UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALEX REIMANN CUNHA LIMA

CLASSIFICAÇÃO DE COGRAFOS QUANTO AO ÍNDICE CROMÁTICO

CHAPECÓ
2014

ALEX REIMANN CUNHA LIMA

CLASSIFICAÇÃO DE COGRAFOS QUANTO AO ÍNDICE CROMÁTICO

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Leandro Miranda Zatesko

CHAPECÓ

2014

Lima, Alex Reimann Cunha

Classificação de cografos quanto ao índice cromático / Alex Reimann Cunha Lima. – 2014.
101 f.: il.; 30 cm.

Orientador: Leandro Miranda Zatesko.
Trabalho de conclusão de curso (graduação) - Universidade Federal da Fronteira Sul, Curso de Ciência da Computação, Chapecó, SC, 2014.

1. Coloração de arestas. 2. Cografos. 3. Conjectura *Overfull*. I. Zatesko, Leandro Miranda, orient. II. Universidade Federal da Fronteira Sul. III. Título.

E-mail: cunha.lima@ufrgs.br

ALEX REIMANN CUNHA LIMA

CLASSIFICAÇÃO DE COGRAFOS QUANTO AO ÍNDICE CROMÁTICO

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Leandro Miranda Zatesko

Este trabalho de conclusão de curso foi defendido e aprovado pela banca em: 12/12/2014

BANCA EXAMINADORA:

Prof. Leandro Miranda Zatesko - UFFS

(Presidente)

Profª Drª Sheila Morais de Almeida - UTFPR (em vídeo-conferência)

Prof. Marcelo Cezar Pinto - UFFS

Ao meu irmão Erich, em quem confio e me espelho para conseguir trilhar com honra e perseverança os rumos da vida, dedico este trabalho.

AGRADECIMENTOS

Ao Professor Leandro Miranda Zatesko, por todos os seus ensinamentos e pela dedicação despendida ao longo da elaboração deste trabalho.

À minha mãe, ao meu pai e ao meu irmão, pelo incentivo e apoio constantes aliados à educação, à formação e ao caráter que me proporcionaram, e também por serem a razão da minha existência.

Às minhas amigas e aos meus amigos, pela compreensão e por serem parte do que tenho de mais valioso na vida.

"Há uma única ciência, a Matemática, a qual ninguém se pode jactar de conhecer porque suas conquistas são, por natureza, infinitas; dela toda gente fala, sobretudo os que mais a ignoram."

— Malba Tahan

RESUMO

Coloração de arestas, um problema que consiste em descobrir quantas cores são necessárias para obter uma coloração mínima das arestas de um grafo, é $\mathcal{NP}$-completo (ainda que só haja duas possibilidades para o número mínimo de cores, conforme o *Teorema de Vizing*). Contudo, quando se trata de algumas subclasses de grafos, já restou provado que o problema admite solução polinomial ou linear. No caso dos cografos (grafos que não possuem um P_4 como subgrafo induzido), esse problema ainda não foi resolvido. Entretanto há indícios de que ele admita não apenas solução polinomial mas também linear, haja vista a existência da Conjectura *Overfull* e outros resultados relacionados à coloração de arestas de subclasses de cografos. A partir dessa constatação, é proposta uma conjectura, baseada em evidências extraídas de um programa especialmente projetado para a geração de cografos, acerca da operação de junção quando quando um deles possui núcleo acíclico. Além disso, são apresentados dois teoremas para casos mais restritos da conjectura: quando o núcleo acíclico não possuir arestas e quando houver um emparelhamento que não cubra o núcleo do outro grafo.

Palavras-chave: Coloração de arestas. Cografos. Conjectura *Overfull*.

ABSTRACT

Edge colouring, a problem which consists in figuring out how many colours are needed to provide a minimal colouring of the edges of a graph, is $\mathcal{NP}$-complete (even though, according to *Vizing's Theorem*, there are only two possibilities for the minimum number of colours). However, when dealing with some graph subclasses, it admits a polynomial or linear solution. For cograph class (graphs which do not have a P_4 as an induced subgraph), this problem has not been solved yet. Nevertheless there are indications that it admits not only a polynomial solution but also a linear one, given the existence of the Overfull Conjecture and other results related to edge colouring of cograph subclasses. From this observation, we propose a conjecture, based on evidence drawn from a computer program specially designed for generation of cographs. The conjecture applies to the join operation when one of the operands has an acyclic core. In addition, we present two theorems for more restricted cases of the conjecture: when the acyclic core is edgeless and when there is a matching which does not cover the core of the other graph.

Keywords: Edge colouring. Cographs. Overfull Conjecture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

CQP	Condição dos Quatro Pontos
GPL	*General Public License* (Licença Pública Geral)
LexBFS	*Lexicographic Breadth-First Search* (Busca em Largura Lexicográfica)
NO	vizinhança-*overfull*
O	*overfull* (sobrecarregado)
PSV	Propriedade do Subconjunto da Vizinhança
SGBD	Sistema de Gerenciamento de Banco de Dados
SO	subgrafo-*overfull*

LISTA DE SÍMBOLOS

$\chi(G)$	número cromático de G (p. 29)
$\chi'(G)$	índice cromático de G (p. 30)
$\Delta(G)$	grau máximo de G (p. 33)
$V, V(G)$	conjunto de vértices de G (p. 32)
$E, E(G)$	conjunto de arestas de G (p. 32)
$\|V(G)\|, n$	ordem de G (p. 32)
$\|E(G)\|, m$	número de arestas de G (p. 32)
$N(v)$	vizinhança de v (p. 33)
$d(v)$	grau do vértice v (p. 33)
$d(S)$	soma dos graus dos vértices de S (p. 33)
$\delta(G)$	grau mínimo de G (p. 33)
$G[S]$	subgrafo induzido de G por S (p. 34)
$\Lambda[G]$	núcleo de G (p. 34)
$N[\Lambda[G]]$	seminúcleo de G (p. 34)
$G \cup H$	união de G e H (p. 34)
$G * H$	junção de G e H (p. 34)
$\overline{G}$	complemento de G (p. 35)
P_n	caminho de ordem n (p. 35)
C_n	ciclo de ordem n (p. 35)
K_n	grafo completo de ordem n (p. 35)
$\text{lca}(u, v)$	ancestral comum mais baixo de u e v (p. 37)
$T(G)$	coárvore de G (p. 40)
$\alpha(i)$	conjunto de vértices da subárvore induzida pelo i-ésimo ramo (p. 40)
$a(i)$	cardinalidade de $\alpha(i)$ (p. 40)
T_{ji}^{x}	i-ésima subárvore de P_{xR} com raiz de rótulo j (p. 41)
σ	permutação resultante de uma passada da LexBFS (p. 43)
$\sigma(v)$	posição de v em σ (p. 43)
$u \prec_\sigma v$	u ocorre antes de v em σ (p. 43)
$\sigma^{-1}(i)$	i-ésimo vértice de σ (p. 43)
$N'(\alpha)$	conjunto dos vizinhos de α ainda não enumerados (p. 44)
$\overline{\sigma^{-}}$	permutação resultante de uma passada da LexBFS^{-} (p. 45)
$S(x)$	*fatia* de σ iniciada em x (p. 47)

$S^A(x)$	*subfatia* de $S(x)$ em σ que contém os vizinhos de x (p. 47)
$S^N(x)$	sequência de $S(x)$ em σ que contém os não-vizinhos de x (p. 47)
$S_i(x)$	i-ésima célula de $S^N(x)$ em σ (p. 47)
x_i	primeiro vértice de $S_i(x)$ (p. 47)
$\overline{S}(x)$	*fatia* de $\overline{\sigma^-}$ iniciada em x (p. 47)
$\overline{S^A}(x)$	*subfatia* de $S(x)$ em $\overline{\sigma^-}$ que contém os vizinhos de x (p. 47)
$\overline{S^N}(x)$	sequência de $S(x)$ em $\overline{\sigma^-}$ que contém os não-vizinhos de x (p. 47)
$\overline{S}_i(x)$	i-ésima célula de $S^N(x)$ em $\overline{\sigma^-}$ (p. 47)
$N_<(S)$	vizinhança à esquerda de S (p. 47)
$N^\ell(S_i(x))$	vizinhança local de $S_i(x)$ (p. 47)
B_G	grafo bipartido formado por uma junção (p. 63)
G_M	grafo obtido de G pela remoção das arestas que estão em B_G mas não fazem parte de M (p. 64)
$x \bmod y$	resto da divisão de x por y (p. 65)

SUMÁRIO

1 INTRODUÇÃO

Em 1852, ao colorir um mapa que representava as divisões administrativas da Inglaterra, o matemático Francis Guthrie percebeu que apenas quatro cores diferentes eram necessárias para distinguir as diferentes regiões. Diante de tal constatação, lançou-a como uma conjectura: "Quatro cores são suficientes para colorir um mapa". Se por um lado essa propriedade não era de grande valia para desenhistas de mapas, por outro, despertou um grande interesse entre os matemáticos, já que se tratava de uma peculiaridade um tanto intrigante; além do mais, sua demonstração foi muito difícil — de fato, ela só ocorreu mais de 100 anos depois, na década de 1960, com o auxílio de computadores [4].

