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 em Ciências

Programa: Mestrado Acadêmico em Computação

Orientador: Prof. Dr. Carlos Eduardo Ferreira

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

São Paulo, junho de 2011

Caminhos mínimos com recursos limitados

Esta versão definitiva da tese/dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante a defesa realizada
por Joel Silva Uchoa em XX/06/2011.

Comissão Julgadora:

- Profa. Dra. Nome Completo (orientadora) - IME-USP [sem ponto final]
- Prof. Dr. Nome Completo - IME-USP [sem ponto final]
- Prof. Dr. Nome Completo - IMPA [sem ponto final]

Resumo

Um problema bastante conhecido é o de escolher uma rota para se fazer uma viagem, tal que a rota minimize a distância do percurso. Nesta forma básica, esse problema é o problema de caminho mínimo em grafos onde as arestas são possíveis trechos, valorados por seu comprimento. Algumas vezes um caminho mínimo desta forma é bom, outras vezes não. Existem ocasiões, onde tal caminho possui propriedades indesejáveis. Por exemplo, alguns trechos podem ter tráfego denso e nos fazer perder muito tempo na travessia, ou existem muitos pedágios com taxas que, acumuladas pelo caminho, vão exceder o dinheiro que temos disponível. Isso nos leva a considerar um ou mais parâmetros adicionais para a escolha do caminho. Os casos mais comuns de parâmetros a considerar envolvem o consumo de recursos em um orçamento que limita a quantidade disponível desses recursos. Um caminho mínimo com essas limitações adicionais é chamado de **caminho mínimo com recursos limitados** (*resource constrained shortest path* - RCSP).

Este trabalho possui dois objetivos principais: apresentar um histórico bibliográfico do problema RCSP, tendo como foco algoritmos exatos para o caso onde possuímos um único recurso; e implementar e comparar os principais algoritmos conhecidos, observando-os em situações práticas.

Palavras-chave: caminhos mínimos com recursos limitados.

Abstract

Abstract ...

Keywords: keyword1, keyword2, keyword3.

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*)

Lista de Figuras

Lista de Tabelas

Capítulo 1

IntroduÃ§Ã£o

O problema de caminhos mÃnimos (SP – *shortest path problem*) Ã© um dos problemas fundamentais da computaÃ§Ã£o. Vem sendo estudado com profundidade e hÃ¡ uma grande quantidade de publicaÃ§Ãµesarespeitodomesmo.$Inclusive, s£oconhecidosvriassolu§µesesficientes(algor$
$ficosemumarede.Tala§£odeveserexecutadadeforma^3timaemrela§£oaalgumcrit©rio, porexemploomen$
$vel, ouomenorgastodetempoouomximodeconfiabilidade/seguran§a.$

Conforme essas soluÃ§Ãµesparao SP $foramsendoapresentadas, novasnecessidadesforamlevantadase$
$nimosmulti-objetivo.Comon£o©possvelotimizarsobretodososcrit©riosdeumas^3vez, n^3sescolhemosum$
$nimoscomrestri§µesporrecursos, oucomopreferimoschamar,$ **problemadecaminhosmnimoscomrecur**
$-resource\ constrained\ shortest\ path\ problem$[2]$), oqualseroobjetodeestudonestetrabalho.$

A adiÃ§Ã£o de restriÃ§Ãµesporrecursosno SP, $infelizmentetornaoproblema$ NP$-difcil, mesmoemgrafosacclicos, comrestri§µessobreum^onicorecurso, ecomtodososconsumosderecursosposit$
$ceis$ Mochila *e* Partição *paraonossoproblema.*

Em contextos diversos sÃ£o encontrados problemas de cunho teÃ³$ricoeprticoquepodemserformulados$
$nimoscomrecursoslimitados, oquenosmotivouaestud-loafimdedesenvolverumtrabalhoqueresumisseinf$
$mosum^onicorecursoeaimplementa§£oecompara§£odosprincipaisalgoritmosconhecidos, observando-$
$osemsitua§µesprticas.$

1.1 AplicaÃ§Ãμes

O problema de caminhos mÃnimos com recursos limitados pode ser aplicado em uma imensa quantidade de problemas prÃ¡ticos. Esta seÃ§Ã£o vai descrever algumas destas aplicaÃ§Ãµes.

[1] RestriÃ§Ãµesdestetipo, ondetemosoconsumoderecursosemumor§amentoquelimitaaquantidadedisponveldestesrecursoss£ochamadasdeknapsack constraints(?).

[2] **?** foi um dos primeiros a chamar o problema desta forma, antes disso era comum utilizar apenas CSP (constrained shortest path problem). A versÃ£o com um Ãºnico recurso pode ser referenciada como SRCSP(*single* RCSP), ou ainda como WCSP (*weight constrained shortest path problem*) (**?**).

1.1.1 Qualidade de serviÃ§o em redes de computadores

A qualidade de serviÃ§o (QoS – *quality of service*) Ã© 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 aplicaÃ§Ãµes $fim-a-fim, como vdeo eudio conferência e transferência$

Existem redes de computadores que estÃ£o oferecendo garantias de QoS para diversas aplicaÃ§Ãµes. Estas aplicaÃ§Ãµes possuem diversos requisitos que precisam ser atendidos para o seu bom funcionamento, tais como banda, tempo máximo de atraso e quantidade máxima de perda de pacotes. O problema em que estamos in

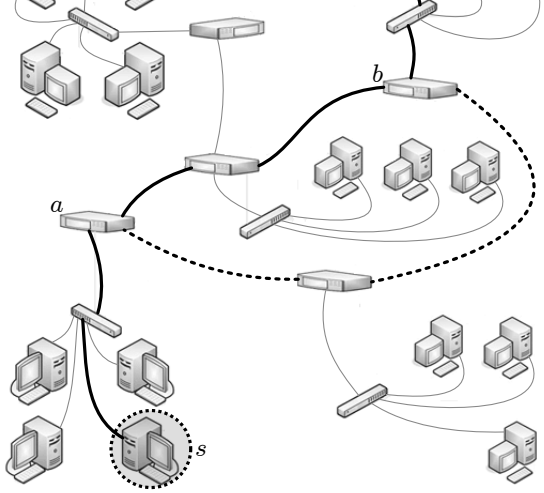


Figura 1.1: *RepresentaÃ§Ã£o de uma rede de computadores. Os seguimentos pretos e contÃnuos compÃµem uma rota entre os computadores s e t. Substituindo – se o trecho entre os roteadores a e b pelo trecho pontilhado, temos uma outra rota que pode ser usada para a comunicaÃ§Ã£*

Formalizando um pouco melhor o problema, podemos representar nossa rede como um grafo direcionado $G = (V, A)$, onde V Ã© o conjunto de vÃ©rtices e A Ã© o conjunto de arcos. Cada arco $uv \in A$ Ã© associado com um custo c_{uv} e M pesos nÃ£o negativos w^k_{uv}, $k = 1, 2, \cdots, M$ que representam a valoraÃ§Ã£o dos requisitos no arco (ambos, pesos e custos sÃ£o aditivos em um caminho). Dada uma aplicaÃ§Ã£o que exige M requisitos com valoraÃ§Ã£o mÃ¡xima R^k , $k = 1, 2, \cdots, M$, o problema Ã© encontrar um caminho P da origem s atÃ© o destino t que respeita os requisitos e que minimiza o custo total do caminho. Problema este que pode ser modelado da seguinte forma:

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

Este problema Ã© uma aplicaÃ§Ã£o direta do problema de caminhos mÃnimos com recursos limitados. Ã comum, neste tipo de aplicaÃ§Ã£o, querermos uma rota passando pelo menor nÃºmero de vÃ©rtices possÃvel, neste caso a funÃ§Ã£o de custo possui valor

unitÃ¡rio para todos os arcos. Ã comum tambÃ©m existirem restriÃ§Ãμes quen£os£oaditivas pelo caminh
las geralmente com algum pre processamento ou pequenas altera§μes nos algoritmos.

1.1.2 Roteamento de trÃ¡fego de veÃculos

Quando precisamos nos deslocar de um ponto a outro em uma rede de trÃ¡fego veicular Ã© natural nos preocuparmos com uma sÃ©rie de fatores a respeito da rota escolhida para o trajeto. Geralmente desejamos percorrer o caminho no menor tempo possÃvel dentro de determinadas restriÃ§Ãμes, *como o consumo de combustvel, gastos com pedgio e distnciamxima percorrida. Outra sticas£o mais subjetivas, geralmente baseadas em dados estatsticos*).

