Caminhos mínimos com recursos limitados

Joel Silva Uchoa

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre

Programa: Ciência da Computação

Orientador: Prof. Dr. Carlos Eduardo Ferreira

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, agosto de 2012

Caminhos mínimos com recursos limitados

Esta dissertação trata-se da versão original
do aluno (Joel Silva Uchoa).

Agradecimentos

Primeiramente gostaria de agradecer a Deus, que me deu forças e condições para concretizar este trabalho. Ele tem sido meu guia, autor e consumador da minha fé. Grande inspirador e fortalecedor da minha vida, companheiro em todos os momentos.

À minha amada esposa Tainara Alcântara, pelos inúmeros puxões de orelha, que sempre soaram como incentivo, pela compreensão, amor e carinho oferecidos a mim sem medidas.

À minha família, por sua capacidade de acreditar e investir em mim, e por suas incessantes orações. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir.

Aos meus amigos, pelas alegrias, tristezas, derrotas, conquistas e conhecimentos compartilhados. Em especial a meus grandes amigos, Wanderley Guimarães e Márcio Oshiro, pelo companheirismo e ajuda em todos os momentos.

Um agradecimento a todos os professores que tive no IME-USP, em especial ao professor Carlos Eduardo Ferreira. Em seu papel de orientador, teve uma paciência fenomenal e tentou exaustivamente me ensinar e ajudar de todas as formas possíveis.

Resumo

Caminhos mínimos com recursos limitados

O problema de caminhos mínimos (SP – *shortest path problem*) é um dos problemas fundamentais da computação. Vem sendo estudado com profundidade e há uma grande quantidade de publicações a respeito do mesmo. Inclusive, são conhecidos várias soluções eficientes (algoritmos de tempo polinomial) para o problema. O SP é frequentemente colocado em prática em uma grande variedade de aplicações em diversas áreas, não somente em computação. Nessas aplicações geralmente se deseja realizar algum tipo de deslocamento ou transporte entre dois ou mais pontos específicos em uma rede. Tal ação deve ser executada de forma ótima em relação a algum critério, por exemplo o menor custo possível, ou o menor gasto de tempo ou o máximo de confiabilidade/segurança.

Conforme essas soluções para o SP foram sendo apresentadas, novas necessidades foram levantadas e surgiram variações do problema para modelar tais necessidades. Uma dessas variantes, advém do fato de que, na prática, muitas vezes não desejamos apenas o menor custo ou o menor tempo, mas desejamos otimizar uma combinação de diferentes critérios, por exemplo, um caminho que seja rápido e barato. Este é conhecido como o problema de caminhos mínimos multi-objetivo. Como não é possível otimizar sobre todos os critérios de uma só vez, nós escolhemos um dos critérios para representar a função custo, que será minimizada, e para os demais critérios representamos como recursos e definimos os limites que julgamos aceitáveis para o consumo de cada um desses recursos. Esta variação é chamada de problema de caminhos mínimos com restrições por recursos, ou como preferimos chamar, **problema de caminhos mínimos com recursos limitados** (RCSP – *resource constrained shortest path problem*), o qual será o objeto de estudo neste trabalho.

A adição de restrições por recursos no SP, infelizmente torna o problema $\mathcal{NP}$-difícil, mesmo em grafos acíclicos, com restrições sobre um único recurso, e com todos os consumos de recursos positivos. Temos reduções dos famosos problemas $\mathcal{NP}$-difíceis Mochila e Partição para o nosso problema.

Em contextos diversos são encontrados problemas de cunho teórico e prático que podem ser formulados como problemas de caminhos mínimos com recursos limitados, o que nos motivou a estudá-lo a fim de desenvolver um trabalho que resumisse informações suficientes para auxiliar pesquisadores ou desenvolvedores que tenham interesse no problema. Nós apresentamos aqui, uma detalhada revisão bibliográfica do RCSP, tendo como foco o

desenvolvimento de algoritmos exatos para o caso onde possuímos um único recurso e a implementação e comparação dos principais algoritmos conhecidos, observando-os em situações práticas.
Palavras-chave: Otimização combinatória, caminhos mínimos com restrições.

Abstract

Resource constrained shortest path

The problem of choosing a route to a trip, where we want minimize the distance of the path is a major problem in computing. In this basic form, this is the shortest path problem. But sometimes, besides the length we need to consider more parameters for selecting a good path. A common parameters to consider is the consumption of resources in a limited budget. A shortest path with these additional constraints is called resource constrained shortest path - RCSP.

This paper has two main objectives: to present a literature review of the problem RCSP, focusing on exact algorithms for the case where we have a single resource, and implement and compare some algorithms, observing them in practical situations.

The Shortest Path (SP) problem is among the fundamental problems of computer science. It's been deeply studied and subject of many publications. Also, many efficient solutions (polynomial time algorithms) are known for this problem. The SP is widely applied in many fields of science, not only computer science. These situations usually need to transport a load between two or more specific spots of a network. This action must be taken optimally regarding to some criterion, for instance the least cost, or the least time or maximum reliability.

While new solutions for SP were presented, new demands were issued too, with new variations for the problem. One of these variations comes from the fact that, in a real scenario, a combination of many criteria must be optimized, for instance a path with least cost and least time. This problem is known as Multiobjective Shortest Path. Since it's not possible to optimize all criteria at once, one of them is chosen to represent the cost function to be minimized and the others to represent resources with defined boundary. This variation, known as Resource Constrained Shortest Path (RCSP), was the object of the present study.

By adding resource constraints, the SPbecomes $\mathcal{NP}$-hard, even in acyclic graphs with only one resource constrained and all resource consumption being positive. There are reductions from the famous NP-hard problems Knapsack and Partition to our problem.

In many fields, are found theoretical and practical problems that may be expressed as a Resource Constrained Shortest Path Problem, which motivated us to study this problem in order to summarize enough information to researchers and developers involved with this problem. This paper presents a detailed bibliographic revision to RCSP, focusing on the

development of exact algorithms for the case when there is only only one resource and on the implementation and comparison of the main known algorithms in practical situations.
Keywords: Combinatorial Optimization, shortest path with constraints.

Sumário

Lista de Abreviaturas

SP	Problema do caminho mínimo. (*[single-source, single-sink] shortest path problem*)
CSP	Problema do caminho mínimo com restrições. (*constrained shortest path problem*)
RCSP	Problema do caminho mínimo com recursos limitados. (*resource constrained shortest path problem*)
SRCSP	RCSP com um único recurso. (*single resource constrained shortest-path problem*)
VCSP	Problema do caminho mínimo com restrições sobre vértices. (*vertex constrained shortest-path problem*)
TCSP	Problema do caminho mínimo com restrições de tempo. (*time constrained shortest-path problem*)
FPTAS	Esquema de aproximação totalmente polinomial. (*fully polinomial-time approximation scheme*)

Lista de Símbolos

λ Limite de recursos disponível para uma instância do RCSP.

ψ Função predecessor.

$\mathcal{X}$ Conjunto de soluções viáveis para o problema (P) (seção 3.8).

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O problema de caminhos mínimos (SP – *shortest path problem*) é um dos problemas fundamentais da computação. Vem sendo estudado com profundidade e há uma grande quantidade de publicações a respeito do mesmo. Inclusive, são conhecidas várias soluções eficientes (algoritmos de tempo polinomial) para o problema. O SP é frequentemente colocado em prática em uma grande variedade de aplicações em diversas áreas, não somente em computação. Nessas aplicações geralmente se deseja realizar algum tipo de deslocamento ou transporte entre dois ou mais pontos específicos em uma rede. Tal ação deve ser executada de forma ótima em relação a algum critério, por exemplo o menor custo possível, ou o menor gasto de tempo ou o máximo de confiabilidade/segurança.

Conforme essas soluções para o SP foram sendo apresentadas, novas necessidades foram levantadas e surgiram variações do problema para modelar tais necessidades. Uma dessas variantes advém do fato de que na prática, muitas vezes não desejamos apenas o menor custo ou o menor tempo, mas desejamos otimizar uma combinação de diferentes critérios, por exemplo, um caminho que seja rápido e barato. Este é conhecido como o problema de caminhos mínimos multi-objetivo. Como não é possível otimizar sobre todos os critérios de uma só vez, nós escolhemos um dos critérios para representar a função custo, que será minimizada, e os demais critérios representamos como recursos e definimos os limites que julgamos aceitáveis para o consumo de cada um deles [1]. Esta variação é chamada de problema de caminhos mínimos com restrições de recursos, ou como preferimos chamar, **problema de caminhos mínimos com recursos limitados** (RCSP – *resource constrained shortest path problem* [2]), o qual será o objeto de estudo neste trabalho.

A adição de restrições por recursos no SP torna o problema $\mathcal{NP}$-difícil, mesmo em grafos acíclicos, com restrições sobre um único recurso, e com todos os consumos de recursos positivos. Temos reduções dos famosos problemas $\mathcal{NP}$-difíceis Mochila e Partição para

[1] Restrições deste tipo, onde temos o consumo de recursos em um orçamento que limita a quantidade disponível destes recursos são chamadas de *knapsack constraints* (Borndörfer, 2009).

[2] Beasley e Christofides (1989) foi um dos primeiros a chamar o problema desta forma, antes disso era comum utilizar apenas CSP (constrained shortest path problem). A versão com um único recurso pode ser referenciada como SRCSP(*single* RCSP), ou ainda como WCSP (*weight constrained shortest path problem*) (Dumitrescu e Boland, 2003).

o nosso problema.

Em contextos diversos são encontrados problemas de cunho teórico e prático que podem ser formulados como problemas de caminhos mínimos com recursos limitados, o que nos motivou a estudá-lo a fim de desenvolver um trabalho que resumisse informações suficientes para auxiliar pesquisadores ou desenvolvedores que tenham interesse no problema. Nós apresentamos aqui, uma detalhada revisão bibliográfica do RCSP, tendo como foco o desenvolvimento de algoritmos exatos para o caso onde possuímos um único recurso e a implementação e comparação dos principais algoritmos conhecidos, observando-os em situações práticas.

1.1 Aplicações

O problema de caminhos mínimos com recursos limitados pode ser aplicado em uma imensa quantidade de problemas práticos. Esta seção vai descrever algumas destas aplicações.

1.1.1 Qualidade de serviço em redes de computadores

A qualidade de serviço (QoS – *quality of service*) é um aspecto importante para as redes de pacotes como um todo e para as redes IP em particular. Tem um valor fundamental para o desempenho de determinadas aplicações fim-a-fim, como vídeo e áudio conferência e transferência de dados.

Existem redes de computadores que estão oferecendo garantias de QoS para diversas aplicações. Estas aplicações possuem diversos requisitos que precisam ser atendidos para o seu bom funcionamento. Como exemplo de requisitos podemos citar largura mínima de banda, tempo máximo de atraso e quantidade máxima de perda de pacotes. O problema em que estamos interessados, ou seja, serviços baseados em QoS, é determinar uma rota que usa os recursos da rede de forma eficiente e satisfaz, ao mesmo tempo, requisitos como qualidade de conexão. Este problema é conhecido como roteamento baseado em QoS (Aurrecoechea *et al.*, 1998).

Formalizando um pouco melhor o problema, podemos representar nossa rede como um grafo direcionado $G = (V, A)$, onde V é o conjunto de vértices e A é o conjunto de arcos. Cada arco $uv \in A$ é associado com um custo c_{uv} e M pesos não negativos w^k_{uv}, $k = 1, 2, \cdots, M$ que representam a valoração dos requisitos no arco (ambos, pesos e custos são aditivos em um caminho). Dada uma aplicação que exige M requisitos com valoração máxima R^k , $k = 1, 2, \cdots, M$, o problema é encontrar um caminho P da origem s até o destino t que respeita os requisitos e que minimiza o custo total do caminho. Problema este que pode ser modelado da seguinte forma:

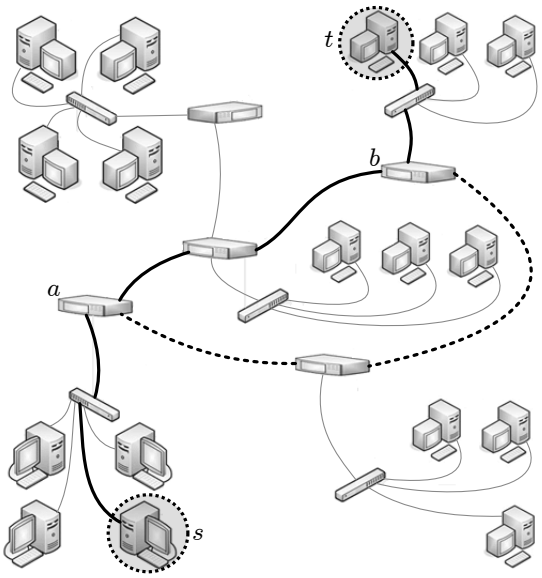


Figura 1.1: *Representação de uma rede de computadores. Os segmentos pretos e contínuos compõem uma rota entre os computadores s e t. Substituindo-se o trecho entre os roteadores a e b pelo trecho pontilhado, temos uma outra rota que pode ser usada para a comunicação entre s e t. Dependendo das propriedades destas rotas e das necessidades dos usuários, uma ou outra pode ser mais apropriada para o uso.*

$$\begin{aligned} \text{minimize} \quad & c(P) = \sum_{uv \in P} c_{uv} \\ \text{sujeito a} \quad & \sum_{uv \in P} w_{uv}^k \leq R^k \qquad \text{para } k = 1, \ldots, M \end{aligned}$$

Este problema é uma aplicação direta do problema de caminhos mínimos com recursos limitados. É comum, neste tipo de aplicação, querermos uma rota passando pelo menor número de vértices possível, neste caso a função de custo possui valor unitário para todos os arcos. É comum também existirem restrições que não são aditivas pelo caminho, mesmo com este tipo de restrição as soluções para o RCSP podem ser aplicadas, podemos contorná-las geralmente com algum preprocessamento ou pequenas alterações nos algoritmos.

1.1.2 Roteamento de tráfego de veículos

Quando precisamos nos deslocar de um ponto a outro em uma rede de tráfego veicular é natural nos preocuparmos com uma série de fatores a respeito da rota escolhida para o trajeto. Geralmente desejamos percorrer o caminho no menor tempo possível dentro de determinadas restrições, como consumo de combustível, gastos com pedágio e distância máxima percorrida. Outras restrições relevantes, são por exemplo, segurança e possibilidade de congestionamentos (essas características são mais subjetivas, geralmente baseadas em dados estatísticos).

Dentro deste contexto, Jahn *et al.* (2005) propõem uma aplicação interessante que usa o RCSP como subproblema. Nesta aplicação, não se deseja minimizar apenas o tempo de

percurso de cada usuário individualmente, o objetivo é minimizar o tempo total de percurso do sistema como um todo. Com a função objetivo da forma que foi descrita acima, é possível que para atingir a eficiência global, alguns usuários precisem realizar caminhos muito piores do que realizariam caso utilizassem uma estratégia egoísta. Assim, em um sistema de apoio à decisão, poderia acontecer dos usuários não seguirem as recomendações apresentadas. Pensando nisto, Jahn *et al.* (2005) aplicam uma restrição para que o caminho de cada usuário não seja demasiadamente pior do que o melhor caminho possível. Desta forma, é possível obter uma solução aceitável para cada usuário, que objetiva aumentar a eficiência global do sistema [3].

Hoje em dia existem sistemas de informação e orientação projetados para auxiliar motoristas a tomarem decisões de rota. Vamos idealizar um sistema que pode fornecer informações como congestionamentos, bloqueios ou acidentes, e dar recomendações baseado nestes dados. O motorista cadastraria suas preferências, entraria com o seu destino e o sistema calcularia a rota baseado em mapas digitais, preferências do usuário, especificidades do veículo (dimensões, quantidade de combustível disponível e consumo, por exemplo) e nas condições atuais do trânsito. Sistemas deste tipo poderiam ter o seguinte cenário, as condições de tráfego seriam obtidas através de sensores e transmitidas a uma central de controle de tráfego, que por sua vez, receberia, através do computador de bordo do carro, as preferências dos usuários, as características dos veículos, as posições atuais e seus destinos, podendo assim fazer certas distribuições de rotas aos motoristas.

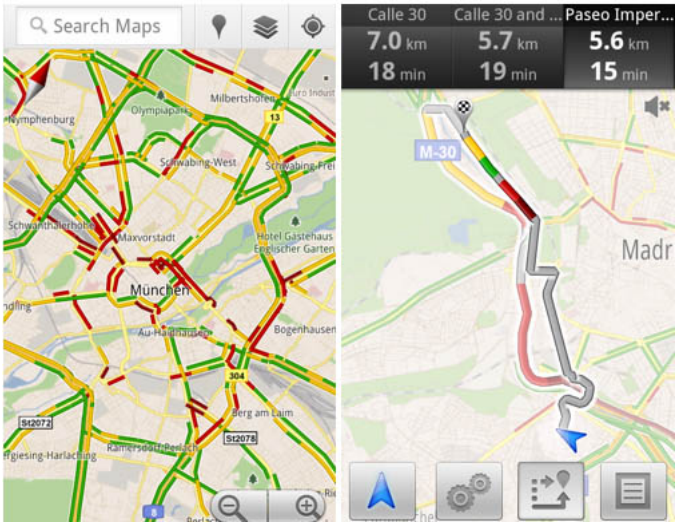


Figura 1.2: *Exemplo de um sistema de orientação para motoristas. No lado esquerdo podemos ver para uma determinada região o tráfego em cada trecho (vermelho, amarelo e verde representam respectivamente tráfego pesado, médio e leve). No lado direito temos a representação de uma rota, começando no triângulo azul (posição atual do veículo) e terminando na marcação com um circulo quadriculado xadrez.(Figura retirada de Google Mobile Blog – http://googlemobile.blogspot.com.br/2011/07/live-traffic-information-for-13.html)*

[3] Soluções onde se atribui o caminho mais rápido para cada usuário sob as condições correntes, são chamadas de solução *user optimal* ou *user equilibrium*. Soluções onde minimiza-se o tempo total do sistema, são chamadas de *system optimal*.

Descrevemos todo um conjunto de restrições complexo, porém, na nossa formulação levaremos em consideração apenas os níveis de congestionamento e um limitante superior para a degradação de um caminho em relação ao melhor caminho (consideramos um caminho viável apenas se este é mais demorado que o caminho mais rápido até um certo limite).

Representamos nossa rede rodoviária por um grafo direcionado $G = (V, A)$ com dois atributos em cada arco $uv \in A$: $\tau_{uv} \geq 0$ que representa uma estimativa do tempo de travessia quando não há congestionamento; e uma função $l_{uv}(x_{uv})$ (x é um fluxo na rede, x_{uv} é a parte deste fluxo correspondente ao arco uv) que computa uma estimativa do tempo de travessia do arco uv considerando o fluxo dado.

Nós modelamos os veículos que possuem a mesma origem e destino como um par $k = (s, t)$, definimos K como o conjunto de todos estes pares. Podemos representar cada $k \in K$ como (s_k, t_k). Definimos a demanda $d_k > 0$ para cada $k \in K$ como sendo a quantidade de fluxo a ser roteada através de k (veículos por unidade de tempo). Denotamos todos os caminhos para o par k por $\mathcal{P}_k = \{P \mid P \text{ é um caminho de } s_k \text{ até } t_k\}$, e o conjunto completo de caminhos por $\mathcal{P} = \cup_{k \in K} \mathcal{P}_k$. Para um caminho $P \in \mathcal{P}$, o tempo para percorrermos P, dado um fluxo x representando o estado atual da rede, é $l_P(x) = \sum_{uv \in P} l_{uv}(x_{uv})$, o tempo estimado de percurso sem considerar congestionamento é $\tau_P = \sum_{uv \in P} \tau_{uv}$.

Definimos um fator de tolerância $\varphi \geq 1$. Através deste fator, assumimos que para um caminho $P \in \mathcal{P}_k$ ser viável, $\tau_P \leq \varphi T_k$, onde $T_k = \min_{P \in \mathcal{P}_k} \tau_P$ é o menor tempo possível para se partir de s_k e chegar a t_k desconsiderando-se o fluxo na rede. Assim podemos denotar $\mathcal{P}_k^\varphi$ como o conjunto de todos os caminhos viáveis que partem de s_k e terminam em t_k, e $\mathcal{P}^\varphi = \cup_{k \in K} \mathcal{P}_k^\varphi$ como sendo o conjunto de todos os caminhos viáveis para os pares em K.

O sistema ótimo com restrições (CSO – *constrained system optimum*) proposto, pode ser modelado como o seguinte fluxo multicomodidade de custo mínimo (*min-cost multicommodity flow*):

$$
\begin{aligned}
\text{minimize} \quad & C(x) = \sum_{uv \in A} l_{uv}(x_{uv}) \cdot x_{uv} \\
\text{sujeito a} \quad & \sum_{P \in \mathcal{P}_k^\varphi} x_P = d_k && \text{para } k \in K \\
& \sum_{P \in \mathcal{P}^\varphi \mid uv \in P} x_P = x_{uv} && \text{para } uv \in A \\
& x_P \geq 0 && \text{para } P \in \mathcal{P}^\varphi
\end{aligned}
$$

Jahn *et al.* (2005) usam um algoritmo de geração de colunas para resolver o problema CSO (para uma descrição do método ver Frank e Wolfe (1956) e LeBlanc *et al.* (1985)). Neste algoritmo, surge como sub-problema computar um caminho ótimo em $\mathcal{P}_k^\varphi$ que é precisamente o RCSP: Neste caso, cada arco $uv \in A$ tem dois parâmetros, o tempo de travessia l_{uv} e o comprimento τ_{uv}. Dado um par (s, t), o objetivo é computar um caminho mais rápido

de s a t cujo tamanho não excede a um dado limite T. Ou seja, o problema é

$$\min\{l_P \mid P \text{ é uma caminho de } s \text{ até } t \text{ e } \tau_p \leq T\},$$

onde $l_P = \sum_{uv \in P} l_{uv}$ e $\tau_P = \sum_{uv \in P} \tau_{uv}$.

1.1.3 Compressão de imagens (aproximação de curvas)

Uma curva/função é linear por partes (*piecewise linear curve*) se pudermos subdividi-la em intervalos que são lineares. Este tipo de curva/função é frequentemente usado para aproximar funções complexas ou objetos geométricos.

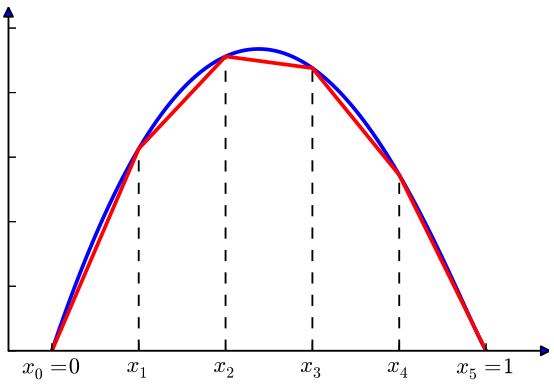


Figura 1.3: *Exemplo de uma função linear por partes. A função em azul é uma função não linear, e a função em vermelho é uma aproximação da primeira que atende a nossa definição de curva linear por partes.*

O uso dessas curvas é muito comum em áreas como computação gráfica, programação matemática, processamento de imagens e cartografia. Curvas lineares por partes são populares porque são fáceis de se criar e manipular, além de fornecerem, em geral, aproximações suficientemente boas para os problemas estudados.

Aplicações nas áreas citadas no parágrafo anterior, frequentemente incluem uma enorme quantidade de dados (as curvas em geral possuem uma grande quantidade de partes ou pontos de quebra). Isto causa dificuldades, por exemplo, com o espaço de armazenamento, taxa de transmissão ou tempo gasto para renderizar a curva em um dispositivo gráfico. Isto naturalmente nos faz pensar no problema de redução/compressão de dados, onde nós queremos determinar uma nova curva linear por partes que é tão aproximada quando possível da original, mas tem um número pequeno de pontos de quebra.

