ANÁLISE COMPARATIVA DE ALGORITMOS EFICIENTES PARA O PROBLEMA DE CAMINHO MÍNIMO

ANÁLISE COMPARATIVA DE ALGORITMOS EFICIENTES PARA O PROBLEMA DE CAMINHO MÍNIMO

Jorge von Atzingen
Cláudio Barbieri da Cunha
Francisco Yastami Nakamoto
Fábio Rogério Ribeiro
André Schardong
Escola Politécnica, Universidade de São Paulo

RESUMO
Motivados por novas aplicações práticas que ocorrem em logística e transportes no âmbito de sistemas de apoio à decisão (SAD) em tempo real apoiados em Sistemas de Informações Geográficas (SIG's), neste artigo apresenta-se uma análise comparativa entre algoritmos eficientes aplicados ao problema de caminho mínimo (PCM). O PCM consiste em encontrar um caminho entre dois nós de uma rede, de tal forma que o custo total seja minimizado; no caso de aplicações em tempo real, respostas têm que ser obtidas em poucos segundos. Neste trabalho são comparados cinco algoritmos eficientes para resolver o problema proposto: *Dijkstra*, *Radix Heap*, *Heap Binomial*, *L-Threshold* e o recente *Dijkstra com Heurística*. O objetivo principal é avaliar o seu desempenho, testando-os com dados reais de quatro malhas viárias de grande porte (i.e. com milhões de arcos). Os resultados obtidos permitem avaliar a eficiência de cada um dos algoritmos, com destaque para o Dijkstra com Heurística.

ABSTRACT
Motivated by novel practical applications that arise in logistics and transportation in the context of real-time GIS-based decision support systems (DSS), in this paper we present a comparative analysis between efficient algorithms applied to the shortest path problem (SPP). The SPP consists of finding a path between two nodes of the network, in such a way that the total cost is minimized; in real-time applications a solution must be reached within a few seconds. In this paper five efficient algorithms are compared: Dijkstra, Radix Heap, Binomial Heap, L-Threshold and the new Heuristic Dijkstra. The main objective is to evaluate the performance of each algorithm for large-scale transportation networks (i.e., corresponding to millions of edges). The results evidence the efficiency of algorithm, especially the Heuristic Dijkstra.

1. INTRODUÇÃO

Dada uma rede (ou malha), composta de nós (ou vértices) e arcos (ou ligações) entre esses nós, o problema de caminho mínimo pode ser genericamente enunciado como o de encontrar o menor caminho entre dois nós dessa rede. Por exemplo, considerando-se um conjunto de cidades de uma determinada região como vértices, as vias que ligam essas cidades podem ser representadas como os arcos dessa rede. O caminho mínimo entre duas cidades consiste na seqüência de vias que resulta na menor distância. Não necessariamente o "menor" está associado à menor distância total percorrida; o conceito é mais genérico, e relacionado ao mínimo custo (ou impedância), considerando algum atributo quantificável, como, por exemplo, distância, tempo, risco, etc. (Boaventura Netto, 1996; Cormen *et al.*, 2002; Drozdek, 2002; Ziviani, 2004). Pode-se considerar ainda uma ponderação entre atributos; por exemplo, o caminho de mínimo custo generalizado entre dois pontos de uma cidade, muito utilizado em redes de transporte público de passageiros, que leva em conta uma soma ponderada do custo (ou desembolso) da viagem e das diferentes parcelas de tempo (de deslocamento, de espera, andando a pé, etc.) associadas à mesma.

O problema de caminho mínimo é um dos problemas genéricos intensamente estudados e utilizados em diversas áreas como Engenharia de Transportes, Pesquisa Operacional, Ciência da Computação e Inteligência Artificial. Isso decorre do seu potencial de aplicação a inúmeros problemas práticos que ocorrem em transportes, logística, redes de computadores e

de telecomunicações, etc. (Peer e Sharma, 2007), como também sua utilização para resolver problemas mais complexos de otimização combinatória tais como o problema de roteirização de veículos através do método de geração de colunas (Chabrier, 2007) em que o caminho mínimo deve ser recalculado diversas vezes. Portanto, seu desempenho é crucial para o sucesso de tais abordagens para problemas mais complexos.

Mais recentemente, têm-se observado um aumento do interesse pelo problema de caminho mínimo, em virtude do seu uso em várias aplicações em engenharia de transportes, motivadas pelo avanço e disseminação dos Sistemas de Informação Geográfica, ou SIG's (do inglês *Geographic Information Systems* ou GIS), que permitem tratamento computacional a dados geográficos ou geo-referenciados. Inúmeras aplicações baseadas em SIG's vêm sendo disponibilizadas na internet, muitas delas na forma de mapas ou guias eletrônicos, que permitem não só localizar um endereço, como também encontrar o "melhor" caminho entre dois pontos. A cada nova solicitação, um algoritmo de caminho mínimo deve ser processado, a fim de encontrar o caminho mínimo entre os dois pontos indicados pelo usuário, que podem ser desde endereços em uma mesma cidade ou região metropolitana até endereços correspondentes a localidades distantes entre si. Em qualquer caso, é fundamental que a consulta *on-line* retorne uma resposta em tempo muito reduzido. Para tanto, é fundamental um algoritmo de caminho mínimo rápido e eficiente, que permita que muitas consultas possam ser tratadas simultaneamente.

