k-menores caminhos

Fábio Pisaruk

Dissertação apresentada ao
Instituto de Matemática e
Estatística da
Universidade de São Paulo
como parte dos requisitos
para obtenção do grau de
Mestre em Ciência da Computação

Orientador: Prof. Dr. José Coelho de Pina Jr.

— São Paulo, 23 de julho de 2012—

Aos meus pais Paulo e Roseli
e meu irmão Marcos.

Resumo

Tratamos da generalização do problema da geração de caminho mínimo, no qual não apenas um, mas vários caminhos de menores custos devem ser produzidos. O problema dos k-menores caminhos consiste em listar os k-caminhos de menores custos conectando um par de vértices.

Esta dissertação trata de algoritmos para geração de k-menores caminhos em grafos simétricos com custos não-negativos, bem como algumas implementações destes.

Palavras-chave: caminhos mínimos, k-menores caminhos.

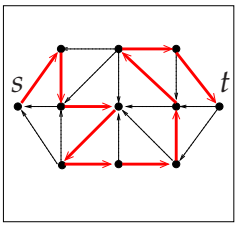
Índice

Lista de Figuras

Lista de códigos

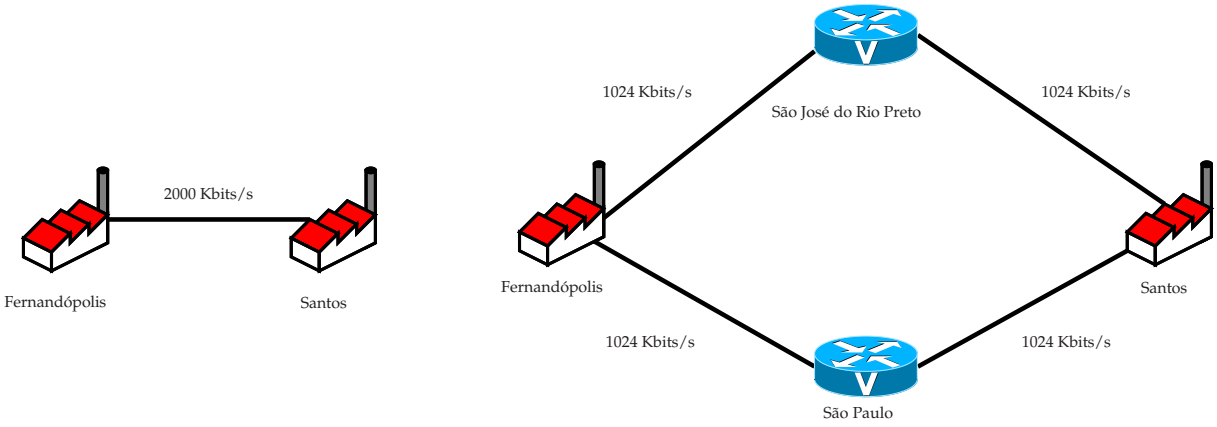
Introdução

[l]30mm



Uma certa empresa de telecomunicações, cujo nome real não será citado por razões de confidencialidade, mas que para nossa comodidade será chamada de TeleMax, fornece linhas de transmissão aos seus clientes de modo que estes possam, por exemplo, ligar-se às suas filiais por linhas privadas. Para tal, conta com uma infra-estrutura de rede bastante complexa compreendendo cabos e diversos equipamentos de junção. Esta rede é *full-duplex*, ou seja, possui passagem de dados em ambos os sentidos. Para entender o processo de fornecimento de linhas de transmissão, passaremos a um exemplo. A empresa SoftSOF possui duas sedes, uma em Santos e outra em Fernandópolis e, deseja interligar suas filiais com uma qualidade mínima de 200 Kbits/s, em pico de uso. A SoftSOF possui 10 computadores em cada uma de suas filiais, logo precisaríamos de um link de 200 Kbits/s $\times 10 = 2000$ Kbits/s. É solicitado à TeleMax um *link* de 2000 Kbits/s ligando as suas sedes. A TeleMax não possui uma ligação direta entre as duas cidades, entretanto possui uma ligação que passa por São José do Rio Preto, ou seja, um caminho Santos - São José Do Rio Preto - Fernandópolis. Infelizmente, este link só dispõe de 1024 Kbits/s. No entanto, observando-se sua infra-estrutura, descobre-se que existe um outro caminho: Santos - São Paulo - Fernandópolis, também com capacidade de 1024 Kbits/s. Pronto. A TeleMax pode fornecer o link requerido

pela SoftSOF, bastando para isso utilizar os dois caminhos acima descritos, totalizando os 2048 Kbits/s, um pouco acima do requerido.



(*a*) (*b*)

Figura 1: Em (*a*) vemos um esquema solicitação da SoftSOF, um link de 2000 Kbits/s entre as filiais. Em (*b*) está descrita a solução encontrada pela TeleMax, com base na disponibilidade de sua rede.

Vamos a algumas considerações relevantes. O custo de um caminho é função da quantidade de equipamentos usada e não da distância total dos cabos que o compõe. Isto se deve ao custo elevado dos equipamentos se comparado ao dos cabos. Assim, passa a ser melhor utilizar uma ligação que percorra uma distância maior mas que passa por um número menor de equipamentos, do que uma com menor distância mas que se utiliza de mais equipamentos.

A justificativa para a geração de diversos caminhos no lugar de apenas um está relacionada à capacidade de transmissão disponível por cabo. A motivação para a geração dos menores caminhos, ou seja, com utilização mínima de equipamentos, requer uma explicação mais detalhada. Até agora fomos simplistas ao tratarmos das relações entre cabos e equipamentos como se um equipamento se ligasse a apenas um cabo. Na verdade, cada equipamento se liga a um grande número de cabos. Assim, podemos ter diversos caminhos entre dois equipamentos, um para cada cabo. A fim de utilizarmos bem os recursos da rede é interessante que o menor número de equipamentos esteja alocado para cada cliente pois, desta maneira, um número maior de ligações poderá ser oferecido pela TeleMax. Embora a utilização do menor número possível de equi-

pamentos para cada cliente não seja suficiente para garantir que a rede esteja sendo utilizada de maneira eficiente, não nos importaremos com isto neste trabalho. Feitas as devidas considerações, vamos agora justificar a automação do processo.

Imagine levar a cabo o processo de fornecimento de linhas manualmente. Podemos salientar alguns problemas da abordagem manual. Devido às dimensões da rede, o operador responsável levará muito tempo para obter uma lista de caminhos entre os pontos. Durante o tempo em que o operador gastar analisando a rede, esta poderá ter sofrido alterações, as quais não serão levadas em conta por ele. Além disso, sabemos como as pessoas são suscetíveis a falhas, ainda mais quando expostas a atividades maçantes e repetitivas. Por conta destes fatores, a TeleMax sentiu a necessidade de uma ferramenta computacional que gerasse de maneira rápida e confiável uma série de caminhos entre dois pontos da sua rede.

Na construção da ferramenta, consideramos a rede como um grafo simétrico, por ser full-duplex, onde as arestas são representadas pelos cabos e os vértices pelos equipamentos. A ferramenta tinha como núcleo o algoritmo desenvolvido por Naoki Katoh, Toshihide Ibaraki e H. Mine [20], de geração de menores caminhos. Os caminhos de mesmo custo, ou seja, que se utilizam de igual quantidade de equipamentos, são posteriormente reordenados crescentemente pela distância total percorrida por seus cabos. Esta dissertação trata de algoritmos que produzem caminhos de menor custo em grafos. Embora algoritmos para tal sejam de interesse teórico, é curioso observar que foi uma aplicação prática, demandada por uma necessidade surgida no âmbito empresarial, que nos levou ao estudo destes.

Organização da dissertação

O capítulo 1 contém a maior parte das notações, conceitos e definições que são usados ao longo desta dissertação.

Em seguida, no capítulo 2, o método de Dijkstra para geração de caminho mínimo é descrito.

No capítulo 3 descrevemos o problema dos k-menores caminhos. Passamos, em seguida a descrever as árvores de prefixos e finalizamos explicamos o algoritmo de YEN que as utiliza.

Em seguida, no capítulo 4 o método desenvolvido por Naoki Katoh, Toshihide Iba-

raki e H. Mine [20] é descrito detalhadamente. Além disso, a implementação feita é exibida juntamente com uma simulação de sua execução.

Seguimos, no capítulo 5 com uma série de gráficos e análises experimentais do desempenho da nossa implementação do algoritmo de KIM.

Finalmente, no capítulo 6, relatamos as nossas conclusões, frustrações e possíveis trabalhos futuros.

CAPÍTULO 1

Preliminares

Neste capítulo apresentamos notações e definições que serão extensivamente empregadas ao longo deste trabalho.

A maior parte das definições seguem de perto as empregadas por Paulo Feofiloff [12].

1.1 Notação básica

O conjunto dos números inteiros será denotado por $\mathbb{Z}$. O conjunto dos números inteiros não-negativos e positivos $\mathbb{Z}_{\geq}$. $\mathbb{Z}$ $\mathbb{Z}_{\geq}$

É escrito S é uma **parte** de um conjunto V significando que S é um subconjunto de V.

Uma **lista** é uma seqüência $\langle v_1, v_2, \ldots, v_k \rangle$ de itens.

Um **intervalo** $[j\,..\,k]$ é uma seqüência de inteiros $j, j+1, \ldots, k$. Se i é um número em $[j\,..\,k]$, então i é um número inteiro tal que $j \leq i \leq k$. $[j..k]$

1.2 Grafos, passeios e caminhos

Um **grafo** é um objeto da forma (V, A), onde V é um conjunto finito e A é um conjunto de pares ordenados de elementos de V.

Os elementos de V são chamados **vértices** e os elementos de A são chamados **arcos**.

Para cada arco (u, v), os vértices u e v representam a **ponta inicial** e a **ponta final** de (u, v), respectivamente.

Um arco (u, v) também será representado por uv. Diremos que um tal arco **sai** de u e **entra** em v. O **grau de entrada** de um vértice v é o número de arcos que entram em v; o **grau de saída** de v é o número de arcos que saem de v.

O conjunto de todos os arcos que têm ponta inicial em um dado vértice v é denotado
$A(v)$ por $A(v)$.

Um grafo é **simétrico** se para cada arco uv existir também o arco vu. Diremos às vezes que o arco vu é **reverso** do arco uv e que o par $\{(u, v), (v, u)\}$ é uma **aresta**.

Um grafo pode ser naturalmente representado através de um diagrama, como o da figura 1.1, onde os vértices são pequenas bolas e os arcos são as flechas ligando estas bolas.

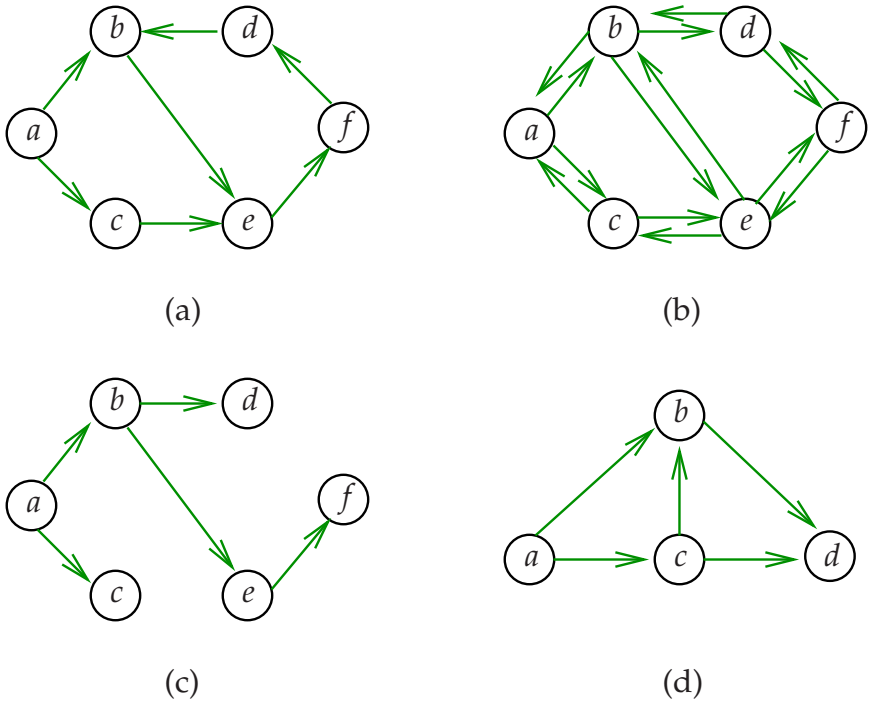


Figura 1.1: (a), (b), (c) e (d) são exemplos de grafos. (b) é um grafo simétrico.

Um **passeio** num grafo (V, A) é qualquer seqüência da forma

$$\langle v_0, a_1, v_1, \ldots, a_t, v_t \rangle \tag{1.1}$$

onde $v_0, \ldots, v_t$ são vértices, $a_1, \ldots, a_t$ são arcos e, para cada i, a_i é um arco com ponta

inicial v_{i-1} e ponta final v_i. O vértice v_0 é o **início** ou **ponta inicial** do passeio e o v_t é seu **término** ou **ponta final**.

Na figura 1.1(a) a seqüência $\langle a, ab, b, be, e, ef, f, fd, d, db, b, be, e, ef, f\rangle$ é um passeio com início em a e término em f.

Se $P := \langle v_0, a_1, v_1, \ldots, a_t, v_t\rangle$ é um passeio, então qualquer subseqüência da forma

$$\langle v_i, a_{i+1}, v_{i+1}, \ldots, a_j, v_j\rangle \tag{1.2}$$

com $0 \leq i \leq j \leq t$ será um **subpasseio** de P. Além disso, se $i = 0$, então o subpasseio será dito um **prefixo** de P. Na figura 1.1(a) a seqüência $\langle a, ab, b, be, e, ef, f, fd, d\rangle$ é um subpasseio e prefixo de do passeio $\langle a, ab, b, be, e, ef, f, fd, d, db, b, be, e, ef, f\rangle$.

Um **caminho** é um passeio sem vértices repetidos. Na figura 1.1(a) a seqüência $\langle a, ab, b, be, e, ef, f\rangle$ é um caminho com início em a e término em f.

Por conveniência, nossa definição de grafos não têm '"arcos paralelos": dois arcos diferentes não podem ter a mesma ponta inicial e a mesma ponta final. Assim, podemos representar o passeio em (1.1) simplesmente por

$$\langle v_0, v_1, v_2, \ldots, v_t\rangle.$$

1.3 Grafos no computador

Existem pelo menos três maneiras populares de representar um grafo em um computador, são elas: (1) matriz de adjacência; (2) matriz de incidência e (3) listas des adjacência. Nesta dissertação, matriz de adjacência e listas de adjacência são as representação utilizadas.

Matriz de adjacência

Uma **matriz de adjacência** de um grafo (V, A) é uma matriz com valores em $\{0, 1\}$, e indexada por $V \times V$, onde cada entrada (u, v) da matriz tem valor 1 se existe no grafo um arco de u a v, e 0 caso contrário. Para grafos simétricos a matriz de adjacências é simétrica. O espaço gasto com esta representação é proporcional a n^2, onde n é o número de vértices do grafo. Uma matriz de adjacência é mostrada na figura 1.2.

	a	b	c	d
a	0	1	1	0
b	0	0	0	1
c	0	1	0	1
d	0	0	0	0

Figura 1.2: Matriz de adjacência do grafo da figura 1.1(d).

Matriz de incidência

Uma **matriz de incidência** de um grafo (V, A) é uma matriz com valores em $\{-1, 0, +1\}$ e indexada por $V \times A$, onde cada entrada (u, a) é -1 se u é ponta inicial de a, $+1$ se u é ponta final de a, e 0 caso contrário. O espaço gasto com esta representação é proporcional a nm, onde n é o número de vértices e m é o número de arcos do grafo. Uma matriz de incidência da figura 1.1(d) pode ser vista em 1.3.

	ab	ac	cb	cd	bd
a	-1	-1	0	0	0
b	$+1$	0	$+1$	0	-1
c	0	$+1$	-1	-1	0
d	0	0	0	$+1$	$+1$

Figura 1.3: Matriz de incidência do grafo da figura 1.1(d).

Listas de adjacência

Na representação de um grafo (V, A) através de **listas de adjacência** tem-se, para cada vértice u, uma lista dos arcos com ponta inicial u. Desta forma, para cada vértice u, o conjunto $A(u)$ é representado por uma lista. O espaço gasto com esta representação é proporcional a $n + m$, onde n é o número de vértices e m é o número de arcos do grafo. Uma lista de adjacência está ilustrada na figura 1.4.

1.4 Filas de prioridade

Sempre que representamos dados em um computador nós consideramos cada um dos seguintes aspectos:

$A(a)$: ab, ac
$A(b)$: bd
$A(c)$: cb, cd
$A(d)$:

Figura 1.4: Listas de adjacência do grafo da figura 1.1(d).

(1) a maneira que essas informações (ou objetos do mundo real) são modelados como objetos matemáticos;

(2) o conjunto de operações que definiremos sobre estes objetos matemáticos;

(3) a maneira na qual estes objetos serão armazenados (representados) na memória de um computador;

(4) os algoritmos que são usados para executar as operações sobre os objetos com a representação escolhida.

Para proceguir, precisamos entender a diferença entre os seguintes termos, tipo de dados, tipo abstrato de dados e estrutura de dados.

O **tipo de dado** de uma variável é o conjunto de valores que esta variável pode assumir. Por exemplo, uma variável do tipo boolean só pode assumir os valores TRUE e FALSE.

Os itens (1) e (2) acima dizem respeito ao **tipo abstrato de dados**, ou seja, ao modelo matemático junto com uma coleção de operações definidas sobre este modelo. Um exemplo de tipo abstrado de dados é o conjunto dos números inteiros com as operações de *adição*, *subtração*, *multiplicação* e *divisão* sobre inteiros.

Já os itens (3) e (4) estão relacionados aos aspectos de implementação.

Para representar um tipo abstrato de dados em um computador nós usamos uma **estrutura de dados**, que é uma coleção de variáveis, possivelmente de diferentes tipos, ligadas (relacionas) de diversas maneiras.

Uma **fila de prioridades** [1, 6] é um tipo abstrato de dados que consiste de uma coleção de itens, cada um, com um valor ou prioridade associada.

Uma fila de prioridade têm suporte para as seguintes operações:

- INSERT(v, val): adiciona o vértice v com valor val para a coleção.

- DELETE(v): remove o vértice v da coleção.
- EXTRACT-MIN( ): devolve o vértice com o menor valor e o remove da coleção.
- DECREASE-KEY(v, val): Muda para val o valor associado ao vértices v; assume-se que val não é maior que o valor corrente associado a v. Note que DECREASE-KEY sempre pode ser implementada como um DELETE seguido por um INSERT.

Uma seqüência de operações é chamada **monótona** se os valores retornados por sucessivos EXTRACT-MIN's são não-decrescentes. O algoritmo DIJKSTRA (seção 2.5) executa uma seqüência monótona de operações.

1.5 Java Universal Network/Graph Framework

O Java Universal Network/Graph Framework(JUNG) é uma biblioteca de software livre, escrita em Java, desenvolvida para permitir a modelagem, análise e visualização de dados passíveis de serem representados na forma de grafos. Além de permitir a visualização de grafos, conta com diversos algoritmos implementados e estruturas de dados pertinentes à área de grafos.

A biblioteca foi criada de modo a abranger as mais diversas necessidades, sendo assim bastante genérica, capaz de representar grafos na forma de matrizes e listas de adjacência e tratar de grafos simétricos e assimétricos. No nível mais elevado, ou seja, de maior abstração, temos as interfaces com prefixo Archetype, as quais definem as diretrizes dos tipos mais genéricos de elementos componentes de um grafo: vértices, arcos e o grafo em si. São representantes deste nível as interfaces: *ArchetypeGraph*, *ArchetypeVertex* e *ArchetypeEdge*.

No nível imediatamente inferior, especializando as interfaces anteriores, encontramos as interfaces: *Graph*, *Vertex* e *Edge*. Estas objetivam representar elementos de grafos sem arestas paralelas, uma vez que estes permitem que novos operações sejam definidas.

Para se trabalhar com grafos simétricos e distingui-los dos demais foram criadas duas interfaces: *DirectedGraph* e *UndirectedGraph* e as respectivas *DirectedEdge* e *UndirectedEdge*. Estas interfaces permitem validações em tempo de compilação. Nenhuma delas possue métodos próprios, apenas estendem a interface *Graph*. A validação em tempo de compilação ocorre por conta da comparação entre a assinatura das funções

que as utilizam. Se uma dada função possuir na sua assinatura um parâmetro do tipo *UndirectedGraph* e for invocada com um argumento do tipo *DirectedGraph* teremos um erro de compilação. Observe, no entanto, que a interface *UndirectedGraph* por si só não valida a simetria do grafo. Esta valição fica a cargo da implementação da mesma.

A seguir temos as implementações das interfaces citadas acima. Numa camada intermédiaria, existem três classes abstratas implementando funcionalidades comuns a grafos simétricos e não-simétricos, são elas: *AbstractSparseGraph*, *AbstractSparseVertex* e *AbstractSparseEdge*. No momento, apenas temos implementada a representação de grafos como listas de adjacência, como pode ser notado nos próprios nomes das classes, os quais contêm a palavra Sparse; vale lembrar que grafos esparsos, ou seja, com poucas arestas, se comparado à quantidade máxima possível, são, em geral, representados de maneira mais eficiente e econônica usando-se listas de adjacência e grafos densos como matrizes.

Por fim, temos as implementações específicas para grafos simétricos: *UndirectedSparseGraph*, *UndirectedSparseVertex* e *UndirectedSparseEdge* , e não-simétricos: *DirectedSparseGraph*, *DirectedSparseVertex* e *DirectedSparseEdge*

Visto um pouco da arquitetura vamos passar a alguns exemplos e aplicações. Criar um grafo é um processo bem simples, basta instar a classe referente ao tipo grafo desejado, como no exemplo a seguir:

```
Graph g = new DirectedSparseGraph();
```

cria um grafo não-simétrico baseado numa representação na forma de lista de adjacência.

Também é possivel criar um grafo lendo seus dados de um arquivo. Apresentaremos apenas um exemplo usando o padrão Pajek uma vez que o JUNG é capaz de ler e escrever apenas neste formato (o formato GraphML é suportado somente no modo leitura). O formato Pajek é muito abrangente, permitindo definições bem complexas de grafos contudo, apresentaremos apenas dois exemplos que ilustram muito bem a simplicidade deste padrão.

Grafo simétrico sem custos nas arestas

```
/*Cria um grafo simétrico com 14 vértices.*/
/*Os vértices saõ numerados a partir do 1.*/
/*Cada vértice pode ter um rótulo que deve vir após seu número. */
/*No caso o vértice 2 tem o rótulo W2.*/
/*Para definir as arestas usamos o *Edges.*/
/*Cada aresta é definida a partir dos vértices que ela interliga*/
/*O arquivo não pode conter linhas em branco ao final.*/
*Vertices 14
1 I1
```

```
2  I3
3  W1
4  W2
5  W3
6  W4
7  W5
8  W6
9  W7
10  W8
11  W9
12  S1
13  S2
14  S4
*Edges
1  5
3  5
3  6
5  6
9  10
9  11
10  11
3  12
5  12
6  12
10  14
11  14
9  12
```

A figura 1.5 mostra o grafo descrito acima.

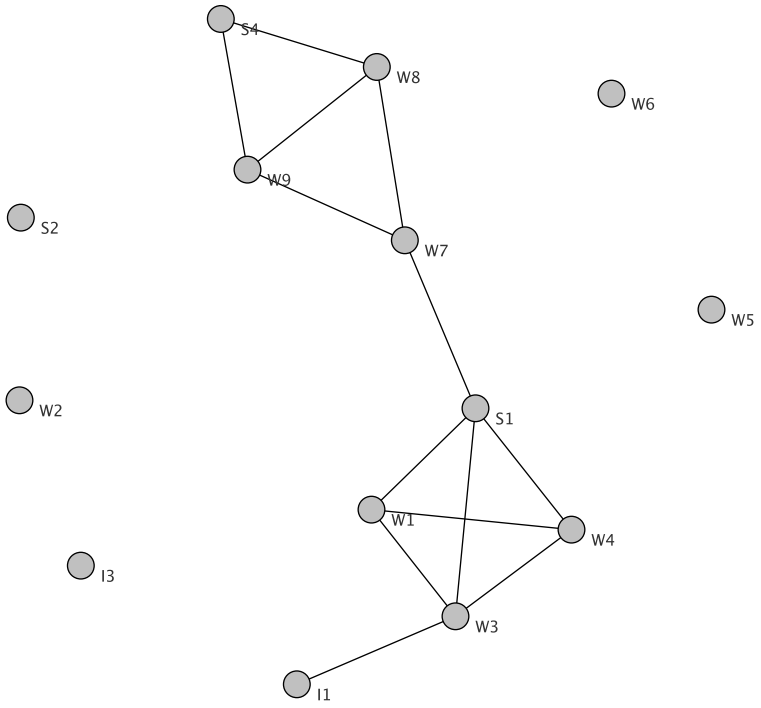


Figura 1.5: Grafo simétrico sem custos nas arestas

Grafo simétrico com custos nas arestas

```
/*Cria um grafo simétrico com 14 vértices.*/
/*Os vértices saõ numerados a partir do 1.*/
/*Cada vértice pode ter um rótulo que deve vir após seu número. */
/*No caso o vértice 2 tem o rótulo W2.*/
/*Para definir as arestas usamos o *Edges.*/
/*Cada aresta é definida a partir dos vértices que ela interliga*/
/*O custo de cada aresta vem logo após o segundo vértice que o compõe. */
Por exemplo, a aresta que liga os vértices 3 e 5 tem custo 2.9 */
/*O arquivo não pode conter linhas em branco ao final.*/
*Vertices 14
1 I1
2 I3
3 W1
4 W2
5 W3
6 W4
7 W5
8 W6
9 W7
10 W8
11 W9
12 S1
13 S2
14 S4
*Edges
1 5 1
3 5 2.9
3 6 3
5 6 90
9 10 1
9 11 2
10 11 3
3 12 66
5 12 6.7
6 12 21
10 14 2
11 14 33
9 12 1
```

A figura 1.5 mostra o grafo descrito acima.

