k-menores caminhos

Fábio Pisaruk

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: Mestrado em Ciência da Computação
Orientador: Prof. Dr. José Coelho de Pina Júnior.

São Paulo, julho de 2009

k-menores caminhos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Fábio Pisaruk
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. José Coelho de Pina Júnior (Presidente) - IME-USP.
- Prof. Dr. José Augusto Ramos Soares - IME-USP.
- Prof. Dr. Orlando Lee - UNICAMP.

Aos meus pais Paulo e Roseli
e meu irmão Marcos.

Agradecimentos

Gostaria de agradecer a Deus, meus pais e amigos.

Ao professor José Coelho, pela sua atenção, motivação e grande paciência.

A minha namorada Fabiana que esteve comigo durante todo o tempo da faculdade.

A todo o pessoal da Telefonica que me acompanhou durante a implementação do algoritmo, em especial, Arcindo, Givanildo e Manoel, grande amigos.

Aos amigos de trabalho e faculdade: Lucas Petri, Flávio Daher e Luciana Delfini, que me acompanharam durante boa parte deste trabalho.

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.

Abstract

We consider a long-studied generalization of the shortest path problem, in which not one but several short paths must be produced. The k-shortest (simple) paths problem is to list the k paths connecting a given source-destination pair in the digraph with minimum total length.

This dissertation deals with k-shortest simple paths algorithms designed for non-negative costs, undirected graphs and some implementations of them.

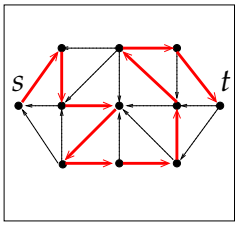
Keywords: shortest paths, k-shortest paths.

Sumário

Lista de Figuras

Lista de códigos

Introdução



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.

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

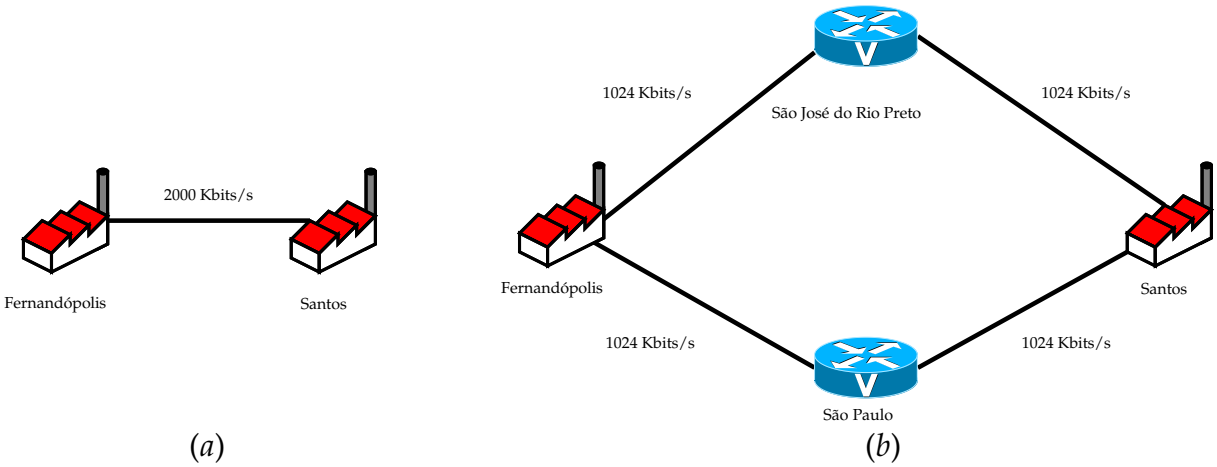


Figura 1: Em (*a*) vemos um esquema solicitação da SoftSOF, um link de 2000 Kbits/s entre as filiais. Em (*b*) está descrita a solução encontrada pela TeleMax, com base na disponibilidade de sua rede.

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 equipamentos 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 [36], 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 usadas 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 explicando o algoritmo de YEN que as utiliza.

No capítulo 4 apresentamos o algoritmo desenvolvido por John Hershberger, Matthew Maxel e Subhash Suri que é um refinamento do algoritmo de Yen e é semelhante ao algoritmo desenvolvido por Naoki Katoh, Toshihide Ibaraki e H. Mine, o qual é apresentado no capítulo 5.

No capítulo 5 o método desenvolvido por Naoki Katoh, Toshihide Ibaraki e H.

Mine [36] é descrito mais detalhadamente. Além disso, a implementação feita é exibida juntamente com uma simulação de sua execução.

Seguimos, no capítulo 6, com uma série de gráficos e análises experimentais do desempenho da nossa implementação do algoritmo de KIM.

Finalmente, no capítulo 7, 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 [18].

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 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 $\{uv, vu\}$ 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.

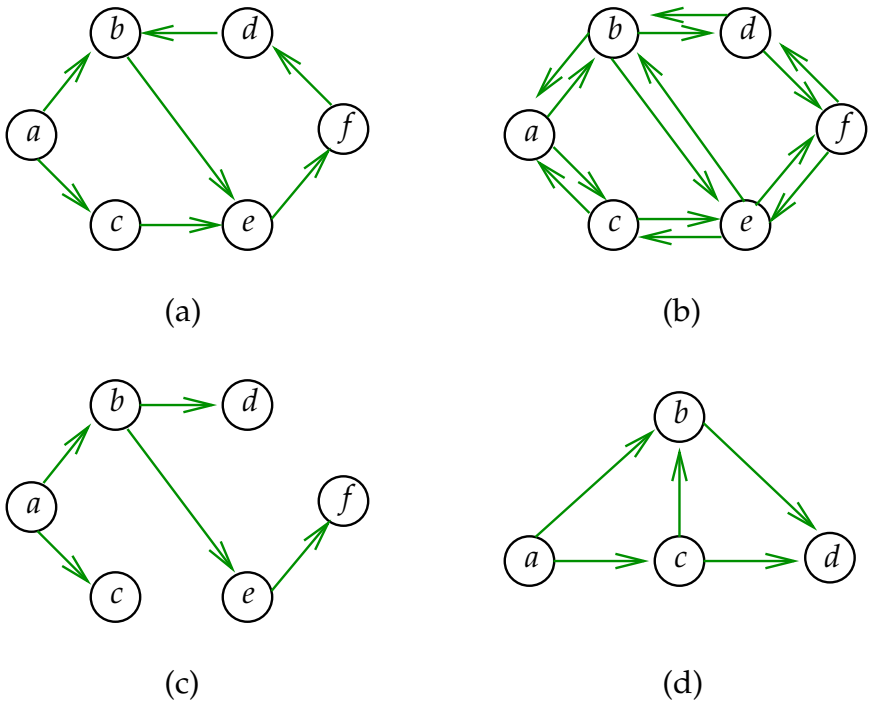


Figura 1.1: (a), (b), (c) e (d) são exemplos de grafos. (b) é um grafo simétrico.

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

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

onde $v_0, \ldots, v_q$ são vértices, $a_1, \ldots, a_q$ 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**.

Um ciclo é um passeio onde $v_0 = v_t$, ou seja começa e termina no mesmo vértice. Um grafo é chamado de acíclico se não possuir ciclos.

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 $P := \langle v_0, a_1, v_1, \ldots, a_t, v_q\rangle$ 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 $0 \leq i \leq j \leq q$ será um **subpasseio** de P. Além disso, se $i = 0$, então o subpasseio será dito um **prefixo** de P e se $j = q$ então o subpasseio é dito um **sufixo** 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 $\langle a, ab, b, be, e, ef, f, fd, d, db, b, be, e, ef, f\rangle$ e a seqüência $\langle e, ef, f, fd, d, db, b, be, e, ef, f\rangle$ é um sufixo.

Um **caminho** é um passeio sem vértices repetidos. Na figura 1.1(a) o passeio $\langle a, ab, b, be, e, ef, f\rangle$ é um caminho com início em a e término em f. Se num ciclo apenas a ponta inicial e a ponta final coincidem, então dizemos que esse ciclo é um **circuito**. Na figura 1.1(a) o passeio $\langle b, be, e, ef, f, fd, d, db, b\rangle$ é um circuito.

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

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

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 de 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. Esta representação é particularmente útil quando modelamos problemas de otimização combinatória através de programas lineares [11, 45].

	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.

$$\begin{array}{ll} A(a): & ab, \quad ac \\ A(b): & bd \\ A(c): & cb, \quad cd \\ A(d): & \end{array}$$

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

1.4 Filas de prioridade

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

(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 prosseguir, 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 abstrato 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 (relacionadas) de diversas maneiras.

Uma **fila de prioridades** [1, 12] é um tipo abstrato de dados que consiste de uma coleção de itens, cada um com um valor ou prioridade associada. Nos algoritmos tratados neste texto os itens serão, basicamente, vértices.

Uma fila de prioridade tem suporte para as seguintes operações:

- INSERT(v, val): adiciona o vértice v com valor val na 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 vértice 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 sobre os vértices de um grafo.

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 e grafos simétricos. 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 e grafos simétricos, distinguindo-os dos demais, foram criadas duas interfaces: DirectedGraph e UndirectedGraph e as respectivas DirectedEdge (arco) e UndirectedEdge (aresta). Estas interfaces permitem validações em tempo de compilação. Nenhuma delas possui 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 validação fica a cargo da implementação da mesma.

A seguir temos as implementações das interfaces citadas acima. Numa camada intermediária, existem três classes abstratas implementando funcionalidades comuns a grafos e grafos 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 poucos arcos, se comparado à quantidade máxima possível, são, em geral, representados de maneira mais eficiente e econômica usando-se listas de adjacência e grafos densos como matrizes de adjacência.

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

Criação de grafos e grafos simétricos

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 baseado numa representação na forma de lista de adjacência,

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

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

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);
```

e depois adicionar-lhe os arcos:

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

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.

Também é possível criar um grafo a partir de dados de um arquivo. Apresentaremos apenas um exemplo usando o padrão Pajek [3] uma vez que o JUNG é capaz de ler e escrever apenas neste formato (o formato GraphML é suportado somente no modo

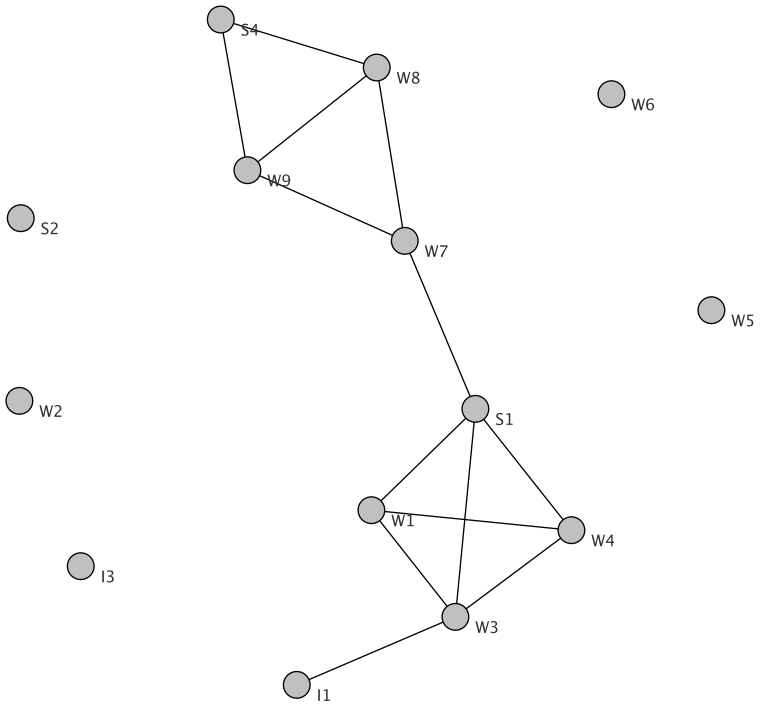
leitura). O formato Pajek é muito abrangente, permitindo definições bem complexas de grafos, contudo apresentaremos apenas alguns exemplos simples de seu uso.

Começaremos mostrando um arquivo no formato Pajek representando um grafo simétrico com 14 vértices, sem custos nas arestas:

```
1  *Vertices 14
2  1 I1
3  2 I3
4  3 W1
5  4 W2
6  5 W3
7  6 W4
8  7 W5
9  8 W6
10 9 W7
11 10 W8
12 11 W9
13 12 S1
14 13 S2
15 14 S4
16 *Edges
17 1 5
18 3 5
19 3 6
20 5 6
21 9 10
22 9 11
23 10 11
24 3 12
25 5 12
26 6 12
27 10 14
28 11 14
29 9 12
```

Os vértices são numerados a partir do 1. Cada vértice pode possuir um rótulo, que deve vir após seu número, por exemplo, o vértice número 2 tem o rótulo *I3*. Para definir as arestas usamos *Edges, como exibido na linha 16. Cada aresta é definida pelos dois vértices que conecta, por exemplo, na linha 17 temos uma aresta conectando os vértices 1 e 5. Vale lembrar que o arquivo não pode conter linhas em branco.

A figura a seguir mostra uma ilustração do grafo simétrico descrito acima.



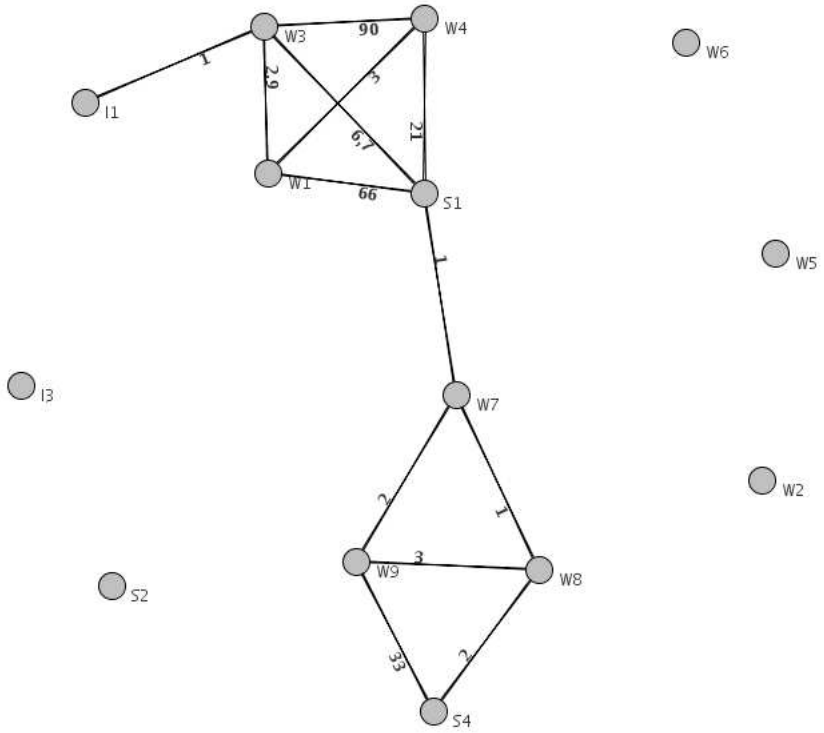
Agora vamos criar um grafo simétrico como o anterior, mas com custos em suas arestas:

```
*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
```

```
20          5  6  90
21          9  10  1
22          9  11  2
23          10  11  3
24          3  12  66
25          5  12  6.7
26          6  12  21
27          10  14  2
28          11  14  33
29          9  12  1
```

Observe que as arestas contêm três números, onde os dois primeiros correspondem aos vértices que ela conecta e o terceiro ao seu custo. Por exemplo, na linha 18, temos a aresta conectando os vértices 3 e 5 com custo 2.9.

A figura a seguir mostra uma ilustração do grafo simétrico descrito acima.



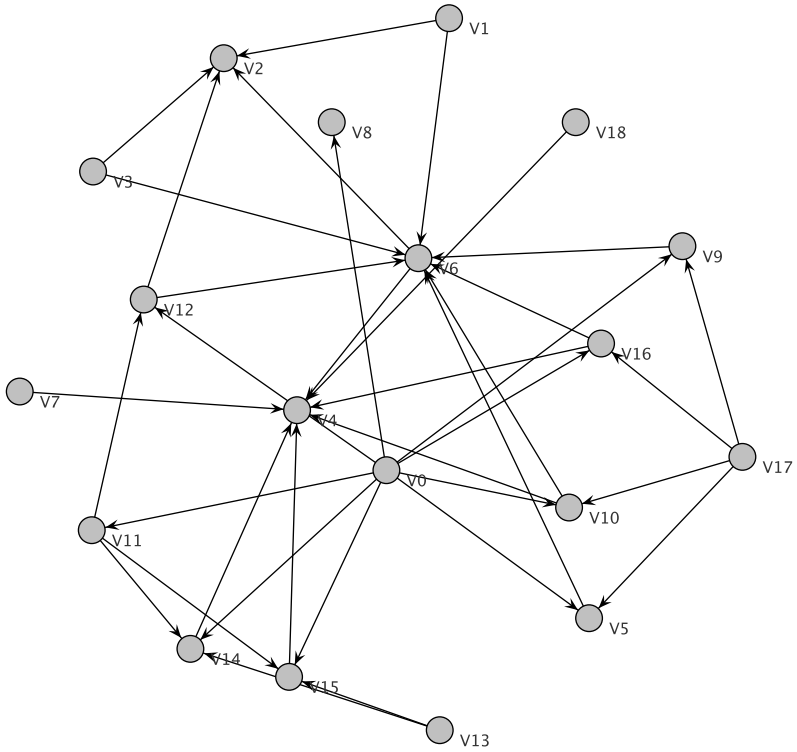
A criação de grafos é um pouco diferente. Lembramos que em nosso trabalho estamos apenas interessados em grafos simétricos. No entanto, apenas à título de curiosidade, exibiremos um exemplo.

A seguir temos a descrição de um grafo com 19 vértices, sem custo nos arcos:

```
1  *Vertices 19
2  *Arcslist
3  1 4 6 17 5 13 12 9 8 7
4  3 13 12 8 7
5  4 6 5 9
6  2 6 5
7  13 16
8  12 16
9  9 16
10 9 19
11 8 16 18
12 7 16 18
13 15 16 19
14 14 16 19
15 6 18
16 5 18
17 16 19 18
18 11 18
19 10 18
```

Observe que não há nenhuma linha contendo o número do vértice seguido de seu rótulo. Sendo assim, assume-se que os vértices são numerados de 1 a 19 e rotulados de V0 a V18. Note que o vértice de número 1 não é necessariamente rotulado por V0, 2 por V1, e assim por diante. No nosso exemplo, o vértice 3 é rotulado como V17. A definição dos arcos é um pouco diferente do que vimos anteriormente quando trabalhamos com grafos simétricos e arestas. Em primeiro lugar, usamos *Arcslist no lugar de *Edges e, uma vez que nosso grafo não tem custo nos arcos, podemos defini-los em função dos vértices adjacentes. Por exemplo, na linha 4 definimos arcos que conectam o vértice 3 aos vértices: 13, 12, 8 e 7.

A figura a seguir mostra uma ilustração do grafo descrito acima.



A leitura de um arquivo contendo um grafo ou grafo simétrico no formato Pajek consiste dos seguintes passos:

(1) criação de um leitor PajekNet;

(2) criação de um objeto a partir de uma classe referente ao tipo de grafo desejado. Lembre-se que temos grafos e grafos simétricos;

(3) definição da maneira pela qual os custos são atribuídos às arestas ou aos arcos.

O trecho de código a seguir ilustra os passos descritos acima:

```
1        PajekNetReader pajekNetReader = new PajekNetReader(false);
2        UndirectedGraph g = new UndirectedSparseGraph();
3        NumberEdgeValue nev = new UserDatumNumberEdgeValue(g);
4        g = (UndirectedGraph) pajekNetReader.load("data/pajNetTest.dat", g,
             nev);
```

Na linha 1 temos a criação do leitor PajekNet, usando a classe PajekNetReader. Na linha 2 criamos o objeto g referente a um grafo simétrico. Em seguida, na linha 3 defi-

nimos como os custos são atribuídos aos vértices, neste caso o repositório [1] do usuário contém esses custos, os quais são lidos do arquivo PajekNet. Finalizamos, na linha 4, criando o grafo simétrico a partir dos dados do arquivo: data/pajNetTest.dat.

Atribuição de custos às arestas ou aos arcos

O JUNG permite flexibilidade na maneira como os custos são atribuídos aos arcos e às arestas. Para tal, existe a interface NumberEdgeValue que define dois métodos: getNumber e setNumber. A idéia é deixar o desenvolvedor livre para criar qualquer tipo de implementação que defina os custos dos arcos/arestas do seu grafo/grafo simétrico. A biblioteca JUNG já conta com quatro implementações desta interface: ConstantDirectionalEdgeValue, ConstantEdgeValue, EdgeWeightLabeller e UserDatumNumberEdgeValue Em nosso trabalho usamos apenas duas dessas:

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 UserDatumNumberEdgeValue.

Armazenando dados no grafo

Citamos, anteriormente, o repositório de dados do usuário, o qual é apenas uma das formas disponibilizadas pelo JUNG para permitir ao usuário armazenar dados no grafo. Além disso, é possível adicionar dados aos arcos (grafo), arestas (grafo simétrico) e vértices. Para isso, o usuário pode optar por especializar uma classe que implemente a interface ArchetypeVertex ou utilizar os métodos de anotações oferecidos. Explicaremos melhor como funcionam estes dois métodos a seguir:

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

[1] O repositório do usuário será apresentado quando tratarmos do armazenamento de informações nos elementos contituintes de um grafo na seção Anotações.

definam e obtenham este dado, como é mostrado no exemplo a seguir:

```
1  class MeuVertice extends DirectedSparseVertex {
2      private String nome;
3
4      public MeuVertice( String nome ) {
5          this.nome = nome;
6      }
7
8     public String getNome(){
9       return nome;
10    }
11
12    public void setNome(String nome){
13      this.nome=nome;
14    }
15 }
```

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 armazenado 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 armazenado em udc;

getUserDatumKeyIterator(): retorna um objeto de iteração que permite navegar pelos dados armazenados pelo usuário no seu repositório;

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

Adicionando a informação nome a um vértice:

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

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. O JUNG fornece três diferentes soluções, sendo que o usuário pode criar outras implementando a interface CopyAction, são elas:

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.

Mudanças na versão 2.0 do JUNG

Quando iniciamos a implementação do algoritmo KIM em Java usamos a versão mais recente da biblioteca JUNG naquele momento, ou seja, a versão 1.7. Pouco antes da finalização, a versão 2.0 foi disponibilizada e decidimos usá-la por diversos motivos, dentre eles:

- Flexibilidade para usar qualquer classe como vértice ou arco. Na versão anterior era preciso trabalhar com as interfaces AchetypeVertex e ArchetypeEdge;
- Simplificação da arquitetura de classes;
- Uso de *Generics* do Java.

Para começar, temos a interface Graph<V,E> a qual define as operações básicas que podem ser realizadas em um grafo, dentre elas:

- Adição e remoção de vértices e/ou arcos;
- Obtenção das pontas de um arco, ou seja, os vértices que este conecta;
- Obtenção de informações referentes aos vértices, tais como: grau, predecessores e sucessores.

Há também três interfaces que especializam a interface Graph<V,E>:

DirectedGraph<V, E>: usada para indicar que a classe que a implementa suportará apenas grafos;

UndirectedGraph<V, E>: usada para indicar que a classe que a implementa suportará apenas grafos simétricos;

SimpleGraph<V, E>: usada para indicar que a classe que a implementa suportará apenas grafos sem arcos paralelos ou loops.

As três interfaces apresentadas anteriormente são chamadas de interfaces de marcação e tem o propósito de validar, em tempo de compilação, o tipo de objeto sendo usado. Como dissemos no início deste capítulo, a validação quanto a simetria ou não do grafo deve ser implementada pela classe em questão. Por exemplo, a classe UndirectedSparseGraph<V,E>, a qual implementa a interface UndirectedGraph<V,E> é responsável por garantir que apenas arestas sejam adicionadas ao grafo simétrico, pois a interface, por si só, não garante tal restrição.

Na versão 2.0 do JUNG, diferentemente da anterior, qualquer objeto pode ser uma aresta, arco ou vértice. Isto permite uma maior flexibilidade, principalmente quando se trata de armazenar informações nos elementos do grafo. A seguir, exibiremos um exemplo de criação de um grafo simétrico na versão 2.0 da biblioteca.

```
1  Graph<Integer, String> g = new SparseUndirectedGraph<Integer, String>();
2  g.addVertex((Integer)1);
3  g.addVertex((Integer)2);
4  g.addVertex((Integer)3);
5  g.addEdge("Aresta-A", 1, 2);
6  g.addEdge("Aresta-B", 2, 3);
```

Na linha 1 criamos um grafo simétrico onde os vértices são objetos do tipo Integer e as arestas do tipo String. Em seguida, nas linhas 2-4, adicionamos três vértices ao grafo simétrico, representados pelos inteiros 1,2 e 3. Finalizamos, nas linhas 5 e 6, adicionando duas arestas conectando o vértice 1 ao 2 e 2 ao 3.

O uso de *Generics* juntamente com a flexibilidade do JUNG em permitir que qualquer tipo seja usado como elemento de um grafo, facilita muito a programação. Observe no exemplo anterior que não tivemos absolutamente nenhum trabalho extra para informar que os nosso vértices armazenavam inteiros. Além disso, quando obtivermos um vértice do grafo, teremos em mãos um inteiro e não um objeto do tipo Vertex que armazena um inteiro em seu repositório de dados.

Caso nossos vértices e/ou arcos sejam tipos mais complexos que simples inteiros ou textos, podemos utilizá-los como elementos do grafo bastando defini-los usando *Generics*. Suponha que desejamos que nossos vértices e arestas sejam dos tipos Estacao e Link, respectivamente, definidos da seguinte maneira:

```
1  class Estacao {
2    private String nome;
3    private int equipamentos;
4
5    public MyNode(String nome, in equipamentos) {
6      this.nome = nome;
7      this.equipamentos=equipamentos;
8    }
9
10   public String toString() {
11     return nome + " - " + equipamentos;
12   }
13
14   public String getNome(){
15     return nome;
16   }
17
18   public String getEquipamentos(){
19     return equipamentos;
20   }
21 }
22
23 class Link {
24   int numeroDeFibrasUsadas;
25   int numeroDeFibras;
26   double distancia;
27
28   public Link(int numeroDeFibrasUsadas, int numeroDeFibras, double
       distancia ) {
29     this.numeroDeFibrasUsadas=numeroDeFibrasUsadas;
30     this.numeroDeFibras=numeroDeFibras;
31     this.distancia=distancia;
32   }
33 }
```

Para criar um grafo simétrico utilizando eses dois tipos poderíamos, simplemente, escrever: Graph<Estacao, Link> g = **new** SparseUndirectedGraph<Estacao, Link>(); e, adicionar arestas e vértices de maneira semelhante ao exemplo anterior:

```
1  Estacao ipiranga = new Estacao("IPIRANGA",1000);
2  Estacao jabaquara = new Estacao("JABAQUARA",200);
3  g.addVertex(ipiranga);
4  g.addVertex(jabaquara);
5  g.addEdge("ipiranga-jabaquara",ipiranga,jabaquara);
```

Um outro ponto que sofreu grande modificação nesta nova versão se refere a atribuição de custos às arestas. Na versão anterior, tudo girava em torno da interface NumberEdgeValue e suas respectivas implementações. Na versão 2.0, o JUNG começou a fazer uso do padrão de projeto (design pattern [22]) *Transformer*. O padrão *Transformer* é, de certa forma, uma versão do padrão *Visitor*, onde a operação a ser realizada no elemento é a transformação deste em outro tipo.

O uso desse padrão, em conjunto com a parametrização de tipos permitida pelo uso do *Generics* permite a atribuição de custos às arestas de maneira bem simples e flexível. Usando o tipo Link do exemplo anterior poderíamos criar, por exemplo, uma atribuição de custos que levasse em conta a distância, da seguinte forma:

```
1  Transformer<Link, Number> transformer = new Transformer<Link, Number>() {
2    public Number transform(Link link) {
3      return link.getdistancia();
4    }
5  };
```

Usando esta implementação, o algoritmo de Dijkstra, por exemplo, poderia ordenar os vértices por ordem crescente de distância.

Supondo agora que a utilização do "link" seja o seu custo, procederíamos assim:

```
1  Transformer<Link, Number> transformer = new Transformer<Link, Number>() {
2    public Number transform(Link link) {
3      return link.getNumeroDeFibrasUsadas()/link.getNumeroDeFibras();
4    }
5  };
```

Por fim, caso queiramos que todas as arestas possuam o mesmo custo bastaria implementarmos:

```
1  Transformer<Link, Number> transformer = new Transformer<Link, Number>() {
2    public Number transform(Link link) {
3      return 1;
4    }
5  };
```

A leitura de grafos a partir de arquivos sofreu uma pequena mudança. Anteriormente, o leitor *PajekNet*, no processo de criação dos elementos do grafo, executava apenas operações **new**. Agora, o leitor espera que lhe sejam fornecidas fábricas de criação desses elementos. A idéia é usar o padrão de projetos *Factory*, cuja função é encapsular o processo de criação de instâncias de certos tipos. Um exemplo de uma fábrica de vértices do tipo Estacao seria:

```
1 Factory<Link> fabricaVertices = new Factory<Link>() {
2   private int id=0;
3   @Override
4   public Link create() {
5     id++;
6     return new Estacao(Integer.toString(id),0);
7   }
8 }
```

Para obter os rótulos dos vértices definidos no arquivo no formato PajkNet basta invocar o método SettableTransformer<V, String> getVertexLabeller() da classe PajekNetReader. Novamente vemos o JUNG 2.0 utilizando o padrão *Transformer*, desta vez recebendo um vértice e retornando seu rótulo.

Embora haja outras mudanças, consideramos as anteriormente citadas como as mais relevantes para nosso trabalho.

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 [14] que resolve o problema do caminho mínimo. As referências básicas para este capítulo são as notas de aula de Paulo Feofiloff [18] e a dissertação de Shigueo Isotani [31].

2.1 Descrição

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 $c(uv)$ 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 um 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 custo 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 quando a função custo estiver subentendida, simplesmente por $\text{dist}(s, t)$. $\text{dist}_c(s,t)$ $\text{dist}(s,t)$

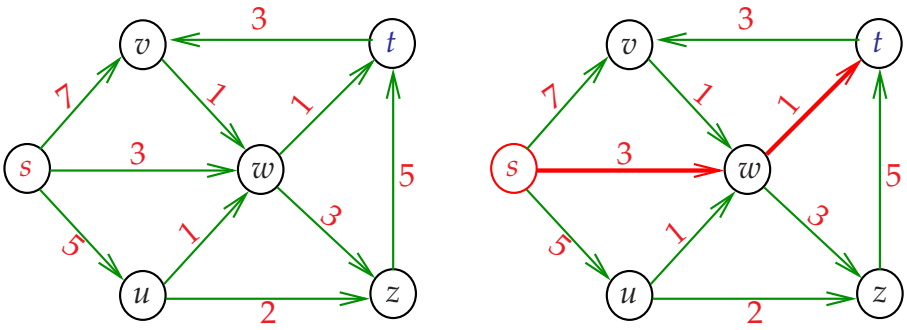


Figura 2.1: Um grafo com custos nos arcos. O custo do caminho $\langle s, u, w, z, t\rangle$ é 14. À direita o caminho de custo mínimo $\langle s, w, t\rangle$ está em destaque.

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 [14], 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

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

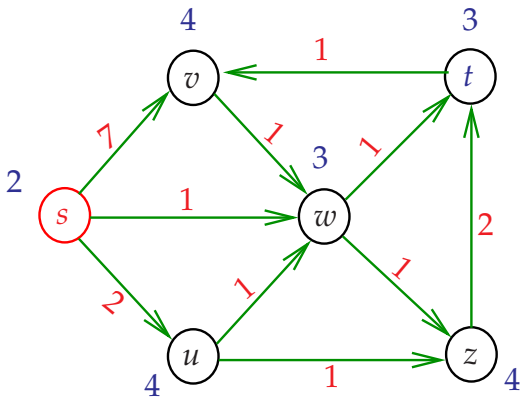


Figura 2.2: (a) Um grafo com custos nos arcos. e um c-potencial. Os números próximos aos vértices são os seus potenciais.

onde $\delta^+(u)$ representa o conjunto de arcos com ponta inicial no vértice u e $\delta^-(u)$ representa o conjunto de arcos com ponta final no vértice u. 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

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

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 combinatório 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 acessibilidade 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 [17].

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 vale que*

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

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

$$\begin{aligned} c(P) &= c(v_0v_1) + \ldots + c(v_{k-1}v_k) \\ &\geq (y(v_1) - y(v_0)) + (y(v_2) - y(v_1)) + \ldots + (y(v_k) - y(v_{k-1})) \\ &= y(v_k) - y(v_0) = y(t) - y(s). \end{aligned}$$

∎

Do lema 2.1 tem-se imediatamente os seguintes corolários.

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

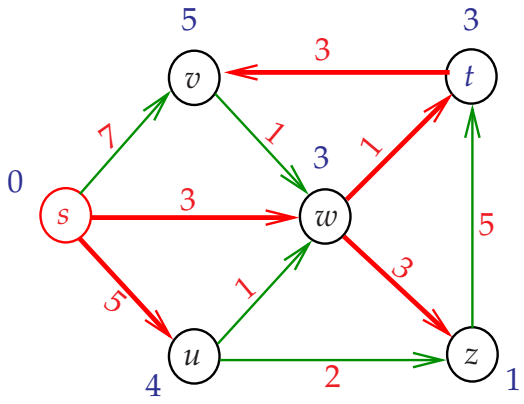


Figura 2.3: Um grafo com custos nos arcos e um potencial que certifica que qualquer um dos caminhos com arcos em destaque têm custo mínimo.

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

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

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

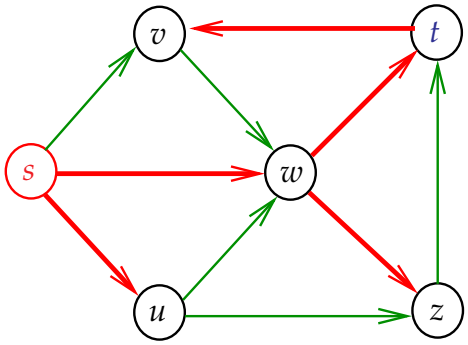
Seja ψ uma função-predecessor e $\Psi := \{uv \in A : u = \psi(v)\}$. Dizemos em (V, Ψ) é o **grafo de predecessores**.

Os algoritmos descritos neste texto utilizam funções predecessor para, compactamente, representarem todos os caminhos de custos mínimos a partir de um dado vértice. Conforme ilustrado na figura 2.4.

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* [13], *labeling step* [46]). Examinar um arco uv consiste em verificar se y respeita c em uv e, caso não respeite, ou



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 arborescência. A tabela ao lado mostra os valores de ψ.

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 tentativa, 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** $y(v) \leftarrow y(u) + c(uv)$
3 $\psi(v) \leftarrow u$

O consumo de tempo para examinar um arco é constante.

Outra operação básica é **examinar um vértice**. Se u é um vértice, examiná-lo 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 u.

2.5 Algoritmo de Dijkstra

Nesta seção é descrito o celebrado algoritmo de Edsger Wybe Dijkstra [14] 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, Q do conjunto de vértices. O algoritmo conhece caminhos de s a cada vértice em S e a uma parte dos vértices em Q. Para os vértices em S o caminho conhecido tem custo mínimo. Cada iteração consiste em remover um vértice apropriado de Q, incluí-lo em 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 se ψ determina um caminho de s a um vértice t, então este caminho tem custo mínimo. Por outro lado, a condição de inacessibilidade (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 [18].

DIJKSTRA (V, A, c, s) $\quad \triangleright\ c \geq 0$

1 **para cada** v em V **faça**
2 $\quad y(v) \leftarrow nC + 1 \quad \triangleright\ nC + 1$ *faz o papel de* ∞
3 $\quad \psi(v) \leftarrow$ NIL
4 $y(s) \leftarrow 0$
5 $Q \leftarrow N \quad \triangleright\ Q$ *func. como uma fila de prioridades*
6 **enquanto** $Q \neq \langle\rangle$ **faça**
7 $\quad$ *retire de* Q *um vértice* u *com* $y(u)$ *mínimo*
8 $\quad$ EXAMINE-VÉRTICE (u)
9 **devolva** ψ e y

A versão mais expandida é

DIJKSTRA (V, A, c, s) $\quad \triangleright\ c \geq 0$

1 **para cada** v em V **faça**
2 $\quad y(v) \leftarrow nC + 1 \quad \triangleright\ nC + 1$ *faz o papel de* ∞
3 $\quad \psi(v) \leftarrow$ NIL
4 $y(s) \leftarrow 0$
5 $Q \leftarrow N \quad \triangleright\ Q$ *func. como uma fila de prioridades*
6 **enquanto** $Q \neq \langle\rangle$ **faça**
7 $\quad$ *retire de* Q *um vértice* u *com* $y(u)$ *mínimo*
8 $\quad$ **para cada** uv em $A(u)$ **faça**
9 $\quad\quad$ **se** $y(v) > y(u) + c(uv)$ **então**
10 $\quad\quad\quad y(v) \leftarrow y(u) + c(uv)$
11 $\quad\quad\quad \psi(v) \leftarrow u$
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

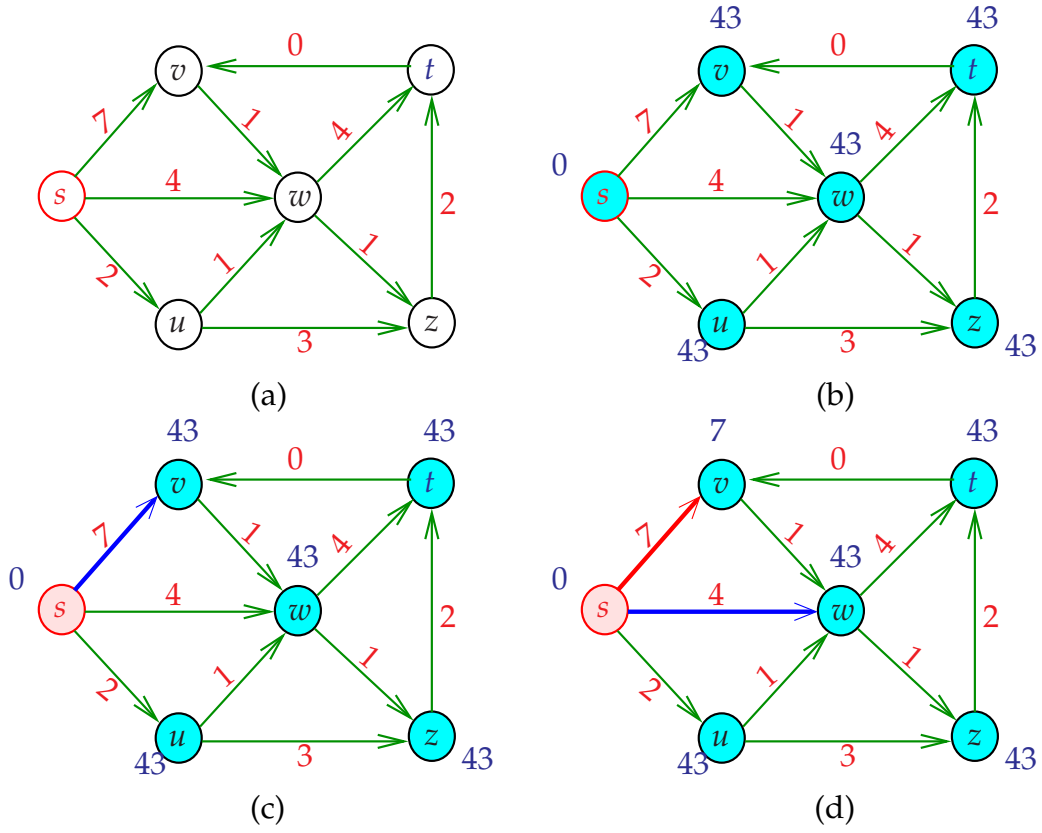
arco indica o seu custo. Por exemplo, $c(tv) = 0$ e $c(sw) = 4$. Nas ilustrações os vértices com interior azul claro são aqueles que estão em Q. Assim, na figura (b) vemos que todos os vértices foram inseridos em Q. A função-potencial y é indicada pelos números em azul próximos cada vértice. Assim, na figura (b), vemos que $y(s) = 0$ e o potencial dos demais vértices é $n \times C + 1 = 6 \times 7 + 1 = 43$.

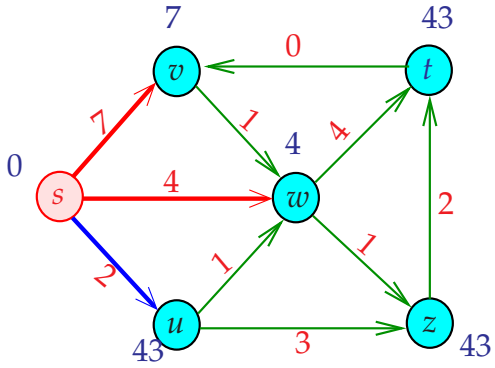
Durante a simulação, o vértice sendo examinado têm a cor rosa no seu interior, enquanto o arco sendo examinado é mais espesso e tem a cor azul. Por exemplo, na figura (c) o vértice sendo examinado é s e o arco sendo examinado é sv.

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.

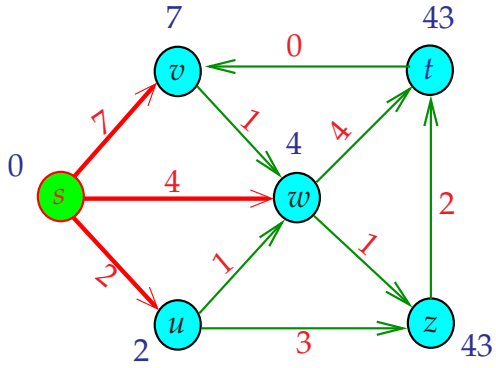
Os arcos já examinados são espessos e têm a cor vermelha ou são finos e têm a cor azul escura. Na figura (k) vemos que o arco wz está sendo examinado e os arcos já examinados são sv, sw, su, uw, uz e wt.

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.

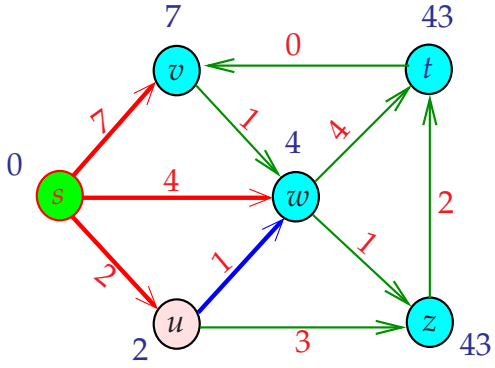



7
43
v
t
0
7
1
4
4
s
w
2
4
2
1
1
u
z
43
3
43

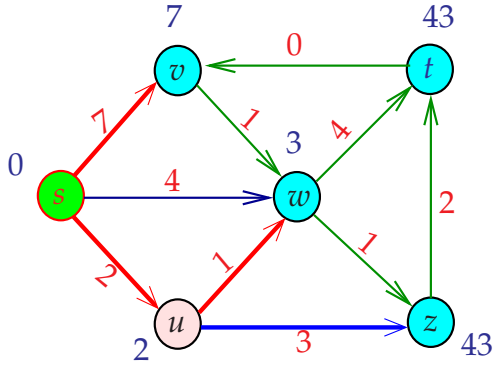
(e)


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

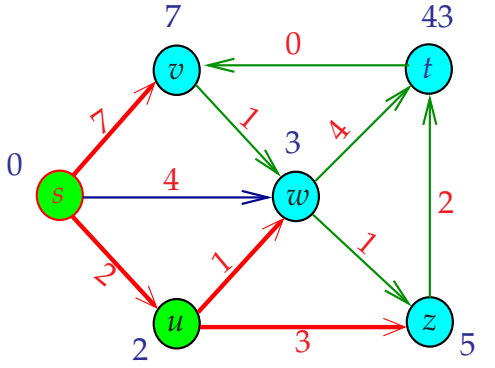
(f)


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

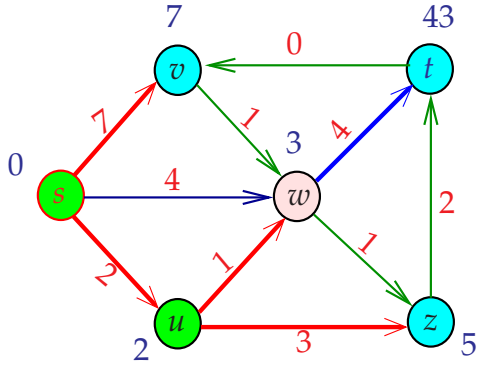
(g)


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

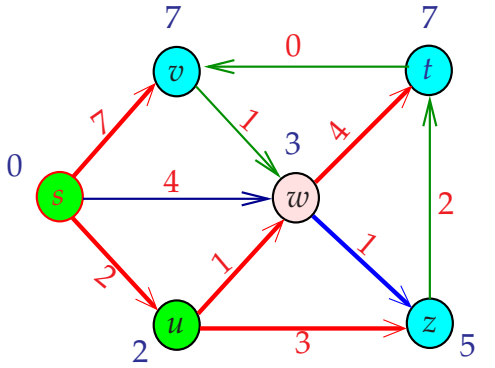
(h)


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

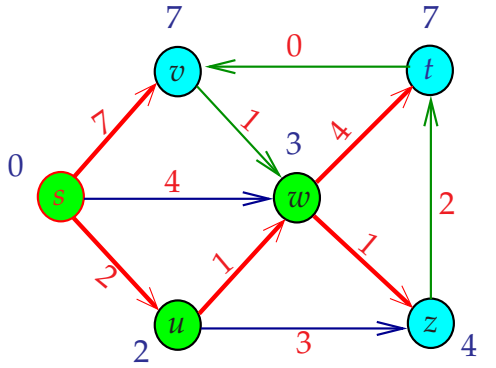
(i)


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

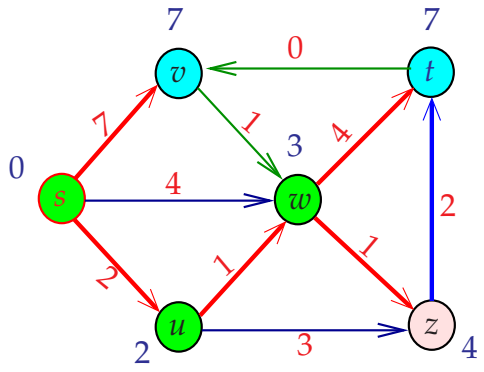
(j)


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

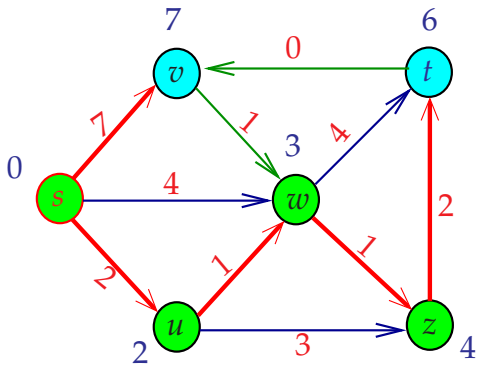
(k)


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

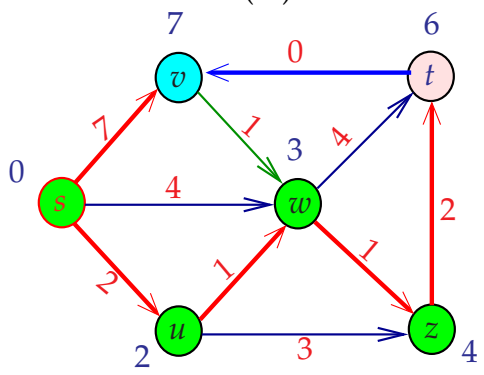
(l)



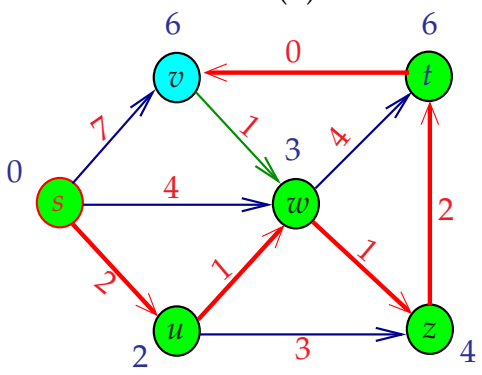
(m)



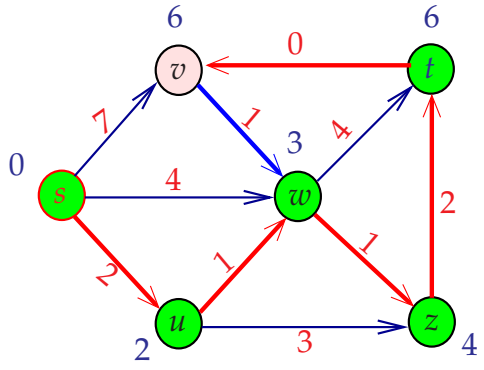
(n)



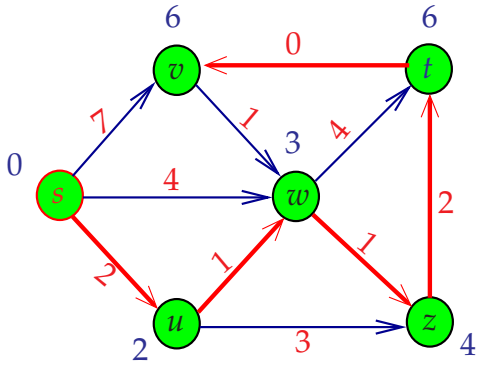
(o)



(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, $y(v) < nC + 1 \Leftrightarrow \psi(v) \neq$ 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 $V - Q$ e v em Q, vale que

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

Teorema 2.4 (da correção de DIJKSTRA): *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:

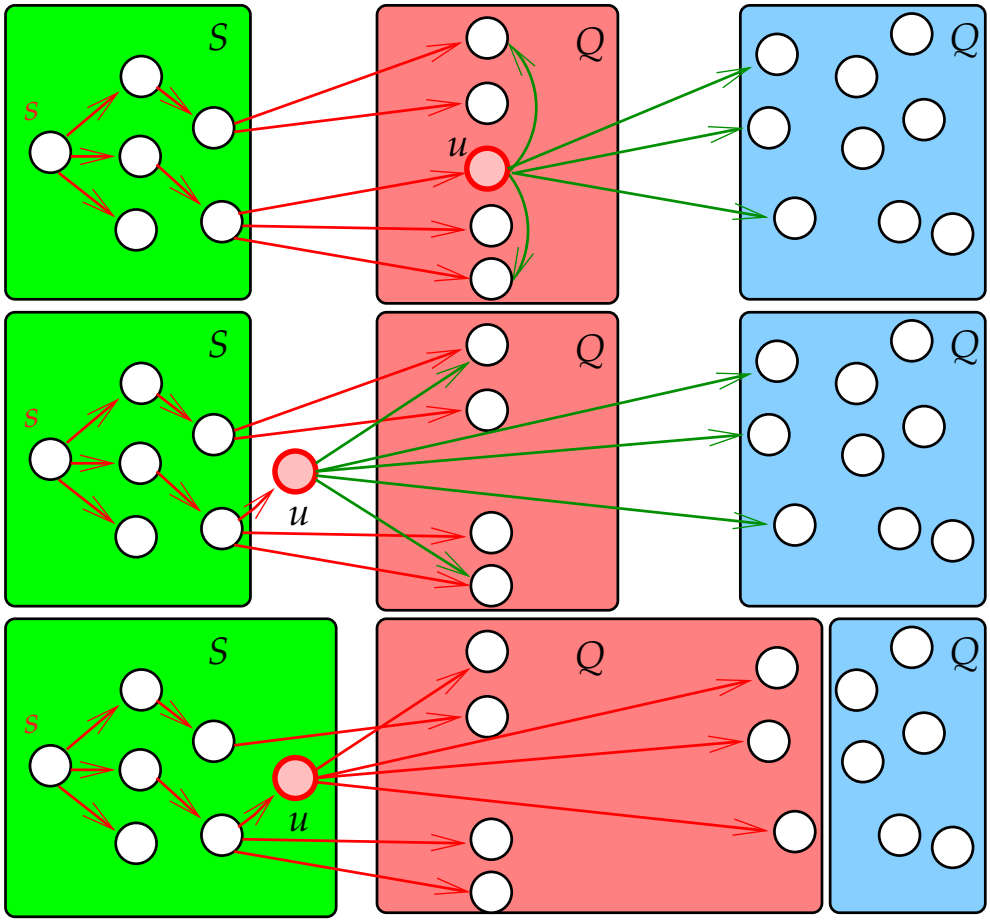


Figura 2.5: Ilustração de uma iteração do algoritmo DIJKSTRA. O arcos em vermelho formam o grafo de predecessores.

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 potencial 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 (do consumo de tempo de DIJKSTRA): *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 Donald B. Johnson [34], o Fibonacci heap de Michael L.

Fredman e Robert Endre Tarjan [21], que permitem diminuir o tempo gasto para encontrar um vértice com potencial mínimo, 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 fila de prioridades é implementada na forma de um **min-heap**, onde quanto menor o potencial maior a prioridade. A descrição do algoritmo de Dijkstra logo a seguir faz uso das operações EXTRACT-MIN e DECREASE-KEY, especificadas na secão 1.4, e BUILD-MIN-HEAP que recebe o conjunto V de vértices em que cada vértice v tem prioridade $y(v)$ e constrói uma fila de prioridades.

