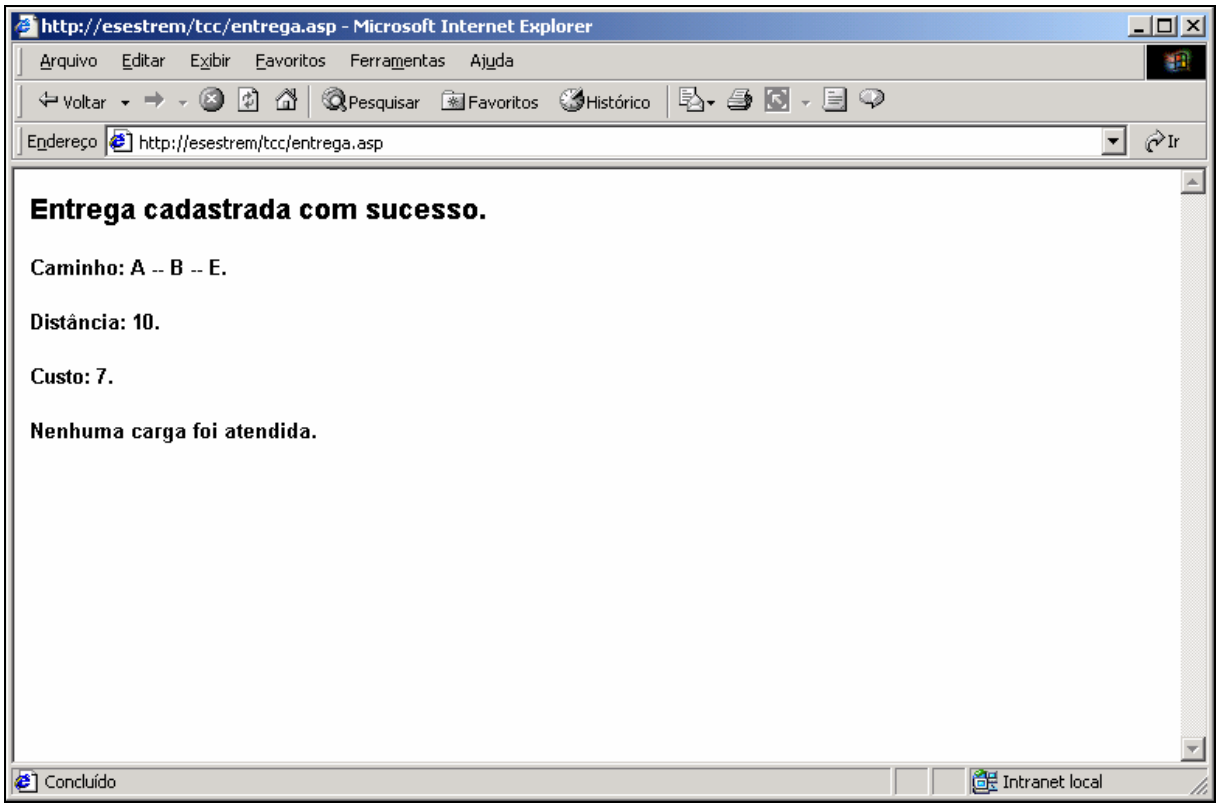
Neste exemplo, a entrega atende as cargas de código 59, 60 e 61 já cadastradas. A tela mostrada na figura 27 mostra ao usuário estas informações. A carga 61 não está na rota original da entrega (A – B – E), mas como esta entrega pode ter um aumento da distância de até 50%, foi possível atender também a carga 61.

Figura 27 - Tela de entrega cadastrada



Ao cadastrar uma nova entrega com as mesmas informações da figura 26 nenhuma carga será atendida, pois as cargas disponíveis já foram atendidas. O resultado pode ser visto na figura 28.