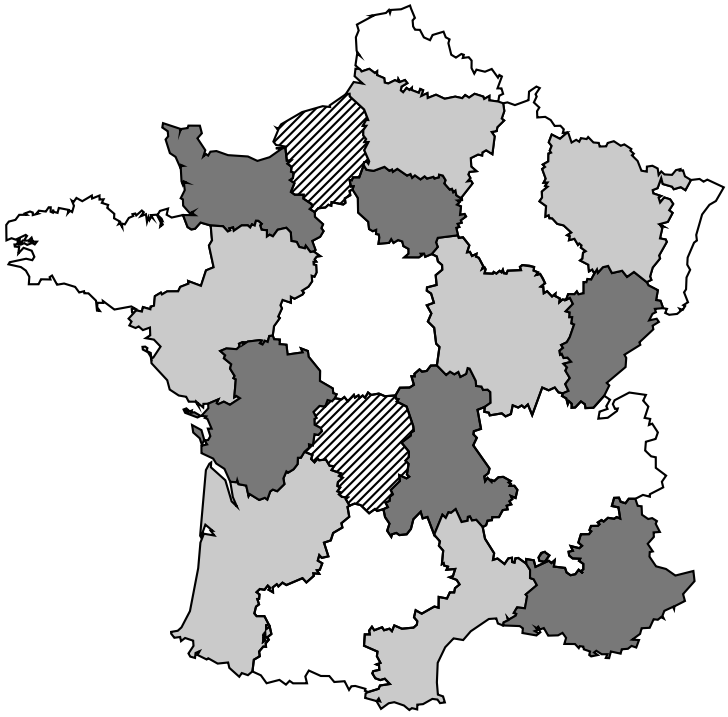


Figura 1.1: Coloração do mapa da França continental [1]

Em razão da possibilidade de modelar esse problema com grafos, passou-se a estudar a coloração dessas estruturas matemáticas. Na próxima seção, apresentaremos a relação entre os problemas de coloração de vértices e de arestas de grafos e, finalmente, os cografos e a coloração de suas arestas.

[1] Adaptado de: http://www.amcharts.com/lib/3/maps/svg/franceLow.svg.

1.1 Apresentação e importância do problema

O problema proposto por Guthrie consistia na seguinte pergunta:

Problema das Quatro Cores (4CORES)**.** *Quantas cores são necessárias para colorir um mapa de tal forma que países fronteiriços tenham cores distintas?*

Para melhor compreensão, convém mencionar que, se a fronteira de duas ou mais regiões for apenas um ponto, elas não são consideradas vizinhas e podem, portanto, levar a mesma cor. A Figura 1.1 contém o mapa das regiões administrativas da França continental. Esse mapa foi escolhido pelo fato de possuir regiões de tamanhos parecidos (o que facilita sua visualização) e também por não poder ser colorido com um número menor de cores.

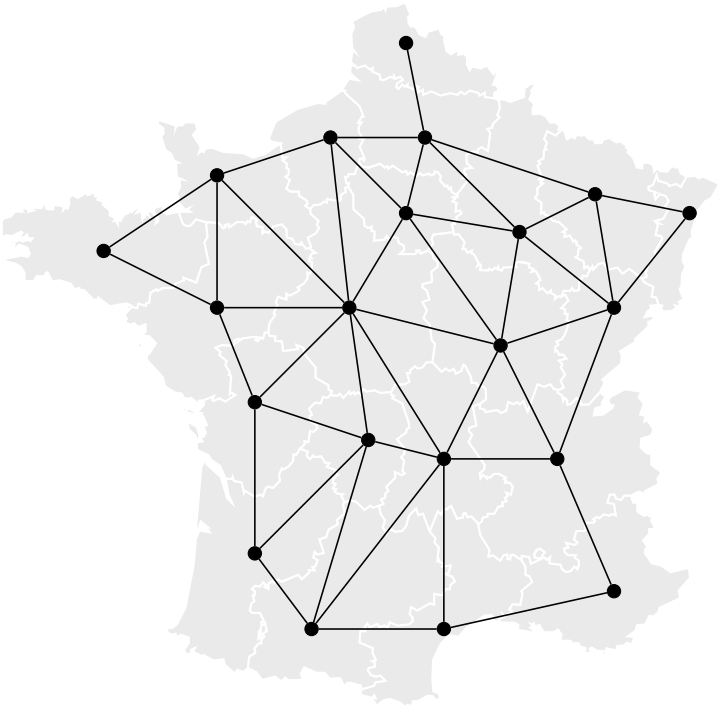


Figura 1.2: Conversão do mapa da França continental para um grafo

Convém transformar 4CORES no Problema da Coloração de Grafos:

Problema da Coloração de Grafos (COLORAÇÃO)**.** *Quantas cores são necessárias para colorir os vértices de um grafo de forma que vértices adjacentes tenham cores distintas?*

Desde já, é importante ressaltar que esses problemas não são equivalentes: a equivalência se dá entre 4CORES e COLORAÇÃO quando este último é restrito a grafos

planares. Contudo, a análise do problema mais geral nos proporcionará uma melhor compreensão para outro problema derivado, visto mais adiante.

A conversão é realizada de tal modo que o grafo correspondente terá um vértice associado a cada região do mapa. Vértices serão adjacentes (ligados por uma aresta) se as respectivas regiões forem vizinhas. Pode-se verificar essa conversão no grafo da Figura 1.2, obtido a partir do mapa da Figura 1.1.

O número mínimo de cores necessárias para colorir os vértices de um grafo G chama-se número cromático e é denotado por $\chi(G)$. Na Figura 1.3a, temos um exemplo de um grafo com número cromático igual a 2 (as cores são representadas por números), porquanto essa é a quantidade de cores suficientes para que sua coloração seja válida. A adição de uma determinada aresta provoca uma mudança na estrutura do grafo suficiente para que seu número cromático passe a ser 3 (Figura 1.3b).

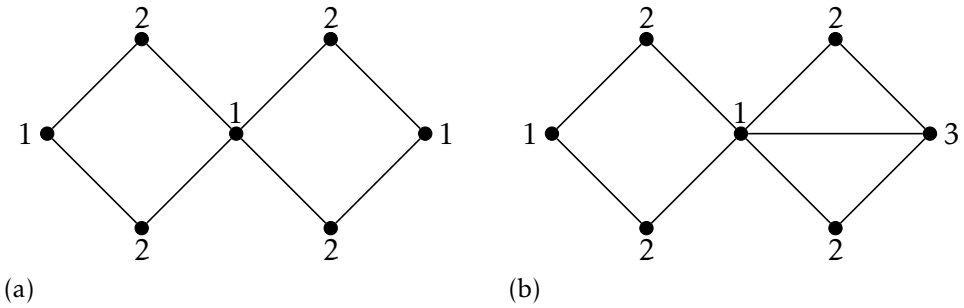


Figura 1.3: Coloração de grafos

A Tabela 1.1 apresenta alguns problemas associados ao número cromático de um grafo. Saber se 2 cores são suficientes (ou seja, verificar se o grafo é bipartido) é um problema que admite solução polinomial. Saber se 3 ou mais cores são suficientes é um problema $\mathcal{NP}$-completo.

Problema	**Classe**
2-Coloração	$\mathcal{P}$
3-Coloração	$\mathcal{NP}$-completo
4-Coloração	$\mathcal{NP}$-completo
k-Coloração ($k > 2$)	$\mathcal{NP}$-completo

Tabela 1.1: Classes de complexidade para problemas de coloração de grafos [9]

Além do Problema da Coloração de Grafos, ainda mais intrigante — conforme justificaremos no Capítulo 3 — é o Problema da Coloração de Arestas:

Problema da Coloração de Arestas (ARESTA-COLORAÇÃO). *Quantas cores são necessárias para colorir as arestas de um grafo de forma que arestas adjacentes tenham cores distintas?*

Analogamente ao problema anterior, neste o objetivo é colorir as arestas de um grafo de modo válido, ou seja, objetivando-se que arestas incidentes nos mesmos vértices (adjacentes) tenham cores distintas. Nesse caso, sendo G o grafo em questão, o número mínimo de cores necessárias chama-se índice cromático e é denotado por $\chi'(G)$. A Figura 1.4a apresenta um exemplo de um grafo com índice cromático igual a 4. É importante destacar desde já que o índice cromático está intimamente relacionado com o grau máximo do grafo (representado por Δ e definido mais precisamente na p. 33). Isso ocorre porque não é possível colorir as arestas com menos do que Δ cores, caso contrário os vértices com esse grau seriam extremos de arestas com a mesma cor (Figura 1.4b). Por outro lado, um importantíssimo resultado encontrado por Vizing [22] coloca $\Delta(G)+1$ como limite superior do índice cromático de qualquer grafo G. Ou seja, só há duas possibilidades para o índice cromático, pois $\Delta(G) \leqslant \chi'(G) \leqslant \Delta(G)+1$. Quando $\chi'(G) = \Delta(G)$, dizemos que G é *Classe 1*; quando $\chi'(G) = \Delta(G)+1$, dizemos que G é *Classe 2*.

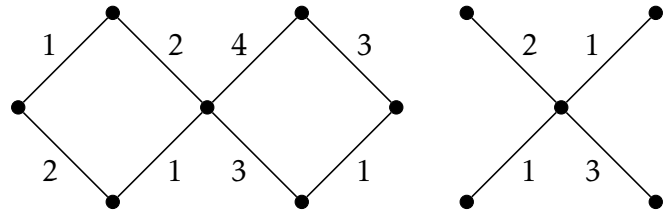


Figura 1.4: Coloração de arestas (a) válida e (b) inválida

Decidir se um grafo G é *Classe 1* é um problema $\mathcal{NP}$-completo — um certificado de tamanho polinomial verificável em tempo polinomial é a própria Δ-coloração (coloração com Δ cores). A seu turno, decidir se o grafo é *Classe 2* é um problema co$\mathcal{NP}$-completo, já que se trata do complementar de um problema $\mathcal{NP}$-completo.

Muito embora descobrir se um grafo é *Classe 2* seja um problema co$\mathcal{NP}$-completo, ainda há a possibilidade de se chegar a um resultado de modo imediato em alguns casos, pois grafos com $m > \Delta(G)\lfloor n/2 \rfloor$ (conhecidos como *overfull*[2] ou so-

[2] Adotamos neste trabalho o termo em inglês, inclusive em palavras compostas como vizinhança-*overfull* e subgrafo-*overfull*, a fim de manter a coerência com os nomes das classes *NO* e *SO* (p. 58).