Grafo assimétrico

```
/*Cria um grafo assimétrico com 19 vértices.*/
/*Os vértices saõ numerados a partir do 1.*/
/*Cada vértice pode ter um rótulo que deve vir após seu número.*/
/*No caso, optamos por não rotulá-los.*/
/*Para definir os arcos usamos o *Arcslist.*/
/*Cada arco é definido a partir do vértice de origem e uma lista de vértices de chegada*/
Por exemplo, a linha "1 4 6 17 5 13 12 9 8 7" significa que saem arcos com origem no vértice 1 e chegada nos vértices "4 6 17 5 13 12 9 8 7".
/*O arquivo não pode conter linhas em branco ao final.*/
*Vertices 19
*Arcslist
1 4 6 17 5 13 12 9 8 7
3 13 12 8 7
4 6 5 9
2 6 5
13 16
12 16
9 16
9 19
8 16 18
7 16 18
15 16 19
14 16 19
6 18
5 18
16 19 18
11 18
10 18
```

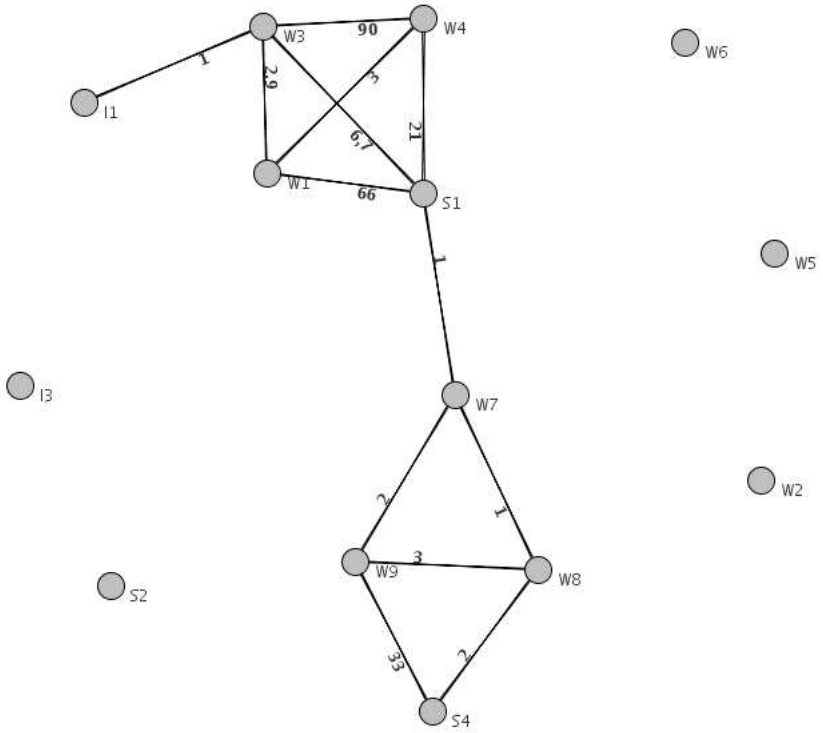


Figura 1.6: Grafo simétrico com custos nas arestas

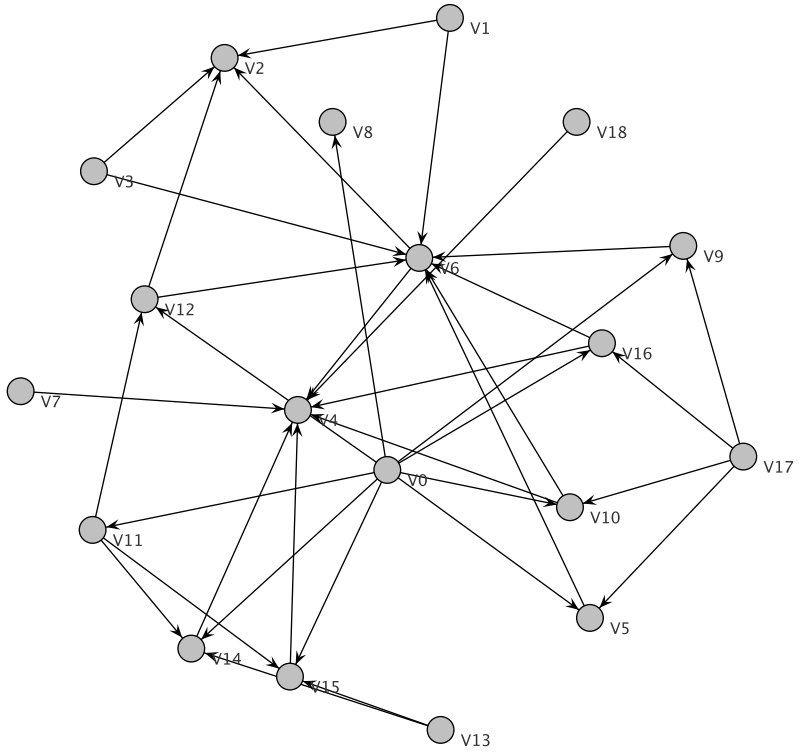


Figura 1.7: Grafo assimétrico sem custos nas arestas

Para ler o grafo a partir do arquivo basta criar um leitor PajekNet, instar a classe referente ao tipo de grafo desejado(simétrico ou não-simétrico), definir como os custos são atribuídos e informar ao leitor o arquivo com o grafo.

```
PajekNetReader pajekNetReader = new PajekNetReader(false);
UndirectedGraph g = new UndirectedSparseGraph();
NumberEdgeValue nev = new UserDatumNumberEdgeValue(g);
Path.setNumberEdgeValue(nev);
g = (UndirectedGraph) pajekNetReader.load("data/pajNetTest.dat", g, nev);
```

Código 1.1: Leitura usando PajNetReader

O JUNG permite flexibilidade na maneira como os custos são atribuidos às arestas. Existe uma interface chamada *NumberEdgeValue* que define apenas dois métodos: *getNumber* e *setNumber*. A idéia é que o desenvolvedor fique livre para criar qualquer tipo de implementação que defina os custos dos arcos/arestas do seu grafo. A biblioteca JUNG já conta com quatro implementações desta interface: *ConstantDirectionalEdgeValue*, *ConstantEdgeValue*, *EdgeWeightLabeller* e *UserDatumNumberEdgeValue* Nosso projeto fez uso apenas de duas destas:

ConstantEdgeValue define todos os arcos como tendo o mesmo custo.

UserDatumNumberEdgeValue obtém os custos dos arcos no repositório de dados do usuário.

Caso o grafo tenha sido obtido a partir de um arquivo no formato Pajek contendo custos devemos usar o UserDatumNumberEdgeValue.

Uma vez criado o grafo, podemos adicionar-lhe vértices da seguinte forma:

```
Vertex v1 = (Vertex) new DirectedSparseVertex();
Vertex v2 = (Vertex) new DirectedSparseVertex();
g.addVertex(v1);
g.addVertex(v2);
```

Código 1.2: Criação e adição de vértices ao grafo

e depois adicionar-lhe os arcos:

```
DirectedEdge e = (DirectedEdge) new DirectedSparseEdge(v1, v2);
g.addEdge(e);
```

Código 1.3: Criação e adição de arcos ao grafo

Observe nos exemplos acima que tanto arcos quanto vértices são independentes do grafo: primeiramente são criados e só então adicionados a ele. Algumas observações importantes:

- Um vértice/arco só pode pertencer a um grafo.
- Um vértice/arco só pode ser adicionado uma vez a um grafo.
- A direcionalidade de um vértice deve coincidir com a do grafo no qual ele será inserido. Por exemplo, não é possível adicionar um vértice *DirectedSparseVertex* a uma implementação de *UndirectedGraph*.
- A direcionalidade de um arco deve coincidir com a dos vértices que este conecta e também com a do grafo.

Citamos anteriormente o repositório de dados do usuário. O JUNG permite que o usuário adicione dados a cada um dos elementos do grafo. Para tal, o usuário pode optar por especializar uma classe que implemente a interface *ArchetypeVertex* ou utilizar os métodos de anotação oferecidos.

Especialização: Suponha que cada vértice contenha um nome. Usando-se especialização de classes, o usuário pode criar a classe *MeuVertice* contendo o atributo nome e métodos que definam e obtenham este dado, como é mostrado no exemplo a seguir:

```
class MeuVertice extends DirectedSparseVertex {
    private String nome;

    public MeuVertice( String nome ) {
       this.nome = nome;
    }

        public String getNome(){
                return nome;
        }

        public void setNome(String nome){
                this.nome=nome;
        }
```

```
}
```

Código 1.4: Especialização da classe DirectedSparseVertex

Anotações: Podemos realizar a mesma tarefa utilizando uma solução bem mais flexível: anotações. Cada uma das implementações das interfaces *Vertex*, *Edge* e *Graph* implementa também a interface *UserData* a qual define operações que permitem adicionar dados a cada um dos elementos do grafo. São elas:

addUserDatum(key, datum, copyaction): Adiciona o objeto datum usando o objeto key como chave além de especificar o copyaction.

getUserDatum(key): Obtém o objeto armanezado com a chave key.

removeUserDatum(key): Remove o objeto armazenado com a chave key.

setUserDatum(key, datum, copyaction): Adiciona ou substitui o objeto cuja chave seja key, além de redefinir o copyaction.

importUserData(udc): Importa os dados do repositório de usuário udc.

getUserDatumKeyIterator(): Retorna um objeto de iteração que permite navegadar pelos dados armazenados pelo usuário no seu repositório.

getUserDatumCopyAction(key): retorna o copyaction especificado pelo usuário para o objeto armazenado segundo a chave key.

Como adicionar a informação nome a um vértice:

```
Vertex v = (Vertex) g.addVertex(new DirectedSparseVertex());
v.addUserdatum("nome","Pisaruk",UserData.SHARED);
g.addUserdatum("id","10",UserData.CLONE);
```

Código 1.5: Adicionando dados ao repositório do usuário

Quando um grafo ou qualquer de seus elementos constituintes é copiado, o destino dos dados do repositório do usuário de cada um deles é determinado pelo seu copyaction. JUNG fornece três diferentes soluções, sendo que o usuário pode criar outras implementando a interface CopyAction:

UserData.CLONE: retorna uma cópia dos dados armazenados segundo a implementação do método clone(), definido na classe Object do Java.

UserData.REMOVE: retorna null, ou seja, o dado não é copiado.

UserData.SHARED: retorna uma referência ao objeto armazenado, ou seja, qualquer mudança será refletida nas duas referências.

CAPÍTULO 2

Caminhos mínimos e Dijkstra

Estão descritos neste capítulo os elementos básicos que envolvem o problema do caminho mínimo, tais como função-custo, função-potencial, função-predecessor, critério de otimalidade e o celebrado algoritmo de Edsger Wybe Dijkstra [8] que resolve o problema do caminho mínimo. A referência básica para este capítulo são as notas de aula de Paulo Feofiloff [12] e a dissertação de Shigueo Isotani [17].

2.1 Descrição

Uma **função-custo** em (V, A) é uma função de A em $\mathbb{Z}_{\geq}$. Se c é uma função-custo em (V, A) e uv estiver em A, então $c(uv)$ será o valor de c em uv. Se P for um caminho em um grafo (V, A) e c uma função-custo, então $c(P)$ é o **custo do caminho** P, ou seja, $c(P)$ é o soma dos custos de todos os arcos em P. Um caminho P tem **custo mínimo** se $c(P) \leq c(P')$ para todo caminho P' com o mesmo início e término que P (figura 2.1).

Como a nossa função-custo é não-negativa, então no grafo há sempre um passeio de custo mínimo que é uma caminho. Por esta razão, um passeio de custo mínimo é simplesmente chamado de **caminho mínimo**.

Se (V, A) é um grafo simétrico e c é uma função comprimento em (V, A), então c é **simétrica** se $c(uv) = c(vu)$ para todo arco uv. O **maior custo** de um arco será denotado por C, ou seja, $C := \max\{c(uv) : uv \in A\}$. No grafo da figura 2.1 temos que $C = 7$. C

A **distância** de um vértice s a um vértice t é o menor custo de um caminho de s a t. A distância de s a t em relação a c será denotada por $\text{dist}_c(s, t)$, ou simplesmente, $\text{dist}_c(s,t)$ quando a função custo estiver subentendida, por $\text{dist}(s, t)$ denota a distância de s a t. $\text{dist}(s,t)$

Na figura 2.1 a distância de s a t é 4.

Um problema fundamental em otimização combinatória que tem um papel de destaque nesta dissertação é o **problema do caminho mínimo**, denotado por CM:

CM **Problema** CM(V, A, c, s, t): Dado um grafo (V, A), uma função custo c e dois vértice s e t, encontrar um caminho de custo mínimo de s a t.

Na literatura essa versão é conhecida como *single-pair shortest path problem*. O celebrado algoritmo de Edsger Wybe Dijkstra [8], apresentado na seção 2.5, resolve o problema do caminho mínimo.

2.2 Funções potenciais e critério de otimalidade

Como é possível provar que um dado caminho de um vértice s a um vértice t é de custo mínimo? Algoritmos para o CM fornecem certificados de otimalidade de suas respostas. Esses certificados vêm de dualidade de programação linear. De fato, o seguinte programa linear, que chamamos de primal, é uma relaxação do problema do caminho mínimo: encontrar um vetor x indexado por A que

$$\begin{array}{rlcrl} \text{minimize} & cx & & & \\ \text{sob as restrições} & x(\delta^+(s)) - x(\delta^-(s)) & = & 1 & \\ & x(\delta^+(t)) - x(\delta^-(t)) & = & -1 & \\ & x(\delta^+(v)) - x(\delta^-(v)) & = & 0 & \text{para cada } v \text{ em } V \setminus \{s,t\} \\ & x[uv] & \geq & 0 & \text{para cada } uv \text{ em } A. \end{array}$$

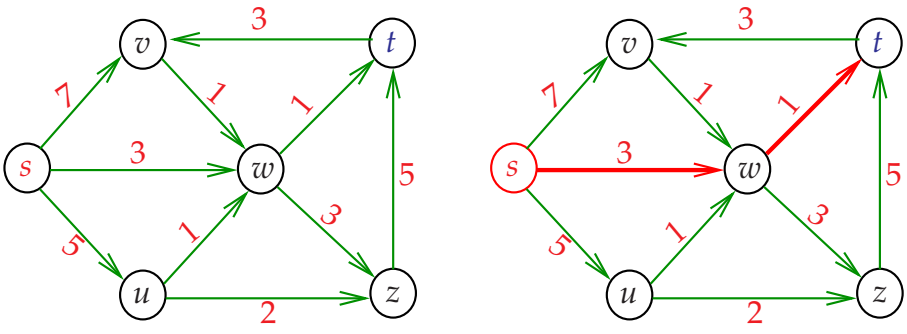


Figura 2.1: Um grafo com custos nos arcos. O custo do caminho $\langle s, u, w, z, t\rangle$ é 14. À direita o caminho de custo mínimo $\langle s, w, t\rangle$ está em destaque.

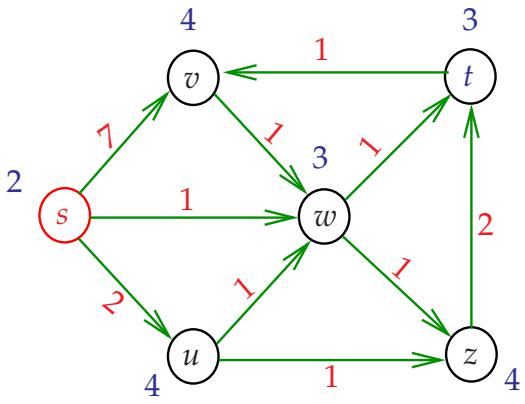


Figura 2.2: (a) Um grafo com custos nos arcos. e um c-potencial. Os números próximos aos vértices são os seu ponteciais.

De fato, cada vetor característico de um caminho se s a t é uma solução viável do problema primal.

O respectivo problema dual consiste em encontrar um vetor y indexado por V que

$$\begin{array}{rl} \text{maximize} & y(t) - y(s) \\ \text{sob as restrições} & y(v) - y(u) \leq c(uv) \quad \text{para cada } uv \text{ em } A. \end{array}$$

Se um vértice t não é acessível a partir de s um algoritmo pode, para comprovar este fato, devolver uma parte S de V tal que $s \in S$, $t \notin S$ e não existe uv com u em S e v em $V \setminus S$, ou seja $A(S) = \emptyset$. Este seria um certificado combinatória de **não-acessibilidade** de t por s. Entretanto, os certificados fornecidos pelos algoritmos, baseados em funções potencial, serão um atestado compacto para certificar ambos: a otimalidade dos caminhos fornecidos, e a não acessabilidade de alguns vértices por s.

Uma **função-potencial** é uma função de V em $\mathbb{Z}$. Se y é uma função-potencial e c é uma função-custo, então, dizemos que y é um *c*-**potencial** se

$$y(v) - y(u) \leq c(uv) \text{ para cada arco } uv \text{ em } A \text{ (figura 2.2).}$$

Limitantes inferiores para custo de caminhos são obtidos através de c-potenciais. Este fato está no lema a seguir, que é uma particularização do conhecido lema da dualidade de programação linear [11].

Lema 2.1 (lema da dualidade): *Seja (V, A) um grafo e c uma função-custo sobre V. Para todo caminho P com início em s e término em t e todo c-potencial y*

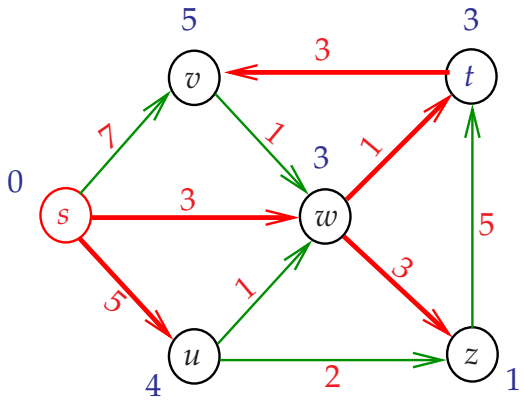


Figura 2.3: Um grafo com custos nos arcos e um potencial que certifica que qualquer um dos caminhos com arcos em destaque têm custo mínimo.

vale que

$$c(P) \geq y(t) - y(s).$$

Demonstração: Suponha que P é o caminho $\langle s = v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k = t \rangle$. Temos que

$$\begin{aligned} c(P) &= c(\alpha_1) + \ldots + c(\alpha_k) \\ &\geq (y(v_1) - y(v_0)) + (y(v_2) - y(v_1)) + \ldots + (y(v_k) - y(v_{k-1})) \\ &= y(v_k) - y(v_0) = y(t) - y(s). \end{aligned}$$

∎

Do lema 2.1 tem-se imediatamente os seguintes corolários.

Corolário 2.2 (condição de inacessabilidade): *Se (V, A) é um grafo, c é uma função-custo, y é um c-potencial e s e t são vertices tais que*

$$y(t) - y(s) \geq nC + 1$$

então, t não é acessível a partir de s ∎

Corolário 2.3 (condição de otimalidade): *Seja (V, A) um grafo e c é uma função-custo. Se P é um caminho de s a t e y é um c-potencial tais que $y(t) - y(s) = c(P)$, então P é um caminho que tem custo mínimo.* ∎

2.3 Representação de caminhos

Uma maneira compacta de representar caminhos de dado vértice até cada um dos demais vértices de um grafo é através de uma função-predecessor. Uma **função-predecessor** é uma função "parcial" ψ de V em V tal que, para cada v em V,

$$\psi(v) = \text{NIL} \quad \text{ou} \quad (\psi(v), v) \in A$$

Se (V, A) é um grafo, ψ uma função predecessor sobre V e $v_0, v_1, \ldots, v_k$ são vértices tais que

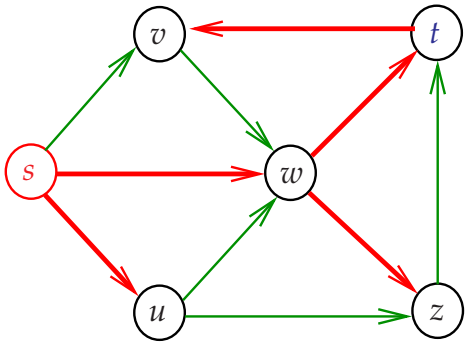
(1) $v_0 = \psi(v_1), v_2 = \psi(v_3), \ldots, v_{k-1} = \psi(v_k)$; e

(2) $\alpha_i := v_{i-1}v_i$ está em A para $i = 1, \ldots, k$.

então dizemos que $\langle v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k \rangle$ é um **caminho determinado por** ψ.

Seja ψ uma função-predecessor e $\Psi := \{uv \in A : u = \psi(v)\}$. Dizemos em (V, Ψ) é o **grafo de predecessores**.

Os algoritmos descritos nesta dissertação utilizam funções predecessor para compactamente representar todos os caminhos de custo mínimo a partir de um dado vértice. Conforme ilustrado na figura 2.4.



vértice	ψ
s	s
w	s
u	s
t	w
z	w
v	t

Figura 2.4: Representação de caminhos através da função-predecessor ψ com vértice inicial s. Os arcos em destaque formam uma arborecência. A tabela ao lado mostra os valores de ψ.

2.4 Examinando arcos e vértices

Algoritmos para encontrar caminhos mínimos mantém, tipicamente, além de uma função-predecessor, uma função-potencial. O valor desta função-potencial para cada vértice é um limitante inferior para o custo dos caminhos que tem como origem o vértice s, como mostra o lema da dualidade. Esta função é intuitivamente interpretada como uma **distância tentativa** a partir de s.

Seja y uma função-potencial e ψ uma função-predecessor. Uma operação básica envolvendo as funções ψ e y é **examinar um arco** (*relaxing* [7], *labeling step* [27]). Examinar um arco uv consiste em verificar se y respeita c em uv e, caso não respeite, ou seja,

$$y(v) - y(u) > c(uv) \text{ ou, equivalentemente } y(v) > y(u) + c(uv)$$

fazer

$$y(v) \leftarrow y(u) + c(uv) \text{ e } \psi(v) \leftarrow u.$$

Intuitivamente, ao examinar um arco uv tenta-se encontrar um "atalho"para o caminho de s a v no grafo de predecessores, passando por uv. O passo de examinar uv pode diminuir o valor da distância tentativa dos vértices v e atualizar o predecessor, também tentativo, de v no caminho de custo mínimo de s a v.

```
EXAMINE-ARCO (uv)    ▷ examina o arco uv
1   se y(v) > y(u) + c(u,v)
2       então y(v) ← y(u) + c(uv)
3             ψ(v) ← u
```

O consumo de tempo para examinar um arco é constante.

Outra operação básica é **examinar um vértice**. Se u é um examinar um consiste em examinar todos os arcos da forma uv. Em linguagem algorítmica tem-se

```
EXAMINE-VÉRTICE (u)    ▷ examina o vértice u
1   para cada uv em A(u) faça
2       EXAMINE-ARCO (uv)
```

ou ainda, de uma maneira expandida

```
EXAMINE-VÉRTICE (u)    ▷ examina o vértice u
1   para cada uv em A(u) faça
2       se y(v) > y(u) + c(uv)
3           então y(v) ← y(u) + c(uv)
4                 ψ(v) ← u
```

O consumo de tempo para examinar um vértice é proporcional ao número de arcos com ponta inicial no vértice.

2.5 Algoritmo de Dijsktra

Nesta seção é descrito o celebrado algoritmo de Edsger Wybe Dijkstra [8] que resolve o problema do caminho mínimo

A idéia geral do algoritmo de Dijkstra para resolver o problema é a seguinte. O algoritmo é iterativo. No início de cada iteração tem-se uma partição S e Q dos conjuntos vertices. O algoritmo conhece caminhos de s a cada vértice em S e a uma parte dos vértices em Q. Para os vértice em S o caminho conhecido tem custo mínimo. Cada iteração consiste em remover um vértice apropriado de Q, incluí-lo S e examiná-lo, atualizando, eventualmente, o custo dos caminhos a alguns vértices em Q.

O algoritmo recebe um grafo (V, A), uma função-custo c de A em $\mathbb{Z}_{\geq}$ e um vértice s e devolve uma função-predecessor ψ e uma função-potencial y que respeita c tais que, para cada vértice t, se t é acessível a partir de s, então ψ determina um caminho de s a t que tem comprimento $y(t) - y(s)$, caso contrário $y(t) - y(s) = nC + 1$, onde $C := \max\{c(uv) : uv \in A\}$. Da condição de otimalidade (corolário 2.3) tem-se que ψ determina um caminho de s a um vértice t, então este caminho tem custo mínimo. Por outro lado, a condição de inacessabilidade (corolário 2.2) diz que se y é um c-potencial com $y(t) - y(s) = nC + 1$, então não existe caminho de s a t. A correção do algoritmo de Dijkstra fornecerá a recíproca dessas condições.

A descrição a seguir é a mesma das notas de aula de Feofiloff [12].

```
DIJKSTRA (V, A, c, s)    ▷ c ≥ 0
1   para cada v em V faça
```

```
2       y(v) ← nC + 1    ▷ nC + 1 faz o papel de ∞
3       ψ(v) ← NIL
4   y(s) ← 0
5   Q ← N    ▷ Q func. como uma fila com prioridades
6   enquanto Q ≠ ⟨⟩ faça
7       retire de Q um vértice u com y(u) mínimo
8       EXAMINE-VÉRTICE (u)
9   devolva ψ e y
```

A versão mais expandida é

```
DIJKSTRA (V, A, c, s)    ▷ c ≥ 0
 1  para cada v em V faça
 2      y(v) ← nC + 1    ▷ nC + 1 faz o papel de ∞
 3      ψ(v) ← NIL
 4  y(s) ← 0
 5  Q ← N    ▷ Q func. como uma fila com prioridades
 6  enquanto Q ≠ ⟨⟩ faça
 7      retire de Q um vértice u com y(u) mínimo
 8      para cada uv em A(u) faça
 9          se y(v) > y(u) + c(uv) então
10              y(v) ← y(u) + c(uv)
11              ψ(v) ← u
12  devolva ψ e y
```

Simulação

A seguir vemos ilustrada a simulação do algoritmo para a chamada DIJKSTRA (s). Na figura (a) vemos o grafo (V, A) dado, onde o número em vermelho próximo a cada arco indica o seu custo. Por exemplo, $c(tv) = 0$ e $c(sw) = 4$. Nas ilustrações os vértices com interior azul claro são aqueles que estão em Q. Assim, na figura (b) vemos que

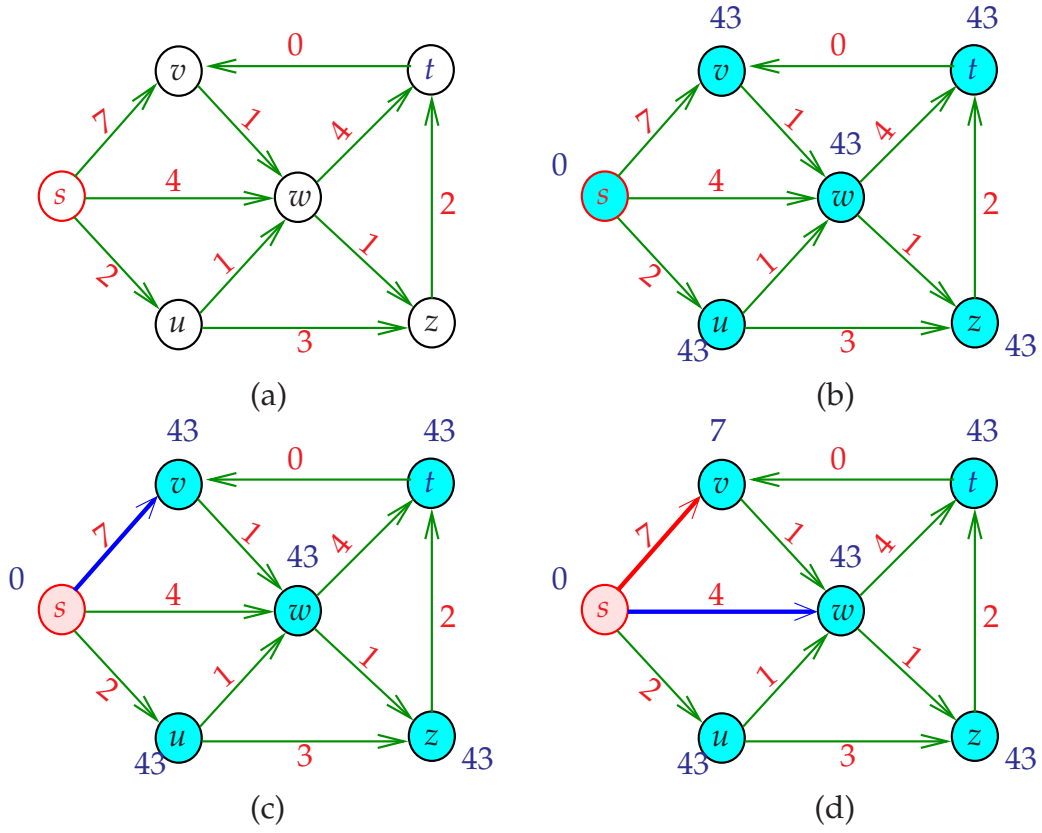
todos os vértices foram inseridos em Q. A função-potencial y é indicada pelos números em azul próximos cada vértice. Assim, na figura (b), vemos que $y(s) = 0$ e o potencial dos demais vértices é $n \times C + 1 = 6 \times 7 + 1 = 43$.

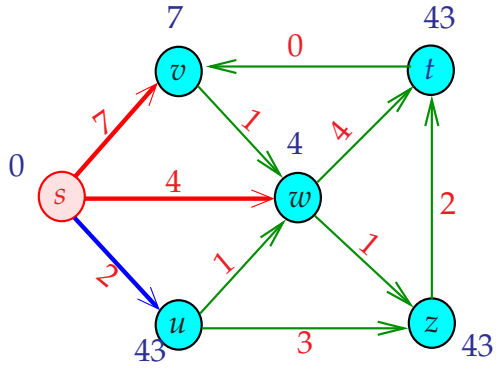
Durante a simulação, o vértice sendo examinado têm a cor rosa no seu interior, enquanto o arco sendo examinado é mais espesso e tem a cor azul. Por exemplo, na figura (c) o vértice sendo examinado é s e o arco sendo examinado é sv.

Os vértices que já foram examinados e, portanto estão em S têm a cor verde em seu interior. Na figura (j) vemos que w está sendo examinado, s e u são os vértices em S e os demais estão em Q.

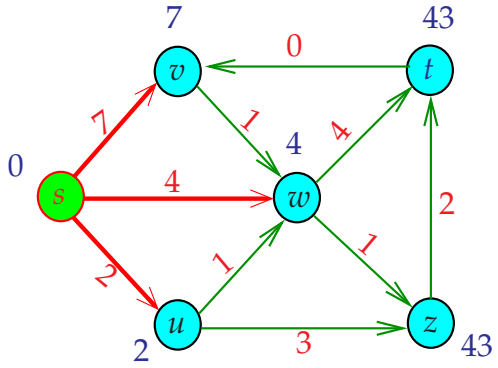
Os arcos já examinados são espessos e têm a cor vermelha ou são finos e têm a cor azul escura. Na figura (k) vemos que o arco wz está sendo examinado e os arcos já examinados são sv, sw, su, uw, uz e wt.

Os arcos em vermelho são os que formam o grafo de predecessores com raiz s. Na figura (k) os arcos do grafo de predecessores são sv, su, uw, e wt.

