O Problema do Multicorte Dirigido Mínimo

Juan Gutiérrez Alva

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

Programa: Mestrado em Ciência da Computação
Orientador: Prof. Dr. Paulo Feofiloff

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

São Paulo, 23 de janeiro de 2013

O Problema do Multicorte Dirigido Mínimo

Esta tese/dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Juan Gutiérrez Alva em 7/12/2012. O original encontra-se disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr Paulo Feofiloff (orientador) - IME-USP
- Prof. Dr. José Coelho de Pina - IME-USP
- Prof. Dr. Orlando Lee - IC-UNICAMP

Dedico este trabalho a meus pais e irmãos.

Agradecimentos

Ao professor Paulo, por aceitar ser meu orientador, por seus conselhos e comentários acertados, e por sua infinita paciência. A todos meus companheiros que conheci estes dois anos e meio no IME, por sua companhia e amizade. A meus familiares, que estiveram perto de mim ainda estando longe. A todos os professores com os que cursei disciplinas no IME, por seu conhecimento oferecido. E a Carla, por sua companhia, carinho e paciência.

Resumo

O Problema do Multicorte Dirigido Mínimo é um problema clássico em otimização combinatória. Ele é NP-difícil mesmo para instâncias muito simples. Este trabalho faz uma análise dos algoritmos exatos e de aproximação para resolver o problema. Também implementa alguns desses algoritmos e compara seus desempenhos.

Palavras-chave: teoria dos grafos, algoritmos em grafos, multicorte dirigido, multicommodity disconnecting set.

Abstract

The directed multicut problem is a classical problem in combinatorial optimization. It is NP-hard even for very simple families of instances. This work makes an analysis of the exact and approximation algorithms for the problem. It also implements some of these algorithms and compares their performances.

Keywords: graph theory, algorithms in graphs, directed multicut, multicommodity disconnecting set.

Sumário

Capítulo 1

Introdução

Um par de terminais em um digrafo é um par ordenado de vértices diferentes. Dado um digrafo e um conjunto de k pares de terminais, dizemos que um conjunto de arcos X do digrafo é um multicorte se, para cada par de terminais st, qualquer caminho de s a t contém pelo menos um arco de X. O Problema do Multicorte Dirigido de Custo Mínimo consiste em, dado um digrafo com custos nos arcos e um conjunto de pares de terminais, encontrar um multicorte de custo mínimo. No caso $k = 1$, o problema é o clássico problema do Corte Mínimo/Fluxo Máximo.

Esse problema tem muitas aplicações em roteamento de telecomunicações e transporte [BCF00]. Multicortes são importantes no estudo de cadeias de Markov, clustering, técnicas de divisão e conquista, e desenho de circuitos integrados [ENRS00].

A dualidade entre fluxos e cortes é um fenômeno fundamental na otimização combinatória. O dual do Problema do Multicorte Dirigido de Custo Mínimo é o Problema do Multifluxo Máximo. Porém, o clássico teorema de Fluxo Máximo/Corte Mínimo de Ford e Fulkerson [FF56] não pode ser generalizado para o caso de multicortes (ou seja, o multifluxo máximo pode ser estritamente menor que o multicorte mínimo).

Se no lugar de um digrafo temos um grafo, definimos O Problema do Multicorte Não Dirigido de Custo Mínimo de maneira similar. Para esse problema existe uma $O(\log k)$-aproximação apresentada por Garg *et al.* [GVY93], que faz uso da técnica de "region growing" introduzida por Leighton e Rao [LR88].

Foi impossível estender essas técnicas ao Problema do Multicorte Dirigido de Custo Mínimo, exceto em alguns casos simétricos (por exemplo se para cada par de terminais st existe o par de terminais ts). Even *et al.* exibiram uma $O(\log^2 k)$-aproximação para o esse tipo de instâncias [ENSS98]. Esse algoritmo faz uso da técnica de "region growing", mas o fator alcançado é pior que o fator atingido no caso do Problema do Multicorte Não Dirigido de Custo Mínimo. Lamentavelmente o caso não simétrico do Problema do Multicorte Dirigido de Custo Mínimo dista muito do caso simétrico e portanto os esforços feitos para esse tipo de instâncias não têm quase relevância para o caso geral. Tudo indica que o Problema do Multicorte Dirigido de Custo Mínimo é mais difícil que

o Problema do Multicorte Não Dirigido de Custo Mínimo.

O Problema do Multicorte Dirigido Mínimo de Custo Mínimo é NP-difícil. Além disso, não existe algoritmo polinomial que produza uma aproximação com fator constante a menos que uma certa hipótese mais fraca que P=NP se confirme [CK07].

Em seguida farei um resumo dos principais algoritmos de aproximação conhecidos para o problema. Cheriyan *et al.* [CKR05] apresentaram uma $O(\sqrt{n \log n})$-aproximação, onde n é o número de vértices do digrafo. Esse fator de aproximação foi melhorado por Gupta [Gup03]. Ele mostrou uma $O(\sqrt{n})$-aproximação, que é uma das melhores aproximações conhecidas. Os dois resultados anteriores foram feitos usando programação linear. Kortsarts *et al.* [KKN05] exibiram uma $O(n^{2/3})$-aproximação puramente combinatória.

No capítulo 2 definirei o problema tanto na versão geral como na versão com custos unitários para posteriormente analisar sua complexidade.

No capítulo 3 exporei dois algoritmos simples que resolvem o problema de maneira aproximada, mas com fatores de aproximação muito altos.

No capítulo 4 formularei duas relaxações lineares do problema. Posteriormente mostrarei o algoritmo de Bellmore *et al.* [BGJ70], que resolve o problema de maneira exata fazendo uso de técnicas de programação linear.

No capítulo 5 exibirei o algoritmo de Aneja e Vemuganti [AV77], Bellmore e Ratliff [BR71], que resolve o problema de maneira exata fazendo uso de técnicas similares às usadas no método Simplex para redes.

No capítulo 6 aduzirei o algoritmo de Costa *et al.* [CLR03], que resolve em tempo polinomial o problema em árvores divergentes.

No capítulo 7 apresentarei o algoritmo de Kortsarts *et al.* [KKN05], que resolve o problema de maneira aproximada caso os arcos tenham custos unitários.

No capítulo 8 analisarei o algoritmo de Gupta [Gup03], que se baseia no algoritmo de Cheriyan *et al.* [CKR05] e dá um dos melhores fatores de aproximação conhecidos para o problema.

Finalmente no capítulo 9 exibirei os resultados feitos no estudo experimental.

Capítulo 2

Problema do multicorte dirigido de custo mínimo

Neste capítulo começaremos por definir o problema tanto na sua versão geral como na sua versão com custos unitários. Em seguida mostraremos que o problema é NP-difícil.

O leitor interessado em revisar os conceitos básicos da teoria dos grafos e a notação usada ao longo do documento poderá consultar o apêndice A.

2.1 Definição

Dizemos que um par ordenado de vértices st de um digrafo G é um *par de terminais* se $s \neq t$. Uma *rede* é uma dupla (G, Q) tal que G é um digrafo e Q é um conjunto de pares de terminais em G. Dada uma rede (G, Q), dizemos que um conjunto de arcos $X \subseteq E_G$ *separa* um elemento st de Q se não existe nenhum caminho de s a t em $G - X$. Um conjunto que separa todos os elementos de Q é chamado *multicorte* da rede (G, Q). Um multicorte X^* de (G, Q) é *mínimo* se para qualquer multicorte X de (G, Q), $|X^*| \leq |X|$.

> **Problema do Multicorte Dirigido Mínimo (MM)**: Dada uma rede (G, Q), encontrar um multicorte mínimo de (G, Q).

O exemplo da figura 2.1 mostra uma rede com três pares de terminais a separar. Nesse exemplo um multicorte mínimo tem cardinalidade 2.

Consideremos agora uma função c que associa um custo a cada arco de G. Estenda a definição de rede a uma tripla (G, c, Q) tal que G é um digrafo, $c_e \geq 0$ para todo $e \in E_G$, e Q é um conjunto de pares de terminais em G. Um multicorte X^* de (G, Q) é *c-mínimo* se para qualquer multicorte X de (G, Q), $c(X^*) \leq c(X)$, onde $c(X)$ significa $\sum_{e \in X} c_e$ conforme descrito no apêndice A.

> **Problema do Multicorte Dirigido de Custo Mínimo (MCM)**: Dada uma rede (G, c, Q), encontrar um multicorte c-mínimo de (G, Q).

No caso $|Q| = 1$, o problema MCM é resolvido pelo algoritmo de Ford e Fulkerson [FF56] em tempo polinomial. Já para $|Q| = 2$, o problema é NP-difícil (veja a seção 2.2).

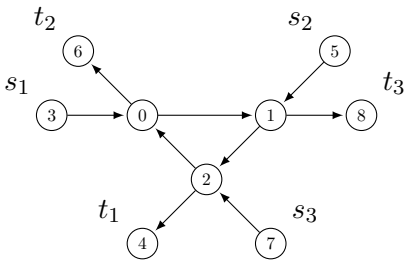

Figura 2.1: O gráfico mostra um digrafo G e um conjunto de pares de terminais $Q = \{s_1t_1, s_2t_2, s_3t_3\}$ em G. O conjunto $X = \{01, 12\}$ é um multicorte mínimo de (G, Q). Esta rede foi tomada do artigo de Bellmore *et al.* [BGJ70].

Dada uma rede (G, Q) ou uma rede (G, c, Q), no resto do documento a variável n vai representar o número de vértices de G, a variável m o número de arcos de G e a variável k o número de elementos de Q. Também,

$$\mu(G, Q)$$

vai representar a cardinalidade de um multicorte mínimo de (G, Q) e

$$\mu(G, c, Q)$$

vai representar o custo de um multicorte c-mínimo de (G, Q). O tamanho de uma instância depende dos três parâmetros n, m e k os quais não são independentes, já que $1 \leq k \leq n^2 - n$ e $0 \leq m \leq n^2 - n$.

2.2 Complexidade computacional

Nesta seção mostraremos que o problema MM, e portanto também o problema MCM, é NP-difícil. Para a prova, reduziremos uma instância do Problema do Multisseparador Dirigido Mínimo (Multiway Cut) a uma instância do MM. Já que o Problema do Multisseparador Mínimo é NP-difícil, concluiremos que o problema MM é NP-difícil.

Dados um digrafo G e um subconjunto S de V_G, diremos que um conjunto $X \subseteq E_G$ é um *multisseparador* de (G, S) se não existe caminho de s a s' em $G - X$ para cada par ss' de elementos distintos de S. Um multisseparador X^* de (G, Q) é *mínimo* se para qualquer multisseparador X de (G, Q), $|X^*| \leq |X|$.

> **Problema do Multisseparador Dirigido Mínimo**: Dados um digrafo G e um subconjunto S de V_G, encontrar um multisseparador mínimo de (G, S).

O Problema do Multisseparador Dirigido Mínimo (também conhecido como *Multiway Cut*) é um caso particular do problema MM. A versão não dirigida daquele problema é definida de maneira óbvia: no lugar do digrafo temos um grafo G e X é um multisseparador se não existe caminho entre s e s' em $G - X$ para todo par ss' de elementos

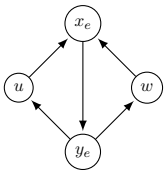

Figura 2.2: Redução da versão não dirigida à versão dirigida do Problema do Multisseparador Mínimo. Em cada passo do algoritmo escolhemos uma aresta arbitrária $e = uw$ de E_G e adicionamos dois novos vértices, x_e e y_e, e 5 novos arcos, $ux_e, wx_e, x_ey_e, y_eu, y_ew$, a D.

de S.

Dalhaus *et al.* [DJP$^+$94] mostraram que a versão não dirigida do problema é NP-difícil. A prova faz uma redução a partir do problema de Corte Máximo. Ela é um tanto complicada e não a reproduziremos aqui. Tomaremos esse resultado como ponto de partida para mostrar que o Problema do Multisseparador Dirigido Mínimo é NP-difícil.

Teorema 2.1 O Problema do Multisseparador Dirigido Mínimo é NP-difícil.

Prova: Redução da versão não dirigida à versão dirigida do Problema do Multisseparador Mínimo. Considere o seguinte algoritmo que constrói uma instância (D, S) da versão dirigida a partir de uma instância (G, S) da versão não dirigida do problema. No início do algoritmo, $V_D := V_G$ e $E_D := \emptyset$. Em cada passo do algoritmo escolhemos uma aresta arbitrária $e = uw$ de E_G e adicionamos dois novos vértices, x_e e y_e, e 5 novos arcos, $ux_e, wx_e, x_ey_e, y_eu, y_ew$, a D (veja figura 2.2). Diremos que o arco x_ey_e é *especial*. Retiramos uw de E_G e repetimos o processo até que $E_G = \emptyset$.

Seja D o digrafo resultante da aplicação do algoritmo ao digrafo G. Provaremos que para todo multisseparador M_D de (D, S) existe um multisseparador M'_D que só usa arcos especiais e $|M'_D| \leq |M_D|$. Considere o seguinte algoritmo que constrói M'_D a partir de M_D. No início do algoritmo, $M'_D = \emptyset$. Tome qualquer arco em M_D. Se esse arco é especial, então adicione-o a M'_D. Se o arco não é especial, então ele é da forma ux_e, wx_e, y_eu ou y_ew. Em qualquer caso, adicione o arco x_ey_e a M'_D. Retire o arco de M_D e repita o processo até que $M_D = \emptyset$. Provaremos que M'_D é um multisseparador de (D, S). Suponha por contradição que existe algum caminho de s a s' em $D - M'_D$ para algum par de vértices ss' em S. Esse mesmo caminho existe em $D - M_D$ e M_D não separa s de s'. Portanto M_D não é um multisseparador de (D, S). Contradição. Como para cada arco escolhido em M_D é adicionado no máximo um arco a M'_D, no final do algoritmo, $|M'_D| \leq |M_D|$.

Considere o seguinte algoritmo polinomial que obtém um multisseparador mínimo M_G de (G, S) a partir de um multisseparador mínimo M_D de (D, S). Primeiro o algoritmo usa como sub-rotina o algoritmo do parágrafo anterior obtendo um multisseparador mínimo M'_D que só usa arcos especiais. Considere o conjunto $M_G = \{e \in E_G : x_ey_e \in M'_D\}$. Provaremos que M_G é um multisseparador de (G, S).

Suponha por contradição que existe um caminho em $G - M_G$ entre algum par de vértices ss' em S. Então existe um caminho de s a s' em $D - M_D$. Portanto D não é um multisseparador de (D, S). Contradição.

Por último, provaremos que M_G é mínimo. Suponha que existe um multisseparador M_G^* de (G, S) com cardinalidade menor que $|M_G|$. O conjunto $M_D^* = \{x_e y_e \in E_D : e \in E_G\}$ é um multisseparador de (D, S) com cardinalidade menor que $|M'_D|$. Portanto M'_D não é um multisseparador mínimo de (D, S). Contradição. □

Corolário 2.1 O problema MM é NP-difícil.

Prova: Reduzimos o Problema do Multisseparador Dirigido Mínimo para o problema MM. Definimos uma instância (G, Q) do problema MM a partir de uma instância (D, S) do Problema do Multisseparador Dirigido Mínimo segundo: $G := D$ e $Q = \{st : s \neq t \in S\}$. É claro que qualquer multisseparador de (D, S) é também um multicorte de (G, Q). Analogamente, qualquer multicorte de (G, Q) é também um multisseparador de (D, S). Portanto basta encontrar um multisseparador mínimo de (D, S) para encontrar um multicorte mínimo de (G, Q). □

Com uma prova um pouco mais cuidadosa, Garg *et al.* [GVY94] provaram que o problema MM é NP-difícil mesmo quando restrito às instâncias em que $|Q| = 2$.

Capítulo 3

Dois algoritmos de aproximação

Neste capítulo mostraremos dois algoritmos de aproximação óbvios e muito fáceis de implementar, mas com fatores de aproximação elevados. Antes, daremos uma definição importante.

Seja A um algoritmo que, para toda rede (G, Q), devolve um multicorte $A(G, Q)$ de (G, Q). Se $|A(G, Q)| \leq \alpha \cdot \mu(G, Q)$ para toda rede (G, Q), dizemos que A é uma α-*aproximação* para o problema MM e α é chamado *fator de aproximação* do algoritmo A. Analogamente, um algoritmo A é uma α-*aproximação* para o problema MCM se $c(A(G, Q)) \leq \alpha \cdot \mu(G, c, Q)$ para toda rede (G, c, Q).

3.1 Algoritmo de aproximação com fator k

Iremos descrever uma k-aproximação para o problema MCM, onde k é o número de pares de terminais da rede. O algoritmo MFMC-Iterado recebe uma rede (G, c, Q) e devolve um multicorte X de (G, Q) tal que $c(X) \leq k \cdot \mu(G, c, Q)$, onde $\mu(G, c, Q)$ é o custo de um multicorte c-mínimo de (G, Q).

Algoritmo MFMC-Iterado(G, c, Q)

```
1  X ← ∅
2  para todo st ∈ Q faça
3      G' ← G − X
4      se dist(s, t, G') < ∞ então
5          C ← FordFulkerson(G', c, s, t)
6          X ← X ∪ C
7  devolva X
```

O algoritmo MFMC-Iterado usa como sub-rotina o algoritmo FordFulkerson que recebe um digrafo G' com custos não negativos c nos arcos e dois vértices s, t de G' e devolve um corte de custo mínimo que separa s de t em G'.

Teorema 3.1 O algoritmo MFMC-Iterado devolve um multicorte X de (G, Q) tal

que $c(X) \leq k \cdot \mu(G, c, Q)$.

Prova: Começaremos provando que, no final do algoritmo, X é um multicorte de (G, Q). Considere a seguinte invariante no início de cada iteração do processo iterativo das linhas 2-6:

(I) X é um multicorte de (G', Q'), sendo Q' o conjunto de pares de terminais de Q que já foram escolhidos na linha 2.

No início da primeira iteração a invariante (I) é válida, já que $Q' = \emptyset$. Suponha agora que a invariante é válida no início de uma iteração qualquer das linhas 2-6. Vamos mostrar que a invariante é válida no início da iteração seguinte. Se $dist(s, t, G') = \infty$ então, no início da iteração atual, X separa s de t em G'. A prova da invariante segue do fato que, no final da iteração, Q' aumenta seu valor em st e X não muda de valor. Se $dist(s, t, G') < \infty$, imediatamente depois da execução da linha 5, C separa s de t em G'. A prova da invariante segue do fato que, na linha 6, X aumenta seu valor em C e, no final da iteração, Q' aumenta seu valor em st. Portanto a invariante é válida no início da iteração seguinte.

A invariante (I) prova que, no final do algoritmo, X é um multicorte de (G, Q). Vamos agora provar que, no final do algoritmo, $c(X) \leq k \cdot \mu(G, c, Q)$. Seja $X_1, X_2, \ldots, X_p$ a sequência de valores da variável X imediatamente depois da execução da linha 6. Seja $(s_1, t_1, G'_1, C_1), \ldots, (s_p, t_p, G'_p, C_p)$ a correspondente sequência de valores de (s, t, G', C). Seja X^* um multicorte c-mínimo de (G, Q). Como X^* é um multicorte de (G, Q) e como cada G'_i é um subgrafo de G, X^* separa s_i de t_i em G'_i para cada i. Como C_i é um corte de custo mínimo que separa s_i de t_i em G'_i então, para cada i,

$$c(C_i) \leq c(X^*).$$

No final do algoritmo, $X = C_1 \cup C_2 \cup \ldots \cup C_p$, e, como todos os custos nos arcos de G' são não negativos, então

$$c(X) \leq \sum_{i=1}^{p} c(C_i) \leq p \cdot c(X^*) \leq k \cdot c(X^*) = k \cdot \mu(G, c, Q).$$

Com isso, concluímos a prova do teorema. □

Em seguida analisaremos a complexidade do algoritmo. A linha 5 pode ser implementada com o algoritmo de Edmonds-Karp [EK72] em tempo $O(nm^2)$. O número de iterações do processo iterativo das linhas 2-6 está limitado superiormente por k. Concluímos que o tempo de execução do algoritmo MFMC-Iterado é

$$O(knm^2).$$

Na implementação do algoritmo fizemos testes com redes geradas aleatoriamente e com famílias de digrafos encontrados na Internet. Conseguimos construir uma instância muito simples onde a k-aproximação é justa, mas em geral os custos devolvidos pelo algoritmo estiveram muito perto do custo do multicorte c-mínimo. Para mais detalhes da implementação do algoritmo, veja o capítulo 9.

3.2 Algoritmo de aproximação com fator n

Iremos descrever uma n-aproximação para o problema MM, onde n é o número de vértices da rede. A ideia é muito simples: pegar um caminho mínimo para cada par de terminais e adicioná-lo ao multicorte. Esta n-aproximação é interessante porque voltará a ser usada (implicitamente) nas linhas 11-15 do algoritmo MULTICORTE-TODOS (capítulo 7). O algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS recebe uma rede (G, Q) e devolve um multicorte X de (G, Q) tal que $|X| \leq n \cdot \mu(G, Q)$, onde $\mu(G, Q)$ é a cardinalidade de um multicorte mínimo.

Algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS(G, Q)

```
1  X ← ∅
2  para todo st ∈ Q faça
3      enquanto dist(s, t, G − X) < ∞ faça
4          P ← CAMINHO(G − X, s, t)
5          X ← X ∪ E_P
6  devolva X
```

O algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS usa como sub-rotina o algoritmo CAMINHO, que recebe um digrafo G' e dois vértices s, t e devolve um caminho de s a t em G'.

Teorema 3.2 O algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS devolve um multicorte X de (G, Q) tal que $|X| \leq n \cdot \mu(G, Q)$.

Prova: Começaremos provando que, no final do algoritmo, X é um multicorte de (G, Q). Considere a seguinte invariante no início de cada iteração do processo iterativo das linhas 2-5:

(I) X é multicorte de (G, Q'), sendo Q' o conjunto de pares de terminais de Q que já foram escolhidos na linha 2.

No início da primeira iteração a invariante (I) é válida, já que $Q' = \emptyset$. Suponha agora que a invariante é válida no início de uma iteração qualquer das linhas 2-5. Vamos mostrar que a invariante é válida no início da iteração seguinte. Imediatamente depois do processo iterativo das linhas 3-5, $dist(s, t, G-X) = \infty$ e, portanto, no final da iteração, X separa s de t em G. A prova da invariante segue do fato que, no final da iteração, Q' aumenta

seu valor em st. Portanto a invariante é válida no início da iteração seguinte.

A invariante (I) prova que, no final do algoritmo, X é um multicorte de (G, Q). Vamos agora provar que, no final do algoritmo, $|X| \leq n \cdot \mu(G, Q)$. Seja r o número de caminhos obtidos em todas as chamadas a CAMINHO. Então $|X| < n \cdot r$, pois todo caminho encontrado tem comprimento menor a n. Por outro lado $r \leq \mu(G, Q)$, já que esses r caminhos são disjuntos nos arcos e um multicorte precisa conter pelo menos um arco de cada um dos r caminhos. Segue que, no final do algoritmo,

$$|X| < n \cdot \mu(G, Q).$$

Com isso, concluímos a prova do teorema. □

Em seguida analisaremos a complexidade do algoritmo. Cada chamada à função CAMINHO pode ser implementada com uma busca em largura em tempo $O(n + m)$. Vamos delimitar o total de chamadas a essa função. Seja $P_1, P_2, \ldots, P_r$ a sequência de caminhos obtidos em todas as chamadas a CAMINHO. Seja $X_1, X_2, \ldots, X_r$ a correspondente sequência de valores de X em todas as chamadas a CAMINHO. É claro que cada P_i têm comprimento como máximo $|E_G \setminus X_i|$. Então,

$$|P_r| \leq |E_G \setminus X_r| = m - \sum_{i=1}^{r-1} |P_i|,$$

e portanto

$$r \leq \sum_{i=1}^{r} |P_i| \leq m,$$

onde a primeira desigualdade vale pois, por causa da condição da linha 3, cada P_i tem comprimento pelo menos um.

Concluímos que o tempo de execução do algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS é

$$O(m(n + m)).$$

O fator de aproximação n é muito grosseiro, mas a complexidade do algoritmo MULTICORTE-COLEÇÃO-DE-CAMINHOS justifica sua implementação. Só é razoável implementar esse algoritmo quando $n < k$, já que em outro caso o algoritmo MFMC-ITERADO (veja seção 3.1) tem melhor fator de aproximação.

Capítulo 4

Duas relaxações lineares

Neste capítulo formularemos duas relaxações lineares do Problema do Multicorte Dirigido de Custo Mínimo (MCM). Uma delas faz uso de um número potencialmente exponencial de restrições. Na outra formulação, o número de restrições é sempre polinomial. Para cada relaxação formularemos também o dual respectivo.

Formularemos os programas lineares inteiros associados a cada uma dessas relaxações e descreveremos o algoritmo de Bellmore *et al.* [BGJ70], que resolve o problema MCM de maneira exata com base na primeira formulação.

Este capítulo é importante para o resto do documento, já que serão constantes as referências às formulações feitas.

4.1 Primeira relaxação linear

Dada uma rede (G, c, Q), um *Q-caminho* em G é um caminho de s a t em G para algum $st \in Q$. Defina $\mathcal{P} = \{E_P : P \text{ é um } Q\text{-caminho em } G\}$. O seguinte programa linear, que chamaremos de *PL1*, é uma relaxação do problema MCM. Ele consiste em encontrar um vetor racional x indexado por E_G que

$$\text{minimize} \sum_{uv \in E_G} c_{uv} x_{uv} \tag{4.1}$$

sujeito a

$$\sum_{uv \in P} x_{uv} \geq 1 \quad \text{para cada } P \in \mathcal{P}, \tag{4.2}$$

$$x_{uv} \geq 0 \quad \text{para cada } uv \in E_G. \tag{4.3}$$

O vetor característico x de qualquer multicorte de (G, Q) satisfaz (4.2) e (4.3). Reciprocamente, qualquer vetor x de zeros e uns que satisfaz (4.2) e (4.3) representa um multicorte de (G, Q). Portanto, *PL1* é uma relaxação do problema MCM.

Um vetor racional x que é viável em *PL1* é chamado de *multicorte fracionário* de (G, c, Q). Um *multicorte fracionário c-mínimo* de (G, Q) é uma solução ótima de *PL1*.

Note que *PL1* não precisa da restrição $x_{uv} \leq 1$ para cada $uv \in E_G$, já que toda solução ótima de *PL1* satisfaz essa restrição automaticamente.

O dual de *PL1* consiste em encontrar um vetor racional y indexado por $\mathcal{P}$ que

$$\text{maximize} \sum_{P \in \mathcal{P}} y_P \tag{4.4}$$

sujeito a

$$\sum_{P: uv \in P} y_P \leq c_{uv} \quad \text{para cada } uv \in E_G, \tag{4.5}$$

$$y_P \geq 0 \quad \text{para cada } P \in \mathcal{P}. \tag{4.6}$$

Um vetor racional y que satisfaz (4.5) e (4.6) é chamado de *multifluxo* de (G, c, Q).

Lema 4.1 Dada um rede (G, c, Q), para cada multicorte fracionário x de (G, Q) e cada multifluxo y de (G, c, Q),

$$\sum_{P \in \mathcal{P}} y_P \leq \sum_{uv \in E_G} c_{uv} x_{uv}. \tag{4.7}$$

Prova:

$$\begin{aligned} \sum_{P \in \mathcal{P}} y_P &\leq \sum_{P \in \mathcal{P}} \big(y_P \cdot \sum_{uv \in P} x_{uv}\big) && (4.8) \\ &= \sum_{P \in \mathcal{P}} \big(\sum_{uv \in P} y_P \cdot x_{uv}\big) \\ &= \sum_{uv \in E_G} \big(\sum_{P: uv \in P} y_P \cdot x_{uv}\big) \\ &\leq \sum_{uv \in E_G} c_{uv} x_{uv}, && (4.9) \end{aligned}$$

onde (4.8) vale por (4.2) e (4.9) vale por (4.5). □

Pelo teorema da dualidade [Chv83, p.54], vale a igualdade em (4.7) quando e somente quando x é um multicorte fracionário c-mínimo de (G, Q) e y é um multifluxo máximo de (G, c, Q).

Seja K_n um digrafo completo com n vértices. Ou seja, um digrafo tal que, para cada dois vértices u,v de K_n, existe um arco uv. Sejam s e t dois vértices de K_n. Há pelo menos 2^n caminhos disjuntos de s a t. Logo, o número de restrições em (4.2) é exponencial quando $G = K_n$.