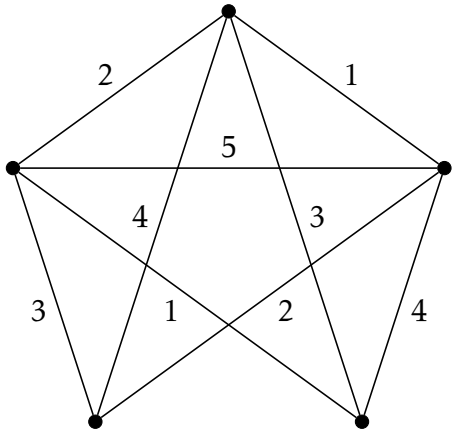


Figura 1.5: Grafo *overfull*

brecarregados) necessariamente são *Classe 2* (Figura 1.5). Ademais, um grafo G que tenha algum subgrafo induzido H que seja um grafo *overfull* e que satisfaça $\Delta(H) = \Delta(G)$ é dito subgrafo-*overfull* e também é *Classe 2* [18].

Nesse contexto, volta-se a atenção a uma classe de grafos para a qual não se sabe se ARESTA-COLORAÇÃO está em $\mathcal{P}$ ou se é $\mathcal{NP}$-completo: os cografos, que emergem naturalmente de diversas áreas [16], e destacam-se pela sua simplicidade e estrutura recursiva. Cografos são grafos que não possuem um P_4 (caminho de 4 vértices) como subgrafo induzido. O exemplo da Figura 1.6a não é um cografo, pois possui o P_4 $axyz$

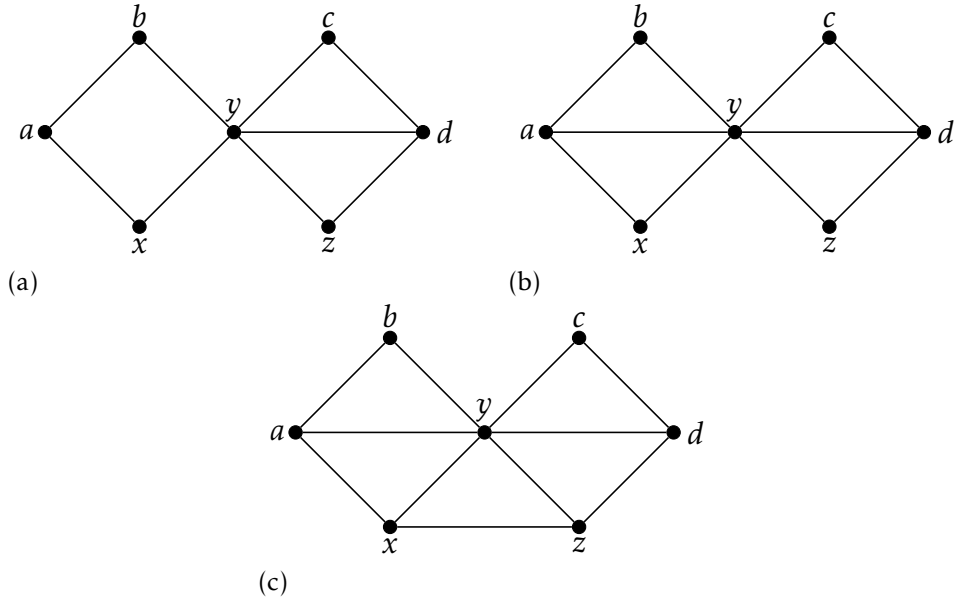


Figura 1.6: Grafos com e sem P_4 induzido

como subgrafo induzido. Ao adicionarmos a aresta ay, obtemos o grafo da Figura 1.6b, que é um cografo (não possui nenhum P_4 como subgrafo induzido). Podemos ver que

a simples adição (ou remoção) de arestas pode transformar um cografo em um não-cografo (ou vice-versa): com a aresta xz, o grafo da Figura 1.6c (p. 31) já não é mais um cografo ($baxz$ é um P_4 induzido).

Em razão de sua característica recursiva, muitos problemas $\mathcal{NP}$-completos, quando restritos a cografos, podem ser resolvidos em tempo polinomial. A título de exemplo, já se sabe que COLORAÇÃO pode ser resolvido em tempo linear [9], assim como o reconhecimento dos cografos [2]. Além disso, suspeita-se que ARESTA-COLORAÇÃO também esteja em $\mathcal{P}$ por conta da Conjectura *Overfull*, a qual afirma que grafos com $\Delta > n/3$ são *Classe 2* se e somente se forem subgrafo-*overfull* [5] — precondição satisfeita pelos cografos conexos, pois já foi demonstrado que possuem $\Delta \geqslant n/2$ [8]. Ademais, cografos podem ser considerados conexos sem perda generalidade, conforme exibido na Proposição 4.6 (p. 62). Note-se ainda que é possível decidir se um grafo com $\Delta(G) \geqslant n/2$ é subgrafo-*overfull* em tempo linear [18], sugerindo assim que ARESTA-COLORAÇÃO restrita a cografos não apenas esteja em $\mathcal{P}$, mas sim em $\mathcal{TIME}(n+m)$. Outra evidência nesse sentido é o fato de que a Conjectura *Overfull* já foi demonstrada para os grafos multipartidos completos (definição na p. 36), que constituem uma conhecida subclasse dos cografos [12].

1.2 Definições preliminares

Ao longo deste trabalho, utilizaremos as seguintes definições:

Definição 1.1. Um *grafo* $G = (V, E)$ é uma estrutura matemática composta por um conjunto de vértices V e um conjunto de arestas E. Todos os grafos considerados neste trabalho são simples, finitos, não dirigidos e sem pesos nas arestas. Além disso:

(i) o conjunto de vértices é denotado por $V(G)$;

(ii) o conjunto de arestas é denotado por $E(G)$;

(iii) toda aresta une dois vértices distintos (seus extremos), que são ditos *adjacentes*;

(iv) duas arestas distintas são adjacentes se incidem no mesmo vértice;

(v) a cardinalidade de $V(G)$ (ou *ordem* de G) é representada por $|V(G)|$ ou n;

(vi) a cardinalidade de $E(G)$ é representada por $|E(G)|$ ou m.

Definição 1.2. Dado um vértice v de um grafo G, o conjunto de vértices que lhe são adjacentes (sua *vizinhança*) é denotado por $N(v)$. Utilizamos as seguintes notações:

(i) o *grau* de v é $d(v) = |N(v)|$;

(ii) para $S \subseteq V(G)$, $d(S) = \sum_{u \in S} d(u)$;

(iii) o *grau máximo* de um grafo G é $\Delta(G) = \max_{v \in V(G)} d(v)$;

(iv) o *grau mínimo* de um grafo G é $\delta(G) = \min_{v \in V(G)} d(v)$;

(v) Δ-*vértice* de G é um vértice v tal que $d(v) = \Delta(G)$;

(vi) Δ-*vértice próprio* de G é um vértice v tal que $d(N(v)) \geqslant \Delta(G)^2 - \Delta(G) + 2$;

(vii) o número de Δ-vértices na vinhança de v é denotado por $d^*(v)$;

(viii) o número de Δ-vértices próprios na vinhança de v é denotado por $d^{**}(v)$;

(ix) se $d(v) = |V(G)| - 1$, v é dito *vértice universal*.

Na Figura 1.7a, $d(x) = 3$ e $d(y) = 2$. O vértice z é o único Δ-vértice, pois $d(z) = \Delta(G) = 4$.

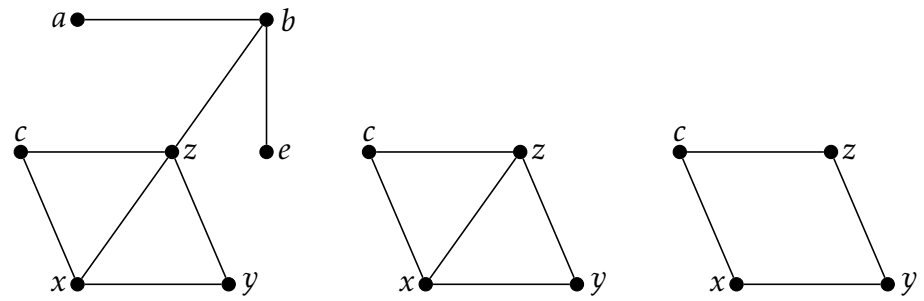


Figura 1.7: Um grafo e dois de seus subgrafos

Definição 1.3. Chama-se *emparelhamento* um conjunto de arestas que não são adjacentes entre si (duas a duas).

Na Figura 1.7a, as arestas ab, cz, xy formam um emparelhamento.

Definição 1.4. *Módulo* é um conjunto de vértices M tal que $(\forall x, y \in M) N(x) \setminus M = N(y) \setminus M$. Ademais, se $\{x, y\}$ for um módulo, x e y são ditos *gêmeos*.

Portanto, tanto vértices isolados quanto o conjunto de todos os vértices de um grafo formam módulos triviais. Na Figura 1.7a (p. 33), os conjuntos de vértices $\{c, x, y\}$, $\{c, y\}$ e $\{a, e\}$ são módulos. Já o conjunto $\{c, x, y, z\}$ não é um módulo, pois $zb \in E(G)$ mas $cb, xb, yb \notin E(G)$.

Definição 1.5. Um *subgrafo* G' de um grafo $G(V, E)$ é um grafo $G'(V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$. Destacam-se os seguintes subgrafos de G:

(i) Seja $S \subseteq V(G)$, o *subgrafo induzido* $H = G[S]$ é o subgrafo de G tal que $V(H) = S$ e, para todo x e todo y em $S, xy \in E(H)$ se e somente se $xy \in E(G)$.

(ii) O subgrafo induzido pelos Δ-vértices de G, indicado por $\Lambda[G]$, é chamado de núcleo de G.

(iii) O subgrafo induzido pelos Δ-vértices e seus vizinhos, indicado por $N[\Lambda[G]]$, é chamado de seminúcleo de G.

O grafo da Figura 1.7b (p. 33) é um subgrafo induzido do grafo da Figura 1.7a (p. 33). Já o grafo da Figura 1.7c (p. 33), embora seja um subgrafo do grafo da Figura 1.7a (p. 33), não é induzido em razão da ausência da aresta xz.