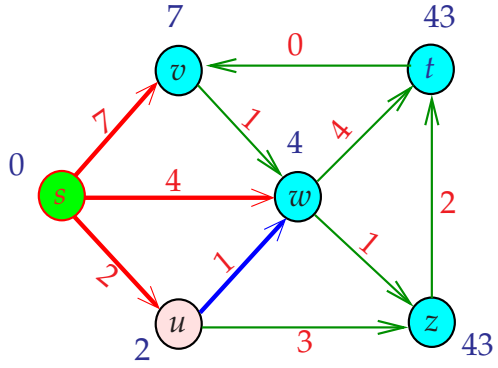



7
43
v
t
0
7
1
4
4
s
w
2
4
2
1
1
u
z
43
3
43

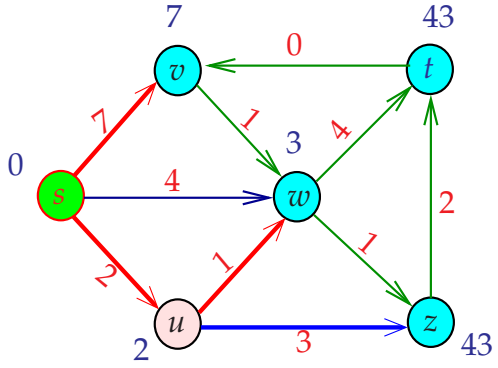
(e)


7
43
v
t
0
7
1
4
4
s
w
2
4
2
1
1
u
z
2
3
43

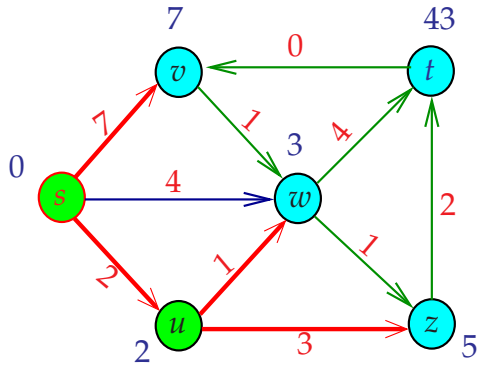
(f)


7
43
v
t
0
7
1
4
4
s
w
2
4
2
1
1
u
z
2
3
43

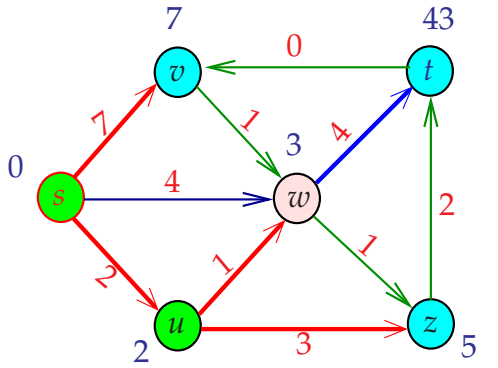
(g)


7
43
v
t
0
7
1
3
4
s
w
2
4
2
1
1
u
z
2
3
43

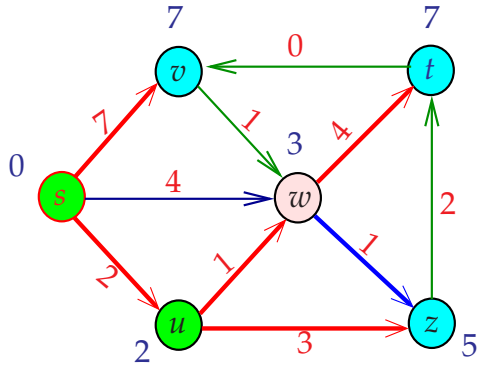
(h)


7
43
v
t
0
7
1
3
4
s
w
2
4
2
1
1
u
z
2
3
5

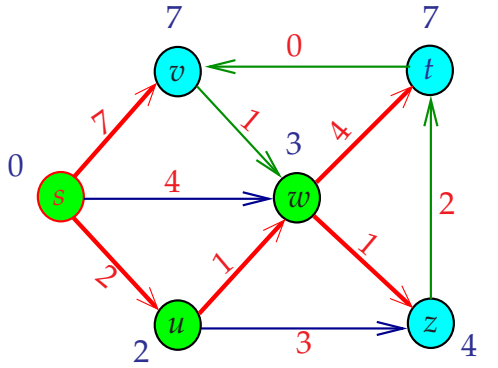
(i)


7
43
v
t
0
7
1
3
4
s
w
2
4
2
1
1
u
z
2
3
5

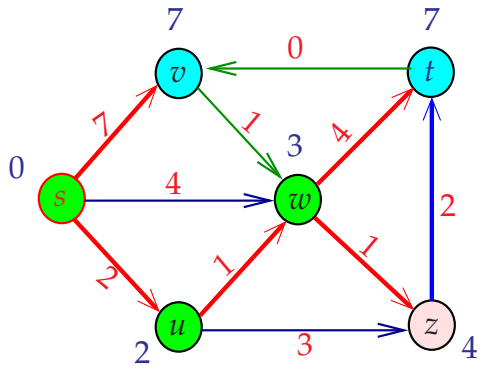
(j)


7
7
v
t
0
7
1
3
4
s
w
2
4
2
1
1
u
z
2
3
5

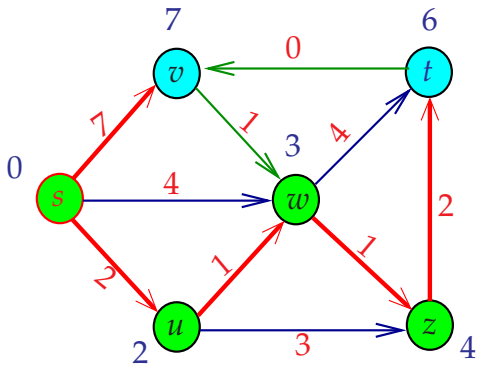
(k)


7
7
v
t
0
7
1
3
4
s
w
2
4
2
1
1
u
z
2
3
4

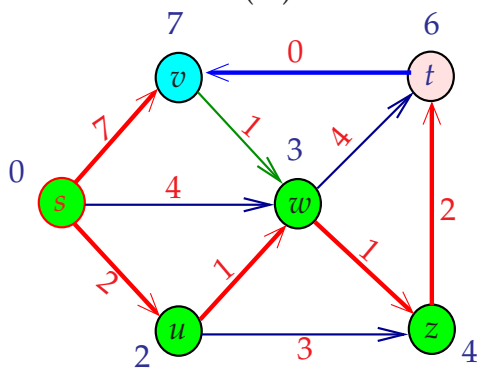
(l)



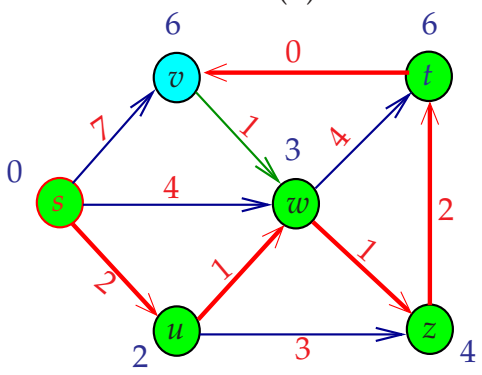
(m)



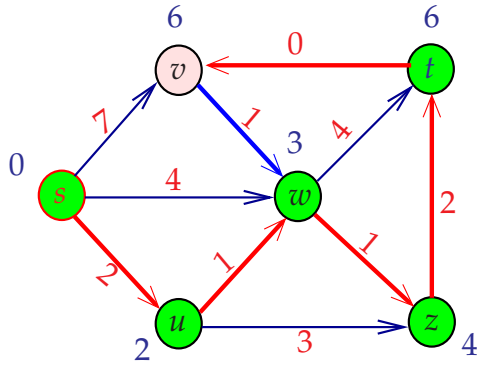
(n)



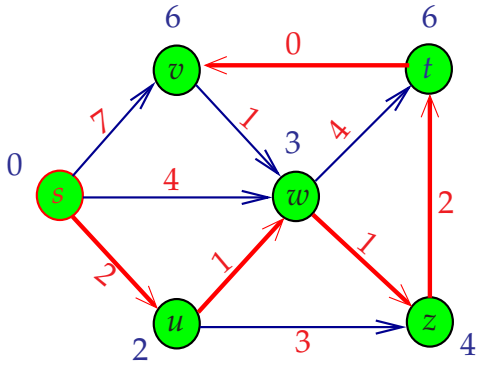
(o)



(p)



(q)



(r)

Correção

A correção do algoritmo de Dijkstra baseia-se nas demonstrações da validade de uma série de relações invariantes, enunciadas a seguir. Estas relações são afirmações envolvendo os dados do problema V, A, c e s e os objetos y, ψ, S e Q. As afirmações são válidas no início de cada iteração do algoritmo e dizem como estes objetos se relacionam entre si e com os dados do problema.

Na linha 6, antes da verificação da condição "$Q \neq \langle\rangle$" valem as seguintes invariantes:

(dk0) para cada arco pq no grafo de predecessores tem-se $y(q) - y(p) = c(pq)$;

(dk1) $\psi(s) =$ NIL e $y(s) = 0$;

(dk2) para cada vértice v distinto de s, $yr(v) < nC + 1 \Leftrightarrow pi(v) \neq$ NIL;

(dk3) para cada vértice v, se $\psi(v) \neq$ NIL então existe um caminho de s a v no grafo de predecessores.

(dk4) para cada arco pq com $y(q) - y(p) > c(pq)$ tem-se que p e q estão Q;

(dk5) (monotonicidade) para quaisquer u em $V - Q$, v em Q vale que

$$y(u) \leq y(v).$$

Teorema 2.4 (da correção): *Dado um grafo (V, A), uma função custo c e um vértice s o algoritmo* DIJKSTRA *corretamente encontra um caminho de custo mínimo de s a t, para todo vértice t acessível a partir de s.*

Demonstração: Como Q é vazio no início da última iteração, então devido a (dk4) a função y é um c-potencial. Se $y(t) < nC + 1$ então, por (dk2), vale que $\psi(t) \neq$ NIL. Logo, de (dk3), segue que existe um st-caminho P no grafo de predecessores. Desta forma, (dk0) e (dk1) implicam que

$$c(P) \geq y(t) - y(s) = y(t).$$

Da condição de otimalidade, concluímos que P é um st-caminho (de custo) mínimo.

Já, se $y(t) = nC + 1$, então (dk1) implica que $y(t) - y(s) = nC + 1$ e da condição de inexistência concluímos que não existe caminho de s a t no grafo (V,A).

Concluímos portanto que o algoritmo faz o que promete. ∎ ∎

Consumo de tempo

As duas seguintes operações são as principais responsáveis pelo consumo de tempo assintótico do algoritmo:

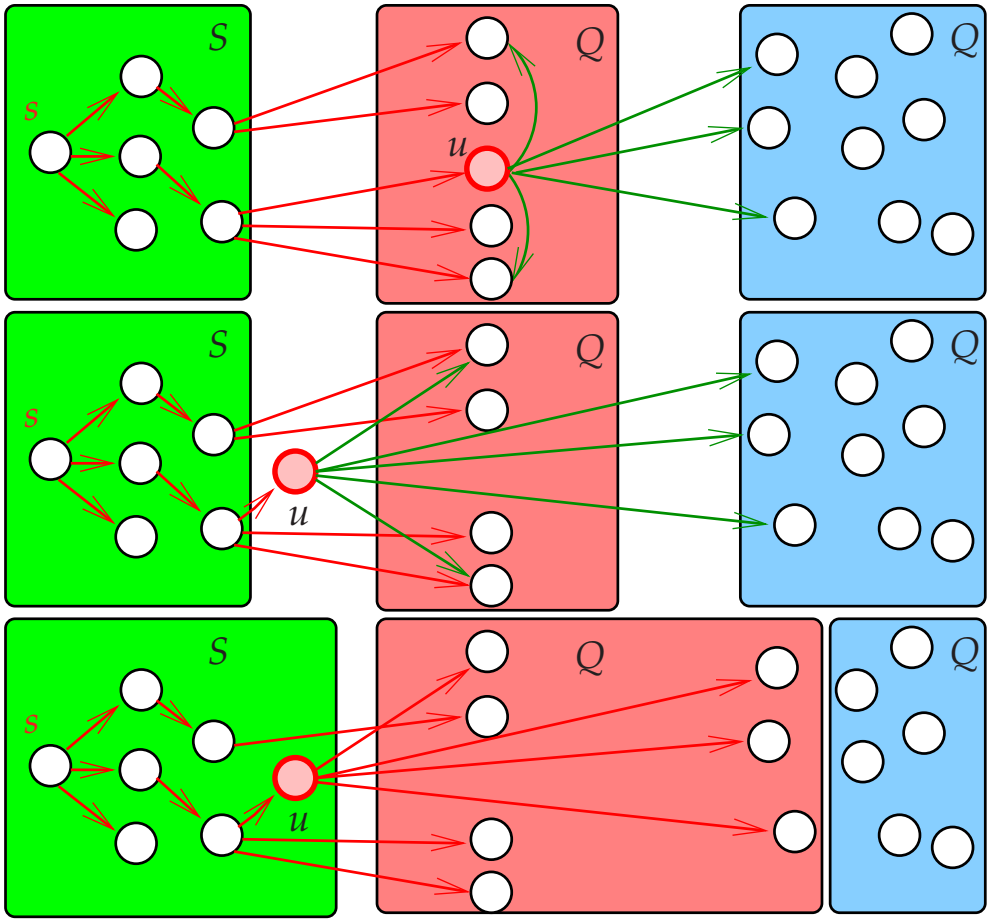


Figura 2.5: Ilustração de uma iteração do algoritmo DIJKSTRA. O arcos em vermelho formam o grafo de predecessores.

Escolha de um vértice com potencial mínimo. Cada execução desta operação gasta tempo $O(n)$. Como o número de ocorrências do caso 2 é no máximo n, então o tempo total gasto pelo algoritmo para realizar essa operação é $O(n^2)$.

Atualização do potencial. Ao examinar um arco o algoritmo eventualmente diminui o pontencial da ponta final. Essa atualização de potencial é realizada não mais que $|A(u)|$ para examinar o vértice u. Ao todo, o algoritmo pode realizar essa operação não mais que $\sum_{u \in V} |A(u)| = m$ vezes. Desde que cada atualização seja feita em tempo constante, o algoritmo requer uma quantidade de tempo proporcional a m para atualizar potenciais.

O número de iterações é $< n$.

linha	consumo de todas as execuções da linha
1-3	$\mathrm{O}(n)$
4	$\mathrm{O}(1)$
5	$\mathrm{O}(n)$
6	$n\,\mathrm{O}(1) = \mathrm{O}(n)$
7	$n\,\mathrm{O}(n) = \mathrm{O}(n^2)$
8-11	$m\,\mathrm{O}(1) = \mathrm{O}(m)$
12	$\mathrm{O}(n)$
total	$\mathrm{O}(1) + 4\,\mathrm{O}(n) + \mathrm{O}(m) + \mathrm{O}(n^2)$ $= \mathrm{O}(n^2)$

Assim, o consumo de tempo do algoritmo no pior caso é $O(n^2 + m) = O(n^2)$. O teorema abaixo resume a discussão.

Teorema 2.5 (consumo de tempo): *O algoritmo* DIJKSTRA *quando executado em um grafo com n vértices e m arcos, consome tempo $O(n^2)$.* ∎

Para grafos densos, ou seja, grafos onde $m = \Omega(n^2)$, o consumo de tempo de $O(n^2)$ do algoritmo de Dijkstra é ótimo, pois, é necessário que todos os arcos do grafo sejam examinados. Entretanto, se $m = O(n^{2-\epsilon})$ para algum ϵ positivo, existem métodos sofisticados, como o heap de Johnson [18], o fibonacci heap de Fredman e Tarjan [13], que permitem diminuir o tempo gasto para encontrar um vértice com potencial mí-

nimo, gerando assim implementações que consomem menos tempo para resolver o problema.

2.6 Dijkstra e filas de prioridades

A maneira mais popular para implementar o algoritmo de Dijkstra é utilizando uma fila de prioridades para representar Q, onde a prioridade de cada vértice v é o seu potencial $y(v)$. A descrição do algoritmo de Dijkstra logo a seguir faz uso das operações BUILD-MIN-HEAP EXTRACT-MIN e DECREASE-KEY, especificadas na secão 1.4.

```
HEAP-DIJKSTRA (V, A, c, s)    ▷ c ≥ 0
 1   para cada v em V faça
 2       y(v) ← nC + 1    ▷ nC + 1 faz o papel de ∞
 3       ψ(v) ← NIL
 4   y(s) ← 0
 5   Q ← BUILD-MIN-HEAP (V)    ▷ Q é um min-heap
 6   enquanto Q ≠ ⟨⟩ faça
 7       u ← EXTRACT-MIN (Q)
 8       para cada uv em A(u) faça
 9           custo ← y(u) + c(uv)
10           se y(v) > custo então
11               DECREASE-KEY(custo, v, Q)
12               ψ(u) ← v

13   devolva ψ e y
```

O número de iterações é $< n$.

linha	consumo de todas as execuções da linha
1-4	$O(n)$
5	$O(n)$
6	$n\,O(1) = O(n)$
7	$n\,O(\lg n) = O(n \lg n)$
8-10	$m\,O(1) = O(m)$
11	$m\,O(\lg n) = O(m \lg n)$
12	$m\,O(1) = O(m)$
13	$O(n)$
total	$4\,O(n) + 2\,O(m) + O(n \lg n) + O(m\ \lg n)$ $= O(m \lg n)$ (supondo (V, A) conexo)

O teorema a seguir resume o número de operação feitas pela implementação acima para manipular a fila de prioridades que representa Q.

Teorema 2.6 (número de operações): *O algoritmo de Dijkstra, quando executado em um grafo com n vértices e m arcos, realiza uma seqüência de n operações* INSERT, *n operações* EXTRACT-MIN *e no máximo m operações* DECREASE-KEY. ∎

O consumo de tempo do algoritmo Dijkstra pode variar conforme a implementação de cada uma dessas operações da fila de prioridade: INSERT, DELETE e DECREASE-KEY.

Existem muitos trabalhos envolvendo implementações de filas de prioridade com o intuito de melhorar a complexidade do algoritmo de Dijkstra. Para citar alguns exemplos temos [2, 5, 13].

As estruturas de dados utilizadas na implementação das filas de prioridade podem ser divididas em duas categorias, conforme as operações elementares utilizadas:

(1) (modelo de comparação) estruturas baseadas em comparações; e

(2) (modelo RAM) estruturas baseadas em "bucketing".

Bucketing é um método de organização dos dados que particiona um conjunto em partes chamadas **buckets**. No que diz respeito ao algoritmo de Dijkstra, cada bucket

agrupa vértices de um grafo relacionados através de prioridades, que nesse caso, são os potenciais.

A tabela a seguir resumo o consumo de tempo de várias implementações de um min-heap e o respectivo consumo de resultante para o algoritmo de Dijkstra [7].

	heap	D-heap	fibonacci heap	bucket heap	radix heap
BUILD-MIN-HEAP	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(\log(nC))$
EXTRACT-MIN	$O(\log n)$	$O(\log_D n)$	$O(\log n)$	$O(C)$	$O(\log(nC))$
DECREASE-KEY	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(1)$
Dijkstra	$O(m \log n)$	$O(m \log_D n)$	$O(m + n \log n)$	$O(m + nC)$	$O(m + n \log(nC))$

2.7 Dijkstra e ordenação

O problema do caminho mínimo, na sua forma mais geral, ou seja, aceitando custos negativos, é NP-difícil; o problema do caminho hamiltoniano pode facilmente ser reduzido a este problema. Devido a relação invariante da monotonicidade (dk5) tem-se que:

> O algoritmo DIJKSTRA retira os nós da fila Q na linha 7 do algoritmo em ordem não-decrescente das suas distância a partir de s.

Conclusão: o consumo de tempo do algoritmo DIJKSTRA é $\Omega(n \log n)$

Fredman e Tarjan [13] observaram que como o algoritmo de Dijkstra examina os vértices em ordem de distância a partir de s (invariante (dk3)) então o algoritmo está, implicitamente, ordenando estes valores. Assim, qualquer implementação do algoritmo de Dijkstra para o modelo comparação adição realiza, no pior caso, $\Omega(n \log n)$ comparações. Portanto, qualquer implementação do algoritmo para o modelo de comparação-adição faz $\Omega(m + n \log n)$ operações elementares.

2.8 Implementação de Dijkstra no JUNG

Como dissemos anteriormente, a biblioteca JUNG contém implementados algoritmos para diversos problemas em grafos. Um desses algoritmos é o desenvolvido por Dikstra para resolver o problema do caminho mínimo em grafos sem custos negativos. Participam da implementação uma série de classes e interfaces que permitem ao

usuário reaproveitar parte do código na criação de versões modificadas do mesmo.

Começaremos pela interface *Distance* cujo objetivo é definir métodos para classes que calculam distãncia entre dois vértices. Possui dois métodos:

Number getDistance(ArchetypeVertex source, ArchetypeVertex target) Responsável por retornar a distância de um caminho ligando o vértice *source* ao *target*. O retorno na forma de *Number* permite que os tipos númericos: *byte, double, float, int, long e short*, sejam usados indistintamente. Fica a cargo do usuário saber o tipo de dado armazenado para posterior obtenção.

Map getDistanceMap(ArchetypeVertex source) Responsável por retornar um mapeamento onde a chave representa um vértice acessível a partir do *source* e o valor corresponde à distância de um caminho até ele partindo de *source*.

A interface *Distance* é implementada pela classe *DijkstraDistance* cujo objetivo é calcular distâncias entre os vértices usando o algortimo de Dijkstra. Além disso, permite que resultados parcias, - caminhos e distâncias, sejam armazenados para reutilização posterior. Descreveremos os seus métodos principais bem como os derivados da interface *Distance*.

Métodos derivados da interface *Distance*:

Number getDistance(ArchetypeVertex source, ArchetypeVertex target) Retorna a distância do menor caminho de *source* a *target*. Caso *target* não seja acessível retorna *null*.

Map getDistanceMap(ArchetypeVertex source) Retorna o mapeamento como descrito na interface *Distance*, com a diferença de que os vértices do mapeamento, quando percorridos por um *iterator* serão obtidos em ordem crescente de distância.

Métodos usados para melhorar o desempenho pelo aplicação de certas restrições:

LinkedHashMap getDistanceMap(ArchetypeVertex source, int n) Subrotina do método *getDistanceMap* cujo objetivo é calcular as distâncias entre o vértice *source* e os *n* vértices mais próximos dele, retornando esta informação na forma de um mapeamento, como o do método *Map getDistanceMap(ArchetypeVertex source)*.

setMaxDistance(double maxDist) Limita a distância máxima para alcançar um vértice no valor de *maxDist*. Desta maneira, vértices, cujas menores distâncias para

serem alcançados a partir de um vértice sejam superiores a *maxDist*, serão considerados inalcançáveis.

setMaxTargets(int maxDestinos) Limita o número de vértices cujas distâncias mínimas devam ser calculadas. Retornando à descrição do algoritmo de Dijkstra da seção 2.5, isto equivale a limitar o número de elementos do conjunto S ao valor *maxDestinos*.

O algoritmo de Dijkstra está implementado em duas partes complementares: uma rotina iterativa, como a descrita na seção 2.5, e algumas estruturas de dados. Como dito anteriormente, o consumo de tempo do algoritmo depende fortemente da estrutura de dados utilizada no armazenamento dos vértices ainda não analisados, ou seja, no conjunto Q. No JUNG, a estrutura utilizada foi um *heap* binário armazenada na forma de um *array*. Os principais métodos usados na manipulação de um *heap* estão implementados nas seguintes rotinas:

add(Object o) Insere o objeto *o* no *heap*.

Object pop() Retorna e retira o menor elemento do *heap*.

Object peek() Apenas retorna o menor elemento.

update(Object o) Informa ao *heap* que a chave do elemento *o* foi alterada, de modo que o *heap* precisa ser atualizado.

Para que o *heap* possa ser construído e atualizado é preciso que os seus elementos tenham uma ordenação. Por isso, a classe *MapBinaryHeap*(nome da classe que implementa a estrutura de *heap* no JUNG), possui construtores que permitem definir um *Comparator* a ser utilizado. Caso nenhum *comparator* seja passado, utiliza-se o padrão, que nada mais faz que tentar comparar os objetos, devendo estes implementarem a interface *Comparable*. Lembramos que muitas classes do JavaSDK já implementam a interface *Comparable*, por exemplo: *Integer, Double, BigInteger*, entre outras. Assim, poderíamos criar um *heap* com elas sem a necessidade de informar um *Comparator*.

O *heap* no JUNG não é implementado apenas com o uso de um *array*. O autor optou por armazenar referências dos objetos contidos no *heap* num *HashMap*, onde a chave é o proprio objeto e o valor associado corresponde à posição do objeto no *heap*, permitindo que o método *update* localize em $O(1)$(consumo de tempo para a localização de um

elemento num *hash*) a posição no heap do objeto cuja chave fora alterada, para em seguida atualizar o *heap*.

Agora, vamos nos ater ao método principal, aquele que realmente calcula as menores distâncias de uma origem aos outros vértices:

LinkedHashMap singleSourceShortestPath(ArchetypeVertex source, Set targets, int numDests).

O primeiro parâmetro indica o vértice de origem, a parti do qual as distâncias aos demais serão calculadas. O segundo corresponde a uma lista de vértices de destino. Caso a opção de *cache* esteja habilitada, todos os destinos informados ao método, cujas distãncias já tenham sido calculadas e armazenadas em chamadas anteriores, serão automaticamente excluídos da lista de destinos a serem calculados na chamada corrente. Usar ou não *cache* para armazenar resultados previamente calculados é opcional e pode ser definido tanto nos construtores da classe quanto alterados através do método *enableCaching*. O seu uso garante melhores desempenhos em chamadas sucessivas para obtenção de diversas distâncias ou predecessores, sempre mantendo fixo a origem. No entanto, vale ressaltar que no caso de alterações do grafo, exclusão/adição de arestas e/ou vértices ou até mesmo mudanças no comprimento das arestas, podem invalidar as distâncias previamente calculadas, sendo que fica a cargo do usuário da classe executar uma chamada do método *reset* para que as novas distâncias possam ser retornadas corretamente. As estruturas de dados utilizadas pelo algoritmo estão centralizadas numa classe chamada *SourceData*. Os principais dados armazenados são:

distances : Mapeamento contendo as menores distâncias a partir da origem. A chave é o vértice e o valor armazenado é a menor distância para alcançá-lo a partir do vértice de origem.

estimatedDistances : semelhante ao *distances*, com a diferença de guardar a menor distância até o momento, ou seja, esta distância pode diminuir.

unknownVertices : Conjunto de vértices que ainda não foram analisados.

[1] O uso de uma outra classe no armazenamento desses dados permite que as estruturas utilizadas sejam alteradas através da especialização da classe *SourceData*. Isso será de

[1]Embora haja outros dados, salientamos que ou são auxiliares ou estão relacionados às restrições que visam melhorar empiricamente o desempenho do algoritmo e, por isso, serão omitidas na nossa descrição.

extrema importância quando estudarmos o algoritmo de geracao de *k*-menores caminhos. A rotina começa obtendo o *SourceData*, o qual é indexado pelo vértice de origem. Podem haver tantos quanto o número de vértices do grafo e o seu armazenmaneto em memória entre chamadas sucessivas está vinculado ao uso ou não do *cache*. Caso não exista *SourceData* para o vértice de origem, um novo será criado: as estruturas citadas acima são inicializadas, a distância à origem definida como zero e a origem adicionada a lista *unknowVertices*. A seguir o funcionamento, *grosso modo*, segue a descrição feita na seção 2.5:

1. O vértice com menor custo será retirado da lista de vértices não analisados(*unknownVertices*).

2. Para cada aresta partindo dele, a nova distância será comparada com a anteriormente armazenada em *estimatedDistances*. Se for inferior, o método *update*, da classe *SourceData*, será chamado. Caso não exista distância previamente calculada, o método *createRecord* será invocado.

3. Uma vez que todas as arestas de um vértice foram analisadas, este entra na lista de vértices cujas distância mínimas já foram calculadas: *distances*.

Ao final, teremos a estrutura *distance* devidamente preenchida, e podemos obter as distâncias a partir da origem de todos os vértices alcançáveis.

A classe *DijkstraDistance*, no entanto, não armazena uma lista de predecessores, não permitindo assim que caminhos sejam reconstruídos. Para, além de informar distâncias, permitir reconstrução de caminhos, o autor especializou a classe *DijkstraDistance*, criando a classe *DijkstraShortestPath*. As principais mudanças se referem a quatro métodos que foram adicionados:

Map getIncomingEdgeMap(Vertex origem) : Retorna um mapeamento indexado pelos vértices acessíveis a partir do vértice *origem* e, para cada um destes vértices, armazena o correspondente arco incidente pertencente ao caminho de custo mínimo até ele. O mapeamento é salvo na forma de um *LinkedHashMap* cuja iteração respeita o retorno dos vértices com menores custos.