Dentro deste contexto, ? propÃμe uma aplica§£o interessante que usa o RCSP *como subproblema. Nesta a velque para atingir a efici*ª*ncia global, algun suurios precisem realizar caminhos muito piores do que realizar sta. Assim, em um sistema de apoio decis£o, poderia acontecer dos usurios n£o seguirem as recomenda§μes ap vel. Desta forma, ©possvel obter uma solu§£o aceitvel para cada usurio, que objetiva aumentar a efici*ª*ncia gl*

Hoje em dia existem sistemas de informaÃ§Ã£o e orientaÃ§Ã£o projetados para auxiliar motoristas a tomarem decisÃμes de rota. *Vamos idealizar um sistema que pode fornecer informa§μes como o culo (dimensμes, quantidade de combustvel disponvel e consumo, por exemplo) e nas condi§μes atuais do trnsito sticas dos veculos, as posi§μes atuais e seus destinos, podendo assim fazer certas distribui§μes de rotas aos moto*

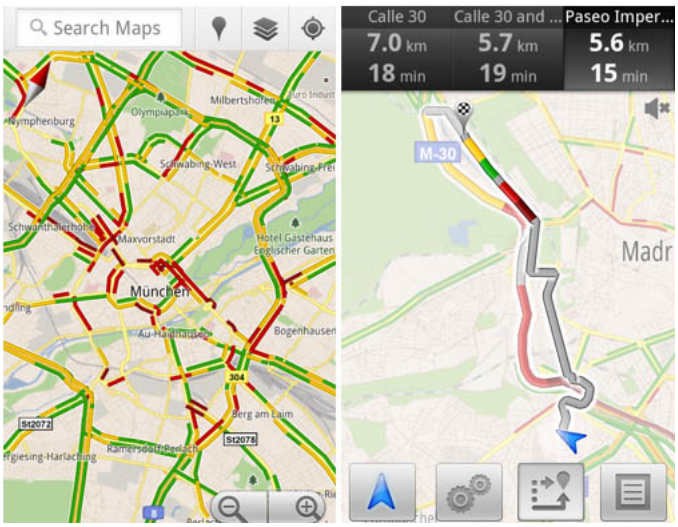


Figura 1.2: *Exemplo de um sistema de orientaÃ§Ã£o para motoristas. No lado esquerdo podemos ver para uma determinada regiÃ£o o trÃ¡fego em cada trecho (vermelho, amarelo e verde representam respectivamente trÃ¡fego pesado, mÃ©dio e leve). No lado direito temos a representaÃ§Ã£o de uma rota, comeÃ§ando no triÃngulo azul (posiÃ§Ã£o atual do veÃculo) e terminando na marcaÃ§Ã£o com um circulo quadriculado xadrez.(Figura retirada de Google Mobile Blog – http://googlemobile.blogspot.com.br/2011/07/live-traffic-information-for-13.html)*

Descrevemos todo um conjunto de restriÃ§Ãμes complexo e detalhista, por©m, na nossa formula§£o leve veis de congestionamento e um limitante superior para a degrada§£o de um caminho em rela§£o a melhor cami

[3] SoluÃ§Ãμes onde se atribui o caminho mais rpido para cada usurio sob as condi§μes correntes, s£o chamadas de solu§£o ou seo tempo total do sistema, s£o chamadas de system optimal.

Representamos nossa rede rodoviÃ¡ria 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 nÃ£o hÃ¡ congestionamento; e uma funÃ§Ã£o $l_{uv}(x_{uv})$ (x Ã© um fluxo na rede, x_{uv} Ã© a parte deste fluxo correspondente ao arco uv) que computa uma estimativa do tempo de travessia do arco uv considerando o fluxo dado[4].

NÃ³s modelamos os veículos que possuem a mesma origem e destino como um par $k = (s, t)$, definimos 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 atravÃ©s de k (veÃculos por unidade de tempo). Denotamos todos os caminhos para o par k por $\mathcal{P}_k = \{P \mid P$ Ã© um caminho de s_k atÃ© $t_k\}$, 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, Ã© $l_P(x) = \sum_{uv \in P} l_{uv}(x_{uv})$, o tempo estimado de percurso sem considerar congestionamento Ã© $\tau_P = \sum_{uv \in P} \tau_{uv}$.

Definimos um fator de tolerÂncia $\varphi \geq 1$. AtravÃ©s deste fator, assumimos que para um caminho $P \in \mathcal{P}_k$ ser viÃ¡vel, $\tau_P \leq \varphi T_k$, onde $T_k = \min_{P \in \mathcal{P}_k} \tau_P$ Ã© o menor tempo possÃvel 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 viÃ¡veis 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 viÃ¡veis para os pares em K.

O sistema Ã³timo com restriÃ§Ãµes (CSO – *constrained system optimum*) proposto, pode ser modelado como o problema de custo mínimo de fluxo multi-commodity (*min-cost multi-commodity flow*) :

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

? usa um algoritmo de geraÃ§Ã£o de colunas para resolver o problema CSO (para uma descriÃ§Ã£o do mÃ©todo ver **?** e **?**). Neste algoritmo, surge como sub-problema computar um caminho Ã³timo em P_k^{φ} que Ã© 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 Ã© computar um caminho mais rÃ¡pido de s a t cujo tamanho nÃ£o excede a um dado limite T. Ou seja, o problema Ã©

$$\min\{l_P \mid P \text{ Ã© um caminho de } s \text{ atÃ© } t \text{ e } \tau_p \leq T\},$$

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

[4] TODO – InformaÃ§Ãµes sobre a função.

1.1.3 CompressÃ£o de imagens (aproximaÃ§Ã£o de curvas)

Uma curva/funÃ§Ã£o Ã© linear por partes (*piecewise linear curve*) se pudermos subdividi-la em intervalos que sÃ£o lineares. Este tipo de curva/funÃ§Ã£o Ã© frequentemente usado para aproximar funÃ§Ã£*µescomplexasouobjetosgeomÃ©tricos*.

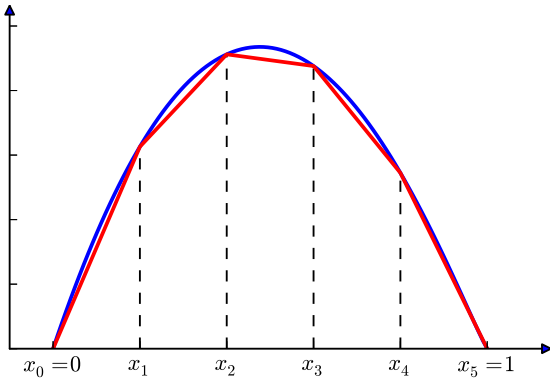


Figura 1.3: *Exemplo de uma funÃ§Ã£o linear por partes. A funÃ§Ã£o em azul Ã© uma funÃ§Ã£o nÃ£o linear, e a funÃ§Ã£o em vermelho Ã© uma aproximaÃ§Ã£o da primeira que atende a nossa definiÃ§Ã£o de curva linear por partes.*

O uso dessas curvas Ã© muito comum em Ã¡reas como computaÃ§Ã£o grÃ¡fica, programaÃ§Ã£o matemÃ¡tica, processamento de imagens e cartografia. Curvas lineares por parte sÃ£o populares porque sÃ£o fÃ¡cil de se criar e manipular, alÃ©m de fornecer, em geral, uma aproximaÃ§Ã£o suficientemente boa para os problemas estudados.

Aplicaã§Ã*µesnasreascitadasnopargrafoanterior*, *frequentementeincluemumaenormequantidadedec veldaoriginal*, *mastemumnºmeromenordepontosdequebra*.

?? estudaram este problema e mostraram como ele poderia ser modelado como um problema de caminhos mÃnimos com recursos limitados: Os pontos de quebra $v_1, v_2, \cdots, v_{n-1}, v_n$ sÃ£o os vÃ©rtices do grafo $G = (V, A)$ e para todo para $1 \leq i \leq j \leq n$ nÃ³*stemosumarcov*$_i v_j$. O custo de um arco c_{uv} Ã© o erro introduzido na aproximaÃ§Ã£o por tomar o "atalho" indo direto de u para v ao invÃ©s da curva original[5]. O recurso de uma aresta r_{uv} Ã© 1 para $uv \in A$. Agora, nÃ³*spodemoscomputaramelhoraproximaÃ§Ã£ousandonomximokpontosdequebra*[6].

1.2 Objetivos

Os principais objetivos deste trabalho, de forma sucinta sÃ£o:

- levantar um conjunto de referÃªncias bibliogrÃ¡ficas relevantes, cobrindo o mÃ¡ximo possÃvel de variaÃ§Ã*µeseaplicaÃ§µesdoproblema*; *apresentaro*RCSP *esuasdiversasabordagenscomu*
- implementar um subconjunto dos principais algoritmos conhecidos;
- avaliar o desempenho prÃ¡tico dos algoritmos implementados.

[5] Existem diversos tipos de mÃ©tricas que podem ser usadas para calcular este erro.

[6] Alternativamente, nÃ³*spodemoslimitaroerrodeaproximaÃ§Ãoecomputaromenornºmerodepontosdequebra*.