Outro aspecto que tem levado a esse verdadeiro renascimento do interesse pelos algoritmos de caminho mínimo é dado, segundo Fu *et al.* (2006), pelos recentes avanços dos chamados Sistemas Inteligentes de Transporte (do inglês *Intelligent Transportation Systems* ou ITS), definidos genericamente como sistemas de transporte que usam tecnologias de informática e de telecomunicações para assegurar a sua melhor utilização e operação, buscando controlar e reduzir congestionamentos e filas. Entre as inúmeras aplicações, incluem-se sistemas de apoio à navegação em tempo real, cuja finalidade é auxiliar motoristas a encontrar os melhores caminhos ou rotas para atingirem seus locais de destino, sistemas automáticos de despacho de veículos (por exemplo, para despacho em tempo real de veículos de atendimento ou socorro), entre outros.

Todas essas aplicações dependem de se encontrar os caminhos mínimos entre pares de pontos, repetidas vezes, e cujos tempos de resposta para encontrar as soluções devem ser bastante reduzidos, compatíveis com aplicações em tempo real. Embora existam na literatura inúmeros artigos que descrevam algoritmos eficientes para problemas de caminho mínimo, verificou-se a carência de um artigo atualizado, em que os principais algoritmos eficientes encontrados na literatura pudessem ser avaliados e comparados, considerando classes de problemas como as encontradas na prática em aplicações em engenharia de transportes, em logística e em engenharia de tráfego.

Assim, o objetivo deste artigo é apresentar uma avaliação comparativa de algoritmos eficientes para o problema de caminho mínimo, considerando instâncias reais encontradas na prática, tendo em vista os desafios proporcionados pelas aplicações acima citadas. Busca-se, dessa forma, orientar os pesquisadores e profissionais em diversas áreas, entre as quais Engenharia de Transportes, Logística e Geoprocessamento, quanto à escolha do melhor algoritmo de caminho mínimo para as aplicações críticas em termos de tempo de processamento e memória necessária. Adicionalmente, espera-se, como esse artigo, preencher

uma lacuna na literatura que é a análise comparativa desses algoritmos considerados os mais eficientes, tendo em vista que alguns deles correspondem a artigos publicados há mais de dez anos.

Este artigo está organizado da seguinte forma: inicialmente, na próxima seção, é apresentada uma breve conceituação do problema de caminho mínimo, seguindo-se a descrição de todos os algoritmos implementados computacionalmente, na quarta seção são apresentados os testes computacionais realizados com dados reais de malhas rodoviárias dos Estados Unidos e do Brasil e, na última seção são apresentadas as considerações finais sobre este trabalho.

2. O PROBLEMA DE CAMINHO MÍNIMO

Uma rede de transporte (que pode ser uma malha viária, rodoviária, etc.) pode ser representada por uma grafo $G = (N, A)$, onde N é o conjunto de nós e A é o conjunto de arcos os quais interligam estes nós. Neste trabalho será considerado o número de nós $|N| = n$ e o número de arcos $|A| = m$; para cada arco $(i, j) \in A$ está associado um custo unitário c_{ij}. O caminho entre um nó origem (s) e um nó destino (t) é definido por uma seqüência de arcos: $(s,i),...(k,l),...(j,t)$. O Problema de Caminho Mínimo (PCM) consiste em determinar um caminho entre s e t tal que a somatória dos custos unitários dos arcos (ou alguma outra medida de impedância) que compõem este caminho seja o mínimo.

O PCM vem sendo estudado há mais de 40 anos em diversas áreas de conhecimento como ciência da computação, matemática e engenharia de transportes. Devido à dificuldade computacional em tratar este problema, a maioria das pesquisas nesta área se concentra no desenvolvimento de algoritmos eficientes para resolver o PCM. A maioria dos algoritmos para encontrar o valor ótimo do caminho mínimo é baseada em aplicações da teoria de programação dinâmica para encontrar o caminho mínimo em uma rede. Seja $L(i)$ o custo acumulado no caminho desde o nó de origem s até o nó i, e $P(i)$ o nó predecessor do nó i, isto é, o nó imediatamente anterior ao nó i no caminho, a partir do qual o nó i é atingido. O par $\{L(i), P(i)\}$ é normalmente denominado etiqueta ou rótulo (do inglês "*label*") do nó i. O conjunto Q corresponde aos nós a serem examinados durante o processo de busca do caminho mínimo. A seqüência de passos de um algoritmo genérico de caminho mínimo é dada por:

Passo 1: Inicialização:	Faça $i = s$; $L(i) = 0$; $L(j) = \infty \ \forall j \neq i$; $P(i)$=NULO; $Q = \{i\}$
Passo 2: Seleção de nó:	Selecionar e remover um nó i do conjunto Q
Passo 3: Expansão de nó:	Varrer a lista de arcos (i, j) emanando do nó i Se $L(i) + c_{ij} < L(j)$ então $L(j) = L(i) + c_{ij}$ e $P(j) = i$ $Q = Q \cup \{j\}$
Passo 4: Critério de parada:	Se $Q = \varnothing$ então parar; senão retornar ao Passo 2

A grande diferença entre os diversos algoritmos aplicados ao PCM reside na maneira como o conjunto Q é examinado a fim de selecionar e remover um nó i no Passo 2, definindo duas classes de algoritmos: os algoritmos de fixação de etiqueta (do inglês "*label setting*") e os algoritmos de correção de etiqueta (do inglês "*label correcting*"). Nos algoritmos do tipo "*label setting*" o nó $i \in Q$ é selecionado com base em algum critério de ordenação, de maneira a assegurar que a sua etiqueta não tenha como ser alterada em iterações subseqüentes, tornando-se permanente; em outras palavras, i é escolhido de tal forma que se garante que o valor $L(i)$ não possa ser reduzido por outros caminhos que venham a ser percorridos em

iterações subseqüentes. Já os algoritmos da classe "*label correcting*" consideram alguma estrutura do tipo lista ou similar, sem uma ordenação especial, e que não garante que um nó *i* selecionado no Passo 2 não possa retornar ao conjunto *Q* em uma iteração futura. Em outras palavras, em cada uma das classes de algoritmos, a grande distinção reside na estrutura de dados utilizada e na forma em que a rede é percorrida durante a busca pelo próximo nó a ser considerado como parte do caminho mínimo. Boas referências sobre problemas de caminho mínimo podem ser encontradas em Ahuja *et al.* (1993) e Fu *et al.* (2006).

Tendo em vista que as redes de transporte para os quais os algoritmos de caminho mínimo devem ser aplicados normalmente apresentam um número muito elevado de nós e arcos, uma implementação computacional eficiente é de fundamental importância para assegurar um bom desempenho. Isto é representado pelo algoritmo e pela estrutura de dados escolhidos. Os requisitos de hardware também exercem um papel importante; a evolução da informática proporciona processadores cada vez mais velozes, possibilitando tempos de processamento cada vez menores para volume de dados cada vez maiores. Este cenário propicia a proposição de novos algoritmos e estruturas de dados, além das melhorias em algoritmos existentes com a implementação de estruturas de dados antes inviáveis computacionalmente.

3. ALGORITMOS IMPLEMENTADOS

Nesta seção são apresentados os cinco algoritmos analisados neste trabalho. Todos foram implementados utilizando estruturas de dados eficientes, visando obter a solução ótima para o problema de caminho mínimo no menor tempo computacional possível.

3.1. Algoritmo Radix Heap

O algoritmo *Radix Heap* foi proposto inicialmente por Ahuja *et al.* (1990) e ainda é considerado no meio científico como um dos algoritmos mais eficientes para resolver o problema do caminho mínimo. Seu funcionamento é descrito na Figura 1.

Cada nó da rede possui um rótulo com a distância entre este nó e o nó origem. O algoritmo inicia todos os rótulos como temporários e com valor infinito, cria *K buckets* vazios com faixas de valores que variam entre 2^{k-1} e 2^k-1 com $k = 1, 2, ..., K$. Em seguida, insere cada nó *i* no *bucket* cuja faixa de valores corresponda ao valor atual do rótulo do nó *i*.

Em seguida, o algoritmo seleciona o nó com o menor rótulo temporário e o rotula permanentemente. A partir deste último nó rotulado permanentemente, o algoritmo varre todos os arcos que saem deste nó e atualiza as etiquetas temporárias dos nós que foram alcançados a partir deste último nó rotulado permanentemente.

O rótulo temporário de um nó somente é atualizado quando o novo valor do rótulo for inferior ao atual, isto é, o novo caminho encontrado possui o custo inferior ao do caminho atual. Quando um rótulo temporário é atualizado o algoritmo verifica se o nó referente a este rótulo deve ser realocado para um novo *bucket* ou se deve continuar no mesmo *bucket*.

Quando um *bucket* contiver muitos nós dentro dele, as faixas de valores de todos os *buckets* serão recalculadas e o conteúdo deste *bucket* será redividido entre todos os demais *buckets*. O critério de parada do *Radix Heap* ocorre quando o nó destino for rotulado ou quando todos os nós receberem rótulos permanentes. Um fator importante a ser observado do algoritmo *Radix Heap* é que as dimensões dos buckets crescem exponencialmente e a sua faixa de valores mudam dinamicamente enquanto o algoritmo é executado.