Dahl *et al.* (1996); Nygaard *et al.* (1998) estudaram este problema e mostraram como ele poderia ser modelado como um problema de caminhos mínimos com recursos limitados: Os pontos de quebra $v_1, v_2, \cdots, v_{n-1}, v_n$ são os vértices do grafo $G = (V, A)$ e para todo para $1 \leq i \leq j \leq n$ nós temos um arco $v_i v_j$. O custo de um arco c_{uv} é o erro introduzido na aproximação por tomar o "atalho" indo direto de u para v ao invés da curva original[4].

[4]Existem diversos tipos de métricas que podem ser usadas para calcular este erro.

O recurso de uma aresta r_{uv} é 1 para $uv \in A$. Agora, nós podemos computar a melhor aproximação usando no máximo k pontos de quebra [5].

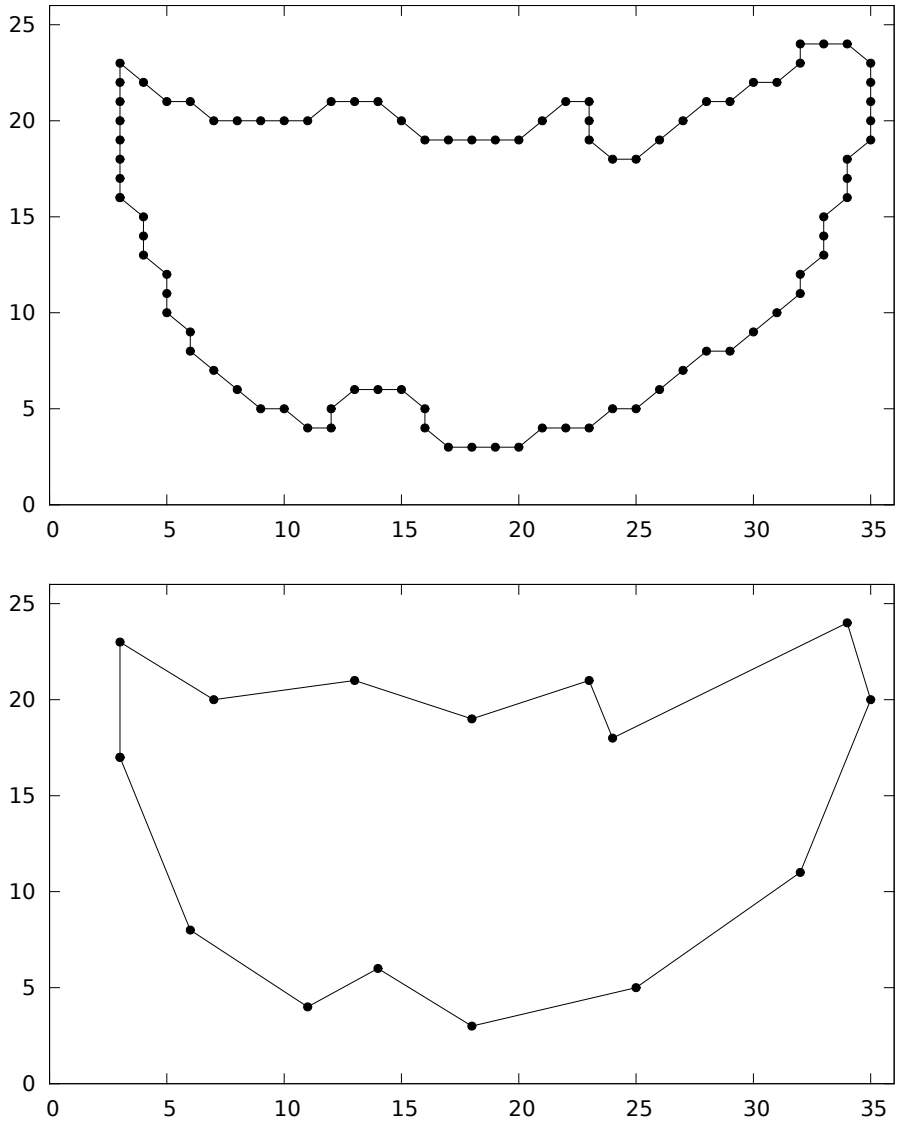


Figura 1.4: *Exemplo de uma curva e sua aproximação. A curva original possui* 90 *pontos, enquanto a sua aproximação possui* 15 *pontos.*

1.2 Objetivos

Os principais objetivos deste trabalho, de forma sucinta são:

- levantar um conjunto de referências bibliográficas relevantes, cobrindo o máximo possível de variações e aplicações do problema;
- apresentar o RCSP e suas diversas abordagens com uma notação padronizada;

[5] Alternativamente, nós podemos limitar o erro de aproximação e computar o menor número de pontos de quebra.

- implementar um subconjunto dos principais algoritmos conhecidos;
- avaliar o desempenho prático dos algoritmos implementados.

1.3 Preliminares

Para o perfeito entendimento do conteúdo desde trabalho devemos salientar a necessidade de conhecimento prévio em alguns assuntos que enumeraremos a seguir. A maioria dos estudantes de computação deve estar familiarizado com os conceitos, mas faremos indicações de publicações que definem e usam notações próximas da que estamos utilizando.

Neste texto supomos que o leitor tem um conhecimento prévio em diversos temas de computação tais como teoria dos grafos, fluxo em redes, programação linear/inteira e relaxação, teoria de complexidade computacional etc. Recomendamos fortemente ao leitor que não está familiarizado com algum destes assuntos que leia as seguintes publicações que sugerimos abaixo, elas foram utilizadas no preparo desde trabalho.

No que se refere às partes de teoria dos grafos, fluxo em redes e algoritmos em grafos, seguimos de a nomenclatura e conceitos definidos em Feofiloff (2004). Um livro completo em relação aos conceitos de programação linear e relaxação que fazemos questão de indicar é o Wolsey (1998). Carvalho *et al.* (2001) é uma ótima referencia sobre algoritmos de aproximação. Por fim temos o livro Cormen *et al.* (2001) que pode ser usado para o estudo de complexidade computacional.

1.4 Organização

O trabalho é organizado da seguinte forma.

O Capítulo 1 – Introdução, apresenta uma visão geral do problema e da dissertação. Apresentamos textualmente uma definição do RCSP, além de citar informações sobre complexidade e descrever alguns problemas interessantes aos quais o RCSP pode ser aplicado. Enumeramos também os tópicos relevantes para o entendimento do nosso trabalho fazendo referência a textos que podem ajudar os leitores a adquirir tais conhecimentos. Discorremos um pouco também a respeito do foco e objetivos deste trabalho.

O Capítulo 2 – Caminhos mínimos (sem restrições), trás uma descrição do problema de caminhos mínimos clássicos, problema este que deu origem ao RCSP. Além da descrição, apresentamos algoritmos eficientes para o problema, e também definimos alguns conceitos importantes, usados no decorrer da dissertação.

No Capítulo 3 – Caminhos mínimos com recursos limitados, definimos formalmente o problema de caminhos mínimos com recursos limitados, que é o foco deste trabalho. Apresentamos um breve histórico listando as principais soluções conhecidas para o problema. Expomos também uma prova que mostra que o RCSP é um problema $\mathcal{NP}$-difícil. Por fim descrevemos alguns algoritmos relevantes que despertaram nosso interesse.

O Capítulo 4 – Experimentos, expõe estatísticas e percepções a respeito dos experimentos realizados com os algoritmos implementados.

Capítulo 2

Caminhos mínimos (sem restrições)

Como dito anteriormente, o problema de caminhos mínimos com recursos limitados é uma generalização do problema de caminho mínimo clássico (SP – *shortest-path problem*). Tal problema consiste em encontrar um caminho de um vértice origem até um vértice destino com menor custo em um grafo direcionado. A importância do SP se deve a suas inúmeras aplicações e generalizações.

Dada a sua relevância, vamos dedicar este capítulo ao SP. Na primeira parte descrevemos os principais conceitos relacionados ao problema, em seguida definimos o problema formalmente e por fim fazemos uma exposição de alguns algoritmos que resolvem o problema. Este capítulo foi desenvolvido baseado em Feofiloff (1997), Pisaruk (2009) e Isotani (2002).

2.1 Definições básicas

Podemos definir uma **função custo** c em um grafo direcionado $G = (V, A)$ como sendo uma função sobre A, onde, para todo $uv \in A$, $c(uv)$ é o valor de c em uv (o custo do arco uv).

Seja um caminho P e uma função custo c em um grafo $G = (V, A)$, definimos o **custo do caminho** P como $c(P) = \sum_{uv \in P} c(uv)$ a soma dos custos de todos os arcos em P.

Dizemos ainda que um caminho P tem **custo mínimo** se, seja s e t o início e o fim de P respectivamente, vale que $c(P) \leq c(Q)$ para todo caminho Q que começa em s e termina em t.

Definimos a **distância** de um vértice s a um vértice t como o custo de um menor caminho de s a t. Representamos a distância de s a t por $\text{dist}(s, t)$. A distância de s a t, na figura 2.1, é 4.

Vamos denotar por $C = \max\{c(uv) : uv \in A\}$ o **maior custo** de um arco. No grafo representado na figura 2.1 temos que o maior custo é $C = 7$.

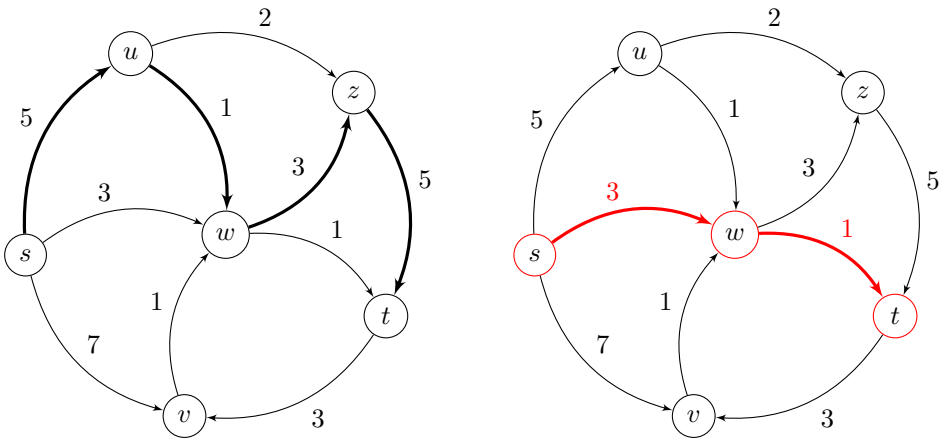


Figura 2.1: *Exemplo de um grafo com uma função custo sobre os arcos. No lado esquerdo temos um caminho* $\langle s, u, w, z, t\rangle$ *com custo igual a 14. À direita temos o caminho* $\langle s, w, t\rangle$ *em vermelho, que é um caminho de custo mínimo de* s *à* t*.*

2.2 Definição formal do problema

Com as definições que acabamos de apresentar podemos fazer uma definição formal para o **problema do caminho mínimo**, denotado por SP:

Problema SP(G, c, s, t): *Como parâmetros do problema são dados*

- *um grafo direcionado* $G = (V, A)$,
- *uma função custo* c *sobre* G,
- *um vértice origem* s *e*
- *um vértice destino* t.

O problema consiste em encontrar um caminho de custo mínimo de s *a* t.

Na literatura essa versão é conhecida como *single-pair shortest path problem* ou ainda como *single-source, single-sink shortest-path problem* (Zhu, 2005).

2.3 Funções potenciais

Vamos definir o seguinte programa linear, que chamamos de primal: encontrar um vetor x indexado por A que

$$\text{minimize} \quad \sum_{uv \in A} c(uv) x_{uv}$$

$$\text{sob as restrições} \quad \sum_{vw \in A} x_{vw} - \sum_{uv \in A} x_{uv} = \begin{cases} 1 & \text{para } v = s \\ 0 & \text{para todo } v \in V \setminus \{s, t\} \\ -1 & \text{para } v = t \end{cases}$$

$$x_{uv} \geq 0 \quad \text{para todo } uv \in A.$$

O vetor característico de qualquer caminho de s a t é uma solução viável do problema primal. Dessa forma, o problema é uma relaxação do SP. Vamos definir agora, o respectivo problema dual, que consiste em encontrar um vetor y indexado por V que

$$\begin{aligned} &\text{maximize} && y(t) - y(s) \\ &\text{sob as restrições} && y(v) - y(u) \;\leq\; c(uv) \quad \text{para todo } uv \in A. \end{aligned}$$

Uma **função-potencial** é uma função sobre V que associa a cada vértice um valor. Se y é uma função potencial e c é uma função custo, então, dizemos que y é um c**-potencial** se

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

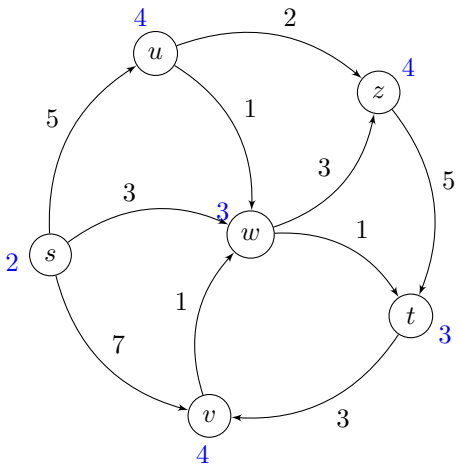


Figura 2.2: *Grafo com uma função custo c sobre os arcos e um c-potencial associado aos vértices em azul.*

É interessante que um algoritmo que resolve o SP, apresente, juntamente com a solução, certificados como garantia que sua solução é correta. O primeiro seria um certificado que garanta que os caminhos fornecidos são mínimos (**certificado de otimalidade**), este pode ser extraído a partir de uma particularização do lema da dualidade de programação linear (Feofiloff, 2000). O segundo seria o certificado de **não acessibilidade**, que pode ser apresentado da seguinte forma: se não é possível atingir um vértice t a partir de s, mostrar uma parte S de V tal que $s \in S$, $t \notin S$ e não existe $uv \in A$ com u em S e v em $V \setminus S$. A partir de um c-potencial, podemos extrair ambos os certificados de otimalidade dos caminhos encontrados, e o certificado de não acessibilidade de alguns vértices a partir de s.

Lema 2.1 (lema da dualidade): *Seja $G = (V, A)$ 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 $\langle s = v_0, \alpha_1, v_1, \ldots, \alpha_k, v_k = t\rangle$. Temos que

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

∎

Do lema 2.1 temos imediatamente os seguintes corolários.

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

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

então, t não é acessível a partir de s. ∎

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

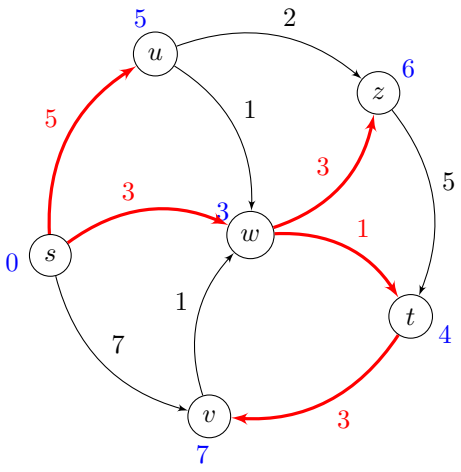


Figura 2.3: *Grafo com custos nos arcos e um potencial nos vértices. O potencial exibido garante que qualquer caminho formado por vértices vermelhos a partir de s é um caminho de custo mínimo.*

2.4 Representação de caminhos

Uma **função predecessor** ψ é uma função sobre V tal que, para cada v em V,

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

Funções desse tipo são uma maneira compacta e eficiente de representar caminhos de um dado vértice até cada um dos demais vértices de um grafo.

Dado um grafo direcionado $G = (V, A)$, uma função predecessor ψ sobre V e um caminho $P = \langle v_0, v_1, \ldots, v_k \rangle$, dizemos que P é um **caminho determinado por** ψ se

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

Dado um grafo $G = (V, A)$ e uma função predecessor ψ em V, dizemos que o grafo induzido por ψ, da forma (V, Ψ) é o **grafo de predecessores**, onde $\Psi = \{uv \in A : u = \psi(v)\}$.

vértice	ψ
s	NIL
w	s
u	s
t	w
z	w
v	t

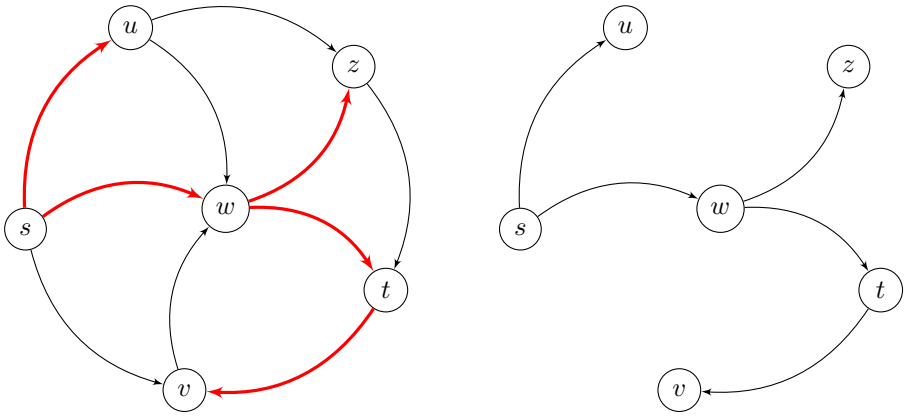


Figura 2.4: *Exemplo de uma função predecessor ψ e de um grafo de predecessores induzido por ela. Acima temos os valores de ψ para cada vértice. O grafo da esquerda mostra os arcos induzidos por ψ em vermelho. A grafo da direita é o grafo de predecessores a partir de ψ.*

2.5 Examinando arcos e vértices

Temos que além de uma função predecessor para representar os caminhos, um outro elemento muito útil em algoritmos que resolvem o SP é uma função potencial. Os custos dos caminhos que têm como origem o vértice s são limitados inferiormente por esta função.

Examinar um arco ou **relaxar/rotular um arco** (*relaxing* Cormen *et al.* (2001), *labeling step* Tarjan (1983)) é uma operação básica envolvendo uma função predecessor ψ e uma função potencial y, e consiste em verificar se y respeita c em uv e, caso não respeite, ou seja,

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

fazer

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

Podemos interpretar esta operação como a tentativa de encontrar um "atalho"para o caminho de s a v no grafo de predecessores, passando pelo arco uv.

Algoritmo EXAMINE-ARCO (uv) ▷ *examina o arco uv*

```
1  se y(v) > y(u) + c(u, v)
2      então y(v) ← y(u) + c(uv)
3            ψ(v) ← u
```

Podemos estender a operação de examinar um arco em outra operação básica, que seria **examinar um vértice**. Examinar um vértice u consiste em examinar todos os arcos da forma uv, $v \in V$.

Algoritmo EXAMINE-VÉRTICE (u) ▷ *examina o vértice u*

```
1  para cada uv em A faça
2      se y(v) > y(u) + c(uv)
3          então y(v) ← y(u) + c(uv)
4                ψ(v) ← u
```

As operações de examinar arcos ou vértices sempre diminuem o valor da função potencial em um vértice visando respeitar a função custo no elemento em que se está examinando. Ou seja, tentando tornar y em um c-potencial.

2.6 Algoritmos

Existem vários algoritmos eficientes para resolver o problema de caminho mínimo, sendo os mais conhecidos os algoritmos de Dijkstra e o de Ford. Ambos aplicam-se ao problema como definimos, caminho mínimo de um vértice inicial s para um vértice final t ou de s para todos os outros vértices. Apresentamos com detalhes o algoritmo de Dijkstra.

2.6.1 Algoritmo de Dijsktra

Vamos descrever agora o famoso algoritmo de Edsger Wybe Dijkstra (Dijkstra, 1959) que resolve o problema do caminho mínimo em grafos para o caso em que a função custo possui apenas custos não negativos, ou seja, $c(uv) \geq 0$ para todo $uv \in A$. Nosso texto segue de perto Pisaruk (2009) e Feofiloff (2004).

Descrição

O algoritmo é iterativo. No início de cada iteração tem-se os conjuntos S e Q, que são uma partição do conjunto de vértices do grafo ($S \cap Q = \emptyset$ e $S \cup Q = V$). O algoritmo define caminhos partindo de s a cada vértice em S, caminhos estes que são garantidamente de custo mínimo, e define caminhos a uma parte dos vértices em Q. Antes da primeira iteração temos $S = \emptyset$ e $Q = V$. Cada iteração consiste em retirar um determinado vértice de Q, examiná-lo e adicioná-lo a S. Eventualmente, ao examinar tal vértice, descobrimos caminhos a vértices em Q até então não alcançados, ou melhores que os já conhecidos. Se o conjunto Q é vazio, já examinamos todos os vértices e podemos parar o processamento.

O algoritmo recebe um grafo direcionado $G = (V, A)$, 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 tal 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 $C = \max\{c(uv) : uv \in A\}$ [1]. Convenientemente definimos $y(s) = 0$.

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

```
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 ← V      ▷ Q é uma fila com prioridades
6   enquanto Q ≠ ⟨⟩ faça
7       retire de Q um vértice u com y(u) mínimo
8       para cada uv em A faça
9           se y(v) > y(u) + c(uv) então
10              y(v) ← y(u) + c(uv)
11              ψ(v) ← u
12  devolva ψ e y
```

Simulação

A seguir, temos uma simulação para uma chamada do algoritmo DIJKSTRA. No estado (1) temos o grafo $G = (V, A)$ com custo sobre os arcos. Nos estados os vértices em Q têm interior azul claro. A função-potencial y é indicada pelos números em azul próximos cada vértice. No

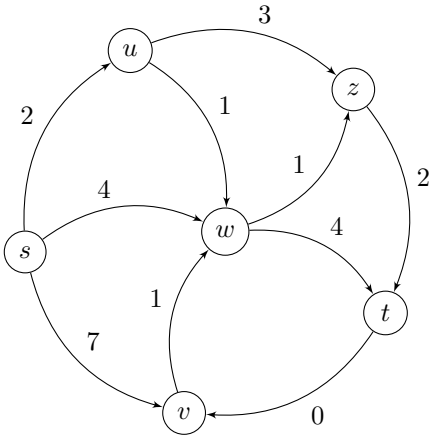
[1] Se ψ determina um caminho de s a um vértice t, então este caminho tem custo mínimo (condição de otimalidade, corolário 2.3). Se y é um c-potencial com $y(t) - y(s) = nC + 1$, então não existe caminho de s a t (condição de inacessibilidade, corolário 2.2).

estado (2) temos que todos os vértices foram inseridos em Q e os potencias foram inicializados ($y(s) = 0$ e o potencial dos demais vértices é $n \times C + 1 = 6 \times 7 + 1 = 43$).

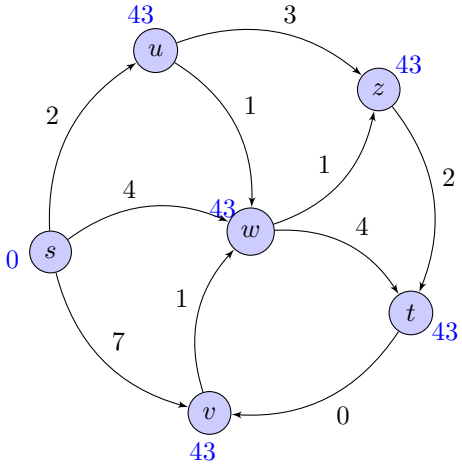
Um vértice que está sendo examinado tem a cor rosa. O arco que está sendo examinado tem a cor azul (no estado (3) o vértice que está sendo examinado é s e o arco sendo examinado é sv).

Os vértices em S (já examinados) têm a cor verde. No estado (10) vemos que w está sendo examinado, s e u são os vértices em S e os demais estão em Q.