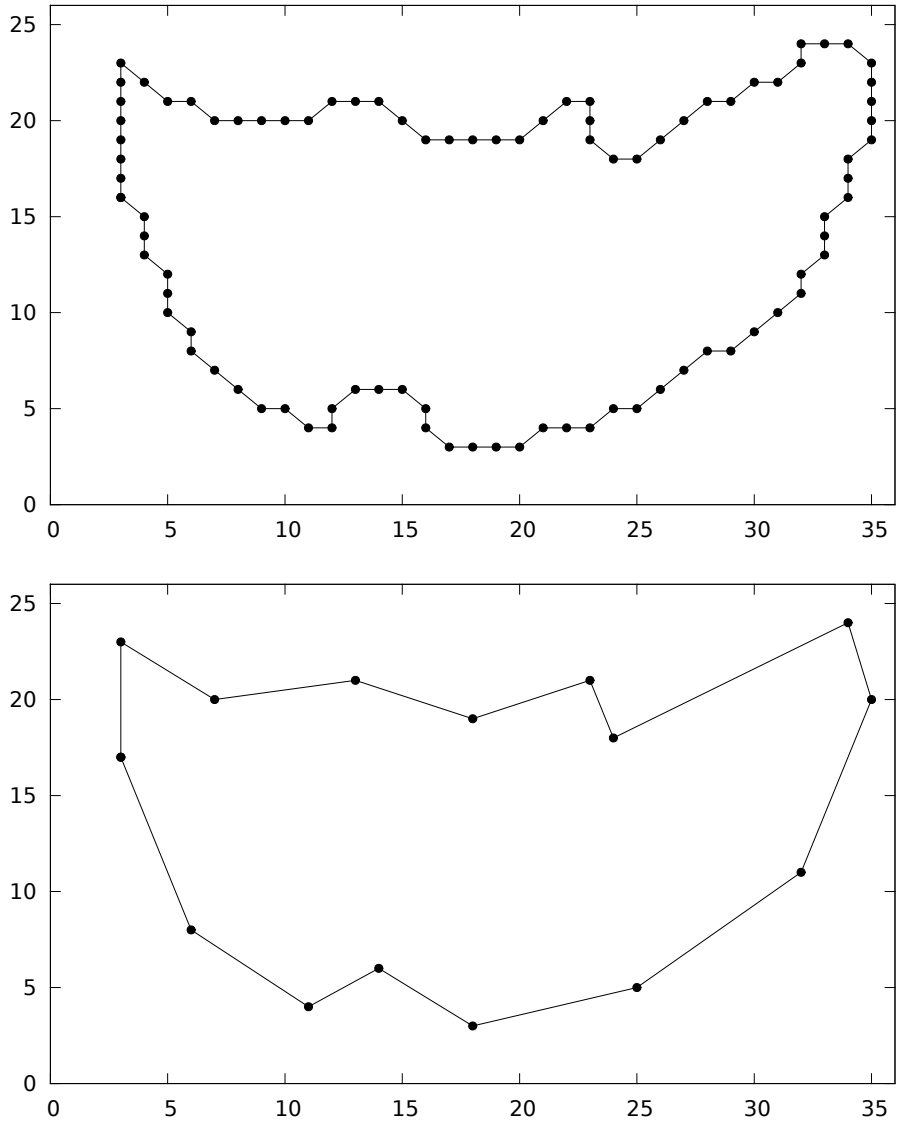


Figura 1.4: *Exemplo de uma curva e sua aproximaÃ§Ã£o. A curva original possui* 90 *pontos, enquanto a sua aproximaÃ§Ã£o possui* 15 *pontos.*

1.3 Preliminares

Para o perfeito entendimento do conteÃºdo desde trabalho devemos salientar a necessidade de conhecimento prÃ©vio em alguns assuntos que enumeraremos a seguir. A maioria dos estudantes de computaÃ§Ã£o deve estar familiarizado com os conceitos, mas faremos indicaÃ§Ã£$esdepublica§esquedefinemeusamnota§espr^3ximasdaqueestamosutilizando.$

Um conhecimento prÃ©vio dos seguintes assuntos Ã© recomendado:

- Teoria dos Grafos
- Algoritmos em Grafos
- Fluxo em Redes
- ProgramaÃ§Ã£o Linear

- ProgramaÃ§Ã£o Inteira
- RelaxaÃ§Ã£o Lagrangiana
- Algoritmos de AproximaÃ§Ã£o
- Complexidade Computacional

No que se trata da parte de teoria dos grafos, fluxo em redes e algoritmos em grafos, seguimos de perto a nomenclatura e conceitos definidos em **?**. Um livro muito completo em relaÃ§Ã£o aos conceitos de programaÃ§Ã£o linear e relaxaÃ§Ã£o que fazemos questÃ£o de indicar Ã© o **?**. **?** fala sobre algoritmos de aproximaÃ§Ã£o, e Ã© uma Ã³*timarefer*ê*nciasobreoassunto.Porfimtemosolivro*?*quepodeserusadoparaoestudodecomplexidadecomp*

1.4 OrganizaÃ§Ã£o

O trabalho Ã© organizado da seguinte forma.

O CapÃtulo 1 – IntroduÃ§Ã£o, apresenta uma visÃ£o geral do problema e da dissertaÃ§Ã£o. Apresentamos textualmente uma definiÃ§Ã£o do RCSP, alem de citar informaÃ§Ã*µessobrecomp*

O CapÃtulo 2 – Caminhos mínimos (sem restrições), trÃ¡s uma descriÃ§Ã£o do problema de caminhos mÃnimos clÃ¡ssicos, problema este que deu origem ao RCSP. Alem da descriÃ§Ã£o apresentamos algoritmos eficientes para o problema, e tambÃ©m definimos alguns conceitos importantes, usados no decorrer da dissertaÃ§Ã£o.

No CapÃtulo 3 – Caminhos mínimos com recursos limitados, definimos formalmente o problema de caminhos mÃnimos com recursos limitados, que Ã© o foco deste trabalho. Apresentamos um breve histÃ³*ricolistandoasprincipaissolu*§*µesconhecidasparaoproblema.Expomostamb*
cil.Porfimdescrevemosalgunsalgoritmosrelevantesquedespertaramnossointeresse.

O CapÃtulo 4 – Experimentos, expÃ*µeestatsticasepercep*§*µesarespeitodosexperimentosrealizadoscom*

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 por 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 **?**, **?** e **?**.

2.1 Definições básicas

Podemos definir uma **função custo** c em um grafo $G = (V, A)$ 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 inicio 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 $C = 7$.

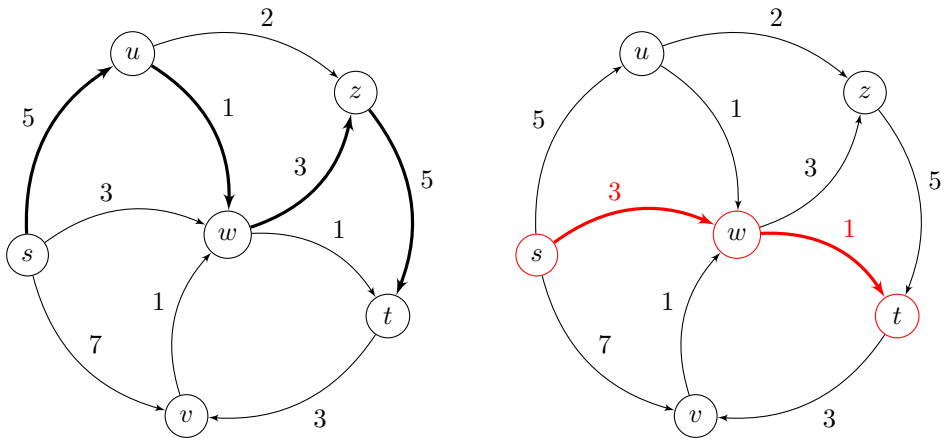


Figura 2.1: *Exemplo de um grafo com uma função custo sobre os arcos. No lado esquerdo temos um caminho* $\langle s, u, w, z, t\rangle$ *com custo igual a 14. À direita temos o caminho* $\langle s, w, t\rangle$ *em vermelho, que é um caminho de custo mínimo de* s *à* t*.*

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* (**?**).

2.3 Funções potenciais

Vamos definir o seguinte programa linear, que chamamos de primal, e é uma relaxação do problema do caminho mínimo: encontrar um vetor x indexado por A que

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

O vetor característico de qualquer caminho de s a t é uma solução viável do problema primal. 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

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

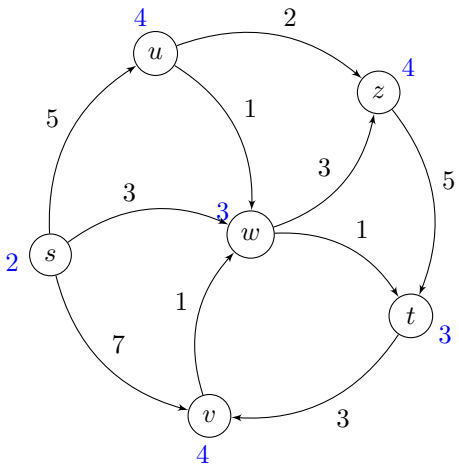


Figura 2.2: *Grafo com uma função custo c sobre os arcos e um c-potencial associado aos vértices em azul.*

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 (**?**). 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 por s.