Segundo Ahuja *et al.* (1990), a complexidade do algoritmo Radix Heap, o qual é uma evolução do Algoritmo *Dial*, é $O(m + n \log(nC))$. A Figura 1 apresenta o pseudocódigo do algoritmo *Radix Heap*.

```
AlgoritmoRadixHeap
        inicializa os buckets
        Para i = 0 até i = Número de buckets Faça
                IniBucket = 2^(k-1) ; FimBucket = 2^k-1;
        Fim Para;
        Para i = 1 até Número de nós Faça
                s[i]= 0, nó i não rotulado; d[i] = MAXINT, distancia da origem ao nó i igual a infinito;
        Fim Para;
        Enquanto todos os nós não forem rotulados permanentemente Faça
                escolha o nó com o menor rótulo temporário;
                remove do heap o no que será rotulado permanentemente;
                Se o bucket estiver muito cheio Faça
                        Recalcula as faixas de valores; Redistribui os nós;
                Fim Se
                rotula o nó i permanentemente; k = o primeiro arco que sai do nó i para o nó j;
                Enquanto k diferente de 0 Faça
                        d[j]= d[i]+dist[k], atualiza a distância do nó origem ao nó j;
                        insere o nó j no bucket que contenha a sua faixa de valores;
                        k= linkout[k], k recebe o próximo arco ou zero caso tenha
                                        acabado de percorrer todos os arcos que saem do nó i
                Fim Enquanto
        Fim Enquanto
Fim RadixHeap
```

Figura 1: Pseudocódigo do algoritmo *Radix Heap*

3.2. Algoritmo de Dijkstra

O algoritmo de Dijkstra permite determinar a solução ótima através da adição de vértices à árvore de caminho mínimo pelo processo de relaxamento de uma aresta; em outras palavras, consiste em verificar se há a possibilidade de melhorar o caminho obtido até o momento (Boaventura Netto, 1996; Cormen *et al.*, 2002). Uma etapa de relaxamento pode diminuir o valor da estimativa de caminho mínimo e atualizar o predecessor (Cormen *et al.*, 2002). O algoritmo requer que nenhum peso no grafo seja negativo (Netto, 1996; Cormen *et al.*, 2002). A Figura 2 apresenta o pseudocódigo do algoritmo de Dijkstra conforme apresentado em Cormen *et al.* (2002).

```
Algoritmo Dijkstra
Início
        inicialize a distância para todos os nós em G = infinito
        inicialize o predecessor de todos os nós em G = vazio
        enquanto H não estiver vazio faça
                u = o nó com menor rótulo extraído de H
                para cada v adjacente a u:
                        se rótulo[v] > rótulo[u] + distância[u, v]:
                                rótulo[v] = rótulo[u] + distância[u, v];
                        predecessor[v] = u;
                atualiza a posição de v em H
                        Fimse
                Fim para
        Fim enquanto
Fim Dijkstra
```

Figura 2: Pseudocódigo do algoritmo *Dijkstra*

O algoritmo de Dijkstra resolve o problema de caminho mínimo de uma única origem em um grafo orientado (Cormen *et al.*, 2002). O algoritmo mantém um conjunto de nós cujos pesos desde a origem já foram calculados. O algoritmo seleciona repetidamente o nó que possui a menor estimativa (peso) e calcula os pesos dos nós adjacentes.

3.3. Algoritmo Dijkstra com *Heap* Binomial

A estrutura de dados *Heap* Binomial, conhecida também como fila de prioridades (Tenenbaum *et al.*, 1995), é uma alternativa de implementação do algoritmo de *Dijkstra* a fim de encontrar o nó com etiqueta de menor custo, cujo pseudocódigo é apresentado na Figura 3. O objetivo é reduzir o tempo de execução do algoritmo *Dijkstra* quanto à determinação deste nó em questão. O tempo de execução para determinar o menor custo em um *heap* binomial é sempre *O(log n)* (Cormen *et al.*, 2002). Um *heap binomial* é uma coleção de árvores binomiais. A árvore binomial é uma árvore ordenada que possui as seguintes propriedades:

1. Existem 2^k nós;
2. A altura da árvore é k;
3. Existem exatamente (k^i) nós na profundidade i para $i = 0, 1, ..., k$;
4. A raiz tem grau k, o qual é maior que o de qualquer outro nó;
5. Se os filhos da raiz são numerados da esquerda para a direita por $k-1$, $k-2$, ..., 0, o filho i é a raiz de uma subárvore B_i.

Um *heap* binomial H é definido como um conjunto de árvores binomiais que satisfaz as seguintes propriedades:

1. Cada árvore binomial H obedece à propriedade de *heap* mínimo, ou seja, a chave de um nó é maior ou igual à chave de seu pai;
2. Existe no máximo uma árvore binomial em H cuja raiz tem grau k, para qualquer inteiro não negativo k.

