Partição de Grafos em Subgrafos Conexos Balanceados

Renato Pinheiro Fréme Lopes Lucindo

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

Área de Concentração: **Ciência da Computação**

Orientadora: **Prof[a]. Dr[a]. Yoshiko Wakabayashi**

– São Paulo, junho de 2007 –

Partição de Grafos em Subgrafos Conexos Balanceados

Renato Pinheiro Fréme Lopes Lucindo

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Renato Pinheiro Fréme Lopes Lucindo
e aprovada pela comissão julgadora.

São Paulo, 26 de março de 2007.

Banca examinadora:

Profª. Drª. Yoshiko Wakabayashi (orientadora) – IME-USP
Prof. Dr. Carlos Eduardo Ferreira – IME-USP
Profª. Drª. Liliane Rose Benning Salgado – CIN-UFPE

Resumo

Nesta dissertação estudamos — do ponto de vista algorítmico — o seguinte problema, conhecido como *problema da partição conexa balanceada*. Dado um grafo conexo G com pesos atribuídos a seus vértices, e um inteiro $q \geq 2$, encontrar uma partição dos vértices de G em q classes, de forma que cada classe da partição induza um grafo conexo e que, ao considerar as somas dos pesos dos vértices de cada classe, a menor das somas seja o maior possível. Em outras palavras, o objetivo é encontrar q classes cujos pesos sejam tão balanceados quanto possível. Sabe-se que este problema é $\mathcal{NP}$-difícil. Mencionamos alguns resultados sobre complexidade computacional e algoritmos que são conhecidos para este problema. Apresentamos algumas heurísticas que desenvolvemos, todas elas baseadas no uso do algoritmo polinomial para árvores, devido a Perl e Schach, que apresentamos com detalhe. Implementamos quatro heurísticas e um algoritmo de 3/4-aproximação conhecido para o caso $q = 2$. Exibimos os resultados obtidos com os vários testes computacionais conduzidos com instâncias aleatórias, com grafos de diferentes pesos e densidades. Os resultados computacionais indicam que o desempenho dessas heurísticas — todas elas polinomiais — é bem satisfatório. No caso especial em que $q = 2$, observamos que a heurística mais onerosa sistematicamente produziu soluções melhores ou iguais às do algoritmo de aproximação

Palavras-chave: Algoritmo de aproximação, grafos, heurísticas, otimização combinatória, partição conexa balanceada

Abstract

In this dissertation we study algorithmic aspects of the following problem, known as the *balanced connected partition*. Given a connected graph G with weights defined on its vertices, and an integer $q \geq 2$, find a partition of the vertices of G into q classes such that each class induces a connected graph, and furthermore, when we consider the sum of the weights of the vertices in each class, the smallest sum is as large as possible. In other words, the q classes must have weights that are as balanced as possible. This problem is known to be $\mathcal{NP}$-hard. We mention some computational complexity and algorithmic results that are known for this problem. We present some heuristics that we designed, all of them based on the use of the polynomial algorithm for trees, due to Perl and Schach, which we show in detail. We implemented four heuristics and a 3/4-approximation algorithm that is known for $q = 2$. We run tests on many random instances, of graphs with different weights and densities. The computational results indicate that the performance of these heuristics — all of polynomial time complexity — are very satisfactory. For $q = 2$, we observed that the most expensive heuristic produced solutions with values which are systematically better or equal to those produced by the approximation algorithm.

Keywords: Approximation algorithm, balanced connected partition, combinatorial optimization, graphs, heuristics

Sumário

Prefácio

Grafos são estruturas muito usadas para representar a existência ou não de relações entre elementos de um dado conjunto. Assim, redes de comunicação, fluxos em redes de transporte, mapas geográficos e relações binárias em geral podem ser representados por grafos, e neste caso, várias questões de interesse podem ser investigadas. Por exemplo, qual o seu grau de vulnerabilidade (a eliminação de quantas arestas ou vértices causam a perda de conexidade), sua estabilidade (qual o número máximo de vértices independentes que contém), qual o seu diâmetro (maior distância entre quaisquer dois de seus vértices), se podem ser particionados em um certo número de subgrafos conexos, entre outras.

Essa última questão pode ser de interesse tanto pela sua aplicabilidade direta em diversas situações práticas, mas também no seguinte contexto. Algoritmos para problemas em grafos podem utilizar o seguinte procedimento recursivo: dado um grafo conexo G, particiona G em subgrafos conexos de tamanhos aproximadamente iguais, resolve os correspondentes subproblemas para esses subgrafos e de suas soluções obtém uma solução para o grafo original G.

A exigência de que a partição seja tão balanceada quanto possível objetiva minimizar a profundidade da recursão e a discrepância dos tamanhos dos grafos usados em qualquer nível do processo de recursão. Estamos interessados em problemas dessa natureza, onde G é um grafo conexo com pesos associados aos seus vértices.

O problema que é o tema central desta dissertação, que chamamos de **Partição Conexa Balanceada**, denotada por PCB, é o seguinte:

> dado um inteiro $q \geq 2$, um grafo conexo $G = (V, A)$ com uma função peso $w : V \to \mathbb{Z}_+$ definida sobre seus vértices, encontrar uma q-partição $(V_1, V_2, \ldots, V_q)$ de V tal que $G[V_i]$ (subgrafo induzido por V_i) seja conexo para $1 \leq i \leq q$ e o peso do subgrafo mais leve seja o maior possível. Mais formalmente, queremos encontrar uma tal partição que maximiza a função $\min\{w(V_i) : i = 1, \ldots, q\}$, onde $w(X)$ denota o peso do conjunto X, definido como $w(X) = \sum_{x \in X} w(x)$.

As correspondentes variantes desse problema quando q é fixo (não faz parte da entrada), são denotadas por $\mathsf{PCB_q}$.

O foco central dessa dissertação é o estudo — do ponto de vista algorítmico — dos problemas PCB e $\mathsf{PCB_2}$, ambos sabidamente $\mathcal{NP}$-difíceis, conforme mostraram Camerini, Galbiati e Maffioli [11] em 1983 e Dyer e Frieze [16] em 1985, respectivamente.

Antes de mencionarmos alguns resultados conhecidos sobre esses problemas, vejamos por que um um algoritmo ingênuo estaria totalmente fora de cogitação. Considere, a título de ilustração, um grafo com 30 vértices para o qual queremos encontrar uma 10-partição. Neste caso, o número de possíveis q-partições (em classes não-vazias) é

$$173373343599189364594756.$$

Num computador que consiga calcular 10 milhões de possibilidades por segundo, sem contar o tempo para testar viabilidade, esse algoritmo levaria no mínimo 549398697 anos para terminar. Assim, é preciso pensar em estratégias mais espertas para tratar o PCB.

Os nossos estudos começaram com um artigo de Perl e Schach [31] que apresenta um algoritmo polinomial para o PCB restrito a árvores. O algoritmo desenvolvido por esses autores é relativamente simples, já a prova de que o algoritmo de fato encontra uma solução ótima não é tão trivial. Esses estudos e a implementação do algoritmo conduziram-nos, de maneira natural, ao desenvolvimento de heurísticas para o PCB em grafos arbitrários. Essa motivação surgiu da constatação de que não há algoritmos, para o caso de grafos arbitrários, que possam ser úteis do ponto de vista prático e também como rotinas para serem usadas em esquemas de *branch-and-bound* ou *branch-and-cut*.

Assim, essa dissertação contempla a implementação de um algoritmo polinomial para o problema PCB restrito a árvores, e a implementação de quatro heurísticas para o PCB em grafos arbitrários. Estudamos também o algoritmo de aproximação desenvolvido por Chlebíková [13] para o PCB_2, cuja razão de desempenho é 3/4. Implementamos esse algoritmo e realizamos testes computacionais para comparar o desempenho desse algoritmo com o das heurísticas que desenvolvemos.

Os resultados que obtivemos nos testes computacionais foram bastante satisfatórios. Os testes realizados mostraram vários aspectos interessantes, que procuramos ilustrar em gráficos e tabelas.

Nesta dissertação mencionamos alguns resultados conhecidos sobre o PCB e PCB_q. Dentre eles mencionamos os seguintes: para o PCB são conhecidos algoritmos polinomiais apenas para escadas e árvores. Em termos de algoritmos de aproximação, excetuando o algoritmo de Chlebíková [13] para o PCB_2, conhece-se uma 1/2-aproximação para o PCB_3 em grafos 3-conexos, obtida por Salgado [32]. Vale notar que foi provado recentemente por Chataigner, Salgado e Wakabayashi [12] que o problema PCB não admite um PTAS (esquema de aproximação polinomial), a menos que $\mathcal{P}$=$\mathcal{NP}$. Não se sabe, porém, se um tal resultado vale para o PCB_2.

• **Organização do texto**. Esta dissertação está organizada da seguinte forma. Inicialmente, no Capítulo 1, apresentamos alguns conceitos e fixamos a notação que é utilizada no texto.

No Capítulo 2 introduzimos formalmente o problema central de nossos estudos. Apresentamos alguns resultados conhecidos sobre a sua complexidade e alguns algoritmos para casos especiais. Também mencionamos algumas aplicações práticas do problema.

O Capítulo 3 apresenta um algoritmo polinomial para o PCB restrito a árvores, bem como sua implementação e testes computacionais.

A seguir, no Capítulo 4, apresentamos algumas heurísticas baseadas em árvores geradoras que fazem uso do algoritmo discutido no Capítulo 3. Mostramos os resultados dos testes computacionais que realizamos e as comparações entre as diferentes heurísticas.

No Capítulo 5, apresentamos o algoritmo devido a Chlebíková para o PCB_2, que é uma 3/4-aproximação. Discutimos alguns aspectos relativos à implementação desse algoritmo, e apresentamos resultados computacionais comparando o desempenho desse algoritmo com uma das heurísticas mencionadas no Capítulo 4.

Encerramos o texto com o Capítulo 6, no qual apresentamos um resumo das contribuições dessa dissertação e discutimos possíveis trabalhos futuros.

• ***Software* e *Hardware* utilizados**. O texto da dissertação foi produzido em LaTeX. As figuras descritivas foram feitas no Inkscape. Os gráficos foram produzidos com Gnumeric e GNUPlot. Os desenhos de grafos grandes apresentando soluções dadas pelos algoritmos foram gerados com o yEd. A implementação dos algoritmos foi desenvolvida em C++ utilizando a coleção de compiladores GCC. Todos esses são sistemas livres e rodam sobre o sistema operacional Linux (no caso foi utilizada a versão 2.6+).

Todos os testes das implementações feitas foram executados em um AMD Athlon 64 3500+ (2.2GHz, 4417.14 bogomips), com 1.0GB de memória RAM.

• **Agradecimentos**. Agradeço em primeiro lugar à minha família, pelo apoio, força e por acreditarem junto comigo.

Agradeço ã minha orientadora, Profª. Drª. Yoshiko Wakabayashi, por ter aceitado me orientar, pela sua paciência, compreensão e incentivo todos esses anos.

Sou muito grato ao amigo Cassio Campos, por toda ajuda desde o ingresso no programa de mestrado, até a conclusão. Agradeco também todo apoio e paciência dos amigos e colegas de trabalho.

Meu muito obrigado a todos vocês.

Capítulo 1

Preliminares

Este capítulo tem por objetivo apresentar os conceitos e a notação que serão usados nesta dissertação.

1.1 Números e conjuntos

Denotamos por $\mathbb{Z}_+$ o conjunto dos números inteiros estritamente positivos.

Seja P um conjunto e $P_1, P_2, \ldots, P_k$ subconjuntos não-vazios de P. Dizemos que $P = (P_1, P_2, \ldots, P_k)$ é uma partição de P se $P_1 \cup P_2 \cup \ldots \cup P_k = P$ e $P_i \cap P_j = \emptyset$ para todo $1 \leq i < j \leq k$. Chamamos $P_1, P_2, \ldots, P_k$ de **componentes** ou **classes** da partição. Uma partição com k componentes é chamada de **k-partição**. Uma 2-partição também é chamada de **bipartição**.

Lembramos aqui que, o número de k-partições de um conjunto com n elementos é conhecido como **número de Stirling do segundo tipo**, e é dado por

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n.$$

Esses números também satisfazem a seguinte relação de recorrência:

$$S(n,k) = S(n-1,k-1) + kS(n-1,k),$$

onde $S(1,1) = S(2,1) = S(2,2) = 1$. Para dar uma idéia do rápido crescimento desses números, observamos que $S(6,3) = 90$, $S(12,6) = 1323652$, $S(24,12) = 24930204590758260$ e $S(48,24)$ é da ordem de 10^{40}. Na página 52 o leitor encontra uma tabela com alguns valores do número de Stirling do segundo tipo relevantes ao texto.

Se n é um número inteiro então denotamos por $\lceil n \rceil$ (**teto** de n) o menor inteiro maior ou igual a n e por $\lfloor n \rfloor$ (**chão** de n) o maior inteiro menor ou igual a n.

1.2 Teoria dos grafos

O leitor não familiarizado com teoria dos grafos poderá recorrer a algum texto sobre o assunto, dentre os quais indicamos Bollobás [9], Bondy e Murty [10] ou Diestel [15]. Conceitos que não forem apresentados aqui poderão ser encontrados nesses textos. Nosso objetivo aqui é mais estabelecer a terminologia e a notação adotadas.

Um **grafo não-orientado**, ou simplesmente **grafo**, é um par ordenado (V, A) de conjuntos disjuntos, onde cada elemento de A corresponde a um par *não-ordenado* de elementos distintos de V. Chamamos os elementos de V de **vértices** e os elementos de A de **arestas**. A Figura 1.1 mostra um exemplo de grafo. Se G é um grafo, também nos referimos ao seu conjunto de vértices por $V(G)$ e ao seu conjunto de arestas por $A(G)$. No final desta seção nos referimos a grafos orientados.

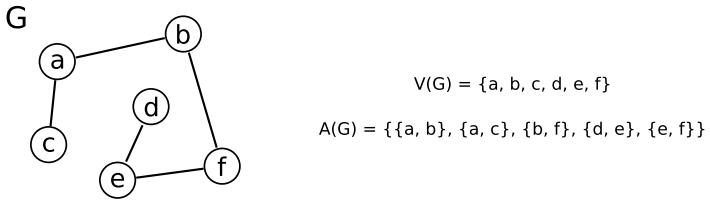

Figura 1.1: Exemplo de grafo

• Adjacência e incidência de vértices e arestas. Denotamos por $\alpha = \{u, v\}$, ou simplesmente uv, a única aresta que corresponde ao par $\{u, v\}$ de elementos de V. Dizemos que α **liga** os vértices u e v ou que α **incide** nos vértices u e v. Chamamos u e v de **extremos** de α. Dizemos também que u é **vizinho** ou **adjacente** a v. Arestas com um extremo em comum são chamadas **adjacentes**.

• Grau de vértices e de grafos. O **grau** de um vértice v, denotado por $g_G(v)$, é o número de arestas que incidem em v. Chamamos de vértice **isolado** um vértice com grau nulo. O **grau mínimo** de um grafo, denotado por $\delta(G)$, é definido como $\delta(G) = \min\{g_G(v) : v \in V(G)\}$. O **grau máximo**, denotado por $\Delta(G)$, é definido como $\Delta(G) = \max\{g_G(v) : v \in V(G)\}$. Dizemos que G é **k-regular** se todos os seus vértices têm grau k.

• Passeios, trilhas, caminhos e circuitos. Um **passeio** P em um grafo é uma seqüência finita de vértices e arestas da forma $(v_1, a_1, v_2, a_2, \ldots, a_{n-1}, v_n)$, em que toda aresta a_i tem v_i e v_{i+1} como extremos. Os vértices v_1 e v_n são chamados de **origem** e **término** do passeio P, respectivamente; os demais vértices são chamados de **internos**. Dizemos que P é um passeio de v_1 a v_n. Quando $v_1 = v_n$ o passeio é dito **fechado**. Se P é um passeio, o seu conjunto de vértices é denotado por $V(P)$, e o seu conjunto de arestas é denotado por $A(P)$. Definimos o **comprimento** de um passeio como $|A(P)|$ e o denotamos por $||P||$. Chamamos de **trilha** um passeio sem arestas repetidas. Um **caminho** é um passeio sem vértices repetidos. Um **circuito** é uma trilha fechada cuja origem e vértices internos são todos distintos.

• Subgrafo e grafo induzido. Dizemos que um grafo H é **subgrafo** de um grafo G se $V(H) \subseteq V(G)$ e $A(H) \subseteq A(G)$, e escrevemos apenas $H \subseteq G$. Podemos dizer que H está contido em G, ou que G contém H. Chamamos H de **subgrafo gerador** de G se $H \subseteq G$ e $V(H) = V(G)$.

Se G é um grafo e $\emptyset \neq X \subseteq V(G)$, denotamos por $G[X]$ o **grafo induzido** por X, definido como o subgrafo de G cujo conjunto de vértices é X e cujo conjunto de arestas consiste de todas as arestas de $A(G)$ que têm os dois extremos em X.

Analogamente, se G é um grafo e $\emptyset \neq B \subseteq A(G)$, denotamos por $G[B]$ o **grafo induzido** por B, definido como o subgrafo de G cujo conjunto de arestas é B e cujo conjunto de vértices consiste de todos os vértices de $V(G)$ que são extremos das arestas em B.

Se G é um grafo e X é um subconjunto de vértices de G, então $G - X$ é o subgrafo de G obtido pela remoção de todos os vértices em X e todas as arestas de G com pelo menos um extremo em X. Se B é um subconjunto de arestas de G, então $G - B$ é o subgrafo de G com conjunto de vértices $V(G)$ e conjunto de arestas $A(G) \backslash B$.

A Figura 1.2 mostra à esquerda um grafo e à direita dois de seus subgrafos: em cima temos um subgrafo gerador e logo abaixo o grafo induzido por $\{a, b, c, d, e, f, g\}$.

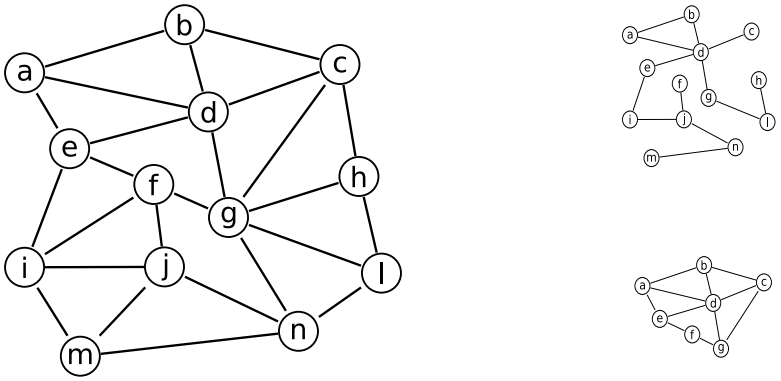

Figura 1.2: Um grafo e dois de seus subgrafos

• Conexidade e cortes. Dizemos que um grafo é **conexo** se para quaisquer dois de seus vértices u e v, existe um caminho de u a v. Chamamos de **componentes** os *subgrafos conexos maximais* de um grafo.

