k-árvores de custo mínimo

Marcio Takashi Iura Oshiro

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

Programa: Ciência da Computação
Orientador: Prof. Dr. José Coelho de Pina Jr.

— São Paulo, 12 de julho de 2010 —

– Durante a realização desse trabalho, o aluno recebeu apoio financeiro da CAPES. –

k-árvores de custo mínimo

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Marcio Takashi Iura Oshiro e aprovada pela comissão julgadora.

São Paulo, 28 de maio de 2012

Banca examinadora:

Prof. Dr. José Coelho de Pina Junior (Presidente)	IME-USP
Profa. Dra. Yoshiko Wakabayashi	IME-USP
Prof. Dr. Fabio Henrique Viduani Martinez	UFMS

Agradecimentos

Agradeço

à minha família por todo apoio que sempre me deram;

ao Coelho por tem me orientado e por toda dedicação e paciência, tanto no mestrado quanto na iniciação científica;

aos membros da banca examinadora pela revisão e sugestões;

ao grupo de Combinatória e Otimização Combinatória do IME-USP pelo apoio que me deram;

aos amigos do BCC (Albano, Atoji, Cris, Lobato, Mariana, Pavão, Pedro, ...), da "salinha" (Antônio, Cardonha, Domingos, Ellen, Goca, Hashimoto, Luna, Marcel, Ricardo, ...) e do NUMEC (Álvaro, Chegado, Schouery, ...) pelo apoio e amizade ao longo desses anos;

aos "pesquisadores" Hugo e Jeferson, que sempre participaram comigo da maratona de programação, e ao *coach* Wanderley pelo aprendizado e diversão.

Resumo

Esta dissertação trata do problema da k-árvore de custo mínimo (kMST): dados um grafo conexo G, um custo não-negativo c_e para cada aresta e e um número inteiro positivo k, encontrar uma árvore com k vértices que tenha custo mínimo.

O kMST é um problema NP-difícil e portanto não se conhece um algoritmo polinomial para resolvê-lo. Nesta dissertação discutimos alguns casos em que é possível resolver o problema em tempo polinomial. Também são estudados algoritmos de aproximação para o kMST. Entre os algoritmos de aproximação estudados, apresentamos a 2-aproximação desenvolvida por Naveen Garg, que atualmente é o algoritmo com melhor fator de aproximação.

Abstract

This dissertation studies the minimum cost k-tree problem (kMST): given a connected graph G, a nonnegative cost function c_e for each edge e and a positive integer k, find a minimum cost tree with k vertices.

The kMST is an NP-hard problem, which implies that it is not known a polynomial algorithm to solve it. In this dissertation we discuss some cases that can be solved in polynomial time. We also study approximation algorithms for the kMST. Among the approximation algorithms we present the 2-approximation developed by Naveen Garg, which is currently the algorithm with the best approximation factor.

Índice

Agradecimentos v
Resumo vii
Abstract vii

1 Introdução **1**

2 Preliminares **5**
2.1 Grafos, florestas e árvores 5
2.2 Algoritmos de aproximação 7

3 O problema kMST **9**
3.1 Descrição do problema 9
3.2 kMST com raiz 11
3.3 kMST generalizado 12
3.4 Complexidade computacional 13

4 Algoritmos exatos **15**
4.1 Algoritmo enumerativo 15
4.2 kMST para k específicos 16
4.3 Árvore geradora mínima 17
4.4 kMST em grafos completos 19
4.5 k-árvores de árvores 21

5 Algoritmos combinatórios 27

5.1 k-Kruskal . . . 27

5.2 k-Dijkstra . . . 31

5.3 $2\sqrt{k}$-aproximação . . . 33

5.4 $O((\log k)^2)$-aproximação . . . 36

6 Algoritmos primal-duais 43

6.1 Programação linear . . . 43

6.2 Árvore de Steiner com coleta de prêmios . . . 48

6.3 5-aproximação de Garg . . . 52

7 2-aproximação de Garg 61

7.1 Idéia básica do algoritmo . . . 62

7.2 Coleções laminares . . . 63

7.3 A subrotina GW-Modificado . . . 65

7.4 Custos reduzidos e potenciais . . . 69

7.5 Convexidade das penalidades . . . 71

7.6 Penalidade limiar . . . 75

7.7 k-árvores de custos limitados . . . 81

7.8 Descrição do algoritmo . . . 88

8 Considerações finais 93

A Simulação de GW-Modificado 95

A.1 Execução de GW-Modificado . . . 95

A.2 Custos reduzidos e potenciais . . . 102

B Contra-exemplo 109

Referências Bibliográficas 111

Índice Remissivo 115

CAPÍTULO 1

Introdução

Problemas de otimização combinatória são problemas cujo objetivo é minimizar ou maximizar uma determinada função definida sobre um conjunto discreto. Em muitos casos, tal conjunto é finito. Logo, um método para encontrar o elemento que minimiza ou maximiza a função desejada é avaliar a função para cada elemento do conjunto e devolver aquele com o valor ótimo . Valor ótimo é o máximo no caso em que queremos maximizar, ou o mínimo no caso em que queremos minimizar. Esse método, chamado de enumerativo ou força-bruta, sempre funciona para conjuntos finitos. No entanto, mesmo sendo finitos, esses conjuntos podem ser muito grandes, tornando o método impraticável. Isso porque, mesmo utilizando um computador muito rápido, poderia levar milhões de anos para se obter uma solução do problema. Por isso o estudo e desenvolvimento de algoritmos eficientes para tais problemas é muito importante.

Estudos teóricos mostram a existência de problemas muito difíceis denominados intratáveis, ao ponto de não conhecermos algoritmos eficientes para resolvê-los. Na verdade, muitos acreditam que não existem algoritmos eficientes para resolver problemas intratáveis. Uma abordagem muito utilizada atualmente é a de sacrificar a otimalidade desejada em troca de eficiência. Ou seja, em vez de procurarmos por uma solução com valor ótimo, procuramos por uma com valor possívelmente bem próximo do ótimo. Para muitos algoritmos utilizados atualmente não sabemos o quão próximo ou longe do ótimo está a solução devolvida por ele. Se conseguimos provar matematicamente que existe um limite para o quão longe do ótimo está a solução devolvida por um algoritmo, então o chamamos de algoritmo de aproximação.

Nesta dissertação, estudaremos um problema de otimização combinatória sobre uma estrutura conhecida como árvore e alguns algoritmos de aproximação para tratá-lo. Árvores formam uma classe muito importante de grafos, tanto do ponto de vista teórico quanto prático. O problema a ser estudado é conhecido como kMST. Ele consiste, basicamente, em encontrar uma árvore num dado grafo, contendo pelo menos k vértices e de forma que a soma dos custos de suas arestas seja mínima. Mostraremos que esse problema é computacionalmente difícil e apresentaremos vários algoritmos de aproximação para tratá-lo.

O problema kMST desperta grande interesse teórico por ser computacionalmente difícil, como mostraremos na seção 3.4. Além do interesse teórico, também existem interesses práticos. Por se tratar de um problema em árvores não é difícil encontrar aplicações na área de telecomunicações, por exemplo. Mencionaremos algumas aplicações conhecidas para o kMST e suas variantes.

Algoritmos para roteamento *quorumcast*

Em redes de comunicação, um problema muito importante é o de roteamento de mensagens, ou seja, determinar caminhos pelos quais uma mensagem deve percorrer até chegar ao seu destino. Basicamente, o que queremos é encontrar um grafo que represente os melhores caminhos pelos quais uma mensagem enviada por um determinado nó da rede deve passar para chegar aos demais nós da rede.

O grafo desejado deve ser conexo, pois queremos que cada nó da rede possa se comunicar com todos os demais. Como não existe vantagem em uma mesma cópia da mensagem passar mais de uma vez por um mesmo nó da rede, o grafo que queremos é uma árvore. Chamamos essa árvore de árvore de roteamento.

Uma comunicação *multicast* consiste em enviar uma mensagem para um determinado conjunto de m nós em uma rede com $n \geq m$ nós. Comunicações *multicast* têm se tornado muito popularer em aplicações distribuídas.

Uma generalização de comunicação *multicast* é a comunicação *quorumcast*. Uma comunicação *quorumcast* consiste em enviar um mensagem para quaisquer q nós de um determinado conjunto de $m \geq q$ nós em uma rede com $n \geq m$ nós. Esse tipo de comunicação é também de interesse em aplicações distribuídas.

Duas abordagens naturais para o roteamento *quorumcast* são: mandar mensagem para cada nó individualmente, um nó por vez, até que q nós respondam; e utilizar *multicast* para enviar a mensagem para todos os nós. A primeira abordagem pode causar um atraso excessivo por ter que esperar as respostas dos outros nós. A segunda abordagem pode causar congestionamento na rede.

Shun Yan Cheung e Akhil Kuma [CK94] propõem uma abordagem que utiliza o kMST, com $k = q$, como subproblema. A idéia é resolver o kMST para encontrar, dentre os m nós, os q melhores para se comunicar.

Planejamento em mineração a céu aberto

Um planejamento de extração de minérios deve definir os seguintes três itens:

- limite economicamente viável do poço a ser escavado, isto é, a profundidade a partir da qual deixa de valer a pena continuar a escavação;
- taxa de processamento do minério;
- seqüência de extração do minério.

O limite economicamente viável do poço é um dos fatores determinantes na decisão de escavar um poço ou não. Muitos modelos para determinar esse limite particionam a região a ser minerada em blocos. Esses blocos possuem uma relação de precedência entre si, indicando quais blocos devem ser escavados antes de escavar um determinado bloco.

Esses modelos possuem uma formulação natural na linguagem de grafos orientados. Os vértices correspondem aos blocos e as arestas correspondem às relações de precedência entre os blocos. O objetivo é encontrar um conjunto de vértices que maximiza uma certa função objetivo.

Andrew B. Philpott e Nicholas Charles Wormald [PW97] modelam o problema de encontrar o limite economicamente viável de um poço como um problema de grafos orientados. Um grafo de mineração (G, r, w) é um grafo orientado G com raiz r e uma função de peso w de V_G em $\mathbb{Z}_{\geq}$. O peso de um subgrafo H de G é a soma dos pesos de cada vértice em H. A raiz r é artificial, não representando um bloco, e precede todos os blocos.

O problema considerado é o de encontrar uma árvore em G com k vértices e maior peso possível. Como queremos maximizar uma função nos vértices e não nas arestas, esta não é uma aplicação direta do problema kMST. No entanto, na seção 3.3, definiremos uma generalização do kMST que engloba este problema.

Outras aplicações

Além das citadas anteriormente, existem outras aplicações estudadas que envolvem o problema kMST ou uma variante dele. Algumas dessas aplicações que podemos citar envolvem problemas com campos petrolíferos [HJ93], *facility layout* [FHW98], decomposição de matriz [BFM98].

Organização do texto

O texto está organizado da seguinte forma. No capítulo 2 apresentamos algumas definições básicas de teoria dos grafos e algoritmos de aproximação. Também fixamos algumas notações. No capítulo 3 definimos formalmente o problema e algumas variantes e provamos que ele é NP-difícil. No capítulo 4 discutimos alguns casos particulares nos quais é possível resolver o problema em tempo polinomial. No capítulo 5 apresentamos alguns algoritmos de aproximação para o kMST. No capítulo 6 introduzimos alguns conceitos básicos de programação linear e o problema da árvore de Steiner com coleta de prêmios para em seguida explicar um algoritmo com fator de aproximação 5. No capítulo 7 apresentamos um algoritmo com fator de aproximação 2, que é o melhor atualmente. No capítulo 8 apresentamos algumas considerações finais, e no apêndice A mostramos alguns exemplos ilustrando o funcionamento de alguns algoritmos visto no capítulo 7.

CAPÍTULO 2

Preliminares

Neste capítulo apresentaremos alguns conceitos básicos que são essenciais para entender os problemas e os algoritmos estudados. Como o kMST é um problema sobre grafos, começaremos explicando alguns conceitos necessários de teoria dos grafos e a notação utilizada. Também explicaremos brevemente o conceito de algoritmo de aproximação, que é de principal interesse para esta dissertação.

Como notação básica utilizaremos os símbolos $\mathbb{Z}$, $\mathbb{Q}$ e $\mathbb{R}$ para denotar respectivamente os conjuntos de números inteiros, racionais e reais. Os símbolos $\mathbb{Z}_{\geq}$ e $\mathbb{Q}_{\geq}$ denotam respectivamente os conjuntos de números inteiros não-negativos e racionais não-negativos.

Em geral, se f é uma função definida sobre um conjunto X, então, f_x é o valor da função f em x, onde x é um elemento de X. Se X' é um subconjunto de X, então $f(X') = \sum_{x \in X'} f_x$.

2.1 Grafos, florestas e árvores

Um **grafo** G é um par (V, E), onde V é um conjunto finito qualquer e E é um conjunto de pares não-ordenados de elementos de V. Os elementos de V são chamados **vértices** e os elementos de E são chamados **arestas**. Em alguns casos é interessante considerar que as arestas são pares ordenados, neste caso chamamos o grafo de **orientado**. Um grafo é **completo** se seu conjunto de arestas é $E = \{\{u, v\} \mid u, v \in V, u \neq v\}$. Denotamos um grafo completo com n vértices por K^n.

Em toda esta dissertação, a menos que seja dito o contrário, usaremos a notação $n := |V|$ e $m := |E|$ se não houver dúvidas sobre qual é o grafo (V, E) referido. Para todo grafo G, denotaremos por V_G seu conjunto de vértices e por E_G seu conjunto de arestas quando estes não estiverem explícitos, ou seja $G = (V_G, E_G)$.

Se u e v são vértices em V e $\{u, v\}$, ou simplesmente uv, é uma aresta em E, então u e v são as **pontas da aresta** uv. Dizemos, então que u e v são **adjacentes** ou **vizinhos**. O **grau** de um vértice é o número de vizinhos que possui.

Seja (V, E) um grafo e S um subconjunto de V. Denotamos por $\delta(S)$ o **corte** $(S,\ V \setminus S)$, isto é, o subconjunto de arestas de E que possuem uma ponta em S e outra em $V \setminus S$.

Se (V, E) é um grafo, dizemos que (V', E') é um **subgrafo** de (V, E), ou simplesmente $(V', E') \subseteq (V, E)$, se V' é um subconjunto de V e E' é um subconjunto de E. Um subgrafo (V', E') é chamado **gerador** de (V, E) se $(V', E') \subseteq (V, E)$ e $V' = V$.

Sejam G e X um subconjunto não-vazio de V. Denotamos por $G[X]$ o subgrafo de G **induzido** por X, isto é, $V_{G[X]} = X$ e $E_{G[X]} = \{uv \in E_G \mid u, v \in X\}$. De forma análoga definimos subgrafos induzidos por um subconjunto não vazio de E.

Um **caminho** em um grafo (V, E) é uma seqüência $\langle v_0, a_1, v_1, a_2, v_2, \ldots, a_p, v_p \rangle$ tal que v_i é um vértice em V para todo $i = 0, 1, \ldots, p$, a_i é uma aresta em E para todo $i = 1, 2, \ldots, p$, os vértices são dois a dois distintos e $a_i = v_{i-1}v_i$. Dizemos que v_0 e v_p são **pontas do caminho** e os demais vértices são chamados de **vértices internos** . Podemos denotar um caminho apenas por sua seqüência de vértices ou sua seqüência de arestas. Caminhos podem ser tratados como subgrafos de (V, E).

O **tamanho** de um caminho é o número de arestas nesse caminho. Chamamos de caminho **mais curto** entre dois vértices, um caminho com o menor número de arestas, dentre os caminhos que ligam esses vértices. Se existe uma função de custo nas arestas $c : E \to \mathbb{Q}_{\geq}$, então chamamos de caminho **mais curto** aquele que minimiza a soma de c_e para cada aresta e no caminho.

Um **circuito** em um grafo (V, E) é uma seqüência $\langle v_1, a_1, v_2, a_2, v_3, \ldots, v_p, a_p \rangle$ tal que v_i é um vértice em V para todo $i = 1, 2, \ldots, p$, a_i é uma aresta em E para todo $i = 1, 2, \ldots, p$, os vértices são dois a dois distintos, $a_i = v_iv_{i+1}$ para $i = 1, 2, \ldots, p-1$ e $a_p = v_pv_1$. Podemos denotar um circuito por sua seqüência de vértices ou sua seqüência de arestas. Circuitos podem ser tratados como subgrafos de (V, E).

Um grafo é chamado **conexo** se para todo u e v em V distintos existe um caminho com pontas u e v. Um subgrafo conexo maximal de um grafo é denominado **componente** do grafo. Note que se o grafo for conexo, ele possui apenas um componente que é ele próprio.

Chamamos um grafo sem circuitos de **floresta**. Se o grafo não possui circuitos e for conexo, dizemos que é uma **árvore**. Logo, cada componente de uma floresta é uma árvore. Em muitos casos é interessante identificar um determinado vértice de uma árvore, chamamos tal vértice de **raiz** da árvore.

Uma **trilha** em um grafo (V, E) é uma seqüência $\langle v_1, a_1, v_2, a_2, v_3, \ldots, v_p\rangle$ onde, para todo $i = 1, 2, \ldots, p$, v_i é um vértice em V, para todo $i = 1, 2, \ldots, p-1$, a_i é uma aresta em E com $a_i = v_i v_{i+1}$ e as arestas são duas a duas distintas. Se $v_1 = v_p$ dizemos que a trilha é **fechada**. Se a trilha contém todas as arestas do grafo, então dizemos que a trilha é **euleriana**.

2.2 Algoritmos de aproximação

Um **problema de otimização** consiste em três partes:

- conjunto de **instâncias**;
- conjunto de **candidatos a uma solução**, também conhecido como conjunto de **soluções viáveis**, para cada instância;
- função não-negativa que define o **valor**, ou **custo**, de um candidato a solução.

Problemas de otimização podem ser de **maximização** ou **minimização**. O kMST, MST e ST são problemas de minimização. Uma solução de um problema de otimização é um candidato de custo máximo, no caso de problemas de maximização, ou de custo mínimo, no caso de problemas de minimização.

No caso do kMST com raiz, uma instância é dada por um grafo conexo com custos não-negativos nas arestas, um número inteiro k e um vértice r. Um candidato a uma solução é uma k-árvore que contém r e seu valor é o custo dessa k-árvore. Como o grafo é conexo, se $1 \leq k \leq n$, onde n é o número de vértices do grafo, então sempre

existe um candidato a uma solução. Denotaremos por T^* uma solução do kMST, ou seja, uma k-árvore mínima e por OPT $:= c(T^*)$ o custo de T^*.

Muitos problemas de otimização são NP-difíceis. Isso quer dizer que sob a hipótese de que P $\neq$ NP, sabemos que não existem algoritmos polinomiais para eles. Existem pelo menos duas maneiras de tratar problemas NP-difíceis: buscar uma solução apesar da demora para obtê-la, ou obter eficientemente um candidato a uma solução razoavelmente bom. Os chamados algoritmos de aproximação são do segundo tipo.

Um **algoritmo de aproximação** é um algoritmo polinomial para problemas de otimização. Em vez de procurar por uma solução do problema, ele procura um candidato a uma solução cujo valor não está longe do valor de uma solução. Isto é, se S é uma solução do problema e C é um candidato a uma solução devolvido pelo algoritmo de aproximação, então, no caso de problemas de maximização temos

$$\text{valor}(C) \geq \alpha \cdot \text{valor}(S),$$

e no caso de problemas de minimização temos

$$\text{valor}(C) \leq \alpha \cdot \text{valor}(S),$$

onde α é um número que pode depender da instância. Em ambos os casos dizemos que C é uma **α-aproximação**. Também usamos a nomenclatura α-aproximação para denotar algoritmos de aproximação que devolvem uma α-aproximação para toda instância do problema.

Chamamos α de **fator de aproximação** do algoritmo. Note que em problemas de maximização temos $0 < \alpha \leq 1$ e em problemas de minimização temos $\alpha \geq 1$, sendo que um fator de aproximação $\alpha = 1$ corresponde a um algoritmo que encontra uma solução do problema. Portanto, quanto mais próximo de 1 for o fator de aproximação, melhor será o algoritmo de aproximação.

CAPÍTULO 3

O problema kMST

Neste capítulo definiremos o problema kMST e as suas variantes enzaizada e com peso. Na seção 3.4 apresentaremos uma demonstração de que o kMST é um problema NP-difícil.

3.1 Descrição do problema

Uma **k-árvore** de um grafo (V, E) é uma árvore com pelo menos k vértices. Se existe uma função de **custos** c de E em $\mathbb{Q}_{\geq}$, definimos o custo de uma k-árvore T, denotado por $c(T)$, como a soma dos custos de suas arestas.

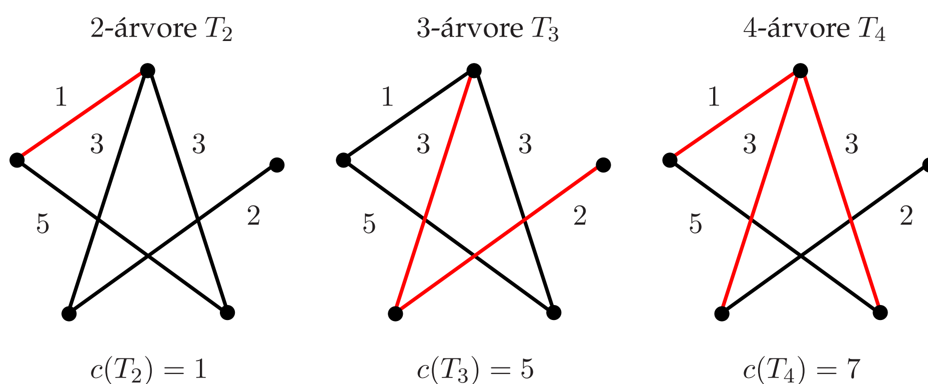


Figura 3.1: Exemplo de k-árvores com custo nas arestas.

Um problema clássico em otimização combinatória é o de encontrar uma árvore geradora de custo mínimo de um dado grafo com custos em suas arestas (MST). Se n é o número de vértices do grafo, esse problema consiste em encontrar uma n-árvore com o menor custo possível dentre todas as n-árvores do grafo dado. Existem algoritmos eficientes para resolver o MST [Kru56, Pri57].

O problema de encontrar uma k-árvore de custo mínimo (kMST) é uma generalização do MST, pois a quantidade de vértices da árvore que queremos encontrar é um dos parâmetros do problema. Uma descrição mais precisa é dada a seguir.

Problema kMST(V, E, c, k): *dados um grafo conexo (V, E), um custo não-negativo c_e para cada aresta e em E e um número inteiro k, encontrar uma k-árvore de custo mínimo.*

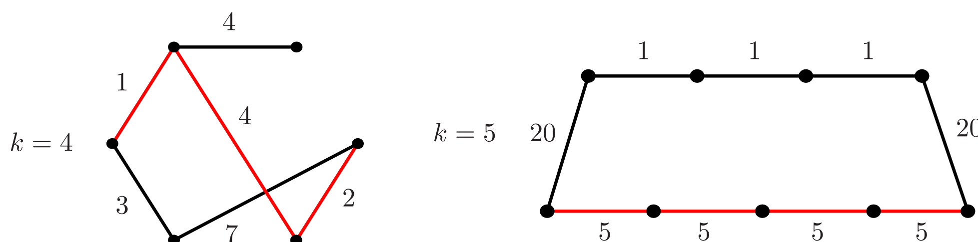


Figura 3.2: k-árvores de custo mínimo.

É claro que $1 \leq k \leq |V|$, caso contrário o problema não faz sentido.

O kMST é um problema NP-difícil [FHJM94, RSM+94, ZL93]. Isso significa que sob a hipótese de que P $\neq$ NP, não existem algoritmos eficientes para resolvê-lo. Assim, uma das maneiras de tratá-lo é por meio de algoritmos de aproximação.

Vários algoritmos de aproximação foram desenvolvidos para o kMST. O primeiro foi apresentado por Ramamurthy Ravi, Ravi Sundaram, Madhav V. Marathe, Daniel J. Rosenkrantz e Sekharipuram S. Ravi [RSM+94] e tem fator de aproximação $2\sqrt{k}$. Baruch Awerbuch, Yossi Azar, Avrim Blum e Santosh Vempala [AABV98] obtiveram um algoritmo melhor, com fator de aproximação $O(\log^2 k)$. Sridhar Rajagopalan e Vijay V. Vazirani [RV95] apresentaram um algoritmo com fator de aproximação $O(\log k)$. Um algoritmo com fator de aproximação 17, o primeiro fator constante, foi proposto por Avrim Blum, Ramamurthy Ravi e Santosh Vempala [BRV99]. Naveen Garg [Gar96]

mostrou em um mesmo artigo um algoritmo com fator de aproximação 5 e como refiná-lo para obter um fator de aproximação 3. Sanjeev Arora e George Karakostas [AK06] apresentaram um algoritmo com fator de aproximação $2 + \varepsilon$. Atualmente o algoritmo com menor fator de aproximação para o kMST foi obtido por Naveen Garg [Gar05b] e tem fator de aproximação 2.

$2\sqrt{k}$	Ravi, Sundaram, Marathe, Rosenkrantz e Ravi (1994)
$O(\log^2 k)$	Awerbuch, Azar, Blum e Vempala (1995)
$O(\log k)$	Rajagopalan e Vazirani (1995)
17	Blum, Ravi e Vempala (1995)
5 e 3	Garg (1996)
$2 + \varepsilon$	Arora e Karakostas (2000)
2	Garg (2000)

Tabela 3.1: Evolução do desenvolvimento de algoritmos de aproximação para o kMST.

3.2 kMST com raiz

Em problemas que envolvem árvores é comum existir duas versões do problema. Uma versão na qual a árvore deve ter uma determinada raiz e uma na qual não é definida uma raiz específica para a árvore. Chamaremos de **enraizada** ou **com raiz**, a variante do kMST definida da seguinte forma.

Problema kMST-R(V, E, c, k, r): *dados um grafo conexo* (V, E)*, um custo não-negativo* c_e *para cada aresta* e *em* E*, um número inteiro* k *e um vértice* r *em* V*, encontrar uma* k*-árvore de custo mínimo que contém* r*.*

É claro que os problemas kMST e kMST-R são diferentes, já que o kMST-R depende de um parâmetro a mais. No entanto, consideramos que esses problemas são equivalentes no sentindo em que um algoritmo eficiente que resolve um pode ser usado como subrotina de um algoritmo eficiente para o outro.

De fato, podemos resolver uma instância (V, E, c, k) do kMST, através de um algoritmo para o kMST-R, adicionando ao grafo (V, E) um vértice novo r conectado a cada vértice em V por arestas com custos suficientemente grandes, por exemplo, a soma dos

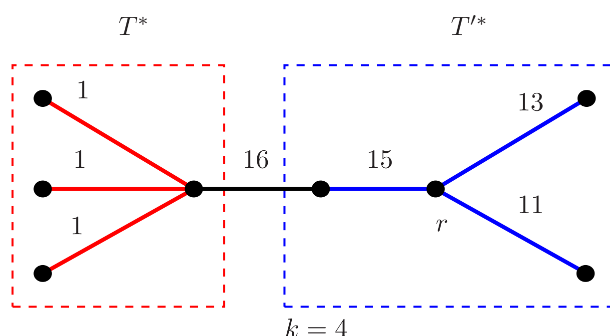


Figura 3.3: T^* é solução do kMST e T'^* é solução do kMST-R.

custos de todas as arestas de E. Dessa forma, basta encontrar nesse novo grafo uma $(k+1)$-árvore de custo mínimo e com raiz r, digamos T_r. Como as arestas incidentes a r têm custos grandes, apenas uma delas estará em T_r. Logo, removendo r de T_r, obtemos uma k-árvore de custo mínimo no grafo(V, E).

Para resolver uma instância (V, E, c, k, r) do kMST-R, através de um algoritmo para o kMST, adicionamos, ao grafo (V, E) dado, n vértices novos conectados a r por arestas de custo 0. Então, o que procuramos é uma $(k + n)$-árvore de custo mínimo. Pela quantidade de vértices adicionados, certamente r estará em uma $(k + n)$-árvore de custo mínimo. Removendo os n vértices adicionados, obtemos uma k-árvore de custo mínimo com raiz r no grafo (V, E).

Consideraremos tanto o problema sem raiz como o com raiz, dependendo de qual for mais conveniente para a situação.

3.3 kMST generalizado

Em alguns casos é interessante considerarmos grafos que além de custos nas arestas, possuem também **pesos** nos vértices. Dado um grafo (V, E), podemos considerar que os peso nos vértices são dados por uma função w de V em $\mathbb{Z}_{\geq}$.

Para grafos com pesos nos vértices, podemos pensar em uma definição mais geral de k-árvore. Uma **k-árvore com pesos** é uma árvore na qual a soma dos pesos de cada um de seus vértices é pelo menos k. Quando for claro que o grafo em questão possui pesos nos vértices chamaremos uma k-árvore com pesos apenas de k-árvore. Com essa definição mais geral, podemos formular uma versão mais geral do kMST-R.

Problema kMST-P(V, E, c, w, k, r): *dados um grafo conexo* (V, E)*, um custo não-negativo* c_e *para cada aresta* e *em* E*, um peso positivo* w_v *para cada vértice* v *em* V*, um número inteiro* k *e um vértice* r *em* V*, encontrar uma* k*-árvore de custo mínimo que contém* r*.*

O problema kMST-R é um caso particular do kMST-P. Para verificar isso, basta tomar w_v como 1 para todo vértice v do grafo. Poderíamos também enunciar uma versão do kMST-P sem raiz, mas neste texto só nos interessará a versão com raiz.

3.4 Complexidade computacional

Para provar que o kMST é NP-difícil vamos considerar a versão sem raiz. Mostraremos que um outro problema NP-difícil, que chamaremos de ST, pode ser reduzido polinomialmente a ele. Isso quer dizer que, dada uma instância do ST, podemos gerar em tempo polinomial uma instância do kMST, cuja solução pode ser transformada, em tempo polinomial, em uma solução para a instância do ST.

Sejam (V, E) um grafo conexo e R um subconjunto de V. Chamamos R de **conjunto de terminais**. Uma **árvore de Steiner** é uma árvore que contém todos os vértices terminais. A descrição do problema ST é dada a seguir.

Problema ST(V, E, R): *dados um grafo conexo* (V, E) *e um subconjunto* R *de* V*, encontrar uma árvore de Steiner com número mínimo de arestas.*

O problema ST é sabidamente NP-difícil [GJ90].

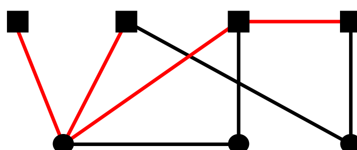


Figura 3.4: Árvore de Steiner mínima. Os vértices quadrados representam os terminais.

Ravi, Sundaram, Marathe, Rosenkrantz e Ravi [RSM+94] provaram o seguinte teorema, mostrando que o kMST é NP-difícil.

Teorema 3.1. O problema ST pode ser reduzido polinomialmente ao kMST.

Prova. Dada uma instância (V, E, R) do ST, criamos uma instância correspondente (V', E', c, k) para o kMST da seguinte forma.

Seja x o número de vértices não-terminais de (V, E), isto é, $x = |V \setminus R|$. O grafo (V', E') possui uma cópia de (V, E) e cada terminal possui um caminho de tamanho x conectado a ele (figura 3.5). Logo, (V', E') tem $|V| + x|R|$ vértices. Para cada aresta em E atribuímos custo 1 e para as demais arestas atribuímos custo 0. Finalmente, tomamos $k = |R| \cdot (x + 1)$.

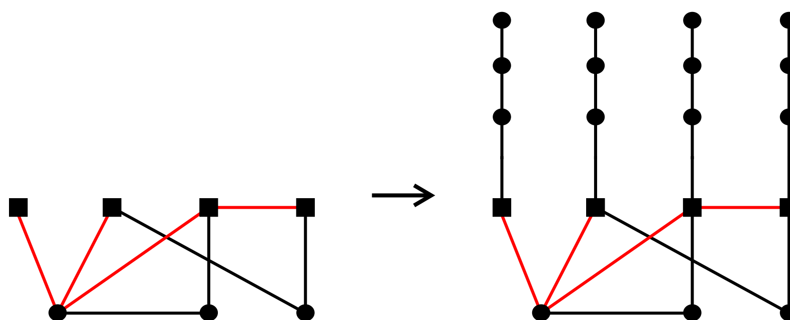


Figura 3.5: Ilustração da construção de (V', E') a partir de (V, E, R). Os vértices quadrados representam o conjunto R.

Uma k-árvore mínima em (V', E') possui o menor número possível de arestas de custo 1. Ademais, pela escolha de k, sabemos que qualquer k-árvore em (V', E') contém todos os vértices de R. Então, uma k-árvore mínima contém uma árvore de Steiner mínima e caminhos de custo 0 saindo de cada terminal. Logo, a partir de uma k-árvore mínima em (V', E'), podemos obter uma árvore de Steiner mínima em (V, E).

Por outro lado, a partir de uma árvore de Steiner mínima em (V, E), basta adicionar os caminhos de custo 0 para obter uma k-árvore mínima em (V', E').

Portanto ST pode ser polinomialmente reduzido ao kMST. □

Também existem outras provas de que o kMST é um problema NP-difícil obtidas de forma independente por Matteo Fischetti, Horst Wilhelm Hamacher, Kurt O. Jørnsten e Francesco Maffioli [FHJM94] e por Alexander Zelikovsky e Dmitrii Lozovanu [ZL93].

CAPÍTULO 4

Algoritmos exatos

Vimos na seção 3.4 que o kMST é um problema NP-difícil, ou seja, não se conhecem formas eficientes de se resolvê-lo. Então, uma maneira de tratá-lo é por algoritmos de aproximação. Mas antes estudaremos alguns casos particulares, para os quais conhecemos algoritmos polinomiais, para tentar entender melhor o problema.

Primeiro apresentaremos um algoritmo ingênuo que resolve o kMST de forma exata, mas não é polinomial. Em seguida, discutiremos o que acontece quando restringimos o valor do parâmetro k. Também veremos os casos nos quais restringimos os valores dos custos das arestas e também o grafo.

4.1 Algoritmo enumerativo

Os grafos de uma instância do problema kMST são finitos, isto é, com número de vértices e arestas finitos. Logo, um algoritmo simples que termina em tempo finito consiste em enumerar todas as k-árvores do grafo e escolher uma de menor custo. Esse algoritmo certamente resolve o problema, mas o número de k-árvores no grafo pode ser muito grande.

Veremos que se $k = n$ existe um algoritmo polinomial para o kMST. Então, se considerarmos o K^n, podemos resolver para cada subconjunto de k vértices e tomar a melhor k-árvore. Mas o número de subconjuntos de k vértices é

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!} = \mathrm{O}(n^k).$$

Assim, poderíamos ter que resolver um número exponencial de subproblemas, tornando o algoritmo muito ineficiente.

Portanto, utilizar o método enumerativo dessa forma é inviável. No entanto, como veremos a seguir, quando temos mais informações sobre a estrutura da solução podemos desenvolver métodos enumerativos eficientes.

4.2 kMST para k específicos

O problema só faz sentido se temos $1 \leq k \leq n$. Podemos considerar ainda que $k \geq 2$, pois $k = 1$ é trivial e nada interessante. O caso $k = 2$, no qual queremos encontrar uma aresta de custo mínimo, também é trivial, mas já exige algum esforço computacional. Basta listar todas as arestas e devolver uma de menor custo.