4.2 Segunda relaxação linear

Dada uma rede (G, c, Q), o seguinte programa linear, que chamaremos de *PL2*, é uma relaxação do problema MCM. Ele consiste em encontrar vetores racionais z^{st}, $st \in Q$, indexados por V_G, e um vetor racional x indexado por E_G que

$$\text{minimizem} \sum_{uv \in E_G} c_{uv} x_{uv} \tag{4.10}$$

sujeito a

$$\begin{aligned} z_t^{st} - z_s^{st} &\geq 1 \quad \text{p.c. } st \in Q, && (4.11)\\ z_u^{st} - z_v^{st} + x_{uv} &\geq 0 \quad \text{p.c. } uv \in E_G \text{ e cada } st \in Q, && (4.12)\\ x_{uv} &\geq 0 \quad \text{p.c. } uv \in E_G, && (4.13) \end{aligned}$$

sendo "p.c." uma abreviatura de "para cada". Note que *PL2* não precisa da restrição $x_{uv} \leq 1$ para cada $uv \in E_G$, já que toda solução ótima de *PL2* satisfaz essa restrição automaticamente.

O dual de *PL2*, que chamaremos de *PD2*, consiste em encontrar vetores racionais y^{st}, $st \in Q$, indexados por E_G, e números w^{st}, $st \in Q$, que

$$\text{maximizem} \sum_{st \in Q} w^{st} \tag{4.14}$$

sujeito a

$$\begin{aligned} \sum_{u:uv \in E_G} y_{uv}^{st} - \sum_{u:vu \in E_G} y_{vu}^{st} &= 0 \quad \text{p.c. } st \in Q \text{ e cada } v \in V_G \setminus \{s,t\}, && (4.15)\\ \sum_{u:su \in E_G} y_{su}^{st} - \sum_{u:us \in E_G} y_{us}^{st} &= w^{st} \quad \text{p.c. } st \in Q, && (4.16)\\ \sum_{u:ut \in E_G} y_{ut}^{st} - \sum_{u:tu \in E_G} y_{tu}^{st} &= w^{st} \quad \text{p.c. } st \in Q, && (4.17)\\ \sum_{st \in Q} y_{uv}^{st} &\leq c_{uv} \quad \text{p.c. } uv \in E_G, && (4.18)\\ y_{uv}^{st} &\geq 0 \quad \text{p.c. } st \in Q \text{ e cada } uv \in E_G, && (4.19)\\ w^{st} &\geq 0 \quad \text{p.c. } st \in Q. && (4.20) \end{aligned}$$

A variável y_{uv}^{st} pode ser vista como o *fluxo relativo a st* no arco uv. O fluxo relativo a st que entra em um vértice v é a soma de todos os fluxos relativos a st de todos os arcos que entram em v. O fluxo relativo a st que sai de um vértice v é definido analogamente. A restrição (4.15) estabelece que o fluxo relativo a st que entra em um vértice, que não é

nem s nem t, é igual ao fluxo relativo a st que sai do vértice. O *fluxo total* em um arco é a soma de todos os fluxos nesse arco. A restrição (4.18) estabelece que o fluxo total em um arco é limitado pelo custo desse arco.

Lema 4.2 Para cada par (x, z) viável em *PL2* e cada par (y, w) viável em *PD2*,

$$\sum_{st \in Q} w^{st} \leq \sum_{uv \in E_G} c_{uv} x_{uv}. \tag{4.21}$$

Prova: De (4.15-4.17), para cada $st \in Q$,

$$\begin{aligned}
&\sum_{uv \in E_G} y^{st}_{uv}(z^{st}_u - z^{st}_v) \\
&= \sum_{uv \in E_G} y^{st}_{uv} z^{st}_u - \sum_{uv \in E_G} y^{st}_{uv} z^{st}_v \\
&= \sum_{u \in V_G} z^{st}_u \Big(\sum_{v:uv \in E_G} y^{st}_{uv} - \sum_{v:vu \in E_G} y^{st}_{vu} \Big) \\
&= z^{st}_s \Big(\sum_{v:sv \in E_G} y^{st}_{sv} - \sum_{v:vs \in E_G} y^{st}_{vs} \Big) + z^{st}_t \Big(\sum_{v:tv \in E_G} y^{st}_{tv} - \sum_{v:vt \in E_G} y^{st}_{vt} \Big) \\
&= z^{st}_s w^{st} - z^{st}_t w^{st} \\
&= w^{st}(z^{st}_s - z^{st}_t). \qquad (4.22)
\end{aligned}$$

Para cada st em Q,

$$\begin{aligned}
w^{st} &\leq (z^{st}_t - z^{st}_s) w^{st} && (4.23) \\
&\leq \sum_{uv \in E_G} (y^{st}_{uv} \cdot (z^{st}_u - z^{st}_v + x_{uv})) + (z^{st}_t - z^{st}_s) w^{st} && (4.24) \\
&= \sum_{uv \in E_G} y^{st}_{uv} x_{uv} + \sum_{uv \in E_G} (z^{st}_u - z^{st}_v) y^{st}_{uv} + (z^{st}_t - z^{st}_s) w^{st} \\
&= \sum_{uv \in E_G} y^{st}_{uv} x_{uv}, && (4.25)
\end{aligned}$$

onde (4.23) vale por (4.11), (4.24) vale por (4.12) e (4.19), e (4.25) vale por (4.22).

Por último, de (4.18) e (4.25),

$$\begin{aligned}
\sum_{st \in Q} w^{st} &\leq \sum_{st \in Q} \Big(\sum_{uv \in E_G} y^{st}_{uv} x_{uv} \Big) \\
&= \sum_{uv \in E_G} \Big(x_{uv} \sum_{st \in Q} y^{st}_{uv} \Big) \\
&\leq \sum_{uv \in E_G} c_{uv} x_{uv}.
\end{aligned}$$

Com isso, concluímos a prova do lema. □

Pelo teorema da dualidade em programação linear [Chv83, p.54], vale a igualdade em (4.21) quando e somente quando (x, z) é solução ótima do *PL2* e (y, w) é solução ótima de *PD2*.

Vamos mostrar que o número de restrições de *PL2* é polinomial no tamanho da rede. Lembre que $m := |E_G|$, $n := |V_G|$ e $k := |Q|$. Por causa do conjunto de restrições em (4.11) existem k restrições. O conjunto de restrições em (4.12) dá $m \cdot k$ restrições. Finalmente, o conjunto de restrições em (4.13) dá m restrições. Portanto, para cada rede (G, c, Q), o *PL2* tem como número de restrições:

$$k(m + 1) + m.$$

4.3 Comparação entre as relaxações

Começaremos mostrando que *PL1* e *PL2* são equivalentes.

Suponha que x é viável em *PL1*. Defina os vetores racionais z^{st}, $st \in Q$, segundo $z^{st}_u := dist_x(s, u, G)$ (ou seja, a distância de s a u em G, sendo que x faz o papel de comprimento em cada arco). Mostraremos que (x, z) é viável em *PL2*. Por (4.3), (x, z) satisfaz (4.13). Sejam uv em E_G e st em Q. Por desigualdade triangular,

$$z^{st}_u - z^{st}_v + x_{uv} = dist_x(s, u, G) - dist_x(s, v, G) + x_{uv} \geq 0.$$

Portanto, (x, z) satisfaz (4.12). Dado um par st em Q, seja P^* um caminho mínimo de s a t sendo que x faz o papel de comprimento em cada arco. Levando em conta (4.2),

$$1 \leq \sum_{uv \in P^*} x_{uv} = dist_x(s, t, G) = dist_x(s, t, G) - dist_x(s, s, G) = z^{st}_t - z^{st}_s.$$

Portanto (x, z) satisfaz (4.11). Como (x, z) satisfaz (4.11), (4.12) e (4.13), então (x, z) é viável em *PL2*.

Suponha que (x, z) é viável em *PL2*. Por (4.13), x satisfaz (4.3). Dado um par st em Q, seja P um caminho de s a t e P^* um caminho mínimo de s a t, sendo que x faz o papel de comprimento em cada arco. Levando em conta (4.11) e (4.12),

$$\sum_{uv \in P} x_{uv} \geq \sum_{uv \in P^*} x_{uv} \geq \sum_{uv \in P} (z^{st}_v - z^{st}_u) = z^{st}_t - z^{st}_s \geq 1.$$

Portanto, x satisfaz (4.2). Como x satisfaz (4.2) e (4.3), então x é viável em *PL1*.

Como a função objetivo de *PL1* é a mesma que a função objetivo de *PL2*, concluímos

que os dois programas lineares são equivalentes. Defina

$$\mu^*(G, c, Q)$$

como o custo de uma solução ótima de qualquer um dos programas lineares *PL1* e *PL2*.

Comparemos agora o número de restrições de cada programa linear. O número de restrições de *PL2* é $k(m+1)+m$. Embora seja polinomial, essa quantidade é perto de n^4 em instâncias onde k e m estão perto de n^2. Por outro lado, embora em algumas instâncias o número de restrições de *PL1* seja exponencial, na prática muitas vezes esse número é menor que $k(m+1)+m$. Foi feita uma análise experimental para comparar o tempo de execução de ambas formulações. A conclusão é que, para a maioria de instâncias, resolver *PL1* demora menos tempo que resolver *PL2*. Para mais detalhes dos resultados dessa implementação, veja o capítulo 9.

4.4 Programas lineares inteiros

É claro que nenhum dos programas lineares *PL1* e *PL2* resolvem o problema MCM, já que nem sempre a solução devolvida é inteira. Para ver isso, considere o seguinte exemplo da formulação de *PL1* baseado na rede da figura 2.1:

$$\text{minimize} \sum_{i=1}^{9} x_i \tag{4.26}$$

sujeito a

$$\begin{aligned} x_{30} + x_{01} + x_{12} + x_{24} &\geq 1, \\ x_{51} + x_{12} + x_{20} + x_{06} &\geq 1, \\ x_{72} + x_{20} + x_{01} + x_{18} &\geq 1, \\ x_{30}, x_{01}, x_{12}, x_{24}, x_{51}, x_{12}, x_{20}, x_{06}, x_{72}, x_{20}, x_{01}, x_{18} &\geq 0. \end{aligned}$$

Nesse exemplo $Q = \{34, 56, 78\}$. Uma solução ótima para esse programa linear é $x_{01} = x_{12} = x_{20} = 0.5$ e $x_{uv} = 0$ em caso contrário. Porém, uma solução inteira ótima é $x_{01} = x_{12} = 1$ e $x_{uv} = 0$ em caso contrário.

Se em *PL1* substituímos (4.3) por

$$x_{uv} \in \{0, 1\} \quad \text{para cada } uv \in E_G, \tag{4.27}$$

obtemos o programa linear inteiro *PI1*, que é uma reformulação do problema MCM.

Se em *PL2* substituímos (4.13) por

$$x_{uv} \in \{0, 1\} \quad \text{para cada } uv \in E_G, \tag{4.28}$$

e adicionamos

$$z_u^{st} \in \mathbb{Z} \text{ para cada } u \in V_G \text{ e cada } st \in Q, \tag{4.29}$$

obtemos o programa linear inteiro *PI2*, que é uma reformulação do problema MCM.

Mediante um argumento similar ao exibido na seção 4.3, podemos afirmar que os programas lineares inteiros *PI1* e *PI2* são equivalentes. É claro que $\mu(G, c, Q)$ é o custo de uma solução ótima de qualquer um dos programas lineares *PI1* e *PI2*.

O *gap de integralidade* para uma instância (G, c, Q) é a razão entre $\mu(G, c, Q)$ e $\mu^*(G, c, Q)$. O algoritmo MFMC-Iterado (seção 3.1) dá um limitante superior trivial de k para o gap de integralidade de qualquer instância. Saks *et al.* [SSZ04] mostraram que esse limitante superior não pode ser melhorado. Eles mostraram que, para cada ϵ tal que $0 < \epsilon < 1$, existe uma família de instâncias cujo gap de integralidade é maior ou igual a

$$k(1 - \epsilon).$$

Em função de n, Chuzhoy e Khanna [CK07] mostraram que, para cada ϵ tal que $0 < \epsilon < 1$, existe uma família de instâncias cujo gap de integralidade é maior ou igual a

$$n^{1/7} / \log n.$$

4.5 Algoritmo de Bellmore, Greenberg e Jarvis

Já que o programa linear *PI1* reformula o problema MCM, basta resolvê-lo para resolver o problema MCM. Porém, existe uma dificuldade em enumerar todos os Q-caminhos da rede, já que essa quantidade pode ser exponencial no tamanho da rede. Não é preciso conhecer explicitamente todos os Q-caminhos para resolver *PI1*. O seguinte algoritmo, apresentado por Bellmore *et al.* [BGJ70], resolve *PI1* enumerando implicitamente os Q-caminhos da rede. O algoritmo recebe uma rede (G, c, Q) e devolve um multicorte c-mínimo de (G, Q).

Algoritmo Multicorte-BellGreJar(G, c, Q)

1 $\mathcal{P} \leftarrow \emptyset$
2 $X \leftarrow \emptyset$

```
3  repita
4      X ← IntProgLin(c, P)
5      P' ← P
6      para todo st ∈ Q faça
7          se existe caminho P de s a t em G − X então
8              P ← P ∪ {E_P}
9  até que P = P'
10 devolva X
```

O algoritmo trabalha com uma coleção $\mathcal{P}$ de conjuntos de arcos. Dada uma tal coleção $\mathcal{P}$, dizemos que um conjunto de arcos X é uma *cobertura* de $\mathcal{P}$ se $X \cap P \neq \emptyset$ para todo $P \in \mathcal{P}$. Também dizemos que X *cobre* $\mathcal{P}$. Uma cobertura X^* de $\mathcal{P}$ é c-mínima se $c(X^*) \leq c(X)$ para toda cobertura X de $\mathcal{P}$. Se $\widehat{\mathcal{P}}$ é a coleção dos conjuntos de arcos de todos os Q-caminhos em G, então é claro que toda cobertura de $\widehat{\mathcal{P}}$ é um multicorte de (G, Q).

O algoritmo IntProgLin, na linha 4, recebe uma coleção $\mathcal{P}$ e devolve uma cobertura c-mínima de $\mathcal{P}$. A formulação do programa linear inteiro para resolver o problema da cobertura c-mínima de $\mathcal{P}$ é formalmente igual à formulação de *PI1* visto na seção 4.4. O algoritmo IntProgLin pode fazer uso de qualquer método de programação linear inteira para resolver o programa linear inteiro que formula o problema da cobertura c-mínima. Pela definição de IntProgLin, em cada iteração, imediatamente depois da execução da linha 4,

$$X \text{ é uma cobertura } c\text{-mínima de } \mathcal{P}. \tag{4.30}$$

Teorema 4.1 No final do algoritmo, X é um multicorte c-mínimo de (G, Q).

Prova: As linhas 6-8 garantem que, no final do algoritmo, X é um multicorte de (G, Q). Suponha por contradição que existe um multicorte $\widehat{X}$ de (G, Q) tal que $c(\widehat{X}) < c(X)$. Seja $\widehat{\mathcal{P}} = \{E_P : P \text{ é um } Q\text{-caminho em } G\}$. É claro que $\widehat{X}$ cobre $\widehat{\mathcal{P}}$. Como, no final do algoritmo, $\mathcal{P} \subseteq \widehat{\mathcal{P}}$, então $\widehat{X}$ cobre $\mathcal{P}$. Mas, por (4.30), X é uma cobertura c-mínima de $\mathcal{P}$. Contradição. □

O número de iterações do processo iterativo das linhas 3-9 é, no pior dos casos, o número de Q-caminhos em G, que pode ser maior ou igual que 2^n (veja seção 4.1).

Na implementação do algoritmo fizemos testes com famílias de instâncias tomadas da Internet. Nesses testes, encontramos alguns que demoraram muito em ser resolvidos (um deles com apenas 30 vértices). Percebemos que a principal causa para esta demora foi o número excessivo de execuções de IntProgLin. Isso ocasionou ter que fazer uma heurística para melhorar o desempenho do algoritmo. A heurística não adiciona apenas um caminho (linha 8 do algoritmo), senão uma coleção maximal de caminhos para cada

par st. Para mais detalhes dos resultados da implementação do algoritmo, veja o capítulo 9. Para detalhes do código da implementação, veja o apêndice B.

Capítulo 5

Algoritmo de Aneja e Vemuganti, Bellmore e Ratliff

Neste capítulo apresentamos o algoritmo concebido por Aneja e Vemuganti [AV77], que resolve de maneira exata o Problema do Multicorte Dirigido de Custo Mínimo (MCM). Ele está baseado no algoritmo exibido por Bellmore e Ratliff [BR71], que faz uso de técnicas similares às usadas no método Simplex para redes. O seguinte algoritmo é apenas uma versão básica dos algoritmos mostrados nos artigos, já que em ambos artigos foram propostos métodos mais eficientes para algumas partes do algoritmo. Mencionamos alguns desses na seção 5.4.

5.1 O algoritmo

O algoritmo MULTICORTE-ANEVEM-BELLRAT recebe uma rede (G, c, Q) e devolve um multicorte c-mínimo de (G, Q).

Algoritmo MULTICORTE-ANEVEM-BELLRAT(G, c, Q)

1 $\mathcal{R} \leftarrow \emptyset$
2 $X_0 \leftarrow E_G$

3 **enquanto** $\emptyset \notin \mathcal{R}$ **faça**
4 $X \leftarrow E_G$
5 **para todo** $e \in X$ **faça**
6 $X \leftarrow X \setminus \{e\}$
7 **para todo** $R \in \mathcal{R}$ **faça**
8 **se** $X \cap R = \emptyset$ **então**
9 $P_e \leftarrow R$
10 $X \leftarrow X \cup \{e\}$
11 **interrompa**
12 **se** P_e não está definido **então**
13 **para todo** $st \in Q$ **faça**
14 **se** existe um caminho P de s a t em $G - X$ **então**
15 $P_e \leftarrow E_P$
16 $X \leftarrow X \cup \{e\}$
17 **interrompa**
18 **se** $c(X) < c(X_0)$ **então**
19 $X_0 \leftarrow X$
20 **para todo** $f \in E_G$ **faça**
21 $c'_f \leftarrow c_f$
22 **para todo** $e \in X$ **faça**
23 **se** $f \in P_e$ **então**
24 $c'_f \leftarrow c'_f - c_e$
25 $R \leftarrow \{f \in E_G : c'_f < 0\}$
26 $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
27 **devolva** X_0

5.2 Análise do algoritmo

Dada uma coleção de conjuntos de arcos $\widehat{\mathcal{P}}$, dizemos que um conjunto de arcos X é uma *cobertura* de $\widehat{\mathcal{P}}$ se $X \cap P \neq \emptyset$ para todo $P \in \widehat{\mathcal{P}}$. Seja $\mathcal{P} = \{E_P : P \text{ é um } Q\text{-caminho em } G\}$. É fácil ver que um conjunto X é multicorte de (G, Q) se e só se X é cobertura de $\mathcal{P}$.

Em cada iteração, imediatamente depois da execução do bloco de linhas 5-17, podemos ver que

$$X \text{ é uma cobertura de } \mathcal{P} \cup \mathcal{R}. \tag{5.1}$$

Podemos ver também que X é uma cobertura minimal e portanto para cada $e \in X$ existe um conjunto de arcos $P_e \in \mathcal{P} \cup \mathcal{R}$ tal que

$$X \cap P_e = \{e\}. \tag{5.2}$$

O bloco de linhas 20-24 calcula os "custos residuais" dos arcos, exatamente como faria o algoritmo Simplex. Em cada iteração, imediatamente depois da execução do bloco de linhas 20-24, para cada arco f defina $X(f) = \{e \in X : f \in P_e\}$. Esse conjunto está bem definido por causa de (5.2). É fácil ver que, imediatamente depois da execução do bloco de linhas 20-24,

$$c'_f = c_f - \sum_{e \in X(f)} c_e. \tag{5.3}$$

De (5.2) e (5.3) deduzimos que, imediatamente depois da execução do bloco de linhas 20-24, para todo $e \in X$,

$$c'_e = 0. \tag{5.4}$$

Lema 5.1 Em cada iteração, imediatamente depois da execução da linha 25, se $\widehat{X}$ é um multicorte de (G, Q) que cobre $\mathcal{R}$ e $c(\widehat{X}) < c(X)$, então $\widehat{X} \cap R \neq \emptyset$.

Prova: Note que

$$\begin{aligned}
\sum_{f \in \widehat{X}} \big(\sum_{e \in X(f)} c_e \big) &= \sum_{e \in X} (c_e \cdot |\widehat{X} \cap P_e|) \\
&\geq \sum_{e \in X} c_e && (5.5) \\
&= c(X), && (5.6)
\end{aligned}$$

onde (5.5) vale pois $\widehat{X}$ é um multicorte de (G, Q) que cobre $\mathcal{R}$. Portanto,

$$\begin{aligned}
c'(\widehat{X}) &= \sum_{f \in \widehat{X}} c'_f \\
&= \sum_{f \in \widehat{X}} c_f - \sum_{f \in \widehat{X}} \sum_{e \in X(f)} c_e && (5.7) \\
&= c(\widehat{X}) - \sum_{f \in \widehat{X}} \sum_{e \in X(f)} c_e \\
&\leq c(\widehat{X}) - c(X), && (5.8)
\end{aligned}$$

onde (5.7) vale por (5.3), e (5.8) vale por (5.6). Finalmente, suponha por contradição que $\widehat{X} \cap R = \emptyset$. Então,

$$\begin{aligned}
0 &\leq \sum_{f \in \widehat{X}} c'_f \\
&= c'(\widehat{X}) \\
&\leq c(\widehat{X}) - c(X) && (5.9) \\
&< 0,
\end{aligned}$$

onde (5.9) vale por (5.8). Contradição. Com isso, concluímos a prova do lema. □

Lema 5.2 No início de cada iteração do processo iterativo das linhas 3-26 são válidas as seguintes invariantes:

(I_1) X_0 é multicorte de (G, Q),

(I_2) para qualquer multicorte $\widehat{X}$ de (G, Q), se $c(\widehat{X}) < c(X_0)$ então $\widehat{X}$ cobre $\mathcal{R}$.

Prova: A prova de (I_1) segue diretamente de (5.1) e o fato de que toda cobertura de $\mathcal{P}$ é um multicorte de (G, Q). Para a prova de (I_2) precisamos mais cuidado.

No início da primeira iteração a invariante (I_2) é válida já que $\mathcal{R} = \emptyset$. Suponha agora que a invariante é válida no início de uma iteração qualquer das linhas 3-26. Vamos mostrar que a invariante é válida no início da iteração seguinte. Seja $\widehat{X}$ um multicorte de (G, Q) tal que, no começo da linha 20, $c(\widehat{X}) < c(X_0)$. Precisamos provar que, imediatamente depois da execução da linha 26, $\widehat{X}$ cobre $\mathcal{R}$. É fácil ver que X_0 não incrementa seu valor no bloco de linhas 18-19 e portanto, no início da iteração, $c(\widehat{X}) < c(X_0)$. Por causa de (I_2), no início da iteração, $\widehat{X}$ é um multicorte de (G, Q) que cobre $\mathcal{R}$. Pelo lema 5.1, imediatamente depois da execução da linha 25, $\widehat{X} \cap R \neq \emptyset$. A prova segue do fato que na linha 26, $\mathcal{R}$ aumenta seu valor em exatamente R. □

Lema 5.3 O processo iterativo das linhas 3-26 é finito.

Prova: Precisamos provar que a linha 27 do algoritmo é executada. Suponha por contradição que a linha 27 do algoritmo não é executada. Começaremos mostrando o seguinte fato: em cada iteração, imediatamente depois da execução da linha 25, $R \notin \mathcal{R}$. Suponha por contradição que, imediatamente depois da execução da linha 25, $R \in \mathcal{R}$. Então, como X cobre $\mathcal{R}$, $X \cap R \neq \emptyset$. Mas isso é impossível já que por (5.4), $c'_e = 0$ para todo $e \in X$, e pela linha 25, $c'_e < 0$ para todo $e \in R$. Contradição. Como o número de subconjuntos de E_G é finito, então em alguma iteração $R = \emptyset$ e a linha 27 é executada. □

Teorema 5.1 No final do algoritmo, X_0 é um multicorte c-mínimo de (G, Q).

Prova: Por (I_1), no final do algoritmo, X_0 é um multicorte de (G, Q). Suponha por contradição que existe um multicorte $\widehat{X}$ de (G, Q) tal que $c(\widehat{X}) < c(X_0)$. Por (I_2), no final do algoritmo, $\widehat{X}$ cobre $\mathcal{R}$. Mas isso é impossível já que, no final do algoritmo, $\emptyset \in \mathcal{R}$. □

5.3 Análise da complexidade

Na prova do lema 5.3, vimos que na linha 25 de cada iteração sempre é obtido um R distinto dos obtidos nas iterações anteriores. Então, o número de iterações do processo

iterativo das linhas 3-26 é, no pior dos casos, o número de subconjuntos de E_G, ou seja 2^m.

5.4 Pontos fracos do algoritmo

O algoritmo apresenta dois pontos fracos. Bellmore e Ratliff propuseram heurísticas para melhorar esses pontos.

O primeiro ponto fraco está na linha 4. Nessa linha, a cobertura X de $\mathcal{P} \cup \mathcal{R}$ é o total de arcos do digrafo. É claro que para que o algoritmo esteja correto, basta considerar uma cobertura qualquer de $\mathcal{P} \cup \mathcal{R}$. Bellmore e Ratliff propõem, entre outras opções, formular e resolver uma relaxação linear para o problema da cobertura c-mínima. A formulação desse programa linear, que chamaremos de *PL1′*, é similar à do programa linear *PL1* visto na seção 4.1. Seja x^* uma solução ótima para o programa linear *PL1′* e seja $X = \{e \in E_G : x^*_e > 0\}$. É claro que X é uma cobertura de $\mathcal{P} \cup \mathcal{R}$. Um ponto ainda obscuro é a maneira de resolver esse programa linear, já que não temos todos os Q-caminhos enumerados explicitamente. Aneja e Vemuganti [AV77] propõem um algoritmo que resolve o programa linear *PL1′* enumerando implicitamente os caminhos.

O segundo ponto fraco está no bloco de linhas 5-17. Vimos que, imediatamente depois da execução do bloco de linhas 5-17, X é uma cobertura minimal de $\mathcal{P} \cup \mathcal{R}$. É claro que há muitas coberturas minimais diferentes. Dois fatores vão alterar a maneira em que é encontrado um tal X minimal: (1) a ordem em que são processados os arcos na linha 5 e (2) qual é o P_e, para cada e, achado no bloco de linhas 6-17. Para ambos casos Bellmore e Ratliff procuraram achar X de um jeito tal que consiga-se o melhor conjunto R possível na linha 25. Infelizmente não é fácil saber que propriedade deve ter esse R. Bellmore e Ratliff procuraram escolher X de um jeito tal que $\sum_{e \in R} x^*_e$ seja o menor possível. A razão para essa escolha vem do fato que se $\sum_{e \in R} x^*_e < 1$ então a solução fracionária ótima obtida na iteração atual não poderá ser obtida na próxima iteração. Dessa maneira obtemos um melhor limitante inferior para o multicorte c-mínimo obtido na próxima iteração.

5.5 Implementação

O algoritmo não foi implementado e portanto não observamos seu desempenho na prática.

Capítulo 6

Multicortes dirigidos de custo mínimo em árvores divergentes

Foi provado no capítulo 2 que o Problema do Multicorte Dirigido de Custo Mínimo (MCM) é NP-difícil. Contudo, se o digrafo é uma árvore divergente, ou seja, se é uma árvore que admite um vértice, a raiz, tal que existe um caminho da raiz até qualquer outro vértice, existe um algoritmo que devolve um multicorte c-mínimo e consome tempo polinomial no tamanho da árvore. Esse algoritmo foi proposto por Costa *et al.* [CLR03] e faz uso da técnica primal-dual. Neste capítulo analisaremos esse algoritmo, sua correção e complexidade.

6.1 Árvores divergentes

Um digrafo G é uma *árvore divergente* se tem as seguintes propriedades:

(1) existe um único vértice de G, chamado raiz, cujo grau de entrada é 0,

(2) todo os demais vértices de G têm grau de entrada 1,

(3) para todo vértice v de G existe um caminho da raiz até v,

onde o *grau de entrada* de um vértice v é o número de arcos do digrafo que entram em v. Segue da definição que o caminho da raiz até um vértice qualquer é único.

Se st é um par de terminais em uma árvore divergente, existe no máximo um caminho de s a t. Isso pode ser deduzido intuitivamente. Comece por t e ande na direção oposta ao arco que entra nele. Desse modo, uma das duas afirmações será verdadeira: alcançaremos s e portanto esse é o único caminho de s a t, ou alcançaremos a raiz e não há caminho de s a t.

6.2 Relaxação linear

Na seção 4.1 mostramos uma relaxação linear para o problema MCM em digrafos arbitrários. Mas o que acontece se o digrafo é uma árvore divergente? No caso de árvores

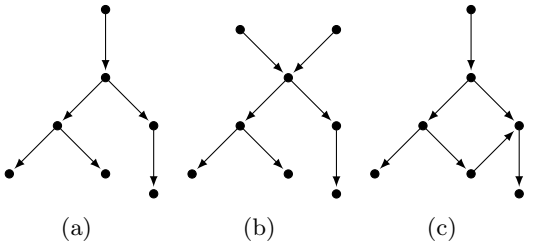

Figura 6.1: Só o primeiro digrafo é uma árvore divergente.