Um grafo G é **k-conexo** se o subgrafo $G - X$ é conexo para todo subconjunto X de $V(G)$ com $|X| < k$. Grafos 2-conexos também são chamados de **biconexos**. Se v é um vértice de um grafo G tal que o subgrafo $G - \{v\}$ não é conexo, então chamamos v de **vértice-de-corte**.

• Partições conexas. Se G é um grafo conexo e $P = (V_1, V_2, \ldots, V_q)$ é uma q-partição de $V(G)$ tal que $G[V_i]$ é conexo para todo $1 \leq i \leq q$, chamamos P de **q-partição conexa** de G.

• Tipos especiais de grafos. Grafos com certas propriedades recebem nomes especiais. Chamamos de grafo **completo** o grafo no qual quaisquer dois vértices distintos são adjacentes. Um grafo completo com n vértices é denotado por K_n.

Um grafo que não contém circuitos é chamado de **floresta**. Uma **árvore** é um floresta conexa. Os vértices de grau 1 de uma floresta são chamados de **folhas**. Um subgrafo gerador que é uma árvore é chamado de **árvore geradora**. Uma árvore pode ter um vértice especial r chamado **raiz**; neste caso a denominamos de **árvore enraizada** (em r) ou árvore com raiz r.

Se T é uma árvore geradora de um grafo G, e α é uma aresta de G que não pertence a T, então $T + \{\alpha\}$ contém precisamente um circuito. Um tal circuito é chamado **circuito fundamental** de G (relativamente a T).

Um grafo G é dito **bipartido** se existe uma bipartição (X, Y) de $V(G)$ tal que todas as arestas de G têm um extremo em X e o outro em Y. Um tal grafo é **bipartido completo** se cada vértice de X for adjacente a cada vértice de Y; neste caso ele é denotado por $K_{n,m}$, onde $n = |X|$ e $m = |Y|$.

Um grafo $G = (V, A)$ é uma **grade** de n *linhas* e m *colunas*, denotada por $\mathcal{G}_{n,m}$, se $V = \{(i, j) : 1 \leq i \leq n,\ 1 \leq j \leq m\}$ e $A = \{\{(i, j), (k, l)\} : (i = k \text{ e } |j - l| = 1) \text{ ou } (j = l \text{ e } |i - k| = 1)\}$. Chamamos de **escada** a grade $\mathcal{G}_{n,2}$ ou $\mathcal{G}_{2,n}$, onde $n \geq 2$.

A Figura 1.3 mostra alguns grafos especiais: (a) uma árvore, (b) uma escada, (c) uma grade $\mathcal{G}_{3,3}$, (d) um grafo completo K_5, (e) um grafo bipartido.

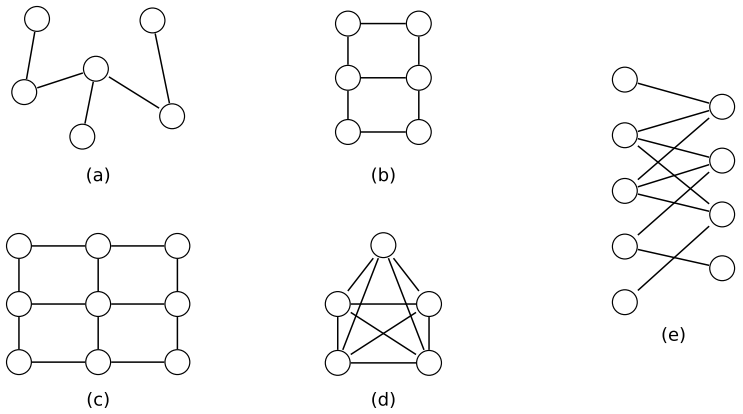

Figura 1.3: Exemplos de alguns grafos especiais

• Grafos com pesos definidos nos vértices e/ou arestas. Em problemas de otimização sobre grafos é natural tratar de grafos nos quais há funções definidas sobre o seu conjunto de vértices e/ou de arestas. Neste texto, no problema de nosso interesse central, lidamos com grafos com uma função peso definida sobre o seu conjunto de vértices. Dado um grafo $G = (V, A)$, no qual está definida uma função peso $w : V \to \mathbb{Z}_+$, adotaremos a seguinte notação: se $X \subseteq V$, então $w(X)$ denota o **peso do conjunto** X, definido como $w(X) := \sum_{x \in X} w(x)$. Também dizemos que $w(X)$ é o **peso**

do subgrafo $G[X]$. Se X e Y são subconjuntos de V, então dizemos que X é **mais leve** do que Y (ou Y é **mais pesado** do que X) se $w(X) \leq w(Y)$.

Neste texto, em geral vamos nos referir a grafos. Apenas no Capítulo 3, na descrição de um dos algoritmos vamos nos referir a grafos orientados: informalmente, podemos dizer que neste caso estamos nos referindo a um grafo no qual as arestas são orientadas. Mais formalmente, um **grafo orientado** é um par ordenado (V, A) de conjuntos disjuntos, onde V é um conjunto de elementos chamados vértices, A é um conjunto de elementos chamados arestas, sendo que cada aresta é um *par ordenado* de elementos de V. Se (u, v) é uma aresta, dizemos que essa aresta é **orientada** (no sentido de u para v). Vários dos conceitos sobre grafos aqui apresentados se transpõem para o caso de grafos orientados, de maneira natural (exemplo: adjacência, conexidade, subgrafo induzido). Não iremos definir esses conceitos, pois não há perigo de confusão no contexto em que serão usados.

• Convenção. Neste texto, reservamos as letras n e m para denotar o *número de vértices* e o *número de arestas*, respectivamente, de um grafo. Assim, quando não houver menção explícita, em contexto de grafos, principalmente quando é feita uma análise da complexidade de um algoritmo, isso deve ficar subentendido.

Dizemos que o **tamanho de um grafo** é o número de vértices desse grafo. Ressaltamos que alguns textos usam o conceito de tamanho como sendo a soma do número de vértices e o número de arestas de um grafo.

1.3 Complexidade e algoritmos de aproximação

É praticamente impossível falar de problemas em otimização combinatória sem esbarrar em questões de complexidade computacional. Vamos supor que o leitor tenha alguma familiaridade com tais conceitos. Se não for este o caso, sugerimos os textos de Sipser [34] e Papadimitriou [30]. Usaremos aqui os símbolos $\mathcal{P}$, respectivamente $\mathcal{NP}$, para denotar a classe dos problemas que podem ser resolvidos por um algoritmo determinístico, respectivamente não-determinístico, em tempo polinomial no tamanho da instância do problema.

Chamamos de **algoritmo de aproximação** um algoritmo polinomial em que temos uma garantia da qualidade da solução obtida, garantia esta expressa por uma medida chamada **razão de aproximação** ou **razão de desempenho**. Essa medida nos informa qual é a pior razão, considerando-se todas as instâncias, entre o valor da solução devolvida pelo algoritmo e o valor de uma solução ótima.

Mais formalmente, seja P um problema de maximização e $\mathcal{I}$ uma instância desse problema. Denotamos por $opt(\mathcal{I})$ o valor de uma solução ótima da instância $\mathcal{I}$. Analogamente, sendo $\mathcal{A}$ um algoritmo para o problema P, denotamos por $\mathcal{A}(\mathcal{I})$ o valor de uma solução obtida ao executarmos $\mathcal{A}$ com a instância $\mathcal{I}$. Dizemos que um algoritmo $\mathcal{A}$ é uma **ρ-aproximação** para o problema P se

$$\mathcal{A}(\mathcal{I}) \geq \rho \cdot opt(\mathcal{I}) \text{ para toda instância } \mathcal{I} \text{ de } \mathcal{P}.$$

Chamamos ρ de **razão de aproximação**. Uma 1-aproximação é um algoritmo exato para o problema. Observamos que ρ pode ser constante, ou uma função que depende do tamanho da instância $\mathcal{I}$ do problema. Em ambos os casos, $0 < \rho \leq 1$.

Chamamos de PTAS (*polynomial time approximation scheme*), esquema de aproximação polinomial, uma família de algoritmos de aproximação para um problema P cuja razão de aproximação é $1-\epsilon$ e o tempo de execução do algoritmo é polinomial no tamanho da instância de P para todo ϵ fixo.

Um FPTAS (*fully polynomial time approximation scheme*), esquema de aproximação completamente polinomial, é uma família de algoritmos de aproximação para um problema P cuja razão de aproximação é $1 - \epsilon$ e o tempo de execução do algoritmo é polinomial no tamanho da instância de P e $1/\epsilon$.

No caso de problemas de minimização, define-se analogamente esses conceitos, trocando-se a desigualdade para $\mathcal{A}(\mathcal{I}) \leq \rho \cdot opt(\mathcal{I})$, sendo que neste caso, $\rho \geq 1$. No caso de PTAS e FPTAS, em vez de $1 - \epsilon$-aproximação, queremos uma $1 + \epsilon$-aproximação.

Caso o leitor tenha interesse em textos sobre algoritmos de aproximação indicamos Ausiello et al. [2], Fernandes et al. [17] e Vazirani [39].

Capítulo 2

Introdução

Neste capítulo definimos os problemas que serão objeto de estudo dessa dissertação, e mencionamos os resultados a respeito que são conhecidos.

2.1 Os problemas

O problema da **Partição Conexa Balanceada**, denotada por PCB, é o seguinte:

> dado um inteiro $q \geq 2$, um grafo conexo $G = (V, A)$ com uma função peso $w : V \to \mathbb{Z}_+$ definida sobre seus vértices, encontrar uma q-partição $(V_1, V_2, \ldots, V_q)$ de V tal que $G[V_i]$ (subgrafo induzido por V_i) seja conexo para $1 \leq i \leq q$ e o peso do subgrafo mais leve seja o maior possível. Mais formalmente, queremos encontrar uma tal partição que maximiza a função $\min\{w(V_i) : i = 1, \ldots, q\}$.

Lembramos aqui que $w(X)$ denota o peso de um conjunto X, definido como a soma dos pesos de seus elementos.

Note que, no PCB, o número de classes da partição desejada, o inteiro q, faz parte da entrada. Uma outra variante do PCB, na qual o inteiro q é *fixo* (não faz parte da entrada), será denotada aqui por $\mathsf{PCB_q}$. Chamamos esse último problema de **q-Partição Conexa Balanceada**. A notação $\mathsf{PCB_{>k}}$ refere-se a qualquer um dos problemas $\mathsf{PCB_q}$ onde $q > k$.

Os casos especiais dos problemas PCB e $\mathsf{PCB_q}$, nos quais os pesos são todos uniformes (ou equivalentemente unitários), serão denotados por $1{-}\mathsf{PCB}$ e $1{-}\mathsf{PCB_q}$.

Na Figura 2.1 vemos um exemplo de duas soluções para o $\mathsf{PCB_2}$. No grafo da esquerda exibimos uma solução viável, enquanto que no grafo da direita exibimos uma solução ótima.

Temos interesse no estudo do PCB e $\mathsf{PCB_q}$ sob o ponto de vista algorítmico. Neste sentido, é natural começar pela investigação de sua complexidade computacional. Descrevemos brevemente os resultados conhecidos a este respeito, e também mencionamos alguns resultados algorítmicos conhecidos.

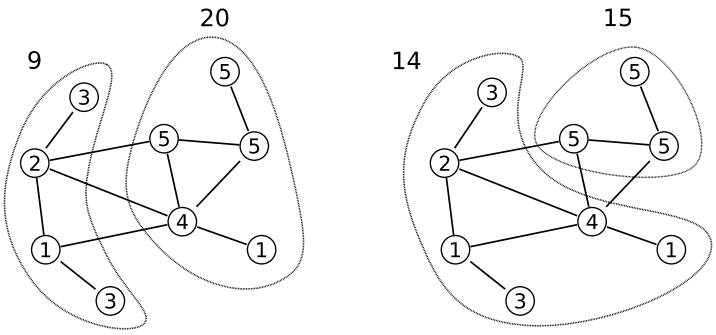

Figura 2.1: Exemplos de solução para o PCB_2

2.2 Alguns resultados conhecidos

O PCB_2 é um problema de grande interesse, e o mais investigado do ponto de vista algorítmico. Em 1983, Camerini, Galbiati e Maffioli [11] mostraram que esse problema é $\mathcal{NP}$-difícil, mesmo no caso de pesos uniformes. Sabemos também que este problema é $\mathcal{NP}$-difícil mesmo para grades com pelo menos 3 colunas [3], e portanto para grafos planares.

Em 1981, Perl e Schach [31] exibiram um algoritmo polinomial para o PCB_2 restrito a *árvores*. Em 2001, Becker, Lari, Lucertini e Simeone [4] mostraram que o PCB_2 pode ser polinomialmente resolvido para *escadas*.

No caso de pesos uniformes, não é difícil provar que o $1{-}\mathsf{PCB}_2$ pode ser resolvido em tempo polinomial se o grafo de entrada é 2-conexo. Assim, é surpreendente o fato de que o $1{-}\mathsf{PCB}_\mathsf{q}$ é $\mathcal{NP}$-difícil mesmo para grafos bipartidos (para todo $q \geq 2$), resultado esse provado por Dyer e Frieze [16] em 1985.

Não é difícil ver que o problema PCB_2 restrito a grafos completos é $\mathcal{NP}$-difícil. Basta notar que o problema da Partição pode ser reduzido à versão de decisão do PCB_2. Como o problema da Partição é $\mathcal{NP}$-completo [19], o resultado segue. Observamos também que podemos resolver o PCB_2 usando o algoritmo pseudo-polinomial (baseado em programação dinâmica) que resolve o problema da Max Mochila [19], combinado com busca binária. Assim, o PCB_2 restrito a grafos completos admite um FPTAS.

Os resultados mencionados acima sobre o PCB_2 podem ser resumidos conforme indicamos na Tabela 2.1.

Em vista dos resultados para o PCB_2, segue que em todos os casos em que o PCB_2 é $\mathcal{NP}$-difícil, o problema PCB também é $\mathcal{NP}$-difícil. Chataigner, Salgado e Wakabayashi [12] provaram que PCB_q é $\mathcal{NP}$-difícil no sentido forte, mesmo para grafos q-conexos ($q \geq 2$). Já no caso de pesos uniformes, o seguinte resultado foi provado por Lovász [22].

> Seja G um grafo q-conexo com n vértices, $q \geq 2$, e sejam $n_1, n_2, \ldots, n_q$ números naturais tais que $n_1 + n_2 + \ldots + n_q = n$. Então G tem uma q-partição conexa $(V_1, V_2, \ldots, V_q)$ tal que $|V_i| = n_i$ para $i = 1, 2, \ldots, q$.

Tipo de grafo	PCB_2	
	pesos uniformes	pesos quaisquer
conexo	$\mathcal{NP}$-difícil	$\mathcal{NP}$-difícil
2-conexo	$\mathcal{P}$	$\mathcal{NP}$-difícil
bipartido	$\mathcal{NP}$-difícil	$\mathcal{NP}$-difícil
árvore	$\mathcal{P}$	$\mathcal{P}$
grade	$\mathcal{P}$	$\mathcal{NP}$-difícil
escada	$\mathcal{P}$	$\mathcal{P}$
completo	$\mathcal{P}$	$\mathcal{NP}$-difícil

Tabela 2.1: Resumo dos resultados conhecidos para o PCB_2

Um algoritmo polinomial para obter a partição tal como descrita no teorema de Lovász pode ser obtido da prova (construtiva) apresentada por Györi [20]).

Algoritmos polinomiais para o 1-BCP$_q$ em grafos q-conexos surgiram nos anos 90. Suzuki, Takahashi e Nishizeki [36] obtiveram um algoritmo linear para o caso $q = 2$, e Suzuki et al. [37] obtiveram um algoritmo polinomial para o caso $q = 3$. Ma e Ma [26] obtiveram algoritmos polinomiais para $q \geq 2$. No caso em que $q = 4$ e o grafo é planar, em 1977 Nakano, Rahman e Nishizeki [29] obtiveram um algoritmo linear.

Para o PCB e $\mathsf{PCB_q}$ não são muitos os algoritmos que foram desenvolvidos. Os casos especiais para os quais há algoritmos polinomiais restringem-se aos casos acima mencionados: escadas e árvores.

Para o PCB_2, o melhor resultado conhecido é um algoritmo de aproximação obtido por Chlebíková [13] que resolve o problema dentro de uma razão de 3/4. Existe também um PTAS para o PCB_2 devido a Salgado [32] para a classe especial de grafos 2-conexos em que o conjunto dos vértices de peso maior que 1 induz um grafo completo.

Para o $\mathsf{PCB_3}$ em grafos 3-conexos, Chataigner, Salgado e Wakabayashi [12] obtiveram uma 1/2-aproximação.

Em suma, para grafos arbitrários e $q \geq 3$, o problema $\mathsf{PCB_q}$ ainda foi pouco investigado. Um resumo dos poucos resultados conhecidos está na Tabela 2.1. Os casos indicados com ´?´ são aqueles para os quais não se conhecem algoritmos polinomiais ou de aproximação.

Tipo de grafo	PCB_2	$\mathsf{PCB_3}$	$\mathsf{PCB}_{>3}$
árvore	polinomial	polinomial	polinomial
escada	polinomial	polinomial	polinomial
conexo	3/4-aproximação	?	?
2-conexo	3/4-aproximação	?	?
3-conexo	3/4-aproximação	1/2-aproximação	?

Tabela 2.2: Resumo dos algoritmos conhecidos para o $\mathsf{PCB_q}$

Encerramos esta seção mencionando um resultado de inaproximabilidade sobre o PCB. Chataigner, Salgado e Wakabayashi [12] provaram que o PCB não pode ser aproximado dentro de uma razão maior do que 5/6, e portanto não admite um PTAS. Um resultado dessa natureza não é conhecido para o PCB_q, nem mesmo para o caso PCB_2.

2.3 Algumas aplicações

Existem várias aplicações do PCB. Algumas delas o leitor interessado poderá encontrar na tese de doutorado de Liliane Salgado [32]. A seguir reproduzimos (salvo pequenas alterações) uma dessas aplicações.

Suponha que um órgão público deseje fazer uma partição de um terreno constituído de vários lotes. Tal terreno é uma área de mineração retangular, que deve ser distribuída a q companhias, para extração de minérios. Essa partição deve ser tal que cada companhia opere livremente nos seus lotes, sem ter que transitar em lotes que não lhe cabe, ou seja, cada conjunto de lotes atribuídos a cada companhia deve ser conexo. Para fazer a distribuição, é feita uma prospecção geológica e obtida uma estimativa da probabilidade de descobrir minérios em cada lote. Para que a distribuição seja justa (balanceada), o orgão público deseja fazer uma subdivisão que maximize a probabilidade mínima de descobrir minérios em cada conjunto de lotes. Neste caso temos uma aplicação do PCB, cujo grafo de entrada é uma grade, e o peso dos vértices é uma estimativa de probabilidade. Essa aplicação é mencionada no artigo de Becker et al. [3]. Nesse mesmo artigo é mencionada uma aplicação em que o problema consiste em atribuir supervisores a setores dentro de uma mesma fábrica, atendendo questões de tempo de supervisão e conexidade da área sob responsabilidade de cada supervisor.