```
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	número de execuções da linha
1	$n+1$
2-3	n
4-5	1
6	$n+1$
7	n
8-10	m
11-12	m
13	1

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 de DIJKSTRA): *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, 9, 21].

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 [13].

	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	$O(m \log n)$	$O(m \log_D n)$	$O(m + n \log n)$	$O(m + nC)$	$O(m + n \log(nC))$

Lembramos que no Fibonacci heap o consumo de tempo é amortizado.

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.

Fredman e Tarjan [21] 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 Dijkstra, 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.

Interface Distance

Começaremos pela interface Distance cujo objetivo é definir métodos para classes que calculam distância entre dois vértices. Ela 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 numéricos: **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.

Interface DijkstraDistance

A interface Distance é implementada pela classe DijkstraDistance cujo objetivo é calcular distâncias entre os vértices usando o algoritmo de Dijkstra. Além disso, permite que resultados parciais, – 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 não-decrescente de distância.

Além dos métodos acima, há implementados métodos para melhorar o desempenho através do uso 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 alcança-

dos a partir de um vértice forem superiores a maxDist, serão considerados inacessíveis;

setMaxTargets(**int** maxDestinos): limita o número de vértices cujas distâncias mínimas devem 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 de maxDestinos.

Fila de prioridades

O algoritmo 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 (secão 2.6).

No JUNG, a estrutura utilizada foi um *heap* binário armazenado 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, a qual 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 próprio 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:

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

O primeiro parâmetro indica o vértice de origem, a partir 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 distâncias 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 fixa a origem. No entanto, vale ressaltar que alterações do grafo, como 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:

LinkedHashMap 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. O conjunto de todas as chaves corresponde ao conjunto S (seção 2.5) e os valores a $y(v), v \in S$.

Map estimatedDistances: semelhante ao distances, com a diferença de guardar a menor distância até o momento, ou seja, esta distância pode diminuir. O conjunto formado pelas chaves corresponde ao conjunto Q, cujas distâncias sejam diferentes de $nC + 1$, apresentado na secão 2.6 e o valores correspondem a $y(v), v \in Q$.

MapBinary unknownVertices: conjunto de vértices que ainda não foram analisados. Corresponde a todos os elementos do conjunto Q, cujas distâncias sejam diferentes de $nC + 1$.

O uso de uma outra classe no armazenamento desses dados [1] permite que as estruturas utilizadas sejam alteradas através da especialização da classe SourceData. Isso será de extrema importância quando estudarmos o algoritmo de geração 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 armazenamento 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.

Uma iteração da implementação do algoritmo de Dijkstra, feita no JUNG, pode ser resumida nos seguintes passos:

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

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

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

A descrição acima segue a apresentada na seção 2.5. Ao final, temos a estrutura distance devidamente preenchida, e podemos obter as distâncias, a partir da origem, a todos os vértices acessíveis.

Representação de caminhos

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,

[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.

criando a classe DijkstraShortestPath, cujas principais mudanças se referem a adição de quatro novos métodos:

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 armazenado na forma de um LinkedHashMap cuja iteração retorna os vértices em ordem não-decrescente dos custos. Este mapeamento corresponde ao grafo de predecessores apresentado na seção 2.3;

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. A função tratada corresponde à aplicação da função predecessor ao vértice target, ou seja $\psi($ target $)$, definida no grafo de predecessores retornado pela função Map getIncomingEdgeMap(Vertex source);

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. A função retorna o caminho definido pela função predecessor definida no grafo de predecessores obtido pela chamada à função Map getIncomingEdgeMap(Vertex source).

Para que esses métodos pudessem funcionar foi preciso 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 pertencentes 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 são 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

k-menores caminhos e Yen

Estão descritos neste capítulo os elementos básicos que envolvem o problema dos k-menores caminhos junto com um método e algoritmo central para o problema. Começamos relembrando os ingredientes do problema do caminho mínimo que serão empregados neste capítulo. Em seguida, na seção 3.2, apresentamos o problema dos k-menores caminhos. Antes de abordamos os métodos para resolução do problema dos k-menores caminhos, mostramos uma maneira compacta para representarmos uma coleção de caminhos com um início comum: a "árvore dos prefixos", muito útil na descrição de dois métodos genéricos, feitas na seção 3.4, para o problema do k-menores caminhos. A seção 3.5 trata de uma partição dos caminhos candidatos a estarem na solução do problema dos k-menores caminhos. Esta partição, junto com a árvore dos prefixos, são fundamentais nos algoritmos que serão apresentados neste e nos próximos capítulos. Na seção 3.6, vemos o algoritmo de Jin Y. Yen [48, 49]. A relevância do algoritmo de Dijkstra para o problema dos caminhos mínimos é semelhante à do algoritmo de Yen para o problema dos k-menores caminhos: até o momento, o algoritmo de Yen é o assintoticamente mais eficiente para o problema e todas as melhorias encontradas o têm como plano de fundo.

Finalmente, terminamos este capítulo com algumas considerações históricas sobre os algoritmos conhecidos para o problema.

3.1 Caminhos mínimos

Lembremos que 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 $c(uv)$ será o valor de c em uv. As

vezes diremos que (V, A, c) é um **grafo com custo**. 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**.

No restante deste texto, utilizamos extensivamente como subrotina um algoritmo para o **problema do caminho mínimo**, denotada 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 [14], descrito no capítulo 2, resolve o problema do caminho mínimo.

Denotamos, por n e m os números $|V|$ e $|A|$, respectivamente. Além disso, representamos 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 é uma implementação do algoritmo de Dijkstra projetada por Michael L. Fredman e Robert Endre Tarjan [21] que consome tempo $O(m + n \log n)$ (seção 2.6). Existe ainda um algoritmo que consome tempo linear *sob um outro modelo de computação* que foi desenvolvido por Mikkel Thorup [47] e que se encontra esmiuçado na dissertação de mestrado de Shigueo Isotani [31].

3.2 k-menores caminhos

O problema central deste texto se assemelha muito ao do k-ésimo menor elemento, que é estudado em disciplinas básicas de análise de algoritmos [13]:

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 $\mathcal{S}$. Além disto, consumindo tempo extra $\Theta(k \log k)$ podemos ter esses k menores elementos em ordem crescente. Sendo assim, podemos obter os k menores elementos de $\mathcal{S}$ em ordem crescente consumindo tempo $\Theta(|\mathcal{S}| + k \log k)$.

A diferença entre o problema k-ÉSIMO e o problema central deste texto é que o conjunto $\mathcal{S}$ dado é "muito grande" e nos é dado de uma maneira compacta, o que torna o problema sensivelmente mais difícil do ponto de vista computacional. A seguir tornamos essa digressão mais precisa.

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 os st-caminhos, ou seja, todos os caminhos com início em s e término em t. Uma lista $\langle P_1, \ldots, P_k \rangle$ de st-caminhos distintos é de **custo mínimo** se

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

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 apresentado na introdução deste trabalho é o **problema dos k-menores caminhos**, denotado por k-MC:

> **Problema** k-MC(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 k-menores caminhos de s a t. k-MC

É evidente que o CM nada mais é que o k-MC com $k = 1$. O k-MC é uma generalização natural do problema do caminho mínimo.

O k-MC 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-MC utilizando meramente algoritmos inspirados no k-ÉSIMO como subrotinas.

Na próxima seção é descrito o método genérico para resolver o k-MC. Este método é um passo intermediário para chegarmos ao método desenvolvido por Jin Y. Yen [48] para o k-MC.

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, neste capítulo, discutirmos o algoritmo 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 e existir um caminho da raiz a cada um dos demais vértices. Será conveniente tratarmos os vértices de uma arborescência por **nós**. Uma arborescência está ilustrada na figura 3.1. A raiz dessa arborescência é o nó r. Uma **folha** de uma arborescência é um nó que não é ponta inicial de nenhum arco.

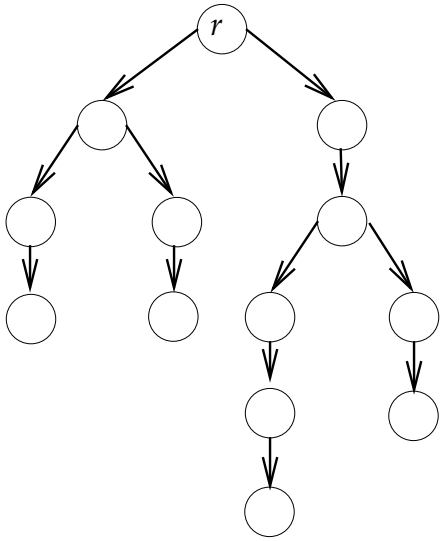


Figura 3.1: Ilustração de uma arborescência que tem com raiz o nó r.

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

$$R := \langle u_0, u_1, \ldots, u_q \rangle$$

for um caminho em (N, E), então

$$f(R) := \langle f(u_0), f(u_1), \ldots, f(u_q) \rangle$$

será uma seqüência de vértices 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 $\mathcal{Q}$;

(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 $\mathcal{Q}$. Basta existirem em $\mathcal{Q}$ caminhos com pontas iniciais distintas para que não seja possível definir a função rótulo para a raiz da arborescência de forma a satisfazer (p1). No entanto, se todos os caminhos em $\mathcal{Q}$ tiverem a mesma ponta inicial, então existe uma árvore dos prefixos de $\mathcal{Q}$ e esta é única. Na figura 3.2(b) vemos a ilustração da árvore dos prefixos de quatro caminhos de s a t no grafo da figura 3.2(a). Na árvore da ilustração, w, x, y e z são nós e $f(w) = s, f(x) = a, f(y) = d$ e $f(z) = d$.

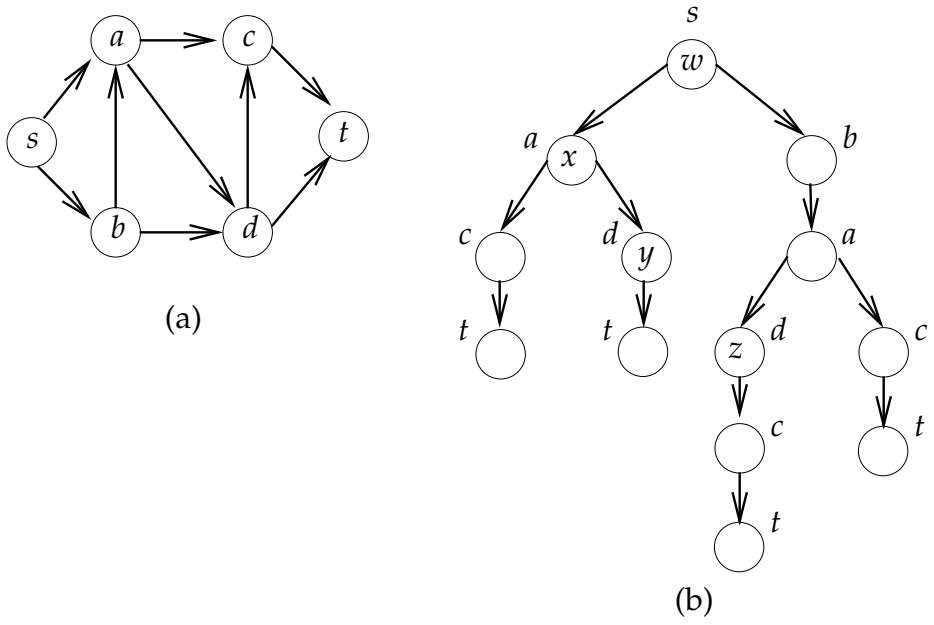


Figura 3.2: (b) mostra a árvore dos prefixos dos caminhos $\langle s, a, c, t\rangle$, $\langle s, a, d, t\rangle$, $\langle s, b, a, c, t\rangle$ e $\langle s, b, a, d, c, t\rangle$ no grafo em (a). Na árvore, um símbolo ao lado de um nó é o rótulo desse nó. O símbolo dentro de um nó é o seu nome.

Teorema 3.1 (da árvore dos prefixos): *Se $\mathcal{Q}$ é uma coleção de caminhos em um grafo, todos com ponta inicial em um vértice s, então existe uma árvore dos prefixos de $\mathcal{Q}$. Ademais, a árvore dos prefixos de $\mathcal{Q}$ é única (a menos de isomorfismo).*

Demonstração: A demonstração é por indução no número de caminhos em $\mathcal{Q}$.

Se $\mathcal{Q}$ tem apenas um caminho, então tomamos

$$\begin{aligned} N &:= V(\mathcal{Q}), \\ E &:= A(\mathcal{Q}), \text{ e} \\ f(v) &:= v, \text{ para todo } v \text{ em } N. \end{aligned}$$

É evidente que (N, E, f) satisfaz (p1), (p2) e (p3) e portanto é uma árvore de prefixos de $\mathcal{Q}$.

Suponha agora que $\mathcal{Q}$ tem mais do que um caminho e seja P um caminho qualquer em $\mathcal{Q}$. Defina $\mathcal{Q}' := \mathcal{Q} - \{P\}$. Por indução, existe (N', E', f') árvore dos prefixos de $\mathcal{Q}'$. Seja Q_P o maior prefixo de P para o qual existe um caminho R_P em (N', E') com início na raiz e tal que $f(R_P) = Q_P$. Temos que Q_P tem pelo menos um vértice pois todos os caminhos em $\mathcal{Q}'$ tem início em s e portanto, por (p1), o rótulo da raiz de (N', E') é s.

Suponha

$$\begin{aligned} P &= \langle s = v_0, v_1, \ldots, v_k, v_{k+1}, \ldots, v_q \rangle, \\ Q_P &= \langle s = v_0, v_1, \ldots, v_k \rangle, \text{ e} \\ R_P &= \langle u_0, u_1, \ldots, u_k \rangle . \end{aligned}$$

Como (N', E', f') é arvore de prefixos de $\mathcal{Q}'$, então $f'(u_i) = v_i$ para $i = 0, \ldots, k$.

Para $i = k+1, \ldots, q$, seja u_i um elemento que não está em N' e defina

$$\begin{aligned} N &:= N' \cup \{u_{k+1}, \ldots, u_q\}, \\ E &:= E' \cup \{u_i u_{i+1} : i = k, \ldots, q-1\}, \text{ e} \\ f(u) &:= \begin{cases} f'(u), & \text{se } u \in N', \\ v_i, & \text{se } u = u_i \text{ para algum } i \text{ em } \{k+1, \ldots, q\}. \end{cases} \end{aligned}$$

Esta construção está ilustrada na figura 3.3.

É claro que (N, E) é uma arborescência. Passamos a mostrar que (N, E, f) é árvore de prefixos de $\mathcal{Q}$.

Seja R um caminho em (N, E) com início na raiz. Se R é um caminho em (N', E'), então $f(R) = f'(R)$ é prefixo de algum caminho em $\mathcal{Q}'$ que é um subconjunto de $\mathcal{Q}$. Se R utiliza algum nó em N que não é nó de N', então $f(R)$ é prefixo de P que está em $\mathcal{Q}$. Logo, (N, E, f) satisfaz (p1).

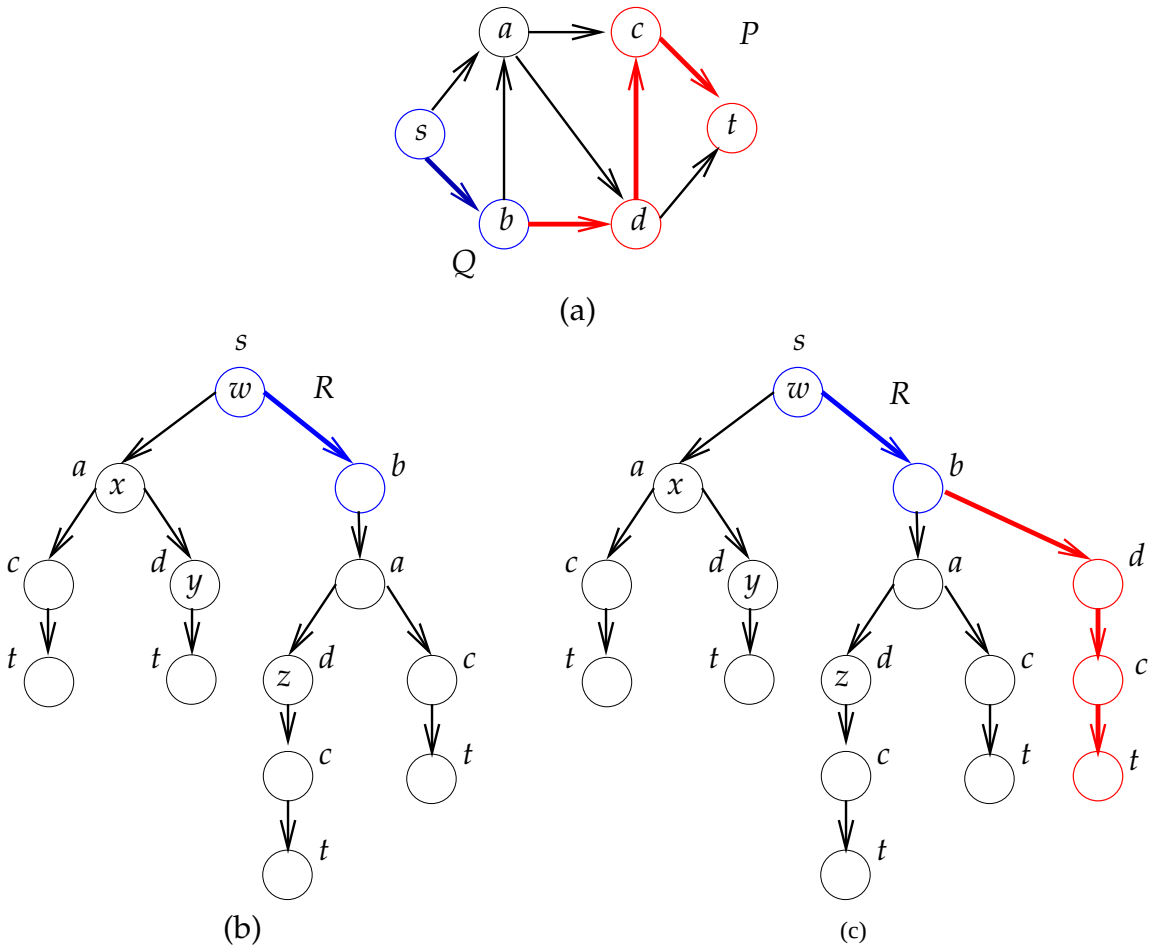


Figura 3.3: Ilustração da construção da árvore dos prefixos de $\mathcal{Q}$ a partir da árvore dos prefixos de $\mathcal{Q}'$ feita no teorema 3.1. A figura (a) mostra o caminho $P = \langle s, b, d, c, t\rangle$ com vértices e arcos de cor azul e vermelha. O prefixo $Q = \langle s, b\rangle$ tem vértices e arco azul. Na figura (b) vemos a árvore dos prefixos de $\mathcal{Q} = \{\langle s, a, c, t\rangle, \langle s, a, d, t\rangle, \langle s, b, a, c, t\rangle, \langle s, b, d, c, t\rangle\}$. Na figura (c) está a árvore dos prefixos de $\mathcal{Q}' = \mathcal{Q} - \{P\}$. Nas árvores dos prefixos os nós e arcos azuis indicam o caminho R e um símbolo ao lado de um nó é o rótulo desse nó.

Para cada prefixo Q de um caminho em $\mathcal{Q}'$ existe uma caminho R em (N', E') com início na raiz tal que $f'(R) = Q$. Como R é um caminho em (N, E) com início na raiz e $f(R) = f(R')$, concluímos que para todo caminho Q em $\mathcal{Q}'$ existe um caminho R em (N, E) tal que $f(R) = Q$. Como o único caminho em $\mathcal{Q} - \mathcal{Q}'$ é P e $R = \langle u_0, \ldots, u_q \rangle$ é um caminho em (N, E) com início na raiz tal que $f(R) = P$, concluímos que (N, E, f) satisfaz (p2).

Se Q é prefixo de um caminho em $\mathcal{Q}'$, então a unicidade do caminho R em (N, E) com início na raiz tal que $f(R') = Q'$ segue do fato de (N', E', f') ser árvore dos prefixos de $\mathcal{Q}'$. Se Q é um prefixo de algum caminho em $\mathcal{Q}$ que não é prefixo de um caminho em $\mathcal{Q}'$ então $Q = \langle v_0, \ldots, v_k, v_{k+1}, \ldots, v_r \rangle$ com $k + 1 \leq r \leq q$ e portanto, da maximalidade de Q_P, segue que $R = \langle u_0, \ldots, u_k, u_{k+1}, \ldots, u_r \rangle$ é o único caminho em (N, E) com origem na raiz e tal que $f(R) = Q$. Com isto acabamos de mostrar que (N, E, f) também satisfaz (p3) e portanto é árvore dos prefixos de $\mathcal{Q}$.

Verifiquemos agora que se (N_1, E_1, f_1) e (N_2, E_2, f_2) são árvores dos prefixos de uma coleção de caminhos $\mathcal{Q}$ com início em um vértice s, então essas árvores são essencialmente a mesma, isto é, são isomorfas.

Definimos uma função h de N_1 em N_2 que é um isomorfismo entre (N_1, E_1, f_1) e (N_2, E_2, f_2). A cada nó u_1 em N_1 associaremos um nó $h(u_1)$ em N_2 da seguinte maneira. Como (N_1, E_1) é uma arborescência temos que existe um único caminho R_{u_1} que tem como início a raiz da arborescência e como término u_1. Devido a (p1) sabemos que $Q := f_1(R_{u_1})$ é prefixo de um caminho em $\mathcal{Q}$. Já, de (p2) e (p3) temos que R_{u_1} é o único caminho com início na raiz tal que $f_1(R_{u_1}) = Q$. Novamente, devido a (p2) e (p3), temos que existe na arborescência (N_2, E_2) um único caminho R_{u_2} com início na raiz tal que $f_2(R_{u_2}) = Q$. Seja u_{u_2} a ponta final de R_{u_2}. Definimos $h(u_{u_1}) := u_2$.

Da unicidade dos caminhos R_{u_1} e R_{u_2} acima segue que h é uma bijeção entre N_1 e N_2 e que se $u_1 w_1$ é um arco em E_1 então $h(u_1)h(w_1)$ é um arco em E_2. Além disso, para todo u_1 em N_1 é evidente que $f_1(u_1) = f_2(h(u_1))$, pois ambos são os términos de um único prefixo Q de um caminho em $\mathcal{Q}$. Segue do que foi exposto que h é um isomorfismo entre (N_1, E_1, f_1) e (N_2, E_2, f_2).

∎

3.4 Métodos genéricos

A descrição que fazemos aqui de um método para o k-MC é, de certa forma, *top-down*. Começaremos com um método genérico que será refinado a cada passo incluindo, convenientemente, algumas subrotinas auxiliares. A nossa intenção aqui é apresentar uma descrição mais conceitual em que a correção e o consumo de tempo do método sejam um tanto quanto evidentes, apesar do consumo de tempo ser exponencial. Deixaremos para a próxima seção a descrição de um algoritmo que é mais um refinamento dos métodos desta seção e atinge o menor consumo de tempo conhecido, o algoritmo de Yen. Assim, esta seção pretende ser uma ponte entre idéias genéricas e muito simples e o algoritmo da próxima seção.

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

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

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

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

4 $\quad$ ***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 algoritmo de Yen é uma elaboração do método GENÉRICO. Em vez do conjunto $\mathcal{P}$, YEN-GENÉRICO, descrito logo adiante, mantém uma partição Π de $\mathcal{P}$. Na linha 6 do método a partição Π é atualizada. Em cada iteração i, é escolhido o caminho mais barato P_i dentre os caminhos em um conjunto $\mathcal{L}$ formado por *um* caminho mínimo P_π representante de cada parte π de Π. Na linha 6 do método a partição Π é atualizada. O método mantém ainda um conjunto $\mathcal{Q}$ com cada caminho em $\langle P_1, \ldots, P_i \rangle$.

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

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

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

2 ***para*** $i = 1, \ldots, k$ ***faça***
3 $\quad \mathcal{L} \leftarrow \{P_\pi : P_\pi$ *é caminho mínimo na parte* π *de* $\Pi\}$
4 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$
5 $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{P_i\}$
6 $\quad \Pi \leftarrow$ ATUALIZE-GENÉRICO $(V, A, s, t, \mathcal{Q})$
7 ***devolva*** $\langle P_1, \ldots, P_k \rangle$

Na próxima seção tratamos de Π de uma maneira mais precisa e só depois, ao final da seção, consideramos os detalhes da subrotina ATUALIZE-GENÉRICO. Como veremos, a eficiência do algoritmo de Yen, descrito na seção 3.6, depende fortemente da estrutura restrita dos caminhos nas partes de Π: cada parte é formada por caminhos que têm um certo prefixo comum.

Neste ponto, como no início de cada execução do bloco de linhas 2–6 do método YEN-GENÉRICO temos que

Π é uma partição de $\mathcal{P} = \mathcal{P}_{st} - \mathcal{Q}$,

então a correção do método é evidente.

Teorema 3.2 (da correção de YEN-GENÉRICO): *Dado um grafo* (V, A)*, uma função custo* c *e vértices* s *e* t *o método* YEN-GENÉRICO *corretamente encontra os* k*-menores caminhos de* s *a* t*.* ∎

3.5 Partição de caminhos

Seja $\mathcal{P}_{st}$ a coleção dos caminhos de s a t em (V, A). Suponha que $\mathcal{Q}$ seja uma coleção de caminhos distintos de s a t, como, por exemplo, a atualizada na linha 5 do método YEN-GENÉRICO. Passamos a descrever a partição Π dos caminhos em $\mathcal{P} := \mathcal{P}_{st} - \mathcal{Q}$. Para isto é conveniente utilizarmos a *árvore dos prefixos* de $\mathcal{Q}$, como foi feito por John Hershberger, Matthew Maxel e Subhash Suri [26].

No que segue suponha que (N, E, f) é a árvore dos prefixos de $\mathcal{Q}$ e u é um nó qualquer 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 3.2(b) temos que $R_y = \langle w, x, y \rangle$ e $f(R_y) = \langle s, a, d \rangle$.

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 3.2 temos que: π_u

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

Notemos que para cada nó u que é folha de (N, E, f) temos que π_u é formado pelo caminho $f(R_u)$ de $\mathcal{Q}$. Portanto, o número de folhas de (N, E, f) é exatamente $|\mathcal{Q}|$.

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

$$\Pi := \{\pi_u : u \in N, u \text{ não é folha}\}.$$

Notemos que para cada coleção $\mathcal{Q}$ de caminhos temos uma única árvore dos prefixos (N, E, f) de $\mathcal{Q}$ e associada a essa árvore temos uma única partição Π de $\mathcal{P}$. Assim, algumas vezes nos referiremos a Π como sendo a **partição associada** à coleção $\mathcal{Q}$.

Podemos verificar que cada caminho em $\mathcal{Q}$ não pertence a nenhuma parte de Π.

Teorema 3.3 (da partição): *Sejam (V, A) um grafo, s e t dois de seus vértices e $\mathcal{Q}$ uma coleção de caminhos de s a t em (V, A). Se P é um caminho de s a t que não está em $\mathcal{Q}$, então P pertence a uma única parte de Π.*

Demonstração: Considere um caminho P que não está em $\mathcal{Q}$. Suponha que Q é o maior prefixo de P que também é prefixo de algum caminho em $\mathcal{Q}$. Como P não está em $\mathcal{Q}$ então Q é um prefixo próprio de P. Como (N, E, f) é árvore dos prefixos de $\mathcal{Q}$, então, por (p2) e (p3), existe um único caminho R com início na raiz e término em um nó interno da árvore tal que $f(R) = Q$ é prefixo de P. Se u é o nó interno da árvore que é a ponta final de R, então, pela maximalidade de Q, temos que P não possui arcos em A_u. Portanto, P está na parte π_u e portanto esta é a única parte de Π que contém P. ∎

Notemos que, cada parte π_u em que u é folha, é formada pelo único caminho P em $\mathcal{Q}$ tal que $f(R_u) = P$. Assim, do teorema 3.3 da partição segue o colorário abaixo.

Corolário 3.4: *Seja $\mathcal{Q}$ uma coleção de caminhos em um grafo (V, A) tal que todos os caminhos em $\mathcal{Q}$ têm ponta inicial em um vértice s e ponta final em um*

vértice t. Se (N, E, f) é a árvore dos prefixos de $\mathcal{Q}$, então

$$\mathcal{P}_{st} = \bigcup_{u \in N} \pi_u \,,$$

onde $\mathcal{P}_{st}$ é a coleção de todos os caminhos de s a t em (V, A). ∎

Agora temos em mãos todos os elementos necessários para voltarmos a discutir o método YEN-GENÉRICO e completá-lo. No início de cada iteração da linha 2 do método YEN-GENÉRICO, o número de partes em Π é igual ao número de nós, que não são folhas, na árvore dos prefixos de $\mathcal{Q}$ e, portanto, é certamente não superior a $i \times n$. Logo, no início de cada iteração, o número de caminhos em $\mathcal{L}$ é não superior a $i \times n$, já que a árvore dos prefixos de $\mathcal{Q}$ não possui mais do que n nós para cada caminho em $\mathcal{Q}$.

O algoritmo ATUALIZE-GENÉRICO resume toda a discussão acima. Este algorimo recebe um grafo (V, A), dois vértices s e t do grafo e uma coleção $\mathcal{Q}$ de caminhos de s a t e devolve uma partição Π dos caminhos em $\mathcal{P} = \mathcal{P}_{st} - \mathcal{Q}$.

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

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

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

2 $\quad$ ***para cada*** $u \in N$ *que não é uma folha* ***faça***

3 $\quad\quad \pi_u \leftarrow \{$*caminhos em* $\mathcal{P}$ *com prefixo* $f(R_u)$
e que não possuem arcos em $A_u\}$

4 $\quad\quad \Pi \leftarrow \Pi \cup \{\pi_u\}$

5 $\quad$ ***devolva*** Π

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 que um refinamento do algoritmo ATUALIZE-GENÉRICO obtenha, mais eficientemente, uma nova partição a partir da anterior. Veremos este e outro refinamento na próxima seção.

3.6 Algoritmo de Yen

O algoritmo que Jin Y. Yen [48] desenvolveu para resolver o k-MC parece ter um papel central dentre os algoritmos que foram posteriormente projetados para o k-MC ou mesmo para versões mais restritas do problema [16, 36, 26]. Várias melhorias práticas do método de Yen foram implementadas e testadas [7, 24, 39, 40, 42].

Antes de prosseguirmos, mencionamos que o algoritmo de Yen foi generalizado por Eugene L. Lawler [37] para problemas de otimização combinatória, contanto que seja fornecida uma subrotina para determinar uma solução ótima sujeita à condição de que certas variáveis tenham seus valores fixados. Por exemplo, no caso do algoritmo de Yen para o k-MC essa subrotina resolve o seguinte **problema do subcaminho mínimo**, denotado por SCM:

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

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

Resolver o CM(V, A, c, s, t) é o mesmo que resolver o 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 subgrafo (V', A') apropriado de (V, A):

$$V' := V - (V(Q) - \{s'\})$$
$$A' := A - F - A^+(Q - s') - A^-(Q)\,,$$

onde $V(Q)$ é o conjunto dos vértices em Q, $A^+(Q)$ são os arcos com ponta inicial em $V(Q)$, $A^-(Q)$ são os arcos com ponta final em $V(Q)$ e s' é a ponta final de Q. Desta forma, o CM e o SCM são computacionalmente equivalentes e ambos podem ser resolvidos consumindo-se tempo $T(n, m)$.

Voltemos ao algoritmo de Yen. Conceitualmente, o algoritmo de Yen é uma elaboração do algoritmo YEN-GENÉRICO. Como no método YEN-GENÉRICO, no início de cada iteração $\mathcal{L}$ é uma lista dos candidatos a i-ésimo caminho mínimo de s a t. Esta lista é formada por um caminho mínimo em cada parte da partição Π. O algoritmo de Yen mantém o conjunto $\mathcal{Q}$ dos caminhos previamente selecionados através de sua

árvore dos prefixos (N, E, f). Diferentemente do método YEN-GENÉRICO, o algoritmo de Yen armazena a partição Π implicitamente através da árvore dos prefixos (N, E, f) e de $\mathcal{L}$:

> para cada nó u em (N, E, f) que não é folha a lista $\mathcal{L}$ contém um caminho de custo mínimo na parte π_u de Π. (3.1)

Pelo teorema 3.3 da partição temos que $\mathcal{L}$ contém um caminho de custo mínimo de cada parte da partição Π de $\mathcal{P}_{st} - \mathcal{Q}$. A tarefa de cada iteração é escolher o caminho P mais barato dentre todos em $\mathcal{L}$ e, em seguida, atualizar $\mathcal{L}$ e a árvore dos prefixos (N, E, f) dos caminhos selecionados. Essa atualização é feito na linha 5 pelo algoritmo ATUALIZE-YEN.

O algoritmo ATUALIZE-YEN recebe um grafo (N, A), uma função custo c, vértices s e t, um caminho P de s a t, a árvore dos prefixos (N', E', f') de uma coleção $\mathcal{Q}'$ de st-caminhos e uma lista $\mathcal{L}'$ de st-caminhos tais que:

> (N', E', f') e $\mathcal{L}'$ satisfazem (3.1) nos papéis de (N, E, f) e $\mathcal{L}$

e devolve:

> uma árvore dos prefixos (N, E, f) da coleção $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$ de st-caminhos e uma lista $\mathcal{L}$ de st-caminhos que satisfazem (3.1).

Algoritmo YEN (V, A, c, s, t, k)

```
0   P ← um caminho de custo mínimo de s a t
1   L ← {P}
2   (N, E, f) ← árvore dos prefixos de L
3   para i = 1, ..., k faça
4       P_i ← caminho de custo mínimo em L
5       (N, E, f, L) ← ATUALIZE-YEN (V, A, c, s, t, P_i, N, E, f, L)
6   devolva ⟨P_1, ..., P_k⟩
```

Tendo em vista a especificação do algoritmo ATUALIZE-YEN a correção do algoritmo YEN é evidente.

Teorema 3.5 (da correção de YEN): *Dado um grafo (V, A), uma função custo c e vértices s e t o algoritmo* YEN *corretamente encontra os k-menores caminhos de s a t.* ∎

O algoritmo ATUALIZE-YEN que está logo a seguir utiliza como subrotina o algoritmo ÁRVORE-DOS-PREFIXOS que recebe uma árvore dos prefixos de uma coleção de caminhos $\mathcal{Q}'$ com ponta inicial em um vértice s e um caminho P com ponta inicial em s e devolve a árvore dos prefixos de $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$. O serviço feito por ÁRVORE-DOS-PREFIXOS é essencialmente descrito na demonstração do teorema 3.1 e está ilustrado na figura 3.2.

Algoritmo ATUALIZE-YEN $(V, A, c, s, t, P, N', E', f', \mathcal{L}')$

0 $\quad \mathcal{L} \leftarrow \mathcal{L}' - \{P\}$

1 $\quad R_P \leftarrow$ *maior caminho em* (N', E') *com ponta inicial na raiz tal que*
$$Q_P := f(R_P) \text{ é prefixo de } P$$

2 $\quad (N, E, f) \leftarrow$ ÁRVORE-DOS-PREFIXOS(N', E', f', P)

3 $\quad$ *Suponha que* $\langle u_0, \ldots, u_k, u_{k+1}, \ldots, u_q \rangle$ *é o caminho em* (N, E) *tal que*
$$R_P = \langle u_0, \ldots, u_k \rangle \text{ e } f(\langle u_0, \ldots, u_k, u_{k+1}, \ldots, u_q \rangle) = P.$$

4 $\quad$ ***para cada*** $u \in \{u_k, \ldots, u_{q-1}\}$ ***faça***

5 $\quad\quad P_u \leftarrow$ *caminho de s a t de custo mínimo com prefixo $f(R_u)$ e que não possui arcos em A_u*

6 $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{P_u\}$

7 $\quad$ ***devolva*** $(N, E, f, \mathcal{L})$

Primeiramente, como o algoritmo ATUALIZE-YEN recebe a árvore dos prefixos (N', E', f') de uma coleção de caminhos $\mathcal{Q}'$ com origem em um vértice s e um caminho P com origem em s então, devido à especificação do algoritmo ÁRVORE-DOS-PREFIXOS invocado na linha 2, é claro que ATUALIZE-YEN corretamente devolve a árvore dos prefixos (N, E, f) de $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$.

Pela linha 1 do algoritmo ATUALIZE-YEN temos que Q_P é o maior prefixo de P para o qual existe um caminho R_P na árvore dos prefixos (N', E', f') de $\mathcal{Q}'$ tal que $f'(R_p) = Q_P$. Devido à definição do caminho $\langle u_0, \ldots, u_k, u_{k+1}, \ldots, u_q \rangle$ feita na linha 3 e à definição de R_P temos que o caminho P é o representante da parte π'_{u_k} da partição

Π' associada à árvore dos prefixos (N', E', f'). Seja Π a partição associada à árvore dos prefixos (N, E, f) construída na linha 2. Antes da primeira execução do bloco de linhas 4–6 temos que $\mathcal{L}$ contém um caminho de custo mínimo de cada parte $\pi'_u = \pi_u$, onde u é um nó em $N - \{u_k, \ldots, u_{q-1}\}$. Nas linhas 4-5, são acrescidos a $\mathcal{L}'$ os caminhos de custo mínimo em π_u para u em $\{u_k, u_{k+1}, \ldots, u_{q-1}\}$, que são os nós em $N - N'$ que não são folhas, já que a única folha de (N, E, f) em $N - N'$ é u_q. Assim, concluímos que $\mathcal{L}$ devolvida contém, para cada nó u da árvore dos prefixos (N, E, f) de $\mathcal{Q}$ que não é folha, um caminho de custo mínimo na parte π_u. Com isto concluímos que o algoritmo ATUALIZE-YEN faz o que promete.

Em cada execução do bloco de linhas 3–5 do algoritmo YEN o número de caminhos em $\mathcal{L}$ é não superior a $k\,n$. Assim, o consumo de tempo de todas as execuções da linha 4 é $O(k \lg kn)$, se utilizamos um min-heap para armazenarmos os custos dos caminhos em $\mathcal{L}$. O consumo de tempo do algoritmo YEN é dominado pelo consumo de tempo de todas as execução da linha 5, ou seja, pelo consumo de tempo de das k execuções do algoritmo ATUALIZE-YEN.

A cada execução da linha 5 do algoritmo ATUALIZE-YEN, na verdade, estamos resolvendo o problema $\mathrm{SCM}(V, A, c, s, t, f(R_u), A_u)$. Assim, o consumo de tempo de cada execução dessa linha é $T(n, m)$. Essa linha 5 é executada uma vez para cada nó na árvore dos prefixos dos k-menores caminhos $\langle P_1, \ldots, P_k \rangle$. Como o número de nós nessa árvore é não superior a $k\,n$, concluímos que o consumo de tempo total das execuções dessa linha é $O(k\,n\,T(n, m))$.

Teorema 3.6 (do consumo de tempo de YEN): *O consumo de tempo do algoritmo* YEN *é* $O(k\,n\,T(n, m))$*, onde* n *é o número de vértices e* m *é o número de arcos do grafo dado, respectivamente.* ∎

3.7 Revisão algorítmica

Antes de Yen, o problema k-MC é uma generalização do problema dos caminhos mínimos que havia sido considerado por vários autores. Aqui fazemos uma breve revisão dos trabalhos produzidos por esses autores.

Todos os algoritmos para o k-MC podem ser aplicados a grafos com custos negativos mas sem circuito negativo. No caso de grafos com custos negativos, os algoritmos devem utilizar o algoritmo de Richard Ernest Bellman [5] e Lester Randolph Ford [19]

em vez do algoritmo de Dijkstra. O consumo de tempo do algoritmo de Bellman-Ford é $O(mn)$.

Bock, Kantner e Haynes [6], em 1957, desenvolveram um algoritmo semelhante ao método GENÉRICO que enumerava todos os caminhos entre s e t, ordenava-os de acordo com seus custos e depois devolvia os k-menores caminhos. Essa estratégia utiliza tempo e memória exponencial e só pode fazer sentido em utilizá-la se k for extremamente grande.

Em 1961, Pollack [43] propôs um algoritmo que para obter os k-menores caminhos primeiro construía os $(k-1)$-menores caminhos e depois, para todo conjunto de $k-1$ arcos, um de cada caminho, encontrava um caminho mínimo de s a t que não passava por esses arcos. Depois disto, o algoritmo, escolhia determinava o k-ésimo menor caminho dentre todos os caminho encontrados. O algoritmo de Pollack pode ser aplicável quando k é pequeno, mas o seu consumo de tempo cresce exponencialmente com o valor de k.

Algoritmo POLLACK (V, A, c, s, t, k)
0 $P_1 \leftarrow$ *um st-caminho de custo mínimo em* (V, A)
1 ***para*** $i = 2, \ldots, k$ ***faça***
2 $\quad \mathcal{L} \leftarrow \emptyset$
3 $\quad \mathcal{A} \leftarrow \{\{a_1, \ldots, a_{i-1}\} : a_k$ *é arco em* $P_k\}$
4 $\quad$ ***para cada*** U *em* $\mathcal{A}$ ***faça***
5 $\quad\quad P_U \leftarrow$ *st-caminho de custo mínimo em* $(V, A - U)$
6 $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{P_U\}$
7 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$
8 ***devolva*** $\langle P_1, \ldots, P_k \rangle$

Clarke, Krikorian e Rausan [10], em 1963, introduziram um procedimento de *branch-and-bound* para encontrar os k-menores caminhos. O procedimento primeiro encontra o caminho mínimo se s a t e depois produz os k-menores caminhos enumerando *todos* os caminhos que "desviam" do caminho mínimo em algum vértice e selecionando o de menor custo.

A eficiência desse procedimento depende do grafo e custos sobre consideração. Se em cada iteração o menor caminho procurando for rapidamente gerado pelo proce-

dimento, então o custo desse caminho serve como um limitante (*bound*) que faz com que menos caminhos sejam gerados. Em geral, entretanto, é de se esperar que esse procedimento use uma quantidade de tempo e espaço muito grande.

Em seu algoritmo, Sakarovitch [44], em 1966, produz k^* passeios, $k^* \geq k$, através de uma rotina semelhante a de Hoffman e Pavley [30] para o problema dos k-menores *passeios*, que é brevemente discutido no capitulo 7. Em seguida, os k-menores caminhos são encontrado entre os k^* passeios produzidos.

A eficiência do algoritmo de Sakarovitch como o algoritmo de Clarke, Krikorian e Rausan, depende do grafo e dos custos sob consideração. Se os k^* passeios produzidos pelo algoritmo são na verdade caminhos, então o algoritmo termina muito rápidamente. Caso haja muitos desses passeios que contêm circuitos, então o número k^* de passeios produzidos pode ser muito grande.

O algoritmo de Yen [48, 49], apresentado neste capítulo, foi o primeiro algoritmo de consumo de tempo polinomial para o problema k-MC. Até hoje este é o algoritmo mais eficiente conhecido. O algoritmo de Yen, resolve $O(n)$ instâncias do CM para obter cada um dos k-menores caminhos. Seu consumo de tempo é $O(nkT(n,m)$.

Para grafos simétricos, Naoki Katoh, Toshihide Ibaraki e H. Mine [36], propuseram um refinamente do algoritmo de Yen que resolve $\Theta(1)$ instâncias do CM para cada um dos k-menores caminhos. O algoritmo de Katoh, Ibaraki e Mine e sua implementação são objeto de estudo do capítulo 5.

Finalmente, John Hershberger and Matthew Maxel e Subhash Suri [25, 26] propuseram um algoritmo para o k-MC que pode ser entendido como uma extensão das idéias de Katoh, Ibaraki e Mine.

A apresentação do algoritmo de Yen deste capítulo faz uso da linguagem utilizada por Hershberger, Maxel e Suri para apresentar o seu algoritmo, em particular, fazemos uso da árvore dos prefixos de uma coleção de caminhos. No próximo capítulo descrevemos as idéias de Hershberger, Maxel e Suri.

CAPÍTULO 4

Algoritmo de Hershberger, Maxel e Suri

John Hershberger, Matthew Maxel e Subhash Suri [25, 26] propuseram um algoritmo para o k-MC que é um refinamento do algoritmo YEN e que generaliza as idéias de Naoki Katoh, Toshihide Ibaraki e H. Mine [36] para o k-MC restrito a grafos simétricos. O algoritmo de Katoh, Ibaraki e Mine, apresentado no próximo capítulo, faz $\Theta(1)$ invocações a uma subrotina que resolve o CM para cada um dos k menores caminhos desejados e, portanto, tem consumo de tempo $\Theta(k\,T(n,m))$.

Em seu algoritmo, Hershberger, Maxel e Subhash, utilizam uma subrotina de um trabalho de Hershberger e Suri [27] e, inicialmente, achavam que esse novo algoritmo alcançava o mesmo consumo de tempo de $\Theta(k\,T(n,m))$ do algoritmo de Katoh, Ibaraki e Mine. Logo em seguida, porém, descobriram que a subrotina utilizada podia falhar em algumas situações [28]. A correção dessa falha fez com o novo algoritmo tivesse, no pior caso, o mesmo consumo de tempo de $\Theta(kn\,T(n,m))$ do algoritmo YEN.