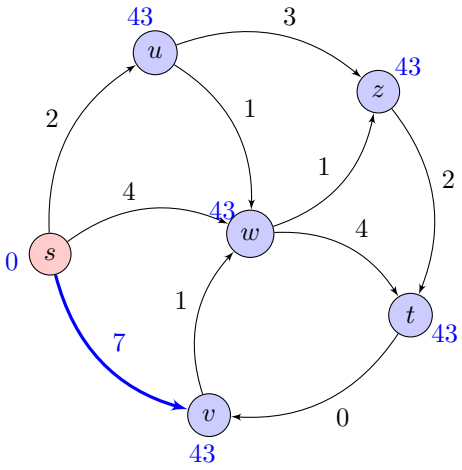
Os arcos já examinados têm a cor vermelha se fazem parte do grafo de predecessores, caso contrário são tracejados.



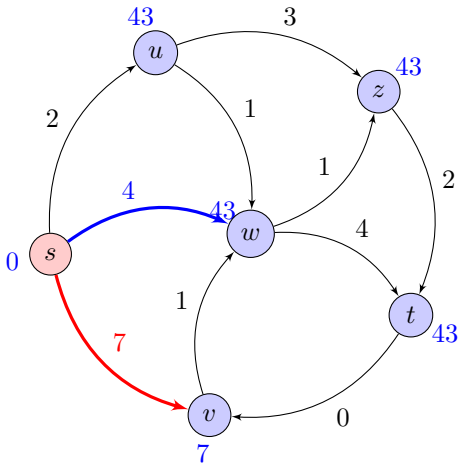
(1)



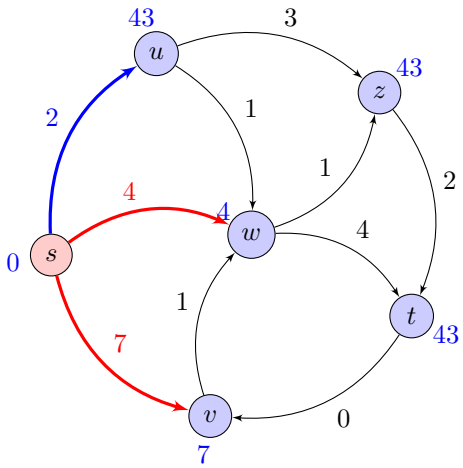
(2)



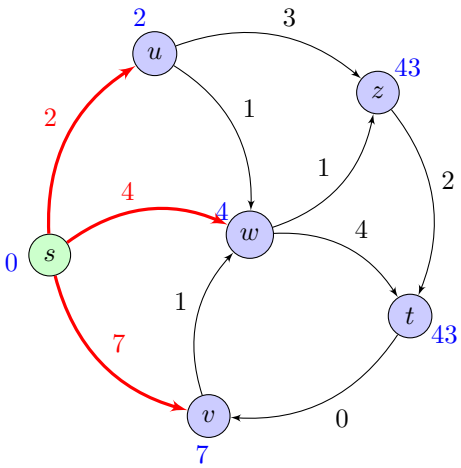
(3)



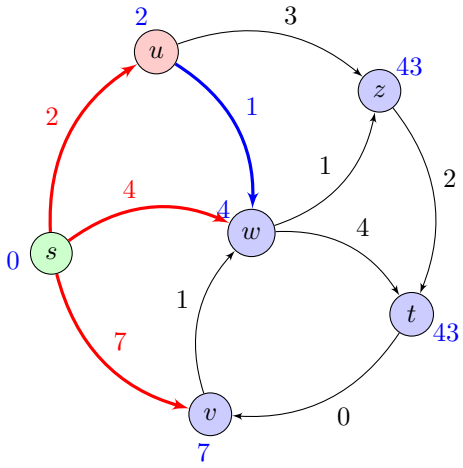
(4)


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

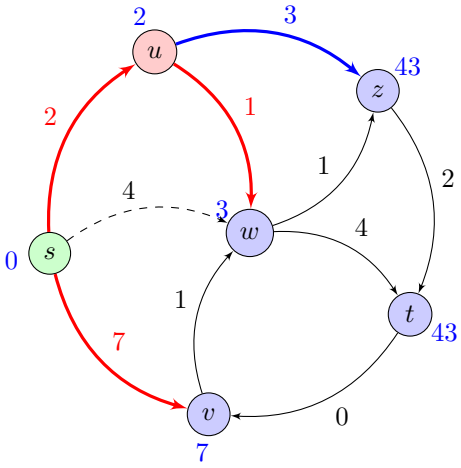
(5)


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

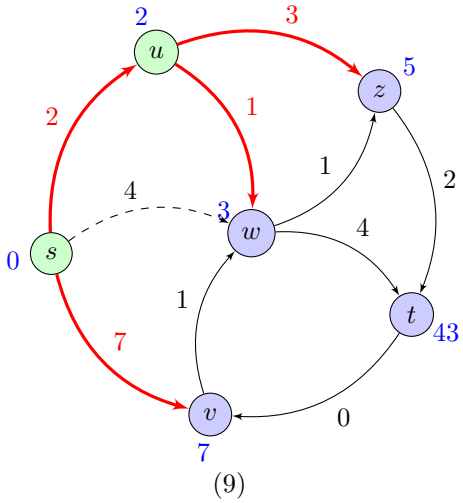
(6)


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

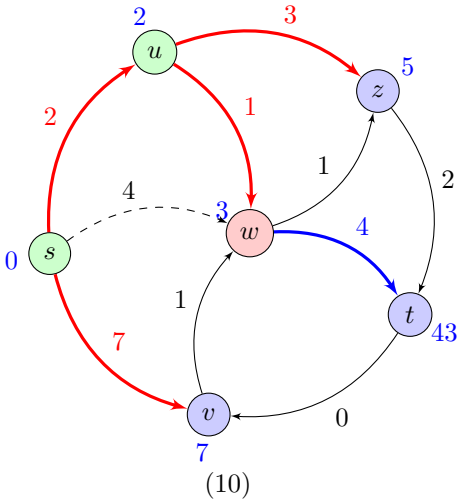
(7)


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

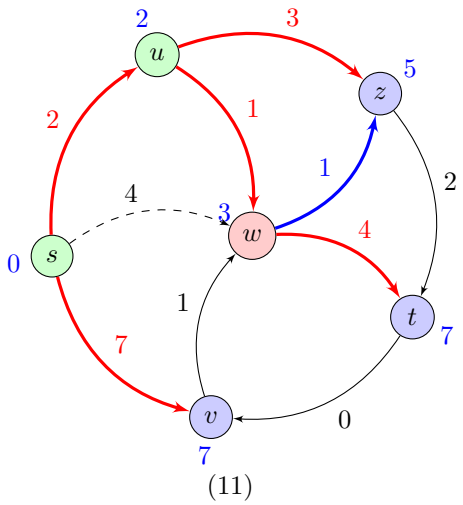
(8)


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

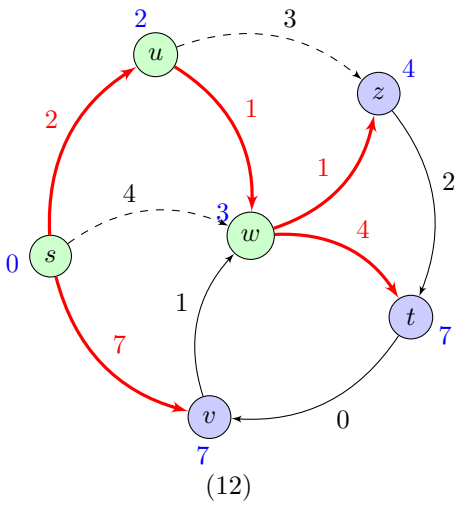
(9)


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

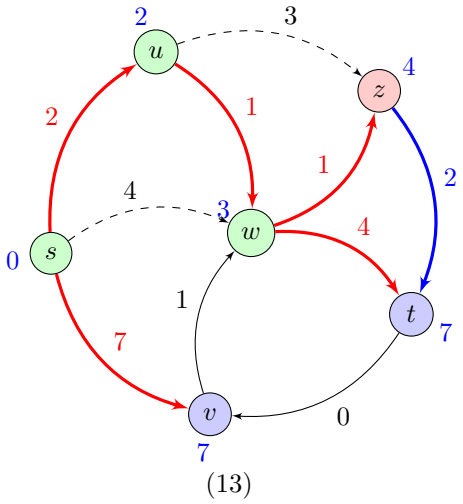
(10)



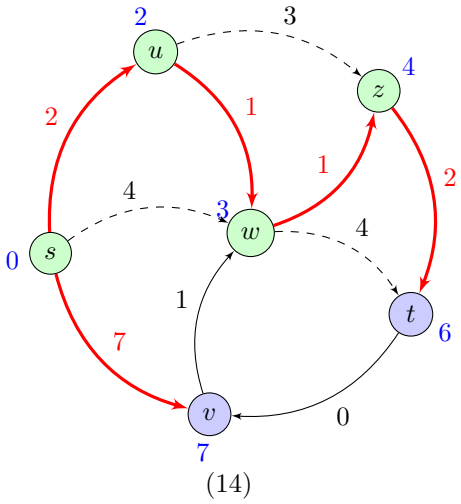
(11)



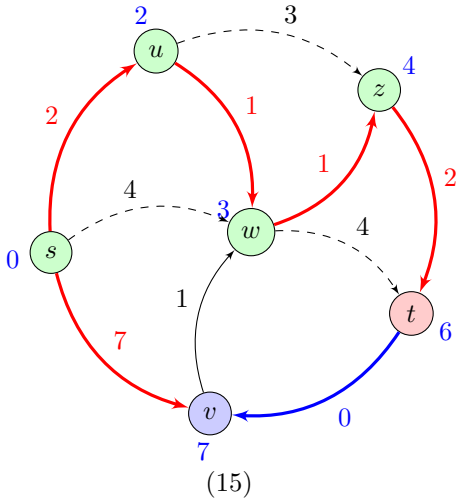
(12)



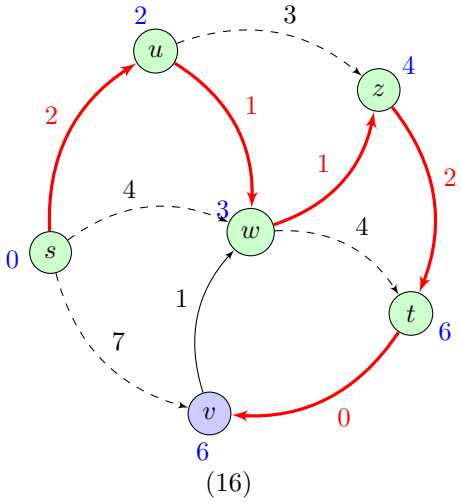
(13)



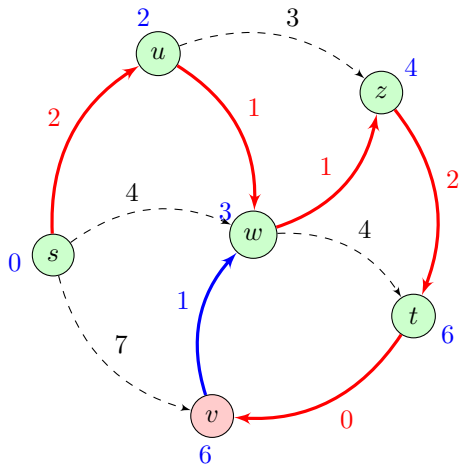
(14)



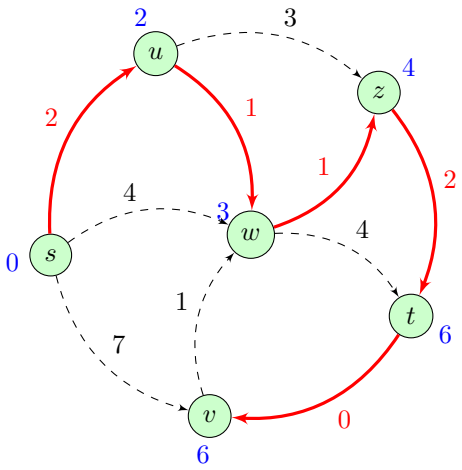
(15)



(16)



(17)



(18)

Corretude

A corretude do algoritmo de Dijkstra baseia-se nas demonstrações da validade de uma série de relações invariantes. Estas relações são afirmações envolvendo os dados do problema V, A, c e s e os objetos y, ψ, S e Q. Para uma prova detalhada recomendamos a leitura de Isotani (2002). Aqui faremos apenas uma argumentação básica para mostrar que o algoritmo é correto.

Teorema 2.4 (da corretude): *Dado um grafo $G = (V, A)$, 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.*

A seguir um prova indutiva do teorema 2.4.

Demonstração: Como Q é vazio no final do processo, vale que todos os vértices, e portanto todos os arcos, foram examinados, o que garante que a função y é um c-potencial. Se $y(t) < nC + 1$ então o valor de $y(t)$ foi atualizado ao menos uma vez, e assim vale que $\psi(t) \neq$ NIL. Logo, segue que existe um st-caminho P no grafo de predecessores e P é uma caminho de custo mínimo pela condição de otimalidade porque

$$y(v) = y(u) + c(uv), \forall uv \in \Psi \Rightarrow c(P) = y(t) - y(s) = y(t).$$

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

Concluímos portanto que o algoritmo faz o que promete. ∎

Consumo de tempo

As duas principais operações no algoritmo são as seguintes (analisadas no pior caso):

1. escolher um vértice com potencial mínimo, que pode exigir tempo $O(n)$ e é executada até n vezes (até Q ficar vazio), ou seja, consome tempo $O(n^2)$;

2. atualizar o potencial, que pode acontecer para todas as arestas, ou seja, exige tempo $O(m)$.

Assim, o consumo de tempo do algoritmo no pior caso é $O(n^2+m) = O(n^2)$. Para grafos esparsos, existem métodos sofisticados, como o heap de Johnson (Johnson, 1977), o heap de Fibonacci (Fredman e Tarjan, 1987), que permitem diminuir o tempo consumido para encontrar um vértice com potencial mínimo, gerando assim implementações que consomem menos tempo para resolver o problema.

Dijkstra e filas de prioridades

Uma **fila de prioridades** (Ahuja *et al.*, 1993; Cormen *et al.*, 1999) é uma estrutura de dados que armazena uma coleção de itens, cada um com uma prioridade associada. Os itens serão basicamente vértices em nosso contexto.

Temos as seguintes operações para uma fila de prioridade Q:

- INSERT(v, p, Q): adiciona o vértice v com prioridade p em Q.
- DELETE(v, Q): remove o vértice v de Q.
- EXTRACT-MIN(Q): devolve o vértice com a menor prioridade e o remove de Q.
- DECREASE-KEY(v, p, Q): muda para p a prioridade associada ao vértice v em Q (p deve ser menor que a atual prioridade associada a v).
- BUILD-MIN-HEAP(V): recebe o conjunto V de vértices em que cada vértice v tem prioridade $y(v)$ e constrói uma fila de prioridades Q.

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

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

```
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
```

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

Tabela 2.1: *Complexidade do algoritmo de Dijkstra de acordo com as filas de prioridade.*

```
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 consumo de tempo do algoritmo de 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 Ahuja *et al.* (1990); Cherkassky *et al.* (1999); Fredman e Tarjan (1987).

A tabela 2.1 resume as complexidades de tempo de várias implementações de filas de prioridade e o respectivo consumo de tempo resultante para o algoritmo de Dijkstra Cormen *et al.* (2001).

Fredman e Tarjan (1987) observaram que como o algoritmo de Dijkstra examina os vértices em ordem de distância a partir de s, o algoritmo está, implicitamente, ordenando estes valores. Assim, qualquer implementação do algoritmo de Dijkstra realiza pelo menos $\Omega(n \log n)$ comparações e faz $\Omega(m + n \log n)$ operações elementares.

2.6.2 Algoritmo de Ford

Ao contrário do algoritmo DIJKSTRA, este algoritmo, que foi proposto por Ford (Bellman, 1958), pode ser aplicado a grafos cujos arcos têm associado custos negativos, não podendo contudo existir circuitos de custo negativo (Barrico, 1998).

Este algoritmo consiste em examinar os vértices do grafo, corrigindo os potenciais tantas vezes quantas forem necessárias, até que todos os potenciais satisfaçam a condição de otimalidade. Neste algoritmo, os potenciais dos vértices têm o mesmo significado dos potenciais utilizados no algoritmo DIJKSTRA, só que agora os potenciais só se tornam definitivos após terminar a execução do algoritmo.

O algoritmo recebe um grafo $G = (V, A)$, uma função custo c de A em $\mathbb{Z}$ 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 $C := \max\{c(uv) : uv \in A\}$. Versão genérica do algoritmo.

Algoritmo BELLMAN-FORD (V, A, c, s) $\triangleright$ (V, A, c) *não possui ciclos negativos*

```
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  enquanto y(v) > y(u) + c(uv) para algum uv ∈ A faça
6      y(v) ← y(u) + c(uv)
7      ψ(v) ← u
8  devolva ψ e y
```

A complexidade de tempo desta versão é $O(n^2mC)$ (Feofiloff, 2004). A complexidade de tempo é tão elevado porque o algoritmo pode examinar cada arco muitas vezes (o valor de $y(v)$ pode diminuir muitas vezes antes de atingir seu valor final). A seguir temos a melhor versão conhecida, do ponto de vista do consumo assintótico de tempo, para o problema do caminho mínimo com custos arbitrários.

Algoritmo BELLMAN-FORD (V, A, c, s) $\triangleright$ (V, A, c) *não possui ciclos negativos*