Mais aplicações foram apresentadas por Becker et. al. [4]: há exemplos sobre balanceamento de tarefas de supervisão em uma linha de montagem de circuitos eletrônicos, e em problema de patrulhamento fluvial. Outras aplicações que surgem são nas áreas de processamento de imagens [1, 23, 24, 28], paginação em banco de dados [7] e análise de *clusters* [27].

Capítulo 3

Partição conexa balanceada de árvores

Encontramos na literatura vários algoritmos polinomiais para se obter partições conexas de árvores, que consideram as mais variadas funções objetivo [25, 21, 31, 8, 5, 18, 6, 7, 27]. Neste capítulo, o problema de nosso interesse é o PCB em árvores, que consiste em encontrar uma partição conexa balanceada de uma árvore. Este problema foi primeiramente resolvido por Perl e Schach [31] em 1981. Esses autores introduziram uma técnica de *deslocamento de cortes* que foi aplicada posteriormente a outros problemas de particionamento de árvores [8, 7].

Neste capítulo mostraremos o algoritmo para o PCB em árvores devido a Perl e Schach, que servirá como base para o desenvolvimento de algumas heurísticas para o PCB em grafos arbitrários. A apresentação que faremos aqui é baseada num artigo mais recente de Becker e Perl [7].

3.1 Algoritmo de Perl & Schach

Antes de descrever o algoritmo, vamos apresentar algumas definições e estabelecer a notação a ser usada. Consideraremos aqui que as árvores são orientadas e enraizadas. Se $\alpha = (v_1, v_2)$ é uma aresta (orientada de v_1 a v_2), vamos nos referir ao vértice v_1 como CAUDA(α) e ao vértice v_2 como PONTA(α). Também dizemos que v_2 é **filho** de v_1; e que v_1 é **pai** de v_2, escrito como PAI(v_2).

É imediato que uma q-partição conexa P de uma árvore $T = (V, A)$ fica determinada pela remoção de $q-1$ arestas distintas de T. Tais arestas são chamadas **P-cortes**. A idéia básica do algoritmo que veremos é a seguinte. Para encontrar tais $q-1$ arestas que definem uma q-partição ótima (no sentido da função objetivo do PCB), o algoritmo irá atribuir $q-1$ **rótulos** $c_1, c_2, \ldots, c_{q-1}$, também chamados **cortes**, às arestas de T. Inicialmente esses cortes são atribuídos a uma mesma aresta (incidente à raiz r de T). À medida que o algoritmo é executado, esses cortes vão sendo deslocados a outras arestas. Na Figura 3.4 indicamos alguns deslocamentos.

Mais precisamente, um **deslocamento** de um corte c é a operação que consiste em transferir c de uma aresta α_1 (à qual c estava atribuída) para uma aresta adjacente α_2 à qual não há, nesse momento, nenhum corte atribuído, e tal que $\text{PONTA}(\alpha_1) = \text{CAUDA}(\alpha_2)$. Ou seja, os deslocamentos são sempre *top-down* (afastando-se da raiz). Caso não exista uma tal aresta α_2 com a propriedade mencionada, dizemos que o corte c não pode ser deslocado. Note que essa restrição imposta para fazer o deslocamento garante que nenhum corte será atribuído a uma mesma aresta mais de uma vez (isto é importante na análise da complexidade do algoritmo).

Inicialmente, o algoritmo faz uso do procedimento AJUSTA-ÁRVORE (T, w). Este procedimento recebe uma árvore $T = (V, A)$ e uma função peso w (definida nos vértices de T), e devolve uma nova árvore com $|V(T)| + 1$ vértices. Tal árvore consiste de uma cópia de T na qual escolhemos um vértice arbitrário, digamos v, então adicionamos um novo vértice para ser a raiz, digamos r, de peso nulo, e depois adicionamos uma aresta orientada de r para v. As arestas originais de T são orientadas no sentido da raiz para as folhas de T. Estendemos a função peso w de maneira natural para a nova árvore, e definimos $w(r) = 0$. Notamos que r é um vértice artificial, não pertencente ao grafo original (introduzido apenas para inicializar a atribuição dos cortes). Essa construção pode ser vista na Figura 3.1.

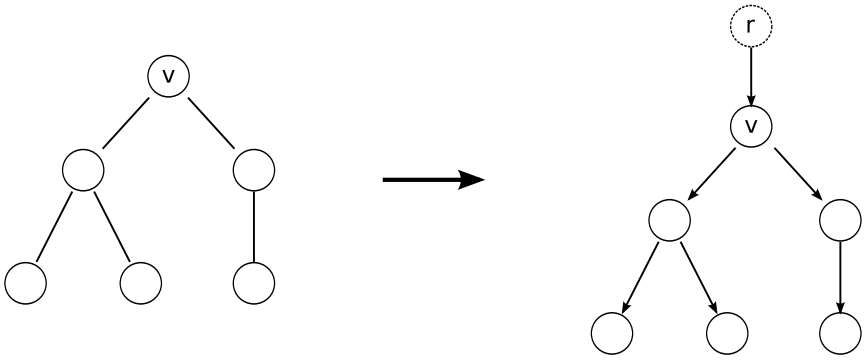

Figura 3.1: Construção feita pelo procedimento AJUSTA-ÁRVORE

As seguintes definições serão úteis nas demonstrações que apresentaremos. Neste contexto, T é uma árvore enraizada orientada. A **componente inferior de um vértice** v é a subárvore de T com raiz v que contém todos os vértices e arestas que podem ser visitados por uma *busca no sentido das arestas* em T começando em v e sem utilizar arestas com cortes atribuídos. A **componente inferior de uma aresta** α é a componente inferior de $\text{PONTA}(\alpha)$. A **componente inferior de um corte** c atribuído a uma aresta α é a componente inferior da aresta α. A **componente superior de um corte** c atribuído a uma aresta α é a componente inferior de $\text{CAUDA}(\alpha)$. Veja a Figura 3.2.

Uma **subárvore parcial de um vértice** v é uma subárvore de T enraizada em v que contém exatamente um filho de v e todos os seus descendentes. Chamamos de **subárvore completa de um vértice** v a subárvore de T enraizada em v que contém todos os filhos de v e seus descendentes. A Figura 3.3 mostra à esquerda a subárvore

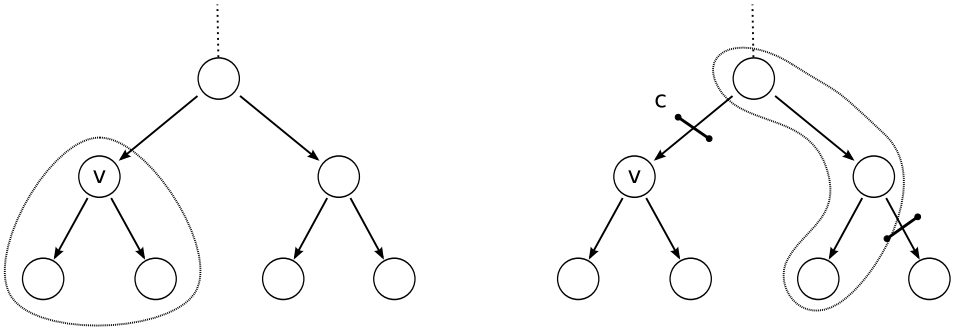

Figura 3.2: (a) componente inferior de v, (b) componente superior de c

completa de um vértice v e à direita as duas subárvores parciais desse mesmo vértice v.

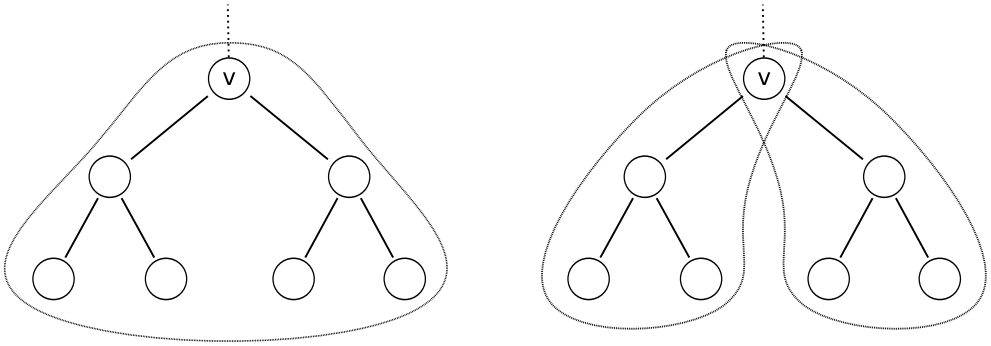

Figura 3.3: (a) subárvore completa de v, (b) subárvores parciais de v

O algoritmo que descreveremos faz uso do procedimento PESO-COMPONENTE (T, w, v), que recebe como entrada uma árvore T, a função peso w e um vértice v e calcula o peso da *componente inferior* de v. Esse procedimento garante apenas que a seguinte propriedade é satisfeita: se T_1 e T_2 são subárvores de T, com raiz v_1 e v_2, respectivamente, e se T_1 é uma subárvore de T_2, então PESO-COMPONENTE $(T_1, w, v_1) \leq$ PESO-COMPONENTE (T_2, w, v_2). Essa é a única propriedade do procedimento levada em consideração na prova do algoritmo (isto nos permite considerar diferentes funções-peso para o problema e obter resultados análogos).

PartiçãoDePerlSchach (G, w, q)

Entrada: Uma árvore G, uma função $w : V(G) \to \mathbb{Z}_+$ e um inteiro q.
Saída: uma q-partição conexa balanceada ótima de G

```
1  T ← AJUSTA-ÁRVORE(G, w)
2  Atribua todos os q − 1 cortes à única aresta da raiz r de T
3  repeat  CR ← −1
4          W_min ← peso de uma componente mais leve ▷ floresta corrente
5          for corte c' de T
6               do if c' pode ser deslocado para uma aresta α' sem cortes
7                     then p ← PESO-COMPONENTE(T, w, PONTA(α'))
8                          if p > CR
9                             then CR ← p, c ← c', α ← α'
10         if CR ≥ W_min
11            then faça o deslocamento do corte c para a aresta α
12    until CR < W_min
13 return a partição definida pelos q − 1 cortes
14 ▷ W_min é o peso de uma componente mais leve dessa partição
```

Uma simulação da execução do algoritmo PartiçãoDePerlSchach $(G, w, 4)$ é exibida na Figura 3.4, onde G é uma árvore com 6 vértices, no qual estão indicados os pesos definidos pela função w. Em cada iteração, os testes dos possíveis deslocamentos estão indicados em tamanho menor, no lado direito de cada árvore; dentre estes, as árvores representadas nas caixas cinzas são aquelas cujo deslocamento indicado foi o selecionado.

3.1.1 Análise do algoritmo

Primeiramente vamos provar que o algoritmo PartiçãoDePerlSchach de fato determina uma partição conexa balanceada ótima.

Vamos chamar de Q uma q-partição conexa balanceada ótima de T. Queremos comparar a partição devolvida pelo algoritmo com a partição ótima Q. Como o algoritmo atribui cortes às arestas de T, e vai obtendo partições conexas de T, vamos chamar de A uma partição de T obtida pelo algoritmo (numa iteração genérica) após realizar alguns deslocamentos. Na iteração em que A está definida, os cortes c_i's estão atribuídos às arestas de T. Referimos a tais arestas como **arestas que contêm A-cortes** ou às quais estão atribuídos A-cortes. Por analogia, as arestas de T que são Q-cortes também serão vistas como arestas que **contêm Q-cortes** ou arestas às quais estão atribuídos Q-cortes.

Para comparar duas partições de T, vamos considerar a relação de ordem parcial definida a seguir.

Seja v um vértice de T, e seja T' uma subárvore parcial de v. Para uma dada partição P, denotamos por $\mathbf{C(T', v, P)}$ o conjunto das arestas de T' que contêm P-cortes.

Dizemos que uma partição P está **acima** de uma partição P', e escrevemos $\mathbf{P \geq P'}$, se para cada vértice v de T temos que

$$|C(T', v, P)| \leq |C(T', v, P')| \text{ para cada subárvore parcial } T' \text{ de } v.$$

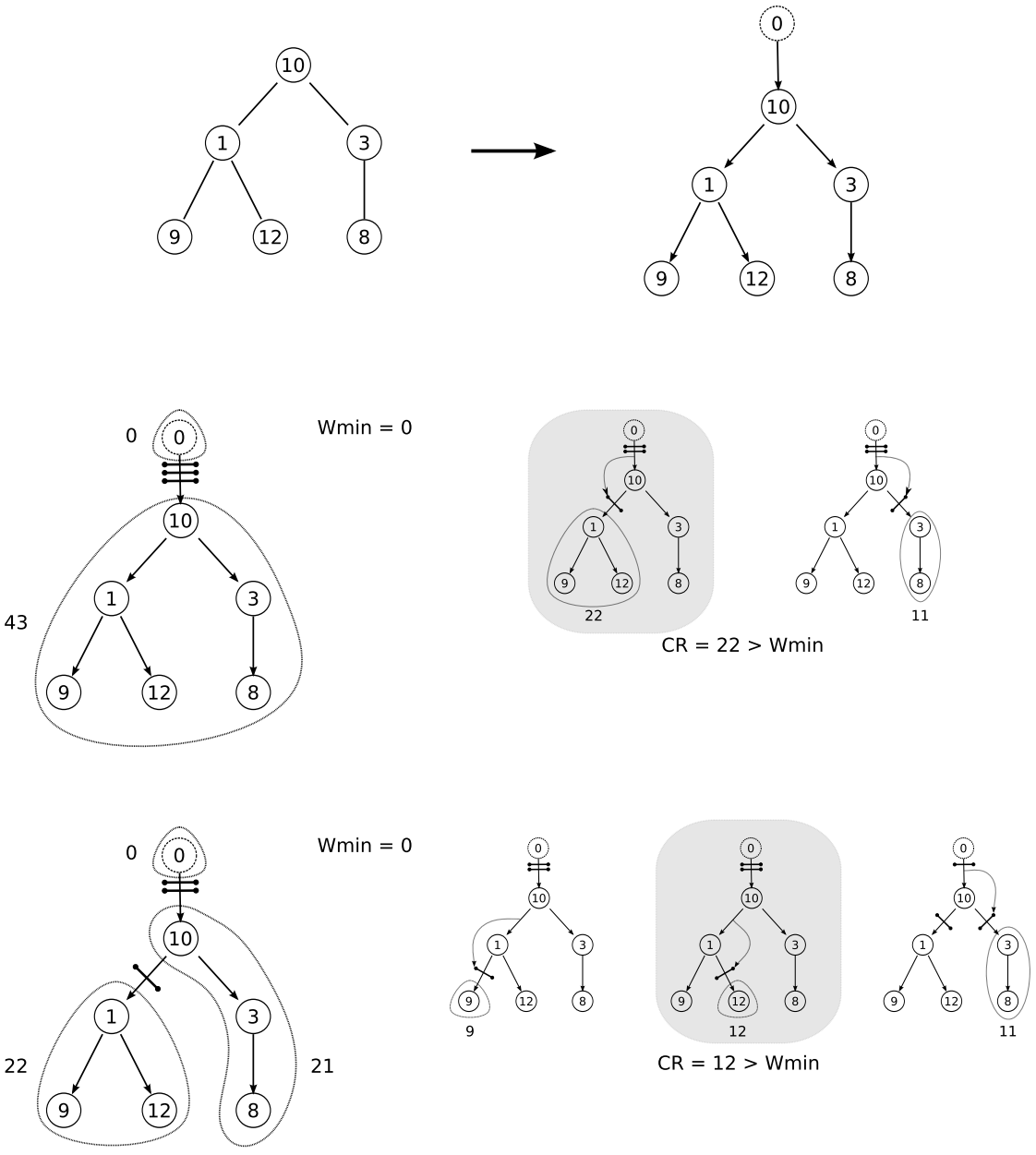

Figura 3.4: Simulação do algoritmo PartiçãoDePerlSchach $(G, w, 4)$ (continua)

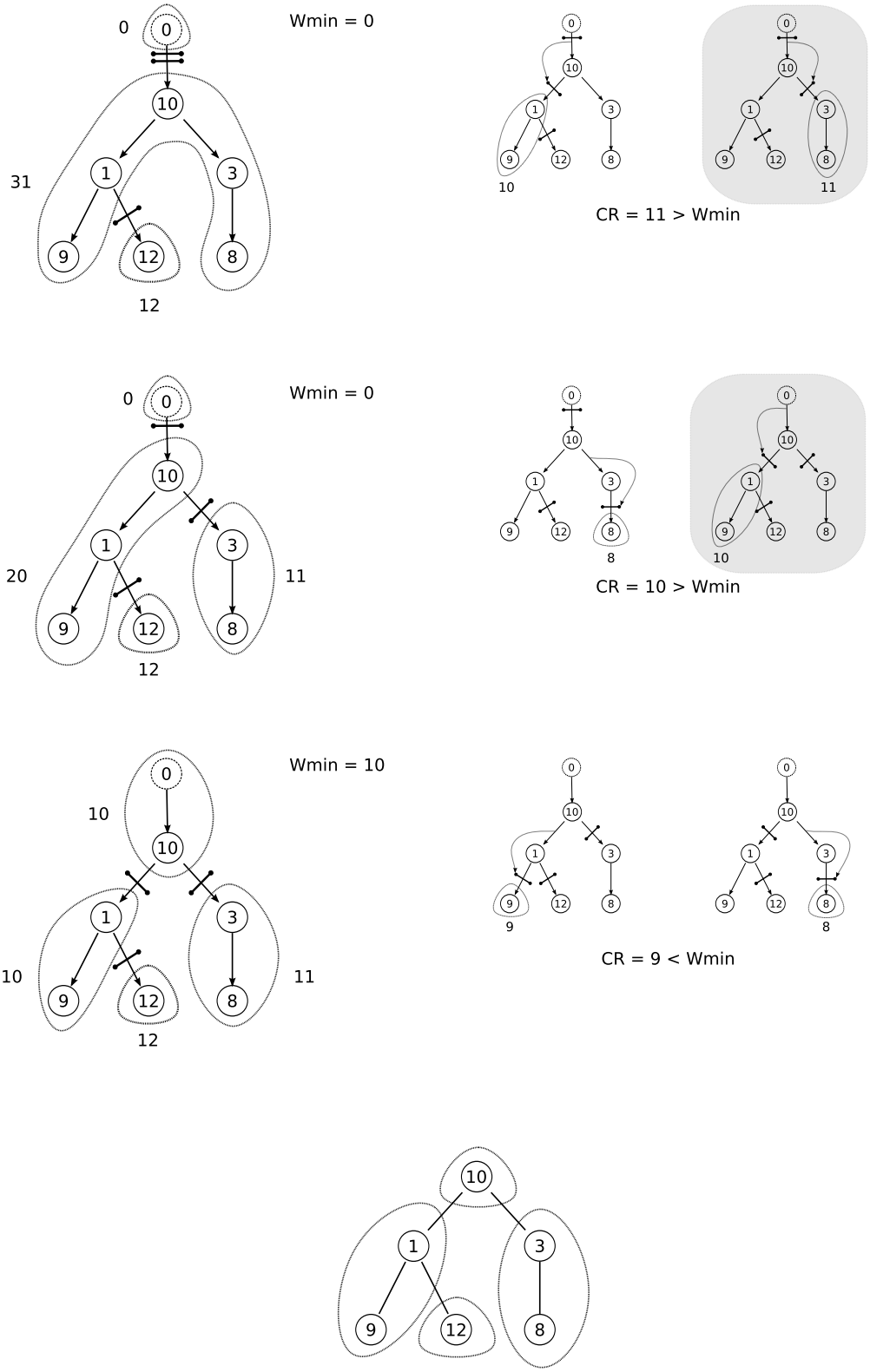

Figura 3.5: Continuação da simulação do algoritmo PartiçãoDePerlSchach $(G, w, 4)$

Se $P \geq P'$ e $P \neq P'$ então dizemos que P está **estritamente acima** de P', e escrevemos $\mathbf{P} > \mathbf{P'}$.