O caso $k = 3$ também é simples. Basta observar que uma 3-árvore com 3 vértices é sempre um caminho com duas arestas. Então, sempre tem um vértice de grau 2 e dois de grau 1. Logo, para cada vértice do grafo com grau pelo menos 2, encontramos uma 3-árvore selecionando duas arestas incidentes a ele com o menor custo possível. Dentre as 3-árvores encontradas, devolvemos a de menor custo.

Para $k = 4$ já fica um pouco mais complicado, mas ainda podemos resolvê-lo. Uma 4-árvore com 4 vértices tem um vértice de grau 3 e três de grau 1 ou é um caminho com 3 arestas. No primeiro caso, basta fazer uma enumeração parecida com a que fazemos para $k = 3$. No segundo caso, para cada aresta uv do grafo, encontramos uma 4-árvore selecionando, quando existir, uma aresta incidente a u e uma incidente a v, distintas de uv e com menor custo possível. Dentre todas as 4-árvores encontradas, devolvemos a de menor custo.

De forma geral, se k for uma constante, ou seja, o valor de k é o mesmo para todas as instâncias do problema, então podemos resolver o kMST em tempo polinomial. Para k constante, o algoritmo ingênuo da seção anterior fica polinomial no tamanho da entrada. É claro que na prática esse algoritmo pode continuar sendo ineficiente.

Um caso bem conhecido ocorre quanto $k = n$. Neste caso temos o problema da árvore geradora mínima. Veremos esse problema na próxima seção.

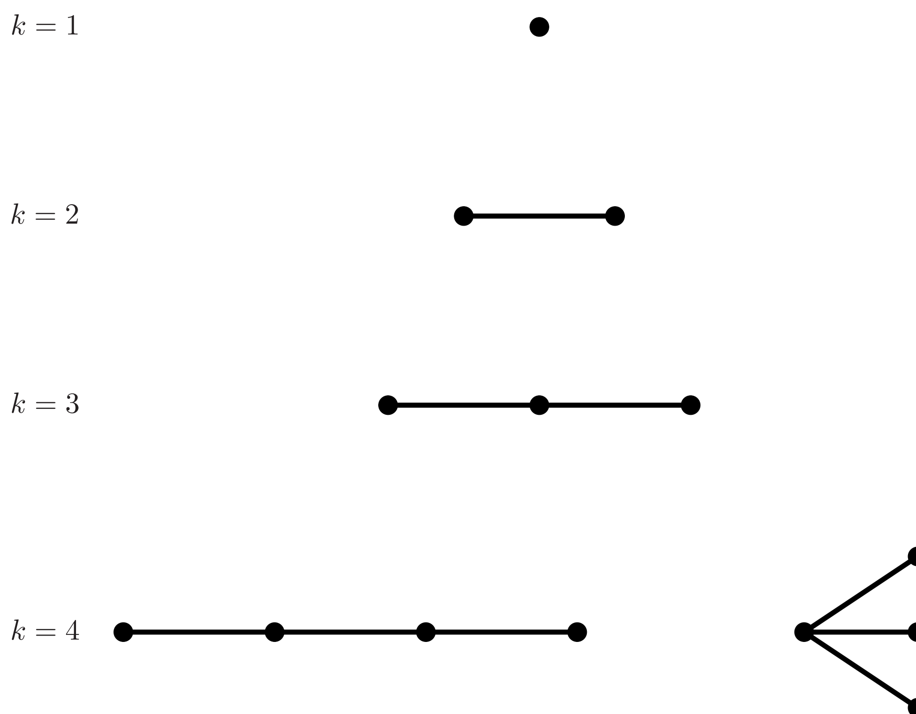


Figura 4.1: Forma de k-árvores com k vértices para $k \in \{1, 2, 3, 4\}$.

4.3 Árvore geradora mínima

Uma **árvore geradora** de um grafo G é uma árvore que é subgrafo gerador de G. Em outras palavras, é uma árvore de G contendo todos os seus vértices. Chamamos de **árvore geradora mínima** uma árvore geradora cuja soma dos custos de suas arestas é a menor possível.

Existem algoritmos polinomiais para resolver o problema da árvore geradora mínima. Um exemplo é o conhecido algoritmo de Joseph Bernard Kruskal Jr. [Kru56], que recebe um grafo conexo (V, E) e uma função custo $c : E \to \mathbb{Q}_{\geq}$ e devolve uma árvore geradora mínima F do grafo.

Algoritmo KRUSKAL(V, E, c)

1 $F \leftarrow (V, \emptyset)$
2 enquanto não existe uma n-árvore em F faça
3 seja e uma aresta com c_e mínimo e pontas em componentes distintos de F.
4 $F \leftarrow F + e$
5 devolva F

Basicamente o algoritmo de Kruskal começa com uma floresta F na qual cada vértice é um componente. Em cada iteração inserimos em F uma aresta que possui pontas em componentes de F distintos. Logo, F continua sendo uma floresta. As arestas são examinadas em ordem não decrescente de custo. Dessa forma o algoritmo mantém as seguintes relações invariantes no início de cada iteração:

- F é uma floresta de custo mínimo dentre as florestas de (V, E) com $|E_F|$ arestas;
- cada componente C de F é uma árvore geradora mínima de $G[V_C]$.

Um outro algoritmo conhecido para o problema MST é o algoritmo de Robert Clay Prim [Pri57]. O algoritmo recebe um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$ e um vértice r de V e devolve uma árvore geradora mínima F do grafo. Denotaremos por $\delta(S)$ o corte $(S, V \setminus S)$, onde S é um subconjunto de V.

Algoritmo PRIM(V, E, c, r)

1 $F \leftarrow (\{r\}, \emptyset)$
2 enquanto $|V_F| < n$ faça
3 seja uv, $u \in V_F$, uma aresta de menor custo em $\delta(V_F)$
4 $V_F \leftarrow V_F \cup \{v\}$
5 $E_F \leftarrow E_F \cup \{uv\}$
6 devolva F

O vértice r pedido no algoritmo de Prim pode ser um vértice qualquer de V, pois serve apenas para indicar ao algoritmo por onde começar a construir a árvore geradora. A partir de r o algoritmo cresce uma árvore até que essa árvore possua todos os vértices de V. O algoritmo mantém a seguinte relação invariante no início de cada iteração:

- F é uma árvore geradora mínima em $G[V_F]$.

O consumo de tempo dos algoritmos de Kruskal e de Prim é O$(m \log n)$ [CLRS01]. Dependendo da implementação e das estruturas de dados utilizadas, o consumo de tempo dos algoritmos de Kruskal e de Prim podem ser melhorados. Mais detalhes sobre isso podem ser encontrados no livro de Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, Clifford Stein [CLRS01].

4.4 kMST em grafos completos

Nesta seção vamos considerar apenas instâncias do kMST nas quais o grafo (V, E) é completo e $c_e \in \{1, 2\}$ para todo $e \in E$. Na verdade, os valores 1 e 2 para os custos das arestas foram escolhidos para facilitar o entendimento e poderiam ser quaisquer dois valores positivos distintos. O importante é que só existam dois valores possíveis para os custos das arestas.

A seguinte adaptação do algoritmo KRUSKAL resolve o kMST restrito às instâncias descritas acima. Esse algoritmo recebe um grafo completo (V, E), um número inteiro k e uma função custo $c : E \to \{1, 2\}$ e devolve uma k-árvore mínima T do grafo.

Algoritmo KRUSKAL-MODIFICADO(V, E, k, c)

1 $E_1 \leftarrow \{e \in E \mid c_e = 1\}$
2 $E_2 \leftarrow \{e \in E \mid c_e = 2\}$
3 $F \leftarrow (V, \emptyset)$
4 enquanto não existe uma k-árvore em F e existe aresta e_1 em E_1
  com pontas em componentes distintos de F faça
5   $F \leftarrow F + e_1$
6 enquanto não existe uma k-árvore em F faça
7   seja e_2 uma aresta em E_2 com pontas em componentes
    distintos de F e com maior número de vértices.
8   $F \leftarrow F + e_2$
9 $T \leftarrow k$-ÁRVORE(F, k)
10 devolva T

Assim como no algoritmo KRUSKAL a cada iteração, nos dois laços, estamos conectando dois componentes e, assim, obtendo um componente maior. Note que dessa forma, ao chegar na linha 9 do algoritmo, F tem apenas um componente com pelo menos k vértices.

A rotina k-ÁRVORE(F, k) devolve uma k-árvore de F que tem exatamente k vértices. Isso não é uma tarefa difícil, pois, sabemos que F é uma floresta com apenas um componente contendo pelo menos k vértices. Tal componente é uma k-árvore. Se essa k-árvore tem mais do que k vértices, basta remover folhas da árvore até que ela fique com exatamente k vértices.

Teorema 4.1. Dados um grafo completo (V, E), um número inteiro $k \leq |V|$ e uma função custo $c : E \rightarrow \{1, 2\}$, o algoritmo KRUSKAL-MODIFICADO devolve uma k-árvore mínima.

Prova. Sejam T^* uma k-árvore de custo mínimo e T a k-árvore devolvida pelo algoritmo KRUSKAL-MODIFICADO. Denotaremos por $\mathscr{C} = \{C_1, C_2, \ldots, C_p\}$ o conjunto dos componentes do grafo $(V, \{e \in E \mid c_e = 1\})$. Suponha que $|V_{C_1}| \geq |V_{C_2}| \geq \cdots \geq |V_{C_p}|$.

Certamente temos que $|V_T| = |V_{T^*}| = k$ e $|E_T| = |E_{T^*}|$. Então queremos mostrar que $c(T) = c(T^*)$. Como $|E_T| = |E_{T^*}|$, isso ocorre se e somente se T e T^* possuem a mesma quantidade de arestas de custo 2. Do algoritmo, temos que toda aresta de custo 2 em T tem pontas em componentes de $\mathscr{C}$ diferentes. O mesmo ocorre em T^*, caso contrário, poderíamos remover uma aresta de custo 2 com ambas as pontas em um componente, digamos C, de $\mathscr{C}$ e adicionar uma de custo 1 que também possui ambas as pontas em C. Com isso obteríamos uma k-árvore com custo menor que $c(T^*)$, o que é uma contradição.

Logo, para mostrar que $c(T) = c(T^*)$, basta mostrar que T e T^* intersectam o mesmo número de componentes em $\mathscr{C}$, já que (V, E) é completo.

Suponha que T^* intersecta $y \leq p$ componentes em $\mathscr{C}$. A soma do número de vértices desses y componentes é pelo menos k. Logo

$$\sum_{i=1}^{y} |V_{C_i}| \geq k.$$

Então temos que T intersecta no máximo y componentes em $\mathscr{C}$. Se T intersectasse menos que y componentes, teríamos $c(T) < c(T^*)$, contradizendo o fato de T^* ser mínima. Assim, T intersecta exatamente y componentes em $\mathscr{C}$. Portanto $c(T) = c(T^*)$. □

Uma execução do algoritmo KRUSKAL-MODIFICADO é praticamente uma execução do algoritmo de Kruskal podendo, possivelmente, parar antes. Logo, o algoritmo KRUSKAL-MODIFICADO tem consumo de tempo polinomial.

4.5 k-árvores de árvores

O último caso particular do kMST que veremos é o seguinte.

Problema kMST-A(V, E, c, k, r): *dados uma árvore* (V, E)*, um custo não-negativo* c_e *para cada aresta* e *em* E*, um número inteiro* k *e um vértice* r *em* V*, encontrar uma* k*-árvore de custo mínimo que contém* r*.*

Uma idéia natural para o kMST-A seria utilizar o algoritmo PRIM com r como vértice inicial, mas parando no momento em que a árvore construída no algoritmo possuir k vértices. No entanto, essa idéia não funciona. Um exemplo disso é o grafo da figura 4.2, com $k = 4$. Note que independentemente da raiz escolhida, a k-árvore devolvida por tal algoritmo terá sempre custo 200. No entanto, se $r \in \{v_1, v_2, v_4, v_5\}$ temos que o custo da k-árvore de custo mínimo com raiz r é 102.

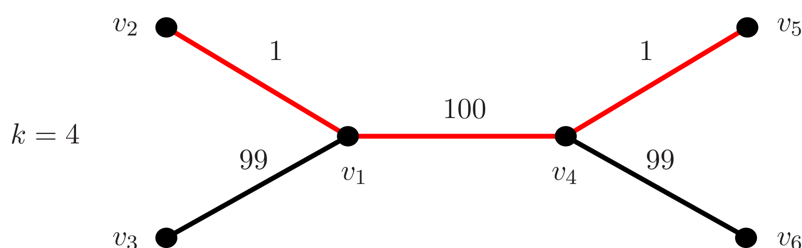


Figura 4.2: Exemplo de que PRIM não resolve kMST-A.

Francesco Maffioli [Maf91] propôs um algoritmo de programação dinâmica para encontrar o custo de uma k-árvore mínima com raiz r em árvores. Christian Blum [Blu07] mostra as estruturas de dados necessárias para que esse algoritmo também devolva uma k-árvore mínima.

Para desenvolver um algoritmo de programação dinâmica, um problema de otimização precisa ter a chamada propriedade da **subestrutura ótima**. Isto é, podemos construir a solução do problema através da solução de subproblemas. Mostramos no lema 4.2 que o kMST-A possui essa propriedade.

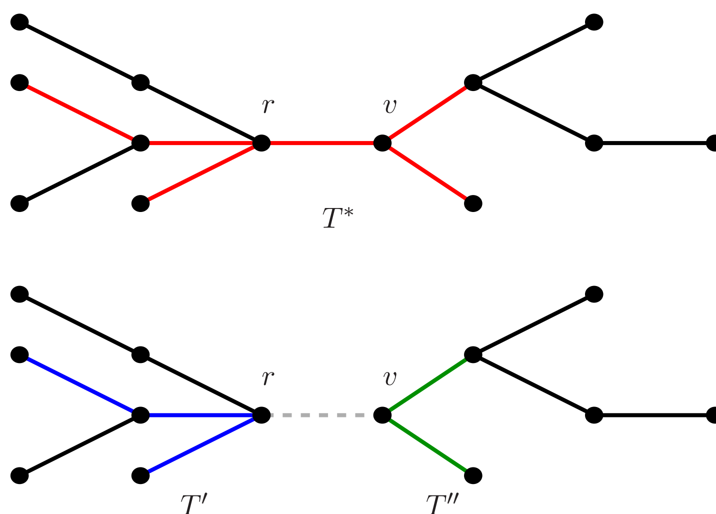


Figura 4.3: Ilustração da subestrutura ótima do problema.

Lema 4.2. O problema kMST-A possui a propriedade da subestrutura ótima.

Prova. Primeiro mostraremos como são os subproblemas do kMST-A para a instância (V, E, c, k, r) e depois mostraremos que conseguimos construir uma solução para tal instância através de soluções para esses subproblemas.

Seja $T = (V, E)$ e T^* uma k-árvore mínima em T contendo r. Consideraremos uma aresta rv em T^*. Note que $T - rv$ consiste de duas árvores disjuntas T_r e T_v, uma contendo o vértice r e a outra contendo v respectivamente. Denotaremos por c_r os custos c restritos a T_r e $k_r := |V_{T^*} \cap V_{T_r}|$. Definimos c_v e k_v de maneira análoga.

Considere os subproblemas kMST-A$(V_{T_r}, E_{T_r}, c_r, k_r, r)$ e kMST-A$(V_{T_v}, E_{T_v}, c_v, k_v, v)$. Temos que $T' = T^*[V_{T^*} \cap V_{T_r}]$ é uma solução de kMST-A$(V_{T_r}, E_{T_r}, c_r, k_r, r)$, caso contrário existiria uma k-árvore em T de custo menor que $c(T^*)$. Analogamente, temos que $T'' = T^*[V_{T^*} \cap V_{T_v}]$ é uma solução de kMST-A$(V_{T_v}, E_{T_v}, c_v, k_v, v)$.

Note que $T^* = T' + T'' + rv$. Logo, podemos obter a T^* da solução de subproblemas. Portanto, o problema kMST-A possui a propriedade da subestrutura ótima. $\square$

Seja T uma árvore com raiz r. Para cada vértice v em $V_T \setminus \{r\}$, denotaremos por pai(v) o vértice adjacente a v no caminho de v a r em T. E para cada vértice v em V, denotaremos por T_v a **subárvore** de T **enraizada** em v. Ou seja, T_v é tal que um vértice v' em V está em T_v se e somente se o caminho de r a v' em T contém v. Vamos supor que $V_T = \{v_1, v_2, \ldots, v_n\}$, $v_1 = r$, e para cada $i \geq 2$ temos que pai$(v_i) \in \{v_1, v_2, \ldots, v_{i-1}\}$.

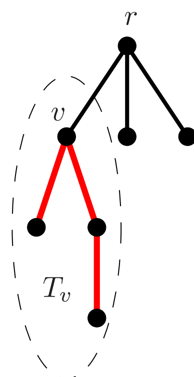


Figura 4.4: Exemplo de uma árvore enraizada em v.

Sejam v um vértice em V e j tal que $1 \leq j \leq k$. Denotaremos por $d_{v,j}$ o custo de uma j-árvore com raiz v de custo mínimo em T_v. Então temos que $d_{v,j}$ é o custo de uma solução de kMST-A$(V_{T_v}, E_{T_v}, c, j, v)$. Considere o conjunto F_v de todos os vértices de V cujo pai é v. Considere também uma função $\alpha : \mathbb{Z}_{\geq} \to \{0, 1\}$, tal que $\alpha(x) = 0$ se e somente se $x = 0$. Então temos a seguinte recorrência:

$$d_{v,j} = \begin{cases} 0, \text{ se } j = 1 \\ \min\left\{\sum_{u \in F_v} \alpha(j_u)(c_{uv} + d_{u,j_u})\right\}, \text{ se } 2 \leq j \leq |V_{T_v}| \\ \infty, \text{ se } j > |V_{T_v}| \end{cases} \tag{4.1}$$

onde o mínimo é tomado sobre todos os conjuntos de valores j_u tais que para todo u em F_v temos $j_u \geq 0$ e $\sum_{u \in F_v} j_u = j - 1$.

Se $|F_v| = 1$, o cálculo da recorrência é trivial. Se $|F_v| = 2$, o cálculo ainda é simples, pois, supondo que $F_v = \{u_1, u_2\}$, temos que $1 \leq j_{u_1} \leq j - 2$ e $j_{u_2} = j - j_{u_1} - 1$. No entanto, se $|F_v| > 2$, o cálculo começa a ser inviável, pois o número de combinações possíveis para os j_u fica exponencial.

Mas, existe uma maneira de calcular a recorrência (4.1) em tempo polinomial. Seja $F_v = \{u_1, u_2, \ldots, u_z\}$. Dado que já conhecemos $d_{u_q,i}$, para $1 \leq q \leq z$ e $1 \leq i \leq j - 1$,

a idéia é calcular o custo das j-árvores em $T[\{v\} \cup V_{T_{u_1}}]$, $2 \leq j \leq k$, e atualizamos o valor de $d_{v,j}$. Em seguida fazemos o mesmo em $T[\{v\} \cup V_{T_{u_1}} \cup V_{T_{u_2}}]$ e atualizamos $d_{v,j}$. E assim sucessivamente até considerarmos $T[\{v\} \cup V_{T_{u_1}} \cup V_{T_{u_2}} \cup \cdots \cup V_{T_{u_z}}] = T_v$. Isso equivale a considerar em cada uma das vezes o caso em que $|F_v| = 2$.

Então, o que precisamos fazer é calcular, para cada $u \in F_v$, o valor

$$\widetilde{d}_{v,j} = \min_{1 \leq i \leq j-1,} \{d_{v,j-i} + c_{vu} + d_{u,i}\}$$

para todo $j \in \{2, 3, \ldots, k\}$. Antes de passar para o próximo vértice em F_v, atualizamos $d_{v,j}$ com o valor de $\widetilde{d}_{v,j}$ para todo j.

Paulo Feofiloff [Feo] descreve um algoritmo de programação dinâmica baseado nessa recorrência com consumo de tempo $O(nk^2) = O(n^3)$, pois $k \leq n$.

Algoritmo PD(V, E, c, k, r)

1 considere que $V = \{v_1, v_2, \ldots, v_n\}$, $v_1 = r$, e para cada $i \geq 2$ temos que pai$(v_i) \in \{v_1, v_2, \ldots, v_{i-1}\}$
2 para cada v em V faça
3 $\quad d_{v,1} \leftarrow 0$
4 $\quad d_{v,2} \leftarrow d_{v,3} \leftarrow \cdots \leftarrow d_{v,k} \leftarrow \infty$
5 para q de $|V|$ decrescendo até 2 faça
6 $\quad v_p \leftarrow$ pai(v_q)
7 $\quad$ para j de 2 até k faça
8 $\quad\quad \widetilde{d}_j \leftarrow d_{v_p,j}$
9 $\quad\quad$ para i de 1 até $j-1$ faça
10 $\quad\quad\quad \widetilde{d}_j \leftarrow \min\{\widetilde{d}_j,\ d_{v_p,j-i} + c_{v_p v_q} + d_{v_q,i}\}$
11 $\quad$ para j de 2 até k faça
12 $\quad\quad d_{v_p,j} \leftarrow \widetilde{d}_j$
13 devolva $d_{v_1,k}$

O algoritmo acima apenas devolve o custo de uma k-árvore mínima com raiz r. Para obter a correspondente k-árvore mínima, bastaria adicionar uma nova matriz, digamos d', com mesma dimensão de d, tal que $d'_{v,j}$ armazena a árvore associada ao custo $d_{v,j}$. Assim, toda vez que atualizarmos $d_{v,j}$, também atualizamos $d'_{v,j}$.

A seguir mostramos uma breve simulação do algoritmo PD para a instância definida pela figura abaixo.

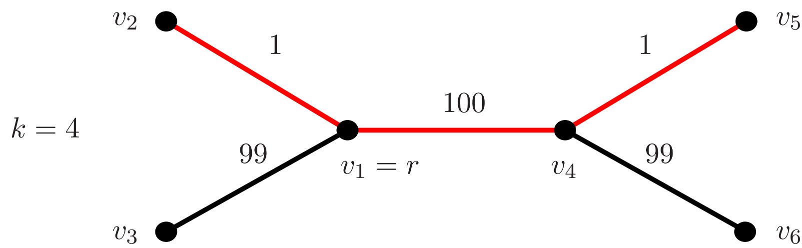


Figura 4.5: Em destaque mostramos a solução.

	1	2	3	4
v_1	0	∞	∞	∞
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	∞	∞	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.1: Matriz d inicial.

	1	2	3	4
v_1	0	∞	∞	∞
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	99	∞	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.2: Matriz para $q = 6$ ($v_p = v_4$).

	1	2	3	4
v_1	0	∞	∞	∞
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	1	100	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.3: Matriz para $q = 5$ ($v_p = v_4$).

	1	2	3	4
v_1	0	100	101	200
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	1	100	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.4: Matriz para $q = 4$ ($v_p = v_1$).

	1	2	3	4
v_1	0	99	101	200
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	1	100	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.5: Matriz para $q = 3$ ($v_p = v_1$).

	1	2	3	4
v_1	0	1	100	102
v_2	0	∞	∞	∞
v_3	0	∞	∞	∞
v_4	0	1	100	∞
v_5	0	∞	∞	∞
v_6	0	∞	∞	∞

Tabela 4.6: Matriz para $q = 2$ ($v_p = v_1$).

CAPÍTULO 5

Algoritmos combinatórios

Neste capítulo veremos os primeiros algoritmos de aproximação desenvolvidos para o kMST. Todos eles são baseados em idéias de algoritmos bem conhecidos como o algoritmo de Dijkstra para caminho mínimo e o algoritmo de Kruskal (seção 4.3) para árvore geradora mínima.

5.1 k-KRUSKAL

Na seção 4.3 vimos um algoritmo polinomial para o kMST no caso específico em que $k = n$. Uma idéia natural seria tentar adaptar esse algoritmo para resolver o kMST em tempo polinomial. No entanto, por se tratar de um problema NP-difícil, isso provavelmente não é possível.

Se não sabemos adaptar o algoritmo para resolver o kMST em tempo polinomial, podemos tentar adaptá-lo para ser um algoritmo de aproximação para o kMST. Um algoritmo de aproximação bem simples, que chamaremos de k-KRUSKAL, obtido ao adaptarmos o algoritmo de Kruskal é descrito a seguir.

O k-KRUSKAL recebe um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$ e um número inteiro k e devolve uma k-árvore T do grafo.

Algoritmo k-KRUSKAL (V, E, c, k)

1 $F \leftarrow (V, \emptyset)$
2 enquanto não existe uma k-árvore em F faça
3 seja e uma aresta com c_e mínimo e pontas em componentes distintos de F.
4 $F \leftarrow F + e$
5 $T \leftarrow k$-ÁRVORE(F, k)
6 devolva T

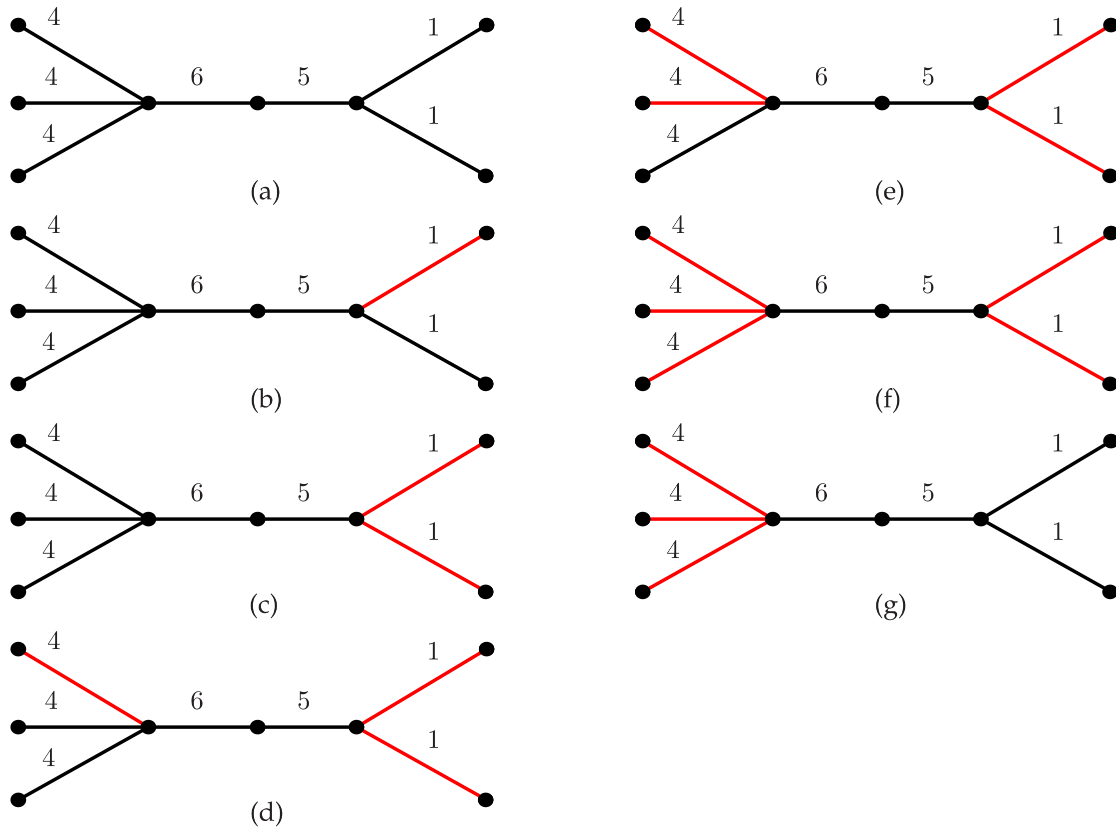


Figura 5.1: Simulação do algoritmo k-KRUSKAL ($k = 4$). (a) Mostra o grafo considerado. De (b) a (f) são inseridas arestas uma a uma, ordenadas pelo custo, até se obter uma k-árvore. Em (g), arestas que não fazem parte da k-árvore são removidas.

O k-KRUSKAL difere do Kruskal apenas na condição de parada (linha 2) e no objeto devolvido (linha 5). Como ele executa no máximo o mesmo número de iterações que o algoritmo de Kruskal, temos que k-KRUSKAL também é polinomial.

Enquanto Kruskal devolve uma árvore geradora mínima, ou uma n-árvore de custo mínimo, k-KRUSKAL devolve uma k-árvore T, não necessariamente mínima. Vamos supor, sem perda de generalidade, que T tem exatamente k vértices, caso contrário podemos remover folhas de T, sem aumentar o seu custo, até que T tenha exatamente k vértices.

Teorema 5.1. O fator de aproximação do algoritmo k-KRUSKAL é $k - 1$.

Prova. Seja T a k-árvore devolvida pelo algoritmo. Seja e a última aresta inserida na floresta F pelo algoritmo. Denotaremos por OPT o custo de uma k-árvore mínima.

Note que antes de adicionar e todo componente da floresta tem menos de k vértices. Logo, qualquer k-árvore mínima tem pelo menos uma aresta, que não está em $F - e$. Como no algoritmo as arestas são examinadas em ordem não-decrescente de custo, então vale que $c_e \leq \text{OPT}$.

Portanto, $c(T) \leq (k-1)c_e \leq (k-1)\text{OPT}$. □

O teorema 5.1 nos garante que a k-árvore devolvida por k-KRUSKAL tem custo no máximo $k - 1$ vezes maior que o custo de uma k-árvore mínima. Esse fator de aproximação não é bom. A princípio isso não quer dizer que o algoritmo também não é bom, pois usando um outro limitante para o custo da k-árvore devolvida poderíamos obter um fator de aproximação melhor para o mesmo algoritmo. No entanto, não é este o caso do k-KRUSKAL. Os exemplos da figura 5.2 mostram que é possível construir instâncias nas quais o k-KRUSKAL devolve k-árvores com custos cada vez mais próximos de $k - 1$ vezes o custo de uma k-árvore mínima. Neste caso dizemos que o fator de aproximação do algoritmo é **justo**.

Lema 5.2. Seja T^* uma k-árvore mínima com custo OPT. Seja x o número de arestas de T^* que não estão em F. Se $x > 0$, então o custo de uma árvore geradora mínima de um componente C qualquer é no máximo $\frac{|V_C|}{x}\text{OPT}$.

Prova. Dentre as arestas de T^* que não estão em F, seja e uma de menor custo. Então temos que $x \cdot c_e \leq \text{OPT}$.

Seja T_C uma árvore geradora mínima de C. Dado que as arestas são processadas em ordem não-decrescente de custo, temos que o custo de qualquer aresta de T_C é no máximo c_e. Portanto, $c(T_C) \leq (|V_C| - 1)c_e \leq \frac{|V_C|}{x}\text{OPT}$. □

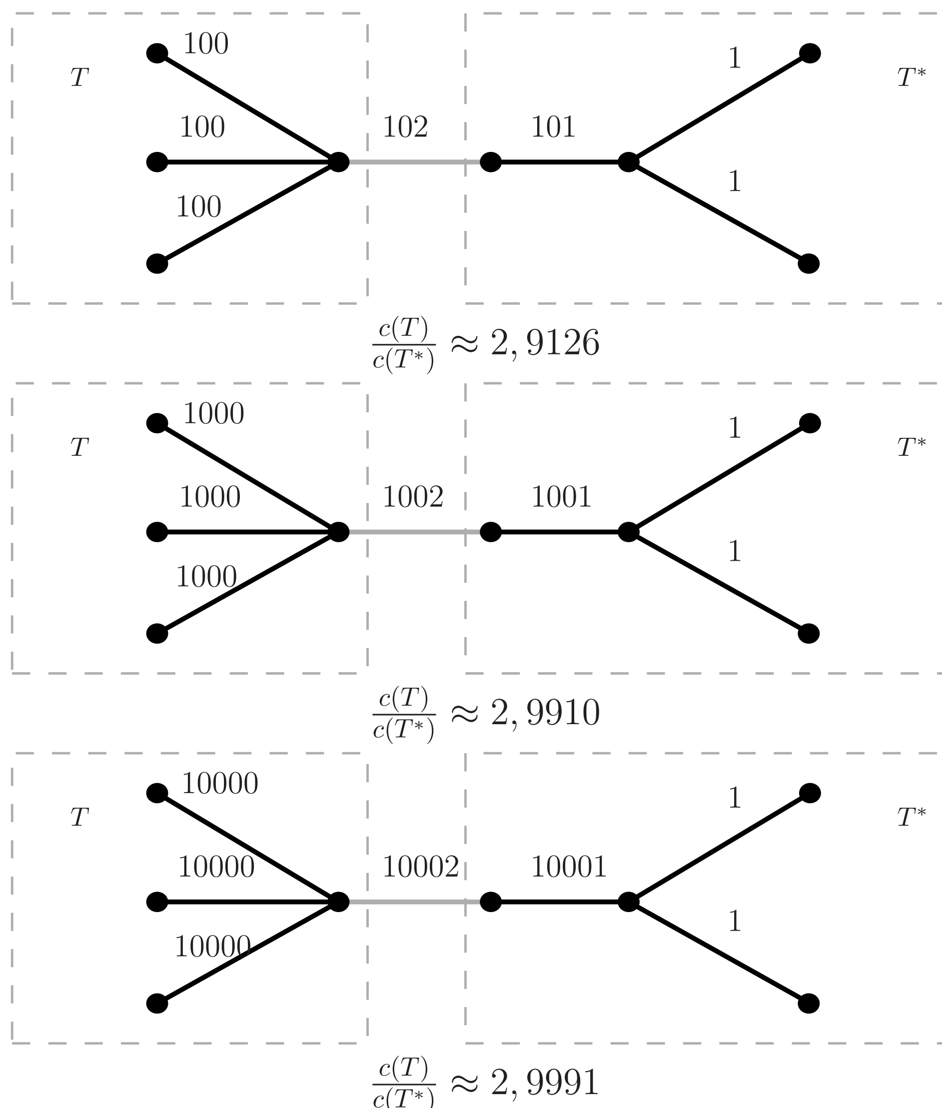


Figura 5.2: Exemplos de que a k-árvore devolvida por k-KRUSKAL pode ter custo tão próximo quanto se queira de $k-1$ vezes o custo de uma k-árvore mínima ($k=4$).

O lema 5.2 implica que, no final do algoritmo, quanto mais espalhados pelos componentes de F estiverem os vértices de uma k-árvore mínima, melhor será o candidato a uma solução encontrado pelo algoritmo. No entanto, se os vértices de uma k-árvore mínima estiverem concentrados em poucos componentes de F, então o candidato a uma solução pode não ser muito bom.

5.2 k-DIJKSTRA

Um dos problemas mais conhecidos envolvendo grafos é o de encontrar caminhos de custo mínimo (CCM). Um caminho de um vértice v até um vértice u é de custo mínimo se a soma dos custos de suas arestas é a menor possível, dentre todos os caminhos de v até u.

Problema CCM(V, E, c, r): *dados um grafo conexo* (V, E)*, um custo não-negativo* c_e *para cada aresta* e *em* E *e um vértice* r *em* V*, encontrar os caminhos de custo mínimo de* r *para os demais vértices de* V*.*

Um fato importante é que toda solução do problema CCM pode ser representada por uma árvore. Chamamos essa árvore de **árvore de caminhos mínimos**. Essa árvore tem r como raiz e todo caminho, na árvore, de r para qualquer outro vértice em V é um caminho de custo mínimo no grafo original.