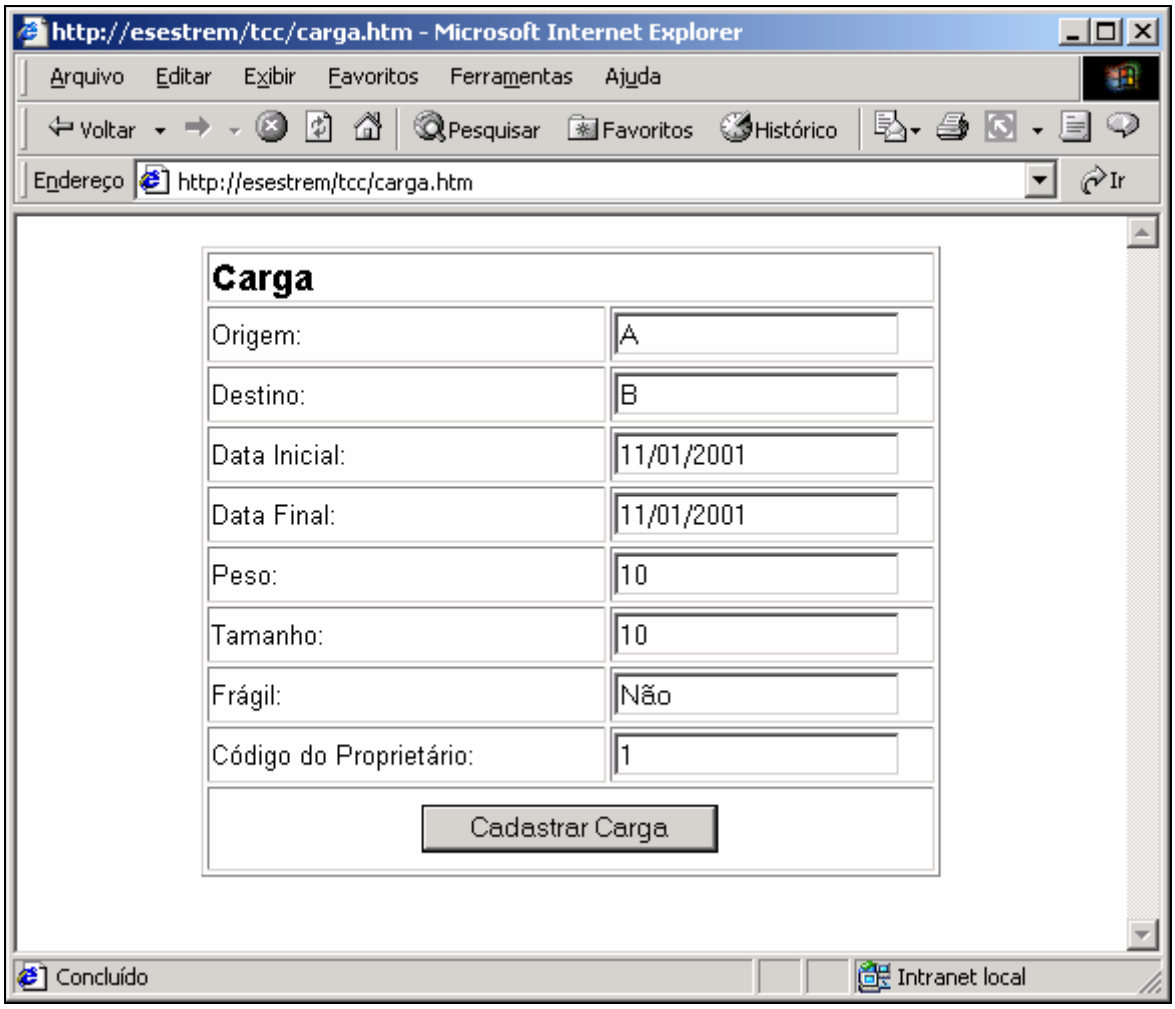
Figura 28 - Tela de entrega cadastrada



4.3.2 CASO DE USO “CADASTRAR CARGA”

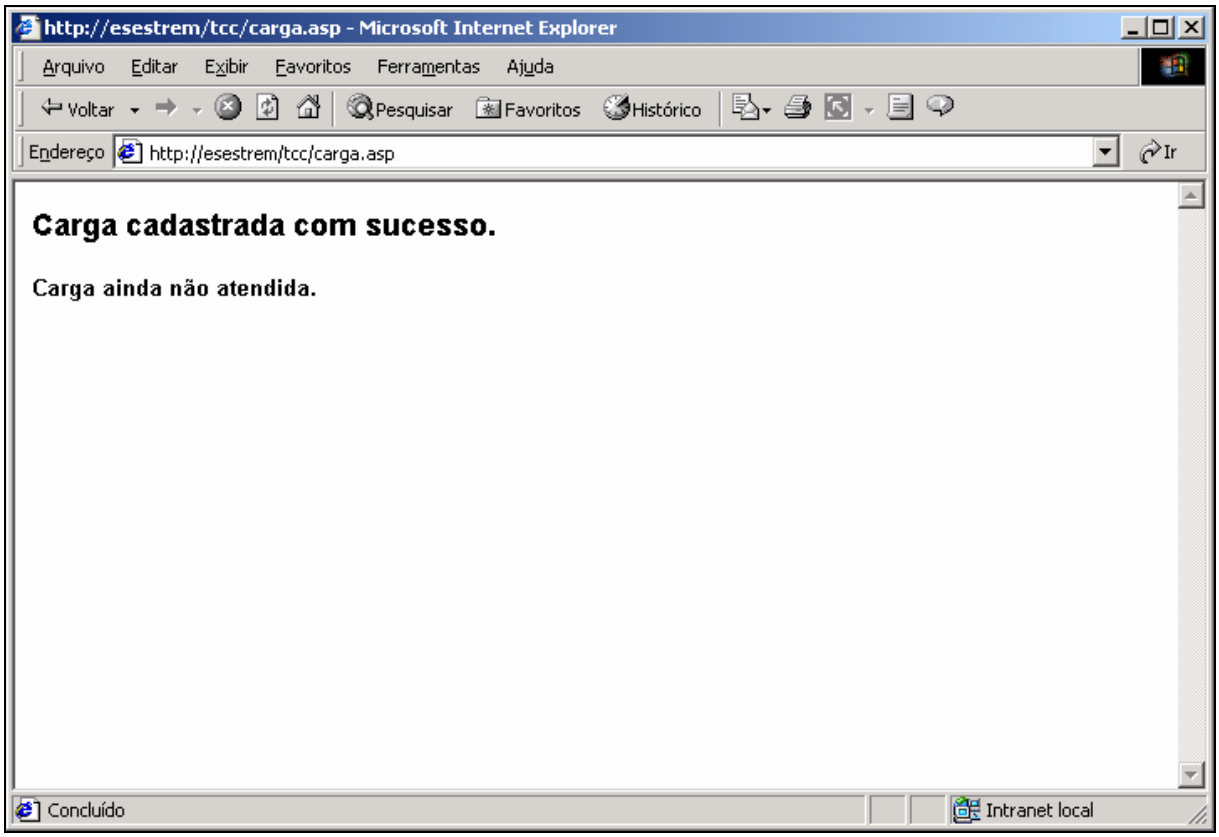
O caso de uso “Cadastrar carga” ocorre quando o proprietário de uma carga faz o cadastro desta carga acessando, através de um *browser*, a tela para informar dados sobre a carga a ser entregue. Esta tela bem como os dados da carga que está sendo cadastrada podem ser vista na figura 29. O código executado pelo botão “Cadastrar carga” pode ser visto no Anexo I.

Figura 29 - Tela de cadastro de carga



Neste exemplo, não há entrega disponível para atender a carga. A tela mostrada na figura 30 mostra ao usuário estas informações.

Figura 30 - Tela de carga não atendida



5 CONSIDERAÇÕES FINAIS

Este capítulo irá apresentar as conclusões com relação ao trabalho desenvolvido como também as dificuldades encontradas e sugestões para trabalhos futuros nesta área.

5.1 CONCLUSÕES

O objetivo principal deste trabalho, que é implementar um protótipo que auxilie a distribuição de cargas, foi atendido.

Foram estudados 2 algoritmos para determinar os menores caminhos entre vértices dentre aqueles que foram encontrados durante o desenvolvimento do trabalho: *Dijkstra* e *Bellman-Ford*. O algoritmo de *Dijkstra* com listas de *Fibonacci* mostrou-se bastante eficiente. As listas de *Fibonacci* contribuem para a otimização do algoritmo de *Dijkstra*, porém, sua implementação pode ser considerada complexa.

O protótipo desenvolvido pode auxiliar as empresas a reduzir seus custos e aumentar sua eficiência, porém, vale lembrar que não foram realizados testes com dados de uma situação real.