```
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  repita n − 1 vezes
6      para cada uv em A faça
7          se y(v) > y(u) + c(uv) então
8              y(v) ← y(u) + c(uv)
9              ψ(v) ← u
10 devolva ψ e y
```

O algoritmo consome $O(nm)$ unidades de tempo. Como $m = O(n^2)$ no pior caso tem complexidade de tempo $O(n^3)$.

Algoritmo de Ford e programação dinâmica

Podemos descrever o método de Ford de uma forma alternativa que deixa mais claro que o algoritmo proposto se trata de uma programação dinâmica.

Vamos descrever a recorrência sobre a qual o algoritmo é implementado da seguinte forma. Definimos $d_k(v)$ como sendo o menor comprimento de uma caminho de s a v que possui no máximo k arcos. Caso não exista nenhum caminho de s a v, definimos $d_k(v) = \infty$. Com essa definição temos que o caminho mínimo de s a t é igual a $d_n(t)$ pois um caminho contém no máximo $n - 1$ arcos e precisamos de uma iteração a mais para detectar se há ciclos negativos.

Em termos algorítmicos, a função d_0 pode ser calculada de forma trivial, onde $d_0(s) = 0$ e $d_0(v) = \infty$ para $v \in V \setminus \{s\}$. Os demais valores podem ser calculados da seguinte forma:

$$d_k(v) = \min \left\{ d_{k-1}(v), \min_{u|uv \in A} \{d_{k-1}(u) + c_{uv}\} \right\}$$

A partir dessa recorrência, podemos de forma direta, extrair um algoritmo recursivo e a partir da recursão extrair um algoritmo iterativo que resolve o problema.

Capítulo 3

Caminhos mínimos com recursos limitados

3.1 Definição do problema

Problema RCSP$(G, s, t, k, r, \lambda, c)$: *Como parâmetros do problema são dados*

- *um grafo direcionado* $G = (V, A)$,
- *um vértice origem* $s \in V$ *e um vértice destino* $t \in V$, $s \neq t$,
- *um número* $k \in \mathbb{N}$ *de recursos disponíveis* $\{1, \ldots, k\}$,
- *o consumo de recursos* $r^i_{uv} \in \mathbb{N}_0$ *de cada arco de* G *sobre os* k *recursos disponíveis,* $i = 1, \ldots, k$, $uv \in A$,
- *o limite* $\lambda^i \in \mathbb{N}_0$ *que dispomos de cada recurso,* $i = 1, \ldots, k$,
- *o custo* $c_{uv} \in \mathbb{N}_0$, *para cada arco,* $uv \in A$.

O consumo de um recurso i, $i = 1, \ldots, k$ *em um st-caminho* P *é* $r^i(P) = \sum_{uv \in P} r^i_{uv}$. *Um st-caminho* P *é limitado pelos recursos* $1, \ldots, k$ *se este consome não mais que o limite disponível de cada recurso, ou seja, se* $r^i(P) \leq \lambda^i$, $i = 1, \ldots, k$. *O custo de um st-caminho* P *é* $c(P) = \sum_{uv \in P} c_{uv}$. *O problema* RCSP *consiste em encontrar o caminho limitado pelos recursos de menor custo.*

Usaremos no decorrer deste trabalho $n = |V|$ e $m = |A|$. Quando estivermos tratando de um contexto onde existe apenas um recurso (SRCSP), ou seja, $k = 1$, usaremos apenas λ para representar λ^1 e apenas r_{uv} para representar r^1_{uv}.

3.2 Revisão bibliográfica

Pelo fato do RCSP surgir como subproblema em um grande número de problemas práticos, ele tem sido extensivamente estudado como pode ser visto na tabela 3.1 que mostra uma lista com algumas características dos principais algoritmos disponíveis.

Basicamente, duas famílias de algoritmos têm sido propostas: uma envolve resolver o problema relaxado usando relaxação linear ou relaxação lagrangiana e a outra usa programação dinâmica. Métodos baseados em relaxações geralmente consistem em três passos: (1) computar limites inferior e superior para uma solução ótima resolvendo o problema relaxado, (2) usar os resultados do primeiro passo para reduzir o grafo, e (3) eliminar a folga da dualidade.

Seguindo esta linha, Handler e Zang (1980) resolveram um dual lagrangiano (para a versão com um único recurso) no passo (1), para eliminar a folga da dualidade, eles usaram o algoritmo de Yen (1971) (que é um algoritmo que calcula todos os caminhos entre um par de vértices em ordem crescente de custo, desenvolvido para o problema do k-ésimo menor caminho, ou KSP – *k shortest path problem*). A ideia era que, embora estivessem usando o algoritmo de Yen, que complexidade de tempo exponencial, o número de caminhos computados seria bastante reduzido. Beasley e Christofides (1989) apresentaram uma abordagem baseada em relaxação lagrangiana que usa o método do sub-gradiente para resolver o dual lagrangiano como primeiro passo e *branch and bound* para eliminar a folga da dualidade. Mehlhorn e Ziegelmann (2000) propuseram o método do envoltório que resolve uma relaxação linear para o caso com múltiplos recursos, para eliminar a folga eles usam um método de enumeração de caminhos como em Hassin (1992).

Referencia	Versão do RCSP	Método	Custo	Grafo
Handler e Zang (1980)	único recurso	RL + YEN	$c_{uv} \geq 0$	gerais
Beasley e Christofides (1989)	múltiplos recursos	RL + BB	irrestrito	gerais
Mehlhorn e Ziegelmann (2000)	múltiplos recursos	PL	$c_{uv} \geq 0$	gerais
Joksch (1966)	único recurso	PD	$c_{uv} > 0$	gerais
Aneja *et al.* (1983)	múltiplos recursos	LS	$c_{uv} \geq 0$	gerais
Hassin (1992)	único recurso	PD	$c_{uv} > 0$	acíclicos
Dumitrescu e Boland (2003)	múltiplos recursos	LS	$c_{uv} \geq 0$	gerais

Tabela 3.1: *Principais algoritmos disponíveis para o* RCSP.
RL – relaxação lagrangiana; KSP – k-ésimo menor caminho; BB – *branch and bound*; PL – relaxação linear; PD – programação dinâmica; LS – rotulamento permanente.

Um número considerável de publicações abordam soluções baseadas em programação dinâmica para o RCSP. Joksch (1966) propôs um algoritmo baseado em programação dinâmica, bem como Hassin (1992), que apresentou dois algoritmos exatos para uso com grafos acíclicos. Aneja *et al.* (1983) adaptaram o esquema de rotulamento permanente (*label-setting*) de Dijkstra (1959). A abordagem de rotulamento permanente é uma generalização do algoritmo DIJKSTRA, que rotula e processa os vértices em uma ordem baseada nos consumos de recursos. Além do método de rotulamento permanente, temos a abordagem de rotulamento corretivo (*label-correcting*) que é uma generalização do algoritmo BELLMAN-FORD; este trata cada vértice várias vezes, atualizando os rótulos quando necessário. Dumitrescu e Boland (2003) investigaram variações do algoritmo de rotulamento corretivo. Eles apresentaram uma versão melhorada do algoritmo de Aneja *et al.* (1983),

além de apresentar um algoritmo baseado em um método de escalar pesos. **?**.

3.3 Complexidade

Handler e Zang (1980) e Jaffe (1984) mostraram que o RCSP é $\mathcal{NP}$-difícil, mesmo em grafos acíclicos, com restrições sobre um único recurso, e com todos os consumos de recursos positivos (Dumitrescu e Boland, 2003). Garey e Johnson (1979) apresentaram uma redução do problema da partição para o RCSP, enquanto Handler e Zang (1980) reduziram o problema da mochila para o nosso problema.

Hassin (1992) mostrou que o SRCSP tem solução polinomial se os custos dos arcos e consumos de recursos são limitados. A seguir, mostraremos uma redução do **problema da mochila** (*knapsack*), definido a seguir, ao problema RCSP(baseada em Handler e Zang (1980)).

Problema MOCHILA(N,w,v,D): *Como parâmetros do problema são dados:*

- *um conjunto de itens* $N = \{1, \ldots, n\}$,
- *pesos* $w_i \in \mathbb{N}$, $i = 1, \ldots, n$, *para esses itens,*
- *valores* $v_i \in \mathbb{N}$, $i = 1, \ldots, n$, *para esse itens,*
- *um peso limite* $D \in \mathbb{N}_0$.

O peso de um subconjunto $I \subseteq N$ *é* $w(I) = \sum_{i \in I} w_i$, *e seu valor é* $v(I) = \sum_{i \in I} v_i$. *O problema* MOCHILA *consiste em encontrar um subconjunto de itens com valor máximo, cujo peso não excede o limite* D.

Teorema 3.1: *O* RCSP *é* $\mathcal{NP}$*-difícil.*

Demonstração: A prova se dá pela redução do problema MOCHILA ao RCSP. Vamos tomar uma instância I do problema MOCHILA. Nós podemos construir uma instância I' para o RCSP como se segue:

- $V = N \cup \{0\}$.
- Entre cada par de vértice $i-1$ e i, onde $i = 1, 2, \cdots, n$, teremos duas arestas paralelas $(i-1, i)$ que estarão separadas, uma em cada um dos dois subconjuntos A_1 e A_2 que compõem A.
  - $A = A_1 \cup A_2$,
  - $A_1 = \{(i-1, i) : i = 1, \ldots, n\}$,
  - $A_2 = \{(i-1, i) : i = 1, \ldots, n\}$.
- $r_{uv} = \begin{cases} w_i, & \text{se } uv \in A_1, \\ 0, & \text{caso contrário} \end{cases}$ para todo $uv \in A$.

- $c_{uv} = \begin{cases} M - v_i, & \text{se } uv \in A_1, \\ M, & \text{caso contrário} \end{cases}$ para todo $uv \in A$.
- $s = 0$.
- $t = n$.
- $k = 1$.
- $\lambda = D$.

A constante M pode ser definida como um grande inteiro de tal forma que $M - v_i$, para qualquer i, seja não negativo. Por questão de praticidade, vamos convencionar que representaremos um arco $(i-1, i) \in A_1$ como a_i^1 e um arco $(i-1, i) \in A_2$ como a_i^2.

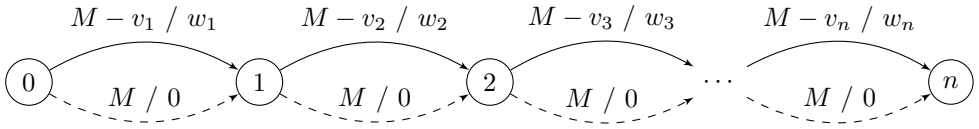


Figura 3.1: *Os arcos contínuos representam os arcos no conjunto A_1 e os arcos tracejados representam os arcos no conjunto A_2. O rótulo de cada arco uv representa o seu custo e o seu consumo de recurso, respectivamente (c_{uv} / r_{uv}).*

Como $s = 0$ e $t = n$; podemos ver que qualquer st-caminho P em $G = (V, A)$ contém ou a_i^1 ou a_i^2, $i = 1, \ldots, n$. Vamos dividir os arcos de P em dois conjuntos X e Y, onde X contém os arcos em P que estão em A_1, e Y contém os demais arcos. A partir de X, vamos definir um subconjunto $S \subseteq N$, tal que $i \in S$ se e somente se $a_i^1 \in X$. Com isso,

$$\begin{aligned} c(P) &= \sum_{a_i^1 \in X} (M - v_i) + \sum_{a_i^2 \in Y} M \\ &= n \cdot M - \sum_{a_i^1 \in X} v_i \\ &= n \cdot M - v(S) \end{aligned} \tag{3.1}$$

$$\begin{aligned} r(P) &= \sum_{a_i^1 \in X} w_i + \sum_{a_i^2 \in Y} 0 \\ &= \sum_{a_i^1 \in X} w_i \\ &= w(S) \end{aligned} \tag{3.2}$$

Daí, concluímos que todo subconjunto $S \subseteq N$ contém um st-caminho P associado, e vice-versa, por meio da equivalência $i \in S \Leftrightarrow a_i^1 \in P$. Pelas equações 3.1 e 3.2, um conjunto S e um caminho P associados, possuem $r(P) = w(S)$ e $c(P) = n \cdot M - v(S)$. E daí temos dois resultados:

$$\begin{aligned} r(P) \leq \lambda &\iff w(S) \leq D \\ \text{minimizar } c(P) &\iff \text{maximizar } v(S) \end{aligned}$$

∎

3.4 Preprocessamento

Nesta seção nós revisamos algumas possibilidades de redução do tamanho do problema pela eliminação de vértices e/ou arcos que não podem fazer parte de uma solução ótima.

3.4.1 Redução baseada nos recursos

Vamos denotar por R_{uv} a menor quantidade de recurso que podemos usar partindo do vértice u até o vértice v. Qualquer vértice v para o qual vale que

$$R_{sv} + R_{vt} > \lambda$$

pode ser removido do grafo, tendo em vista que este vértice não pode pertencer a qualquer caminho viável. De forma similar, qualquer arco uv para o qual vale que

$$R_{su} + r_{uv} + R_{vt} > \lambda$$

pode ser removido do grafo pelo mesmo motivo. Aneja *et al.* (1983) foram os primeiros a apresentar reduções baseadas nos recursos como descrito acima. Podemos executar esta redução com duas computações do algoritmo Dijkstra (para calcular R_{sv} e R_{vt} para qualquer $v \in A$) seguidas pela iteração sobre todos os vértices e arcos verificando as condições acima descritas ($O(n^2 + m + n)$).

3.4.2 Redução baseada nos custos

Caso tenhamos um limite superior UB conhecido para uma solução ótima do problema, nós podemos estender a ideia da redução baseada em recursos para uma redução baseada em custos. Vamos denotar C_{uv} como sendo o menor custo possível para um caminho partindo de u até v. Qualquer vértice v para o qual vale que

$$C_{sv} + C_{vt} > UB$$

pode ser removido, da mesma forma que qualquer arco uv para o qual vale que

$$C_{su} + c_{uv} + C_{vt} > UB$$

pode ser removido. Neste caso, a efetividade da redução depende diretamente da qualidade do limite superior UB que conhecemos.

Dumitrescu e Boland (2003) propuseram que esta redução fosse executada repetidas vezes, atualizando-se o limite superior UB caso possível, até que não fosse feita nenhuma remoção no grafo. O método é efetivo quando conhecemos um bom limitante inferior.

3.5 Programação dinâmica

Programação dinâmica é uma técnica bastante poderosa para resolver determinados tipos de problemas computacionais. Muitos algoritmos eficientes fazem uso desse método. Basicamente, esta estratégia de projeto de algoritmos, traduz uma recursão para uma forma iterativa que utiliza uma tabela como apoio para as computações.

Da mesma forma que acontece em um algoritmo recursivo, em um algoritmo baseado em programação dinâmica, cada instância do problema é resolvida a partir da solução de subinstâncias. O que distingue a programação dinâmica de uma recursão comum é o uso da tabela que "memoriza" as soluções das subinstâncias, evitando assim o recálculo caso esses valores sejam necessários mais de uma vez.

Para que o paradigma da programação dinâmica possa ser aplicado, é preciso que o problema tenha as seguintes características: subestrutura ótima e superposição de subproblemas. Um problema apresenta uma subestrutura ótima quando uma solução ótima para o problema contém (ou pode ser calculada a partir de) soluções ótimas para subproblemas. A superposição de subproblemas acontece quando um algoritmo recursivo recalcula o mesmo subproblema muitas vezes.

Temos alguns algoritmos baseados em programação dinâmica conhecidos para o RCSP. Todos têm complexidade pseudo-polinomial, ou seja, complexidades de tempo que dependem do tamanho dos custos e consumo de recursos dos arcos. Nesta seção iremos apresentar três destas soluções. A primeira itera sobre os possíveis valores de consumo, minimizando o custo; e iremos referenciá-la como **programação dinâmica primal**. A segunda, muito similar à primeira, itera sobre os possíveis valores de custo, verificando se o consumo de recursos atende ao limite imposto; e iremos referenciá-la como **programação dinâmica dual**. Por último temos uma modelagem baseada em uma generalização do algoritmo Dijkstra, geralmente relacionada a soluções do problema de caminho mínimo multi-objetivo, que aqui chamaremos de **programação dinâmica por rótulos** (vale salientar que embora estejamos descrevendo este algoritmo nesta seção, ele não é um algoritmo por programação dinâmica propriamente dito, mas é baseado na programação dinâmica primal). Embora todos os algoritmos possam ser usados na versão do problema com múltiplos recursos, por questão de simplicidade vamos descrevê-los na versão com um único recurso.

3.5.1 Programação dinâmica primal

Um dos primeiros a descrever um algoritmo baseado em programação dinâmica para o RCSP foi Joksch (1966) (ver também Lawler (1976); Witzgall e Goldman (1965)). Nesta seção iremos descrever esse algoritmo.

Na programação dinâmica primal, iteramos sobre o consumo de recursos, partindo de uma quantidade unitária até o limite imposto, obtendo os caminhos mínimos limitados pelos recursos com custo mínimo partindo de s. Vamos definir a recorrência sobre a qual

o algoritmo é implementado da seguinte forma. Definimos $f_j(r)$ como sendo o menor custo possível para um caminho de s a j, que consome no máximo r unidades de recurso, e assim temos:

$$f_v(r) = \begin{cases} 0, & \text{se } v = s \\ & \text{e } r = 0, \ldots, \lambda \\ \infty, & \text{se } v \neq s \\ & \text{e } r = 0 \\ \min\left\{ f_v(r-1), \min\limits_{u \mid r_{uv} \leq r} \{ f_u(r - r_{uv}) + c_{uv} \} \right\}, & \text{se } v \neq s \\ & \text{e } r = 1, \ldots, \lambda \end{cases}$$

A partir da recorrência podemos extrair de forma direta, uma recursão de complexidade exponencial. Temos como base da recursão $f_s(r) = 0$ para $1 \leq r \leq \lambda$ e $f_j(0) = \infty$ para $j \neq s$. Abaixo temos o algoritmo recursivo denominado de RECURSÃO-PRIMAL[1] [2].

A fim de reconstruirmos os caminhos encontrados vamos manter uma função ψ como apresentado no capítulo anterior. Aqui a função $\psi(v, r)$ armazena o predecessor do vértice v no caminho de custo mínimo que consume recurso no máximo r.

Algoritmo RECURSÃO-PRIMAL (v, r)

```
1   ψ(v, r) ← NIL
2   se v = s então
3       devolva 0
4   se r = 0 então
5       devolva ∞
6   custo ← RECURSÃO-PRIMAL(v, r − 1)
7   ψ(v, r) ← ψ(v, r − 1)
8   para cada uv em A faça
9       se r_uv ≤ r então
10          valor ← RECURSÃO-PRIMAL(u, r − r_uv) + c_uv
11          se custo > valor então
12              ψ(v, r) ← u
```

[1] Para que os parâmetros no algoritmo não formem uma lista muito extensa, vamos considerar que temos acesso de forma global ao grafo $G = (V, A)$, às funções c e r sobre os arcos, ao vértice de origem s e à função predecessor ψ.

[2] Na descrição do algoritmo denotaremos por r (sem índice) o consumo máximo de recursos permitido para a chamada corrente. Sempre que r representar a função de consumo sobre os arcos ele estará acompanhado pelo índice que representa o arco, r_{uv} por exemplo.

```
13              custo ← valor
14      devolva custo
```

É importante salientar que se no grafo existirem ciclos com consumo nulo de recursos, o algoritmo recursivo não pode ser aplicado diretamente pois entraria em "loop infinito". O que garante que o algoritmo termina é o fato de que o parâmetro r sempre diminui nas chamadas recursivas e a base da recursão responde de forma direta aos casos com r nulo. Nestes casos, precisaríamos realizar um preprocessamento no grafo baseado no seguinte fato: Se o arco $uv \in A$ possui $r_{uv} = 0$, temos que todo arco $wu \in A$ pode ser "estendido" como um arco wv onde $c_{wv} = c_{wu} + c_{uv}$ e $r_{wv} = r_{wv} + 0$. O preprocessamento então substituiria todas os arcos com consumo nulo pelos arcos "estendidos" até que não existissem arcos com consumo nulo no grafo. Podemos adaptar o algoritmo de Floyd (1962) para realizar tal processamento em $O(n^3)$.

O valor do caminho ótimo OPT pode ser encontrado pela chamada do algoritmo que corresponde ao valor $f_t(\lambda)$ da recorrência, ou seja, a chamada de RECURSÃO-PRIMAL(t, λ). O caminho ótimo propriamente dito pode ser construído usando a função predecessor ψ montada no decorrer da recursão.

A partir do algoritmo recursivo, podemos implementar um algoritmo iterativo que computa o valor de um caminho ótimo em tempo $O(m\lambda)$ e consumo de memória $O(n\lambda)$. Joksch (1966) apresentou melhorias práticas para este algoritmo, contudo a complexidade de pior caso é não melhor que a obtida com a ideia básica.

Algoritmo PROGRAMAÇÃO-DINÂMICA-PRIMAL (t, λ)

```
1   PD ← [ ]    ▷ tabela de programação dinâmica
2   para cada r em {0, 1, ··· , λ} faça
3       ψ(s, r) ← NIL
4       PD[s, r] ← 0
5   para cada v em V \ {s} faça
6       ψ(v, 0) ← NIL
7       PD[v, 0] ← ∞
8   para r de 1 até λ faça
9       para cada v em V \ {s} faça
10          ψ(v, r) ← ψ(v, r − 1)
11          PD[v, r] ← PD[v, r − 1]
12          para cada uv em A faça
13              se r_uv ≤ r então
14                  se PD[v, r] > PD[v, r − r_uv] + c_uv então
```

15 $\psi(v, r) \leftarrow u$

16 $\text{PD}[v, r] \leftarrow \text{PD}[v, r - r_{uv}] + c_{uv}$

17 ***devolva*** $\text{PD}[t, \lambda]$

A programação dinâmica iterativa é ainda mais sensível a arcos com consumo nulo de recurso que a recursão. Neste caso, um simples arco $uv \in A$ com $r_{uv} = 0$, pode nos fazer acessar uma posição ainda não calculada da tabela e assim computar uma solução incorreta. Desta forma o preprocessamento descrito para remover arcos sem consumo de recursos deve ser executado antes da chamada de Programação-Dinâmica-Primal.

A seguir vamos apresentar um exemplo para o algoritmo. Na figura 3.2 temos representado um grafo, perto dos arcos temos c/r que representam o custo e o consumo de recursos respectivamente. No exemplo tomamos o vértice $s = 0$ para ser nossa origem e o vértice $t = 9$ como nosso destino, o nosso limite de recursos é 10. A solução do exemplo tem custo igual a 14 que pode ser conferida na tabela 3.2. Esta tabela representa o estado da variável PD ao final da execução.

$v \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	∞	∞	∞	∞	∞	∞	∞	1	1	1	1
2	∞	∞	4	4	4	4	4	4	4	4	4
3	∞	7	7	7	7	7	7	7	4	4	4
4	∞	∞	∞	∞	7	7	7	7	7	7	7
5	∞	∞	∞	∞	∞	11	11	11	11	10	10
6	∞	∞	14	14	14	9	9	9	9	9	9
7	∞	∞	∞	∞	9	8	8	8	5	5	5
8	∞	∞	8	8	8	8	8	3	3	3	3
9	∞	∞	∞	20	20	20	15	15	15	14	14

Tabela 3.2: *Tabela de programação dinâmica primal gerada para o grafo da Figura 3.2.*

3.5.2 Programação dinâmica dual

Na seção anterior vimos um procedimento simples baseado em programação dinâmica que objetiva minimizar os custos iterando sobre os recursos. Hassin (1992) descreveu uma versão diferente do algoritmo acima mais útil para seu propósito, que era desenvolver um algoritmo de aproximação para o RCSP, que será discutido mais a frente. Nesta seção descreveremos a versão de Hassin.

Na programação dinâmica dual, iteramos sobre os custos, partindo de uma quantidade unitária até encontrarmos um caminho viável, sempre minimizando o consumo de recursos dos caminhos computados. Digamos que $g_v(c)$ denota o menor consumo de recursos possível para um caminho de s a v que custa no máximo c. Então a seguinte recorrência pode ser

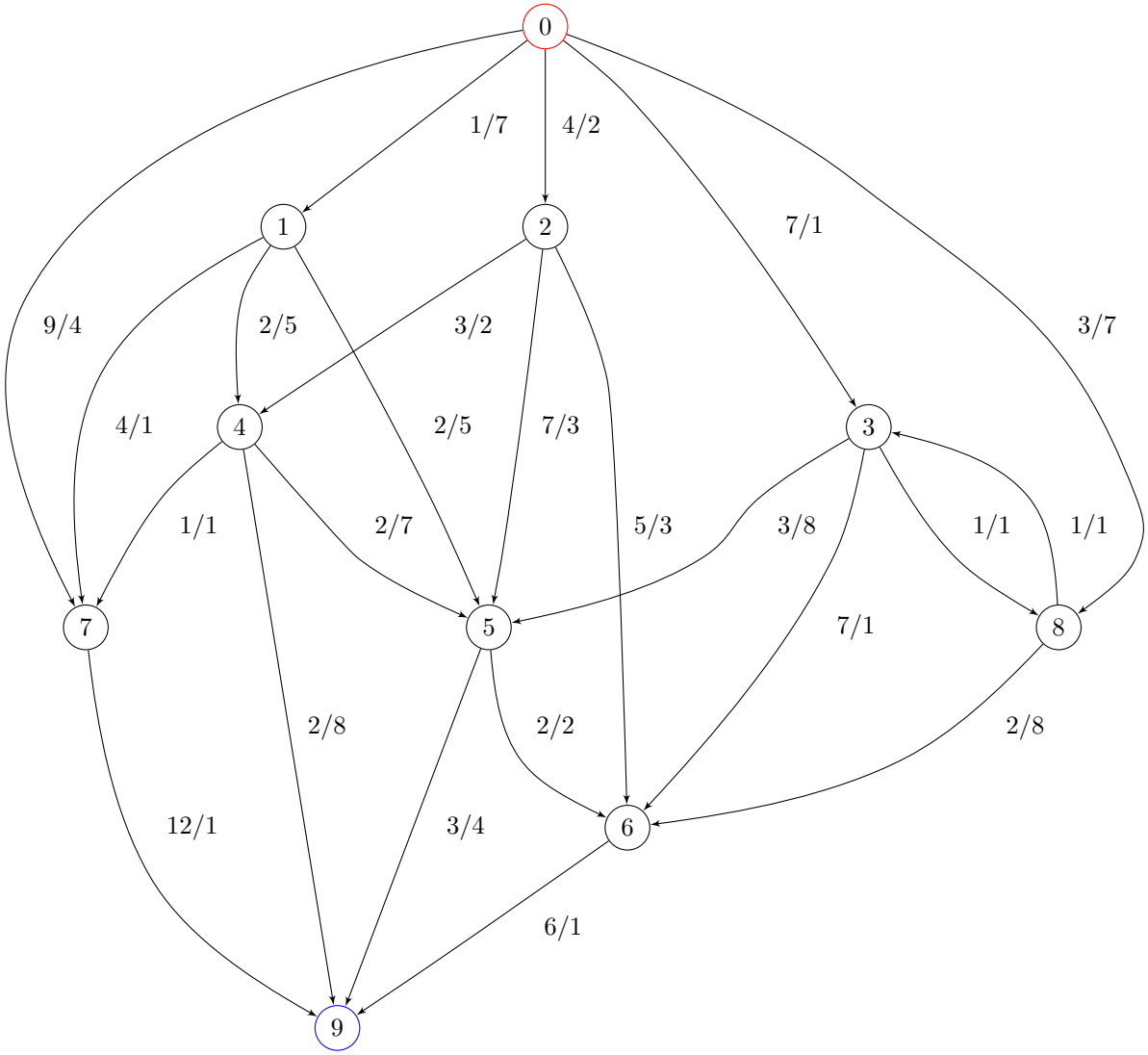


Figura 3.2: *Nesta figura temos a representação de um grafo usado para demonstrar a tabela de programação dinâmica primal gerada pelo algoritmo descrito acima.*

definida.

$$g_v(c) = \begin{cases} 0, & \text{se } v = s \\ & \text{e } c = 0, \ldots, OPT \\ \infty, & \text{se } v \neq s \\ & \text{e } c = 0 \\ \min\left\{ g_v(c-1), \min\limits_{u \mid c_{uv} \leq c} \{ g_u(c - c_{uv}) + r_{uv} \} \right\}, & \text{se } v \neq s \\ & \text{e } c = 1, \ldots, OPT \end{cases}$$

Observe que OPT não é um valor conhecido no início da execução, mas ele pode ser expresso como $OPT = \min\{c \mid g_t(c) \leq \lambda\}$. Devemos computar a função g iterativamente, primeiro para $c = 1$ e $v = 2, \ldots, n$, então para $c = 2$ e $v = 2, \ldots, n$, e assim sucessivamente,

até o primeiro valor c' tal que $g_t(c') \leq \lambda$. Só então teremos o conhecimento do valor $OPT = c'$. A seguir temos o algoritmo desenvolvido a partir da recorrência apresentada.

Algoritmo RECURSÃO-DUAL (v, c)

```
1   ψ(v, c) ← NIL
2   se v = s então
3       devolva 0
4   se c = 0 então
5       devolva ∞
6   recurso ← RECURSÃO-DUAL(v, c − 1)
7   ψ(v, c) ← ψ(v, c − 1)
8   para cada uv em A faça
9       se c_uv ≤ c então
10          valor ← RECURSÃO-DUAL(u, c − c_uv) + r_uv
11          se recurso > valor então
12              ψ(v, c) ← u
13              recurso ← valor
14  devolva recurso
```

A partir da recursão acima podemos implementar o seguinte algoritmo iterativo:

Algoritmo PROGRAMAÇÃO-DINÂMICA-DUAL (t, λ)