Definição 1.6. Dados dois grafos G_1 e G_2 com conjuntos disjuntos de vértices, definimos as seguintes operações:

(i) Sua *união disjunta* é o grafo

$$G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2)). \tag{1.7}$$

(ii) Sua *junção* é o grafo

$$G_1 * G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2) \cup (V(G_1) \times V(G_2))). \tag{1.8}$$

A Figura 1.8a (p. 35) representa a união entre G_1 e G_2, sendo $V(G_1) = \{a, b, c\}$ e $V(G_2) = \{x, y\}$. Por sua vez, a Figura 1.8b (p. 35) mostra a junção entre os mesmos G_1 e G_2; nela, pode-se ver claramente que a junção é o grafo obtido pela união disjunta acrescida de todas as possíveis arestas que têm uma extremidade em G_1 e a outra em G_2. Além disso, é fácil perceber que união disjunta sempre resulta em um grafo desconexo, ao passo que a junção sempre resulta em um grafo conexo.

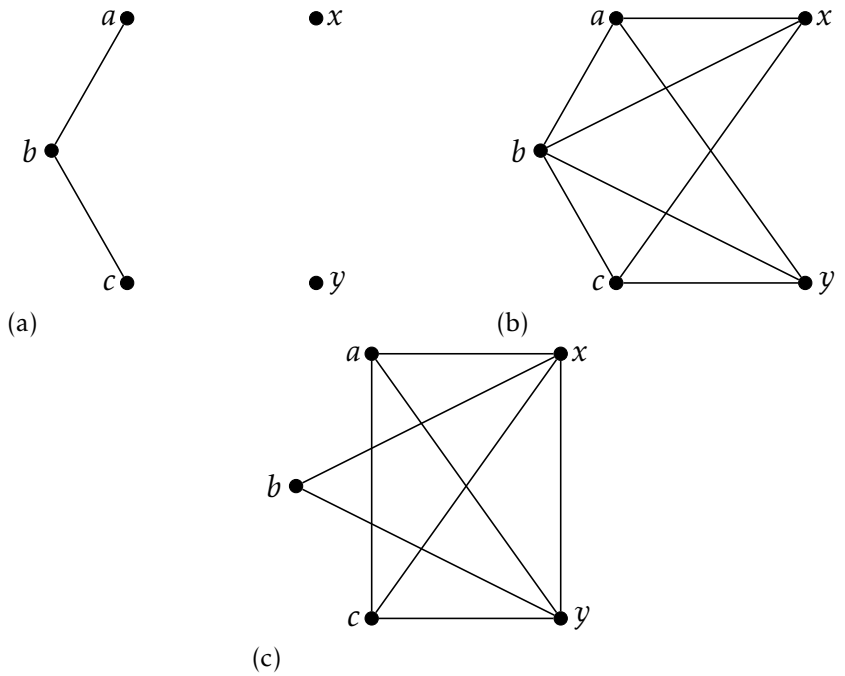


Figura 1.8: Operações de união disjunta, junção e complementação de grafos

Definição 1.9. Dado o grafo G, seu *complemento* é o grafo

$$\overline{G} = \left(V(G), \binom{V(G)}{2} \setminus E(G) \right). \tag{1.10}$$

Ou seja, $xy \in E(G)$ se e somente se $xy \notin E(\overline{G})$. A Figura 1.8c representa o complemento do grafo da Figura 1.8a.

Definição 1.11. *Passeio* é uma sequência alternante de vértices e arestas, que se inicia e termina em um vértice, e na qual cada aresta é incidente no vértice que a antecede de imediato e no vértice que a sucede de imediato. *Caminho* é um passeio no qual todos os vértices e todas as arestas são distintos. Um caminho com n vértices é representado por P_n. *Ciclo* é um passeio no qual todos os vértices e todas as arestas são distintas, à exceção do último vértice, que deve ser igual ao primeiro. Um ciclo com n vértices é representado por C_n. Uma aresta que une dois vértices de um ciclo sem fazer parte dele é chamada de *corda*.

Definição 1.12. Grafo *completo* é o grafo no qual todos os vértices são adjacentes dois a dois. O grafo completo com n vértices é denotado por K_n. Sobre esses conceitos, podemos ainda tecer as seguintes observações:

(i) O grafo com apenas um vértice é também completo (K_1).

(ii) A complementação de um grafo completo resulta em um grafo sem arestas e vice-versa.

(iii) Um conjunto de vértices que induz um subgrafo completo é denominado *clique*.

(iv) Um conjunto de vértices que induz um subgrafo sem arestas é denominado *conjunto independente*.

Definição 1.13. Um grafo *multipartido completo* G é um grafo que possui uma partição de $E(G)$ na qual cada parte é um conjunto independente e dois vértices em partes distintas sempre são adjacentes.

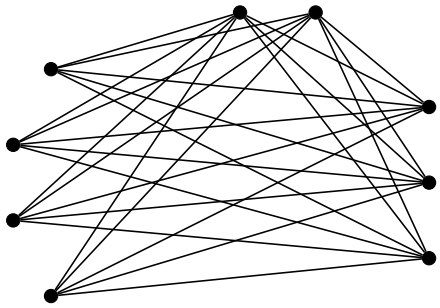


Figura 1.9: Um grafo multipartido completo

Definição 1.14. Os vértices externos de um caminho são os vértices de início e de fim do caminho. Os demais vértices desse caminho são chamados de vértices internos.

Também utilizaremos o conceito de árvore neste trabalho, exemplificada na Figura 1.10.

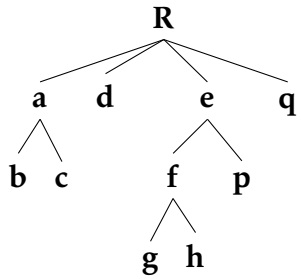


Figura 1.10: Uma árvore

Definição 1.15. Árvore é um grafo conexo e acíclico, tal como mostra a Figura 1.10. Todas as árvores aqui tratadas são enraizadas, ou seja, possuem um vértice especial denominado raiz (ou R). Além disso:

(i) Precisamos por vezes fazer referências a caminhos das folhas até a raiz, e o fazemos da maneira usual. Denominamos o caminho de uma folha h até a raiz R por P_{hR}. Na Figura 1.10, o caminho P_{hR} é a sequência h, f, e, R.

(ii) Dados dois vértices x e y, seu *ancestral comum mais baixo*, denotado por $\text{lca}(x, y)$ [3], é o primeiro vértice em comum entre as sequências P_{xR} e P_{yR}. Na Figura 1.10, temos que $\text{lca}(g, p) = e$ e $\text{lca}(b, d) = R$.

[3] Também conhecido como ancestral comum mais próximo, ou nca.

2 COGRAFOS

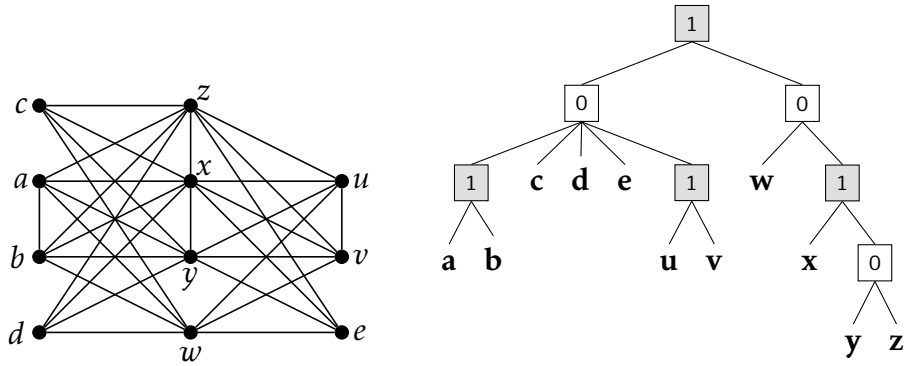
A classe dos cografos (exemplificados na Figura 2.1a), subclasse dos P_4-redutíveis e superclasse dos multipartidos completos, é a menor classe de grafos que contém o K_1 e é fechada sob as operações de união disjunta e complementação [3]. De forma análoga, a complementação pode ser substituída pela junção, já que, dados dois grafos G_1 e G_2, $G_1 * G_2 = \overline{\overline{G_1} * \overline{G_2}}$. Assim, pode-se adotar um método recursivo para a definição construtiva de cografos.

Definição 2.1 ([6])**.** Um cografo pode ser construído de maneira recursiva da seguinte forma:

(i) K_1 é um cografo;

(ii) se G_1 e G_2 são cografos, $G_1 \cup G_2$ é um cografo;

(iii) se G_1 e G_2 são cografos, $G_1 * G_2$ é um cografo.



(a) (b)

Figura 2.1: Um cografo e sua coárvore

Além da definição construtiva, pode-se também defini-los de forma desconstrutiva: cografos são grafos que podem ser reduzidos a vértices isolados por meio da complementação recursiva de suas componentes conexas (daí a razão do nome cografo, que advém do inglês *cograph — complement reducible graph*, ou grafo redutível por complementação). Ademais, nenhum cografo possui um P_4 como subgrafo induzido, fato que constitui uma caracterização por subgrafo proibido (por consequência também não possuem nenhum C_n induzido para $n > 4$). A Figura 2.1a apresenta um exemplo de um cografo com 11 vértices, que ilustra a inexistência de um P_4 como subgrafo induzido.