Durante o desenvolvimento do protótipo foram utilizadas 3 ferramentas: *Rational Rose* para fazer a especificação, *Borland Delphi* para construir as classes e transformá-los em objetos COM e ASP para fazer a camada de interface e instanciar os objetos COM criados. Cabe ressaltar a reusabilidade de código que a utilização de tecnologias como COM proporcionam possibilitando que uma ferramenta de desenvolvimento utilize objetos desenvolvidos com outras ferramentas.

5.2 DIFICULDADES ENCONTRADAS

As maiores dificuldades encontradas foram na implementação do algoritmo da lista de *Fibonacci* e na comunicação entre a camada em HTML e a camada dos objetos COM, devido aos poucos tipos de dados que o VBScript suporta.

5.3 EXTENSÕES

Pode-se estudar outros algoritmos de menor caminho e/ou utilizar outros tipos de filas de prioridades, como a lista binomial, para fazer uma comparação de tempo da execução dos algoritmos em uma situação real.

Pode-se estudar também formas para otimizar o caminho da entrega depois que a rota for alterada para atender uma carga.

Seria interessante fazer um estudo para planejar entregas que atendam as cargas cadastradas, levando em consideração não só a distância, mas também o custo financeiro e as datas de entregas.

Outra possibilidade é a criação de bibliotecas de algoritmos de menor caminho, para serem utilizadas em qualquer categoria de problemas.

ANEXO I

Neste anexo encontra-se o algoritmo de Dijkstra implementado utilizando listas de Fibonacci e o código executado pelo botão “Cadastrar carga” do protótipo:

a) algoritmo Dijkstra com listas de Fibonacci:

```
procedure TMenoresCaminhos.Dijkstra( aVerticeOrigem: Integer; aGrafo: TGrafo );
var
  I: Integer;
  W: Integer;
  xMenorCaminho: TMenorCaminho;
  xFibonacciHeap: TFibonacciHeap;
  xMFibonacciItem: TFibonacciItem;
  xAMenorCaminho: TMenorCaminho;
  xCurrentVertice: TVertice;
begin
  xFibonacciHeap := TFibonacciHeap.Create;
  try
    // for all vertices v
    for I := 0 to (aGrafo.QtdeVertices -1) do
    begin
      xCurrentVertice := aGrafo.Vertices[I];
      xMenorCaminho := TMenorCaminho.Create;
      xMenorCaminho.Vertice := xCurrentVertice;
      // if v = s then
      if (aGrafo.Vertices[I].Codigo = aVerticeOrigem) then
        // v.Cost = 0
        xMenorCaminho.Distancia := 0
      else
      // else v.Cost = cInfinity
        xMenorCaminho.Distancia := cInfinity;

      // Insert v into H
      xFibonacciHeap.InsertFibonacciItem(xMenorCaminho);

      FMenorCaminho.Add(xMenorCaminho);
    end;

    while (xFibonacciHeap.Count > 0) do
    begin
      // M := ExtractMin(H)
      xMFibonacciItem := xFibonacciHeap.ExtractMin;
      // for each vertex attached to M
      for I := 0 to (xMFibonacciItem.MenorCaminho.Vertice.QtdeArestas -1) do
      begin
        xAMenorCaminho := ObterMenorCaminhoParaVertice(
         xMFibonacciItem.MenorCaminho.Vertice.Arestas[I].VerticeDestino.Codigo);
        // w := cost of edge from M to A
        W := xMFibonacciItem.MenorCaminho.Vertice.Arestas[I].Distancia;
        // if (M.Cost + w < A.Cost)
        if (xMFibonacciItem.MenorCaminho.Distancia + W < xAMenorCaminho.Distancia) then
        begin
          // DecreaseKey(A, M.Cost + w)
          xFibonacciHeap.DecreaseKey(xAMenorCaminho.FibonacciItem,
xMFibonacciItem.MenorCaminho.Distancia + W, xMFibonacciItem.MenorCaminho.Custo +
xMFibonacciItem.MenorCaminho.Vertice.Arestas[I].Custo);
          xAMenorCaminho.EspecificarCaminhoCompleto(xMFibonacciItem.MenorCaminho);
        end;
      end;
    end;
  finally
    xFibonacciHeap.Free;
  end;
end;
```