Edge getIncomingEdge(Vertex source, Vertex target) : Retorna o arco incidente em *target* pertencente ao caminho de custo mínimo cuja ponta inicial é *source*. Usa o método acima como base.

List getPath(Vertex source, Vertex target) : Retorna uma lista de arcos que fazem parte do caminho de custo mínimo com ponta final *source* e ponta final *target*. A lista encontra-se ordenada de acordo com a ordem e que os arcos aparecem no caminho.

Para que esses métodos pudessem funcionar foi preciso mudar especializar a classe *SourceData*, a qual passou a armazenar duas novas estruturas de dados:

Map tentativeIncomingEdges : Um mapeamento indexado pelos vértices acessíveis e os seus respectivos arcos incidentes pertencente ao caminho de custo mínimo corrente. Este arco pode vir a ser substituído caso exista um outro pertencente a um caminho de custo menor que venha a ser calculado posteriormente. Suas entradas sao alteradas durante a chamada da função *update*.

LinkedHashMap incomingEdges : Um mapeamento semelhante ao anterior, mas contendo valores definitivos. Uma vez que um vértice é analisado, uma entrada definitiva é criada em *incomingEdges* contendo a entrada correspondente a este vértice no mapeamento *tentativeIncomingEdges*.

Para maiores detalhes recomendamos a leitura direta do código do JUNG.

CAPÍTULO 3

Problema dos k-menores caminhos

Estão descritos neste capítulo os ingredientes básicos que envolvem o problema dos k-menores caminhos, tais como função comprimento, função potencial, função predecessor, critério de otimalidade, etc. A referência básica para este capítulo é Feofillof [10]

3.1 Caminho mínimo

Uma **função custo** em (V, A) é uma função de A em $\mathbb{Z}_{\geq}$. Se c for uma função custo em (V, A) e uv estiver em A, então $c(u, v)$ será o valor de c em uv. Se P for um passeio em um grafo (V, A) e c uma função custo, denotaremos por $c(P)$ o **custo do caminho** P, ou seja, $c(P)$ é o somatório dos custos de todos os arcos em P. Um passeio P tem **custo mínimo** se $c(P) \leq c(P')$ para todo passeio P' que tenha o mesmo início e término que P. Um passeio de custo mínimo é comumente chamado de **caminho mínimo**.

Um problema fundamental em otimização combinatória que tem um papel de destaque neste projeto é o **problema do caminho mínimo**, denotado por CM:

> **Problema** CM(V, A, c, s, t): Dado um grafo (V, A), uma função custo c e dois vértice s e t, encontrar um caminho de custo mínimo de s a t. CM

Na literatura essa versão é conhecida como *single-pair shortest path problem*. O celebrado algoritmo de Edsger Wybe Dijkstra [8], que foi descrito no capítulo 2 resolve o problema do caminho mínimo.

Denotaremos, quando não houver ambigüidade, por n e m os números $|V|$ e $|A|$, respectivamente. Além disso, representaremos por $T(n, m)$ o consumo de tempo de

uma subrotina genérica para resolver o CM em um grafo com n vértices e m arestas. O algoritmo mais eficiente conhecido para o CM foi projetado por Michael L. Fredman e Robert Endre Tarjan [13] e consome tempo $O(m + n \log n)$. Existe ainda um algoritmo que consome tempo linear *sob um outro modelo de computação* que foi desenvolvido por Mikkel Thorup [28].

Uma **função comprimento** em (V, A) é uma função de A em $\mathbb{Z}_{\geq}$. Se c é uma função comprimento em (V, A) e uv está em A, então, denotaremos por $c(u, v)$ o valor de c em uv. Se (V, A) é um grafo simétrico e c é uma função comprimento em (V, A), então c é **simétrica** se $c(u, v) = c(v, u)$ para todo arco uv. O **maior comprimento** de um arco será denotado por C, ou seja, $C = \max\{c(u, v) : uv \in A\}$.

C

Se P é um passeio em um grafo (V, A) e c é uma função comprimento, denotaremos por $c(P)$ o **comprimento do caminho** P, ou seja, $c(P)$ é o somatório dos comprimentos de todos os arcos em P. Um passeio P tem **comprimento mínimo** se $c(P) \leq c(P')$ para todo passeio P' que tenha o mesmo início e término que P. A **distância** de um vértice s a um vértice t é o menor comprimento de um caminho de s a t. A distância de s a t em relação a c será denotada por $\text{dist}_c(s, t)$, ou simplesmente, quando a função comprimento estiver subentendida, por $\text{dist}(s, t)$ denota a distância de s a t.

$\text{dist}_c(s,t)$

$\text{dist}(s,t)$

3.2 k-menores caminhos

O problema central deste projeto se assemelha muito ao do k-ésimo menor elemento, que é estudado em disciplinas básicas de análise de algoritmos:

k-ÉSIMO

> **Problema** k-ÉSIMO$(\mathcal{S}, k)$: Dado um conjunto $\mathcal{S}$ de números inteiros e um número inteiro positivo k, encontrar o k-ésimo menor elemento de $\mathcal{S}$.

Os algoritmos conhecidos para o problema k-ÉSIMO são facilmente adaptáveis para, além do k-ésimo menor, fornecerem, em tempo linear, os k menores elementos de $\mathcal{S}$ *em ordem crescente*.

A diferença entre o problema k-ÉSIMO e o problema que consideraremos é que o conjunto $\mathcal{S}$ dado é "muito grande" e, portanto, nos é dado de uma maneira compacta, o que torna o problema sensivelmente mais difícil do ponto de vista computacional. Adiante tornamos o problema mais preciso.

Suponha que (V, A) seja um grafo, c uma função custo e s e t dois de seus vértices. Considere o conjunto $\mathcal{P}_{st}$ de todos st-caminhos, ou seja, caminhos de s a t. Uma lista $\langle P_1, \ldots, P_k \rangle$ st-caminhos distintos é de **custo mínimo** se

$$c(P_1) \leq c(P_2) \leq \cdots \leq c(P_k) \leq \min\{c(P) : P \in \mathcal{P} - \{P_1, \ldots, P_k\}\}.$$

De uma maneira mais breve, diremos que $\langle P_1, \ldots, P_k \rangle$ são **k-menores caminhos** (de s a t).

Em termos da teoria dos grafos o problema que foi discutido na introdução é o **problema dos k-menores caminhos**, denotado por k-CM:

> **Problema** k-CM(V, A, c, s, t, k): Dado um grafo (V, A), uma função custo c, dois vértice s e t e um inteiro positivo k, encontrar k-caminhos de custos mínimos de s a t. — k-CM

É evidente que o CM nada mais é que o k-CM com $k = 1$.

O k-CM é, em essência, o problema k-ÉSIMO com $\mathcal{P}_{st}$ no papel do conjunto $\mathcal{S}$. A grande diferença computacional é que o conjunto $\mathcal{P}_{st}$ não é fornecido explicitamente, mas sim de uma maneira compacta: um grafo, uma função custo e um par de vértices. Desta forma, o número de elementos em $\mathcal{P}_{st}$ é potencialmente exponencial no tamanho da entrada, tornando impraticável resolvermos o k-CM utilizando meramente algoritmos para o k-ÉSIMO como subrotina.

Na próxima seção é descrito o método genérico para resolver o k-CM. Este método é um passo intermediário para chegarmos no método desenvolvido por Jin Y. Yen [29] para o k-CM. Antes disto, apresentamos aqui um problema intimamente relacionado ao k-CM e que também é considerado por este projeto:

> **Problema** K-CM(V, A, c, s, t, K): Dado um grafo (V, A), uma função custo c, dois vértice s e t e um inteiro positivo K, encontrar os caminhos de s a t de custos não superiores a K. — K-CM

Formalmente, nesta dissertação, estamos interessados no seguinte **problema do caminho mínimo**:

> **Problema** PCM(V, A, c, s): Dado um grafo (V, A), uma função comprimento c e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V. — PCM

Na literatura essa versão é conhecida como *single-source shortest path problem.*

3.3 Árvores dos prefixos

Descrevemos aqui uma "arborescência rotulada" que de certa forma codifica os prefixos dos caminhos em uma dada coleção. Esta representação será particularmente útil quando, mais adiante, discutirmos o método de Yen. No que segue $\mathcal{Q}$ é uma coleção de caminhos de um grafo e $V(\mathcal{Q})$ e $A(\mathcal{Q})$ são o conjunto dos vértices e o conjunto dos arcos presentes nos caminhos, respectivamente.

Um grafo acíclico (N, E) com $|N| = |E| + 1$ é uma **arborescência** se todo vértice, exceto um vértice especial chamado de **raiz**, for ponta final de exatamente um arco. Será conveniente tratarmos os vértices de uma arborescência por **nós**. Uma arborescência está ilustrada na figura 1.1(c). A raiz dessa arborescência é o nó a. Uma **folha** de uma arborescência é um nó que não é ponta inicial de nenhum arco.

Suponha que (N, E) seja uma arborescência e f uma **função rótulo** que associa a cada nó em N um vértice em $V(\mathcal{Q})$ e a cada arco em E um arco em $A(\mathcal{Q})$. Se

$$R = \langle u_0, e_1, u_1, \ldots, e_t, u_t \rangle$$

for um caminho em (N, E), então

$$f(R) := \langle f(u_0), f(e_1), f(u_1), \ldots, f(e_t), f(u_t) \rangle$$

será uma seqüência de vértices e arcos dos caminhos em $\mathcal{Q}$. Diremos que (N, E, f) é **árvore dos prefixos** de $\mathcal{Q}$ se

(p1) para cada caminho R em (N, E) com início na raiz, $f(R)$ for prefixo de algum caminho em $\mathcal{Q}$; e

(p2) para cada prefixo Q de algum caminho em $\mathcal{Q}$ existir um caminho R em (N, E) com início na raiz tal que $f(R) = Q$; e

(p3) o caminho R do item anterior for único.

Não é verdade que para cada coleção $\mathcal{Q}$ de caminhos em um grafo existe uma árvore dos prefixos de $\mathcal{Q}$. No entanto, se todos os caminhos em $\mathcal{Q}$ tiverem a mesma ponta inicial, então existe uma árvore dos prefixo de $\mathcal{Q}$ e esta é única. Na figura 3.1(b) vemos a ilustração da árvore dos prefixos de quatro caminhos de s a t no grafo da figura 3.1(a). Na árvore da ilustração w, x, y e z são nós e $f(w) = s, f(x) = a, f(y) = d$ e $f(z) = d$.

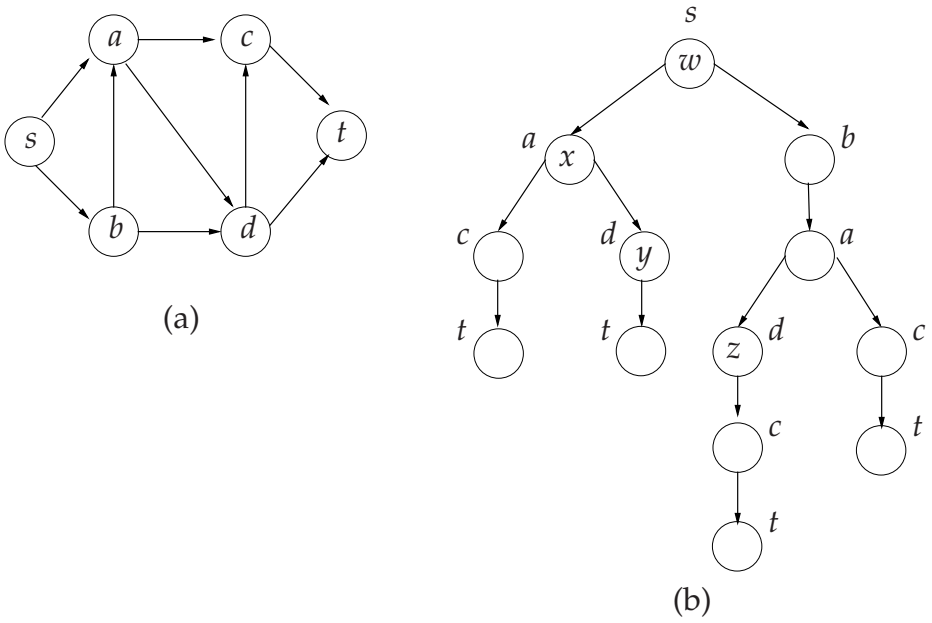


Figura 3.1: (b) mostra a árvore dos prefixos dos caminhos $\langle s, a, c, t\rangle$, $\langle s, a, d, t\rangle$, $\langle s, b, a, c, t\rangle$ e $\langle s, b, a, d, c, t\rangle$ no grafo em (a). Na árvore, um símbolo ao lado de um nó é o rótulo desse nó. Os rótulos dos arcos não estão representados na figura. O símbolo dentro de um nó é o seu nome.

3.4 Método genérico

A descrição que fazemos é, de certa forma, *top-down*. Começaremos com um método genérico que será refinado a cada passo incluindo, convenientemente, algumas subrotinas auxiliares. O nosso interesse aqui é numa descrição mais conceitual em que a correção e o consumo de tempo polinomial do método sejam um tanto quanto evidentes. Não temos a intenção de descrever um algoritmo com o menor consumo de tempo.

O método abaixo recebe um grafo (V, A), uma função custo, dois vértices s e t e um inteiro positivo k e devolve uma lista $\langle P_1, \ldots, P_k \rangle$ de k-menores caminhos de s a t.

Método GENÉRICO (V, A, c, s, t, k)

0 $\mathcal{P} \leftarrow$ *conjunto dos caminhos de* s *a* t

1 ***para*** $i = 1, \ldots, k$ ***faça***

2 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{P}$

3 $\quad \mathcal{P} \leftarrow \mathcal{P} - P_i$

4 ***devolva*** $\langle P_1, \ldots, P_k \rangle$

No início de cada iteração da linha 1 o conjunto $\mathcal{P}$ contém os candidatos a i-ésimo caminho mínimo de s a t. O método de Yen é uma elaboração do método GENÉRICO. Em vez do conjunto $\mathcal{P}$, Yen mantém, pelo menos conceitualmente, uma partição Π de $\mathcal{P}$. Em cada iteração, é escolhido o caminho mais barato dentre um conjunto $\mathcal{L}$ formado por *um* caminho mínimo P_π representante de cada parte π de Π e depois a partição é atualizada.

Método YEN-GENÉRICO (V, A, c, s, t, k)

0 $\Pi \leftarrow \{\{$*conjunto dos caminhos de* s *a* $t\}\}$

1 $\mathcal{Q} \leftarrow \emptyset$

2 ***para*** $i = 1, \ldots, k$ ***faça***

3 $\quad \mathcal{L} \leftarrow \langle P_\pi : P_\pi$ *é caminho mínimo da parte* π *de* $\Pi \rangle$

4 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$

5 $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{P_i\}$

6 $\Pi \leftarrow$ ATUALIZE-GENÉRICO $(V, A, \mathcal{Q})$

7 ***devolva*** $\langle P_1, \ldots, P_k \rangle$

Como veremos, a eficiência do método de Yen dependerá fortemente da estrutura restrita dos caminhos nas partes de Π: cada parte é formada por caminhos que têm um certo prefixo comum.

Seja $\mathcal{P}_{st}$ a coleção dos caminhos de s a t em (V, A). Suponha que $\mathcal{Q}$ seja a lista de caminhos distintos de s a t na linha 5 do método YEN-GENÉRICO. Passamos a descrever a partição Π dos caminhos em $\mathcal{P} := \mathcal{P}_{st} \setminus \mathcal{Q}$. Para isto é conveniente utilizarmos a *árvore dos prefixos* de $\mathcal{Q}$, como foi feito por John Hershberger, Matthew Maxel e Subhash Suri [16].

No que segue suponha que (N, E, f) seja a árvore dos prefixos de $\mathcal{Q}$ e u seja um nó em N. Representaremos por R_u o caminho da raiz a u na árvore. Assim, $f(R_u)$ é o prefixo de um caminho em $\mathcal{Q}$. Por exemplo, na árvore dos prefixos da figura 3.1(b) temos que $R_y = \langle w, wx, x, xy, y \rangle$ e $f(R_y) = \langle s, sa, a, ad, d \rangle$. R_u $f(R_u)$

Seja A_u

$$A_u := \{(f(u), f(w)) : uw \in E\}.$$

e seja π_u o conjunto dos caminhos em $\mathcal{P}$ com prefixo $f(R_u)$ e que não possuem arcos em A_u. Para o exemplo na figura 3.1 temos que π_u

$$A_w = \{sa, sb\}, A_x = \{ac, ad\}, A_y = \{dt\} \text{ e } A_z = \{dc\}$$
$$\pi_w = \emptyset, \pi_x = \emptyset, \pi_y = \{\langle s, a, d, c, t \rangle\}, \text{ e } \pi_z = \{\langle s, b, a, d, t \rangle\}.$$

A partição Π é formada por uma parte π_u para cada vértice u em N, ou seja, Π

$$\Pi := \{\pi_u : u \in N\}.$$

No início de cada iteração da linha 2 o número de partes é certamente não superior a $n \times i$. O algoritmo ATUALIZE-GENÉRICO resume toda a discussão acima.

Algoritmo ATUALIZE-GENÉRICO $(V, A, \mathcal{Q})$

0 $\Pi \leftarrow \emptyset \qquad \mathcal{P} \leftarrow \mathcal{P}_{st} \setminus \mathcal{Q}$

1 $(N, E, f) \leftarrow$ *árvore dos prefixos de* $\mathcal{Q}$

2 ***para cada*** $u \in N$ ***faça***
3 $\quad \pi_u \leftarrow \{$*caminhos em* $\mathcal{P}$ *com prefixo* $f(R_u)$
$\qquad\qquad\qquad\qquad$ *e que não possuem arcos em* $A_u\}$
4 $\quad \Pi \leftarrow \Pi \cup \{\pi_u\}$
5 ***devolva*** Π

Podemos verificar que cada caminho em $\mathcal{Q}$ não pertence a nenhuma parte de Π. Também podemos verificar que cada caminho P em $\mathcal{P}$ está em uma única parte de Π. De fato, seja P' o maior prefixo de P que é prefixo de algum caminho em $\mathcal{Q}$. Pela definição de árvore de prefixos, existe um único caminho R' em (N, E) com início na raiz e tal que $P' = f(R')$. Para o vértice u término de R' temos que P está em π_u e é a única parte que possui P.

Desta forma, no início de cada iteração das linhas 2–6 do método YEN-GENÉRICO, Π é uma partição de $\mathcal{P}$, portanto a correção do método é evidente.

As árvores dos prefixos de duas execuções consecutivas do algoritmo ATUALIZE-GENÉRICO são muito semelhantes: apenas um novo caminho é acrescentado à árvore anterior. Isto, em particular, significa que as partições de duas iterações consecutivas das linhas 2–6 do método YEN-GENÉRICO são muitos semelhantes. Esta observação pode ser utilizada para o algoritmo ATUALIZE-GENÉRICO obter mais eficientemente uma partição a partir da partição anterior.

3.5 Método de Yen

O método que Jin Y. Yen [29] desenvolveu para resolver o k-CM parece ter um papel central entre os algoritmos que foram posteriormente projetados para o k-CM ou mesmo para versões mais restritas do problema [9, 20, 16]. Várias melhorias práticas do método de Yen têm sido implementadas e testadas [3, 15, 23, 24, 26]

Antes de prosseguirmos, mencionamos que o método de Yen foi generalizado por Eugene L. Lawler [21] para problemas de otimização combinatória, contanto que seja fornecida uma subrotina para determinar uma solução ótima sujeita a condição de que certas variáveis têm seus valores fixados. Por exemplo, no caso do método de Yen para o k-CM essa subrotina resolve o seguinte **problema do sub-caminho mínimo**, denotado por SCM:

Problema SCM(V, A, c, s, t, P, F): Dado um grafo (V, A), uma função custo c, dois vértice s e t, um caminho P e uma parte F de A, encontrar um caminho de custo mínimo de s a t que tem P como prefixo e não contém arcos em F. SCM

É evidente que se P não tem início em s então o problema é inviável. Do ponto de vista de método Lawler, o prefixo P e o conjunto F são as 'variáveis' com valores fixados.

Resolver o CM(V, A, c, s, t) é o mesmo que resolver SCM$(V, A, c, s, t, \langle s\rangle, \emptyset)$. Por outro lado, o SCM pode ser solucionado aplicando-se um algoritmo para o CM em um sub-grafo apropriado de (V, A). Desta forma, o CM e o SCM são computacionalmente equivalentes e podem ser resolvidos em tempo $T(n, m)$.

Conceitualmente, o método de Yen é uma elaboração do método YEN-GENÉRICO. No início de cada iteração da linha 2, $\mathcal{L}$ é uma lista dos candidatos a i-ésimo caminho mínimo de s a t. Ao invés da partição Π de $\mathcal{P}$, Yen mantém em $\mathcal{L}$ um caminho mínimo de cada parte de Π. Em cada iteração é escolhido o caminho mais barato entre todos em $\mathcal{L}$ e a partição é *levemente* atualizada.

Método YEN (V, A, c, s, t, k)

1 $\mathcal{L} \leftarrow \{$*um caminho de custo mínimo de* s *a* $t\}$

2 $\mathcal{Q} \leftarrow \emptyset$

3 ***para*** $i = 1, \ldots, k$ ***faça***

4 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$

5 $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{P_i\}$

6 $\quad \mathcal{L} \leftarrow$ ATUALIZE $(V, A, c, \mathcal{Q})$

7 ***devolva*** $\langle P_1, \ldots, P_k\rangle$

Algoritmo ATUALIZE $(V, A, c, \mathcal{Q})$

0 $\mathcal{L} \leftarrow \emptyset$

1 $(N, E, f) \leftarrow$ *árvore dos prefixos de* $\mathcal{Q}$

2 ***para cada*** $u \in N$ ***faça***

3 $\quad P_u \leftarrow$ *caminho de* s *a* t *de custo mínimo com prefixo* $f(R_u)$

$\qquad\qquad$ *e que não possui arcos em* A_u

4 $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{P_u\}$

5 ***devolva*** $\mathcal{L}$

Na linha 3 do algoritmo ATUALIZE, na verdade, estamos resolvendo o problema SCM$(V, A, c, s, t, f(R_u), A_u)$. Assim, o consumo de tempo do algoritmo resultante é $n\, i\, T(n, m))$. Em chamadas consecutivas do algoritmo ATUALIZE, as árvores dos prefixos calculadas são muito semelhantes. De fato, o algoritmo pode ser implementado de tal maneira que o consumo o seu consumo de tempo seja $n\, T(n, m)$.

O método de YEN pode ser implementado de tal maneira que o seu consumo de tempo seja proporcional a $k\, n\, T(n, m)$.

CAPÍTULO 4

Algoritmo de Katoh, Ibaraki e Mine

Neste capítulo trataremos, propriamente dito, do algoritmo de KIM. Para entendê-lo fizemos usos de vários artigos diferentes. Começamos pelo artigo original de Naoki Katoh, Toshihide Ibaraki e H. Mine [20] (KIM), sob o qual a implementação foi feita. O artigo é bem preciso do ponto de vista da implementação, trabalhando com índices e citando até estruturas para implementação. Do ponto de vista do entedimento do algoritmo em termos gerais não refrescou muito.

Uma vez que o artigo de Jin Y. Yen [29] fora citado pelo do KIM, achamos que seria de grande utilidade lê-lo, além do mais, sabemos que o algoritmo de KIM é uma melhoria do de Yen, sendo específico para grafos simétricos. Com a leitura do algoritmo de YEN, e sua simplicidade, começamos a vislumbrar melhor o de KIM e a entendê-lo com mais propriedade.

En seguida encontramos o artigo de John Hershberger, Matthew Maxel e Subhash Suri [16], o qual foi muito elucidativo. Este trata de uma extensão da idéia central do algoritmo de KIM para grafos não-simétricos. O que mais nos chamou a atenção foi a maneira como o algoritmo foi descrito: os autores procuraram trabalhar mais com idéias gerais e lidar com estruturas mais sofisticadas deixando de lado a quantidade exorbitante de índices e descrições de baixo nível apresentadas no artigo original de KIM.

Nosso plano é apresentar o algoritmo de KIM, bem como suas subrotinas e principais idéias, em seguida passaremos a demonstração de correção seguindo então para a análise de desempenho assintótico e, por fim, de modo a ajudar no entendimento, faremos uma simulação mostrando também o código implementado utilizando-se o pacote JUNG.

4.1 Caminhos derivados

O algoritmo de KIM funciona, basicamente, gerando caminhos candidatos em cada iteração. Inicialmente calcula-se o caminho $P1$, menor caminho entre s e t, para tanto utiliza-se o próprio algoritmo de Dijkstra. Em seguida, utilizando-se a rotina 4.2, obtemos o caminho $P2$ que é o menor caminho dentre todos os que desviam de $P1$ em algum vértice. A partir dos caminhos $P1$ e $P2$ são gerados três novos caminhos os quais serão candidatos para $P3$, são eles:

Pa Pa : menor caminho que se desvia de $P2$ em algum momento depois que $P2$ se desviou de $P1$.

Pb Pb : menor caminho que se desvia de $P1$ depois do vértice comum a $P1$ e $P2$.

Pc Pc : menor caminho que se desvia de P1 antes do vértice comum a $P1$ e $P2$.

A figura 4.1 mostra como são esses caminhos. É possível observar que só existem estas três possibilidades para o caminho $P3$, graficamente isto se torna até mais intuitivo. Os caminhos candidatos Pa, Pb e Pc são então colocados na lista de caminhos candidatos e, no início da próxima iteração, o caminho de menor custo é retirado da lista e, tornando-se o $P3$. A partir então dos caminhos $P3$ e seu caminho gerador, o qual chamaremos de caminho pai, faremos o mesmo processo, ou seja, geraremos três caminhos candidatos Pa, Pb, Pc. Os pais de cada caminho candidato são definidos da seguinte maneira:

- O caminho pai de Pa é o $P2$, uma vez que é aquele com o qual compartilha o maior número de vértices consecutivos partindo de s.
- O caminho pai de Pb é o $P1$, uma vez que é aquele com o qual compartilha o maior número de vértices consecutivos partindo de s.
- O caminho pai de Pc é o $P1$, uma vez que é aquele com o qual compartilha o maior número de vértices consecutivos partindo de s.

Durante a execução do algoritmo fica mais fácil identificar os pais de cada caminho, bastando observar qual o último caminho do qual este desvia.

4.2 Árvores Ts e Tt

O algoritmo de KIM está baseado fortemente na geração de duas árvores de menores caminhos, as quais são geradas utilizando-se uma versão um pouco modificada do algortimo de Dijkstra. São duas as mudanças requeridas:

- Rotulação utilizando-se índices ϵ na árvore Ts e ζ na árvore Tt.
- Garantia de que um certo caminho base faça parte de ambas as árvores.

A árvore Ts corresponde a árvore de menores caminhos cuja raiz é s. Ts

A árvore Tt corresponde a árvore de menores caminhos cuja raiz é t. Tt

A rotulação das árvores funciona da seguinte maneira:

Seja $G = (V, A)$ um **grafo** simétrico e $P = \langle u_1, u_2, \ldots, u_n \rangle$ um caminho em G. Definimos a rotulação ϵ dos vértices da árvore Ts da seguinte forma: rotulação ϵ

- Se o vértice x pertencer ao caminho P, então $\epsilon(x)$ corresponde a sua posição no caminho, começando a contagem no 1. No exemplo acima, $\epsilon(u_1) = 1$, $\epsilon(u_2) = 2$ e $\epsilon(u_n) = n$
- Caso contrário, o valor de $\epsilon(x)$ corresponde ao $\epsilon(y)$, onde y representa o vértice pertencente à árvore Ts tal que o arco (x, y) também pertence à árvore Ts.

Graficamente, podemos observar que a rotulação ϵ nada mais é que atribuir a cada vértice não pertencente ao caminho base P o valor de ϵ do seu último vértice comum ao caminho base P. Na figura 4.2(b), temos o caminho base $P = \langle a, d, e \rangle$ cuja rotulação corresponde a sua posição no caminho, ou seja, $\epsilon(a) = 1$, $\epsilon(d) = 2$ e $\epsilon(e) = 3$.

$\epsilon(b) = 1$, uma vez que o ϵ do último vértice pertencente ao caminho P comum aos caminhos P e $\langle a, b \rangle$, na árvore Ts, é 1. O mesmo vale para o vértice c.

Antes de definir a rotulação ζ, vamos definir P_r como o reverso de P, ou seja, $P_r = \langle u_n, u_{n-1}, \ldots, u_1 \rangle$

Definimos a rotulação ζ dos vértices da árvore Tt da seguinte forma: rotulação ζ

- Se o vértice x pertencer ao caminho P_r, então $\zeta(x) = \epsilon(x)$. No exemplo acima, $\zeta(u_1) = 1$, $\zeta(u_{n-1}) = n - 1$ e $\zeta(u_n) = n$.