Levando-se em consideração a definição construtiva, associa-se a cada cografo uma árvore, chamada *coárvore*, que rotula em seus nós internos as operações e tem como folhas os vértices do grafo [2]. Ela traz em si todas as informações necessárias para a construção do respectivo cografo. A título de exemplo, a coárvore da Figura 2.1b (p. 39) representa as operações realizadas para a construção do cografo da Figura 2.1a (p. 39). Cada nó interno é uma raiz de uma ou mais subárvores que, por sua vez, são coárvores dos subgrafos induzidos pelos vértices representados pelas respectivas folhas. O nó indica, por meio do rótulo, a operação realizada entre os subgrafos a que correspondem as subárvores induzidas pelos seus filhos (0 é união disjunta e 1 é junção). Se todos os caminhos da raiz da coárvore até as folhas passarem por nós com rótulos alternantes (entre 0 e 1) e cada nó tiver ao menos dois filhos (exceto quando $n < 2$), ter-se-á uma coárvore canônica. A cada cografo corresponde uma e somente uma coárvore canônica [8].

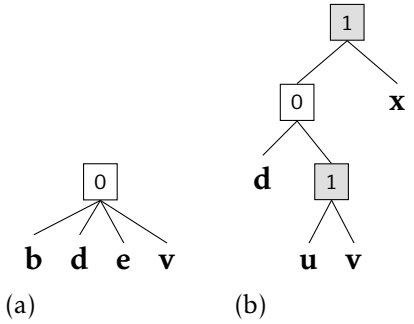


Figura 2.2: Coárvores de alguns subgrafos induzidos do cografo da Figura 2.1a

Cumpre destacar que, dados dois vértices distintos v_i e v_j, pode-se verificar por meio da coárvore se eles são vizinhos, já que $\text{lca}(v_i, v_j) = 1$ se e somente se $v_i, v_j \in E(G)$. Diante de tal constatação, fica claro que as folhas de qualquer subárvore induzida pelos nós internos correspondem a um *módulo*, fato que será de grande valia para o reconhecimento de cografos descrito na Seção 2.1.

A coárvore de um cografo G é denotada por $T(G)$. Consideramos que uma coárvore $T(G)$ possui r ramos (em outras palavras, a raiz de $T(G)$ possui r filhos). Sabemos que cada um desses r filhos induz uma subárvore, a qual possui um ou mais vértices. Designamos por $\alpha(i)$ o conjunto de vértices da subárvore induzida pelo ramo i de $T(G)$ ($1 \leqslant i \leqslant r$). Além disso, denotamos a cardinalidade desses conjuntos da seguinte forma: $a(i) = |\alpha(i)|$. É importante notar que $\{\alpha(1), \alpha(2), \alpha(3) \ldots \alpha(r)\}$ é uma partição de $V(G)$ (chamada de α-partição). A Figura 2.3 (p. 41) mostra uma coárvore e sua

α-partição.

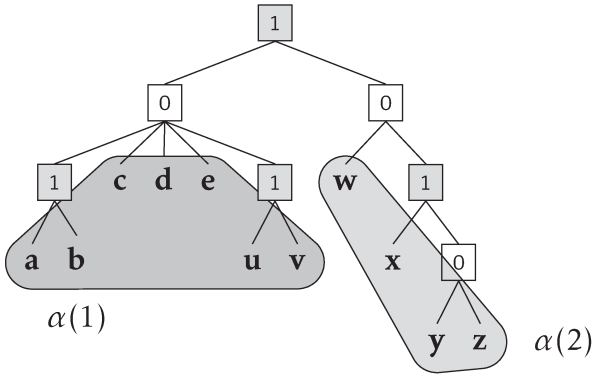


Figura 2.3: A α-partição de uma coárvore

Precisamos fazer referência a algumas subárvores bastante específicas de uma coárvore T. Assim, considerando z um vértice qualquer de T (uma de suas folhas) e P_{zR} o caminho de z à raiz R, designamos por T^z_{ji} a i-ésima subárvore induzida de T com rótulo j, sendo excluídos da subárvore os filhos dos vértices de P_{zR}. A Figura 2.4 mostra um exemplo dessa construção. Pode-se perceber que o vértice z, na figura referida, não induz esse tipo de subárvore, já que ele é folha e, nessa condição, não possui rótulo.

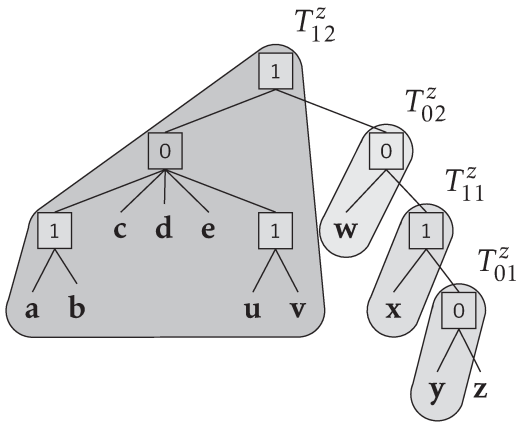


Figura 2.4: Exemplos de subárvores T^z_{ji}

2.1 O reconhecimento de cografos[4]

Saber se um dado grafo pertence a uma determinada família é um problema importante, pois, uma vez classificado, pode-se lançar mão de uma série de características inerentes à dita classe em prol da otimização de algoritmos que se apliquem

[4] A fim de não poluir o texto, resultados secundários ou muito técnicos foram movidos para um apêndice próprio, encontrado a partir da p. 79.

nesse grafo. No caso do presente trabalho, ao reconhecer um cografo, teremos a possibilidade de aplicar o conhecimento de sua estrutura para, de alguma forma, extrair resultados acerca da coloração das arestas (tratada no Capítulo 3) de forma mais imediata. De outro norte, o algoritmo apresentado nesta seção (Algoritmo 2.1), além de reconhecer cografos em tempo linear, devolve a coárvore em caso positivo ou exibe os vértices que formam um P_4 induzido em caso negativo.[5]

Todos os algoritmos, lemas, teoremas, corolários, observações e definições expostos nesta seção, embora por vezes ligeiramente adaptados em alguns pontos, são oriundos de um artigo de Bretscher [2], que apresenta o reconhecimento de cografos em tempo linear. Os próprios autores mencionam a existência de outros algoritmos mais antigos que também realizam o reconhecimento de cografos; contudo, esses algoritmos demandam estruturas de dados ou técnicas de algoritmos mais complexas [2]. O reconhecimento, tal como exposto no Algoritmo 2.1, será feito com uma passada do algoritmo LexBFS sobre o grafo, seguida da passada de uma variante, chamada LexBFS$^-$. Em seguida, para o caso positivo, é construída a coárvore do grafo.[6]

Reconheça-cografo(G):
Entrada: *um grafo G.*
Saída: *a coárvore T se G for um cografo ou* **F** *caso contrário.*

1 seja τ uma permutação qualquer de $V(G)$;
2 σ = LexBFS(G, τ);
3 $\overline{\sigma}^-$ = LexBFS$^-$(G, σ);
4 **se** σ e $\overline{\sigma}^-$ obedecem à PSV (definição na p. 48) em G e $\overline{G}$ respectivamente:
5 **devolva** Construa-coárvore(G);
6 **senão**:
7 **devolva F**.

Algoritmo 2.1: Reconhecimento de cografos

LexBFS é uma abreviatura do sintagma "busca em largura lexicográfica". Ele recebe como entrada um grafo e uma permutação inicial de seus vértices e devolve uma outra permutação que obedece à disposição de vértices comum às buscas em largura (decorrente da inserção dos vértices em uma fila, de onde são retirados para visitação). O adjetivo "lexicográfica" se deve a uma das possíveis implementações desse algoritmo, que tarja os vértices para estabelecer o critério de desempate desejado. Lança-

[5] A exibição do P_4 induzido não se mostrou tão relevante para os objetivos do presente trabalho, portanto foi suprimida.
[6] A construção da coárvore é abordada em um apêndice próprio (p. 83).

remos mão de uma implementação diversa, mas equivalente, em razão das estruturas intermediárias que ela proporciona.

A variante do algoritmo LexBFS que utilizaremos (Algoritmo 2.2) funciona a partir da divisão do conjunto de vértices em *células*. No presente contexto, célula é simplesmente uma sequência ordenada de vértices. Cria-se uma lista de células L que, no começo do algoritmo, é inicializada com apenas uma célula, a qual contém todos os vértices do grafo (linha 1). A célula não-vazia mais à esquerda dessa lista é tomada em cada passo do algoritmo, e dela é extraído um vértice, conhecido como pivô (designado por α), que é inserido na permutação de saída (linhas 4 a 6). A partir daí, analisa-se a vizinhança do pivô, e as demais células, uma a uma, são divididas em duas: em uma célula ficam os vizinhos do pivô e na outra os não-vizinhos (linhas 9 e 10). A célula dos vizinhos sempre é posicionada imediatamente à esquerda da célula dos não-vizinhos (linha 11). Para fins de otimização, essa divisão não ocorrerá se a célula for composta inteiramente por vizinhos ou por não-vizinhos do pivô (linha 8).

LexBFS(G, τ):
Entrada: *um grafo G e uma permutação τ dos vértices de G.*
Saída: *uma permutação σ dos vértices de G.*
1 seja L uma lista de células inicializada com a célula $V(G)$;
2 seja σ uma sequência de vértices inicialmente vazia;
3 **enquanto** L tiver uma célula não-vazia, **faça**:
4 seja X a primeira célula não-vazia de L:
5 extraia um vértice α de X segundo a ordem τ;
6 adicione α ao final de σ;
7 **para toda** célula Y de L, **faça**:
8 **se** $N(\alpha) \cap Y \neq \varnothing$ e $N(\alpha) \cap Y \neq Y$:
9 crie uma nova célula Z;
10 mova todos os vértices vizinhos de α de Y para Z;
11 insira Z em L imediatamente à esquerda de Y;
12 **devolva** σ.