Parece-nos que esse novo trabalho trouxe avanços significativos para a solução e compreensão do k-MC. Entre esses estes avanços estão a utilização de:

(a1) árvore dos prefixos de caminhos (seção 3.3) para a descrição do algoritmo;

(a2) uma partição que tem como refinamento a partição Π (seção 3.4) explcitamente usada pelo método GENÉRICO e implicitamente utilizada pelo algoritmo YEN que faz com que em cada iteração o número de caminhos na lista $\mathcal{L}$ de candidatos a i-ésimo menor caminho seja $O(i)$; e

(a3) uma heurística eficiente para o problema do "desvio de custo mínimo" (*replacement paths*) que apesar de falhar algumas vezes essa falha pode ser facilmente

detectada e, nesse caso, o algoritmo passa a executar uma subrotina mais lenta, porém correta.

O avanço (a1) ajudou muito na descrição do algoritmo YEN e, principalmente, na compreensão do algoritmo de Katoh, Ibaraki e Mine, que, na sua descrição, utilizava extenuante pseudo-código, muito próximo de código. Já o avanço (a2) fez com o número de caminhos na lista $\mathcal{L}$ de candidatos a i-ésimo menor caminho do algoritmo YEN passe de $O(in)$ para $O(i)$. Finalmente, (a3) traz a tona um problema que é naturalmente relacionado ao k-MC e que é interessante por si só.

Este capítulo contém os ingredientes do algoritmo de Hershberger, Maxel e Suri. Inicialmente, na próxima seção, fazemos uma revisão da partição dos caminhos, associada à árvore dos prefixos, vista no capítulo anterior. Em seguida, na seção 4.2, apresentamos uma descrição mais conceitual do algoritmo de Hershberger, Maxel e Suri que é discutido na seção 4.3. Terminamos este capítulo apresentando o problema do desvio de custo mínimo e uma heurística que procura resolvê-lo eficientemente.

4.1 Revisão da partição de caminhos

Sejam (V, A) um grafo e s e t dois de seus vértices. Seja $\mathcal{P}_{st}$ a coleção de todos os caminhos de s a t em (V, A). Suponha que $\mathcal{Q}$ seja uma coleção de caminhos de s a t. Na seção 3.4 foi descrita uma partição Π dos caminhos em $\mathcal{P} := \mathcal{P}_{st} - \mathcal{Q}$. Nesta seção definiremos uma outra partição Γ de $\mathcal{P}$. Esta partição é tal que Π é um refinamento de Γ e, portanto, o número de partes de Γ é não superior ao número de partes de Π. Na verdade, o ponto aqui é: o número de parte em Γ é, em geral, sensivelmente menor que o número de partes em Π.

No que segue suponha que (N, E, f) é a árvore dos prefixos de $\mathcal{Q}$ (seção 3.3). Na partição Π havia uma parte para cada nó u de N que não é folha. Cada parte da nova partição Γ será indexada pelas partes de uma partição dos nós da arborescência (N, E) que não são folhas, ao invés de apenas por um único nó, como em Π.

N_0, N_1 Para $i = 0, 1$, seja N_i o conjunto de nós da arborescência (N, E) com grau de saída i. Seja ainda $N_{\geq 2}$ o conjunto dos nós de grau de saída maior ou igual a 2. Assim, os nós em N_0 são as folhas de (N, E), os nós em N_1 induzem caminhos em (N, E) e $N_{\geq 2} = N - N_0 - N_1$. Sendo assim, N_0, N_1 e $N_{\geq 2}$ formam uma partição de N. As definições estão ilustrada na figura 4.1(b).

$N_{\geq 2}$

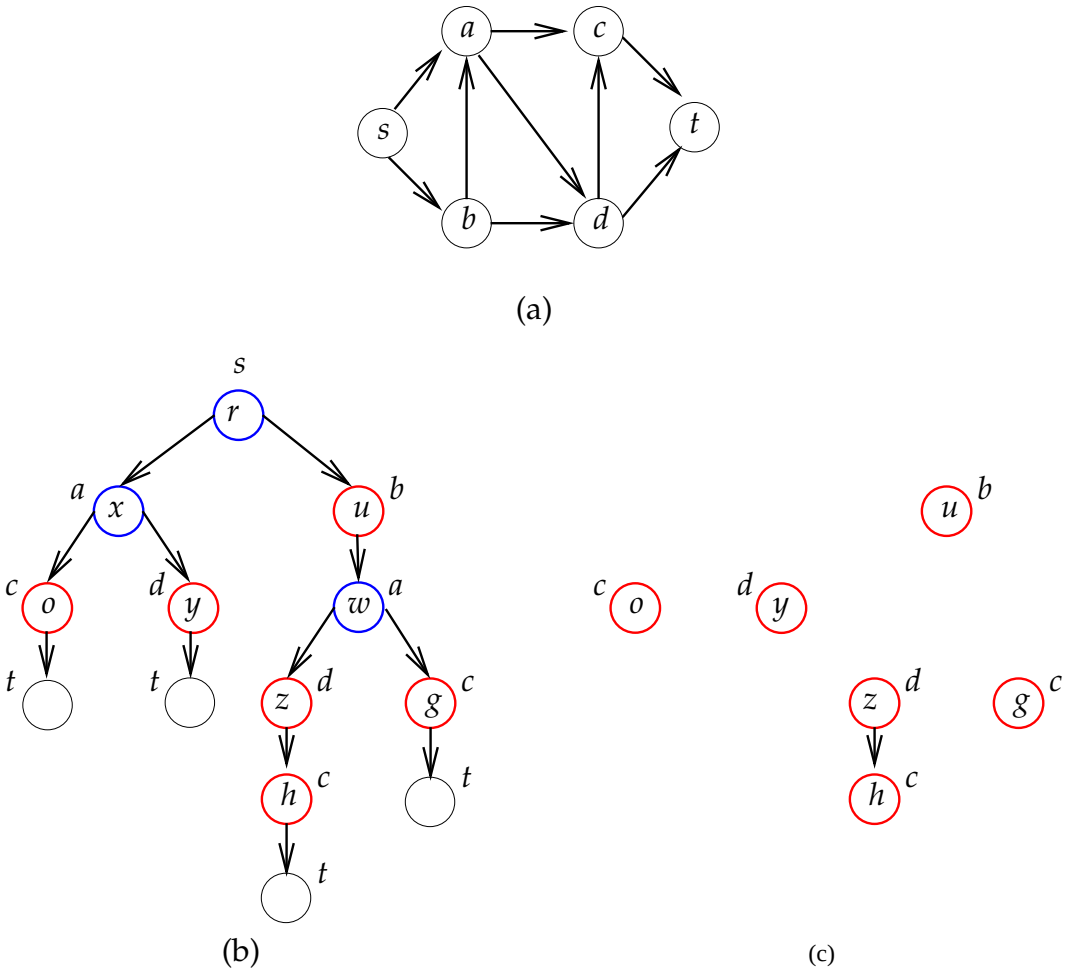


Figura 4.1: (b) mostra a árvore dos prefixos dos caminhos $\langle s, a, c, t\rangle$, $\langle s, a, d, t\rangle$, $\langle s, b, a, c, t\rangle$ e $\langle s, b, a, d, c, t\rangle$ no grafo em (a). Na árvore, um símbolo ao lado de um nó é o rótulo desse nó. O símbolo dentro de um nó é o seu nome. Na árvore dos prefixos temos que os nós em preto são folhas e estão em N_0, os nós vermelhos estão em N_1 e os nós azuis estão em $N_{\geq 2}$. Na figura (c) são mostrados os caminhos da arborescência induzidos pelos nós em N_1.

Definimos uma partição $\mathcal{U}$ dos nós de uma arborescência (N, E) que não são folhas
u da seguinte maneira. Seja $\mathcal{U} := \mathcal{U}_1 \cup \mathcal{U}_{\geq 2}$ a partição dos nós de $N - N_0$ tal que

$$\mathcal{U}_1 = \{\{u_0, u_1, \ldots, u_k\} : \langle u_0, u_1, \ldots, u_k\rangle \text{ é um caminho maximal em } (N, E) \text{ formado apenas por nós em } N_1\} \text{ e}$$
$$\mathcal{U}_{\geq 2} = \{\{u\} : u \text{ é um nó em } N_{\geq 2}\}$$

$\mathcal{U}_1, \mathcal{U}_{\geq 2}$ No exemplo ilustrado pela figura 4.1 temos que

$$\mathcal{U}_1 = \{\{g\}, \{o\}, \{u\}, \{y\}, \{z, h\}\} \text{ e}$$
$$\mathcal{U}_{\geq 2} = \{\{r\}, \{x\}, \{w\}\}$$

Pela definição de $\mathcal{U}_1$ e $\mathcal{U}_{\geq 2}$ é evidente que $\mathcal{U}$ é uma partição de $N - N_0$.

Γ A partição Γ dos caminhos em $\mathcal{P}$ é formada por uma parte γ_U para cada parte U de $\mathcal{U}$ onde

$$\gamma_U := \bigcup_{u \in U} \pi_u \,,$$

e π_u é a parte Π como definida na seção 3.4. De maneira semelhante ao que ocorre com a partição Π, a cada coleção $\mathcal{Q}$ de caminhos temos uma única árvore dos prefixos (N, E, f) de $\mathcal{Q}$ e associada a essa árvore temos uma única partição Γ de $\mathcal{P}$. Algumas vezes nos referimos a Γ como sendo a **partição associada** à coleção $\mathcal{Q}$.

Da definição de Γ e do teorema 3.3 da partição segue imediatamente o seguinte teorema.

Teorema 4.1 (da partição revista): *Sejam (V, A) um grafo, s e t dois de seus vértices e $\mathcal{Q}$ uma coleção de caminhos de s a t. Se P é um caminho de s a t que não está em $\mathcal{Q}$, então P pertence a uma única parte de Γ.* ∎

4.2 Método HMS-GENÉRICO

Agora, podemos facilmente reescrever o método YEN-GENÉRICO em termos da partição Γ em vez de Π. A seguir, no método HMS-GENÉRICO, a subrotina ATUALIZE-GENÉRICO-HMS é a responsável pela atualização da partição Γ após o caminho P_i ser incluído em $\mathcal{Q}$ na linha 5.

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

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

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

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

3 $\quad \mathcal{L} \leftarrow \{P_\gamma : P_\gamma$ *é caminho mínimo na parte* γ *de* $\Gamma\}$

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

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

6 $\quad \Gamma \leftarrow$ ATUALIZE-GENÉRICO-HMS $(V, A, s, t, \mathcal{Q})$

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

Neste ponto estamos preparados para limitar o número de partes em Γ. Limitaremos esse número a partir de um fato básico que só tem a ver com a estrutura de arborescências.

Fato 4.2: *Se* (N, E) *é uma arborescência então* $|\mathcal{U}| = |\mathcal{U}_1| + |\mathcal{U}_{\geq 2}| \leq 3|N_0| - 2$.

Demonstração: A demonstração é por indução no número de folhas $|N_0|$ da arborescência (N, E).

Se $|N_0| = 1$, então $|\mathcal{U}_1| \leq 1$ e $|\mathcal{U}_{\geq 2}| = 0$ e portanto $|\mathcal{U}_1| + |\mathcal{U}_{\geq 2}| \leq 3|N_0| - 2$.

Podemos supor que $|N_0| \geq 2$. Seja $\langle u_0, \ldots, u_q \rangle$ um caminho em (N, E) em que $\{u_0\}$ está em $\mathcal{U}_{\geq 2}$ ($u_0 \in N_{\geq 2}$) e u_q é folha. Assim, se $q > 1$, então $\{u_1, \ldots, u_{q-1}\}$ está em $\mathcal{U}_1$. Seja ainda (N', E') a arborescência definida por

$$N' := N - \{u_1, \ldots, u_q\} \quad \text{e}$$
$$E' := E - \{u_0u_1, \ldots, u_{q-1}u_q\}.$$

Por definição temos que $|N'_0| = |N_0| - 1$. Se $\mathcal{U}'_1$ e $\mathcal{U}'_{\geq 2}$ são as partições associadas à arborescência (N', E'), então, por indução temos que

$$|\mathcal{U}'| = |\mathcal{U}'_1| + |\mathcal{U}'_{\geq 2}| \leq 3|N'_0| - 2. \tag{4.1}$$

Passamos agora a considerar dois casos dependendo do grau de saída de u_0 em (N', E'). As análises desses casos concluem a demonstração.

Caso 1. u_0 tem grau de saída maior ou igual a 2 em (N', E')

Nesse caso $\{u_0\}$ está em $\mathcal{U}'_{\geq 2}$ e assim

$$\begin{aligned} |\mathcal{U}_1| &\leq |\mathcal{U}'_1| + 1 \quad \text{e} \\ |\mathcal{U}_{\geq 2}| &= |\mathcal{U}'_{\geq 2}| \,. \end{aligned}$$

Portanto,

$$\begin{aligned} |\mathcal{U}| &= |\mathcal{U}_1| + |\mathcal{U}_{\geq 2}| \\ &\leq |\mathcal{U}'_1| + |\mathcal{U}'_{\geq 2}| + 1 \\ &\leq 3|N'_0| - 2 + 1 \qquad (4.2) \\ &= 3(|N_0| - 1) - 1 \\ &= 3|N_0| - 4 \\ &< 3|N_0| - 2 \,, \end{aligned}$$

onde (4.2) é devido a hipótese de indução (4.1).

Caso 2. u_0 tem grau de saída igual a 1 em (N', E')

Nesse caso $\{u_0\}$ não está em $\mathcal{U}'_{\geq 2}$ e portanto

$$\begin{aligned} |\mathcal{U}_1| &\leq |\mathcal{U}'_1| + 2 \quad \text{e} \\ |\mathcal{U}_{\geq 2}| &= |\mathcal{U}'_{\geq 2}| + 1 \,. \end{aligned}$$

Logo,

$$\begin{aligned} |\mathcal{U}| &= |\mathcal{U}_1| + |\mathcal{U}_{\geq 2}| \\ &\leq |\mathcal{U}'_1| + |\mathcal{U}'_{\geq 2}| + 3 \\ &\leq 3|N'_0| - 2 + 3 \qquad (4.3) \\ &= 3(|N_0| - 1) + 1 \\ &= 3|N_0| - 2 \,, \end{aligned}$$

onde (4.3) é devido a hipótese de indução (4.1). ∎

Teorema 4.3 (do número de partes): *Sejam (V, A) um grafo, s e t dois de seus vértices, $\mathcal{Q}$ uma coleção de caminhos de s a t e (N, E, f) a árvore dos prefixos de $\mathcal{Q}$. Se Γ é a partição associada à $\mathcal{Q}$, então $|\Gamma| \leq 3|\mathcal{Q}| - 2$.*

Demonstração: Seja (N, E, f) a árvore dos prefixos de $\mathcal{Q}$. Temos que o número $|N_0|$ de folhas da arborescência (N, E) é igual a $|\mathcal{Q}|$. Portanto,

$$\begin{aligned} |\Gamma| &= |\mathcal{U}| \\ &= |\mathcal{U}_1| + |\mathcal{U}_2| \\ &\leq 3|N_0| - 2 \qquad (4.4) \\ &= 3|\mathcal{Q}| - 2\,, \end{aligned}$$

onde a desigualdade (4.4) é devida ao fato 4.2. ∎

Do teorema 4.3 do número de partes segue imediatamente que

> após cada execução da linha 3 do método HMS-GENÉRICO temos que $|\mathcal{L}| \leq 3i - 2$. (4.5)

4.3 Algoritmo HMS

O algoritmo HMS de Hershberger, Maxel e Suri é textualmente idêntico ao algoritmo YEN. Isto não é uma surpresa já que desde o início deste capítulo foi mencionado que o algoritmo deste capítulo é um refinamento do algoritmo YEN. O refinamente está escondido, pelo menos por enquanto, na rotina de atualização da lista $\mathcal{L}$ de candidatos a i-ésimo menor caminho: o algoritmo YEN atualiza $\mathcal{L}$ usando, implicitamente, a partição Π enquanto HMS faz o mesmo serviço através da partição Γ. YEN mantém em $\mathcal{L}$ um caminho de custo mínimo de em para cada parte de Π e HMS mantém em $\mathcal{L}$ um caminho de custo mínimo para cada parte de Γ. Como $|\Gamma| \leq |\Pi|$ o algorimo HMS freqüentemente armazena menos caminhos que YEN.

Algoritmo HMS (V, A, c, s, t, k)

1 $\quad \mathcal{L} \leftarrow \{$*um caminho de custo mínimo de* s *a* $t\}$
2 $\quad (N, E, f) \leftarrow$ *árvore dos prefixos de* $\emptyset \quad \triangleright$ *árvore vazia*
3 $\quad$ ***para*** $i = 1, \ldots, k$ ***faça***
4 $\quad\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$
5 $\quad\quad (N, E, f, \mathcal{L}) \leftarrow$ ATUALIZE-HMS $(V, A, c, s, t, P_i, N, E, f, \mathcal{L})$
6 $\quad$ ***devolva*** $\langle P_1, \ldots, P_k \rangle$

A correção do algoritmo HMS segue da correção de YEN.

Teorema 4.4 (da correção de HMS): *Dado um grafo (V, A), uma função custo c e vértices s e t o algoritmo* HMS *corretamente encontra os k-menores caminhos de s a t.* ∎

O consumo de tempo da linha 1 de HMS é $T(n, m)$. A linha 2 consome tempo $\Theta(1)$. Devido a (4.5), cada execução da linha 4 consume tempo $O(\lg i)$ se utilizarmos um min-heap para armazenarmos os custos dos caminhos em $\mathcal{L}$. Portanto, O consumo de tempo de todas as execuções da linha 4 é $O(k \lg k)$. O consumo de tempo do algoritmo HMS é dominado pelo consumo de tempo das k execuções de ATUALIZE-HMS.

O algoritmo ATUALIZE-HMS, que está logo a seguir, utiliza como subrotina o algoritmo ÁRVORE-DOS-PREFIXOS que recebe uma árvore dos prefixos de uma coleção de caminhos $\mathcal{Q}'$ com ponta inicial em um vértice s e um caminho P com ponta inicial em s e devolve a árvore dos prefixos de $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$. Essa subrotina já foi utilizada pelo algoritmo ATUALIZE-YEN da seção 3.6. O serviço feito por ÁRVORE-DOS-PREFIXOS é essencialmente descrito na demonstração do teorema 3.1 e está ilustrado na figura 3.2.

Na descrição do algoritmo, da mesma maneira que na seção 3.5, se u é um nó da árvore do prefixos (N, E, f), então R_u é o caminho da raiz a u em (N, E) e $A_u = \{f(u)f(w) : uw \in E\}$.

Algoritmo ATUALIZE-HMS $(V, A, c, s, t, P, N', E', f', \mathcal{L}')$

0 $\quad \mathcal{L} \leftarrow \mathcal{L}' - \{P\}$

1 $\quad$ *Seja* $\mathcal{U}' = \mathcal{U}'_1 \cup \mathcal{U}'_{\geq 2}$ *a partição dos nós da arborescência* (N', E') *que não são folhas*

2 $\quad$ *Seja* $U' \in \mathcal{U}'$ *tal que* $P \in \gamma_{U'}$

3 $\quad (N, E, f) \leftarrow$ ÁRVORE-DOS-PREFIXOS(N', E', f', P)

4 $\quad$ *Seja* $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_{\geq 2}$ *a partição dos nós da arborescência* (N, E) *que não são folhas*

5 $\quad$ ***para cada*** $U \in \mathcal{U} - (\mathcal{U}' - \{U'\})$ ***faça***

6 $\quad\quad P_U \leftarrow$ *caminho de s a t de custo mínimo com prefixo $f(R_u)$ e que não possui arcos em A_u* ***com*** *u* ***em*** *U*

7 $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{P_U\}$

8 ***devolva*** $(N, E, f, \mathcal{L})$

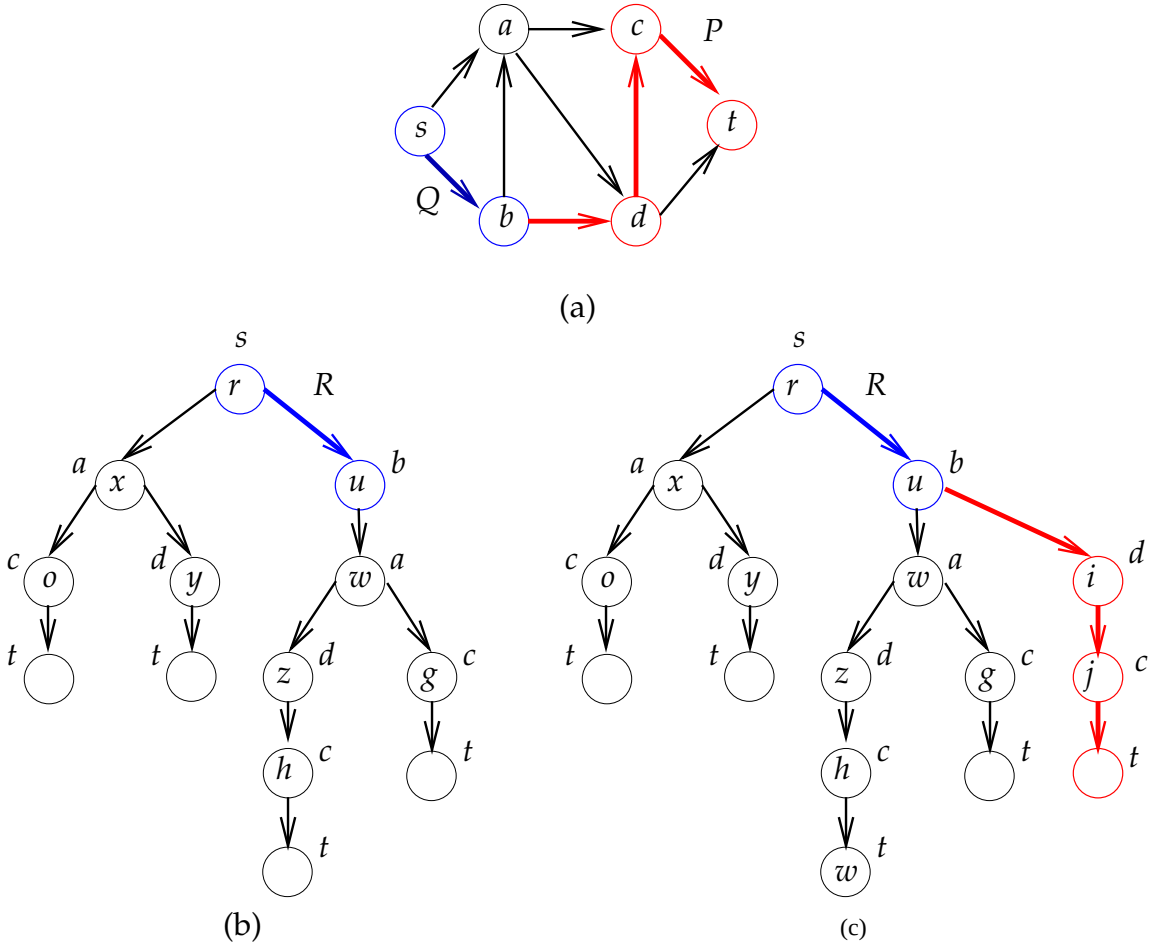


Figura 4.2: Ilustração de duas possíveis árvores dos prefixos (N', E', f') e (N, E, f) na execução do algoritmo ATUALIZE-HMS. A figura (a) mostra o caminho $P = \langle s, b, d, c, t \rangle$ com vértices e arcos de cor azul e vermelha. Na figura (b) vemos a árvore dos prefixos (N', E', f') de $\mathcal{Q}' = \{\langle s, a, c, t \rangle, \langle s, a, d, t \rangle, \langle s, b, a, c, t \rangle, \langle s, b, d, c, t \rangle\}$. Na figura (c) está a árvore dos prefixos (N, E, f) de $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$ computada na linha 3.

Para exemplificarmos um pouco a execução de ATUALIZE-HMS considere a figura 4.2 que mostra duas possíveis árvores dos prefixos (N', E', f') e (N, E, f) durante uma execução do algoritmo ATUALIZE-HMS.[1]

[1]Esta mesma ilustração já foi exibida na seção 3.3 para ilustrar a demonstração do teorema 3.1.

Na ilustração, o caminho P recebido pelo algoritmo é $\langle s, b, d, c, t\rangle$. Na figura 4.2(b) vemos a árvore dos prefixos de $\mathcal{Q}' = \{\langle s, a, c, t\rangle, \langle s, a, d, t\rangle, \langle s, b, a, c, t\rangle, \langle s, b, d, c, t\rangle\}$.

Para a arborescência (N', E') da figura 4.2(b) temos que, na linha 1 do algoritmo,

$$\begin{aligned} \mathcal{U}'_1 &= \{\{u\}, \{o\}, \{y\}, \{z, h\}, \{g\}\} \quad \text{e} \\ \mathcal{U}'_{\geq 2} &= \{\{r\}, \{x\}, \{w\}\}\,. \end{aligned}$$

Na linha 2, $U' = \{u\}$ é a parte de $\mathcal{U}'$ tal que P está em $\gamma_{U'}$. A figura 4.2(c) mostra a árvore dos prefixos de $\mathcal{Q} = \mathcal{Q}' \cup \{P\}$ computada na linha 3. Para a arborescência (N, E) temos que, na linha 4 do algoritmo,

$$\begin{aligned} \mathcal{U}_1 &= \{\{o\}, \{y\}, \{z, h\}, \{g\}, \{i, j\}\} \quad \text{e} \\ \mathcal{U}_{\geq 2} &= \{\{r\}, \{x\}, \{w\}, \{u\}\} \end{aligned}$$

Portanto, as partes em $\mathcal{U} - (\mathcal{U}' - \{U'\})$ na linha 5 são

$$\{\{u\}, \{i, j\}\}$$

Assim, o bloco de linhas 5–7 será executado duas vezes:

- uma vez para determinar o caminho de custo mínimo em $\gamma_{\{u\}} = \pi_u = \emptyset$, já que $R_u = \langle r, u\rangle$, $f(R_u) = \langle s, b\rangle$ e $A_u = \{ba, bd\}$;
- outra vez para determinar o caminho de custo mínimo em $\gamma_{\{i,j\}} = \langle s, b, d, t\rangle$, já que:
  - $R_i = \langle r, u, i\rangle$, $f(R_i) = \langle s, b, d\rangle$ e $A_i = \{dc\}$; e
  - $R_j = \langle r, u, i, j\rangle$, $f(R_j) = \langle s, b, d, c\rangle$ e $A_j = \{ct\}$.

Muito do trabalho feito pelo algoritmo ATUALIZE-HMS é conceitual. O consumo de tempo do algoritmo é dominado pelo consumo de tempo de todas as execuções da linha 6, que pode ser reescrita da seguinte maneira mais expandida:

6a $\quad \mathcal{F} \leftarrow \emptyset$

6b $\quad$ ***para cada*** $u \in U$ ***faça***

6c $\quad\quad P_u \leftarrow$ *caminho de* s *a* t *de custo mínimo com prefixo* $f(R_u)$ *e que não possui arcos em* A_u

6d $\quad\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{P_u\}$

6e $\quad P_U \leftarrow$ *caminho de custo mínimo em* $\mathcal{F}$

Devido a (4.5), sabemos que o bloco de linhas 4,5 e 6 é executado no máximo 4 vezes cada vez que o algoritmo ATUALIZE-HMS é invocado. Se U está em $\mathcal{U}_{\geq 2}$, estão U contém apenas um nó e o consumo de tempo de uma execução do bloco de linhas 6a–6e é $O(T(n,m))$, já que a execução dessas linhas se reduz a resolver uma instância do problema do subcaminho mínimo SCM em um subgrafo apropriado de (V,A) (seção 3.6). Já, se U está em $\mathcal{U}_1$, então U é formado por um conjunto de nós $\{u_0,u_1,\ldots,u_k\}$ em N_1 tal que $\langle u_0,u_1,\ldots,u_k\rangle$ é um caminho na arborescência (N,E). Como k pode ser proporcional ao número de nós do grafo (V,A) concluímos que, para U em $\mathcal{U}_1$ o consumo de tempo de uma execução do bloco de linhas 6a–6e poder ser $\Theta(nT(n,m))$ no pior caso. Com isto podemos afirmar apenas que o consumo de tempo do algoritmo ATUALIZE-HMS é $O(nT(n,m))$ e que o HMS é $O(k\,n\,T(n,m))$.

Teorema 4.5 (do consumo de tempo de HMS): *O consumo de tempo do algoritmo* HMS *é* $O(k\,n\,T(n,m))$*, onde* n *é o número de vértices e* m *é o número de arcos do grafo dado, respectivamente.* ∎

Na verdade, se trocarmos a linha 6 pelas linhas 6a–6e, o algoritmo HMS torna-se idêntico ao algoritmo YEN. Na próximo seção tratamos de um problema que é naturalmente derivado do desejo de executar o bloco de linhas 6a–6e de uma maneira mais eficiente para U em $\mathcal{U}_1$. Veremos uma heurística para o problema de encontrar um caminho mínimo P_U, em que U é uma parte em $\mathcal{U}_1$, que consome tempo $O(T(n,m))$, mas que as vezes falha. Quando a heurística falha, HMS executa, precisamente como YEN, as linhas 6a–6e.

4.4 Desvios mínimos

O problema e a heurística desta seção foram propostos por Hershberger, Maxel e Suri [27, 28, 25, 26].

Suponha que (V,A) é um grafo, s e t são dois de seus vértices, $\mathcal{Q}$ é uma coleção de caminhos de s a t. Suponha ainda que (N,E,f) é a árvore dos prefixos de $\mathcal{Q}$. Lembremos que

$$\mathcal{U}_1 = \{\{u_0,u_1,\ldots,u_k\} : \langle u_0,u_1,\ldots,u_k\rangle \text{ é um caminho maximal em } (N,E) \text{ formado apenas por nós em } N_1\}\,.$$

No final da seção anterior, desejávamos uma subrotina eficiente para executar o bloco

de linhas 6a–6e do algoritmo HMS quando U está em $\mathcal{U}_1$. Assim, no que segue, suponha que U está em $\mathcal{U}_1$ e que $\langle u_0, u_1, \ldots, u_k \rangle$ é o caminho em (N, E) formado pelos nós que estão em U. Devido à definição de $\mathcal{U}_1$ temos que para cada u em U o conjunto A_u possui apenas um único arco. De fato, para $i = 0, \ldots, k$,

$$A_{u_i} = \{f(u_i)f(u_{i+1})\},$$

onde u_{k+1} é o nó tal que $u_k u_{k+1}$ é o único arco na arborescência (N, E) com ponta inicial em u_k. Assim, u_{k+1} é uma folha ou um nó com grau de saída pelo menos dois na arborescência (N, E), ou seja $u_k \in N_0 \cup N_{\geq 2}$.

O caminho P_U obtido na linha 6e é aquele caminho de custo mínimo de s a t em (V, A) tal que, para algum u_i em U, $0 \leq i \leq k$,

- $f(R_{u_i})$ é prefixo de P_U; e
- P_U não possui o arco em A_{u_i}.

Nesse caso, dizemos que P_U é o caminho de custo mínimo de s a t em (V, A) que **desvia** do caminho de $f(R_{u_k})$ no vértice $f(u_i)$.

Para o algoritmo HMS desejamos, portanto, uma subrotina que resolve o seguinte **problema do desvio mínimo**, denotado por DM:

DM **Problema** DM(V', A', c', s', t', Q): Dado um grafo (V', A'), uma função custo c', dois vértice s' e t' e um caminho Q, encontrar um caminho de custo mínimo de s' a t' que **não** tem Q como prefixo.

Encontrar o caminho P_U é equivalente a resolver o problema DM onde

- (V', A') é o grafo resultante de (V, A) após a remoção de todos os vértices no caminho $f(R_{u_0})$, exceto $f(u_0)$,
- c' é a restrição da função custo c aos arcos em A',
- $s' = f(u_0)$, $t' = t$ e
- $Q = f(\langle u_0, \ldots, u_k \rangle)$.

É evidente que se Q não é prefixo de um caminho mínimo de s' a t' em (V', A') então o problema pode ser resolvido através de apenas uma invocação de um subrotina para

o CM. Assim, o algoritmo DM-HMS adiante supõe que Q é prefixo de um caminho mínimo de s' a t'. Um algoritmo ingênuo para resolver o DM pode, para vw em Q encontrar o caminho de custo mínimo de s' a t' no grafo $(V', A' - \{vw\})$; um caminho de menor custo dentre estes é uma solução do problema. O consumo tempo desta solução ingênua é $O(|Q|T(n', m'))$, onde $n' = |V'|$ e $m' = |A'|$. A heurística DM-HMS mais adiante consome tempo $O(T(n', m'))$. Apesar de falhar em algumas situações a falha pode ser facilmente detectada e, nesse caso, outro algoritmo mais lento deve ser executado.

Para resolver o problema DM o algoritmo DM-HMS faz uso de duas funções potencias (seção 2.2) especiais.

Uma função potencial y dos vértices de V' em $\mathbb{Z}$ é $(s', *)$-**ótima** se

> para cada vértice v em V', $y(v) - y(s')$ é o menor custo de um caminho de s' a v.

Ao invocarmos DIJKSTRA(V', A', c', s') (seção 2.5) o algoritmo devolve uma arborescência (V', S') dos caminhos mínimos de s' aos demais vértices do grafo e uma função potencial $(s', *)$-ótima tal que $y(s') = 0$. Assim, nesse caso, para todo v em V' temos que $y(v)$ é o menor custo de um caminho de s' a v.

Um função potencial z dos vértices de V' em $\mathbb{Z}$ é $(*, t')$-**ótima** se

> para cada v em V', $z(v) - z(t')$ é o menor custo de um caminho de v a t'.

Também podemos obter uma função $(*, t')$-ótima através do algoritmo de Dijkstra. Ao invocarmos DIJKSTRA$(V', \tilde{A}, \tilde{c}, t')$, onde $\tilde{A} = \{wv : vw \in A'\}$ e $\tilde{c}(wv) = c'(vw)$ para todo arco wv em $\tilde{A}$ podemos supor que o algoritmo devolve uma arborescência (V', T') dos caminhos de custos mínimos de cada vértice do grafo até t' e uma função potencial $(*, t')$-ótima z tal que $z(t') = 0$. Portanto, para todo v em V' temos que $z(v)$ é o menor custo de um caminho de v a t'.

A heurística a seguir supõe que $Q = \langle s'{=}v_0, v1, \ldots, v_k \rangle$ está em um caminho de custo mínimo de s' a t'. O algoritmo supõe ainda que Q é subcaminho das arborescências (V', S') e (V', T') computadas nas linhas 1 e 2. Na linha 10, S'_v é o caminho de s' a v na arborescência (V', S') e T'_w é o caminho de w a t' na arborescência (V', T').

Heurística DM-HMS (V', A', c', s', t', Q)

```
1   (V', S'), y ← DIJKSTRA(V', A', c', s')
2   (V', T'), z ← DIJKSTRA(V', Ã, c̃, t')
3   para cada arco v_i v_{i+1} em Q faça
4       Seja X_i o componente de (V', S' − {v_i v_{i+1}}) que contém s'
5       Seja E_i o conjunto dos arcos com ponta inicial em X_i
            e ponta final em V' − X_i
6       d ← ∞    P ← ⟨⟩
7       para cada vw em E_i faça
8           se d > y(v) + c'(vw) + z(w)
9               então d ← y(v) + c'(vw) + z(w)
10                  P ← S'_v · vw · T'_w    ▷ concatenação dos caminhos
11                  v_0 ← v    w_0 ← w
12  devolva P, v_0, w_0
```

A execução das linhas 1 e 2 da heurística consome tempo $O(T(n', m'))$, onde $n' = |V'|$ e $m' = |A'|$. Ao longo da execução da heurística, um arco vw em $A' - S'$ precisa ser examinado no bloco de linhas 8–11 no máximo uma vez. Como o consumo de tempo de cada execução desse bloco de linhas é $\Theta(1)$, então o consumo de tempo total gasto pela heurística executando o bloco de linha 8–11 é $O(|A'|)$. Para determinar os arcos para os quais as linhas 7–11 deverão ser executadas basta, para cada vértice v' em V', determinamos o índice $p(v') = i$ do vértice v_i em Q mais próximo de v' na arborescência (V', S'). Um arco vw em A' deverá ser submetido ao exame das linhas 8–11 se $p(v) < p(w)$. O algoritmo DIJKSTRA pode ser adaptado para determinar $p(v')$ para cada vértice v' em V', sem custo assintótico adicional. Assim, o bloco de linhas 4–7 pode, essencialmente, ser executado juntamente com o pré-processamento feito na linha 1. A discussão aqui apresentada encontra-se resumida no teorema a seguir.

Teorema 4.6 (do consumo de tempo de DM-HMS): *O consumo de tempo da heurística* DM-HMS *é* $O(T(n', m'))$*, onde* n' *é o número de vértices e* m' *é o número de arcos do grafo dado, respectivamente.* ∎

Consideremos agora como a heurística pode falhar em algumas situações. Para isto

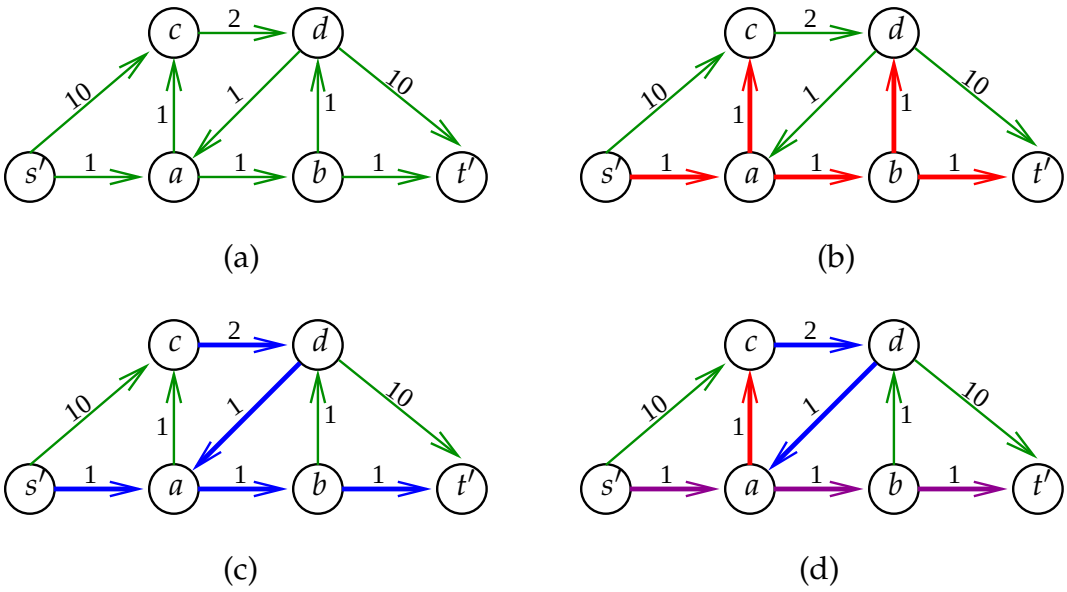


Figura 4.3: Exemplo em que a heurística DM-HMS falha. Na figura (a) vemos o grafo (V', A') e a função custo c'. O caminho Q recebido pela heurística é $\langle s', a, b \rangle$. Nas figuras (b) e (c), os arcos em destaque formam as arborescências S' e T' computadas nas linhas 1 e 2 da heurística, respecticamente. Em (d), vemos o passeio $\langle s', a, c, d, a, b, t' \rangle$, de custo 7, devolvido, erroneamente, pela heurística. O caminho que deveria ter sido devolvido é $\langle s', a, b, d, t' \rangle$ de custo 13.

considere o grafo (V', A') ilustrado na figura 4.3(a). Na figura, um número próximo a um arco representa o seu custo. O caminho Q é $\langle v_0, v_1, v_2 \rangle = \langle s', a, b \rangle$ que é prefixo do único caminho de custo mínimo de s' a t'. As figuras 4.3(b) e 4.3(c) mostram as arborescências (V', S') e (V', T') computadas na linhas 1 e 2.

Para $i = 0$ e $v_i v_{i+1} = s'a$, a execução do bloco de linhas 4–11 determina que $\langle s', c, d, a, b, t' \rangle = S'_{s'} \cdot s'c \cdot T'_c$ é o caminho de custo mínimo de s' a t' que desvia do prefixo Q em s'. Aqui temos que $S'_{s'} = \langle s' \rangle$ e $T'_c = \langle c, d, a, b, t' \rangle$ e que o caminho resultante tem custo 15.

Já, na iteração do bloco de linhas 4–11 em que $i = 1$ e $v_i v_{i+1} = ab$, a heurística determina que $\langle s', a, c, d, a, b, t' \rangle = S'_c \cdot cd \cdot T'_d$ é o "caminho" de custo mínimo que desvia de Q em a. Esse "caminho" tem custo 7, $S'_{c'} = \langle s', a, c \rangle$ é prefixo de P e $T'_d = \langle d, a, b, t' \rangle$ é sufixo de P. A heurística, nesse caso, devolve erroneamente, $P = \langle s', a, c, d, a, b, t' \rangle$ como sendo o caminho de custo mínimo de s' a t' que não tem Q como prefixo, no entanto, P não é um caminho.

A falha aqui foi devida ao fato que o sufixo T'_c de P contém o arco $v_i v_{i+1} = ab$ da iteração. Para que esse falha seja detecta basta para cada vértice v' determinarmos o menor índice $q(v') = i$ tal que o caminho $T_{v'}$ de v' a t' contém o vértice v_i de Q. Para cada vw em algum E_i o passeio P computado na linha 10 é um caminho se $q(w) > p(u)$. Assim, P devolvido na linha 12 pela heurística não é solução do problema do desvio de custo mínimo se $q(w_0) < p(v_0)$.

Hershberger, Maxel e Suri [26] implementaram o algoritmo HMS utilizando como subrotina a heurística DM-HMS e reportaram que resultados empíricos mostram que a heurística falha em menos de 1% das vezes em que foi executada. Isto indica que, na prática, HMS tem um desempenho significativamente melhor que YEN.

Em grafos simétricos a heurística DM-HMS não falha. Esta é a subrotina central do algoritmo de Naoki Katoh, Toshihide Ibaraki e H. Mine [36] para o k-MC restrito a grafos simétricos. O algoritmo de Katoh, Ibaraki e Mine, tópico central do próximo capítulo é, essencialmente, a restrição do algoritmo HMS para grafos simétricos. Na figura 4.4 vemos a versão para grafos simétricos do exemplo na figura 4.3.

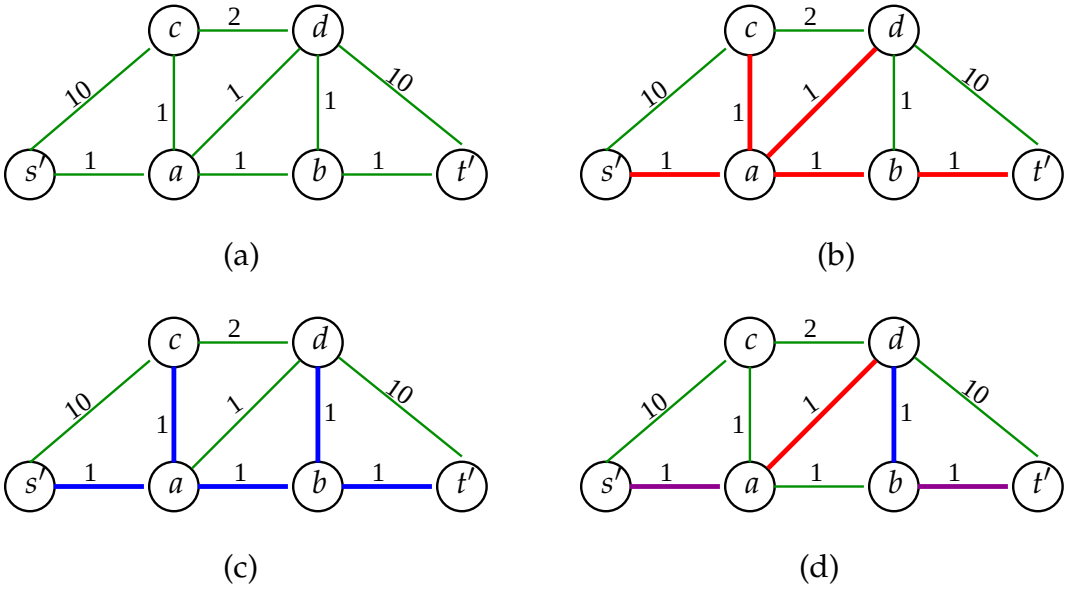


Figura 4.4: Exemplo da figura 4.3 em que o grafo é simétrico. Na figura (a) vemos o grafo simétrico (V', A') e a função custo c'. O caminho Q recebido pela heurística é $\langle s', a, b \rangle$. Nas figura (b) e (c), os arcos em destaque formam as árvore de custo mínimo S' e T' com raizes s' e t', respectivamente. Em (d) vemos o caminho que é corretamente devolvido pela versão da heurística DM-HMS para grafos simétricos devida a Katoh, Ibaraki e Mine.

CAPÍTULO 5

Algoritmo de Katoh, Ibaraki e Mine

Neste capítulo trataremos, propriamente dito, do algoritmo desenvolvido por Katoh, Ibaraki e Mine, o qual chamaremos de KIM. Para entendê-lo fizemos usos de vários artigos diferentes. Começamos pelo artigo original de Naoki Katoh, Toshihide Ibaraki e H. Mine [36] (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. Entretanto, em se tratando do entendimento em termos gerais não ajudou muito.

Uma vez que o artigo de Jin Y. Yen [48] fora citado pelo de KIM, achamos que seria de grande utilidade lê-lo. Além do mais, sabemos que o algoritmo KIM é uma melhoria do YEN, sendo específico para grafos simétricos. Após a leitura do artigo de Yen, começamos a vislumbrar melhor o algoritmo KIM e a entendê-lo com mais propriedade.

Em seguida, encontramos o artigo de John Hershberger, Matthew Maxel e Subhash Suri [26], o qual foi muito elucidativo. Este trata de uma extensão da idéia central do algoritmo KIM para grafos. O que mais nos chamou a atenção foi a maneira como o algoritmo foi descrito. Os autores procuraram trabalhar com idéias mais gerais e lidar com estruturas mais sofisticadas, deixando de lado a quantidade exorbitante de índices e descrições de baixo nível apresentadas no artigo de KIM.

Finalmente, vale citar o artigo de Eleni Hadjiconstantinou e Nicos Christofides [24] que trata de uma implementação do algoritmo KIM utilizando algumas mudanças que levam a um melhor desempenho na prática. Neste artigo, a descrição do algoritmo é feita com mais clareza, razão pela qual decidimos citá-lo aqui.

Nosso plano é apresentar o algoritmo KIM, bem como suas subrotinas e principais

idéias, comentando sobre o desempenho assintótico das principais rotinas, em seguida exibiremos uma simulação e finalizaremos exibindo pontos importantes sobre a nossa implementação.

5.1 Visão Geral

O algoritmo KIM é, de certa forma, uma versão do algoritmo YEN específica para grafos simétricos. Mais especificamente, o algoritmo HMS apresentado no capítulo 4, pode ser visto como a aplicação de algumas das idéias do algoritmo KIM à grafos. O algoritmo KIM aplica uma técnica mais fina de particionamento de caminhos, semelhante à apresentada no capítulo 4, sendo que o número de partições é sempre menor a $3 \times |Q|$, onde $|Q|$ é a quantidade de caminhos calculados até um dado momento da execução, e a quantidade de caminhos candidatos é sempre menor ou igual a $2 \times |Q| - 1$. Em cada iteração, no máximo, três novos caminhos são gerados e adicionados à lista de caminhos candidatos, sendo que na iteração seguinte, um de menor custo é selecionado.

Seja (V, A, c) um grafo simétrico com uma função custo definida, s e t dois vértices distintos, k um inteiro positivo representando a quantidade de caminhos que se deseja calcular entre s e t. O algoritmo KIM, inicialmente, calcula um caminho de custo mínimo entre s e t usando algum algoritmo que resolva o problema CM (5.2). Em seguida, gera um caminho de custo mínimo que desvia do anterior em algum de seus vértices, utilizando para tal uma subrotina que resolve o problema do desvio mínimo (4.4). A partir dos dois caminhos gerados até o momento, o conjunto $\mathcal{P}_{st}$ é particionado em três: P_a, P_b e P_c. Um caminho de menor custo de cada uma das partições, seu representante, é gerado e adicionado à lista de caminhos candidatos $\mathcal{L}$. A geração de cada representante utiliza como subrotina um algoritmo para o problema do desvio mínimo restrito (5.2). Um caminho de menor custo é retirado da lista de caminhos candidatos tornando-se o P_3. A partir de P_3 e seu pai, geramos, no máximo, mais três caminhos candidatos, cada qual representante de um das partições P_a, P_b e P_c definidas por P_3 e seu pai. Na seção 5.3, as partições serão apresentadas com mais detalhes. O processo continua até que o k-ésimo menor caminho seja retirado da lista $\mathcal{L}$ ou não haja mais caminhos em $\mathcal{L}$.