Também utilizamos a seguinte notação. Se P é uma q-partição de uma árvore T, o peso de uma componente mais leve de P é denotado por $\mathbf{W_{min}(T, P)}$ ou apenas $\mathbf{W_{min}(P)}$ quando a árvore em questão é óbvia pelo contexto. Da mesma forma, usamos $\mathbf{W(C)}$ como sendo o peso de uma componente C.

Os seguintes resultados relativos ao algoritmo PartiçãoDePerlSchach são a base para provarmos que ele está correto.

Nos dois lemas a seguir, A denota uma partição da árvore T obtida após realizar alguns deslocamentos, de acordo com o algoritmo PartiçãoDePerlSchach; e Q denota uma partição ótima.

Lema 3.1.1. *Se $A > Q$ o algoritmo* PartiçãoDePerlSchach *não pára, e sim efetua o deslocamento de um A-corte c. Neste caso, após o deslocamento de c, na nova partição que se obtém de A o peso da componente inferior de c é maior ou igual ao peso de uma componente mais leve da partição Q.*

Demonstração. Como $A > Q$, alguma subárvore parcial de T tem mais Q-cortes do que A-cortes. Então existe pelo menos uma aresta em T que contém um A-corte e não contém um Q-corte. Vamos chamar de c' esse A-corte mais distante da raiz (veja a Figura 3.6). Neste caso, na subárvore parcial de c' há pelo menos tantos Q-cortes quanto A-cortes (pela definição acima). O mesmo vale para a subárvore completa de $\text{PONTA}(c')$; logo, esta tem pelo menos uma aresta com apenas um Q-corte, digamos s'. Seja c'' o A-corte imediatamente acima do corte s' (primeiro encontrado a partir de s' caminhando na direção da raiz). Tal c'' existe, e pode ser que $c'' = c'$. O deslocamento do corte c'' para uma aresta no caminho entre c'' e s' resultaria numa componente inferior de c'' com peso maior ou igual ao da componente inferior de s', que por sua vez é maior ou igual a $W_{\min}(Q)$. Assim, se C é uma componente inferior resultante mais pesada, temos que $W(C) \geq W_{\min}(Q) \geq W_{\min}(A)$ (a última desigualdade segue da otimalidade de Q). Portanto, o algoritmo não pára, e efetua o deslocamento de um corte c, cuja componente inferior resultante tem peso maior ou igual ao peso de uma componente mais leve da partição Q. □

Lema 3.1.2. *Suponha que $A > Q$. Seja A' a partição que resulta de A após fazer o deslocamento de um corte. Então existe uma partição ótima Q' tal que $A' \geq Q'$.*

Demonstração. Como $A > Q$, pelo Lema 3.1.1 o algoritmo faz um deslocamento, digamos de um corte c. Suponha que c seja deslocado de uma aresta a_1 para uma aresta a_2. Vejamos como construir Q'. Para isso, considere os seguintes casos.

1. Caso 1 (Figura 3.7 (a)): Cada subárvore parcial T' de $\text{PONTA}(a_1)$ é tal que

$$|C(T', \text{PONTA}(a_1), A)| = |C(T', \text{PONTA}(a_1), Q)|. \tag{3.1}$$

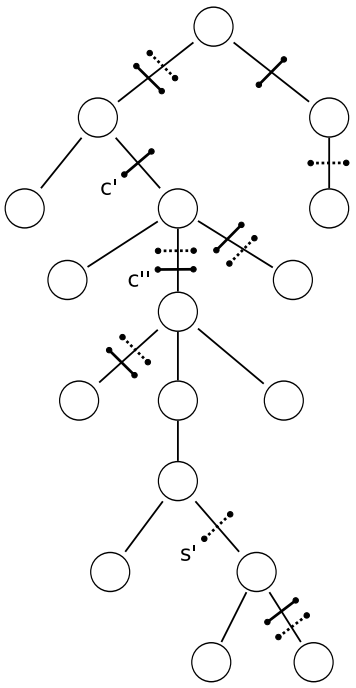

Figura 3.6: Situação descrita na prova do Lema 3.1.1. Os A-cortes estão indicados em linhas cheias e os Q-cortes em linhas pontilhadas.

Então após o deslocamento de c para a_2 a subárvore parcial de a_2 tem um A-corte a mais do que Q-cortes. Assim, $A' \leq Q$. Já cada subárvore parcial de $\text{PONTA}(a_1)$ tem o mesmo número de A-cortes e Q-cortes. Logo, a_1 deve conter um Q-corte. Pelo Lema 3.1.1, a componente inferior de a_2 em A' tem peso maior ou igual a $W_{min}(Q)$. Usando (3.1), o fato de que $A > Q$, segue que, após o deslocamento, a componente inferior de a_2 em Q satisfaz a mesma desigualdade. Fazemos então o deslocamento do Q-corte de a_1 para a_2, para gerar a partição Q'. Dessa forma Q' ainda é ótimo e $A' \geq Q'$.

2. Caso 2 (Figura 3.7 (b)): Existe uma subárvore parcial T' de $\text{PONTA}(a_1)$ tal que

$$|C(T', \text{PONTA}(a_1), A)| < |C(T', \text{PONTA}(a_1), Q)|. \tag{3.2}$$

Se na subárvore completa de $\text{PONTA}(a_2)$ o número de A-cortes é menor do que o número de Q-cortes, então $A' \geq Q$. Caso contrário, a subárvore completa de $\text{PONTA}(a_2)$ tem, pela definição de $A > Q$, o mesmo número de A-cortes e Q-cortes. Com isso, de (3.2) concluímos que existe uma aresta a_3 incidente em $\text{PONTA}(a_1)$ tal que a subárvore parcial enraizada em $\text{PONTA}(a_1)$ e cuja aresta inicial é a_3 tem mais Q-cortes do que A-cortes. Vamos chamar de $SP(\text{PONTA}(a_1), a_3)$ essa tal subárvore parcial. Agora seja c_1 o Q-corte mais próximo da raiz em $SP(\text{PONTA}(a_1), a_3)$.

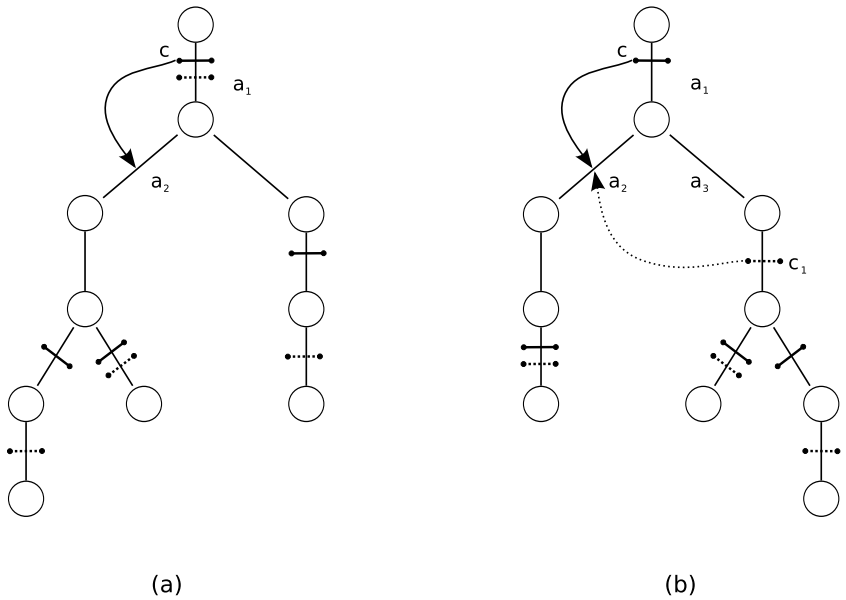

Figura 3.7: Casos 1 e 2 descritos na prova do Lema 3.1.2

Construímos a partição Q' reatribuindo c_1 à aresta a_2. Como no Caso 1, a componente inferior de a_2 em Q tem peso maior ou igual a $W_{min}(Q)$. A componente superior do corte c_1 em Q' também tem peso maior ou igual a $W_{min}(Q)$, pois ela contém a componente inferior de c_1 em Q, que por sua vez tem peso maior ou igual a $W_{min}(Q)$. Dessa forma, essa reatribuição resulta em uma partição Q' tal que $W_{min}(Q') \geq W_{min}(Q)$. Como a reatribuição afeta apenas o número de cortes na subárvore parcial de PONTA(a_1) e cuja aresta inicial é a_2 e $SP(\text{PONTA}(a_1), a_3)$, não é difícil verificar que a condição para ter $A' \geq Q'$ está satisfeita.

□

Usamos esses lemas para mostrar que as partições geradas durante a execução do algoritmo PartiçãoDePerlSchach estão sempre *estritamente acima* de uma partição ótima até o momento em que uma partição ótima é encontrada. Além disso, o algoritmo PartiçãoDePerlSchach não pára antes que isso ocorra. Assim:

1. No inicio do algoritmo PartiçãoDePerlSchach a partição A está *estritamente acima* de Q.

2. O Lema 3.1.1 mostra que enquanto A está *estritamente acima* da partição ótima Q o algoritmo não pára.

3. Usando (2) o Lema 3.1.2 mostra que se a partição A não é ótima o algoritmo PartiçãoDePerlSchach não devolve A (porque existe uma partição ótima Q' para a qual temos que $A \geq Q'$ e $A \neq Q'$; e portanto $A > Q'$).

4. Como o algoritmo pára após um número finito de passos, (3) implica que ele encontra uma partição ótima.

Teorema 3.1.1. *Seja A_f a partição devolvida pelo algoritmo* PartiçãoDePerlSchach. *Então A_f é uma partição ótima.*

Demonstração. A partição inicial do algoritmo PartiçãoDePerlSchach está *estritamente acima* de qualquer outra partição, e portanto, *acima* de uma partição ótima. Seja $A_1, \ldots, A_f$ a seqüência de partições obtidas durante a execução do algoritmo, sendo A_f a partição final devolvida. Agora suponha que nenhuma das partições A_i seja ótima. Pelo Lema 3.1.2, existe uma partição ótima Q_1 tal que $A_1 \geq Q_1$, e como A_1 não é uma partição ótima, temos que $A_1 > Q_1$. De maneira similar podemos encontrar partições ótimas $Q_2, \ldots, Q_f$, tais que $A_2 > Q_2, \ldots, A_f > Q_f$. Neste caso, o Lema 3.1.1 implica que o algoritmo PartiçãoDePerlSchach não pára devolvendo A_f, uma contradição. □

3.1.2 Complexidade computacional

Vamos analisar a seguir a complexidade do algoritmo PartiçãoDePerlSchach. Utilizaremos n para denotar o número de vértices da árvore de entrada. Pelo Lema 3.1.1, a cada iteração o algoritmo efetua um *deslocamento* de um corte. Vimos que um corte não é atribuído mais de uma vez a uma mesma aresta, pela própria definição de *deslocamento*. Assim, o número de iterações do algoritmo PartiçãoDePerlSchach é limitado por $O(q.n)$. Agora, em cada iteração todos os possíveis deslocamentos são analisados, e o peso de cada componente inferior é calculado. Esse último cálculo pode ser feito em tempo $O(n)$, e temos no máximo q *deslocamentos* possíveis em cada iteração. A cada iteração um *deslocamento* é executado e ao se efetuar um *deslocamento* precisamos apenas recalcular o peso da componente que foi afetada pelo deslocamento, o que pode ser feito em tempo limitado por $O(n)$. Assim, cada iteração requer no máximo $O(q.n)$ operações. Isso mostra que a complexidade do algoritmo PartiçãoDePerlSchach é no pior caso $O(q^2.n^2)$. Veremos a seguir que na prática o algoritmo PartiçãoDePerlSchach tem um desempenho melhor.

3.2 Resultados computacionais

O algoritmo descrito acima foi implementado em C++ [35] usando uma abordagem orientada a objetos. Desenvolvemos classes específicas para trabalhar com cortes e manipulação de arestas de uma maneira eficiente, visto que essas operações são bem exploradas pelo algoritmo PartiçãoDePerlSchach, e dado que tais recursos não estão facilmente disponíveis em bibliotecas de manipulação de grafos de uso geral.

Realizamos testes para garantir que a implementação está correta, bem como testes de carga para verificar o limite de consumo de memória e processamento da implementação do algoritmo PartiçãoDePerlSchach.

Os testes de desempenho foram baseados na execução do algoritmo em diversas árvores com tamanhos e pesos aleatórios. As árvores foram geradas segundo o algoritmo

ArvoreAleatória, que recebe como entrada um inteiro positivo n e retorna ao final de sua execução uma árvore com n vértices de pesos arbitrários (pertencentes a $\mathbb{Z}_+$), e arestas escolhidas de forma aleatória.

O algoritmo ArvoreAleatória faz uso de alguns procedimentos, aqui listados:

- número-aleatório ( ) devolve um número aleatório pertencente a $\mathbb{Z}_+$. Também usamos o procedimento numero-aleatório $(\min, \max)$, que devolve um número aleatório inteiro k tal que $\min \leq k \leq \max$.
- Union-Find-Init (n) inicializa uma estrutura com n conjuntos unitários, cada um deles com um elemento distinto no intervalo $1 \,..\, n$.
- Find (x) devolve a qual conjunto o elemento x pertence.
- Union (x, y) une os conjuntos aos quais x e y pertencem, de modo que ao final da operação Find (x) = Find (y).

As estruturas de dados e algoritmos dos procedimentos Union & Find utilizados pelo algoritmo ArvoreAleatória são problemas bem resolvidos e comumente estudados. O leitor pode encontrar informações mais detalhadas nos livro de Sedgewick [33] e Cormen et al. [14] entre outros.

ArvoreAleatória (n)

Entrada: um inteiro n
Saída: um par constituído por uma árvore (V, A)
com $|V| = n$ e uma função peso $w : V \to \mathbb{Z}_+$

```
1  V ← {1, ..., n}
2  A ← ∅
3  nArestas ← 0
4  for i ← 1 to n
5        do w(i ∈ V) ← número-aleatório ( )
6  Union-Find-Init (n)
7  while nArestas ≠ n − 1
8        do u ← número-aleatório (1, n)
9           v ← número-aleatório (1, n)
10          if Find (u) ≠ Find (v)
11             then  Union (u, v)
12                   A ← A ∪ {u, v}
13                   nArestas ← nArestas + 1
14 return ((V, A), w)
```

O procedimento para os testes foi o seguinte: geramos 50 árvores aleatórias usando o algoritmo ArvoreAleatória com tamanho variando de 100 a 2000 vértices, com intervalo de 100. Ou seja, geramos 50 árvores aleatórias de tamanho 100, outras 50 de tamanho 200, e assim por diante até gerar 50 árvores com 2000 vértices, num total de

1000 árvores aleatórias distintas. Para cada uma dessas árvores executamos o algoritmo PartiçãoDePerlSchach para obter partições com $2, 4, 8, 16, 32$ e 64 componentes.

Para cada um dos grupos de 50 árvores calculamos a média de tempo (em segundos) que o algoritmo PartiçãoDePerlSchach levou para gerar cada uma das partições. Com esses resultados obtivemos o gráfico da Figura 3.8.

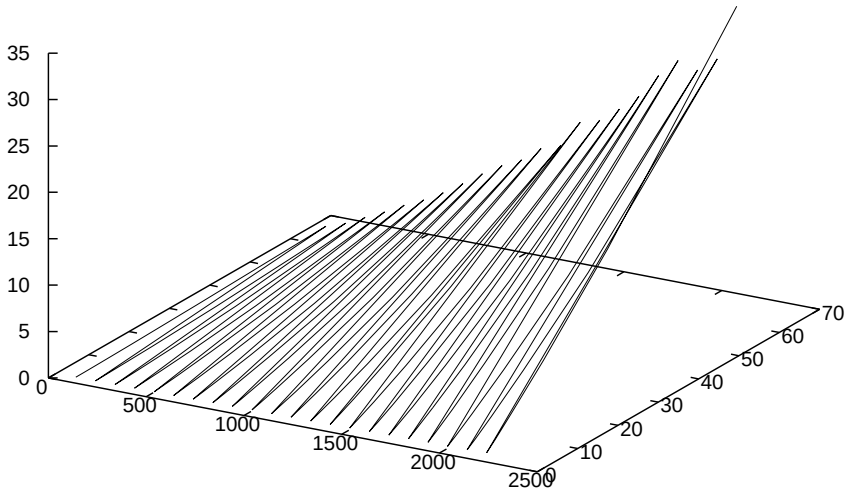

Figura 3.8: Resultados do desempenho do algoritmo PartiçãoDePerlSchach

Este gráfico mostra a relação tempo × tamanho da árvore × número de componentes da partição. Podemos constatar pela curva do gráfico que o desempenho do algoritmo PartiçãoDePerlSchach tem um comportamento inferior ao quadrático, dado pela análise de complexidade do algoritmo.

Além desses testes de desempenho, executamos testes para descobrir os limites de consumo de memória da nossa implementação, particionando árvores grandes, com milhares de vértices, chegando a 100000. A representação visual da partição de uma árvore relativamente grande pode ser vista na parte inferior da Figura 3.9. Na parte superior dessa figura exibimos essa mesma árvore na forma hierárquica enraizada.