As raízes das árvores binomiais dentro de um *heap* binomial estão organizadas em uma lista de raízes que estão em ordem decrescente de grau (quantidade de filhos). A estrutura do nó armazena 3 ponteiros: ponteiro para o pai (ancestral), para o filho e para o irmão. A única situação em que o ponteiro para o pai é nulo será quando o nó for uma raiz.

```
Algoritmo Dijkstra Heap Binomial

Início
        inicialize a distância para todos os nós em G = infinito
        inicialize o predecessor de todos os nós em G = vazio
        monte o heap binomial
        enquanto Heap binomial não estiver vazio faça
                u = extrai mínimo do heap binomial
                para cada v adjacente a u:
                        se rótulo[v] > rótulo[u] + distância[u, v]:
                                rótulo[v] = rótulo[u] + distância[u, v];
                                atualiza o heap binomial;
                        Fimse
                        predecessor[v] = u;
                Fim para
                atualiza a posição de v no heap binomial
        Fim enquanto
Fim Dijkstra Heap Binomial
```

Figura 3: Pseudocódigo do algoritmo *Heap* Binomial

3.4. Algoritmo de Dijkstra com Heurística

Segundo Fu *et al.* (2006), a abordagem tradicional do algoritmo de Dijkstra pode ser muito custosa em termos computacionais quando aplicada a malhas viárias reais. Os autores, e também Dai (2005) descrevem várias heurísticas que podem ser aplicadas ao problema do caminho mínimo a fim de reduzir seu tempo de processamento, descrevendo as principais características de cada uma delas.

Neste trabalho foram experimentadas duas das heurísticas mais promissoras que podem ser combinadas com o algoritmo de Dijkstra, com o objetivo de acelerar o seu processamento: uma de eliminação de arcos da região de busca, e a outra, que utiliza uma função de custo ou mérito para eleger os nós de mínimo custo, conhecida como *A**. Ambas são descritas a seguir.

3.4.1. Heurística de Eliminação de Arcos

Nesta heurística os arcos emanados de um nó candidato *i* somente são examinados se atenderem à seguinte condição:

$$d[i] + d\_dist[j] <= Kheur * distOD \quad (1)$$

Onde: *d_dist: é* a distância euclidiana entre o nó j a ser examinado e o nó de destino;
Kheur: é um coeficiente que indica a área a ser explorada. Quanto maior o valor do coeficiente, maior a chance de se encontrar o valor ótimo do caminho mínimo;
distOD: distância euclidiana entre o nó de origem e destino.

A distância euclideana *distOD* entre dois pontos é calculada através da seguinte expressão:

$$R * (\arccos(\cos(YLat) * \cos(XLat) + \sin(YLat) * \sin(XLat) * \cos(|XLong - YLong|))) * 1000 \quad (2)$$

Onde: *R:* raio da Terra*;*
XLat, YLat, XLong, YLong: coordenadas dos pontos *X* e *Y* (i.e., *s* e *t*, respectivamente).

Como no algoritmo de Dijkstra são rotulados todos os nós a partir do nó de origem *s*, sem que seja possível verificar se estão ou não na direção do nó de destino *t*, a heurística visa eliminar aqueles nós que claramente correspondem a caminhos que levam a nós que se afastam ainda mais de *t* e que claramente não resultarão no caminho mínimo entre *s* e *t.* A Figura 4 apresenta o pseudocódigo para essa versão do algoritmo.

```
Algoritmo Dijkstra EliminiçãodeArcos
início
        inicialize a distância para todos os nós em G = infinito
        inicialize o predecessor de todos os nós em G = vazio
        enquanto H não estiver vazio faça
                u = o nó com menor rótulo extraído de H
                para cada v adjacente a u:
                        se rótulo[v] + distância[u, destino] <= k * distância[origem, destino]
                                se rótulo[v] > rótulo[u] + distância[u, v]:
                                        rótulo[v] = rótulo[u] + distância[u, v];
                                Fimse
                                predecessor[v] = u;
                        atualiza a posição de v em H
                        Fimse
                Fim para
        Fim enquanto
Fim Dijkstra EliminiçãodeArcos
```

Figura 4: Dijkstra com Heurística de Eliminação de Arcos

Neste trabalho o valor de *Kheur* foi calibrado para as malhas rodoviárias utilizada nos testes, e desta forma foi possível chegar à solução ótima. Num sistema de traçado de rotas, porém, seria necessário fazer um auto-ajuste deste coeficiente. Fu *et al.* (2006) propõem utilizar a velocidade para auxiliar na estimativa do *Kheur*. Os autores também indicam a possibilidade de armazenar em uma lista auxiliar os nós excluídos, para que possam ser examinados, caso não seja encontrado um caminho até o nó de destino.

*3.4.2. Heurística A**

Nesta heurística os nós são rotulados segundo uma função de custo e não somente pela distância da origem. A função de custo é dada pela fórmula abaixo:

$$custo[j] = DHeurBal * d[j] + (1 - DHeurBal) * d\_dist[j]); \quad (3)$$

Onde: *d*: é a distância entre a origem e o nó *j*;

d_dist: é a distância euclidiana entre o nó *j* e o nó destino;

DheurBal: é o coeficiente que indica o peso de cada um dos termos da função: 0 será considerada somente a distância euclidiana (Heurística pura) e, 1 será considerada apenas a distância viária (Dijkstra puro). Neste trabalho foi utilizado o valor de 0,5 para *DheurBal*, com resultados satisfatórios.

```
Algoritmo DijkstraHeuristicaA*
início
        inicialize a distância para todos os nós em G = infinito
        inicialize o predecessor de todos os nós em G = vazio
        enquanto H não estiver vazio faça
                u = o nó com menor rótulo extraído de H
                para cada v adjacente a u:
                        se rótulo[v] > rótulo[u] + merit[u, v]:
                                rótulo[v] = rótulo[u] + merit[u, v];
                                predecessor[v] = u;
                                atualiza a posição de v em H
                        Fim se
                Fim para
        Fim enquanto
Fim DijkstraHeuristicaA*
```