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

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

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

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

∎

Do lema 2.1 tem-se imediatamente os seguintes corolários.

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

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

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

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

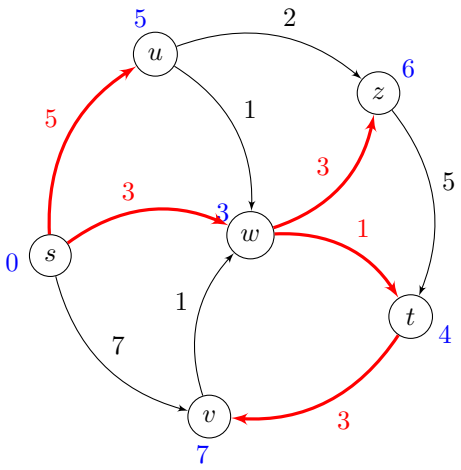


Figura 2.3: *Grafo com custos nos arcos e um potencial nos vértices. O potencial exibido garante que qualquer caminho formado por vértices vermelhos a partir de s é um caminho de custo mínimo.*

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 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

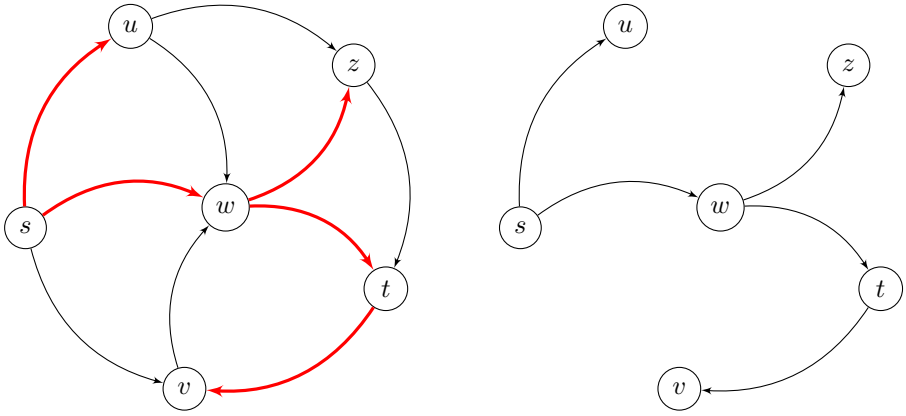


Figura 2.4: *Exemplo de uma função predecessor ψ e de uma grafo de predecessores induzido por ela. Acima temos os valores de ψ para cada vértice. O grafo da esquerda mostra as arestas induzidas por ψ em vermelho. A grafo da direita é o grafos de predecessores a partir de ψ.*

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. O custo dos caminhos que tem como origem o vértice s são limitados inferiormente por esta função.

Examinar um arco ou **relaxar/rotular um arco** (*relaxing* **?**, *labeling step* **?**) 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       EXAMINE-ARCO (uv)
```

Abaixo temos uma versão expandida.

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 nó 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 (**?**) que resolve o problema do caminho mínimo em grafos onde a função custo possui apenas custos não negativos, ou seja, $c(uv) \geq 0$ para todo $uv \in A$. Nosso texto segue de perto **?** e **?**.

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 conhece caminhos partindo de s a cada vértice em S, caminhos estes que são garantidamente de custo mínimo, e conhece caminhos a uma parte dos vértices em Q. 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.

O algoritmo recebe um grafo $G = (V, A)$, 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 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 $C := \max\{c(uv) : uv \in A\}$ [1].

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 func. como uma fila com prioridades
6  enquanto Q ≠ ⟨⟩ faça
7      retire de Q um vértice u com y(u) mínimo
8      EXAMINE-VÉRTICE (u)
9  devolva ψ e y
```

Abaixo temos uma versão expandida.

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 ∞
```

[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).