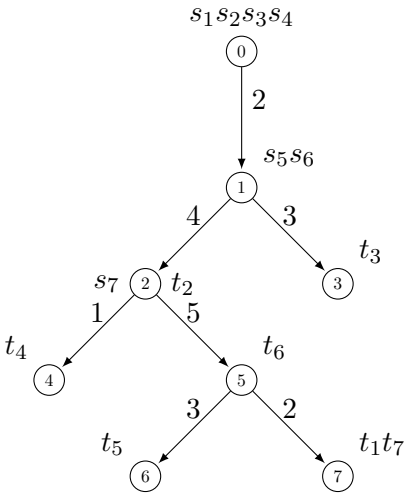

Figura 6.2: Exemplo de árvore divergente com uma função custo e um conjunto de pares de terminais (essa figura está no artigo de Costa *et al.* [CLR03]). Um multicorte de custo mínimo é $\{01, 25\}$ e tem custo 7.

divergentes existe no máximo um caminho para cada par de terminais. Portanto, podemos simplificar essa relaxação linear no caso de árvores divergentes e observar a relação primal-dual nessa estrutura.

Seja (G, c, Q) uma rede tal que G é uma árvore divergente com raiz r e $Q = \{s_1t_1, s_2t_2, \ldots, s_kt_k\}$. Para cada $i \in \{1, 2, \ldots, k\}$, denote por A_i o conjunto de arcos do único caminho de s_i a t_i em G. Suporemos que existe tal caminho para todo i, caso contrário podemos eliminar de Q os pares de terminais que não têm essa propriedade. O programa linear *PL1* (veja seção 4.1) assume a seguinte forma: encontrar um vetor racional x indexado por E_G que

$$\text{minimize} \sum_{e \in E_G} c_e x_e \tag{6.1}$$

sujeito a

$$\sum_{e \in A_i} x_e \geq 1 \quad \text{para cada } i \in \{1, 2, \ldots, k\}, \tag{6.2}$$

$$x_e \geq 0 \quad \text{para cada } e \in E_G. \tag{6.3}$$

Chame esse programa linear de *PLA*. O dual do *PLA*, chamado de *PDA* , assume a seguinte forma: encontrar um vetor racional y indexado por $\{1, 2, \ldots, k\}$ que

$$\text{maximize} \sum_{i=1}^{k} y_i \tag{6.4}$$

sujeito a

$$\sum_{i:e\in A_i} y_i \leq c_e \quad \text{para cada } e \in E_G, \tag{6.5}$$

$$y_i \geq 0 \quad \text{para cada } i \in \{1, 2, \ldots, k\}. \tag{6.6}$$

A seguinte inequação (dualidade fraca) deriva-se diretamente do lema 4.1: para cada x viável em *PLA* e cada y viável em *PDA*,

$$\sum_{i=1}^{k} y_i \leq \sum_{e\in E_G} c_e x_e. \tag{6.7}$$

Sejam x^* e y^* soluções ótimas de *PLA* e *PDA* respectivamente. A propriedade de folgas complementares [Chv83, p.62] nos diz que, para cada $i \in \{1, 2, \ldots, k\}$,

$$\text{se } y_i^* > 0 \text{ então } \sum_{e\in A_i} x_e^* = 1, \tag{6.8}$$

e, para cada $e \in E_G$,

$$\text{se } x_e^* > 0 \text{ então } \sum_{i:e\in A_i} y_i^* = c_e. \tag{6.9}$$

A relação (6.8) diz que se x^* é um vetor de zeros e uns, ou seja, o vetor característico de um multicorte, então cada A_i tal que $y_i > 0$ é cortado por exatamente um arco do multicorte. A relação (6.9) diz que todos os arcos de um multicorte c-mínimo estão "saturados".

6.3 O algoritmo

O algoritmo procura implementar as relações de folgas complementares (6.8) e (6.9). Ele recebe uma rede (G, c, Q) tal que G é uma árvore divergente e $Q = \{s_1t_1, s_2t_2, \ldots, s_kt_k\}$ e devolve um multicorte c-mínimo de (G, Q). Ajuste a notação de modo que, para cada i, haja um caminho de s_i a t_i em G. Sendo r a raiz da árvore, defina $dist(v) := dist(r, v, G)$ para cada vértice v e ajuste a notação de modo que $dist(s_1) \leq dist(s_2) \leq \ldots \leq dist(s_k)$.

Algoritmo MULTICORTE-DE-ÁRVORE(G, c, Q)

```
1  para todo a ∈ E_G faça
2      z_a ← 0
3  para i ← 1 até k faça
4      P_i ← caminho de s_i a t_i em G
5      A_i ← E_{P_i}
6  X ← ∅
7  para i ← k decrescendo até 1 faça
8      y_i ← min{c_a − z_a : a ∈ A_i}
9      para todo a ∈ A_i faça
10         z_a ← z_a + y_i
11         se c_a − z_a = 0 então
12             X ← X ∪ {a}
13 para i ← 1 até k faça
14     se y_i > 0 então
15         e ← primeiro arco de P_i que está em X
16         X ← X \ (A_i \ {e})
17 devolva X
```

6.4 Análise do algoritmo

Lema 6.1 No final do primeiro processo iterativo (linhas 7-12) do algoritmo:

(I'_2) $\sum_{j:a\in A_j} y_j \leq c_a$ para cada $a \in E_G$,

(I'_3) $\sum_{j:a\in A_j} y_j = c_a$ para cada $a \in X$,

(I'_4) X é um multicorte de (G, Q),

(I'_5) para cada par (j, l) tal que $1 \leq j < l \leq k$, se $y_j > 0$ então $X \cap (A_l \setminus A_j) \neq \emptyset$.

Prova: Considere as seguintes invariantes no início de cada iteração do primeiro processo iterativo (linhas 7-12) do algoritmo:

(I_1) $z_a = \sum\{y_j : a \in A_j \text{ e } i < j \leq k\}$ para cada $a \in E_G$,

(I_2) $z_a \leq c_a$ para cada $a \in E_G$,

(I_3) $z_a = c_a$ para cada $a \in X$,

(I_4) $X \cap A_j \neq \emptyset$, para $j = i+1, \ldots, k$,

(I_5) para cada par (j, l) tal que $i < j < l \leq k$, se $y_j > 0$ então $X \cap (A_l \setminus A_j) \neq \emptyset$.

Começaremos mostrando que, no início da primeira iteração do primeiro processo iterativo, as invariantes $(I_1), (I_2), (I_3), (I_4)$ e (I_5) são válidas. A invariante (I_1) é vacuamente válida já que, no início da primeira iteração, $i = k$ e $z = 0$. A invariante (I_2) é

válida já que, de acordo com a nossa definição de rede (seção 2.1), temos $c_a \geq 0$ para toda a e portanto, no início da primeira iteração, para cada $a \in E_G$, $z_a = 0 \leq c_a$. A invariante (I_3) é vacuamente válida já que, no início da primeira iteração, temos $X = \emptyset$. As invariantes (I_4) e (I_5) são vacuamente válidas já que, no início da primeira iteração, temos $i = k$.

Suponha agora que as invariantes $(I_1), (I_2), (I_3), (I_4)$ e (I_5) são válidas no início de uma iteração qualquer do primeiro processo iterativo (linhas 7-12). Vamos mostrar que as invariantes são válidas no início da iteração seguinte.

Prova de (I_1): Se $a \notin A_i$, o valor de z_a não muda e a invariante é válida no início da iteração seguinte. Se $a \in A_i$, o valor de z_a no início da iteração atual é $\sum\{y_j : a \in A_j$ e $i < j \leq k\}$. No final da iteração, o valor de i diminui em 1 e o valor de z_a aumenta em y_i e portanto a invariante é válida no início da iteração seguinte.

Prova de (I_2): Se $a \notin A_i$, o valor de z_a não muda e a invariante é válida no início da iteração seguinte. Se $a \in A_i$, temos que $z_a \leq c_a$ no início da iteração atual. No final da iteração, o valor de z_a incrementa-se em y_i. Como y_i é no máximo $c_a - z_a$, então z_a é no máximo c_a no início da iteração seguinte.

Prova de (I_3): As linhas 11 e 12 garantem que um arco a é acrescentado a X na iteração atual somente se $z_a = c_a$. Portanto, se $a \in X$ no início da iteração seguinte então $z_a = c_a$.

Prova de (I_4): Se $j > i$ então, por hipótese, $X \cap A_j$ no início da iteração seguinte $\neq \emptyset$. Se $j = i$, seja $b \in A_i$ tal que $c_b - z_b = \min\{c_a - z_a : a \in A_i\}$. Depois da execução da linha 10, $z_b = c_b$ e portanto $b \in X$ depois da execução da linha 12. Concluímos que, no final da iteração atual, $X \cap A_i \neq \emptyset$ e, portanto, a invariante é válida no início da iteração seguinte.

Prova de (I_5): Basta mostrar que, no final da iteração atual, (I_5) vale para os pares (i, l) com $i < l \leq k$. Supondo $y_i > 0$, provaremos que, para cada $l > i$, $X \cap (A_l \setminus A_i) \neq \emptyset$ no final da iteração atual. Se $A_l \cap A_i = \emptyset$, então $X \cap (A_l \setminus A_i) = X \cap A_l \neq \emptyset$ em virtude de (I_4). Agora considere o caso em que $A_l \cap A_i \neq \emptyset$. Como $y_i > 0$, no início da iteração atual temos $z_a < c_a$ para cada a em A_i. Isso, somado à invariante (I_3), resulta em $X \cap A_i = \emptyset$ no início da iteração atual. A invariante (I_4) nos diz que $X \cap A_l \neq \emptyset$, com o que concluímos que $X \cap (A_l \setminus A_i) \neq \emptyset$ no início e portanto também no final da iteração atual.

Com isso, concluímos a prova do lema 6.1. □

Lema 6.2 No final do algoritmo:

(I_6') X é um multicorte de (G, Q),

(I_7') para $i = 1, \ldots, k$, se $y_i > 0$ então $|X \cap A_i| \leq 1$.

Prova: Considere as seguintes invariantes no início de cada iteração segundo processo iterativo (linhas 13-16) do algoritmo:

(I_6) $|X \cap A_l| \geq 1$ para $l = 1, \ldots, k$,

(I_7) para $l = 1, \ldots, i-1$, se $y_l > 0$ então $|X \cap A_l| \leq 1$,

(I_8) para cada par (j, l) tal que $i \leq j < l \leq k$, se $y_j > 0$ então $X \cap (A_l \setminus A_j) \neq \emptyset$.

Começaremos mostrando que, no início da primeira iteração do segundo processo iterativo, as invariantes $(I_6), (I_7)$ e (I_8) são válidas: A invariante (I_6) é válida já que, por (I'_4), temos que, no início da primeira iteração, $X \cap A_j \neq \emptyset$ para $j = 1, \ldots, k$. A invariante (I_7) é vacuamente válida já que, no início da primeira iteração, $i = 1$. A invariante (I_8) é válida por (I'_5).

Suponha agora que as invariantes $(I_6), (I_7)$ e (I_8) são válidas no início de uma iteração qualquer do segundo processo iterativo (linhas 13-16). Vamos mostrar que as invariantes são válidas no início da iteração seguinte.

Observe que há uma ordem parcial no conjunto dos arcos da árvore: um arco *a precede* um arco b se o caminho da raiz da árvore até a ponta final de b contém o arco a. É claro que os arcos de todo conjunto A_p estão na relação de ordem: se a e b pertencem a A_p, então a precede b ou b precede a.

Prova de (I_6): Seja l um índice em $\{1, 2, \ldots, k\}$, vamos mostrar que $X \cap A_l \neq \emptyset$ no final da iteração atual. Se $y_i = 0$, então isso é verdade pois X não muda na iteração atual. Se $y_i > 0$, basta provar que, no início da iteração atual, $(X \setminus (A_i \setminus \{e\})) \cap A_l \neq \emptyset$, sendo e o arco escolhido na linha 15. Temos dois casos. Se $X \cap (A_l \setminus A_i) \neq \emptyset$ então, no início da iteração atual,

$$(X \setminus (A_i \setminus \{e\})) \cap A_l \supseteq (X \setminus A_i) \cap A_l = X \cap (A_l \setminus A_i) \neq \emptyset.$$

Se $X \cap (A_l \setminus A_i) = \emptyset$ então $l \leq i$ por causa de (I_8). Observe que por (I_6), no início da iteração atual, $X \cap A_l \neq \emptyset$. Seja f um arco em $X \cap A_l$, como $X \cap A_l \subseteq A_i$ então $f \in X \cap A_i$. Logo, $e = f$ ou e precede f em A_i. Como $f \in X \cap A_l \cap A_i$ e $l \leq i$, temos $e \in A_l$. Portanto, no início da iteração atual,

$$(X \setminus (A_i \setminus \{e\})) \cap A_l \supseteq \{e\} \cap A_l \neq \emptyset.$$

Com isso, concluímos a prova de (I_6).

Prova de (I_7): Se $y_i \leq 0$, então X não muda na iteração atual, portanto a invariante é válida no início da iteração seguinte. Se $y_i > 0$, então é claro que $|X \cap A_i| = 1$ no final da iteração atual. Além disso, para todo $l < i$, a propriedade $|X \cap A_l| \leq 1$ se preserva no início da iteração seguinte, pois X não aumenta durante a iteração atual.

Prova de (I_8): Se $y_i = 0$, então (I_8) continua válida no início da iteração seguinte

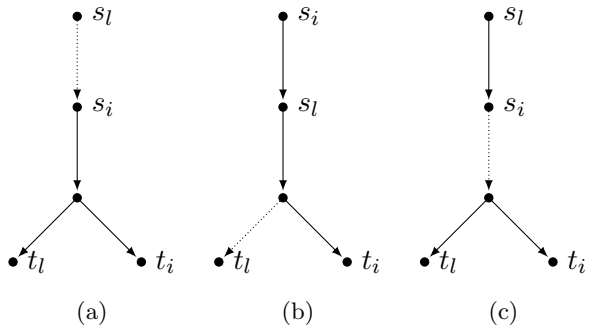

Figura 6.3: Prova de (I_6). As figuras mostram o que acontece no início de uma iteração. Um arco pontilhado indica que ele está em X. As figuras (a) e (b) mostram o que acontece no caso $X \cap (A_l \setminus A_i) \neq \emptyset$. A figura (c) mostra o que acontece no caso $X \cap (A_l \setminus A_i) = \emptyset$.

pois X não muda de valor na iteração atual. Suponha agora que $y_i > 0$. Seja (j, l) um par tal que $i < j < l \leq k$. Suponha também que $y_j > 0$. Para provar que (I_8) continua valendo no início da iteração seguinte, basta mostrar que no início da iteração atual temos $(X \setminus A_i) \cap (A_l \setminus A_j) \neq \emptyset$.

Para continuar com a prova, precisamos de um fato importante, cuja prova será deixada para o final:

$$\text{ou } A_l \cap A_i \subseteq A_j, \text{ ou } A_l \cap A_j \subseteq A_i. \tag{6.10}$$

De acordo com (6.10), basta considerar os dois casos a seguir. Se $A_l \cap A_j \subseteq A_i$, então temos $A_l \setminus A_j = A_l \setminus (A_l \cap A_j) \supseteq A_l \setminus A_i$. Portanto, no início da iteração atual,

$$(X \setminus A_i) \cap (A_l \setminus A_j) \supseteq (X \setminus A_i) \cap (A_l \setminus A_i) \neq \emptyset,$$

já que o caso $j = i$ de (I_8) vale no início da iteração atual. Se $A_l \cap A_i \subseteq A_j$, temos $A_l \setminus A_i = A_l \setminus (A_l \cap A_i) \supseteq A_l \setminus A_j$. Portanto, no início da iteração atual,

$$\begin{aligned}
(X \setminus A_i) \cap (A_l \setminus A_j) &= (X \setminus A_j) \cap (A_l \setminus A_i) \\
&\supseteq (X \setminus A_j) \cap (A_l \setminus A_j) \\
&= X \cap (A_l \setminus A_j) \\
&\neq \emptyset,
\end{aligned}$$

onde a última desigualdade vale porque (I_8) é válida no início da iteração atual.

Finalizamos com a prova de (6.10). Precisamos de um fato prévio fácil de ver. Sejam a e b arcos de A_l tais que a precede b. Para todo índice $i' \leq l$,

$$\text{se } b \in A_{i'} \text{ então também } a \in A_{i'}. \tag{6.11}$$

Suponha que $A_l \cap A_i \not\subseteq A_j$. Seja a um elemento de $(A_l \cap A_i) \setminus A_j$. Seja b um elemento de $A_l \cap A_j$. Vamos provar que $b \in A_i$. Como $a \notin A_j$, $b \in A_j$ e $j \leq l$, então, por (6.11), a não precede b. Portanto b precede a. Como $a \in A_l$ e $i \leq l$ então, por (6.11), $b \in A_i$. Com isso, concluímos a prova de (6.10) e de (I_8).

Com isso, concluímos a prova do lema 6.2. □

Teorema 6.1 O algoritmo MULTICORTE-DE-ÁRVORE devolve um multicorte c-mínimo de (G, Q).

Prova: Por (I'_6), no final do algoritmo, o vetor característico x de X é viável em PLA. É fácil ver que (I'_2) continua valendo no final de algoritmo e portanto y é viável em PDA. Note que

$$\begin{aligned}
\sum_{e \in E_G} c_e x_e &= c(X) \\
&= \sum_{a \in X} c_a \\
&= \sum_{a \in X} \Big(\sum_{i : a \in A_i} y_i \Big) \qquad (6.12) \\
&= \sum_{i=1}^{k} \Big(\sum_{a \in X \cap A_i} y_i \Big) \\
&= \sum_{i=1}^{k} \big(y_i \cdot |X \cap A_i| \big) \\
&\leq \sum_{i=1}^{k} y_i, \qquad (6.13)
\end{aligned}$$

onde (6.12) vale por (I'_3), e (6.13) vale por (I'_7). De (6.7) e (6.13) concluímos que X é um multicorte c-mínimo de (G, Q). □

6.5 Análise da complexidade

Dada uma rede (G, Q), lembre que $n := |V_G|, k := |Q|$ e o número de arcos de G é $n - 1$. Suporemos que para a implementação do algoritmo MULTICORTE-DE-ÁRVORE a representação da árvore é feita usando listas de adjacência. Alternativamente a árvore pode ser implementada com um vetor indexado pelos vértices, que guarda para cada vértice seu predecessor mais imediato. É claro que o bloco de linhas 1-2 consome tempo $O(n)$. Para encontrar o caminho de um vértice v a um vértice w em uma árvore divergente, partimos de w e recuamos em direção ao único arco que entra em w. Repetindo esse processo, chegaremos a v em no máximo n passos. Tendo em conta esse argumento, cada execução da linha 4 consome tempo $O(n)$ e portanto o bloco de linhas 3-5 consome

tempo $O(kn)$. Pelo mesmo argumento, cada execução da linha 8 consome tempo $O(n)$, e cada execução do bloco de linhas 9-12 também consome tempo $O(n)$. Portanto, o bloco de linhas 7-12 consome tempo $O(kn)$. Da mesma maneira, a linha 15 consome tempo $O(n)$ e portanto o bloco de linhas 13-16 consome tempo $O(kn)$. A análise nos diz que o tempo de execução do algoritmo é

$$O(kn).$$

Vamos tentar expressar esse tempo em função de n apenas. Note que k pode ser tão grande quanto $\frac{n^2-n}{2}$, mas mostraremos que cada rede com $k \geq n$ pode ser transformada em outra equivalente tal que $k < n$. Suponha que temos uma rede onde $k > n$; então, existem dois índices i, j tais que $t_i = t_j$ e portanto $A_i \subseteq A_j$ ou $A_j \subseteq A_i$. Suponha sem perda de generalidade que $A_i \subseteq A_j$. Então qualquer conjunto de arcos que separa A_i também separa A_j. Podemos portanto remover o par de terminais (s_j, t_j) da rede. Repetir esse processo até que $t_i \neq t_j$ para cada par de índices (i, j) consome tempo $O(n^2)$. Levando em conta esse pré-processamento, o algoritmo consome tempo $O(n^2) + O(kn)$. Como depois desse pré-processamento $k < n$, o algoritmo consome tempo

$$O(n^2).$$

6.6 Implementação

Na implementação do algoritmo fizemos testes com árvores divergentes geradas aleatoriamente. Em cada teste medimos o tempo de execução do algoritmo no eixo y e, no eixo x, o número de vértices da instância. Os resultados parecem sugerir que o algoritmo é $\Theta(n^2)$, o que é consistente com a análise da complexidade.

Para mais detalhes dos resultados da implementação do algoritmo, veja o capítulo 9.

6.7 Outros tipos de árvores

Um digrafo é uma *árvore dirigida* se existe um conjunto de arcos do digrafo tal que, se invertidos, o digrafo resultante é uma árvore divergente. Um digrafo é uma *árvore convergente* se a inversão de todos seus arcos produz uma árvore divergente.

É fácil ver que podemos modificar o algoritmo Multicorte-de-Árvore para aplicá-lo a árvores convergentes. No caso de uma árvore dirigida arbitrária, embora o algoritmo Multicorte-de-Árvore não possa ser aplicado, o problema MCM também pode ser resolvido em tempo polinomial. Isso ocorreo porque a matriz de restrições do *PL1* (veja seção 4.1) de uma árvore dirigida qualquer é totalmente unimodular, propriedade que explicaremos em seguida. Uma matriz é *totalmente unimodular* se qualquer submatriz quadrada dela tem determinante -1, 0 ou 1. Essa propriedade é válida para a matriz de restrições de uma árvore dirigida qualquer porque ela é do tipo "network matrix" [Sch03, p.213]. Como a matriz de restrições de uma árvore dirigida qualquer é totalmente

unimodular, as soluções do programa linear (4.1-4.3) são inteiras. Podemos então resolver o problema usando algum algoritmo polinomial de programação linear como o algoritmo dos elipsóides [Sch03, p.68].

6.8 Caso particular: caminhos com custos unitários

A presente seção trata do Problema do Multicorte Dirigido Mínimo (MM) em caminhos. É claro que todo caminho é uma árvore divergente, portanto podemos aplicar o algoritmo MULTICORTE-DE-ÁRVORE a uma rede (G, c, Q) tal que G é um caminho. Porém, quando G é um caminho e $c_e = 1$ para todo e em G, o problema apresenta uma propriedade minimax interessante em relação a outro problema conhecido, o problema dos intervalos disjuntos [CLRS01, p.371] [KT05, p.116].

Seja (G, Q) uma rede tal que G é um caminho e $Q = \{s_1t_1, s_2t_2, \ldots, s_kt_k\}$. Para cada $i \in \{1, 2, \ldots, k\}$, denote por A_i o conjunto de arcos do único caminho de s_i a t_i em G. Suporemos que existe tal caminho para todo i, caso contrário podemos eliminar de Q os pares de terminais que não têm essa propriedade. Um *conjunto de intervalos disjuntos* de (G, Q) é um subconjunto D de $\{A_1, A_2, \ldots, A_k\}$ tal que $A_i \cap A_j = \emptyset$ para cada $A_i, A_j \in D$.

> **Problema dos Intervalos Disjuntos**: Dada uma rede (G, Q), encontrar um conjunto de intervalos disjuntos de cardinalidade máxima.

A seguinte inequação deriva-se diretamente do lema 4.1. Seja (G, Q) uma rede tal que G é um caminho. Se X é um multicorte de (G, Q) e D um conjunto de intervalos disjuntos de (G, Q), então

$$|D| \leq |X|. \tag{6.14}$$

É bem conhecido um algoritmo guloso para o Problema dos Intervalos Disjuntos. A ideia do algoritmo é a seguinte. Ajuste a notação de modo que $d(s_1) \leq d(s_2) \leq \cdots \leq d(s_k)$, onde $d(s_i) := dist(r, s_i, G)$ e r é o primeiro vértice de G. O algoritmo começa com $D = \{A_k\}$ e $i = k - 1$, diminuindo em cada iteração o valor de i em 1. Em cada iteração, o algoritmo verifica se A_i é disjunto de todos os elementos em D. Se isso é verdade, então adiciona A_i a D. No final do algoritmo, D é um conjunto de intervalos disjuntos de cardinalidade máxima.

Considere o conjunto $X = \{a_i : A_i \in D\}$, sendo a_i o primeiro arco de A_i. Mostraremos que X é um multicorte de (G, Q). Suponha por contradição que existe um índice i tal que $X \cap A_i = \emptyset$. É claro que $A_i \notin D$. Logo, A_i não é disjunto de todos os elementos em D que foram processados antes que ele. Seja A_j um elemento em D que foi processado antes que A_i (ou seja, $j > i$) tal que $A_i \cap A_j \neq \emptyset$. Como j foi processado antes de i, temos $d(s_i) \leq d(s_j)$ e portanto $a_j \in A_i$ e $X \cap A_i \neq \emptyset$. Contradição. Concluímos que X é um multicorte de (G, Q). É claro também que $|X| = |D|$. Logo, por (6.14), X é um

multicorte mínimo de (G, Q). Portanto, para encontrar um multicorte mínimo de (G, Q) basta resolver o Problema dos Intervalos Disjuntos.

Capítulo 7

Algoritmo de Kortsarts, Kortsarz e Nutov

Neste capítulo apresentamos o algoritmo MULTICORTE-KKN concebido por Kortsarts *et al.* [KKN05], que resolve de maneira aproximada o Problema do Multicorte Dirigido de Mínimo (MM) fazendo uso de técnicas puramente combinatórias. Esse algoritmo tem como fator de aproximação $O((n \lg n)^{2/3})$.

7.1 Algoritmo Multicorte-KKN

Dizemos que um par ordenado de vértices distintos (u, v) de um digrafo é *ligado* se existe algum caminho de u a v em G. O algoritmo MULTICORTE-KKN recebe uma rede (G, Q) tal que $Q \neq \emptyset$ e todo par de terminais em Q é ligado. Devolve um multicorte X de (G, Q) tal que $|X| \leq 13(n \lg n)^{2/3} \cdot \mu(G, Q)$, onde n é o número de vértices de G e $\mu(G, Q)$ é a cardinalidade de um multicorte mínimo de (G, Q).

Algoritmo MULTICORTE-KKN(G, Q)

```
1  se n < 4 então
2      devolva E_G
3  X ← E_G
4  para l ← ⌈8(lg n)^{2/3}⌉ até ⌈8(n lg n)^{2/3}⌉ faça
5      M ← MULTICORTE-TODOS(G, Q, l)
6      se |M| < |X| então
7          X ← M
8  devolva X
```

O algoritmo MULTICORTE-KKN usa como sub-rotina na linha 5 o algoritmo MULTICORTE-TODOS que, com entrada (G, Q, l), produz um multicorte M de (G, Q) tal que

$$|M| \leq l \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{l^2}. \tag{7.1}$$

7.2 Análise do algoritmo Multicorte-KKN

Teorema 7.1 O algoritmo MULTICORTE-KKN devolve um multicorte X de (G, Q) tal que $|X| \leq 13(n \lg n)^{2/3} \cdot \mu(G, Q)$.

Prova: Se $n < 4$ então

$$\begin{aligned} |E_G| &\leq n^2 \\ &< 13(n \lg n)^{2/3} \\ &\leq 13(n \lg n)^{2/3} \cdot \mu(G, Q), \end{aligned}$$

onde a última desigualdade vale porque $\mu(G, Q) \geq 1$, já que $Q \neq \emptyset$ e todo par de terminais em Q é ligado.

Agora analisaremos o caso $n \geq 4$. Seja $\widehat{l} = \lceil 8 \cdot (n \lg n)^{2/3} / \mu(G, Q)^{1/3} \rceil$. Seja $M_{\widehat{l}}$ o valor de M depois da execução da linha 5 do algoritmo quando $l = \widehat{l}$. Note que $\lceil 8 \cdot (\lg n)^{2/3} \rceil \leq \widehat{l} \leq \lceil 8 \cdot (n \lg n)^{2/3} \rceil$, já que $1 \leq \mu(G, Q) \leq n^2$. Então, imediatamente antes da execução da linha 8,

$$\begin{aligned} |X| &\leq |M_{\widehat{l}}| \\ &\leq \widehat{l} \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{\widehat{l}^2} \\ &= \left\lceil \frac{8 \cdot (n \lg n)^{2/3}}{\mu(G, Q)^{1/3}} \right\rceil \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{\lceil 8 \cdot (n \lg n)^{2/3} / \mu(G, Q)^{1/3} \rceil^2} \\ &\leq \left(\frac{8 \cdot (n \lg n)^{2/3}}{\mu(G, Q)^{1/3}} + 1 \right) \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{(8 \cdot (n \lg n)^{2/3} / \mu(G, Q)^{1/3})^2} \\ &= 8 \cdot (n \lg n)^{2/3} \cdot \mu(G, Q)^{2/3} + \mu(G, Q) + \frac{200}{64} \cdot (n \lg n)^{2/3} \cdot \mu(G, Q)^{2/3} \\ &\leq (8 + 1 + 200/64) \cdot (n \lg n)^{2/3} \cdot \mu(G, Q) \\ &\leq 13 \cdot (n \lg n)^{2/3} \cdot \mu(G, Q), \end{aligned}$$

onde a primeira desigualdade vale pelas linhas 6-7 do algoritmo e a segunda desigualdade vale por (7.1). □

Em seguida detalharemos o algoritmo MULTICORTE-TODOS.