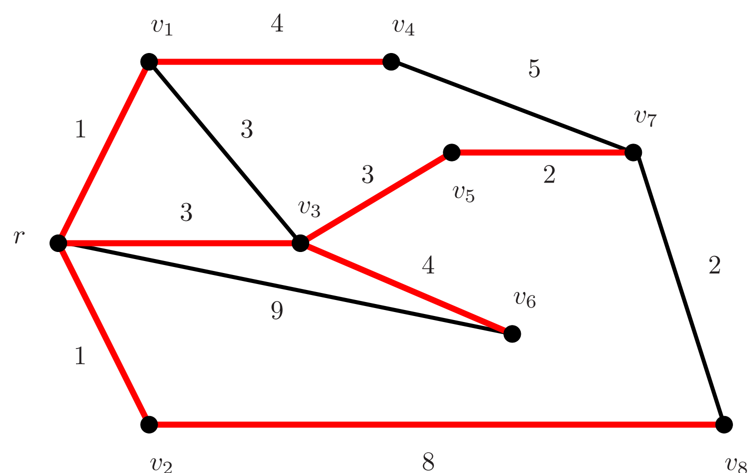


Figura 5.3: Exemplos de uma árvore de caminhos mínimos. As distâncias de r a $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$ são respectivamente 1, 1, 3, 5, 6, 7, 8, 9.

A princípio, esse problema não tem relação direta com o problema kMST. Mas, veremos como adaptar o algoritmo de Edsger Wybe Dijkstra [Dij71], que encontra caminhos de custo mínimo, em um algoritmo de aproximação para o kMST, que chamaremos de k-DIJKSTRA. Assim como o algoritmo de Dijkstra, o k-DIJKSTRA pode ser implementado com complexidade $\mathrm{O}((|V| + |E|) \log |V|)$.

O algoritmo k-DIJKSTRA funciona da seguinte maneira. A cada iteração temos uma árvore de caminhos mínimos com raiz r. Dentre os vértices fora dessa árvore, escolhemos o que tem menor distância de r, digamos v, e o adicionamos na árvore. Então, atualizamos, se for o caso, as distâncias dos vizinhos de v fora da árvore. A **distância** de um vértice v a r é o custo de um caminho de custo mínimo entre v e r.

Para decidir qual vértice deve ser inserido na árvore atual, o algoritmo utiliza uma estrutura de dados chamada fila de prioridade[1] . Dessa estrutura usaremos duas operações: CRIA-FILA(Q, X, f) e EXTRAI-MIN(Q). A primeira, cria uma fila de prioridade Q contendo X e associando o valor $f(x)$, para cada elemento x em X. A segunda devolve, removendo da fila, um elemento que tem menor valor de f.

O algoritmo k-DIJKSTRA recebe um grafo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$, um número inteiro k e um vértice r em V. Devolve uma árvore de caminhos mínimos T com raiz r contendo pelo menos k vértices. Note que T é uma k-árvore.

Algoritmo k-DIJKSTRA (V, E, c, k, r)

1 $T \leftarrow (\{r\}, \emptyset)$
2 $d(r) \leftarrow 0$
3 para cada $v \in V \setminus \{r\}$ faça
4   se $rv \in E$ então
5         $d(v) \leftarrow c_{rv}$
6         $p(v) \leftarrow r$
7   senão
8         $d(v) \leftarrow \infty$
9         $p(v) \leftarrow$ NIL
10 CRIA-FILA$(Q, V \setminus \{r\}, d)$
11 $v \leftarrow r$
12 enquanto T não é k-árvore faça
13     $v \leftarrow$ EXTRAI-MIN(Q)
14     $T \leftarrow T + p(v)v$

[1]Mais detalhes sobre filas de prioridade podem ser encontrados no livro de Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest e Clifford Stein [CLRS01].

```
15          para cada $u \in V \setminus V_T$ com $vu \in E$ faça
16              se $d(u) > d(v) + c_{vu}$ então
17                  $d(u) \leftarrow d(v) + c_{vu}$
18                  $p(u) \leftarrow v$
19      devolva $T$
```

Teorema 5.3. O fator de aproximação do algoritmo k-DIJKSTRA é $k-1$.

Prova. Seja T a k-árvore devolvida pelo algoritmo. Sejam T^* uma k-árvore mínima e $\text{OPT} = c(T^*)$. É claro que a distância, em T^*, de r para os demais vértice de V_{T^*} não é maior que OPT. Então, como por construção T é uma árvore de caminhos mínimos, temos que a distância, em T, de r para os demais vértice de V_T também não é maior que OPT. Logo, cada aresta em T tem custo no máximo OPT.

Portanto, $c(T) = \sum_{e \in E_T} c_e \leq \sum_{e \in E_T} \text{OPT} = (k-1)\text{OPT}$. □

Um fato interessante que usaremos mais adiante é que o algoritmo k-DIJKSTRA pode facilmente ser adaptado para o caso com pesos nos vértices, visto na seção 3.3. Para isso, na linha 12, ao invés de verificar se T é uma k-árvore, basta verificar se T é uma k-árvore com pesos. E o fator de aproximação dado pelo teorema 5.3 continua valendo.

5.3 $2\sqrt{k}$-aproximação

Ramamurthy Ravi, Ravi Sundaram, Madhav Vishnu Marathe, Daniel J. Rosenkrantz e Sekharipuram S. Ravi [RSM+94] desenvolveram um algoritmo de aproximação para o kMST baseado no algoritmo k-KRUSKAL e k-DIJKSTRA. Esse é o primeiro algoritmo de aproximação para o kMST publicado.

Assim como no algoritmo k-KRUSKAL, crescemos uma floresta F adicionando, em cada iteração, uma aresta de menor custo possível até que algum componente de F tenha pelo menos k vértices. Deste componente obtemos uma k-árvore. Além disso, em cada iteração, após adicionarmos uma nova aresta na floresta, tentamos obter uma k-árvore conectando alguns componentes. Dessa forma, o algoritmo, possivelmente, obtém várias k-árvores e no final devolve a de menor custo.

O procedimento de conectar componentes da floresta recebe como parâmetros o grafo com custos nas arestas, a floresta F atual, o número inteiro k e um número t, funcionando da seguinte forma. Se não existir um conjunto de no máximo t componentes da floresta, totalizando pelo menos k vértices no conjunto, não devolvemos nada. Então suponha que tal conjunto existe.

Para facilitar o entendimento vamos considerar o **grafo dos componentes** da floresta. Os vértices desse grafo são os componentes da floresta. Para cada vértice desse grafo atribuímos um peso igual ao número de vértices do componente correspondente. O conjunto de arestas do grafo dos componentes da floresta é o conjunto de arestas que conectam componentes distintos da floresta. No caso de mais de uma aresta conectar os mesmos componentes consideramos apenas a de menor custo.

Para cada vértice C do grafo dos componentes de F calculamos o menor valor d_C. Esse valor d_C é tal que, dentre os vértices que estão no máximo a distância d_C de C, existe um conjunto $\mathcal{C}_C$ de no máximo t vértices cuja soma dos pesos é pelo menos k. O valor d_C está bem definido pela suposição feita anteriormente. Note que C pertence a $\mathcal{C}_C$. Seja C' o vértice cujo valor $d_{C'}$ é o menor possível. Então conectamos o componente de F correspondente a C' aos componentes correspondentes aos vértices em $\mathcal{C}_{C'}$, obtendo assim uma k-árvore.

O algoritmo RSMRR recebe um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_\geq$ e um número inteiro k e devolve uma k-árvore T do grafo.

Algoritmo RSMRR (V, E, c, k)

1 $F \leftarrow (V, \emptyset)$ $\quad \mathcal{T} \leftarrow \emptyset$
2 enquanto não existe uma k-árvore em F faça
3 seja e uma aresta com c_e mínimo e pontas em componentes distintos de F.
4 $F \leftarrow F + e$
5 $T_C \leftarrow$ CONECTA-COMPONENTES(V, E, F, c, k, t)
6 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T_C\}$
7 $T_K \leftarrow k$-ÁRVORE(F, k)
8 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T_K\}$
9 $T \leftarrow$ árvore de $\mathcal{T}$ com custo mínimo
10 devolva T

Uma maneira de implementar o procedimento CONECTA-COMPONENTES é utilizando o algoritmo k-DIJKSTRA da seção 5.2 para cada vértice do grafo dos componentes da floresta F. Uma adaptação que deve ser feita é verificar se para a raiz passada ao k-DIJKSTRA existe uma k-árvore com pesos com no máximo t vértices. Se não existir devolvemos uma árvore vazia. Dessa forma o algoritmo consome tempo $\mathrm{O}(|V|^2((|V|+|E|)\log|V|))$, pois o procedimento CONECTA-COMPONENTES é chamado no máximo $k \leq |V|$ vezes e em cada chamada executa o k-DIJKSTRA no máximo $|V|$ vezes.

Vamos supor que todas as k-árvores obtidas pelo algoritmo têm exatamente k vértices, pois, se alguma dessas k-árvores tiver mais do que k vértices, podemos remover folhas dela até que fique com exatamente k. A remoção de folhas não aumenta o custo da k-árvore.

Teorema 5.4. O fator de aproximação do algoritmo RSMRR é $t\mathrm{OPT} + \frac{k}{t}\mathrm{OPT}$, onde $t \leq k$ é um parâmetro do procedimento de conectar componentes.

Prova. Considere uma k-árvore mínima T^*, com custo OPT. Seja x o número de arestas de T^* que não estão em F no final do algoritmo. Note que o lema 5.2 também vale para o algoritmo RSMRR.

Se $x \geq t$, então pelo lema 5.2 temos que $c(T_K) \leq \frac{k}{x}\mathrm{OPT} \leq \frac{k}{t}\mathrm{OPT}$.

Se $x < t \leq k$, então em algum momento anterior tínhamos $x = t$, pois a cada iteração x pode diminuir de apenas uma unidade. Dessa forma, existem no máximo t componentes de F que contêm vértices de T^*. Logo, existe um conjunto de no máximo t componentes da floresta, totalizando pelo menos k vértices no conjunto.

Seja C um componente de F que contém vértices de T^*. O custo da k-árvore T_C obtida conectando componentes a C é a soma do custo de cada componente a ser conectado mais o custo para conectá-los.

Pelo lema 5.2, a soma dos custos dos componentes a serem conectados é menor ou igual a $\frac{k}{t}\mathrm{OPT}$. Ademais, temos que $d_C \leq \mathrm{OPT}$, pois usando apenas arestas de T^* podemos conectar C aos demais componentes que contêm vértices de T^*. Logo, $c(T_C) \leq t\mathrm{OPT} + \frac{k}{t}\mathrm{OPT}$.

Portanto, como devolvemos a k-árvore com o menor custo, dentre as obtidas, temos que o custo da k-árvore devolvida é menor ou igual a $t\mathrm{OPT} + \frac{k}{t}\mathrm{OPT}$. □

Corolário 5.5. O fator de aproximação do algoritmo RSMRR utilizando $t = \sqrt{k}$ é $2\sqrt{k}$.

Prova. Do teorema 5.4, temos que o fator de aproximação do algoritmo RSMRR utilizando $t = \sqrt{k}$ é $\sqrt{k}\text{OPT} + \frac{k}{\sqrt{k}}\text{OPT} = 2\sqrt{k}\text{OPT}$. □

5.4 O$((\log k)^2)$-aproximação

Como vimos, o fator de aproximação do algoritmo RSMRR é $\sqrt{k}$, enquanto o do algoritmo k-KRUSKAL é k. Apesar da melhora ser significativa, o fator de aproximação do algoritmo RSMRR não é considerado bom. Isso porque o valor da função $\sqrt{k}$ se afasta de 1 à medida que o valor de k aumenta e, claramente, quanto mais próximo de 1 melhor é o fator de aproximação.

Baruch Awerbuch, Yossi Azar, Avrim Blum e Santosh Vempala [AABV98] desenvolveram um algoritmo de aproximação para o kMST com fator de aproximação polilogarítmico. Esse fator de aproximação também tem o problema de ser uma função crescente, mas seu crescimento é menor que o da função $\sqrt{k}$.

O algoritmo AABV usa uma modificação do algoritmo k-KRUSKAL, que chamaremos de AABV-AUX, para encontrar uma $\frac{k}{4}$-árvore. Dessa forma, ainda ficam faltando $k' := \frac{3k}{4}$ vértices para completar k. Então novamente através do algoritmo AABV-AUX encontramos uma $\frac{k'}{4}$-árvore disjunta da anterior e ficam faltando $\frac{3k'}{4}$ vértices. Esse processo continua até que a soma da quantidade de vértices de cada árvore encontrada seja pelo menos k. No final conectamos as árvores para obter uma k-árvore.

Diferentemente do algoritmo k-KRUSKAL, que conecta componentes de F através de arestas, o algoritmo AABV-AUX conecta esses componentes através de caminhos de custo mínimo entre eles. A cada iteração, são escolhidos dois componentes de F que minimizam a função $\gamma : \mathcal{C} \to \mathbb{Q}$, onde $\mathcal{C}$ é o conjuntos dos atuais componentes de F. A função γ é definida da seguinte maneira:

$$\gamma(C, D) = \begin{cases} \frac{\text{dist}(C,D)}{\min(|V_C|,|V_D|)}, & \text{se } C, D \in \mathcal{C} \text{ e } C \neq D \\ \infty, & \text{caso contrário} \end{cases},$$

onde dist(C, D) é o custo de um menor caminho entre C e D. A idéia por trás do uso da função γ é tentar escolher componentes que não estão distantes e possuem quantidade grande de vértices. Observe que essa função muda a cada iteração.

O algoritmo AABV-AUX recebe como entrada um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$ e um número inteiro k e devolve uma $\frac{k}{4}$-árvore T do grafo.

Algoritmo AABV-AUX (V, E, c, k)

1 $F \leftarrow (V, \emptyset)$
2 enquanto não existe uma $\frac{k}{4}$-árvore em F faça
3 seja $\mathcal{C}$ o conjunto dos componentes de F e C_i, C_j componentes distintos de $\mathcal{C}$ tal que $\gamma(C_i, C_j) = \min_{C_p, C_q \in \mathcal{C}}(\gamma(C_p, Cq))$
4 seja P um caminho mínimo entre C_i, C_j
5 $F \leftarrow F + P$
6 $T \leftarrow k$-ÁRVORE$(F, \frac{k}{4})$
7 devolva T

Uma observação importante é que ao inserir caminhos em F (linha 5) podemos obter circuitos em F. No entanto, isso não é um problema, pois podemos considerar que o laço do algoritmo (linhas 2 a 5) serve para selecionar o conjunto de vértices da árvore a ser devolvida. Dado esse conjunto basta encontrar a árvore geradora mínima do subgrafo de (V, E) induzido por esse conjunto de vértices. Portanto, por conveniência, podemos supor que F sempre é uma floresta. O algoritmo AABV-AUX consome tempo $\mathrm{O}(k|V|^2((|V| + |E|) \log |V|)) = \mathrm{O}(|V|^3((|V| + |E|) \log |V|))$.

Awerbuch, Azar, Blum e Vempala [AABV98] provaram os seguintes lemas.

Lema 5.6. Em qualquer iteração do algoritmo AABV-AUX, se o maior valor usado da função $\gamma(\cdot, \cdot)$ até o momento foi $\gamma_{\max}$, então qualquer árvore de F com n' vértices terá custo no máximo $\gamma_{\max} n' \log_2 n'$.

Prova. Dada uma árvore T' de F com n' vértices, vamos atribuir créditos aos vértices de T' de forma que a soma dos créditos de todos os seus vértices será um limitante superior para o seu custo. Inicialmente todo vértice começa com crédito igual a zero.

As árvores de F são obtidas através da conexão de componentes. Logo o custo da árvore é igual ao custo de todas as conexões feitas para formá-la. Considere a conexão de dois componentes, digamos C_1 e C_2. Suponha $|V_{C_1}| \leq |V_{C_2}|$. Sabemos do algoritmo que C_1 e C_2 são conectados por um caminho de custo $\gamma(C_1, C_2)|V_{C_1}| \leq \gamma_{\max}|V_{C_1}|$. Então adicionamos $\gamma_{\max}$ aos créditos dos vértices de C_1.

Note que o número de vértices de um componente obtido aos conectar dois outros componentes é pelo menos o dobro do número de vértices do menor dos dois. Ademais, para cada conexão, adicionamos créditos apenas para vértices do menor componente. Logo, cada vértice de T' tem no máximo $\gamma_{\max} \log_2 n'$ de crédito.

Portanto, como T' tem n' vértices, o custo de T' é no máximo $\gamma_{\max} n' \log_2 n'$. □

Lema 5.7. O algoritmo AABV-AUX nunca usa um valor da função $\gamma(\cdot,\cdot)$ maior que $\frac{8\log_2 k}{k}$OPT, onde OPT é o custo de uma k-árvore mínima.

Prova. Para provar o lema vamos analisar uma iteração qualquer do algoritmo. Podemos supor que nesta iteração todos os componentes de F possuem menos do que $\frac{k}{4}$ vértices, caso contrário a última iteração já teria terminado.

Suponha por contradição que o valor da função $\gamma(\cdot,\cdot)$ utilizado na iteração atual seja maior que $r := \frac{8\log_2 k}{k}$OPT. Isso significa que não existem dois componentes de F, digamos C_1 e C_2 com $|V_{C_1}| \leq |V_{C_2}|$, tal que a distância entre eles seja menor que $r|V_{C_1}|$.

Fixe uma k-árvore mínima T^* e daqui em diante considere apenas os componentes de F cujo conjunto de vértices intersecta o conjunto de vértices de T^*.

Seja $\mathcal{W} := \{W_2, W_3, \ldots, W_{(\log_2 k)-1}\}$ uma partição do conjunto de componentes de F tal que um componente C está em W_i se e somente se $\frac{k}{2^{i+1}} \leq |V_C| < \frac{k}{2^i}$. Como

$$\sum_{i=2}^{\log_2 k} \frac{k}{2^i} < \frac{k}{2}$$

e o número total de vértices dos componentes de F que estamos considerando é pelo menos k, então, pelo princípio da casa dos pombos, temos que existe um W em $\mathcal{W}$ tal que $|W| \geq 2$.

Podemos afirmar que W contém pelo menos $\frac{k}{2\log_2 k}$ vértices. Isso porque mesmo no pior caso em que colocamos exatamente um componente em cada W_i ainda sobram pelo menos $\frac{k}{2}$ vértices em componentes que ainda não pertencem a nenhum W_i. Se distribuirmos esses vértices igualmente[2] entre todos os W_i temos que W recebe pelo menos $\frac{k}{2} \cdot \frac{1}{(\log_2 k)-1} > \frac{k}{2\log_2 k}$.

[2]Distribuir igualmente minimiza a maior quantidade recebida por uma parte de $\mathcal{W}$. Note que isso é inviável, servindo apenas para estimar quantidade de vértices.

Considere que os componentes que estão em W possuem uma quantidade de vértices entre s e $2s$. Então W possui pelo menos $\frac{k}{2\log_2 k} \cdot \frac{1}{2s} = \frac{k}{4s\log_2 k}$ componentes. Pela hipótese inicial, o custo para conectar dois componentes quaisquer em W é pelo menos rs. Assim, consideramos que cada componente em W paga pelo menos $\frac{rs}{2}$ por conexão.

Todos os componentes em W intersectam T^*, logo o custo de T^* é pelo menos o custo de conectar os componentes em W. Então OPT $\geq \frac{k}{4s\log_2 k} \cdot \frac{rs}{2} = \frac{kr}{8\log_2 k} =$ OPT, o que é uma contradição.

Portanto, o algoritmo AABV-AUX nunca usa um valor maior que $\frac{8\log_2 k}{k}$OPT para a função $\gamma(\cdot,\cdot)$. □

Teorema 5.8. O algoritmo AABV-AUX devolve uma $\frac{k}{4}$-árvore do grafo (V, E) com custo no máximo $4(\log_2 k)^2$OPT, onde OPT é o custo de uma k-árvore de custo mínimo.

Prova. A $\frac{k}{4}$-árvore devolvida pelo algoritmo é obtida ao conectar dois componentes de F com menos do que $\frac{k}{4}$ vértices cada. Logo, juntos esses componentes têm menos do que $\frac{k}{2}$ vértices. No entanto, como os componentes são conectados por um caminho podemos acabar com uma árvore que tem mais do que $\frac{k}{2}$ vértices.

Suponha que a árvore devolvida pelo algoritmo tenha no máximo $\frac{k}{2}$ vértices. Pelo lema 5.7 temos que o maior valor de $\gamma(\cdot,\cdot)$ usado pelo algoritmo é no máximo $\frac{8\log_2 k}{k}$OPT. Logo, segundo o lema 5.6, o custo da $\frac{k}{4}$-árvore devolvida pelo algoritmo é no máximo

$$\left(\frac{8\log_2 k}{k}\text{OPT}\right)\frac{k}{2}\log_2\frac{k}{2} = 4(\log_2 k)\left(\log_2\frac{k}{2}\right)\text{OPT} \leq 4(\log_2 k)^2\text{OPT}.$$

No caso em que a árvore devolvida tem mais do que $\frac{k}{2}$ vértices a mesma análise vale. Isso porque, pela análise do lema 5.6, quem paga o custo do caminho que vai conectar os componentes são os vértices do componente que tem menor quantidade de vértices. Logo, os vértices do caminho não entram na conta do custo da árvore devolvida. Portanto a mesma análise vale. □

Para descrever o algoritmo AABV vamos considerar a versão com raiz do kMST. Para a versão sem raiz basta utilizar o algoritmo n vezes, deixando um vértice diferente como raiz em cada vez.

O algoritmo AABV recebe um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$, um número inteiro k, um vértice r de V e uma distância d e devolve, se existir, uma k-árvore T do grafo tal que todo vértice de T tem distância menor que d de r.

Algoritmo AABV (V, E, c, k, r, d)

1 $V' \leftarrow \{v \in V \mid \text{dist}(v, r) \leq d\}$
2 se $|V'| < k$ então devolva $\emptyset$
3 $E' \leftarrow \{uv \in E \mid u, v \in V'\}$
4 considere o grafo (V', E')
5 $i \leftarrow 0$
6 enquanto $k > 0$ faça
7   $i \leftarrow i + 1$
8   $T_i \leftarrow$ AABV-AUX(V', E', c, k)
9   contraia T_i em (V', E')
10   $k \leftarrow k - |V_{T_i}|$
11 $T \leftarrow$ CONECTA$(r, T_1, T_2, \ldots, T_i)$
12 devolva T

Primeiramente o algoritmo remove do grafo todos os vértices cuja distância à raiz r é maior que d. Isso é feito para garantir que cada árvore encontrada pode ser conectada à raiz com um custo menor que d.

Após encontrar uma árvore T_i com AABV-AUX, o algoritmo faz uma operação que chamaremos de **contração de árvore** no grafo. Seja X o subconjunto de vértices de V que possuem um vizinho em T_i. Para todo u e v em X defina $\tilde{P_{uv}}$ como um caminho de menor custo de u a v que tem a forma $\langle uu', P_{u'v'}, v'v \rangle$, onde u' e v' são vértices de T_i e $P_{u'v'}$ é o caminho de u' a v' em T_i. Contrair a árvore T_i no grafo (V', E') consiste em inserir em E' arestas uv, para todo u e v em X distintos, com custo $c_{uv} = c(\tilde{P_{uv}})$, caso ainda não exista a aresta uv em E' ou a que existe tem custo maior. Depois de inserir as arestas removemos de (V', E') os vértices de T_i e as arestas incidentes a esses vértices.

No final, o algoritmo conecta a raiz r às árvores encontradas através de caminhos de custo mínimo. O consumo de tempo dominante será o das chamadas a AABV-AUX. Logo o algoritmo AABV consome tempo O$(|V|^4((|V| + |E|) \log |V|))$.

Teorema 5.9. O algoritmo AABV tem fator de aproximação $\mathrm{O}((\log k)^3)$ se $d = \mathrm{OPT}$.

Prova. Pelo teorema 5.8, sabemos que as árvores encontradas nas $\mathrm{O}(\log k)$ execuções do algoritmo AABV-AUX têm custo $\mathrm{O}((\log k)^2)\mathrm{OPT}$. Ademais, o custo para conectar essas árvores à raiz r é no máximo $d = \mathrm{OPT}$. Portanto o custo da árvore devolvida pelo algoritmo AABV é $\mathrm{O}((\log k)^3)\mathrm{OPT} + \mathrm{O}(\log k)\mathrm{OPT} = \mathrm{O}((\log k)^3)\mathrm{OPT}$. □

Um ponto importante sobre o teorema 5.9 é que ele supõe o conhecimento prévio do valor de OPT. O que fazemos na prática é executar o algoritmo no máximo $n - k$ vezes da seguinte forma. Sejam $d_1 \geq d_2 \geq \ldots \geq d_n$ as distâncias de cada vértice em V à raiz. Na i-ésima execução do algoritmo, $1 \leq i \leq n - k$, utilizamos $d = d_i$. Das k-árvores obtidas devolvemos a de menor custo. Isso funciona porque para algum d_i obteremos um V' igual ao que obteríamos se tomássemos $d = \mathrm{OPT}$.

Podemos modificar o algoritmo AABV para melhorar o fator de aproximação para $\mathrm{O}((\log k)^2)$. Para isso supomos novamente o conhecimento do valor OPT e usaremos como caixa-preta um algoritmo $(3, 6)$-aproximador.

Considere um grafo G com n vértices e com custos nas arestas. Sejam $\varepsilon > 0$ e $L_\varepsilon > 0$ números dados. Se existir em G uma árvore enraizada com pelo menos $(1 - \varepsilon)n$ vértices e custo no máximo L_ε, então, um algoritmo $(3, 6)$**-aproximador** encontra em G, em tempo polinomial, uma árvore com mesma rair, custo no máximo $6L_\varepsilon$ e com pelo menos $(1 - 3\varepsilon)n$ vértices.

O algoritmo de Michel Xavier Goemans e David Paul Williamson [GW95], que veremos na seção 6.2, tem essa propriedade.

A idéia é encontrar uma k-árvore com custo $\mathrm{O}((\log k)^2\mathrm{OPT})$ ou então uma $\frac{k}{4}$-árvore com custo $\mathrm{O}(\mathrm{OPT})$. No último caso, conseguimos uma $\frac{k}{4}$-árvore melhor do que a do teorema 5.8 e dessa forma removemos o $(\log k)^2$ do fator de aproximação do algoritmo AABV. Esse novo algoritmo funciona da seguinte maneira.

Executamos o algoritmo AABV até encontrar uma $\frac{15k}{16}$-árvore. Isso pode ser obtido com menos de 10 iterações do algoritmo AABV. Vamos considerar que essa árvore tem exatamente $\frac{15k}{16}$ vértices e seja S o conjunto desses vértices.

Utilizando um algoritmo $(3, 6)$-aproximador, que chamaremos de B, com $\varepsilon = \frac{3}{15}$, no subgrafo induzido pelo conjunto de vértices S. Fixe uma k-árvore mínima T^*. Se T^* contém pelo menos $(1 - \frac{3}{15})|S|$ vértices de S, então B encontra um caminho de com-

primento no máximo $6\mathrm{OPT}$ com pelo menos $(1-\frac{9}{15})\frac{15k}{16}=\frac{3k}{8}>\frac{k}{4}$ vértices. Como B é $(3,6)$-aproximador, uma árvore geradora mínima no grafo induzido pelos vértices desse caminho é uma $\frac{k}{4}$-árvore com custo $\mathrm{O}(\mathrm{OPT})$.

Se o caminho encontrado por B tem custo maior que $6\mathrm{OPT}$ ou ele não contém pelo menos $\frac{3k}{8}$ vértices, então sabemos que T^* contém menos do que $(1-\frac{3}{15})|S|$ vértices de S. Logo, T^* contém pelo menos $\frac{k}{4}$ vértices em $V\setminus S$. Então executamos o algoritmo AABV-AUX com argumento $\frac{k}{4}$ no subgrafo induzido por $V\setminus S$ para encontrar uma $\frac{k}{16}$-árvore. Conectando essa nova árvore à $\frac{15k}{16}$-árvore que já tínhamos, obtemos uma k-árvore de custo $\mathrm{O}((\log k)^2\mathrm{OPT})$. Isso porque o custo para conectar é no máximo OPT, a $\frac{k}{16}$-árvore tem custo $\mathrm{O}((\log k)^2\mathrm{OPT})$, pelo teorema 5.8, e a $\frac{15k}{16}$-árvore também tem custo $\mathrm{O}((\log k)^2\mathrm{OPT})$, pois foi obtida com menos de 10 iterações do algoritmo AABV-AUX.

CAPÍTULO 6

Algoritmos primal-duais

Os algoritmos de aproximação para o kMST com fator de aproximação constante, conhecidos atualmente, são primal-duais. Eles usam como subrotina um algoritmo de aproximação para o problema da árvore de Steiner com coleta de prêmios (*prize-collecting Steiner tree*), ou simplesmente PCST.

Neste capítulo, primeiramente introduziremos alguns conceitos necessários de programação linear e dualidade. Em seguida, enunciaremos alguns resultados sobre o problema PCST e um algoritmo de aproximação para ele, desenvolvido por Michel Xavier Goemans e David Paul Williamson [GW95, GW97]. Finalmente, apresentaremos um algoritmo com fator de aproximação 5 para o kMST.

6.1 Programação linear

Programação linear é uma ferramenta muito poderosa e amplamente usada em otimização combinatória. Começou como um ramo da matemática aplicada que estuda problemas de minimizar ou maximizar uma função linear, denominada **função objetivo**, sujeita a restrições lineares.

Sejam M e N conjuntos de índices, $\{M_1, M_2, M_3\}$ e $\{N_1, N_2, N_3\}$ partições de M e N respectivamente. Considere uma matriz real A indexada por $M \times N$, um vetor real b indexado por M e um vetor real c indexado por N.

Um **problema de programação linear** ou **programa linear** de minimização (ou, de forma análoga, de maximização) é formulado da seguinte maneira:

$$\begin{array}{llcll} \text{minimize} & cx & & & \\ \text{sujeito a} & (Ax)_i & \geq & b_i & \text{para cada } i \text{ em } M_1 \\ & (Ax)_i & = & b_i & \text{para cada } i \text{ em } M_2 \\ & (Ax)_i & \leq & b_i & \text{para cada } i \text{ em } M_3 \\ & x_j & \geq & 0 & \text{para cada } j \text{ em } N_1 \\ & x_j & \leq & 0 & \text{para cada } j \text{ em } N_3. \end{array} \tag{6.1}$$

No programa linear (6.1), cx corresponde à função objetivo e, para cada linha i da matriz A, a relação entre $(Ax)_i$ e b_i corresponde a uma restrição sobre o valor de x. Também temos restrições sobre o sinal das entradas de x.

Se existe algum x que satisfaz todas as restrições dizemos que ele é um **candidato a uma solução** e o problema é **viável**. Se tal x não existe dizemos que o problema é **inviável**, ou seja, não possui solução.

Existem algoritmos polinomiais para resolver programas lineares, como por exemplo o método dos elipsóides e o método dos pontos interiores. Há também o algoritmo simplex que, apesar de ser muito usado, não é polinomial, mas tem tempo esperado polinomial.

Em programas lineares para problemas de otimização combinatória é comum adicionarmos restrições que exigem que algumas variáveis sejam inteiras. Programas lineares com essas restrições adicionais são chamados **programas lineares inteiros**.

Uma formulação do problema kMST-R(V, E, c, k, r) como programa linear inteiro é a seguinte. Considere c como um vetor indexado por E. Tomaremos como variáveis vetores x e z, onde x é indexado por E e z é indexado por 2^V. Queremos que, para e em E, x_e tenha valor 1, se e pertencer à solução, e valor 0 caso contrário. Também queremos que z_S tenha valor 1, se S for o subconjunto de V contendo todos os vértices que não fazem parte da solução, e valor 0 em caso contrário.

Então, temos o seguinte:

$$
\begin{array}{lrcll}
\text{minimize} & cx & & & \\
\text{sujeito a} & \sum_{e\in\delta(S)} x_e + \sum_{X:S\subseteq X} z_X & \geq & 1 & \text{para cada } S \subseteq V \setminus \{r\} \\
& \sum_{S:S\subseteq V\setminus\{r\}} |S| z_S & \leq & n-k & \\
& x_e & \in & \{0,1\} & \text{para cada } e \in E \\
& z_S & \in & \{0,1\} & \text{para cada } S \subseteq V \setminus \{r\}.
\end{array}
\qquad (6.2)
$$

Na formulação acima, o primeiro tipo de restrição serve para garantir que todo candidato a uma solução seja um subgrafo conexo que contém r. Como o problema é de minimização, a função objetivo garante que esse subgrafo será uma árvore. O segundo tipo de restrição garante que todo candidato a uma solução tem pelo menos k vértices. Os dois últimos servem para garantir que o valor atribuído às variáveis sejam inteiros, mais especificamente, sejam 0 ou 1. Note que essa formulação é apenas uma possível formulação. Existem outras formulações igualmente corretas.

Restrições de integralidade tornam o problema difícil. De fato, é sabido que resolver um programa inteiro é um problema NP-difícil.

Relaxação linear e lagrangeana

Como resolver um programa inteiro é um problema NP-difícil, formular o kMST-R como um programa inteiro não é suficiente para se resolvê-lo eficientemente. No entanto, a formulação pode nos ajudar, pois podemos obter muitas informações dela. Por exemplo, podemos encontrar um limitante inferior (no caso de problema de minimização) para o valor da solução.

Um método para encontrar limitantes inferiores é através da relaxação do problema original. Isto é, transformar o problema original em um problema mais fácil cuja solução ótima não é superior ao do problema original.

Uma definição mais formal de relaxação é a seguinte. Considere um programa linear $\min\{cx : x \in X \subseteq \mathbb{R}^n\}$. Um programa linear $\min\{fx : x \in X' \subseteq \mathbb{R}^n\}$ é uma **relaxação** do anterior se $X \subseteq X'$ e $fx \leq cx$ para todo x em X.

Pode-se observar que se uma relaxação é inviável, então o problema original também é inviável.

Existem várias formas de se relaxar uma formulação de um problema. Consideraremos dois tipos de relaxação bem conhecidos: a relaxação linear e a lagrangeana.

A **relaxação linear** basicamente consiste em trocar as restrições de integralidade das variáveis por restrições de não-negatividade. Dessa forma todos os candidatos a uma solução do problema original continuam sendo candidatos a uma solução no problema relaxado.

Uma relaxação linear da formulação do kMST-R dada anteriormente é a seguinte.

$$\begin{array}{lrcll} \text{minimize} & cx & & & \\ \text{sujeito a} & \sum_{e\in\delta(S)} x_e + \sum_{X:S\subseteq X} z_X & \geq & 1 & \text{para cada } S \in V \setminus \{r\} \\ & \sum_{S:S\subseteq V\setminus\{r\}} |S| z_S & \leq & n-k & \\ & x_e & \geq & 0 & \text{para cada } e \in E \\ & z_S & \geq & 0 & \text{para cada } S \subseteq V \setminus \{r\}. \end{array} \tag{6.3}$$

Uma observação interessante é que se a relaxação linear admite solução inteira, então essa solução também é solução do problema original.

A **relaxação lagrangeana** consiste em remover uma das restrições do problemas e incorporá-la na função objetivo. Dessa forma, todos os candidatos a uma solução do problema original continuam sendo candidatos a uma solução no problema relaxado.