```
3       ψ(v) ← NIL
4   y(s) ← 0
5   Q ← V     ▷ Q func. como uma fila com prioridades
6   enquanto Q ≠ ⟨⟩ faça
7       retire de Q um vértice u com y(u) mínimo
8       para cada uv em A 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
```

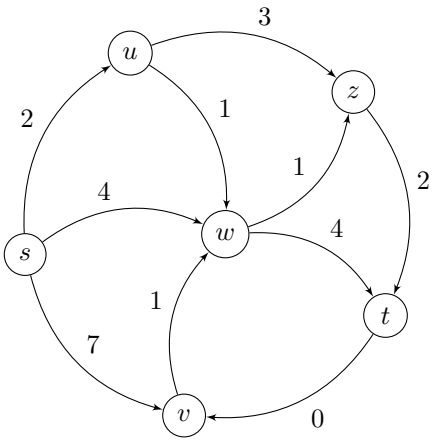
Simulação

A seguir, temos uma simulação para uma chamada do algoritmo DIJKSTRA. Na figura (1) temos o grafo (V, A) com custo sobre os arcos. Nas figuras os vértices em Q têm interior azul claro. A função-potencial y é indicada pelos números em azul próximos cada vértice. Na figura (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 $n \times C + 1 = 6 \times 7 + 1 = 43$).

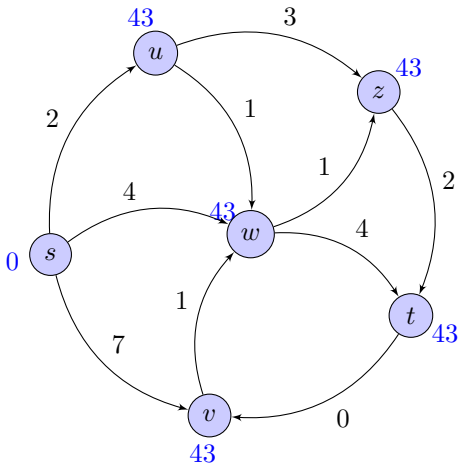
Um vértice sendo examinado tem a cor rosa. O arco sendo examinado tem a cor azul (na figura (3) o vértice sendo examinado é s e o arco sendo examinado é sv).

Os vértices em S (já examinados) têm a cor verde. Na figura (10) vemos que w está sendo examinado, s e u são os vértices em S e os demais estão em Q.

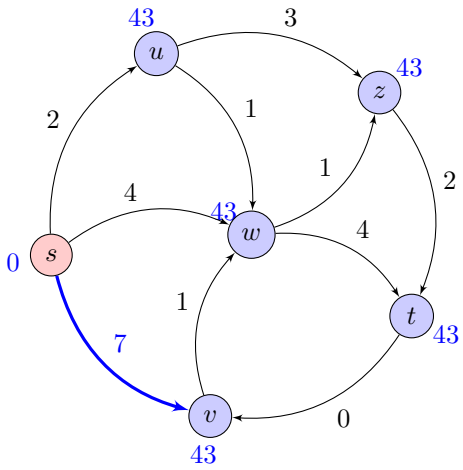
Os arcos já examinados têm a cor vermelha se fazem parte do grafo de predecessores, caso contrário são tracejados.



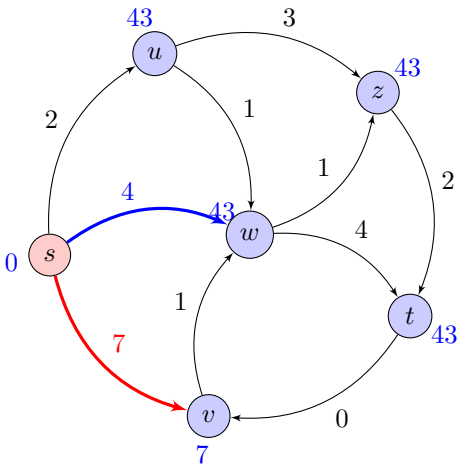
(1)



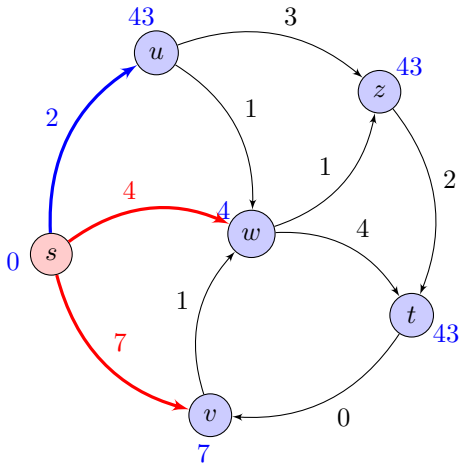
(2)


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

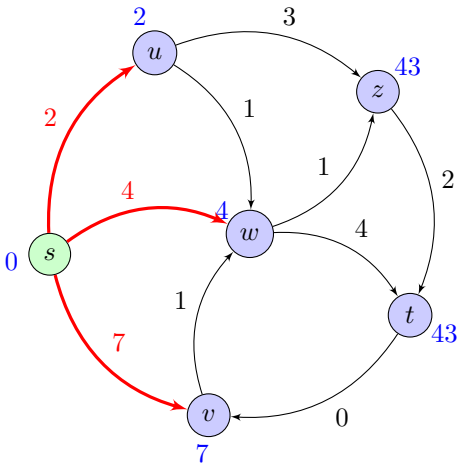
(3)


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

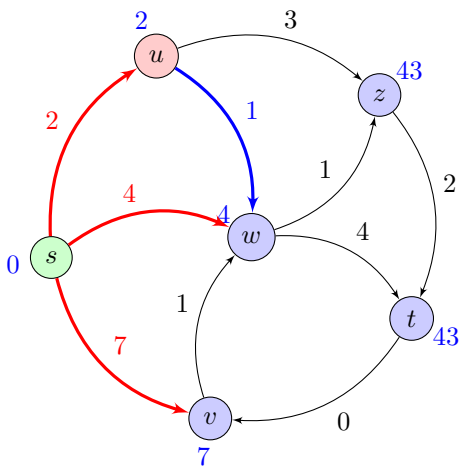
(4)


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

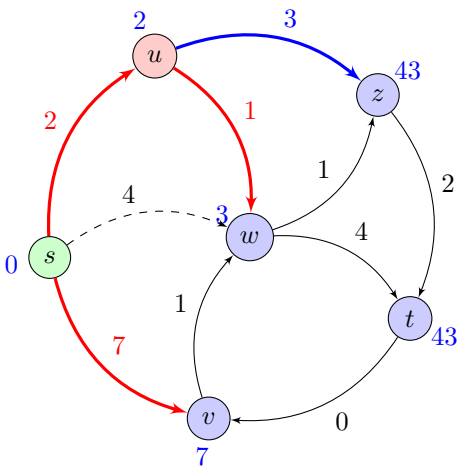
(5)


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

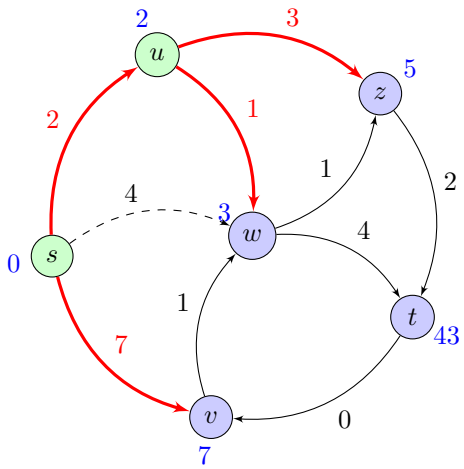
(6)


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

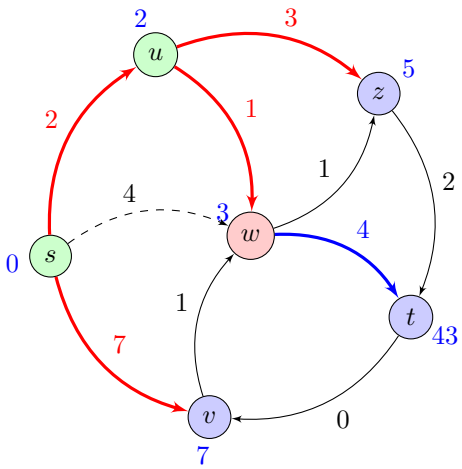
(7)


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

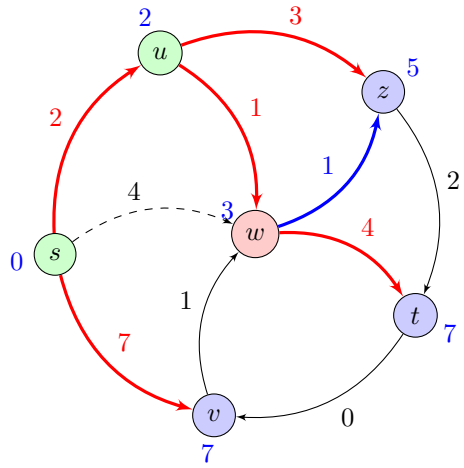
(8)



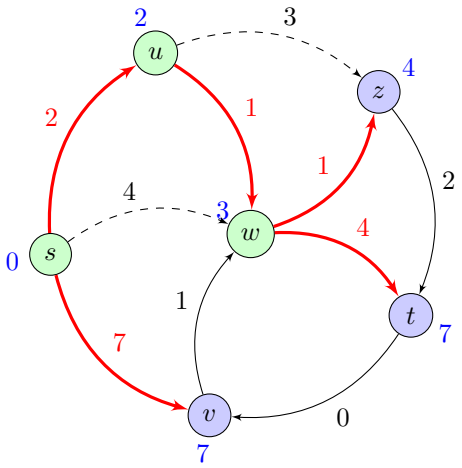
(9)



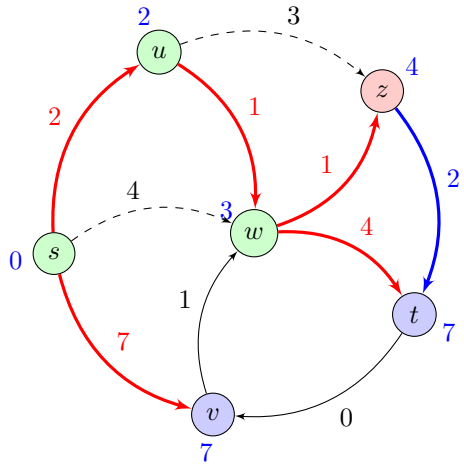
(10)



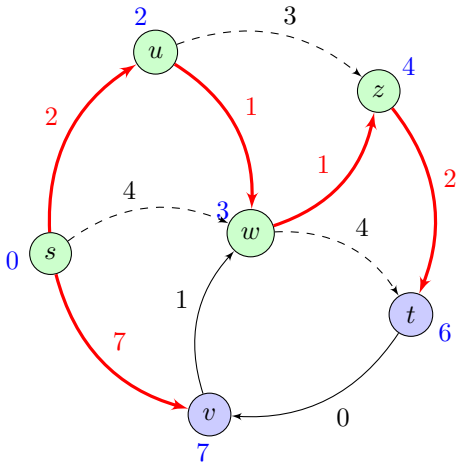
(11)



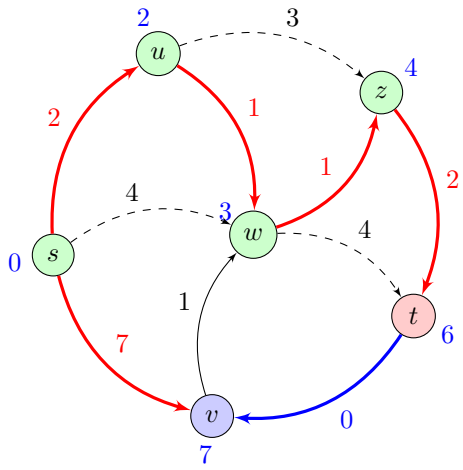
(12)



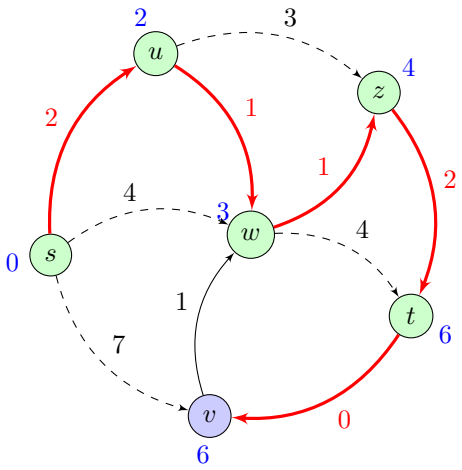
(13)



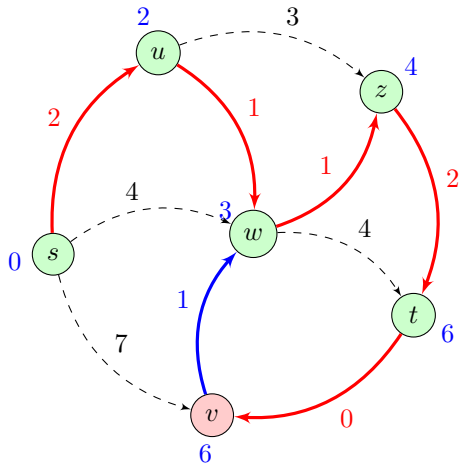
(14)



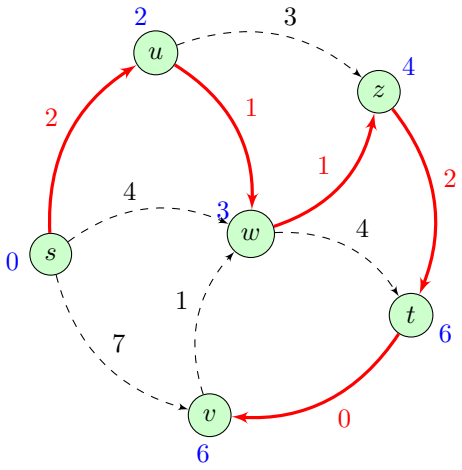
(15)



(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 **?**. Aqui faremos apenas uma argumentação básica para mostrar que o algoritmo é correto.

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

Demonstração: Como Q é vazio no 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 gastar tempo $O(n)$ e é executada até n vezes (até Q ficar vazio), ou seja, consome tempo $O(n^2)$;
2. e atualizar o potencial, que pode acontecer para todas as arestas, ou seja, gasta 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 (**?**), o heap de Fibonacci (**?**), que permitem diminuir o tempo gasto 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 ??** 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.

	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.*

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

```
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
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 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 **???**.