Algoritmo 2.2: Permutação LexBFS

A permutação resultante de uma passada da LexBFS é designada por σ. A posição de um vértice na permutação é denotada pelo mesmo símbolo, mas utilizado como uma função: $\sigma(v) = i$ indica que v é o i-ésimo vértice de σ. De forma sucinta, indicamos que $\sigma(v) < \sigma(w)$ por $v \prec_\sigma w$. A função inversa toma a posição e devolve o vértice: $\sigma^{-1}(i) = v$ indica que o i-ésimo vértice de σ é v. Em algumas situações, é necessário *congelar* a execução do algoritmo e analisar seu estado naquele instante. Por exemplo,

após um vértice α ser tomado como pivô, denotamos por $N'(\alpha)$ o conjunto de seus vizinhos ainda não enumerados em σ (ainda não tomados como pivôs). Um passo-a-passo da execução do algoritmo LexBFS sobre o cografo da Figura 2.1a é exibido na Tabela 2.1, utilizando como permutação de entrada a seguinte sequência:

$$\tau : \mathbf{z\ d\ y\ e\ u\ v\ b\ c\ x\ a\ w} \tag{2.2}$$

$\sigma(\alpha)$	α	$N'(\alpha)$	Lista de células L
			z d y e u v b c x a w
1	**z**	{**d**, **e**, **u**, **v**, **b**, **c**, **x**, **a**}	**d e u v b c x a** | **y w**
2	**d**	{**x**, **y**, **w**}	**x** | **e u v b c a** | **y w**
3	**x**	{**e**, **u**, **v**, **b**, **c**, **a**, **y**}	**e u v b c a** | **y** | **w**
4	**e**	{**y**, **w**}	**u v b c a** | **y** | **w**
5	**u**	{**v**, **y**, **w**}	**v** | **b c a** | **y** | **w**
6	**v**	{**y**, **w**}	**b c a** | **y** | **w**
7	**b**	{**a**, **y**, **w**}	**a** | **c** | **y** | **w**
8	**a**	{**y**, **w**}	**c** | **y** | **w**
9	**c**	{**y**, **w**}	**y** | **w**
10	**y**	{}	**w**
11	**w**	{}	

Tabela 2.1: Exemplo de execução do algoritmo LexBFS

Conforme já adiantado, são necessárias duas passadas da LexBFS para que seja possível o reconhecimento do cografo (linhas 2 e 3 do Algoritmo 2.1 — p. 42). A segunda passada deve ocorrer sobre $\overline{G}$, além de utilizar como permutação de entrada a permutação de saída da primeira passada (σ). Contudo, não é necessário computar $\overline{G}$: basta que se use um novo algoritmo, chamado de LexBFS$^-$, obtido pela seguinte alteração na linha 11 do Algoritmo 2.2 (p. 43):

---

11 insira Z em L imediatamente à direita de Y;

Ora, posicionar os vizinhos à direita dos não-vizinhos é o mesmo que posicionar

os vizinhos em $\overline{G}$ à esquerda dos não-vizinhos em $\overline{G}$. Ou seja, o algoritmo LexBFS$^-$ realiza a mesma permutação, mas dessa vez sobre o complemento de G. A permutação de uma passada LexBFS$^-$ é representada por $\overline{\sigma}^-$. A Tabela 2.2, por sua vez, exibe o passo-a-passo da execução do algoritmo LexBFS$^-$ sobre o cografo da Figura 2.1a (p. 39), utilizando como permutação de entrada a permutação LexBFS construída anteriormente, qual seja:

$$\sigma : \mathbf{z\ d\ x\ e\ u\ v\ b\ a\ c\ y\ w} \tag{2.3}$$

$\overline{\sigma}(\alpha)$	α	$\overline{N}'(\alpha)$	Lista de células L
			z d x e u v b a c y w
1	**z**	{**y**, **w**}	**y w** | **d x e u v b a c**
2	**y**	{**w**}	**w** | **d x e u v b a c**
3	**w**	{**x**}	**x** | **d e u v b a c**
4	**x**	{}	**d e u v b a c**
5	**d**	{**e**, **u**, **v**, **b**, **a**, **c**}	**e u v b a c**
6	**e**	{**u**, **v**, **b**, **a**, **c**}	**u v b a c**
7	**u**	{**b**, **a**, **c**}	**b a c** | **v**
8	**b**	{**c**, **v**}	**c** | **a** | **v**
9	**c**	{**a**, **v**}	**a** | **v**
10	**a**	{**v**}	**v**
11	**v**	{}	

Tabela 2.2: Exemplo de execução do algoritmo LexBFS$^-$

Duas características de uma permutação LexBFS são bastante úteis para o reconhecimento de cografos. A primeira estabelece que toda permutação LexBFS obedece à CQP (Condição dos Quatro Pontos), conforme esboçado na Figura 2.5 (p. 46).

Propriedade 2.4 (Condição dos Quatro Pontos — em inglês FPC (*Four Point Condition*))**.** *Sejam σ uma permutação LexBFS do grafo G e $x, y, z \in V(G)$ tais que $x \prec_\sigma y \prec_\sigma z$. Se $xy \notin E(G)$ e $xz \in E(G)$, então existe um vértice $w \in V(G)$ tal que $w \prec_\sigma x$, $wy \in E(G)$ e $wz \notin E(G)$ (Lema A.1, p. 79)*

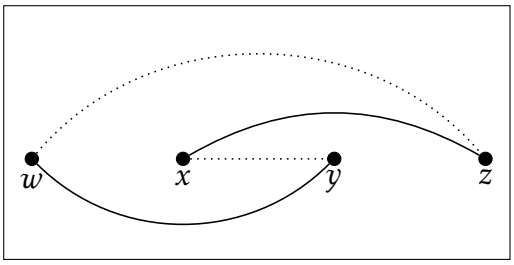


Figura 2.5: A Condição dos Quatro Pontos

Por outro lado, toda permutação LexBFS de um cografo G é livre de *guarda-chuvas*, conforme mostra a Figura 2.6.

Propriedade 2.5. *Sejam σ uma permutação LexBFS do cografo G e $x, y, z \in V(G)$ tais que $x \prec_\sigma y \prec_\sigma z$. Não pode ocorrer o caso em que $xy \notin E(G)$, $yz \notin E(G)$ e $xy \in E(G)$ (Lema A.2, p. 79).*

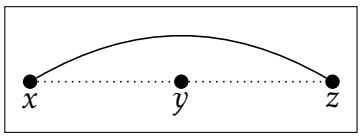
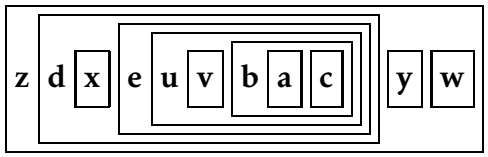


Figura 2.6: Um *guarda-chuva*

Ademais, é preciso levar em conta também os conceitos de células e *fatias*, que fazem parte das estruturas construídas ao longo da execução do algoritmo. No instante em que o vértice x foi escolhido como pivô, não fosse a ordem τ passada como parâmetro para o algoritmo, qualquer outro vértice da mesma célula poderia ter sido escolhido em seu lugar. Esses outros vértices só estão na mesma célula em que x se encontra porque, até então, são gêmeos de x no que diz respeito aos vértices já enumerados (em outras palavras, sejam a e b dois elementos da mesma célula no instante que o pivô x dessa célula é escolhido e c um vértice qualquer já enumerado, $ac \in E(G)$ se e somente se $bc \in E(G)$). A cada vértice tomado como pivô, portanto, corresponde um conjunto de vértices *empatados* (afora o critério de desempate ditado por τ, por óbvio) ao qual damos o nome de *fatia*. Podemos visualizar a sequência de *fatias* de nossa ordenação em (2.6) (p. 47). Observe-se que cada vértice dá início a uma (e somente uma) nova *fatia* e pode haver *fatias* umas dentro das outras. Dizemos que a *fatia* interna é uma *subfatia* da externa.

σ : 

(2.6)

Dados um vértice x e uma permutação LexBFS σ tais que $x \in \sigma$, representamos a *fatia* que se inicia nesse vértice por $S(x)$. Além disso, dentro de $S(x)$ há duas importantes sequências de vértices (possivelmente vazias). A primeira, denotada por $S^A(x)$, é uma *subfatia* formada pelos vértices de $S(x)$ que são adjacentes a x. A outra, denotada por $S^N(x)$, é formada pelos demais vértices (ou seja, não-vizinhos) exceto o próprio x, que não está em $S^A(x)$ nem em $S^N(x)$. Quanto à sequência dos não-vizinhos ($S^N(x)$), é ainda subdividida em células da seguinte forma: assim que o o último vértice de $S^A(x)$ tiver sido utilizado como pivô, os demais vértices de $S(x)$ — prestes a serem tomados como pivôs — estarão espalhados em células de $S^N(x)$ denotadas por $S_1(x)$, $S_2(x)$, $S_3(x)$... Essas células são chamadas de x-células e possuem as seguintes propriedades:

(i) nenhum de seus elementos (vértices) é adjacente a x;

(ii) todos os seus elementos possuem a mesma vizinhança em $S^A(x)$ (por óbvio, já que são enumerados após essa célula).

Denotamos por x_i o primeiro vértice da x-célula $S_i(x)$. A Tabela 2.3 (p. 48) exibe a *fatia* $S^A(\alpha)$ e as α-células (células de $S^N(\alpha)$) para cada pivô α do passo-a-passo realizado na Tabela 2.1 (p. 44).

No caso de uma permutação LexBFS$^-$, utilizamos uma notação análoga. A *fatia* que se inicia no vértice x é denotada por $\overline{S}(x)$. A *subfatia* relativa aos vizinhos de x é indicada por $\overline{S^A}(x)$ e a subsequência de não-vizinhos, por $\overline{S^N}(x)$. Por sua vez, as x-células são denotadas por $\{\overline{S}_1(x), \overline{S}_2(x), \overline{S}_3(x), \ldots\}$. A Tabela 2.4 (p. 49) exibe a *fatia* $\overline{S^A}(\alpha)$ e as α-células para cada pivô α do passo-a-passo realizado na Tabela 2.2 (p. 45).