A seguir apresentamos a rotina principal do algoritmo KIM, de certa forma semelhante a do YEN.

Algoritmo KIM (V, A, c, s, t, k)

1 $P_1 \leftarrow$ *um caminho de custo mínimo de* s *a* t
2 $P_2 \leftarrow$ FSP (V, A, c, P_1, s, t)
2 $\mathcal{L} \leftarrow P_2$
3 $\mathcal{Q} \leftarrow P_1$
4 ***para*** $i = 2, \ldots, k$ ***faça***
5 $\quad P_i \leftarrow$ *caminho de custo mínimo em* $\mathcal{L}$
6 $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{P_i\}$
7 $\quad \mathcal{L} \leftarrow$ ATUALIZE-KIM $(V, A, c, \mathcal{Q}, P, \mathcal{L})$
8 ***devolva*** $\langle P_1, \ldots, P_k \rangle$

Na rotina ATUALIZE-KIM, o conjunto dos caminhos candidatos é incrementado de, no máximo, três caminhos, cada qual representante de uma das partições: P_a, P_b e P_c.

Algoritmo ATUALIZE-KIM $(V, A, c, \mathcal{Q}, P, \mathcal{L})$

1 $P_a \leftarrow$ PA$(V, A, c, \mathcal{Q}, P)$
2 $P_b \leftarrow$ PB$(V, A, c, \mathcal{Q}, P)$
3 $P_c \leftarrow$ PC$(V, A, c, \mathcal{Q}, P)$
4 $\mathcal{L} \leftarrow \mathcal{L} \cup \{P_a, P_b, P_c\}$
5 ***devolva*** $\mathcal{L}$

5.2 Problema do desvio mínimo

Na seção 4.4 o problema do desvio mínimo foi apresentado e no capítulo 4 uma heurística foi proposta. Essa heurística é bastante semelhante a que iremos apresentar, no entanto o faremos de uma maneira um pouco diferente.

Nesta seção mostraremos como o problema do desvio mínimo é resolvido pelo algoritmo KIM. O plano desta seção é apresentar cada um dos elementos que fazem parte da solução do problema DM, pelo algoritmo KIM. Começaremos pelas árvores de menores caminhos, apresentadas no capítulo 4 como arborescências com funções potenciais. Em seguida, mostraremos como caminhos de custos mínimos podem ser

gerados a partir destas árvores. Prosseguiremos explicando um método de rotulação dos nós dessas árvores, o qual será empregado na obtenção de um desvio mínimo. Exibiremos um caso específico em que a solução não funciona. Trataremos brevemente do desvio mínimo restrito, o qual está, de fato, no algoritmo KIM. Finalizaremos, utilizando as árvores rotuladas na solução do problema do desvio mínimo.

Árvores de menores caminhos T_s e T_t

Seja (V, A, c) um grafo simétrico com uma função custo c nas arestas e v um de seus vértices. Uma árvore de menores caminhos com raiz em v é uma arborescência com raiz v formada pelos caminhos de menores custos de v a cada um dos seus vértices acessíveis. Cada vértice acessível possui um potencial que é igual ao custo de um caminho de custo mínimo a partir de v. Na figura a seguir temos um grafo e duas árvores de menores custos, uma com raiz em s e outra em t.

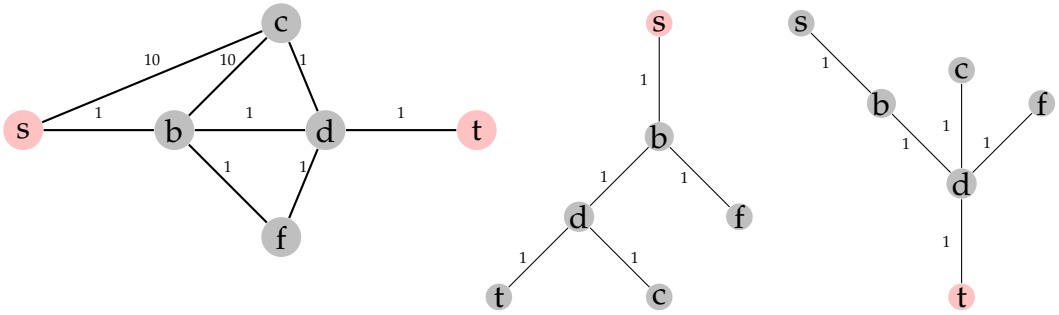


Figura 5.1: Exemplos de árvores de menores caminhos

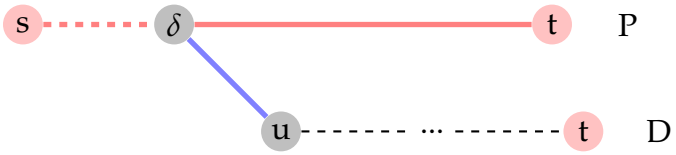
Observe que cada vértice é acessado por um caminho de custo mínimo a partir da raiz. Por exemplo: na árvore com raiz em a o vértice c é acessado pelo caminho $\langle a, b, d, c\rangle$ de custo 3.

Desvios mínimos

Na seção o problema do desvio mínimo foi apresentado. Apenas relembrando, o problema do desvio mínimo consiste em, dado um caminho de custo mínimo, encontrar um caminho diferente, com mesma ponta inicial e com custo mínimo. Na figura a seguir vemos inicialmente um caminho de s a t.

P

Um desvio mínimo de P corresponde a um caminho de custo mínimo entre s a t, diferente de P. na figura a seguir vemos o desvio mínimo D, que corresponde ao caminho que tem o prefixo comum $\langle s, \ldots, \delta \rangle$, tornando-se diferente de P a partir daí.



Na seção anterior vimos as árvores de menores caminhos. Agora iremos utilizá-las na construção de desvios mínimos.

Seja T_s uma árvore de menores caminhos com origem em s, T_t uma árvore de menores caminhos com raiz em t, P um caminho de custo mínimo entre s e t tal que $P \in T_s$ e $P \in T_t$. Suponha que exista um desvio mínimo D de P. Sendo D um desvio mínimo de P, compartilha com este um prefixo maximal: $\langle s, \ldots, \delta \rangle$, para algum vértice $\delta \in P$. Seja u o vértice seguinte a δ no caminho D. Naturalmente $\delta u \notin P$. Sendo assim, temos duas opções para a aresta δu:

- $\delta u \in T_s$
  Neste caso, o caminho D corresponde ao formado pela concatenação do caminho de s a u na árvore T_s ao caminho de u a t na árvore T_s. Observe que $u \in T_s$, pois é acessível a partir de s. Além disso, como $\delta u \in T_s$, temos que o caminho de s a u pertence a T_s. Pela definição de árvores de menores caminhos, o caminho de u a t em T_t é de custo mínimo. Sendo assim, $D = s \underset{T_s}{\longrightarrow} u \underset{T_t}{\longrightarrow} t$.
- $\delta u \notin T_s$
  Neste caso, podemos afirmar que o caminho de s a δ concatenado à aresta δu é um caminho de custo mínimo de s a u, pois sendo D um caminho de custo mínimo, qualquer de seus prefixos deve ser de custo mínimo. Como u é acessível a partir de t e sabendo que o caminho de u a t em T_t é de custo mínimo, temos que: $D = s \underset{T_s}{\longrightarrow} u \underset{\in A}{\rightarrow} v \underset{T_t}{\longrightarrow} t$.

Formalizaremos esses dois casos em dois tipos de caminhos na seção seguinte.

Tipos de caminhos

Seja (V, A, c) um grafo simétrico com uma função custo definida em suas arestas, s e t dois vértices distintos de V, T_s uma árvore de menores caminhos com raiz em s, T_t

uma árvore de menores caminhos com raiz em t, P um caminho de custo mínimo de s a t, P_r o reverso de P, $P \in T_s$ e $P_r \in T_t$. Um desvio mínimo construído através de $u \in V$ é de um dos dois tipos definidos a seguir:

Tipo I **Tipo I** : $s \underset{T_s}{\longrightarrow} u \underset{T_t}{\longrightarrow} t$.

Tipo II **Tipo II** : $s \underset{T_s}{\longrightarrow} u \underset{\in A}{\rightarrow} v \underset{T_t}{\longrightarrow} t$.

Na figura a seguir temos um grafo simétrico, um caminho de custo mínimo de s a t: $\langle s, b, d, t\rangle$, uma árvore de menores caminhos com raiz em s e outra em t, tal que $\langle s, b, d, t\rangle \in T_s$ e $\langle t, d, b, s\rangle \in T_t$.

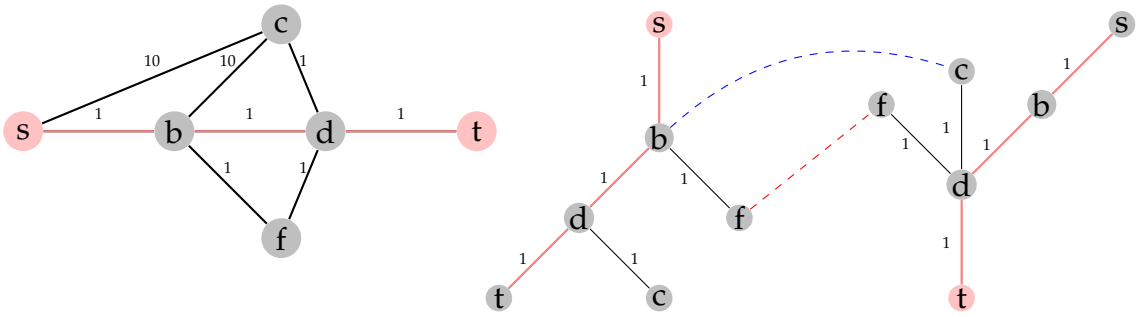


Figura 5.2: Exemplos de caminhos **tipo I** e **tipo II**

Em vermelho, temos o caminho $\langle s, b, f, d, t\rangle$ do **tipo I**: $s \underset{T_s}{\longrightarrow} f \underset{T_t}{\longrightarrow} t$. Em azul, temos caminho $\langle s, b, c, d, t\rangle$ do **tipo II**: $s \underset{T_s}{\longrightarrow} b \underset{\in A}{\rightarrow} c \underset{T_t}{\longrightarrow} t$.
O desvio mínimo de $\langle s, b, d, t\rangle$ corresponde ao caminho de custo mínimo $\langle s, b, c, d, t\rangle$.

Rotulação das árvores

Apresentaremos uma rotulação de árvores de menores caminhos que será utilizada em seguida para geração eficiente de desvios mínimos.

Seja (V, A, c) um grafo simétrico com uma função custo definida em suas arestas, s e t dois vértices distintos de V, T_s uma árvore de menores caminhos com raiz em s, T_t uma árvore de menores caminhos com raiz em t, P um caminho de custo mínimo de s a t, P_r o reverso de P, $P \in T_s$ e $P_r \in T_t$.

Para as rotulações utilizamos a função-predecessor apresentada na seção 2.3.

Rotularemos os vértices de T_s utilizando a **rotulação** ϵ definida por: rotulação ϵ

(1) $\epsilon(s) = 1$

(2) $u \neq s$

- Se $u \in P$ então $\epsilon(u) = \epsilon(\psi_{T_s}(u)) + 1$
- Se $u \notin P$ então $\epsilon(u) = \epsilon(\psi_{T_s}(u))$

Rotularemos os vértices de T_t utilizando a **rotulação** ζ definida por: rotulação ζ

(1) $\zeta(t) = |P|$

(2) $u \neq t$

- Se $u \in P$ então $\zeta(u) = \zeta(\psi_{T_t}(u)) - 1$
- Se $u \notin P$ então $\zeta(u) = \zeta(\psi_{T_t}(u))$

Pelas definições acima podemos perceber que $\forall u \in P$ temos $\epsilon(u) = \zeta(u)$.

Na figura a seguir, temos o caminho destacado $\langle s, b, d, t \rangle$ e as rotulações ϵ e ζ das árvores de menores caminhos T_s e T_t.

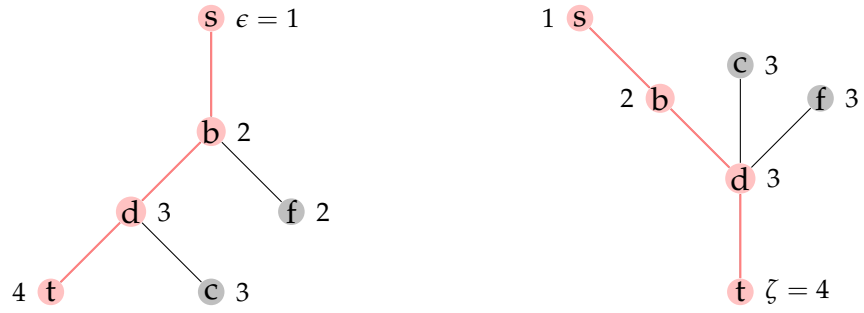


Figura 5.3: Exemplo de rotulações ϵ e ζ

Para gerar as árvores tratadas aqui, é preciso utilizar um versão modificada do algoritmo CM, a qual chamaremos **CM modificada**. As alterações requeridas são:

- rotulação ϵ na árvore T_s e ζ na árvore T_t;
- garantia de que um certo caminho faça parte de ambas as árvores.

Desvios mínimos utilizando rotulações

Agora, temos em mãos os ingredientes necessários para apresentar uma maneira simples e eficiente de gerar desvios mínimos: árvores de menores caminhos, tipos de caminhos e rotulações. Veremos, primeiramente, como é possível determinar que tipo de caminho pode ser gerado a partir de um vértice, utilizando as rotulações das árvores.

Seja (V, A, c) um grafo simétrico com uma função custo definida em suas arestas, s e t dois vértices distintos de V, T_s uma árvore de menores caminhos com raiz em s, T_t uma árvore de menores caminhos com raiz em t, P um caminho de custo mínimo de s a t, P_r o reverso de P, $P \in T_s$ e $P_r \in T_t$.

Para cada vértice u, acessível a partir de s, podemos determinar que tipos de caminhos podem ser gerados através dele, da seguinte maneira:

$\epsilon(u) < \zeta(u)$: O novo caminho será do tipo I.

$\epsilon(u) = \zeta(u)$: Se $u \in P$ então isso sempre ocorre pela própria definição. No entanto, podem existir vértices não pertencentes a P tal que isto também ocorra.

Neste caso, se tentássemos gerar um caminho do tipo I este conteria vértices repetidos, sendo assim um passeio e não um caminho, ou seria o próprio P.

Vamos tratar de dois casos para mostrar que não é possível montar um caminho do tipo I, diferente de P, a partir de u, quando $\epsilon(u) = \zeta(u)$:

$u \in P$: Seja $u \in P = \langle s = v_1, \ldots, u, \ldots, v_n = t \rangle$, $s \xrightarrow[T_s]{} u$ o caminho de s a u em T_s, $P_r\langle t = v_n, \ldots, u, \ldots, v_1 = s \rangle$ o reverso de P em T_t e $t \xrightarrow[T_t]{} u$ o caminho de t a u em T_t. Se concatenarmos os caminhos $s \xrightarrow[T_s]{} u$ e $u \xrightarrow[T_t]{} t$ teremos o caminho: $\langle s = v_1, \ldots, u, \ldots, v_n = t \rangle = P$.

$u \notin P$: Seja $u \notin P$ e escolha em P o vértice c tal que $\epsilon(c) = \epsilon(u)$, ou seja, escolha o vértice de desvio. Vamos montar agora o caminho do tipo I: $s \xrightarrow[T_s]{} u \xrightarrow[T_t]{} t$. Como u se desvia de P em c, temos $s \xrightarrow[T_s]{} u = s \xrightarrow[T_s]{} c \xrightarrow[T_s]{} u$. Como $\epsilon(c) = \zeta(c)$, pois $c \in P$, $\epsilon(u) = \zeta(u)$ e $\epsilon(c) = \epsilon(u)$, podemos afirmar que c se desvia do reverso de P no vértice c logo, o caminho $u \xrightarrow[T_t]{} t$ é igual ao $u \xrightarrow[T_t]{} c \xrightarrow[T_t]{} t$. Sendo assim, montaríamos o passeio: $s \xrightarrow[T_s]{} c \xrightarrow[T_s]{} u \xrightarrow[T_t]{} c \xrightarrow[T_t]{} t$, onde observamos a repetição do vértice c, não sendo, portanto, um caminho.

Podemos gerar caminhos do tipo II, através de arcos $(u,v) \notin T_s \cup T_t$, contanto que $\epsilon(u) < \zeta(v)$. Se $\epsilon(u) \geq \zeta(v)$, geraríamos passeios e não caminhos. Usaremos um argumento parecido ao anteriormente apresentado. Se $u \notin P$ tomaremos o vértice de desvio, ou seja, c onde $\epsilon(u) = \epsilon(c) = \zeta(c)$, se $u \in P$ tomaremos $c = u$. Supondo que $\epsilon(u) \geq \zeta(v)$, como $\epsilon(u) = \epsilon(c)$, podemos escrever $\epsilon(c) \geq \zeta(v)$ ou $\zeta(c) \geq \zeta(v)$. Tomemos agora o vértice de desvio de v, o qual chamaremos de d, tal que $\zeta(v) = \zeta(d) = \epsilon(d)$. Montaremos agora o caminho do tipo II: $s \xrightarrow[T_s]{} u \to v \xrightarrow[T_t]{} t = s \xrightarrow[T_s]{} c \xrightarrow[T_s]{} u \to v \xrightarrow[T_t]{} d \xrightarrow[T_t]{} t$. Como $\zeta(c) \geq \zeta(v)$ e $\zeta(v) = \zeta(d)$ temos que $\zeta(c) \geq \zeta(d)$, assim, podemos reescrever o caminho anterior como: $s \xrightarrow[T_s]{} c \xrightarrow[T_s]{} u \to v \xrightarrow[T_t]{} c \xrightarrow[T_t]{} d \xrightarrow[T_t]{} t$, onde fica clara a repetição do vértice c, caraterizando um passeio e não um caminho.

$\epsilon(u) > \zeta(u)$: Só podemos gerar passeios neste caso. Veremos, na seção 5.2, que isto ocorre apenas quando houver arestas com custos iguais a zero.

Na figura a seguir, vemos um grafo simétrico, onde o caminho $P = \langle s, b, d, t \rangle$ está destacado, as árvores T_s e T_t, tal que P está T_s e o seu reverso está em T_t.

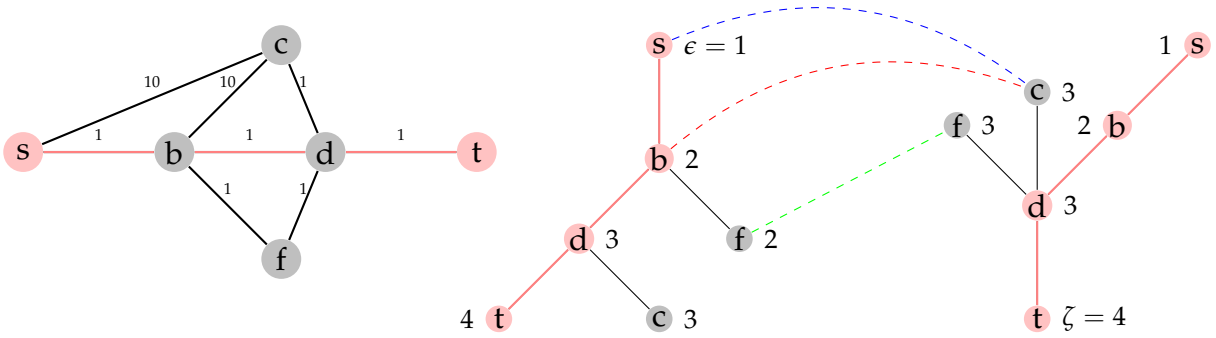


Figura 5.4: Exemplo de desvios utilizando as rotulações ϵ e ζ.

Observamos três desvios:

- usando o vértice f, onde $\epsilon(f) \leq \zeta(f)$, montamos o caminho do tipo I: $s \xrightarrow[T_s]{} u \xrightarrow[T_t]{} t = \langle s, b, f, d, t \rangle$;
- usando o vértice c, onde $\epsilon(c) = \zeta(c)$, montamos os caminhos do tipo II:
  (a) $s \xrightarrow[T_s]{} s \xrightarrow[\in A]{} c \xrightarrow[T_t]{} t = \langle s, c, d, t \rangle$. Observe que $\epsilon(s) < \zeta(c)$;

(b) $s \xrightarrow[T_s]{} b \underset{\in A}{\rightarrow} c \xrightarrow[T_t]{} t = \langle s, b, c, d, t \rangle$. Observe que $\epsilon(b) < \zeta(c)$.

Neste caso, o desvio mínimo corresponde ao caminho $\langle s, b, f, d, t \rangle$.

Custo zero nas arestas

Na seção anterior, dissemos que quando $\epsilon(u) > \zeta(u)$, para algum vértice u, só é possível gerar passeios através dele, situação que pode ocorrer se existir alguma aresta de custo zero no grafo. Veremos um exemplo onde isso ocorre e o que acontece, com o algoritmo de geração de desvio mínimo apresentado, neste caso.

Na figura a seguir, temos um grafo simétrico com um caminho mínimo de s a t destacado e as árvores T_s e T_t correspondentes. Queremos encontra um desvio mínimo de $\langle s, b, d, t \rangle$ utilizando o algoritmo de desvio mínimo utilizando rotulações.

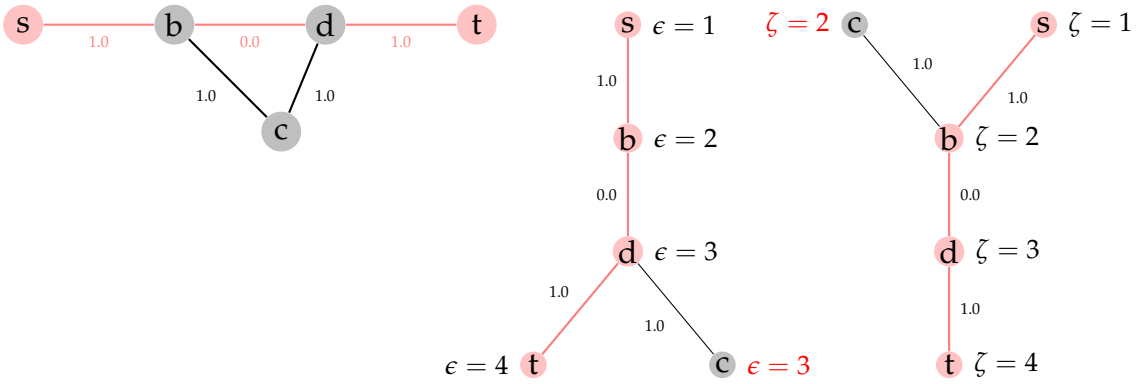


Figura 5.5: Exemplo de falha no cálculo do desvio mínimo no algoritmo KIM

Observe que em T_s o predecessor de c é o vértice d, enquanto que em T_t é o b. A aresta bd ter custo zero torna isso possível. Sendo assim, se tentássemos gerar um desvio de custo mínimo do tipo I através de c acabaríamos com o passeio: $\langle s, b, d, c, b, d, t \rangle$, onde os vértices b e d aparecem repetidos. Não há como gerar um desvio mínimo do tipo II, uma vez que não existe nenhuma aresta do grafo que não esteja nas árvores. Além disso, mesmo que houvesse tais arestas, ainda assim geraríamos um passeio.

Pelo algoritmo apresentado, não há desvio mínimo, no entanto, está claro que o desvio mínimo de $\langle s, b, d, t \rangle$ é $\langle s, b, c, d, t \rangle$.

Problema do desvio mínimo restrito

O problema do desvio mínimo é resolvido no algoritmo KIM através de duas rotinas: FSP e SEP. O algoritmo KIM lida com uma versão mais restrita do problema do desvio mínimo, pois permite que seja informado o vértice antes do qual o desvio deve ocorrer.

DM restrito

Problema DM Restrito$(V, A, c, s, t, P, \alpha)$: Dado um caminho $P = \langle u_1 = s, \ldots, u_n = t\rangle$, um grafo simétrico (V, A, c) e um inteiro $1 \leq \alpha \leq n$, encontrar um desvio mínimo de $\langle s, \ldots, u_\alpha\rangle$.

Se $\alpha = n$, qualquer caminho de custo mínimo diferente de P serve como resposta.

Começaremos tratando da rotina FSP.

A rotina FSP gera duas árvores de menores caminhos rotuladas, T_s e T_t, tal que o caminho P pertence a T_s e o seu reverso pertence a T_t e delega à rotina SEP o trabalho de tentar encontrar um desvio mínimo restrito.

A seguir temos o algoritmo que resume o que foi dito:

Algoritmo FSP $(V, A, c, s, t, P, \alpha)$

1 $T_s \leftarrow$ *árvore de menores caminhos com raiz em* s, *rotulada com os* ϵ *e contendo o caminho* P.

2 $T_t \leftarrow$ *árvore de menores caminhos com raiz em* t, *rotulada com os* ζ *e contendo o caminho reverso de* P.

3 $R \leftarrow \text{SEP}(V, A, T_s, T_t, s, t, \alpha, P)$

4 ***se*** $R = x, x \in V$ ***então devolva*** $s \xrightarrow[T_s]{} x \xrightarrow[T_t]{} t$

5 ***se*** $R = (x, y), (x, y) \in A$ ***então devolva*** $s \xrightarrow[T_s]{} x \underset{\in A}{\rightarrow} y \xrightarrow[T_t]{} t$

6 ***devolva*** R

Nas linhas 1 e 2, as construções das duas árvores correspondem a duas chamadas a um rotina que resolve o CM modificado. Veremos adiante que a rotina SEP, executada na linha 3, consome tempo $\Theta(m + n)$. Nas linhas 4 e 5, a concatenação dos caminhos consome tempo $\Theta(n)$, pois corresponde a passear nas árvores através de, no máximo n vértices. Portanto, o consumo de tempo da função FSP é $\Theta(T(n, m))$.

A rotina SEP é a alma do algoritmo KIM. Ela trabalha testando desvios dos tipos I e II, retornando um de custo mínimo de s a t, que desvia do caminho P em algum vértice u onde $\epsilon(u) \leq \alpha$, analisando as arestas e vértices do grafo (V, A) em profundidade. Veremos que seu consumo de tempo é $O(m)$.

Algoritmo SEP $(V, A, T_s, T_t, s, t, \alpha, P)$

1 *empilha s em S*
2 $C \leftarrow \infty, R \leftarrow \emptyset$
3 ***enquanto*** $S \neq \emptyset$ ***faça***
4 *desempilha u de S*
5 $F_u \leftarrow$ *conjunto de vértices em T_s cujo predecessor seja o vértice u.*
6 $A_u \leftarrow$ *conjunto dos vértices vizinhos do vértice u.*
7 ***se*** $\epsilon(u) = \zeta(u)$ ***então***
8 ***para*** $v \in A_u - F_u$ *e* $\epsilon(u) < \zeta(v)$ ***faça***
9 ***se*** $c(s \underset{T_s}{\longrightarrow} u) + c(u, v) + c(t \underset{T_t}{\longrightarrow} v) < C$ ***então*** *//caminho tipo II*
10 $C \leftarrow c(s \underset{T_s}{\longrightarrow} u) + c(u, v) + c(t \underset{T_t}{\longrightarrow} v)$
11 $R \leftarrow (u, v)$
12 ***se*** $\epsilon(u) < \zeta(u)$ ***então***
13 ***se*** $c(s \underset{T_s}{\longrightarrow} u) + c(t \underset{T_t}{\longrightarrow} u) < C$ ***então*** *//caminho tipo I*
14 $C \leftarrow c(s \underset{T_s}{\longrightarrow} u) + c(t \underset{T_t}{\longrightarrow} u)$
15 $R \leftarrow u$
16 ***para*** $v \in F_u$ ***faça se*** $\epsilon(v) < \alpha$ ***então*** *empilha v em S*
17 ***devolva*** R

O problema do desvio mínimo restrito é, efetivamente resolvido pela rotina SEP. Como vimos anteriormente, dado qualquer vértice pertencente à árvore T_s se existirem desvios mínimos que passem por ele então existe um dentre eles que é do tipo I ou II. A rotina SEP faz justamente o trabalho de tentar gerar esses desvios e retornar o de menor custo. Note que percorrer os vértices de T_s em profundidade diminui o número de vértices que precisam ser testados, pois adicionamos à pilha somente os anteriores aos que possuem $\epsilon = \alpha$. Observe na linha 17 que, apenas os vértices vizinhos ao que

está sendo analisado cujos valores de ϵ sejam inferiores a α, são adicionados à pilha S afinal, queremos apenas caminhos que desviem de P antes do vértice u_α.

Para calcularmos o consumo de tempo da rotina SEP, podemos notar, que o conjunto S armazena vértices pertencentes à T_s, cujos valores de ϵ sejam inferiores a α. Como os vértices são analisados apenas uma vez, podemos afirmar que o laço da linha 3 é executado $O(n)$, já que existem, no máximo n vértices em T_s. A linha 8 contém um laço que percorre uma série de arcos em A. Para cada vértice x sendo analisado, as linhas 9-11 podem ser executadas, no máximo $|A_x|$ onde A_x corresponde às arestas com ponta em x. Considerando-se todos os vértices, as linhas 9-11 podem ser executadas para todos as arestas do grafo. Com base nesta constatação e considerando que as outras linhas internas ao laço da linha 3 consomem tempo $\Theta(1)$, temos que o consumo da rotina SEP é igual a $O(m)$.

5.3 Partições

Nesta seção trataremos das partições definidas em cada iteração do algoritmo KIM. Relembrando o funcionamento do algoritmo, inicialmente calculamos o caminho P_1, menor caminho entre s e t, utilizando um algoritmo CM. Em seguida, utilizando a rotina FSP, obtemos um desvio mínimo de P_1, chamado de P_2. A partir de P_2 e seu pai, P_1, definimos três partições do conjunto de caminhos de s a t:

P_a : caminhos que desviam de P_2 em algum momento depois que P_2 se desviou de P_1. P_a

P_b : caminhos que desviam de P_1 depois do vértice comum a P_1 e P_2. P_b

P_c : caminhos que desviam de P_1 antes do vértice comum a P_1 e P_2. P_c

Cada uma das partições é representada por um caminho de menor custo pertencente a ela. A partir de P_1 e P_2 são gerados, no máximo, três novos caminhos, os representantes das partições descritas acima, os quais serão candidatos para P_3.

Na figura a seguir vemos a disposição esquemática das partições P_a, P_b e P_c.

É possível observar que só existem estas três possibilidades para o caminho P_3. Os representantes das partições: P_a, P_b e P_c, são então colocados na lista de caminhos candidatos e, no início da próxima iteração, um caminho de menor custo é retirado da lista,

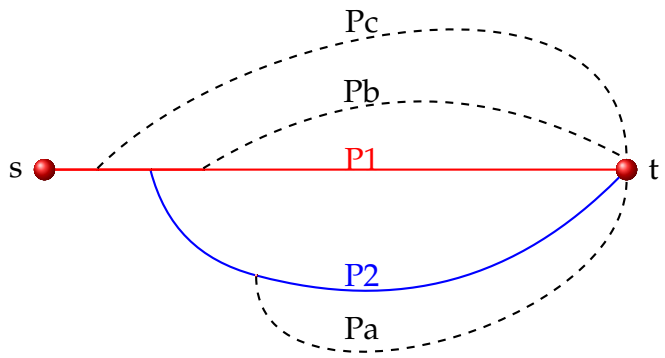


Figura 5.6: Disposição esquemáticas dos representantes das partições P_a, P_b e P_c

tornando-se o P_3. A partir dos caminhos P_3 e seu caminho pai, faremos o mesmo processo, ou seja, geraremos três caminhos candidatos em cada uma das partições:P_a, P_b, P_c, definidas por P_3 e seu pai. O pai de cada caminho é definido da seguinte maneira:

- O pai de P_a é o P_2;
- O pai de P_b é o P_1;
- O pai de P_c é o P_1;
- O pai de P_2 é o P_1.

Uma definição melhor de caminho pai seria: Dada uma seqüência ordenada de caminhos $\langle P_1, P_2, \ldots P_k \rangle$, onde $\forall j \leq i, c(P_j) \leq c(P_i)$, diremos que P_j é pai de P_i, quando $i = min\{x | 1 < x < j,$ tal que P_j e P_i compartilham o maior prefixo comum$\}$.

Seja, por exemplo,
$P = \langle \langle a, b, e \rangle, \langle a, c, e \rangle, \langle a, d, e \rangle, \langle a, b, c, e \rangle \rangle$.
O pai do caminho $\langle a, d, e \rangle$ é o $\langle a, b, e \rangle$. Embora o caminho $\langle a, c, e \rangle$ compartilhe o mesmo prefixo que $\langle a, b, e \rangle$, ele possui um índice maior e, na definição dizemos que o i deve ser o menor índice tal que tenha o maior prefixo.

Agora que vimos como são as partições definidas para os caminhos P_1 e P_2, mostraremos sua definição em função de um $i-$ésimo caminho qualquer.

Antes de prosseguirmos vale lembrar alguns pontos sobre árvore dos prefixos (seção 3.3). Dado um grafo e uma coleção de caminhos com ponta inicial s, existe uma

árvore dos prefixos desses caminhos representada por (N, E, f), onde N é ao conjunto de nós, E é o conjunto de arcos e f é uma função rótulo que relaciona nós é caminhos na árvore a vértices e caminhos no grafo.

Seja u um nó em (N, E, f), então:

- R_u é o caminho, com origem na raiz, com ponta final em u;
- A_u é conjunto de arcos com ponta inicial em u;
- $f(R_u)$ mapeia o caminho R_u na árvore ao caminho correspondente de a a $f(u)$ no grafo.

Partição P_a

Seja (V, A, c) um grafo simétrico com um função custo definida em suas arestas, $\mathcal{Q} = \langle P_1, \ldots, P_i \rangle$, onde $i \geq 2$, o conjunto dos $i-$menores caminhos de s a t, P_j o caminho pai de P_i, δ o vértice de desvio entre P_j e P_i, β o índice de δ em $P_i = \langle s = u_1, \ldots, u_n = t \rangle$, (N, E, f) a árvore dos prefixos de $\mathcal{Q}$ e $f(R_p) = \langle s = u_1, \ldots, u_\beta \rangle$.

A partição P_a, definida por P_j e P_i, corresponde ao conjunto de caminhos de s a t, com prefixo $f(R_q)$ e que sejam diferentes de P_i.

Para gerar caminhos mínimos em P_a, removemos do grafo os vértices do prefixo $f(R_p)$ juntamente com suas arestas, ou seja, removemos os vértices $\langle u_1, \ldots, u_\beta = \delta \rangle$ e as arestas com pontas nestes vértices. Logo após, geramos um desvio mínimo de $u_{\beta+1}$ a $u_n = t$, executando para isso a rotina FSP no grafo alterado, tomando $u_{\beta+1}$ como origem, $u_n = t$ como destino, $P = \langle u_{\beta+1}, \ldots, u_n = t \rangle$ e $\alpha = |P|$, a qual nos retorna um caminho de custo mínimo de $u_{\beta+1}$ a t diferente de P. Concatenando o caminho encontrado ao prefixo $f(R_p)$ obtemos um caminho de custo mínimo na partição P_a.

Todo caminho na partição P_a, definida por P_j e P_i, é filho de P_i.

A seguir, o algoritmo que sintetiza o que foi descrito:

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

0 P *definido por* $\langle u_1, \ldots u_n \rangle$

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

2 $\beta \leftarrow$ *índice do vértice* δ

3 $V' \leftarrow V - \langle u_1, \ldots, u_\beta \rangle$

4 $A' \leftarrow A - \{uv | u \text{ ou } v \in \langle u_1, \ldots, u_\beta \rangle\}$

5 ***devolva*** $f(R_p) \cup$ FSP $(V', A', c, u_{\beta+1}, t, \langle u_{\beta+1}, \ldots, t \rangle, |\langle u_{\beta+1}, \ldots, t \rangle|)$

As alterações no grafo executadas nas linhas 3 e 4 consomem tempo $O(m + n)$. O custo da linha 5 é o mesmo da função FSP, consumo este, dominante na criação do caminho de menor custo na partição P_a. Assim, o consumo de tempo da rotina PA é $\Theta(T(n, m))$.

Na figura a seguir, temos uma árvore dos prefixos de $\mathcal{Q}$, onde apenas exibimos os caminhos P_j e P_i, representados, respectivamente, pelos nós t_j e t_i. Os rótulos externos

representam vértices no grafo, enquanto que os internos, nós na árvore dos prefixos. As linhas tracejadas representam uma seqüência de nós e arcos. As linhas contínuas representam um nó ou arco específico.

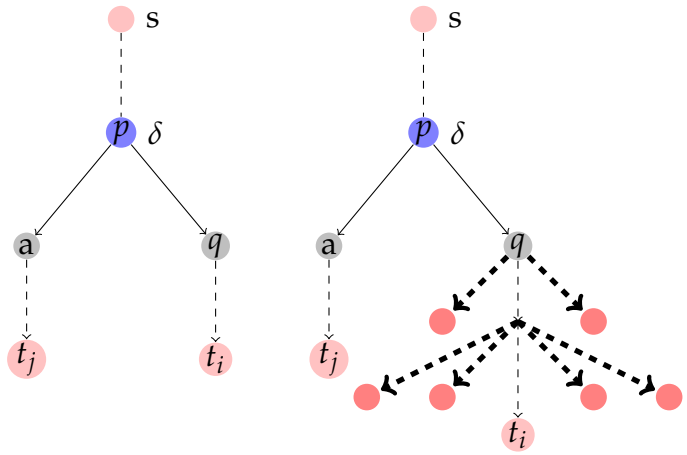


Figura 5.7: Esquema dos caminhos na partição P_a definida por P_j e P_i

O vértice de desvio entre P_j e P_i é o δ. Na figura à direita, vemos os caminhos da partição P_a, definida por P_j e P_i, que correspondem aos caminhos que desviam de P_i após o nó q.

Partição P_b

Seja (V, A, c) um grafo simétrico com um função custo definida em suas arestas, $\mathcal{Q} = \langle P_1, \ldots, P_i \rangle$, onde $i \geq 2$, o conjunto dos $i-$menores caminhos de s a t, P_j o caminho pai de P_i, δ o vértice de desvio entre P_j e P_i, β o índice de δ em $P_j = \langle s = u_1, \ldots, u_n = t \rangle$, (N, E, f) a árvore dos prefixos de $\mathcal{Q}$, $f(R_p) = \langle s = u_1, \ldots, u_\beta \rangle$ e $f(R_q) = \langle u_1, \ldots, u_{\beta+1} \rangle$.

A partição P_b, definida por P_j e P_i, corresponde ao conjunto de caminhos com prefixo $f(R_p)$ que não possuem arcos em $A_p - (p, q)$ e que desviam de P_j antes do vértice u_γ. Para o cálculo de γ, escolha o caminho P_l que compartilha com P_j o menor prefixo $f(R_r)$ tal que $f(R_p) \subset f(R_r)$, e temos $u_\gamma = f(r)$.

Para gerar caminhos nesta partição, removemos do grafo os vértices do prefixo $f(R_p)$, com exceção de $f(p)$ juntamente com suas arestas, ou seja, removemos os vértices $\langle u_1, \ldots, u_{\beta-1} \rangle$ e as arestas com pontas nestes vértices. Além disso, removemos as arestas $f(a)f(b), \forall (a, b) \in A_p$, com exceção da aresta $u_\beta u_{\beta+1}$.

Logo após, executamos a rotina FSP com o novo grafo, δ como origem, $u_n = t$ como destino, $P = \langle \delta, \ldots, u_n = t \rangle$ e $\alpha = |\langle \delta, \ldots, u_\gamma \rangle|$, obtendo então um caminho de custo mínimo de δ a t diferente de P e que desvia deste antes do vértice u_γ. Concatenando o prefixo $f(R_p)$ ao caminho encontrado obtemos o caminho de menor custo na partição P_b.

Todo caminho na partição P_b definida por P_j e P_i é filho de P_j.

A seguir o algoritmo que sintetiza o procedimento descrito:

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

0 $P_j \leftarrow$ *pai de* P

1 P_j *definido por* $\langle u_1, \ldots u_n \rangle$.

2 $(N, E, f) \leftarrow$ *árvore dos prefixos de* $\mathcal{Q}$

3 $\delta \leftarrow$ *vértice de desvio de* P *e* P_j

4 $\beta \leftarrow$ *índice do vértice* δ

5 $V' \leftarrow V - \langle u_1, \ldots, u_{\beta-1} \rangle$

6 $A' \leftarrow A - \{uv | u \text{ ou } v \in \langle u_1, \ldots, u_{\beta-1} \rangle\}$

7 $A' \leftarrow A' - f(a)f(b), \forall (a, b) \in A_p$

8 $A' \leftarrow A' \cup u_\beta u_{\beta+1} \in P_j$

9 $u_\gamma = f(r)$ *sendo* $f(R_r), com f(R_p) \subset f(R_r)$, *o menor prefixo compartilhado por* P_j *e algum caminho de* $\mathcal{Q}$

10 ***devolva*** $f(R_p) \cup$ FSP $(V', A', c, \delta, t, \langle u_\beta, \ldots, u_n = t \rangle, |\langle u_\beta, \ldots, u_\gamma \rangle|)$

Assim como aconteceu no cálculo do consumo de tempo do caminho de custo mínimo na partição P_a, o consumo do algoritmo PB é dominado pela chamada à função FSP, sendo portanto $\Theta(T(n, m))$.

Na figura a seguir temos em (a) uma árvore dos prefixos dos caminhos em $\mathcal{Q}$, onde exibimos os caminhos P_i e P_j representados, respectivamente, pelos caminhos da raiz às folhas t_i e t_j. As folhas em verde representam caminhos filhos de P_j, que desviam deste após o nó δ. Em (b) vemos como é feita a escolha do nó r a partir dos filhos de P_j. Em (c), exibimos, usando folhas vermelhas, os caminhos da partição P_b definida pelos caminhos P_i e P_j, correspondentes aos caminhos com prefixo $f(R_p)$ e que desviam de P_j antes do vértice u_γ.

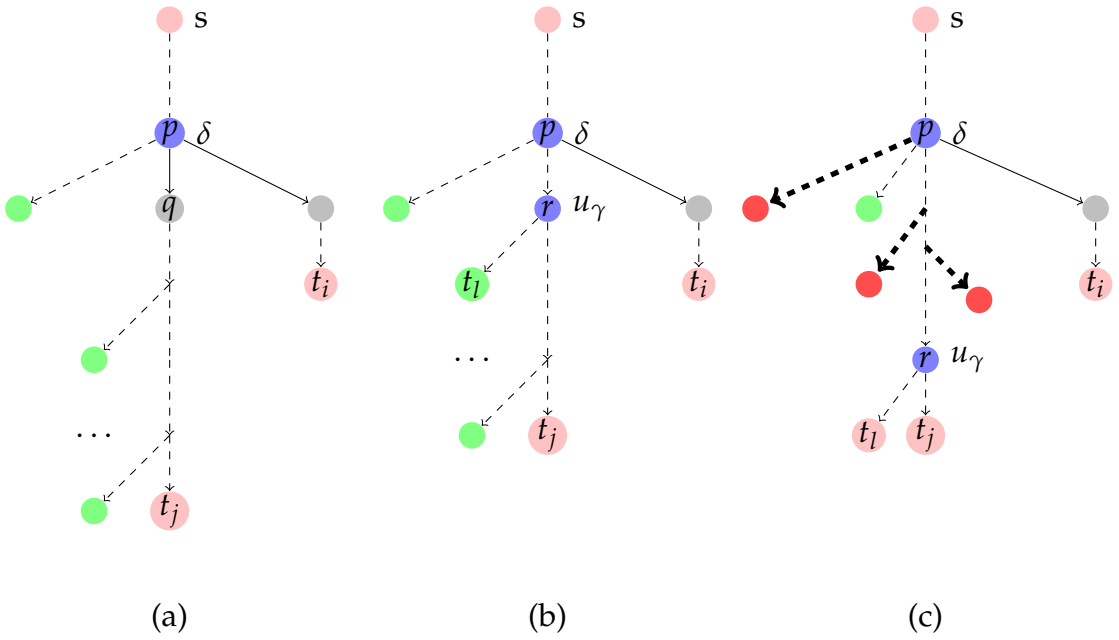


Figura 5.8: Esquema dos caminhos na partição P_b definida por P_j e P_i

Partição P_c

Seja (V, A, c) um grafo simétrico com um função custo definida em suas arestas, $\mathcal{Q} = \langle P_1, \ldots, P_i \rangle$, onde $i \geq 2$, o conjunto dos $i-$menores caminhos de s a t, P_j o caminho pai de P_i, δ o vértice de desvio entre P_j e P_i, β o índice de δ em $P_j = \langle s = u_1, \ldots, u_n = t \rangle$, (N, E, f) a árvore dos prefixos de $\mathcal{Q}$, $f(R_p) = \langle s = u_1, \ldots, u_\beta \rangle$, $f(R_q)$ o maior prefixo de um caminho em $\mathcal{Q}$ tal que $f(R_q) \subset f(R_p)$ e $u_\lambda = f(q)$.

A partição P_c, definida por P_j e P_i, corresponde ao conjunto de caminhos com prefixo $f(R_q)$ que não possuem arcos em A_p e que desviam de P_j antes do vértice $f(p)$.

Para gerar caminhos nesta partição, removemos do grafo os vértices do prefixo $f(R_q)$, com exceção de $f(q)$, juntamente com suas arestas, ou seja, removemos os vértices $\langle u_1, \ldots, u_{\lambda-1} \rangle$ e as arestas com pontas nestes vértices. Além disso, removemos as arestas $f(a)f(b)\ \forall (a, b) \in A_q$.

Executamos então a rotina FSP com o novo grafo, u_λ como origem, t como destino, $P = \langle u_\lambda, \ldots, t \rangle$ e $\alpha = |\langle u_\lambda, \ldots, u_\beta \rangle|$, obtendo então um caminho de custo mínimo de u_λ a t diferente de P e que desvia deste antes do vértice u_β. Concatenando o prefixo $f(R_q)$ ao caminho encontrado obtemos o caminho de menor custo na partição P_c.

Todo caminho na partição P_c definida por P_j e P_i é filho de P_j.

A seguir o algoritmo que sintetiza o procedimento descrito:

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

0 $P_j \leftarrow$ *pai de* P

1 P_j *definido por* $\langle u_1, \ldots u_n \rangle$.

2 $(N, E, f) \leftarrow$ *árvore dos prefixos de* $\mathcal{Q}$

3 $\delta \leftarrow$ *vértice de desvio de* P *e* P_j

4 $\beta \leftarrow$ *índice do vértice* δ

5 $f(R_p) \leftarrow \langle u_1, \ldots, u_\beta \rangle$

6 $u_\lambda = f(q)$ *onde* $f(R_q)$ *é o maior prefixo*
de um caminho em $\mathcal{Q}$ *tal que* $f(R_q) \subset f(R_p)$

7 $V' \leftarrow V - \langle u_1, \ldots, u_{\lambda-1} \rangle$

8 $A' \leftarrow A - \{uv | u \text{ ou } v \in \langle u_1, \ldots, u_{\lambda-1} \rangle\}$

9 $A' \leftarrow A' - f(a)f(b), \forall (a, b) \in A_q$

10 ***devolva*** $f(R_q) \cup$ FSP $(V', A', c, u_\lambda, t, \langle u_\lambda, \ldots, u_n = t \rangle, |\langle u_\lambda, \ldots, u_\beta \rangle|)$

Assim como aconteceu no cálculo do consumo de tempo do caminho de custo mínimo na partição P_b, o consumo do algoritmo PC é dominado pela chamada à função FSP, sendo portanto $\Theta(T(n, m))$.

Na figura a seguir exibimos uma árvore dos prefixo resumida dos caminhos em $\mathcal{Q}$, onde destacamos os caminhos P_j e P_i definidos pelos caminhos com início na raiz e término, respectivamente, nas folhas t_j e t_i.