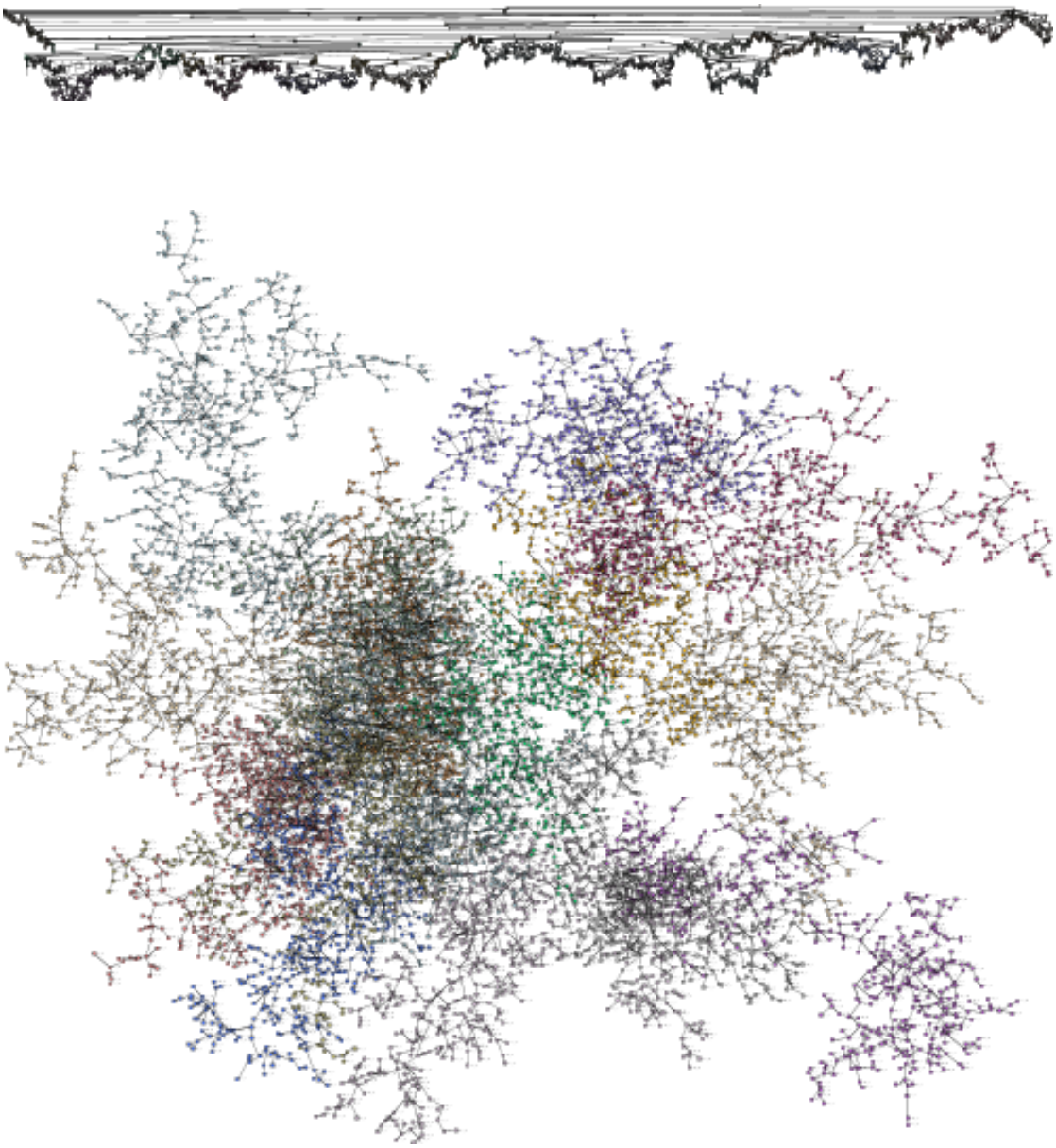

Figura 3.9: Uma 16-partição de uma árvore com 10000 vértices

Capítulo 4

Heurísticas baseadas em árvores geradoras

Neste capítulo vamos apresentar algumas heurísticas para o PCB baseadas na seguinte idéia: obter (algumas) árvores geradoras do grafo de entrada, usar o algoritmo PartiçãoDePerlSchach para árvores geradoras, e escolher a melhor solução encontrada. As heurísticas diferem na maneira em que as árvores são geradas: se árvores anteriores são usadas para gerar outras (e como isso é feito), ou se estas são geradas independentemente.

4.1 Introdução

Os algoritmos descritos nesse capítulo utilizam alguns procedimentos que foram previamente discutidos no Capítulo 3 (número-aleatório, Union-Find-Init, Find e Union). Utilizamos também o procedimento medida-partição (Q, w) que recebe como parâmetro uma partição Q de um grafo e uma função peso w definida sobre seus vértices e devolve a soma dos pesos de uma componente mais leve da partição Q.

Lembramos que os símbolos n e m denotam o número de vértices e o número de arestas do grafo em consideração. Assim, nas análises de complexidade dos algoritmos, referem-se ao grafo de entrada do algoritmo em análise.

Uma premissa para desenvolver heurísticas para um problema é conseguir avaliar os resultados encontrados por essas heurísticas. Para o PCB, implementamos um procedimento de construção de grafos de maneira que sabemos *a priori* o valor de uma solução ótima para o mesmo, e com isso conseguimos mensurar a qualidade da solução encontrada por uma heurística.

O procedimento que implementamos constrói um grafo de maneira que ele seja composto por q árvores disjuntas (nos vértices) de peso igual, digamos p. Dessa forma um tal grafo tem peso total $q.p$ e possui uma q-partição onde todas as componentes têm peso p. Geramos q árvores com quantidades de vértices variável. Para gerar tais árvores implementamos um procedimento que constrói uma árvore arbitrária com pesos atribuídos a seus vértices de forma aleatória e tal que a soma desses pesos seja algum

valor pré-estabelecido. Após gerar q árvores com um tal procedimento, usamos um outro procedimento que acrescenta arestas às essas q-árvores de forma aleatória, de modo que o grafo resultante seja conexo (tanto arestas com ambas as pontas numa mesma árvore, como arestas com pontas em árvores distintas são geradas).

O algoritmo ArvoreAleatóriaComPeso descrito a seguir recebe dois números inteiros como entrada, n e *peso*, sendo que obrigatoriamente $peso \geq n$ (pois cada vértice deve ter peso pelo menos 1). Ele devolve uma árvore gerada de forma aleatória e uma função peso definida sobre os vértices dessa árvore de forma que a soma dos pesos de todos os vértices seja exatamente igual a *peso*.

ArvoreAleatóriaComPeso $(n, peso)$

Entrada: dois inteiros, n e *peso* ($peso \geq n$).
Saída: um par constituído por uma árvore (V, A) arbitrária com $|V| = n$ e uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices da árvore sendo que $\sum_{v \in V} w(v) = peso$.

```
 1  V ← {1, ..., n}
 2  A ← ∅
 3  nArestas ← 0
 4  Union-Find-Init (n)
 5  while nArestas ≠ n − 1
 6        do u ← número-aleatório (1, n)
 7           v ← número-aleatório (1, n)
 8           if Find (u) ≠ Find (v)
 9              then  Union (u, v)
10                    A ← A ∪ {u, v}
11                    nArestas ← nArestas + 1
12  for i ← 1 to n
13        do w(i ∈ V) ← 1
14  pesoRestante ← peso − n
15  while pesoRestante > 0
16        do v ← número-aleatório (1, n)
17           w(v ∈ V) ← w(v ∈ V) + 1
18           pesoRestante ← pesoRestante − 1
19  return ((V, A), w)
```

A construção do grafo é feita pelo por GrafoAleatórioParaQPartição, um algoritmo que recebe como entrada três inteiros:

- n, o tamanho do grafo a ser gerado,
- *densidade*, o número de arestas que o grafo deve ter, e
- q, quantas componentes deve ter a partição conexa.

Esse algoritmo faz uso do procedimento ArvoreAleatóriaComPeso para gerar q árvores aleatórias e as une utilizando uma quantidade de arestas definida pelo procedimento arestas-pela-densidade.

arestas-pela-densidade $(n, densidade)$

Entrada: dois inteiros: n e *densidade*.
Saída: total de arestas que um grafo com n vértices tem segundo a densidade dada em porcentagem (100 indica o grafo completo).

1 **return** $\frac{1}{100}(\frac{n.(n-1)}{2}.densidade)$

GrafoAleatórioParaQPartição $(n, densidade, q)$

Entrada: três inteiros, n, *densidade* e q
Saída: uma tripla composta por um grafo (V, A) com $|V| = n$ e com o total de arestas de acordo com a *densidade*, uma função peso w definida sobre os vértices do grafo e um inteiro que é o valor ótimo de uma q-partição desse grafo.

1 ▷ Sorteamos o peso único que cada componente da q-partição terá
2 $otimo \leftarrow$ número-aleatório $(n, 10.n)$
3 ▷ Sorteamos também quantos vértices devem ter cada componente da q-partição
4 $tamanhos[1\,..\,q] \leftarrow \langle 1, \ldots, 1\rangle$ ▷ Valor inicial de 1 para cada componente
5 $tamanhoTotal \leftarrow n - q$
6 **while** $tamanhoTotal > 0$
7 **do** $t \leftarrow$ número-aleatório $(1, q)$
8 $tamanhos[t] \leftarrow tamanhos[t] + 1$
9 $tamanhoTotal \leftarrow tamanhoTotal - 1$
10 ▷ Usamos um *array* temporário para armazenar todas as q árvores geradas
11 $arvores[1\,..\,q] \leftarrow \langle((\emptyset, \emptyset), w), \ldots, ((\emptyset, \emptyset), w)\rangle$
12 **for** $i \leftarrow 1$ **to** q
13 **do** $arvores[i] \leftarrow$ ArvoreAleatóriaComPeso $(tamanhos[i], otimo)$
14 $w \leftarrow \bigcup_{i=1}^{q} arvores[i](w)$
15 $V \leftarrow \bigcup_{i=1}^{q} arvores[i](V)$
16 $A \leftarrow \bigcup_{i=1}^{q} arvores[i](A)$
17 **Ligamos cada uma das q árvores com $q - 1$ arestas e acrescentamos em A**
18 ▷ Depois adicionamos as arestas restantes de forma aleatória
19 $nArestas \leftarrow |A|$
20 $totalArestas \leftarrow$ arestas-pela-densidade $(n, densidade)$
21 **while** $nArestas < totalArestas$
22 **do** $u \leftarrow$ número-aleatório $(1, n)$
23 $v \leftarrow$ número-aleatório $(1, n)$
24 **if** $\{u, v\} \notin A$
25 **then** $A \leftarrow A \cup \{u, v\}$
26 $nArestas \leftarrow nArestas + 1$
27 **return** $((V, A), w, otimo)$

Utilizando o algoritmo GrafoAleatórioParaQPartição obtemos grafos cuja solução ótima é conhecida; dessa forma, podemos definir a qualidade de uma partição Q (dada a função peso w) como sendo:

$$Qualidade(Q, w) = \frac{\text{MEDIDA-PARTIÇÃO}(Q, w)}{valor\ ótimo.}$$

Lembramos que MEDIDA-PARTIÇÃO(Q, w) é o procedimento que devolve a medida (ou valor) da partição Q, definida como o peso da componente mais leve dessa partição. Quando $Q = \emptyset$, definimos a medida de Q como sendo ∞.

Nas seções seguintes apresentamos algumas heurísticas para o PCB.

4.2 Árvores geradoras aleatórias

A primeira heurística que vamos apresentar se baseia numa idéia bem simples: determinar algumas árvores geradoras aleatórias[1] do grafo de entrada, calcular uma partição conexa ótima dessas árvores utilizando o algoritmo PartiçãoDePerlSchach e devolver a melhor solução encontrada. Essa heurística utiliza ArvoreGeradoraAleatória, um procedimento que, dado um grafo de entrada, computa uma árvore geradora aleatória.

ArvoreGeradoraAleatória (G)

Entrada: grafo conexo G.
Saída: uma árvore geradora arbitrária de G.

```
1  V ← V(G)
2  A ← ∅
3  nArestas ← 0
4  Union-Find-Init (n)
5  while nArestas ≠ |V| − 1
6        do α ← número-aleatório (1, |A(G)|)
7           {u, v} ← α−ésimo elemento do conjunto A(G)
8           if Find (u) ≠ Find (v)
9              then Union (u, v)
10                  A ← A ∪ {u, v}
11                  nArestas ← nArestas + 1
12 return (V, A)
```

A seguir descrevemos o procedimento principal da heurística.

[1] Ressaltamos que todos os procedimentos que fazem uso de números aleatórios foram implementadas de maneira a gerar uma *semente* nova a cada execução

HeurísticaArvoresAleatórias $(G, w, q, tentativas)$

Entrada: grafo conexo G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G, um inteiro q e um inteiro *tentativas*.
Saída: uma q-partição conexa de G.

```
1  melhor ← ∅
2  for t ← 1 to tentativas
3        do T ← ArvoreGeradoraAleatória (G)
4           P ← PartiçãoDePerlSchach (T, w, q)
5           if medida-partição (melhor, w) < medida-partição (P, w)
6              then melhor ← P
7  return melhor
```

Fizemos testes com essa heurística com o valor de *tentativas* sempre polinomial no tamanho da entrada (n ou n^2). Esses testes estão descritos na seção a seguir.

4.2.1 Testes: HeurísticaArvoresAleatórias

Dividimos os testes da seguinte maneira: agrupamos os grafos por tamanho, digamos n. Para cada tamanho geramos grafos com três faixas de densidade (30%, 60% e 90%), e executamos a heurística quatro vezes, usando quatro valores diferentes para q (2, $n/4$, $n/2$ e $3n/4$). Usamos o procedimento GrafoAleatórioParaQPartição para gerar 20 grafos aleatórios de cada um dos tamanhos e densidades. Por fim calculamos a média da *Qualidade* obtida em cada uma das execuções.

A Tabela 4.1 mostra o resultado dos testes para grafos que variam de tamanho entre 10 e 70 vértices, em que a HeurísticaArvoresAleatórias foi executada com o parâmetro $tentativas = n$. Com essa escolha o algoritmo fica com complexidade $O(q^2.n^3)$, pois obtém n árvores aleatórias de G e chama o algoritmo PartiçãoDePerlSchach para cada uma delas (lembramos que esse algoritmo tem complexidade $O(q^2.n^2)$).

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.993	0.980	0.998	1.000	0.982	0.929	0.922	0.955	0.982	0.967	0.877	0.869
20	0.994	0.903	0.789	0.705	0.995	0.904	0.803	0.609	0.993	0.904	0.799	0.664
30	0.995	0.858	0.766	0.588	0.997	0.859	0.774	0.586	0.995	0.858	0.774	0.572
40	0.997	0.857	0.746	0.611	0.999	0.861	0.759	0.563	0.997	0.855	0.748	0.555
50	1.000	0.829	0.730	0.538	0.999	0.830	0.724	0.537	0.999	0.834	0.734	0.533
60	1.000	0.829	0.719	0.536	0.998	0.832	0.715	0.536	0.998	0.820	0.728	0.532
70	0.999	0.810	0.707	0.538	0.999	0.810	0.730	0.524	0.999	0.811	0.712	0.532

Tabela 4.1: Testes da HeurísticaArvoresAleatórias com $tentativas = n$

Na Tabela 4.2 temos o resultado da execução da mesma bateria de testes, só que dessa vez o parâmetro *tentativas* é n^2. Com essa escolha a complexidade do algoritmo é $O(q^2.n^4)$.

Junto com esses testes, além da média calculamos também o desvio padrão da qualidade obtida. A Tabela 4.3 mostra este resultado.

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	1.000	1.000	1.000	1.000	1.000	0.992	0.981	1.000	0.999	0.988	0.981	1.000
20	1.000	0.962	0.900	0.966	1.000	0.957	0.900	0.909	1.000	0.963	0.880	0.914
30	1.000	0.920	0.854	0.800	1.000	0.919	0.842	0.828	1.000	0.918	0.853	0.736
40	1.000	0.914	0.823	0.737	1.000	0.911	0.827	0.718	1.000	0.910	0.825	0.728
50	1.000	0.891	0.809	0.658	1.000	0.891	0.806	0.660	1.000	0.895	0.806	0.628
60	1.000	0.881	0.787	0.618	1.000	0.883	0.798	0.589	1.000	0.884	0.792	0.605
70	1.000	0.869	0.787	0.565	1.000	0.868	0.786	0.572	1.000	0.868	0.787	0.603

Tabela 4.2: Testes da HeurísticaArvoresAleatórias com $tentativas = n^2$

tent.	Tamanho	30%				60%				90%			
	do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
n	10	0.002	0.005	0.007	0.000	0.002	0.023	0.020	0.088	0.000	0.007	0.005	0.009
	30	0.000	0.005	0.010	0.010	0.001	0.007	0.003	0.001	0.000	0.002	0.010	0.001
	50	0.000	0.000	0.003	0.004	0.000	0.001	0.006	0.002	0.000	0.004	0.003	0.005
	70	0.000	0.004	0.009	0.004	0.000	0.001	0.000	0.001	0.000	0.001	0.017	0.004
n^2	10	0.000	0.000	0.000	0.000	0.000	0.003	0.014	0.000	0.002	0.003	0.004	0.000
	30	0.000	0.003	0.001	0.020	0.000	0.003	0.011	0.002	0.000	0.004	0.004	0.031
	50	0.000	0.000	0.004	0.030	0.000	0.003	0.002	0.025	0.000	0.001	0.006	0.042
	70	0.000	0.004	0.001	0.028	0.000	0.001	0.005	0.010	0.000	0.001	0.004	0.017

Tabela 4.3: Desvio padrão da HeurísticaArvoresAleatórias

4.3 Heurísticas que fazem uso de circuitos fundamentais

A seguir descreveremos duas heurísticas que trabalham com a idéia de melhorar a solução inicial obtida por uma árvore geradora arbitrária. Basicamente, a idéia consiste em acrescentar uma aresta que não esteja na árvore corrente, gerando dessa forma um circuito fundamental; depois, tal circuito é destruído (removendo-se uma aresta) resultando uma outra árvore geradora. Este processo é repetido, guardando-se a melhor solução obtida pelo algoritmo PartiçãoDePerlSchach.

4.3.1 HeurísticaCircuitoSimples

A idéia dessa primeira heurística que faz uso de circuitos fundamentais é simples. Dada uma árvore geradora do grafo calculamos uma partição ótima com o algoritmo PartiçãoDePerlSchach. Depois tentamos "conectar" duas das componentes obtidas, usando uma aresta do grafo de entrada que não pertence à árvore corrente. As duas componentes escolhidas são: uma de menor peso e uma de maior peso. Removemos uma aresta arbitrária do circuito formado (qualquer uma diferente da que acabamos de adicionar). Calculamos uma partição ótima para a nova árvore e continuamos nesse processo enquanto a solução melhora. A idéia de unir uma componente de menor peso com uma de maior peso é o de dar uma chance ao algoritmo PartiçãoDePerlSchach de calcular uma partição em que o peso da menor componente seja maior.

A HeurísticaMelhoraÁrvore, que será descrita a seguir, faz uso do procedimento ÁrvoreUneComponente que recebe um grafo, uma árvore geradora dele, um inteiro q e uma q-partição conexa desse grafo, e devolve uma nova árvore. A árvore devolvida é a que resulta após conectar uma componente mais leve com uma mais pesada, formando

um circuito fundamental, e removendo-se uma aresta do circuito formado.