Definição 2.7. Seja S um conjunto de vértices consecutivos em uma permutação LexBFS σ e x o primeiro vértice de S com respeito a σ. A vinhança à esquerda de S é definida por:

$$N_<(S) = \{y \in \sigma : y \prec_\sigma x \text{ e } \forall z \in S, y \in N(z)\} \tag{2.8}$$

Definição 2.9. Dada uma permutação LexBFS σ e um vértice x, a vinhança local da x-célula $S_i(x)$, denotada por $N^\ell(S_i(x))$, é o conjunto de vértices de $S(x)$ que obedecem às seguintes regras:

$\sigma(\alpha)$	α	$S(\alpha)$	$S^A(\alpha)$	$S^N(\alpha)$
1	**z**	**z d x e u v b a c y w**	**d x e u v b a c**	**y** | **w**
2	**d**	**d x e u v b a c**	**x**	**e u v b a c**
3	**x**	**x**	∅	∅
4	**e**	**e u v b a c**	∅	**u v b a c**
5	**u**	**u v b a c**	**v**	**b a c**
6	**v**	**v**	∅	∅
7	**b**	**b a c**	**a**	**c**
8	**a**	**a**	∅	∅
9	**c**	**c**	∅	∅
10	**y**	**y**	∅	∅
11	**w**	**w**	∅	∅

Tabela 2.3: Células de cada S^N de uma passada de LexBFS

(i) ocorrem antes de x_i;

(ii) são vizinhos de ao menos um vértice em $S_i(x)$.

É importante destacar o fato de que a vizinhança local não exige que a adjacência exista em relação a todos os vértices da célula. Isso ocorre porque uma célula nem sempre é uma *fatia*: quando não se trata de um cografo, ela pode ser uma união de *fatias*, e nesse caso alguns vizinhos de células anteriores provocarão essa cisão da célula. No caso da vinhança à esquerda (Definição 2.7, p. 47), aí sim há a necessidade de que a vizinhança ocorra com relação a todos os vértices da sequência. Podemos, portanto, observar que os cografos têm uma certa peculiaridade no que tange à relação entre vizinhança local e vizinhança à esquerda:

Observação 2.10. *No caso dos cografos, em razão do Lema A.3 (p. 79), temos que* $N^\ell(S_i(x)) = N_<(S_i(x)) \cap S^A(x)$.

Estimulada pela Observação 2.10, a seguinte definição merece destaque pelo fato de que é uma propriedade de permutações LexBFS:

$\overline{\sigma}(\alpha)$	α	$\overline{S}(\alpha)$	$\overline{S^A}(\alpha)$	$\overline{S^N}(\alpha)$
1	**z**	**z y w x d e u b c a v**	**y w**	**x** | **d e u b c a v**
2	**y**	**y w**	**w**	∅
3	**w**	**w**	∅	∅
4	**x**	**x**	∅	∅
5	**d**	**d e u b c a v**	**e u b c a v**	∅
6	**e**	**e u b c a v**	**u b c a v**	∅
7	**u**	**u b c a v**	**b c a**	**v**
8	**b**	**b c a**	**c**	**a**
9	**c**	**c**	∅	∅
10	**a**	**a**	∅	∅
11	**v**	**v**	∅	∅

Tabela 2.4: Células de cada $\overline{S^N}$ de uma passada de LexBFS$^-$

Definição 2.11 (Propriedade do Subconjunto da Vizinhança)**.** Dizemos que uma permutação LexBFS obedece à PSV (Propriedade do Subconjunto da Vizinhança — *Neighbourhood Subset Property* em inglês) quando

$$\forall v \, \forall i < j \text{ tais que } S_j(v) \neq \varnothing, N^\ell(S_j(v)) \subsetneq N^\ell(S_i(v))$$

O Teorema 2.12, a seu turno, sela a a equivalência entre cografos e grafos com permutações LexBFS e LexBFS$^-$ que obedecem à PSV, de forma que poderemos usá-la no reconhecimento da classe.

Teorema 2.12. *Sejam* σ=LexBFS(G,τ) *para uma permutação* τ *qualquer de* $V(G)$ *e* $\overline{\sigma}^-$=LexBFS$^-(G,\sigma)$*. G é um cografo se e somente se* σ *e* $\overline{\sigma}^-$ *obedecem à PSV.*

Demonstração. ($\Rightarrow$) Suponhamos que G seja um cografo (necessariamente $\overline{G}$ também será). Nesse caso, o Lema A.3 (p. 79) aliado à Observação 2.10 (p. 48), implica que σ e $\overline{\sigma}^-$ obedecem à PSV.

($\Leftarrow$) Vamos assumir que G não é um cografo para, a partir daí, concluir que σ ou $\overline{\sigma}^-$ não obedecem à PSV. Em tal contexto, G possui um P_4 induzido. Denotamos por w o primeiro vértice de σ que pertence a um P_4 induzido. Seja $p = abcd$ um P_4

induzido qualquer que contenha o vértice w (nesse caso, $\overline{G}$ possui o P_4 induzido $\overline{p}$ — os extremos de p são os vértices internos de $\overline{p}$ e vice-versa). Denotamos a *fatia* mínima que contém $\overline{p}$ por $\overline{S}(v)$ e passamos à análise dos três possíveis casos:

(i) Se o vértice v for um dos extremos de $\overline{p}$, pelo Lema A.10 (p. 81) concluímos que $\overline{\sigma}^{-}$ não atende à PSV.

(ii) Se o vértice v for um dos vértices internos de $\overline{p}$ (sem perda de generalidade, seja $v = a$), então, pela Observação A.7 (p. 80), o vértice a (um extremo de p) é o primeiro vértice de p em σ. Denotamos por $S(u)$ a *fatia* mínima que contém p em σ. Se $u = w = a$, então, pelo Lema A.10 (p. 81), concluímos que σ não atende à PSV. Caso contrário, u não pode pertencer a nenhum P_4 induzido (já que ele ocorre antes de p e, por consequência, antes de qualquer P_4 induzido). Pelo Lema A.11 (p. 82), concluímos que σ não atende à PSV.

(iii) Se o vértice v não pertence a $\overline{p}$, então, pela Observação A.7 (p. 80), o vértice v ocorre antes de p em σ. Sabemos, portanto, que v não pertence a P_4 induzido algum. Logo, aplicamos o Lema A.11 (p. 82) e então concluímos que $\overline{\sigma}^{-}$ não atende à PSV. ♦

Podemos construir o Algoritmo Teste-PSV (Algoritmo 2.3) para testar se uma permutação LexBFS (ou LexBFS^{-}) σ atende ou não à PSV. Além disso, pode-se demonstrar que tal algoritmo é executado em tempo linear (Lema 2.13, p. 50).

Teste-PSV(v):
Entrada: *o vértice v de uma fatia $S(v)$ de uma permutação LexBFS σ.*
Saída: **V** *se σ atende à PSV ou* **F** *caso contrário.*

```
1  para cada i tal que S_i(v), S_{i+1}(v) ∈ S^N(v);
2    se |N^ℓ(S_i(v))| < |N^ℓ(S_{i+1}(v))|, devolva F;
3    j ← 1 e k ← 1;
4    enquanto k ⩽ |N^ℓ(S_{i+1}(v))|:
5      enquanto N^ℓ(S_i(v))[j] ≠ N^ℓ(S_{i+1}(v))[k]:
6        j ← j + 1;
7        se j > |N^ℓ(S_i(v))|, devolva F;
8      k ← k + 1;
9  devolva V.
```

Algoritmo 2.3: Teste da PSV para uma permutação LexBFS

Lema 2.13. *Seja* σ=LexBFS(G,τ) *para uma permutação* τ *qualquer de* $V(G)$ *e dadas as vizinhanças locais* $N^{\ell}(S_i(v))$ *associadas a cada* $S_i(v)$ *de cada vértice* v *de* σ*, a PSV pode ser verificada em* $\mathcal{O}(|V(G)|+|E(G)|)$.

Demonstração. É fácil constatar que o Algoritmo 2.3 (p. 50) verifica a PSV para a permutação σ, pois, para cada vértice v, passa por cada par de conjuntos $N^{\ell}(S_i(v))$ e $N^{\ell}(S_{i+1}(v))$ e realiza um teste de inclusão. Nesse caso, o teste é feito apenas nas vizinhanças locais de células adjacentes duas a duas, valendo-se da transitividade da relação de inclusão. Considerando que cada vizinhança local é varrida no máximo 2 vezes, a complexidade de tempo é linear no tamanho total de todas as vizinhanças locais. Precisamos, portanto, saber que tamanho é esse. Já que há no máximo n *fatias*, há no máximo n vizinhanças locais. O tamanho de cada vizinhança local de uma v-célula $S_i(v)$ é limitado pela quantidade de arestas à esquerda dos vértices em $S_i(v)$. Assim, somando-se os tamanhos de todas as vizinhanças locais, temos como resultado uma complexidade de tempo $\mathcal{O}(|V(G)|+|E(G)|)$. ♦

Finalmente, unindo-se o resultado do Lema 2.13 com a complexidade de tempo da construção da coárvore (Algoritmo B.1, p. 85), é possível concluir que o Algoritmo 2.1 (p. 42) reconhece cografos em tempo linear.

2.2 Alguns problemas em cografos

Os cografos são importantes porque muitos problemas $\mathcal{NP}$-completos admitem solução polinomial ou linear em cografos (inclusive a coloração de vértices), como pode ser visto na Tabela 2.5 (p. 52). Seu próprio reconhecimento é feito em tempo linear, conforme visto na Seção 2.1. Contudo, como pode ser verificado na tabela indicada, ainda não se sabe se ARESTA-COLORAÇÃO restrito a cografos é um problema $\mathcal{NP}$-completo (suspeita-se que esteja em $\mathcal{P}$ — mais precisamente em $\mathcal{TIME}(n+m)$ — em razão da Conjectura *Overfull*, conforme será abordado no Capítulo 4).