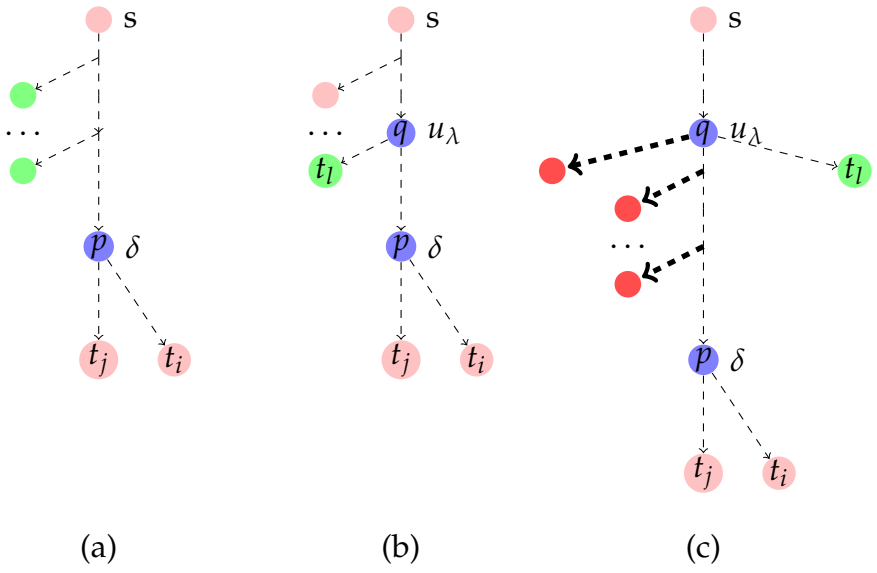
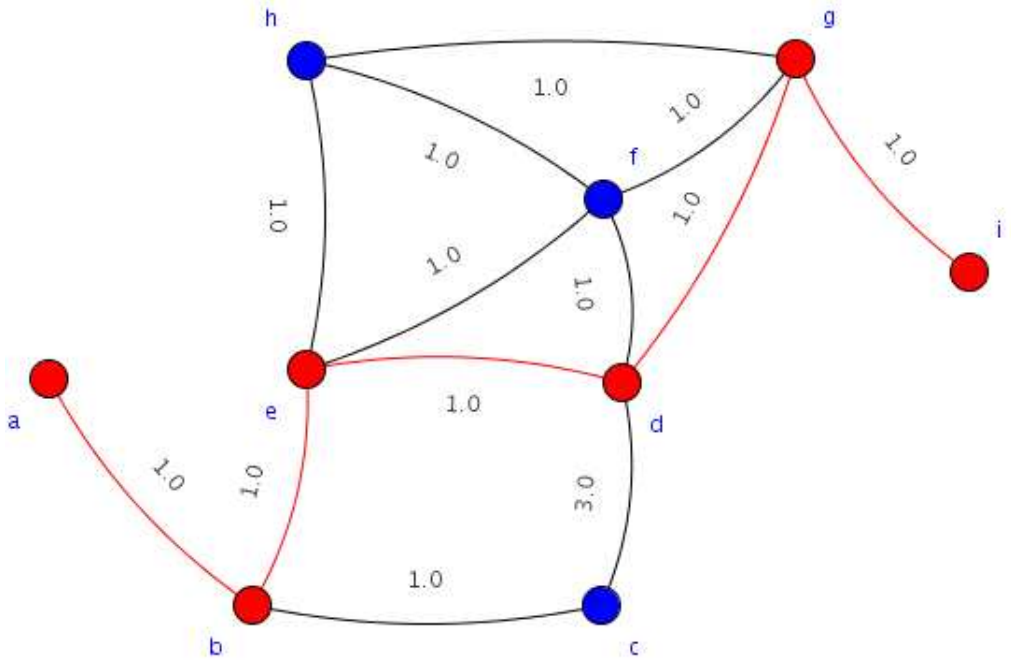


Figura 5.9: Em (a) vemos o nó de desvio entre P_j e P_i, bem como, destacados em verde, os filhos de P_j que desviam antes de δ. Em (b) observamos como é feita a escolha do vértice u_λ com base nos filhos de P_j e do desvio δ. Em (c) temos os caminhos da partição P_c definida por P_j e P_i, que corresponde aos caminhos com prefixo $f(R_q)$ que desviam de P_j entre u_λ e δ.

5.4 Simulação

Simularemos uma execução do algoritmo KIM num grafo simples, exibindo passo a passo as operações realizadas nas obtenções dos primeiros cinco caminhos.

Utilizaremos na nossa simulação o grafo apresentado a seguir, no qual procuraremos caminhos com ponta inicial no vértice a e final em i.



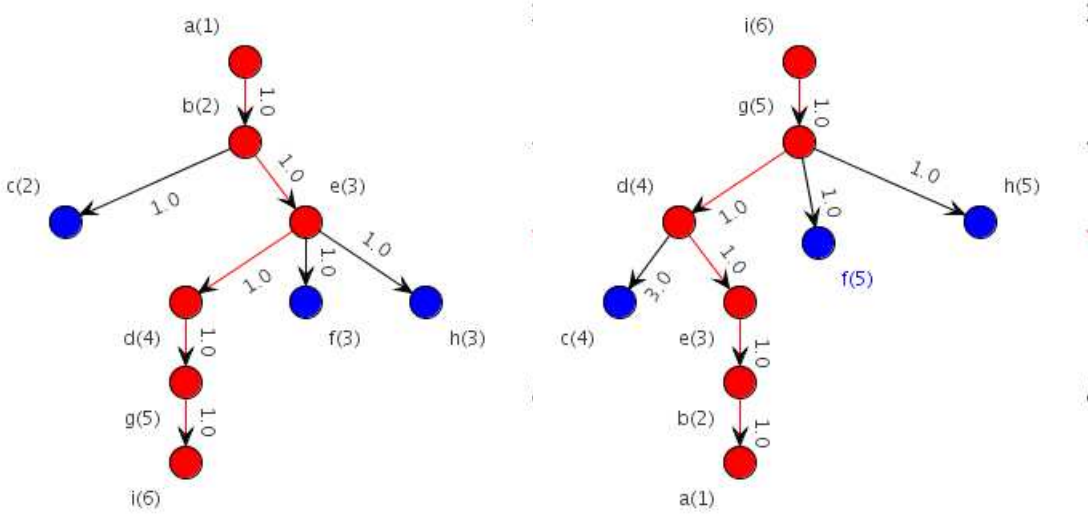
Inicialmente, calculamos o menor caminho de a a i, chamado de P_1, o qual está destacado no grafo acima pelas arestas vermelhas. A obtenção deste caminho resume-se a uma execução do algoritmo de Dijkstra no grafo apresentado.

Em seguida, a partir do caminho P_1, através da rotina FSP calculamos o segundo menor caminho: P_2. O caminho P_2 é o menor caminho de a a i, que desvia do caminho P_1 em algum de seus vértices. A geração de P_2 merece ser mais detalhada.

Primeiramente, são geradas duas árvores de menores caminhos, chamadas respectivamente de T_a e T_i. A árvore T_a corresponde a árvore de menores caminhos enraizada em a, devidamente rotulada com os valores de ϵ e contendo o caminho P_1. A árvore T_i corresponde a árvore de menores caminhos enraizada em i, devidamente rotulada com os valores de ζ e contendo o caminho reverso de P_1: P_{1_r}.

A seguir exibimos as árvores T_a e T_i usadas na geração de P_2: O rótulo de cada vértice está indicado pelo valor entre parênteses. Os números entre parênteses da árvore

à esquerda correspondem aos valores de ϵ enquanto que os da direita aos de ζ.



A partir dessas duas árvores, a rotina SEP, a qual é responsável por obter o menor caminho de a a i diferente de P_1, é chamada com $\alpha = 6$, pois queremos um caminho que desvia de P_1 antes do vértice i, cuja posição no caminho P_1 é 6. A rotina SEP examina os vértices da árvore T_a em profundidade, tentando obter caminhos dos tipos I e II, que desviem de P_1 antes do vértice i, retornando o de menor custo dentre eles.

Inicialmente adicionamos o vértice a à pilha S e definimos o custo atual do menor caminho encontrado até o momento, C, como ∞. O conjunto R armazena um vértice ou um arco[1], dependendo do tipo de caminho gerado. Caso o caminho seja do tipo II então R armazenará um vértice, se for tipo II então armazenará um arco.

Começamos analisando o vértice a, para o qual temos: $F_a = \{b\}, A_a = \{a, b\}$ e $\epsilon(a) = \zeta(a)$, já que pertence ao caminho P_1. Como $\epsilon(a) = \zeta(a)$ tentamos montar um caminho do tipo II, ou seja, um que faça uso de uma aresta não pertencente à árvore T_a. Procuramos todos os vértices vizinhos de a no grafo e que não sejam vizinhos dele em T_a. Neste caso não existe nenhum. Adicionamos o único vizinho de a em $T_a = \{b\}$ à S e procedemos à próxima iteração.

Seguimos analisando o vértice b, para o qual temos: $F_b = \{c, e\}, A_b = \{a, c, e\}$ e $\epsilon(b) = \zeta(b)$, já que pertence ao caminho P_1. Tentamos, novamente, montar um caminho do tipo II e, para isso, procuramos todos os vértices vizinhos de b no grafo que não

[1]OS termos aresta e arco são intercambiáveis e podem ser usados sem prejuízo algum uma vez que estamos trabalhando com grafos simétricos.

sejam vizinhos dele em T_a, ou seja, $A_b - F_b = \{a\}$, mas como $\epsilon(a) < \epsilon(b)$, temos que ignorá-lo. Observe que caso esta última verificação não fosse feita, poderíamos gerar passeios que não fossem caminhos, por possuírem vértices repetidos. Empilhamos os vértices c e e e passamos à próxima iteração.

Desempilhamos o vértice e, onde $F_e = \{d, f, h\}, A_e = \{b, d, f, h\}$ e $\epsilon(e) = \zeta(e)$. Como $F_e - A_e = b$ e $\epsilon(b) < \epsilon(e)$ não temos nenhum vértice para analisar. Empilhamos os vértices d, f e h em S. Observe que o vértice b não é adicionado.

Desempilhamos o vértice h, obtendo $F_h = \emptyset, A_h = \{e, f, g\}$ e $\epsilon(h) \neq \zeta(h)$. Tentamos agora montar um caminho do tipo I, concatenando o caminho $a \xrightarrow[T_a]{} h$ ao $h \xrightarrow[T_i]{} i$, formando um caminho de custo 5. Uma vez que seu custo é menor que o menor custo atual fazemos: $R = h$ e $C = 5$.

Desempilhamos o vértice f, onde $F_h = \emptyset$ e $\epsilon(f) \neq \zeta(f)$. Montamos o caminho $a \xrightarrow[T_a]{} f \xrightarrow[T_i]{} i$, de custo 5. Mantemos o caminho anterior, já que o custo deste novo caminho não é inferior a 5.

Desempilhamos o vértice d, onde $F_d = g, A_d = \{c, e, f, g\}$ e $\epsilon(d) = \zeta(d)$. Testamos apenas o f, uma vez que $\epsilon(c) < \epsilon(d)$ e $\epsilon(e) < \epsilon(d)$, obtendo o caminho do tipo II: $a \xrightarrow[T_a]{} d \rightarrow f \xrightarrow[T_i]{} i$, de custo 6. Empilhamos o vértice g.

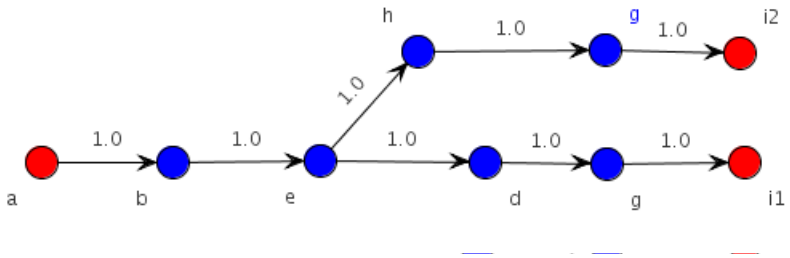
Desempilhamos o vértice g, onde $F_g = i, A_g = \{d, f, h, i\}$ e $\epsilon(g) = \zeta(g)$.

Desempilhamos o vértice c, onde $F_c = \emptyset, A_c = \{b, d\}$ e $\epsilon(c) \neq \zeta(c)$. Montamos o caminho do tipo I: $a \xrightarrow[T_a]{} c \xrightarrow[T_i]{} i$, de custo 7.

A rotina SEP é concluída com o caminho: $a \xrightarrow[T_a]{} h \xrightarrow[T_i]{} i$, cujo custo é 5.

Agora que acabamos de mostrar como o caminho P_2 é construído, vamos tratar da geração dos próximos caminhos e exibir a estrutura dos caminhos P_a, P_b e P_c.

Até o momento temos a seguinte árvore de menores caminhos, formada pelos caminhos P_1 e P_2:

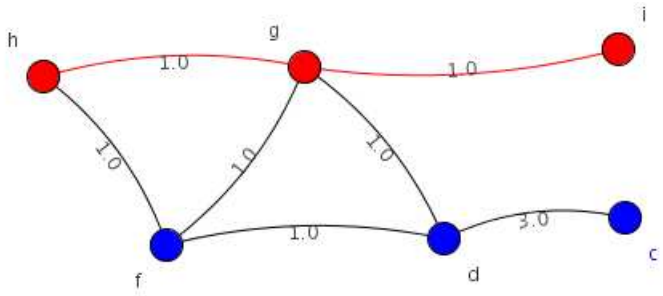


Onde o caminho P_1 é o que termina no nó i_1 e o caminho P_2 no i_2. O último vértice comum aos caminhos P_1 e P_2 é o e, a partir do qual os caminhos se diferenciam. O vértice e desempenhará um papel chave da geração dos caminhos derivados de P_1 e P_2 e será chamado vértice de desvio.

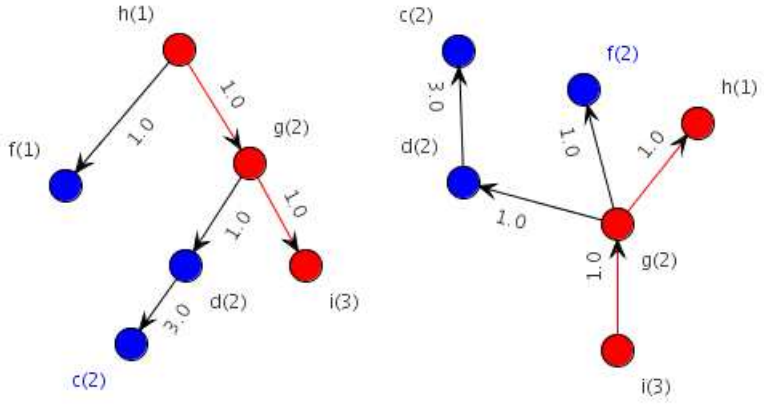
Começaremos pela geração do caminho P_a, o menor dentre todos os caminhos que desviam de P_2 após o último vértice comum a P_1 e P_2.

Para a geração de P_a necessitamos do último caminho gerado e de seu caminho gerador (pai), que neste caso são, respectivamente, os caminhos: P_2 e P_1.

A idéia é gerar o menor caminho de h a i e concatená-lo ao prefixo comum aos caminhos P_1 e $P_2 = \langle a, b, e \rangle$.
Para evitar a geração de caminhos repetidos precisamos remover do grafo os vértices do prefixo comum e suas respectivas arestas: $\{(a, b), (b, e)\}$, obtendo o grafo:



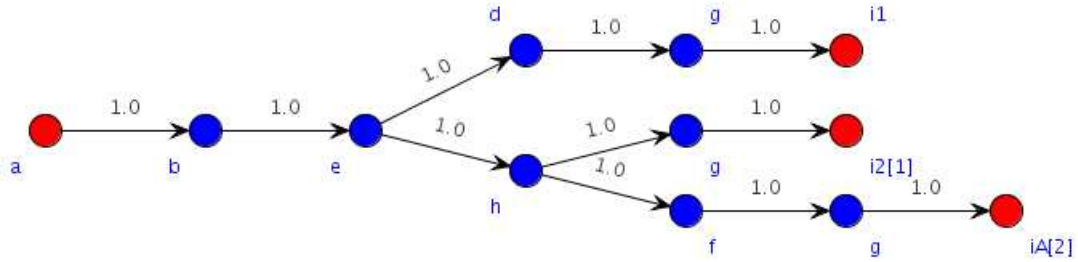
A partir do novo grafo, oriundo dessas remoções, executamos uma chamada à função FSP, utilizando h como origem, i como destino, o caminho base $P = \langle h, g, i \rangle$ e $\alpha = 3$, obtendo as seguintes árvores T_h e T_i:



Por fim, a rotina SEP é chamada retornando o caminho do tipo I: $h \xrightarrow[T_h]{} f \xrightarrow[T_i]{} i = \langle h, f, g, i \rangle$, de custo 3, o qual é concatenado ao prefixo $\langle a, b, e \rangle$ formando o caminho

$P_a = \langle a, b, e, h, f, g, i \rangle$, de custo 6.

Na figura a seguir, temos os caminhos P_1, P_2 e P_a, com pontas iniciais a e pontas finais, respectivamente : i_1, i_2 e i_A, onde fica fácil observar que o caminho P_a desvia de P_2 após o vértice e. O caminho P_a é filho do caminho P_2 o que está indicado pelo número entre colchetes no rótulo da sua ponta final.

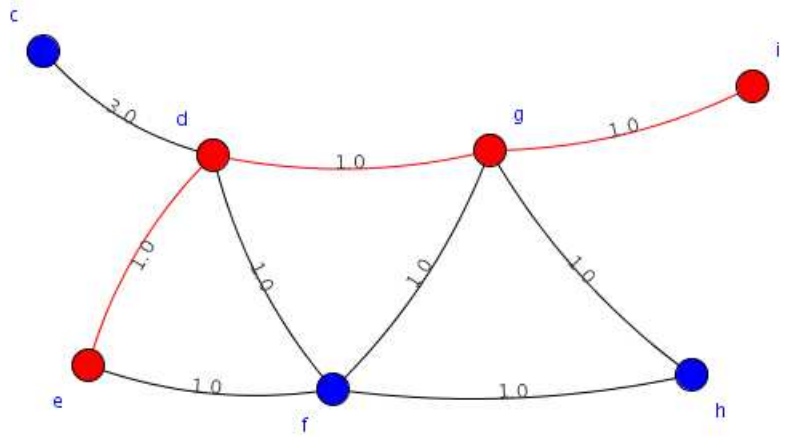


Vamos agora tratar do caminho P_b, o menor caminho que desvia do caminho P_1 em algum ponto entre o vértice de desvio e e o i. Para evitar a geração de caminhos repetidos precisamos excluir do grafo:

- Os vértices pertencentes ao prefixo comum aos caminhos P_1 e P_2 (excetuando-se o vértice de desvio), bem como suas respectivas arestas.
  Neste caso, os vértices a serem removidos são:$\{a, b\}$ e as arestas (a, b) e (b, e).

- As arestas com pontas nos vértices de desvio dos caminhos que têm P_1 como pai.
  No momento, o único caminho gerado a partir de P_1 é o P_2.
  Observando-se a árvore formada pelos menores caminhos encontrados até o momento (P_1 e P_2), temos que aresta a ser removida é a (e, h), pois tem ponta no vértice de desvio e do caminho P_2 o qual tem P_1 como pai.

Após estas alterações, obtemos o novo grafo, o qual possui o caminho $\langle e, d, g, i \rangle$ destacado:



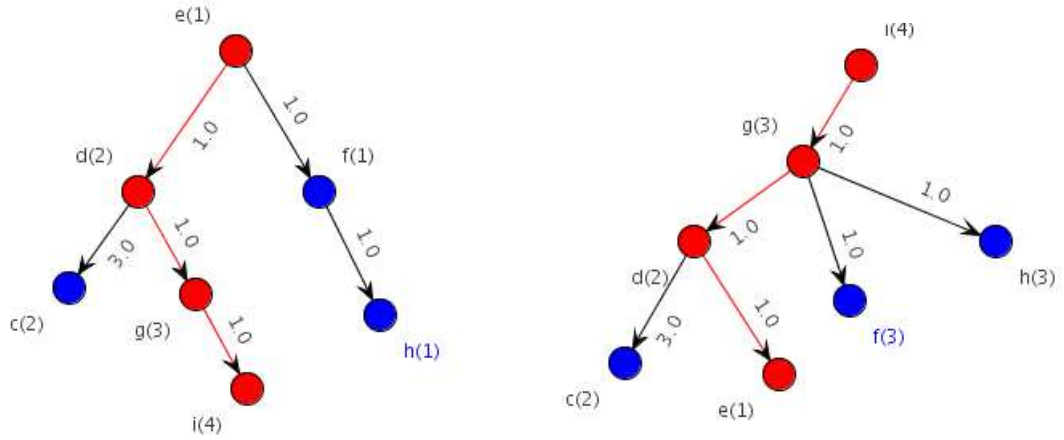
Agora precisamos informar à rotina FSP o valor de α, para que ela saiba antes de qual vértice do caminho $\langle e, d, g, i\rangle$ o novo caminho tem que desviar. Para a determinação do α faremos o seguinte:

1. Tomaremos todos os caminhos gerados a partir de P_1 e seus respectivos vértices de desvio.
   No momento o único caminho gerado a partir de P_1 é o P_2, cujo nó de desvio é o e;

2. Escolhemos o caminho que compartilha o menor prefixo com P_2, e que contenha o prefixo $\langle a, b, e\rangle$. Ao vértice de desvio deste caminho daremos o nome de γ. Caso γ exista $\alpha = |\langle e, \ldots, \gamma\rangle|$. Caso não exista $\alpha = |\langle e, \ldots, i\rangle|$

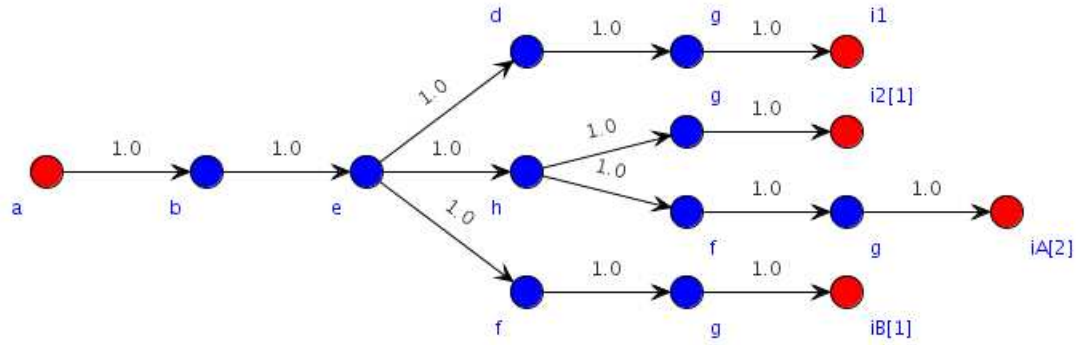
No nosso caso, γ não existe, logo $\alpha = |\langle e, d, g, i\rangle| = 4$.

Executamos a rotina FSP no novo grafo, utilizando o vértice e como ponta inicial, i como final, e $\alpha = 4$, obtendo as árvores de menores caminhos T_e e T_i exibidas na figura a seguir:



Com as árvores em mãos, a rotina SEP retorna o menor caminho do tipo I: $e \xrightarrow[T_e]{} f \xrightarrow[T_i]{} i = \langle e, f, g, i \rangle$.
Concatenando-o com o prefixo do caminho P_1: $\langle a, b \rangle$, obtemos o caminho $\langle a, b, e, f, g, i \rangle$, de custo 5.

A figura a seguir mostra a disposição do caminho P_b calculado, o qual é representado pelo caminho com ponta inicial em a e final em iB:



Entre colchetes vemos o número 1, indicando que seu pai é o caminho P_1.

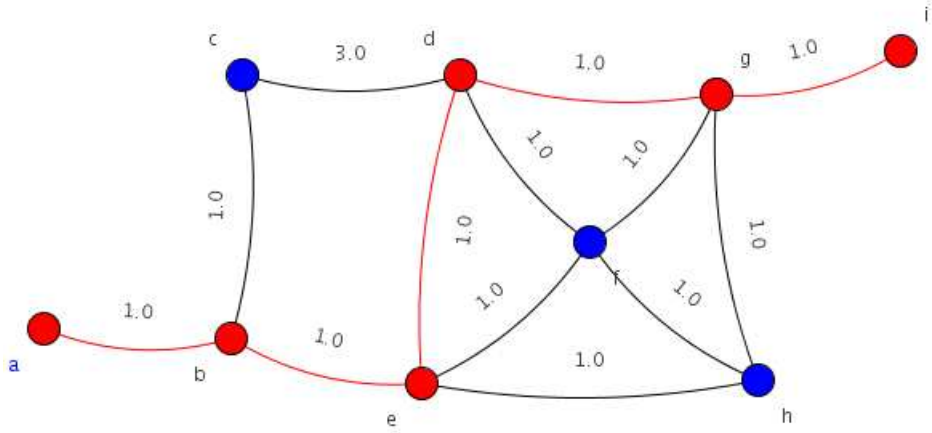
Vamos agora tratar do caminho P_c, o menor caminho que desvia do caminho P_1 em algum ponto antes do vértice de desvio e.

Dados todos os caminhos filhos de P_1, escolhemos aquele compartilha com P_1 o maior prefixo e se desvia antes do vértice de desvio e. Chamaremos de δ o vértice de desvio deste caminho com seu pai P_1. Caso não exista, δ será igual à ponta inicial de P_1. O α que será passado à função FSP e SEP, corresponde ao número de vértices do subcaminho de P_1 com ponta inicial em δ e ponta final em e.

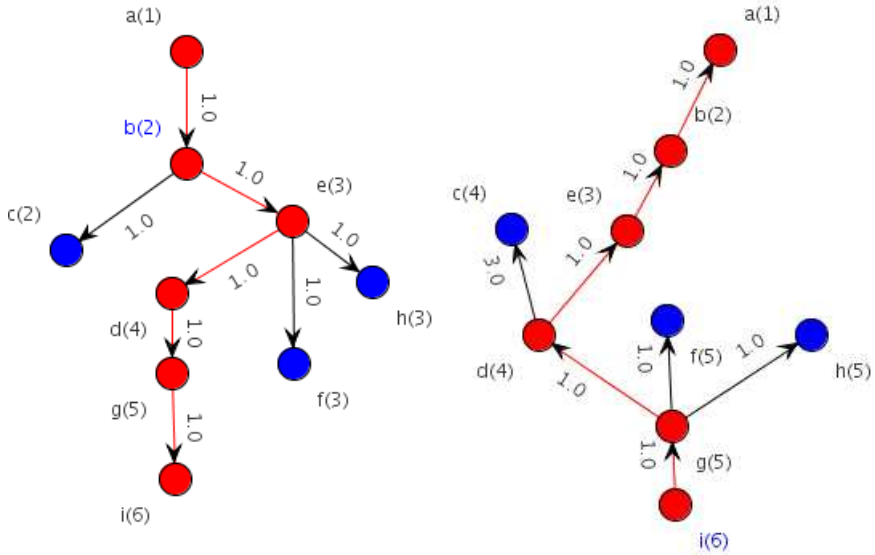
Neste momento, P_2 é o único filho de P_1 e se desvia de P_1 no vértice e, logo $\delta = a$

e $\alpha = |\langle a, b, e \rangle| = 3$. A fim de evitar a geração de caminhos repetidos, removemos do grafo todas as arestas com pontas no vértice δ pertencentes aos caminhos filhos de P_1. Neste caso, não há nenhuma aresta a ser removida

O grafo para cálculo de P_c é o próprio grafo original, onde o caminho P_1 está destacado:



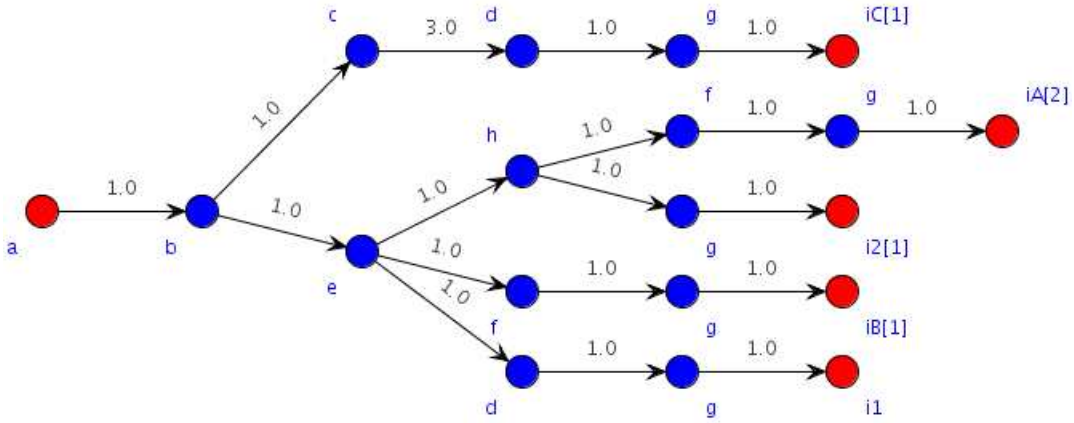
Executamos a rotina FSP, com ponta inicial a, ponta final i e $\alpha = 3$, no grafo obtido, a qual nos retorna as seguintes árvores T_a e T_i:



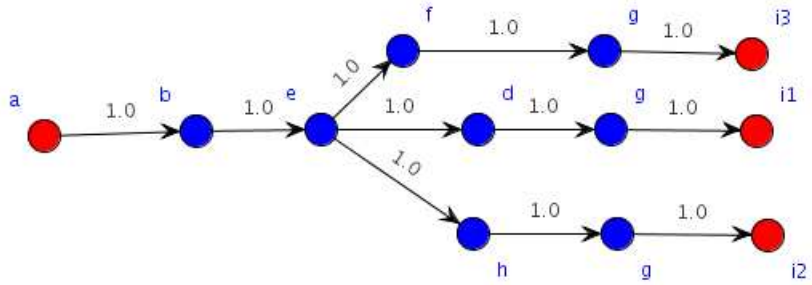
Finalizamos chamando a rotina SEP obtendo o caminho do tipo I: $a \xrightarrow[T_a]{} c \xrightarrow[T_i]{} i = \langle a, b, c, d, g, i \rangle$ de custo 7. Observe que, como passamos $\alpha = 3$, a rotina SEP só testou caminhos que desviavam antes do vértice e. Nenhum vértice com $\epsilon > 3$ foi adicionado à pilha S. Agora é possível perceber a vantagem de se analisar os vértices da árvore em profundidade. Se tivéssemos analisado os vértice de qualquer maneira, teríamos

tentado gerar caminhos para muito mais vértices. O uso da pilha diminui isto, apenas os vértices *a*,*b* e *c* foram analisados.

A figura a seguir exibe os caminhos gerados até o momento:



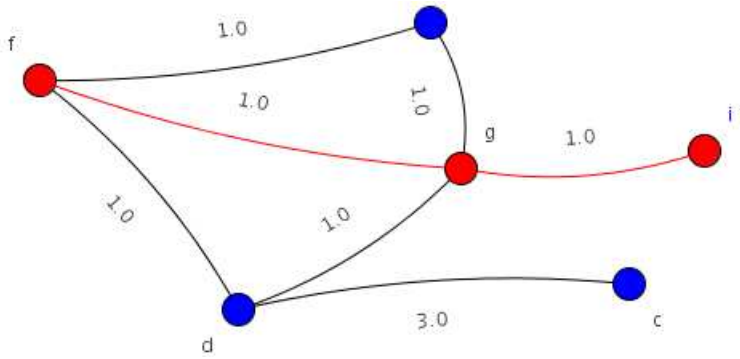
Neste momento estamos com os caminhos candidatos: P_a, P_b e P_c e os menores caminhos: P_1 e P_2. Retiramos o caminho de menor custo da lista de candidatos, neste caso P_b, e o adicionamos à lista de caminhos mais curtos, obtendo a seguinte árvore de caminhos mais curtos:



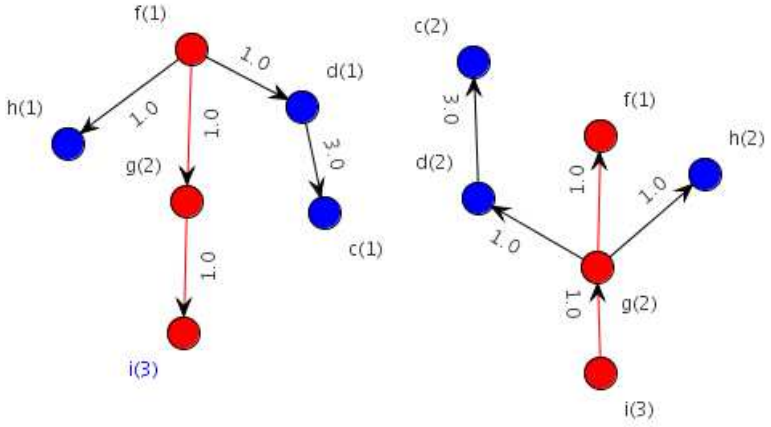
Esses três caminhos formarão a base para a geração do 4º caminho mais curto.

Tomaremos o caminho P_3, cujo pai é P_1 e geraremos os caminhos P_a, P_b e P_c, correspondentes. O vértice de desvio desses dois caminhos é o *e*.

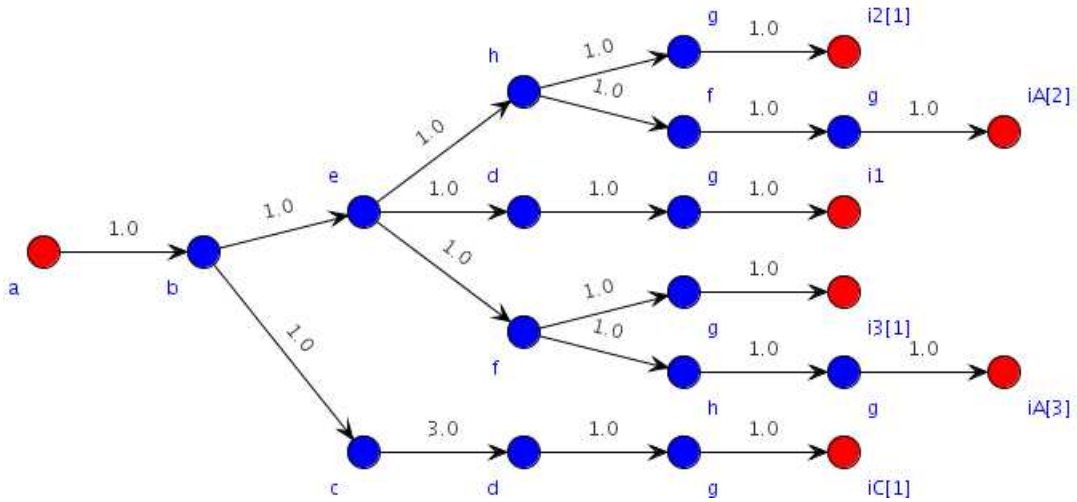
A geração de P_a é relativamente simples.
Retiramos do grafo os vértices do prefixo comum a P_1 e P_3, ou seja, $\{a, b, e\}$, obtendo o grafo:



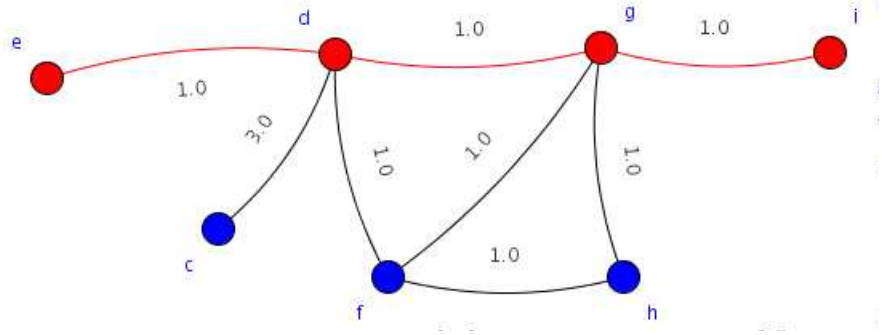
Executamos a função FSP com $\alpha = 3$ para nos retornar o menor caminho de f a i diferente de $\langle f, g, i \rangle$, a qual nos retorna o caminho $\langle f, h, g, i \rangle$ através da execução do algoritmo SEP nas árvores:



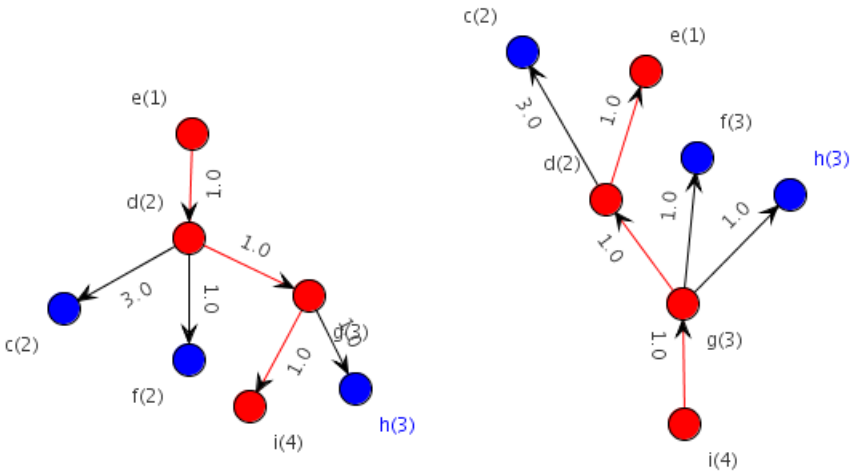
Concatenamos o prefixo $\langle a, b, e \rangle$ ao caminho $\langle f, h, g, i \rangle$ obtendo o caminho $\langle a, b, e, f, h, g, i \rangle$, exibido na figura a seguir:



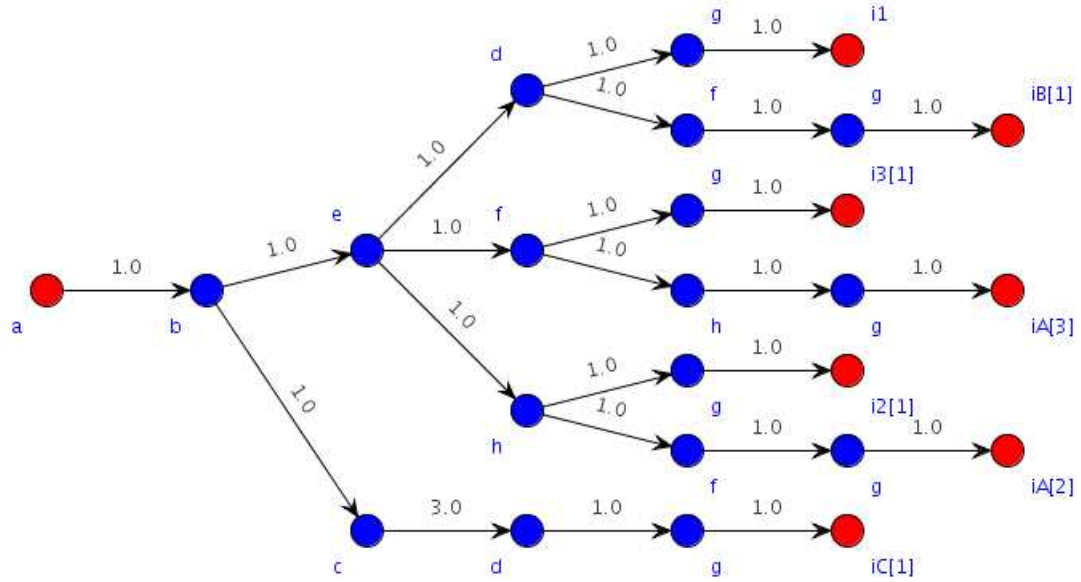
A geração de P_b requer o cálculo de um caminho que desvia de P_1 depois do vértice de desvio e e seja diferente de todos os filhos de P_1. Para isso, primeiro removemos os vértices do prefixo comum a P_1 e $P_3 = \langle a, b, e \rangle$ com exceção do vértice e. Depois, para evitar a geração de caminhos repetidos, apagamos as arestas: (e, h) e (e, f), obtendo o grafo:



Em seguida executamos rotina FSP, com $s = e$, $t = i$ e $\alpha = 4$, uma vez que não existe nenhum caminho filho de P_1 que desvia deste num vértice posterior a e. Todos os caminhos gerados a partir de P_1, até o momento, ou se desviam exatamente no vértice e ou antes dele. Obtemos então as seguintes árvores T_e e T_i:

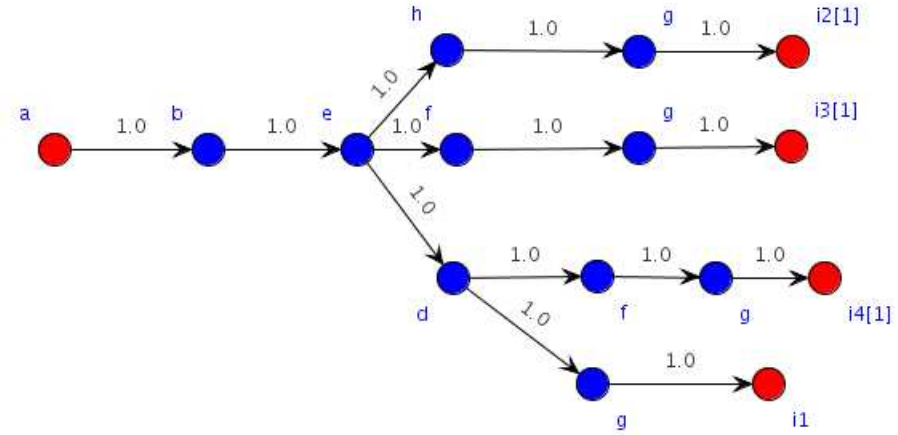


Por fim, a rotina SEP nos devolve o caminho do tipo I: $e \xrightarrow[T_e]{} f \xrightarrow[T_i]{} i = \langle e, d, f, g, i \rangle$, que concatenado ao $\langle a, b, e \rangle$ forma o caminho $\langle a, b, e, d, f, g, i \rangle$ de custo 7, exibido a seguir.



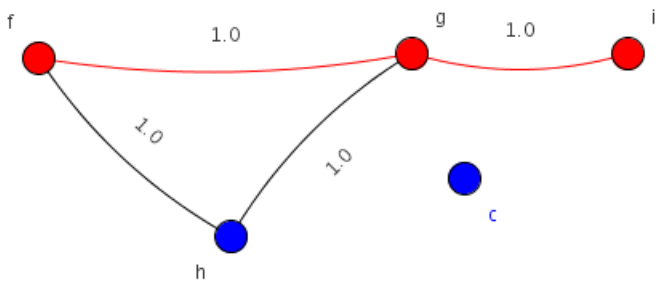
Não é possível gerar nenhum caminho P_c com base nos caminhos P_1 e P_3, pois o menor caminho filho de P_1 que desvia antes do vértice de desvio e já foi calculado nas iterações anteriores.

Retiramos da lista de candidatos o caminho de menor custo, ou seja, $\langle a, b, e, d, f, g, i \rangle$ e o inserimos na lista de menores caminhos. A árvore dos menores caminhos, contendo os quatro menores caminhos se torna então:

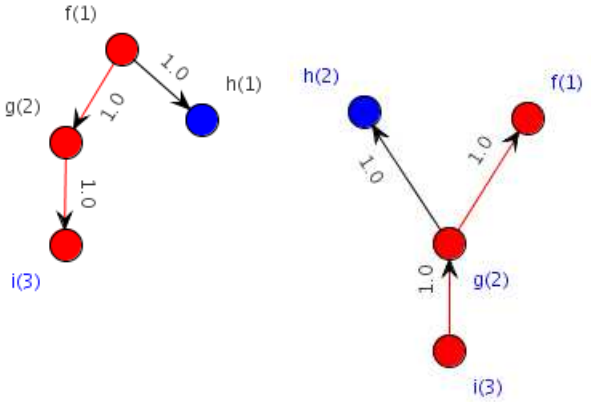


Para a geração do 5º caminho mais curto de a a i tomamos os caminhos: $P_4 = \langle a, b, e, d, f, g, i \rangle$ e seu caminho gerador: P_1.

Começamos gerando um caminho na partição P_a. O vértice de desvio entre P_1 e P_4 é o d. Removemos do grafo os vértices do prefixo comum aos dois caminhos, bem como suas arestas correspondentes, obtendo o grafo:

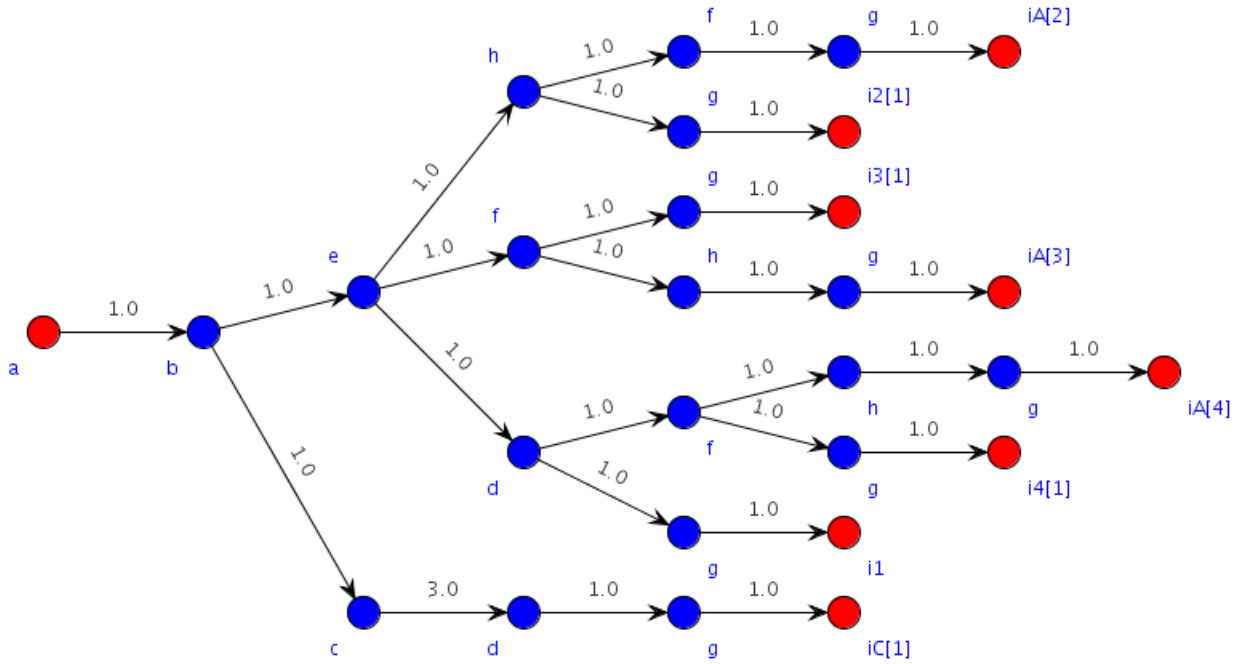


A partir deste grafo resolvemos o problema do desvio de custo mínimo, através da execução da rotina SEP nas árvores de menores caminhos T_f e T_i e o caminho base $\langle f, g, i \rangle$:

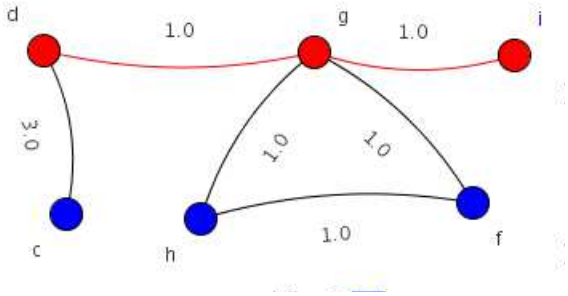


A rotina SEP nos retorna o desvio de custo mínimo gerado a partir do vértice h: $\langle f, h, g, i \rangle$, um caminho do tipo I: $f \xrightarrow[T_f]{} h \xrightarrow[T_i]{} i$.

O novo caminho na partição P_a é obtido concatenando-se o prefixo $\langle a, b, e, d \rangle$ ao caminho $\langle f, h, g, i \rangle$, obtendo-se o caminho: $\langle a, b, e, d, f, h, g, i \rangle$, exibido a seguir:

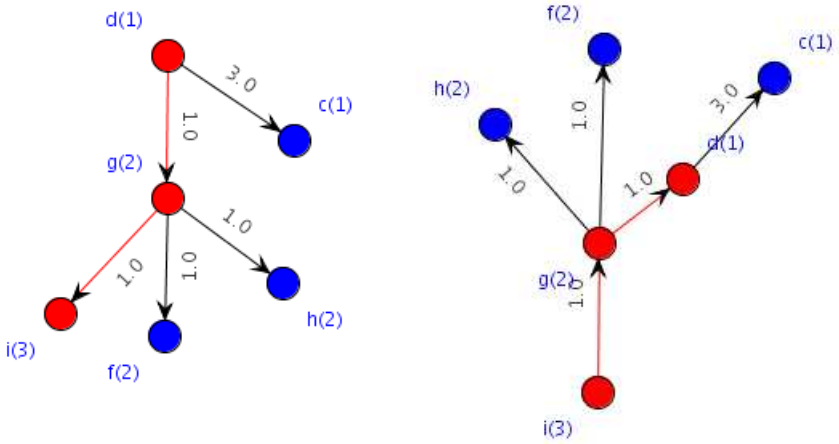


Vamos tratar agora da geração do novo caminho na partição P_b. Retiramos do grafo os vértices do prefixo comum, com exceção do vértice de desvio, juntamente com suas arestas, ou seja, apagamos os vértices: $\{a, b, e\}$ e as arestas:
$\{(a,b),(b,c),(b,e),(e,h),(e,f),(e,d)\}$. Queremos então obter um caminho na partição P_b usando o desvio d e gerar um que desvia de P_1 a partir deste vértice e que seja diferente. Para evitar a geração de caminhos repetidos, é preciso remover algumas arestas. Vamos considerar todos os caminhos gerados a partir de P_1, com exceção de P_4, que compartilham com este o menor prefixo tal que seja maior que o prefixo compartilhado entre P_1 e P_4, ou seja, queremos o caminho com menor prefixo tal este contenha o prefixo $\langle a, b, e, d\rangle$. Não há nenhum caminho nestas condições. Com isto, basta remover a aresta (d, f) do grafo, evitando assim que o caminho P_4 venha ser novamente gerado. A seguir o grafo com as alterações mencionadas:



Pelo grafo é possível perceber que não existe outro caminho de d a i diferente de

$\langle d, g, i \rangle$. No entanto, vamos exibir as árvores T_d e T_i:

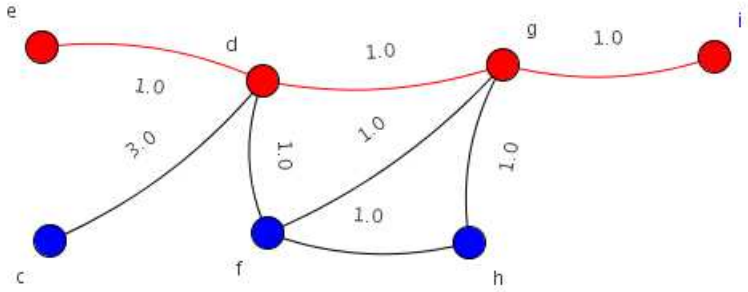


Vale notar que os vértices $\{c, f, h\}$ possuem $\epsilon = \zeta$, sem no entanto fazerem parte do caminho base $\langle d, g, i \rangle$. Com tais árvores o problema do desvio de custo mínimo não tem solução.