```
1   PD ← [ ]    ▷ tabela de programação dinâmica
2   ψ(s, 0) ← NIL
3   PD[s, 0] ← 0
4   para cada v em V \ {s} faça
5       ψ(v, 0) ← NIL
6       PD[v, 0] ← ∞
7   c ← 0
8   enquanto PD[t, c] > λ faça
9       c ← c + 1
10      ψ(s, c) ← NIL
11      PD[s, c] ← 0
12      para cada v em V \ {s} faça
```

13 $\psi(v, c) \leftarrow \psi(v, c-1)$

14 $\text{PD}[v, c] \leftarrow \text{PD}[v, c-1]$

15 ***para cada*** uv *em* A ***faça***

16 ***se*** $c_{uv} \leq c$ ***então***

17 ***se*** $\text{PD}[v, c] > \text{PD}[v, c - c_{uv}] + r_{uv}$ ***então***

18 $\psi(v, c) \leftarrow u$

19 $\text{PD}[v, c] \leftarrow \text{PD}[v, c - c_{uv}] + r_{uv}$

20 $OPT \leftarrow c$

21 ***devolva*** OPT, $\text{PD}[t, OPT]$

Um cuidado a se tomar com o algoritmo iterativo que acabamos de apresentar é que ele pressupõe que a instância é viável, ou seja, que possui solução. O "enquanto" da linha número oito só é interrompido quando encontramos uma solução ótima. Para se verificar a viabilidade da instância, pode-se executar o algoritmo de DIJKSTRA, usando a função de consumo de recurso como função custo, se o caminho encontrado possui consumo menor que o nosso limite, este caminho já é uma candidata a solução ótima.

Todas as recomendações e observações feitas a PROGRAMAÇÃO-DINÂMICA-PRIMAL são aplicáveis à PROGRAMAÇÃO-DINÂMICA-DUAL trocando-se os papéis entre custo e consumo de recursos. A complexidade de tempo do algoritmo sugerido acima é $O(mOPT)$ e a complexidade de espaço é $O(nOPT)$. Observe ainda que na PROGRAMAÇÃO-DINÂMICA-DUAL, devolvemos além do custo ótimo o valor mínimo de consumo de recurso encontrado, isso é necessário para construir o caminho associado ao custo ótimo.

Da mesma forma que acontece com o problema MOCHILA, o algoritmo de tempo pseudo-polinomial baseado em programação dinâmica pode ser adaptado para fornecer um algoritmo de aproximação para o RCSP, com o uso da técnica de escalar e arredondar. Discutiremos isso com mais detalhes seção 3.6.

A seguir, temos a tabela gerada com nossa programação dinâmica dual para o exemplo exibido para o algoritmo de programação dinâmica primal (figura 3.2). Como era de se esperar, obtivemos a mesma resposta, o caminho com menor custo até o vértice 9 que consome no máximo 10 unidades de recursos tem custo 14.

3.5.3 Programação dinâmica por rótulos

A abordagem de programação dinâmica por rótulos (*labeling*) pode ser vista como uma extensão dos métodos por programação dinâmica clássicos. Por causa deste motivo, mesmo esta abordagem não sendo exatamente uma abordagem por programação dinâmica, a mantivemos nesta seção.

$v \setminus c$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	∞	7	7	7	7	7	7	7	7	7	7	7	7	7	7
2	∞	∞	∞	∞	2	2	2	2	2	2	2	2	2	2	2
3	∞	∞	∞	∞	8	8	8	1	1	1	1	1	1	1	1
4	∞	∞	∞	12	12	12	12	4	4	4	4	4	4	4	4
5	∞	∞	∞	12	12	12	12	12	12	11	9	5	5	5	5
6	∞	∞	∞	∞	∞	14	14	14	14	5	5	5	5	5	2
7	∞	∞	∞	∞	13	8	8	8	5	4	4	4	4	4	4
8	∞	∞	∞	7	7	7	7	7	2	2	2	2	2	2	2
9	∞	∞	∞	∞	∞	20	16	16	16	12	12	12	12	12	9

Tabela 3.3: *Tabela de programação dinâmica dual gerada para o grafo da Figura 3.2.*

Motivação

Para entender a motivação por trás desta ideia vamos observar a estrutura da tabela de programação dinâmica construída pelo algoritmo Programação-Dinâmica-Primal. Como na tabela 3.2 que foi dada como exemplo, é comum termos várias células consecutivas em uma linha com um mesmo caminho associado. Ou seja, para um vértice v, temos um caminho de s até v associado a um intervalo de consumo de recursos na tabela. Isso ocorre porque quando não seguimos um caminho melhor terminando no vértice v que consume r unidades de recursos, associamos esta célula ao caminho que termina em v que consome no máximo $r-1$ unidades de recursos. Isso acaba gerando uma tabela com muita redundância. Compactar essa tabela de forma a evitar essa redundância é uma motivação da abordagem por rótulos.

Podemos representar a tabela 3.2 compactamente da seguinte forma, onde as células vazias estão associadas a última célula não vazia na linha.

$v \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0	0										
1	∞							1			
2	∞		4								
3	∞	7							4		
4	∞				7						
5	∞					11				10	
6	∞		14			9					
7	∞				9	8			5		
8	∞		8					3			
9	∞			20			15			14	

Tabela 3.4: *Tabela compacta de programação dinâmica primal gerada para o grafo da Figura 3.2.*

Podemos compactar ainda mais nossa estrutura se descartamos os caminhos inválidos (associados aos valores infinitos na tabela) e ao invés de usarmos uma matriz, usarmos um vetor indexado pelos vértices contendo listas de pares (ou rótulos) (c, r) para representar as

células relevantes (as não vazias), onde c representa o custo e r o consumo de recursos. A tabela 3.5 representa tal compactação da tabela 3.4.

v	
0	$(0, 0)$
1	$(1, 7)$
2	$(4, 2)$
3	$(7, 1)$, $(4, 8)$
4	$(7, 4)$
5	$(11, 5)$, $(10, 9)$
6	$(14, 2)$, $(9, 5)$
7	$(9, 4)$, $(8, 5)$, $(5, 8)$
8	$(8, 2)$, $(3, 7)$
9	$(20, 3)$, $(15, 6)$, $(14, 9)$

Tabela 3.5: *Tabela compacta de programação dinâmica primal gerada para o grafo da Figura 3.2.*

Definição

Temos várias versões de programação dinâmica por rótulos conhecidas, entre elas estão as descritas por Aneja *et al.* (1983); Desrochers e Soumis (1988); Desrosiers *et al.* (1995); Stroetmann (1997). Elas usam um conjunto de rótulos para cada vértice. Cada rótulo representa um caminho q partindo de s até o vértice que tal rótulo está associado, e é representado pelo par (c_q, r_q), o custo e consumo de recursos do caminho.

Um vw-caminho p é **dominado** por um vw-caminho q se $c_p \geq c_q$ e $r_p \geq r_q$, ou seja, q é mais eficiente que p tanto a respeito de custo quanto ao consumo de recursos [3]. Somente caminhos não dominados podem ter seus correspondentes rótulos armazenados na lista de cada vértice em ordem crescente de custo, o que implica que estão em ordem decrescente de consumo de recursos (no caso com um único recurso). Na figura 3.3 podemos ver esta ordem exemplificada. Joksch (1966) observou que a lista de rótulo não dominados são vértices de uma função escada (*step-function*) e apenas esses devem ser considerados para procurar uma solução ótima.

Pode-se dizer que um algoritmo baseado em programação dinâmica por rótulos, procura todos os caminhos não dominados, para cada vértice. Começando com o rótulo $(0, 0)$ na lista de rótulos do vértice s e as listas dos demais vértices vazias. O algoritmo estende a lista de rótulos conhecidos adicionando um arco ao final do caminho associado a cada rótulo, esses novos rótulos são armazenados caso não sejam dominados e sejam soluções viáveis.

Algoritmo PD-Por-Rótulos-Genérico (t, λ)

```
1  PD ← [ ]    ▷ tabela de programação dinâmica
2  ψ(s, (0, 0)) ← NIL
```

[3] Embora tenhamos descrito para o caso com um único recurso, a definição pode ser estendida para o caso com múltiplos recursos.

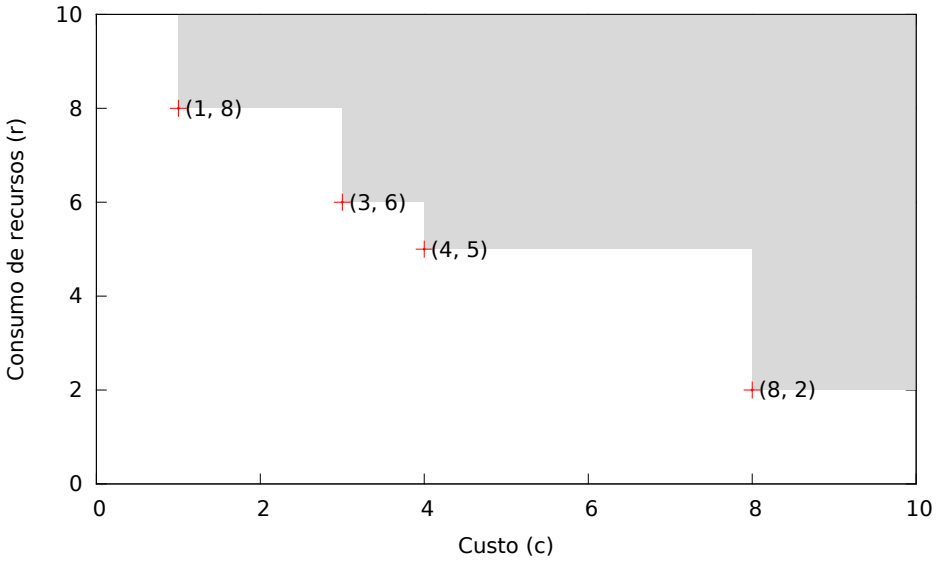


Figura 3.3: *Nesta figura temos o exemplo de uma lista de rótulos, exibidos em um plano para exemplificar a função escada. A área em cinza representa a área que é dominada pelos rótulos da lista.*