A tabela a seguir resume o consumo de tempo de várias implementações de filas de prioridade e o respectivo consumo de tempo resultante para o algoritmo de Dijkstra **?**.

Fredman e Tarjan **?** 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, no pior caso, $\Omega(n \log n)$ 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 (**?**), pode ser aplicado a grafos cujos arcos têm associado custos negativos, não podendo contudo existir circuitos de custo negativo (**?**).

Este algoritmo consiste em examinar os vértices do grafo, corrigindo os potencias as vezes que for necessário, até que todos os potencias satisfaçam a condição de otimalidade. Neste algoritmo, os potenciais dos vértices têm o mesmo significado dos potencias 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
```

O consumo de tempo desta versão é $O(n^2mC)$ (**?**). O consumo de tempo é tão elevado porque o algoritmo pode examinar cada arco muitas veze (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 visa 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 $\quad\quad\quad\quad\quad\quad\quad\quad \psi(v) \leftarrow u$

10 $\quad\quad$ ***devolva*** ψ *e* y

O algoritmo consome $O(nm)$ unidades de tempo. Como $m = O(n^2)$ no pior caso, pode-se dizer que o algoritmo é $O(n^3)$.

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 dirigido* $G = (V, A)$,
- *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 estar "embutido" em um grande número de problemas práticos, ele tem sido extensivamente estudado como é mostrado na tabela 3.1 que mostra uma lista com algumas características dos principais algoritmos disponíveis.

Basicamente, duas famílias de algoritmos tem 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 a 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, **?** 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 **?** (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 é exponencial, o número de caminhos computados seria bastante reduzido. **?** 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. **?** 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 **?**.

Referencia	Versão do RCSP	Método	Custo	Grafo
?	único recurso	RL + Yen	$c_{uv} \geq 0$	gerais
?	múltiplos recursos	RL + BB	irrestrito	gerais
?	múltiplos recursos	PL	$c_{uv} \geq 0$	gerais
?	único recurso	PD	$c_{uv} > 0$	gerais
?	múltiplos recursos	LS	$c_{uv} \geq 0$	gerais
?	único recurso	PD	$c_{uv} > 0$	acíclicos
?	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. **?** propôs uma programação dinâmica, bem como **?**, que apresentou dois algoritmos exatos para uso com grafos acíclicos. **?** adaptou o esquema de rotulamento permanente (*label-setting*) de **?**. A abordagem de rotulamento permanente é uma generalização do algoritmo Dijkstra, que rotula e processa os vértices em ordem, baseado nos consumos de recursos. Alem 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. **?** investigaram variações do algoritmo de rotulamento corretivo, eles apresentaram uma versão melhorada do algoritmo de **?**, alem de apresentar um algoritmo baseado em um método de escalar pesos.

3.3 Complexidade

?? 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 (**?**). **?** apresentaram uma redução do problema da partição para o RCSP, enquanto **?** reduziu o problema da mochila para o nosso problema.

? mostrou que o SRCSP tem solução polinomial se os custos dos arcos e consumos de recursos são limitados. Temos por **?** que a solução do RCSP é garantidamente elementar (não passa por um mesmo vértice duas vezes) se os custos dos arcos são não-negativos e temos apenas restrições que limitam a quantidade máxima de recursos consumidos ou o grafo é acíclico. A seguir, mostraremos uma redução do **problema da mochila** (*knapsack*), definido a seguir, ao problema RCSP(baseada em **?**).

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.

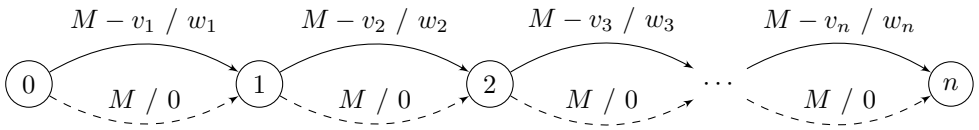


Figura 3.1: *Os arcos contínuos representam os arcos no conjunto A_1 e os arcos tracejados representam os arcos no conjunto A_2. O rótulo de cada arco uv representa o seu custo e o seu consumo de recurso, respectivamente (c_{uv} / r_{uv}).*

Como $s = 0$, $t = n$; 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 R_{uv} como sendo 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. **?** foi o primeiro a apresentar reduções baseadas nos recurso como descrito acima. Podemos executar esta redução com duas computações do algoritmo Dijkstra (para calcular R_{sv} e R_{vt} para qualquer $v \in A$) seguido 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 a solução ótima do problema, nós podemos extender 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.

? propôs 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 funciona bem quando o instância possui restrições bem "apertadas".

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 menores da instância original. O que distingue a programação dinâmica de uma recursão comum é o uso da tabela que "memoriza" as soluções das subinstâncias evitando assim o recalculo 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 problema muitas vezes.

Temos alguns algoritmos baseados em programação dinâmica conhecidos para o RCSP, todos eles tem complexidade pseudo-polinomial, pois suas complexidades de tempo dependem do tamanho dos custos e consumo de recursos dos arcos. Neste capítulo iremos falar um pouco sobre três destas soluções. A primeira itera sobre os possíveis valores de consumo, minimizando o custo; aqui iremos referenciá-la como **programação dinâmica primal**. A segunda, muito similar a primeira, itera sobre os possíveis valores de custo, verificando se o consumo de recursos atende ao limite imposto; 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**. Embora todos os algoritmos possam ser usados na versão do problema com múltiplos recursos, por questão de praticidade 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 **?** (ver também **??**). 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:

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

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].

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

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
13             custo ← valor
14 devolva custo
```

É importante salientar que se no grafo, existir 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 baseados 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 existisse arcos com consumo nulo no grafo. Podemos adaptar o algoritmo de **?** para realizar tal processamento em $O(n^3)$.

O valor do caminho ótimo OPT pode ser encontrado pela chamada RECURSÃO-PRIMAL(t, λ) do algoritmo que corresponde ao valor $f_t(\lambda)$ da recorrência, e o caminho ótimo propriamente dito pode ser construído usando a função predecessor ψ montada no decorrer da recursão [3].

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 $O(n\lambda)$. **?** apresentou melhoras práticas para este algoritmo, contudo a complexidade de pior caso é não melhor que a obtida com a ideia básica.

[3] Definimos função predecessor na seção 2.4 do capítulo 2. Lá, ela era indexada pelos vértices, aqui, temos um versão estendida que é indexada por um par vértice e consumo de recursos, porém o uso é igual em ambos os casos.