- Caso contrário, o valor de $\zeta(x)$ corresponde ao $\zeta(y)$, onde y representa o vértice pertencente à árvore Tt tal que o arco (x, y) também pertence à árvore Tt.

Graficamente, podemos observar que a rotulação ζ nada mais é que atribuir a cada vértice não pertencente ao caminho base P_r o valor de ζ do seu último vértice comum ao caminho base P_r. Na figura 4.2(c), temos o caminho base $P_r = \langle e, d, a \rangle$ cuja rotulação é a mesma de ϵ, ou seja, $\zeta(e) = 3$, $\zeta(d) = 2$ e $\zeta(a) = 1$.

$\zeta(b) = 3$, uma vez que o ζ do último vértice pertencente ao caminho P_r comum aos caminhos P_r e $\langle e, b \rangle$, na árvore Tt, é 3.

$\zeta(c) = 3$, uma vez que o ζ do último vértice pertencente ao caminho P_r comum aos caminhos P_r e $\langle e, b, c \rangle$, na árvore Tt, é 3.

Sendo os custos das aresta do grafo $G = (V, A)$ todos maiores que zero, temos que: $\forall x \in V, \epsilon(x) \leq \zeta(x)$.

Tipos de caminhos

Dadas duas árvores, Ts e Tt e um α o algoritmo de KIM pode gerar caminhos de dois tipos distintos:

tipo I **tipo I** : $s \longrightarrow u \longrightarrow t$ e $\epsilon(u) < \alpha$. O caminho de s a u está em Ts enquanto o caminho de u a t está em Tt.

tipo II **tipo II** : $s \longrightarrow u \rightarrow v \longrightarrow t$ e $\epsilon(u) < \alpha$. O caminho de s a u está em Ts enquanto o caminho de v a t está em Tt. O arco $(u, v) \in G = (V, A)$ não pertence à árvore Ts e nem a árvore Tt.

Utilizando a figura 4.2 e $\alpha = 2$ teríamos, por exemplo, o seguinte caminho to tipo I: $\langle a, b, e \rangle$. Neste exemplo, usamos $s = a, t = e, u = b, \epsilon(b) = 1 < \alpha$ e obtemos então o caminho de s a b em Ts e o concatenamentos ao caminho de b a t em Tt, ou seja, concatenamentos os caminhos $\langle a, b \rangle$ e $\langle b, e \rangle$.

Utilizando a mesma figura, o mesmo α e $s = a, t = e, u = c, \epsilon(c) < \alpha$ obtemos o caminho do tipo II: $\langle a, c, d, e \rangle$. Neste caso, estamos concatenando o caminho $\langle a, c \rangle \in Ts$ ao arco $(c, d) \notin Ts \cup Tt$ e ao caminho $\langle d, e \rangle \in Tt$.

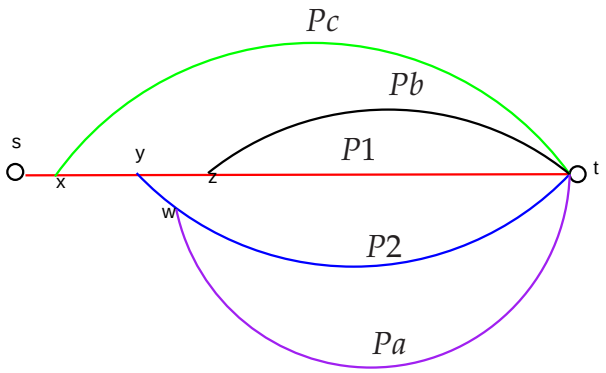
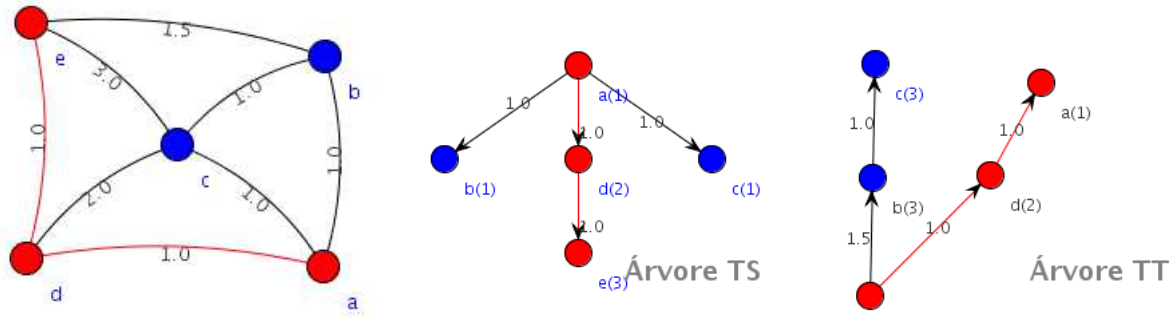


Figura 4.1: A partir do menor caminho, $P1$, calculamos o segundo menor caminho, $P2$. $P1$ e $P2$ formam a base para a obtenção do próximo caminho, $P3$. O caminho Pa corresponde ao menor caminho que se desvia do caminho $P2$ em algum momento depois que o caminho $P2$ se desviou do $P1$. O caminho Pb corresponde ao menor caminho que se desvia do caminho $P1$ depois do vértice comum a $P1$ e $P2$. O caminho Pc corresponde ao menor caminho que se desvia de P1 antes do vértice comum a $P1$ e $P2$. Tomando-se estes três caminhos candidatos para $P3$, basta analisar o de menor custo o qual será o $P3$.

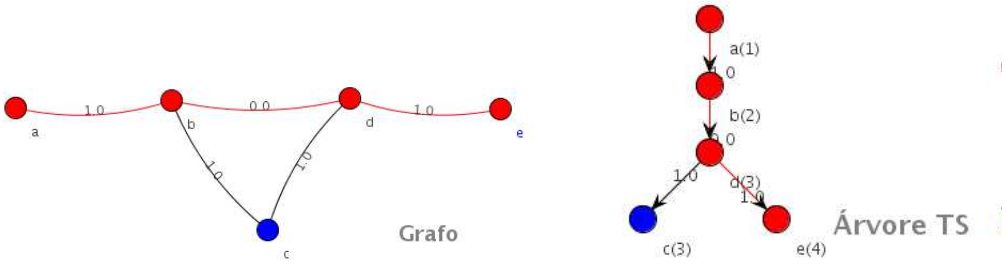


(a) Grafo (b) Árvore Ts (c) Árvore Tt

Figura 4.2: Em (a) temos o grafo de exemplo a partir do qual geraremos árvores com raiz em s=a e t=e. Em vermelho está marcado o menor caminho de a a e. Em (b) temos a árvore de menores caminhos com raiz em a. Em (c) temos a árvore de menores caminhos com raiz em e. Em ambas as árvores a linha vermelha corresponde ao menos caminho de a a e e este é o mesmo que o assinalado no grafo 4.2(a).

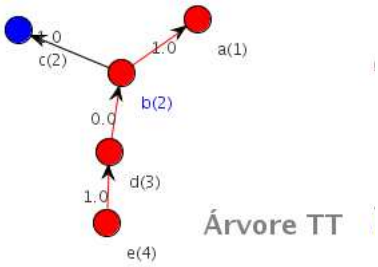
4.3 Arestas com custo zero

Apresentaremos, sucintamente, o problema que pode haver na execução do algoritmo caso exista alguma aresta com custo zero. O funcionamento do algoritmo está baseado na rotulação dos ϵ e ζ respeitar a relação $\epsilon(v) \leq \zeta(v)$ citada na seção 4.2. Quando existem custos zerados nas arestas, é possível que esta relação não seja respeitada. Na figura 4.3 exibimos um grafo com custo zero em uma de suas arestas e na figura as correspondentes árvores *Ts* e *Tt* onde $\epsilon(c) > \zeta(c)$, violando assim a relação básica do algoritmo mencionada na seção 4.2.



(a) Grafo

(b) Árvore Ts

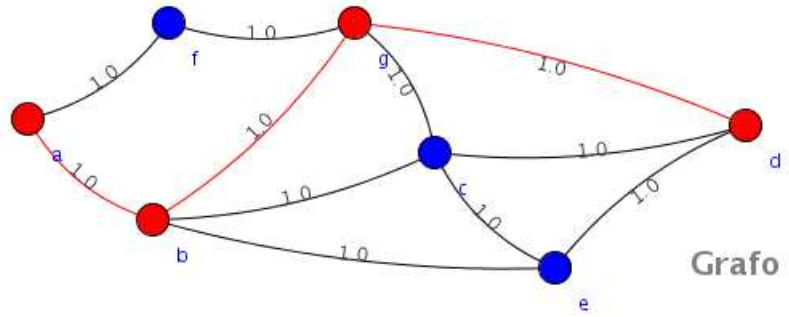


(c) Árvore Tt

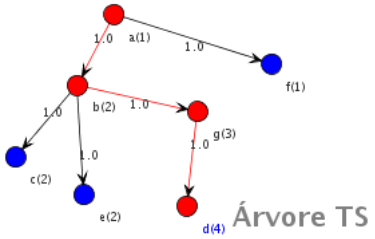
Figura 4.3: Em (a) temos o grafo de exemplo a partir do qual geraremos árvores com raiz em *s*=*a* e *t*=*e*. Em vermelho está marcado o menor caminho de *a* a *e*. Em (b) temos a árvore de menores caminhos com raiz em *a*. Em (c) temos a árvore de menores caminhos com raiz em *e*. Em ambas as árvores a linha vermelha corresponde ao menor caminho de *a* a *e* e este é o mesmo que o assinalado no grafo 4.3(a). Observe que na árvore *Ts* $\epsilon(c) = 3$ e na árvore *Tt* $\zeta(c) = 2$, ou seja, $\epsilon(c) > \zeta(c)$, o que viola a regra básica do algoritmo.

4.4 Simulação

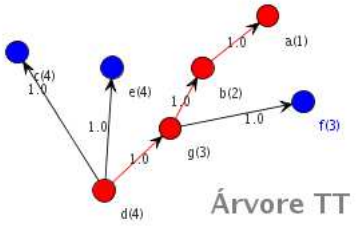
Iremos simular a execução do algoritmo de KIM num grafo simples, exibindo passo a passo de modo a transmitir as operações básicas de seu funcionamento. No grafo da figura 4.4(a) geraremos os três menores caminho com origem em a e destino d. Inicialmente o algoritmo de KIM gera o caminho $P1 = \langle a, b, g, d \rangle$ utilizando o algortimo Dijkstra no grafo da figura 4.4. Em seguida, para gerar o caminho $P2$, realiza uma chamada à rotina FSP 4.2. A rotina FSP calcula as árvores Ts e Tt, apresentadas na figura 4.4(b) e (c).



(a) Grafo



(b) Árvore Ts



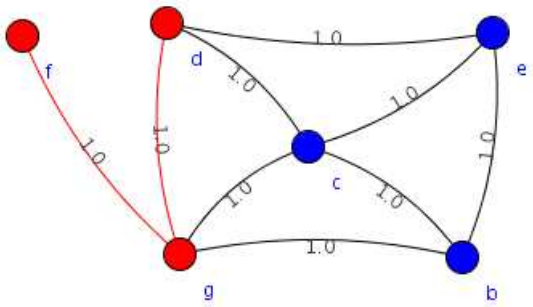
(c) Árvore Tt

Figura 4.4: Em (a) temos o grafo base usado na simulação. Em vermelho temos o caminho $P1$ gerado a partir de uma chamada ao algoritmo Dijkstra com origem a e destino d. Em (b) temos a árvore de menores caminhos com raiz em a. Em (c) temos a árvore de menores caminhos com raiz em e. Em ambas as árvores a linha vermelha corresponde ao menor caminho de a a e e este é o mesmo que o assinalado no grafo em (a).

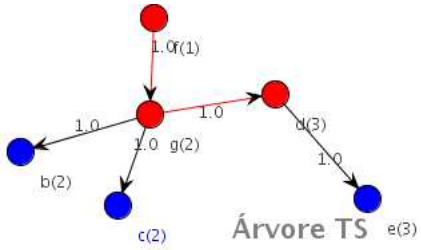
A partir das árvores Ts e Tt apresentadas na figura 4.4(b) e (c), chamamos a rotina SEP 4.3 a qual, utilizando as rotulações ϵ e ζ, testa todos os caminhos dos tipos I e II, retornando o menor dentre eles. Este caminho será o retornado pela função FSP, no

caso, o caminho $P2 = \langle a, f, g, d \rangle$.

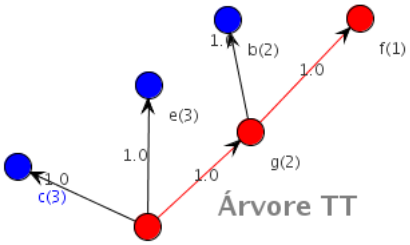
Em seguida, usando como base os caminhos $P1$ e $P2$ geraremos os caminhos candidatos: Pa, Pb, Pc. Na figura 4.5 temos as árvores que darão origem ao caminho Pa, o qual de desvia do caminho $P2$. Utilizando-as, a rotina SEP[4.3] acaba por retornar o caminho $Pa = \langle a, f, g, c, d \rangle$.



(a) Grafo



(b) Árvore Ts

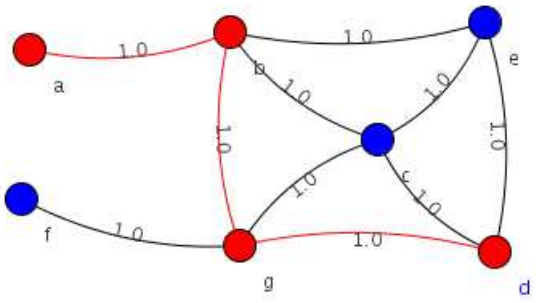


(c) Árvore Tt

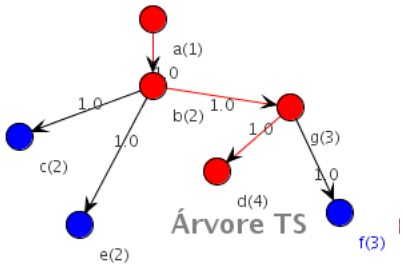
Figura 4.5: Esquema de geração do caminho Pa derivado do caminho $P2 = \langle a, f, g, d \rangle$ Em (a) temos o grafo da figura 4.4(a) com o vértice a removido, bem como suas arestas. Em (b) temos a árvore de menores caminhos com raiz em f. Em (c) temos a árvore de menores caminhos com raiz em d. Em ambas as árvores a linha vermelha corresponde ao menor caminho de f a d e este é o mesmo que o assinalado no grafo em (a).

Para a geração do caminho Pb podemos observar a figura 4.6. Nela, o arco (a, f) foi removido, por fazer parte do prefixo comum aos caminho $P1$ e $P2$. A rotina SEP retornou o caminho $Pb = \langle a, b, e, d \rangle$, como sendo o menor dentre todos os caminhos dos tipos I e II gerados a partir das árvores Ts e Tt da figura 4.6(b) e (c), respectivamente.

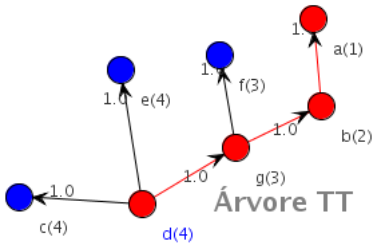
Uma vez que não existe caminho que se desvie antes do último vértice comum aos caminho $P1$ e $P2$, não é possível gerar um caminho Pc na primeira iteração do algoritmo. Neste ponto, retiramos da lista de caminhos candidatos o de menor custo



(a) Grafo



(b) Árvore Ts



(c) Árvore Tt

Figura 4.6: Esquema de geração do caminho Pb derivado do caminho $P1 = \langle a, b, g, d \rangle$. Em (a) temos o grafo da figura 4.4(a) com o arco (a, f) removido. Em (b) temos a árvore de menores caminhos com raiz em a. Em (c) temos a árvore de menores caminhos com raiz em d. Em ambas as árvores a linha vermelha corresponde ao menor caminho de a a d e este é o mesmo que o assinalado no grafo em (a).

entre o caminhos *Pa* e *Pb*, previamente calculados, ou seja, retiramos o caminho *Pb*.

4.5 Implementação

KIM

A implementação foi feita em JAVA utilizando-se a biblioteca JUNG, mencionada anteriormente. Nossa principal missão foi aproveitar ao máximo o código já existente no JUNG.

```
public Path<KIMVertex, KIMEdge>[] getPaths(KIMVertex source,
        KIMVertex target, int nOfPaths) {
    int k = 0, gama, delta;
    TreeSet<Integer>[] W = new TreeSet[nOfPaths];
    DeviationArcsMap<KIMEdge>[] B = new DeviationArcsMap[nOfPaths];
    SortedSet<Path<KIMVertex, KIMEdge>> P = new TreeSet<Path<KIMVertex, KIMEdge>>();
    Path<KIMVertex, KIMEdge>[] bestPaths = new Path[nOfPaths];
    DijkstraShortestPath<KIMVertex, KIMEdge> dsp = new DijkstraShortestPath<KIMVertex, KIMEdge>(
            graph, nev);
    Path<KIMVertex, KIMEdge> p1 = new Path<KIMVertex, KIMEdge>(dsp.getPath(
            source, target), source);
    bestPaths[k] = p1;
    if (p1 == null) return bestPaths;
    p1.setParent(null);
    p1.setOrdem(0);
    W[0] = new TreeSet<Integer>();
    W[0].add(0);
    W[0].add(p1.getVertices().size() - 1);
    if (nOfPaths == 1) return bestPaths;
    insertPath(p1, 0);
    Path<KIMVertex, KIMEdge> p2 = FSP(source, target, p1, p1.getVertices().size());
    removePath(p1, 0);
    if (p2 == null) return bestPaths;
    p2.setParent(p1);
    P.add(p2);
    B[0] = new DeviationArcsMap<KIMEdge>(p1.getVertices().size());
    while (!P.isEmpty()) {
        k++;
        Path<KIMVertex, KIMEdge> pk = P.first();
        pk.setOrdem(k);
        bestPaths[k] = pk;
        if (k == nOfPaths - 1) {
            break;
        }
        P.remove(pk);
        Path<KIMVertex, KIMEdge> pj = pk.getParent();
        W[k] = new TreeSet<Integer>();
        W[k].add(pk.getDevNodeIndex() + 1);
        W[k].add(pk.getVertices().size() - 1);
        B[k] = new DeviationArcsMap<KIMEdge>(pk.getVertices().size());
        B[pj.getOrdem()].put(pk.getDevNodeIndex(), pk.getEdge(pk
                .getDevNodeIndex()));
        if (pk.getDevNodeIndex() + 1 != pk.getVertices().size() - 1) {
            Path<KIMVertex, KIMEdge> pa = getPa(source, target, pk);
            if (pa != null) {
                pa.setParent(pk);
                pa.setOrigem("A");
                P.add(pa);
            }
        }
        gama = getGama(W[pj.getOrdem()], pk.getDevNodeIndex(), pj
                .getVertices().size() - 1);
        Path<KIMVertex, KIMEdge> pb = null;
        if (gama >= 0)
            pb = getPb(B[pj.getOrdem()], source, target, pk, pj, gama);
        if (pb != null) {
            pb.setParent(pk.getParent());
            pb.setOrigem("B");
            logger.debug("Pb=" + pb);
```

```
                        P.add(pb);
                    }
                    if (!W[pj.getOrdem()].contains(pk.getDevNodeIndex())) {
                        W[pj.getOrdem()].add(pk.getDevNodeIndex());
                        delta = getDelta(W[pj.getOrdem()], pj.getDevNodeIndex(), pk
                                .getDevNodeIndex());
                        Path<KIMVertex, KIMEdge> pc = null;
                        if (delta >= 0)
                            pc = getPc(B[pj.getOrdem()], source, target, pk, pj, delta);
                        if (pc != null) {
                            pc.setParent(pk.getParent());
                            pc.setOrigem("C");
                            P.add(pc);
                        }
                    }
                }
                return bestPaths;
        }
```

Código 4.1: Rotina getPaths responsável por calcular e retornar os *k*-menores caminhos

FSP

```
protected Path<KIMVertex, KIMEdge> FSP(KIMVertex s, KIMVertex t,
                Path<KIMVertex, KIMEdge> parent) {
        ShortestPathKIM<KIMVertex, KIMEdge> Ts = getShortestPathAlgorithm(TreeType.TS);
        ShortestPathKIM<KIMVertex, KIMEdge> Tt = getShortestPathAlgorithm(TreeType.TT);
        stopWatchTrees.start("Trees");
        Ts.getIncomingEdgeMap(s);
        Tt.getIncomingEdgeMap(t);
        alfa = parent.getVertices().size();
        Pair resp = SEP(Ts, Tt, s, t, alfa);
        if (resp == null)
                return null;
        if (resp.getSecond().equals(resp.getFirst()))
                logger.debug("Caminho Tipo I");
        else
                logger.debug("Caminho Tipo II");
        Path<KIMVertex, KIMEdge> path = new Path<KIMVertex, KIMEdge>(Ts, Tt,
                        resp, s, t);
        return path;
}
```

Código 4.2: Rotina FSP, dado um caminho base retorna o menor caminho diferente deste.

SEP

```
protected Pair SEP(ShortestPathKIM<KIMVertex, KIMEdge> Ts,
                ShortestPathKIM<KIMVertex, KIMEdge> Tt, KIMVertex s, KIMVertex t,
                int alfa) {
        Stack<KIMVertex> stack = new Stack<KIMVertex>();
        double distance = new Double(Double.MAX_VALUE);
        Pair resp = null;
        stack.push(s);
        while (!stack.isEmpty()) {
                KIMVertex u = stack.pop();
                Set<KIMVertex> sons = getSons(Ts, u, s);
                if (u.getEpsilon().equals(u.getZeta())) {
                        Iterator<KIMEdge> i = graph.getOutEdges(u).iterator();
                        while (i.hasNext()) {
                                KIMEdge atual = i.next();
                                KIMVertex v = graph.getOpposite(u, atual);
                                if (!sons.contains(v)) {
                                        if (u.getEpsilon() < v.getZeta()) {
                                                double newDistance = Ts.getDistance(s, u)
                                                                .doubleValue()
                                                                + nev.transform(atual).doubleValue()
                                                                + Tt.getDistance(t, v).doubleValue();
                                                if (newDistance < distance) {
                                                        distance = newDistance;
                                                        resp = new Pair(u, atual);
                                                }
                                        }
                                }
                        }
                } else {
                        int newDistance = Ts.getDistance(s, u).intValue()
                                        + Tt.getDistance(t, u).intValue();
                        if (newDistance < distance) {
                                distance = newDistance;
                                resp = new Pair(u, u);
                        }
                }
                Iterator<KIMVertex> j = sons.iterator();
                while (j.hasNext()) {
                        KIMVertex son = j.next();
                        if (son.getEpsilon() < alfa) {
                                stack.push(son);
                        }
                }
        }
        return resp;
}
```

Código 4.3: Considerando os rótulos ϵ e ζ, testa caminhos do tipo I e II, retornando o menor dentre eles.

```
public Set<KIMVertex> getSons(ShortestPathKIM<KIMVertex, KIMEdge> T,
                KIMVertex u, KIMVertex s) {
        Iterator<KIMEdge> i = graph.getOutEdges(u).iterator();
        Set<KIMVertex> sons = new HashSet<KIMVertex>();
        while (i.hasNext()) {
                KIMEdge atual = i.next();
                KIMVertex o = graph.getOpposite(u, atual);
                KIMEdge incident = T.getIncomingEdge(s, o);
                if (incident != null && graph.getOpposite(o, incident).equals(u))
                        sons.add(o);
        }
        return sons;
}
```

Código 4.4: Retorna todos os vértices filhos do vértice u na árvore T enraizada pelo vértice s

getPa

```
protected Path<KIMVertex, KIMEdge> getPa(KIMVertex source,
                KIMVertex target, Path<KIMVertex, KIMEdge> pk) {
        KIMVertex start = pk.getVertices().get(pk.getDevNodeIndex() + 1);
        Path<KIMVertex, KIMEdge> p = pk.getSubPath(start);
        LinkedHashMap<KIMEdge, Pair<KIMVertex>> verticesToRestore = removeVertices(pk
                        .getVertices().subList(0, pk.getDevNodeIndex() + 1));
        insertPath(pk, pk.getDevNodeIndex() + 1);
        Path<KIMVertex, KIMEdge> pa = FSP(start, target, p, pk.getVertices()
                        .size() - 1);
        removePath(pk, pk.getDevNodeIndex() + 1);
        Path<KIMVertex, KIMEdge> resp = null;
        restoreVertices(pk.getVertices().subList(0, pk.getDevNodeIndex() + 1),
                        verticesToRestore);
        if (pa != null)
                resp = new Path<KIMVertex, KIMEdge>(pk.getPrefix(start), pa);
        return resp;
}
```

Código 4.5: Calcula o menor caminho *Pa*, vide figura 4.1.

getPb

```
private Path<KIMVertex, KIMEdge> getPb(
                DeviationArcsMap<KIMEdge> devArcsMap, KIMVertex source,
                KIMVertex target, Path<KIMVertex, KIMEdge> pk,
                Path<KIMVertex, KIMEdge> pj, int gama) {
        Path<KIMVertex, KIMEdge> R = pj.getSubPath(pk.getDevNode(), pk
                        .getDevNodeIndex(), gama);
        LinkedHashMap<KIMEdge, Pair<KIMVertex>> arcsToRestore = removeEdges(devArcsMap
                        .get(pk.getDevNodeIndex()));
        LinkedHashMap<KIMEdge, Pair<KIMVertex>> verticesToRestore = removeVertices(pj
                        .getVertices().subList(0, pk.getDevNodeIndex()));
        insertPath(pj, pk.getDevNodeIndex());
        Path<KIMVertex, KIMEdge> pb = FSP(pk.getVertex(pk.getDevNodeIndex()),
                        target, R, pj.getVertices().size() - 1);
        removePath(pj, pk.getDevNodeIndex());
        Path<KIMVertex, KIMEdge> resp = null;
        restoreVertices(pj.getVertices().subList(0, pk.getDevNodeIndex()),
                        verticesToRestore);
        restoreEdges(arcsToRestore);
        if (pb != null)
                resp = new Path<KIMVertex, KIMEdge>(pj.getSubPath(pj.getStart(), 0,
                                pk.getDevNodeIndex()), pb);
        return resp;
}
```

Código 4.6: Calcula o menor caminho *Pb*, vide figura 4.1.

getPc

```
private Path<KIMVertex, KIMEdge> getPc(
        DeviationArcsMap<KIMEdge> devArcsMap, KIMVertex source,
        KIMVertex target, Path<KIMVertex, KIMEdge> pk,
        Path<KIMVertex, KIMEdge> pj, int delta) {
    Path<KIMVertex, KIMEdge> R = pj.getSubPath(pj.getVertex(delta), delta,
            pk.getDevNodeIndex());
    LinkedHashMap<KIMEdge, Pair<KIMVertex>> arcsToRestore = removeEdges(devArcsMap
            .get(delta));
    LinkedHashMap<KIMEdge, Pair<KIMVertex>> verticesToRestore = removeVertices(pj
            .getVertices().subList(0, delta));
    insertPath(pj, delta);
    Path<KIMVertex, KIMEdge> pc = FSP(pj.getVertex(delta), target, R, pj
            .getVertices().size() - 1);
    removePath(pj, delta);
    Path<KIMVertex, KIMEdge> resp = null;
    restoreVertices(pj.getVertices().subList(0, delta), verticesToRestore);
    restoreEdges(arcsToRestore);
    if (pc != null)
        resp = new Path<KIMVertex, KIMEdge>(pj.getSubPath(pj.getStart(), 0,
                delta), pc);
    return resp;
}
```

Código 4.7: Calcula o menor caminho *Pc*, vide figura 4.1.

CAPÍTULO 5

Resultados Experimentais

5.1 Motivação

Recentemente, há um grande interesse em trabalhos relacionados a análise experimental de algoritmos. Em particular, no caso do algoritmo de Dijkstra, uma subrotina do algoritmo KIM, podemos citar os artigos de B.V. Cherkassky, A.V. Goldberg, T. Radzik e Craig Silverstein [4, 14, 5], e do algoritmo de KIM podemos citar o artigo de Eleni Hadjiconstantinou and Nicos Christofides [15].

O interesse em experimentação é devido ao reconhecimento de que os resultados teóricos, freqüentemente, não trazem informações referentes ao desempenho do algoritmo na prática. Porém, o campo da análise experimental é repleto de armadilhas, como comentado por D.S. Johnson [19]. Muitas vezes, a implementação do algoritmo é a parte mais simples do experimento. A parte difícil é usar, com sucesso, a implementação para produzir resultados de pesquisa significativos.