```
3   PD(s) ← PD(s) ∪ {(0, 0)}
4   enquanto existem novos rótulos não "expandidos" faça
5       para cada u em V faça
6           para cada rótulo (c, r) em PD[u] faça
7               para cada uv em A faça
8                   (c', r') ← (c + c_uv, r + r_uv)
9                   se (c', r') não é dominado por nenhum rótulo em PD[v] então
10                      remova os rótulos em PD[v] que são dominados por (c', r')
11                      PD[v] ← PD[v] ∪ {(c', r')}
12                      ψ(v, (c', r')) ← u
13  devolva o rótulo de menor custo em PD[t] que consome não mais que λ recursos
```

O algoritmo acima descreve o núcleo da abordagem de forma genérica. Temos ainda as variantes de rótulo permanente (*labeling setting*) e rótulo corretivo (*labeling correcting*) do algoritmo, que basicamente fazem a expansão dos rótulos em uma ordem específica. Independente da estratégia usada, a complexidade de pior caso permanece o mesmo, temos uma complexidade de tempo de $O(m\lambda \lg \lambda)$. A tabela 3.5 representa o estado da variável PD após a execução do algoritmo para o exemplo da figura 3.2.

3.6 ϵ-Aproximação

Hassin (1992) aplicou a técnica de escalar e arredondar para obter um esquema de apro-

ximação totalmente polinomial (FPTAS – *fully polynomial ϵ-approximation scheme*) para o SRCSP. Vamos ver, resumidamente, este método agora.

Na seção 3.5 nós vimos alguns procedimentos baseados em programação dinâmica. Para nossos propósitos aqui, a programação dinâmica dual (3.5.2) é bastante útil. Nela iteramos sobre o valor de custo c até o primeiro valor c' tal que $g_t(c') \leq \lambda$. Assim, temos o conhecimento do valor $OPT = c'$.

Agora, digamos que V seja um inteiro, e suponha que queremos testar se $OPT \geq V$ ou não. Um procedimento polinomial que responde essa questão poderia ser estendido para obtermos um algoritmo polinomial para encontrar OPT simplesmente usando uma busca binária. Como nosso problema é $\mathcal{NP}$-difícil, temos que nos satisfazer com um teste mais fraco.

Tomemos um ϵ fixo, $0 < \epsilon < 1$. Agora, nós iremos descrever um teste polinomial ϵ-aproximado com as seguintes propriedades: se tal teste devolve uma saída positiva, então definitivamente $OPT \geq V$. Se ele devolver uma saída negativa, então nós sabemos que $OPT < (V + \epsilon)$.

O teste arredonda o custo c_{ij} dos arcos, substituindo seu valor por:

$$\left\lfloor \frac{c_{ij}}{\epsilon V/(n-1)} \right\rfloor \cdot \frac{\epsilon V}{(n-1)}$$

.

Isto diminui todos os custos de arcos em no máximo $\epsilon V/(n-1)$, e todos os custos de caminhos em no máximo ϵV. Agora o problema pode ser resolvido aplicando o algoritmo de programação dinâmica dual ao grafo com os custos dos arcos escalados para $\lfloor c_{ij}/(\epsilon V/(n-1))\rfloor$. Os valores de $g_j(c)$ para $j = 2, ..., n$, são primeiro computados para $c = 1$, depois para $c = 2, 3, \cdots$ até que $g_n(c) \leq \lambda$ para algum $c = \hat{c} < (n-1)/\epsilon$, ou $c \geq (n-1)/\epsilon$.

No primeiro caso, um caminho de custo de no máximo

$$\frac{V\epsilon}{n-1}\hat{c} + V\epsilon < V(1+\epsilon)$$

foi encontrado, e segue que $OPT < V(1+\epsilon)$. No segundo caso, cada caminho tem $c' \geq (n-1)/\epsilon$ ou $c \geq V$, então $OPT \geq V$. Assim, o teste funciona como queríamos.

A complexidade de tempo é polinomial para um ϵ fixado, como explicada a seguir: Tomar a parte inteira de um número não negativo no intervalo $[0, U]$ pode ser feito em tempo $O(\lg(U))$ usando busca binária. Arredondar o custo dos arcos toma tempo $O(mlg(n/\epsilon))$ desde que nós escalamos somente os arcos com custo menor que V (o resultado é no máximo $(n-1)/\epsilon$). Depois executamos $O(n\epsilon)$ iterações do algoritmo acima que novamente toma tempo $O(m \lg(n/\epsilon))$. E essa é também a complexidade do procedimento de teste inteiro.

Agora nós usamos este teste para chegar a um FPTAS baseado em escalar e arredondar: Para aproximar OPT nós primeiramente determinamos um limite superior (UB) e um limite inferior (LB). O limite superior UB pode ser obtido como a soma dos $n-1$ arcos com maior

custo, ou o custo do caminho que consome menos recursos. O limite inferior LB pode ser 0 ou o caminho de menor custo.

Se $UB \leq (1+\epsilon)LB$, então UB é uma ϵ-aproximação de OPT. Suponha que $UB > (1+\epsilon)LB$. Seja V um dado valor $LB < V < UB/(1+\epsilon)$. O nosso procedimento de teste pode agora ser aplicado para melhorar os limites para OPT. Especificamente, ou LB é aumentado ou UB é diminuído para $V(1+\epsilon)$. Executando uma sequência de testes, a razão UB/LB pode ser reduzida. Uma vez que a razão atinja o valor de uma constante pré-definida, então uma ϵ-aproximação pode ser obtida aplicando-se o algoritmo de programação dinâmica para o grafo com os custos dos arcos escalados para $\lfloor c_{ij}/(LB/(n-1))\rfloor$. O erro final é no máximo $\epsilon LB < \epsilon OPT$.

A complexidade de tempo para o último passo é $O(mn/\epsilon)$. A redução da razão UB/LB é melhor alcançada por busca binária no intervalo (LB, UB) em escala logarítmica. Depois de cada teste nós atualizamos os limites. Para garantir uma rápida redução de razão nós executamos o teste com o valor x tal que $UB/x = x/LB$, que é $x = (LB \cdot UB)^{1/2}$. Tomando $\epsilon = 2$ por exemplo, o número de testes necessários para reduzir a razão para abaixo de ϵ é $O(\lg\lg(UB/LB))$ e cada teste toma tempo $O(mn/\epsilon)$. Hassin (1992) mostrou como computar um valor inteiro próximo a $O(UB \cdot LB)^{1/2}$ em tempo $O(\lg\lg(UB/LB))$. Isto dá uma complexidade de tempo, total de $O(\lg\lg(UB/LB)(m(n\epsilon) + \lg\lg(UB/LB)))$ para este algoritmo.

Hassin (1992) também mostrou uma FPTAS cuja complexidade depende somente do número de variáveis e $1/\epsilon$ e possui complexidade de tempo de $O(m(n^2/\epsilon)\lg(n/\epsilon))$. O melhor FPTAS, até o momento, para o RCSP foi obtido por Phillips (1993) que atingiu a complexidade de tempo de $O(m(n/\epsilon) + (n^2/\epsilon)\lg(n/\epsilon))$.

3.7 Ranqueamento de caminhos

O problema de ranqueamento de caminhos, mais conhecido como o problema dos k menores caminhos (KSP – *k shortest path*), consiste me determinar o k-ésimo menor caminho em um grafo. Este problema foi estudado primeiramente por Hoffman e Pavley (1959). Desde então, o problema tem sido bastante estudado e várias soluções foram propostas.

Muitos desses métodos têm complexidade de tempo polinomial para um k fixo. Eppstein (1994) descreveu um algoritmo com complexidade de tempo $O(m + n\lg n + k)$ que resolve o problema quando ciclos são permitidos. Podemos usar o KSP para resolver o RCSP. Neste caso, ignoramos o k e enumeramos os caminhos em ordem não decrescente de custo até encontrar um caminho viável, tal abordagem resulta em um algoritmo de complexidade exponencial para o RCSP.

Algoritmos para resolver o KSP não devem ser usados diretamente para resolver o RCSP. Porém, eles têm sido usados com sucesso como sub-problema para métodos mais sofisticados, como a relaxação lagrangiana proposta por Handler e Zang (1980). Uma extensa bibliografia

para o problema dos k menores caminhos pode ser encontrada em Eppstein (2012).

3.8 Relaxação Lagrangiana

A seguir vamos apresentar o algoritmo proposto por Handler e Zang (1980), que se utiliza de uma relaxação de uma formulação em programação linear.

Nela temos uma variável x_{uv} para cada $uv \in A$, com a seguinte interpretação: $x_{uv} = 1$ significa dizer que o arco uv está na solução e $x_{uv} = 0$ o contrário. Vamos nos referir ao problema abaixo como (P).

$$
\begin{aligned}
\text{minimize} \qquad & c(x) = \sum_{uv \in A} c_{uv} x_{uv} \\
\text{sob as restrições} \qquad & \sum_{vw \in A} x_{vw} - \sum_{uv \in A} x_{uv} = \begin{cases} 1, & \text{para } v = s \\ 0, & \text{para todo } v \in V \setminus \{s, t\} \\ -1, & \text{para } v = t \end{cases} \qquad (1) \\
& \sum_{uv \in A} r_{uv} x_{uv} \leq \lambda \qquad (2) \\
& x_{uv} \in \{0, 1\},\ uv \in A \qquad (3)
\end{aligned}
$$

Na formulação acima, a restrição (3) é responsável por delimitar os possíveis valores que um componente do vetor x pode assumir. A restrição (1), por sua vez, é responsável por garantir que para um vetor x ser solução viável do problema, ele deve "conter" um caminho do vértice s ao vértice t. Por fim, a restrição (2) nos garante que o conjunto de arcos induzido por um vetor x viável, não excede os recursos disponíveis.

Por conveniência de notação, nós iremos definir os seguintes termos. Vamos denotar por $\mathcal{X}$ o conjunto de vetores x que satisfazem as restrições (1) e (3), ou seja, vetores x que contêm um caminho de s a t. Vamos definir também a seguinte função.

$$g(x) = \sum_{uv \in A} r_{uv} x_{uv} - \lambda$$

tal que $x \in \mathcal{X}$.

Com as definições acima, resolver (P) é equivalente a resolver o seguinte.

$$c^* = c(x^*) = min\{\ c(x) \mid x \in \mathcal{X} \text{ e } g(x) \leq 0\ \}$$

Agora iremos aplicar a teoria da dualidade lagrangiana (como apresentada, por exemplo, em Geoffrion (1974), Fisher (1978)) para resolver o RCSP. Tendo em vista que o problema pode ser resolvido em tempo polinomial quando a restrição $g(x) \leq 0$ é relaxada (sem esta restrição, o problema se reduz a buscar caminho mínimo), nossa estratégia será justamente retirar essa "restrição complicada" do conjunto de restrições e a usarmos como penalidade

na função objetivo (técnica essa que é a essência da relaxação lagrangiana).

Para qualquer $u \in \mathbb{R}$, definimos a função Lagrangiana:

$$L(u) = \min_{x \in \mathcal{X}} L(u, x), \text{ onde } L(u, x) = c(x) + ug(x)$$

Perceba que encontrar uma solução para $L(u)$ é o problema de caminho mínimo no grafo original, porém onde os custos dos arcos foram alterados para $c_{uv} + ur_{uv}$, $uv \in A$. Temos que $L(u) \leq c^*$ para qualquer $u \geq 0$ (teorema da dualidade fraca), pois

$$g(x^*) \leq 0 \Rightarrow L(u) \leq c(x^*) + ug(x^*) \leq c(x^*) = c^*,$$

o que nos permite usar $L(u)$ como um limite inferior para o problema original. Para encontrarmos o limite inferior mais justo possível, resolvemos o problema dual a seguir. Vamos nos referir ao problema abaixo como (D).

$$L^* = L(u^*) = \max_{u \geq 0} L(u)$$

Pode ser que exista uma folga na dualidade (*duality gap*), ou seja, pode ser que L^* seja estritamente menor que c^*. Nos casos em que existir essa folga, teremos que trabalhar um pouco mais para eliminá-la.

Vamos, agora, descrever um método para resolver o problema (P), que usa como passo, resolver o problema (D). Por praticidade vamos denotar por $x(u)$ como um caminho que possui valor ótimo associado à função $L(u)$.

O mais natural é que, como primeiro passo, verifiquemos se o menor caminho (não limitado, $\min_{x \in \mathcal{X}} c(x)$) respeita nossas restrições. Vamos chamar esse caminho de $x(0)$, pois $L(0) = c^*$.

- Se $g(x(0)) \leq 0$, então $x(0)$ é claramente uma solução ótima de (P).
- Senão, $x(0)$ nos serve, pelo menos, como limite inferior para uma solução ótima.

Como segundo passo, devemos verificar se o caminho que consome menor quantidade de recursos ($\min_{x \in \mathcal{X}} g(x)$) respeita nossas restrições. Vamos chamar esse caminho de $x(\infty)$, pois para valores muito grandes de u, o parâmetro $c(x)$ na função $L(u)$ é "dominado" por $ug(x)$.

- Se $g(x(\infty)) > 0$, o problema não tem solução, pois o consumo do caminho que consome a menor quantidade de recursos ultrapassa o limite dado.
- Senão, $x(\infty)$ é uma solução viável para a instância e nos serve de limite superior para o problema.

Agora com os resultados dos passos anteriores, se não temos ainda a solução ou a prova de que a instância é inviável, temos a seguinte situação: Dois caminhos, $x(0)$, que **não é**

solução e é um **limite inferior**; e $x(\infty)$, que **é solução viável** e é um **limite superior**, $g(x(0)) > 0$ e $g(x(\infty)) \leq 0$.

Da forma como desenvolvemos a solução até então, podemos interpretar cada caminho no grafo como uma reta no espaço (u, L) da forma $L = c(x) + ug(x)$, onde u é nossa variável, $c(x)$ é nosso termo independente (ponto onde a reta corta o eixo L) e $g(x)$ é nosso coeficiente angular. Isso nos permite dar uma interpretação geométrica para a função $L(u)$, que será o envelope inferior do conjunto de retas (caminhos), ou seja, $L(u)$ será um conjunto de segmentos de retas, tal que cada ponto (u, L) nesses segmentos está abaixo ou na mesma altura de qualquer ponto (u, L') pertencente às retas associadas aos caminhos.