Figura 5: Dijkstra com Heurística A*

3.5. Algoritmo *LQUEUE*

De complexidade de ordem *O(nm)*, é a implementação mais simples, na qual o nó candidato é inserido no final de uma fila, enquanto os demais são removidos um a um do início da fila, como mostrado na Figura 6. Ao contrário dos demais, o algoritmo LQUEUE é do tipo *label correcting*.O algoritmo termina quando a fila de candidatos for esvaziada, de modo que todos os nós recebam sua etiqueta de distância definitiva.

Foi implementada uma segunda versão modificada na qual não é adicionado à fila nenhum nó candidato que venha a ultrapassar a distância já obtida para o nó fim. Desse modo, mesmo que já se tenha passado pelo nó final, só entrar-se-á novamente na iteração de adição de nó candidato, quando essa seleção tiver a possibilidade de redução da distância até então obtida para o nó de destino. Essa mesma modificação foi aplicada nas demais versões de algoritmo nas situações em que o nó de destino foi um dado fornecido.

3.6. Algoritmo *LDQUEUE*

A diferença do algoritmo *LDQUEUE*, de complexidade de ordem $O(n2^n)$ reside somente na diferenciação da reinserção de elementos, como mostrado na Figura 7. Toda vez que um elemento a ser inserido na lista já não apresenta mais a etiqueta de distância como infinito, significa que já passou pela fila e a partir desse momento, se voltar a ser inserido só o será na frente da fila. A modificação para as situações em que o nó de destino é dado é exatamente a mesma que a apresentada no algoritmo *LQUEUE*.

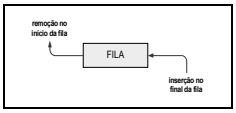

Figura 6: Algoritmo LQUEUE

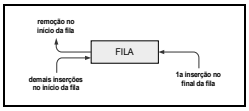

Figura 7: Algoritmo LDQUEUE

3.7. Algoritmo *LTHERSHOLD*

O algoritmo *LTHRESHOLD* é muito semelhante ao *LDQUEUE*, e apresenta a complexidade de mesma ordem $O(n2^n)$. A diferença fundamental, e que resulta num aumento muito grande de sua eficiência está na regra de reinserção. Em vez de simplesmente inserir na frente da fila os nós que já passaram por ela, esse algoritmo mantém uma variável *S* atualizada a cada iteração com o valor da média das etiquetas de distância dos nós que já a receberam (mesmo que não definitiva). No momento de reinserir um nó que já passou pela lista, é feita uma comparação de sua etiqueta de distância com a média mencionada *S*. Se for inferior, o nó será reinserido na frente da fila; se for superior, será reinserido no final.

4. EXPERIMENTOS COMPUTACIONAIS

Os algoritmos propostos foram testados com dados reais de malhas viárias, todas de grande porte. Quatro redes correspondem a malhas rodoviárias de estados americanos (EUA): Distrito de Colúmbia (DC) com 9.559 nós e 29.768 arcos; Virgínia (VA) com 630.639 nós e 1.423.982 arcos; Flórida (FL) com 1.048.506 nós e 2.653.624 arcos; e Califórnia (CA) 1.613.325 nós e 3.970.508 arcos. Já a malha rodoviária brasileira principal (BR) possui 17.092 nós e 23.077 arcos. Os algoritmos propostos foram implementados utilizando C++ e os resultados expostos na Tabela 1 foram obtidos em um computador Pentium IV 3.0GHz com 1 GB de RAM.

Para cada uma das cinco malhas viárias foram resolvidos problemas de caminho mínimo do primeiro nó para todos os demais, e também entre alguns pares de nós origem e destino sorteados aleatoriamente. Todos os algoritmos testados conseguiram chegar aos mesmos resultados para todos os problemas resolvidos, que representam a solução ótima para cada uma das instâncias consideradas, mesmo as duas versões do algoritmo de Dijkstra com heurística. Deste modo, visto que todos os algoritmos implementados foram capazes de encontrar a solução ótima, a questão que permanece é qual o algoritmo mais rápido para encontrar a solução ótima e qual o algoritmo que exigiu menos do computador utilizado nos testes computacionais.

A Tabela 1 apresenta os tempos computacionais necessário para cada um dos algoritmos implementados encontrar a solução ótima de cada problema testado. Podemos observar que, de uma forma geral, o algoritmo de Dijkstra com Heurística foi o mais rápido, seguido pelo *Radix Heap*, o *Heap* Binomial, o Dijkstra puro e por último o algoritmo *L-Threshold*. Outro resultado interessante que esta tabela mostra é que quanto maior o número de nós e arcos da rede maior a diferença de tempo gasto por cada um dos algoritmos. É válido destacar que o

tempo computacional gasto, nos problemas de pequeno porte, para todos os algoritmos foi praticamente o mesmo.