No caso da relaxação linear do kMST-R podemos aplicar a relaxação lagrangeana na restrição que exige pelo menos k vértices no candidato a uma solução.

$$\begin{array}{lrcll} \text{minimize} & cx + \lambda\big(\sum_{S:S\subseteq V\setminus\{r\}} |S| z_S & - & (n-k)\big) & \\ \text{sujeito a} & \sum_{e\in\delta(S)} x_e + \sum_{X:S\subseteq X} z_X & \geq & 1 & \text{para cada } S \subseteq V \setminus \{r\} \\ & x_e & \geq & 0 & \text{para cada } e \in E \\ & z_S & \geq & 0 & \text{para cada } S \subseteq V \setminus \{r\}. \end{array} \tag{6.4}$$

Nesse tipo de relaxação, além de manter os candidatos a uma solução do problema original, também penalizamos o valor dos novos candidatos a uma solução. Isso porque se o candidato a uma solução não satisfaz a restrição relaxada, então terá uma quantidade positiva adicionada a seu valor. O parâmetro λ na função objetivo serve para controlar essa penalidade. Claramente $\lambda \geq 0$.

Dualidade

Existe uma importante e muito explorada relação de dualidade entre programas lineares. O **dual** de um programa linear de minimização, como em (6.1) é um programa de maximização da seguinte forma.

$$\begin{array}{lrcll} \text{maximize} & yb & & & \\ \text{sujeito a} & (yA)_j & \leq & c_j & \text{para cada } j \in N_1 \\ & (yA)_j & = & c_j & \text{para cada } j \in N_2 \\ & (yA)_j & \geq & c_j & \text{para cada } j \in N_3 \\ & y_i & \geq & 0 & \text{para cada } i \in M_1 \\ & y_i & \leq & 0 & \text{para cada } i \in M_3. \end{array} \tag{6.5}$$

É comum chamar o programa do qual o dual se originou de **primal**.

O programa dual da relaxação lagrangeana (6.4) vista na seção anterior é o seguinte.

$$\begin{array}{lrcll} \text{maximize} & \sum_{S \subseteq V \setminus \{r\}} y_s & - & (n-k)\lambda & \\ \text{sujeito a} & \sum_{S : e \in \delta(S)} y_S & \leq & c_e & \text{para cada } e \text{ em } E \\ & \sum_{X : X \subseteq S} y_X & \leq & |S|\lambda & \text{para cada } S \subseteq V \setminus \{r\} \\ & y_S & \geq & 0 & \text{para cada } S \subseteq V \setminus \{r\}. \end{array} \tag{6.6}$$

Existe uma relação fundamental entre os candidatos a uma solução de um problema primal e os candidatos a uma solução do seu dual. Essa relação 6.1 é muitas vezes chamada de **dualidade fraca**.

Lema 6.1. Para todo candidato a uma solução x de um problema primal e todo candidato a uma solução y do problema dual, vale que $cx \geq yb$.

Prova. Podemos supor sem perda de generalidade que $M = M_1$ e $N = N_1$, pois, no caso do problema primal (no dual é análogo), podemos expressar as restrições do tipo $(Ax)_i = b_i$ como $(Ax)_i \geq b_i$ e $(Ax)_i \leq b_i$. As restrições do tipo $(Ax)_i \leq b_i$ podem ser expressas como $(-Ax)_i \geq -b_i$. O mesmo vale para as restrições de sinal de x.

Então, dos programas primal e dual, temos que $Ax \geq b$ e $yA \leq c$. Portanto, vale que $cx \geq (yA)x = y(Ax) \geq yb$. □

6.2 Árvore de Steiner com coleta de prêmios

O problema da árvore de Steiner (ST) visto na seção 3.4 consiste em encontrar, num grafo conexo, uma árvore de Steiner com número mínimo de arestas. Se o grafo tem custos nas arestas, podemos generalizar esse problema minimizando a soma dos custos das arestas da árvore de Steiner, em vez de minimizar o número de arestas.

A seguir veremos uma variante mais geral, chamada de problema da árvore de Steiner com coleta de prêmios (PCST). Nessa variante, consideramos penalidades nos vértices, que devemos pagar caso o vértice não esteja na árvore de Steiner.

Problema PCST(V, E, c, π, r): *dados um grafo conexo* (V, E)*, um custo não-negativo* c_e *para cada aresta* e *em* E*, uma penalidade não-negativa* π_v *para cada vértice* v *em* V *e um vértice* r *de* V*, encontrar uma árvore que minimize a soma do custo de suas arestas mais a soma das penalidades dos vértices fora dela.*

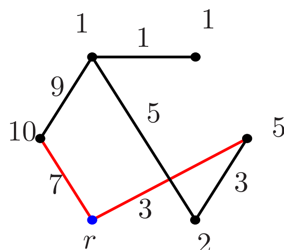


Figura 6.1: Uma solução para o PCST de valor 14.

Para verificar que o problema ST é um caso particular do PCST basta considerar custo 1 para todas as arestas do grafo. Atribua penalidade com valor maior ou igual ao número de arestas do grafo, para vértices terminais, e 0 para os demais. A penalidade alta dos vértices terminais faz com que toda solução os contenha.

Disso obtemos uma observação interessante sobre as soluções do PCST. Quanto maior o valor da penalidade dos vértices, mais vértices terá a solução. Em particular, se a penalidade nos vértices for 0, então a solução será a árvore que contém apenas r. Se as penalidades forem suficientemente grandes (por exemplo, maiores que a soma dos custos de todas as arestas), então toda solução será uma árvore geradora mínima.

Uma formulação como programa inteiro para o PCST é dada a seguir. Seja c o

vetor que representa os custos das arestas e π o vetor que representa as penalidades dos vértices. Tomaremos como variáveis os vetores x, indexado por E, e z, indexado por 2^V. Queremos que x_e, para e em E, tenha valor 1 se e pertencer à solução, e valor 0 em caso contrário. E que z_S tenha valor 1, se S for o subconjunto de V contendo todos os vértices que não fazem parte da solução, e valor 0 em caso contrário. Denotaremos por $\pi(S) = \sum_{v \in S} \pi_v$ e $\delta(S)$ o corte de S, onde S é um subconjunto de V.

$$
\begin{array}{lrcll}
\text{minimize} & cx + \displaystyle\sum_{S \subseteq V \setminus \{r\}} \pi(S) z_S & & & \\
\text{sujeito a} & \displaystyle\sum_{e \in \delta(S)} x_e + \sum_{X : S \subseteq X} z_X & \geq & 1 & \text{para cada } S \subseteq V \setminus \{r\} \\
& x_e & \in & \{0,1\} & \text{para cada } e \in E \\
& z_S & \in & \{0,1\} & \text{para cada } S \subseteq V \setminus \{r\}.
\end{array}
\tag{6.7}
$$

O dual para a relaxação linear de (6.7) é

$$
\begin{array}{lrcll}
\text{maximize} & \displaystyle\sum_{S \subseteq V \setminus \{r\}} y_S & & & \\
\text{sujeito a} & \displaystyle\sum_{S \subseteq V \setminus \{r\} : e \in \delta(S)} y_S & \leq & c_e & \text{para cada } e \in E \\
& \displaystyle\sum_{X : X \subseteq S} y_X & \leq & \pi(S) & \text{para cada } S \subseteq V \setminus \{r\} \\
& y_S & \geq & 0 & \text{para cada } S \subseteq V \setminus \{r\}.
\end{array}
\tag{6.8}
$$

Um algoritmo para o PCST que denotaremos por GW(V, E, c, π, r), ou simplesmente GW, foi criado por Goemans e Williamson [GW95] por volta de 1995. Ele devolve (T, A), onde $T = (V_T, E_T)$ é um candidato a solução do PCST e A é o conjunto dos vértices que não fazem parte de T. Também é devolvido um candidato a solução y para o dual (6.8). Esse algoritmo não será muito discutido neste texto. Iremos apenas usá-lo como "caixa preta".

Pela dualidade fraca e o fato de y ser um candidato a solução de (6.8), sabemos que $\sum_{S \subseteq V \setminus \{r\}} y_S$ é um limitante inferior para o valor de uma solução do PCST. Goemans e Williamson [GW95] provaram o seguinte teorema sobre o algoritmo GW.

Teorema 6.2. O candidato a uma solução do PCST $((V_T, E_T), A)$ e o seu dual, y devolvidos pelo algoritmo GW(V, E, c, π, r), satisfazem a seguinte relação.

$$\sum_{e \in E_T} c_e + \left(2 - \frac{1}{|V|-1}\right)\pi(A) \leq \left(2 - \frac{1}{|V|-1}\right) \sum_{S \subseteq V \setminus \{r\}} y_S.$$

Corolário 6.3. Suponha que $\pi_v = \lambda$ para todo $v \in V$, onde $\lambda \geq 0$ é uma constante. Então para $((V_T, E_T), A)$ e y obtidos de GW(V, E, c, π, r), vale o seguinte.

$$\sum_{e \in E_T} c_e + 2|A|\lambda \leq 2 \sum_{S \subseteq V \setminus \{r\}} y_S.$$

Prova. Segue direto do teorema 6.2, pois $\pi(A) = \sum_{v \in A} \pi_v = \sum_{v \in A} \lambda = |A|\lambda$. Logo

$$\sum_{e \in E_T} c_e + \left(2 - \frac{1}{|V|-1}\right)|A|\lambda \leq \left(2 - \frac{1}{|V|-1}\right) \sum_{S \subseteq V \setminus \{r\}} y_S \iff$$
$$\sum_{e \in E_T} c_e \leq \left(2 - \frac{1}{|V|-1}\right)\left(\sum_{S \subseteq V \setminus \{r\}} y_S - |A|\lambda\right) \iff$$
$$\sum_{e \in E_T} c_e \leq 2\left(\sum_{S \subseteq V \setminus \{r\}} y_S - |A|\lambda\right) \iff$$
$$\sum_{e \in E_T} c_e + 2|A|\lambda \leq 2 \sum_{S \subseteq V \setminus \{r\}} y_S.$$

□

Vimos na seção 5.4 a definição de um algoritmo $(3, 6)$-aproximador. Veremos agora uma generalização dessa definição. Considere um grafo G com n vértices e custos nas arestas. Sejam $\varepsilon > 0$ e $L_\varepsilon > 0$ números dados. Se existir em G uma árvore enraizada com pelo menos $(1 - \varepsilon)n$ vértices e custo no máximo L_ε, então, um algoritmo (a, b)-**aproximador** encontra em G, em tempo polinomial, uma árvore com mesma raiz, de custo no máximo bL_ε e com pelo menos $(1 - a\varepsilon)n$ vértices.

O algoritmo GW é um exemplo de $(3, 6)$-aproximador [GW95]. Um resultado melhor foi obtido por Michel Xavier Goemans e Jon Michael Kleinberg [GK98], provando que o algoritmo GW é um $(2, 4)$-aproximador.

A seguir mostramos um pseudo-código para o algoritmo GW. O algoritmo recebe um grafo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$, um função penalidade $\pi : V \to \mathbb{Q}_{\geq}$ e um vértice raiz r de V. Como saída, o algoritmo devolve uma árvore $T := (V_T, E_T)$ que contém r, um subconjunto A de vértices de V que não estão em T e um vetor y que satisfaz as restrições do programa dual do PCST.

Algoritmo GW (V, E, c, π, r)

1. $V_T \leftarrow \emptyset$ $E_T \leftarrow \emptyset$ $\mathcal{C} \leftarrow \{\{v\} \mid v \in V\}$
2. $y_S \leftarrow 0$ para todo $S \subset V$
3. para cada $v \in V$ faça
4. inicialmente v não possui nenhum rótulo
5. $d(v) \leftarrow 0$
6. $w(\{v\}) \leftarrow 0$
7. se $v = r$ então $\lambda(\{v\}) \leftarrow 0$ senão $\lambda(\{v\}) \leftarrow 1$
8. enquanto existe $C \in \mathcal{C}$ com $\lambda(C) = 1$ faça
9. encontre aresta $e = (i, j)$ com $i \in C_p \in \mathcal{C}$, $j \in C_q \in \mathcal{C}$, $p \neq q$ e que minimiza $\varepsilon_1 = \frac{c_e - d(i) - d(j)}{\lambda(C_p) + \lambda(C_q)}$
10. encontre $\widetilde{C} \in \mathcal{C}$ com $\lambda(\widetilde{C}) = 1$ que minimiza $\varepsilon_2 = \sum_{i \in \widetilde{C}} \pi_i - w(\widetilde{C})$
11. $\varepsilon \leftarrow \min(\varepsilon_1, \varepsilon_2)$
12. $w(C) \leftarrow w(C) + \varepsilon\lambda(C)$ para todo $C \in \mathcal{C}$
13. $y_C \leftarrow y_C + \varepsilon\lambda(C)$ para todo $C \in \mathcal{C}$
14. para todo $v \in C_r \in \mathcal{C}$, onde $r \in C_r$, faça $d(v) \leftarrow d(v) + \varepsilon\lambda(C_r)$
15. se $\varepsilon = \varepsilon_2$ então
16. $\lambda(\widetilde{C}) \leftarrow 0$
17. rotule todos os vértices de $\widetilde{C}$ que não têm rótulo com o rótulo $\widetilde{C}$
18. senão
19. $E_T \leftarrow E_T \cup \{e\}$ $V_T \leftarrow V_T \cup \{i, j\}$
20. $\mathcal{C} \leftarrow (\mathcal{C} \cup \{C_p \cup C_q\}) \setminus \{C_p, C_q\}$
21. $w(C_p \cup C_q) \leftarrow w(C_p) + w(C_q)$
22. se $r \in C_p \cup C_q$ então $\lambda(C_p \cup C_q) \leftarrow 0$ senão $\lambda(C_p \cup C_q) \leftarrow 1$

23 remova o máximo de arestas e vértices de (V_T, E_T) mantendo as seguintes propriedades: (1) existe um caminho de todo vértice sem rótulo até r; (2) se existe um caminho de um vértice v com rótulo C até r, então também existe um caminho de todo vértice com rótulo $C' \supseteq C$ até r

24 $A \leftarrow V \setminus V_T$

25 devolva $(V_T, E_T), A, y$

6.3 5-aproximação de Garg

Naveen Garg [Gar96] obteve uma 5-aproximação para o kMST usando o algoritmo GW como subrotina. A idéia de usar um algoritmo para o PCST está no fato das relaxações lineares dos programas inteiros para o kMST e para o PCST serem similares. Para facilitar o entendimento, vamos considerar aqui a versão enraizada, kMST-R.

Uma formulação do problema kMST-R(V, E, c, k, r) é a seguinte. Considere c como um vetor indexado por E. Tomaremos como variáveis os vetores x, indexado por E, e z, indexado por 2^V. Queremos que x_e, para e em E, tenha valor 1 se e pertencer à solução e valor 0 em caso contrário. E que z_S tenha valor 1, se S for o subconjunto de V contendo todos os vértices que não fazem parte da solução, e valor 0 caso contrário. A formulação abaixo é uma relaxação linear do programa inteiro.

$$
\begin{array}{lrcll}
\text{minimize} & cx & & & \\
\text{sujeito a} & \displaystyle\sum_{e \in \delta(S)} x_e + \sum_{X : S \subseteq X} z_X & \geq & 1 & \text{para cada } S \subseteq V \setminus \{r\} \\
& \displaystyle\sum_{S : S \subseteq V \setminus \{r\}} |S| z_S & \leq & n - k & \\
& x_e & \geq & 0 & \text{para cada } e \in E \\
& z_S & \geq & 0 & \text{para cada } S \subseteq V \setminus \{r\}.
\end{array}
\tag{6.9}
$$

Na formulação acima, o primeiro tipo de restrição serve para garantir que todo candidato a uma solução seja um subgrafo conexo que contém r. A minimização da função objetivo garante que esse subgrafo seja uma árvore. O segundo tipo de restrição garante que todo candidato a uma solução tem pelo menos k vértices. Note que essa formulação é apenas uma dentre várias possíveis. Existem outras formulações igualmente corretas.

Uma relaxação linear para o PCST é a seguinte.

$$\begin{array}{lrcll} \text{minimize} & cx + \displaystyle\sum_{S\subseteq V\setminus\{r\}} \pi(S) z_S & & & \\ \text{sujeito a} & \displaystyle\sum_{e\in\delta(S)} x_e + \sum_{X:S\subseteq X} z_X & \geq & 1 & \text{para cada } S\subseteq V\setminus\{r\} \\ & x_e & \geq & 0 & \text{para cada } e\in E \\ & z_S & \geq & 0 & \text{para cada } S\subseteq V\setminus\{r\}. \end{array} \tag{6.10}$$

Note que (6.9) e (6.10) diferem apenas na função objetivo e em uma restrição a mais que aparece na formulação do kMST-R. Dessa forma, se usarmos a relaxação lagrangeana nessa restrição extra obtemos (6.11).

$$\begin{array}{lrcll} \text{minimize} & cx + \lambda\left(\displaystyle\sum_{S:S\subseteq V\setminus\{r\}} |S| z_S \;-\; (n-k)\right) & & & \\ \text{sujeito a} & \displaystyle\sum_{e\in\delta(S)} x_e + \sum_{X:S\subseteq X} z_X & \geq & 1 & \text{para cada } S\subseteq V\setminus\{r\} \\ & x_e & \geq & 0 & \text{para cada } e\in E \\ & z_S & \geq & 0 & \text{para cada } S\subseteq V\setminus\{r\}. \end{array} \tag{6.11}$$

Agora (6.10) e (6.11) diferem apenas na função objetivo, tendo o mesmo conjunto de candidatos a uma solução. Logo, queremos interpretar o candidato a uma solução T para o PCST, devolvido pelo algoritmo GW, como um candidato a uma solução para o kMST-R. O algoritmo GW também devolve um candidato a uma solução y para o dual, do qual podemos extrair um limitante inferior para o valor de uma solução para o kMST-R.

Para entender isso melhor, considere o dual de (6.11).

$$\begin{array}{lrcll} \text{maximize} & \displaystyle\sum_{S\subseteq V\setminus\{r\}} y_S & - & (n-k)\lambda & \\ \text{sujeito a} & \displaystyle\sum_{S\subseteq V\setminus\{r\}:e\in\delta(S)} y_S & \leq & c_e & \text{para cada } e \text{ em } E \\ & \displaystyle\sum_{X:X\subseteq S} y_X & \leq & |S|\lambda & \text{para cada } S\subseteq V\setminus\{r\} \\ & y_S & \geq & 0 & \text{para cada } S\subseteq V\setminus\{r\}. \end{array} \tag{6.12}$$

Considerando $\pi_v = \lambda$ para todo vértice v em V, temos que (6.8) e (6.12) também diferem apenas na função objetivo. Logo, o y devolvido por GW também é um candidato a uma solução de (6.12). Pela dualidade fraca, sabemos que o valor do candidato a uma solução y não será maior do que o valor de uma solução do kMST-R.

Sejam (V, E, c, k, r) uma instância do kMST-R e OPT o valor de uma solução para essa instância. Faremos o seguinte abuso de notação: denotaremos por λ tanto uma função de penalidade constante nos vértices quanto o valor dessa função, ou seja, todo vértice tem penalidade λ.

Lema 6.4. Se para uma instância (V, E, c, k, r) do kMST-R e um dado λ, o algoritmo GW devolve $((V_T, E_T), A)$ e y, então

$$\sum_{e \in E_T} c_e + 2\lambda(|A| - (n-k)) \leq 2\left(\sum_{S \subseteq V \setminus \{r\}} y_S - \lambda(n-k)\right) \leq 2\text{OPT}.$$

Prova. Do corolário 6.3, temos que

$$\begin{aligned}
&\sum_{e \in E_T} c_e + 2|A|\lambda \leq 2 \sum_{S \subseteq V \setminus \{r\}} y_S \iff \\
&\sum_{e \in E_T} c_e + 2|A|\lambda - 2(n-k)\lambda \leq 2 \sum_{S \subseteq V \setminus \{r\}} y_S - 2(n-k)\lambda \iff \\
&\sum_{e \in E_T} c_e + 2\lambda(|A| - (n-k)) \leq 2\left(\sum_{S \subseteq V \setminus \{r\}} y_S - \lambda(n-k)\right).
\end{aligned}$$

Pela dualidade fraca e por y ser um candidato a uma solução de (6.12), temos que

$$2\left(\sum_{S \subseteq V \setminus \{r\}} y_S - \lambda(n-k)\right) \leq 2\text{OPT}_L,$$

onde OPT_L é o valor de uma solução de (6.11). Mas por se tratar de uma relaxação, temos que $\text{OPT}_L \leq \text{OPT}$. Portanto as desigualdades do lema são verdadeiras. □

Se por acaso o candidato a uma solução $((V_T, E_T), A)$, devolvido por GW, for tal que $|A| = n - k$, isto é, V_T possuir exatamente k vértices, então o lema 6.4 garante que (V_T, E_T) é uma 2-aproximação para o kMST-R. No entanto, se $|A| < n - k$, o lema 6.4 não fornece nenhuma informação sobre limitantes. O caso $|A| > n - k$ não ocorre, pois (V_T, E_T) é um candidato a uma solução. Logo $|A| \leq n - k$.

Baseando-se nessas idéias e no fato de que as penalidades nos vértices influenciam a quantidade de vértices de uma solução do PCST, o algoritmo proposto por Garg tenta encontrar um valor suficientemente bom para a penalidade nos vértices de forma a garantir que o candidato a uma solução seja uma 5-aproximação.

Consideraremos uma instância do kMST-R (V, E, c, k, r) e o valor OPT de uma solução para essa instância. Para facilitar a explicação do algoritmo, vamos supor algumas hipóteses sobre a instância do kMST-R, sem perda de generalidade.

1. Os custos das arestas satisfazem a desigualdade triangular.
   Suponha que os custos das arestas não satisfazem a desigualdade triangular. Então existe no grafo um triângulo formado por arestas v_1v_2, v_2v_3 e v_3v_1, tal que $c_{v_1v_2} > c_{v_2v_3} + c_{v_3v_1}$. Uma solução (V_T, E_T) do kMST-R nunca conteria a aresta v_1v_2, caso contrário $(V_T \cup \{v_3\}, (E_T \setminus \{v_1v_2\}) \cup \{v_2v_3, v_3v_1\})$ seria um subgrafo contendo uma k-árvore de custo menor. Isso contradiz a minimalidade do valor de (V_T, E_T), logo v_1v_2 não faz parte da solução. Portanto podemos supor sem perda de generalidade que os custos das arestas satisfazem a desigualdade triangular.

2. A distância entre um vértice v qualquer e r é no máximo OPT.
   Caso esta hipótese seja falsa, poderíamos fazer o seguinte. Seja v um vértice e d_v a distância de v a r. Removemos todos os vértices cuja distância a r é maior que d_v. Aplicamos o algoritmo no grafo resultante. Repetimos o processo para cada vértice diferente de r e no final devolvemos, dentre as possíveis $n-1$ k-árvores encontradas, uma de menor custo. Certamente, em algum momento, encontramos uma k-árvore, pois se v é um vértice mais distante de r em uma solução, temos que $d_v \leq$ OPT. Este processo exige no máximo $n-1$ execuções do algoritmo.

3. OPT $\geq c_{\min}$, onde $c_{\min}$ é o menor custo não-nulo de uma aresta em E.
   Se essa hipótese não valer, então a solução tem custo zero e esse caso poderia ser encontrado através de um pré-processamento de tempo polinomial.

Como dito anteriormente, o algoritmo tentará encontrar um valor λ suficientemente bom para as penalidades nos vértices, através do algoritmo GW(V, E, c, λ, r). A busca é feita usando a idéia da busca binária com várias chamadas ao algoritmo GW(V, E, c, λ, r), cada vez com uma constante λ diferente para as penalidades.

Como observado na seção 6.2, se tomarmos $\lambda_1 = 0$ como penalidade para cada vértice, a solução para o PCST será a árvore $(\{r\}, \emptyset)$. Se tomarmos $\lambda_2 = \sum_{e \in E} c_e$ como penalidade para cada vértice, a solução será uma árvore geradora mínima. Portanto a busca pelo valor de λ começará no intervalo $[\lambda_1, \lambda_2]$.

Primeiro tomamos $\lambda = \frac{\lambda_1 + \lambda_2}{2}$ e executamos GW(V, E, c, λ, r). Se a árvore obtida possuir menos do que k vértices, atualizamos λ_1 com o valor de λ e repetimos a busca. Se a árvore obtida possuir mais do que k vértices, atualizamos λ_2 com o valor de λ e repetimos a busca. E caso a árvore obtida tenha exatamente k vértices, paramos e a devolvemos. A condição de parada para a busca é satisfeita quando encontramos uma árvore com exatamente k vértices ou obtemos $\lambda_1 < \lambda_2$ tais que

$$\lambda_2 - \lambda_1 \leq \frac{c_{\min}}{2n(2n+1)}.$$

Se encontramos uma árvore com exatamente k vértices, então o lema 6.4 garante que tal árvore é uma 2-aproximação para o kMST-R. Se a busca parar sem encontrar uma árvore com exatamente k vértices, então o algoritmo combinará (T_1, A_1, y^1) e (T_2, A_2, y^2) para obter uma 5-aproximação, onde (T_i, A_i, y^i) são os candidatos a uma solução devolvidos por GW(V, E, c, λ_i, r).

O algoritmo GARG5 recebe um grafo conexo (V, E), uma função custo $c : E \to \mathbb{Q}_{\geq}$, um número inteiro k e uma raiz r de V e devolve uma k-árvore T do grafo.

Algoritmo GARG5(V, E, c, k, r)

1 $\lambda_1 \leftarrow 0$ $\quad \lambda_2 \leftarrow \sum c_e$
2 enquanto a condição de parada não é satisfeita faça
3 $\quad \lambda \leftarrow \frac{\lambda_1 + \lambda_2}{2}$
4 $\quad (T, A, y) \leftarrow$ GW(V, E, c, λ, r)
5 $\quad$ se T tem exatamente k vértices então devolva T
6 $\quad$ se T tem mais do que k vértices então $\lambda_2 \leftarrow \lambda$
7 $\quad$ senão $\lambda_1 \leftarrow \lambda$
8 $(T_1, A_1, y^1) \leftarrow$ GW(V, E, c, λ_1, r)
9 $(T_2, A_2, y^2) \leftarrow$ GW(V, E, c, λ_2, r)
10 devolva COMBINAÇÃO(T_1, T_2)

A combinação de T_1 com T_2 funciona da seguinte maneira. Sejam k_1 e k_2 o número de vértices de T_1 e T_2 respectivamente. Do algoritmo GARG5(V, E, c, k, r) sabemos que $k_1 < k < k_2$. Se sabemos que T_2 é uma 5-aproximação para o kMST-R, então podemos devolvê-la. Caso contrário, adicionamos alguns vértices de T_2 a T_1 para obter uma k-árvore. Para isso, primeiro duplicamos as arestas de T_2 (figura 6.2). Dessa forma todos os vértices de T_2 têm grau par e, assim, T_2 tem uma trilha euleriana fechada. Usando a técnica dos atalhos (*shortcuts*), bastante usada em algoritmos de aproximação para o problema do caixeiro viajante [Vaz01], podemos substituir T_2 por um circuito contendo apenas os vértices de T_2 que não estão em T_1. Seja P um caminho nesse circuito com $k - k_1$ vértices e com o menor custo possível. Conectamos P a r através de um caminho de menor custo. Assim, obtemos uma árvore com pelo menos k vértices.

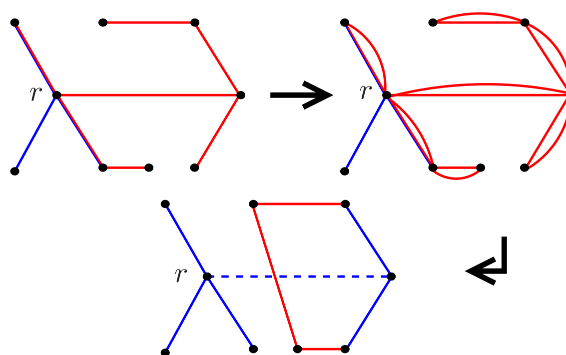


Figura 6.2: Ilustração da combinação de árvores. Arestas escuras representam T_1, arestas claras representam T_2 e temos $k = 7$, $k_1 = 4$ e $k_2 = 8$.

Teorema 6.5. O fator de aproximação do algoritmo GARG5 é 5.

Para verificar que o algoritmo consome tempo polinomial, basta analisar todas as chamadas ao algoritmo GW e a combinação das árvores, pois as demais operações consomem tempo constante.

Considere que a busca termina quando a condição de parada é satisfeita. Como se trata de uma busca binária, o número de iterações é $\mathrm{O}\left(\log \frac{n^2 \sum_{e \in E} c_e}{c_{\min}}\right)$. Ademais, o algoritmo GW, chamado em cada iteração da busca, é polinomial. Logo, a busca consome tempo polinomial. A combinação das árvores também consome tempo polinomial, pois duplicar arestas, a técnica dos atalhos e conectar P a r por um caminho de menor custo podem ser feitos em tempo polinomial.

Agora, em relação à qualidade do candidato a uma solução devolvido, já sabemos que se encontramos uma árvore com exatamente k vértices durante a busca, então essa árvore é uma 2-aproximação para o kMST-R. Então vamos supor que não encontramos tal árvore, ou seja, a árvore devolvida é obtida da combinação de $T_1 = (V_{T_1}, E_{T_1})$ e $T_2 = (V_{T_2}, E_{T_2})$.

Do teorema 6.2 temos que

$$\sum_{e \in E_{T_1}} c_e \leq \left(2 - \frac{1}{n}\right)\left(\sum_{S \subseteq V \setminus \{r\}} y_S^1 - |A_1|\lambda_1\right) \tag{6.13}$$

$$\sum_{e \in E_{T_2}} c_e \leq \left(2 - \frac{1}{n}\right)\left(\sum_{S \subseteq V \setminus \{r\}} y_S^2 - |A_2|\lambda_2\right). \tag{6.14}$$

Vamos analisar a combinação convexa dessas duas desigualdades. Sejam α_1, α_2 números positivos tais que $\alpha_1 + \alpha_2 = 1$ e $\alpha_1 k_1 + \alpha_2 k_2 = k$. Então

$$\alpha_1 = \frac{k - k_2}{k_1 - k_2} \text{ e} \tag{6.15}$$

$$\alpha_2 = \frac{k - k_1}{k_2 - k_1}. \tag{6.16}$$

É possível obter um limitante, dado pelo lema a seguir, para o custo da combinação convexa de T_1 e T_2. Fabián Ariel Chudak, Tim Roughgarden e David Paul Williamson [CRW04] provaram o seguinte lema.

Lema 6.6. Considere que o algoritmo GW não encontra uma k-árvore com exatamanete k vértices. Então,

$$\alpha_1 \sum_{e \in E_{T_1}} c_e + \alpha_2 \sum_{e \in E_{T_2}} c_e \leq 2\text{OPT}.$$

Prova. De (6.13) temos que

$$\begin{aligned}
\sum_{e\in E_{T_1}} c_e &\leq \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S^1 - |A_1|\lambda_1\right)\\
&= \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S^1 - |A_1|\lambda_2 + |A_1|(\lambda_2-\lambda_1)\right)\\
&\leq \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S^1 - |A_1|\lambda_2\right) + \left(2-\frac{1}{n}\right)\frac{c_{\min}|A_1|}{2n(2n+1)}\\
&\leq \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S^1 - |A_1|\lambda_2\right) + \frac{c_{\min}}{2n+1}.
\end{aligned}$$

A última desigualdade vale, pois, $\left(2-\frac{1}{n}\right)\frac{c_{\min}|A_1|}{2n(2n+1)} = \left(\frac{(2n-1)|A_1|}{(2n)n}\right)\frac{c_{\min}}{2n+1}$ e $\left(\frac{(2n-1)|A_1|}{(2n)n}\right) \leq 1$.

Como $\alpha_1+\alpha_2 = 1$ e $\alpha_1 k_1 + \alpha_2 k_2 = k$, então $\alpha_1|A_1| + \alpha_2|A_2| = n-k$, pois $|A_1| = n-k_1$ e $|A_2| = n-k_2$. Considere que $\alpha_1 y_S^1 + \alpha_2 y_S^2 = y_S$ para todo $S \subseteq V \setminus \{r\}$. Logo,

$$\begin{aligned}
\alpha_1 \sum_{e\in E_{T_1}} c_e + \alpha_2 \sum_{e\in E_{T_2}} c_e &\leq \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S - \lambda_2(\alpha_1|A_1| + \alpha_2|A_2|)\right) + \frac{\alpha_1 c_{\min}}{2n+1}\\
&= \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S - \lambda_2(n-k)\right) + \frac{\alpha_1 c_{\min}}{2n+1}.
\end{aligned}$$

Como y é combinação convexa de y^1 e y^2, então y é um candidato a uma solução de (6.12). Pela dualidade fraca, temos que $\sum_{S\subseteq V\setminus\{r\}} y_S - \lambda_2(n-k) \leq \text{OPT}$. E pela hipótese 3 sobre a instância do problema e o fato de $\alpha_1 \leq 1$, temos que $\alpha_1 c_{\min} \leq \text{OPT}$. Portanto,

$$\begin{aligned}
\alpha_1 \sum_{e\in E_{T_1}} c_e + \alpha_2 \sum_{e\in E_{T_2}} c_e &\leq \left(2-\frac{1}{n}\right)\left(\sum_{S\subseteq V\setminus\{r\}} y_S - \lambda_2(n-k)\right) + \frac{\alpha_1 c_{\min}}{2n+1}\\
&\leq \left(2-\frac{1}{n}\right)\text{OPT} + \frac{\text{OPT}}{2n+1}\\
&\leq 2\text{OPT}.
\end{aligned}$$

□

Então, se $\alpha_2 \geq \frac{1}{2}$, a árvore T_2 é uma 4-aproximação para o kMST-R, pois T_2 é uma k-árvore e pelo lema 6.6 temos $\sum_{e\in E_{T_2}} c_e \leq 2\alpha_2 \sum_{e\in E_{T_2}} c_e \leq 2(2\text{OPT}) = 4\text{OPT}$.

No entanto, se $\alpha_2 < \frac{1}{2}$, não sabemos se T_2 é uma 5-aproximação para o kMST-R. Logo, adicionamos vértices em T_1 para que vire uma k-árvore, como descrito anteriormente (figura 6.2). Então o custo de T_1 é acrescido do custo do caminho P obtido de T_2 mais o custo de conectar P a r.

Pela hipótese 2 sobre a instância do problema, temos que o custo de conectar P a r não é maior que OPT.

O caminho P é um caminho de um circuito com pelo menos $k_2 - k_1$ vértices, obtido aplicando a técnica dos atalhoss no grafo obtido pela duplicação das arestas de T_2. Pela hipótese 1 sobre a instância, sabemos que o custo desse circuito não é maior que o dobro do custo de T_2. Além disso, P é um caminho de menor custo e com $k - k_1$ vértices desse circuito. Logo, o custo de P é

$$c(P) \leq 2\frac{k - k_1}{k_2 - k_1} \sum_{e \in E_{T_2}} c_e.$$

De (6.16) sabemos que $\frac{k-k_1}{k_2-k_1} = \alpha_2$. Além disso, $\alpha_1 \geq \frac{1}{2}$, pois $\alpha_2 < \frac{1}{2}$. Logo,

$$\begin{aligned}
\sum_{e \in E_{T_1}} c_e + 2\frac{k - k_1}{k_2 - k_1} \sum_{e \in E_{T_2}} c_e + \text{OPT} &= \sum_{e \in E_{T_1}} c_e + 2\alpha_2 \sum_{e \in E_{T_2}} c_e + \text{OPT} \\
&\leq 2\left(\alpha_1 \sum_{e \in E_{T_1}} c_e + \alpha_2 \sum_{e \in E_{T_2}} c_e\right) + \text{OPT} \\
&\leq 4\text{OPT} + \text{OPT} = 5\text{OPT}.
\end{aligned}$$

Portanto, quando existe uma solução, o algoritmo sempre devolve uma 5-aproximação para o kMST-R. Isso prova o teorema 6.5.