Com a interpretação geométrica dos caminhos, temos a informação que retas **crescentes** são associadas a caminhos **não viáveis** para o nosso problema, enquanto as retas **não crescentes** são **soluções viáveis**. Como estamos procurando o valor de L^* (o ponto "mais alto" da função $L(u)$) vamos analisar o ponto (u', L') que é a intersecção das retas associadas a $x(0)$ e $x(\infty)$.

$$u' = (c(x(\infty)) - c(x(0)))/(g(x(0)) - g(x(\infty)))$$

$$L' = c(x(0)) + u' \cdot g(x(0))$$

É fato que $u' \geq 0$, pois $c(x(0))$ é mínimo, $g(x(\infty)) \leq 0$ e $g(x(0)) > 0$. Claramente, se existem apenas dois caminhos o ponto (u', L') é o que maximiza $L(u)$. O mesmo acontece quando existem vários caminhos e $L(u') = L'$, ou seja, $L(u', x) \geq L'$ para qualquer $x \in \mathcal{X}$. Um último caso "especial" é quando existe um caminho $x_h \in \mathcal{X}$ tal que $g(x_h) = 0$ e $L(u') = L(u', x_h) < L'$. Como a reta associada a x_h é horizontal, ela limita superiormente $L(u)$, e como temos o ponto $(u', L(u'))$ sobre ela, $c^* = c(x_h) = L^* = L(u')$ (neste caso não existe folga na dualidade).

Escrevemos, especificamente, sobre os caminhos $x(0)$ e $x(\infty)$ no parágrafo anterior, mas o que foi dito vale no caso geral, onde temos dois caminhos disponíveis $x^+, x^- \in \mathcal{X}$, tal que $g^+ \equiv g(x^+) > 0$, $g^- \equiv g(x^-) \leq 0$ e $c^- \equiv c(x^-) \geq c^+ \equiv c(x^+)$. Então, temos que $u' = (c^- - c^+)/(g^+ - g^-)$ e $L' = c^+ + u'g^+$ definem o ponto de intersecção, no espaço (u, L), das retas associadas aos caminhos x^+ e x^-. Se $L(u') = L'$ ou se $g(x(u')) = 0$, então $L(u^*) = L(u')$ é uma solução do nosso problema dual (D). Caso contrário, se $g(x(u')) < 0$, então $x(u')$ é o nosso novo caminho x^-, e se $g(x(u')) > 0$, então $x(u')$ é o nosso novo caminho x^+. O procedimento se repete até determinarmos uma solução do problema (D). Com a realização do procedimento temos disponíveis um limite inferior LB (*lower bound*) e um limite superior UB (*upper bound*) para o valor de c^*. Nós temos que $LB = L(u^*) \leq c(x^*)$ (pelo teorema da dualidade fraca); e por definição segue que qualquer x^- usado durante o procedimento é uma solução viável, assim UB é o valor do último c^- ou o valor de $c(x(u'))$ associado com o último caminho $x(u')$ **se** $g(x(u')) \leq 0$.

Tendo resolvido o problema (D), temos limites $LB \leq c^* \leq UB$ e uma solução viável associada a UB para o RCSP. Quando $LB = UB$, esta solução é ótima. Porém, quando $LB < UB$ temos uma folga na dualidade. Para eliminarmos essa folga poderíamos usar

um algoritmo de k-ésimo menor caminho (*k-shortest path*) a partir do primeiro caminho x tal que $c(x) \geq LB$ até o primeiro x_k tal que $g(x_k) \leq 0$. Como esse algoritmo precisa do conhecimento de todos os caminhos anteriores para gerar o próximo, essa abordagem não tomaria nenhum proveito da resolução do dual. Em contraste, determinar o k-ésimo menor caminho em relação à função Lagrangiana $L(u^*, x)$ (o que é equivalente a usar a função c' como custo, $c'_{uv} = c_{uv} + u^* \cdot r_{uv}$, $uv \in A$) é perfeitamente aplicável a partir da solução dual.

Vamos denotar $L_k(u^*)$, para $k = 1, 2, \ldots$, como sendo o valor do k-ésimo menor caminho $x_k \in \mathcal{X}$ em relação a função de custo $L(u^*, x)$. Os caminhos x_1 e x_2 já são conhecidos, eles são x^+ e x^- respectivamente, pois se interceptam no ponto $(u^*, L(u^*))$, o que significa que possuem valor mínimo em relação a função $L(u^*, x)$. Iterando sobre o k-ésimo caminho, $k \geq 3$, nós atualizamos UB quando $g(x_k) \leq 0$ e $c(x_k) < UB$; e atualizamos $LB = L_k(u^*)$, pois essa é uma sequência não decrescente ($L_{k-1}(u^*) \leq L_k(u^*)$). O procedimento continua até que $LB \geq UB$, e então temos a solução do problema (P), associada a $UB = c^*$, solução do RCSP.

Algoritmo RCSP-*LAGRANGIANA*(*G*, *s*, *t*, $k = 1$, r, λ, c)

▷ *Inicialização*

1 $x_0, c_0, g_0 \leftarrow L(0)$
2 **se** $g_0 \leq 0$
3 **então** $x^*, c^* \leftarrow x_0, c_0$
4 **senão** $x^+, c^+, g^+ \leftarrow x_0, c_0, g_0$
5 $x_\infty, c_\infty, g_\infty \leftarrow L(\infty)$
6 **se** $g_\infty > 0$
7 **então** $x^*, c^* \leftarrow NULL, NULL$ ▷ *Não tem solução!*
8 **senão** $x^-, c^-, g^- \leftarrow x_\infty, c_\infty, g_\infty$

▷ *Resolvendo o Dual*

9 **se** $x^+ \neq NIL$ *e* $x^- \neq NIL$ ▷ *Se entrou nos dois "então" acima*
10 $LB \leftarrow 0$; $UB \leftarrow c^-$
11 **enquanto** $LB < UB$ **faça**
12 $u' \leftarrow (c^- - c^+)/(g^+ - g^-)$; $L' \leftarrow c^+ + u'g^+$; $x', c', g' \leftarrow L(u')$
13 **se** $g' = 0$
14 **então** $x^*, c^* \leftarrow x', c'$; $LB \leftarrow UB \leftarrow c'$
15 ***PÁRA*** *o enquanto*
16 **se** $L(u') = L'$ *e* $g' < 0$
17 **então** $LB \leftarrow L'$; $UB \leftarrow \min\{UB, c'\}$; $x^- \leftarrow x'$; $u^* \leftarrow u'$

18 ***PÁRA*** *o enquanto*

19 **se** $L(u') = L'$ e $g' > 0$

20 **então** $LB \leftarrow L'$; $u^* \leftarrow u'$

21 ***PÁRA*** *o enquanto*

22 **se** $L(u') < L'$ e $g' > 0$

23 **então** $x^+, c^+, g^+ \leftarrow x', c', g'$

24 **se** $L(u') < L'$ e $g' < 0$

25 **então** $x^-, c^-, g^- \leftarrow x', c', g'$; $UB \leftarrow \min\{UB, c'\}$

▷ *Eliminando a folga da dualidade*

26 $x_1, x_2 \leftarrow x^+, x^-$; $k \leftarrow 2$

27 **enquanto** $LB < UB$ **faça**

28 $k \leftarrow k + 1$; $x_k, c_k, g_k \leftarrow L_k(u^*)$; $LB \leftarrow L_k(u^*)$

29 **se** $g_k \leq 0$ e $c_k < UB$

30 **então** $x^-, UB \leftarrow x_k, c_k$

31 **se** $LB \geq UB$

32 **então** $x^*, c^* \leftarrow x^-, UB$

33 **devolva** x^*, c^*

A seguir temos um exemplo do algoritmo executando no grafo representado na figura 3.4. No exemplo temos $s = 0$, $t = 9$, e o limite de recursos é 1. Os caminhos e retas em vermelho estão associados à x^+ enquanto os azuis à x^-. Já os caminhos verdes são os caminhos calculados no procedimento que elimina a folga do problema dual.

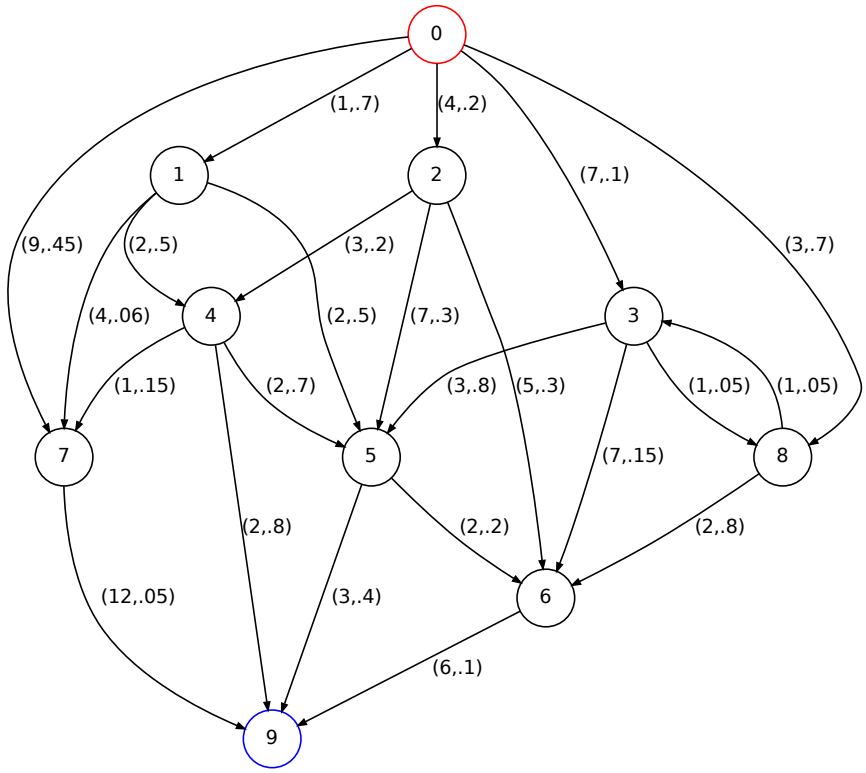


Figura 3.4: *Grafo exemplo; os rótulos dos arcos representam* (c_{uv}, r_{uv}).

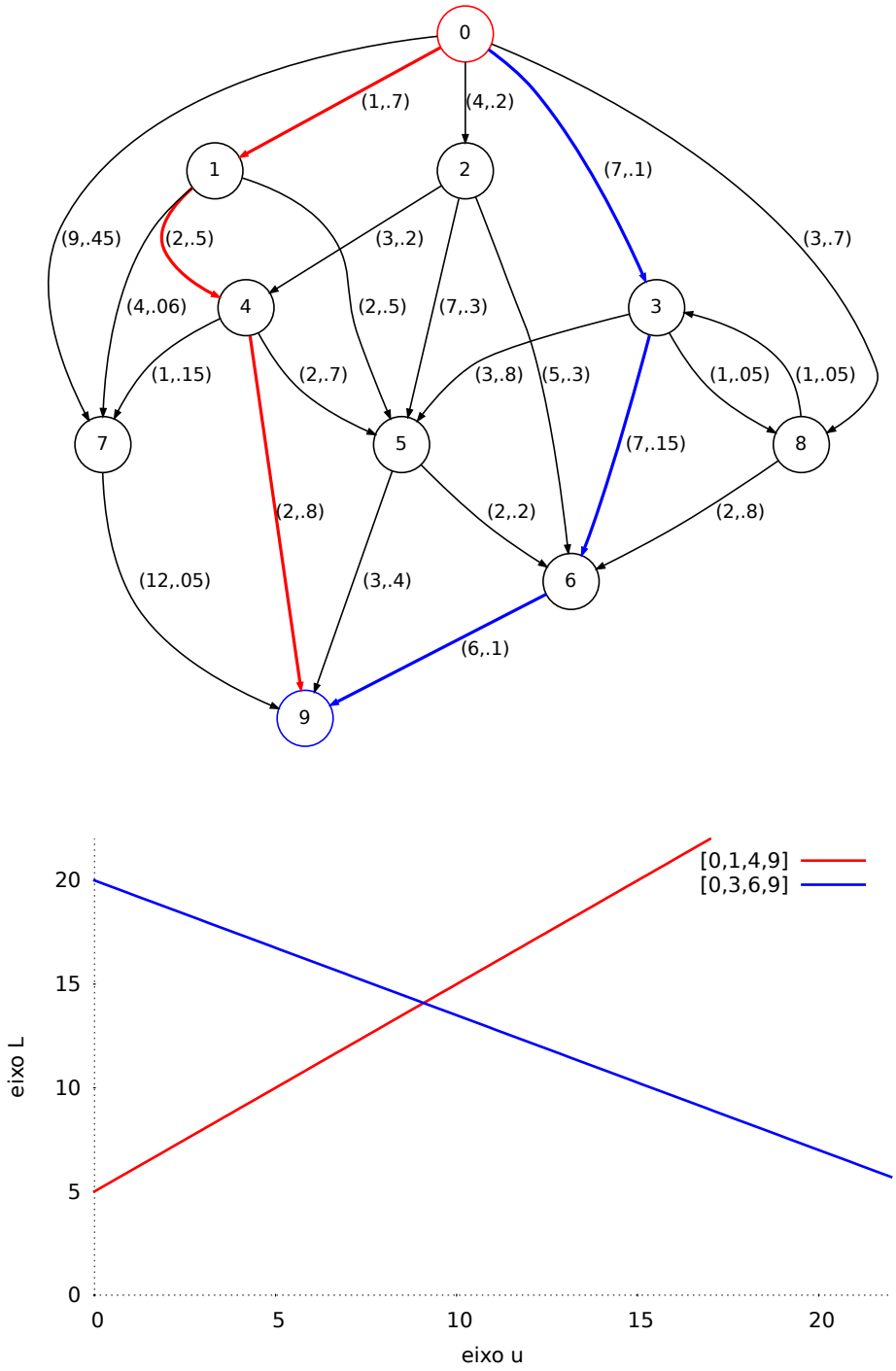


Figura 3.5: *Inicialmente encontramos* $x_0 = [0, 1, 4, 9]$ *e* $x_\infty = [0, 3, 6, 9]$. *As restas correspondentes são* $L = 5 + u$ *e* $L = 20 - 0.65u$ *que se intersectam no ponto* $u' = 9.09$ *e* $L' = 14.09$. *Nesta primeira iteração temos* $LB = 5$ *e* $UB = 20$.

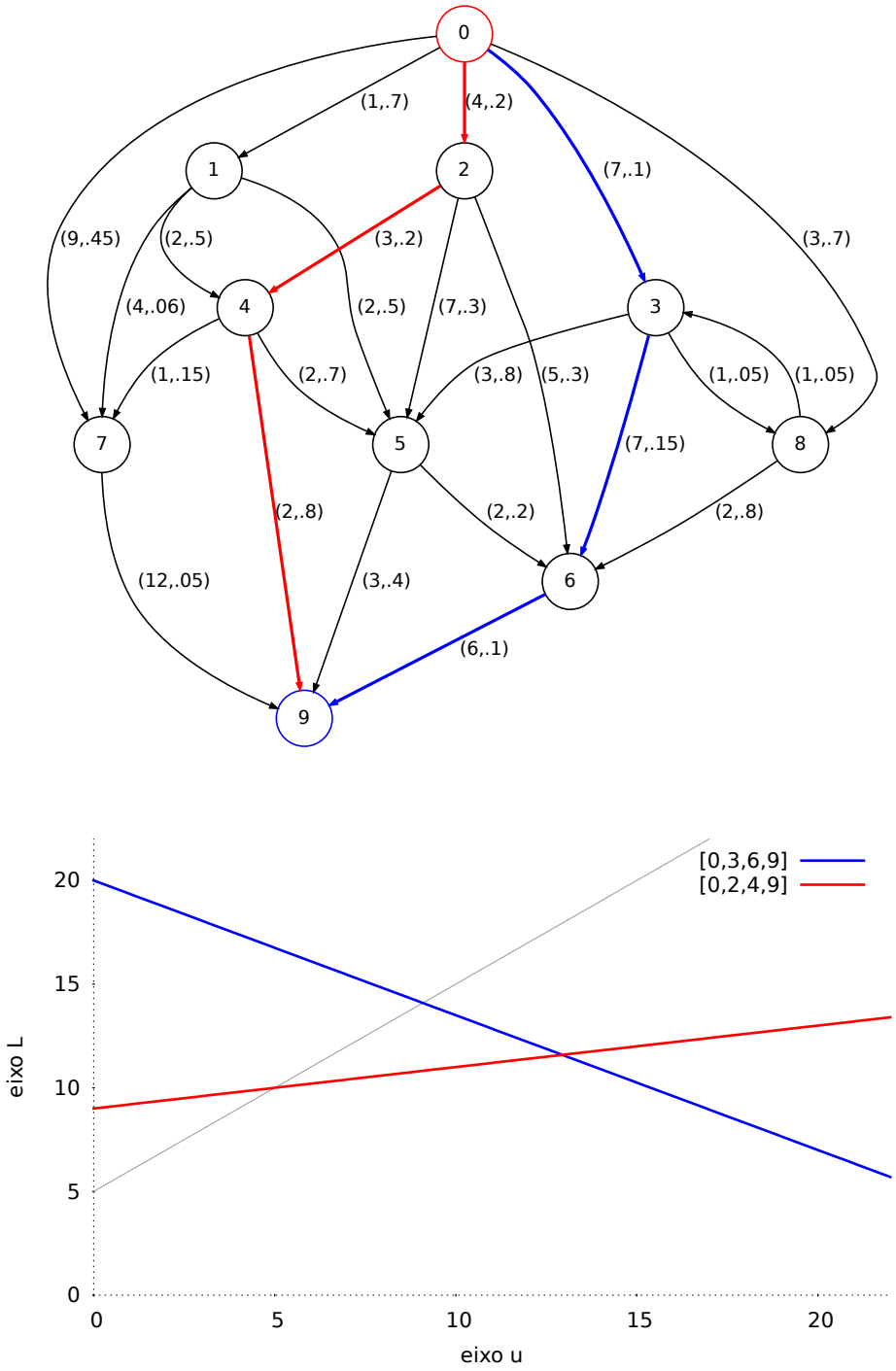


Figura 3.6: *Calculando a função L no ponto u' encontramos um novo caminho para x^+ que passa a ser $[0, 2, 4, 9]$. Temos agora $LB = 10.8$ e $UB = 20$. A interseção das restas é sobre o ponto $u' = 12.94$ e $L' = 11.59$.*

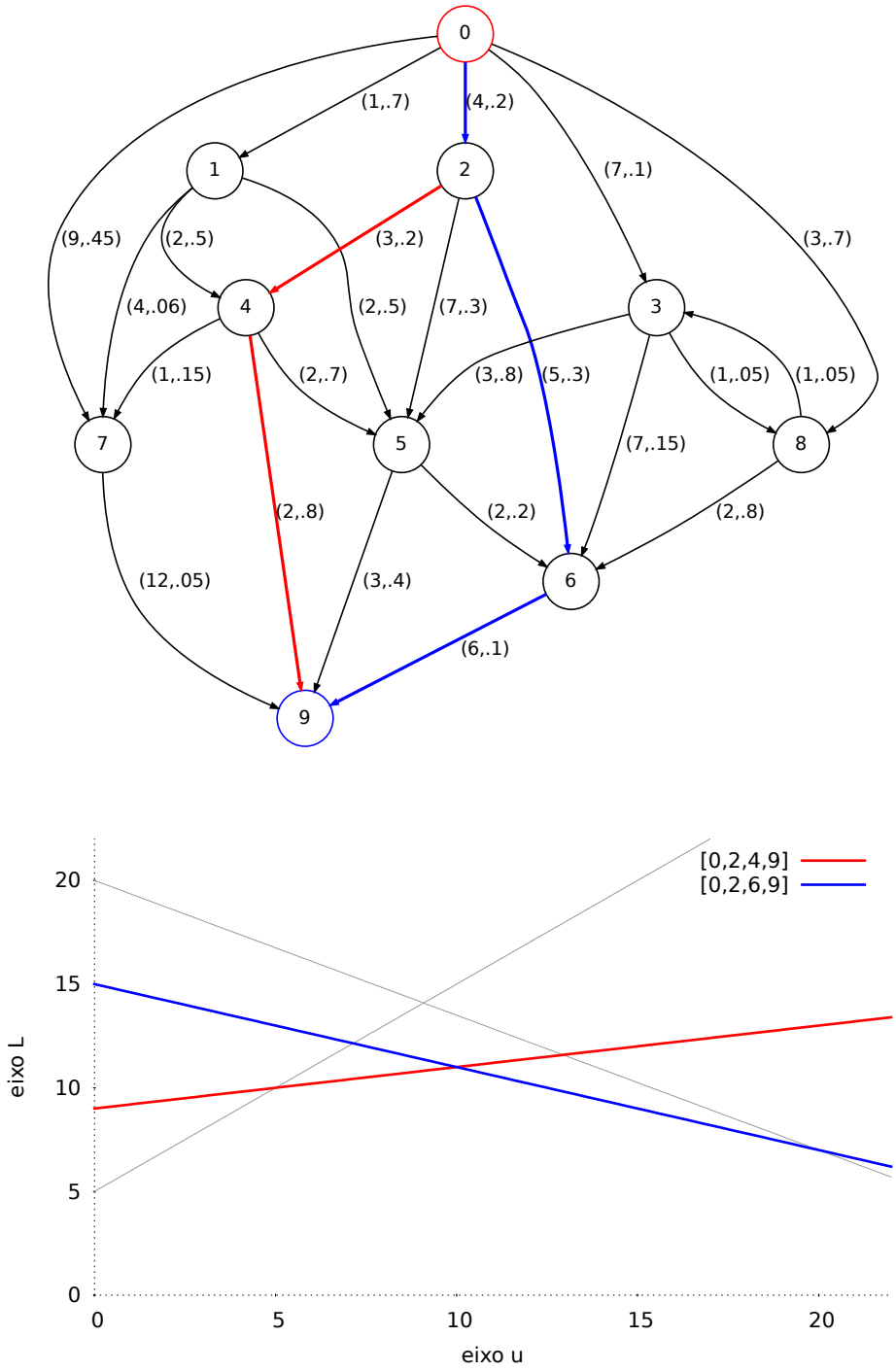


Figura 3.7: *Avaliando a função L no ponto de interseção encontramos um caminho para substituir x^-, a saber $[0, 2, 6, 9]$. Agora na interseção de x^+ e x^- encontramos o ponto máximo da função L que é $L^* = 11$ e $u^* = 10$. Terminamos o dual com $LB = 11$ e $UB = 15$.*

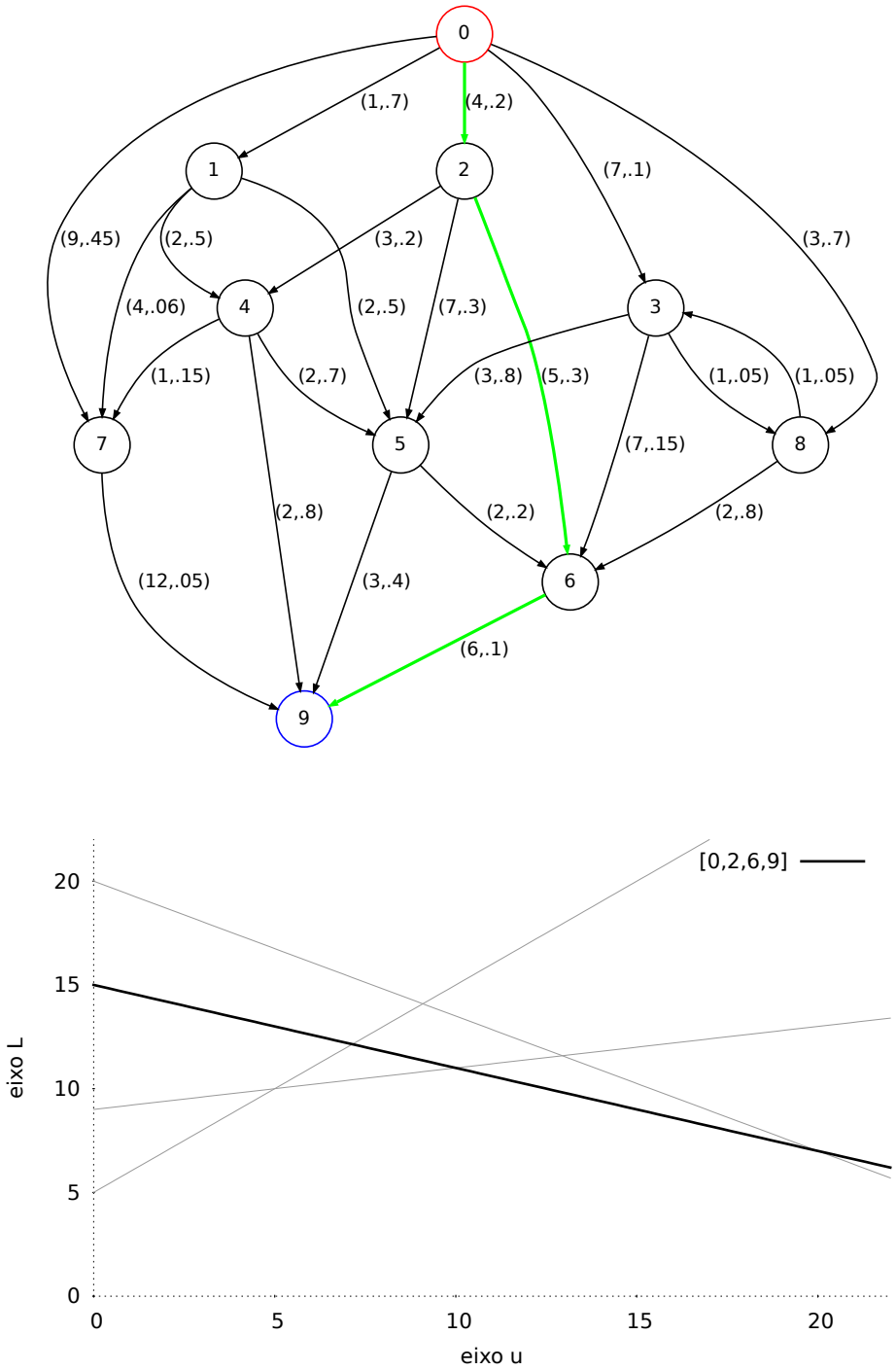


Figura 3.8: *A partir daqui, executamos o algortimo de Yen para eliminação da folga da dualidade.*

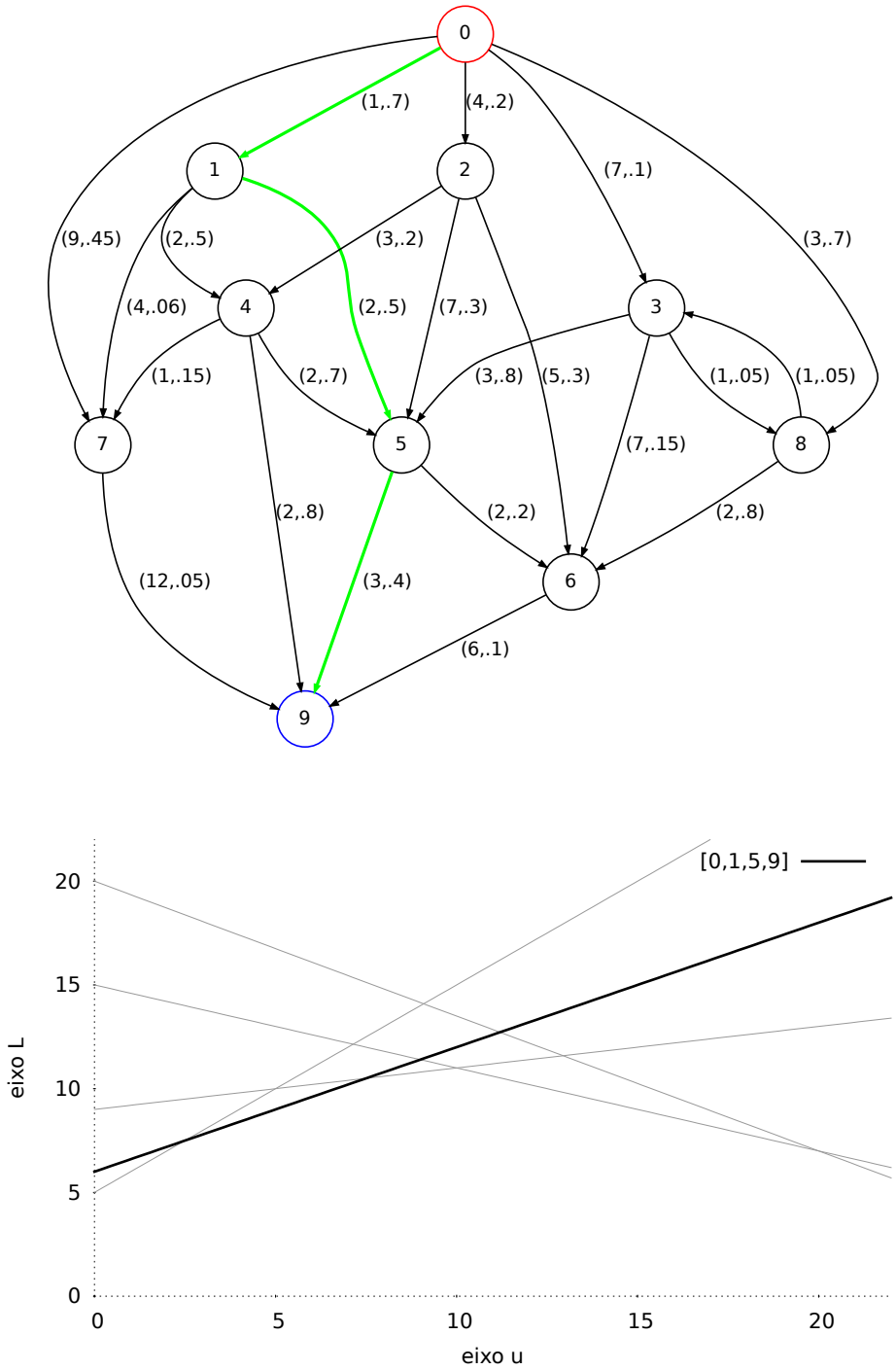


Figura 3.9: *Eliminando a folga da dualidade.*

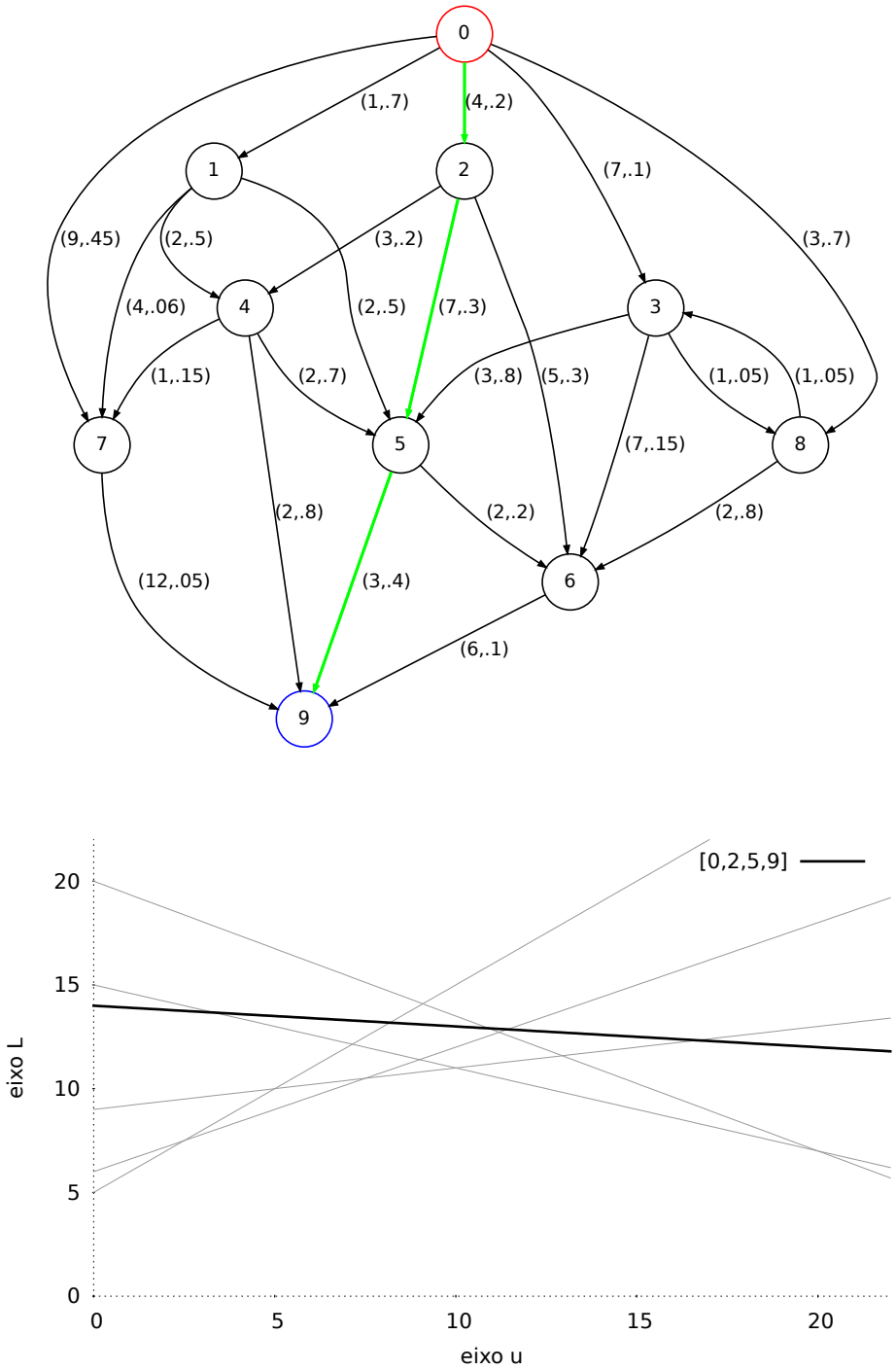


Figura 3.10: *Eliminando a folga da dualidade.*

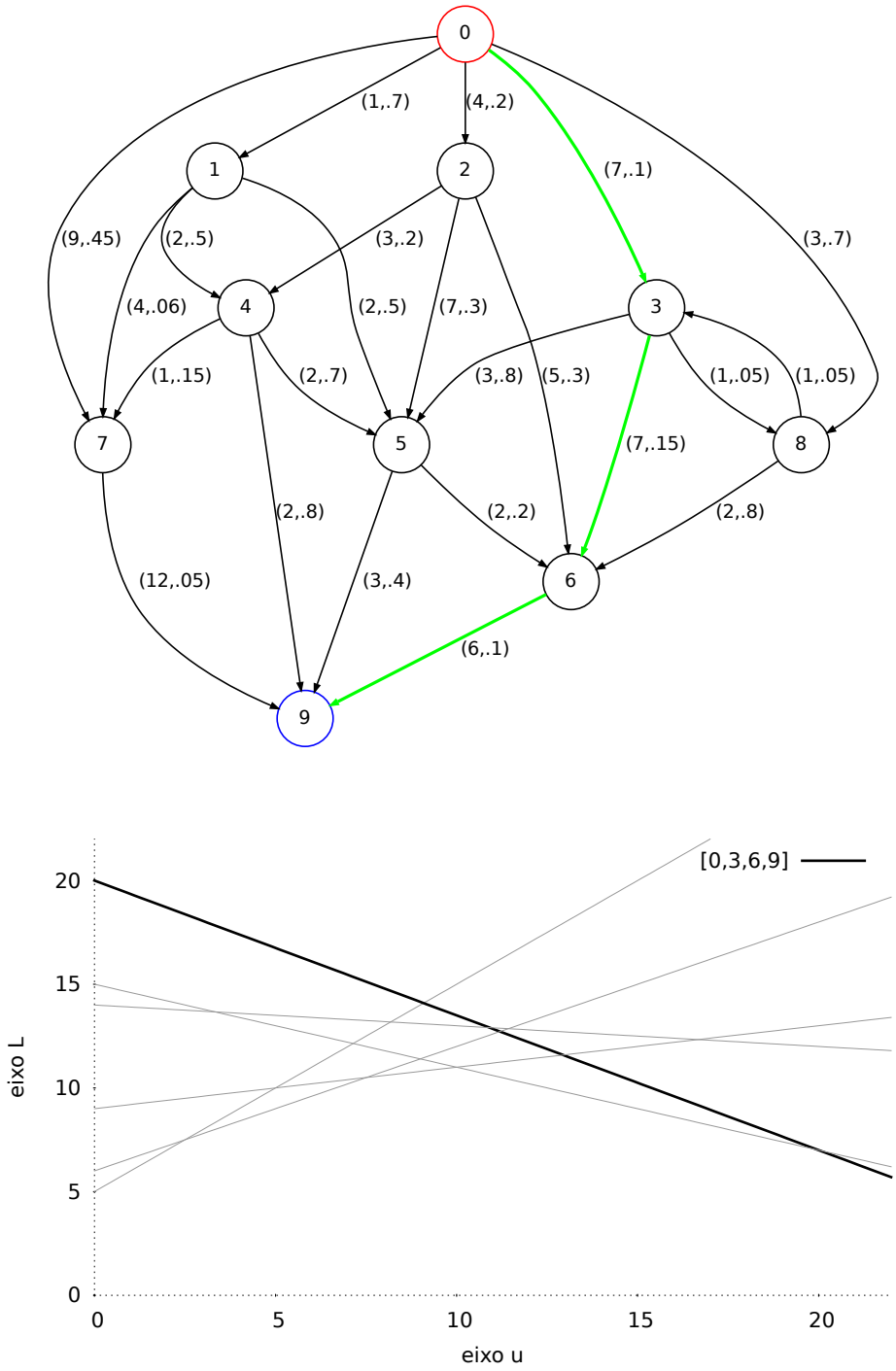


Figura 3.11: *Eliminando a folga da dualidade.*

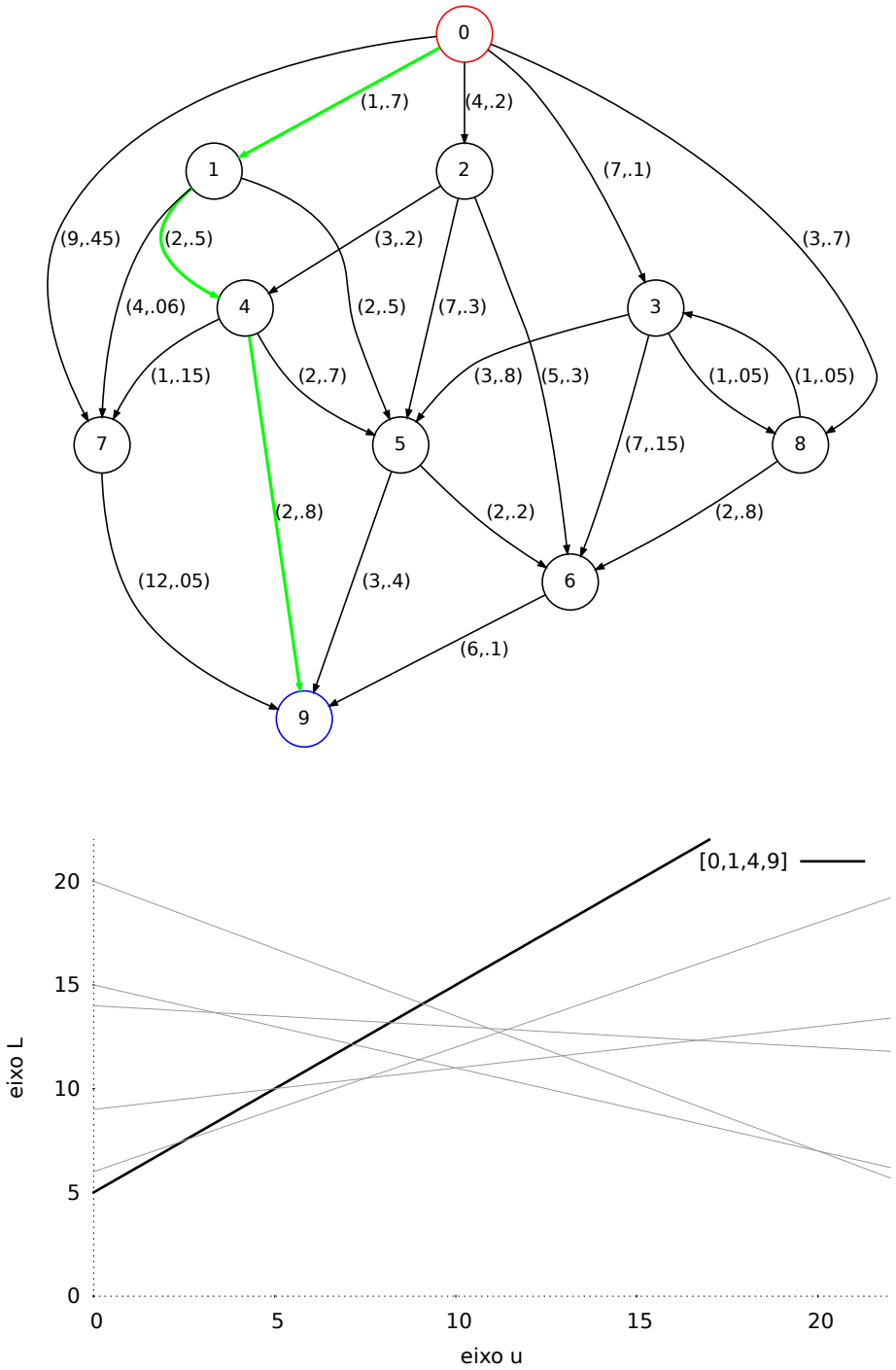


Figura 3.12: *Eliminando a folga da dualidade.*

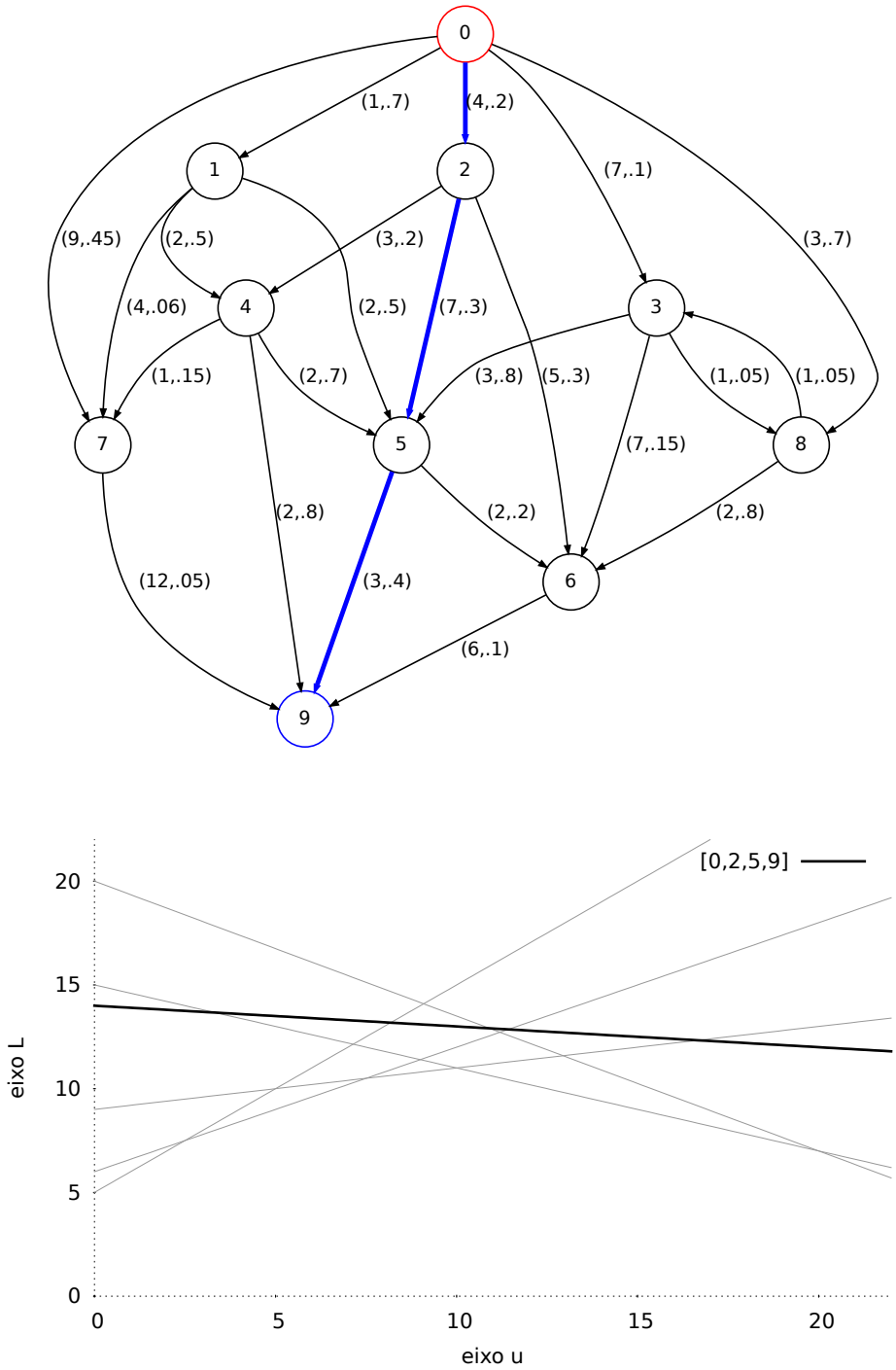


Figura 3.13: *Solução encontrada para o exemplo.*

Capítulo 4

Experimentos

No presente capítulo, mostraremos resultados de experimentos com alguns dos diferentes métodos propostos para o problema de caminhos mínimos com recursos limitados, são eles programação dinâmica primal, algoritmo de Yen e o método proposto por Handler e Zang. Por uma questão de simplicidade, todas as implementações são para a versão do problema com um único recurso. Como a performance dos algoritmos é variável para diferentes tipos de grafos, nós iremos experimentá-los com dados reais e randômicos usando os seguintes tipos de grafos: grafos em grade, grafos de ruas, grafos de curva de aproximação, e grafos aleatórios.

4.1 Ambiente computacional

Todas os nossos experimentos foram executados em um *laptop Sony Vaio*, com processador *Intel Core i3 CPU M 330 @ 2.13GHz* e 2GB de memória RAM. O sistema operacional utilizado é o Ubuntu, versão *12.04 LTS 64 bit, kernel 3.2.0-27-generic*. Os códigos foram implementados usando a linguagem de programação C++ e compilados usando o compilador g++ da GNU, versão 4.6.3.

4.2 Dados de Teste

Nós usamos os seguintes quatro tipos de grafos:

DEM: Modelos digitais de elevação (*digital elevation models*) são grafos em forma de grade onde cada vértice tem um valor de altura associado. Usamos exemplos de modelos de elevações da Europa cedidos por Mark Ziegelmann (Ziegelmann (2001)).

Em nossos exemplos DEM, temos que se o arco uv está no grafo o arco vu também está. O valor de m é aproximadamente $4n$. Utilizamos o valor absoluto das diferenças de altura dos vértices como função custo, esses valores estão no intervalo $[0, 600]$. Nós usamos inteiros aleatórios dentro do intervalo $[10, 20]$ como consumo de recursos. Por fim, estamos interessados em minimizar a diferença de altura acumulada no caminho com limitação do

comprimento do caminho.

	Número de Vertices	Número de Arcos
Austria grande	41600	165584
Austria pequeno	11648	46160

Tabela 4.1: *Casos de teste do tipo* ***DEM***

ROAD: Temos exemplo de grafos de ruas dos Estados Unidos fornecidos por Mark Ziegelmann. Para vértices adjacentes, temos novamente arcos nas duas direções, e a estrutura nos dá m aproximadamente $2.5n$. Nós usamos um índice que avalia o congestionamento como função de custo. Definimos os congestionamentos como inteiros entre $[0, 100]$. Nossa função custo é a distância Euclidiana entre os pontos finais dos arcos. Estas distâncias são números de ponto flutuante no intervalo $[0, 7]$. Estamos interessados em minimizar o congestionamento sujeito a um comprimento limitado.

	Número de Vertices	Número de Arcos
Road 1	77059	171536
Road 2	24086	50826

Tabela 4.2: *Casos de teste do tipo* ***ROAD***

CURVE: Nas instâncias de curvas de aproximação nós queremos aproximar uma função linear por partes (definida no capítulo de introdução) por uma nova curva com menos pontos de quebra. Isto é muito importante para problemas de compressão de dados em áreas como cartografia, computação gráfica e processamento de sinais.

Assumindo que os pontos de quebra na curva dada ocorrem na ordem $v_1, v_2, \cdots, v_n$, nós usamos os pontos de quebra como vértices e adicionamos arcos $v_i v_j$ para cada $i < j$. Os custos dos arcos são atribuídos como um erro de aproximação que é introduzido por tomar o atalho ao invés da curva original.

	Número de Vertices	Número de Arcos
Curva 1	10000	99945
Curva 2	10000	199790
Curva 3	1000	9945
Curva 4	1000	19790
Curva 5	5000	49945
Curva 6	5000	99790

Tabela 4.3: *Casos de teste do tipo* ***CURVE***

BC: Beasley e Christofides (1989) disponibilizaram 24 casos de teste para o problema. Os dados foram gerados de forma randômica e contêm até 500 vértices e 4800 arcos. Para mais informações a respeito da forma de gerar os dados, recomendamos a leitura do artigo original.

	Número de Vértices	Número de Arcos
1	100	955
2	100	955
3	100	959
4	100	959
5	100	990
6	100	990
7	100	999
8	100	999
9	200	2040
10	200	2040
11	200	1971
12	200	1971
13	200	2080
14	200	2080
15	200	1960
16	200	1960
17	500	4858
18	500	4858
19	500	4978
20	500	4978
21	500	4847
22	500	4847
23	500	4868
24	500	4868

Tabela 4.4: *Casos de teste do tipo Beasley e Christofides (1989),*

4.3 Resultados

Inicialmente, diversos fatores devem ser considerados em uma implementação dos algoritmos vistos neste trabalho. Implementamos apenas a programação dinâmica primal, o algoritmo de Yen para ranqueamento de caminhos e o algoritmo proposto por Hander e Zang.

Dentre todos os casos de teste que utilizamos, apenas os 24 casos de Beasley e Christofides (1989) ofereceram condições para comparações entre todos os algoritmos implementados. Isto aconteceu porque os demais casos de teste eram muito grandes, o que impossibilitou usar as implementações de programação dinâmica primal (fora os casos de teste do tipo curva) e do algoritmo de Yen. Devido a pouca memória da máquina usada para os testes, aconteciam travamentos já que essas duas abordagens consomem muita memória.

Um fato que merece destaque é a performance da programação dinâmica primal nos casos de testes do tipo de aproximação de curvas. Como era esperado, esse algoritmo foi extremamente eficiente para tais casos (veja a tabela 4.8). Isso se deve ao fato de que como os consumos de recursos neste conjunto de testes é sempre 1, o problema se torna polinomial, o que favorece a abordagem por programação dinâmica por ter uma implementação simples