7.3 Algoritmo Multicorte-Todos

O algoritmo MULTICORTE-TODOS recebe uma rede (G, Q) com $n \geq 4$ vértices e um inteiro $l \geq 1$ e devolve um multicorte X de (G, Q) tal que $|X| \leq l \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{l^2}$.

Algoritmo MULTICORTE-TODOS(G, Q, l)

```
 9  se l < 11 lg n então
10      devolva E_G
11  Y ← ∅
12  para todo st ∈ Q faça
13      enquanto dist(s, t, G − Y) < l faça
14          P ← CAMINHO(s, t, G − Y)
15          Y ← Y ∪ E_P
16  p ← ⌊(l − 1)/(4 lg n)⌋
17  G' ← G − Y
18  G'' ← G'
19  X ← ∅
20  para todo st ∈ Q faça
21      se dist(s, t, G'') < ∞ então
22          C ← CORTESIMPLES(G'', s, t, p)
23          X ← X ∪ C
24          G'' ← G' − X
25  devolva X ∪ Y
```

O algoritmo MULTICORTE-TODOS usa como sub-rotinas os algoritmos CAMINHO (linha 14) e CORTESIMPLES (linha 22). O algoritmo CAMINHO recebe dois vértices s, t de um digrafo G e devolve um caminho de s a t em G. Para entender o que faz o algoritmo CORTESIMPLES precisamos antes de uma definição importante. Dado um digrafo G,

$$R(G) = \{(u, v) : u \neq v, \mathit{dist}(u, v, G) < \infty\},$$

isso é, o conjunto de pares de vértices ligados em G. O algoritmo CORTESIMPLES, com entrada (G'', s, t, p), devolve um corte C que separa s de t em G'' tal que

$$|C| \leq \frac{4}{p^2} \cdot (|R(G'')| - |R(G'' - C)|). \tag{7.2}$$

7.4 Análise do algoritmo Multicorte-Todos

Levando em conta (7.2), estamos prontos para provar o seguinte lema:

Lema 7.1 O algoritmo MULTICORTE-TODOS devolve um multicorte X de (G, Q) tal que $|X| \leq l \cdot \mu(G, Q) + \dfrac{200(n \lg n)^2}{l^2}$.

Prova: Analisaremos primeiro o caso $l < 11 \lg n$. Nesse caso,

$$\begin{aligned} |E_G| &\leq n^2 \\ &= \frac{(ln)^2}{l^2} \\ &< \frac{(11n \lg n)^2}{l^2} \qquad (7.3) \\ &< \frac{200(n \lg n)^2}{l^2} \\ &\leq l \cdot \mu(G, Q) + \frac{200(n \lg n)^2}{l^2}, \end{aligned}$$

onde (7.3) vale pois $n \geq 2$.

Analisaremos agora o caso $l \geq 11 \lg n$. Seja r o número de caminhos encontrados no processo iterativo das linhas 12 a 15 do algoritmo. No início da execução da linha 16, $|Y| \leq l \cdot r$, pois todos os caminhos encontrados têm comprimento menor que l. Além disso, $r \leq \mu(G, Q)$, já que, como os caminhos são disjuntos nos arcos, um multicorte mínimo precisa conter pelo menos um arco de cada um dos r caminhos. Segue que, no início da execução da linha 16, $|Y| \leq l \cdot \mu(G, Q)$.

Agora provaremos que, no início da execução da linha 25, $|X| \leq \frac{200(n \lg n)^2}{l^2}$. Considere a seguinte invariante no início de cada iteração do processo iterativo das linhas 20-24:

$$|X| \leq \frac{4}{p^2}(|R(G')| - |R(G'')|).$$

Na primeira iteração, $|X| = 0$ e $G' = G''$, e portanto a invariante é válida. Suponha agora que a invariante é válida no início de uma iteração qualquer. Vamos mostrar que vale no início da iteração seguinte. Se $dist(s, t, G'') = \infty$ então os valores das variáveis não mudam na iteração atual e a invariante é válida no início da iteração seguinte. Suponha agora que $dist(s, t, G'') < \infty$. No final da iteração o lado esquerdo da desigualdade incrementa-se em $|C|$. Por (7.2), $|C| \leq \frac{4}{p^2}(|R(G'')| - |R(G'' - C)|)$, então o lado esquerdo da desigualdade incrementa-se no máximo em $\frac{4}{p^2}(|R(G'')| - |R(G'' - C)|)$. No final da iteração, G'' diminui seu valor em exatamente C e portanto o lado direito da desigualdade incrementa-se exatamente em $\frac{4}{p^2}(|R(G'')| - |R(G'' - C)|)$. Portanto, a invariante é válida no início da iteração seguinte.

No final do algoritmo,

$$\begin{aligned} |X| &\leq \frac{4}{p^2}(|R(G') - |R(G' - X)|) \\ &\leq \frac{4}{p^2}|R(G')| \\ &\leq \frac{4n^2}{p^2}. \end{aligned} \tag{7.4}$$

Também,

$$\begin{aligned} l &\geq 11 \lg n \\ &= \frac{28 \lg n}{3} + \frac{5 \lg n}{3} \\ &\geq \frac{28 \lg n}{3} + \frac{7}{3} \qquad (7.5) \\ &= \frac{7(4 \lg n + 1)}{3}, \qquad (7.6) \end{aligned}$$

onde (7.5) vale pois $n \geq 4$, e portanto

$$\begin{aligned} p &= \lfloor \frac{l-1}{4 \lg n} \rfloor \\ &\geq \frac{l-1}{4 \lg n} - 1 \\ &= \frac{l - (4 \lg n + 1)}{4 \lg n} \\ &\geq \frac{l - 3l/7}{4 \lg n} \qquad (7.7) \\ &= \frac{l}{7 \lg n}, \qquad (7.8) \end{aligned}$$

onde (7.7) vale por (7.6). Então, de (7.4) e (7.8),

$$\begin{aligned} |X| &\leq \frac{4n^2}{p^2} \\ &\leq \frac{4n^2}{(l/7 \lg n)^2} \\ &= \frac{4n^2(7 \lg n)^2}{l^2} \\ &\leq \frac{200(n \lg n)^2}{l^2}. \end{aligned} \tag{7.9}$$

Com isso, concluímos a prova do lema. □

Em seguida detalharemos o algoritmo CorteSimples.

7.5 Algoritmo CorteSimples

O algoritmo CorteSimples recebe um digrafo G com n vértices, dois vértices s, t, e um inteiro $p \geq 1$ tais que $4p \lg n + 1 \leq dist(s, t, G) < \infty$. Devolve um corte C que separa s de t tal que $|C| \leq \frac{4}{p^2}(|R(G)| - |R(G - C)|)$.

Algoritmo CorteSimples(G, s, t, p)
26 **para** $i \leftarrow 0$ **até** $\lceil 2 \lg n \rceil - 1$ **faça**
27 $\quad S^*_{ip} \leftarrow \{v : dist(s, v, G) \leq ip\}$
28 $\quad T^*_{ip} \leftarrow \{v : dist(v, t, G) \leq ip\}$
29 $C_{0p} \leftarrow$ CorteMínimo(S^*_0, T^*_0, G)
30 $j \leftarrow 0$
31 **para** $j \leftarrow 0$ **até** $\lceil 2 \lg n \rceil$ **faça**
32 $\quad j \leftarrow j + 1$
33 $\quad C_{jp} \leftarrow$ CorteMínimo(S^*_{jp}, T^*_{jp}, G)
34 $\quad$ **se** $|C_{jp}| \leq 2|C_{(j-1)p}|$ **então**
35 $\quad\quad$ **devolva** C_{jp}

O algoritmo CorteSimples faz uso do algoritmo CorteMínimo , que recebe um digrafo G e dois subconjuntos S e T de V_G tais que $S \cap T = \emptyset$,e devolve um corte de cardinalidade mínima que separa S de T.

7.6 Análise do algoritmo CorteSimples

Para provar a correção de CorteSimples precisamos de um estudo prévio. Dados dois conjuntos $U, W \subseteq V_G$, defina

$$R_G(U, W) = (U \times W) \cap R(G),$$

isso é, o número de pares ligados em $U \times W$.

Dado um digrafo G, um par de terminais st em G, e um inteiro não negativo i, defina

$$S_i := \{v \in V_G : dist(s, v, G) = i\}$$

e

$$T_i := \{v \in V_G : dist(v, t, G) = i\},$$

e sejam $S^*_i = S_0 \cup S_1 \cup \ldots \cup S_i$ e $T^*_i = T_0 \cup T_1 \cup \ldots \cup T_i$.

Lema 7.2 Seja G um digrafo, seja st um par de vértices ligados em G, sejam $i, j \geq 0$ dois inteiros tais que $S^*_{i+1} \cap T^*_{j+1} = \emptyset$, e seja $\mathcal{P}$ uma coleção de caminhos disjuntos nos

arcos de S_i^* a T_j^*. Então

$$|R_G(S_i, T_{j+1})| + |R_G(S_{i+1}, T_j)| \geq |\mathcal{P}|.$$

Prova: Precisamos de algumas definições prévias. Um (S,T)-*emparelhamento* é um subconjunto M de $S_{i+1} \times T_{j+1}$ tal que, para cada dois elementos distintos (u,w) e (u',w') de M, temos que $u \neq u'$ e $w \neq w'$. Para cada $(u,w) \in M$, diremos que M emparelha u e w (e emparelha w e u). Também diremos que M incide em u (e incide em w).

Para cada caminho P em $\mathcal{P}$, o *vértice superior* de P é o primeiro vértice de P que está em S_{i+1}. O *vértice inferior* de P é o último vértice de P que está em T_{j+1}.

Um (S,T)-emparelhamento é *bom* se (1) para cada P em $\mathcal{P}$, ou M incide no vértice superior de P ou M incide no vértice inferior de P; (2) para cada $(u,w) \in M$, algum caminho em $\mathcal{P}$ contém u e w.

Provaremos agora que existe pelo menos um (S,T)-emparelhamento bom. Comece com $M = \emptyset$ e seja P um elemento de $\mathcal{P}$. Seja a o vértice superior e b o vértice inferior de P. Adicione o par (a,b) a M. Agora, seja $\mathcal{P}^a$ o conjunto de elementos em $\mathcal{P}$ cujo vértice superior é a e seja $\mathcal{P}_b$ o conjunto de elementos em $\mathcal{P}$ cujo vértice inferior é b. Repita o procedimento com $\mathcal{P} \setminus (\mathcal{P}^a \cup \mathcal{P}_b)$ no lugar de $\mathcal{P}$. Como resultado M é um (S,T)-emparelhamento bom.

Para provar o lema, basta encontrar uma injeção de $\mathcal{P}$ em $R_G(S_i, T_{j+1}) \cup R_G(S_{i+1}, T_j)$. Para cada elemento P de $\mathcal{P}$, seja $a(P)$ o vértice superior de P e seja $b(P)$ o vértice inferior de P. Seja $a'(P)$ o vértice em P que precede $a(P)$ e seja $b'(P)$ o vértice em P que sucede $b(P)$. Seja M um (S,T)-emparelhamento bom. Pela propriedade (1) de um emparelhamento bom, ou $a(P)$ é emparelhado por M com algum $w \in T_{j+1}$ ou $b(P)$ é emparelhado por M com algum $u \in S_{i+1}$. No primeiro caso seja

$$f(P) := (a'(P), w),$$

no segundo caso

$$f(P) := (u, b'(P)),$$

o que define nossa função f. Em virtude da propriedade (2) de um emparelhamento bom, para cada P em $\mathcal{P}$, o par $f(P)$ é ligado.

Finalmente, mostraremos que a função f é uma injeção. Seja (P_1, P_2) um par de elementos de $\mathcal{P}$ e suponha $f(P_1) = f(P_2)$. Suponha sem perda de generalidade que $f(P_1) = f(P_2) = (a', w)$. Suponha por contradição que $P_1 \neq P_2$, então $(a(P_1), w), (a(P_2), w) \in M$ e $a(P_1) \neq a(P_2)$, o que contradiz a definição de emparelhamento. □

Lema 7.3 Seja G um digrafo, seja st um par de vértices ligados em G, seja p um inteiro

tal que $p \geq 1$, e seja uma coleção de r caminhos disjuntos nos arcos de $S^*_{(j-1)p}$ a $T^*_{(j-1)p}$. Então $|R_G(S^*_{jp}, T^*_{jp})| \geq rp^2/2$.

Prova: Note que

$$\begin{aligned} |R_G(S^*_{jp}, T^*_{jp})| &\geq |R_G(S^*_{jp} \setminus S^*_{(j-1)p-1}, T^*_{jp} \setminus T^*_{(j-1)p-1})| \\ &\geq \frac{1}{2} \cdot \sum_{i=(j-1)p}^{jp-1} \Big(\sum_{l=(j-1)p}^{jp-1} \big(|R_G(S_i, T_{l+1})| + |R_G(S_{i+1}, T_l)|\big)\Big) \\ &\geq rp^2/2, \end{aligned} \tag{7.10}$$

onde (7.10) vale pelo lema 7.2. Com isso, concluímos a prova do lema. □

Agora estamos prontos para provar o lema central do algoritmo CorteSimples.

Lema 7.4 O algoritmo CorteSimples devolve um corte C que separa s de t tal que $|C| \leq \frac{4}{p^2}(|R(G)| - |R(G-C)|)$.

Prova: Precisamos provar que (1) o processo iterativo das linhas 31-35 é finito e (2) $|R(G)| - |R(G-C)| \geq |C|p^2/4$.

Prova de (1): Precisamos provar que a linha 35 do algoritmo é executada. Suponha por contradição que a linha 35 do algoritmo não é executada. Então $|C_{jp}| > 2|C_{(j-1)p}|$ para $j = 1, \ldots, \lceil 2 \lg n \rceil - 1$ (note que $j \leq 2 \lg n$ já que $dist(s,t,G) \geq 4p \lg n + 1$). Como $|C_{0p}| \geq 1$, já que $dist(s,t,G) < \infty$, então para cada j, $|C_{jp}| \geq 2^{j+1} - 1$. Portanto,

$$|C_{(\lceil 2 \lg n \rceil - 1)p}| \geq 2^{\lceil 2 \lg n \rceil} - 1 \geq 2^{2 \lg n} - 1 = 2^{\lg n^2} - 1 = n^2 - 1.$$

Contradição, pois qualquer corte tem cardinalidade no máximo $|E_G| \leq n(n-1)$.

Prova de (2): Seja C_{jp} o corte devolvido pelo algoritmo. Note que $|C_{(j-1)p}| \geq |C_{jp}|/2$ pela condição da linha 34. Como $C_{(j-1)p}$ é um corte mínimo que separa $S^*_{(j-1)p}$ de $T^*_{(j-1)p}$, pelo teorema de Ford e Fulkerson, existem $|C_{(j-1)p}| \geq |C_{jp}|/2$ caminhos disjuntos nos arcos de $S^*_{(j-1)p}$ a $T^*_{(j-1)p}$. Aplicando o lema 7.3, temos que $R_G(S^*_{jp}, T^*_{jp}) \geq |C_{(j-1)p}|p^2/2 \geq |C_{jp}|p^2/4$. Note agora que C_{jp} separa todo (u,v) tal que $u \in S^*_{jp}$ e $v \in T^*_{jp}$ e portanto,

$$|R(G)| - |R(G - C_{jp})| \geq |R_G(S^*_{jp}, T^*_{jp})| \geq |C_j|p^2/4.$$

Com isso, concluímos a prova do lema 7.4. □

7.7 Uma análise grosseira da complexidade dos algoritmos

Dada uma rede (G, Q), lembre que $n := |V_G|$, $m := |E_G|$ e $k := |Q|$. Cada chamada a CorteMínimo, na linha 33 do algoritmo CorteSimples, se implementada com o algoritmo de Edmonds-Karp [EK72] consome tempo $O(nm^2)$. O algoritmo CorteSimples

faz $O(\lg n)$ invocações ao algoritmo CorteMínimo Então o algoritmo CorteSimples consome tempo

$$O(nm^2 \lg n).$$

O algoritmo Multicorte-Todos faz $O(k)$ invocações ao algoritmo CorteSimples. Portanto, o algoritmo Multicorte-Todos consome tempo

$$O(knm^2 \lg n).$$

O algoritmo Multicorte-KKN faz $O(n^2)$ invocações ao algoritmo Multicorte-Todos. Portanto, o tempo de execução do algoritmo Multicorte-KKN é

$$O(kn^3m^2 \lg n).$$

7.8 Uma análise mais fina da complexidade do algoritmo CorteSimples

Seja $C_p, C_{2p}, \ldots, C_{lp}$ a sequência de valores da variável C_{jp} na linha 33 do algoritmo CorteSimples. Como C_{lp} separa S^*_{lp} de T^*_{lp}, e como $S^*_{jp} \subseteq S^*_{lp}$ e $T^*_{jp} \subseteq T^*_{lp}$ para cada $j \leq l$, então C_{lp} separa S^*_{jp} de T^*_{jp} para cada $j \leq l$. Como C_{jp} é um corte de cardinalidade mínima que separa S^*_{jp} de T^*_{jp} então, para cada $j \leq l$,

$$|C_{jp}| \leq |C_{lp}|.$$

Cada chamada a CorteMínimo, na linha 33 do algoritmo CorteSimples, se implementada com o algoritmo de Ford e Fulkerson [CLRS01, p.658], consome tempo $O(m \cdot |C_{jp}|)$, já que os custos são inteiros não negativos. Como

$$\sum_{j=0}^{l} |C_{jp}| \leq (l+1)|C_{lp}| \leq (\lceil 2 \lg n \rceil + 1)|C_{lp}|,$$

então o algoritmo CorteSimples consome tempo

$$O(m \lg n \cdot |C_{lp}|). \tag{7.11}$$

Note que C_{lp} é o corte devolvido pelo algoritmo CorteSimples na linha 35. Portanto, esse tempo de execução depende da saída dada pelo algoritmo.

7.9 Uma análise mais fina da complexidade do algoritmo Multicorte-Todos

A linha 14 do algoritmo Multicorte-Todos pode ser implementada com uma busca em largura em tempo $O(n + m)$. Portanto o bloco de linhas 12-15 do algoritmo

MULTICORTE-TODOS consome tempo

$$O(k(n+m)). \tag{7.12}$$

Seja $C_1, C_2, \ldots, C_q$ a sequência de valores da variável C imediatamente depois da execução da linha 22 do algoritmo MULTICORTE-TODOS. No final do algoritmo MULTICORTE-TODOS,

$$\sum_{i=1}^{q} |C_i| = |X| \leq \frac{200n^2 \lg^2 n}{l^2},$$

onde a última desigualdade vale por (7.9). Por (7.11), cada chamada ao algoritmo CORTESIMPLES consome tempo $O(m \lg n \cdot |C_i|)$. Portanto, o bloco de linhas 20-24 do algoritmo MULTICORTE-TODOS consome tempo $O(\frac{mn^2 \lg^3 n}{l^2})$. De (7.12), o algoritmo MULTICORTE-TODOS consome tempo

$$O(k(n+m) + \frac{mn^2 \lg^3 n}{l^2}). \tag{7.13}$$

É importante observar que a complexidade desse algoritmo depende de mais um parâmetro, l, o qual não é constante.

7.10 Uma análise mais fina da complexidade do algoritmo Multicorte-KKN

O algoritmo MULTICORTE-KKN faz $O(n^2)$ invocações ao algoritmo MULTICORTE-TODOS. De (7.13), cada uma dessas invocações consome tempo $O(k(n+m) + \frac{mn^2 \lg^3 n}{l^2})$. Note que

$$\sum_{l=1}^{n^2} \frac{1}{l^2} \leq \sum_{l=1}^{\infty} \frac{1}{l^2} = \pi^2/6 = O(1).$$

Portanto, o tempo de execução do algoritmo MULTICORTE-KKN é $O(k(n+m) + mn^2 \lg^3 n)$. Se o digrafo é conexo então $m \geq n-1$. Como $k = O(n^2)$, concluímos que o algoritmo MULTICORTE-TODOS consome tempo

$$O(mn^2 \lg^3 n).$$

7.11 Implementação

O algoritmo não foi implementado e portanto não observamos seu desempenho na prática.

7.12 Caso geral: custos arbitrários nos arcos

Kortsarts *et al.* [KKN05] também propuseram um algoritmo que resolve o Problema do Multicorte Dirigido de Custo Mínimo (MCM) de maneira aproximada fazendo uso de técnicas puramente combinatórias. Esse algoritmo tem como fator de aproximação $O(n^{2/3})$ e complexidade $O(nm^2 \lg^3 n)$. Embora esse algoritmo tenha melhor fator de aproximação que o algoritmo MULTICORTE-KKN, se o digrafo é conexo então MULTICORTE-KKN tem um melhor tempo de execução e vale a pena implementá-lo.

Capítulo 8

Algoritmo de Gupta

Neste capítulo apresentamos o algoritmo concebido por Gupta [Gup03] que resolve de maneira aproximada o Problema do Multicorte Dirigido de Custo Mínimo (MCM). Ele é uma $O(\sqrt{n})$-aproximação para o problema MCM, onde n é o número de vértices do digrafo. Embora esse fator seja um dos melhores para o problema MCM, é um fator muito ruim na prática. O algoritmo é basicamente o mesmo proposto por Cheriyan *et al.* [CKR05], que é uma $O(\sqrt{n \log n})$-aproximação. Gupta faz uma melhor análise desse algoritmo, obtendo um fator de aproximação $O(\sqrt{n})$.

8.1 O algoritmo

O algoritmo GUPTA-C-K-R recebe uma rede (G, c, Q) e devolve um multicorte $X \cup Y$ de (G, Q) tal que $c(X \cup Y) \leq 8\sqrt{n} \cdot \mu^*(G, c, Q)$, onde $\mu^*(G, c, Q)$ é o custo de um multicorte fracionário c-mínimo de (G, Q), e n é o número de vértices de G. É claro que se $\mu(G, c, Q)$ é o custo de um multicorte c-mínimo de (G, Q) então $\mu^*(G, c, Q) \leq \mu(G, c, Q)$. Portanto o algoritmo é uma $8\sqrt{n}$-aproximação.

Algoritmo GUPTA-C-K-R(G, c, Q)

```
1  x ← ProgLin(G, c, Q)
2  Y ← ∅
3  para todo e ∈ E_G faça
4      se x_e ≥ 1/(4√n) então
5          Y ← Y ∪ {e}
6  G' ← G − Y
```

```
7  X ← ∅
8  para todo st ∈ Q faça
9      G'' ← G' − X
10     U ← {v ∈ V_{G''} : existe caminho em G'' de s a v}
11     W ← {v ∈ V_{G''} : existe caminho em G'' de v a t}
12     se U ∩ W ≠ ∅ então
13         H ← G''[U ∩ W]
14         S ← {v ∈ V_H : dist_x(s, v, H) ≤ 1/4}
15         T ← {v ∈ V_H : dist_x(s, v, H) ≥ 3/4}
16         Z ← CorteCustoMínimo(H, c, S, T)
17         X ← X ∪ ∂_H(Z)
18 devolva X ∪ Y
```

O algoritmo ProgLin, na linha 1, recebe uma rede (G, c, Q) e devolve um multicorte fracionário c-mínimo de (G, Q). O algoritmo CorteCustoMínimo, na linha 16, recebe um digrafo H com custos c nos arcos, dois subconjuntos S e T de V_H tais que $S \cap T = \emptyset$, e devolve um conjunto Z tal que $S \subseteq Z \subseteq V_H \setminus T$ e $c(\partial_H(Z))$ é mínimo.

O algoritmo Gupta-C-K-R faz umas pequenas modificações ao algoritmo apresentado no artigo de Gupta. Na linha 4, no artigo encontramos $\frac{1}{\sqrt{n}}$ no lugar de $\frac{1}{4\sqrt{n}}$. Na linha 14 encontramos $1/3$ no lugar de $1/4$. Na linha 15 encontramos $2/3$ no lugar de $3/4$. Com essas modificações conseguimos uma melhor constante no fator de aproximação.

8.2 Análise do algoritmo

O algoritmo Gupta-C-K-R funciona em duas etapas. Primeiro encontra um conjunto de arcos Y, obtendo uma aproximação preliminar (linhas 3-5). Depois encontra um conjunto de arcos X mediante uma técnica sofisticada (linhas 8-17). Finalmente devolve a união de X e Y. Provaremos que tanto $c(X)$ quanto $c(Y)$ estão na ordem $O(\sqrt{n} \cdot \mu^*(G, c, Q))$. O segundo resultado é fácil de provar, o primeiro resultado é o coração do algoritmo.

Lema 8.1 Imediatamente antes da execução da linha 6 do algoritmo, $c(Y) \leq 4\sqrt{n} \cdot \mu^*(G, c, Q)$.

Prova: Note que

$$c(Y) = \sum_{e \in Y} c_e \leq \sum_{e \in Y} c_e(x_e \cdot 4\sqrt{n}) \leq 4\sqrt{n} \cdot \sum_{e \in E_G} c_e x_e = 4\sqrt{n} \cdot \mu^*(G, c, Q),$$

onde a primeira desigualdade vale pois $x_e \geq \frac{1}{4\sqrt{n}}$ para todo $e \in Y$. □

Para provar que $c(X)$ é $O(\sqrt{n} \cdot \mu^*(G, c, Q))$ começaremos por encontrar um limitante superior para $c(\partial_H(Z))$ imediatamente depois da cada execução da linha 16.

Lema 8.2 Imediatamente depois de cada execução da linha 16 do algoritmo, $c(\partial_H(Z)) \leq 2\sum_{e\in E_H} c_e x_e$.

Prova: Vamos primeiro dividir o digrafo em camadas. Sejam $\{v_0, v_1, \ldots, v_q\}$ os vértices de H enumerados de modo que $dist_x(s, v_0, H) \leq dist_x(s, v_1, H) \leq \ldots \leq dist_x(s, v_q, H)$. Ponha $r_i := dist_x(s, v_i, H)$ para todo i. Seja $R_i = \{v_0, v_1, \ldots, v_i\}$. Para cada arco v_jv_l ponha $c_{jl} := c_{v_jv_l}$ e $x_{jl} := x_{v_jv_l}$. Ponha $\partial := \partial_H$. Então

$$\begin{aligned}
\sum_{i=0}^{q-1} c(\partial(R_i))(r_{i+1} - r_i) &= \sum_{i=0}^{q-1} \Big(\sum_{jl\in J(i)} c_{jl}(r_{i+1} - r_i)\Big) \\
&= \sum_{jl\in J} \Big(c_{jl} \cdot \sum_{i=j}^{l-1} (r_{i+1} - r_i)\Big) \\
&= \sum_{jl\in J} c_{jl}(r_l - r_j) \\
&\leq \sum_{jl\in J} c_{jl}x_{jl} \qquad (8.1) \\
&\leq \sum_{e\in E_H} c_e x_e, \qquad (8.2)
\end{aligned}$$

sendo J o conjunto de todos os pares jl tais que $0 \leq j < l \leq q$ e $v_jv_l \in E_H$, e $J(i) = \{jl \in J : j \leq i < l\}$. A desigualdade (8.1) vale pois, por desigualdade triangular, $r_l - r_j = dist_x(s, v_l, H) - dist_x(s, v_j, H) \leq x_{uv}$ para todo arco uv de H.

Como x é um multicorte fracionário c-mínimo de (G, Q) e como $H \subseteq G$, então $dist_x(s, t, H) \geq dist_x(s, t, G) = \sum_{e\in E_P} x_e \geq 1$, sendo P um caminho mínimo de s a t em G. Também, $dist_x(s, s, H) = 0$, e portanto podemos definir $g = \max\{i : r_i \leq 1/4\}$ e $h = \min\{i : r_i \geq 3/4\}$. Provaremos que

$$\text{existe } i \text{ tal que } g \leq i < h \text{ e } c(\partial(R_i)) \leq 2 \sum_{e\in E_H} c_e x_e. \qquad (8.3)$$

Suponha por contradição que $c(\partial(R_i)) > 2\sum_{e\in E_H} c_e x_e$ para todo i tal que $g \leq i < h$.