Um dos recursos mais exigidos do computador durante a resolução de um problema de caminho mínimo é a quantidade de memória RAM necessária. Conforme a Tabela 2, o algoritmo *Heap* Binomial apresentou o menor consumo de memória RAM, seguido pelos algoritmos *Radix Heap* e Dijkstra puro empatados com o segundo menor consumo de memória. Os algoritmos Dijkstra com Heurística e *L-Threshold* foram os que mais exigiram memória para resolver os problemas testados.

Tabela 1: Tempos Computacionais

Rede	Origem	Destino	Distância	Tempo (seg)				
				Radix Heap	Heap Binomial	Dijkstra	Dijkstra Heurística	L-Threshold
DC	1	Todos	---	0,015	0,031	0,015	0,003	0,015
DC	4.780	1	11.505	0,015	0,015	0,015	0,001	0,078
DC	4.780	1.594	11.849	0	0,031	0,015	0,002	0,062
DC	4.780	3.187	5.134	0	0,016	0	0	0,015
DC	4.780	6.373	8.129	0	0,016	0	0,001	0,047
DC	4.780	7.966	4.608	0	0,016	0	0	0,016
DC	4.780	9.559	2.763	0	0	0	0	0,016
VA	1	Todos	---	0,453	2,125	2,714	0,248	4,387
VA	315.320	1	308.602	0,469	1,59	3,929	0,13	7,16
VA	315.320	210.213	261.396	0,453	1,558	3,868	0,047	7,212
VA	315.320	315.319	800	0	0	0	0	0,031
VA	315.320	420.425	96.424	0,094	0,343	0,674	0,012	1,236
VA	315.320	525.531	371.214	0,484	1,691	4,029	0,104	7,055
VA	315.320	630.637	152.066	0,281	0,939	2,681	0,106	4,459
FL	1	Todos	---	0,86	3,578	6,115	0,478	30,682
FL	524.254	1	84.750	0,094	0,297	0,548	0,006	1,673
FL	524.254	174.752	95.700	0,125	0,454	1,05	0,007	3,111
FL	524.254	349.503	192.476	0,407	1,578	3,763	0,052	12,962
FL	524.254	699.005	133.506	0,282	0,922	2,52	0,039	7,937
FL	524.254	873.756	155.649	0,328	1,125	2,928	0,025	9,65
CA	1	Todos	---	1,36	5,188	12,116	0,323	68,296
CA	806.663	1	589.905	0,906	3,984	7,201	0,252	28,221
CA	806.663	268.888	208.593	0,547	2,124	5,103	0,116	18,011
CA	806.663	537.775	89.574	0,266	1,14	2,44	0,105	7,692
CA	806.663	1.075.549	138.101	0,422	1,655	3,802	0,23	12,518
CA	806.663	1.344.436	577.618	0,875	3,794	7,071	0,249	27,764
CA	806.663	1.613.323	704.732	1,078	4,761	8,479	0,359	32,983
BR	1	Todos	---	0,01	0,017	0,015	0,007	0,016
BR	1	17.092	2.590	0	0	0	0,005	0
BR	1.409	1	1.043	0	0	0	0,002	0
BR	56	3.345	2.398	0,005	0	0,015	0,005	0,016
BR	4	7.689	719	0	0	0	0	0

Uma estrutura de dados deficiente pode comprometer a implementação de algoritmos para resolver o PCM, a Tabela 3 apresenta os tempos computacionais gastos pelo algoritmo *Radix Heap* com duas estruturas de dados diferentes: lista ligada e *forward star*.

É possível observar que o uso da estrutura de dados *forward star* permitiu que o algoritmo *Radix Heap* fosse ligeiramente mais rápido que o mesmo algoritmo utilizando a estrutura de dados lista ligada.

Tabela 2: Memória Máxima Utilizada

Algoritmo	Memória RAM (MB)
Radix Heap	124
Heap Binomial	111
Dijkstra	124
Dijkstra com Heurística	204
L-Threshold	253

Tabela 3: Estruturas de Dados

Rede	Origem	Destino	FO	Tempo (seg)	
				Lista Ligada	Forward Star
DC	1	Todos	---	0,015	0,015
DC	4.780	1	11.504,8	0	0,015
DC	4.780	1.594	11.848,6	0	0
DC	4.780	3.187	51.34,38	0	0
DC	4.780	6.373	81.28,63	0	0
DC	4.780	7.966	46.08,48	0,015	0
DC	4.780	9.559	2.763,03	0	0
VA	1	Todos	---	0,5	0,453
VA	315.32	1	308.602	0,547	0,469
VA	315.320	210.213	261.396	0,531	0,453
VA	315.320	315.319	800,462	0	0
VA	315.320	420.425	96.424	0,11	0,094
VA	315.320	525.531	371.214	0,547	0,484
VA	315.320	630.637	152.066	0,328	0,281
FL	1	Todos	---	0,969	0,86
FL	524.254	1	84.750,3	0,094	0,094
FL	524.254	174.752	95.699,7	0,156	0,125
FL	524.254	349.503	192.476	0,485	0,407
FL	524.254	699.005	133.506	0,312	0,282
FL	524.254	873.756	155.649	0,375	0,328
CA	1	Todos	---	1,578	1,36
CA	806.663	1	589.905	1,031	0,906
CA	806.663	268.888	208.593	0,656	0,547
CA	806.663	537.775	89.574,3	0,313	0,266
CA	806.663	1.075.549	138.101	0,485	0,422
CA	806.663	1.344.436	577.617	1	0,875
CA	806.663	1.613.323	704732,1	1,234	1,078
BR	1	Todos	---	0,015	0,01
BR	1	17.092	2.590,22	0	0
BR	1.409	1	1.042,73	0	0
BR	56	3.345	2.397,83	0,008	0,005
BR	4	7.689	718,89	0	0