Segundo D.S. Johnson [19], pode-se dizer que existem quatro motivos básicos que levam a realizar um trabalho de implementação de um algoritmo:

(1) Para usar o código em uma aplicação particular, cujo propósito é descrever o impacto do algoritmo em um certo contexto;

(2) Para proporcionar evidências da superioridade de um algoritmo;

(3) Para melhor compreensão dos pontos fortes, fracos e do desempenho das operações algorítmicas na prática; e

(4) Para produzir conjecturas sobre o comportamento do algoritmo no caso-médio sob distribuições específicas de instâncias onde a análise probabilística direta é muito

difícil.

Nesta dissertação estamos mais interessados no motivo (3).

5.2 Ambiente experimental

A plataforma utilizada nos experimentos foi um notebook rodando Linux Ubuntu 8.04, Kernel 2.6.24-23 com dois processadores Intel T7500 de 2.20Ghz e 2GB de RAM.

Para controlar os tempos usamos a classe StopWatch 5.1, implementada por Rod Johnson e Juergen Hoeller.

```
public class StopWatch {
        private final String id;
        private boolean keepTaskList = true;
        private final List taskList = new LinkedList();
        private long startTimeMillis;
        private boolean running;
        private String currentTaskName;
        private TaskInfo lastTaskInfo;
        private int taskCount;
        private long totalTimeMillis;

        public StopWatch() {
                this.id = "";
        }

        public StopWatch(String id) {
                this.id = id;
        }

        public void start(String taskName) throws IllegalStateException {
                if (this.running) {
                        throw new IllegalStateException(
                                        "Can't start StopWatch: it's already running");
                }
                this.startTimeMillis = System.currentTimeMillis();
                this.running = true;
                this.currentTaskName = taskName;
        }

        public void stop() throws IllegalStateException {
                if (!this.running) {
                        throw new IllegalStateException(
                                        "Can't stop StopWatch: it's not running");
                }
                long lastTime = System.currentTimeMillis() - this.startTimeMillis;
                this.totalTimeMillis += lastTime;
                this.lastTaskInfo = new TaskInfo(this.currentTaskName, lastTime);
                if (this.keepTaskList) {
                        this.taskList.add(lastTaskInfo);
                }
                ++this.taskCount;
                this.running = false;
                this.currentTaskName = null;
        }

        public long getLastTaskTimeMillis() throws IllegalStateException {
                if (this.lastTaskInfo == null) {
                        throw new IllegalStateException(
                                        "No tests run: can't get last interval");
                }
```

```
            return this.lastTaskInfo.getTimeMillis();
        }

        public long getTotalTimeMillis() {
            return totalTimeMillis;
        }
}
```

Código 5.1: Classe para controle dos tempos de execução

```
1   StopWatch stopWatch = new StopWatch("KIM");
2   stopWatch.start();
3   stopWatch.stop();
4   int time = stopWatch.getLastTaskTimeMillis();
```

Código 5.2: Exemplo de uso da classe StopWatch5.1

Para calcular o uso de memória utilizamos o seguinte trecho de código:

```
1   long initMem = Runtime.getRuntime().freeMemory();
2   /* leitura ou geracao do grafo */
3   long graphMemUsage = initMem - Runtime.getRuntime().freeMemory();
4   // execucao do algoritmo
5   long kimMemUsage = initMem - graphMemUsage - Runtime.getRuntime().freeMemory();
```

Os testes foram criados levando-se em conta o consumo de tempo assintótico do algoritmo KIM $O(kc(n,m))$, onde $c(m,n)$ é o consumo de tempo da subrotina que calcula uma àrvore de menores caminhos. No caso de grafos sem custos nas arestas, utilizamos uma busca em largura, cuja consumo de tempo é $O(n+m)$, caso contrário vamos para a implementação do Dijkstra feita no JUNG, cujo consumo é $O(m \lg n)$.

5.3 Gerador de instâncias

Implementamos um pequeno gerador de grafos simétricos aleatórios utilizando a interface *GraphGenerator* fornecida pelo JUNG. Inicialmente pensamos em utilizar geradores disponíveis na DIMACS, mas estes geravam apenas grafos dirigidos, desta maneira teríamos que convertê-los para simétricos. O gerador implementado segue a idéia apresentada no artigo de Eleni Hadjiconstantinou and Nicos Christofides [15]:

(1) Criamos os vértices.

(2) Criamos um ciclo hamiltoniano ligando cada um no seu vizinho.

(3) Adicionamos aleatoriamente o restante dos arcos.

Os parâmetros para criação são:

- n número de vértices
- m número de arestas, sendo que $n \leq m \leq n * (n-1)/2$, pois se $m < n$ não é possível construir o ciclo hamiltoniano e, se $m > n * (n-1)/2$ não é possível criar um grafo sem arestas paralelas.
- *EdgeFactory* fábrica de arcos, caso estejamos usando uma representação específica para os arcos.
- *VertexFactory* fábrica de vértices, caso estejamos usando uma representação específica para os vértices.

```
1  public class ConnectedUndirectedGraphGenerator<V, E> implements
2                  GraphGenerator<V, E> {
3
4          private Factory<V> vertexFactory;
5          private Factory<E> edgeFactory;
6          private int mNumVertices;
7          private int mNumEdges;
8          private Random mRandom;
9
10         @Override
11         public Graph<V, E> create() {
12                 UndirectedGraph<V, E> graph = new UndirectedSparseGraph<V, E>();
13                 V prior = vertexFactory.create();
14                 V first = prior;
15                 graph.addVertex(prior);
16                 for (int i = 1; i < mNumVertices; i++) {
17                         V cur = vertexFactory.create();
18                         graph.addVertex(cur);
19                         graph.addEdge(edgeFactory.create(), prior, cur);
20                         prior = cur;
21                 }
22                 graph.addEdge(edgeFactory.create(), prior, first);
23                 List<V> vertices = new ArrayList<V>(graph.getVertices());
24                 while (graph.getEdgeCount() < mNumEdges) {
25                         V u = vertices.get((int) (mRandom.nextDouble() * mNumVertices));
26                         V v = vertices.get((int) (mRandom.nextDouble() * mNumVertices));
27                         /*Não permitimos a criação de loops e nem de arcos paralelos.*/
28                         if (!v.equals(u) && !graph.isSuccessor(v, u)) {
29                                 graph.addEdge(edgeFactory.create(), u, v);
30                         }
31                 }
32                 return graph;
33         }
34
35         public ConnectedUndirectedGraphGenerator(Factory<V> vertexFactory,
36                         Factory<E> edgeFactory, int numVertices, int numEdges) {
37                 if (numEdges < numVertices)
38                         throw new IllegalArgumentException(
39                                         "Numero de arcos deve ser no mínimo igual ao numero de vertices");
40                 if (numEdges > (numVertices * (numVertices - 1) / 2))
41                         throw new IllegalArgumentException(
42                                         "Não é possível criar "
43                                                         + numEdges
44                                                         + " arcos de modo que não haja arcos paralelos e loops num grafo com "
45                                                         + numVertices + " vértices");
46                 this.vertexFactory = vertexFactory;
47                 this.edgeFactory = edgeFactory;
48                 mNumVertices = numVertices;
49                 mNumEdges = numEdges;
50                 mRandom = new Random();
51         }
52
53         static Factory<KIMVertex> defaultVertexFactory = new Factory<KIMVertex>() {
```

```
54                  int id = 1;
55
56                  public KIMVertex create() {
57                          return new KIMVertex(Integer.toString(id++));
58                  }
59          };
60
61          static Factory<KIMEdge> defaultEdgeFactory = new Factory<KIMEdge>() {
62                  public KIMEdge create() {
63                          return new KIMEdge();
64                  }
65          };
66  }
```

Para cada grafo gerado escolhemos aleatoriamente uma origem e um destino, necessariamente diferentes, e rodamos o algoritmo KIM. Cada execução nos retorna os seguintes tempos:

- Tempo total para obtenção dos k caminhos.
- Tempo total gasto na construção das árvores Ts e Tt.
- Tempo total gasto na execução da rotina FSP.
- Tempo total gasto na execução da rotina SEP.

O tempo gasto na criação do grafo não é considerado. A fim de tentar evitar escolhas ruins das origens e destinos, escolhemos cinco origens e destinos e calculamos a média dos tempos acima. Uma vez que o consumo de tempo assintótico do algoritmo de KIM está definido em função do número de caminhos a ser gerado(k), número de arcos(m) e número de vértices(n), nos testes fixaremos sempre duas deixando a outra variável a fim de estudarmos o comportamento do algoritmo.

Além dos tempos, estamos interessados no consumo de memória. Seguindo a mesma idéia apresentada anteriormente, tirar a média entre cinco execuções para cada grafo, calcularemos apenas a mémoria total utilizada no cálculo dos k caminhos.

Usaremos a idéa de densidade de um grafo que consiste em dividir a quantidade de arestas pelo número máximos de arestas possível, ou seja, $d = m/(n^2 - n)$. Isso nos permite fazer comparações mais concisas que se usássemos valores de m e n independentes.

5.4 Gráficos e análises

Começamos com os gráficos exibindo o tempo de execução do algoritmo em função do número de caminhos. Calculamos um gráfico diferente para cada densidade entre 0.1 e 1(grafo completo). Com isto podemos observar se o algoritmo de sai melhor para grafos densos ou esparsos.

Observando os gráficos anteriores é possível constatar o caráter linear do desempenho do algoritmo em função do valor de k, independentemente da densidade escolhida. Isso vem corroborar a análise assíntoca do algoritmo dita anteriormente. Vamos analisar o papel da densidade no desempenho do algoritmo. Façamos uma pequena tabela com os valores de k, densidade e tempo:

Relação entre k, densidade e tempo		
k	densidade	tempo
300	0.1	1500
300	0.2	2100
300	0.4	4000
300	0.8	8000
600	0.1	3200
600	0.2	4600
600	0.4	9000
600	0.8	17000

Quanto mais denso, maior o núemro de arestas que precisa ser testada e isto acaba se refletindo no maior tempo de execução do algoritmo de maneira linear. Observe que fixando-se os valores de k e duplicando-se os valores das densidades, obtemos tempos que são aproxidamente dobrados. Lembrando que o algoritmo de KIm, tem como subrotina o Dijktra para encontrar caminho mínimo e que este está implementado usando um heap temos que o algoritmo de Dijkstra está implementado com consumo de tempo $O(m \lg n)$. Logo, os gráficos mostram a dependência linear entre a densidade e os tempos de execução, mais uma vez corroborando a análise assintótica feita.

O algoritmo de KIM conta, basicamente, com duas subrotinas principais, as quais são responsáveis pela maior parte do tempo consumido pelo algortimo, são elas:

- Rotinas de construção de árvores de menores caminhos. Estas rotinas correspondem a duas execuções do algoritmo de Dijkstra: uma utilizando s como raiz e t como destino, retornando como resposta a árvore Ts e outra onde a raiz é t e o

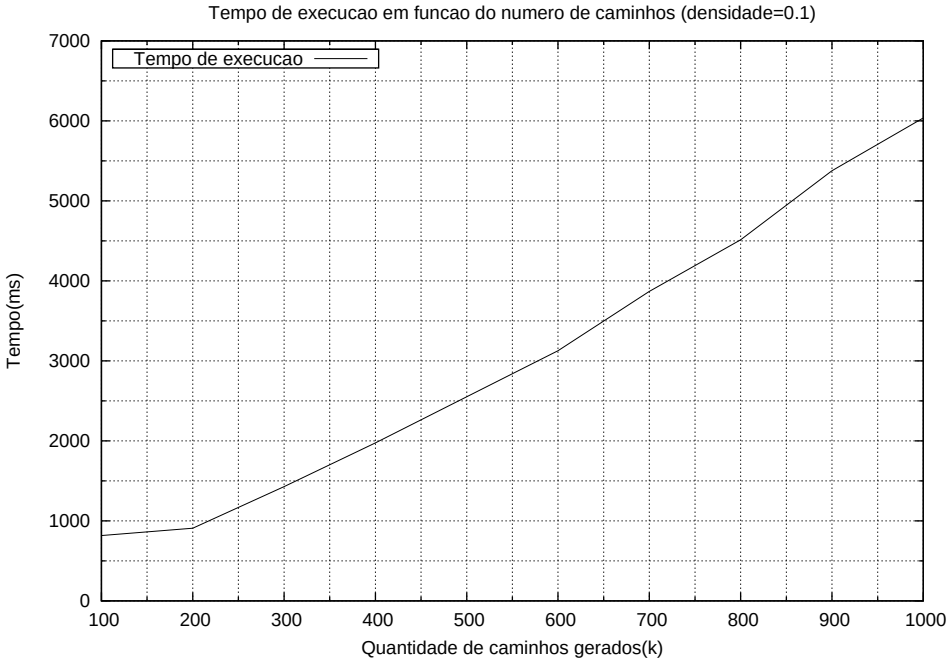


Figura 5.1: Desempenho do algoritmo com um grafo de densidade 0.1 e 100 vértices

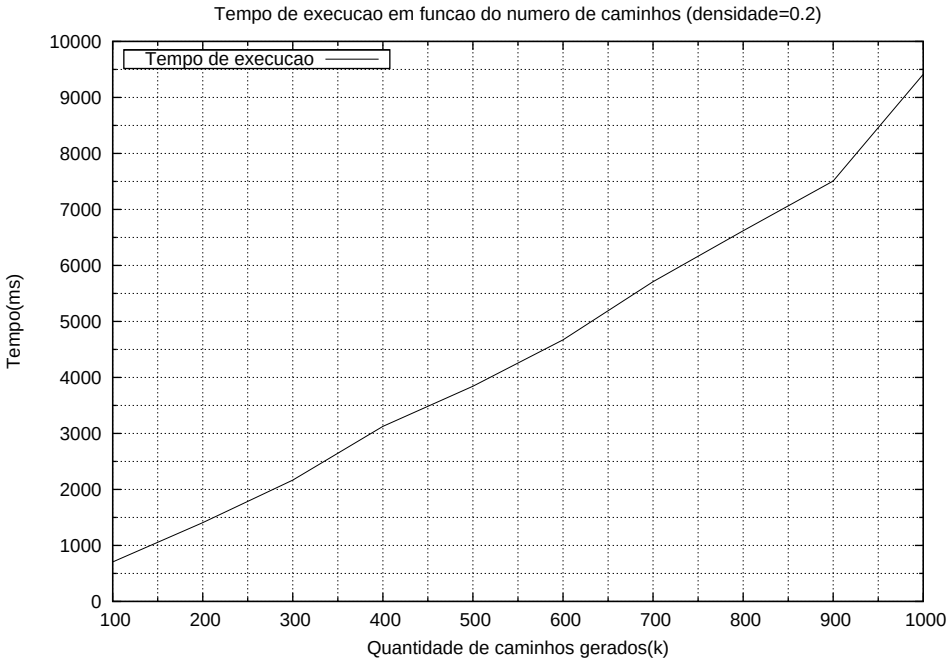


Figura 5.2: Desempenho do algoritmo com um grafo de densidade 0.2 e 100 vértices

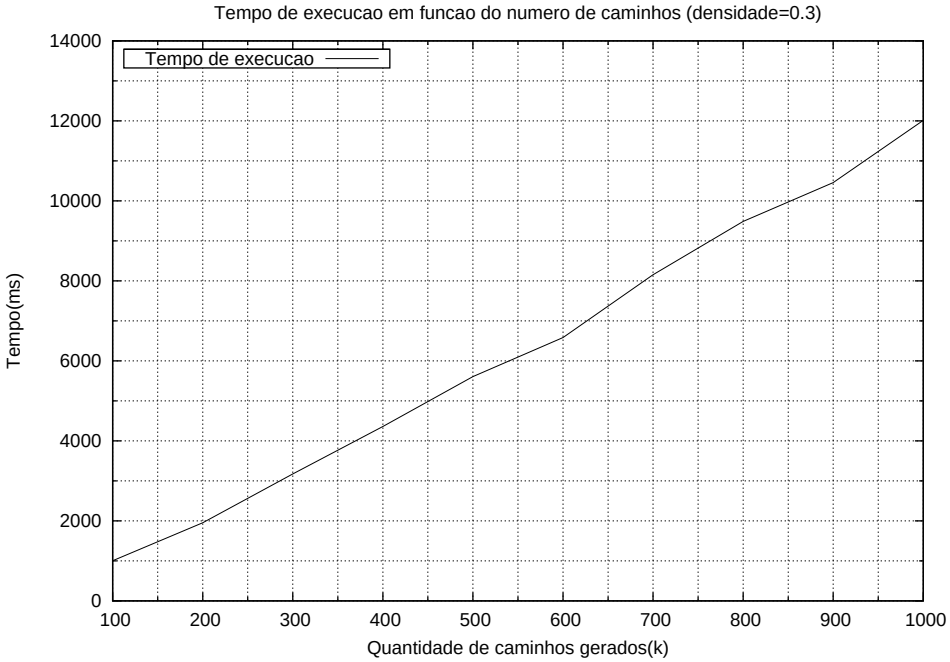


Figura 5.3: Desempenho do algoritmo com um grafo de densidade 0.3 e 100 vértices

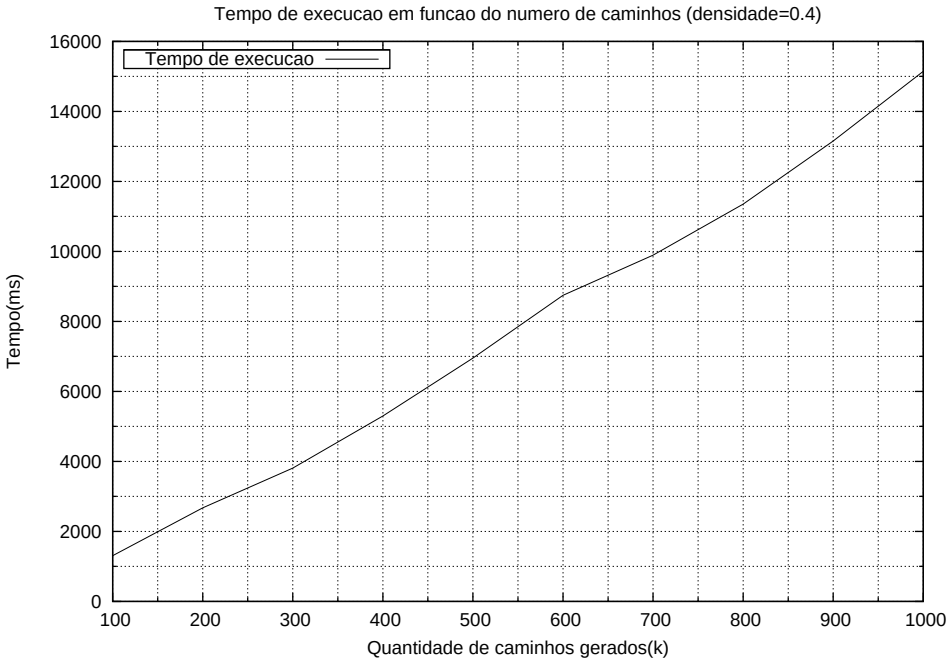


Figura 5.4: Desempenho do algoritmo com um grafo de densidade 0.4 e 100 vértices

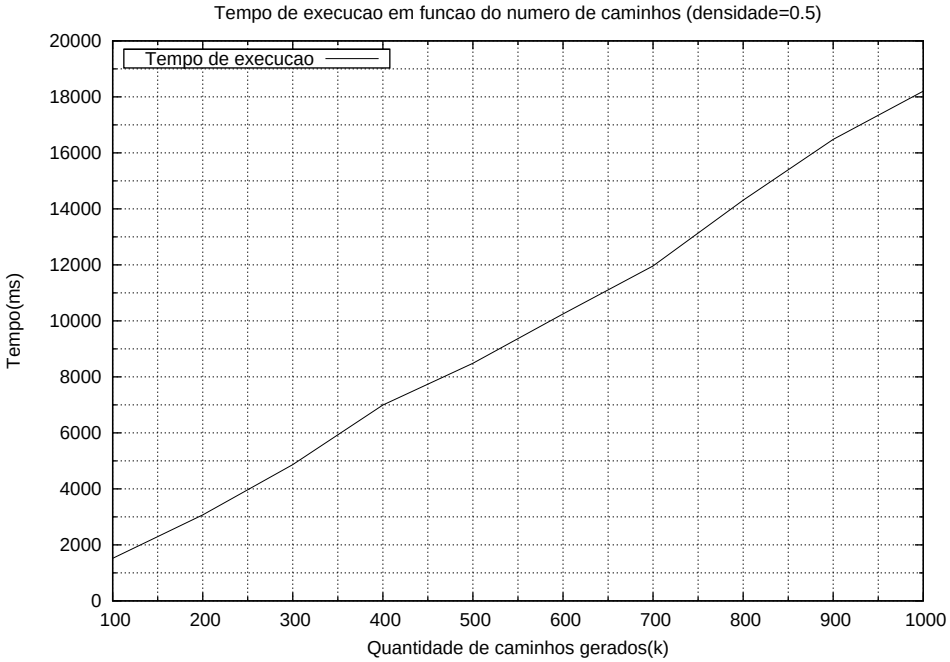


Figura 5.5: Desempenho do algoritmo com um grafo de densidade 0.5 e 100 vértices

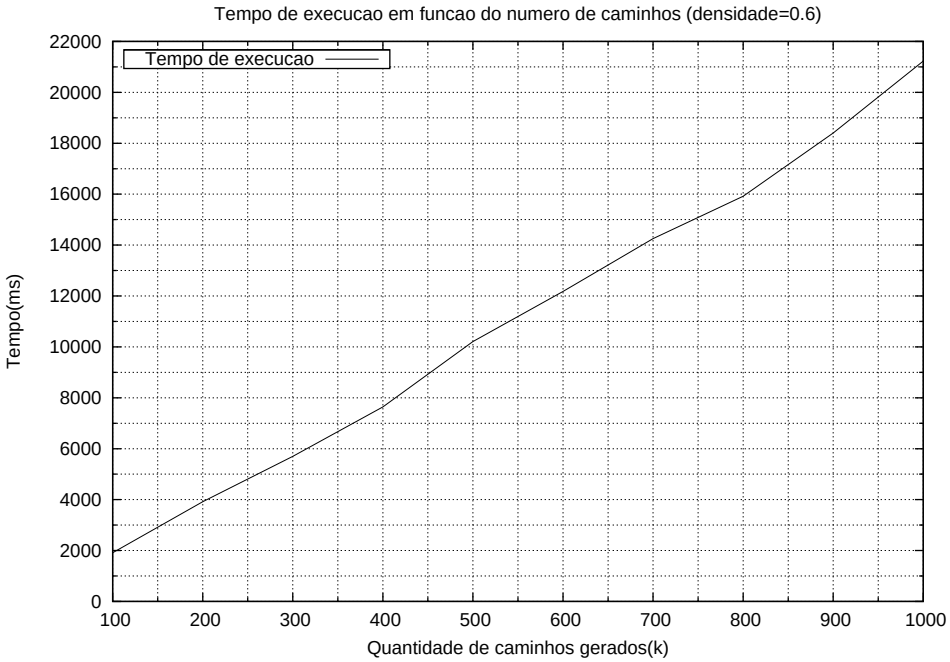


Figura 5.6: Desempenho do algoritmo com um grafo de densidade 0.6 e 100 vértices

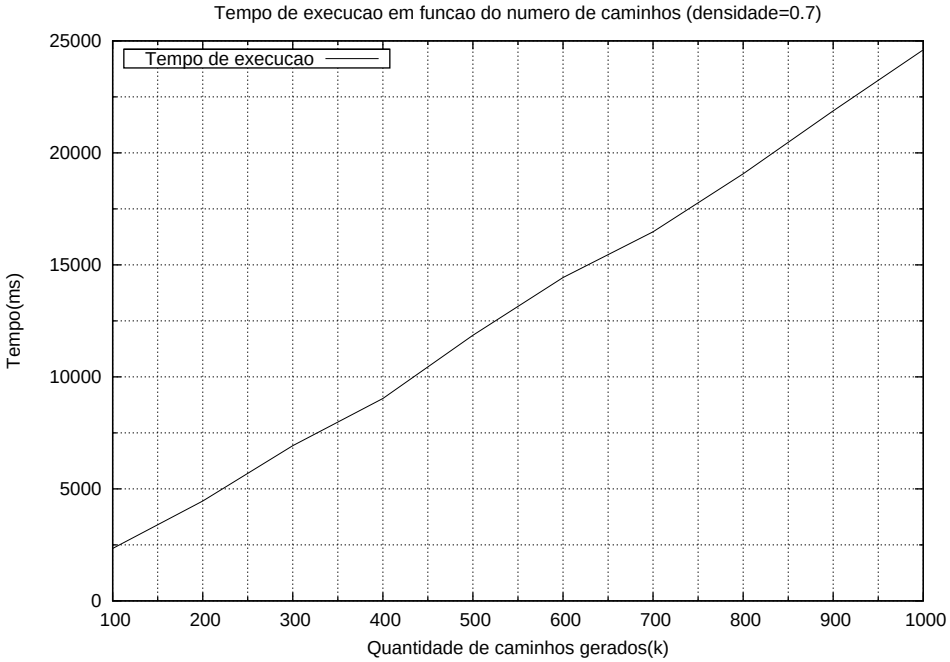


Figura 5.7: Desempenho do algoritmo com um grafo de densidade 0.7 e 100 vértices

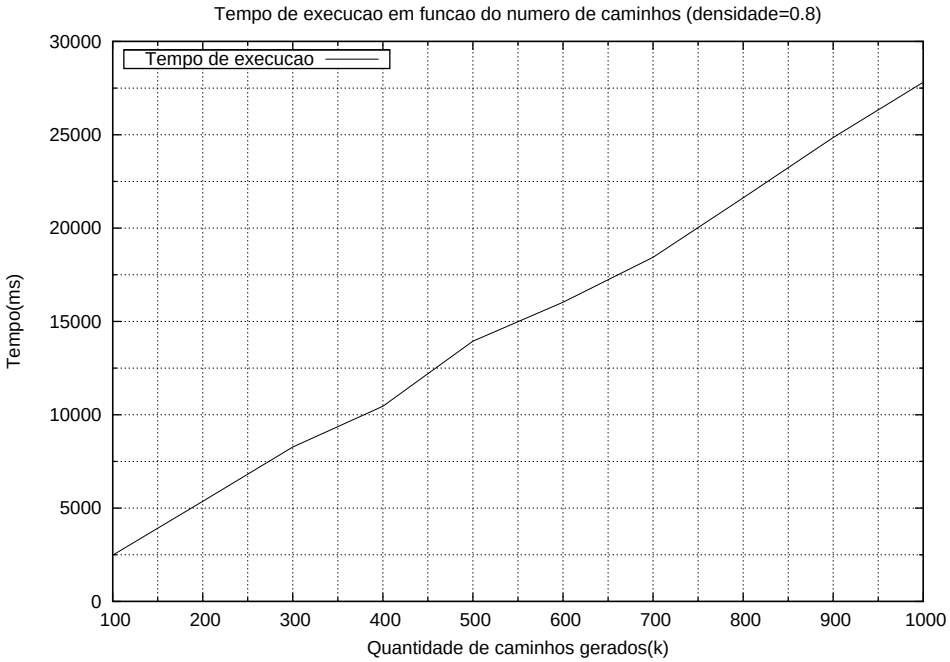


Figura 5.8: Desempenho do algoritmo com um grafo de densidade 0.8 e 100 vértices

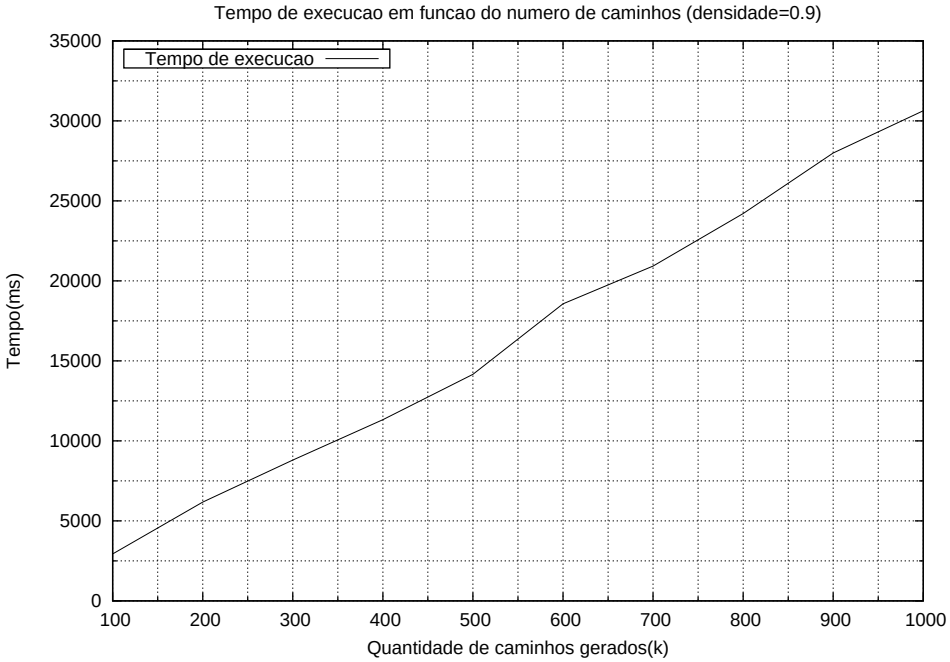


Figura 5.9: Desempenho do algoritmo com um grafo de densidade 0.9 e 100 vértices

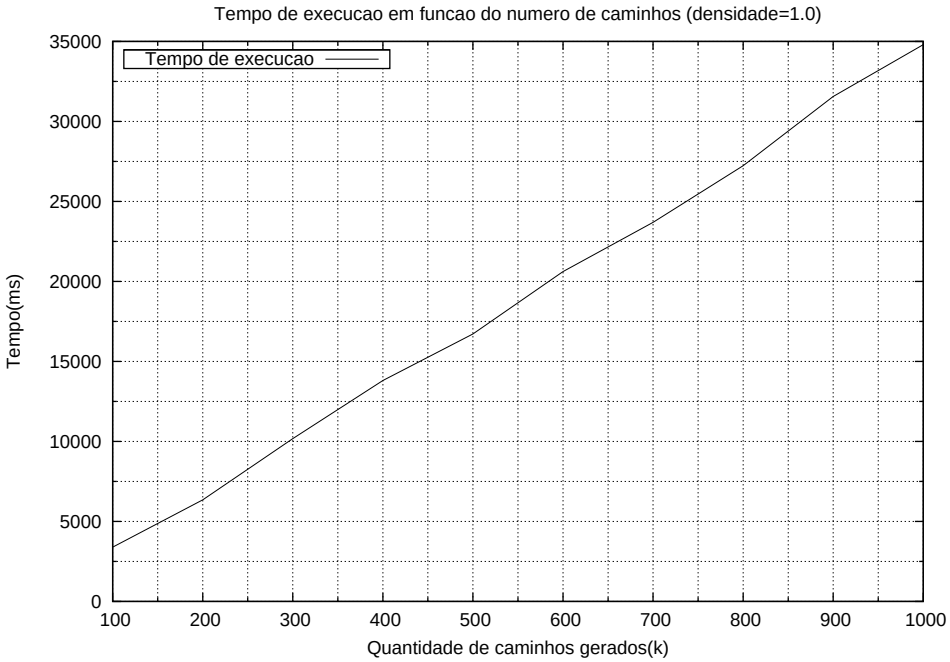


Figura 5.10: Desempenho do algoritmo com um grafo de densidade 1.0 e 100 vértices

destino é s retornando a árvore Tt.