Então

$$\begin{aligned}
\sum_{i=g}^{h-1} c(\partial(R_i))(r_{i+1} - r_i) &> \sum_{i=g}^{h-1} \big((2 \sum_{e \in E_H} c_e x_e)(r_{i+1} - r_i)\big) \\
&= 2 \sum_{e \in E_H} \big(c_e x_e \cdot \sum_{i=g}^{h-1} (r_{i+1} - r_i)\big) \\
&= 2 \sum_{e \in E_H} \big(c_e x_e \cdot (r_h - r_g)\big) \\
&\geq 2 \sum_{e \in E_H} \big(c_e x_e \cdot (3/4 - 1/4)\big) \\
&= 2 \sum_{e \in E_H} (c_e x_e \cdot 1/2) \\
&= \sum_{e \in E_H} c_e x_e.
\end{aligned}$$

Esse resultado contradiz (8.2) e portanto prova (8.3).

Para terminar a prova do lema, seja i um índice que faz válida a propriedade (8.3). É claro que S, achado na linha 14 do algoritmo, é igual a R_g e T, achado na linha 15 do algoritmo, é igual a $V_H \setminus R_{h-1}$. Então $S = R_g \subseteq \ldots \subseteq R_i \subseteq \ldots \subseteq R_{h-1} = V_H \setminus T$, e portanto $\partial(R_i)$ separa S de T. Como $\partial(Z)$ é um corte mínimo dentre os que separam S de T,

$$c(\partial(Z)) \leq c(\partial(R_i)) \leq 2 \sum_{e \in E_H} c_e x_e.$$

Com isso, concluímos a prova do lema. □

Uma vez delimitado o custo do cada corte encontrado na linha 16 de cada iteração, e sabendo que o número de iterações do processo iterativo das linhas 8-17 é delimitado por $k := |Q|$, podemos afirmar que $c(X) \leq k \cdot 2\mu^*(G, c, Q)$. Mas esse resultado não nos satisfaz, já que k pode não ser $O(\sqrt{n})$, chegando perto de n^2. Devemos delimitar $c(X)$ de alguma outra maneira. Fixamos nossa atenção em um arco e qualquer e tentamos delimitar o número de vezes que e aparece em algum H.

Lema 8.3 No final do algoritmo, $c(X) \leq 4\sqrt{n} \cdot \mu^*(G, c, Q)$

Prova: Seja $H_1, H_2, \ldots, H_p$ a sequência de valores da variável H imediatamente depois da execução da linha 13. Seja $(s_1, t_1, S_1, T_1, Z_1), \ldots, (s_p, t_p, S_p, T_p, Z_p)$ a correspondente sequência de valores de (s, t, S, T, Z). Para cada i, seja C_i o corte associado a Z_i, isso é,

$C_i = \partial_{H_i}(Z_i)$. Para cada i, seja $E_i := E_{H_i}$ e $V_i := V_{H_i}$. É claro que, no final do algoritmo

$$\begin{aligned} c(X) &= \sum_{i=1}^{p} c(C_i) \\ &\leq 2\sum_{i=1}^{p}\sum_{e\in E_i} c_e x_e \qquad (8.4) \\ &= 2\sum_{e\in E_{G'}} c_e x_e |N(e)|, \end{aligned}$$

onde (8.4) vale pelo lema 8.2, e sendo $N(e) = \{i \in \{1,\ldots,p\} : e \in E_i\}$. Logo,

$$c(X) \leq 2\sum_{e\in E_{G'}} c_e x_e (|L(e)| + |R(e)|),$$

sendo

$$L(uv) = \{j \in N(uv) : u \in Z_j\}$$

e

$$R(uv) = \{j \in N(uv) : u \in V_j \setminus Z_j\}.$$

Resta mostrar que $|L(uv)| \leq \sqrt{n}$ para todo uv em $E_{G'}$ e analogamente $|R(uv)| \leq \sqrt{n}$.

Fixe um arco uv e, para cada j em $L(uv)$, seja P_j um caminho em H_j de u até t_j. A definição de H_j garante que tal caminho existe. Seja $\widehat{P}_j$ o menor segmento terminal desse caminho que começa em Z_j.

Suponha que $\widehat{P}_j$ começa em w. Temos

$$\begin{aligned} x(\widehat{P}_j) &\geq dist_x(s_j, t_j, H_j) - dist_x(s_j, w, H_j) \qquad (8.5) \\ &\geq 1 - dist_x(s_j, w, H_j) \qquad (8.6) \\ &> 1 - 3/4 \qquad (8.7) \\ &= 1/4, \qquad (8.8) \end{aligned}$$

onde (8.5) vale pela desigualdade triangular; (8.6) vale pois, pela definição de ProgLin na linha 1, $dist_x(s_j, t_j, H_j) \geq 1$; e (8.7) vale pois $dist_x(s, w, H_j) < 3/4$, já que $w \in Z_j \subseteq V_j \setminus T_j$.

Por outro lado, por causa do primeiro processo iterativo (linhas 3-5), para todo arco e de G',

$$x_e < \frac{1}{4\sqrt{n}}. \qquad (8.9)$$

Logo, de (8.8) e (8.9), $|E_{\widehat{P}}| > \sqrt{n}$. Seja $W(\widehat{P}) = V_{\widehat{P}} \cap (V_j \setminus Z_j)$. É claro que

$$|W(\widehat{P})| = |E_{\widehat{P}}| > \sqrt{n}. \qquad (8.10)$$

Agora observe que $W(\widehat{P}_i)$ é disjunto de $W(\widehat{P}_j)$ para todo $i < j$ que estejam em $L(uv)$. Eis a prova desse fato: suponha que $W(\widehat{P}_i)$ e $W(\widehat{P}_j)$ têm um vértice r em comum. Considere a relação entre P_j e C_i. Como $u \in Z_i$ e $r \in W(\widehat{P}_j) \subseteq V_i \setminus Z_i$, o caminho P_j tem um vértice em Z_i e outro em $V_i \setminus Z_i$, e portanto algum arco de P_j está em C_i, donde

$$E_j \cap C_i \neq \emptyset.$$

Mas $E_j \subseteq E_G \setminus (Y \cup X_{j-1}) \subseteq E_G \setminus (Y \cup C_1 \cup \ldots \cup C_i)$ e portanto $E_j \cap C_i = \emptyset$. Assim, temos uma contradição. A contradição mostra que, de fato

$$W(\widehat{P}_i) \cap W(\widehat{P}_j) = \emptyset$$

para todo $i < j$. Segue daí e de (8.10) que $L(uv) < \frac{|V_G|}{\sqrt{n}} = \sqrt{n}$. Mediante um argumento similar, $|R(uv)| < \sqrt{n}$. Com isso, concluímos a prova do lema. □

Lema 8.4 No final do algoritmo, X é um multicorte de (G', Q).

Prova: Basta provar a seguinte invariante no início de cada iteração do processo iterativo das linhas 8-17:

(I) X é multicorte de (G', Q'), sendo Q' o conjunto de pares de terminais de Q que já foram escolhidos na linha 8 do algoritmo.

No início da primeira iteração a invariante (I) é válida já que $Q' = \emptyset$. Suponha agora que a invariante é válida no início de uma iteração qualquer das linhas 8-17. Vamos mostrar que a invariante é válida no início da iteração seguinte. Se, na linha 12, $U \cap W = \emptyset$ então X e Q' não mudam de valor na iteração atual e portanto a invariante é valida no início da iteração seguinte. No caso contrário, note que imediatamente depois da execução da linha 13, todo caminho de s a t em $G' - X$ está em H. Esse fato é fácil de deduzir já que cada vértice de cada um desses caminhos está em $U \cap W$. Portanto, imediatamente depois da execução da linha 16, como $\partial_H(Z)$ separa s de t em H, $\partial_H(Z)$ separa s de t em G'. A prova da invariante segue do fato que, na linha 17, X aumenta seu valor em $\partial_H(Z)$ e, no final da iteração, Q' aumenta seu valor em st. □

Teorema 8.1 O algoritmo GUPTA-C-K-R devolve um multicorte $X \cup Y$ de (G, Q) tal que $c(X \cup Y) \leq 8\sqrt{n} \cdot \mu^*(G, c, Q)$,

Prova: Pelo lema 8.4, no final do algoritmo, X é um multicorte de (G', Q). Pela linha 6, no final do algoritmo, $Y = E_G \setminus E_{G'}$. Portanto $X \cup Y$ é multicorte de (G, Q).

Pelos lemas 8.1 e 8.3,

$$\begin{aligned} c(X \cup Y) &\leq c(X) + c(Y) \\ &= 4\sqrt{n} \cdot \mu^*(G, c, Q) + 4\sqrt{n} \cdot \mu^*(G, c, Q) \\ &= 8\sqrt{n} \cdot \mu^*(G, c, Q). \end{aligned}$$

Com isso, concluímos a prova do teorema. □

8.3 Análise da complexidade

Dada uma rede (G, Q), lembre que $n := |V_G|$, $m := |E_G|$ e $k := |Q|$. A linha 1 pode fazer uso do programa linear *PL2* visto na seção 4.2, o qual tem um número polinomial de restrições no tamanho da rede. Portanto o tempo de execução da linha 1 é $P(m, n, k)$, onde P é um polinômio com variáveis m, n e k. O bloco de linhas 3-5 toma tempo $O(m)$. As linhas 10 e 11 podem ser implementadas com uma busca em largura em tempo $O(n + m)$. As linhas 14 e 15 podem ser implementadas com o algoritmo de Dijkstra [Dij59] em tempo $O((n + m) \lg n)$ se implementado com filas de prioridades (heap) [CLRS01, p.595]. Se o digrafo é conexo então $m \geq n-1$, e portanto o algoritmo de Dijsktra consome tempo $O(m \lg n)$. A linha 16 pode ser implementada com o algoritmo de Edmonds-Karp [EK72] em tempo $O(nm^2)$. Portanto o bloco de linhas do processo iterativo 8-17 consome tempo $O(knm^2)$. A análise nos diz que o tempo de execução do algoritmo é

$$O(knm^2) + P(n, m, k).$$

8.4 Implementação

Mencionamos na introdução que o fator de aproximação dado pelo algoritmo é muito alto. Embora isso seja decepcionante na primeira impressão, a pergunta que fica no ar é se o algoritmo é melhor do que parece, ou seja, se o fator de aproximação é, na verdade, menor que $O(\sqrt{n})$. Não temos informação suficiente para responder a essa pergunta. O mais provável é que esse fator seja justo, ou seja, existe uma família de instâncias para as quais o custo devolvido pelo algoritmo chega muito perto do seu limitante superior.

Além dessa consideração teórica, a implementação do algoritmo foi um tanto decepcionante no seguinte sentido: não tivemos dados de testes suficientes que usem a segunda parte do algoritmo. Ou seja, imediatamente antes da execução da linha 6, o conjunto de arcos Y já é um multicorte. Para testar o correto funcionamento do segundo processo iterativo (linhas 8-17), tivemos que eliminar o bloco de linhas do primeiro processo iterativo. A implementação se comportou corretamente, mas é claro que o fator de aproximação não conseguiu ser garantido.

Finalmente, um fato não tão decepcionante foi que nos testes feitos o custo do mul-

ticorte devolvido pelo algoritmo não dista muito do custo do multicorte c-mínimo, se assemelhando muito aos resultados da k-aproximação apresentada na seção 3.1.

Para mais detalhes dos resultados da implementação do algoritmo, veja o capítulo 9.

8.5 Melhora

Agarwal *et al.* [AAC07] apresentaram uma $O(n^{11/23} \lg n)$-aproximação para o problema MCM. O algoritmo apresentado por Agarwal *et al.* está baseado no algoritmo Gupta-C-K-R. Esse algoritmo faz enfase no conceito de "carga". O lema 8.2 diz que, imediatamente depois de cada execução da linha 16 do algoritmo Gupta-C-K-R, $c(\partial_H(Z)) \leq 2 \sum_{e \in E_H} c_e x_e$. Podemos dizer que o custo do corte $\partial_H(Z)$ está sendo "carregado" pelos arcos de H, sendo que cada arco em H recebe uma carga que é no máximo 2 vezes sua contribuição na resolução do programa linear, ou seja no máximo $2c_e x_e$. O algoritmo apresentado por Agarwal *et al.* modifica o algoritmo de Gupta tal que cada arco em H recebe uma carga que é no máximo $O(n^{1/23})$ vezes sua contribuição na resolução do programa linear, ou seja no máximo $O(n^{1/23})c_e x_e$.

No algoritmo Gupta-C-K-R, cada arco e é carregado no máximo $L(e)+R(e) \leq 2\sqrt{n}$ vezes, como foi mostrado na última parte da prova do lema 8.3. Agarwal *et al.* modificam o algoritmo Gupta-C-K-R de modo que cada arco é carregado r vezes. Se $r < n^{10/23}$ então, com uma analise similar à apresentada no lema 8.3, o custo do multicorte devolvido pelo algoritmo de Agarwal *et al.* é limitado superiormente por $O(n^{11/23}) \cdot \mu^*(G, c, Q)$. Se $r \geq n^{10/23}$, o algoritmo apresentado por Agarwal *et al.* faz uma redistribuição de cargas. Fazendo uso de um jogo combinatório, Agarwal *et al.* conseguem provar que nesse caso o custo do multicorte devolvido é limitado superiormente por $O(n^{11/23}) \cdot \mu^*(G, c, Q)$.

Capítulo 9

Estudo experimental

Em nosso estudo experimental codificamos 5 programas:

- `k-aproximacao`, que implementa o algoritmo MFMC-Iterado,
- `multicorte-gupta`, que implementa o algoritmo Gupta-C-K-R,
- `pl-frac`, que resolve os dois programas lineares vistos no capítulo 4,
- `multicorte-muitos-caminhos`, que implementa o algoritmo Multicorte-BellGreJar, e
- `multicorte-arvores`, que implementa o algoritmo Multicorte-de-Árvore.

Todos esses programas resolvem de maneira exata ou aproximada o Problema do Multicorte Dirigido de Custo Mínimo (MCM). Medimos o desempenho dos programas fazendo testes com instâncias geradas aleatoriamente e com instâncias baixadas da Internet.

9.1 Detalhes das implementações

Todos os algoritmos foram implementados em ANSI C. Os programas `multicorte-muitos-caminhos`, `multicorte-gupta` e `pl-frac` usam a biblioteca CPLEX [IBM] a qual consegue resolver programas lineares e programas lineares inteiros com grande quantidade de restrições em um tempo de execução razoável. O tempo de execução dos programas foi medido usando a função clock(). A função clock() devolve o tempo de CPU decorrido desde o início da execução do programa e ignora o tempo de execução de outros processos no mesmo CPU. O leitor interessado nas implementações e testes feitos durante nosso estudo pode consultar `http://www.ime.usp.br/~juanguti/multicorte/`.

9.2 Detalhes das instâncias baixadas da Internet

Não encontramos benchmarks publicados para o problema MCM. Devido a essa falta, adaptamos algumas das instâncias existentes para outros problemas de otimização combinatória (tipicamente multicommodity flow). As instâncias foram tomados do sítio

Operations Research Group - University of Pisa, `http://www.di.unipi.it/di/groups/optimize/`. Esse sítio contém benchmarks para distintos problemas de otimização, em particular para Multicommodity Problems. Foram tomadas duas famílias de instâncias:

- Família $\mathcal{C}$. Tomadas da seção "The Canad problems", geradas para o problema "Fixed Charge Multicommodity Min-Cost Flow (MMCF)". Essas instâncias foram originalmente publicadas no artigo de Crainic *et al.* [CFG01]. Essa seção contém 31 instâncias, com número de vértices de 20 até 30, número de arcos de 228 até 683, e número de pares de terminais de 39 até 400.

- Família $\mathcal{G}$. Essa família contém instâncias de grades geradas aleatoriamente, tomadas da seção "The Planar and Grid problems", e geradas para o problema "Linear Multicommodity Min-Cost Flow (MMCF)". Essas instâncias foram originalmente publicadas no artigo de Larsson e Yuan [LY04]. Para gerar uma rede, os vértices do digrafo são escolhidos aleatoriamente como pontos no plano. Os arcos são escolhidos tal que o digrafo resultante é uma grade regular, ou seja existem 4 arcos entrando e 4 arcos saindo para cada vértice interno da grade. Os pares de terminais são escolhidos aleatoriamente. Os custos são as distâncias euclidianas entre os vértices. Essa seção contém 15 instâncias de grades, com número de vértices de 25 até 1225, número de arcos de 80 até 4760, e número de pares de terminais de 50 até 32000.

9.3 Detalhe das instâncias geradas aleatoriamente

Para medir o desempenho dos programas `k-aproximacao`, `multicorte-gupta`, `multicorte-muitos-caminhos` e `pl-frac` foram geradas redes aleatórias. Implementamos um algoritmo que gera uma rede aleatória com entrada o número de vértices n, o número de arcos m e o número de pares de terminais k da rede. O algoritmo enumera os vértices da rede de 0 a $n-1$. Para gerar os arcos da rede, o algoritmo escolhe dois vértices $u \neq v$ aleatoriamente. Se uv ainda não é um arco da rede, o algoritmo adiciona uv aos arcos da rede. O custo escolhido para o arco uv é um número inteiro aleatório entre 1 e 100. Para gerar o conjunto de pares de terminais, o algoritmo escolhe aleatoriamente um vértice s e em seguida um vértice t tal que existe um caminho de s a t. Se st ainda não é um par de terminais da rede, o algoritmo adiciona st ao conjunto de pares de terminais da rede.

Foram geradas com esse algoritmo 4 famílias de instâncias de redes aleatórias, que chamaremos de $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$, onde

- $\mathcal{R}_1$ é uma família de redes aleatórias com n vértices, $m = \lfloor n\sqrt{n} \rfloor$ arcos e $k = \lfloor n/2 \rfloor$ pares de terminais, onde n começa em 2 e atinge 300, em passos de 1.

- $\mathcal{R}_2$ é uma família de redes aleatórias com n vértices, $m = \lfloor n^2/2 \rfloor$ arcos e $k = \lfloor n/2 \rfloor$ pares de terminais, onde n começa em 1 e atinge 300, em passos de 1.

- $\mathcal{R}_3$ é uma família de redes aleatórias com n vértices, $m = \lfloor n\sqrt{n} \rfloor$ arcos e $k = \lfloor n^2/2 \rfloor$ pares de terminais, onde n começa em 2 e atinge 300, em passos de 1.

- $\mathcal{R}_4$ é uma família de redes aleatórias com n vértices, $m = \lfloor n^2/2 \rfloor$ arcos e $k = \lfloor n^2/2 \rfloor$ pares de terminais, onde n começa em 1 e atinge 300, em passos de 1.

Para medir o desempenho do programa `multicorte-arvore` foram geradas árvores divergentes aleatórias. Implementamos um algoritmo que gera uma árvore divergente aleatória com entrada o número de vértices n e o número de pares de terminais k da rede. O algoritmo enumera os vértices da árvore de 0 a $n-1$, onde o vértice 0 é a raiz da árvore. Para cada vértice $v > 0$, o algoritmo escolhe aleatoriamente um vértice u entre 0 e $v-1$ e adiciona uv aos arcos da árvore. O custo escolhido para o arco uv é um número inteiro aleatório entre 1 e 100. Para achar o conjunto de pares de terminais, o algoritmo escolhe aleatoriamente um vértice t entre 1 e $n-1$. Posteriormente escolhe aleatoriamente um vértice s que esteja no caminho da raiz da árvore até t e adiciona o par st ao conjunto de pares de terminais. O algoritmo nunca escolhe o mesmo t para dois pares de terminais distintos (uma explicação para isso foi dada na seção 6.5). Portanto, para qualquer k dado como entrada, o algoritmo gera uma rede com no máximo $n-1$ pares de terminais.

Foram geradas com esse algoritmo 2 famílias de instâncias de árvores divergentes aleatórias, que chamaremos de $\mathcal{A}_1$, $\mathcal{A}_2$, onde

- $\mathcal{A}_1$ é uma família de árvores divergentes aleatórias com n vértices e $k = \lfloor n/2 \rfloor$ pares de terminais, onde n começa em 1000 e atinge 50000, em passos de 100.

- $\mathcal{A}_2$ é uma família de redes aleatórias com n vértices e $k = 20000$ pares de terminais, onde n começa em 20200 e atinge 100000, em passos de 200.

9.4 Estudo experimental do algoritmo MFMC-Iterado

O algoritmo MFMC-Iterado apresentado na seção 3.1 é uma k-aproximação para o problema MCM. A implementação desse algoritmo foi feita no programa `k-aproximacao`. Um passo adicional feito no programa `k-aproximacao` é que, depois de achar o multicorte k-aproximado, encontramos um multicorte minimal contido nesse multicorte.

Fizemos testes com as famílias de instâncias $\mathcal{R}_1$ e $\mathcal{R}_3$ para calcular o custo do multicorte produzido pelo programa `k-aproximacao`. Para cada instância, calculamos também o valor do multicorte fracionário c-mínimo com o programa `pl-frac` (que dá uma cota inferior do ótimo), mas não conseguimos encontrar esse valor para todas as instâncias.

Os resultados podem ser observados nas figuras 9.1 e 9.2 respectivamente. Em seguida um resumo dos resultados:

- O custo do multicorte produzido pelo programa `k-aproximacao` nas instâncias de $\mathcal{R}_1$, com $n \leq 245$, é no máximo 2.4 vezes o custo do multicorte fracionário c-mínimo. O programa `pl-frac` só conseguiu terminar em um tempo razoável para n até 245.

- O custo do multicorte produzido pelo programa `k-aproximacao` nas instâncias de $\mathcal{R}_3$ é no máximo 1.3 vezes o custo do multicorte fracionário c-mínimo. O programa `pl-frac` conseguiu terminar para todas as instâncias.

Observe que, para cada instância, o valor da divisão entre o custo produzido pelo programa `k-aproximacao` e o custo do multicorte fracionário c-mínimo, comparado com o valor de k, é muito pequeno. Portanto, nos testes feitos, o programa se comporta bem melhor que o previsto pela análise teórica.

Finalmente fizemos testes com as famílias de instâncias $\mathcal{C}$ e $\mathcal{G}$ descritas na seção 9.2. Os resultados se mostram nas tabelas 9.1 e 9.2. O comportamento da k-aproximação nesses testes é também melhor que o previsto pela análise teórica. Para cada instância da família $\mathcal{C}$, o custo da k-aproximação é no máximo 1.3 vezes o custo do multicorte fracionário c-mínimo. Para cada instância da família $\mathcal{G}$, o custo da k-aproximação é no máximo 1.8 vezes o custo do multicorte fracionário c-mínimo.

9.5 Estudo experimental do algoritmo Gupta-C-K-R

O algoritmo Gupta-C-K-R apresentado no capítulo 8 é uma $8\sqrt{n}$-aproximação para o problema MCM. A implementação desse algoritmo foi feita no programa `multicorte-gupta`. Um passo adicional feito no programa `multicorte-gupta` é que, depois de achar o multicorte aproximado, encontramos um multicorte minimal contido nesse multicorte.

Fizemos testes com as famílias de instâncias $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ e $\mathcal{R}_4$ para calcular o custo do multicorte produzido pelo programa `multicorte-gupta`. Não conseguimos encontrar o multicorte aproximado para todas as instâncias. A razão é que o programa `multicorte-gupta` resolve a relaxação linear associada a cada instância e resolver um programa linear fracionário com um número grande de restrições leva muito tempo. Para cada instância, calculamos também o valor do multicorte fracionário c-mínimo com o programa `pl-frac` (que dá uma cota inferior do ótimo),

Os resultados podem ser observados nas figuras 9.3, 9.4, 9.5 e 9.6. Em seguida um resumo dos resultados:

- Para a família $\mathcal{R}_1$, o programa `multicorte-gupta` só conseguiu terminar em um tempo razoável para n até 211. O custo do multicorte aproximado em uma instância

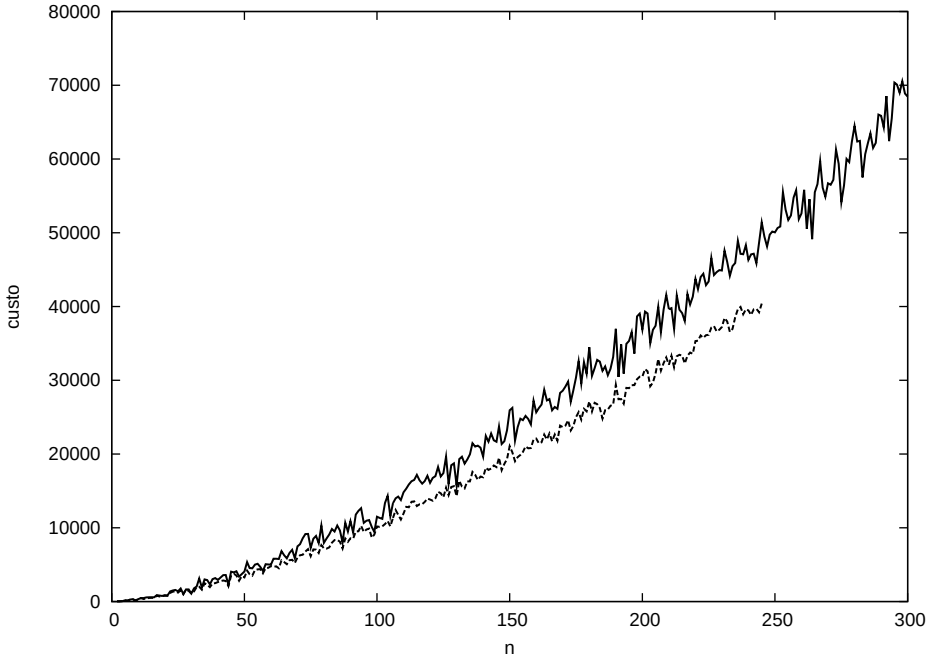

Figura 9.1: Resultados dos testes do programa `k-aproximacao` com a família de instâncias aleatórias $\mathcal{R}_1$. O programa `k-aproximacao` implementa o algoritmo MFMC-ITERADO. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `k-aproximacao`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

de $\mathcal{R}_1$, com $n \leq 211$, é no máximo 2 vezes o custo do multicorte fracionário c-mínimo.

- Para a família $\mathcal{R}_2$, o programa `multicorte-gupta` só conseguiu terminar em um tempo razoável para n até 108. O custo do multicorte aproximado em uma instância de $\mathcal{R}_2$, com $n \leq 108$, é no máximo 1.4 vezes o custo do multicorte fracionário c-mínimo.

- Para a família $\mathcal{R}_3$, o programa `multicorte-gupta` conseguiu terminar em um tempo razoável para todas as instâncias. O custo do multicorte aproximado em uma instância de $\mathcal{R}_3$ é no máximo 1.2 vezes o custo do multicorte fracionário c-mínimo.

- Para a família $\mathcal{R}_4$, o programa `multicorte-gupta` só conseguiu terminar em um tempo razoável para n até 192. O custo do multicorte aproximado em uma instância de $\mathcal{R}_4$, com $n \leq 192$, é no máximo 1.3 vezes o custo do multicorte fracionário c-mínimo.

Observe que, para cada instância, o valor da divisão entre o custo produzido pelo programa `multicorte-gupta` e o custo do multicorte fracionário c-mínimo, comparado com o valor de $8\sqrt{n}$, é muito pequeno Portanto, nos testes feitos, o programa se comporta

bem melhor que o previsto pela análise teórica.

Finalmente fizemos testes com as famílias de instâncias $\mathcal{C}$ e $\mathcal{G}$ descritas na seção 9.2. Os resultados se mostram nas tabelas 9.1 e 9.2. O comportamento do programa `multicorte-gupta` nesses testes é também melhor que o previsto pela análise teórica. Para cada instância da família $\mathcal{C}$, o custo do multicorte devolvido pelo programa `multicorte-gupta` é no máximo 1.3 vezes o custo do multicorte fracionário c-mínimo. Para cada instância da família $\mathcal{G}$, o custo do multicorte devolvido pelo programa `multicorte-gupta` é no máximo 1.6 vezes o custo do multicorte fracionário c-mínimo.

9.6 Estudo experimental das relaxações lineares

No capítulo 4 apresentamos duas relaxações lineares, *PL1* e *PL2*, do problema MCM. A diferença fundamental entre essas duas relaxações é o número de restrições. O programa linear *PL1* tem um número potencialmente exponencial de restrições (no número de vértices do digrafo). O programa linear *PL2* tem um número de restrições polinomial. Os algoritmos que resolvem *PL1* e *PL2* foram implementados no programa `pl-frac`. Nesta seção tentaremos descobrir qual relaxação é a mais eficiente com relação ao tempo de execução.

Resolver *PL2* não precisou de um algoritmo extraordinário pois a a matriz de restrições do *PL2* é conhecida explicitamente. Cada linha da matriz pode ser calculada a partir da matriz de adjacência e a matriz de custos da rede. Detalhes dessa implementação encontram-se no apêndice B.