ÁrvoreUneComponente (G, T, w, P, q)

Entrada: grafo conexo G, uma árvore geradora T de G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G, uma q-partição P de G e um inteiro q.
Saída: uma árvore geradora de G.

```
1  V ← V(T)
2  A ← A(T)
3  ▷ Devolve as componentes da partição em um vetor ordenado
4  componentes[1 . . q] ← ORDENA-PARTIÇÃO (P)
5  for c ← 1 to q
6        do
7           for d ← q downto 1
8                 do if Existe aresta α ∈ A(G) com uma extremidade
9                    em componentes[c] e outra em componentes[d] e α ∉ A(T)
10                      then  A ← A ∪ α
11                            β ← aresta arbitrária do circuito de (V, A)
12                            A ← A\β
13                            return (V, A)
14 return (V, A)
```

HeurísticaMelhoraÁrvore (G, T, w, q)

Entrada: grafo conexo G, uma árvore geradora T de G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G e um inteiro q.
Saída: uma q-partição conexa de G.

```
1  melhor ← PartiçãoDePerlSchach (T, w, q)
2  for t ← 1 to |V(G)|
3        do T ← ÁrvoreUneComponente (G, T, w, melhor, q)
4           P ← PartiçãoDePerlSchach (T, w, q)
5           if MEDIDA-PARTIÇÃO (melhor, w) < MEDIDA-PARTIÇÃO (P, w)
6              then  melhor ← P
7              else  return melhor
8  return melhor
```

O procedimento ÁrvoreUneComponente tem complexidade $O(q^2.m)$, pois no pior caso precisamos procurar em todos os pares possíveis de componentes, e em cada busca podemos levar até $O(m)$ para encontrar uma aresta que pode ser utilizada para conectar dois componentes em T. Assim, como HeurísticaMelhoraÁrvore pode executar até n vezes os procedimentos ÁrvoreUneComponente e PartiçãoDePerlSchach, temos que sua complexidade é $O(q^2.(n^3 + m.n))$.

Utilizando procedimento HeurísticaMelhoraÁrvore podemos escrever a heurística como segue.

HeurísticaCircuitoSimples (G, w, q)

Entrada: grafo conexo G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G e um inteiro q.

Saída: uma q-partição conexa de G.

1 $T \leftarrow$ ArvoreGeradoraAleatória (G)

2 **return** HeurísticaMelhoraÁrvore (G, T, w, q)

A HeurísticaCircuitoSimples é bem simples, executando dois procedimentos apenas uma vez. Claramente sua complexidade está limitada pela complexidade da HeurísticaMelhoraÁrvore, que é $O(q^2.(n^3 + m.n))$.

Testes: HeurísticaCircuitoSimples

Realizamos testes com a HeurísticaCircuitoSimples da mesma forma que testamos a HeurísticaArvoresAleatórias. Os resultados estão na Tabela 4.4.

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.876	0.860	0.858	0.932	0.824	0.818	0.742	0.795	0.887	0.785	0.733	0.755
20	0.866	0.759	0.655	0.640	0.869	0.798	0.719	0.616	0.883	0.793	0.674	0.586
30	0.878	0.783	0.648	0.569	0.855	0.781	0.635	0.599	0.895	0.736	0.664	0.547
40	0.878	0.766	0.633	0.540	0.873	0.763	0.643	0.544	0.888	0.754	0.660	0.520
50	0.855	0.756	0.625	0.531	0.903	0.729	0.614	0.515	0.879	0.712	0.634	0.530
60	0.872	0.761	0.588	0.522	0.915	0.751	0.618	0.517	0.875	0.756	0.618	0.528
70	0.899	0.747	0.619	0.513	0.912	0.755	0.606	0.509	0.863	0.737	0.563	0.524

Tabela 4.4: Testes da HeurísticaCircuitoSimples

Da mesma forma, verificamos o desvio padrão da qualidade das soluções encontradas pela heurística. Os resultados podem ser vistos na Tabela 4.5

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.015	0.014	0.044	0.015	0.019	0.020	0.020	0.052	0.042	0.015	0.022	0.032
30	0.012	0.013	0.006	0.008	0.006	0.015	0.020	0.040	0.003	0.014	0.018	0.005
50	0.020	0.000	0.015	0.006	0.020	0.003	0.005	0.003	0.003	0.001	0.009	0.016
70	0.018	0.012	0.009	0.001	0.009	0.011	0.011	0.002	0.005	0.005	0.007	0.001

Tabela 4.5: Desvio padrão da HeurísticaCircuitoSimples

4.3.2 HeurísticaCircuitoOnerosa

A heurística que descrevemos nesta seção é mais onerosa que a anterior. Dada uma árvore geradora aleatória T, essa heurística busca a melhor solução testando todas as árvores geradoras que estão a distância 1 de T. Dizemos que uma árvore T' está a distância 1 de T se $|A(T) \triangle A(T')| = 2$, ou seja, T' pode ser obtida de T pelo processo de

acrescentar uma nova aresta, digamos α, e remover uma aresta β do circuito fundamental em $T + \{\alpha\}$.

A descrição dessa heurística é dada a seguir.

HeurísticaCircuitoOnerosa (G, w, q)

Entrada: grafo conexo G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G e um inteiro q.
Saída: uma q-partição conexa de G.

```
1  T ← ArvoreGeradoraAleatória (G)
2  melhor ← PartiçãoDePerlSchach (T, w, q)
3  foreach aresta α ∈ A(G) e α ∉ A(T)
4        do A(T) ← A(T) ∪ α
5           foreach aresta β no circuito de T
6                 do A(T) ← A(T)\β
7                    P ← PartiçãoDePerlSchach (T, w, q)
8                    if medida-partição (melhor, w) < medida-partição (P, w)
9                       then melhor ← P
10                   A(T) ← A(T) ∪ β
11          A(T) ← A(T)\α
12 return melhor
```

Vimos que a HeurísticaCircuitoOnerosa usa cada uma das arestas de G que não pertence à árvore geradora aleatória original para formar um circuito fundamental. Depois, remove uma a uma as arestas desses circuitos, formando novas arvores geradoras. Esse procedimento tem complexidade $O(m.n)$. Para cada nova árvore gerada o algoritmo PartiçãoDePerlSchach é executado. Com isso, temos que a complexidade de HeurísticaCircuitoOnerosa é da ordem de $O(q^2.n^3.m)$.

Testes: HeurísticaCircuitoOnerosa

Os testes realizados com a HeurísticaCircuitoOnerosa seguiu o padrão dos testes anteriores. As Tabelas 4.6 e 4.7 apresentam os resultados sobre a qualidade e o desvio padrão.

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.991	0.943	0.919	0.975	0.990	0.941	0.863	0.949	0.984	0.930	0.875	0.901
20	0.992	0.885	0.696	0.663	0.990	0.873	0.749	0.620	0.991	0.902	0.764	0.646
30	0.992	0.852	0.668	0.585	0.995	0.810	0.690	0.564	0.996	0.822	0.687	0.576
40	0.997	0.794	0.619	0.542	0.996	0.773	0.680	0.525	0.997	0.797	0.664	0.539
50	0.998	0.770	0.594	0.519	0.996	0.769	0.616	0.527	0.997	0.759	0.623	0.521
60	0.998	0.745	0.642	0.523	0.998	0.740	0.607	0.519	0.997	0.782	0.626	0.521
70	0.998	0.737	0.613	0.517	0.998	0.734	0.608	0.513	0.999	0.731	0.619	0.523

Tabela 4.6: Testes da HeurísticaCircuitoOnerosa

Tamanho do grafo	30%				60%				90%			
	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.002	0.010	0.018	0.006	0.001	0.013	0.044	0.011	0.004	0.016	0.006	0.016
30	0.008	0.007	0.019	0.010	0.001	0.008	0.031	0.010	0.001	0.013	0.017	0.045
50	0.001	0.010	0.016	0.001	0.001	0.006	0.001	0.004	0.003	0.007	0.037	0.006
70	0.000	0.006	0.027	0.001	0.000	0.019	0.009	0.001	0.001	0.003	0.021	0.002

Tabela 4.7: Desvio padrão da HeurísticaCircuitoOnerosa

4.4 Junção de duas heurísticas

Na heurística descrita nesta seção usamos as duas idéias anteriores simultaneamente, gerando um número polinomial de árvores geradoras aleatórias e para cada uma delas executando uma heurística que faz uso de circuitos fundamentais. Escolhemos a heurística HeurísticaCircuitoSimples por ser a mais eficiente computacionalmente, usando seu procedimento principal HeurísticaMelhoraÁrvore. A descrição dessa heurística é a seguinte.

HeurísticaMista $(G, w, q, tentativas)$

Entrada: grafo conexo G, uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices de G, um inteiro q e um inteiro *tentativas*.
Saída: uma q-partição conexa de G.

```
1  melhor ← ∅
2  for t ← 1 to tentativas
3        do T ← ArvoreGeradoraAleatória (G)
4           P ← HeurísticaMelhoraÁrvore (G, T, w, q)
5           if medida-partição(melhor, w) < medida-partição(P, w)
6              then melhor ← P
7  return melhor
```

A complexidade da HeurísticaMista é $O(tentativas.q^2.(m^3 + m.n))$. Fizemos testes com $tentativas = n$ e $tentativas = n^2$. Com essas escolhas, a complexidade resultante é $O(q^2.(n^4 + m.n^2))$ e $O(q^2.(n^5 + m.n^3))$, respectivamente.

4.4.1 Testes: HeurísticaMista

Os testes da HeurísticaMista foram realizados da mesma forma que os testes feitos com a HeurísticaArvoresAleatórias, com n e n^2 como valores para *tentativas*. Os resultados dos testes de desempenho estão nas Tabelas 4.8 e 4.9. Já a Tabela 4.10 exibe os resultados sobre o desvio padrão.

4.5 Comparação do desempenho das heurísticas

Todas as tabelas com os resultados de qualidade e desvio padrão apresentadas neste capítulo contemplam a execução das heurísticas com a mesma massa de dados.

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	0.994	0.987	0.974	1.000	0.978	0.944	0.914	0.997	0.992	0.949	0.906	0.931
20	0.991	0.922	0.806	0.756	0.993	0.913	0.814	0.764	0.988	0.923	0.848	0.819
30	0.997	0.874	0.800	0.707	0.995	0.881	0.787	0.697	0.997	0.883	0.808	0.648
40	0.997	0.869	0.771	0.640	0.998	0.876	0.769	0.653	0.997	0.865	0.774	0.655
50	0.998	0.850	0.756	0.583	0.998	0.846	0.766	0.582	0.999	0.853	0.757	0.561
60	0.999	0.838	0.739	0.565	0.998	0.847	0.754	0.562	0.998	0.852	0.736	0.557
70	0.998	0.827	0.740	0.539	0.998	0.829	0.748	0.585	0.998	0.834	0.734	0.538

Tabela 4.8: Testes da HeurísticaMista com *tentativas* $= n$

Tamanho	30%				60%				90%			
do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
10	1.000	1.000	1.000	1.000	1.000	0.994	0.987	1.000	1.000	0.991	0.979	1.000
20	1.000	0.966	0.913	0.989	1.000	0.964	0.895	0.898	1.000	0.968	0.889	0.927
30	1.000	0.925	0.867	0.872	1.000	0.930	0.853	0.858	1.000	0.935	0.861	0.869
40	1.000	0.920	0.823	0.776	1.000	0.919	0.840	0.752	1.000	0.916	0.835	0.764
50	1.000	0.899	0.826	0.769	1.000	0.896	0.816	0.701	1.000	0.897	0.816	0.696
60	1.000	0.890	0.810	0.641	1.000	0.887	0.804	0.689	1.000	0.893	0.805	0.719
70	1.000	0.872	0.794	0.650	1.000	0.876	0.799	0.645	1.000	0.875	0.803	0.710

Tabela 4.9: Testes da HeurísticaMista com *tentativas* $= n^2$

tent.	Tamanho	30%				60%				90%			
	do grafo	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$	2	$n/4$	$n/2$	$3n/4$
n	10	0.001	0.003	0.006	0.000	0.002	0.026	0.018	0.001	0.002	0.011	0.018	0.015
	30	0.002	0.005	0.003	0.019	0.000	0.009	0.003	0.023	0.001	0.006	0.004	0.018
	50	0.001	0.004	0.015	0.013	0.000	0.006	0.012	0.013	0.000	0.003	0.004	0.011
	70	0.000	0.006	0.001	0.001	0.002	0.002	0.005	0.014	0.000	0.001	0.010	0.007
n^2	10	0.000	0.000	0.000	0.000	0.000	0.001	0.003	0.000	0.000	0.002	0.001	0.000
	30	0.000	0.005	0.002	0.004	0.000	0.003	0.006	0.013	0.000	0.000	0.006	0.015
	50	0.000	0.002	0.003	0.018	0.000	0.000	0.004	0.029	0.000	0.001	0.001	0.015
	70	0.000	0.002	0.008	0.012	0.000	0.003	0.001	0.022	0.000	0.001	0.004	0.008

Tabela 4.10: Desvio padrão da HeurísticaMista

Na Tabela 4.11 temos um quadro geral das heurísticas apresentadas neste capítulo, mostrando a complexidade computacional, a média da qualidade das soluções e a média do desvio padrão da qualidade.

Porém todos esses resultados são devidos a instâncias que conhecemos previamente o valor de uma solução ótima. Além desses, executamos mais um teste, dessa vez com grafos completamente aleatórios gerados com o procedimento descrito a seguir.

Para esse segundo teste utilizamos o procedimento GrafoAleatório. Este procedimento recebe dois inteiros como entrada, n e *densidade*, e devolve um grafo arbitrário $G = (V, A)$ com $|V| = n$, e com número de arestas de acordo com a *densidade* e uma função $w : V \to \mathbb{Z}_+$ de valores aleatórios. Para essas instâncias aleatórias que geramos, não sabemos *a priori* o valor de uma solução ótima.

Heurística	Complexidade	Qualidade	Desv. Padrão
HeurísticaArvoresAleatórias [$tent. = n$]	$O(q^2.n^3)$	0.814	0.005
HeurísticaArvoresAleatórias [$tent. = n^2$]	$O(q^2.n^4)$	0.880	0.006
HeurísticaCircuitoSimples	$O(q^2.n^2 + q^2.m.n)$	0.721	0.013
HeurísticaCircuitoOnerosa	$O(q^2.n^3.m)$	0.775	0.010
HeurísticaMista [$tent. = n$]	$O(q^2.n^4 + q^2.m.n^2)$	0.838	0.006
HeurísticaMista [$tent. = n^2$]	$O(q^2.n^5 + q^2.m.n^3)$	0.897	0.004

Tabela 4.11: Comparação das complexidades e qualidades das heurísticas

GrafoAleatório $(n, densidade)$

Entrada: dois inteiros, n e *densidade*.
Saída: um par constituído por um grafo (V, A) arbitrário conexo de forma que $|V| = n$ e uma função $w : V \to \mathbb{Z}_+$ definida sobre os vértices do grafo com valores aleatórios.

```
1  V ← {1, ..., n}
2  A ← ∅
3  ▷ Garantimos que o grafo é conexo gerando primeiro uma árvore aleatória
4  nArestas ← 0
5  Union-Find-Init (n)
6  while nArestas ≠ n − 1
7        do u ← número-aleatório (1, n)
8           v ← número-aleatório (1, n)
9           if Find (u) ≠ Find (v)
10             then Union (u, v)
11                  A ← A ∪ {u, v}
12                  nArestas ← nArestas + 1
13 ▷ Adicionamos as arestas arbitrárias de acordo com a densidade
14 totalArestas ← arestas-pela-densidade (n, densidade)
15 while nArestas < totalArestas
16       do u ← número-aleatório (1, n)
17          v ← número-aleatório (1, n)
18          if {u, v} ∉ A
19             then A ← A ∪ {u, v}
20                  nArestas ← nArestas + 1
21 ▷ Por fim distribuímos pesos aleatórios
22 for i ← 1 to n
23       do w(i ∈ V) ← número-aleatório ()
24 return ((V, A), w)
```

Executamos o teste da seguinte maneira. Geramos grafos aleatórios com o número de vértices variando de 10 a 70 e densidades de 30%, 60% e 90%. Para cada par (vértices,densidade) sorteamos 10 grafos. Executamos as quatro heurísticas sobre essa massa de dados usando quatro valores diferentes para q (2, $n/4$, $n/2$ e $3n/4$). A diferença desse teste é que não definimos um número de tentativas nem deixamos as heurísticas executando até alcançarem sua condição de parada. Atribuímos uma fração de tempo proporcional ao número de arestas da instância sendo processada e deixamos os algoritmos executarem no máximo por esse tempo pré-determinado. Dessa forma, todos os algoritmos tiveram o mesmo tempo para calcular uma partição. Os resultados desse teste são apresentados na Tabela 4.12. A coluna "Ganhou" mostra quantas vezes uma heurística apresentou resultados melhores ou iguais (empates) que as demais. No total foram 840 execuções dos algoritmos.

Heurística	Ganhou
HeurísticaArvoresAleatórias	608
HeurísticaCircuitoSimples	46
HeurísticaCircuitoOnerosa	121
HeurísticaMista	617

Tabela 4.12: Comparação das heurísticas

Como podemos ver pelos resultados apresentados, a heurística HeurísticaMista obtém resultados de melhor qualidade quanto comparada às outras heurísticas desenvolvidas.

Tentamos ainda explorar alguns outros aspectos da heurística HeurísticaMista. Para isso executamos alguns testes específicos, cujos resultados podem ser vistos nas figuras a seguir. Na Figura 4.1 temos dois gráficos. O da parte superior mostra a média da qualidade das soluções obtidas pela execução da heurística com $tentativas = n^2$ em 50 grafos aleatórios de 30 vértices e 50% de densidade, com q variando de 1 a n. O gráfico de baixo mostra a distribuição dos valores do número de Stirling do segundo tipo (em escala logarítmica) com $n = 30$ e k variando de 1 a n.

A Figura 4.2 também exibe dois gráficos, sendo que o da parte superior mostra a mesma média de qualidades como na Figura 4.1 só que dessa vez para grafos com 20 vértices. O segundo gráfico mostra a média de qualidades obtidas, para alguns valores de q, quando variamos a densidade dos grafos, desde árvores até grafos completos.