CAPÍTULO 7

2-aproximação de Garg

O algoritmo com menor fator de aproximação conhecido para o kMST foi apresentado por Naveen Garg [Gar05b] em 2005. O fator de aproximação desse algoritmo é 2, melhorando o $2 + \varepsilon$ obtido por Sanjeev Arora e George Karakostas [AK06].

Como os demais algoritmos com fator de aproximação constante, esse algoritmo de Garg utiliza uma subrotina baseada num algoritmo para o PCST. Mais precisamente, ele utiliza uma modificação de uma subrotina para uma versão do PCST sem raiz. Essa subrotina é, essencialmente, o algoritmo desenvolvido por David Stifler Johnson, Maria Minkoff e Steven John Phillips [JMP00] para o PCST, que utiliza a mesma estratégia apresentada por Michel Xavier Goemans e David Paul Williamson [GW95] para o PCST.

Como vimos na seção 6.3 uma das principais idéias de se utilizar o PCST é que, tomando bons valores para as penalidades nos vértices, obtemos boas aproximações para o kMST. Na 5-aproximação de Garg [Gar96] fazemos uma busca binária para encontrar o valor para a penalidade nos vértices. Na 2-aproximação, também é feita uma busca parecida, no entanto, mais complexa.

Na tentativa de facilitar o entendimento, dividiremos a explicação deste algoritmo em partes. Primeiro vamos apresentar uma idéia geral do algoritmo. Em seguida, discutiremos alguns elementos presentes na subrotina central do algoritmo. E, finalmente, apresentaremos o algoritmo de forma mais detalhada.

7.1 Idéia básica do algoritmo

A 2-aproximação de Garg recebe um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_\geq$ e um número inteiro k e devolve uma k-árvore T do grafo.

Como dito anteriormente, esse algoritmo utiliza como subrotina principal uma modificação de um algoritmo para o PCST sem raiz. Chamaremos essa subrotina de GW-MODIFICADO. Assim como o algoritmo GW da seção 6.2, GW-MODIFICADO começa expandindo uma floresta F e no final utiliza a subrotina PODA para remover determinadas arestas de F, obtendo uma floresta $\widehat{F}$.

Uma das características principais da subrotina GW-MODIFICADO para o algoritmo de Garg é a formação de uma coleção laminar de partes do conjunto de vértices do grafo associada ao valor das variáveis duais devolvido pelo algoritmo.

A função penalidade fornecida à subrotina GW-MODIFICADO será sempre constante e por isso muitas vezes será denotada apenas como um número racional. Se q é o valor da penalidade nos vértices fornecida para GW-MODIFICADO, denotaremos por F_q e $\widehat{F_q}$ as florestas obtidas antes e depois do processo de poda respectivamente. Também denotaremos por $\alpha(q)$ a maior quantidade de vértices em uma árvore da floresta $\widehat{F_q}$ devolvida por GW-MODIFICADO.

O valor de penalidade q de interesse é tal que $\alpha(q_-) < k$ e $\alpha(q_+) \geq k$, onde q_- e q_+ representam valores infinitesimalmente[1] menor e maior que q, respectivamente. Chamamos tal valor de **penalidade limiar**. O algoritmo de Garg utiliza a subrotina GARG-LIMIAR para encontrar uma penalidade limiar.

Dada uma penalidade limiar q, o algoritmo invoca a subrotina GARG-k-ÁRVORE para encontrar uma k-árvore T' com exatamente k vértices. Seja $\mathcal{L}$ a coleção laminar dada por GW-MODIFICADO. A idéia chave para obter a 2-aproximação para o kMST é que ou T' é uma 2-aproximação ou, se não é, sabemos que existe uma k-árvore de custo mínimo em algum conjunto de $\mathcal{L}$ que não intersecta $V_{T'}$. No segundo caso, encontramos, por chamadas recursivas do algoritmo de Garg, uma 2-aproximação em subgrafos induzidos por determinados conjuntos maximais em $\mathcal{L}$. Isso será explicado de forma mais detalhada na seção 7.8.

[1]Isso significa que são valores que diferem de q por um $\epsilon > 0$ tão pequeno quanto se queira.

7.2 Coleções laminares

Uma coleção $\mathcal{L}$ de partes de um conjunto X é chamada **laminar** se para todo L_1 e L_2 em $\mathcal{L}$ temos que $L_1 \subseteq L_2$, ou $L_2 \subseteq L_1$, ou $L_1 \cap L_2 = \emptyset$. Para qualquer parte Y de X adotaremos a seguinte notação[2]:

$$\begin{aligned}
&\mathcal{L}[Y] := \{L \in \mathcal{L} \mid L \subseteq Y\} \\
&\mathcal{L}(Y) := \{L \in \mathcal{L} \mid L \subset Y\} \\
&\mathcal{L}_Y := \{L \in \mathcal{L} \mid L \supseteq Y\} \\
&\overline{Y} := X \setminus Y.
\end{aligned}$$

O conjunto dos elementos maximais de $\mathcal{L}$ será denotado por $\mathcal{L}^*$. Podemos observar que o fato de $\mathcal{L}$ ser laminar implica que os elementos em $\mathcal{L}^*$ são dois-a-dois disjuntos.

Sejam G um grafo e $\mathcal{L}$ uma coleção laminar das partes de V_G. Para qualquer aresta e em E_G considere $\mathcal{L}(e) := \{L \in \mathcal{L} \mid e \in \delta(L)\}$. Sejam y uma função de 2^{V_G}, c uma função de E_G e π uma função de V_G. Dizemos que y **respeita** c se

$$y(\mathcal{L}(e)) \leq c_e \tag{7.1}$$

para toda aresta e em E_G. Se para uma determinada aresta temos que (7.1) vale com igualdade, então chamamos a aresta de **justa**. Dizemos que y **respeita** π se

$$y(\mathcal{L}[L]) \leq \pi(L) \tag{7.2}$$

para todo L em $\mathcal{L}$. Se para um determinado L temos que (7.2) vale com igualdade, então dizemos que y **satura** L.

Lema 7.1 (da dualidade)**.** Sejam G um grafo conexo, c uma função custo de E_G em $\mathbb{Q}_\geq$ e π uma função penalidade de V_G em $\mathbb{Q}_\geq$. Sejam $\mathcal{L}$ uma coleção laminar de partes de V_G contendo V_G, y uma função de $\mathcal{L}$ em $\mathbb{Q}_\geq$ e T um subgrafo conexo de G. Se y respeita c e π, então

$$c(T) \geq \pi(V_T) - (\pi(S) - y(\mathcal{L}(S))), \tag{7.3}$$

onde S é um conjunto de vértices minimal em $\mathcal{L}$ que contém V_T. Sabemos que S existe, pois V_G está em $\mathcal{L}$. Ademais, se $\pi_v = p$ para todo vértice v em V_G e T é uma k-árvore, então

$$c(T) \geq kp - (\pi(S) - y(\mathcal{L}(S))). \tag{7.4}$$

[2] Para não confundir a notação basta associá-la à notação de intervalos. Intervalo fechado, que inclui os extremos, é denotado por $[\cdot,\cdot]$. Intervalo aberto, que não inclui os extremos, é denotado por $(\cdot,\cdot)$.

Prova. Seja $\mathcal{A} := \{L \in \mathcal{L} \mid \delta_T(L) \neq \emptyset\}$. Como y respeita c, temos que

$$y(\mathcal{A}) \leq \sum_{L \in \mathcal{A}} |\delta_T(L)| y_L = \sum_{e \in E_T} y(\mathcal{L}(e)) \leq \sum_{e \in E_T} c_e = c(T).$$

Seja S um conjunto minimal em $\mathcal{L}$ contendo V_T. E seja $\mathcal{B} := \{L \in \mathcal{L}[\overline{V_T}] \mid L \subseteq S\}$. Note que $\mathcal{A}$ e $\mathcal{B}$ particionam S. Logo, $y(\mathcal{L}(S)) = y(\mathcal{A}) + y(\mathcal{B})$.

Então,

$$\begin{aligned} c(T) &\geq y(\mathcal{A}) = y(\mathcal{L}(S)) - y(\mathcal{B}) \\ &\geq y(\mathcal{L}(S)) - \pi(S \setminus V_T) \qquad (7.5) \\ &= \pi(V_T) - (\pi(S) - y(\mathcal{L}(S))). \end{aligned}$$

A desigualdade (7.5) vale, pois, como $\mathcal{B}$ é uma coleção laminar e y respeita π, temos que $y(\mathcal{B}) \leq \pi(S \setminus V_T)$.

Claramente, a desigualdade (7.4) é verdadeira, pois, no caso em que $\pi_v = p$ para todo vértice v em V_G e T é uma k-árvore, temos que $\pi(V_T) \geq kp$. □

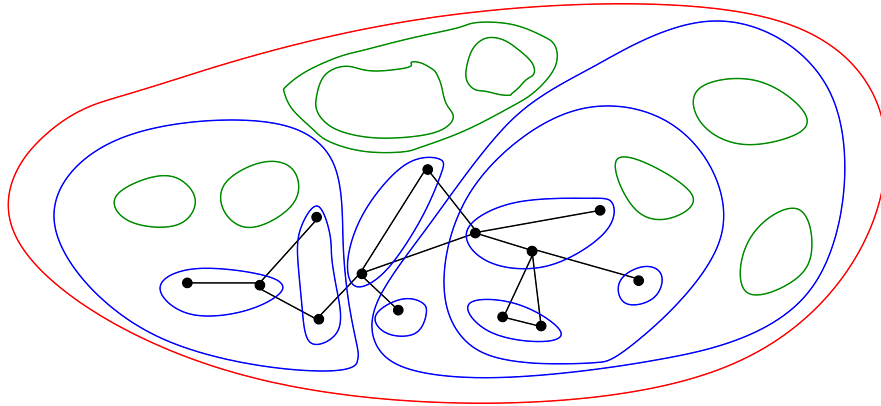


Figura 7.1: Ilustração de um subgrafo conexo T e de uma coleção laminar $\mathcal{L}$ restrita a S. O conjunto S é representado por linhas claras, $\mathcal{A}$ por linhas escuras e $\mathcal{L}[\overline{V_T}]$ por linhas tracejadas.

Como o lema 7.1 (da dualidade) vale para qualquer subgrafo conexo T, então, vale também para uma k-árvore de custo mínimo em particular. Logo, o lema da dualidade dá um limitante para o custo de uma k-árvore mínima.

7.3 A subrotina GW-MODIFICADO

Nesta seção apresentaremos o algoritmo GW-MODIFICADO adaptado por Garg e que é central para seu algoritmo. Como já mencionamos, essa subrotina é essencialmente o algoritmo desenvolvido por Johnson, Minkoff e Phillips [JMP00] para o PCST, que utiliza a mesma estratégia apresentada por Goemans e Williamson [GW95] para o PCST. A descrição que faremos é baseada na apresentada por Paulo Feofiloff, Cristina Gomes Fernandes, Carlos Eduardo Ferreira e José Coelho de Pina Jr. [FFFd07, FFFd09].

Seja G um grafo. Uma aresta é **interna** a uma partição $\mathcal{X}$ de V_G se as suas pontas estão em uma mesma parte de $\mathcal{X}$. As demais arestas são ditas **externas**. As arestas externas possuem dois elementos em $\mathcal{X}$ que possuem as suas pontas. Chamamos esses elementos de **extremos** da aresta em $\mathcal{X}$.

Sejam T um subgrafo de G e $\mathcal{L}$ uma coleção laminar de subconjuntos de V_G. Dizemos que T é **conexo em um conjunto** L de vértices de G se $T[V_T \cap L]$ é conexo. O subgrafo T é $\mathcal{L}$-conexo se é conexo em cada L de $\mathcal{L}$. Dizemos que T **não tem pontes** em $\mathcal{L}$ se $|\delta_T(L)| \neq 1$ para todo L em $\mathcal{L}$.

O algoritmo GW-MODIFICADO recebe, como entrada, um grafo conexo G, uma função custo $c : E_G \rightarrow \mathbb{Q}_{\geq}$ e uma função penalidade $\pi : V_G \rightarrow \mathbb{Q}_{\geq}$ e devolve:

- uma coleção laminar $\mathcal{L}$ de subconjuntos de V_G tal que V_G está em $\mathcal{L}$;
- uma função y de $\mathcal{L}$ em $\mathbb{Q}_{\geq}$ que respeita c e π e que satura V_G;
- uma função coloração χ de $\mathcal{L}$ em $\{\textit{branca}, \textit{cinza}, \textit{preta}\}$ tal que se L é um conjunto em $\mathcal{L}$ não saturado por y, então $\chi_L = \textit{branca}$, caso contrário $\chi_L = \textit{cinza}$ ou $\chi_L = \textit{preta}$.
- uma árvore T de G $\mathcal{L}$-conexa e sem pontes em $\{L \in \mathcal{L} \mid \chi_L = \textit{preta}\}$.

No algoritmo de Garg a função penalidade π será sempre constante, ou seja, $\pi_v = q \in \mathbb{Q}_{\geq}$ para todo v em V_G.

O algoritmo GW-MODIFICADO constrói uma árvore F adicionando uma aresta a cada iteração. Ademais, a cada iteração são mantidas as seguintes relações invariantes:

- $\mathcal{L}$ é a coleção dos subconjuntos L de V_G tal que $y_L > 0$;

- os conjuntos de vértices dos componentes de F são os conjuntos em $\mathcal{L}^*$;
- F é $\mathcal{L}$-conexa;
- y respeita c e π;
- cada aresta de F é justa e interna a $\mathcal{L}^*$;
- os conjuntos de cor *cinza* e *preta* são saturados por y e os de cor *branca* não são saturados por y.

O algoritmo começa com a coleção laminar $\mathcal{L} = \{\{v\} \mid v \in V_G\}$. Durante a execução do algoritmo, dizemos que um componente de F está **ativo** se não for saturado por y. O algoritmo mantém uma lista $\mathcal{A}$ de componentes que atualmente estão ativos. Em cada iteração o algoritmo aumenta uniformemente os valores de y correspondentes aos conjuntos de vértices dos componentes ativos de F até ocorrer um dos seguintes eventos: uma aresta externa a $\mathcal{L}^*$ fica justa ou um componente ativo fica saturado. Para os casos no qual mais de uma aresta fica justa no mesmo instante, consideramos uma certa ordem lexicográfica das arestas. Após o tratamento do evento ocorrido, uma nova iteração é iniciada. O algoritmo pára de iterar quando a lista $\mathcal{A}$ fica vazia.

Quando um componente ativo fica saturado, ele é removido de $\mathcal{A}$. Quando uma aresta externa a $\mathcal{L}^*$ fica justa, ela é inserida em F e a lista $\mathcal{A}$ é atualizada. Cada um de seus extremos recebe uma cor e a união deles é inserida em $\mathcal{L}$. Seja L um dos extremos da aresta que ficou justa. O critério para decidir a cor que um conjunto recebe é a seguinte:

- se L não é saturado por y, então recebe a cor *branca*;
- se L é saturado por y, mas era ativo no início da iteração, então recebe a cor *cinza*;
- caso contrário, recebe a cor *preta*.

Esses três casos são mutuamente exclusivos e um deles sempre ocorre.

Uma tentativa de visualizar esse processo seria imaginar que existe uma "circunferência" contendo cada componente ativo. Aumentar uniformemente os valores de y correspondentes aos componentes ativos, seria equivalente a aumentar uniformemente

o raio dessas circunferências. Quando um componente fica saturado, o raio da circunferência correspondente pára de aumentar. Se duas circunferências se tocarem, então a aresta que liga os componentes correspondentes ficou justa e essas duas circunferências se fundem. Para entender melhor veja a simulação do algoritmo no apêndice A.1.

O algoritmo GW-MODIFICADO utiliza uma subrotina PINTA-CONJUNTO para verificar se um extremo de uma aresta que ficou justa deve receber a cor *branca*, *cinza* ou *preta*. A subrotina PINTA-CONJUNTO recebe um extremo L da aresta saturada, a lista $\mathcal{A}$ de componentes ativos no início da iteração, a função y atualizada e a função coloração atual. A função coloração é devolvida após ser atualizada.

Algoritmo PINTA-CONJUNTO$(L, \mathcal{A}, y, \chi)$

1 se L não é saturado por y então
2 $\chi_L \leftarrow$ *branca*
3 senão se L está em $\mathcal{A}$ então
4 $\chi_L \leftarrow$ *cinza*
5 senão
6 $\chi_L \leftarrow$ *preta*
7 devolva χ

Ao terminar as iterações, isto é, quando a lista $\mathcal{A}$ fica vazia, o algoritmo utiliza a subrotina PODA para remover algumas arestas da floresta F. Essas arestas são removidas para que a floresta F não tenha pontes em $\{L \in \mathcal{L} \mid \chi_L = \textit{preta}\}$. A subrotina PODA recebe uma floresta F, uma coleção laminar $\mathcal{L}$ de partes de V_F e uma coloração χ e devolve uma subárvore maximal de F que não tem pontes em $\{L \in \mathcal{L} \mid \chi_L = \textit{preta}\}$.

Algoritmo PODA$(F, \mathcal{L}, \chi)$

1 enquanto $|\delta_F(L)| = 1$ para algum L em $\{L \in \mathcal{L} \mid \chi_L = \textit{preta}\}$ faça
2 seja e a aresta em $\delta_F(L)$
3 $F \leftarrow F - e$
4 $T \leftarrow$ árvore de F com maior número de vértices
5 devolva T

A seguir apresentamos um pseudo-código do algoritmo GW-MODIFICADO. Vamos supor que $\pi_v > 0$ para todo vértice v em V_G. Logo, inicialmente todo vértice é ativo. Essa hipótese não é absurda, pois, como já dissemos antes, o algoritmo GW-MODIFICADO sempre será chamado com uma função π constante e, como vimos na seção 6.3, se $\pi = 0$ a árvore devolvida conterá no máximo um vértice. No pseudo-código utilizaremos uma variável "cronômetro" t para facilitar a explicação do algoritmo. Esta variável será utilizada apenas para nos referirmos ao instante no qual uma aresta fica justa ou um componente fica saturado. Apesar de ser uma variável discreta, é conveniente considerá-la contínua.

O algoritmo GW-MODIFICADO recebe, como entrada, um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_\geq$ e uma função penalidade $\pi : V_G \to \mathbb{Q}_\geq$. São devolvidas uma árvore T, uma função $y : 2^{V_G} \to \mathbb{Q}_\geq$, uma coleção laminar $\mathcal{L}$ de partes de V_G e uma função coloração χ, como mencionado anteriormente.

Algoritmo GW-MODIFICADO(G, c, π)

1 $F \leftarrow (V, \emptyset)$
2 $\mathcal{L} \leftarrow \{\{v\} : v \in V_G\}$ $y \leftarrow 0$
3 χ inicialmente indefinida
4 $\mathcal{A} \leftarrow \mathcal{L}$ $t \leftarrow 0$
5 enquanto $\mathcal{A} \neq \emptyset$ faça
6 seja ε o maior número em $\mathbb{Q}_\geq$ tal que y' respeita c e π, onde y' é tal que $y'_L = y_L + \varepsilon$ se $L \in \mathcal{A}$ e $y'_L = y_L$ caso contrário
7 $t \leftarrow t + \varepsilon$
8 se alguma aresta e externa a $\mathcal{L}^*$ é justa para y' então
9 sejam L_1 e L_2 os extremos de e em $\mathcal{L}^*$
10 $\chi \leftarrow$ PINTA-CONJUNTO$(L_1, \mathcal{A}, y', \chi)$
11 $\chi \leftarrow$ PINTA-CONJUNTO$(L_2, \mathcal{A}, y', \chi)$
12 $\mathcal{L} \leftarrow \mathcal{L} \cup \{L_1 \cup L_2\}$ $y'_{L_1 \cup L_2} \leftarrow 0$
13 $F \leftarrow F + e$
14 $\mathcal{A} \leftarrow (\mathcal{A} \setminus \{L_1, L_2\}) \cup \{L_1 \cup L_2\}$
15 senão

16 se algum elemento $L \in \mathcal{A}$ é saturado por y' então $\mathcal{A} \leftarrow \mathcal{A} \setminus \{L\}$
17 $y \leftarrow y'$
18 $\mathcal{L} \leftarrow \mathcal{L} \setminus \{L \subseteq V_G \mid y_L = 0\}$
19 $T \leftarrow$ PODA$(F, \mathcal{P})$
20 devolva $T, y, \mathcal{L}, \mathcal{B}, \mathcal{C}$ e $\mathcal{P}$

O algoritmo GW-MODIFICADO pode ser implementado de tal maneira que seu consumo de tempo seja O$(|V_G|^2 \log |V_G|)$ [FFFd02].

7.4 Custos reduzidos e potenciais

Considere uma iteração da subrotina GW-MODIFICADO(G, c, π). Seja y a função de $\mathcal{L}$ em $\mathbb{Q}_\geq$ no início da iteração. O **custo reduzido** $\hat{c}_e$ de uma aresta e é

$$\hat{c}_e := c_e - y(\mathcal{L}(e)).$$

Se L_1 e L_2 em $\mathcal{L}$ são os extremos de uma aresta e, então o custo reduzido de e, em uma determinada iteração da subrotina GW-MODIFICADO, é o valor do maior incremento que podemos fazer à soma $y_{L_1} + y_{L_2}$ nessa iteração de forma que a função y continue respeitando a função custo c.

O **potencial** $\Delta(L)$ de um conjunto ativo L em $\mathcal{L}$ no início da iteração é

$$\Delta(L) := \pi(L) - y(\mathcal{L}[L]).$$

De forma semelhante ao custo reduzido de uma aresta, o potencial de um conjunto ativo L é o maior valor com o qual podemos incrementar y_L de modo que y continue respeitando a função penalidade π.

O **potencial inicial** de $\Delta_0(L)$ de um conjunto L em $\mathcal{L}$ é o potencial de L no momento em que L foi adicionado à $\mathcal{L}$. Logo, temos que

$$\Delta_0(L) := \pi(L) - y(\mathcal{L}(L)).$$

Tanto o custo reduzido de uma aresta quanto o potencial de um conjunto ativo variam a cada iteração, mas omitiremos esse fato da notação para não sobrecarregá-la. Já o potencial inicial de um conjunto L não depende da iteração, mas só está definido a partir do momento no qual L é adicionada à coleção $\mathcal{L}$.

Para cada valor ε calculado durante a execução da subrotina ocorre um dos seguintes **eventos**: uma aresta externa fica justa ou um conjunto ativo fica saturado. Para o início de cada iteração da subrotina GW-MODIFICADO vamos considerar os valores:

$$\varepsilon_0 := \min\{\Delta(L) \mid L \in \mathcal{A}\}\,;$$
$$\varepsilon_1 := \min\{\hat{c}(e) \mid e \text{ é uma aresta externa com apenas um extremo em } \mathcal{A}\}\,;$$
$$\varepsilon_2 := \min\{\frac{\hat{c}(e)}{2} \mid e \text{ é uma aresta externa com ambos extremos em } \mathcal{A}\}\,.$$

Então, o valor de ε calculado na linha 6 de GW-MODIFICADO é $\min\{\varepsilon_0, \varepsilon_1, \varepsilon_2\}$.

Considere uma iteração qualquer da subrotina GW-MODIFICADO que não seja a última. Suponha que $\hat{c}'(e)$ é o custo reduzido de uma aresta externa e e que $\Delta'(L)$ é o potencial de um conjunto ativo L no início da próxima iteração. Temos que

$$\hat{c}'(e) = \hat{c}(e) - j\,\varepsilon \quad \text{para cada aresta externa } e\text{, e que} \tag{7.6}$$
$$\Delta'(L) = \Delta(L) - \varepsilon \quad \text{para cada conjunto ativo } L, \tag{7.7}$$

onde j é o número de extremos ativos da aresta e sob consideração. Para as demais arestas o custo reduzido se mantém e para os demais conjuntos o potencial também se mantém. No apêndice A.2 mostramos um exemplo da evolução do custo reduzido de do potencial.

O algoritmo de Garg invoca a subrotina GW-MODIFICADO apenas para funções penalidade constante, isto é, que têm o mesmo valor para cada vértice. Seja q o valor dessa penalidade. Freqüentemente chamamos q de **penalidade** do algoritmo. Também usaremos q para denotar a função penalidade constante com valor q.

No início da execução de GW-MODIFICADO(G, c, q) temos $\hat{c}_e = c_e$ para cada aresta e em E_G e que $\Delta(\{v\}) = q$ para cada v em V_G. Devido às equações (7.6) e (7.7) e à definição de ϵ vemos indutivamente que ambos $\hat{c}_e$ e $\Delta(L)$ são funções lineares de q. Como este fato será utilizado de maneira fundamental já na próxima seção, o apresentamos como um lema.

Lema 7.2. Na execução de GW-MODIFICADO(G, c, q) temos que, no início de cada iteração os valores de $\hat{c}_e$ de cada aresta externa e de $\Delta(L)$ de cada componente ativo L são funções lineares da penalidade q. Conseqüentemente, os valores de y_L são funções lineares da penalidade q, para cada L em $\mathcal{L}$.

7.5 Convexidade das penalidades

Nesta seção continuamos a investigar o comportamento da subrotina GW-MODIFICADO restrita a penalidades constantes. Veremos que para valores de penalidades em um mesmo intervalo esse comportamento se assemelha a uma certa convexidade.

Por conveniência, vamos considerar a floresta F_q, a coleção $\mathcal{L}_q$ e a coloração χ_q mantidas pela subrotina GW-MODIFICADO(G, c, q) como sendo seqüências. Isto é, a ordem da seqüência indica a ordem em que os elementos foram inseridos em F_q, $\mathcal{L}_q$ e χ_q. Ademais, denotaremos por $F_q^i, \mathcal{L}_q^i$ e χ_q^i o estado dessas seqüências depois da $(i-1)$-ésima aresta justa ser encontrada, mas antes da i-ésima aresta justa ser encontrada.

Lema 7.3. Se p e r são penalidades e i é um inteiro positivo tais que $F_p^i = F_r^i$, então $\mathcal{L}_p^i = \mathcal{L}_r^i$

Prova. Provaremos por indução no número de iterações.

No início da primeira iteração da subrotina GW-MODIFICADO temos que $F_p^1 = F_r^1 = \emptyset$ e $\mathcal{L}_p^1 = \mathcal{L}_r^1 = \{\{v\} : v \in V_G\}$. Considere $i > 1$. Como as seqüências F_p^i e F_r^i são iguais, por hipótese de indução, temos que $\mathcal{L}_p^{i-1} = \mathcal{L}_r^{i-1}$. Logo os extremos da última aresta na seqüência $F_p^i = F_r^i$ são os mesmos em $\mathcal{L}_p^{i-1}$ e $\mathcal{L}_r^{i-1}$. Então o conjunto a ser inserido em $\mathcal{L}_p$ é o mesmo a ser inserido em $\mathcal{L}_r$. Portanto $\mathcal{L}_p^i = \mathcal{L}_r^i$. □

Seja q uma penalidade. Se fixamos a seqüência F_q^i, e conseqüentemente $\mathcal{L}_q^i$, e χ_q^i então podemos parametrizar os valores de y_L para cada L em $\mathcal{L}_q^i$ em função da penalidade q. Dessa forma, os custos reduzidos e potenciais também podem ser parametrizados em função da penalidade q. A seguir mostraremos alguns exemplos.

No exemplo da figura 7.2 temos que $F_q^4 = \langle ab, ad, bc \rangle$, os conjuntos $\{a\}, \{b\}, \{a, b\}$ e $\{a, b, d\}$ têm a cor *branca*, não há conjunto com a cor *cinza* e $\{d\}$ e $\{c\}$ têm a cor *preta*.

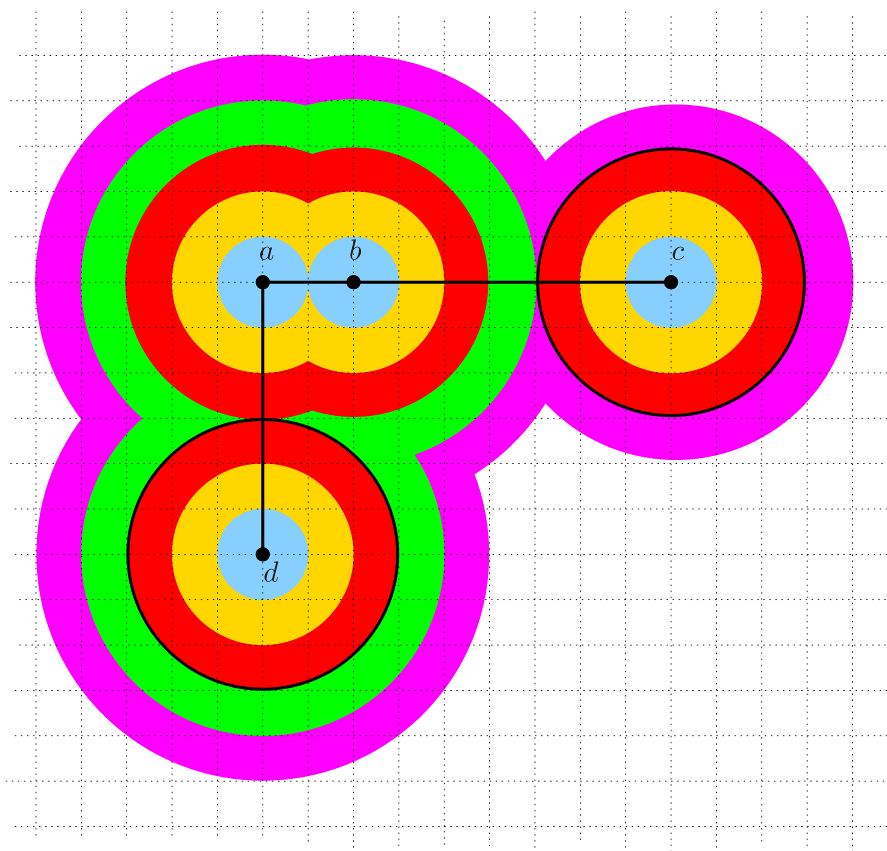


Figura 7.2: O custo das arestas corresponde à distância entre os vértices. Temos que $F_q^4 = \langle ab, ad, bc \rangle$, os conjuntos $\{a\}, \{b\}, \{a,b\}$ e $\{a,b,d\}$ têm a cor *branca* e $\{d\}$ e $\{c\}$ têm a cor *preta*.

Dadas essa seqüência e coloração, podemos calcular os valores de y da seguinte forma:

$$\begin{aligned}
y_{\{a\}} &= \frac{c_{ab}}{2} = 1 \\
y_{\{b\}} &= \frac{c_{ab}}{2} = 1 \\
y_{\{d\}} &= q \\
y_{\{a,b\}} &= c_{ad} - y_{\{a\}} - y_{\{d\}} = 5 - q \\
y_{\{c\}} &= q \\
y_{\{a,b,d\}} &= c_{bc} - y_{\{b\}} - y_{\{a,b\}} - y_{\{c\}} = 1 \\
y_{\{a,b,c,d\}} &= 4q - y_{\{a\}} - y_{\{b\}} - y_{\{d\}} - y_{\{a,b\}} - y_{\{c\}} - y_{\{a,b,d\}} = 3q - 8.
\end{aligned}$$

É evidente que esta parametrização não é, digamos, válida para qualquer valor de penalidade q. Para $q < \frac{8}{3}$ temos que $y_{\{a,b,c,d\}} < 0$. Para $q = \frac{8}{3}$ o conjunto $\{d\}$ deixa de ter a cor *preta* e passa a ter a cor *cinza*. Para $q = 3$ o conjunto $\{d\}$ deixa de ter a cor *preta* e passa a ter a cor *branca*. Podemos verificar que para qualquer valor de penalidade q no intervalo $(\frac{8}{3}, 3)$ a parametrização de y é tal que

- y respeita c e q e
- F_q^4 e χ_q^4 não se alteram.

Em geral, diremos que uma parametrização de y em função de uma penalidade q é **válida** para um número inteiro positivo i e no intervalo (p, r), se para toda penalidade q em (p, r) vale que

- y respeita c e q e
- $F_{p_+}^i = F_q^i = F_{r_-}^i$ e $\chi_{p_+}^i = \chi_q^i = \chi_{r_-}^i$,

onde p_+ representa um número infinitesimalmente maior que p e r_- um número infinitesimalmente menor que r.

Consideremos mais um exemplo mostrado na figura 7.3. Como no exemplo anterior, os custos das arestas são as distâncias entre os vértices. Em particular, o custo da aresta cf é 6,2 e o custo da aresta de é $\gamma := \sqrt{8^2 + 0{,}2^2}$.

Temos que $F_q^5 = \langle ab, ef, ad, cf \rangle$, os conjuntos $\{a\}, \{b\}, \{e\}, \{f\}, \{a, b\}, \{d\}$ e $\{e, f\}$ têm a cor *branca*, não há conjunto com a cor *cinza* e $\{c\}$ tem a cor *preta*. Dadas essa seqüência e coloração, temos que

$$\begin{aligned}
y_{\{a\}} = y_{\{b\}} = y_{\{e\}} = y_{\{f\}} &= 1\,, \\
y_{\{a,b\}} &= 2\,, \\
y_{\{d\}} &= 3\,, \\
y_{\{c\}} &= q\,, \\
y_{\{e,f\}} &= 6{,}2 - 1 - q = 5{,}2 - q\,, \\
y_{\{a,b,d\}} &= 6{,}2 - q - 3 = 3{,}2 - q\,.
\end{aligned}$$

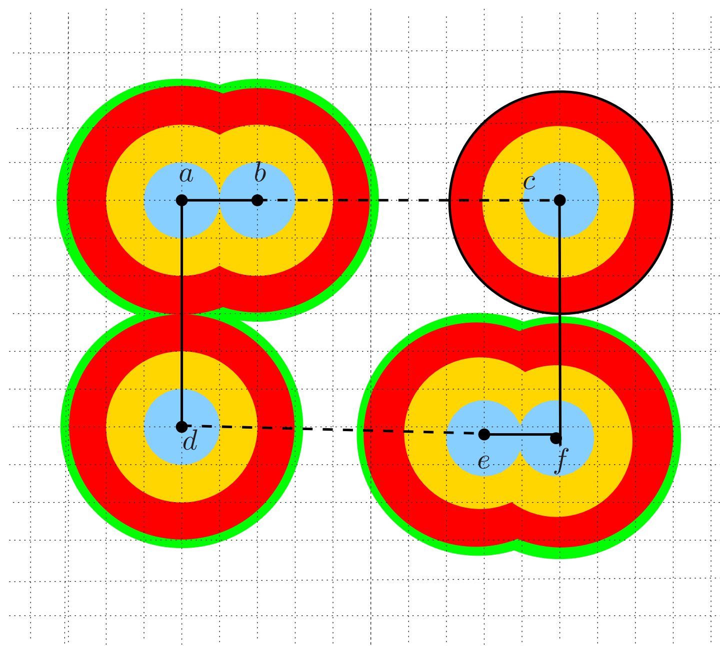


Figura 7.3: O custo das arestas corresponde às distâncias entre os vértices. Temos que $F_q^5 = \langle ab, ef, ad, cf \rangle$, os conjuntos $\{a\}, \{b\}, \{e\}, \{f\}, \{a,b\}, \{d\}$ e $\{e,f\}$ têm a cor *branca* e $\{c\}$ tem a cor *preta*.