Resolver *PL1* precisou mais cuidado. Existe uma dificuldade em enumerar todos os Q-caminhos da rede, já que essa quantidade pode ser exponencial no número de vértices. Porém, não é preciso conhecer explicitamente todos os Q-caminhos para resolver *PL1*. Podemos fazer uma enumeração implícita de caminhos, como na seção 4.5. O algoritmo que resolve *PL1* trabalha em cada iteração com um subconjunto do total de Q-caminhos na rede. Dado um tal subconjunto $\mathcal{P}$, dizemos que um vetor de racionais x indexado por E_G é uma *cobertura fracionária* de $\mathcal{P}$ se $x(E_P) \geq 1$ para todo $P \in \mathcal{P}$. Em cada iteração o algoritmo calcula uma cobertura fracionária c-mínima x de $\mathcal{P}$ e toma x como comprimentos nos arcos. Se existe um par de terminais st tais que a x-distância de s a t é menor que 1 então o algoritmo adiciona a $\mathcal{P}$ um caminho mínimo de s a t. Caso contrário o algoritmo termina e x é um multicorte fracionário c-mínimo da rede.

Fizemos testes com as famílias de instâncias $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ e $\mathcal{R}_4$ para calcular os tempos de execução na resolução dos programas *PL1* e *PL2*. Não conseguimos resolver os programas lineares para todas as instâncias A razão é que resolver um programa linear fracionário com um número grande de restrições leva muito tempo.

Os resultados podem ser observados nas figuras 9.7, 9.8, 9.9, e 9.10. Em seguida um resumo dos resultados:

- Para a família $\mathcal{R}_1$ conseguimos resolver, em menos de meia hora, *PL1* para n até 193 e *PL2* para n até 148. Para todas as instâncias, o tempo na resolução de *PL1* foi menor que o tempo na resolução de *PL2*.

- Para a família $\mathcal{R}_2$ conseguimos resolver, em menos de meia hora, *PL1* para n até 100 e *PL2* para n até 86. Para algumas instâncias, o tempo na resolução de *PL1* foi menor que o tempo na resolução de *PL2*. Para outras, o tempo na resolução de *PL2* foi menor que o tempo na resolução de *PL1*.

- Para a família $\mathcal{R}_3$ conseguimos resolver, em menos de meia hora, *PL1* para n até 300 e *PL2* para n até 65. Para todas as instâncias, o tempo na resolução de *PL1* foi menor que o tempo na resolução de *PL2*.

- Para a família $\mathcal{R}_4$ conseguimos resolver, em menos de meia hora, *PL1* para n até 168 e *PL2* para n até 44. Para todas as instâncias, o tempo na resolução de *PL1* foi menor que o tempo na resolução de *PL2*.

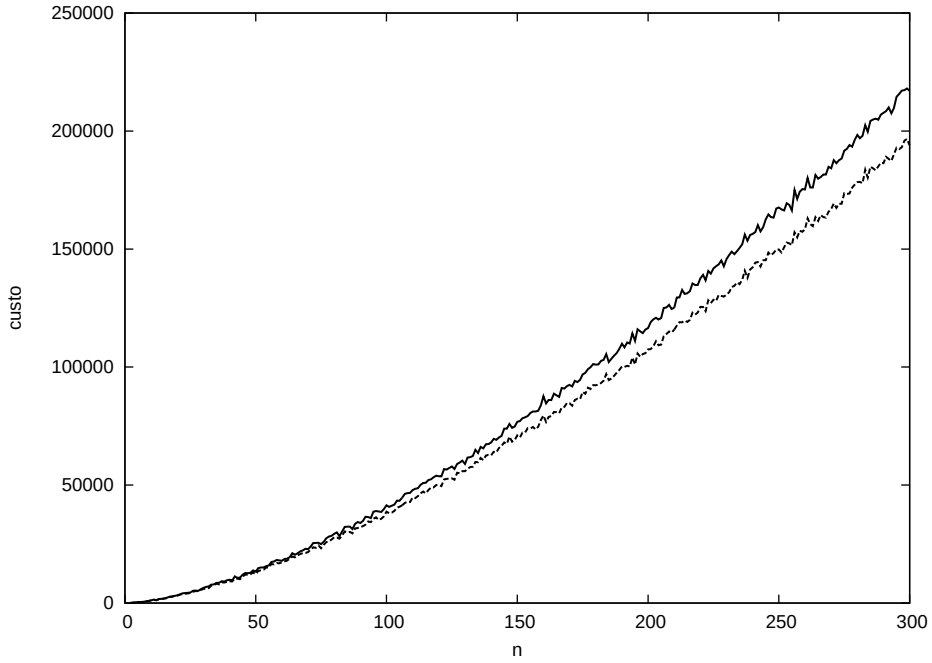

Figura 9.2: Resultados dos testes do programa `k-aproximacao` com a família de instâncias aleatórias $\mathcal{R}_3$. O programa `k-aproximacao` implementa o algoritmo MFMC-ITERADO. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `k-aproximacao`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

Os resultados das figuras nos mostram que, na maioria dos testes, *PL1* é resolvido mais rápido que *PL2*. Uma exceção ocorre na família de instâncias $\mathcal{R}_2$. Nessa família, *PL2* é resolvido mais rápido que *PL1* para alguns valores de n.

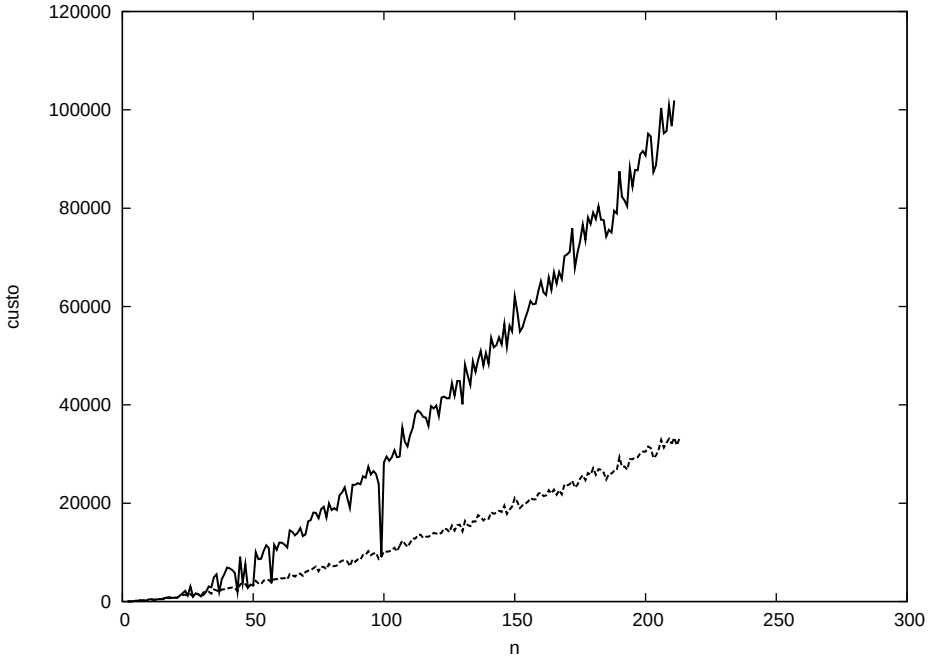

Figura 9.3: Resultados dos testes do programa `multicorte-gupta` com a família de instâncias aleatórias $\mathcal{R}_1$. O programa `multicorte-gupta` implementa o algoritmo GUPTA-C-K-R. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `multicorte-gupta`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

9.7 Estudo experimental do programa multicorte-muitos-caminhos

O algoritmo MULTICORTE-BELLGREJAR apresentado na seção 4.5 resolve o problema MCM de maneira exata. A implementação desse algoritmo foi feita no programa `multicorte-muitos-caminhos`. O algoritmo MULTICORTE-BELLGREJAR faz uma enumeração implícita de caminhos. Em cada iteração trabalha com uma coleção de Q-caminhos e acha uma cobertura c-mínima da coleção. Nestes testes vamos medir, para cada instância, o número de Q-caminhos que o algoritmo enumera antes de terminar, ou seja o tamanho do conjunto $\mathcal{P}$ no final do algoritmo. Esse resultado é importante pois permite comparar o grau de dificuldade do problema entre cada família de instâncias.

Fizemos testes com as famílias de instâncias $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ e $\mathcal{R}_4$. Não conseguimos encontrar o valor do multicorte c-mínimo para todas as instâncias. A explicação para esse fato é a mesma dada na seção 9.6 para o programa `pl-frac`. Também, resolver um programa linear inteiro é mais difícil que resolver um programa linear, por isso o programa `multicorte-muitos-caminhos` consegue resolver menos instâncias que o programa `pl-frac`.

Os resultados podem ser observados nas figuras 9.11, 9.12, 9.13 e 9.14. Em seguida um resumo dos resultados:

- Para a família $\mathcal{R}_1$, o programa `multicorte-muitos-caminhos` só conseguiu terminar em um tempo razoável para n até 72. O número máximo de Q-caminhos encontrados em uma instância de $\mathcal{R}_1$, com $n \leq 72$, é 1999.

- Para a família $\mathcal{R}_2$, o programa `multicorte-muitos-caminhos` só conseguiu terminar em um tempo razoável para n até 55. O número máximo de Q-caminhos encontrados em uma instância de $\mathcal{R}_2$, com $n \leq 55$, é 3651.

- Para a família $\mathcal{R}_3$, o programa `multicorte-muitos-caminhos` só conseguiu terminar em um tempo razoável para n até 149. O número máximo de Q-caminhos encontrados em uma instância de $\mathcal{R}_3$, com $n \leq 149$, é 114610.

- Para a família $\mathcal{R}_4$, o programa `multicorte-muitos-caminhos` só conseguiu terminar em um tempo razoável para n até 47. O número máximo de Q-caminhos encontrados em uma instância de $\mathcal{R}_4$, com $n \leq 47$, é 23829.

Observamos que existe similitude entre a quantidade de Q-caminhos achados nas famílias $\mathcal{R}_1$ e $\mathcal{R}_2$, e entre a quantidade de Q-caminhos achados nas famílias $\mathcal{R}_3$ e $\mathcal{R}_4$. Esse resultado é obvio pois o número de Q-caminhos está em relação diretamente proporcional ao número de pares de terminais. Tanto em $\mathcal{R}_1$ como em $\mathcal{R}_2$, o valor de k é $n/2$. Em $\mathcal{R}_3$ e em $\mathcal{R}_4$, o valor de k é $n^2/2$.

Observe que o número de Q-caminhos não determina a complexidade de uma instância. Para um mesmo n, uma instância da família $\mathcal{R}_4$ chega ter aproximadamente 10 vezes mais caminhos que uma instância da família $\mathcal{R}_1$. Porém, conseguimos resolver mais instâncias da família $\mathcal{R}_4$ que instâncias da família $\mathcal{R}_1$.

Parece ser que a relação entre m e k determina em um grau maior a complexidade de uma instância. Observe a diferença entre os testes de $\mathcal{R}_3$ e $\mathcal{R}_4$. Tanto em $\mathcal{R}_3$ como em $\mathcal{R}_4$ o valor de k é fixo, mas o valor de m em $\mathcal{R}_3$ é menor que o valor de m em $\mathcal{R}_4$. Em $\mathcal{R}_3$ conseguimos resolver mais instâncias que em $\mathcal{R}_4$. Portanto, quando m aumenta com k fixo a complexidade das instâncias aumenta. Uma explicação para esse fato é que quando m aumenta, a possibilidade de existir Q-caminhos disjuntos nos arcos diminui. Em outras palavras, como existem mais arcos, os Q-caminhos encontrados têm mais interseções entre eles, onde uma interseção entre dois Q-caminhos é a quantidade de arcos que compartem. Essa quantidade de interseções causam que resolver o programa linear com o CPLEX seja muito mais difícil que resolvê-lo no caso de ter menos interseções.

Finalmente fizemos testes com as famílias de instâncias $\mathcal{C}$ e $\mathcal{G}$ descritas na seção 9.2. Os resultados se mostram nas tabelas 9.1 e 9.2. O programa `multicorte-muitos-caminhos` conseguiu terminar em um tempo razoável para todas as instâncias da família $\mathcal{C}$. O programa `multicorte-muitos-caminhos` só conseguiu terminar em um tempo razoável para as 4 primeiras instâncias da família $\mathcal{G}$,

não conseguindo terminar para as outras 11 instâncias. Esse fato mostra que resolver um programa linear inteiro é mais difícil que resolver um programa linear.

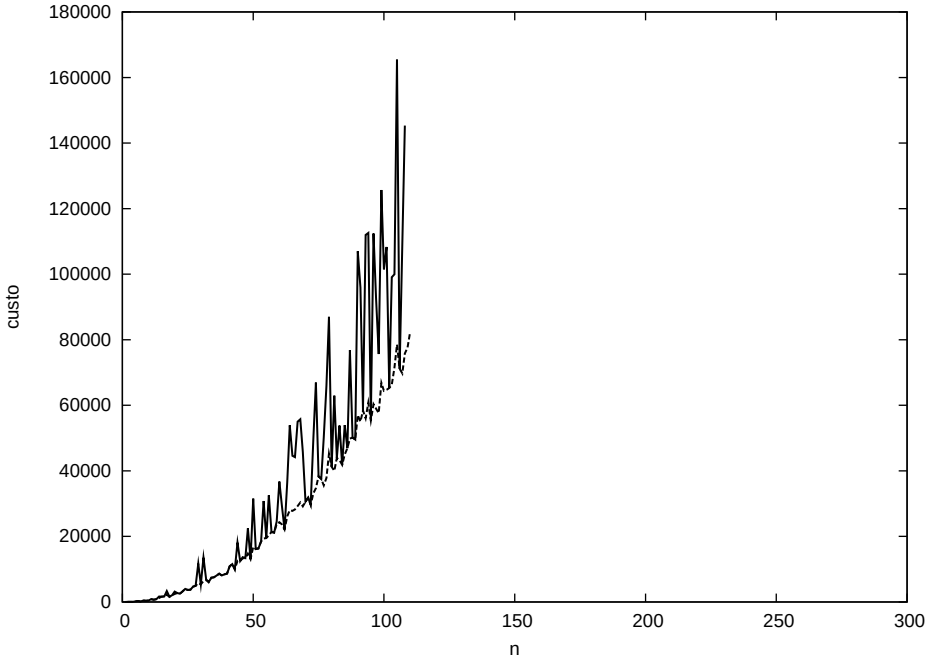

Figura 9.4: Resultados dos testes do programa `multicorte-gupta` com a família de instâncias aleatórias $\mathcal{R}_2$. O programa `multicorte-gupta` implementa o algoritmo GUPTA-C-K-R. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `multicorte-gupta`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

9.8 Estudo experimental do algoritmo Multicorte-de-Árvore

O algoritmo MULTICORTE-DE-ÁRVORE resolve de maneira exata o problema MCM em árvores divergentes. A implementação desse algoritmo foi feita no programa `multicorte-arvores`. Fizemos testes com as famílias de instâncias $\mathcal{A}_1$ e $\mathcal{A}_2$ para medir o tempo de execução do programa. Os resultados podem ser observados nas figuras 9.15 e 9.16 respectivamente.

Nos testes com a família $\mathcal{A}_1$, podemos notar que o formato do gráfico é similar ao gráfico da função $15 \cdot 10^{-8} \cdot n^2$, o que parece sugerir que o algoritmo é $\Theta(n^2)$. Esse resultado parece consistente com a análise da complexidade do algoritmo MULTICORTE-DE-ÁRVORE.

Nos testes com a família $\mathcal{A}_2$, podemos notar que os tempos de execução de cada instância são muito semelhantes entre eles. Na seção 6.5 mostramos que o tempo de execução do algoritmo é $O(kn)$. Já que k é constante, o tempo de execução do algoritmo é $O(n)$. Ou seja, é de esperar que o tempo de execução seja similar ao gráfico de uma função linear em n, porém isso não acontece. Uma explicação para esse fenômeno é a

seguinte. Uma análise mais fina nos diz que o tempo de execução do algoritmo é, na verdade, $O(k \cdot \sum_{i=1}^{k} d_i)$, onde d_i é a distância de s_i a t_i, para cada par de terminais $s_i t_i$. O que acontece é que a soma dessas distâncias não varia muito em relação a n. Como essa soma é quase constante, então o tempo de execução é também quase constante na família de instâncias $\mathcal{A}_2$. O gráfico da figura 9.17 mostra a variação da média das distâncias entre cada par de terminais de cada instância da família $\mathcal{A}_2$, ou seja, no eixo y temos $\sum_{i=1}^{k} d_i/k$ e no eixo x o número de vértices n da árvore. Note que essa média é muito baixa em relação ao número de vértices da árvore. Isso pode ser explicado do fato que o algoritmo que gera a árvore aleatória faz ela não ter muitos níveis de profundidade. Um trabalho futuro é melhorar o algoritmo que gera árvores divergentes aleatórias de um jeito tal que (1) a média das distâncias entre cada par de terminais varie em forma diretamente proporcional a n e (2) a árvore gerada tenha uma maior quantidade de níveis de profundidade para cada n.

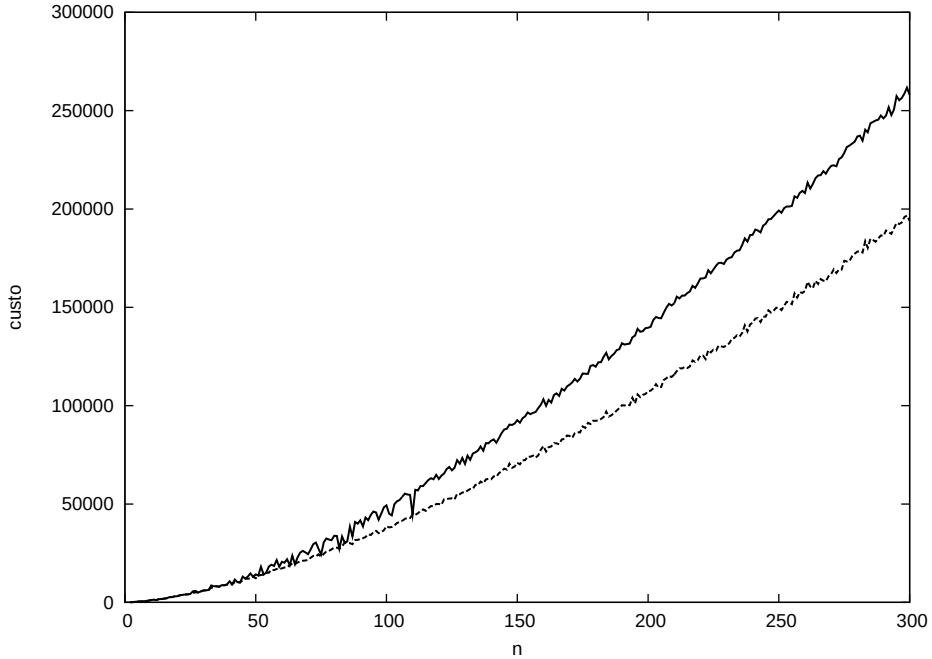

Figura 9.5: Resultados dos testes do programa `multicorte-gupta` com a família de instâncias aleatórias $\mathcal{R}_3$. O programa `multicorte-gupta` implementa o algoritmo Gupta-C-K-R. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `multicorte-gupta`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

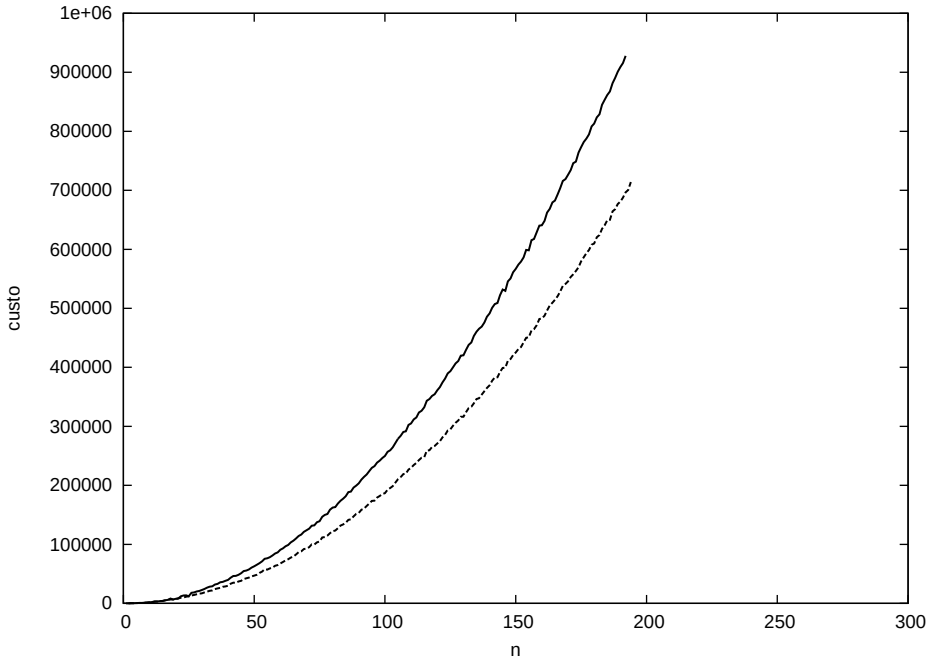

Figura 9.6: Resultados dos testes do programa `multicorte-gupta` com a família de instâncias aleatórias $\mathcal{R}_4$. O programa `multicorte-gupta` implementa o algoritmo GUPTA-C-K-R. A linha não pontilhada é o gráfico do custo do multicorte produzido pelo programa `multicorte-gupta`. A linha pontilhada é o gráfico do custo do multicorte fracionário c-mínimo.

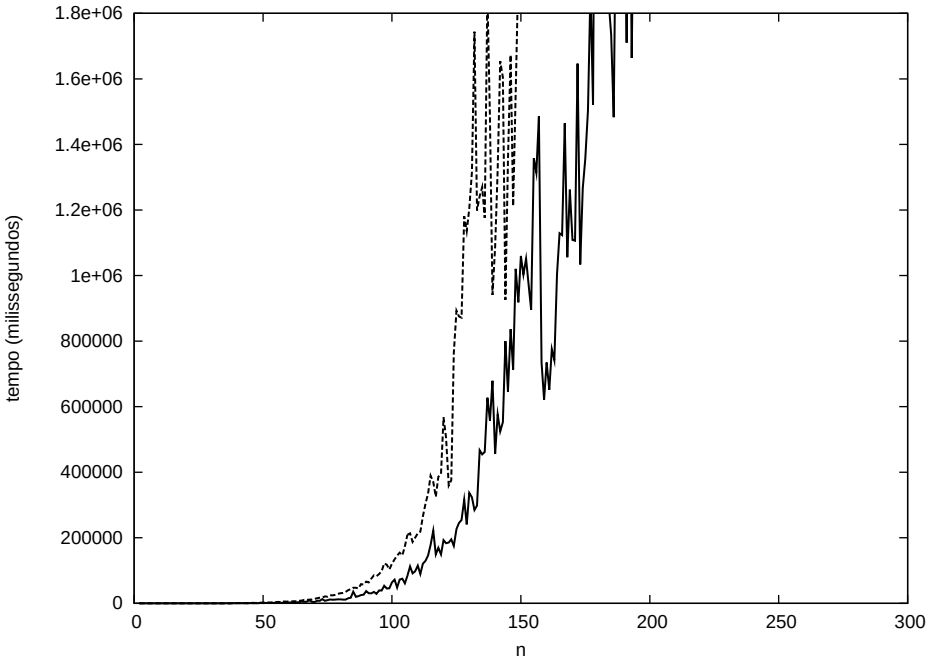

Figura 9.7: Resultados dos testes do programa `pl-frac` com a família de instâncias aleatórias $\mathcal{R}_1$. O programa `pl-frac` resolve os programas lineares *PL1* e *PL2*. A linha não pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL1*. A linha pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL2*.

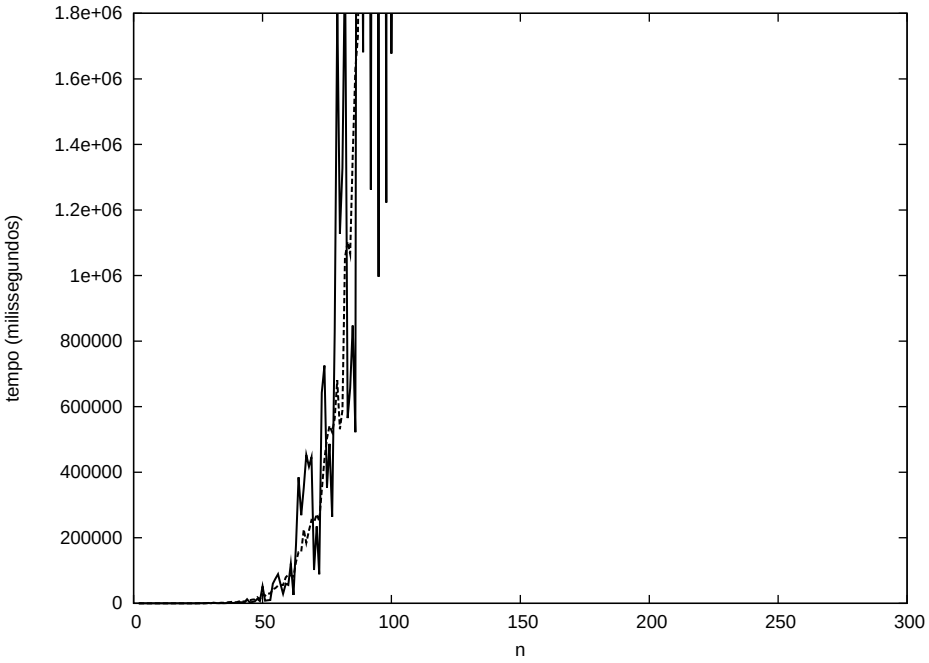

Figura 9.8: Resultados dos testes do programa `pl-frac` com a família de instâncias aleatórias $\mathcal{R}_2$. O programa `pl-frac` resolve os programas lineares *PL1* e *PL2*. A linha não pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL1*. A linha pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL2*.

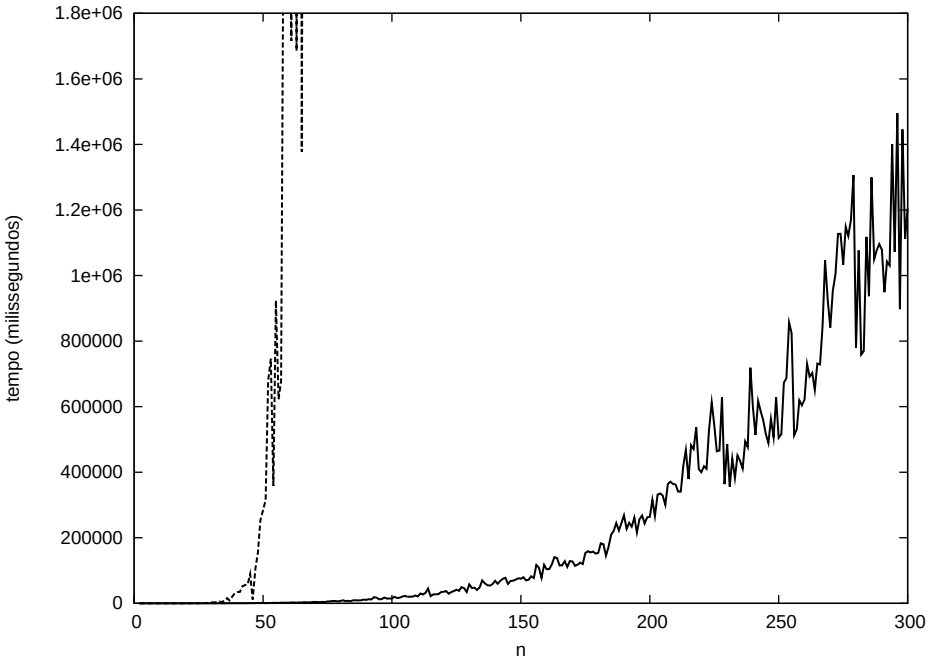

Figura 9.9: Resultados dos testes do programa `pl-frac` com a família de instâncias aleatórias $\mathcal{R}_3$. O programa `pl-frac` resolve os programas lineares *PL1* e *PL2*. A linha não pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL1*. A linha pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL2*.