e direta.

Logo abaixo vemos os resultados obtidos com a execução das nossas implementações usando as instâncias de Beasley e Christofides (1989). O algoritmo proposto por Handler e Zang nos surpreendeu bastante com uma ótima performance de tempo e com uso moderado de memória.

	PDP t(s)	PDP m(KB)	Yen t(s)	Yen m(KB)	HZ t(s)	HZ m(KB)
1	0.06	23248	0.01	6336	0.00	7088
2	0.07	23248	0.02	6336	0.00	7072
3	0.06	23152	0.01	6288	0.00	7056
4	0.07	23152	0.02	6320	0.00	6656
5	0.07	23376	0.01	6352	0.01	7104
6	0.08	23344	0.01	6352	0.01	7104
7	0.06	23168	0.01	6320	0.00	7088
8	0.07	23168	0.01	6336	0.00	7088
9	0.06	23264	0.08	6544	0.01	8496
10	0.07	23264	0.08	6544	0.00	8512
11	0.07	23312	0.01	6496	0.00	7456
12	0.07	23296	0.02	6496	0.00	7456
13	0.07	23376	0.02	7456	0.01	8304
14	0.07	23360	0.02	7456	0.01	8320
15	0.06	23312	0.02	6528	0.00	7456
16	0.07	23296	0.02	6512	0.00	7456
17	0.16	25120	0.03	9536	0.02	13296
18	0.14	24928	0.04	9536	0.02	13296
19	0.07	23728	0.04	9584	0.02	13408
20	0.07	23696	0.05	9584	0.01	11296
21	0.11	24272	0.04	9520	0.02	13280
22	0.10	24208	0.04	9520	0.02	13280
23	0.07	23680	0.04	9520	0.01	13360
24	0.07	23696	0.04	9520	0.01	13344

Tabela 4.5: *Tempo de execução para os testes* ***BC***
Usamos **PDP** para denotar programação dinâmica primal. Usamos **HZ** para denotar o algoritmo de Handler e Zang. As colunas que possuem $t(s)$ representam o consumo de tempo em segundos. As colunas que possuem $m(KB)$ contêm o consumo de memória em kilobytes.

	HZ t(s)	HZ m(KB)	Iterações
Austria grande	208.25	451440	169
Austria pequeno	7.66	119008	79

Tabela 4.6: *Resultados dos casos de teste do tipo* ***DEM***

Apresentados estes resultados, nossos próximos passos em continuidade a este trabalho será revisar todos os códigos dando uma maior atenção a detalhes de implementação que reduzem constantes na complexidade de tempo e consumo de memória. Outra coisa essencial

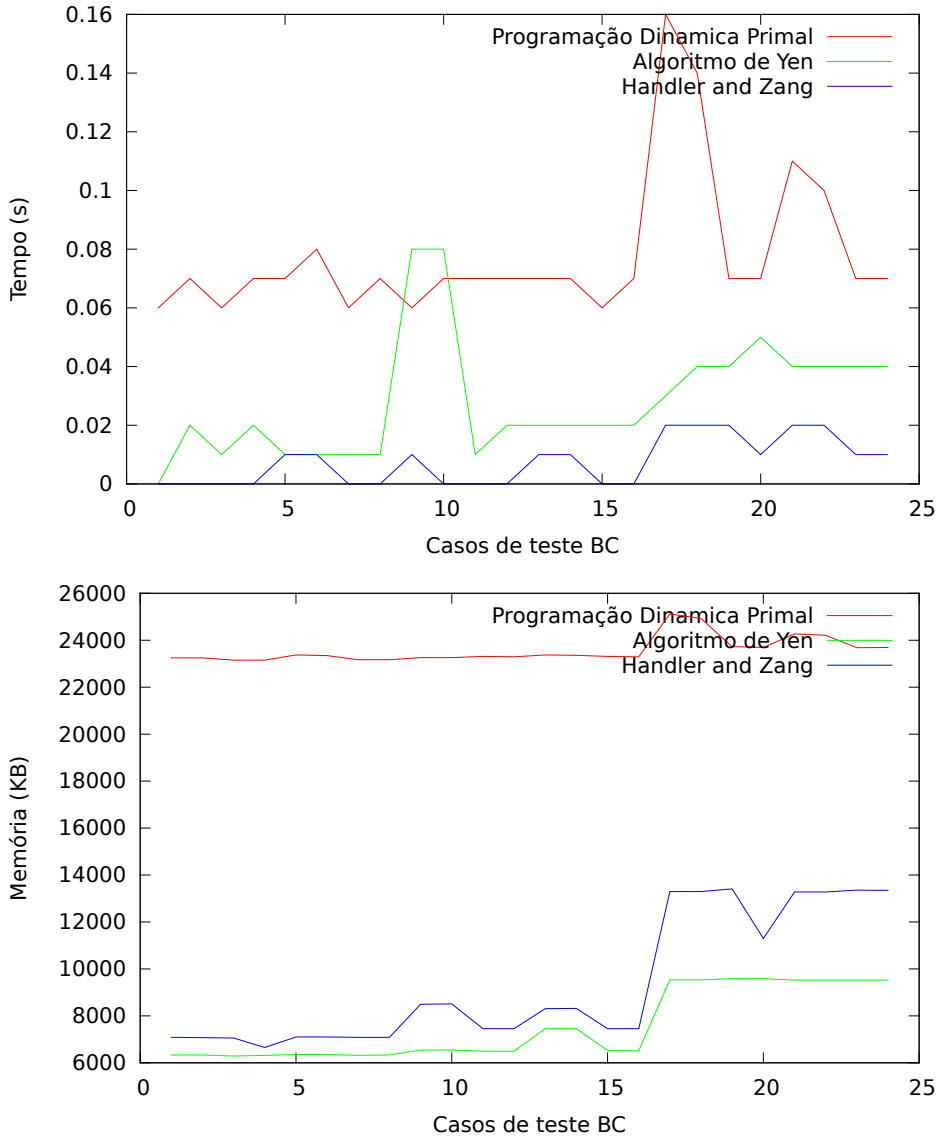


Figura 4.1: *Gráficos comparando consumo de memória e tempo dos algoritmo de programação dinâmica primal, algoritmo de Yen e algoritmo de Handler e Zang.*

	HZ t(s)	HZ m(kb)	Iterações
Road 1	853.88	1303264	1258
Road 2	12.52	189072	214

Tabela 4.7: *Resultados dos casos de teste do tipo* ***ROAD***

	HZ t(s)	HZ m(KB)	HZ Iterações	DPP t(s)	DPP m(KB)
Curva 1	1.76	163648	12	7.62	10280116
Curva 2	80.83	718560	22	12.51	1100832
Curva 3	0.08	20928	8	0.07	27264
Curva 4	0.94	49888	15	0.14	35104
Curva 5	28.56	629776	110	1.48	234832
Curva 6	1.15	154368	11	2.47	270528

Tabela 4.8: *Resultados dos casos de teste do tipo* ***CURVE***

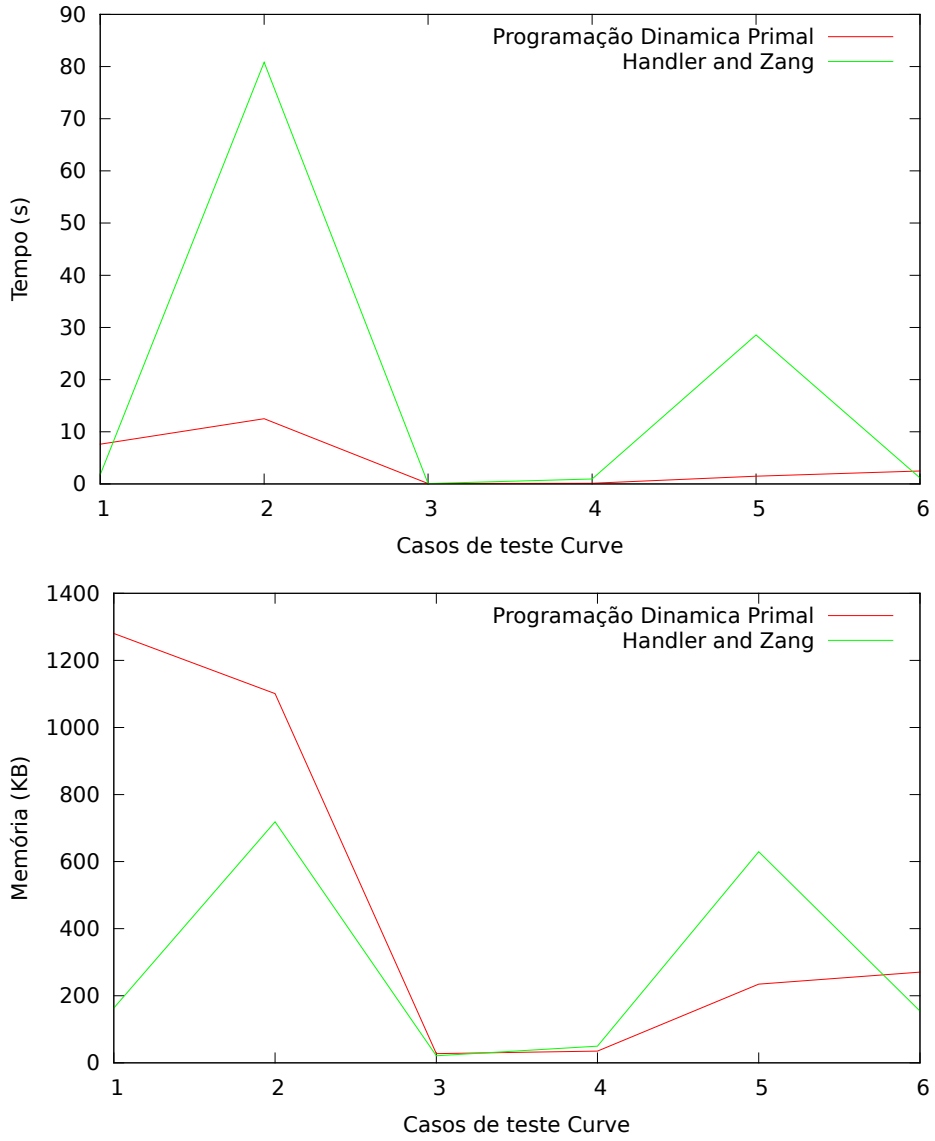


Figura 4.2: *Gráficos comparando consumo de memória e tempo dos algoritmos de programação dinâmica primal Yen e algoritmo de Handler e Zang para os casos do tipo de curva de aproximação.*

a se implementar são as reduções das instâncias sempre que possível, melhores cortes nos preprocessamentos podem trazer grandes benefícios a eficiência das implementações.

Referências Bibliográficas

Ahuja *et al.***(1990)** Ravindra K. Ahuja, Kurt Mehlhorn, J.B. Orlin, e R.E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–223. Citado na pág. 23

Ahuja *et al.***(1993)** Ravindra K. Ahuja, Thomas L. Magnanti, e James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Practice Hall. Citado na pág. 22

Aneja *et al.***(1983)** Y. P. Aneja, V. Aggarwal, e K. P. K. Nair. Shortest chain subject to side constraints. *Networks*, 13(2):295–302. ISSN 1097-0037. doi: 10.1002/net.3230130212. URL http://dx.doi.org/10.1002/net.3230130212. Citado na pág. 28, 31, 40

Aurrecoechea *et al.***(1998)** Cristina Aurrecoechea, Andrew T. Campbell, e Linda Hauw. A survey of qos architectures. *Multimedia Systems*, 6:138–151. ISSN 0942-4962. URL http://dx.doi.org/10.1007/s005300050083. 10.1007/s005300050083. Citado na pág. 2

Barrico(1998) Carlos Manuel Chorro Simões Barrico. Uma Abordagem ao Problema de Caminho Mais Curto Multiobjectivo ¿ Aplicação ao Problema de Encaminhamento em Redes Integradas de Comunicações. Dissertação de Mestrado, Departamento de Engenharia Electrotécnica Faculdade de Ciências e Tecnologia Universidade de Coimbra, Coimbra - Portugal. Citado na pág. 23

Beasley e Christofides(1989) J. E. Beasley e N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394. ISSN 1097-0037. doi: 10.1002/net.3230190402. URL http://dx.doi.org/10.1002/net.3230190402. Citado na pág. xv, 1, 28, 60, 61, 62

Bellman(1958) R. E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90. Citado na pág. 23

Borndörfer(2009) Ralf Borndörfer. CSP Lectures - Combinatorial Optimization at Work. Zuse Institute - Berlin, 2009. Citado na pág. 1

Carvalho *et al.***(2001)** Marcelo H. de Carvalho, Márcia R. Cerioli, Ricardo Dahab, Paulo Feofiloff, Cristina G. Fernandes, Carlos E. Ferreira, Katia S. Guimarães, Flávio K. Miyazawa, José C. de Pina Jr., José A. R. Soares, e Yoshiko Wakabayashi. Uma introdução sucinta a algoritmos de aproximação, 2001. Citado na pág. 8

Cherkassky *et al.***(1999)** Boris V. Cherkassky, Andrew V. Goldberg, e Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM J. Comput.*, 28:1326–1346. Citado na pág. 23

Cormen *et al.***(1999)** Thomas H. Cormen, Charles E. Leiserson, e Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill. Citado na pág. 22

Cormen *et al.***(2001)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, e Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd. edição. Citado na pág. 8, 15, 23

Dahl *et al.***(1996)** Geir Dahl, Geir Dahl, Bjørnar Realfsen, e Bjørnar Realfsen. Curve approximation and constrained shortest path problems. Em *In International Symposium on Mathematical Programming (ISMP97)*. Citado na pág. 6

Desrochers e Soumis(1988) M. Desrochers e F. Soumis. A generalized permanent labeling algorithm for the shortest path problem with time windows. *INFORMS*, 26:191–212. Citado na pág. 40

Desrosiers *et al.***(1995)** Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, e François Soumis. Chapter 2 time constrained routing and scheduling. Em C.L. Monma M.O. Ball, T.L. Magnanti e G.L. Nemhauser, editors, *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, páginas 35 – 139. Elsevier. doi: 10.1016/S0927-0507(05)80106-9. URL http://www.sciencedirect.com/science/article/pii/S0927050705801069. Citado na pág. 40

Dijkstra(1959) E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271. ISSN 0029-599X. URL http://dx.doi.org/10.1007/BF01386390. 10.1007/BF01386390. Citado na pág. 16, 28

Dumitrescu e Boland(2003) I. Dumitrescu e N. Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42 (3):135–153. ISSN 1097-0037. doi: 10.1002/net.10090. URL http://dx.doi.org/10.1002/net.10090. Citado na pág. 1, 28, 29, 31

Eppstein(2012) D. Eppstein. Bibliography files. "http://www.ics.uci.edu/ eppstein/bibs", Agosto 2012. Citado na pág. 44

Eppstein(1994) David Eppstein. Finding the k shortest paths. Em *Proc. 35th Symp. Foundations of Computer Science*, páginas 154–165. IEEE. Citado na pág. 43

Feofiloff(1997) Paulo Feofiloff. Notas de aula de MAC5781 otimização combinatória. "http://www.ime.usp.br/~pf/", 1997. Citado na pág. 11

Feofiloff(2004) Paulo Feofiloff. Fluxo em redes. "http://www.ime.usp.br/~pf/flows/", 2004. Citado na pág. 8, 16, 24

Feofiloff(2000) Paulo Feofiloff. *Algoritmos de Programação Linear*. EDUSP. Citado na pág. 13

Fisher(1978) M. L. Fisher. Lagrangian relaxation methods for combinatorial optimization. *Research Paper, Department of Decision Sciences*. Citado na pág. 44

Floyd(1962) Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–. ISSN 0001-0782. doi: 10.1145/367766.368168. URL http://doi.acm.org/10.1145/367766.368168. Citado na pág. 34

Frank e Wolfe(1956) Marguerite Frank e Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110. ISSN 1931-9193. doi: 10.1002/nav.3800030109. URL http://dx.doi.org/10.1002/nav.3800030109. Citado na pág. 5

Fredman e Tarjan(1987) Michael L. Fredman e Robert Endre Tarjan. Fibonacci heap and their uses in improved network optimization algorithms. *Journal of the ACM*, 34: 596–615. Citado na pág. 22, 23

Garey e Johnson(1979) M. Garey e D. Johnson. *Computers and Intractability: A Guide of the Theory of NP Completeness*. W. H. Freeman, San Francisco. Citado na pág. 29

Geoffrion(1974) A. M. Geoffrion. Lagrangian relaxation for integer programming. *Math Program Study*, 2:82–114. Citado na pág. 44

Handler e Zang(1980) Gabriel Y. Handler e Israel Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309. ISSN 1097-0037. doi: 10.1002/net.3230100403. URL http://dx.doi.org/10.1002/net.3230100403. Citado na pág. 28, 29, 43, 44

Hassin(1992) Refael Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):pp. 36–42. ISSN 0364765X. URL http://www.jstor.org/stable/3689891. Citado na pág. 28, 29, 35, 41, 43

Hoffman e Pavley(1959) Walter Hoffman e Richard Pavley. A method for the solution of the Nth best path problem. *J. Assoc. Comput. Mach.*, 6:506–514. Citado na pág. 43

Isotani(2002) Shigueo Isotani. Algoritmos para caminhos mínimos. Dissertação de Mestrado, Instituto de Matemática e Estatística. Citado na pág. 11, 21

Jaffe(1984) Jeffrey M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116. ISSN 1097-0037. doi: 10.1002/net.3230140109. URL http://dx.doi.org/10.1002/net.3230140109. Citado na pág. 29

Jahn *et al.***(2005)** Olaf Jahn, Rolf H. Möhring, Andreas S. Schulz, e Nicolás E. Stier-Moses. System-optimal routing of traffic flows with user constraints in networks with congestion. *Operations Research*, 53(4):pp. 600–616. ISSN 0030364X. URL http://www.jstor.org/stable/25146896. Citado na pág. 3, 4, 5

Johnson(1977) Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13. Citado na pág. 22

Joksch(1966) H. C. Joksch. The shortest route problem with constraints. *Jornal of Mathematical Analysis and Applications*, 14:191–197. Citado na pág. 28, 32, 34, 40

Lawler(1976) E. L. Lawler. *Combinatotial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York. Citado na pág. 32

LeBlanc *et al.***(1985)** Larry J. LeBlanc, Richard V. Helgason, e David E. Boyce. Improved efficiency of the frank-wolfe algorithm for convex network programs. *Transportation Science*, 19(4):445–462. doi: 10.1287/trsc.19.4.445. URL http://transci.journal.informs.org/content/19/4/445.abstract. Citado na pág. 5

Mehlhorn e Ziegelmann(2000) Kurt Mehlhorn e Mark Ziegelmann. Resource constrained shortest paths. Em Mike Paterson, editor, *Algorithms - ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, páginas 326–337. Springer Berlin / Heidelberg. ISBN 978-3-540-41004-1. Citado na pág. 28

Nygaard *et al.***(1998)** Ranveig Nygaard, John Håkon Husøy, e Dag Haugland. Compression of image contours using combinatorial optimization. Em *In Int. Conf. on Image Processing (ICIP)*, páginas 266–270. Citado na pág. 6

Phillips(1993) Cynthia A. Phillips. The network inhibition problem. Em *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, páginas 776–785, New York, NY, USA. ACM. ISBN 0-89791-591-7. doi: 10.1145/167088.167286. URL http://doi.acm.org/10.1145/167088.167286. Citado na pág. 43

Pisaruk(2009) Fábio Pisaruk. k-menores caminhos. Dissertação de Mestrado, Instituto de Matemática e Estatística. Citado na pág. 11, 16

Stroetmann(1997) Karl Stroetmann. The constrained shortest path problem: A case study in using asms. 3(4):304–319. Citado na pág. 40

Tarjan(1983) Robert Endre Tarjan. *Data structures and network algorithms*. BMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, PA. Citado na pág. 15

Witzgall e Goldman(1965) C. Witzgall e A. J. Goldman. Most profitable routing before maintenance. *Paper presented at the 27th National ORSA Meeting*, B-82. Citado na pág. 32

Wolsey(1998) L.A. Wolsey. *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons. ISBN 9780471283669. URL http://books.google.com.br/books?id=x7RvQgAACAAJ. Citado na pág. 8

Yen(1971) Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):pp. 712–716. ISSN 00251909. URL http://www.jstor.org/stable/2629312. Citado na pág. 28

Zhu(2005) X. Zhu. *The Dynamic, Resource-constrained Shortest Path Problem on an Acyclic Graph with Application in Column Generation and a Literature Review on Sequence-dependent Scheduling*. Texas A&M University. ISBN 9781109850963. URL http://books.google.com.br/books?id=UYP1tgAACAAJ. Citado na pág. 12

Ziegelmann(2001) M. Ziegelmann. *Constrained Shortest Paths and Related Problems*. Tese de Doutorado, Universität des Saarlandes. Citado na pág. 59