b) código do botão Cadastrar Carga:

```
<%
Set xCarga = Server.CreateObject("ObjCarga.TObjCarga")

Dim xDescricaoOrigem
Dim xDescricaoDestino
```

```
Dim xCodigoOrigem
Dim xCodigoDestino
Dim xCodigoEntrega

xCarga.Inicializar

xDescricaoOrigem = Request.Form("edCargaOrigem")
xDescricaoDestino = Request.Form("edCargaDestino")

'Session("Core"): objeto core já criado
'pega o código do vértice a partir da descrição
Session("Core").ObterVerticeCodigoDeDescricao xDescricaoOrigem, xCodigoOrigem
xCarga.Origem = xCodigoOrigem

Session("Core").ObterVerticeCodigoDeDescricao xDescricaoDestino, xCodigoDestino
xCarga.Destino = xCodigoDestino

xCarga.DataInicial = Request.Form("edCargaDataInicial")
xCarga.DataFinal   = Request.Form("edCargaDataFinal")
xCarga.Fragil      = Request.Form("edCargaFragil")
xCarga.CodigoProprietario = Request.Form("edCargaCodigoProprietario")
xCarga.Peso = Request.Form("edCargaPeso")
xCarga.Tamanho = Request.Form("edCargaTamanho")

Session("Core").CadastrarCarga xCarga, xCodigoEntrega

Response.Write "<p><font face=Arial size=4><b>Carga cadastrada com sucesso.</b></font></p>"

If (xCodigoEntrega = -1) then
  Response.Write "<p><font face=Arial size=2><b>Carga ainda não atendida.</b></font></p>"
Else
  Response.Write "<p><font face=Arial size=2><b>Carga atendida.</b></font></p>"
End If
xCarga.Finalizar
xCarga = Nothing
%>
```

REFERÊNCIAS BIBLIOGRÁFICAS

ALVARENGA, Antonio Carlos; NOVAES, Antonio Galvão. **Logística aplicada**: suprimento e distribuição física. São Paulo: Pioneira, 1994.

BALLOU, Ronald H. **Logística empresarial**: transportes, administração de materiais e distribuição física. São Paulo: Atlas, 1993.

BOAVENTURA NETTO, Paulo Osvaldo. **Teoria e modelos de grafos**. São Paulo: Edgard Blücher, 1979.

BOYER, John. The fibonacci heap. **Dr. dobb's journal**, San Mateo, v. 22, n. 261, p. 106-108, jan. 1997.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. **Introduction to algorithms**. Massachusetts: McGrahl-Hill, 1989.

FURTADO, Antonio Luiz. **Teoria dos grafos**: algoritmos. Rio de Janeiro: Livros Técnicos e Científicos, 1973.

HEINZLE, Roberto. **HEINZLE, Roberto**. Disponível em <http://www.inf.furb.rct-sc.br/~heinzle>. Acesso em 21 mai. 2001.

KRUSE, Robert L. **Data structures and program design**. New Jersey: Prentice-Hall, 1994.

LAFORE, Robert. **Aprenda em 24 horas estrutura de dados e algoritmos**. Rio de Janeiro: Campus, 1999.

MESQUITA, Renata. **Estréia portal de logística para carregamentos pesados**. Disponível em: <http://www2.uol.com.br/info/infonews/102000/05102000-3.shl>. Acesso em: 21 mai. 2001.

NAZÁRIO, P. Intermodalidade: importância para a logística e estágio atual no Brasil. In: FLEURY, P.F.; WANKE, P.; FIGUEIREDO K.F. (Org.). **Logística empresarial:** a perspectiva brasileira. São Paulo: Editora Atlas, 2000. p. 126.

PEREIRA, Charles. **Implementação de heurística para determinação do caminho de menor custo**. 1999. 57 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

RABUSKE, Marcia Aguiar. **Introdução a teoria dos grafos**. Florianópolis: Ed. da UFSC, 1992.

SZWARCFITER, Jayme Luiz. **Grafos e algoritmos computacionais**. Rio de Janeiro: Campus, 1984.

SEDGEWICK, Robert. **Algorithms**. Princeton: Addison-Wesley, 1988.

YURI, Flávia. **Apontador cria serviço de multirrotas para logística**. Disponível em: <http://www2.uol.com.br/info/aberto/infonews/052001/07052001-19.shl>. Acesso em: 21 mai. 2001.