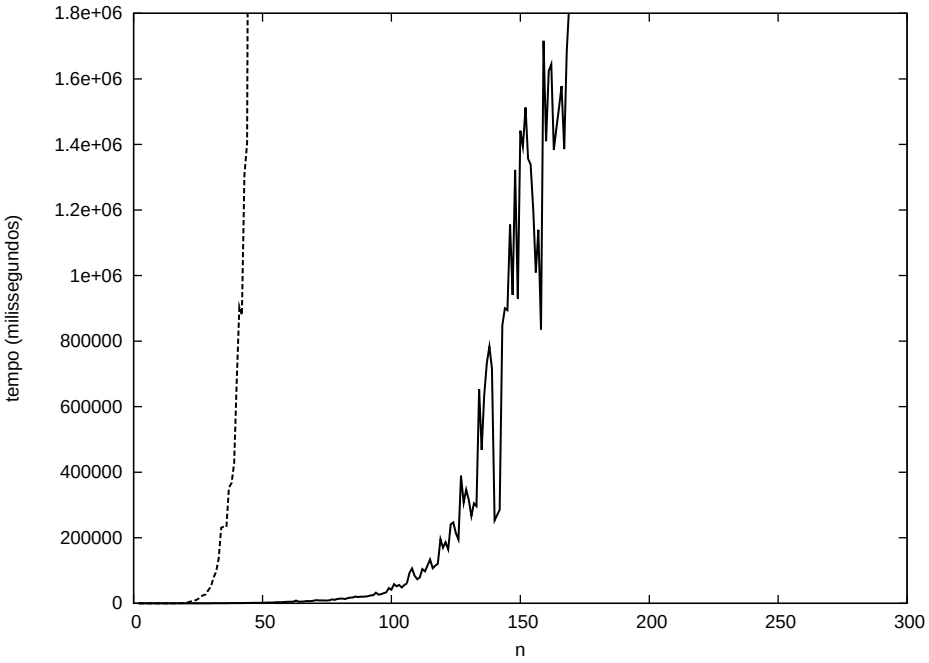

Figura 9.10: Resultados dos testes do programa `pl-frac` com a família de instâncias aleatórias $\mathcal{R}_4$. O programa `pl-frac` resolve os programas lineares *PL1* e *PL2*. A linha não pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL1*. A linha pontilhada é o tempo de execução do programa `pl-frac` quando resolve *PL2*.

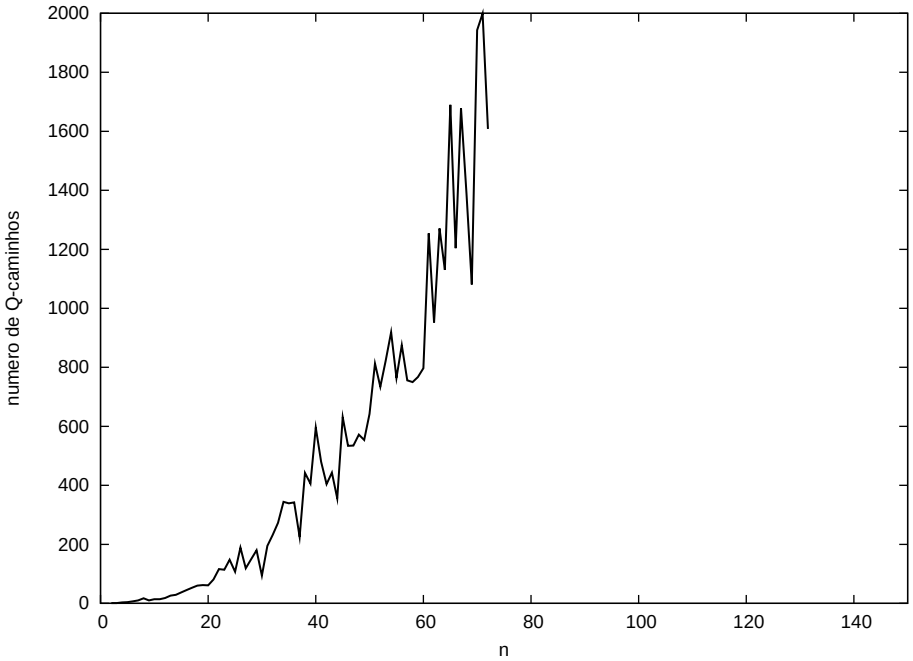

Figura 9.11: Resultados dos testes do programa `multicorte-muitos-caminhos` com a família de instâncias aleatórias $\mathcal{R}_1$. O programa `multicorte-muitos-caminhos` implementa o algoritmo MULTICORTE-BELLGREJAR. O gráfico mostra, para cada instância, o número de Q-caminhos encontrados pelo algoritmo.

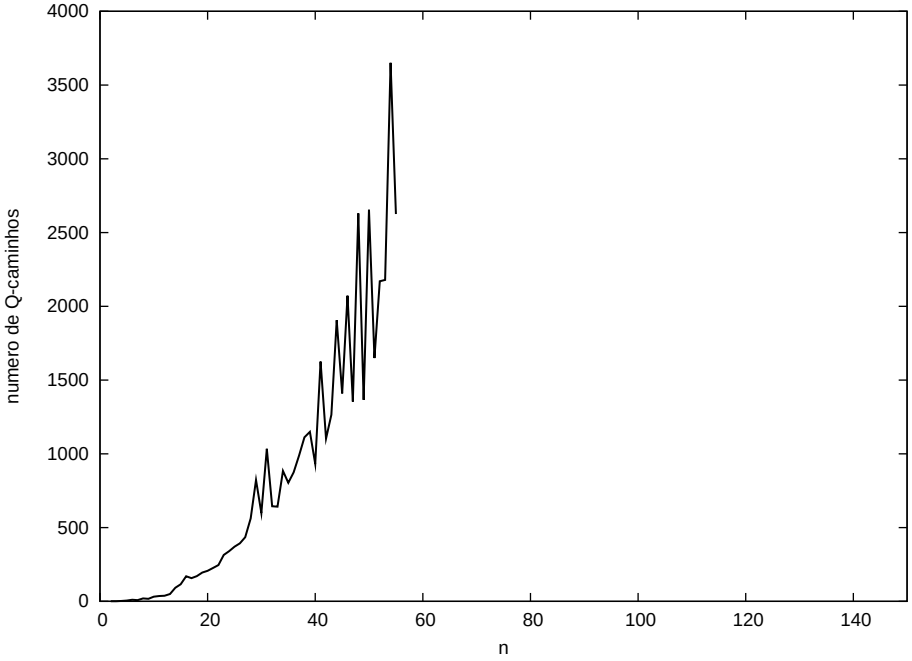

Figura 9.12: Resultados dos testes do programa `multicorte-muitos-caminhos` com a família de instâncias aleatórias $\mathcal{R}_2$. O programa `multicorte-muitos-caminhos` implementa o algoritmo MULTICORTE-BELLGREJAR. O gráfico mostra, para cada instância, o número de Q-caminhos encontrados pelo algoritmo.

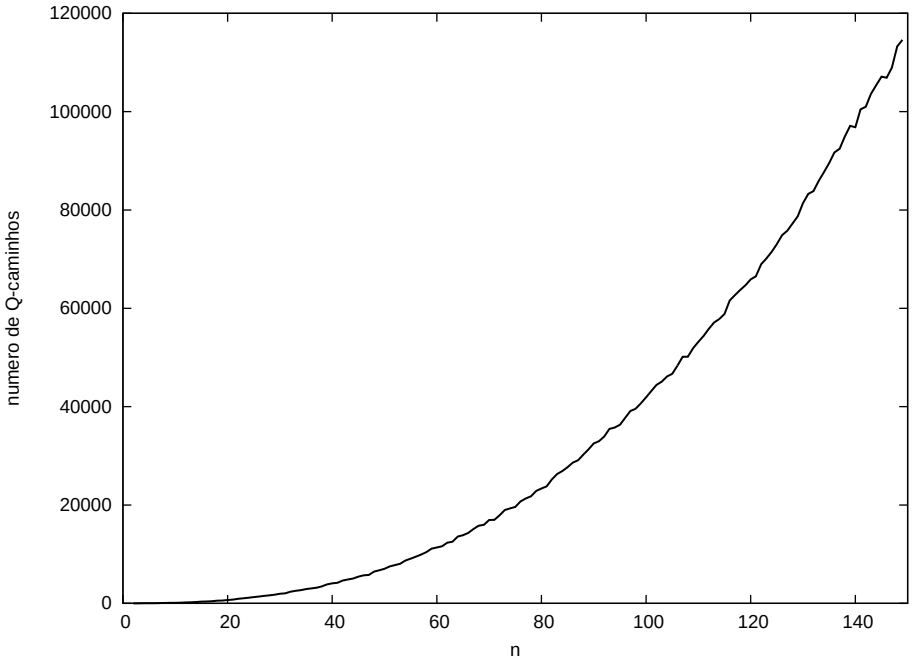

Figura 9.13: Resultados dos testes do programa `multicorte-muitos-caminhos` com a família de instâncias aleatórias $\mathcal{R}_3$. O programa `multicorte-muitos-caminhos` implementa o algoritmo MULTICORTE-BELLGREJAR. O gráfico mostra, para cada instância, o número de Q-caminhos encontrados pelo algoritmo.

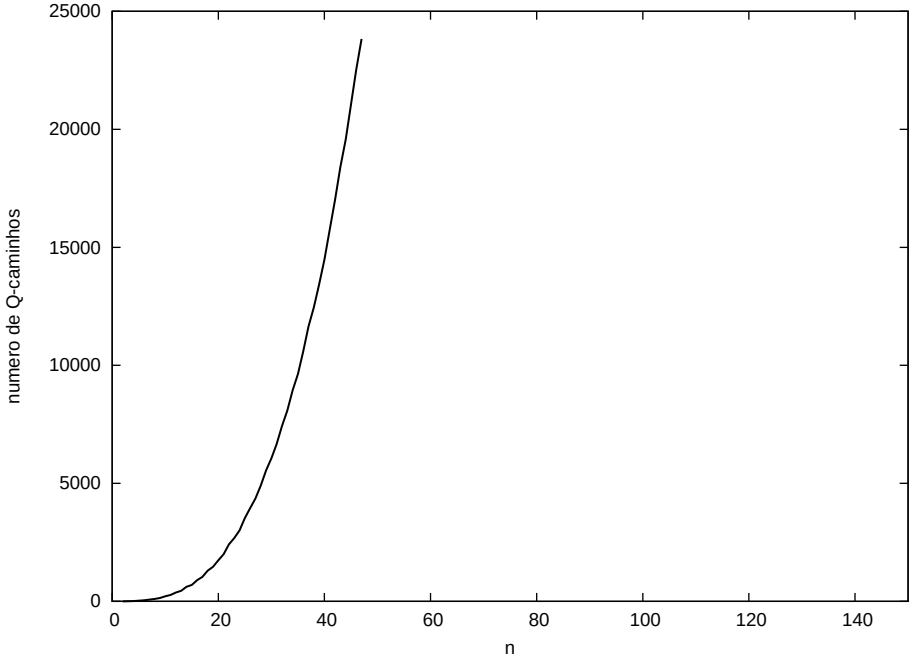

Figura 9.14: Resultados dos testes do programa `multicorte-muitos-caminhos` com a família de instâncias aleatórias $\mathcal{R}_4$. O programa `multicorte-muitos-caminhos` implementa o algoritmo MULTICORTE-BELLGREJAR. O gráfico mostra, para cada instância, o número de Q-caminhos encontrados pelo algoritmo.

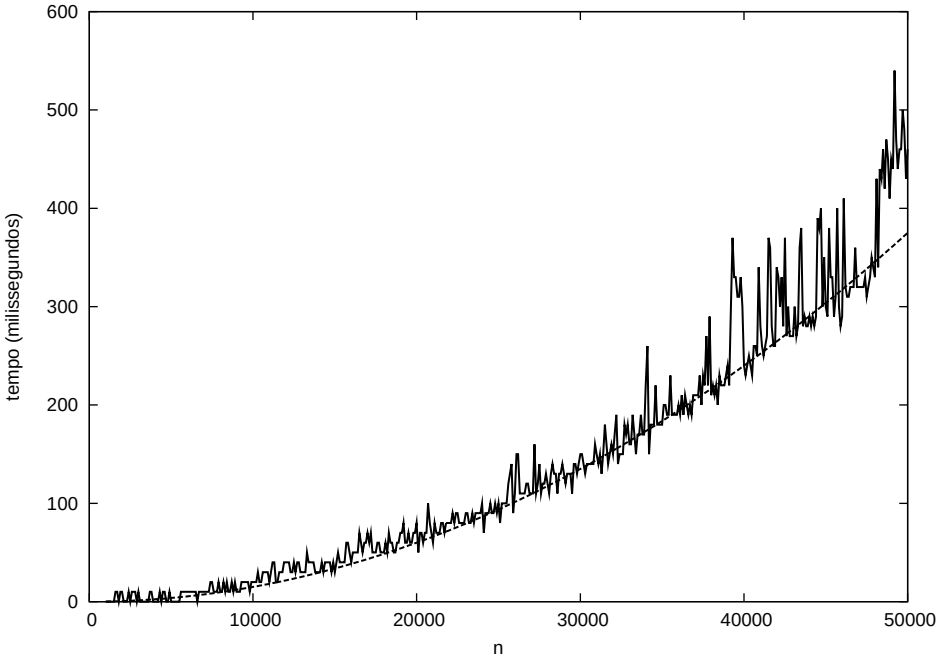

Figura 9.15: Resultados dos testes do programa `multicorte-arvores` com a família de instâncias aleatórias $\mathcal{A}_1$. O programa `multicorte-arvores` implementa o algoritmo MULTICORTE-DE-ÁRVORE. A linha não pontilhada é o tempo de execução em milissegundos do programa `multicorte-arvores`. A linha pontilhada é o gráfico da função $15 \cdot 10^{-8} \cdot n^2$. Ela parece sugerir que o algoritmo MULTICORTE-DE-ÁRVORE é $\Theta(n^2)$.

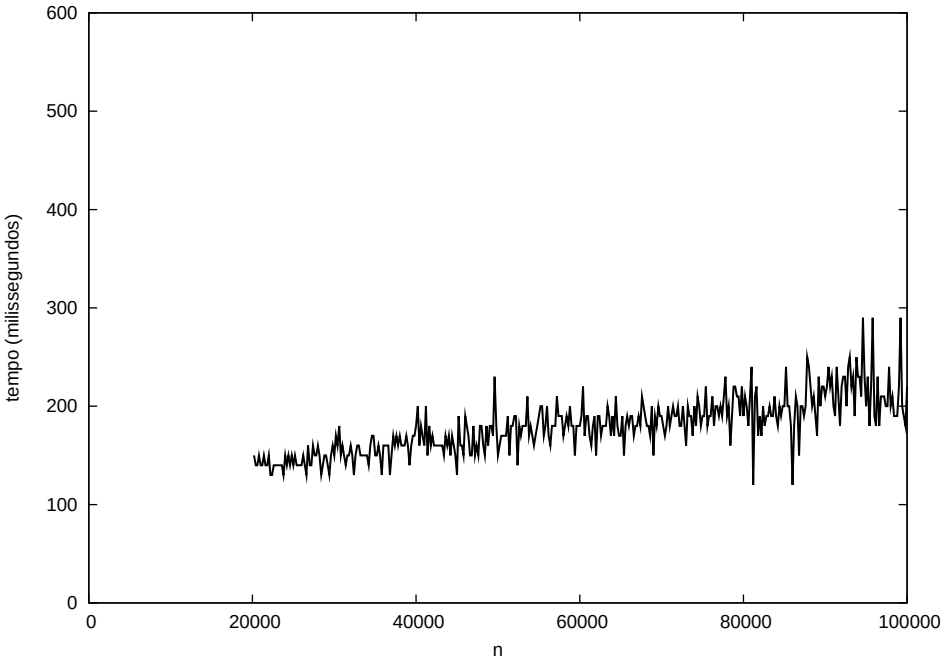

Figura 9.16: Resultados dos testes do programa `multicorte-arvores` com a família de instâncias aleatórias $\mathcal{A}_2$. O programa `multicorte-arvores` implementa o algoritmo MULTICORTE-DE-ÁRVORE. A linha não pontilhada é o tempo de execução em milissegundos do programa `multicorte-arvores`.

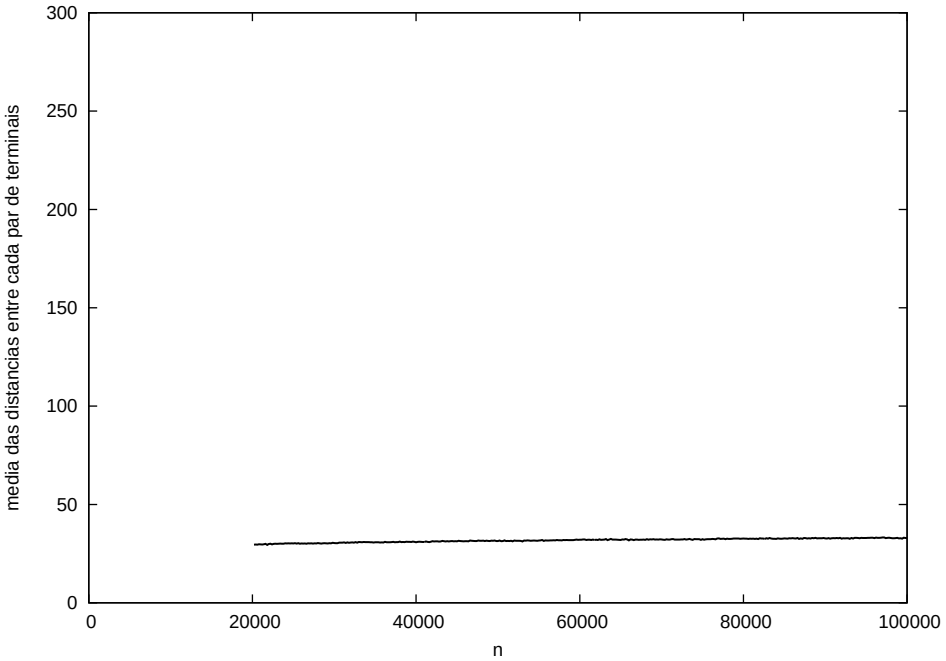

Figura 9.17: Gráfico da média das distâncias entre cada par de terminais da família de instâncias aleatórias $\mathcal{A}_2$. Essa média não varia muito em relação a n, o que explica que o tempo de execução é quase constante na família de instâncias $\mathcal{A}_2$ (veja figura 9.16).

Instância	n	m	k	μ^*	μ	`k-aprox`	`gupta`
c33	20	228	39	2515.00	2515	2927	2515
c35	20	230	40	2612.00	2612	3113	2612
c36	20	230	40	3073.83	3134	3385	3547
c37	20	228	200	810.50	816	854	855
c38	20	230	200	858.00	862	898	911
c39	20	229	200	834.00	834	872	834
c40	20	228	200	835.00	835	839	835
c41	20	228	40	3560.50	3577	3876	3858
c42	20	294	40	3423.00	3423	4372	3423
c43	20	294	40	3910.60	3966	4767	4619
c44	20	294	40	4076.00	4076	4668	4076
c45	20	294	200	1053.00	1053	1116	1053
c46	20	292	200	1039.00	1039	1124	1039
c47	20	291	200	1141.00	1166	1217	1235
c48	20	291	200	1112.00	1112	1194	1208
c49	30	518	100	2841.17	2883	3395	3169
c50	30	516	100	2794.53	2870	3150	3563
c51	30	519	100	2960.36	3067	3368	3508
c52	30	517	100	2858.43	2978	3518	3348
c53	30	520	400	1939.00	2038	2139	2296
c54	30	520	400	1950.50	2031	2172	2124
c55	30	516	400	1854.00	1922	2041	2181
c56	30	518	400	1945.00	2019	2135	2305
c57	30	680	100	3808.00	3839	4367	4165
c58	30	680	100	3876.50	4078	4571	4635
c59	30	687	100	4004.50	4172	4929	4880
c60	30	686	100	3848.00	3981	4428	4644
c61	30	685	400	2457.00	2658	2781	2859
c62	30	679	400	2473.00	2640	2795	2886
c63	30	678	400	2417.50	2597	2772	2928
c64	30	683	400	2618.50	2792	2905	3054

Tabela 9.1: Resultados dos testes com a família de instâncias $\mathcal{C}$ (veja seção 9.2). Para cada instância mostramos o número de vértices (n), o número de arcos (m), o número de pares de terminais (k), o custo do multicorte fracionário c-mínimo (μ^*), o custo do multicorte c-mínimo (μ), o custo do multicorte devolvido pelo programa `k-aproximacao`, e o custo do multicorte devolvido pelo programa `multicorte-gupta`.

Instância	n	m	k	μ^*	μ	`k-aprox`	`gupta`
grid1	25	80	50	2448.50	2452	3127	2537
grid2	25	80	100	3091.67	3129	3479	3385
grid3	100	360	50	4248.84	4300	5440	4441
grid4	100	360	100	5446.13	5478	7188	6712
grid5	225	840	100	6967.69	?	12619	9181
grid6	225	840	200	8976.61	?	15703	13211
grid7	400	1250	400	14827.76	?	25565	20749
grid8	625	2400	500	18749.92	?	34998	28297
grid9	625	2400	1000	23374.53	?	41685	36727
grid10	625	2400	2000	30459.46	?	49294	44951
grid11	625	2400	4000	37500.12	?	56620	51530
grid12	900	3480	6000	51977.49	?	79121	72363
grid13	900	3480	12000	64739.75	?	92520	84021
grid14	1225	4760	16000	78460.69	?	116608	106094
grid15	1225	4760	32000	100049.61	?	135572	129246

Tabela 9.2: Resultados dos testes com a família de instâncias $\mathcal{G}$. (veja seção 9.2). Para cada instância mostramos o número de vértices (n), o número de arcos (m), o número de pares de terminais (k), o custo do multicorte fracionário c-mínimo (μ^*), o custo do multicorte c-mínimo (μ), o custo do multicorte devolvido pelo programa `k-aproximacao`, e o custo do multicorte devolvido pelo programa `multicorte-gupta`.

Apêndice A

Conceitos básicos

Este apêndice é um resumo da terminologia usada ao longo do documento.

Um *digrafo* é uma dupla (V, E), onde V é um conjunto finito e E um conjunto de pares ordenados de elementos distintos de V. Os elementos de V são chamados *vértices* e os de E são chamados *arcos*. Dado um digrafo G, o conjunto de vértices de G é denotado por V_G e o conjunto de arcos de G por E_G. Dado um arco uv, dizemos que uv *entra* em v e *sai* de u. Também dizemos que u é a *ponta inicial* de uv e v é a *ponta final* de uv.

Um digrafo H é *subgrafo* de um digrafo G se $V_H \subseteq V_G$ e $E_H \subseteq E_G$. Dizemos que H é um *subgrafo induzido* de G por V_H se E_H é o conjunto de todos os arcos de G que tem ambas pontas em V_H. O subgrafo induzido de G pelo conjunto de vértices Y é denotado por $G[Y]$. Para qualquer subconjunto Y de V_G, denotamos por $G-Y$ o digrafo $G[V_G \setminus Y]$. Para qualquer vértice v de G, denotamos por $G - v$ o digrafo $G - \{v\}$. Para qualquer subconjunto X de E_G, denotamos por $G - X$ o digrafo com conjunto de vértices V_G e conjunto de arcos $E_G \setminus X$.

Dado um digrafo G e um subconjunto S de V_G, denotamos por $\partial_G(S)$ o conjunto de todos os arcos uv de G tais que $u \in S$ e $v \in V_G \setminus S$. Um *corte* é qualquer conjunto da forma $\partial_G(S)$, onde S é um subconjunto de V_G. Dados dois vértices s, t de G, dizemos que um corte $\partial_G(S)$ *separa* s de t se $s \in S$ e $t \in V_G \setminus S$. Dados dois subconjuntos A, B de V_G, dizemos que um corte $\partial_G(S)$ *separa* A de B se $A \subseteq S$ e $B \subseteq V_G \setminus S$. Dizemos que G é *conexo* se para todo subconjunto próprio não vazio S de V_G, $\partial_G(S) \cup \partial_G(V_G \setminus S) \neq \emptyset$.

Um digrafo G é um *caminho* se V_G admite uma permutação $(v_1, v_2, \ldots, v_n)$ tal que $\{v_i v_{i+1} : 1 \leq i < n\} = E_G$. Os vértices v_1 e v_n são os extremos do caminho. Dizemos que v_1 é o *primeiro vértice* de G e $v_1 v_2$ é o *primeiro arco* de G. Se um caminho P é subgrafo de G, dizemos que P é um caminho em G ou que G contém o caminho P. Se v e w são os dois extremos de um caminho P em G, dizemos que P é um caminho de v a w em G. O *comprimento* de um caminho P é igual a seu número de arcos. Dado um digrafo G e dois vértices v e w de G, um *caminho mínimo* de v a w em G é um caminho de comprimento mínimo de v a w em G. Dois caminhos em G são *disjuntos nos arcos* se não compartem nenhum arco.

Dado um digrafo G, para cada $u, v \in V_G$ definimos a *distância* de u a v em G como o comprimento de um caminho mínimo de u a v em G. Representamos essa distância por $dist(u, v, G)$. Considere agora uma função x que associa um numero real a cada arco de G. Para cada $u, v \in V_G$ definimos a *x-distância* de u a v em G como o comprimento de um caminho mínimo de u a v em G, sendo que x faz o papel de comprimento em cada arco. Representamos essa x-distância por $dist_x(u, v, G)$. Seja (u, v, w) um terno de vértices de um grafo conexo G, a *desigualdade triangular* estabelece que

$$dist(u, v, G) + dist(v, w, G) \geq dist(u, w, G),$$

e analogamente, dada uma função x que associa um numero real a cada arco do digrafo,

$$dist_x(u, v, G) + dist_x(v, w, G) \geq dist_x(u, w, G).$$

Seja G um digrafo. Suponha que c é uma função que associa um número c_e a cada arco e em E_G. Para qualquer subconjunto X de E_G, definimos

$$c(X) := \sum_{e \in X} c_e.$$

Um *grafo* é uma dupla (V, E), onde V é um conjunto finito e E um conjunto de pares não ordenados de elementos distintos de V. Os elementos de V são chamados vértices e os de E são chamados *arestas*. Dado um grafo G, o conjunto de vértices de G é denotado por V_G e o conjunto de arcos de G por E_G. Um grafo H é *subgrafo* de um grafo G se $V_H \subseteq V_G$ e $E_H \subseteq E_G$. Um grafo G é um *caminho* se V_G admite uma permutação $(v_1, v_2, \ldots, v_n)$ tal que $\{v_i v_{i+1} : 1 \leq i < n\} = E_G$. Os vértices v_1 e v_n são os extremos do caminho. Se um caminho P é subgrafo de G, dizemos que P é um caminho em G ou que G contém o caminho P. Se v e w são os dois extremos de um caminho P em G, dizemos que P é um caminho entre v e w em G.

Apêndice B

Código

Em nosso estudo experimental codificamos, entre outros, os seguintes programas:

- `multicorte-muitos-caminhos`, que implementa o algoritmo MULTICORTE-BELLGREJAR,
- `pl-frac`, que resolve os dois programas lineares vistos no capítulo 4.

Neste anexo mostraremos trechos interessantes desses programas, escritos em linguagem C. O código completo está em `http://www.ime.usp.br/~juanguti/multicorte/`. Esses programas usam o pacote IBM ILOG CPLEX [IBM], que resolve programas lineares e programas lineares inteiros.

B.1 Estruturas de dados

As estruturas usadas nas implementações foram matrizes e vetores. Embora implementar uma rede com listas de adjacência é mais eficiente que a implementação com matrizes e vetores, preferimos usar as últimas devido à sua simplicidade na programação e leitura do código. Estas estruturas de dados ficam como um protótipo que pode ser melhorado como desafio futuro.

Dada uma rede (G, c, Q), suporemos que o conjunto de vértices de G é $\{0, 1, \ldots, n-1\}$. O digrafo G é representado por uma matriz `M` com linhas e colunas indexadas pelos vértices de G tal que `M[v][w]` $= 1$ se o arco vw existe em G, e `M[v][w]` $= 0$ caso contrário. A função de custos c é representada por uma matriz `C` com linhas e colunas indexadas pelos vértices de G tal que `C[v][w]` é igual a c_{vw}. O conjunto Q de pares de terminais é representado por um par de vetores `(ss, tt)` indexados por $\{0, 1, \ldots, k-1\}$, sendo que $k = |Q|$ e para cada par de terminais st em Q existe um único i em $\{0, 1, \ldots, k-1\}$ tal que `ss[i]` $= s$ e `tt[i]` $= t$. Uma rede (G, c, Q) é representada por `(M, C, ss, tt)`, onde `M` representa G, `C` representa c, e `(ss, tt)` representa Q. Analogamente, uma rede (G, Q) é representada por `(M, ss, tt)`.

Um multicorte X de uma rede (G, Q) é representado por uma matriz `XX` com linhas e colunas indexadas pelos vértices de G tal que `XX[v][w]` $= 1$ se o arco vw está

em X, e `XX[v][w]` $= 0$ caso contrário. Um multicorte fracionário x de (G, Q) é representado por uma matriz `Xfrac` com linhas e colunas indexadas pelos vértices de G tal que `Xfrac[v][w]` $= x_{uv}$.

B.2 Funções de verificação

Nesta seção mostramos duas funções encarregadas de verificar os resultados dos programas. Elas são `verificaMulticorte` e `verificaMulticorteFrac`.