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                      ψ(v, r) ← u
16                      PD[v, r] ← PD[v, r − r_uv] + c_uv
17  devolva PD[t, λ]
```

A programação dinâmica iterativa é ainda mais sensível a arcos com consumo nulo de recurso que a recursão, neste caso, não precisamos ter ciclos com consumo nulo, 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 arestas sem consumo de recursos deve ser executado antes da chamada de PROGRAMAÇÃO-DINÂMICA-PRIMAL.

3.5.2 Programação dinâmica dual

Na sessão anterior vimos um procedimento simples baseado em programação dinâmica que objetiva minimizar os custos iterando sobre os recursos. **?** 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 recursão pode ser definida.

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

Observe que OPT não é um valor conhecido no inicio 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 custo > 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          ψ(v, c) ← ψ(v, c − 1)
14          PD[v, c] ← 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 a solução. Para se verificar a viabilidade da instância, pode-se rodar um 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 um candidato a solução.

Todas a recomendações e observações feitas a PROGRAMAÇÃO-DINÂMICA-PRIMAL são aplicáveis a 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 o 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 pseudo-polinomial baseado em programação dinâmica pode ser adaptado para fornecer um FPTAS para o RCSP com o uso da técnica de escalar e arredondar. Discutiremos isso com mais detalhes seção 3.6.

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. 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 [4].

Temos várias versões de programação dinâmica por rótulos conhecidas, entre elas estão as descritas por **????**. 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. 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

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

recursos (no caso com um único recurso). Na figura 3.2 podemos ver esta ordem exemplificada. **?** 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.

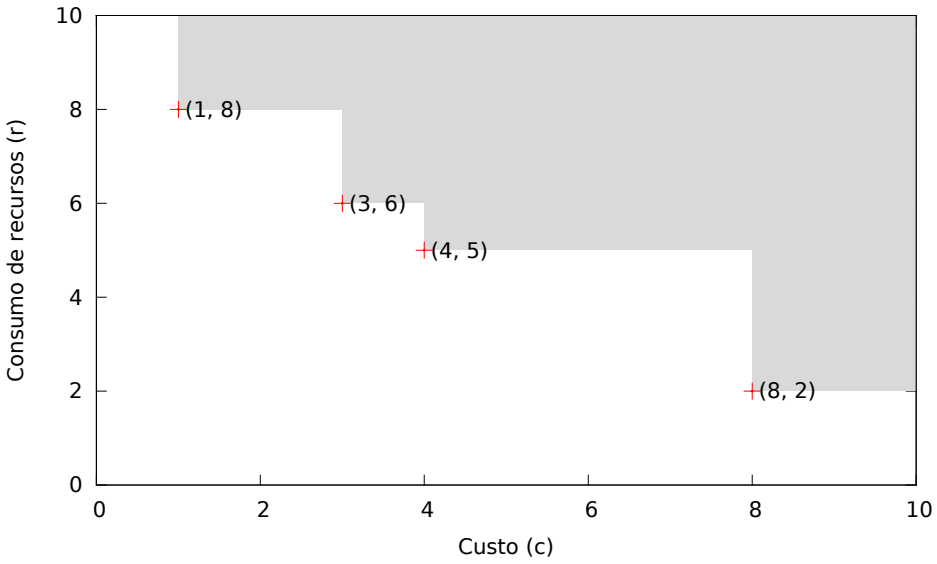


Figura 3.2: *Nesta figura temos o exemplo de uma lista de rótulos, exibidos em um plano para exemplificar a função escada. A área em cinza representa a área que é dominada pelos rótulos da lista.*

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   PD(s) ← PD(s) ∪ {(0,0)}
4   enquanto existe 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 $\psi(v,(c',r')) \leftarrow u$

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, o pior caso permanece o mesmo, temos uma complexidade de tempo de $O(m\lambda)$.

3.6 ϵ-Aproximação

? aplicou a técnica de escalar e arredondar para obter um esquema de aproximação totalmente polinomial (FPTAS – *fully polynomial ϵ-approximation scheme*) para o SRCSP. Vamos rever, resumidamente, este método agora.

Na sessão 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 $g_t(c') \leq R$. Assim, temos o conhecimento do valor $OPT = c'$.

Agora, digamos que V seja um certo valor, e suponha que queremos testar se $OPT \geq V$ ou não. Um procedimento polinomial que responde essa questão pode ser estendido em um algoritmo polinomial parar encontrar OPT simplesmente usando uma 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 revolver uma saída negativa, então nós sabemos que $OPT < (V + \epsilon)$.

O teste arrendonda 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 anterior ao grafo com os custo 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 $c = 1, 2, \cdots$ até que $g_n(c) \leq R$ 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 cara fixado o ϵ é explicada em seguida: Tomar a parte inteira de um número não negativo no intervalo $\{0, \cdots, U\}$ pode ser feito em tempo $O(\lg(U))$ usando busca binária. Arrendondar 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 $O(|E| \lg(n/\epsilon))$. 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 arrendondar: Para aproximar OPT nós primeiramente determinamos um limite superior (UB) e um limite inferior (LB). O limite superior UB pode ser setado como a soma das $n-1$ arcos com maior custo, ou o custo da caminho que consome menos recursos. O limite inferior LB pode ser setado como 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 procedimento 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 atinga o valor de uma constante predefinida, digamos 2, então uma ϵ-aproximação pode ser obtida aplicando-se o algoritmo por programação dinâmica para o grafo com os custo 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 $O(|E|n/\epsilon)$. 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 $x = (LB \cdot UB)^{1/2}$. O número de testes necessários para reduzir a razão para abaixo de 2 é $O(\lg\lg(UB/LB))$ e cada teste toma tempo $O(|E|n/\epsilon)$. **?** 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 $O(\lg\lg(UB/LB)(|E|(n\epsilon) + \lg\lg(UB/LB)))$ para este algoritmo ϵ-aproximado como um FPTAS para o *CSP*.

? também mostrou uma FPTAS cuja a complexidade depende somente do número de variáveis e $1/\epsilon$ e possui complexidade de tempo de $O(|E|(n^2/\epsilon)\lg(n/\epsilon))$. O melhor FPTAS foi obtido por **?** que atingiu a complexidade de tempo de $O(|E|(n/\epsilon) + (n^2/\epsilon)\lg(n/\epsilon))$.

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 **?**. Desde então, o problema

tem sido massivamente estudado e várias soluções foram propostas.

Muitos desses métodos tem complexidade de tempo polinomial para um k fixo. **?** 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.

O KSP não é recomendado para ser usado diretamente para resolver o RCSP. Porem, ele tem sido usado com sucesso como sub-problema para métodos mais sofisticados como a relaxação lagrangiana proposta por **?**. Uma extensa bibliografia para o problema dos k menores caminhos pode ser encontrada em **?**.

3.8 Relaxação Lagrangiana

A seguir vamos apresentar o algoritmo proposto por **?**, que se utiliza de uma relaxação de um problema de programação linear que modela o RCSPcom um único recurso.

Inicialmente, vamos apresentar uma formulação para o problema RCSP usando programação linear. Nela teremos uma variável x_{uv} para cada $uv \in A$, $x_{uv} = 1$ 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 definir $\mathcal{X}$ denotando o conjunto de vetores x que satisfazem as equações (1) e (3), ou seja, vetores x

que contêm um caminho de s a t. Vamos definir também a seguinte função.

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

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 **?**, **?**) como primeiro passo para resolver o RCSP. Tendo em vista que o problema é relativamente mais simples de resolver quando a restrição $g(x) \leq 0$ é relaxada (sem essa restrição, o problema se reduz a caminho mínimo simples), 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 a solução de $L(u)$ é o problema de caminho mínimo no grafo original, porém com os custos dos arcos alterados para $c_{uv} + ur_{uv}$, $uv \in A$. Temos que $L(u) \leq c^*$ para qualquer $u \geq 0$ (teorema fraco da dualidade), 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 que existir essa folga, teremos que trabalhar um pouco mais para eliminá-la.

Vamos, agora, descrever um método para resolver o programa (P), que usa como passo, resolver o problema (D). Por praticidade vamos denotar $x(u)$ 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 a solução.

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 caminho que consume a menor quantidade de recursos consome uma quantidade maior do que o limite.

- Senão, $x(\infty)$ é uma solução viável para a instância e nos serve de limite superior para 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 as 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 intercessão 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).

Falamos, 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 intercessão, 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 a 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 a 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 fraco da dualidade); 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 um folga na dualidade. Para eliminarmos essa folga poderíamos considerar 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 a 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⁺ ≠ NIL e x⁻ ≠ NIL    ▷ Se entrou nos dois "então" acima
10        LB ← 0; UB ← c⁻
11        enquanto LB < UB faça
12            u′ ← (c⁻ − c⁺)/(g⁺ − g⁻); L′ ← c⁺ + u′g⁺; x′, c′, g′ ← L(u′)
13            se g′ = 0
14                então x*, c* ← x′, c′; LB ← UB ← c′
15                      PÁRA o enquanto
16            se L(u′) = L′ e g′ < 0
17                então LB ← L′; UB ← min{UB, c′}; x⁻ ← x′; u* ← u′
18                      PÁRA o enquanto
19            se L(u′) = L′ e g′ > 0
20                então LB ← L′; u* ← u′
21                      PÁRA o enquanto
22            se L(u′) < L′ e g′ > 0
23                então x⁺, c⁺, g⁺ ← x′, c′, g′
24            se L(u′) < L′ e g′ < 0
25                então x⁻, c⁻, g⁻ ← x′, c′, g′; UB ← min{UB, c′}
    ▷ Eliminando a folga da dualidade
26        x1, x2 ← x⁺, x⁻; k ← 2
27        enquanto LB < UB faça
28            k ← k + 1; x_k, c_k, g_k ← L_k(u*); LB ← L_k(u*)
29            se g_k ≤ 0 e c_k < UB
30                então x⁻, UB ← x_k, c_k
31            se LB ≥ UB
32                então x*, c* ← x⁻, UB
33    devolva x*, c*
```

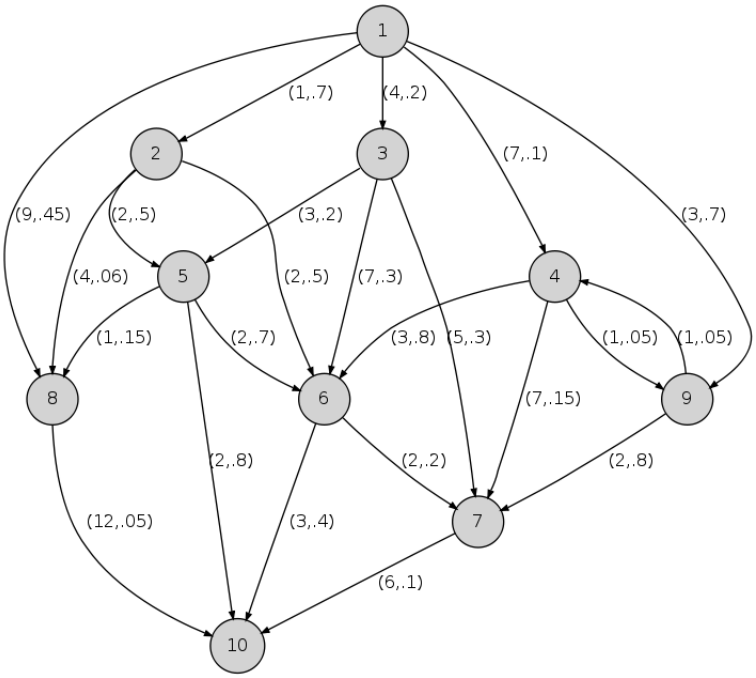


Figura 3.3: *Grafo exemplo; os rótulos dos arcos representam* (c_{uv}, r_{uv}).