- SEP 4.3 é a rotina responsável por calcular o menor caminho de s a t utilizando para tal as árvores Ts e Tt as quais são rotuladas usando-se ϵ e ζ, como explicado no capítulo 4. Esta rotina percorre todos os vértices pertencentes às árvores, tendo portanto desempenho assintotico $O(n)$

A seguir exibimos gráficos com os tempos de execução da rotina SEP (código 4.3), das construções das árvores e o totais do algoritmo KIM. O objetivo é visualizar o qual significativas são estas funções no que diz respeito ao consumo de tempo e, notar que elas realmente são as mais relevantes neste quesito, sendo portanto os primeiros pontos onde qualquer melhoria deveria ser pensanda.

A partir dos gráficos é possível perceber que estas rotinas realmente correspondem a uma importante fatia do tempo total de execução do algoritmo. É interessante notar que quanto maior o valor de k menor a fatia do tempo total utilizada pelas subrotinas, o que nos leva a entender que outras rotinas passam a se tornam mais relevantes. Vamos estudar agora a variação da densidade e o quanto ela influencia nas fatias de tempos das subrotinas de nosso interesse. Para tal, exibiremos gráficos mostrando a porcentagem do tempo total utilizada por cada uma destas subrotinas.

Concluímos que a importância da função SEP 4.3 cresce com a densidade e a da construção das árvores decresce em contrapartida. A explicação se encontra na função getSons 4.4, a qual se torna mais custosa com o aumento da quantidade de arcos no grafo. Outra fato marcante é que independentemente da densidade quanto maior o valor de k menor é a fatia de tempo utilizada por ambas as funções. Isso pode ser explicado pela exclusão de arestas e vértices que ocorre durante a geração de novos caminhos, levando o consumo de tempo das funções que dependem de m e n a diminuir. Sendo assim, de uma maneira amortizada, o custo total das subrotinas de construção de árvores e a SEP 4.3 acabam consumindo cada vez menos com o aumento de k.

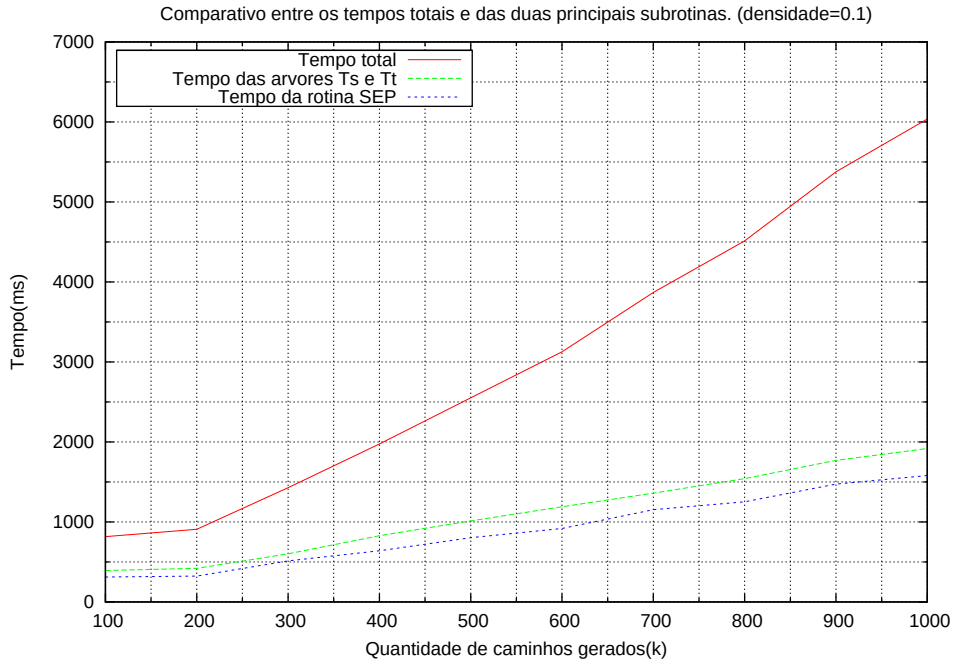


Figura 5.11: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.1

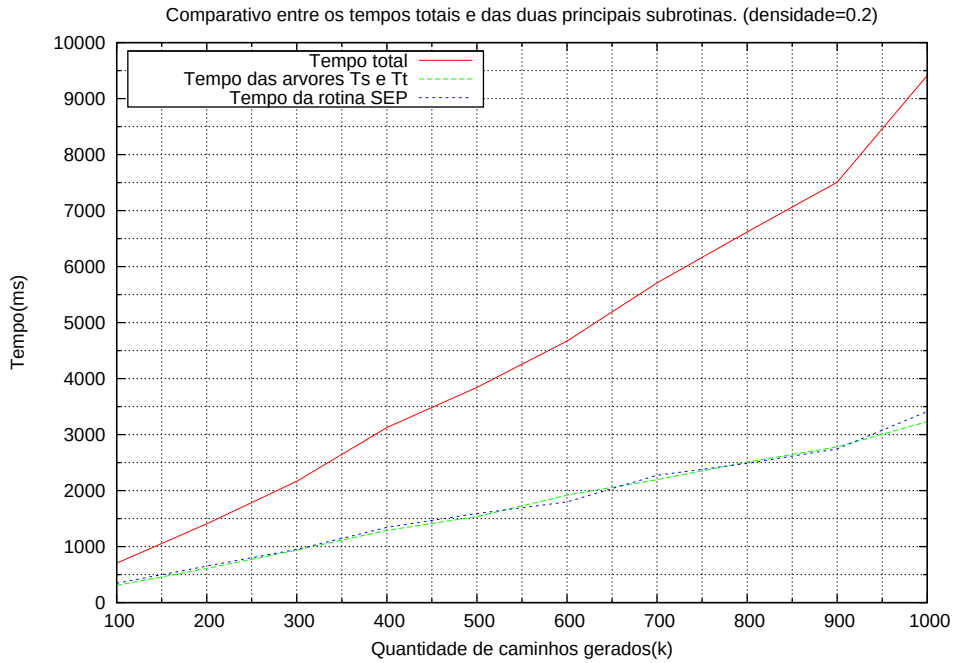


Figura 5.12: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.2

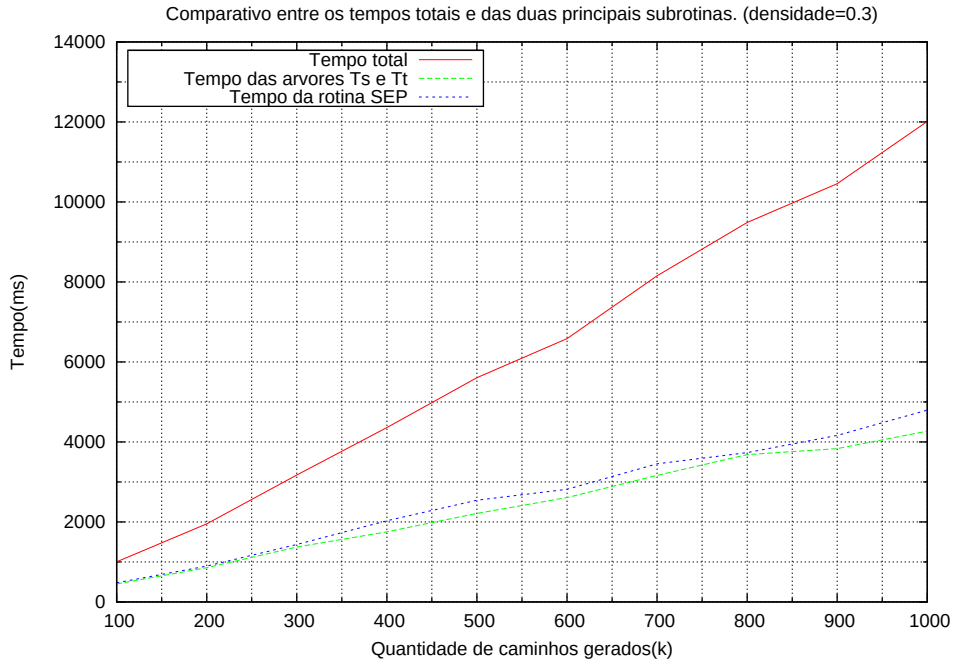


Figura 5.13: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.3

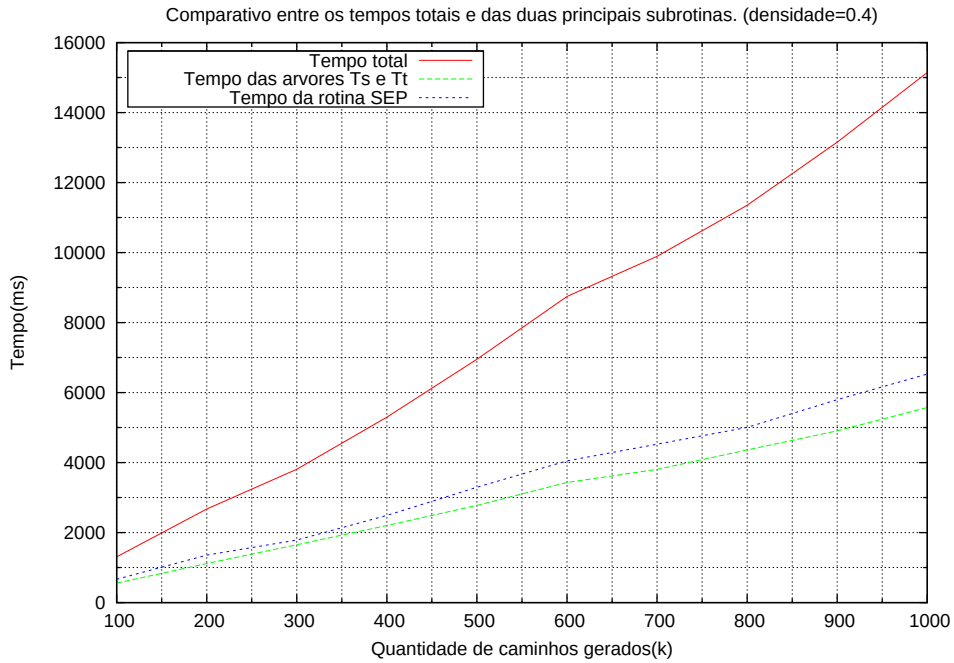


Figura 5.14: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.4

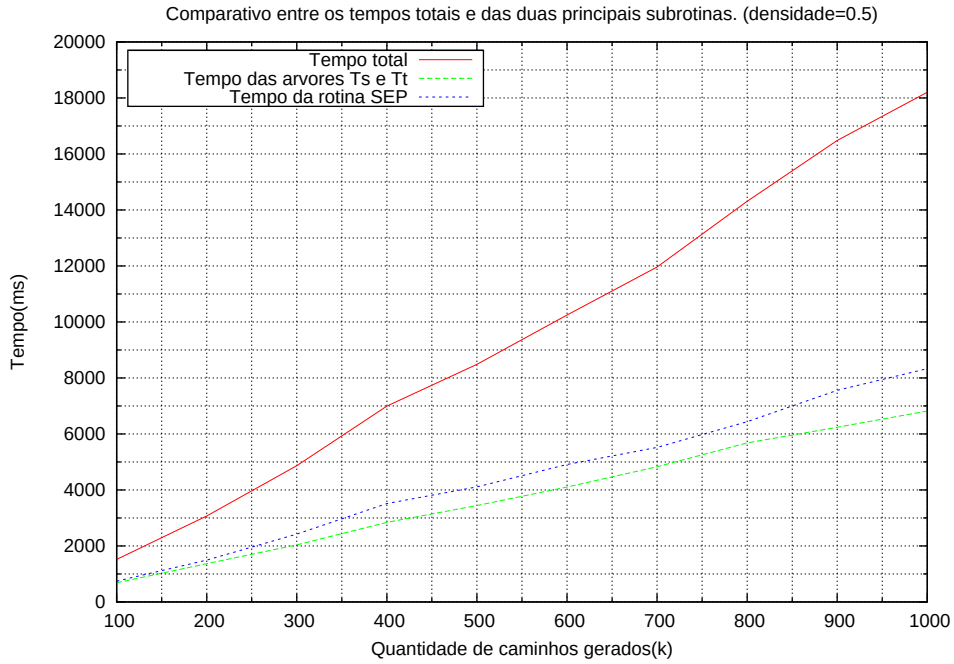


Figura 5.15: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.5

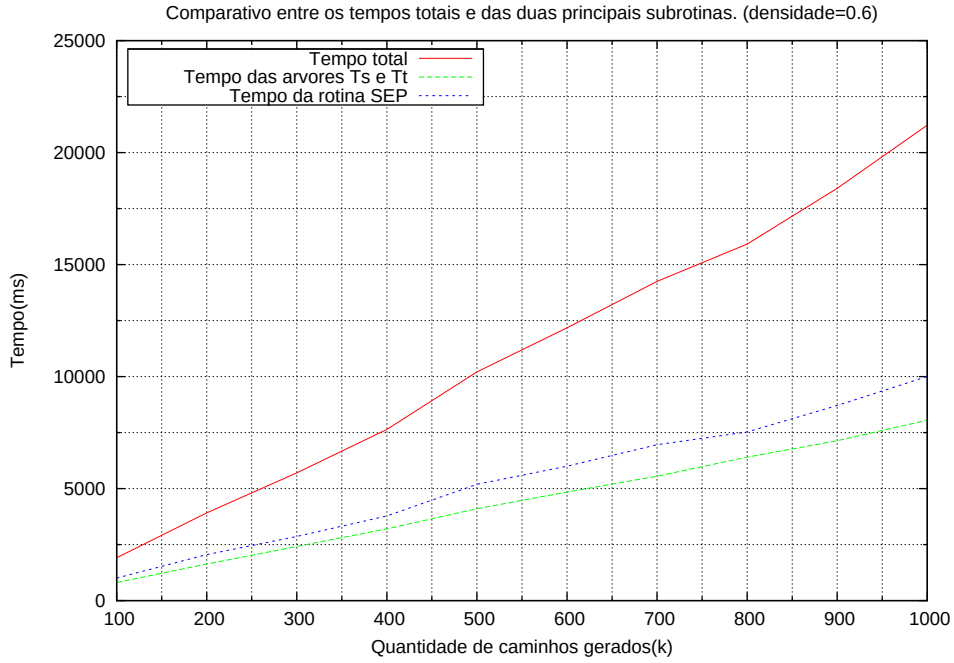


Figura 5.16: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.6

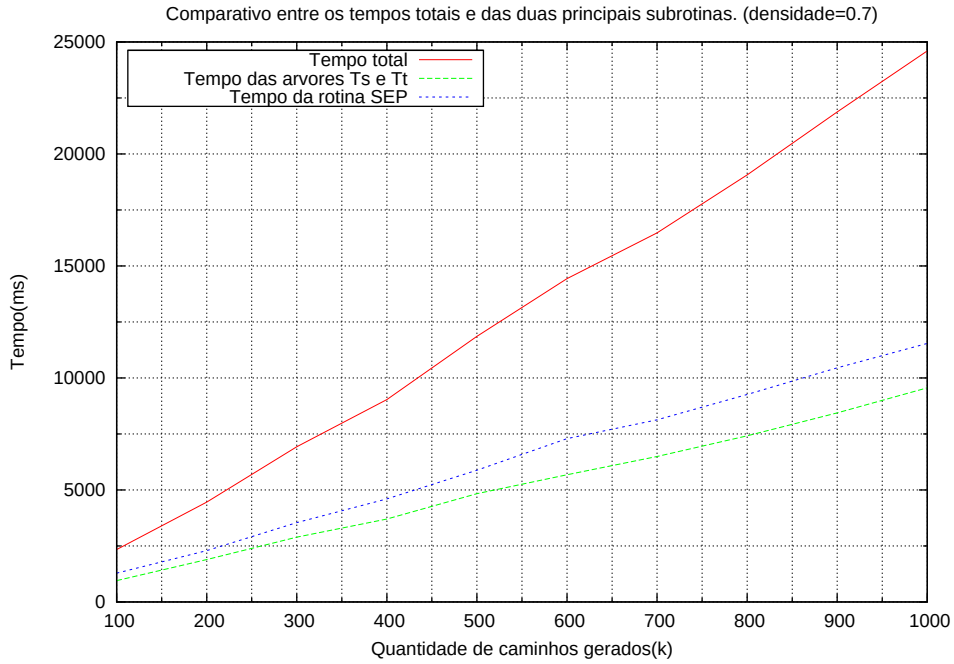


Figura 5.17: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.7

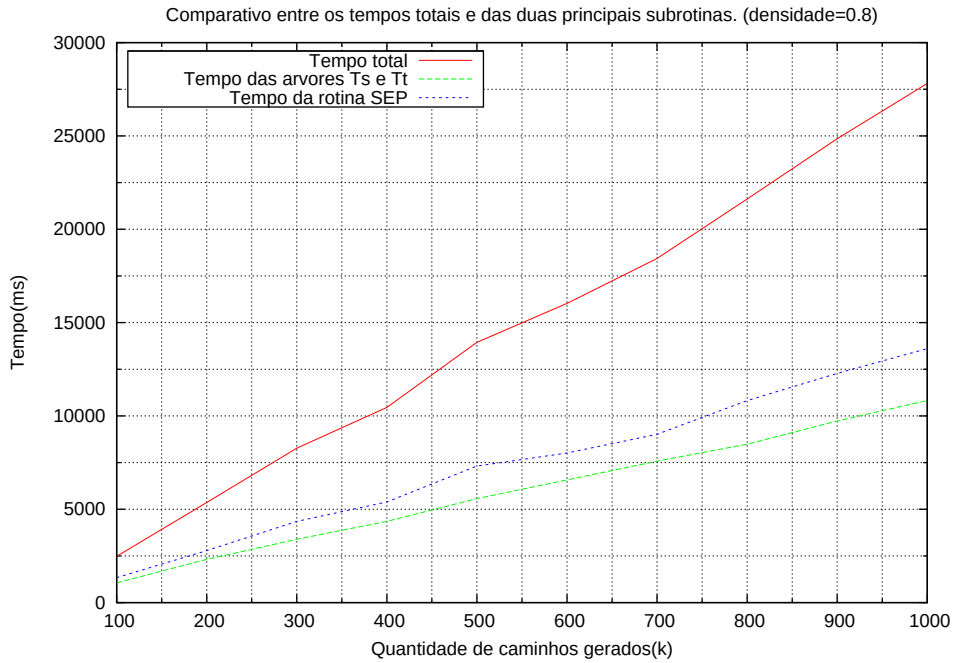


Figura 5.18: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.8

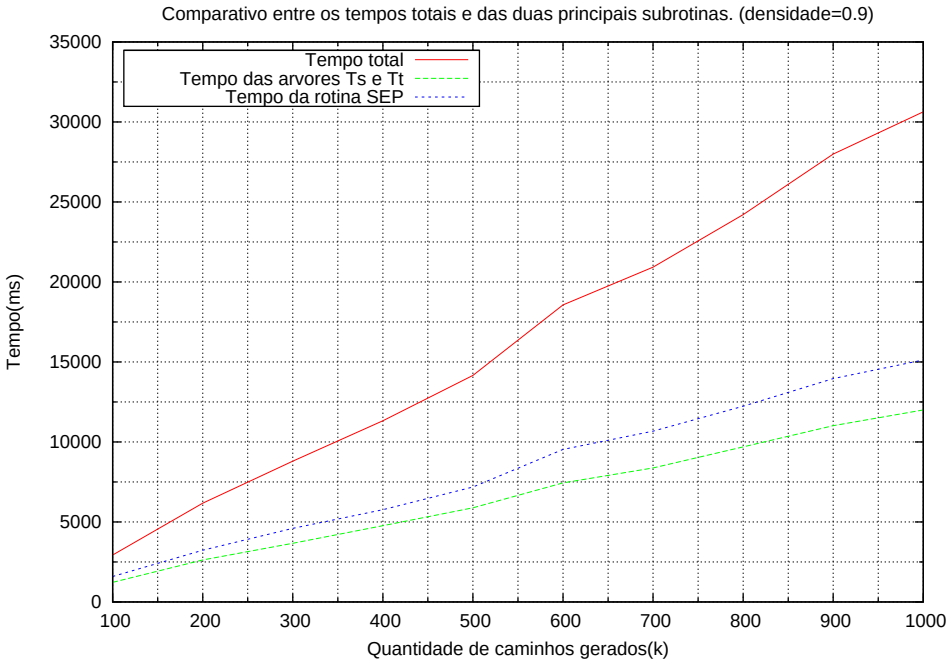


Figura 5.19: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 0.9

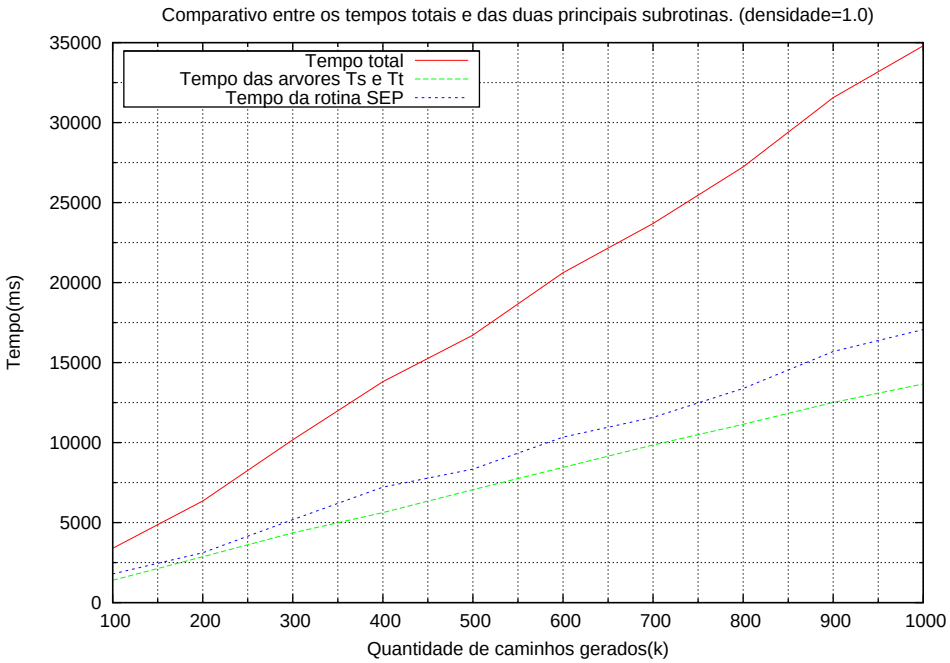


Figura 5.20: Comparativo entre as principais subrotinas e o tempo total do KIM. Densidade 1.0