Portanto, temos os seguintes custos reduzidos

$$\hat{c}_{bc} = 8 - 3 - q - (3{,}2 - q) = 1{,}8\,,$$
$$\hat{c}_{de} = \gamma - 3 - 3 - 2(3{,}2 - q) = \gamma - 12{,}4 + 2q$$

e potenciais

$$\Delta(\{a,b,d\}) = 3q - 1 - 1 - 1 - 2 - 2 - 0{,}2 = 3q - 5{,}2\,,$$
$$\Delta(\{c,e,f\}) = 3q - 1 - 1 - q - 2{,}2 = 2q - 3{,}2\,.$$

Temos que $\{d\}$ deixa de ter a cor *branca* e passa a ser *cinza* se $q = 3$, e passa a ter a cor *preta* se $q < 3$. Por outro lado, $\{c\}$ muda da cor *preta* para a cor *cinza* se $q = 3{,}1$ e passa a ter a cor *branca* se $q \geq 3{,}1$. Assim, a parametrização acima é válida para q em $(3{,}0, 3{,}1)$.

Para um dado valor de penalidade no intervalo de uma parametrização válida, podemos determinar o valor de ε utilizado na subrotina GW-MODIFICADO apenas calculando os custos reduzidos e os potenciais através da parametrização.

O fato de uma parametrização ser válida em um determinado intervalo é fortemente explorado pelo algoritmo de Garg.

Lema 7.4 (do intervalo)**.** Se p e r são penalidades, $p \leq r$, e i é um inteiro positivo tais $F^i_{p_+} = F^i_{r_-}$ e $\chi^i_{p_+} = \chi^i_{r_-}$ então $F^i_{p_+} = F^i_q = F^i_{r_-}$ e $\chi^i_{p_+} = \chi^i_q = \chi^i_{r_-}$ para todo q no intervalo (p, r).

7.6 Penalidade limiar

Sejam G um grafo conexo e c uma função custo. Denotaremos por $\alpha(p)$ o número de vértices da árvore devolvida ao invocarmos a subrotina GW-MODIFICADO(G, c, p). O valor de $\alpha(p)$ também depende de G e c, mas eles estarão implícitos para não sobrecarregar a notação, pois esses parâmetros são fixos. Não é difícil ver que $\alpha(0) \leq 1$ e $\alpha(\infty) = |V_G|$, onde ∞ corresponde a um valor suficientemente grande[3].

Dado um número p, denotaremos respectivamente por p_- e p_+ números infinitesimalmente menor e maior que p. Dizemos que p é uma **penalidade limiar** para um número inteiro k se $\alpha(p_-) < k$ e $\alpha(p_+) \geq k$. Encontrar uma penalidade limiar é uma parte importante do algoritmo de Garg e é feita pela subrotina GARG-LIMIAR.

O algoritmo GARG-LIMIAR recebe, como entrada, um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_\geq$ e um número inteiro positivo $k > 1$, e devolve uma penalidade limiar q. A idéia do algoritmo GARG-LIMIAR se assemelha à da busca binária.

O algoritmo é iterativo e mantém penalidades p e r e um número inteiro positivo i. No início de cada iteração vale que

(p1) $\alpha(p_+) < k \leq \alpha(r_-)$.

(p2) $F^{i-1}_{p_+} = F^{i-1}_{r_-}$ e $\chi^{i-1}_{p_+} = \chi^{i-1}_{r_-}$.

Em cada iteração, a penalidade limiar é encontrada e devolvida ou o intervalo $[e, d]$ de busca é diminuído. O valor de i é incrementado depois de um certo número de iterações, como veremos.

[3]Um valor suficientemente grande para este caso é $\sum_{e \in E_G} c_e$.

O algoritmo GARG-LIMIAR utiliza a subrotina PENALIDADE. Esta recebe o grafo G, a função custo c, as penalidades p e r e o número positivo i, satisfazendo $\alpha(p_+) < \alpha(r_-)$ e (p2). A subrotina PENALIDADE devolve uma penalidade q, $p < q < r$, e um número inteiro positivo i', $i \leq i' \leq i+1$. Além disso, se $i' = i+1$ a subrotina promete que

> o invariante (p2) vale com $i+1$ no papel de i e com q no papel de p ou r.

O fato importante aqui é que a subrotina PENALIDADE devolve $i' = i+1$ depois de não mais de que $|E_G| + 2$ invocações pelo algoritmo GARG-LIMIAR. Esse fato segue do lema 7.5 que veremos mais adiante.

Algoritmo GARG-LIMIAR(G, c, k)

1. $p \leftarrow 0 \quad r \leftarrow \max\{c_e : e \in E_G\}$
2. $F_{p_+}, T_{p_+}, y^{p_+}, \mathcal{L}_{p_+}, \chi_{p_+} \leftarrow$ GW-MODIFICADO(G, c, p_+)
3. $F_{r_-}, T_{r_-}, y^{r_-}, \mathcal{L}_{r_-}, \chi_{r_-} \leftarrow$ GW-MODIFICADO(G, c, r_-)
4. $i \leftarrow 1$
5. repita
6. $q, i \leftarrow$ PENALIDADE(G, c, p, r, i)
7. $F_{q_-}, T_{q_-}, y^{q_-}, \mathcal{L}_{q_-}, \chi_{q_-} \leftarrow$ GW-MODIFICADO(G, c, q_-)
8. $F_{q_+}, T_{q_+}, y^{q_+}, \mathcal{L}_{q_+}, \chi_{q_+} \leftarrow$ GW-MODIFICADO(G, c, q_+)
9.   se $\alpha(q_-) \geq k$ então
10. $r \leftarrow q$
11.   senão se $\alpha(q_+) < k$ então
12. $p \leftarrow q$
13. até que $\alpha(q_-) < k \leq \alpha(q_+)$
14. devolva q

Para verificar a corretude do algoritmo, primeiro notemos que as relações invariantes (p1) e (p2) valem no início da primeira iteração, pois $k > 1$, $c_e > 0$ para cada aresta e em E_G, $\alpha(p_+) = 1$ e $\alpha(r_-) = |V_G|$. Além disso, dado que (p1) e (p2) valem no início de uma iteração, então também valem no início da próxima. Isso porque a subrotina PENALIDADE devolve uma penalidade q, $p < q < r$. Se o valor de i não é incrementado,

então (p2) vale devido ao lema do intervalo 7.4. Já, se i é incrementado, então (p2) vale devido à especificação da subrotina PENALIDADE. O bloco de linhas 9–12 garante que, se a penalidade devolvida pela subrotina PENALIDADE não é a limiar, então a relação invariante (p1) vale no início da próxima iteração.

Assim, é claro que se o algoritmo GARG-LIMIAR pára, então devolve a penalidade limiar prometida. Por outro lado, é fácil verificar que o algoritmo pára. De fato, a relação invariante (p2) implica que para $i = |V_G| + 1$ vale que as listas $F_{p_+}^{|V_G|} = F_{r_-}^{|V_G|}$ e $\chi_{p_+}^{|V_G|} = \chi_{r_-}^{|V_G|}$. Portanto, devido à especificação da subrotina PODA, invocada por GW-MODIFICADO, temos que $T_{p_+} = T_{r_-}$, e portanto $\alpha(p_+) = \alpha(r_-)$, o que contradiz a relação invariante (p1). Como o valor de i será incrementado depois de não mais que $|E_G|+2$ invocações da subrotina PENALIDADE (lema 7.5) temos que o algoritmo GARG-LIMIAR pára e corretamente devolve uma penalidade limiar q depois de não mais do que $|V_G|(|E_G| + 2) = \mathrm{O}(|V_G|\,|E_G|)$ iterações.

Como foi argumentado logo acima, o número de iterações é $\mathrm{O}(|V_G|\,|E_G|)$. Mais adiante veremos que o consumo de tempo da subrotina PENALIDADE é $\mathrm{O}((|V_G|+|E_G|)$. Assim, em cada iteração o consumo de tempo dominante será o da subrotina GW-MODIFICADO que é $\mathrm{O}(|V_G|^2 \log |V_G|)$. Portanto, o consumo de tempo do algoritmo é $\mathrm{O}(|E_G|\,|V_G|^3 \log |V_G|)$.

Descrição do algoritmo PENALIDADE

O algoritmo PENALIDADE recebe um grafo G, uma função custo c, penalidades p e r e um número positivo i, satisfazendo $\alpha(p_+) < \alpha(r_-)$ e (p2) e devolve uma penalidade q, $p < q < r$, e um número inteiro positivo i', $i \leq i' \leq i + 1$. Além disso, se $i' = i + 1$ a subrotina promete que

> o invariante (p2) vale com $i + 1$ no papel de i e com q no papel de p ou r.

O algoritmo PENALIDADE recebe ainda todos os objetos devolvidos pelas invocações de GW-MODIFICADO feitas pelo algoritmo GARG-LIMIAR além da parametrização de $y, \hat{c}$ e Δ válida para $i - 1$ e $[p_+, r_-]$. Esses parâmetros foram omitidos para não sobrecarregarmos a descrição do algoritmo.

Algoritmo PENALIDADE(G, c, p, r, i)

1 se F_{p_+} não tem i arestas então ▷ Caso 1
2 $q \leftarrow \min\{x \in [p, r] : F_x$ tem i arestas$\}$
3 senão se $F^i_{p_+} \neq F^i_{r_-}$ então ▷ Caso 2
4 $e_{p_+} \leftarrow i$-ésima aresta de $F^i_{p_+}$
5 $e_{r_-} \leftarrow i$-ésima aresta de $F^i_{r_-}$
6 $q \leftarrow \min\{x \in [p, r] : e_{r_-}$ fica justa antes de e_{p_+} em $F_x\}$
7 senão se $F^i_{p_+} = F^i_{r_-}$ e $\chi^i_{p_+} \neq \chi^i_{r_-}$ então ▷ Caso 3
8 $e \leftarrow i$-ésima aresta de $F^i_{p_+}$
9 $L_1, L_2 \leftarrow$ extremos de e em $(\mathcal{L}^{i-1}_{p_+})^*$ com $\chi_{p_+}(L_2) =$ *preta*
10 $q \leftarrow \min\{x \in [p, r] : \chi_x(L_2) =$ *cinza*$\}$
11 senão se $F^i_{p_+} = F^i_{r_-}$ e $\chi^i_{p_+} = \chi^i_{r_-}$ então ▷ Caso 4
12 $q \leftarrow (p + r)/2$
13 $i \leftarrow i + 1$
14 devolva q e i

Examinamos separadamente cada um dos quatro casos considerados pelo algoritmo. É evidente que um, e só um, dos casos considerados ocorre.

Caso 1 F_{p_+} não tem i arestas. Pelo invariante (p2), concluímos que F_{p_+} tem exatamente $i - 1$ arestas. Como, por hipótese $\alpha(p_+) < \alpha(r_-)$, então F_{r_-} contém pelo menos i arestas. Portanto, a penalidade q calculada na linha 2 e devolvida pelo algoritmo é tal que $p < q < r$.

Notemos que, se $F^i_{p_+}$ tem i-arestas, então, $F^i_{r_-}$ tem i arestas, pois p, r e i satisfazem (p2) e $\alpha(p_+) < \alpha(r_-)$.

Caso 2 $F^i_{p_+} \neq F^i_{r_-}$. Como p, r e i satisfazem (p2), então as listas $F^i_{p_+}$ e $F^i_{r_-}$ diferem na i-ésima aresta. Logo, o valor da penalidade q calculada na linha 6 satisfaz $p < q < r$.

Caso 3 $F^i_{p_+} = F^i_{r_-}$ e $\chi^i_{p_+} \neq \chi^i_{r_-}$. Seja e definida na linha 8 e sejam L_1 e L_2 seus extremos em $(\mathcal{L}^{i-1}_{p_+})^*$. Pelo lema 7.3, temos que $\mathcal{L}^i_{p_+} = \mathcal{L}^i_{r_-}$. Como, pela relação invariante (p2) $\chi^{i-1}_{p_+} \neq \chi^{i-1}_{r_-}$, então $\chi^i_{p_+}$ e $\chi^i_{r_-}$ diferem apenas na coloração de L_1 ou L_2.

Para todo L em $\mathcal{L}^i_{p+}$, se $\chi^i_{p+}(L) = $ *branca*, então $\chi^i_{r-}(L) = $ *branca*. Logo, temos que $\chi^i_{p+}(L_1) \neq $ *branca* ou $\chi^i_{p+}(L_2) \neq $ *branca*.

Como p_+ é infinitesimalmente maior que p podemos supor que $\chi^i_{p+}(L_1) \neq $ *cinza* e $\chi^i_{p+}(L_2) \neq $ *cinza*.

Assim, podemos supor que $\chi^i_{p+}(L_1) = $ *branca* e $\chi^i_{p+}(L_2) = $ *preta* e portanto $\chi^i_{r-}(L_1) = \chi^i_{r-}(L_2) = $ *branca*. Logo, o valor da penalidade q calculada na linha 10 satisfaz $p < q < r$.

Caso 4 $F^i_{p+} = F^i_{r-}$ e $\chi^i_{p+} = \chi^i_{r-}$. Nesse caso é evidente que o valor de q calculado na linha 12 satisfaz $p < q < r$ e que, pelo lema do intervalo, (p2) vale com $i + 1$ no papel de i e com q no papel de p ou r.

Aqui terminamos a demonstração da correção do algoritmo PENALIDADE.

Estamos agora preparados para demonstrar o fato que na linha 6 do algoritmo GARG-LIMIAR o valor de i é incrementado depois de não mais do que $|E_G| + 2$ iterações. Este fato é conseqüência do próximo lema.

Lema 7.5. Um seqüência de iterações do bloco de linhas 5–13 do algoritmo GARG-LIMIAR em que o valor de i permanece constante tem comprimento não superior a $|E_G| + 2$.

Prova. Considere uma seqüência $[p_1, r_1], [p_2, r_2], \ldots, [p_s, r_s]$ de intervalos $[p, r]$ considerados em uma seqüência de iterações do algoritmo GARG-LIMIAR em que o valor de i permanece constante. Equivalentemente, nessa seqüência de iterações não ocorre o caso 4 em uma invocação de subrotina PENALIDADE. Queremos mostrar que $s \leq |E_G| + 2$.

Como o valor q devolvido pela subrotina PENALIDADE é tal que $p < q < r$, então o comprimento do intervalo $[p, r]$ diminui estritamente a cada iteração e conseqüentemente

$$p_1 \leq p_2 \leq \cdots \leq p_s \quad \text{e} \quad r_1 \geq r_2 \geq \cdots \geq r_s\,,$$

onde exatamente metade dessas desigualdades são estritas.

Se $F_{p_{j+}}$ tem i arestas para algum j, $0 \leq j \leq s$, então $F_{p_{t+}}$ tem i arestas para todo $t, j \leq t \leq s$, pois $p_t \geq p_j$. Portanto, o caso 1 da subrotina PENALIDADE ocorre no máximo uma vez durante toda a seqüência de iterações.

Suponha que em uma invocação da subrotina PENALIDADE ocorre o caso 2. Como p, r e i satisfazem (p2), então as listas F^i_{p+} e F^i_{r-} diferem na i-ésima aresta. Sejam e_{p+}, e_{r-} e q como determinados nas linhas 4, 5, e 6. Pela definição de q temos que para toda penalidade no intervalo (p, q) vale que e_{p+} fica justa antes de e_{r-}. Isto implica que para toda penalidade x em (p, q) a i-ésima aresta de F_x não é e_{r-}. Analogamente, para toda penalidade x em (q, r) a i-ésima aresta de F_x não é e_{p+}. Assim, após cada ocorrência do caso 2 na seqüência de iterações o número de arestas candidatas a serem a i-ésima aresta justa da floresta diminui de pelo menos um. Portanto, nessa seqüência de iterações o caso 2 ocorre não mais do que $|E_G|$ vezes.

Finalmente, considere agora uma execução do caso 3 e sejam L_1, L_2 e q como definidos nas linhas 9 e 10 da subrotina. Como $\chi^i_{p+}(L_1) =$ *branca* então $\chi^i_x(L_1) =$ *branca* para todo $p < x \leq r$. Logo, $\chi^i_{q-}(L_1) = \chi^i_{q+}(L_1) =$ *branca*. Por outro lado, como $\chi_q(L_2) =$ *cinza* então $\chi^i_{q-}(L_2) =$ *preta* $= \chi^i_{p+}(L_2)$ e $\chi^i_{q+}(L_2) =$ *branca* $= \chi^i_{r-}(L_2)$. Logo, independentemente da atualização de p ou r feita nas linhas 9–12 do algoritmo GARG-LIMIAR teremos que no início da próxima iteração $F^i_{p+} = F^i_{r-}$ e $\chi^i_{p+} = \chi^i_{r-}$ e ocorre o caso 4. Logo, cada ocorrência do caso 3 é seguida por uma ocorrência do caso 4, se q não for a penalidade limiar. Portanto, o caso 3 ocorre no máximo uma vez na seqüência de iterações.

Portanto, $s \leq 1 + |E_G| + 1 = |E_G| + 2$. □

Afirmamos que o consumo de tempo do algoritmo PENALIDADE é $O(|V_G| + |E_G|)$. Não é difícil de ver que, com exceção feitas às linhas 2, 6 e 10, o consumo de tempo de cada linha é $O(1)$.

Como p, r e i satisfazem (p2), então $F^{i-1}_{p+} = F^{i-1}_{r-}$ e $\chi^{i-1}_{p+} = \chi^{i-1}_{r-}$. Em particular, $\mathcal{L}^{i-1}_{p+} = \mathcal{L}^{i-1}_{r-}$. Pelo lema 7.4, $F^{i-1}_{p+} = F^{i-1}_x = F^{i-1}_{r-}$ e $\chi^{i-1}_{p+} = \chi^{i-1}_x = \chi^{i-1}_{r-}$ para todo x no intervalo (p, r). Assim, podemos considerar uma parametrização de y em função das penalidades que é válida para $i - 1$ no intervalo (p, r) (seção 7.5). Agora, pelo lema 7.2 da penalidade linear, os valores de $\hat{c}_e$ de cada aresta externa de $(\mathcal{L}^{i-1}_{p+})^*$ de $\Delta(L)$ de cada componente ativo L em $(\mathcal{L}^{i-1}_{p+})^*$ são funções lineares da penalidade x.

Vejamos como as linhas 2, 6 e 10 podem ser executadas. Inicialmente, para determinarmos os instantes em que cada conjunto fica saturado basta resolvermos $O(|V_G|)$ equações lineares. Como podemos resolver cada equação linear em tempo constante o consumo de para determinar esses instantes é $O(|V_G|)$.

Para calcular o valor de q na linha 2 calculamos o instante em que cada aresta externa fica justa. Precisamos resolver $O(|E_G|)$ equações lineares, levando em consideração o instante em que cada um dos extremos da aresta fica saturado. Dada a aresta que fica justa mais rapidamente, podemos calcular o valor q para que a aresta fique justa no instante calculado. O consumo de tempo extra necessário neste caso é $O(|E_G|)$.

Calcular o valor de q na linha 6 é mais simples que o da linha 2. Para isto basta calcularmos o menor instante em que e_{r_-} fica justa. Dado esse instante podemos determinar o valor de q. O consumo de tempo extra necessário nesse caso é constante.

O calculo do valor de q na linha 10 é o mais simples de todos. Precisamos apenas determinar o menor instante em que a aresta e fica justa. O valor de q é o da penalidade que faz com que o potencial de L_2 seja nulo nesse instante. O consumo de tempo extra necessário nesse caso é constante.

7.7 k-árvores de custos limitados

O algoritmo GARG-k-ÁRVORE recebe um grafo conexo G, um custo não-negativo c_e para cada aresta e em E_G, um número inteiro positivo k, a penalidade limiar q e devolve uma k-árvore T tal que

$$c(T) \leq 2(kq - \Delta_0(Q)),$$

onde Q é um conjunto em $\mathcal{L}_q$ tal que $V_T \subseteq Q$. O conjunto Q que vai nos interessar é aquele que tem o maior valor de Δ_0, pois ele fornecerá um limitante melhor.

Basicamente o algoritmo GARG-k-ÁRVORE alterará a árvore devolvida por GW-MODIFICADO de forma que a árvore T resultante possua as características a seguir:

- T tem exatamente k vértices;
- toda aresta de T é justa em relação a y;
- T é $\mathcal{L}_q$-conexa;
- todo vértice em $Q' \setminus V_T$ está em algum conjunto de cor *preta* de $\mathcal{L}(Q')$, onde Q' é o conjunto minimal que contém V_T.

Para provar o limitante do custo dessa árvore, utilizaremos o seguinte lema devido a Feofiloff, Fernandes, Ferreira e Pina [FFFd09].

Lema 7.6. Seja G um grafo conexo, $\mathcal{W}$ uma partição de V_G e $(\mathcal{X}, \mathcal{Y})$ uma bipartição de $\mathcal{W}$. Seja T uma árvore em G. Se T é $\mathcal{W}$-conexo e para todo $Y \in \mathcal{Y}$ temos que $|\delta_T(Y)| \neq 1$ e $V_T \not\subseteq Y$, então

$$\frac{1}{2}\sum_{X \in \mathcal{X}} |\delta_T(X)| + |\mathcal{X}[\overline{V_T}]| \leq |\mathcal{X}| - 1.$$

Teorema 7.7. Sejam $F_q, T_q, y^q, \mathcal{L}_q, \chi_q$ os objetos devolvidos pela invocação de GW-MODIFICADO(G, c, q). Suponha que T_q tenha mais do que k vértices. Seja T'_q uma subárvore de T'_q com as características citadas acima. Então, $c(T'_q) \leq 2(kq - \Delta_0(Q'))$, onde $Q' \in \mathcal{L}_q$ é o conjunto minimal que contém $V_{T'_q}$.

Prova. Para demonstrar este teorema vamos considerar que no início do algoritmo cada vértice do grafo tem q de créditos para pagar o aumento do valor das variáveis duais. Isto é, se em uma iteração, um conjunto S teve sua variável dual aumentada de ε, então cada vértice de S pagou $\frac{\epsilon}{|S|}$ dos seus créditos.

Temos que $kq - \Delta_0(Q') = kq - (|Q'|q - y(\mathcal{L}(Q'))) = y(\mathcal{L}(Q')) - (|Q' \setminus V_{T'}|)q$. Ademais todo vértice em $Q' \setminus V_{T'}$ está contido em algum conjunto de cor *preta*, ou seja, gastou todos os seus créditos antes mesmo de fazer parte de Q'. Portanto $kq - \Delta_0(Q')$ equivale à quantidade de créditos gastos pelos vértices de $V_{T'_q}$.

Seja $\mathcal{P} \subseteq \mathcal{L}_q$ a coleção dos conjuntos cuja variável dual aumentou apenas com créditos de vértices em $V_{T'_q}$. Logo, $\sum_{L \in \mathcal{P}} y_L \leq kq - \Delta_0(Q')$.

Seja $\mathcal{B} := \{L \in \mathcal{L}_q \mid |\delta_{T'_q}(L)| \neq \emptyset\}$. Como as arestas de T'_q são justas, temos que

$$c(T'_q) = \sum_{e \in E_{T'_q}} c_e = \sum_{e \in E_{T'_q}} y(\mathcal{L}(e)) = \sum_{L \in \mathcal{B}} |\delta_{T'_q}(L)| y_L.$$

Logo, basta mostrar que $\sum_{L \in \mathcal{B}} |\delta_{T'_q}(L)| y_L \leq 2\sum_{L \in \mathcal{P}} y_L$. A cada iteração de GW-MODIFICADO temos que $\sum_{L \in \mathcal{B}} |\delta_{T'_q}(L)| y_L$ aumenta no máximo $\varepsilon \sum_{L \in \mathcal{A}} |\delta_{T'_q}(L)|$ enquanto $\sum_{L \in \mathcal{P}} y_L$ aumenta de pelo menos $\varepsilon(|\mathcal{A}| - 1)$, onde $\mathcal{A}$ é a coleção dos conjuntos ativos restritos a T'_q. Pelo lema 7.6, tomando $\mathcal{W} = \mathcal{L}^*$, $\mathcal{X} = \mathcal{A}$ e $\mathcal{Y} = \mathcal{W} \setminus \mathcal{X}$, temos que

$$\sum_{L \in \mathcal{A}} |\delta_{T'_q}(L)| \leq 2|\mathcal{A}| - 1.$$

Portanto $c(T'_q) \leq 2(kq - \Delta_0(Q'))$. □

O teorema 7.7 prova que uma árvore $V_{T'_q}$ com as características citadas anteriormente tem custo no máximo $2(kq - \Delta_0(Q'))$, onde $Q' \in \mathcal{L}_q$ é o conjunto minimal que contém $V_{T'_q}$. Veremos mais adiante que a partir dessa árvore podemos encontrar outra com custo no máximo $2(kq - \Delta_0(Q))$, onde $Q \in \mathcal{L}_q$ é um conjunto que contém $V_{T'_q}$ e tem Δ_0 máximo.

Para encontrar a árvore desejada, a idéia do algoritmo GARG-k-ÁRVORE é iterar num processo de transformação da tupla $(F_{q_-}, \mathcal{L}_q, \chi_{q_-})$ na tupla $(F_{q_+}, \mathcal{L}_q, \chi_{q_+})$. Vimos na seção 6.3 que, se $\alpha(q_+) = k$, então temos uma 2-aproximação para o kMST. Assim, vamos supor que $\alpha(q_+) > k$.

Primeiro vamos notar que em ambas as tuplas estamos considerando a coleção laminar $\mathcal{L}_q$ de vértices de G. Isso é possível, pois, se S em $\mathcal{L}_q$ tem variável dual positiva durante a execução de GW-MODIFICADO(G, c, q) então S também terá variável dual positiva durante a execução de GW-MODIFICADO(G, c, q_-) e GW-MODIFICADO(G, c, q_+). Logo, S também pertence a $\mathcal{L}_{q_-}$ e $\mathcal{L}_{q_+}$.

Essa transformação é feita passo a passo. Seja $(F, \mathcal{L}_q, \chi)$ o estado atual dessa transformação. Certamente, em algum passo, uma árvore de F passará a ter mais do que k vértices. Esse passo é o que nos interessa. Consideramos duas etapas: etapa 1, na qual transformamos F_{q_-} em F_{q_+}, e etapa 2, na qual transformamos χ_{q_-} em χ_{q_+}.

Após cada passo, em ambas as etapas, chamamos a subrotina PODA para verificar se existe uma árvore com mais do que k vértices em F. Em caso positivo, paramos esse processo de transformação e passamos ao processo que obtém uma k-árvore com exatamente k vértices.

Etapa 1

Apesar de F_{q_-} e F_{q_+} poderem ser diferentes, temos que ambas são $\mathcal{L}_q$-conexas. Ademais, uma aresta em $E_{F_{q_+}} \setminus E_{F_{q_-}}$ só não é justa em relação a y^{q_-} por uma diferença infinitesimal. Logo, para cada L em $\mathcal{L}_q$, podemos transformar o subgrafo de F_{q_-} induzido por L pelo subgrafo de F_{q_+} induzido por S através de trocas e adições de arestas. Os elementos de $\mathcal{L}_q$ são considerados na ordem em que foram formados.

Considere que o passo atual ocorreu nesta etapa e foi responsável por F possuir uma árvore H com mais de k vértices.

Sejam e_- e e_+ as arestas de F_{q_-} e F_{q_+}, respectivamente, que foram trocadas no passo atual. Vamos considerar que e_- e e_+ estão em H, ou seja, e_- ainda não foi removida. Como a inserção de e_+ aumenta o número de vértices de H após a subrotina PODA, então, e_+ conecta dois conjuntos *pretos* que seriam podados.

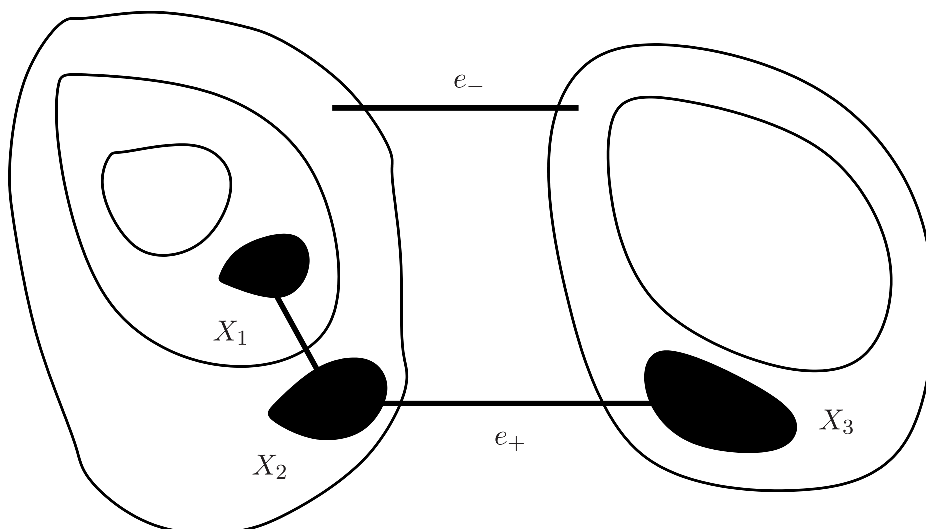


Figura 7.4: Ilustração do caminho formado por conjuntos *pretos* X_1, X_2 e X_3.

Sejam $X_1, X_2, \ldots, X_s$ os conjuntos *pretos* maximais que seriam podados em $F - e_+$. Note que ao inserir e_+, tais conjuntos formam um caminho. Consideraremos que a ordem na qual esse caminho é percorrido é $X_1, X_2, \ldots, X_s$. Removendo todos esses conjuntos *pretos* H fica com menos do que k vértices, então adicionamos vértices desses conjuntos à árvore utilizando a subrotina PEGA-VÉRTICE.

Considere a subárvore de H após a poda, isto é, sem os vértices em $X_1, X_2, \ldots, X_s$. Como ela tem menos do que k vértices, inserimos todos os vértices de X_1 nela. Se continuar com menos do que k vértices, inserimos todos os vértices de X_2 e continuamos o processo até encontrarmos um conjunto X_d que ao ser inserido nessa árvore, faz com que ela fique com mais do que k vértices. Seja H' essa subárvore de H.

A subrotina PEGA-VÉRTICE recebe as árvores H e H', um conjunto S em $\mathcal{L}_q$, um vértice v em S e um número inteiro $\widetilde{k}$ e devolve a árvore H' adicionada de $\widetilde{k}$ vértices de S, sendo um deles o vértice v.

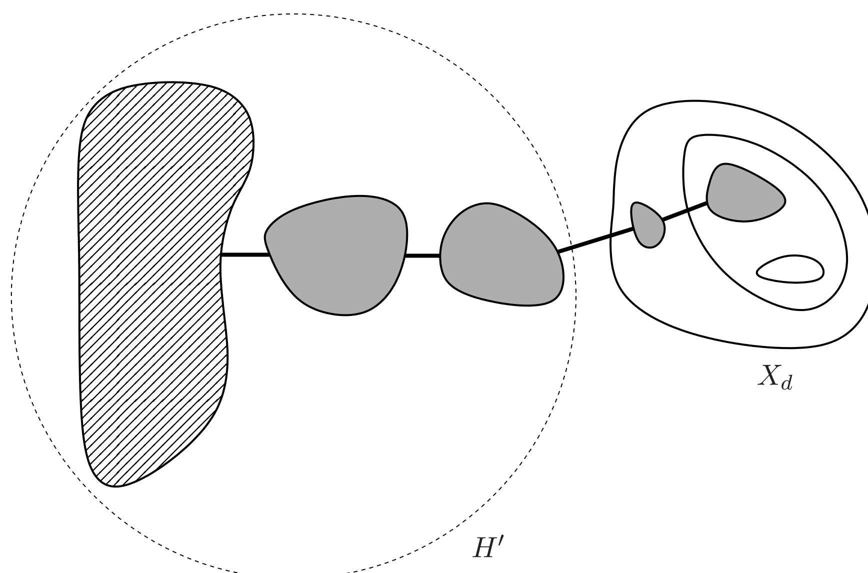


Figura 7.5: Conjunto hachurado corresponde à subárvore obtida após a poda. Conjuntos cinzas correspondem aos vértices adicionados para a árvore final ter exatamente k vértices.

Algoritmo PEGA-VÉRTICE$(H, H', S, v, \widetilde{k})$

1 sejam S_1 e S_2 os componentes que se juntaram para formar S.
  Suponha $v \in S_1$.
2 seja $u'v'$ a aresta entre S_1 e S_2 tal que $u' \in S_1$ e $v' \in S_2$.
3 se $|S_1| = \widetilde{k}$ então
4   $T' \leftarrow H[V_{H'} \cup S_1]$
5 senão se $|S_1| > \widetilde{k}$ então
6   $T' \leftarrow$ PEGA-VÉRTICE$(H, H', S_1, v, \widetilde{k})$
7 senão $T' \leftarrow$ PEGA-VÉRTICE$(H, H[V_{H'} \cup S_1], S_2, v', \widetilde{k} - |S_1|)$
8 devolva T'

A árvore T' obtida desta maneira satisfaz as hipóteses do teorema 7.7.

No algoritmo GARG-k-ÁRVORE esta etapa é tratada pela subrotina ETAPA1 que recebe F, F_{q+}, $\mathcal{L}_q$ e χ e devolve a árvore T', se ela for encontrada nesta etapa, ou uma árvore vazia.

Etapa 2

Para a transformação das colorações, note que χ_{q-} e χ_{q+} restritos a $\mathcal{L}_q$ diferem apenas nos conjuntos que têm cor *cinza* em χ_q. Tais conjuntos têm cor *preta* em χ_{q-} e cor *branca* em χ_{q+}. Logo, basta atribuir cor *branca* para tais conjuntos na coloração χ. Isso é possível, pois a diferença de cor se deve a uma diferença infinitesimal no valor da variável dual. Nesta etapa, os elementos de $\mathcal{L}_q$ também são considerados na ordem em que foram formados.

Considere que o passo atual ocorreu nesta etapa e foi responsável por F possuir uma árvore H com mais de k vértices.

Seja S o conjunto em $\mathcal{L}_q$ que teve sua cor mudada de *preta* para *branca* em χ. Sejam $X_1, X_2, \ldots, X_s$ os conjuntos maximais que seriam podados em F se $\chi(S)$ fosse *preta*. Note que $X_1, X_2, \ldots, X_s$ formam um caminho. Consideraremos que a ordem na qual esse caminho é percorrido é $X_1, X_2, \ldots, X_s = S$. Removendo todos esses conjuntos *pretos*, H fica com menos do que k vértices, então adicionamos vértices desses conjuntos à árvore utilizando a subrotina PEGA-VÉRTICE da mesma maneira que na etapa anterior. A árvore T' obtida desta maneira satisfaz as hipóteses do teorema 7.7.