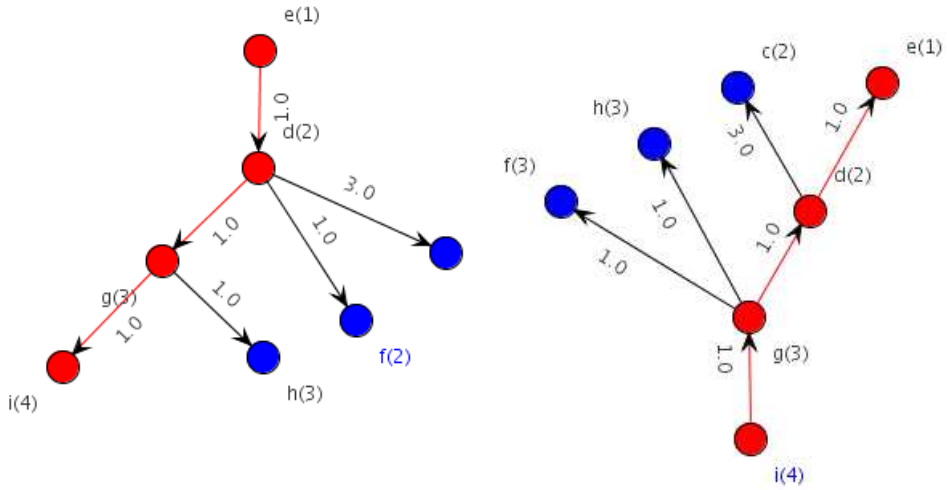
Vamos passar para o caminho na partição P_c. O representante da partição P_c é um caminho de custo mínimo que desvia de P_1 antes do vértice de desvio d e que não seja igual a nenhum dos filhos de P_1.

Olhando a árvore de caminhos contendo os quatro primeiros caminhos observamos que os gerados a partir de P_1 são: P_2, P_3 e P_4, cujos vértices de desvio são: e, e e d. Precisamos então gerar o caminho com ponta em e e que desvia de P_1 antes de d. A escolha da ponta e é como segue: escolha dos caminhos gerados a partir de P_1 aquele que com este compartilhar o maior prefixo tal que seja menor que o prefixo comum a P_1 e P_4 e selecione o vértice de desvio deste caminho. No nosso caso, o caminho escolhido é o P_2 cujo vértice de desvio é o e.

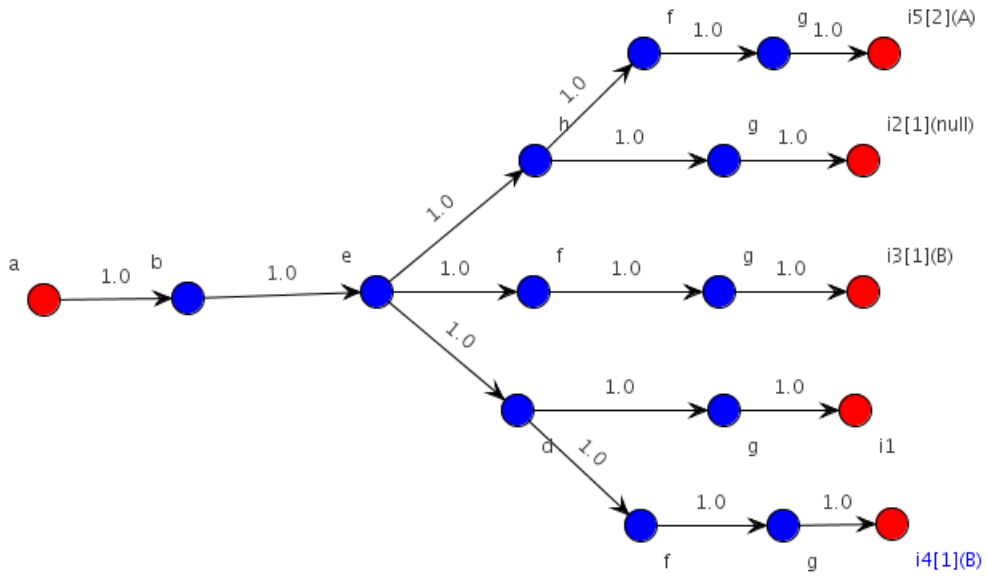
Excluímos do grafo as arestas com ponta no vértice de desvio e e que pertençam aos filhos de P_1, ou seja, excluímos as arestas $\{(e, h), (e, f)\}$. Além disso, é preciso remover os vértices do prefixo $\langle a, b \rangle$ juntamente com suas arestas. O grafo oriundo destas alterações é:



Geramos as árvores T_e e T_i, com o caminho base $\langle e, d, g, i\rangle$ e pedimos à função SEP que nos retorne um desvio de custo mínimo tal que o desvio ocorra antes do vértice d. Não existe tal caminho.

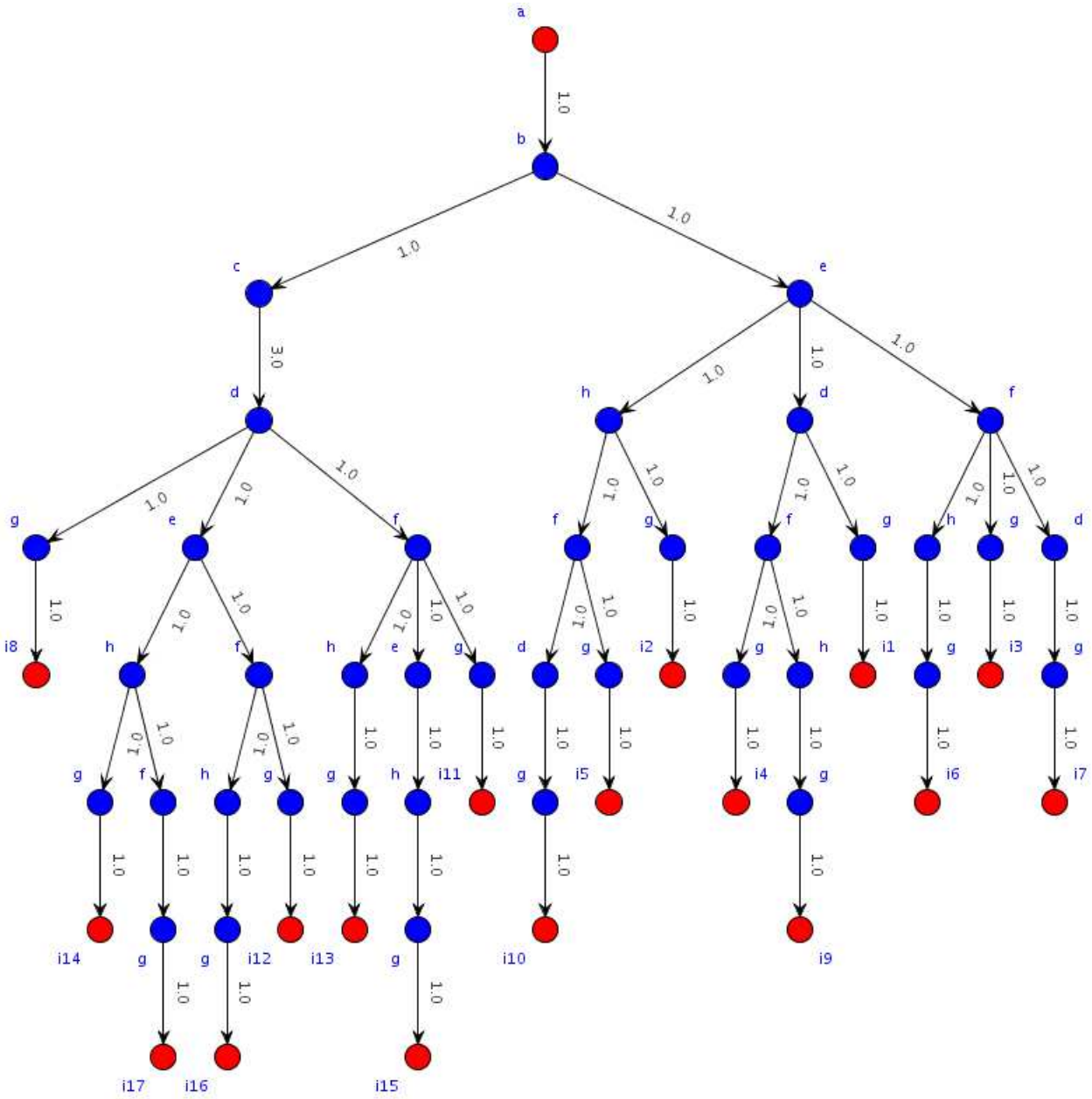


Retiramos da lista de candidatos um caminho de custo mínimo, $\langle a, b, e, h, f, g, i\rangle$ e o adicionamos à lista de menores caminhos que se torna:



O índice após o i corresponde à posição do caminho na lista de menores caminhos. Por exemplo: $i1$ corresponde ao caminho P_1. O número entre colchetes se refere ao índice do pai do caminho. Por exemplo: $i3[1]$ indica que o caminho P_3 foi gerado a partir do P_1. A letra entre parênteses indica a partição que este caminho representa. Por exemplo: $i3[1](B)$ indica que o caminho P_3 é o representante da partição P_b definida pelos caminhos P_1 e P_3.

Todos os 17 caminhos gerados:



5.5 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. Não exibiremos todo o código nesta seção, mas apenas pequenos trechos, com o intuito de explicar como foi feita a implementação dos pontos que consideramos mais importantes.

Como já dissemos, o algoritmo KIM depende de uma rotina capaz de gerar uma árvore de menores caminhos, ou seja, uma rotina que resolva o problema CM, com algumas pequenas mudanças, de forma a garantir a presença de um certo caminho, além das rotulações ϵ e ζ.

Atualmente a biblioteca JUNG possui apenas uma rotina que resolve o problema CM, utilizando o algoritmo de Dijkstra, implementado usando uma fila de prioridade baseada num min-heap. Optamos então por utilizá-la.

De forma a garantir a presença de um certo caminho $P = \langle u_1 = s, \ldots, u_n = t \rangle$ na árvore de menores caminhos gerada pelo Dijkstra, permitimos que vértices que fazem parte do caminho sejam apenas alcançados por outros que também façam parte do caminho e, além disso, sejam imediatamente anteriores a ele. Lembrando do algoritmo de Dijkstra, apresentado na seção 2.5, sempre que um arco está sendo examinado, ou seu vértice incidente está em Q ou não. Se não estiver, deve ser inserido com o custo correspondente. Se já estiver, e o novo custo para alcançá-lo for inferior ao anteriormente calculado, devemos atualizar o seu custo. A ação de inserir um vértice em Q está mapeada na função createRecord(V w, E e, **double** new_dist) apresentada no código a seguir. Observe que, caso o vértice, do arco incidente atualmente sendo examinado, não esteja em P nada de especial precisa ser feito, mas caso contrário precisamos garantir que seu predecessor seja um vértice pertencente à P e, imediatamente anterior a este em P. Para testar que está no caminho basta verificar os retornos das chamadas da função isOnPath, nas linha 5 e 6. A fim de testar que seja imediatamente anterior, subtraímos os valores das rotulações entre os dois vértices em questão. Caso estejamos gerando a árvore de menores caminhos T_s, utilizamos a rotulação ϵ, senão utilizamos a ζ.

```
1 protected class SourcePathDataKIM extends SourcePathData {
2   @Override
```

```
3   public void createRecord(V w, E e, double new_dist) {
4     V pred = ((UndirectedGraph<V, E>) g).getOpposite(w, e);
5     if (w.isOnPath()) {
6       if (!pred.isOnPath())
7         return;
8       if (getTreeType().equals(ShortestPathKIM.TreeType.TS)) {
9         if (w.getEpsilon().intValue()-pred.getEpsilon().intValue() != 1)
10          return;
11      } else if (getTreeType().equals(ShortestPathKIM.TreeType.TT)) {
12        if (w.getZeta().intValue()-pred.getZeta().intValue() != -1)
13          return;
14      }
15    }
16    super.createRecord(w, e, new_dist);
17  }
18 }
```

Ainda temos o problema das rotulações para resolver. Vale lembrar que:

Sejam T_s e T_t duas árvores de menores caminhos com raízes s e t, respectivamente, geradas a partir de execuções do CM no grafo simétrico com custo (V, A, c) e $P = \langle s, \ldots, t \rangle$ o caminho base em T_s então:

$$\forall u \in V, u \in T_s, u \in P \rightarrow \epsilon(u) = \zeta(u) = |prefixo(\langle s, \ldots, u \rangle)|$$

$$\forall u \in V, u \in T_s, u \notin P \rightarrow \epsilon(u) = \epsilon(\psi_{T_s}(u))$$

$$\forall u \in V, u \in T_t, u \notin P \rightarrow \zeta(u) = \zeta(\psi_{T_t}(u)$$

Sendo assim, definimos a função de rotulação como exibida no código a seguir.

```
1   private void setLabel(V w, V pred) {
2     if (!w.isOnPath()) {
3       if (getTreeType().equals(ShortestPathKIM.TreeType.TS))
4         w.setEpsilon(pred.getEpsilon());
5       else if (getTreeType().equals(ShortestPathKIM.TreeType.TT))
6         w.setZeta(pred.getZeta());
7     }
```

Observe que rotulamos apenas os vértices não pertencentes ao caminho P. A rotulação dos vértices em P é feita em um momento anterior ao da execução do algoritmo CM modificado. Antes de todas as chamadas a função FSP, executamos a rotina insertPathOnTree, apresentada a seguir.

```
1 protected void insertPathOnTree(Path<KIMVertex, KIMEdge> path, int ini) {
```

```
2   for (int i = ini, epsilon = 1; i < path.getVertices().size(); i++,
        epsilon++) {
3     KIMVertex atual = path.getVertex(i);
4     atual.setIsOnPath(true);
5     atual.setEpsilon(epsilon);
6     atual.setZeta(epsilon);
7   }
8 }
```

O seu código é bem simples e nada mais faz que aplicar a definição das rotulações ϵ e ζ aos vértices pertencentes ao caminho P.

Agora precisamos definir quando as rotulações dos vértices não pertencentes à P devem ser feitas. Observamos que sempre que um vértice é adicionado a S podemos, seguramente, rotulá-lo, uma vez que seu predecessor não mais mudará.

```
1 @Override
2   public Entry<V, Number> getNextVertex() {
3     Map.Entry<V, Number> p = super.getNextVertex();
4     V v = p.getKey();
5     E incomingEdge = incomingEdges.get(v);
6     if (incomingEdge != null) {
7       V pred = ((UndirectedGraph<V, E>) g).getOpposite(v, incomingEdges.get
            (v));
8       setLabel(v, pred);
9       sons.get(pred).add(v);
10    }
11    sons.put(v, new LinkedHashSet<V>());
12    return p;
13  }
```

O código completo da classe que implementa as estruturas de dados utilizadas no algoritmo CM modificado é apresentada a seguir:

```
1   protected class SourcePathDataKIM extends SourcePathData {
2     @Override
3     public Entry<V, Number> getNextVertex() {
4       Map.Entry<V, Number> p = super.getNextVertex();
5       V v = p.getKey();
6       E incomingEdge = incomingEdges.get(v);
7       if (incomingEdge != null) {
8         V pred = ((UndirectedGraph<V, E>) g).getOpposite(v, incomingEdges.
              get(v));
9         setLabel(v, pred);
```

```
10          sons.get(pred).add(v);
11        }
12        sons.put(v, new LinkedHashSet<V>());
13        return p;
14      }
15
16      @Override
17      public void createRecord(V w, E e, double new_dist) {
18        V pred = ((UndirectedGraph<V, E>) g)
19            .getOpposite(w, e);
20        if (w.isOnPath()) {
21          if (!pred.isOnPath())
22            return;
23          if (getTreeType().equals(ShortestPathKIM.TreeType.TS)) {
24            if (w.getEpsilon().intValue()-pred.getEpsilon().intValue() != 1)
25              return;
26          } else if (getTreeType().equals(ShortestPathKIM.TreeType.TT)) {
27            if (w.getZeta().intValue()-pred.getZeta().intValue() != -1)
28              return;
29          }
30        }
31        super.createRecord(w, e, new_dist);
32      }
33
34      private void setLabel(V w, V pred) {
35        if (!w.isOnPath()) {
36          if (getTreeType().equals(ShortestPathKIM.TreeType.TS))
37            w.setEpsilon(pred.getEpsilon());
38          else if (getTreeType().equals(ShortestPathKIM.TreeType.TT))
39            w.setZeta(pred.getZeta());
40        }
41      }
42
43      public SourcePathDataKIM(V source) {
44        super(source);
45      }
46    }
```

Achamos importante tratar de um detalhe da função SEP, à qual implementa o algoritmo SEP apresentado anteriormente. Na linha 5 do algoritmo temos a definição de F_u como o conjunto de vértices em T_s cujo predecessor seja o vértice u. O problema é que temos em mãos apenas a árvore de menores caminhos, na forma de um grafo

de predecessores. Para obter o conjunto F_u, a partir da árvore de menores caminhos, usamos o código a seguir, o qual dada uma árvore de menores caminhos T, um vértice u e um vértice s tal que $\psi_T(s) = \emptyset$, retorna os filhos de u na árvore T com raiz s:

```
1  protected Set<KIMVertex> getSons(ShortestPathKIM<KIMVertex, KIMEdge> T,
2                          KIMVertex u, KIMVertex s) {
3      Iterator<KIMEdge> i = graph.getOutEdges(u).iterator();
4      Set<KIMVertex> sons = new HashSet<KIMVertex>();
5      while (i.hasNext()) {
6        KIMEdge atual = i.next();
7        KIMVertex o = graph.getOpposite(u, atual);
8        KIMEdge incident = T.getIncomingEdge(s, o);
9        if (incident != null  && graph.getOpposite(o, incident).equals(u))
10         sons.add(o);
11     }
12     return sons;
13   }
```

O código anterior exige que todas as arestas de u sejam testadas quanto a sua existência na árvore T, ocasionando um consumo de tempo $O(m)$. Descobrimos, posteriormente, que poderíamos armazenar, para cada vértice u, seu conjunto F_u durante a execução do algoritmo CM modificado. Observamos que sempre que um novo vértice é adicionado ao conjunto S, uma vez que seu predecessor não mais mudará, podemos adicioná-lo à lista de filhos de seu predecessor. O código a seguir resume o que foi dito.

```
1      public Entry<V, Number> getNextVertex() {
2        Map.Entry<V, Number> p = super.getNextVertex();
3        V v = p.getKey();
4        E incomingEdge = incomingEdges.get(v);
5        if (incomingEdge != null) {
6          V pred = ((UndirectedGraph<V, E>) g).getOpposite(v, incomingEdges.
               get(v));
7          setLabel(v, pred);
8          sons.get(pred).add(v);
9        }
10       sons.put(v, new LinkedHashSet<V>());
11       return p;
12     }
```

Na linha 8 obtemos a lista de filhos do vértice predecessor e lhe adicionamos o novo vértice v. Na linha 10 criamos a lista de filhos do vértice v. Aplicando esta mudança conseguimos ganhos de desempenho de até 40%. Notamos que quanto maior for a densidade de um grafo tanto mais tempo gastaremos executando a rotina getSons.

Decidimos implementar uma solução para o CM modificado quando a função custo é constante. Quando a função custo é constante, uma simples busca em largura é suficiente para retornar as árvores de menores caminhos a partir da origem. O JUNG possuía uma implementação para grafos sem custos, no entanto, uma vez que o código e as estruturas estavam juntas, não conseguimos reaproveitá-lo, pois precisávamos alterá-lo com as rotulações e a presença do caminho base.

Utilizando busca em largura, garantir que um certo caminho base faça parte da árvore gerada pode ser realizado forçando cada vértice do caminho a ser analisado apenas pelo vértice anterior a ele no caminho. Observa que isso é feito nas linhas 16 e 17 quando impedimos que o vértice neighbor entre para $\mathcal{S}$ caso ele faça parte do caminho base e o vértice cur não.

Veja que nas linhas 25 a 32 as rotulações ϵ e ζ são feitas de acordo com a definição apresentada anteriormente. Mais uma vez, apenas rotulamos, na implementação do algoritmo CM, vértices que não façam parte do caminho. Os vértices que fazem parte do caminho são rotulados num momento anterior.

```
1  private void bfs(KIMVertex source) {
2    LinkedList<KIMVertex> unknown = new LinkedList<KIMVertex>();
3    distanceMap = new HashMap<KIMVertex, Number>();
4    incomingEdgeMap = new HashMap<KIMVertex, KIMEdge>();
5    unknown.add(source);
6    distanceMap.put(source, 0);
7    while (!unknown.isEmpty()) {
8      KIMVertex cur = unknown.pollFirst();
9      Iterator<KIMEdge> i = g.getIncidentEdges(cur)
10         .iterator();
11     while (i.hasNext()) {
12       KIMEdge incomingEdge = i.next();
13       KIMVertex neighbor = g.getOpposite(cur,
14           incomingEdge);
15       if (!distanceMap.containsKey(neighbor)
16           && !(neighbor.isOnPath() && !cur
17               .isOnPath())) {
18         unknown.addLast(neighbor);
19         incomingEdgeMap.put(neighbor,
20             incomingEdge);
21         distanceMap
22             .put(neighbor,
23                 (Integer) distanceMap
24                     .get(cur) + 1);
```

```
25              if (!neighbor.isOnPath()
26                  && treeType != null) {
27                if (treeType.equals(ShortestPathKIM.TreeType.TS))
28                  neighbor.setEpsilon(cur
29                      .getEpsilon());
30                else if (treeType
31                    .equals(ShortestPathKIM.TreeType.TT))
32                  neighbor.setZeta(cur.getZeta());
33              }
34            }
35          }
36        }
37    }
```

Tendo as duas implementações para o CM modificado: uma baseada no algoritmo de Dijkstra e utilizando grafos com custo não-negativos e outra baseada numa simples busca em largura e lidando com grafos com custos constantes, era preciso fornecer uma maneira de utilizá-los de uma maneira uniforme pelo algoritmo KIM. Para isso, inicialmente definimos a interface ShortestPathKIM:

```
1  public interface ShortestPathKIM<V extends KIMVertex, E extends KIMEdge>
2      extends ShortestPath<V, E>, Distance<V> {
3    public static final Class BFSShortestPathKIMAlgorithm = BFSKIM.class;
4    public static final Class DijkstraShortestPathKIMAlgorithm =
       DijkstraSPKIM.class;
5
6    public List<E> getPath(V source, V target);
7
8    public E getIncomingEdge(V source, V target);
9
10   public LinkedHashSet<V> getSons(V source, V vertex);
11 }
```

Assim, toda classe utilizada para resolver o CM modificado durante a execução do algoritmo KIM deveria implementar os métodos definidos por essa interface. Além disso, criamos uma fábrica de instâncias do CM modificado, para facilitar nas suas criações.

```
1  public class Factory {
2      Transformer<KIMEdge, Number> nev;
3      UndirectedGraph<KIMVertex, KIMEdge> g;
4      Class shortestPathKIM;
5      public Constructor construtor;
```

```
6
7       public Factory(Transformer<KIMEdge, Number> nev,
8           UndirectedGraph<KIMVertex, KIMEdge> g,
9           Class<ShortestPathKIM<KIMVertex, KIMEdge>> shortestPathKIM) {
10        this.nev = nev;
11        this.g = g;
12        this.shortestPathKIM = shortestPathKIM;
13        try {
14          construtor = shortestPathKIM.getConstructor(new Class[] {
15              UndirectedGraph.class, Transformer.class,
16              ShortestPathKIM.TreeType.class });
17        } catch (Exception e) {
18          RuntimeException err = new RuntimeException();
19          err.initCause(e);
20          throw err;
21        }
22      }
23
24      public ShortestPathKIM<KIMVertex, KIMEdge> newInstance(
25          ShortestPathKIM.TreeType treeType) {
26        try {
27          return (ShortestPathKIM<KIMVertex, KIMEdge>) construtor
28              .newInstance(new Object[] { g, nev, treeType });
29        } catch (Exception e) {
30          RuntimeException err = new RuntimeException();
31          err.initCause(e);
32          throw err;
33        }
34      }
35    }
```

O construtor da classe KIM recebe uma fábrica de instâncias do CM modificado e uma função custo na forma de um *Transformer* conseguindo resolver o problema k-MC sem se preocupar com a implementação do CM modificado informada.

Precisamos criar classe específicas para vértices e arestas, uma vez que desejávamos armazenar informações específicas relativas ao algoritmo KIM. Nossa classe KIMVertex possui dois campos que armazenam os valores das rotulações ϵ e ζ, uma variável **boolean** chamada isOnPath usada para indicar se o vértice faz parte do caminho base ou não e, para depuração do código, adicionamos um campo nome. Para facilitar nas leituras de grafos a partir de arquivos *PajekNet* adicionamos à classe uma fábrica padrão de vértices.

```
1 public class KIMVertex {
2   public static class KIMVertexFactory implements Factory<KIMVertex> {
3     int id = 1;
4     @Override
5     public KIMVertex create() {
6       return new KIMVertex(Integer.toString(id++));
7     }
8   }
9   public static Factory<KIMVertex> getFactory() {
10    return new KIMVertex.KIMVertexFactory();
11  }
12  private Integer epsilon;
13  private boolean isOnPath = false;
14  private String nome;
15  private Integer zeta;
16
17  public KIMVertex(String nome) {
18    this(nome, false);
19  }
20  public KIMVertex(String nome, boolean isOnPath) {
21    this.nome = nome; this.isOnPath = isOnPath;
22  }
23  @Override
24  public boolean equals(Object obj) {
25    if (obj == null)
26      return false;
27    try {
28      KIMVertex other = (KIMVertex) obj;
29      return getNome().equals(other.getNome());
30    } catch (ClassCastException e) { return false;}
31  }
32 }
```

Os métodos get e set para cada campo foram suprimidos na listagem anterior, por conveniência.

A nossa classe KIMEdge possui um campo para armazenar um custo, uma fábrica padrão de arestas e duas funções custos mapeadas nos Transformers: defaultCost e constantCost . A seguir o código da classe KIMEdge:

```
1 public class KIMEdge {
2   @Override
3   public boolean equals(Object obj) {
4     if (obj == null)
```

```
5       return false;
6     try {
7       return ((KIMEdge) obj).getID().equals(myID);
8     } catch (ClassCastException e) {return false;}
9   }
10  private static int id;
11  private static Transformer<KIMEdge, Number> defaultCost = new Transformer
        <KIMEdge, Number>() {
12    public Number transform(KIMEdge link) {
13      return link.getCost();
14    }
15  };
16  private static Transformer<KIMEdge, Number> constantCost = new
        Transformer<KIMEdge, Number>() {
17    public Number transform(KIMEdge link) {
18      return 1;
19    }
20  };
21  private int myID;
22  @Override
23  public String toString() {
24    return "[E" + myID + "," + cost + "]";
25  }
26  private Number cost;
27  public KIMEdge() {
28    this(null);
29  };
30  public KIMEdge(Number cost) {
31    synchronized (KIMEdge.class) {
32      id++;
33      this.myID = id;
34    }
35    this.cost = cost;
36  }
37  public static final Transformer<KIMEdge, Number> getDefautTransformer() {
38    return defaultCost;
39  }
40  public static final Transformer<KIMEdge, Number> getConstantTransformer()
        {
41    return constantCost;
42  }
43  public static class KIMEdgeFactory implements Factory<KIMEdge> {
44    @Override
```

```
45      public KIMEdge create() {
46        return new KIMEdge();
47      }
48    }
49    public static Factory<KIMEdge> getFactory() {
50      return new KIMEdge.KIMEdgeFactory();
51    }
52  }
```

Alguns *getters* e *setters* foram omitidos.

Não havia no JUNG uma classe que representasse a idéia de caminho. Para o JUNG um caminho era sempre uma lista de arcos e/ou arestas. Nós precisamos armazenar outras informações, como por exemplo, o caminho pai, o vértice de desvio, o custo, entre outros. Além disso, dado que a rotina SEP retorna ora um vértice ora um arco, seria interessante que a nossa classe soubesse criar um caminho com base nestas duas opções. A seguir exibimos um excerto da interface Path.

```
1  public interface Path<V extends KIMVertex, E extends KIMEdge> {
2    public abstract int getIteracao();
3    public abstract void setIteracao(int iteracao);
4    public abstract Double getCost();
5    public abstract KIMVertex getDevVertex();
6    public abstract int getDevNodeIndex();
7    public abstract KIMEdge getEdge(int pos);
8    public abstract List<KIMEdge> getEdges();
9    public abstract int getOrdem();
10   public abstract String getOrigem();
11   public abstract Path<KIMVertex, KIMEdge> getParent();
12   public abstract Path<KIMVertex, KIMEdge> getPrefix(KIMVertex last);
13   public abstract Path<KIMVertex, KIMEdge> getReverse();
14   public abstract KIMVertex getStart();
15   public abstract Path<KIMVertex, KIMEdge> getSubPath(KIMVertex v);
16   public abstract Path<KIMVertex, KIMEdge> getSubPath(KIMVertex v, int from
       ,
17       int to);
18   public abstract KIMVertex getTarget();
19   public abstract KIMVertex getVertex(int pos);
20   public abstract List<KIMVertex> getVertices();
21   public abstract void setOrdem(int ordem);
22   public abstract void setOrigem(String origem);
23   public abstract void setParent(Path<KIMVertex, KIMEdge> parent);
24   public class Factory {
```

```
25      private UndirectedGraph<KIMVertex, KIMEdge> g;
26      public UndirectedGraph<KIMVertex, KIMEdge> getGraph() {
27        return g;
28      }
29      public Transformer<KIMEdge, Number> getTransformer() {
30        return nev;
31      }
32
33      private Transformer<KIMEdge, Number> nev;
34
35      public Factory(UndirectedGraph<KIMVertex, KIMEdge> g,
36          Transformer<KIMEdge, Number> nev) {
37        this.g = g;
38        this.nev = nev;
39      }
40
41      public final Path<KIMVertex, KIMEdge> newInstance(
42          Path<KIMVertex, KIMEdge> p1, Path<KIMVertex, KIMEdge> p2) {
43        return new BasicPath<KIMVertex, KIMEdge>(p1, p2, g, nev);
44      }
45
46      public final Path<KIMVertex, KIMEdge> newInstance(List<KIMEdge> edges,
47          KIMVertex vertex) {
48        return new BasicPath<KIMVertex, KIMEdge>(edges, vertex, g, nev);
49      }
50
51      public final Path<KIMVertex, KIMEdge> newInstance(
52          ShortestPathKIM<KIMVertex, KIMEdge> Ts,
53          ShortestPathKIM<KIMVertex, KIMEdge> Tt, Pair pair, KIMVertex s,
54          KIMVertex t) {
55        return new BasicPath<KIMVertex, KIMEdge>(Ts, Tt, pair, s, t, g, nev);
56      }
57    }
58  }
```

Boa parte dos métodos definidos nas linhas 2 a 15 corresponde a campos utilizados para propósito de depuração. Na linha 25 vemos a definição de uma fábrica de Path que armazena um grafo e uma função custo mapeada na forma de um Transformer. A fábrica utiliza essas informações para gerar caminhos de diversas maneiras, dentre as quais destacamos:

linha 41 : a partir da concatenação de dois caminhos;

linha 51 : a partir das árvores T_s e T_t, s e t e um pair que ora representa uma aresta ora um vértice. A implementação deve gerar um caminho do tipo I quando pair contiver dois vértices iguais e caminhos do tipo II quando forem diferentes.

As demais rotinas não valem a pena serem exibidas. Lembramos que o código implementado pode ser baixado no sítio: http://code.google.com/p/k-cmc/.

CAPÍTULO 6

Resultados Experimentais

6.1 Motivação

Recentemente, há um grande interesse em trabalhos relacionados a análise experimental de algoritmos. Em particular, no caso do algoritmo DIJKSTRA, uma subrotina do algoritmo KIM, podemos citar os artigos de B.V. Cherkassky, A.V. Goldberg, T. Radzik e Craig Silverstein [8, 23, 9], e do algoritmo KIM podemos citar o artigo de Eleni Hadjiconstantinou and Nicos Christofides [24].

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 [33]. 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 [33], 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).

6.2 Ambiente experimental

Nos experimentos utilizamos duas plataformas:

(1) Um servidor Linux 32 bits com 4 processadores Intel Pentium 4 Xeon 3,39MHz e 3,5 GB de RAM;

(2) Um notebook rodando Linux Ubuntu 8.04, Kernel 2.6.24-23 com dois processadores Intel T7500 de 2.20Ghz e 2GB de RAM.

Os experimentos das seções 6.5, 6.6, 6.7 e 6.8 foram executados na plataforma (1), enquanto que os demais na (2). Como o que nos interessa não é especificamente o tempo absoluto de execução, mas a curva do tempo de execução, acreditamos que essa mudança de plataforma não chega a alterar nossos resultados.

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

```
1  public class StopWatch {
2    private final String id;
3    private boolean keepTaskList = true;
4    private final List taskList = new LinkedList();
5    private long startTimeMillis;
6    private boolean running;
7    private String currentTaskName;
8    private TaskInfo lastTaskInfo;
9    private int taskCount;
10   private long totalTimeMillis;
11
12   public StopWatch() {
13     this.id = "";
14   }
15
16   public StopWatch(String id) {
17     this.id = id;
18   }
```

```
19
20   public void start(String taskName) throws IllegalStateException {
21     if (this.running) {
22       throw new IllegalStateException(
23           "Can't start StopWatch: it's already running");
24     }
25     this.startTimeMillis = System.currentTimeMillis();
26     this.running = true;
27     this.currentTaskName = taskName;
28   }
29
30   public void stop() throws IllegalStateException {
31     if (!this.running) {
32       throw new IllegalStateException(
33           "Can't stop StopWatch: it's not running");
34     }
35     long lastTime = System.currentTimeMillis() - this.startTimeMillis;
36     this.totalTimeMillis += lastTime;
37     this.lastTaskInfo = new TaskInfo(this.currentTaskName, lastTime);
38     if (this.keepTaskList) {
39       this.taskList.add(lastTaskInfo);
40     }
41     ++this.taskCount;
42     this.running = false;
43     this.currentTaskName = null;
44   }
45
46   public long getLastTaskTimeMillis() throws IllegalStateException {
47     if (this.lastTaskInfo == null) {
48       throw new IllegalStateException(
49           "No tests run: can't get last interval");
50     }
51     return this.lastTaskInfo.getTimeMillis();
52   }
53
54   public long getTotalTimeMillis() {
55     return totalTimeMillis;
56   }
57 }
```

O uso da classe StopWatch é bem simples. Eis um pequeno exemplo:

```
1  StopWatch stopWatch = new StopWatch("KIM");
2  stopWatch.start();
```

```
3  /*Aqui vai a chamada à tarefa cuja tempo deseja-se medir*/
4  stopWatch.stop();
5  int time = stopWatch.getLastTaskTimeMillis();
```

Na linha 1 criamos uma instância da classe. Em seguida, na linha 2, inicializamos o cronômetro. Na linha 3 fica a chamada à função cujo tempo será medido. Paramos o cronômetro na linha 4 e chamamos o método getLastTaskTimeMillis() para obter o tempo decorrido.

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  // execução do algoritmo
5  long kimMemUsage = initMem - graphMemUsage - Runtime.getRuntime().
       freeMemory();
```

Na linha 1 a quantidade de memória livre do sistema é armazenada na variável initMem . Na linha 3, após a leitura e armazenamento do grafo em memória, guardamos a memória consumida pelo grafo. Executamos, em seguida, o nosso algoritmo e por fim, na linha 5, calculamos a diferença entre a quantidade inicial de memória livre e as quantidades de memória consumidas pelo grafo e pelo algoritmo.

Os testes foram criados levando-se em conta o consumo de tempo assintótico do algoritmo KIM, $\Theta(kT(n,m))$, onde $T(n,m)$ é o consumo de tempo da subrotina que calcula uma árvore de menores caminhos. No caso de grafos sem custos nas arestas, utilizamos uma busca em largura, cuja consumo de tempo é $\Theta(n+m)$, caso contrário utilizamos a implementação do Dijkstra feita no JUNG (seção 2.8), cujo consumo é $\Theta(m \log n)$. Nos testes realizados, utilizamos apenas a implementação do Dijkstra feita no JUNG.

6.3 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, desta maneira teríamos que convertê-los para grafos simétricos. O gerador implementado segue a idéia apresentada no artigo de Eleni Hadjiconstantinou and Nicos Christofides [24]:

(1) Inicialmente criamos os vértices;

(2) Em seguida, produzimos um ciclo hamiltoniano ligando cada vértice ao seu vizinho, garantindo assim que o grafo gerado seja conexo;

(3) Finalizamos adicionando, aleatoriamente, o restante dos arestas.

Na criação dos grafos aleatórios utilizamos os seguintes parâmetros:

- 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.

Preferimos, ao invés de utilizar valores de n e m independentes, usar o conceito **densidade** de um grafo, que consiste em dividir o número de arestas pelo número máximos de arestas de um grafo simétrico sem circuitos, ou seja, $m/\binom{n}{2}$, o que nos permite fazer comparações mais concisas. Assim, nosso gerador aceita receber os parâmetros n e m ou n e densidade, sendo que ao passar a densidade, o número de arestas é calculado segundo a definição.

A seguir exibimos o código da classe responsável pela geração de grafos aleatórios:

```
1  public class ConnectedUndirectedGraphGenerator<V extends KIMVertex, E
       extends KIMEdge>
2      implements GraphGenerator<KIMVertex, KIMEdge> {
3
4    private Factory<KIMVertex> vertexFactory;
5    private Factory<KIMEdge> edgeFactory;
6    private int n;
7    private int m;
8    private Random mRandom;
9
10   @Override
11   public Graph<KIMVertex, KIMEdge> create() {
12     UndirectedGraph<KIMVertex, KIMEdge> graph = new UndirectedSparseGraph<
           KIMVertex, KIMEdge>();
13     KIMVertex prior = vertexFactory.create();
14     KIMVertex first = prior;
15     graph.addVertex(prior);
16     for (int i = 2; i <= n; i++) {
```

```
17        KIMVertex 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<KIMVertex> vertices = new ArrayList<KIMVertex>(graph.getVertices()
          );
24      while (graph.getEdgeCount() < m) {
25        KIMVertex u = vertices.get((int) (mRandom.nextDouble() * n));
26        KIMVertex v = vertices.get((int) (mRandom.nextDouble() * n));
27        if (!v.equals(u) && !graph.isSuccessor(v, u)) {
28          graph.addEdge(edgeFactory.create(), u, v);
29        }
30      }
31      return graph;
32    }
33
34    public ConnectedUndirectedGraphGenerator(int n, double densidade) {
35      this(n, ((int) (densidade * n * (n - 1) / 2)));
36    }
37
38    public ConnectedUndirectedGraphGenerator(Factory<KIMVertex> vertexFactory
        ,
39        Factory<KIMEdge> edgeFactory, int n, int m) {
40      if (m < n)
41        throw new IllegalArgumentException(
42            "Numero de arcos deve ser no mï¿½nimo igual ao "
43                + "numero de vertices");
44      if (m > (n * (n - 1) / 2))
45        throw new IllegalArgumentException("Nï¿½o ï¿½ possï¿½vel criar " + m
46            + " arcos de modo que nï¿½o haja arcos "
47            + "paralelos e loops num grafo com " + n + " vï¿½rtices");
48      this.vertexFactory = vertexFactory;
49      this.edgeFactory = edgeFactory;
50      this.n = n;
51      this.m = m;
52      mRandom = new Random();
53    }
54
55    public ConnectedUndirectedGraphGenerator(int n, int m) {
56      this(KIMVertex.getFactory(), KIMEdge.getFactory(), n, m);
57    }
```

```
58
59   public static void atribuiCustosNasArestas(Graph<?, KIMEdge> grafo,
60       double min, double max) {
61     if (min > max)
62       throw new IllegalArgumentException("min deve ser inferior a max");
63     if (min < 0)
64       throw new IllegalArgumentException("min deve ser maior que zero");
65     Iterator<KIMEdge> i = grafo.getEdges().iterator();
66     Random random = new Random();
67     while (i.hasNext()) {
68       KIMEdge edge = i.next();
69       edge.setCost(random.nextDouble() * (max - min) + min);
70     }
71   }
72 }
```

A classe implementa a interface Generator definindo a função create, que deve devolver um grafo. Para usá-la podemos escolher um dentre os três construtores apresentados nas linhas 34, 38 e 55. Por exemplo, para construir um grafo conexo com 100 vértices e densidade 0.1, ou seja, 495 arestas, fazemos:

```
1  Graph<KIMVertex, KIMEdge> graph =
2    new ConnectedUndirectedGraphGenerator<KIMVertex, KIMEdge>(100,0.1);
```

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 na obtenção dos k-menores caminhos;
- Tempo total gasto nas construções das árvores de menores caminhos: T_s e T_t;
- Tempo total gasto nas execuções da rotina SEP;
- Tempo total gasto na obtenção do i-ésimo menor caminho.

O tempo gasto na criação do grafo não será considerado. A fim de tentar evitar escolhas ruins das origens e destinos, escolhemos cinco origens e destinos e calculamos a média dos tempos. O mesmo é feito para o cálculo do consumo de memória.

Uma vez que o consumo de tempo assintótico do algoritmo KIM está definido em função do número de caminhos k a serem gerados, número m de arestas e número n

de vértices, fixaremos nos testes duas destas variáveis, deixando a outra livre, a fim de estudarmos o comportamento do algoritmo. Para efeito de análise trabalharemos sempre com a densidade e nunca diretamente com o número de arestas.

6.4 Gráficos e análises

Começaremos nossa análise exibindo um gráfico com os tempos de execução, em função do número k de caminhos gerados, para cada uma das densidades: 0.1, 0.5 e 1.0.

Em seguida, faremos uma análise semelhante, mas trocando a densidade pelo número de caminhos, ou seja, a densidade entrará no eixo x e cada uma das curvas corresponderá a uma quantidade de caminhos.

Variaremos então o número de vértices para as densidades: 0.1, 0.5 e 1.0, e geraremos curvas para execuções do algoritmo KIM na geração de: $100, 200, \ldots 1000$-menores caminhos.

Passaremos então a avaliar a influência que as principais subrotinas do algoritmo KIM têm no seu tempo de execução. Para isto exibiremos gráficos considerando os tempos de execução das rotinas SEP e das árvores de menores caminhos.

Estamos interessados também em saber o custo de cada caminho. Queremos descobrir a evolução dos custos de obtenção de um novo caminho. Para isto, fixando-se uma densidade e um número de vértices, exibiremos um gráfico com os custos de geração do primeiro, segundo, $\ldots, k$-ésimo caminho e observaremos a curva correspondente à ligação dos pontos calculados.

Finalizaremos com gráficos que nos permitirão analisar como se comporta o consumo de memória, primeiramente pela densidade e depois pelo número de caminhos.

Para cada gráfico aproximaremos cada uma das curvas utilizando uma função apropriada, usando o "curve-fitting"do GNUPlot 4.2, com o objetivo de verificar a correlação entre as variáveis envolvidas bem como as aderências das funções às análises teóricas.

Todos os gráficos foram gerados a partir de execuções do algoritmo KIM em grafos simétricos e com custos maiores que zero nas arestas

Quando não for especificada a quantidade de vértices utilizada, essa será igual a 100.

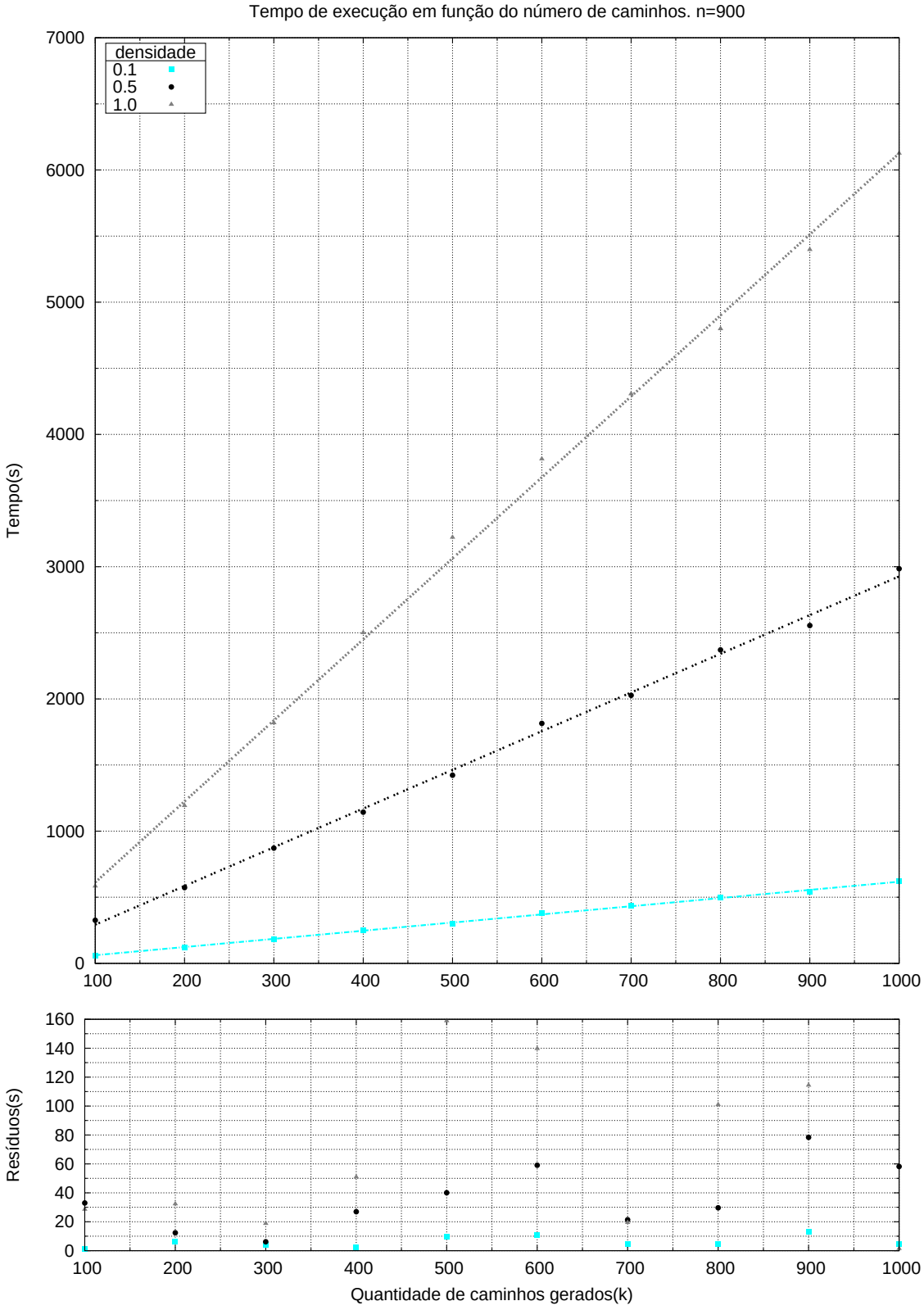
6.5 Tempo em função de k

A seguir o gráfico exibindo os tempos de execução do algoritmo KIM, em segundos, em função do número de caminhos gerados. Cada curva corresponde a uma certa densidade. Aproximamos os pontos obtidos para cada valor de k utilizando regressões lineares através de funções da família $tempo(k) = ak$. Observe que não utilizamos $tempo(k) = ak + b$, uma vez que quando k é zero, o algoritmo não tem trabalho nenhum [1], ou seja, leva tempo desprezível.

Abaixo do gráfico, exibimos os valores absolutos dos resíduos correspondentes à diferença entre o valor de cada ponto obtido a partir da execução do algoritmo KIM e o valor calculado pela curva $tempo(k) = ak$ correspondente, com o objetivo de mostrar a aderência da curva aos dados reais.

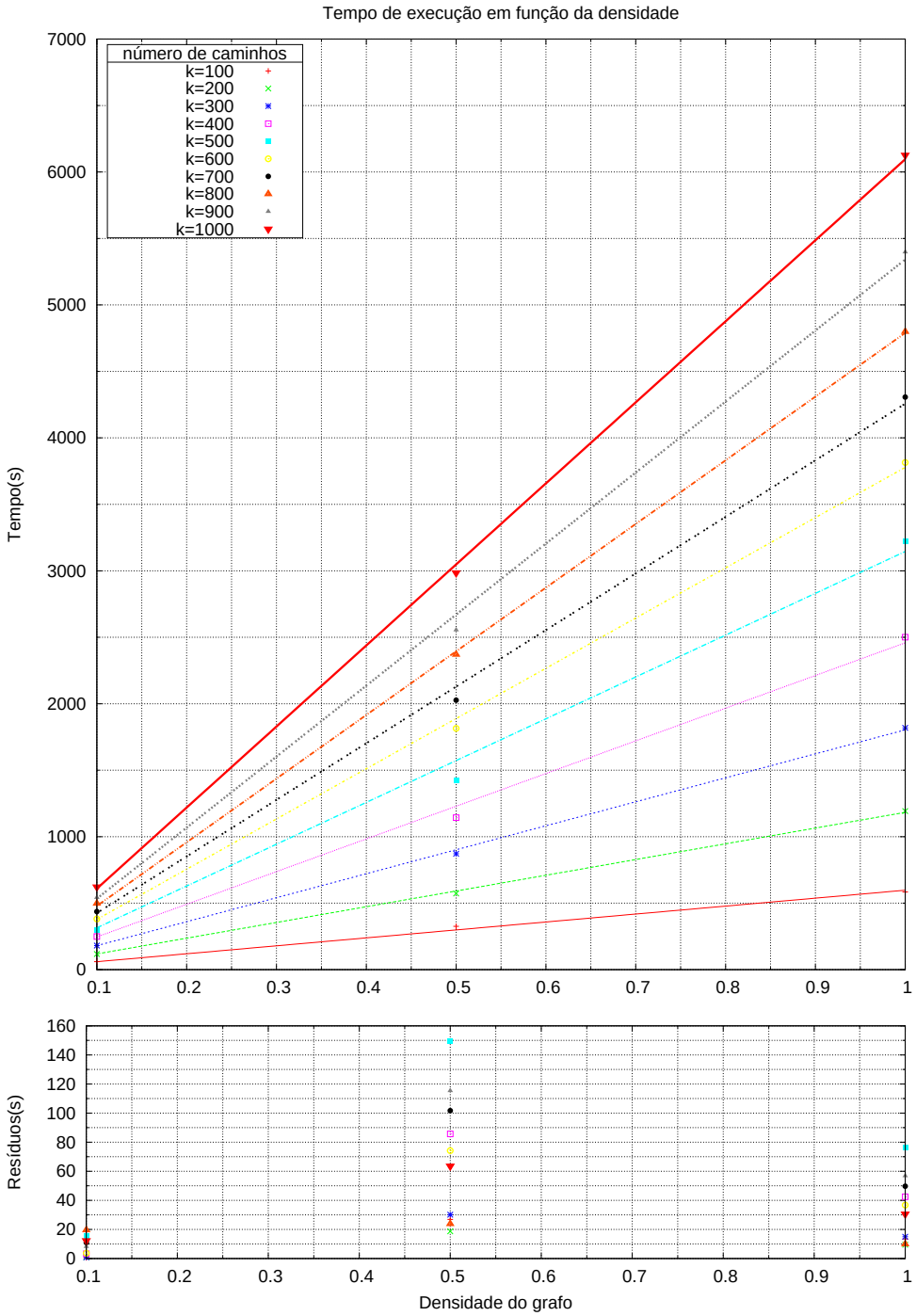
Fica bem clara a dependência linear entre o tempo de execução e a quantidade de caminhos gerados, para todas as densidades escolhidas. Isto vem apoiar a análise assintótica: $tempo = \Theta(kT(n, m))$. Observe que fixando-se n e m, ou seja, fixando n e a densidade, um aumento linear em k implica num aumento também linear no tempo.

[1] Obviamente o algoritmo tem algum trabalho, afinal gasta-se algum tempo para descobrir que não é preciso calcular nenhum caminho, mas esse tempo será desconsiderado nas nossas análises.


Tempo de execução em função do número de caminhos. n=900
densidade
0.1
0.5
1.0
Tempo(s)
0
1000
2000
3000
4000
5000
6000
7000
100
200
300
400
500
600
700
800
900
1000
Resíduos(s)
0
20
40
60
80
100
120
140
160
Quantidade de caminhos gerados(k)

6.6 Tempo em função da densidade

Agora colocaremos o número de caminhos k no eixo x, exibindo então uma curva para cada k de 100 a 1000 com intervalo de 100.



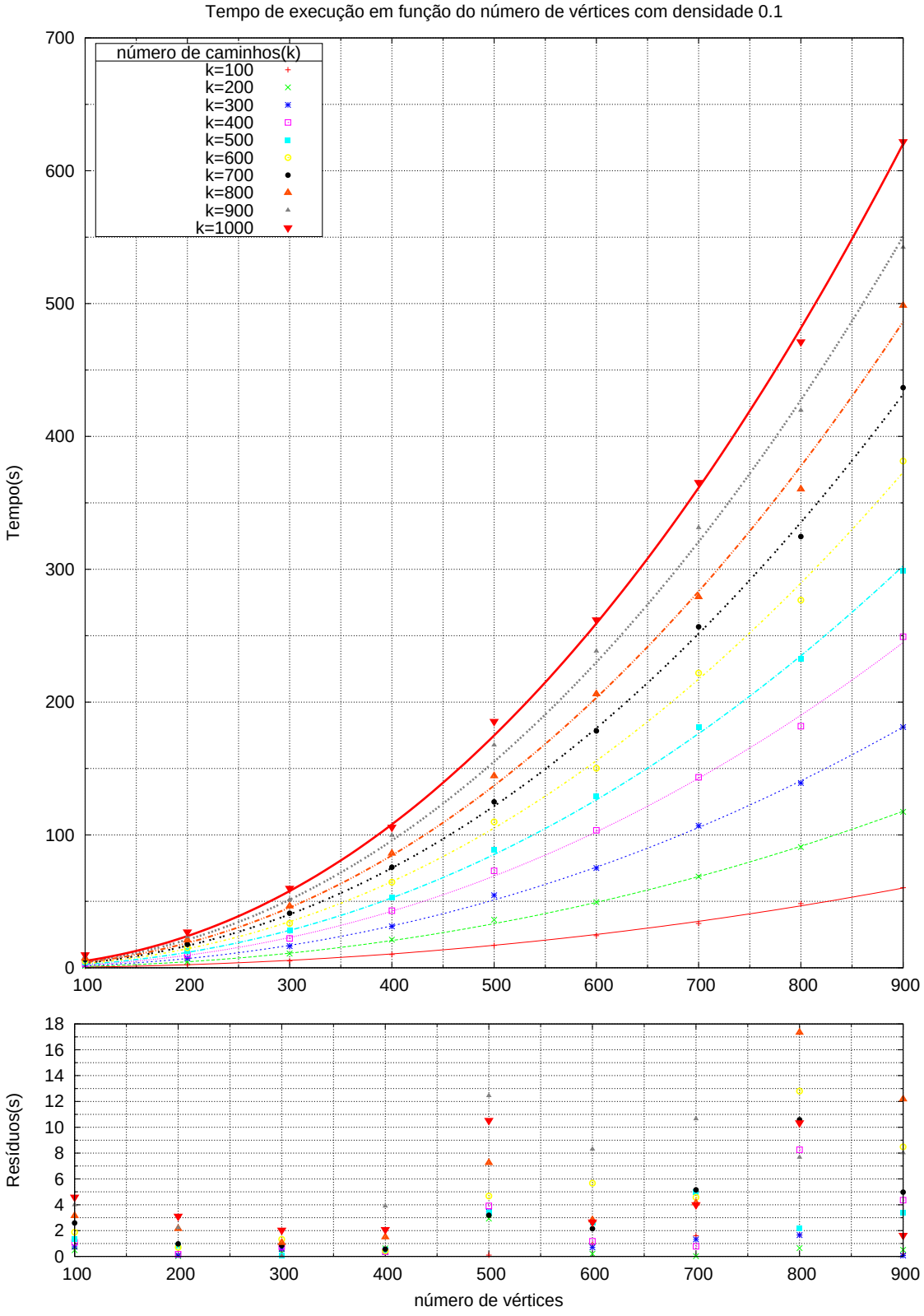
Nos nossos experimentos, estamos utilizando o algoritmo DIJKSTRA com min-heap na implementação da rotina que resolve o CM, sendo assim, $T(n,m) = \Theta(m \log n)$.

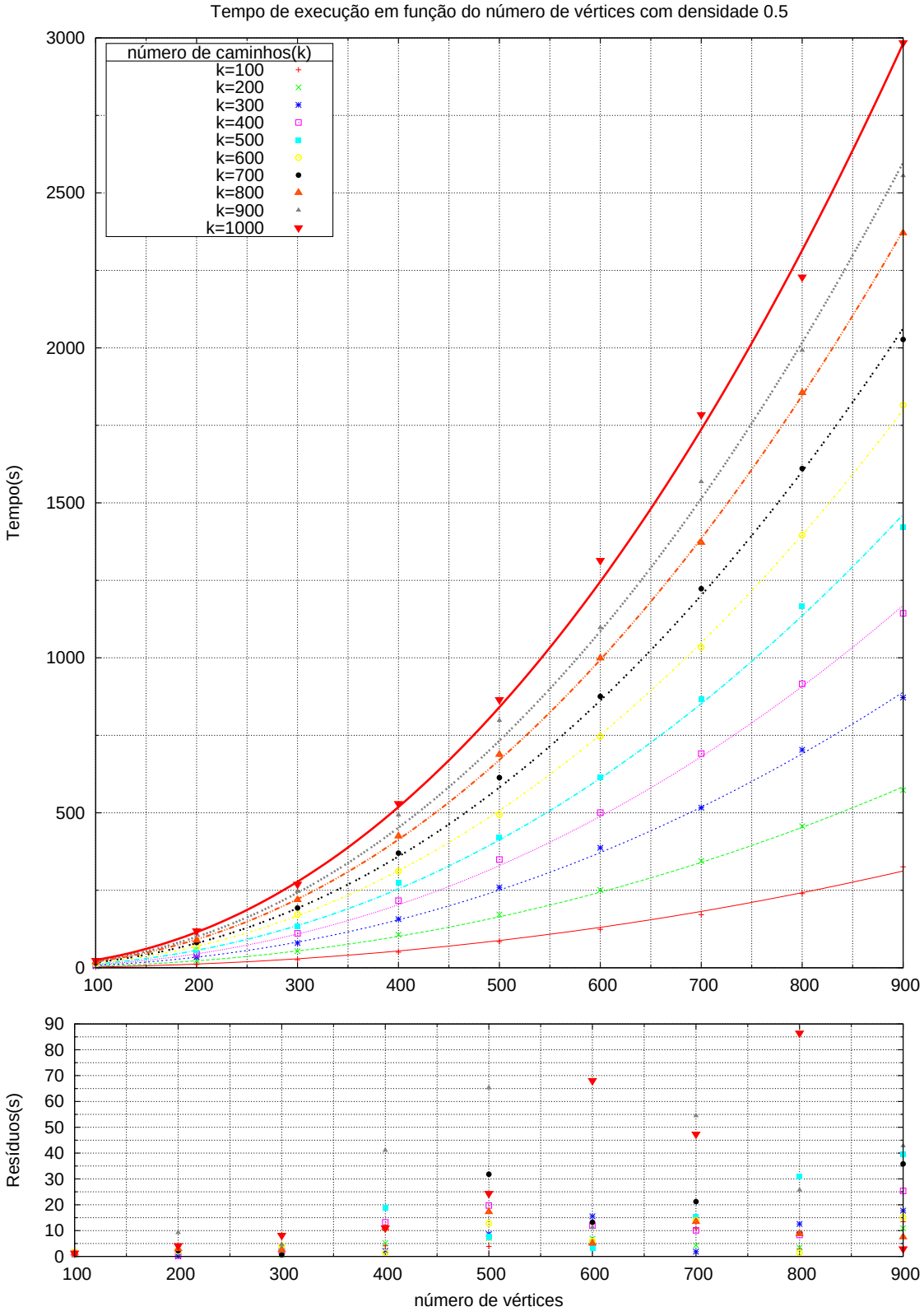
Utilizando-se o conceito de densidade (d), podemos escrever o consumo de tempo assintótico do algoritmo KIM como $\Theta(kdn^2 \log n)$. Em cada curva o valor de k é fixo e a densidade varia entre os valores 0.1, 0.5 e 1.0. Uma vez que o valor de n está fixo no gráfico, podemos aproximar as curvas por funções da família $tempo(d) = ad$. Pelo gráfico, observamos a dependência linear entre densidade (d) e tempo para cada valor de k, o que vem a comprovar a análise assintótica, $\Theta(km \log n)$ ou, usando a definição de densidade, $\Theta(kdn^2 \log n)$.

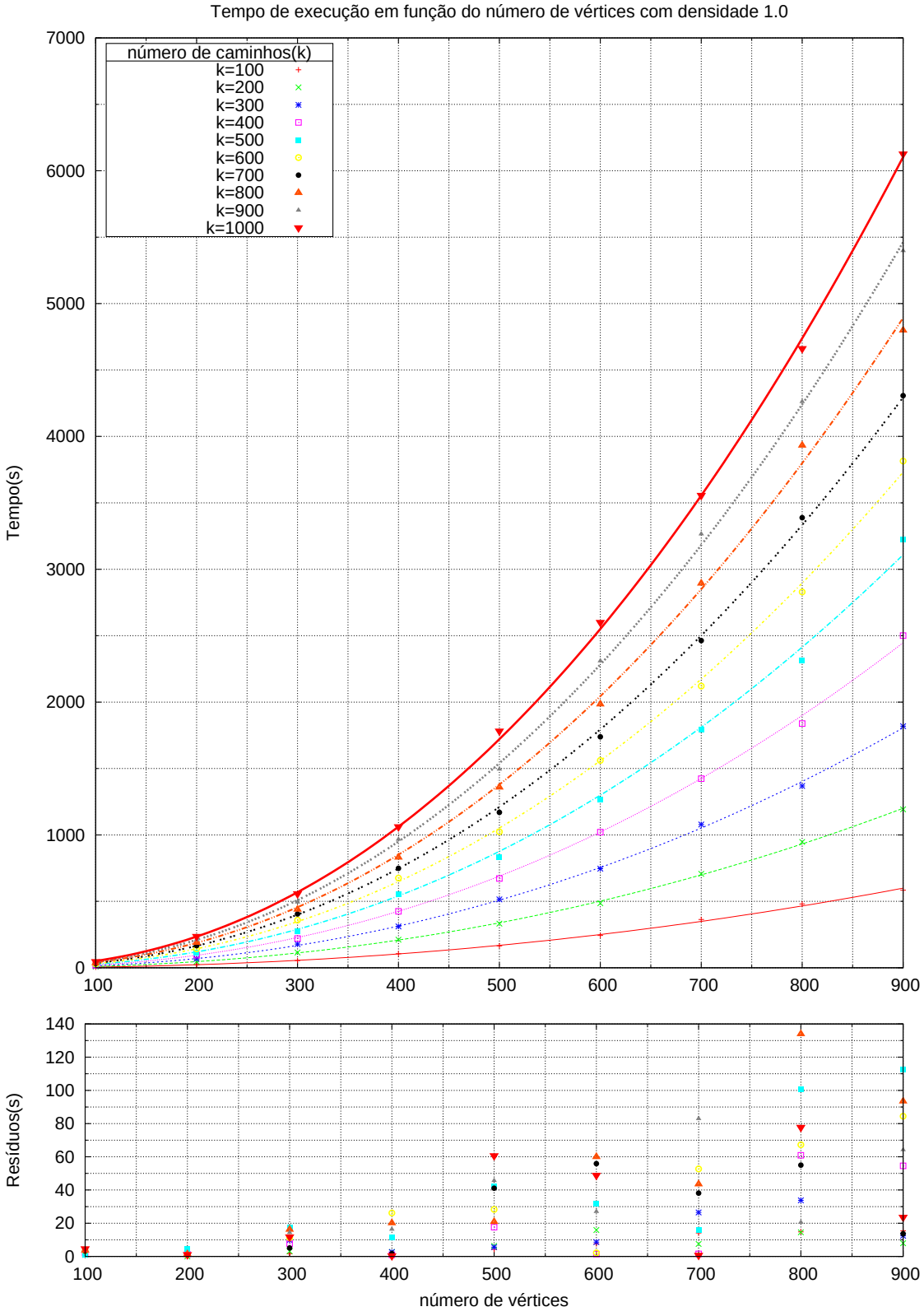
6.7 Tempo em função de n

Uma vez que o consumo de tempo do algoritmo KIM usando DIJKSTRA com min-heap é $\Theta(km \log n)$, decidimos gerar gráficos para execuções variando o número n de vértices.

Nos gráficos a seguir exibimos 10 curvas, uma para cada número k de caminhos a serem gerados, em função do número n de vértices do grafo, usando grafos de densidades: 0.1, 0.5 e 1.0. Observamos que um aumento linear no número de vértices não resulta num aumento linear no consumo de tempo. Lembrando a definição de densidade: $d = \frac{m}{\binom{n}{2}}$ e sabendo que em cada gráfico ela é fixa, temos que $m = d\binom{n}{2} = \frac{d(n^2-n)}{2} = \Theta(n^2)$, assim, podemos escrever $\Theta(km \log n)$ como $\Theta(kn^2 \log n)$. Por conta disso, um aumento linear em n resulta num aumento não linear no consumo de tempo. Decidimos aproximar as curvas de cada gráfico utilizando funções $tempo(n,k) = akn^2 \log n$ através do recurso "curve-fitting"do GNUPlot 4.2. Abaixo de cada gráfico, exibimos um gráfico com os valores absolutos dos resíduos, ou seja, as diferenças entre as curvas de aproximação geradas e os valores obtidos, experimentalmente, para cada ponto.


Tempo de execução em função do número de vértices com densidade 0.1
número de caminhos(k)
k=100
k=200
k=300
k=400
k=500
k=600
k=700
k=800
k=900
k=1000
Tempo(s)
Resíduos(s)
número de vértices


Tempo de execução em função do número de vértices com densidade 0.5
número de caminhos(k)
k=100
k=200
k=300
k=400
k=500
k=600
k=700
k=800
k=900
k=1000
Tempo(s)
Resíduos(s)
número de vértices


Tempo de execução em função do número de vértices com densidade 1.0
número de caminhos(k)
k=100
k=200
k=300
k=400
k=500
k=600
k=700
k=800
k=900
k=1000
Tempo(s)
Resíduos(s)
número de vértices
0
1000
2000
3000
4000
5000
6000
7000
0
20
40
60
80
100
120
140
100
200
300
400
500
600
700
800
900

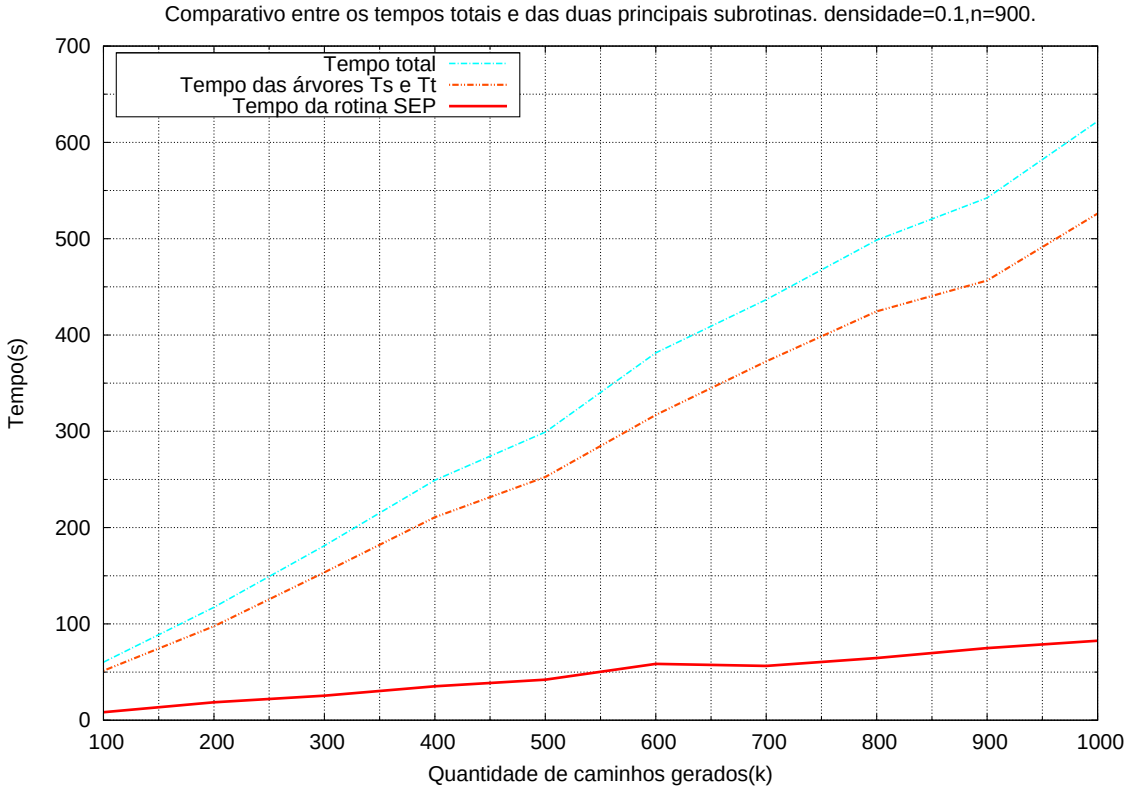
6.8 Função SEP e árvores de menores caminhos

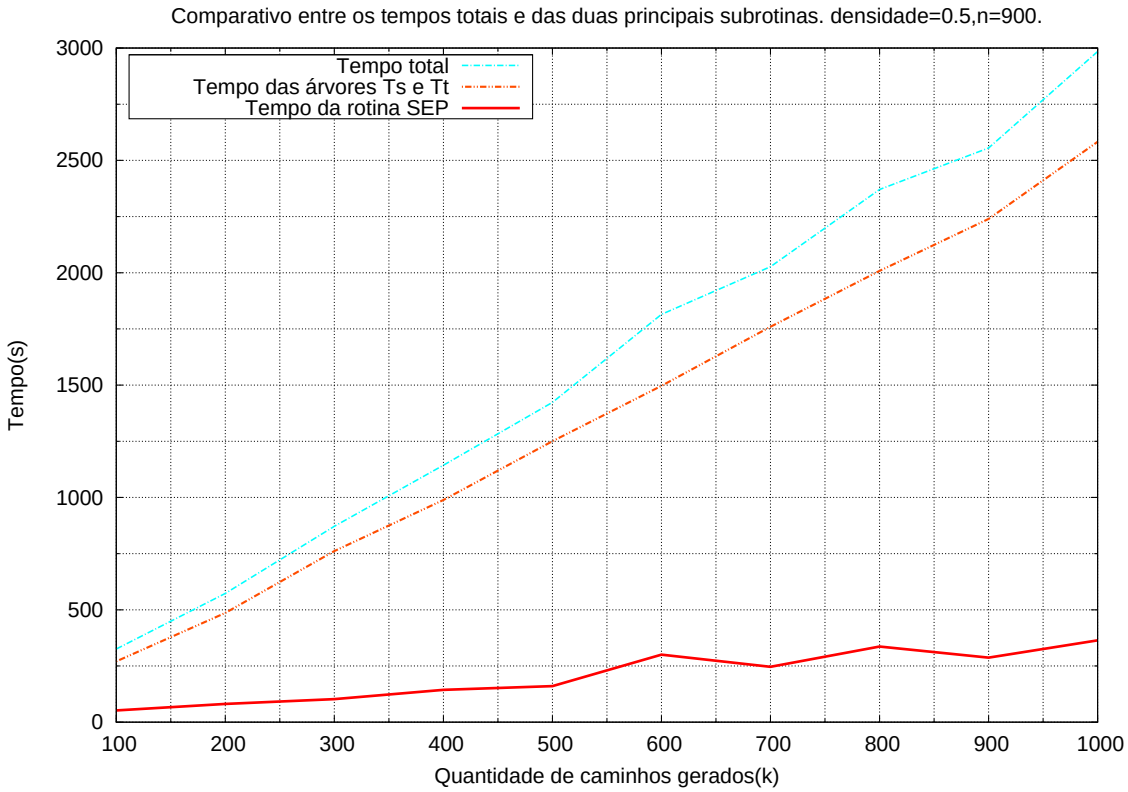
Nas seções anteriores, analisamos o tempo total de execução do algoritmo KIM levando em conta as densidades, quantidade de caminhos gerados e número de vértices. Agora, nos aprofundaremos um pouco, procurando avaliar o custo de algumas rotinas que compõe o algoritmo.

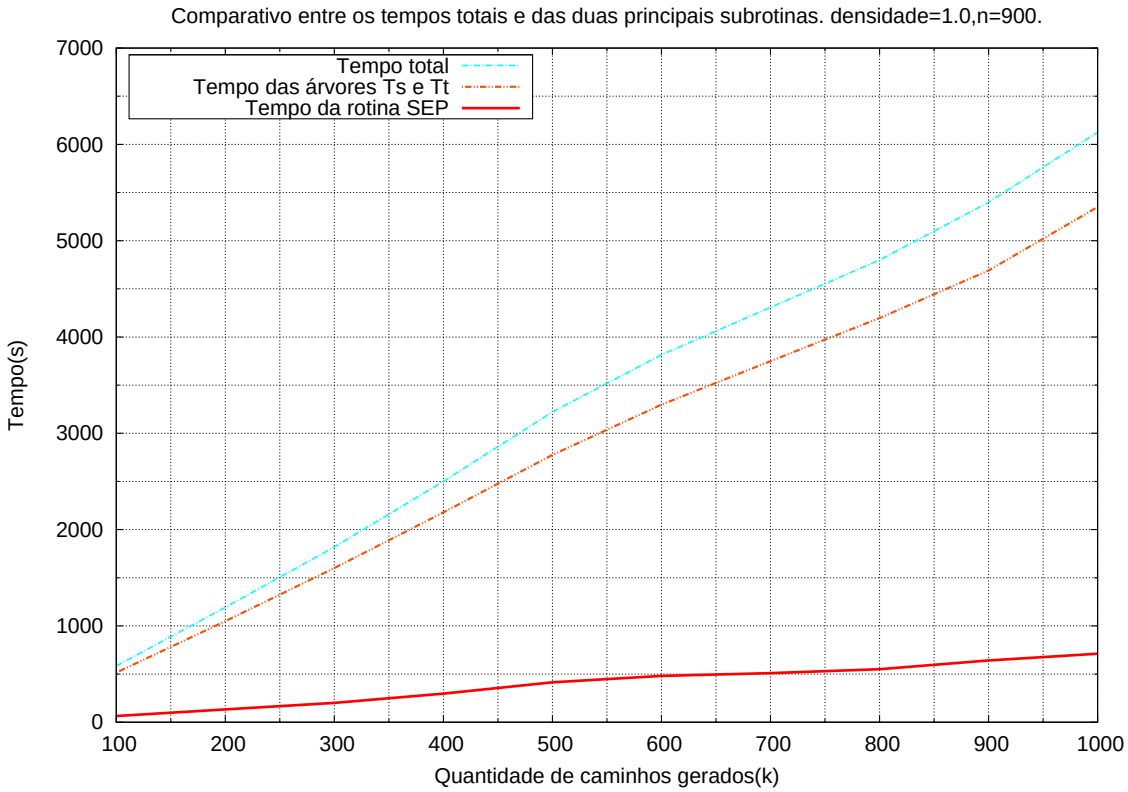
O algoritmo KIM conta, basicamente, com duas subrotinas principais, as quais são responsáveis pela maior parte do seu consumo de tempo, são elas:

- Rotinas de construções de árvores de menores caminhos. Estas rotinas correspondem a duas execuções de um algoritmo que resolva o CM modificado, cuja implementação está apresentada na seção 5.5, uma utilizando s como raiz retornando como resposta a árvore T_s e outra onde a raiz é t retornando a árvore T_t.
- SEP é a rotina responsável por calcular um desvio mínimo restrito de s a t, utilizando para tal, as árvores T_s e T_t as quais são rotuladas usando-se ϵ e ζ, como explicado na seção 5.2.

A seguir, exibimos gráficos com os tempos de execução da rotina SEP, das construções das árvores e os totais do algoritmo KIM. O objetivo é visualizar o quão 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 quaisquer melhorias deveriam ser pensadas.

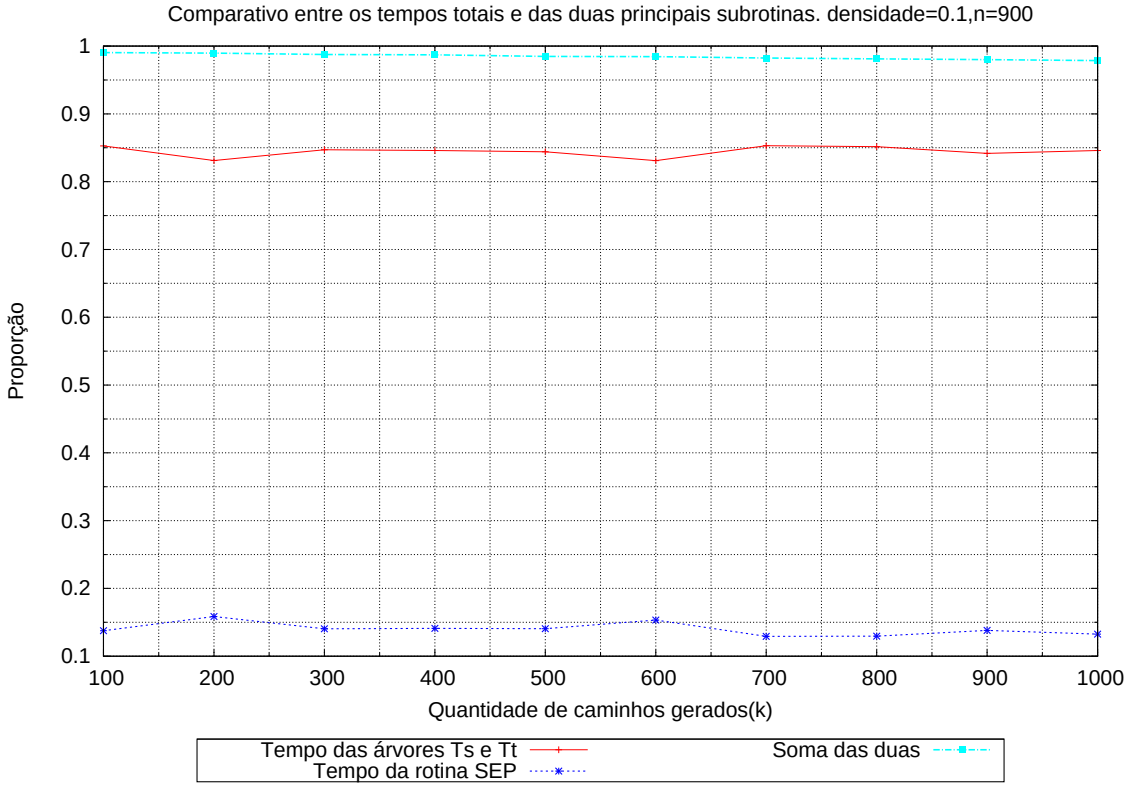

Comparativo entre os tempos totais e das duas principais subrotinas. densidade=0.1,n=900.
Tempo total
Tempo das árvores Ts e Tt
Tempo da rotina SEP
Tempo(s)
Quantidade de caminhos gerados(k)
0
100
200
300
400
500
600
700
800
900
1000

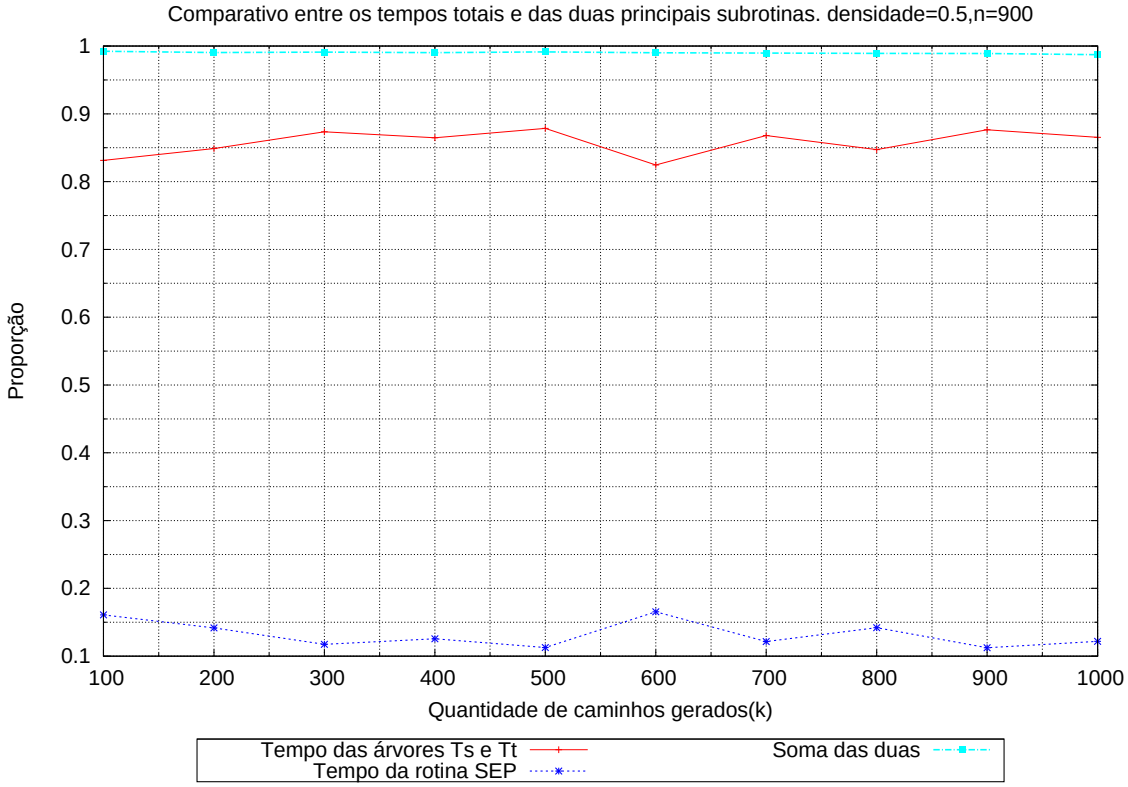

Comparativo entre os tempos totais e das duas principais subrotinas. densidade=0.5,n=900.
Tempo total
Tempo das árvores Ts e Tt
Tempo da rotina SEP
Tempo(s)
Quantidade de caminhos gerados(k)
0
500
1000
1500
2000
2500
3000
100
200
300
400
500
600
700
800
900
1000

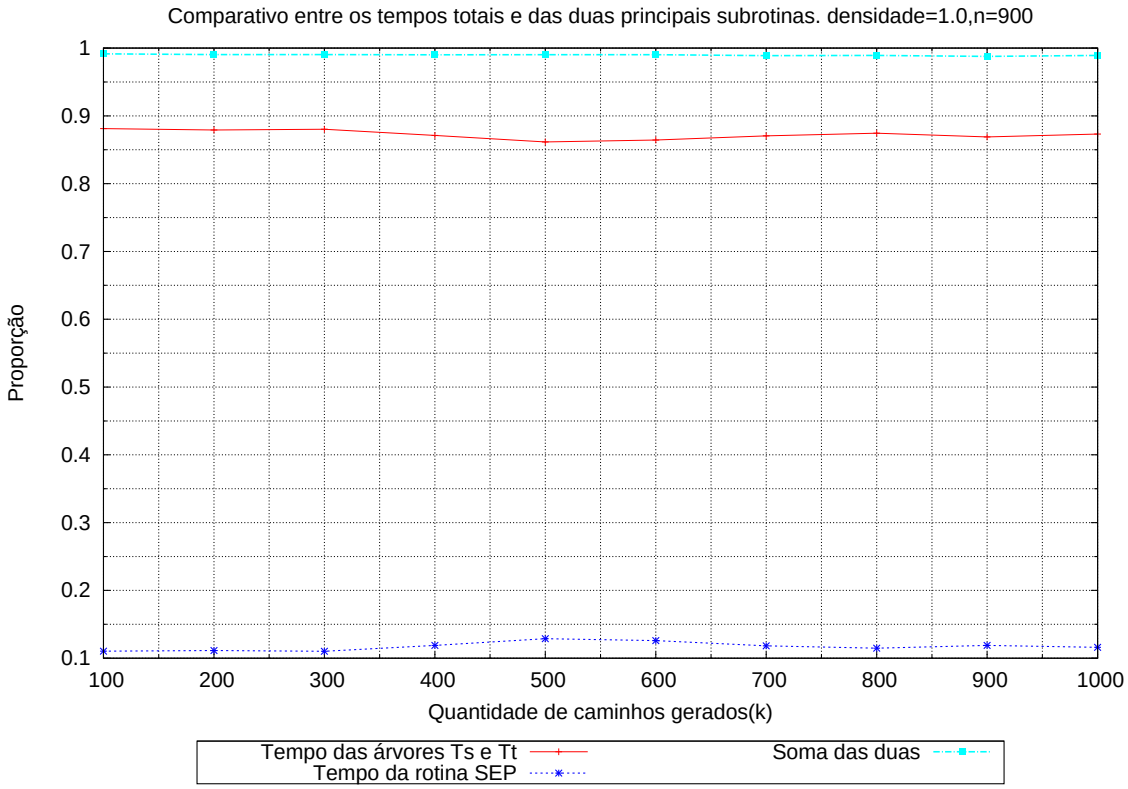


A partir dos gráficos é possível perceber que as rotinas citadas realmente correspondem a uma importante fatia do tempo total de execução do algoritmo.

Notamos que as construções das árvores de menores caminhos consomem a maior parte do tempo total de execução. Exibiremos, a seguir, gráficos com as proporções de tempo utilizadas por cada uma das rotinas anteriormente citadas, bem como a proporção total de tempo consumida por ambas juntas.

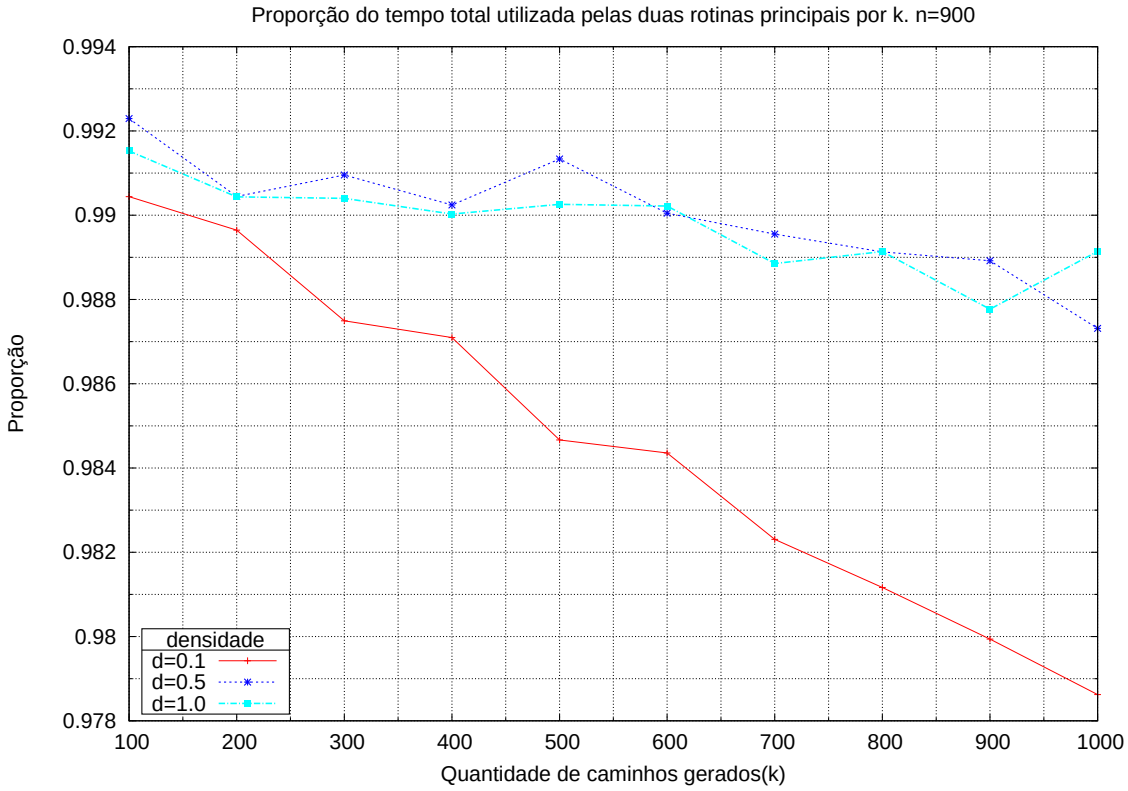

Comparativo entre os tempos totais e das duas principais subrotinas. densidade=0.1,n=900
Proporção
Quantidade de caminhos gerados(k)
Tempo das árvores Ts e Tt
Tempo da rotina SEP
Soma das duas


Comparativo entre os tempos totais e das duas principais subrotinas. densidade=0.5,n=900
Proporção
Quantidade de caminhos gerados(k)
Tempo das árvores Ts e Tt
Tempo da rotina SEP
Soma das duas

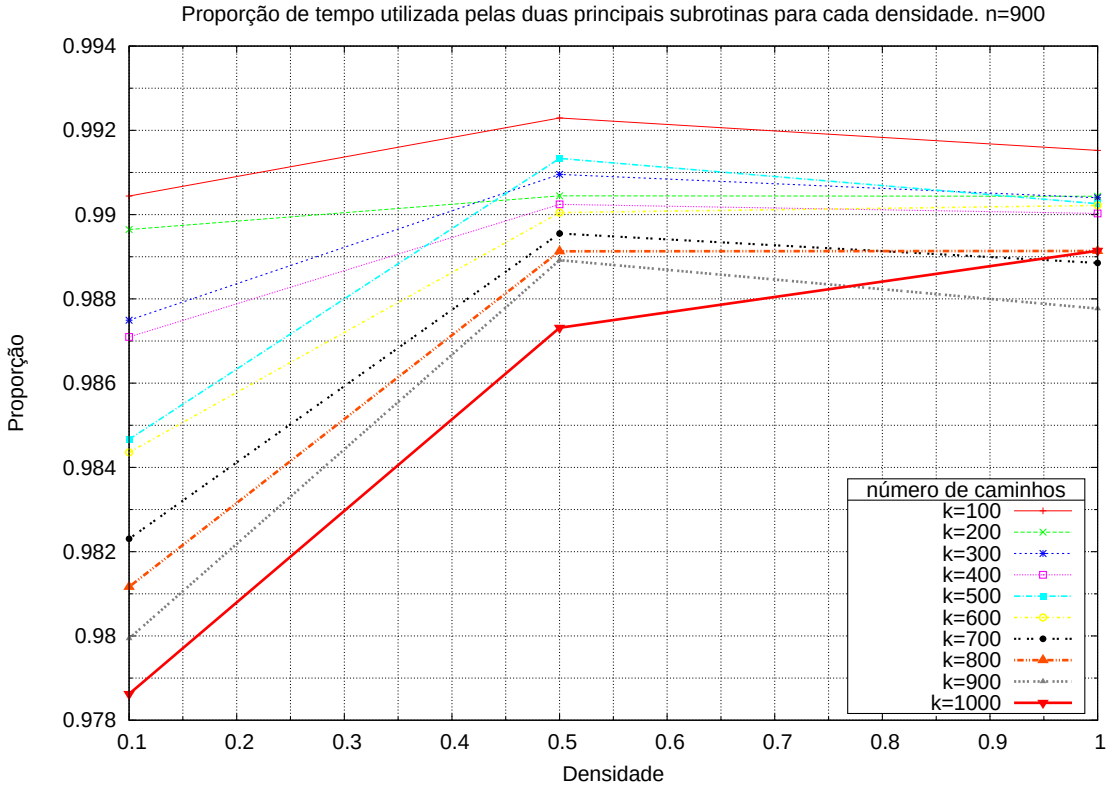


Observamos que, para as densidade 0.1 e 0.5, com o aumento de k, a proporção de tempo utilizada nas construções das árvores de menores caminhos cresce, ao passo que a utilizada pela rotina SEP diminui. Para a densidade 1.0, as proporções se mantém estáveis. Mais uma vez fica clara a importância que essas rotinas têm no tempo de execução do algoritmo KIM.

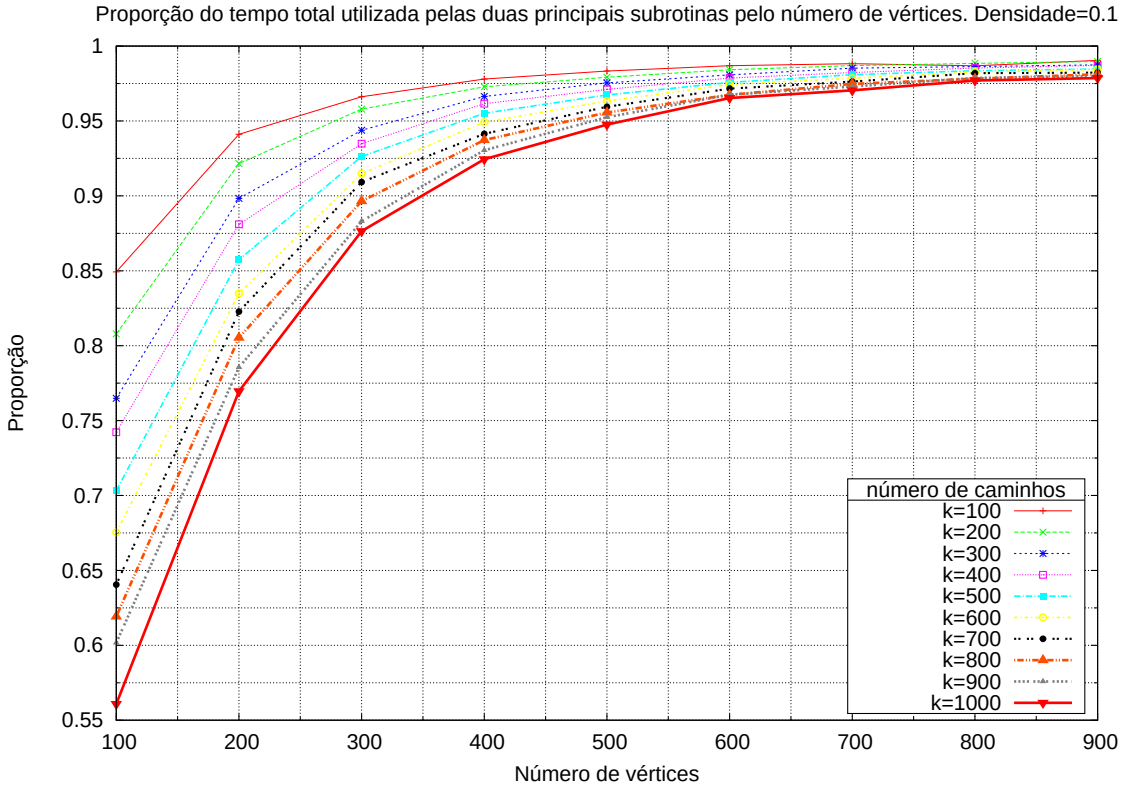
Vamos sintetizar em um único gráfico a proporção de tempo utilizada pelas duas subrotinas para as densidades 0.1, 0.5 e 1.0.

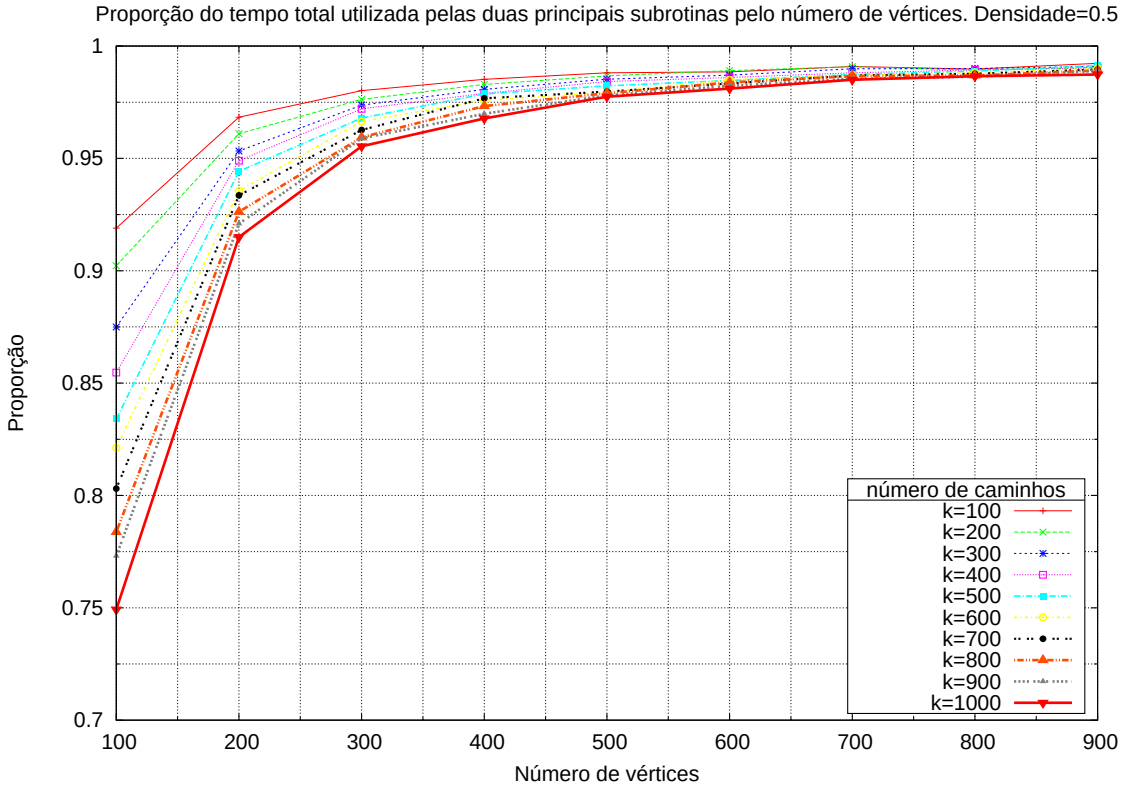


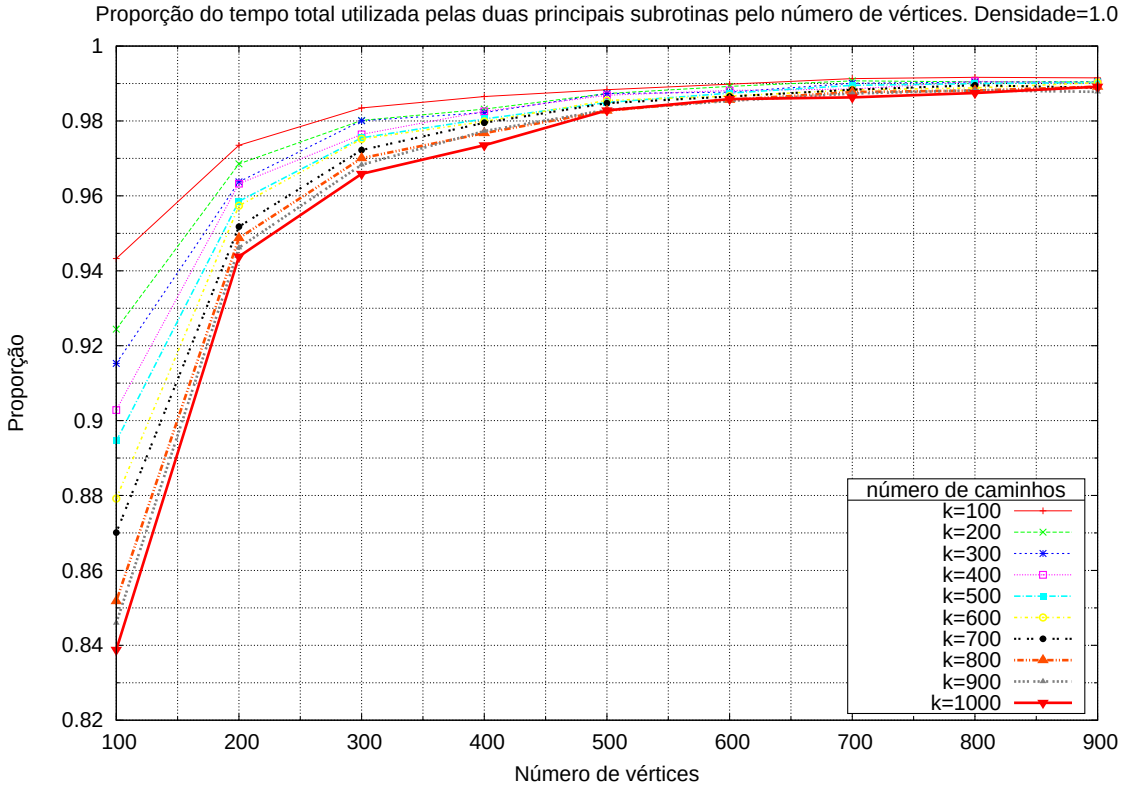
Façamos o mesmo colocando a densidade no eixo x.



Façamos o mesmo colocando o número de vértices no eixo x.


Proporção do tempo total utilizada pelas duas principais subrotinas pelo número de vértices. Densidade=0.1
Proporção
Número de vértices
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
100
200
300
400
500
600
700
800
900
número de caminhos
k=100
k=200
k=300
k=400
k=500
k=600
k=700
k=800
k=900
k=1000


Proporção do tempo total utilizada pelas duas principais subrotinas pelo número de vértices. Densidade=0.5
Proporção
Número de vértices
1
0.95
0.9
0.85
0.8
0.75
0.7
100
200
300
400
500
600
700
800
900
número de caminhos
k=100
k=200
k=300
k=400
k=500
k=600
k=700
k=800
k=900
k=1000



6.9 Custo por caminho

Até o momento realizamos análises considerando o tempo de execução do algoritmo para um certo número k de caminhos gerados. Queremos agora verificar se o custo de cada caminho é independente dos demais ou se existe alguma relação entre eles. Será que os 100 primeiros caminhos consomem mais tempo que os 100 últimos?

O primeiro caminho é mais lento ou mais rápido que os demais?

Vamos, antes de prosseguir, relembrar o funcionamento do algoritmo KIM para encontrar k-menores caminhos entre dois vértices s e t:

(1) Primeiramente o menor caminho de s a t é calculado, P_1, usando um algoritmo para o CM modificado;

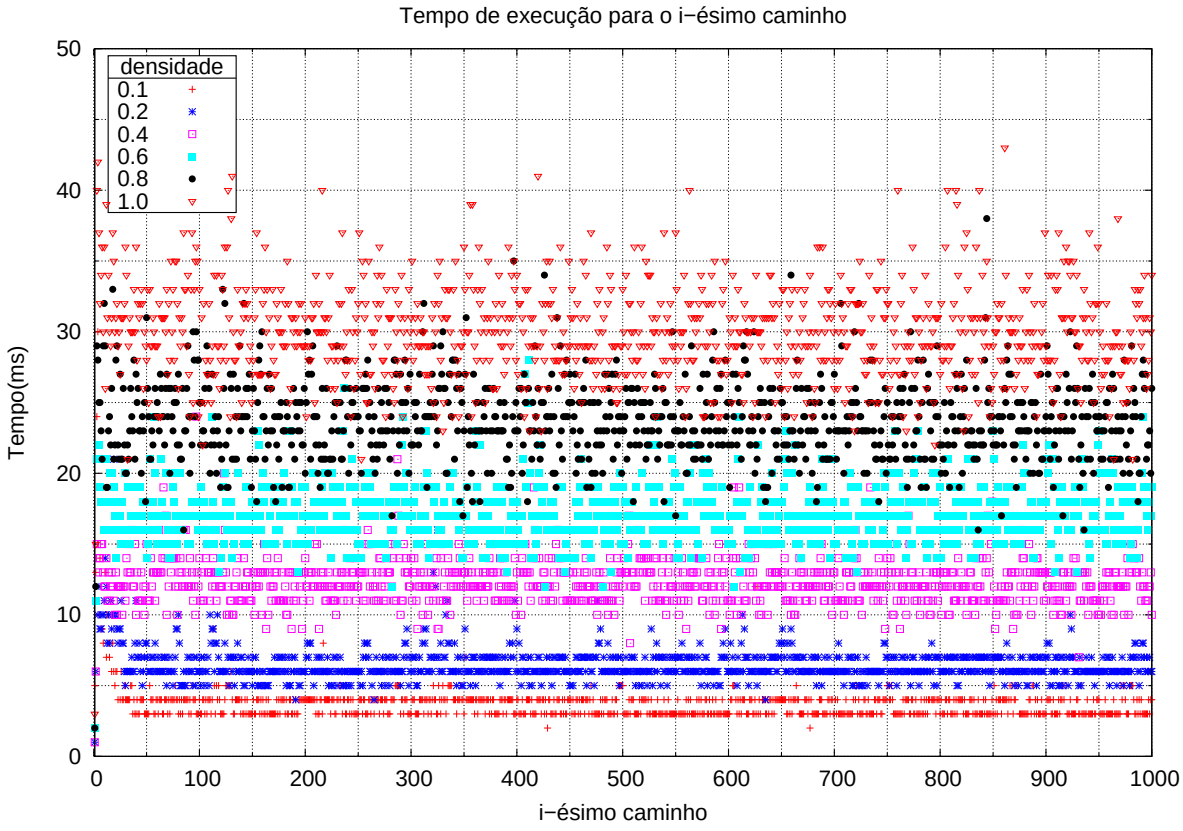
(2) Em seguida, para o cálculo do segundo menor caminho,P_2, chamamos a função FSP, a qual constrói a árvore de menores caminhos com raiz em s (T_s) e a de menores caminhos com raiz em t (T_t), sendo que cada árvore corresponde a uma exe-

cução do CM modificado. Em seguida, a rotina SEP é chamada, percorrendo os vértices da árvore T_s e retornando um desvio mínimo de P_1.

(3) Sejam P_1 e P_2, respectivamente, o primeiro e segundo menores caminhos de s a t, calculamos os menores caminhos P_a, P_b e P_c, distintos entre si e diferentes de P_1 e P_2. Cada um deles é colocado na lista de caminhos candidatos, sendo que o menor deles é retirado.

(4) Cada iteração do algoritmo consistirá então em retirar da lista de candidatos o caminho de menor custo e, a partir dele e de seu caminho pai gerar outros três: (P_a, P_b e P_c).

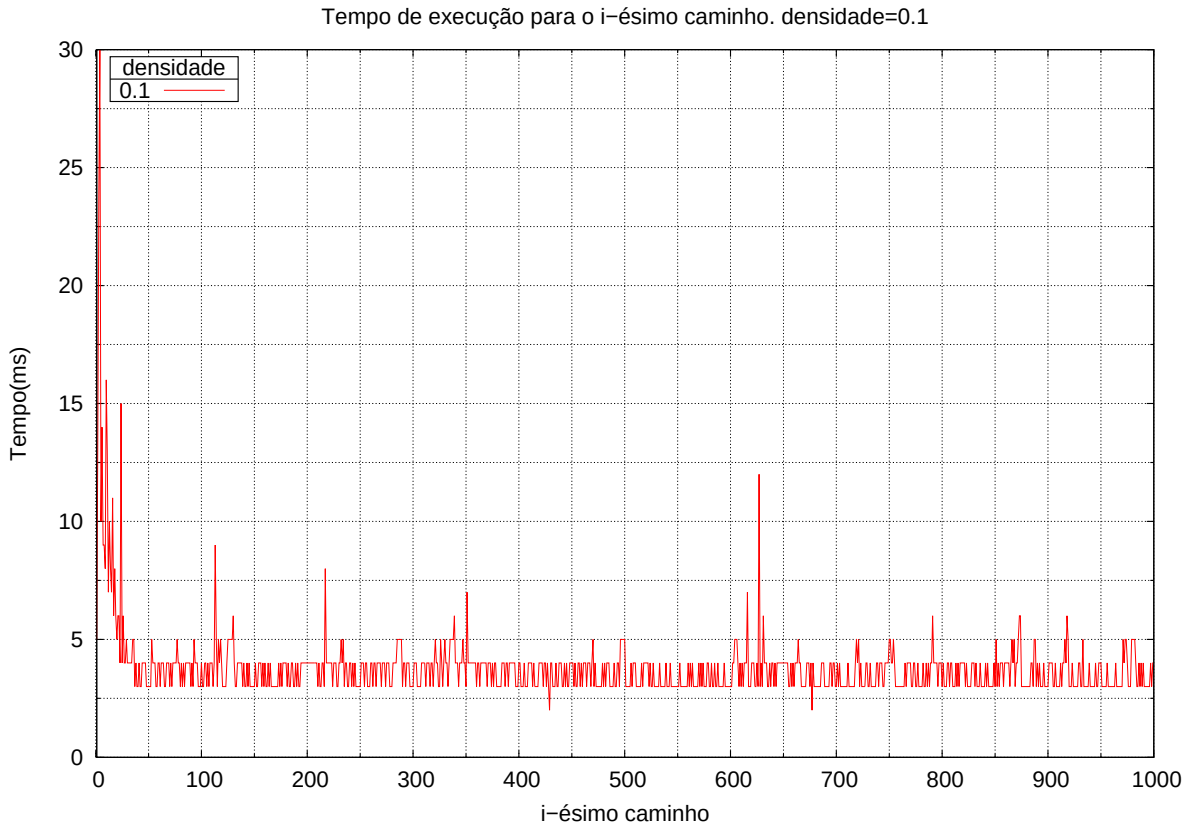
Começaremos exibindo os custos de geração de cada i-ésimo caminho para densidades entre 0.1 e 1.0 com intervalo de 0.2.



A partir do gráfico podemos observar que os custos de obtenção de caminhos em grafos mais densos são em geral maiores que em grafos esparsos.

Vamos trabalhar agora apenas com as densidades 0.1, 0.5 e 1.0, pois acreditamos que sejam suficientes para a nossa análise.

Gráfico com o custo para o *i*-ésimo caminho em um grafo de densidade 0.1.



Notamos que os primeiros caminhos têm custos mais elevado que vão decrescendo chegando a estabilizar a partir do 40º caminho, aproximadamente. Os quatro primeiros caminhos têm os seguintes custos: 5, 13, 24 e 30ms. O primeiro caminho tende a ser o mais rápido, uma vez que apenas uma chamada à função Dijkstra é realizada.

O segundo caminho possui um custo mais elevado, um pouco superior ao dobro do primeiro, uma vez que duas chamadas à função Dijkstra são realizadas, além da execução da rotina SEP, subrotina da FSP.

O terceiro, em geral, é mais custoso que o segundo, já que podem ser gerados até três caminhos candidatos, um em cada uma das partições: P_a, P_b, P_c, cada qual consumindo aproximadamente o mesmo que o P_2.

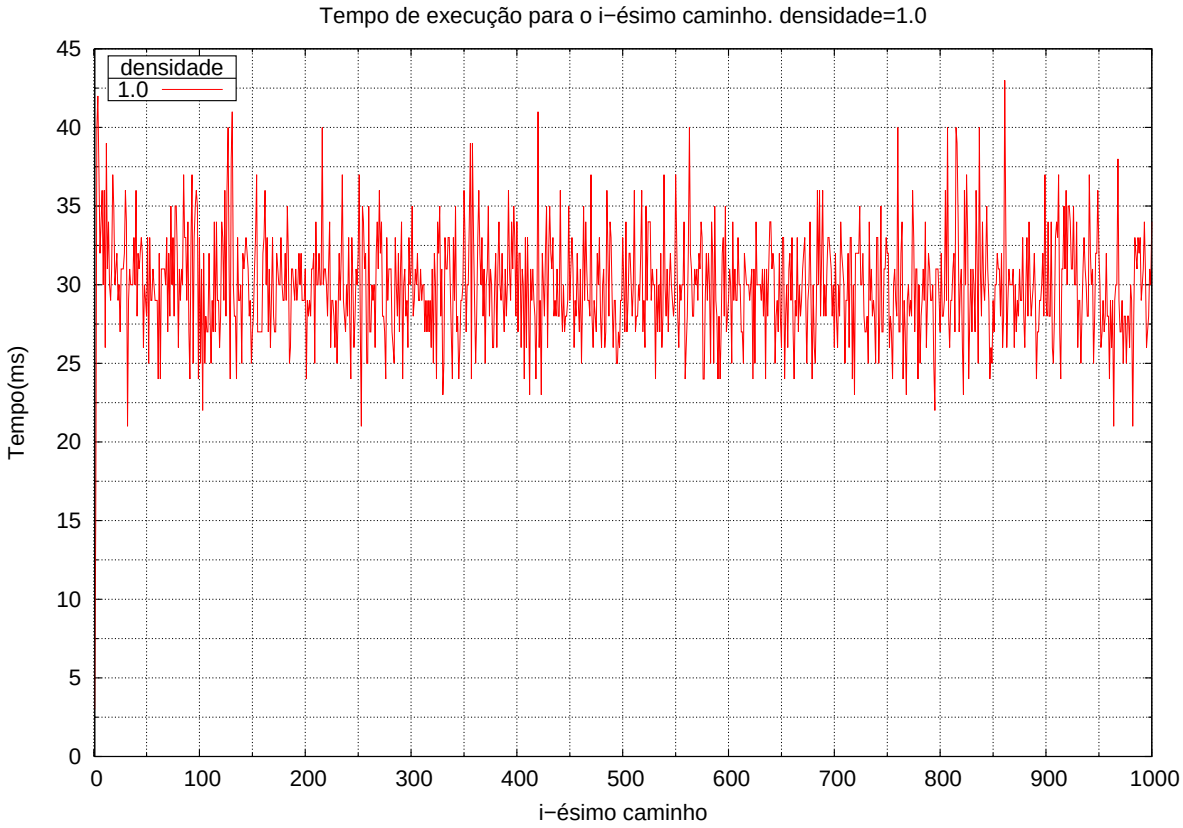
A partir de um certo i, o tempo começa a se estabilizar devido, principalmente, a diminuição do número de arestas e vértices no grafo usado na geração dos caminhos derivados. Lembramos que durante a execução do algoritmo, arestas e vértices são removidos do grafo original e as chamadas à função Dijkstra são executadas neste novo

grafo, a fim de evitar a geração de caminhos repetidos. Mesmo após a estabilização observamos certos picos, por exemplo para k=113 o tempo é 9ms.

Vale recordar que dados dois caminhos: P_k e P_j, com P_k derivado de P_j, ou seja, P_j é o caminho pai de P_k, nem sempre é possível gerar os três caminhos derivados: P_a, P_b, P_c, às vezes conseguimos gerar apenas dois, um ou até nenhum caminho derivado. Desta maneira os picos exibidos no gráfico correspondem aos pontos onde um número maior de caminhos derivados pôde ser calculado.

Da mesma forma, os vales podem ser explicados por pontos onde um menor número de caminhos candidatos pôde ser gerado. Por exemplo, para i=429 o tempo de execução foi de 2ms.

Vamos agora estudar o comportamento do algoritmo para densidades maiores: 0.5 e 1.0.

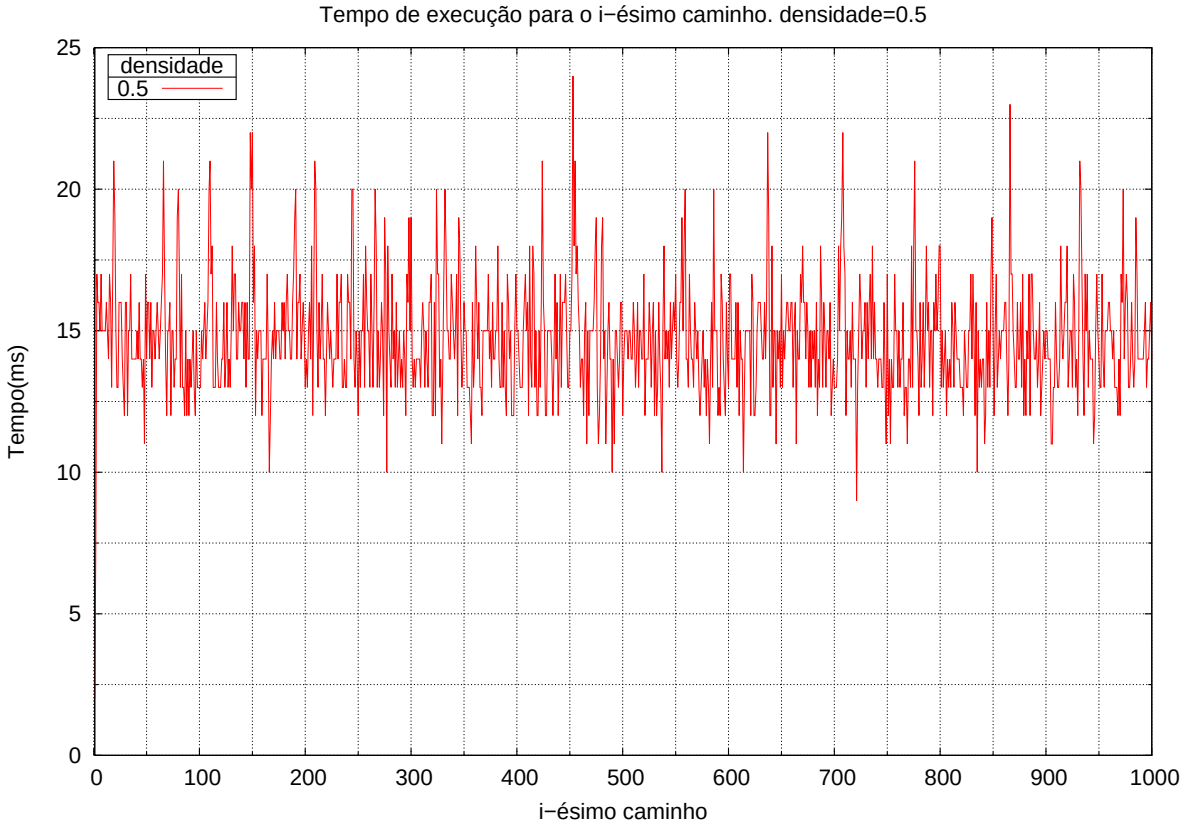


Os quatro primeiros caminhos consomem tempo: 3,15,40 e 42, respectivamente. O resultado fica dentro do esperado, seguindo a mesma explicação dada para o gráfico de densidade 0.1. O que vale notar é o aumento de catorze vezes entre o tempo do primeiro e quarto caminhos calculados.