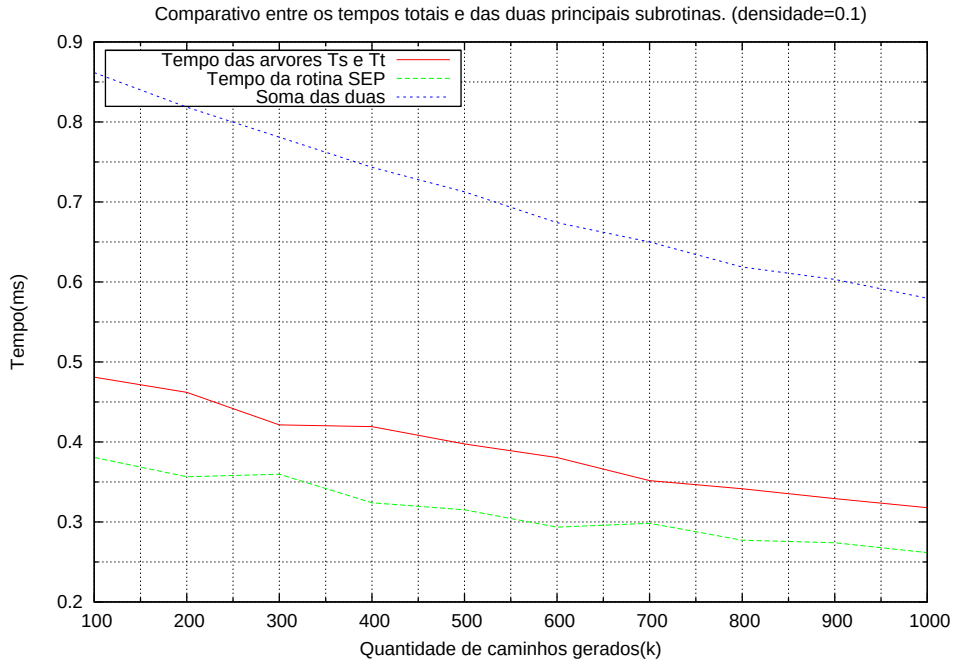


Figura 5.21: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.1

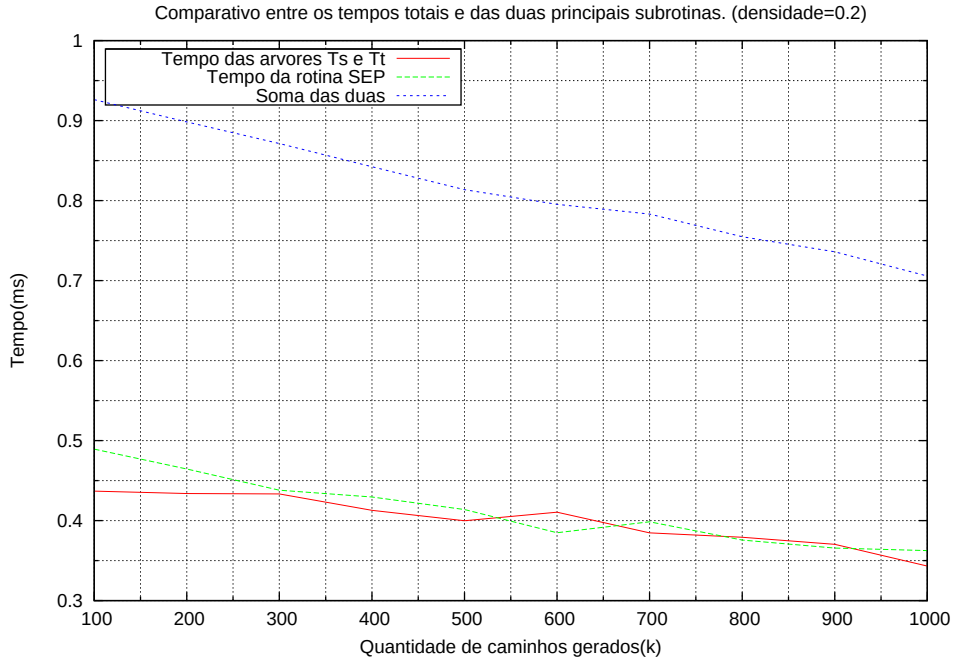


Figura 5.22: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.2

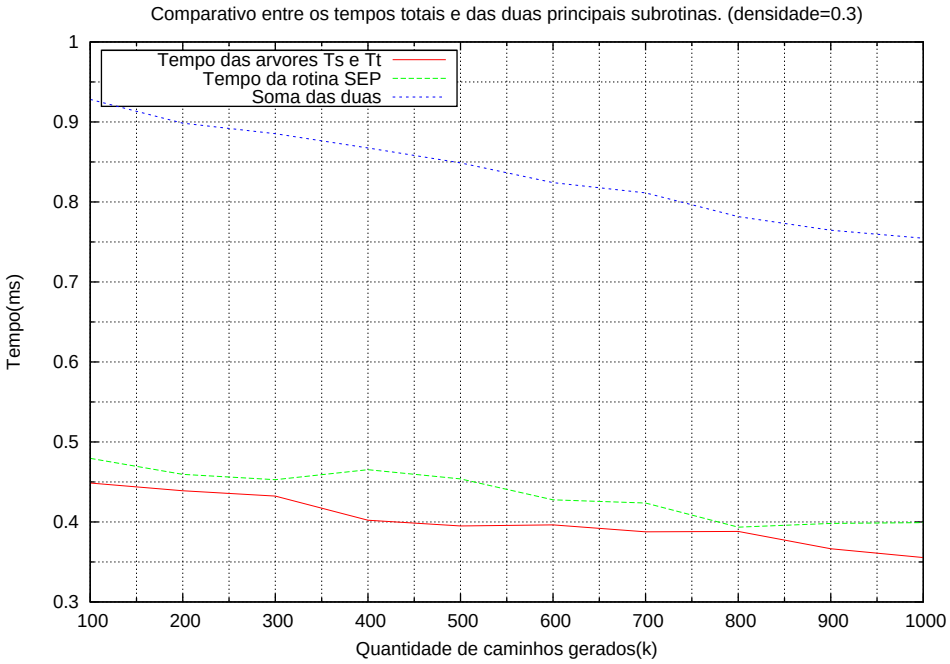


Figura 5.23: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.3

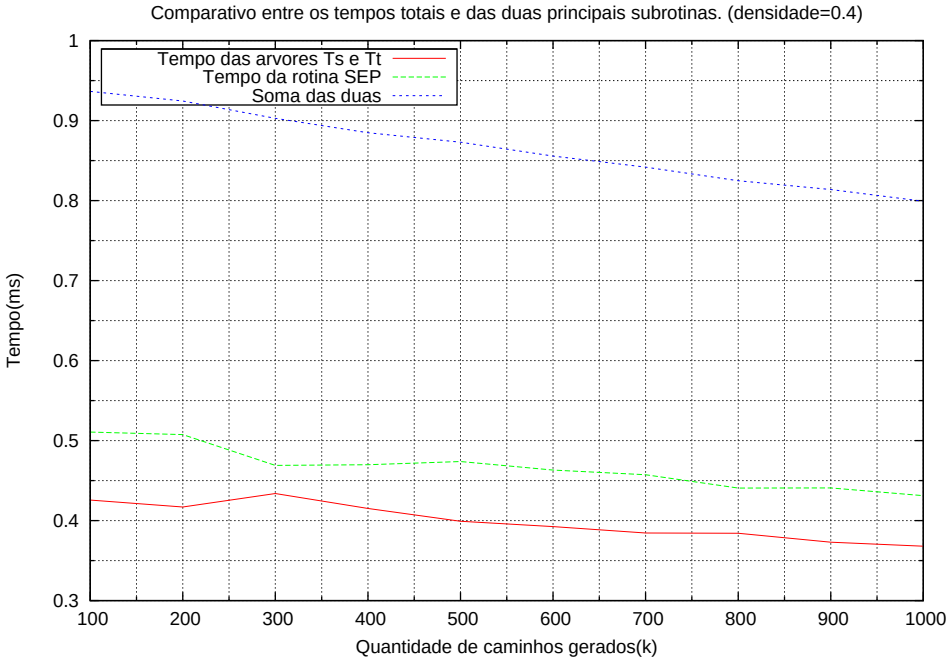


Figura 5.24: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.4

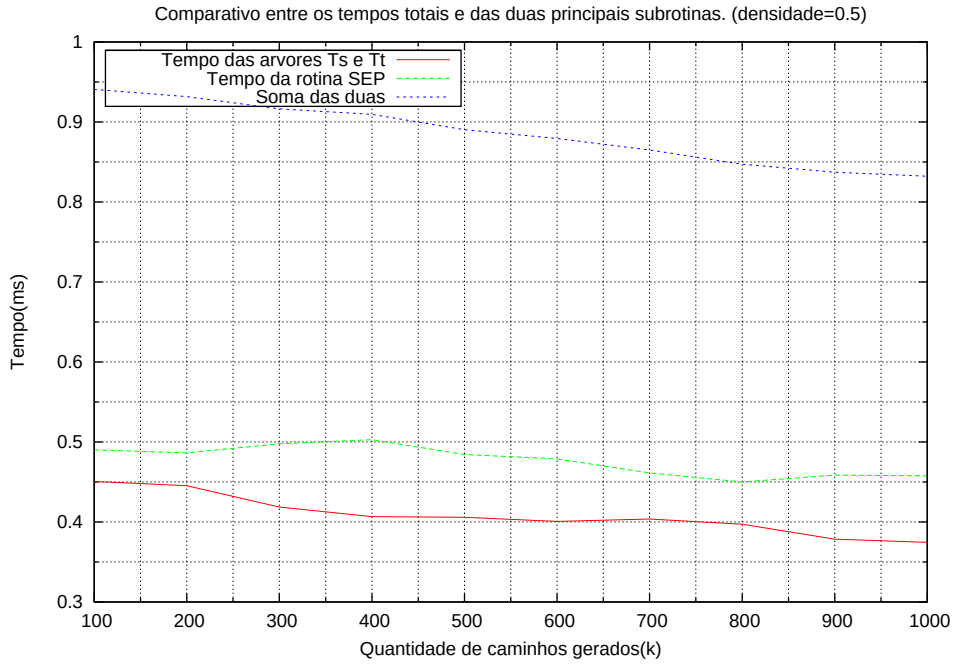


Figura 5.25: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.5

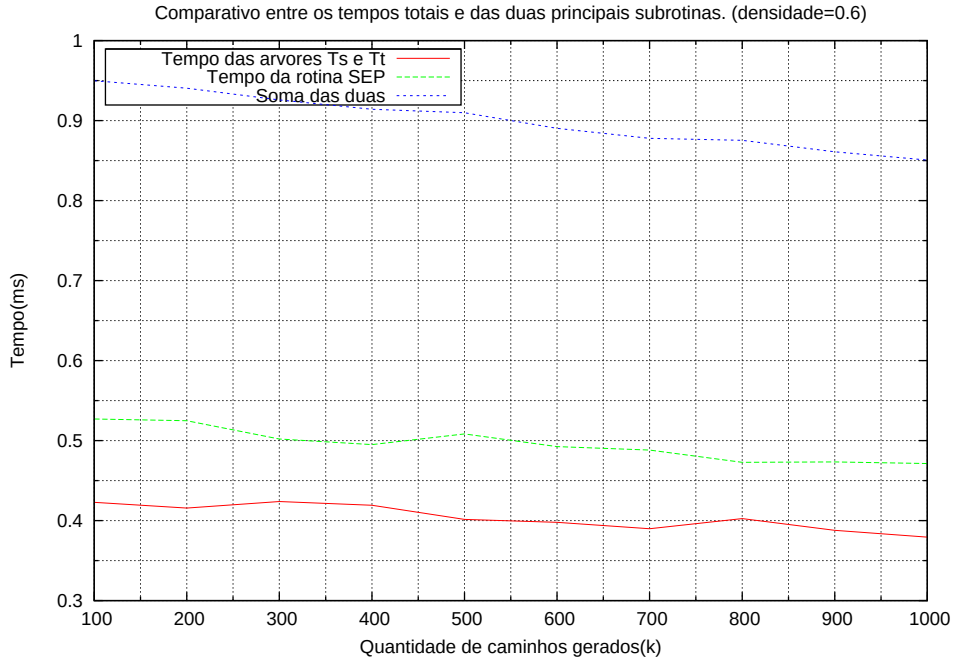


Figura 5.26: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.6

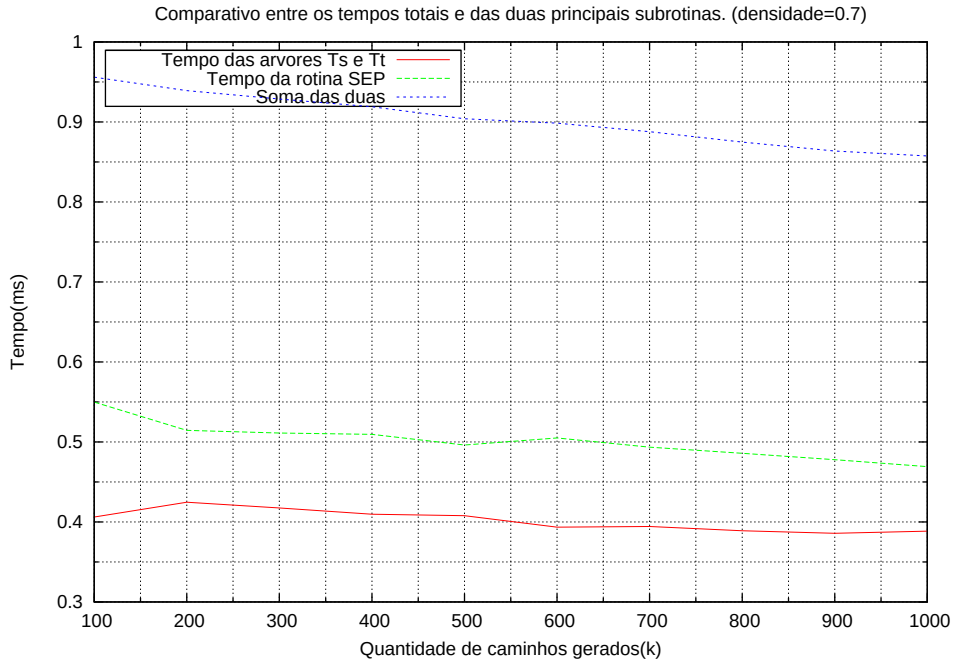


Figura 5.27: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.7

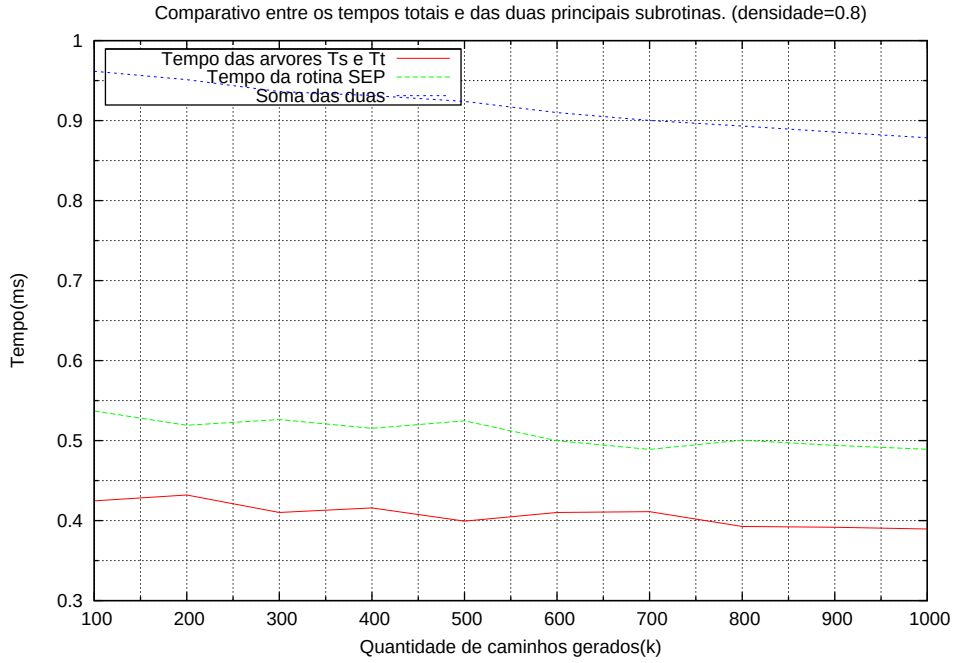


Figura 5.28: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.8

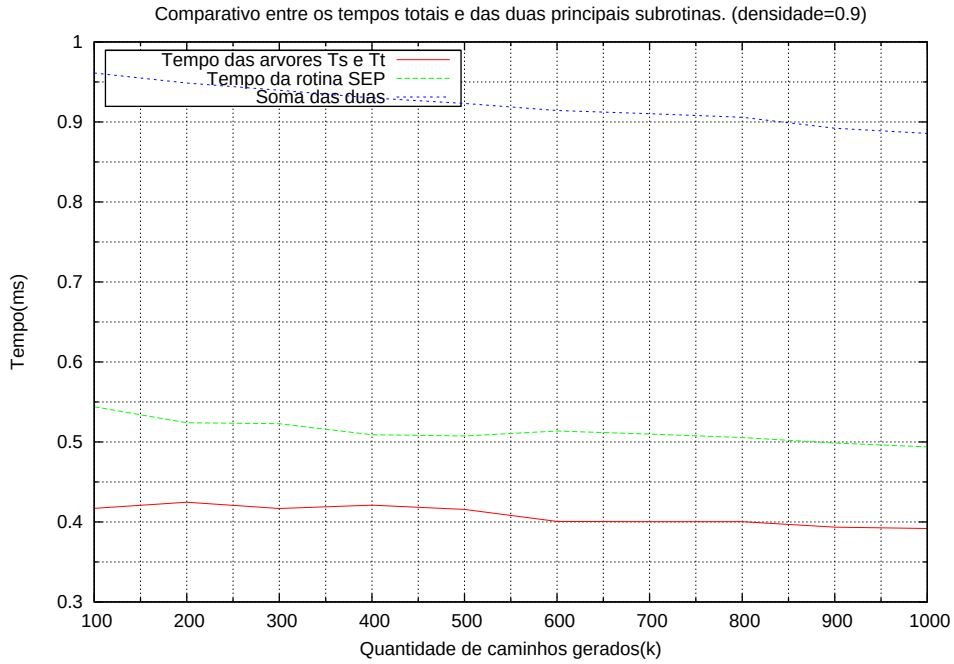


Figura 5.29: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 0.9

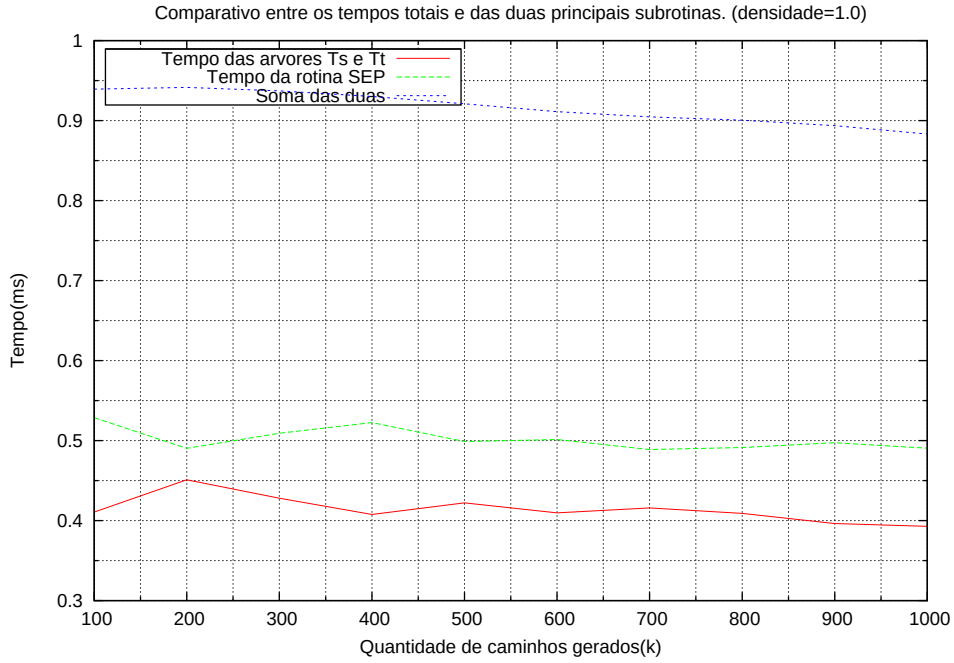


Figura 5.30: Comparativo entre a proporção de tempo utilizada pelas duas principais subrotinas do KIM. Densidade 1.0

CAPÍTULO 6

Considerações finais

6.1 Histórico

Meu primeiro contato com o *k*-CM foi através do problema apresentado na introdução. Inicialmente, estudei o algoritmo de Naoki Katoh, Toshihide Ibaraki e H. Mine [20] (KIM) de uma maneira não-acadêmica, objetivando implementá-lo. Mais tarde, decidimos usar o *k*-CM e os algoritmos para resolvê-los como tema central do mestrado e passamos a estudar o algoritmo de KIM sobre um novo ponto de vista: entender sua essência, subrotinas, peculiaridades e, além disso, procurarmos semelhanças e diferenças com os demais algoritmos existentes para o mesmo problema.

O foco inicial do trabalho foi entender o algoritmo KIM para, num momento posterior, estudar a viabilidade de algumas mudanças experimentais que pudessem melhorar seu desempenho em grafos especiais ou, quem sabe, até em grafos genéricos. Durante a implementação do algoritmo KIM realizada na TeleMax, tive algumas idéias para melhorar o seu desempenho para o grafo que representava a rede de dados da TeleMax. Devido aos prazos curtos e, principalmente ao fato da implementação ter atendido aos requisitos de desempenho, não foi possível justificar orçamento para a análise e implementação das melhorias pensadas. Gostaria, nesta dissertação de mestrado, de apresentar a motivação do estudo do algoritmo KIM para o problema *k*-CM, estudá-lo à luz do método de Yen do qual ele é derivado, descrevê-lo de uma maneira mais simples, sem toda a especificidade de um pseudo-código, apresentar algumas melhorias, implementá-las e avaliar os seus desempenhos.

Estudamos o artigo de KIM, bastante adequado para aqueles que desejam apenas implementar o algoritmo, uma vez que o pseudo-código é apresentado em grande de-

talhe. A linguagem bastante carregada dificultou um entendimento do algoritmo em linhas gerais, razão que nos levou a buscar outra fonte. Embora não seja apenas um novo artigo sobre o algoritmo KIM, o trabalho de John Hershberger, Matthew Maxel e Subhash Suri [16] é classificado pelos autores como uma extensão do algoritmo KIM para grafos dirigidos. O grande mérito deste artigo, do nosso ponto de vista, não é o da apresentação de um novo algoritmo para o problema k-CM, mas sim pela descrição do problema k-CM e das idéias subjacentes na elaboração do algoritmo, apresentadas de uma maneira bem mais simples de ser compreendida, sem o abuso de notações pesadas, como as do artigo KIM.

Após algumas horas de estudo do artigo de Hershberger, Maxel e Suri, decidimos dar mais atenção ao método base para o problema k-CM, o método de Yen. Nosso objetivo era encontrar os fundamentos e as idéias mais gerais que permeavam, segundo nosso entendimento, todos os algoritmos para o problema k-CM.

A descrição do método de Yen [29] é bem sucinta, mas foi suficiente para entendermos algumas idéias. O plano agora é nos debruçarmos sobre o artigo de KIM, tendo como bagagem o aprendizado ganho dos trabalhos de Yen e de Hershberger, Maxel e Suri, lembrando que uma revisão destes artigos é recomendada. Após um entendimento melhor, passaremos à redação de um descrição mais "alto-nível" do algoritmo de KIM.

6.2 Trabalhos futuros

Nem tudo o que pretendíamos coube no tempo estipulado. Dentre os trabalhos futuros que gostaríamos de realizar, ou ao menos deixar como sugestão ao leitor, citamos experimentar algumas mudanças no algoritmo KIM, e avaliar o quanto elas significam em ganho de desempenho. De antemão, sabemos que estas mudanças não acarretarão em melhoras assintóticas, mas acreditamos que conseguiremos alcançar desempenhos significativamente superiores. Como o algoritmo KIM tem como subrotina a geração de árvores de caminhos mínimos e, como foi constatado em Eleni Hadjiconstantinou e Nicos Christofides [15] que essa subrotina responde pela maior parte do processamento do algoritmo, estudaríamos o algoritmo para a reconstrução de árvores de caminhos mínimos descrito por Enrico Nardelli, Guido Proietti e Peter Widmayer [25]. Resumidamente, se trata de um algoritmo para o seguinte problema: dada uma árvore de caminhos mínimos para um grafo G encontrar a árvore de caminhos mínimos para

o grafo G' derivado de G pela remoção de algumas arestas e vértices. Acreditamos que melhorias neste ponto do algoritmo possam levar a grandes ganhos de desempenho. O artigo de Alberto Marchetti-Spaccamela e Umberto Nanni [22] também está relacionado ao problema de reconstrução de árvores e mereceria um estudo também.

6.3 Experiência

Sobre a implementação gostaríamos de dizer que foi uma experiência bastante enriquecedora. Inicialmente possuíamos uma implementação própria, a qual não se mostrou adequada, primeiramente por funcionar apenas para grafos sem custos e segundo por estar muito vinculado ao trabalho realizado na empresa onde trabalhei. Pretendíamos fazer uma implementação que pudesse ser usada no caso mais geral possível e, que fizesse uso de alguma biblioteca pública para manipulação de grafos. As razões vão desde a maior aceitação do código por parte da comunidade open-source, até a reutilização de código, passando também pela possibilidade de utilizar código para visualização gráfica.

Começamos buscando uma biblioteca bem aceita e utilizada e que fosse implementada em JAVA, devido a minha maior familiaridade com esta. Encontramos a biblioteca JUNG, a qual se mostrou bem apropriada aos nossos propósitos. O passo seguinte foi entender um pouco do seu funcionamento e suas estruturas de dados. Em seguida, começamos a implementação do algoritmo de KIM. Neste momento, nos deparamos com diversos problemas na reutilização de código, cito aqui, principalmente, a rotina de geração de menor caminho utilizando o algoritmo de Dijkstra. Foi preciso pensar bastante até descobrirmos uma maneira de aproveitar esta rotina e foi muito prazeroso ver o resultado obtido. Durante a implementação nos valemos bastante das saídas gráficas de alguns pontos do algoritmo o que ajudou bastante na identificação de erros. Houve diversos obstáculos durante a implementação e, sempre é possível que se faça uma implementação mais enxuta e eficiente quanto maior for o conhecimento sobre o assunto e a biblioteca. Esperamos que nosso código venha a ser incorporado ao rol de funções existentes no JUNG e com isso contribuir para o projeto open source que foi de grande ajuda no desenvolvimento deste trabalho.

Referências Bibliográficas

[1] Ravindra K. Ahuja, Thomas L. Magnanti e James B. Orlin, *Network flows: Theory, algorithms, and applications*, Practice Hall, 1993. Citado na(s) página(s) 9

[2] Ravindra K. Ahuja, Kurt Mehlhorn, J.B. Orlin e R.E. Tarjan, Faster algorithms for the shortest path problem, *Journal of the ACM* **37** (1990), 213–223. Citado na(s) página(s) 34

[3] A.W. Brander e Mark C. Sinclair, A comparative study of *k*-shortest path algorithms, *11th UK Performance Engineering Workshop for Computer and Telecommunications Systems*, 1995, pp. 370–379. Citado na(s) página(s) 48

[4] B.V. Cherkassky, A.V. Goldberg e T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, *SODA: ACM-SIAM Symposium on Discrete Algorithms(A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994. Citado na(s) página(s) 67

[5] B.V. Cherkassky, A.V. Goldberg e C. Silverstein, Buckets, heaps, lists and monotone priority queues, *SIAM J. Comput.* **28** (1999), 1326–1346. Citado na(s) página(s) 34, 67

[6] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest, *Introduction to algorithms*, McGraw-Hill, 1999. Citado na(s) página(s) 9

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, *Introduction to algorithms*, 2nd. ed., The MIT Press and McGraw-Hill, 2001. Citado na(s) página(s) 24, 35

[8] Edsger Wybe Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik* **1** (1959), 269–271. Citado na(s) página(s) 19, 20, 25, 41

[9] David Eppstein, Finding the *k* shortest paths, *SIAM Jounal on Computing* **28** (1998), no. 2, 652–673. Citado na(s) página(s) 48

[10] Paulo Feofiloff, *Notas de aula de MAC5781 otimização combinatória*, `"http://www.ime.usp.br/~pf/"`, 1997. Citado na(s) página(s) 41

[11] _____, *Algoritmos de programação linear*, EDUSP, 2000. Citado na(s) página(s) 21

[12] _____, *Fluxo em redes*, `"http://www.ime.usp.br/~pf/flows/"`, 2003. Citado na(s) página(s) 5, 19, 25

[13] Michael L. Fredman e Robert Endre Tarjan, Fibonacci heap and their uses in improved network optimization algorithms, *Journal of the ACM* **34** (1987), 596–615. Citado na(s) página(s) 32, 34, 35, 42

[14] A.V. Goldberg e C. Silverstein, *Implementation of Dijkstra's algorithm based on multilevel buckets*, Tech. report, NEC Research Institute, Princeton, NJ, 1995. Citado na(s) página(s) 67

[15] Eleni Hadjiconstantinou e Nicos Christofides, An efficient implementation of an algorithm for finding *k* shortest simple paths, *Networks* **34** (1999), no. 2, 88–101. Citado na(s) página(s) 48, 67, 69, 90

[16] John Hershberger, Matthew Maxel e Subhash Suri, Finding the *k* shortest simple paths: A new algorithm and its implementation, *ACM Transactions on Algorithms* **3** (2007), no. 4, 19 pages. Citado na(s) página(s) 47, 48, 51, 90

[17] Shigueo Isotani, *Algoritmos para caminhos mínimos*, Master's thesis, Instituto de Matemática e Estatística, 2002. Citado na(s) página(s) 19

[18] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977), 1–13. Citado na(s) página(s) 32

[19] D.S. Johnson, A theoretician's guide to the experimental analysis of algorithms, *To appear in Proceedings of the 5th and 6th DIMACS Implementation Challenges*, 2002. Citado na(s) página(s) 67

[20] Naoki Katoh, Toshihide Ibaraki e H. Mine, An efficient algorithm for *k* shortest simple paths, *Networks* **12** (1982), no. 4, 411–427. Citado na(s) página(s) 3, 4, 48, 51, 89

[21] Eugene L. Lawler, A procedure for computing the *k* best solutions to discrete optimization problems and its application to the shortest path problem, *Management Science* **18** (1972), no. 7, 401–405. Citado na(s) página(s) 48

[22] Alberto Marchetti-Spaccamela e Umberto Nanni, Fully dynamic algorithms for maintaining shortest path trees, *Journal of Algorithms* **34** (2000), 251–281. Citado na(s) página(s) 91

[23] Ernesto Martins e Marta Pascoal, A new implementation of Yen's ranking loopless paths algorithm, *4OR Quart. J. Belgian, French, Italian Oper. Res. Soc.* **1** (2003), no. 2, 121–134. Citado na(s) página(s) 48

[24] Ernesto Martins, Marta Pascoal e José Santos, *A new algorithm for ranking loopless paths*, Tech. report, Universidade de Coimbra, May 1997. Citado na(s) página(s) 48

[25] Enrico Nardelli, Guido Proietti e Peter Widmayer, Swapping a falling edge of a single source shortest paths tree is good and fast, *Algorithmica* **35** (2003), 56–74. Citado na(s) página(s) 90

[26] A. Perko, Implementation of algorithms for *k* shortest loopless paths, *Networks* **16** (1986), 149–160. Citado na(s) página(s) 48

[27] Robert Endre Tarjan, *Data structures and network algorithms*, BMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA, 1983. Citado na(s) página(s) 24

[28] Mikkel Thorup, Undirect single source shortest paths with positive integer weights in linear time, *Journal of the ACM* **46** (1999), 362–394. Citado na(s) página(s) 42

[29] Jin Y. Yen, Finding the *k* shortest loopless paths in a network, *Management Science* **17** (1971), no. 11, 712–716. Citado na(s) página(s) 43, 48, 51, 90

Índice Remissivo