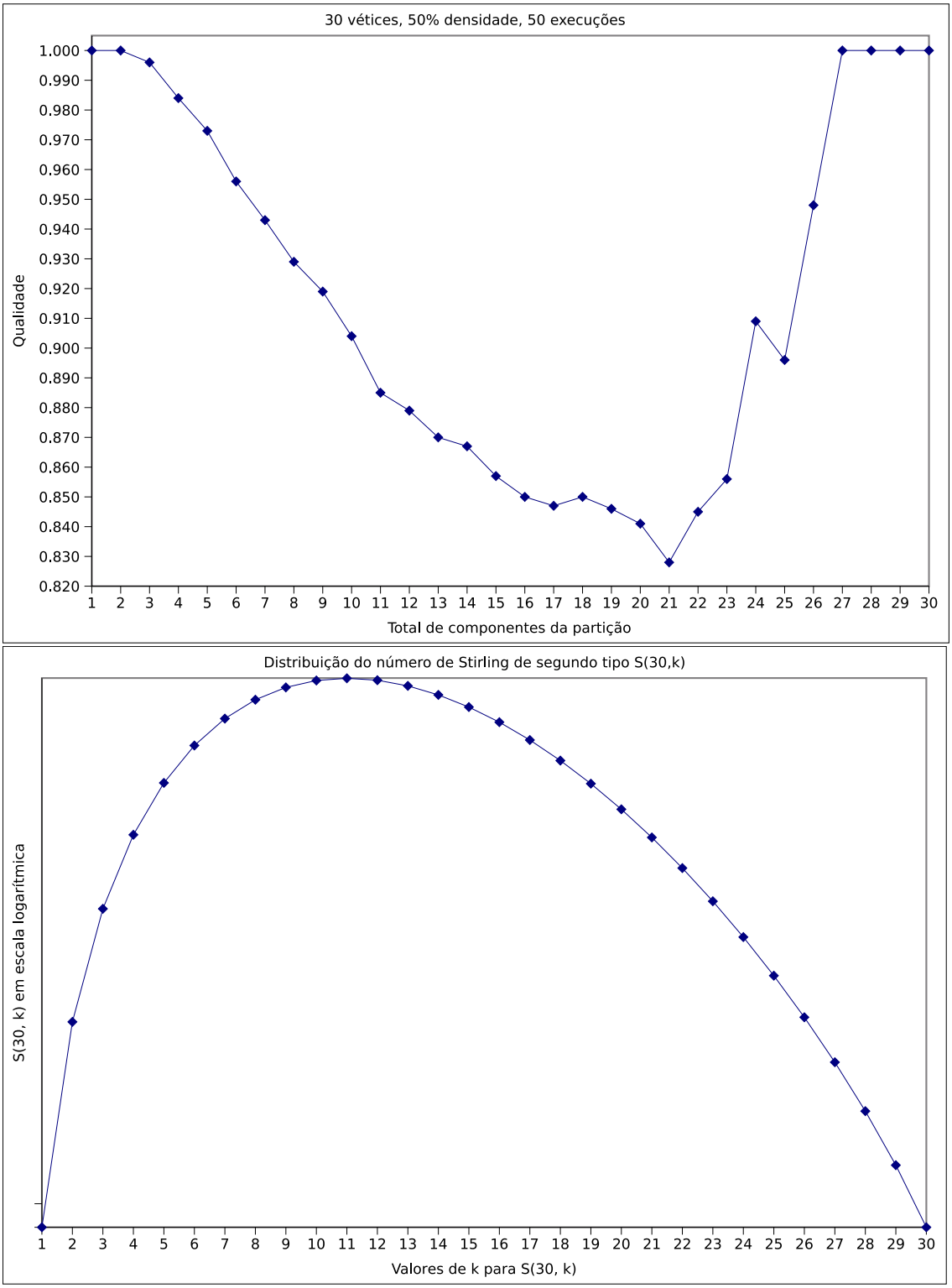

Figura 4.1: Execução da heurística para grafos com 30 vértices (gráfico ao alto), e a distribuição do número de Stirling do segundo tipo $S(30, k)$ em escala logarítmica

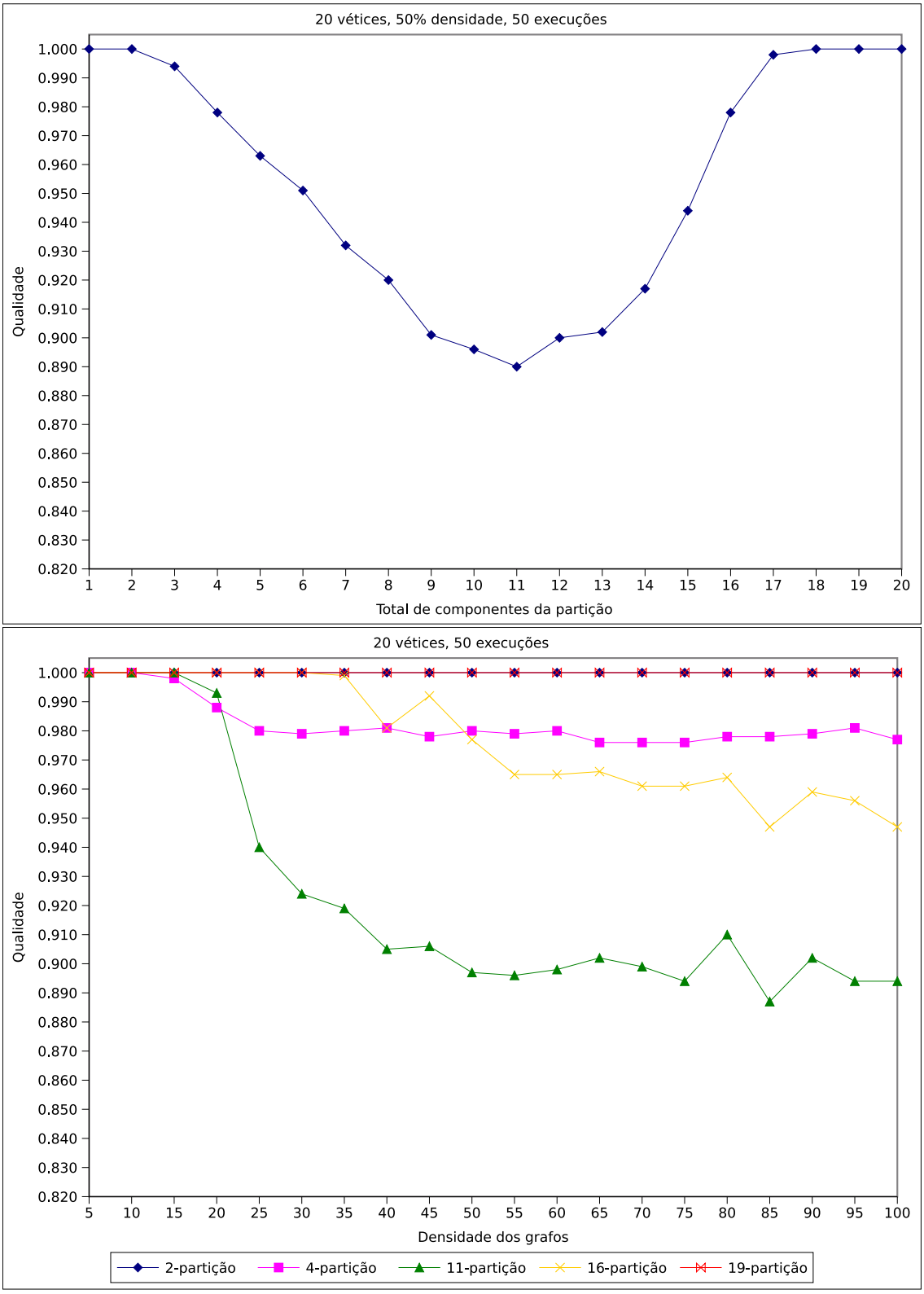

Figura 4.2: Execução da heurística para grafos com 20 vértices

Capítulo 5

Algoritmo de aproximação para bipartição

O melhor algoritmo de aproximação conhecido para o PCB_2 é uma 3/4-aproximação que foi obtido por Chlebíková [13] em 1996. Neste capítulo discutimos esse algoritmo e sua implementação. Também apresentamos uma comparação entre a qualidade das soluções produzidas por uma das heurísticas apresentada no capítulo anterior e essa aproximação.

5.1 Algoritmo de aproximação de Chlebíková

Um estudo detalhado do algoritmo apresentado a seguir pode ser encontrado na tese de doutorado de Liliane Salgado [32]. O leitor pode recorrer a esse texto para ver uma análise completa do algoritmo bem como provas de teoremas e lemas aqui mencionados. Os resultados aqui apresentados seguem a linha dessa tese, que faz uma análise parametrizada da razão de aproximação do algoritmo de Chlebíková [13].

Antes de descrever a aproximação vamos definir o conceito de *vértice admissível* usado pelo algoritmo. Seja G um grafo biconexo e (V_1, V_2) uma bipartição conexa de G. Dizemos que um vértice $u \in V_2$ é **admissível** para V_1 se u é adjacente a um vértice de V_1 e se a bipartição $(V_1 \cup \{u\}, V_2 \backslash \{u\})$ é uma bipartição conexa de G. Em outras palavras, um vértice $u \in V_2$ é **admissível** se ele não é um vértice de corte de $G[V_2]$ e é adjacente a um vértice de V_1.

O algoritmo também faz uso do seguinte lema.

Lema 5.1.1. *Seja G um grafo biconexo e (V_1, V_2) uma bipartição conexa de G tal que $|V_2| \geq 2$. Então existem pelo menos dois vértices distintos em V_2 admissíveis para V_1.*

BipartiçãoDeChlebíkováBiconexo (G, w)

Entrada: um grafo biconexo $G = (V, A)$ e uma função $w : V(G) \to \mathbb{Z}_+$.
Saída: uma 2-partição conexa (V_1, V_2) de G.

1 $v_1 \leftarrow \arg\max\{w(v) \in V\}$ ▷ Seja v_1 um vértice de peso máximo
2 $V_1 \leftarrow \{v_1\}$
3 $V_2 \leftarrow V \backslash V_1$
4 $\beta \leftarrow w(V)/2$
5 **while** $w(V_1) < \beta$
6 **do** Escolha um vértice *admissível* $u \in V_2$ de peso mínimo
7 **if** $w(u) \geq 2(\beta - w(V_1))$
8 **then break** ▷ Caso 1: $w(V_1) \geq \beta - \frac{1}{2}w(u)$
9 **else** $V_1 \leftarrow V_1 \cup \{u\}$ ▷ Caso 2: $w(V_1) < \beta + \frac{1}{2}w(u)$
10 $V_2 \leftarrow V_2 \backslash \{u\}$
11 **return** (V_1, V_2)

Mostramos a seguir a análise da razão de aproximação desse algoritmo.

Teorema 5.1.1. *Seja I uma instância do PCB_2 que consiste de um grafo biconexo $G = (V, A)$ e uma função $w : V \to \mathbb{Z}_+$. Considere $V = \{v_1, v_2, \ldots, v_n\}$, $n \geq 3$, $w(v_1) \geq w(v_2) \geq \ldots \geq w(v_n)$ e $t = \frac{w(V)}{w(v_3)}$. Então* BipartiçãoDeChlebíkováBiconexo *aplicado à instância I devolve em tempo polinomial uma bipartição conexa de G com $\mu = min\{w(V_1), w(V_2)\}$ (valor da função objetivo), tal que*

$$Se\ w(v_1) \geq \frac{1}{2}\, w(V) \quad então \quad \mu = opt(I) \tag{5.1}$$

$$Se\ w(v_1) \leq \frac{1}{2}\, w(V) \quad então \quad \mu \geq \frac{1}{2}\,(w(V) - w(v_3)) \geq \frac{t-1}{2t}\, w(V) \tag{5.2}$$

$$Se\ 3 \leq t \leq 4 \quad então \quad \mu \geq \frac{t-1}{2t-4}\, opt(I) \tag{5.3}$$

$$Se\ t \geq 4 \quad então \quad \mu \geq \frac{t-1}{t}\, opt(I) \tag{5.4}$$

Demonstração. Claramente o algoritmo BipartiçãoDeChlebíkováBiconexo é polinomial e devolve uma bipartição conexa de G. A afirmação 5.1 também é trivial.

Vamos provar a afirmação 5.2. Suponhamos primeiramente que $w(v_1) \geq w(V)/2$. Neste caso, qualquer vértice u escolhido no passo 6 do algoritmo satisfaz $w(u) \leq w(v_3)$, dado que $v_1 \notin V_2$ e pelo Lema 5.1.1 o conjunto V_2 tem pelo menos dois vértices admissíveis.

Temos que considerar as duas possíveis situações de parada do algoritmo. Imediatamente antes de devolver a bipartição (V_1, V_2) o algoritmo executou ou o Caso 1 (passo 8) ou o Caso 2 (passo 9). Vamos analisar essas duas possibilidades. Seja $\beta = \frac{w(V)}{2}$, como na descrição do algoritmo.

Caso 1: Neste caso, $w(V_1) \geq \beta - \frac{1}{2}w(u)$, e portanto, $\mu \geq \beta - \frac{1}{2}w(u) \geq \beta - \frac{1}{2}w(v_3)$. A última desigualdade segue do fato de que $w(u) \leq w(v_3)$. Como $t = \frac{w(V)}{w(v_3)}$, temos

que

$$\mu \geq \beta - \frac{1}{2t}w(V) = \frac{1}{2}w(V) - \frac{1}{2t}w(V) = \frac{t-1}{2t}w(V).$$

Caso 2: Neste caso, $w(V_1) < \beta + \frac{1}{2}w(u)$. Então

$$\mu = w(V_2) = w(V) - w(V_1) > w(V) - \beta - \frac{1}{2}w(u) = \frac{w(V) - w(u)}{2}.$$

Como $w(u) \leq w(v_3) = \frac{w(V)}{t}$, temos que

$$\mu > \frac{w(V) - w(v_3)}{2} = \frac{t-1}{2t}w(V).$$

Por fim, se $w(v_1) = \frac{1}{2}w(V)$, a afirmação 5.2 é claramente verdadeira, o que conclui a prova dessa afirmação.

Vamos provar as afirmações 5.3 e 5.4. Primeiramente observamos que é imediato que $opt(I) \leq w(V)/2$. Quando $3 \leq t \leq 4$, podemos porém provar um limite melhor para o valor ótimo. Afirmamos que neste caso $opt(I) \leq (1 - \frac{2}{t})w(V)$. De fato, para qualquer bipartição (X, Y) de G, uma das duas classes contém pelo menos dois elementos de $\{v_1, v_2, v_3\}$. Como cada um desses vértices tem peso pelo menos $w(V)/t$, segue que o peso dessa tal classe é pelo menos $2\,w(V)/t \geq w(V)/2$. Assim, a classe mais leve de uma solução ótima tem peso no máximo $w(V) - 2\,w(V)/t$, e portanto, $opt(I) \leq (1 - \frac{2}{t})w(V)$, como afirmamos. Usando 5.2, temos que

$$opt(I) \leq \left(1 - \frac{2}{t}\right)\left(\frac{2t}{t-1}\right)\mu. \quad \text{Portanto, } \mu \geq \frac{t-1}{2t-4}\,opt(I).$$

Suponhamos agora que $t \geq 4$. Neste caso, usando o limite trivial para o valor ótimo juntamente com a afirmação 5.2, deduzimos que

$$opt(I) \leq \frac{1}{2}\left(\frac{2t}{t-1}\right)\mu. \quad \text{Portanto, } \mu \geq \frac{t-1}{t}\,opt(I).$$

Concluímos assim a prova do teorema. □

Corolário 5.1.1. BipartiçãoDeChlebíkováBiconexo *é uma* $\frac{3}{4}$*-aproximação para o* PCB_2 *restrito a grafos biconexos.*

Demonstração. O resultado é obtido aplicando-se o Teorema 5.1.1 no qual podemos constatar que o pior caso se dá quando $t = 4$. □

A seguir mostraremos o algoritmo BipartiçãoDeChlebíková para o PCB_2 em grafos arbitrários. Esse algoritmo utiliza o algoritmo BipartiçãoDeChlebíkováBiconexo, que é restrito a grafos biconexos. Veremos que a razão de aproximação do algoritmo BipartiçãoDeChlebíkováBiconexo também vale para o BipartiçãoDeChlebíková.

Esse algoritmo faz uso de alguns procedimentos não detalhados aqui. O primeiro deles é componentes-biconexas que recebe um grafo conexo G como entrada e devolve um

conjunto CB. Cada um dos elementos de CB é um subconjunto de $V(G)$ que induz uma componente biconexa de G.

Outro procedimento utilizado pelo algoritmo é AJUSTA-PESOS que recebe um grafo G, sua função peso $w : V(G) \to \mathbb{Z}_+$ e um subgrafo G' de G que é uma componente biconexa de G. Esse procedimento devolve uma função peso $w' : V(G') \to \mathbb{Z}_+$ tal que que $w'(v) = w(v)$ se $v \in V(G')$ não é um vértice-de-corte em G. Caso v seja um vértice-de-corte em G então ele recebe o seu peso em G somado ao peso de todos os vértices em G separados por ele da componente biconexa. Podemos ver um exemplo desse ajuste de pesos na Figura 5.1. Note neste exemplo que, se G' é a componente biconexa com 4 vértices, o vértice de G' com peso 2 em G passa a ter peso $2 + 6 + 5 + 1$ em G'.

Uma vez que esses ajustes de pesos são feitos, podemos agora tratar cada componente biconexa com pesos ajustados, usando o algoritmo visto para grafos biconexos.

Detalhes da implementação desses procedimentos bem como do algoritmo de aproximação de Chlebíková serão discutidos na próxima seção deste capítulo.

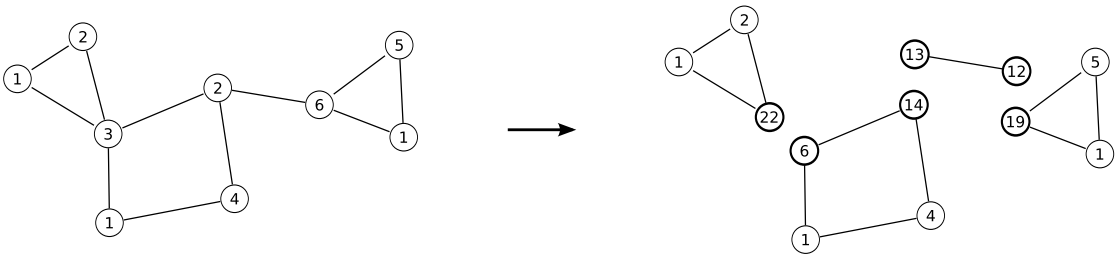

Figura 5.1: Exemplo de ajuste dos pesos das componentes biconexas

Damos abaixo a descrição do algoritmo BIPARTIÇÃODECHLEBÍKOVÁ.

BIPARTIÇÃODECHLEBÍKOVÁ (G, w)

Entrada: um grafo G e uma função $w : V(G) \to \mathbb{Z}_+$.
Saída: uma 2-partição conexa (V_1, V_2) de G.

$V_1 \leftarrow \emptyset$
$V_2 \leftarrow \emptyset$
$ComponentesBiconexas \leftarrow$ COMPONENTES-BICONEXAS (G)
foreach componente biconexa $CB \in ComponentesBiconexas$
    do $G' \leftarrow G[CB]$
        $w' \leftarrow$ AJUSTA-PESOS (G', G, w)
        $(V_1', V_2') \leftarrow$ BIPARTIÇÃODECHLEBÍKOVÁBICONEXO (G', w')
        if $w(min\{V_1', V_2'\}) > w(V_1)$
            then $V_1 \leftarrow min\{V_1', V_2'\}$
                 $V_2 \leftarrow max\{V_1', V_2'\}$
return (V_1, V_2)

O seguinte teorema sobre o algoritmo BIPARTIÇÃODECHLEBÍKOVÁ pode ser provado [32].