Primeiramente o baixo custo do primeiro caminho advém justamente da completude do grafo. Com densidade 1, todos os vértices estão conectados dois a dois o que torna a rotina Dijkstra extremamente veloz. A alta do segundo caminho fica por conta das duas execuções do Dijkstra somadas a da função SEP, a qual aumenta seu consumo em função do número de arestas.

No cálculo dos próximos caminhos podemos ter até 6 chamadas à função Dijkstra e até 3 chamadas à função SEP, o que explica o aumento no consumo de tempo.

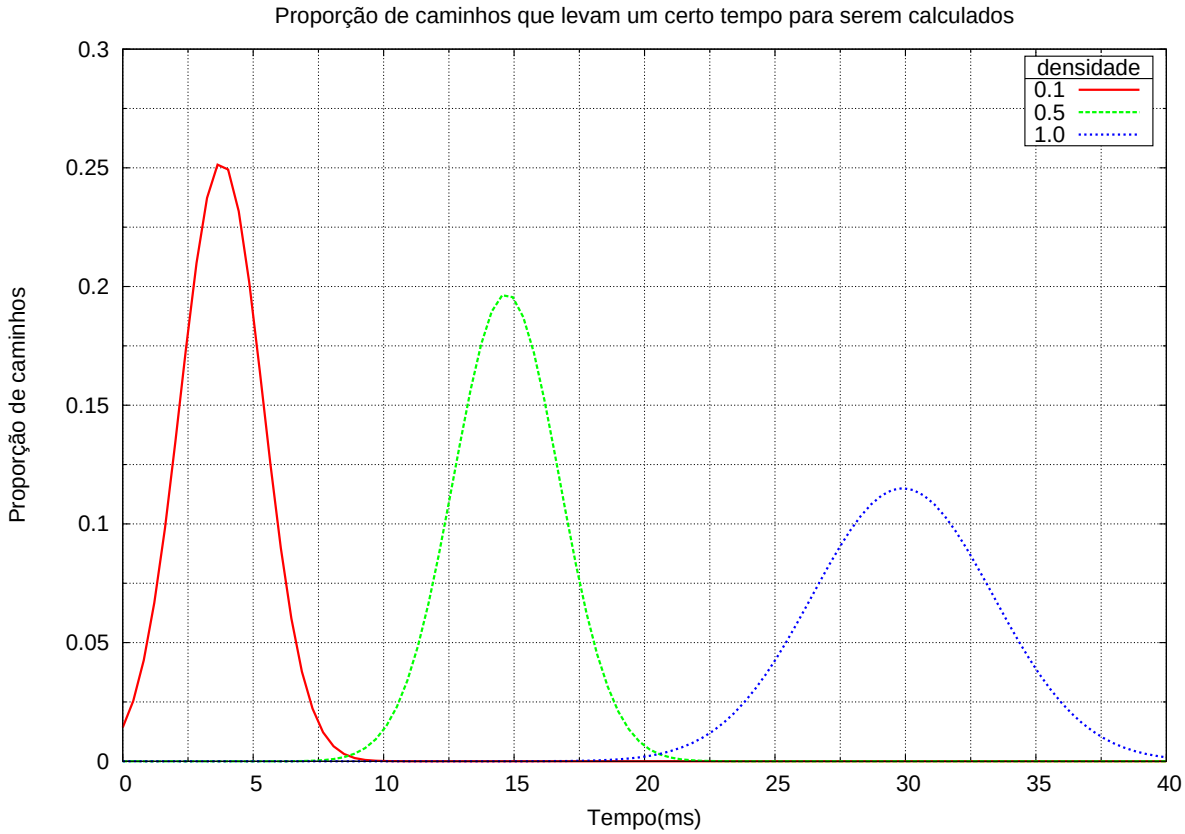
A partir de um certo ponto o tempo começa a se estabilizar, pela mesma razão dita anteriormente: exclusão de arestas e vértices acabam tornando a execução do algoritmo DIJKSTRA e da função SEP mais rápidas.



Utilizando a densidade 0.5, intermediária entre 0.1 e 1.0, não há muito o que ser explicado. Vemos novamente o aumento do tempo de execução para os quatro primeiros caminhos: 2,7,17 e 16.

Notamos, no entanto, um aumento menor entre os tempos de execução do primeiro e quarto caminhos, oito vezes.

Para permitir uma análise mais compacta, decidimos tirar as médias e desvios padrões de cada um dos conjuntos de dados usados nas gerações dos três gráficos anteriores e, com essas informações, construir um único gráfico exibindo três curvas normais.



Observando as três curvas da figura anterior, cada qual referente a uma densidade: 0.1, 0.5 e 1.0, notamos que a abertura aumenta com o aumento da densidade, ou seja, o custo de obtenção de cada caminho se torna mais variável. Basta observar que para a densidade 0.1 a maior parte dos caminhos é obtida com tempos entre 2.5 e 5, enquanto que para a densidade 1.0 está entre 25 e 35.

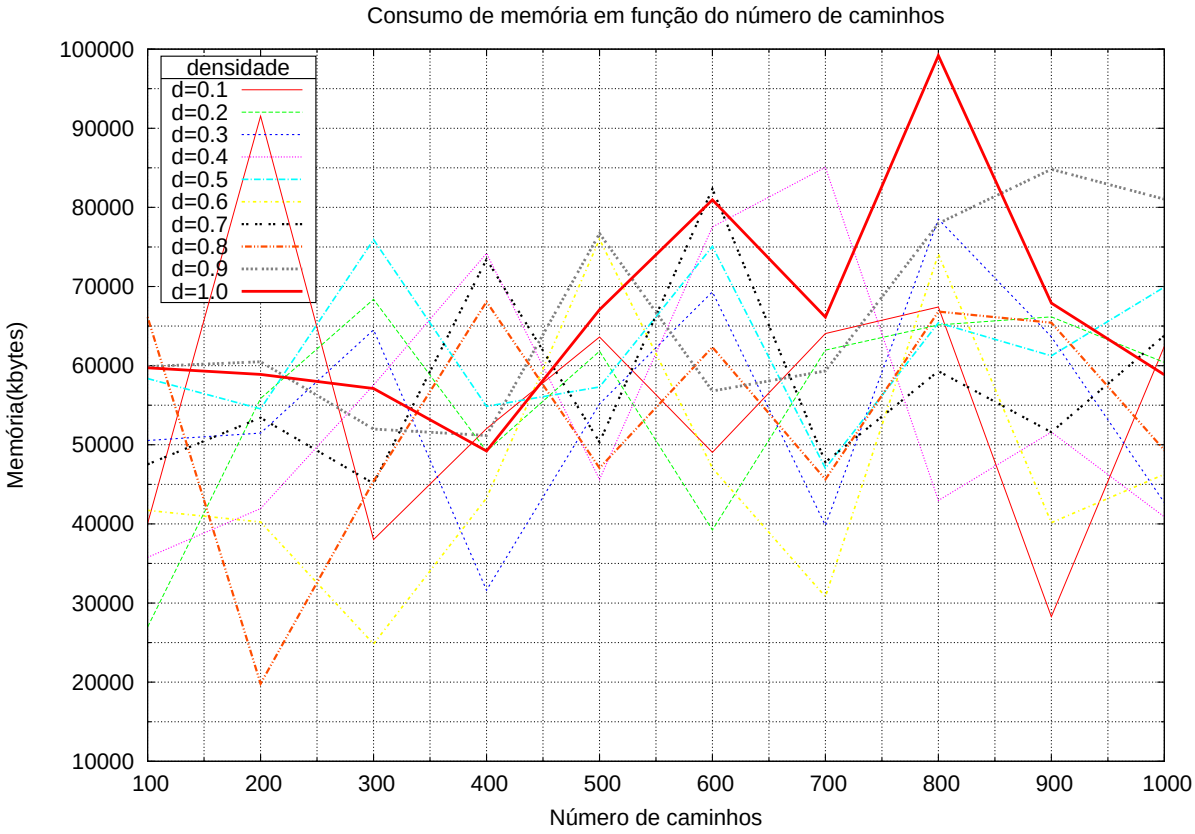
6.10 Consumo de memória

Para calcular o consumo de memória nos deparamos com diversas questões e obstáculos. Primeiramente, como nossa implementação está feita em Java teríamos que levar em conta o coletor de lixo, o qual é responsável por liberar espaço em memória removendo desta objetos não mais utilizados. Esse trabalho pode atrapalhar as nossas medidas de memória. Suponha que tenhamos alocado uma certa quantidade de memória para nossos testes e que antes de iniciar a execução do algoritmo tenhamos anotado a quantidade de memória disponível. Se o coletor de lixo for chamado durante a execução do algoritmo, é possível que a quantidade de memória disponível seja maior que a inicial e, por causa disso, se subtraíssemos uma da outra, com o intuito de obter a memória utilizada, chegaríamos a conclusão de que o processo gasta quantidade de memória negativa.

Surge então a pergunta: por que não desligar o coletor de lixo? De certa forma essa abordagem é justificável, mas não é possível desligá-lo por completo [2]. Além da dificuldade em desligá-lo, não consideramos correto contabilizar como memória gasta pelo algoritmo todo e qualquer uso temporário de memória.

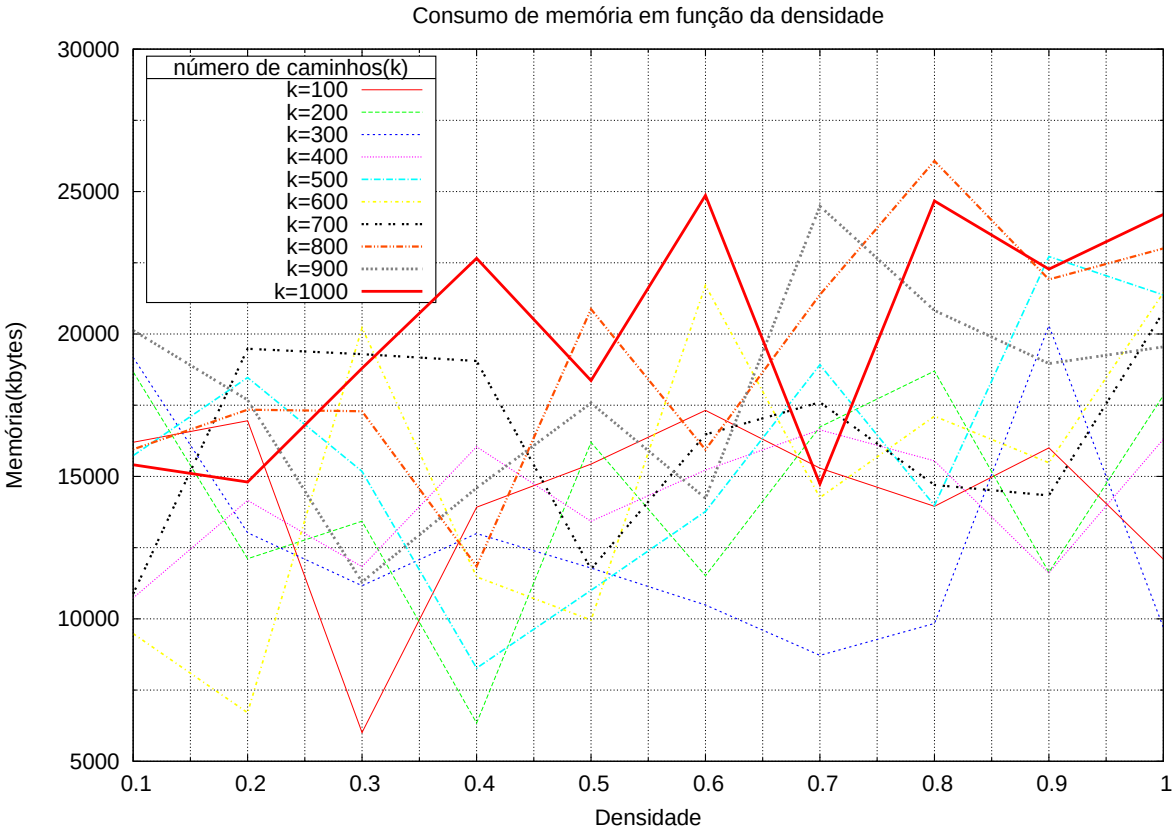
A seguir exibimos um gráfico com os consumos de memória de execuções do algoritmo para densidades de 0.1 a 1.0 com intervalo de 0.1.

[2] Estamos dizendo que não é possível, mas no entanto não temos certeza disso. Tentamos a opção -Xnoclassgc do Java a qual não desliga completamente o coletor de lixo.

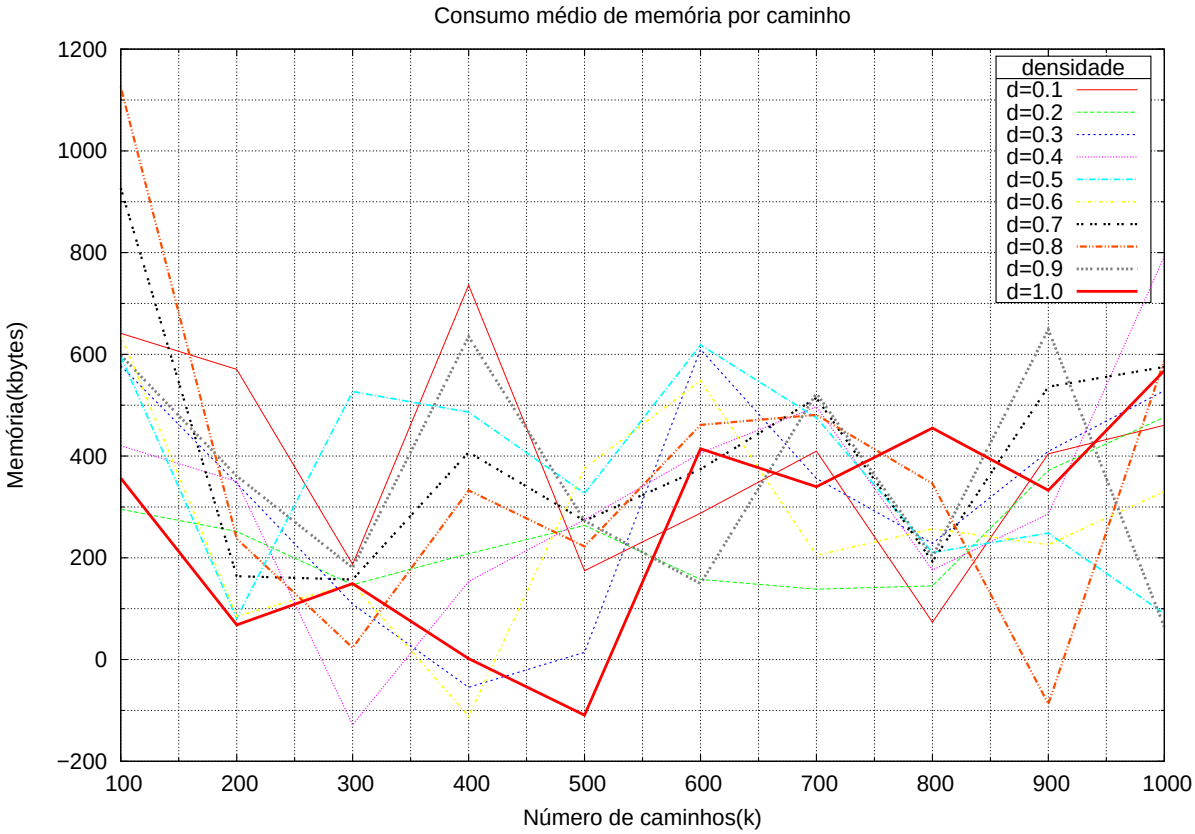


O gráfico serve bem o propósito de mostrar o quanto o trabalho do coletor de lixo atrapalha na nossa análise de consumo de memória. Basta observar como as linhas se cruzam, dificultando no entendimento da relação consumo de memória em função do número de caminhos para cada densidade. O consumo de memória oscila muito impedindo-nos de fazer muitas afirmações quanto as relações entre as variáveis apresentadas.

Exibimos o gráfico da figura anterior colocando a densidade no eixo x.



Devido a dificuldade em medir o consumo de memória entre o início e fim da execução do algoritmo, optamos por uma outra abordagem: calcular o uso de memória entre o início e fim da obtenção de cada um dos caminhos. Desta forma, acreditamos que ao diminuir o intervalo entre as medições sofreremos em menor escala os danos provocados pelas execuções do coletor de lixo. A seguir exibimos um gráfico com os consumos médios de memória por caminho em função do número de caminhos pedidos.



Cada curva representa uma densidade. Calculamos a quantidade de memória consumida para a obtenção de cada caminho e ao final dividimos pelo número de caminhos encontrados. Infelizmente, como nos casos anteriores, não podemos tirar muitas conclusões a partir deste último gráfico. Os pontos com uso negativo de memória se referem as chamadas do coletor de lixo do Java, o qual acaba liberando mais memória que no momento inicial da contagem havia, tornando a diferença entre inicial e final negativa.

CAPÍTULO 7

Considerações finais

7.1 Histórico

Meu primeiro contato com o *k*-MC foi através do problema apresentado na introdução. Inicialmente, estudei o algoritmo de Naoki Katoh, Toshihide Ibaraki e H. Mine [36] (KIM) de uma maneira não-acadêmica, objetivando implementá-lo. Mais tarde, decidimos usar o *k*-MC 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. Decidimos, nesta dissertação de mestrado, apresentar a motivação do estudo do algoritmo KIM para o problema *k*-MC, 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 [26] é classificado pelos autores como uma extensão do algoritmo KIM para grafos não simétricos. O grande mérito deste artigo, do nosso ponto de vista, não é o da apresentação de um novo algoritmo para o problema k-MC, mas sim pela descrição do problema k-MC 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-MC, 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-MC.

A descrição do método de YEN [48] é bem sucinta, mas foi suficiente para entendermos algumas idéias gerais. Debruçarmo-nos então sobre o artigo de KIM, tendo como bagagem o aprendizado ganho dos trabalhos de Yen e de Hershberger, Maxel e Suri, e o reescrevemos com base nos conceitos apresentados nos artigos citados.

7.2 k-menores passeios

Apesar de não ter sido considerado neste texto consideramos que devemos pelo menos mencionar o **problema do k-menores passeios**, denotado por k-MP, tendo em vias a sua semelhança com o k-MC:

k-MP

> **Problema** k-MP(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 k-menores **passeios** de s a t.

Os primeiros para esse problema foram propostos [48] por W. Hoffman e R. Pavley [30], R. Bellman e R. Kalaba [4] e M. Sakarovitch [44], entre outros.

Apesar desse problema parecer levemente diferente do k-MC, ele admite soluções muito mais eficientes. Em 1975, Bennett L. Fox [20] propôs um algoritmo para o k-MP que consome tempo $O(m + kn \log n)$, onde m é o número de arcos e n o número de vértices do grafo. David Eppstein [16] apresentou um algoritmo para o k-MP que

consome tempo $O(m + k + n \log n)$: o algoritmo computa uma representação implícita para os passeios e a partir dessa representação cada passeio pode ser explicitamente obtido em tempo $O(n)$. Para alguns grafos esse consumo de tempo pode ser na prática diminuído por um versão do algoritmo de Eppstein [32].

A técnica de David Eppstein [16] também se aplica para o próximo problema:

> **Problema** K-MC(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 **passeios** de s a t de custos não superiores a K. *K*-MC

Um problema similar, o de encontrar as *k*-**menores árvores geradoras**:

> **Problema** k-MAG(V, A, c, k): Dado um grafo simétrico (V, A), uma função custo c, e um inteiro positivo k, encontrar as k-menores árvores geradoras. *k*-MAG

também foi considerado por alguns pesquisadores [15].

7.3 Alguns limites

A tabela 7.1 mostra os limites inferiores e superiores de para alguns problemas de caminhos mínimos. Todos os limites inferiores utilizam o *modelo de comparação-adição*. Na tabela, o valor $m + \log n$ é devido a uma aplicação do algoritmo de Dijkstra com o *Fibonacci heap* de Michael L. Fredman and Robert Endre Tarjan [21]. Esse valor pode ser substituído por $T(n, m)$.

Problema	**Limite inferior**	**Limite superior**
caminho mínimo (CM)	$\Theta(m + n \log n)$ [21]	
All-pairs shortest path	$\Omega(nm)$ [35]	$O(n(m + n \log n))$ [21]
k-menores caminhos (k-MC)	$\Omega(\min\{n^2, m\sqrt{n}\})$ [29]	$O(kn(m + n \log n))$ [48, 49]
k-menores passeios (k-MP)	$\Theta(k + m + n \log n)$ [16]	
Replacement paths	$\Omega(\min\{n^2, m\sqrt{n}\})$ [29]	$O(n(m + n \log n))$ [29]

Figura 7.1: Limite inferiores e superiores para alguns problemas de caminhos mínimos.

7.4 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, 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 [24] 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 [41]. 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 [38] também está relacionado ao problema de reconstrução de árvores e mereceria um estudo também.

7.5 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 sua reutilização, 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 baseada no 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, usamos bastante as saídas gráficas de alguns pontos do algoritmo para ajudar-nos a identificar 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) 10

[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) 40

[3] Vladimir Batagelj, *Networks / Pajek*, `"http://vlado.fmf.uni-lj.si/pub/networks/pajek/"`. Citado na(s) página(s) 12

[4] R. Bellman e R. Kalaba, On *k*th best policies, *Journal of SIAM* **8** (1960), no. 4, 582–588. Citado na(s) página(s) 170

[5] Richard Ernest Bellman, On a routing problem, *Quartely of Applied Mathematics* **16** (1958), 87–90. Citado na(s) página(s) 62

[6] F. Bock, H. Kantner e J. Haynes, *An algorithm (The rth best path algorithm) for finding and ranking paths through a network*, Tech. report, Armour Research Foundation, Chicago, November 1957. Citado na(s) página(s) 63

[7] 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) 59

[8] Boris V. Cherkassky, Andrew V. Goldberg e Tomasz 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) 135

[9] Boris V. Cherkassky, Andrew V. Goldberg e Craig Silverstein, Buckets, heaps, lists and monotone priority queues, *SIAM J. Comput.* **28** (1999), 1326–1346. Citado na(s) página(s) 40, 135

[10] S. Clarke, A. Krikorian e J. Rausan, Computing the n best loopless paths in a network, *Journal of SIAM* **11** (1963), no. 4, 1096–1102. Citado na(s) página(s) 63

[11] William J. Cook, William H. Cunningham, Willian R. Pulleyblank e Alexander Schrijver, *Combinatorial otimization*, Wiley-Interscience series in discrete mathematics and optimization, John Wiley & Son's, New York, 1998. Citado na(s) página(s) 8

[12] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest, *Introduction to algorithms*, McGraw-Hill, 1999. Citado na(s) página(s) 10

[13] 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) 29, 41, 48

[14] Edsger Wybe Dijkstra, A note on two problems in connection with graphs, *Numerische Mathematik* **1** (1959), 269–271. Citado na(s) página(s) 25, 26, 31, 48

[15] David Eppstein, *Finding the k smallest spanning trees*, texttttthttp://www.ics.uci.edu/~eppstein/pubs/Epp-BIT-92.pdf. Citado na(s) página(s) 171

[16] ———, Finding the k shortest paths, *SIAM Jounal on Computing* **28** (1998), no. 2, 652–673. Citado na(s) página(s) 59, 170, 171

[17] Paulo Feofiloff, *Algoritmos de programação linear*, Addison-Wesley, 2000. Citado na(s) página(s) 27

[18] ———, *Fluxo em redes*, `"http://www.ime.usp.br/~pf/flows/"`, 2003. Citado na(s) página(s) 5, 25, 31

[19] Lester Randolph Ford, *Network flow theory*, Tech. Report Paper P-923, RAND Corporation, Santa Monica, California, 1956. Citado na(s) página(s) 62

[20] Bennett L. Fox, *k*-th shortest paths and applications to the probabilistic networks, *ORSA/TIMS Joint National Meeting*, vol. 23, 1975, p. B263. Citado na(s) página(s) 170

[21] 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) 39, 40, 41, 48, 171

[22] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, *Design patterns: Elements of reusable object-oriented software*, EDUSP, 1995. Citado na(s) página(s) 23

[23] A.V. Goldberg e C. Silverstein, *Implementation of Dijkstra's algorithm based on multi-level buckets*, Tech. report, NEC Research Institute, Princeton, NJ, 1995. Citado na(s) página(s) 135

[24] 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) 59, 83, 135, 138, 172

[25] John Hershberger, Matthew Maxel e Subhash Suri, Finding the *k* shortest simple paths: A new algorithm and its implementation, *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments (ALENEX)* (Baltimore), 2003, pp. 11–14. Citado na(s) página(s) 64, 65, 75

[26] _____, Finding the *k* shortest simple paths: A new algorithm and its implementation, *ACM Transactions on Algorithms* **3** (2007), no. 4, Article 45, `http://doi.acm.org/10.1145/1290672.1290682`. Citado na(s) página(s) 56, 59, 64, 65, 75, 80, 83, 170

[27] John Hershberger e Subhash Suri, Vickrey prices and shortest paths: What is an edge worth?, *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, 2001, pp. 252–259. Citado na(s) página(s) 65, 75

[28] _____, Erratum to "vickrey prices and shortest paths: What is an edge worth?", *Proceedings of the 43rd IEEE symposium on Foundations of Computer Science*, 2002, p. 809. Citado na(s) página(s) 65, 75

[29] John Hershberger, Subhash Suri e Amit Bhosle, On the difficulty of some shortest path problem, *ACM Trasactions on Algorithms* **3** (2007), no. 1, Article 5, texttttthttp://doi.acm.org/10.1145/1219944.1219951. Citado na(s) página(s) 171

[30] W. Hoffman e R. Pavley, A method for the solution of the *n*th best problem, *Journal of the ACM* **6** (1959), no. 4, 506–514. Citado na(s) página(s) 64, 170

[31] Shigueo Isotani, *Algoritmos para caminhos mínimos*, Master's thesis, Instituto de Matemática e Estatística, 2002. Citado na(s) página(s) 25, 48

[32] Víctor M. Jiménez e Andrés Marzal, A lazy version of Eppstein's K shortest paths algorithm, *2nd International Workshop on Experimental and Efficient Algoritms (WEA)* (Springer, ed.), Lecture Notes in Computer Science, vol. 2647, 2003, pp. 179–190. Citado na(s) página(s) 171

[33] David 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) 135

[34] Donald B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977), 1–13. Citado na(s) página(s) 38

[35] David R. Karger, Daphne Koller e Steven J. Phillips, Finding the hidden path: Time bound for all-pairs shortest path, *SIAM Journal on Computing* **22** (1993), 1199–1217. Citado na(s) página(s) 171

[36] 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, 59, 64, 65, 80, 83, 169

[37] 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) 59

[38] 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) 172

[39] 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) 59

[40] 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) 59

[41] 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) 172

[42] A. Perko, Implementation of algorithms for *k* shortest loopless paths, *Networks* **16** (1986), 149–160. Citado na(s) página(s) 59

[43] M. Pollack, The *k*th best route through a network, *Operations Research* **9** (1961), no. 4, 578. Citado na(s) página(s) 63

[44] M. Sakarovitch, *The k shortest routes and the k shortest chains in a graph*, Tech. Report Report ORC-32, Operations Research Center, University of California, Berkeley, October 1966. Citado na(s) página(s) 64, 170

[45] Alexander Schrijver, *Combinatorial optimization: Polyhedra and efficiency*, Algorithms and Combinatorics, vol. 24, Springer, 2003. Citado na(s) página(s) 8

[46] 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) 29

[47] 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) 48

[48] 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) 47, 49, 59, 64, 83, 170, 171

[49] ———, Another algorithm for finding the *k* shortest loopless network path, *Proceedings of the 41st Meeting of the Operations Research Society of America*, vol. 20, 1972. Citado na(s) página(s) 47, 64, 171

Índice Remissivo