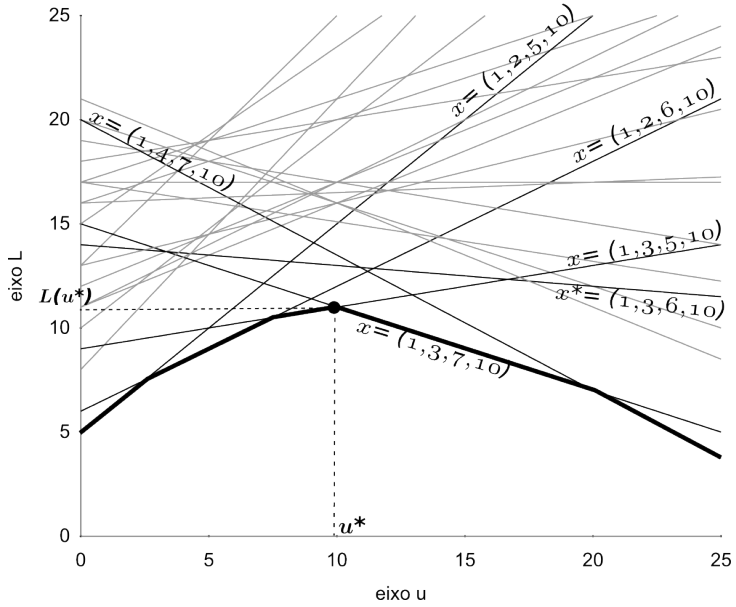


Figura 3.4: *Representação geométrica do grafo da Figura 3.3. As retas pretas representam os caminhos que são relevantes ao algoritmo. A "curva" de segmentos mais espessos representa o função* $L(u)$.

Capítulo 4

Experimentos

No presente capítulo, iremos fazer experimentos com alguns dos diferentes métodos propostos para o problema de caminhos mínimos com recursos limitados. Por uma questão de praticidade, 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 as nossas experiências foram executadas 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 (**?**).

Em nossos exemplos DEM, temos que arcos são bi-direcionados, ou seja, 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.

ROAD: Temos exemplo de grafos de ruas dos Estados Unidos fornecidos por Mark Ziegelmann. Os arcos modelando ruas são novamente bi-direcionados, e a estrutura nos dá

	Número de Vertices	Número de Arcos
Austria grande	41600	165584
Austria pequeno	11648	46160
Escócia grande	63360	252432
Escócia pequeno	16384	65024

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

m aproximadamente $2.5n$. Nós usamos um indice que avalia o congestionamento como função de custo. Definimos os congestionamentos como inteiros entre $[0, 100]$. Nossa função custo é a distancia 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: Nos problema 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 $i < j$. O custo dos arcos é atribuído 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: **?** 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 dos 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* **?***,*

4.3 Resultados

O que primeiro se percebe, é que os algoritmo não são nada triviais de serem implementados. Existe uma grande quantidade de fatores relevantes a implementação. 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 **?** ofereceram boas comparações entre os algoritmos. Isto aconteceu porque os demais casos de teste eram muito grandes, e o fato de as nossas implementações não serem tão performáticas, nos impossibilitou de usar tais casos de teste de uma melhor forma. Devido a pouca memória da máquina usada para os testes, aconteciam travamentos por exemplo.

Logo abaixo vemos os resultados obtidos com a execução das nossas implementações usando as entradas de **?**. O algoritmo proposto por Handler e Zang nos surpreendeu bastante com uma ótima performance de tempo e com uso moderado de memória.

Dado estes resultados, nossos próximos passos em continuidade a este trabalho será revisar todos os códigos dando uma maior atenção a detalhes de implementação que diminuem constantes e consumo de memória. Outra coisa essencial a se implementar não as reduções

	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 $m(KB)$ contem o consumo de mémoria em kilobytes.

das instâncias sempre que possível, não podemos esquecer o problema de caminhos mínimos com recursos limitados é um problema não polinomial, e qualquer corte que seja, pode trazer grandes benefícios a eficiência das implementações.

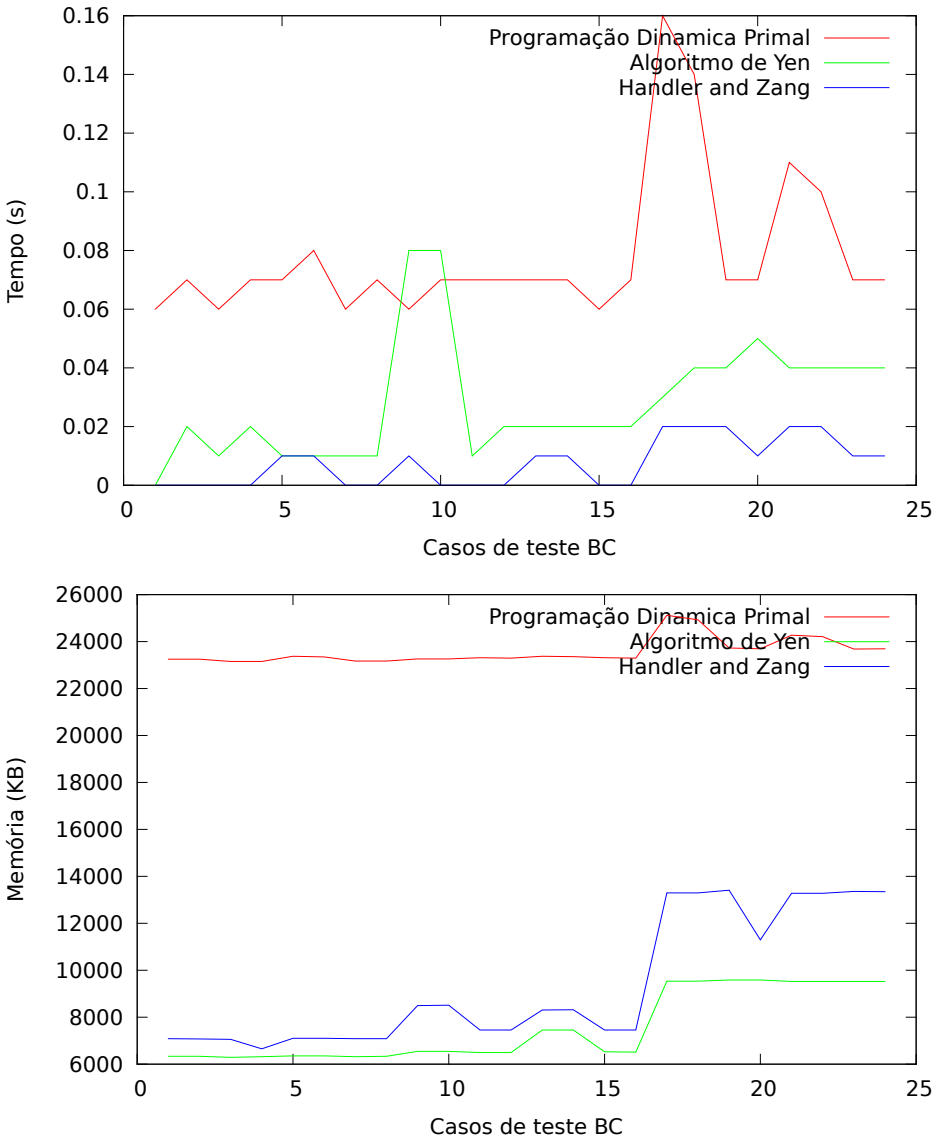


Figura 4.1: *Gráficos comparando consumo de memória e tempo dos algoritmo de programação dinâmica primal, algoritmo de Yen e algoritmo de Handler e Zang.*