No algoritmo GARG-k-ÁRVORE esta etapa é tratada pela subrotina ETAPA2 que recebe F, $\mathcal{L}_q$, χ e χ_{q+} e devolve a árvore T'.

Etapa 3

O processo de transformar $(F_{q-}, \mathcal{L}_q, \chi_{q-})$ em $(F_{q+}, \mathcal{L}_q, \chi_{q+})$ serve para obtermos uma árvore T'_q com custo $c(T'_q) \leq 2(kq - \Delta_0(Q'))$, onde $Q' \in \mathcal{L}_q$ é o conjunto minimal que contém $V_{T'_q}$, pois T'_q satisfaz as condições do teorema 7.7.

Se não existe $Q \in \mathcal{L}_q$ tal que $Q' \subset Q$ e $\Delta_0(Q) > \Delta_0(Q')$, então devolvemos T'_q. Caso contrário, a partir de T'_q, vamos obter uma árvore T com custo $c(T) \leq 2(kq - \Delta_0(Q))$, onde $Q \in \mathcal{L}$ é tal que $Q' \subset Q$ e $\Delta_0(Q)$ é máximo. Para encontrar T fazemos o seguinte.

Seja H a árvore com pelo menos k vértices na floresta $F[Q]$ após o processo de poda que ignora conjuntos que contêm Q'. Sabemos que H existe, pois Q' induz uma árvore com pelo menos k vértices. Como anteriormente, sejam $X_1, X_2, \ldots, X_s$, os conjuntos *pretos* maximais que são eliminados ao fazermos a poda de H em relação a χ.

Da mesma forma como anteriormente, utilizamos a subrotina PEGA-VÉRTICE para obter uma árvore T com exatamente k vértices. Seja S o conjunto minimal em $\mathcal{L}_q$ que contém T. Como T satisfaz as condições do teorema 7.7, $c(T) \leq 2(kq - \Delta_0(S))$. Ademais, $\Delta_0(S) \geq \Delta_0(Q)$, pois todos os vértices em $Q \setminus S$ estão cobertos por conjuntos *pretos*. Portanto $c(T) \leq 2(kq - \Delta_0(Q))$.

No algoritmo GARG-k-ÁRVORE esta etapa é tratada pela subrotina ETAPA3 que recebe F, $\mathcal{L}_q$ e χ e devolve a árvore T.

Algoritmo GARG-k-ÁRVORE

O algoritmo GARG-k-ÁRVORE recebe, como entrada, um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_\geq$, um número inteiro positivo k, a penalidade limiar q e devolve uma k-árvore T com

$$c(T) \leq 2(kq - \Delta_0(Q)),$$

onde Q é um conjunto em $\mathcal{L}_q$ tal que $V_T \subseteq Q$ e $\Delta_0(Q)$ é máximo.

Algoritmo GARG-k-ÁRVORE(G, c, k, q)

1 $F_{q-}, T_{q-}, y^{q-}, \mathcal{L}_{q-}, \chi_{q-} \leftarrow$ GW-MODIFICADO(G, c, q_-)
2 $F_q, T_q, y^q, \mathcal{L}_q, \chi_q \leftarrow$ GW-MODIFICADO(G, c, q)
3 $F_{q+}, T_{q+}, y^{q+}, \mathcal{L}_{q+}, \chi_{q+} \leftarrow$ GW-MODIFICADO(G, c, q_+)
4 se $|V_{T_{q+}}| = k$ então devolva T_{q+}
5 $F \leftarrow F_{q-}$ $\chi \leftarrow \chi_{q-}$
6 $T' \leftarrow$ ETAPA1$(F, F_{q+}, \mathcal{L}_q, \chi)$
7 se $T' = \emptyset$ então $T' \leftarrow$ ETAPA2$(F, \mathcal{L}_q, \chi, \chi_{q+})$
8 $T \leftarrow$ ETAPA3$(F, \mathcal{L}_q, \chi)$
9 devolva T

Note que se T_{q+} tem exatamente k vértices, então a devolvemos. Isso porque neste caso, como vimos na seção 6.3, T_{q+} é uma 2-aproximação para kMST. O consumo de tempo do algoritmo GARG-k-ÁRVORE é dominado pelo consumo de GW-MODIFICADO. Como a subrotina GW-MODIFICADO é executada $O(|V|)$ vezes, temos que GARG-k-ÁRVORE consome $O(|V_G|^3 \log |V_G|)$.

7.8 Descrição do algoritmo

Descrevemos nesta seção o algoritmo GARG2 que recebe um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_{\geq}$ e um número inteiro k, $k > 1$, e devolve uma k-árvore. Se $k > |V_G|$, então o problema é inviável e o algoritmo devolve a árvore vazia. Como veremos, este algoritmo é uma 2-aproximação para o problema kMST.

O algoritmo GARG2 utiliza as subrotinas GW-MODIFICADO(seção 7.3), GARG-LIMIAR(seção 7.6) e GARG-k-ÁRVORE(seção 7.7).

A subrotina GARG-LIMIAR recebe um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_{\geq}$ e um número inteiro positivo k, $k > 1$ e devolve a penalidade limiar q.

Sejam $F_{q+}, T_{q+}, y^{q+}, \mathcal{L}_{q+}$ e χ_{q+} os objetos devolvidos ao invocarmos a subrotina GW-MODIFICADO (G, c, q_+). A subrotina GARG-k-ÁRVORE recebe um grafo conexo G, uma função custo $c : E_G \to \mathbb{Q}_{\geq}$, um número inteiro positivo k, a penalidade limiar q e devolve uma k-árvore T tal que

$$c(T) \leq 2(kq - \Delta_0(Q))\,, \tag{7.8}$$

onde Q é um conjunto em $\mathcal{L}_{q+}$ tal que $V_T \subseteq Q$.

Algoritmo GARG2(G, c, k)

1 se $|V_G| < k$ então
2 devolva $\emptyset$ ▷ árvore vazia
3 senão $q \leftarrow$ GARG-LIMIAR(G, c, k)
4 $F_{q+}, T_{q+}, y^{q+}, \mathcal{L}_{q+}, \chi_{q+} \leftarrow$ GW-MODIFICADO(G, c, q_+)
5 $T_{V_G} \leftarrow$ GARG-k-ÁRVORE(G, c, k, q)
6 $Q \leftarrow L$ em $\mathcal{L}_{q+}$ que contém os vértices de T_{V_G} e $\Delta_0(L)$ é máximo
7 $\mathcal{M} \leftarrow \{L \in \mathcal{L}_{q+} : \Delta_0(L) > \Delta_0(Q)\}$
8 para cada $M \in \mathcal{M}^*$ faça[4]
9 $T_M \leftarrow$ GARG2$(G[M], c, k)$
10 $T \leftarrow k$-árvore de custo mínimo dentre T_{V_G} e T_M com $M \in \mathcal{M}^*$
11 devolva T

[4]Lembrete: o sinal * em uma coleção denota a subcoleção dos elemento maximais.

Na linha 9 do algoritmo GARG2 a função custo c considerada é na verdade a restrição de c às arestas do grafo $G[M]$.

Devido às especificações das subrotinas utilizadas, é evidente que se a instância considerada é viável o algoritmo devolve uma k-árvore. Precisamos mostrar que a k-árvore devolvida é uma 2-aproximação para o kMST.

Considere, na linha 4, os objetos devolvidos ao invocarmos a subrotina GW-MODIFICADO (G, c, q_+).

Pela linha 6 do algoritmo, Q é um conjunto em $\mathcal{L}_{q+}$ com $\Delta_0(Q)$ máximo. Logo, se $\mathcal{M} = \{L \in \mathcal{L} : \Delta_0(L) > \Delta_0(Q)\}$ é a coleção definida na linha 7, então $Q \cap M = \emptyset$ para toda parte M em $\mathcal{M}^*$. Assim,

$$\sum_{M \in \mathcal{M}^*} |M| \leq |V_G| - |Q| \leq |V_G| - k\,,$$

onde a segunda desigualdade é devido ao fato de que os vértices de T_{V_G} estão em Q. Ademais, como todos os elementos em $\mathcal{M}^*$ são disjuntos, temos que o número total de invocações recursivas feitas por GARG2(G, c, k) é no máximo $|V_G| - k$ e portanto o algoritmo pára.

Seja T^* uma k-árvore de custo mínimo e seja L^* um elemento minimal em $\mathcal{L}_{q+}$ que contém os vértices dessa k-árvore. A subrotina GW-MODIFICADO garante que V_G está em $\mathcal{L}_{q+}$, logo existe um tal L^*.

Pela especificação da subrotina GARG-k-ÁRVORE, temos que a k-árvore T_{V_G} obtida na linha 5 é tal que

$$c(T_{V_G}) \leq 2(kq - \Delta_0(Q))\,. \tag{7.9}$$

Por outro lado, devido ao lema da dualidade 7.1 temos que

$$c(T^*) \geq kq - (\pi(L^*) - y(\mathcal{L}(L^*))) = kq - \Delta_0(L^*)\,. \tag{7.10}$$

Assim, se $\Delta_0(L^*) \leq \Delta_0(Q)$ temos que, combinando (7.9) e (7.10),

$$\begin{aligned} c(T_{V_G}) &\leq 2(kq - \Delta_0(Q)) \leq 2(kq - \Delta_0(L^*)) \\ &\leq 2\,c(T^*) = 2\,\mathrm{OPT}(G, c, k). \end{aligned} \tag{7.11}$$

Dessa forma, T_{V_G} é uma 2-aproximação para o kMST e o algoritmo faz o que promete.

Agora, vamos supor que $\Delta_0(L^*) > \Delta_0(Q)$. Consideremos a coleção $\mathcal{M}$ definida na linha 7. Claramente, L^* está em $\mathcal{M}$. Seja M^* o elemento de $\mathcal{M}^*$ que contém L^*. Como $G[M^*]$ contém uma k-árvore mínima de G, uma 2-aproximação em $G[M^*]$ também é uma 2-aproximação em G.

O raciocínio acima é aplicável a cada uma das invocações recursivas, na linha 9, de GARG2$(G[M], c, k)$ para cada M em $\mathcal{M}^*$. O ponto crucial aqui é que em cada nível da recursão vale pelo menos uma das seguintes afirmações:

(g1) ou a parte Q definida na linha 6 é tal que $\Delta_0(L^*) \leq \Delta_0(Q)$ e, portanto, T_{V_G} é uma 2-aproximação

(g2) ou L^* está em $\mathcal{M}$ e o algoritmo realiza as invocações recursivas na linha 9.

Como o algoritmo pára, então (g1) ocorre em algum nível da recursão para cada M em $\mathcal{M}^*$. Portanto, para cada M em $\mathcal{M}^*$ o algoritmo encontra uma k-árvore T_M tal que

$$c(T_M) \leq 2\,\mathrm{OPT}(G[M], c, k)\,.$$

Como M^* está em $\mathcal{M}^*$, então

$$c(T) \leq c(T_{M^*}) \leq 2\,\mathrm{OPT}(G[M^*], c, k) = 2\,\mathrm{OPT}(G, c, k)\,,$$

onde a igualdade é devida ao fato de que $V_{T^*} \subseteq M^*$. Assim, novamente, o algoritmo faz o que promete.

O número total de invocações recursivas feitas pelo algoritmo GARG2 é no máximo $|V_G| - k$. Assim, os consumos total de tempo de todas as execuções da subrotina GARG-LIMIAR, GARG-k-ÁRVORE e GW-MODIFICADO são $\mathrm{O}(|E_G|\,|V_G|^4 \log |V_G|)$, $\mathrm{O}(|V_G|^4 \log |V_G|)$ e $\mathrm{O}(|V_G|^3 \log |V_G|)$, respectivamente. Portanto, o consumo de tempo do algoritmo GARG2 é $\mathrm{O}(|E_G|\,|V_G|^4 \log |V_G|)$.

Versão iterativa do algoritmo GARG2

O algoritmo de Garg que acabamos de ver é recursivo. Podemos reescrevê-lo em uma versão iterativa. A diferença das duas versões é mais no sentido de implementação, pois as idéias envolvidas são as mesmas.

Algoritmo GARG2-ITERATIVO(G, c, k)

1 $\mathcal{M} \leftarrow \{V_G\} \quad \mathcal{T} \leftarrow \emptyset$
2 enquanto $\mathcal{M} \neq \emptyset$ faça
3 seja M um elemento qualquer de $\mathcal{M}$
4 $\mathcal{M} \leftarrow \mathcal{M} \setminus M$
5 $q \leftarrow$ GARG-LIMIAR$(G[M], c, k)$
6 $F_{q+}, T_{q+}, y^{q+}, \mathcal{L}_{q+}, \chi_{q+} \leftarrow$ GW-MODIFICADO$(G[M], c, q_+)$
7 $Q \leftarrow L$ em $\mathcal{L}_{q+}$ tal que $V_{T_{q+}} \subseteq L$ e $\Delta_0(L)$ é máximo.
8 $T_M \leftarrow$ GARG-k-ÁRVORE$(G[M], c, k, q, Q)$
9 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T_M\}$
10 $\mathcal{M} \leftarrow \mathcal{M} \cup \{L \in \mathcal{L}_{q+} : |L| \geq k,\ \Delta_0(L) > \Delta_0(Q)\}^*$
11 $T \leftarrow$ árvore de custo mínimo dentre $\mathcal{T}$
12 devolva T

Nas linhas 6 e 8 do algoritmo GARG2-ITERATIVO a função custo c considerada é na verdade a restrição de c às arestas do grafo $G[M]$.

CAPÍTULO 8

Considerações finais

Apresentamos vários algoritmos de aproximação para o problema kMST. Entre eles estão alguns dos primeiros algoritmos publicados e alguns mais recentes. Alguns dos algoritmos mais simples, como os apresentados nas seções 4.2, 4.4, 5.1 e 5.2, não aparecem em publicações e estão presentes para aumentar o entendimento sobre o problema. Em cada um desses algoritmos vemos variadas técnicas de desenvolvimento de algoritmos e de demonstração do fator de aproximação.

Tentamos descrever os algoritmos de maneira clara e de fácil entendimento. Isso porque notamos que muitos dos artigos estudados se mostravam mais complicados do que o necessário. Podemos destacar o artigo de Garg [Gar05b], no qual é apresentado uma 2-aproximação para o kMST. Grande parte do tempo dedicado a esta dissertação foi gasto tentando entender esse artigo. Inclusive, foi necessário consultar o autor, pois encontramos um contra-exemplo para uma de suas afirmações. Esse contra-exemplo pode ser encontrado no apêndice B.

Apesar de não ter sido mencionado neste texto, uma variante do kMST muito estudada é o kMST-EUCLIDIANO. Nesta variante, os vértices do grafo correspondem a pontos no plano e o custo das arestas é a distância euclidiana entre suas pontas. O kMST-EUCLIDIANO também é um problema NP-difícil.

APÊNDICE A

Simulação de GW-MODIFICADO

Aqui mostraremos algumas simulações do algoritmo GW-MODIFICADO.

A.1 Execução de GW-MODIFICADO

Exemplificamos aqui o funcionamento do algoritmo GW-MODIFICADO através da simulação ilustrada nas figuras a seguir. O grafo que será utilizado na simulação é completo e será dado através do desenho no plano de seus vértices. Inicialmente, somente os vértices serão exibidos na simulação e as arestas estarão implícitas. O custo de cada aresta será a distância euclidiana entre as suas pontas no desenho. As únicas arestas que, à medida que a execução do algoritmo avança, serão exibidas por linhas entre as suas pontas são aquelas que estão na floresta F.

Na ilustração da simulação a seguir supomos que as penalidades de cada vértice são "convenientemente" grandes e que o único conjunto que será saturado por y na última iteração antes da poda será V_G. Assim, durante toda a simulação não teremos conjuntos pintados com a cor *cinza* ou com a cor *preta*. Portanto, a árvore devolvida pelo algoritmo será uma árvore geradora mínima (MST), já que nesta situação o algoritmo GW-MODIFICADO tem o mesmo comportamento que o algoritmo de Kruskal para encontrar uma MST, como pode ser facilmente verificado.

Inicialmente os círculos, e posteriormente os aglomerados de círculos ao redor de um conjunto de vértices, indicam os conjuntos em $\mathcal{L}$. A soma das larguras das faixas desses aglomerados ao redor de um mesmo conjunto $L \in \mathcal{L}$ indica o valor de y_L.

No início da primeira iteração do bloco de linhas 5–17 vale que:

$$\begin{aligned} E_F &= \emptyset, \\ \mathcal{L} &= \{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\}\}, \\ \mathcal{A} &= \{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\}\}. \end{aligned}$$

Na primeira iteração a aresta bc fica justa já que os círculos contendo b e c se tocam devido ao valor de ε calculado na linha 6 e portanto, na linha 8, $y'_{\{b\}} + y'_{\{c\}} = c_{bc}$. A aresta bc é então inserida em F e no início da segunda iteração temos que:

$$\begin{aligned} E_F &= \{\mathbf{bc}\}, \\ \mathcal{L} &= \{\{a\},\dots,\{h\},\{\mathbf{b},\mathbf{c}\}\}, \\ \mathcal{A} &= \{\{a\},\{\mathbf{b},\mathbf{c}\},\{d\},\{e\},\{f\},\{g\},\{h\}\} \end{aligned}$$

e $\{b\}$ e $\{c\}$ são pintados com a cor *branca*.

Na segunda iteração é a vez da aresta de ficar justa por y', ou seja, na linha 8 teremos que $y'_{\{d\}} + y'_{\{e\}} = c_{de}$. No início da terceira iteração temos que:

$$\begin{aligned} E_F &= \{bc,\mathbf{de}\}, \\ \mathcal{L} &= \{\{a\},\dots,\{h\},\{b,c\},\{\mathbf{d},\mathbf{e}\}\}, \\ \mathcal{A} &= \{\{a\},\{b,c\},\{\mathbf{d},\mathbf{e}\},\{f\},\{g\},\{h\}\} \end{aligned}$$

e $\{d\}$ e $\{e\}$ são pintados com a cor *branca*. Na terceira iteração a aresta que fica justa é fh, e no início da quarta iteração vale que:

$$\begin{aligned} E_F &= \{bc,de,\mathbf{fh}\}, \\ \mathcal{L} &= \{\{a\},\dots,\{h\},\{b,c\},\{d,e\},\{\mathbf{f},\mathbf{h}\}\}, \\ \mathcal{A} &= \{\{a\},\{b,c\},\{d,e\},\{\mathbf{f},\mathbf{h}\},\{g\}\} \end{aligned}$$

e $\{f\}$ e $\{h\}$ são pintado com a cor *branca*.

Na quarta iteração a aresta ef fica justa por y', pois $y'_{\{e\}} + y'_{\{d,e\}} + y'_{\{f\}} + y'_{\{f,h\}} = c_{ef}$. No início da quinta iteração vale que:

$$\begin{aligned} E_F &= \{bc,de,fh,\mathbf{ef}\}, \\ \mathcal{L} &= \{\{a\},\dots,\{h\},\{b,c\},\{d,e\},\{f,h\},\{\mathbf{d},\mathbf{e},\mathbf{f},\mathbf{h}\}\}, \\ \mathcal{A} &= \{\{a\},\{b,c\},\{\mathbf{d},\mathbf{e},\mathbf{f},\mathbf{h}\},\{g\}\} \end{aligned}$$

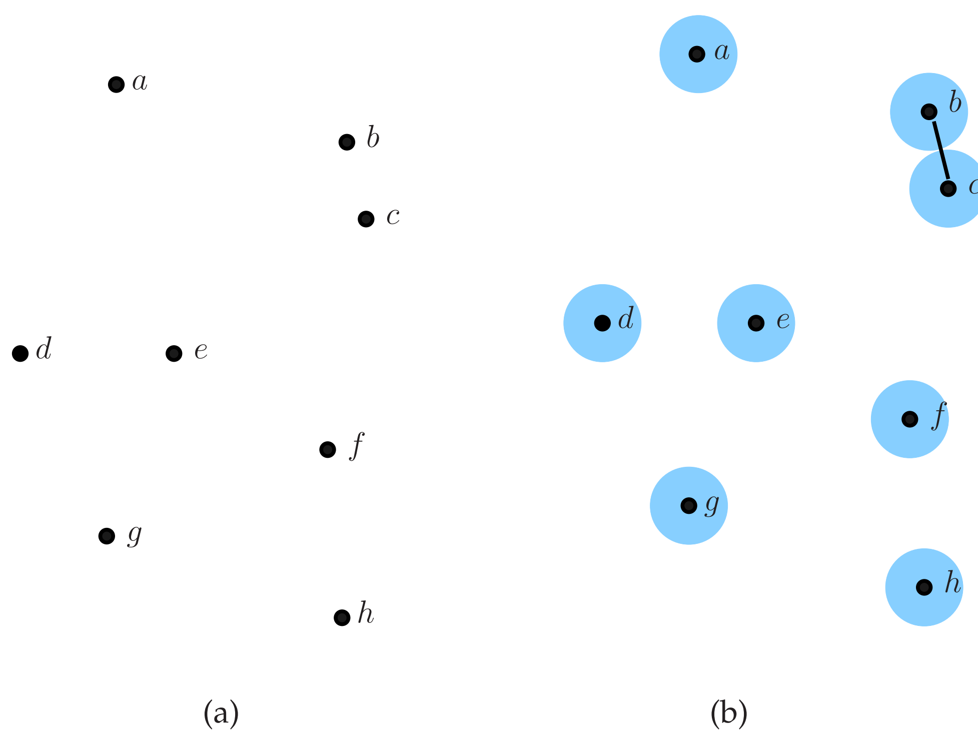
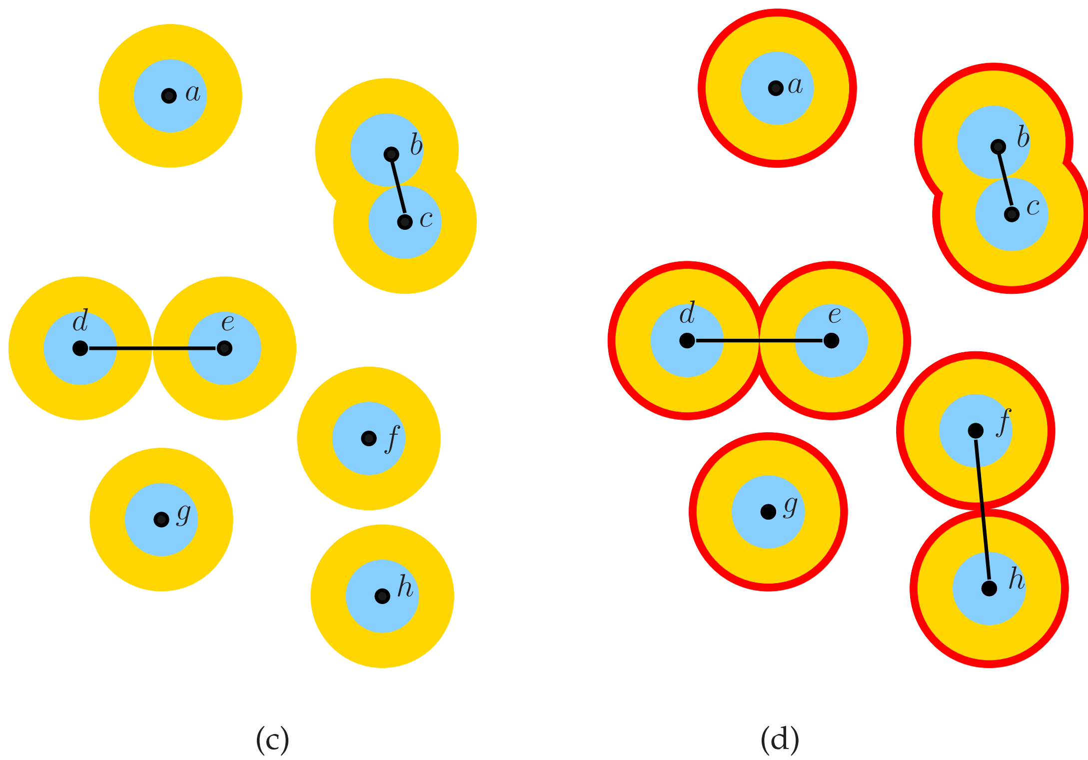


Figura A.1: A figura (a) mostra o grafo G antes da primeira iteração. A situação no início da segunda iteração é mostrada na figura (b).



(c) (d)

Figura A.2: A figura (c) mostra o início da terceira iteração e a figura (d) o início da quarta iteração.

e $\{d, e\}$ e $\{f, h\}$ são pintados com a cor *branca*. Na a quinta iteração a aresta que fica justa é eg, pois $y'_{\{e\}} + y'_{\{d,e\}} + y'_{\{d,e,f,h\}} + y'_{\{g\}} = c_{eg}$, e no início da sexta iteração vale que:

$$\begin{aligned} E_F &= \{bc, de, fh, ef, \mathbf{eg}\}, \\ \mathcal{L} &= \{\{a\}, \ldots, \{h\}, \{b, c\}, \{d, e\}, \{f, h\}, \{d, e, f, h\}, \{\mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}\}, \\ \mathcal{A} &= \{\{a\}, \{b, c\}, \{\mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{g}, \mathbf{h}\}\} \end{aligned}$$

e $\{d, e, f, h\}$ e $\{g\}$ são pintados com a cor *branca*.

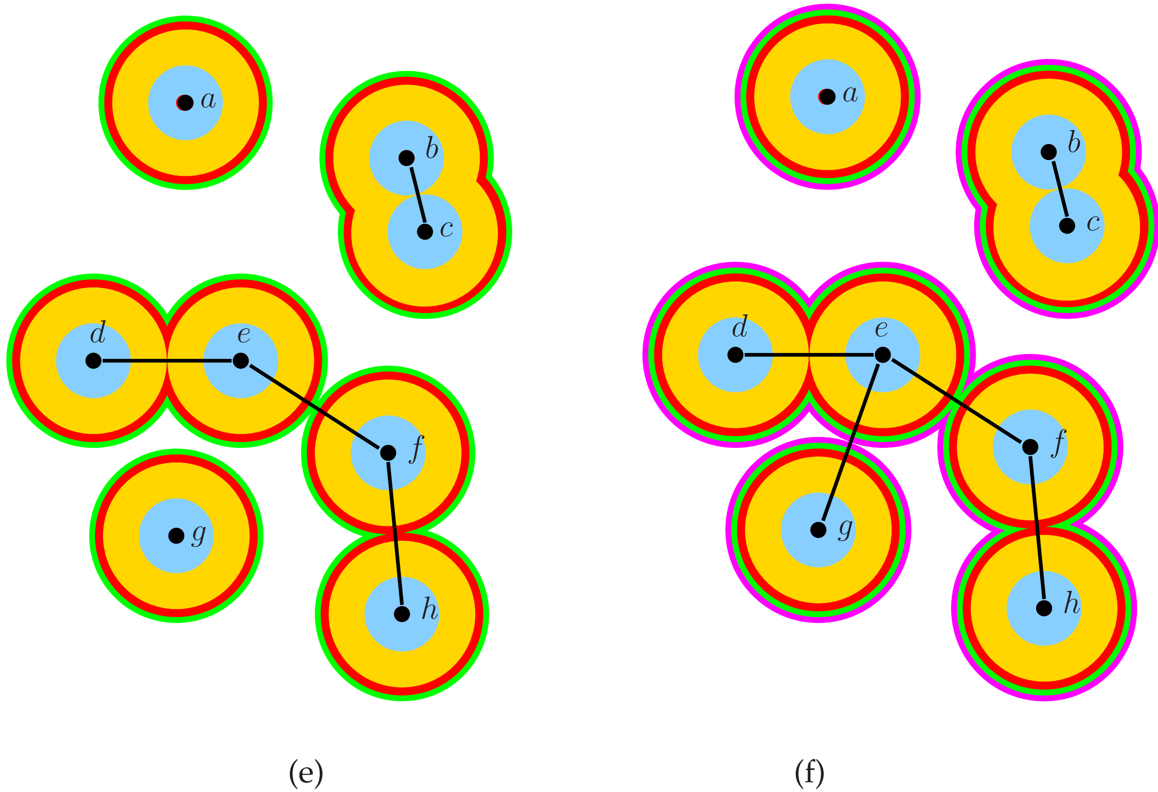


Figura A.3: A figura (e) mostra o início da quinta iteração e (f) o início da sexta iteração.

A aresta que ficou justa na sexta iteração é cf, pois $y'_{\{c\}} + y'_{\{b,c\}} + y'_{\{f\}} + y'_{\{f,h\}} + y'_{\{d,e,f,h\}} + y'_{\{d,e,f,g,h\}} = c_{cf}$. No início da sétima iteração vale que:

$$\begin{aligned} E_F &= \{bc, de, fh, ef, eg, \mathbf{cf}\}, \\ \mathcal{L} &= \{\{a\}, \ldots, \{h\}, \{b, c\}, \{d, e\}, \{f, h\}, \{d, e, f, h\}, \{d, \ldots, h\}, \{\mathbf{b}, \ldots, \mathbf{h}\}\}, \\ \mathcal{A} &= \{\{a\}, \{\mathbf{b}, \ldots, \mathbf{h}\}\} \end{aligned}$$

e $\{b, c\}$ e $\{d, e, f, g, h\}$ são pintados com a cor *branca*.

No início da última iteração temos que:

$$E_F = \{bc, de, fh, ef, eg, cf, \mathbf{ab}\},$$
$$\mathcal{L} = \{\{a\}, \ldots, \{h\}, \{b, c\}, \{d, e\}, \{f, h\}, \{d, e, f, h\}, \{d, \ldots, h\}, \{\mathbf{b}, \ldots, \mathbf{h}\}, \{a, \ldots, h\}\},$$
$$\mathcal{A} = \{\{\mathbf{a}, \ldots, \mathbf{h}\}\}$$

e $\{a\}$ e $\{b, \ldots, h\}$ são pintados com a cor *branca*.

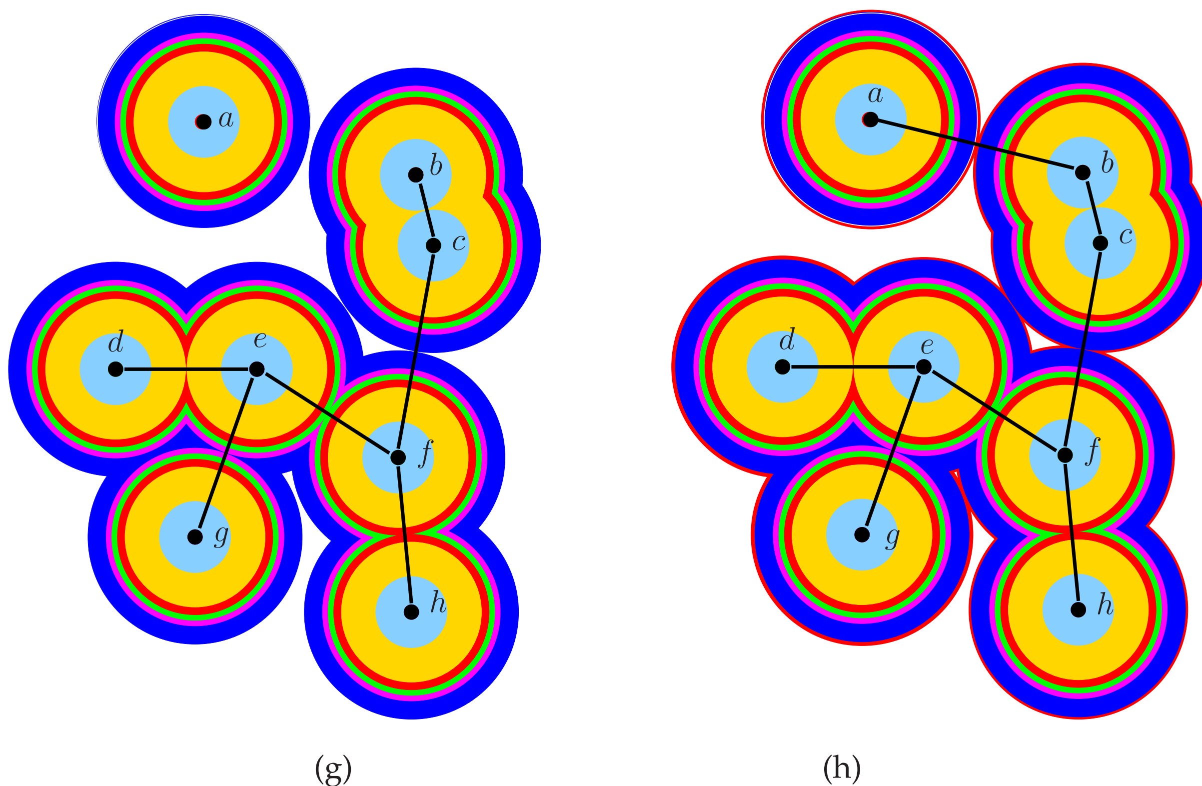


Figura A.4: A figura (g) mostra o início da sétima iteração e (f) o início da última.

Como foi suposto que as penalidades eram convenientemente grandes, o único conjunto saturado por y é V_G. A árvore devolvida, que é $\mathcal{L}$-conexa, está na figura A.5.

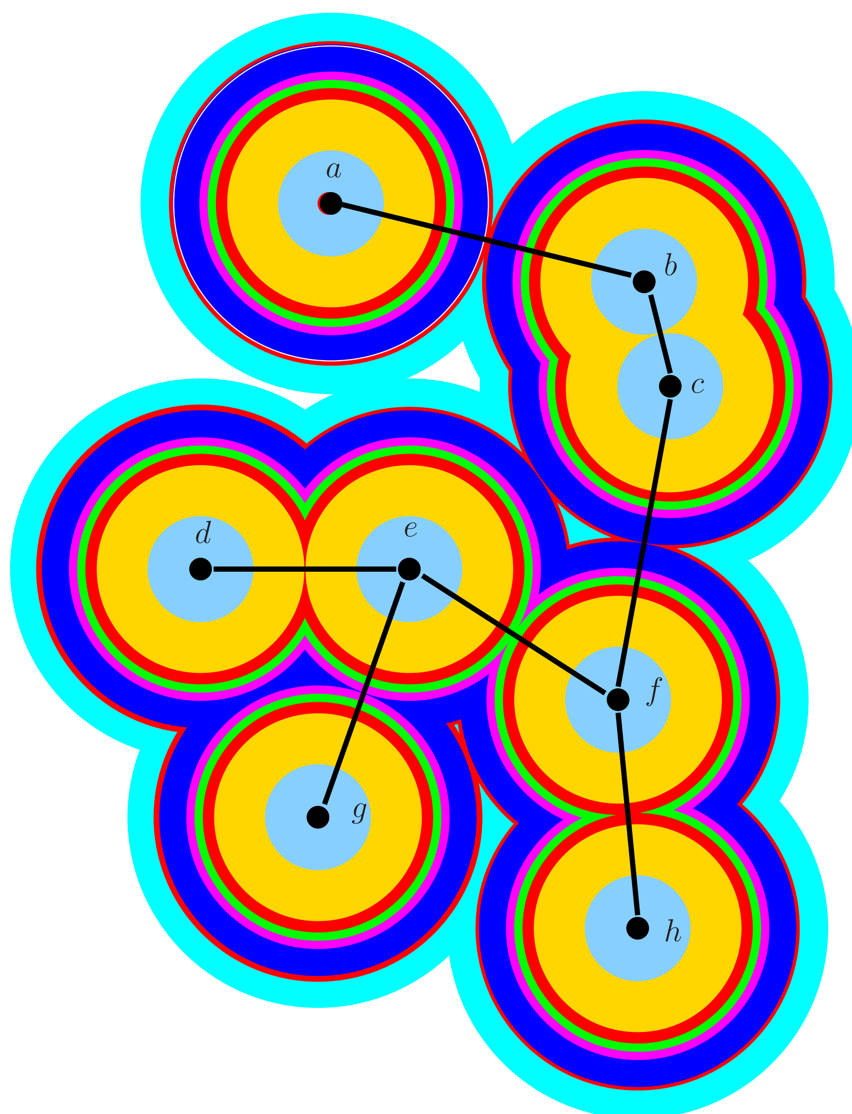


Figura A.5: Árvore devolvida por GW-MODIFICADO.