5. CONSIDERAÇÕES FINAIS

O Problema de Caminho Mínimo é um problema antigo e considerado bem explorado. O "renascimento" do interesse em algoritmos de caminho mínimo capazes de processar redes com um número muito elevado de nós e trechos (da ordem de milhões) decorre das

oportunidades, em particular em logística e transporte, proporcionadas pela disseminação e popularização dos mapas digitais e dos Sistemas de Informação Geográfica, recursos esses muitas vezes disponibilizados de maneira universal em ambiente de internet. É o caso, por exemplo, dos sistemas de mapas oferecidos pelo Google e Yahoo, que permitem a qualquer usuário encontrar, em poucos segundos, o menor caminho entre dois endereços quaisquer, que podem estar em pontos a milhares de quilômetros de distância. Outras aplicações que envolvem o despacho de veículos em tempo real (guinchos, ambulâncias, etc.) também dependem de determinar rapidamente o menor caminho entre dois pontos.

Foram implementados cinco dos melhores algoritmos para o caminho mínimo encontrados na literatura e testados considerando problemas correspondentes a malhas viárias reais, de grande porte, algumas delas com milhões de arcos ou trechos de vias. Os resultados indicam que a maioria deles permite encontrar o menor caminho entre dois pontos quaisquer em tempo de processamento inferior a 1 segundo, utilizando uma máquina normal, sem nenhum recurso diferenciado em termos de memória ou capacidade de processamento. O melhor desempenho foi obtido pelo algoritmo de Dijkstra com Heurística, que resultou tempos de processamento inferiores a 0,5 segundo. Deve-se destacar ainda que o algoritmo *Radix Heap*, apesar de tempos um pouco superiores, foi o segundo algoritmo mais rápido e requereu menos memória.

A estrutura *forward star* comprovou ser uma estrutura de dados rápida e eficiente, fazendo com que um mesmo algoritmo se torna mais rápido e com menor consumo de memória RAM utilizando esta estrutura ao invés da estrutura de dados lista ligada.

REFERÊNCIAS BIBLIOGRÁFICAS

Ahuja, R. K., *et al.* (1990) Faster algorithms for the shortest path problem. *Journal of the Association for Computing Machinery*, v. 37, n. 2, p. 213-223.

Ahuja, R. K., Magnanti, T.L. e Orlin, J.B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, NJ.

Boaventura Netto, P. O. (1996) *Grafos - Teoria, Modelos, Algoritmos*. 2ª edição, Editora Edgard Blücher Ltda.

Chabrier, A. (2007) Vehicle routing problem with elementary shortest path based column generation. *Computers and Operations Research*, v. 33, n. 10, p.2972-2990.

Cormen, T.H., *et al.* (2002) *Algoritmos: Teoria e Prática*. Editora Campus.

Dai L. (2005) *Fast shortest path algoritm for road network and implementation*. Honours Project - Carleton University – School of Computer Science.

Drozdek, A. (2002) *Estrutura de dados e algoritmos em C++*, Thomson Learning, São Paulo.

Fu, L., Sun, D., Rillet, L. R. (2006) Heuristic shortest path algorithms for transportation application: State of the art. *Computers and Operations Research*, v. 33, p.3324-3343.

Peer, S.K.; D.K. Sharma (2007). Finding the shortest path in stochastic networks. *Computers & Mathematics with Applications*, v. 53, n.5, p.729-740.

Tenenbaum, A.M., Langsam, Y., Augenstein, M.J. (1995) *Estruturas de dados usando C*. Editora Makron Books.

Ziviani, N. (2004) *Projeto de algoritmos com implementação em pascal e C*. 2ª Edição, Editora Pioneira Thomson Learning.

Jorge von Atzingen (jorge.reis@poli.usp.br)
Cláudio Barbieri da Cunha (cbcunha@usp.br)
Francisco Yastami Nakamoto (francisco.nakamoto@poli.usp.br)
Fábio Rogério Ribeiro (f2r@uol.com.br)
André Schardong (andreSchardong@gmail.com)
Departamento de Engenharia de Transportes, Escola Politécnica, Universidade de São Paulo
Av. Prof. Almeida Prado, trav. 2, n 83 – São Paulos, SP, Brasil – CEP 05508-970