A função `verificaMulticorte` recebe uma rede `(M, ss, tt)` e uma matriz `X` de zeros e uns com linhas e colunas indexadas pelos vértices da rede. Devolve 1 se `X` representa um multicorte da rede e 0 em caso contrário:

```
int verificaMulticorte (int **M, vertex *ss, vertex *tt, int **X)
{
  int i, r, **MM;
  vertex v, w;
  MM = alocaMatriz (n, n);
  for (v = 0; v < n; v++)
    for (w = 0; w < n; w++)
      MM[v][w] = M[v][w] - X[v][w];
  r = 1;
  for (i = 0; i < k; i++)
    if (caminho_1 (MM, ss[i], tt[i])) {
      r = 0;
      break;
    }
  desalocaMatriz ((void **) MM, n);
  return r;
}
```

O tipo `vertex` é o mesmo que o tipo `int`. A função `verificaMulticorte` faz uso da função `caminho_1`. Essa função recebe a representação `M` de um digrafo e dois vértices `s` e `t` do digrafo e procura um caminho de `s` a `t`. A função devolve 1 se existe tal caminho e 0 em caso contrário.

A função `verificaMulticorteFrac` recebe uma rede `(M, ss, tt)` e uma matriz `Xfrac` com linhas e colunas indexadas pelos vértices da rede. Devolve 1 se `Xfrac` representa um multicorte fracionário e 0 em caso contrário:

```
int verificaMulticorteFrac (int **M, vertex *ss, vertex *tt,
                            double **Xfrac)
{
  int i, r;
  double *d;
  d = malloc (n * sizeof (double));
  r = 1;
  for (i = 0; i < k; i++) {
    dijkstra_1 (M, Xfrac, ss[i], d);
    if (d[tt[i]] < 1.0 - EPSILON) {
      r = 0;
      break;
    }
  }
  free (d);
  return r;
}
```

A função `verificaMulticorteFrac` faz uso da função `dijkstra_1`. Essa função recebe a representação `M` de um digrafo G, uma matriz de reais `Xfrac` com linhas e colunas indexadas pelos vértices do digrafo tal que `Xfrac[u][v]` ≥ 0 para todo par de vértices (u, v), e um vértice `s` de G. Devolve um vetor `d` indexado pelos vértices de G tal que `d[v]` é a distância de `s` a `v` em G tomando `Xfrac[u][v]` como comprimento do arco uv. O valor da constante EPSILON é 10^{-7}.

B.3 Código do programa multicorte-muitos-caminhos

A função `multicorte_muitos_caminhos` implementa o algoritmo MULTICORTE-BELLGREJAR apresentado na seção 4.5, o qual é um algoritmo exato que resolve o problema MCM fazendo uma enumeração implícita de caminhos. Essa função recebe uma rede `(M, C, ss, tt)` e devolve um multicorte `C`-mínimo `X` da rede. Devolve também o número de `(ss, tt)`-caminhos encontrados na enumeração implícita de caminhos.

O processo iterativo das linhas 3-9 do algoritmo MULTICORTE-BELLGREJAR encarrega-se da enumeração implícita de caminhos. Sua implementação aparece no seguinte trecho do código. Esse trecho é interessante pois não é uma implementação direta do algoritmo MULTICORTE-BELLGREJAR, já que fizemos uma heurística que melhora o desempenho do programa. A heurística não adiciona apenas um caminho por cada par st, senão uma coleção maximal de caminhos de s a t.

```
1   for (v = 0; v < n; v++)
2     for (w = 0; w < n; w++)
3       MM[v][w] = M[v][w];
4   while (1) {
5     oldr = r;
6     for (i = 0; i < k; i++) {
7       while (caminho_2 (MM, ss[i], tt[i], pred)) {
8         H = realocaMatriz (H, r, r + 1, m);
9         for (j = 0; j < m; j++) H[r][j] = 0;
10        v = tt[i];
11        while (v != ss[i]) {
12          H[r][B[pred[v]][v]] = 1;
13          v = pred[v];
14        }
15        MM[pred[tt[i]]][tt[i]] = 0;
16        r++;
17      }
18    }
19    if (r == oldr) break;
20    cobertura_otima (H, cc, x, r);
21    for (e = 0; e < m; e++) {
22      if (igual (x[e], 0.0)) X[vv[e]][ww[e]] = 0;
23      else X[vv[e]][ww[e]] = 1;
24    }
25    for (v = 0; v < n; v++)
26      for (w = 0; w < n; w++)
27        MM[v][w] = M[v][w] - X[v][w];
28  }
```

A função `igual` na linha 21 recebe dois números reais e devolve 1 se sua diferença é menor que 10^{-7} e 0 em caso contrário.

A função `caminho_2` na linha 7 faz o mesmo que a função `caminho_1` descrita na seção B.2, com a diferença que `caminho_2` aceita mais um parâmetro, `pred`, que na saída é o vetor de predecessores do caminho de `s` a `t` achado.

A matriz `H` é a matriz de restrições do programa linear inteiro *PL1* (seção 4.4). Em cada iteração do processo iterativo das linhas 7-17 e imediatamente antes da execução da linha 15, `H[r][]` é o vetor característico do conjunto de arcos de um caminho de `ss[i]` a `tt[i]`. Em cada iteração do processo iterativo das linhas 4-28 e imediatamente antes da

execução das linha 25, `X` é cobertura `C`-mínima das linhas de `H`.

Para entender o que as variáveis `B`, `vv`, `ww` e `cc` representam mostramos o seguinte trecho de código. Ele está bem no começo da função `multicorte_muitos_caminhos` e por isso não foi mostrado no trecho anterior:

```
e = 0;
for (v = 0; v < n; v++)
  for (w = 0; w < n; w++)
    if (M[v][w] == 1) {
      cc[e] = C[v][w];
      vv[e] = v;
      ww[e] = w;
      B[v][w] = e;
      e++;
    }
```

A função `cobertura_otima` na linha 20 é a encarregada de invocar o CPLEX. Ela implementa a linha 4 do algoritmo MULTICORTE-BELLGREJAR. A função recebe uma matriz de inteiros `H` com `r` linhas e um vetor de reais `cc`. Devolve um vetor `x`, indexado pelos arcos da rede, com uma solução ótima para o programa linear inteiro com matriz de restrições `H` que minimiza $\texttt{cc} \cdot \texttt{x}$.

É interessante ver como `cobertura_otima` faz uso do CPLEX. A matriz de restrições do programa linear deve ser fornecida ao CPLEX através de 8 vetores: `matcnt`, `matbeg`, `matval`, `matind`, `rhs`, `sense`, `lb` e `ub`. Em seguida descrevemos essas estruturas. O vetor `matcnt` é indexado pelas colunas da matriz de restrições `H`, onde `matcnt[j]` é o número de elementos não nulos na coluna j de `H`. O vetor `matbeg` é indexado pelas colunas de `H`. O vetor `matval` é indexado desde 0 até a quantidade de elementos não nulos de `H` menos um. Os elementos não nulos de `H[i][]` são guardados nas posições `matbeg[i]`, ..., `matbeg[i]+matcnt[i]-1` de `matval`. O vetor `matind` é indexado desde 0 até a quantidade de elementos não nulos de `H` menos um. Suponha que o l-ésimo elemento não nulo da coluna j está na linha i de `H`, então `matind[matbeg[j]+l-1]` $= i$. Os vetores `rhs` e `sense` são indexados pelas linhas de `H`. Esses vetores são tais que `rhs[i]` é o lado direito da i-ésima restrição de `H`, e `sense[i]` é o sentido da i-ésima restrição de `H`, sendo que esse valor é `'G'` se a restrição é da forma maior ou igual. Os vetores `lb` e `ub` são indexados pelas colunas de `H`. Esses vetores são tais que `lb[j]` é o limite inferior da variável correspondente à j-ésima coluna de `H`, e `ub[j]` é o limite superior da variável correspondente à j-ésima coluna de `H`. A função encarregada de copiar essas estruturas de dados para o CPLEX é `CPXcopylp`.

Os tipos de dados das variáveis associadas as colunas de `H` são fornecidas pelo vetor

`ctype`, sendo que o valor de `ctype[j]` é 'B' se a variável associada à coluna j é binária. A função encarregada de copiar essa estrutura ao CPLEX é `CPXcopyctype`.

Em seguida mostramos boa parte do código de `cobertura_otima`:

```
intlp = CPXcreateprob (env, &status, "cob-ot");
matcnt = malloc (m * sizeof (int));
for (j = 0; j < m; j++) {
  matcnt[j] = 0;
  for (i = 0; i < r; i++) matcnt[j] += H[i][j];
}
matbeg = malloc (m * sizeof (int));
matbeg[0] = 0;
for (j = 0; j < m - 1; j++)
  matbeg[j + 1] = matbeg[j] + matcnt[j];
matval = malloc (r * m * sizeof (double));
matind = malloc (r * m * sizeof (int));
for (j = 0; j < m; j++) {
  i_at = 0;
  for (i = 0; i < r; i++)
    if (H[i][j] == 1) {
      matval[matbeg[j] + i_at] = 1.0;
      matind[matbeg[j] + i_at] = i;
      i_at++;
    }
}
lb = malloc (m * sizeof (double));
ub = malloc (m * sizeof (double));
for (j = 0; j < m; j++) {
  lb[j] = 0.0;
  ub[j] = 1.0;
}
rhs = malloc (r * sizeof (double));
sense = malloc (r * sizeof (char));
for (i = 0; i < r; i++) {
  rhs[i] = 1;
  sense[i] = 'G';
}
CPXcopylp (env, intlp, m, r, CPX_MIN, cc, rhs, sense,
           matbeg, matcnt, matind, matval, lb, ub, NULL);
```

```
ctype = malloc (m * sizeof (char));
for (j = 0; j < m; j++)
  ctype[j] = 'B';
CPXcopyctype (env, intlp, ctype);
CPXsetintparam (env, CPX_PARAM_MIPEMPHASIS,
                CPX_MIPEMPHASIS_OPTIMALITY);

CPXmipopt (env, intlp);

lpstat = CPXgetstat (env, intlp);
if (lpstat != CPXMIP_OPTIMAL && lpstat != CPXMIP_OPTIMAL_TOL)
  exit (0);
CPXgetmipx (env, intlp, x, 0, m - 1);
```

A função encarregada de resolver o programa linear inteiro é `CPXmipopt`. A função `CPXgetstat` devolve `CPX_STAT_OPTIMAL` se a solução encontrada por `CPXmipopt` é ótima, e vale `CPXMIP_OPTIMAL_TOL` se a solução devolvida é ótima a menos de um pequeno erro que pode ter sido introduzido pelos arredondamentos.

A função `CPXgetmipx` devolve o vetor característico `x[0...m-1]` de uma solução inteira ótima do programa linear inteiro `intlp`.

B.4 Código do programa pl-frac

O programa `pl-frac` resolve os programas lineares *PL1* e *PL2* apresentados no capítulo 4. A função encarregada de resolver *PL1* é `multicorte_fracionario_caminhos`, a função encarregada de resolver *PL2* é `multicorte_fracionario_arcos`. As seguintes linhas do programa `pl-frac`, permitem ao usuário escolher entre essas duas funções:

```
switch (op) {
  case 1: multicorte_fracionario_caminhos (M, C, ss, tt, Xfrac);
          break;
  case 2: multicorte_fracionario_arcos (M, C, ss, tt, Xfrac);
          break;
}
```

A função `multicorte_fracionario_caminhos` resolve *PL1*. Para isso, ela faz uma enumeração implícita de caminhos semelhante à do algoritmo MULTICORTE-BELLGREJAR. A função recebe uma rede `(M,C,ss,tt)` e devolve um multicorte fracionário `C`-mínimo `Xfrac` da rede. Devolve também o número

de (ss,tt)-caminhos encontrados na enumeração implícita de caminhos. Vejamos como multicorte_fracionario_caminhos faz a enumeração implícita de caminhos:

```
while (1) {
  oldr = r;
  for (i = 0; i < k; i++) {
    dijkstra_2 (M, Xfrac, ss[i], d, pred);
    if (d[tt[i]] < 1.0 - EPSILON) {
      H = realocaMatriz (H, r, r + 1, m);
      for (j = 0; j < m; j++) H[r][j] = 0;
      v = tt[i];
      while (v != ss[i]) {
        H[r][B[pred[v]][v]] = 1;
        v = pred[v];
      }
      r++;
    }
  }
  if (oldr == r) break;
  cobertura_fracionaria_otima (H, cc, x, r);
  for (e = 0; e < m; e++)
    Xfrac[vv[e]][ww[e]] = x[e];
}
```

A função dijkstra_2 faz o mesmo que a função dijkstra_1 descrita na seção B.2, com a diferença que dijkstra_2 aceita mais um parâmetro, pred, que na saída é o vetor de predecessores dos caminhos mínimos a partir de ss[i] sendo que Xfrac[u][v] é o comprimento do arco uv. A função cobertura_fracionaria_otima é a encarregada de invocar o CPLEX. Ela recebe uma matriz de inteiros H com r linhas e um vetor de reais cc. Devolve um vetor x com uma solução ótima para o programa linear com matriz de restrições H que minimiza cc·x. A implementação de cobertura_fracionaria_otima é muito parecida à implementação de cobertura_otima discutida na seção anterior. Para entender o que as variáveis B, vv, ww e cc representam pode-se revisar essa seção.

Para não abusar da paciência do leitor, vamos pular o código da função multicorte_fracionario_arcos, que resolve *PL2*. A complicação fundamental nessa implementação foi entender a partir de (4.11)-(4.13) a estrutura da matriz de restrições do *PL2*. Uma vez entendida essa estrutura, implementá-la com a estrutura de dados usadas pelo CPLEX foi uma tarefa mais mecânica que criativa, porém muito trabalhosa.

B.5 Código do programa multicorte-gupta

Nesta seção mostramos um trecho interessante da implementação do algoritmo GUPTA-C-K-R. A função `subgrafo` encarrega-se da obtenção do subgrafo H na linha 13 do algoritmo. Essa função recebe a representação `M` de um digrafo e dois vértices `s` e `t` do digrafo. Devolve o vetor característico `subconj` de um conjunto de vértices do digrafo tal que para todo vértice `v` nesse conjunto existe um caminho de `s` a `v`, e um caminho de `v` a `t`. O procedimento devolve 1 se esse conjunto é não vazio e 0 em caso contrário.

```
int subgrafo (int **M, vertex s, vertex t, int *subconj)
{
  vertex v, w, *visit;
  int **Mr;
  busca (M, s, subconj);
  if (subconj[t] == 0) return 0;
  Mr = alocaMatriz (n, n);
  for (v = 0; v < n; v++)
    for (w = 0; w < n; w++)
      Mr[w][v] = M[v][w];
  visit = malloc (n * sizeof (int));
  busca (Mr, t, visit);
  for (v = 0; v < n; v++)
    if (visit[v] == 0) subconj[v] = 0;
  free (visit);
  desalocaMatriz ((void **) Mr, n);
  return 1;
}
```

A função `subgrafo` faz uso da função `busca`. Essa função recebe a representação `M` de um digrafo e um vértice `s` do digrafo. Devolve um vetor `subconj` indexado pelos vértices do digrafo tal que `subconj[v]` vale 1 se existe caminho de `s` a `v`, e vale 0 em caso contrário.

Apêndice C

Experiência pessoal

Fazer este trabalho de dissertação foi uma das experiências mas enriquecedoras da minha vida profissional, mas também uma das mais difíceis.

O principal desafio, que acho ainda preciso melhorar, foi organizar e expressar as ideias em um texto matemático bem escrito. Outro desafio importante, muito relacionado com o anterior, foi entender de maneira correta e detalhada os artigos estudados. Muitas vezes esses artigos não dão muitas dicas sobre o que faz o algoritmo, ou deixam de provar certos fatos não tão triviais. Finalmente, implementar os algoritmos de maneira ordenada foi quase tão difícil como escrever o texto. Foi necessário investir muito tempo fazendo debugs e reorganizando o código para fazê-lo o mais eficiente possível. Além disso, eu sempre tive um jeito muito desorganizado de programar. Ajustes de layout, indentação, uma boa nomenclatura das variáveis e comentários adequados no código foram coisas que precisei aprender, já que antes disso minhas habilidades de programação não envolviam quase nenhum desses aspectos. As sugestões dadas pelo professor Paulo foram de grande ajuda para melhorar em cada um destes desafios.

No começo do estudo, li o artigo de Gupta, mas entendi muito pouco. Eu estava no primeiro semestre do mestrado e tinha pouca experiencia em fluxos e programação linear. Fazer as disciplinas de Otimização Combinatória e Análise de Algoritmos no primeiro semestre me ajudou muito a compreender esses tópicos.

Deixei esse artigo para depois e comecei a estudar um caso particular do problema: o que acontece se o digrafo é um caminho com custos unitários? Na minha inocência, achei o problema muito trivial, mas não consegui resolvê-lo facilmente. A seção 6.8 trata desse caso particular, e o algoritmo que o resolve faz uso de uma propriedade minimax. Com esse estudo consegui compreender a importancia do teorema da dualidade em programação linear.

Fiz então um algoritmo de força bruta para resolver o problema MCM. Embora o algoritmo de força bruta não tenha sido difícil de imaginar, escrevê-lo em LaTeX foi difícil, já que eu nunca havia usado esse recurso.

No segundo semestre, implementei esse algoritmo em *C*. A dificuldade de escolher

estruturas de dados adequadas, bem como problemas de programação (ajustes de layout, indentação, nomenclatura das variáveis, comentários adequados), tornaram o processo muito demorado. O resultado dessa implementação não foi positivo, pois não foi possível melhorar o algoritmo e só foi possível rodá-lo para digrafos muito pequenos.

Em seguida estudei o algoritmo para árvores divergentes (capítulo 6). Comecei considerando o caso particular em que a árvore é um caminho com custos arbitrários nos arcos. Já tinha aprendido certas noções de dualidade quando estudei o caso particular de caminhos com custos unitários nos arcos. Estender esse conceito para o caso de caminhos com custos arbitrários não foi tarefa fácil. Tentamos fazer um algoritmo de programação dinâmica, mas isso não funcionou. Só conseguimos aplicar o algoritmo já conhecido para árvores divergentes e adaptá-lo ao caso particular de caminhos com custos arbitrários. A leitura do artigo de Costa *et al.* [CLR03] e também do trabalho feito pelo aluno Pedro Raphael do IME (veja `http://www.ime.usp.br/~cef/mac499-09/monografias/pedro-raphael-rec/`) ajudou a entender o algoritmo para árvores divergentes. Entender o algoritmo foi relativamente fácil; para a prova, no entanto, foi dedicado muito tempo, pois no artigo de Costa *et al.* muitas provas são omitidas. Não existe o estudo detalhado com todas as invariantes necessárias para provar corretamente o algoritmo.

Depois estudei o algoritmo apresentado por Garg *et al.* [GVY93] para o caso não dirigido do problema. Estudei também o algoritmo apresentado por Even *et al.* [ENSS98] para o caso de digrafos simétricos. Foi dedicado um tempo razoável para estudar esses algoritmos. Embora esse estudo não tenha sido incluído na dissertação, ele permitiu entender a técnica de region growing, a mesma aplicada por Gupta e por Kortsarts *et al.*

Já estava no terceiro semestre quando iniciei o estudo dos algoritmos exatos baseados em programação linear (capítulos 4 e 5). Comecei estudando o artigo de Aneja e Ke [AK07], que está muito mal escrito e do qual entendi muito pouco. Foi preciso recomeçar por outro caminho. Estudei então o artigo de Bellmore e Ratliff [BR71], que foi muito bem redigido e me permitiu entender facilmente como funciona o algoritmo proposto. Cabe ressaltar que esse artigo trata do problema de cobertura de conjuntos. A linguagem utilizada nesse artigo é uma linguagem algebraica, que tive que traduzir para uma linguagem puramente combinatória. Grande parte do algoritmo apresentado no capítulo 5 tem como base esse artigo.

Depois comecei o estudo do artigo de Kortsarts *et al.* [KKN05] (capítulo 7), um trabalho bem pequeno, com apenas quatro folhas. Novamente entender a lógica do algoritmo apresentado foi relativamente fácil. A dificuldade estava nos detalhes. Isso me fez refletir que um algoritmo não só é completamente entendido após ser analisado em todos os detalhes. Nesse caso essa análise foi muito mais difícil, já que o artigo não detalha todo o algoritmo. Em primeiro lugar, não são expostos detalhes de teto e chão como descrito no capítulo 7. Além disso, o artigo não mostra os valores das constantes

envolvidas nas equações, lemas e teoremas apresentados. Fazer as contas para calcular esses parâmetros foi um processo muito delicado e trabalhoso. Finalmente, o artigo contém uma prova errada (ou incompleta) do lema 7.2. Corrigir essa prova demandou certo tempo, mas consegui completá-la com ajuda do meu orientador.

No quarto semestre recomecei o estudo do algoritmo de Gupta. Desta vez, entender o algoritmo foi mais fácil. Porém, o artigo de Gupta tem apenas duas páginas e não mostra detalhes suficientes em muitas provas. Não há, por exemplo, a prova do lema 8.2. Essa prova faz uso da técnica de region growing, a mesma usada por Garg *et al.* [GVY93] para o caso não dirigido do problema e por Even *et al.* [ENSS98] para o caso de digrafos simétricos. O estudo prévio desses algoritmos permitiu fazer uma prova desse lema, mas ainda tive certa dificuldade em escrevê-la em uma linguagem correta. No caso do lema 8.3, embora o autor dê a prova para esse lema, completar os detalhes em uma linguagem matemática correta também demandou um tempo considerável.

Estudei então artigo de Cheriyan *et al.* [CKR05]. Cheguei à conclusão de que o algoritmo de Gupta é basicamente o mesmo que o algoritmo de Cheriyan *et al.*. O grande aporte de Gupta foi dar uma melhor análise do algoritmo, obtendo um melhor fator de aproximação.

Terminados todos os estudos teóricos, comecei a implementar alguns dos algoritmos estudados. Eu já tinha começado a programar o algoritmo Multicorte-BellGreJar. da seção 4.5, mas era preciso aprimorá-lo. Começaram as dificuldades com o CPLEX, pois diversos testes demoraram muito para rodar. Tive que alterar alguns parâmetros do CPLEX e otimizar certas regiões do código, o que levou um tempo razoável (para mais detalhes veja as seções 9.7 e B.3).

Depois implementei a formulação do programa linear dado na seção 4.2, o que foi difícil e trabalhoso. A principal dificuldade foi o fato de que a formulação envolve variáveis que não são muito intuitivas à primeira vista. Foram feitas muitas revisões até que eu chegasse à versão final do código.

Em seguida implementei o algoritmo de Gupta, o que não foi difícil de implementar, pois já tinha a experiência dos algoritmos anteriores. A dificuldade veio do fato de que muitas estruturas de dados e muitas funções da biblioteca tiveram que ser revistas e adaptadas para encaixar esse novo algoritmo no código dos algoritmos MFMC-Iterado e Multicorte-BellGreJar. A função que calcula um corte mínimo, por exemplo, teve que ser alterada para receber como entrada dois conjuntos de pares de vértices, S e T, e devolver um corte que separa S de T.

No quinto semestre implementei o algoritmo Multicorte-de-Árvore. Esse algoritmo foi mas rápido de implementar, em parte porque a logica do algoritmo é simples, em parte porque não precisa de muitas funções compartilhadas com o resto dos algoritmos.

Preciso dizer que foi invertida uma quantidade de tempo razoável na implementação

de funções que pudessem verificar os resultados obtidos nas implementações mencionadas anteriormente. Isso fez o código mais confiável.

Como já estava no mestrado havia dois anos, fiz um cronograma para fechar o conteúdo com que me comprometi no exame de qualificação. Exponho abaixo as tarefas que ainda estava por concluir.

Era preciso fechar o capítulo das árvores divergentes. Até então eu só havia feito o algoritmo para o caso particular de caminhos com custos arbitrários. Embora esse algoritmo seja basicamente o mesmo que o algoritmo para árvores divergentes, foi necessário rever a notação em quase sua totalidade e reorganizar as invariantes, o que tornou as suas provas mais complicadas ainda.

O capítulo 5 estava inacabado. A prova da correção do algoritmo não é tão fácil como eu pensava e ainda não estava concluída. Tive que reler o artigo de Bellmore e Ratliff [BR71] a fundo para descobrir uma prova combinatória adequada. Também precisei reler o artigo de Aneja e Vemuganti [AV77] e encaixá-lo no algoritmo de Bellmore e Ratliff.

No capítulo de Kortsarts faltava a análise da complexidade, que não é tão trivial e cuja compreensão e organização exigiram bastante tempo.

Uma vez concluídas essas tarefas, comecei a fazer o capítulo do estudo experimental. Tive que rodar de modo organizado os testes aleatórios e fazer figuras com os resultados obtidos. Para isso, tive que fazer scripts para rodar os programas, organizar os arquivos de saída e rodar scripts para fazer as figuras a serem mostradas no texto. Todos os programas, exceto aqueles para árvores divergentes, foram rodados no servidor brucutu do IME, pois era preciso usar o CPLEX. Todo este processo foi trabalhoso e demandou um tempo considerável. Muitos parâmetros tiveram que ser mudados até que eu chegasse a uma versão final dos testes. Além disso, a compreensão do comportamento de certas figuras demandou uma análise não tão trivial.

Fiz um apêndice com trechos interessantes do código. Esse capítulo teve varias versões, pois inicialmente mostrava quase todo o código que escrevi. Porém, quando tentei explicar por que esses trechos eram interessantes, percebi que não era preciso incluir muito código no apêndice. Foi um tanto frustrante não poder mostrar o código completo no documento, pois foi invertida uma grande quantidade de tempo na implementação dos programas.

Como último passo, li o artigo de Agarwal *et al.* [AAC07], que aprimora o algoritmo de Gupta (veja seção 8.5). Achei esse artigo muito complicado e por isso só consegui extrair informações de alto nível para explicar o que o algoritmo faz. Também reescrevi a seção de caminhos (veja seção 6.8) como caso particular do problema em árvores divergentes. Terminei de redigir a seção de experiência pessoal e revisei bem todo o texto.

Referências Bibliográficas

[AAC07] A. Agarwal, N. Alon, and M. S. Charikar. Improved approximation for directed cut problems. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, pages 671–680, 2007.

[AK07] Y. Aneja and X. Ke. Multicommodity disconnecting set problem. *International Journal of Operations Research*, 4(3):165–171, 2007.

[AV77] Y. P. Aneja and R. R. Vemuganti. A row generation scheme for finding a multi-commodity minimum disconnecting set. *Management Science*, 23(6):652–659, 1977.

[BCF00] L. Brunetta, M. Conforti, and M. Fischetti. A polyhedral approach to an integer multicommodity flow problem. *Discrete Applied Mathematics*, 101(1-3):13–36, 2000.

[BGJ70] M. Bellmore, H. J. Geenberg, and J. J. Jarvis. Multi-commodity disconnecting sets. *Management Science*, 16(6):B427–B433, 1970.

[BR71] M. Bellmore and H. Ratliff. Set covering and involutory bases. *Management Science*, 18(3):194–206, 1971.

[CFG01] T. G. Crainic, A. Frangioni, and B. Gendron. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design problems. *Discrete Applied Mathematics*, 112:73–99, 2001.

[Chv83] V. Chvátal. *Linear Programming*. W.H. Freeman, 1983.

[CK07] J. Chuzhoy and S. Khanna. Polynomial flow-cut gaps and hardness of directed cut problems. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, pages 179–188, 2007.

[CKR05] J. Cheriyan, H. Karloff, and Y. Rabani. Approximating directed multicuts. *Combinatorica*, 25:251–269, 2005.

[CLR03] M. C. Costa, L. Létocart, and F. Roupin. A greedy algorithm for multicut and integral multiflow in rooted trees. *Operations Research Letters*, 31(1):21–27, 2003.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, second edition, 2001.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DJP+94] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23:864–894, 1994.

[EK72] J. Edmonds and R. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[ENRS00] G. Even, J. S. Naor, S. Rao, and B. Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *Journal of the ACM*, 47:585–616, 2000.

[ENSS98] G. Even, J. S. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[Gup03] A. Gupta. Improved results for directed multicut. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 454–455, 2003.

[GVY93] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 698–707, 1993.

[GVY94] N. Garg, V. Vazirani, and M. Yannakakis. Multiway cuts in directed and node weighted graphs. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 487–498. 1994.

[IBM] IBM ILOG. *CPLEX Optimizer*, 12.1 edition. Internet: `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[KKN05] Y. Kortsarts, G. Kortsarz, and Z. Nutov. Greedy approximation algorithms for directed multicut. *Networks*, 45:214–217, 2005.

[KT05] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman, 2005.

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the Twenty-Ninth Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[LY04] T. Larsson and D. Yuan. An augmented lagrangean algorithm for large scale multicommodity routing. *Computational Optimization and Applications*, 27:187–215, 2004.

[Sch03] A. Schrijver. *Combinatorial Optimization*. Springer, 2003.

[SSZ04] M. Saks, A. Samorodnitsky, and L. Zosin. A lower bound on the integrality gap for minimum multicut in directed networks. *Combinatorica*, 24(3):525–530, 2004.

Índice Remissivo