A.2 Custos reduzidos e potenciais

Consideremos agora o seguinte exemplo de uma invocação de GW-MODIFICADO. Nesta ilustração, como na seção anterior, o grafo é completo e o custo de cada aresta é a distância entre suas pontas. O grafo tem 4 vértices e a penalidade inicial em cada vértice é 3. Esta ilustração foi criada por Garg e apresentada por Julian Mestre em uma palestra sobre o algoritmo de Garg [Gar05a]. Esta ilustração mostra como a árvore devolvida pode variar com pequenas alterações na penalidade inicial q de cada vértice.

Utilizamos nesta ilustração a metáfora de "instante em que um evento ocorre". Encaramos a subrotina GW-MODIFICADO como se o valor de ε computado na linha 5 fosse sempre 1 e, nesse caso, a variável t da linha 6 é incrementada de 1 em 1.

No instante $t = 0$ temos que $\mathcal{L} = \{a, b, c, d\}$, $\mathcal{A} = \{\{a\}, \{b\}, \{c\}, \{d\}\}$, $\hat{c}_e = c_e$ para cada aresta e, $\Delta(\{x\}) = 3$ para cada vértice x.

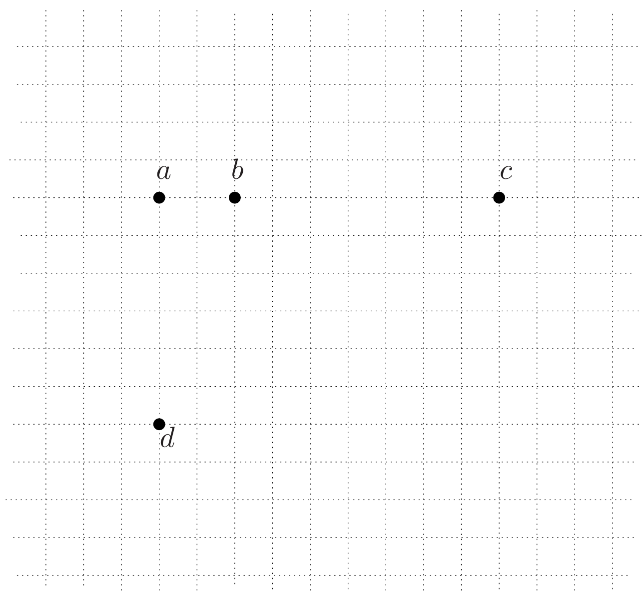


Figura A.6: Situação inicial.

Já, no instante $t = 1$ a aresta ab fica justa e é acrescentada a F, o conjunto $\{a, b\}$ passa a fazer parte de $\mathcal{L}$ e $\{a\}$ e $\{b\}$ são pintados com a cor *branca* e $\mathcal{A} = \{\{\mathbf{a}, \mathbf{b}\}, \{c\}, \{d\}\}$. Temos ainda que $\hat{c}_{\{ad\}} = 4, \hat{c}_{\{bc\}} = 5, \Delta(\{a, b\}) = 4, \Delta(\{c\}) = \Delta(\{d\}) = 2$.

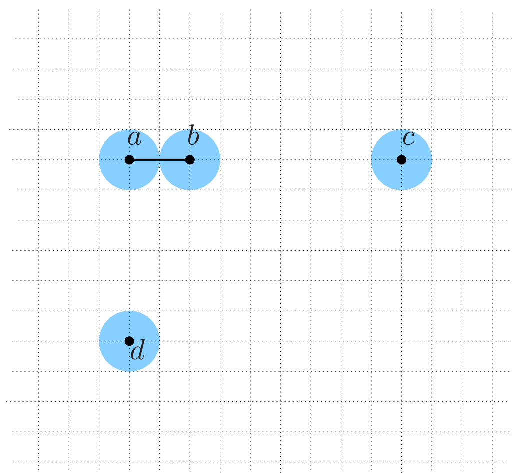


Figura A.7: Instante $t = 1$.

No instante $t = 2$ a situação não se altera muito. Apenas $y_{\{a,b\}}, y_{\{c\}}$ e $y_{\{d\}}$ são incrementados de 1. Assim, $\hat{c}_{\{ad\}} = 2$, $\hat{c}(_{\{bc\}} = 3$, $\Delta(\{a, b\}) = 3$, $\Delta(\{c\}) = \Delta(\{d\}) = 1$.

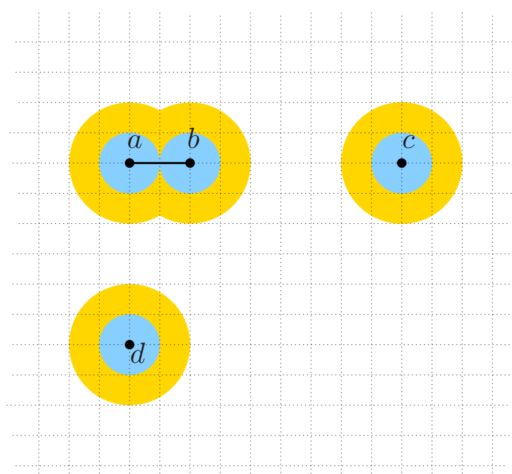


Figura A.8: Instante $t = 2$.

No instante $t = 3$ a aresta ad fica justa e o conjunto $\{c\}$ fica saturado. Teremos assim que $\{a, b\}$ é pintado com a cor *branca* e $\{d\}$ com a cor *cinza*. Além disso, $\mathcal{A} = \{\{\mathbf{a}, \mathbf{b}, \mathbf{d}\}\}$ $\hat{c}_{\{bc\}} = 1, \Delta(\{a, b, d\}) = 2$ e $\Delta(\{c\}) = 0$.

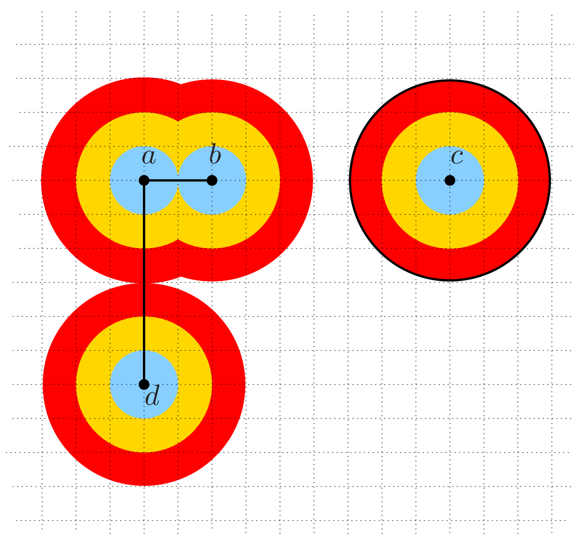


Figura A.9: Instante $t = 3$.

No instante $t = 4$ o valor de $y_{\{a,b,d\}}$ é acrescido de 1, como ilustra a faixa verde ao redor dos vértices $\{a, b, d\}$ e a aresta bc fica justa. Assim, pintamos $\{a, b, d\}$ com a cor *branca* e $\{c\}$ com a cor *cinza*, $\mathcal{A} = \{\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}\}$ e $\Delta(\{a, b, c, d\}) = 1$.

Finalmente, no instante $t = 5$, o valor de $y_{\{a,b,c,d\}}$ é incrementado de 1, o conjunto V_G é saturado por y e conseqüentemente $\mathcal{A} = \emptyset$ e $\Delta(\{a, b, c, d\}) = 0$. A situação atual é $\{a\}, \{b\}, \{a, b\}, \{a, b, d\}$ têm a cor *branca*, $\{d\}$ tem a cor *cinza*, $\{c\}$ tem a cor *preta*, $y_{\{a\}} = y_{\{b\}} = 1$, $y_{\{c\}} = y_{\{d\}} = 3$, $y_{\{a,b\}} = 2$, $y_{\{a,b,d\}} = 1$, e $y_{\{a,b,c,d\}} = 1$.

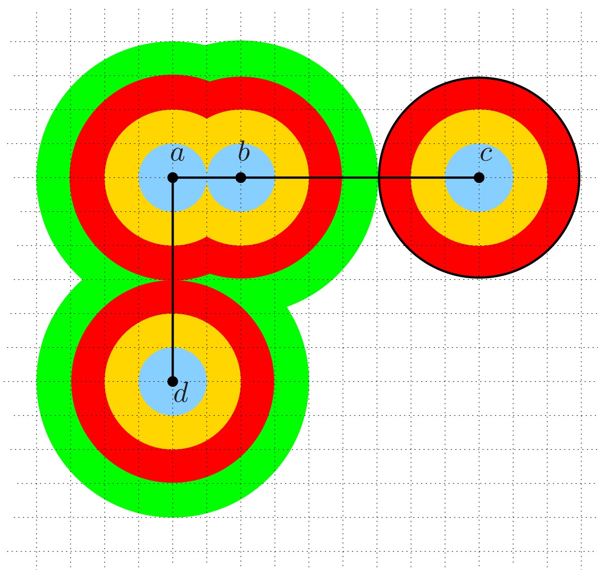


Figura A.10: Instante $t = 4$.

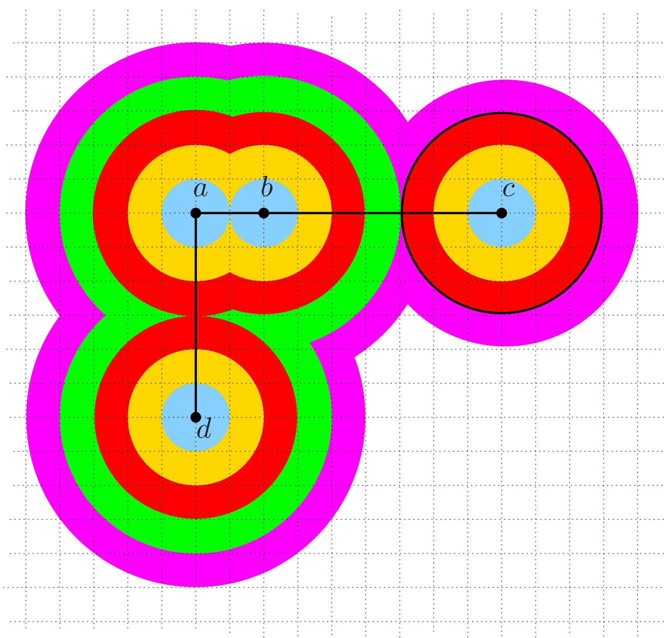


Figura A.11: Instante $t = 5$.

Se o valor da penalidade inicial q de cada vértice fosse um pouco menor que 3 teríamos que o conjunto $\{d\}$ ficaria saturado por y antes da aresta ad ficar justa. Nesse caso, na situação ao final da primeira fase, teríamos que não há conjunto com a cor *cinza* e $\{c\}$ e $\{d\}$ têm a cor *preta*. Os conjuntos com a cor *preta* são indicado na figura A.12 através de uma circunferência tracejada de cor preta.

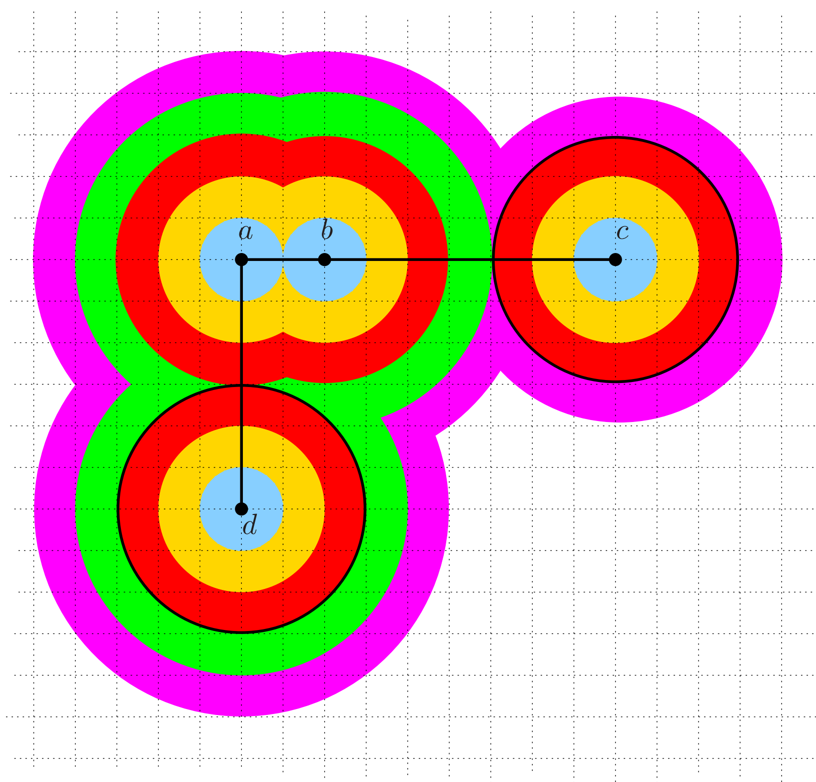


Figura A.12: Exemplo do caso em que o valor de q é um pouco menor que 3.

A figura A.13 mostra a árvore devolvida pela subrotina GW-MODIFICADO com $q = 3$ e a árvore devolvida para um valor de q infinitesimalmente menor do que 3. A fase de poda faz com que as árvores devolvidas sejam diferentes devido à existência de diferentes conjuntos com a cor *preta* nas duas situações. Os conjuntos com a cor *preta* são indicados por circunferências tracejadas pretas.

Como ilustra ainda a figura A.14, mudanças pequenas na penalidade de cada vértice fazem que com que a floresta F mude consideravelmente. Na figura, vemos que se o valor da penalidade q de cada vértice for um pouco maior que 3, então o conjunto

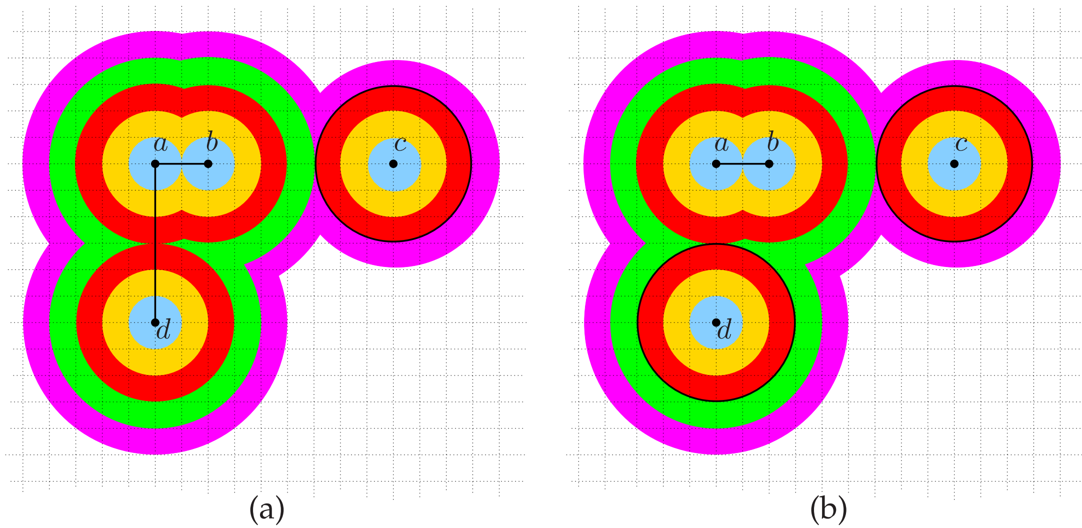


Figura A.13: (a) árvore devolvida quando $q = 3$. (b) árvore devolvida quando q é um pouco menor que 3.

$\{c\}$ fica saturado um pouco mais tarde. Conseqüentemente, a aresta bc fica justa antes da aresta de e é inserida na floresta F, em vez da aresta de. Mas, se o valor de q for um pouco menor do que 3, a aresta de fica justa antes da aresta bc e é inserida na floresta, em vez da aresta bc.

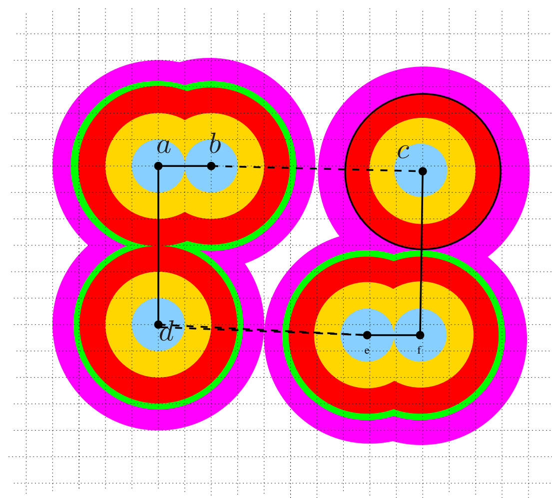


Figura A.14: Exemplo no qual pequena mudança no valor da penalidade pode mudar a árvore devolvida pelo algoritmo.

APÊNDICE B

Contra-exemplo

No artigo de Garg [Gar05b], a subrotina GARG-k-ÁRVORE, diferentemente da forma apresentada na seção 7.7, transforma a tupla $(\mathcal{L}_{q_-}, \chi_{q_-}, F_{q_-})$ em $(\mathcal{L}_{q_+}, \chi_{q_+}, F_{q_+})$. Logo no início da descrição da subrotina (seção 4 do artigo), o autor escreve o seguinte:

> *"Note that the collections $\mathcal{S}_{q+}$, $\mathcal{S}_{q-}$ differ only in sets which have a zero initial potential in* `modified-GW(q)`*. Such sets would be part of $\mathcal{S}_{q+}$ but not of $\mathcal{S}_{q-}$."*

Ou seja, ele afirma que as coleções $\mathcal{L}_{q_-}$ e $\mathcal{L}_{q_+}$ diferem apenas em conjuntos que têm Δ_0 igual a zero em GW-MODIFICADO(G, c, q), onde q é uma penalidade limiar. Ademais, tais conjuntos seriam parte de $\mathcal{L}_{q_+}$, mas não de $\mathcal{L}_{q_-}$. Esse fato é usado para argumentar que inserindo tais conjuntos em $\mathcal{L}$, a coleção continua laminar.

No entanto, no exemplo da figura B.1, temos que podem existir conjuntos em $\mathcal{L}_{q_-}$ que não estão em $\mathcal{L}_{q_+}$. Neste exemplo consideramos que $k = 5$. A penalidade limiar é 4. Como algumas arestas ficam justas simultaneamente, as linhas tracejadas indicam a ordem considerada para elas ficarem justas. No final, temos:

$$\begin{aligned}
\mathcal{L}_{q_-} &= \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_1, v_2\}, \{v_1, v_2, v_3\}, \{v_1, v_2, v_3, v_4\}, \{v_1, v_2, v_3, v_4, v_5\}\} \\
\mathcal{L}_{q} &= \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_1, v_2\}, \{v_1, v_2, v_3, v_4, v_5\}\} \\
\mathcal{L}_{q_+} &= \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_1, v_2\}, \{v_3, v_4\}, \{v_3, v_4, v_5\}, \{v_1, v_2, v_3, v_4, v_5\}\}.
\end{aligned}$$

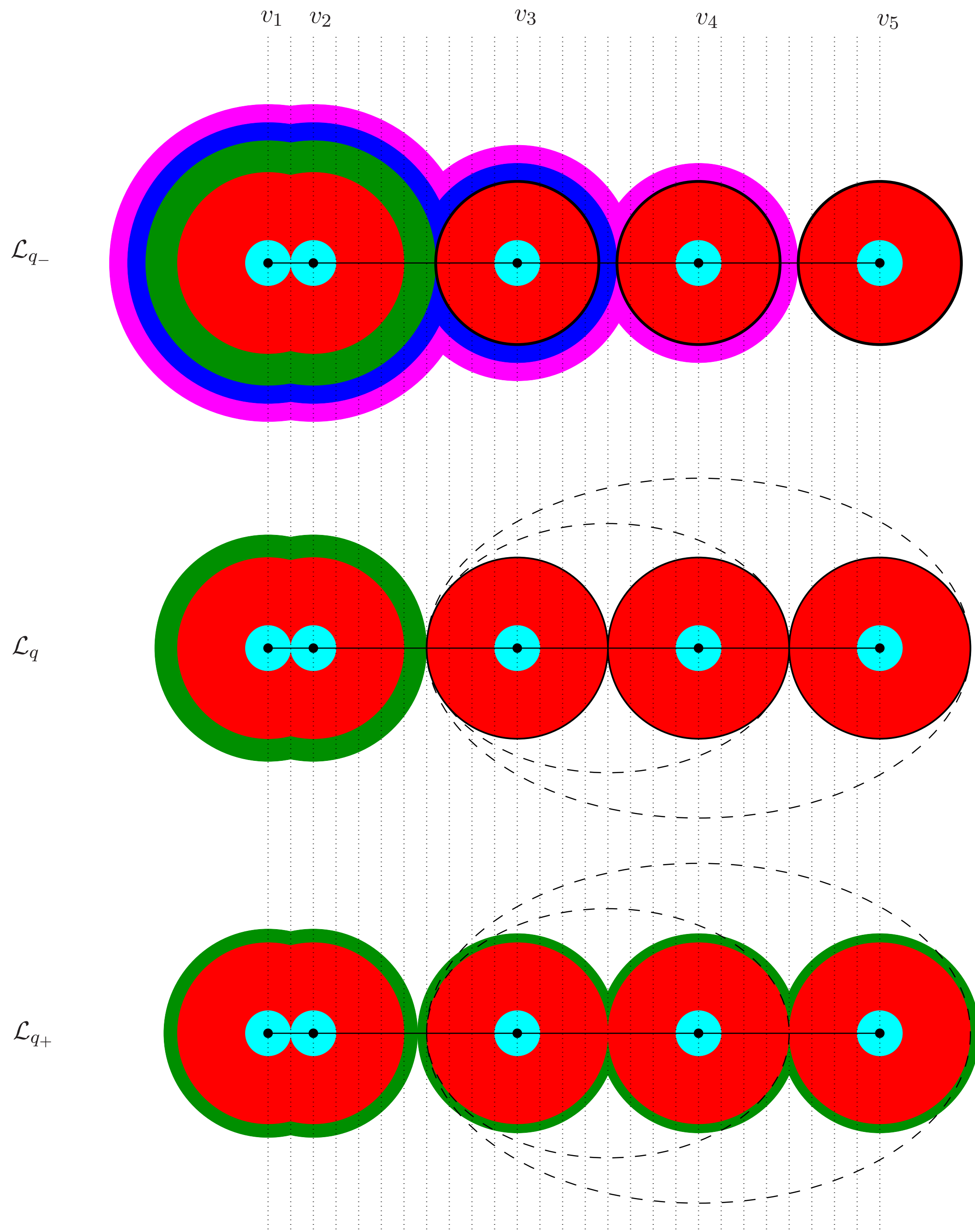


Figura B.1: Contra-exemplo com $k = 5$.

Referências Bibliográficas

[AABV98] Baruch Awerbuch, Yossi Azar, Avrim Blum e Santosh Vempala, *New Approximation Guarantees for Minimum-Weight k-Trees and Prize-Collecting Salesmen*, SIAM Journal on Computing **28** (1998), número 1, 254–262. Citado na(s) página(s) 10, 36, 37

[AK06] Sanjeev Arora e George Karakostas, *A $2 + \varepsilon$ approximation algorithm for the k-MST problem*, Mathematical Programming **107** (2006), número 3, 491–504. Citado na(s) página(s) 11, 61

[BFM98] Ralf Borndörfer, Carlos Eduardo Ferreira e Alexander Martin, *Decomposing Matrices into Blocks*, SIAM Journal on Optimization **9** (1998), número 1, 236–269. Citado na(s) página(s) 4

[Blu07] Christian Blum, *Revisiting dynamic programming for finding optimal subtrees in trees*, European Journal of Operational Research **177** (2007), número 1, 102–115. Citado na(s) página(s) 21

[BRV99] Avrim Blum, Ramamurthy Ravi e Santosh Vempala, *A constant-factor approximation algorithm for the k-MST problem*, Journal of Computer and System Sciences **58** (1999), número 1, 101–108. Citado na(s) página(s) 10

[CK94] Shun Yan Cheung e Akhil Kumar, *Efficient Quorumcast Routing Algorithms*, INFOCOM'94 (Los Alamitos, USA), IEEE Society Press, 1994, páginas 840–847. Citado na(s) página(s) 3

[CLRS01] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest e Clifford Stein, *Introduction to Algorithms*, 2ª edição, The MIT Press, 2001. Citado na(s) página(s) 19, 32

[CRW04] Fabián Ariel Chudak, Tim Roughgarden e David Paul Williamson, *Approximate k-MSTs and k-Steiner trees via the primal-dual method and Lagrangean relaxation*, Mathematical Programming **100** (2004), número 2, 411–421. Citado na(s) página(s) 58

[Dij71] Edsger Wybe Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (269–271), número 1, 48–50. Citado na(s) página(s) 31

[Feo] Paulo Feofiloff, *Notas sobre o problema MKT*, não publicado. Citado na(s) página(s) 24

[FFFd02] Paulo Feofiloff, Cristina Gomes Fernandes, Carlos Eduardo Ferreira e José Coelho de Pina Jr., *$O(n^2 \log n)$ implementation of an approximation for the Prize-Collecting Steiner Tree Problem*, Disponível em `http://www.ime.usp.br/~cris/publ/implpcst.ps.gz` (2002). Citado na(s) página(s) 69

[FFFd07] ______, *Primal-dual approximation algorithms for the Prize-Collecting Steiner Tree Problem*, Information Processing Letters **103** (2007), número 5, 195–202. Citado na(s) página(s) 65

[FFFd09] ______, *A note on Johnson, Minkoff and Phillips' algorithm for the Prize-Collecting Steiner Tree Problem*, arXiv.org **1004.1437v1 [cs.DS]** (2009). Citado na(s) página(s) 65, 82

[FHJM94] Matteo Fischetti, Horst Wilhelm Hamacher, Kurt O. Jørnsten e Francesco Maffioli, *Weighted k-cardinality trees: Complexity and polyhedral structure*, Networks **24** (1994), número 1, 11–21. Citado na(s) página(s) 10, 14

[FHW98] Les R. Foulds, Horst Wilhelm Hamacher e John Mark Wilson, *Integer programming approaches to facilities layout models with forbidden areas*, Annals of Operations Research **81** (1998), 405–418. Citado na(s) página(s) 4

[Gar96] Naveen Garg, *A 3-approximation for the minimum tree spanning k vertices*, FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 1996, páginas 302–309. Citado na(s) página(s) 10, 52, 61

[Gar05a] ______, *Saving an ϵ: a 2-approximation algorithm for the k-MST problem in graphs*, Disponível em `http://www.mpi-inf.mpg.de/~jmestre/slides/stoc-2005.pdf`, 2005, Slides apresentados por Julian Mestre no STOC '05. Citado na(s) página(s) 102

[Gar05b] ______, *Saving an ϵ: a 2-approximation for the k-MST problem in graphs*, STOC '05: Proceedings of the 37th annual ACM Symposium on Theory of Computing (New York, NY, USA), ACM Press, 2005, páginas 396–402. Citado na(s) página(s) 11, 61, 93, 109

[GJ90] Michael Randolph Garey e David Stifler Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman & Co., New York, NY, USA, 1990. Citado na(s) página(s) 13

[GK98] Michel Xavier Goemans e Jon Michael Kleinberg, *An improved approximation ratio for the minimum latency problem*, Mathematical Programming **82** (1998), número 1-2, 296–317. Citado na(s) página(s) 50

[GW95] Michel Xavier Goemans e David Paul Williamson, *A General Approximation Technique for Constrained Forest Problems*, SIAM Journal on Computing **24** (1995), número 2, 296–317. Citado na(s) página(s) 41, 43, 49, 50, 61, 65

[GW97] ______, *The primal-dual method for approximation algorithms and its application to network design problems*, In Approximation algorithms for NP-hard problems (Boston, MA, USA), PWS Publishing Co., 1997, páginas 144–191. Citado na(s) página(s) 43

[HJ93] Horst Wilhelm Hamacher e Kurt O. Jørnsten, *Optimal relinquishment according to the Norwegian petrol law: A combinatorial optimization approach*, Relatório Técnico 7/93, Norwegian School of Economics and Business Administration, Bergen, Noruega, 1993. Citado na(s) página(s) 4

[JMP00] David Stifler Johnson, Maria Minkoff e Steven John Phillips, *The prize collecting Steiner tree problem: theory and practice*, SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2000, páginas 760–769. Citado na(s) página(s) 61, 65

[Kru56] Joseph Bernard Kruskal Jr., *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society **7** (1956), 48–50. Citado na(s) página(s) 10, 17

[Maf91] Francesco Maffioli, *Finding a best subtree of a tree*, Relatório Técnico 91.041, Politecnico di Milano, Dipartimento di Elettronica, Itália, 1991. Citado na(s) página(s) 21

[Pri57] Robert Clay Prim, *Shortest connection networks and some generalizations*, Bell System Technical Journal **36** (1957), 1389–1401. Citado na(s) página(s) 10, 18

[PW97] Andrew B. Philpott e Nicholas Charles Wormald, *On the Optimal Extraction of Ore from an Open-Cast Mine*, Relatório técnico, University of Auckland, Nova Zelândia, 1997. Citado na(s) página(s) 3

[RSM+94] Ramamurthy Ravi, Ravi Sundaram, Madhav Vishnu Marathe, Daniel J. Rosenkrantz e Sekharipuram S. Ravi, *Spanning trees short or small*, SODA '94: Proceedings of the 5th annual ACM-SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 1994, páginas 546–555. Citado na(s) página(s) 10, 13, 33

[RV95] Sridhar Rajagopalan e Vijay V. Vazirani, *Logarithmic approximation of minimum weight k trees*, não publicado, 1995. Citado na(s) página(s) 10

[Vaz01] Vijay V. Vazirani, *Approximation algorithms*, Springer-Verlag New York, Inc., New York, NY, USA, 2001. Citado na(s) página(s) 57

[ZL93] Alexander Zelikovsky e Dmitrii Lozovanu, *Minimal and bounded trees*, Tezele Congresului XVIII al Academiei Romano-Americane (Kishinev, Moldova), 1993, páginas 25–26. Citado na(s) página(s) 10, 14

Índice Remissivo

$(2,4)$-aproximador, 50
$(3,6)$-aproximador, 41, 50
(a,b)-aproximador, 50
α, 62
χ, 65
$\delta(\cdot)$, 6

$\mathcal{A}$, 66
AABV-AUX, 37
algoritmo
  AABV, 40
  AABV-AUX, 37
  GARG-LIMIAR, 76
  GARG2, 88
  GARG2-ITERATIVO, 91
  GARG5, 56
  GARG-k-ÁRVORE, 87
  GW, 51
  GW-MODIFICADO, 65, 68
  k-DIJKSTRA, 32
  k-KRUSKAL, 28
  KRUSKAL, 18
  KRUSKAL-MODIFICADO, 19
  PD, 24
  PEGA-VÉRTICE, 85
  PENALIDADE, 78
  PINTA-CONJUNTO, 67
  PODA, 67
  PRIM, 18
algoritmo de aproximação, 1, 8
α, 62
α-aproximação, 8
aresta, 5
  externa, 65
  extremos de, 65
  interna, 65
  justa, 63
  pontas, 6
árvore, 2, 7
  de Steiner, 13
  de caminhos mínimos, 31
  de roteamento, 2
  geradora, 17
    mínima, 17

busca binária, 55

caminho, 6
  mais curto, 6
  pontas, 6

tamanho, 6
vértices internos, 6
candidato a solução, 7, 44
custo do, 7
valor do, 7
CCM, 31
χ, 65
circuito, 6
$c_{\min}$, 55
coleção laminar, 63
componente, 7
ativo, 66
conjunto
ativo, 66
contração
árvore, 40
corte, 6, 18, 49
custo, 9
reduzido, 69

desigualdade triangular, 55
distância, 32
dual, 53
PCST, 49
dualidade
fraca, 49
dualidade fraca, 47

evento, 70

fator de aproximação, 8
fator de aproximação
justo, 29
$\widehat{F}$, 62
fila de prioridade, 32
floresta, 7
força-bruta, 1
função coloração, 65
função objetivo, 43

GARG-LIMIAR, 76
GARG2, 88
GARG2-ITERATIVO, 91
GARG5, 56
GARG-k-ÁRVORE, 87
grafo, 2, 5
completo, 5
conexo, 7
de mineração, 3
dos componentes, 34
orientado, 3, 5
gw
GW, 51
GW-MODIFICADO, 65, 68

infinitesimalmente, 62
instância, 7

k-ÁRVORE(F, k), 20
k-DIJKSTRA, 31–33
k-KRUSKAL, 27, 28
k-árvore, 9
com pesos, 12, 33
kMST, 2, 10
com pesos, 13
em árvores, 21
enraizado, 11
k-KRUSKAL, 33
kMST-A, 21
kMST-P, 13
kMST-R, 11
Kruskal, 17

KRUSKAL, 18
Kruskal, 27
Kruskal-Modificado, 19
KRUSKAL-MODIFICADO, 19

$\mathcal{L}(\cdot)$, 63
$\mathcal{L}^*$, 63
$\mathcal{L}$-conexo, 65
$\mathcal{L}_\cdot$, 63
$\mathcal{L}[\cdot]$, 63
laminar, 63

m, 6
método enumerativo, 1, 16
MST, 10
multicast, 2

n, 6

ordem lexicográfica, 66
otimização combinatória, 1

pai, 23
parametrização, 71
    válida, 73
PCST, 48
PD, 24
PEGA-VÉRTICE, 85
PENALIDADE, 78
penalidade, 70
penalidade limiar, 62, 75
peso, 12
PINTA-CONJUNTO, 67
PODA, 67
ponte, 65
potencial, 69
    inicial, 69
precedência, 3
Prim, 18
PRIM, 18, 21
problema
    árvore de Steiner, 13
        com coleta de prêmios, 48
    caminhos de custo mínimo, 31
problema de otimização, 7
    maximização, 7
    minimização, 7
programa dual, 47
programa inteiro
    PCST, 48
programa primal, 47
programação linear
    viável, 44
programação dinâmica, 21
programação linear, 43
    inteira, 44
    inviável, 44
    problema de, 44

quorumcast, 2

raiz, 7
redução, 13
relaxação, 45
    lagrangeana, 46, 53
    linear, 46, 52
respeita, 63
roteamento, 2

satura, 63
solução viável, 7
ST, 13
subárvore enraizada, 23

subestrutura ótima, 22
subgrafo, 6
    conexo em conjunto, 65
    gerador, 6
    induzido, 6

terminal, 13
trilha, 7
    euleriana, 7
    fechada, 7

valor ótimo, 1
vértice, 5
    adjacente, 6
    grau de, 6
    vizinho, 6