Teorema 5.1.2. *Seja I uma instância do* PCB_2 *que consiste de um grafo conexo $G = (V, A)$ e uma função $w : V \to \mathbb{Z}_+$. Então o algoritmo* BipartiçãoDeChlebíková*, aplicado à instância I, devolve em tempo polinomial uma bipartição conexa de G com $\mu = w(V_1)$ (valor da função objetivo), tal que*

$$\mu \geq \frac{3}{4} opt(I).$$

5.2 Detalhes da implementação

Implementar a aproximação BipartiçãoDeChlebíková é trabalhoso, embora o algoritmo seja simples, pois exige manipulação de vários grafos. Primeiramente o grafo de entrada é decomposto em componentes biconexas, os pesos dos vértices de cada componente são ajustados, e depois cada componente biconexa é dada como entrada para o procedimento BipartiçãoDeChlebíkováBiconexo.

Implementamos algoritmos que encontram os vértices-de-corte e as componentes biconexas. Esses algoritmos não serão descritos aqui. São algoritmos lineares apresentados por Tarjan [38] como aplicações de busca em profundidade.

O cerne da implementação do procedimento BipartiçãoDeChlebíkováBiconexo é decidir, dada uma bipartição conexa (V_1, V_2), quais vértices de V_2 são admissíveis para V_1. Para isso, geramos o subgrafo $G[V_2]$ e marcamos seus vértices-de-corte. Dessa forma, todos os vértices admissíveis em V_2 são os vértices adjacentes a V_1 e que não estão marcados como vértice-de-corte de $G[V_2]$.

Agora, a parte principal do algoritmo BipartiçãoDeChlebíková está ancorada nos procedimentos componentes-biconexas e ajusta-pesos. Como dito acima, o procedimento componentes-biconexas implementado é um algoritmo linear, que rotula as arestas de cada componente biconexa com um mesmo rótulo: arestas de componentes distintas recebem rótulos distintos.

Para ajustar os pesos também foi implementado um algoritmo linear. Este algoritmo recebe como entrada uma componente biconexa e o grafo original e ajusta o peso dos vértices-de-corte do grafo original presentes na componente. Dado um vértice-de-corte, digamos z, esse ajuste é feito com uma busca no grafo original somando-se o peso de todos os vértices alcançáveis a partir de z sem usar arestas rotuladas com o mesmo rótulo das arestas da componente biconexa que contém z.

5.2.1 Testes

Os testes computacionais foram realizados com entradas geradas pelo procedimento GrafoAleatórioParaQPartição descrito na capítulo anterior, variando-se o tamanho e a densidade dos grafos. Com isso, além de conhecer o valor de uma solução ótima, ainda conseguimos calcular a razão de aproximação para a instância, de acordo com o valor de t, conforme garante o Teorema 5.1.1.

A Tabela 5.1 mostra o resultado dos testes. Cada linha representa a média da execução do algoritmo com 20 grafos aleatórios. A *qualidade* de uma solução é dada pelo

valor da função objetivo encontrada pelo algoritmo dividido pelo valor de uma solução ótima.

Densidade	Tamanho do Grafo	Qualidade	Razão Teórica
30%	10	0.972	0.875
	20	0.971	0.925
	30	0.983	0.947
	40	0.981	0.958
	50	0.989	0.965
	60	0.992	0.972
	70	0.990	0.975
60%	10	0.954	0.875
	20	0.978	0.925
	30	0.987	0.950
	40	0.987	0.960
	50	0.991	0.967
	60	0.991	0.971
	70	0.994	0.975
90%	10	0.956	0.874
	20	0.969	0.929
	30	0.982	0.947
	40	0.987	0.959
	50	0.987	0.967
	60	0.991	0.972
	70	0.994	0.975

Tabela 5.1: Testes computacionais do algoritmo BipartiçãoDeChlebíková

5.3 Comparação com heurística baseada em árvores geradoras

Realizamos dois testes comparando a qualidade das soluções obtidas com a execução do algoritmo BipartiçãoDeChlebíková e da heurística HeurísticaMista apresentada no capítulo anterior. Em todos os testes usamos $|V(G)|^2$ como o parâmetro *tentativas* dessa heurística.

No primeiro teste utilizamos o procedimento GrafoAleatórioParaQPartição para gerar grafos de tamanhos variando entre 10 e 70 com densidades de 30%, 60% e 90%, e cujo valor de uma solução ótima é conhecido. Para cada par $(tamanho, densidade)$ geramos 20 grafos e executamos o algoritmo de aproximação e a heurística.

Usamos aqui o mesmo conceito de *qualidade* de uma bipartição descrito na seção anterior. A Tabela 5.2 mostra os resultados computacionais do primeiro teste, em que

conhecemos o valor do ótimo e conseguimos calcular a qualidade da bipartição. Note que nas colunas "Aproximação Qualidade" e "Heurística Qualidade" temos a média das qualidades das 20 execuções dos algoritmos.

Densidade	Tamanho do Grafo	Aproximação Melhor	Aproximação Qualidade	Heurística Melhor	Heurística Qualidade	Empate
30%	10	0	0.968	10	1.000	10
	20	0	0.974	17	1.000	3
	30	0	0.982	16	1.000	4
	40	0	0.985	17	1.000	3
	50	0	0.987	17	1.000	3
	60	0	0.989	20	1.000	0
	70	0	0.992	16	1.000	4
60%	10	0	0.944	18	1.000	2
	20	0	0.983	13	1.000	7
	30	0	0.981	17	1.000	3
	40	0	0.988	17	1.000	3
	50	0	0.988	17	1.000	3
	60	0	0.991	18	1.000	2
	70	0	0.992	17	1.000	3
90%	10	0	0.944	17	1.000	3
	20	0	0.980	16	1.000	4
	30	0	0.980	18	1.000	2
	40	0	0.985	20	1.000	0
	50	0	0.991	15	1.000	5
	60	0	0.991	18	1.000	2
	70	0	0.991	19	1.000	1

Tabela 5.2: Comparação de BipartiçãoDeChlebíková e HeurísticaMista, com conhecimento do valor ótimo

Para o segundo teste usamos o procedimento GrafoAleatório, descrito no capítulo anterior. Lembramos que este procedimento recebe dois inteiros como entrada, n e *densidade*, e devolve um grafo arbitrário $G = (V, A)$ com $|V| = n$, e com número de arestas de acordo com a *densidade* e uma função $w : V \to \mathbb{Z}_+$ de valores aleatórios. Assim, neste caso, não sabemos *a priori* o valor de uma solução ótima da instância gerada.

Como no primeiro teste, geramos grafos com tamanho variando de 10 e 70, e densidades de 30%, 60% e 90%. Da mesma forma, com cada par (*tamanho*, *densidade*) sorteamos 20 grafos e executamos os algoritmos BipartiçãoDeChlebíková e HeurísticaMista.

A Tabela 5.3 mostra o resultado desse teste. Mostramos quantas vezes um algoritmo apresentou um resultado melhor que o outro e quantos empates (os dois algoritmos encontram o mesmo valor para a função objetivo) aconteceram. Na coluna "Aproximação Diferença" temos a média, de todas as vezes que a aproximação apresentou um resultado pior do que o da heurística, da diferença entre o valor objetivo da solução encontrada pela heurística e o da solução encontrada pela aproximação, dividido pelo peso do grafo. Já na coluna "Heurística Diferença" temos a média análoga, de quando o resultado do

algoritmo de aproximação foi melhor do que o da heurística.

Densidade	Tamanho do Grafo	Aproximação Venceu	Heurística Venceu	Empate	Aproximação Diferença	Heurística Diferença
30%	10	0	11	9	0.025	0.000
	20	0	19	1	0.019	0.000
	30	0	19	1	0.013	0.000
	40	0	19	1	0.008	0.000
	50	0	20	0	0.007	0.000
	60	0	20	0	0.006	0.000
	70	0	20	0	0.004	0.000
60%	10	0	20	0	0.028	0.000
	20	0	18	2	0.020	0.000
	30	0	19	1	0.009	0.000
	40	0	19	1	0.007	0.000
	50	0	20	0	0.006	0.000
	60	0	19	1	0.006	0.000
	70	0	20	0	0.005	0.000
90%	10	0	18	2	0.025	0.000
	20	0	20	0	0.016	0.000
	30	0	18	2	0.010	0.000
	40	0	19	1	0.009	0.000
	50	0	20	0	0.007	0.000
	60	0	20	0	0.007	0.000
	70	0	20	0	0.004	0.000

Tabela 5.3: Comparação de BipartiçãoDeChlebíková e HeurísticaMista, sem conhecimento do valor ótimo

Os testes mostram que o algoritmo de aproximação BipartiçãoDeChlebíková encontra, na prática, soluções cuja razão em relação ao ótimo é melhor do que a razão de aproximação teórica garantida pelo teorema; algumas vezes, este encontra também soluções ótimas. Mesmo assim, em todos os testes a HeurísticaMista sistematicamente encontrou soluções de qualidade melhor ou igual do que das soluções encontradas pelo algoritmo de aproximação.

Capítulo 6

Conclusão

Como já foi dito no inicio deste texto, existem poucos resultados algorítmicos relativos ao PCB e $\mathsf{PCB_q}$. Dentre os resultados relevantes, encontramos um algoritmo polinomial para o caso especial em que o grafo de entrada é uma árvore, e dois algoritmos de aproximação, um para o $\mathsf{PCB_2}$ e outro para o $\mathsf{PCB_3}$ (apenas para grafos 3-conexos). Durante o desenvolvimento deste trabalho implementamos a maioria desses algoritmos e projetamos heurísticas inéditas para o caso geral do problema.

Essas heurísticas apresentaram resultados computacionais muito bons. A comparação com um algoritmo de aproximação apresentada no Capítulo 5 evidencia bem esse fato. Além disso, essas heurísticas têm complexidades diferentes e apresentam, na média, soluções com qualidades diferentes, porém nenhuma delas se mostrou ruim. Isso nos permite decidir pela escolha de uma heurística, levando em conta um compromisso entre *qualidade* e *tempo*, que seja compatível com os propósitos da aplicação e o poder computacional de que dispomos.

Um ponto importante a se destacar é o fato de que modelos de programação linear inteira se mostraram proibitivos para o PCB, mesmo para instâncias pequenas do problema. Fizemos alguns testes dessa natureza e obtivemos resultados pouco satisfatórios. Por exemplo, tentamos resolver com um modelo de programação inteira uma instância com 30 vértices e 40% de densidade, em que queríamos encontrar uma 10-partição. Depois de mais de 6 dias de execução o *solver* não encontrou uma solução ótima. Para essa mesma instância, a HeurísticaMista obteve uma solução ótima (confirmada pelo conhecimento do valor de um limitante superior dado pela relaxação linear).

Agora, com as heurísticas que desenvolvemos podemos pensar em estratégias de *branch-and-bound* ou *branch-and-cut* que certamente se beneficiarão do uso dessas heurísticas para melhorar o seu desempenho. Acreditamos que essas heurísticas permitirão encontrar soluções ótimas para instâncias maiores das que podemos resolver hoje.

Uma outra vantagem dessas heurísticas é que elas se baseiam apenas no algoritmo de Perl e Schach [31] para árvores. Com isso, facilmente podemos alterar as heurísticas para trabalhar com outras funções objetivo, apenas mudando o algoritmo usado para obter soluções ótimas em árvores. É possível, por exemplo, utilizar as mesmas heurísticas para o problema análogo ao PCB, em que queremos minimizar o peso da classe mais pesada da

q-partição. Existe um algoritmo polinomial para esse problema restrito a árvores devido a Becker e Perl [6] e Becker, Schach e Perl [8].

A implementação do algoritmo de aproximação para o PCB_2 devido a Chlebíková abre novas possibilidades para desenvolvermos heurísticas para o caso geral do PCB.

Todos esses algoritmos apresentados no texto foram implementados em C++, em que desenvolvemos estruturas e procedimentos específicos para trabalhar com grafos e seus subgrafos, manipulação de cortes e partições. O trabalho totalizou mais de 7500 linhas de código. O leitor interessado nas implementações e programas produzidos durante esse estudo pode acessar o endereço `http://www.ime.usp.br/~lucindo/graphpar` na *Internet* para obter mais informações.

Números de Stirling do segundo tipo

Aqui apresentamos uma tabela com valores de alguns números de Stirling do segundo tipo $S(n, k)$. Os valores n e k correspondem aos tamanhos de grafos e número de componente de partições usados nos testes dos capítulos 4 e 5.

n	$k = 2$
10	511
20	524287
30	536870911
40	54 9755813887
50	56294 9953421311
60	57646075 2303423487
70	5 9029581035 8705651711
n	$k = \frac{n}{4}$
10	9330
20	74 9206090500
30	263 8301868404 8108297800
40	2364 6841252914 8293635392 5428946680
50	623890 1276275784 8114928617 9482673756 3889288230
60	219308136 6844243711 1542041531 9128230087 8699826605 5315277600
70	81 8706198094 4562319259 1621913430 0515422069 8815385655 7327212922 3140163935
n	$k = \frac{n}{2}$
10	42525
20	591 7584964655
30	128 7986807277 0626040000
40	162 1889095279 7575048788 7236507181
50	745 3802153273 2000833796 2623483762 5465912500
60	9563 5288550944 0274475273 3025368976 2013863174 1164099440
70	287380 8622933777 5761944973 3143478021 8971366396 4121783361 2682183750
n	$k = \frac{3n}{4}$
10	750
20	452329200
30	7182 3880393200
40	7 7277807228 7000494520
50	4571493 4748917557 3737344245
60	20728 1329327164 7989774404 3460870000
70	2 6977285131 7932002545 3690186202 4953405646

Lista de Figuras

Lista de Tabelas

Referências Bibliográficas

[1] E. Aparo and B. Simeone. Un algoritmo di equipartizione e il suo impiego in un problema de contrasto ottico. *Ricerca Operativa*, 3:31–42, 1973.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation (Combinatorial Optimization Problems and Their Approximability Properties)*. Springer-Verlag, 1999.

[3] R. I. Becker, I. Lari, M. Lucertini, and B. Simeone. Max-min partitioning of grid graphs into connected components. *Networks*, 32(2):115–125, 1998.

[4] R. I. Becker, I. Lari, M. Lucertini, and B. Simeone. A polynomial-time algorithm for max-min partitioning of ladders. *Theory of Computing Systems*, 34(4):353–374, 2001.

[5] R. I. Becker and Y. Perl. Shifting algorithms for tree partitioning with general weighting functions. *J. Algorithms*, 4(2):101–120, 1983.

[6] R. I. Becker and Y. Perl. A shifting algorithm for constrained min-max partition on trees. *Discrete Applied Mathematics*, 45(1):1–28, 1993.

[7] R. I. Becker and Y. Perl. The shifting algorithm technique for the partitioning of trees. *Discrete Applied Mathematics*, 62(1-3):15–34, 1995.

[8] R. I. Becker, S. R. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *Journal of ACM*, 29(1):58–67, 1982.

[9] B. Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1998.

[10] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillan/Elsevier, 1976.

[11] P. Camerini, G. Galbiati, and F. Maffioli. On the complexity of finding multi-constrained spanning trees. *Discrete Applied Mathematics*, 5:39–50, 1983.

[12] F. Chataigner, L. R. B. Salgado, and Y. Wakabayashi. Approximability and inapproximability results on balanced connected partitions of graphs. Submetido.

[13] J. Chlebíková. Approximating the maximally balanced connected partition problem in graphs. *Information Processing Letters*, 60:225–230, 1996.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[15] R. Diestel. *Graph Theory*. Graduate Texts in Mathematics. Springer-Verlag, 2000.

[16] M. Dyer and A. Frieze. On the complexity of partitioning graphs into connected subgraphs. *Discrete Applied Mathematics*, 10:139–153, 1985.

[17] C. G. Fernandes, F. K. Miyazawa, M. Cerioli, P. Feofiloff, and et al. *Uma introdução sucinta a algoritmos de aproximação*. Publicações Matemáticas do IMPA. [IMPA Mathematical Publications]. Instituto de Matemática Pura e Aplicada, Rio de Janeiro, 2001. Livro preparado para o 23º Colóquio Brasileiro de Matemática.

[18] G. N. Frederickson. Optimal algorithms for tree partitioning. In *ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1991.

[19] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[20] E. Győri. On division of graphs to connected subgraphs. In *Combinatorics (Proc. Fifth Hungarian Combinatorial Colloq., Keszthely, 1976, Vol. I*, volume 18 of *Colloq. Math. Soc. János Bolyai*, pages 485–494. North-Holland, Amsterdam, 1978.

[21] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM Journal on Scientific Computing*, 6(1):151–154, 1977.

[22] L. Lovász. A homology theory for spanning trees of a graph. *Acta Math. Acad. Sci. Hungar.*, 30(3-4):241–251, 1977.

[23] M. Lucertini, Y. Perl, and B. Simeone. Image enhancement by path partitioning. In V. Cantoni and S. Levialdi, editors, *Recent Issues in Image Analysis*, Lectures Notes in Computer Science. Springer, 1989.

[24] M. Lucertini, Y. Perl, and B. Simeone. Most uniform path partitioning and its use in image processing. *Discrete Applied Mathematics*, 42:227–256, 1993.

[25] J. A. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[26] J. Ma and S. Ma. An $O(k^2n^2)$ algorithm to find a k-partition in a k-connected graph. *Journal of Computer Science and Technology*, 9(1):86 – 91, 1994.

[27] A. Maravalle, B. Simeone, and R. Naldini. Clustering on trees. *Computational Statistics and Data Analysis*, 24:217–234, 1997.

[28] A. Mingozzi, S. Ricciardelli, and M. Spadoni. Partitioning a matrix to minimize the maximum cost. *Discrete Applied Mathematics*, 62(1–3):221–248, 1995.

[29] S. Nakano, M. Rahman, and T. Nishizeki. A linear-time algorithm for four-partitioning four-connected planar graphs. *Information Processing Letters*, 62:315 – 322, 1997.

[30] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[31] Y. Perl and S. R. Schach. Max-min tree partitioning. *Journal of ACM*, 28(1):5–15, 1981.

[32] L. R. B. Salgado. *Algoritmos de Aproximação para Partições Conexas em Grafos*. Tese de doutorado, Universidade de São Paulo, 2004.

[33] R. Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[34] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.

[35] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1992.

[36] H. Suzuki, N. Takahashi, and T. Nishizeki. A linear algorithm for bipartition of biconnected graphs. *Information Processing Letters*, 33(5):227 – 232, 1990.

[37] H. Suzuki, N. Takahashi, T. Nishizeki, H. Miyano, and S. Ueno. An algorithm for tripartitioning 3-connected graphs. *Journal of Information Processing Society of Japan*, 31(5):584 – 592, 1990.

[38] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Scientific Computing*, 1(2):146–160, 1972.

[39] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.