Problema	Grafos	P_4-redutíveis	Cografos	Multipartidos completos
2-Coloração	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
3-Coloração	$\mathcal{NP}$-completo	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
Coloração	$\mathcal{NP}$-completo	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
Clique	$\mathcal{NP}$-completo	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
Ciclo hamiltoniano	$\mathcal{NP}$-completo	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
Caminho hamiltoniano	$\mathcal{NP}$-completo	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$	$\mathcal{TIME}(n+m)$
Corte maximal	$\mathcal{NP}$-completo	?	$\mathcal{P}$	$\mathcal{TIME}(n+m)$
Coloração de arestas	$\mathcal{NP}$-completo	?	?	$\mathcal{TIME}(n+m)$

Tabela 2.5: Classes de complexidade para problemas relacionados a grafos [9]

2.3 Cografos na prática

A maioria dos grafos encontrados na prática podem não ser cografos, mas são muito *próximos* de o serem. Tal proximidade se justifica porque um cografo pode ser obtido mediante a remoção ou acréscimo de poucas arestas [7]. Qualquer grafo pode ser embarcado em um cografo de mesma ordem mediante o acréscimo de arestas (evidente, já que todo grafo completo é um cografo). Esse cografo resultante é chamado de *cografo de completude*. O cografo obtido pelo acréscimo de uma quantia mínima de arestas é chamado de *cografo de completude mínimo*. A seu turno, dado um cografo de completude G para um grafo qualquer H, caso H não possua nenhum outro cografo de completude que seja subgrafo de G, este é chamado de *cografo de completude minimal*. Todo cografo de completude mínimo é também minimal, mas a recíproca não é sempre verdadeira.

Nesse contexto, embora o problema de se encontrar um cografo de completude mínimo para um grafo G seja $\mathcal{NP}$-difícil, foi apresentado um algoritmo polinomial em [16] para encontrar cografos de completude minimais. Daí a importância do estudo de algoritmos em cografos: caso se esteja diante de um problema que possa ser resolvido com mais eficiência em cografos do que em grafos em geral e esse acréscimo minimal de arestas não prejudique a modelagem do problema, ter-se-á então uma forma muito mais eficiente de resolvê-lo.

3 O PROBLEMA DA COLORAÇÃO DAS ARESTAS

A Coloração de Arestas consiste na atribuição de cores às arestas de um grafo, de forma que arestas incidentes no mesmo vértice tenham cores distintas. Embora não poucas vezes se deseje obter a própria coloração válida, em geral o interesse recai apenas na *quantidade* mínima de cores necessárias para se obter uma dessas colorações (conhecida como índice cromático e representada por $\chi'(G)$). Nesses casos, o problema passa a ser enunciado conforme descrito na p. 29 (ARESTA-COLORAÇÃO).

É possível reduzir ARESTA-COLORAÇÃO para COLORAÇÃO, bastando que se faça a conversão do grafo para seu grafo linha (cada aresta é convertida em um vértice e cada incidência entre arestas é convertida em uma aresta). Contudo, ARESTA-COLORAÇÃO possui peculiaridades que tornam seu estudo mais interessante, principalmente no que diz respeito ao fato de que há apenas dois valores possíveis para $\chi'(G)$, conforme demonstrado na Seção 3.1.

3.1 Teorema de Vizing

O *Teorema de Vizing* [22] estabelece que, dado um grafo G, o valor máximo de $\chi'(G)$ é $\Delta(G)+1$. A demonstração decorre da *Recoloração de Vizing* (Lema 3.1), a qual estabelece que, dadas certas condições (existência de um extremo cuja vizinhança possui ao menos uma *cor livre* em cada vértice), sempre é possível colorir uma aresta usando a mesma quantidade de cores.

Lema 3.1 (*Recoloração de Vizing* [22])**.** *Seja uv uma aresta ainda não colorida de um grafo G parcialmente colorido com no máximo C cores. Se todo vértice de $N(u)\cup\{u\}$ ou todo vértice de $N(v)\cup\{v\}$ for adjacente a no máximo $C-1$ arestas coloridas, então uv também poderá ser colorido com uma das C cores.*

Demonstração. Dizemos que c é uma *cor livre* do vértice w se não há nenhuma aresta incidente em w colorida com a cor c. Designaremos por $F(w)$ qualquer uma das *cores livres* de w. Além disso, para simplificar a demonstração e sem perda de generalidade, chamaremos v de v_0 e assumiremos que u é o vértice cuja vinhança possui ao menos uma *cor livre* em cada vértice (a existência de algum vizinho de v_0 sem *cores livres* não interfere no algoritmo). Na Figura 3.1 (p. 54), a linha tracejada unindo os vértices

indica a aresta que pretendemos colorir. As linhas pontilhadas que saem de u e v_0 indicam a existência de pelo menos uma *cor livre* em cada vértice, respectivamente a_0 e a_1.

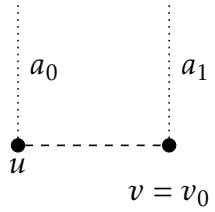


Figura 3.1: A aresta uv_0 e algumas das *cores livres* de u e v_0

Observe-se que $\{u, v_0\} \subseteq N(u) \cup \{u\}$ — daí a existência de a_0 e a_1. Tomaremos os seguintes passos para colorir uv_0:

1. Se u e v_0 tiverem uma *cor livre* comum então podemos colorir uv_0 com tal cor e finalizamos.

2. Caso contrário, sejam $a_0 = F(u)$ e $a_1 = F(v_0)$. Há, portanto, uma aresta incidente em u colorida com a cor a_1, conforme indica a Figura 3.2. Vamos considerar que v_1 designa a outra extremidade dessa aresta ($v_1 \neq u$). Além disso, consideramos também que v_1 e u possuem em comum uma *cor livre*. Nesse caso, recolorimos uv_1 com tal cor e então u passará a ter a_1 como *cor livre*. Ora, podemos aplicar o Passo 1 porque u e v_0 possuem agora um *cor livre* comum (a_1). Finalizamos.

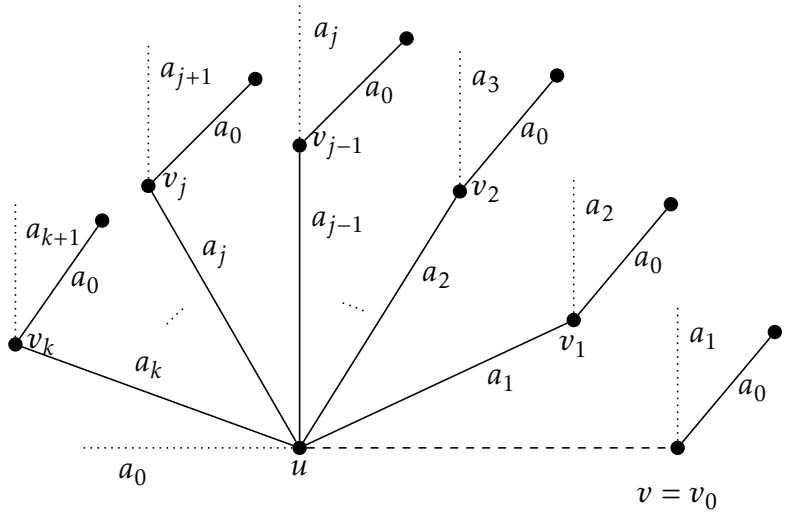


Figura 3.2: As *cores livres* da vizinhança de u

3. Caso v_1 e u não possuam nenhuma *cor livre* comum, sejam $a_2 = F(v_1)$ e v_2 a outra extremidade da aresta incidente em u colorida com a cor a_2. De maneira análoga, caso v_2 e u possuam uma *cor livre* comum, recolorimos uv_2 com tal cor e em seguida usamos a nova *cor livre* de u (a_2) para recolorir uv_1. Esse procedimento tem como resultado uma nova *cor livre* em u (a_1), que será então utilizada para colorir uv_0 (Passo 1). Finalizamos.

4. De maneira geral, portanto, sempre que houver uma *cor livre* comum em v_k e u, recolorimos uv_k com tal cor e utilizamos a_k para recolorir uv_{k-1}, até atingirmos uv_0 e finalizarmos. A este procedimento, chamamos de *caimento*.

5. Contudo, é possível que em algum momento (para algum k), todas as *cores livres* de v_k sejam cores de arestas que já consideramos anteriormente. Assim, não podemos mais continuar esse procedimento. Ou seja, toda *cor livre* de v_k é igual a alguma cor do conjunto $\{a_1, a_2, \dots a_{k-1}\}$. Chamamos de $a_j = a_{k+1}$ a primeira cor em comum (com o menor índice j). Sabemos que em todos os vértices $v_1, v_2, v_3 \dots v_k$ há uma aresta incidente colorida com a cor a_0 — caso contrário já teríamos realizado o *caimento* e finalizado. Consideremos então o passeio de tamanho máximo que se inicia em v_k e cujas arestas foram coloridas alternadamente com as cores a_0 e a_j (esse passeio não é um ciclo porque a_j é uma das *cores livres* de v_k). Há três possibilidades:

   (i) O passeio não termina em nenhum dos vértices do conjunto $\{u, v_1, v_2, v_3 \dots v_k\}$. Nesse caso, trocamos todas as cores do passeio (a aresta colorida com a_0 é recolorida com a_j e vice-versa). É fácil notar que essa troca sempre será possível. Assim, teremos uma *cor livre* comum em u e v_k. Aplicamos o Passo 4 e finalizamos.

   (ii) O passeio termina em u (ou seja, passa pelo vértice v_j e pela aresta incidente em u colorida com a cor a_j). Fazemos a mesma troca de cores do passeio conforme foi indicado no item anterior. Com tal troca, o vértice u passará a ter como *cor livre* a cor a_j em vez de a_0. Assim, teremos a_j como *cor livre* em comum de u e v_{j-1}. Aplicamos o Passo 4 em uv_{j-1} e finalizamos.

   (iii) O passeio termina em algum v_i de $\{v_1, v_2, v_3 \dots v_k\}$. Realizamos mais uma vez a troca de cores do passeio. Com isso, v_i e u terão uma *cor livre* comum (a_0).

Aplicamos o Passo 4 em uv_i e finalizamos. ♦

O Lema 3.1 (p. 53) pode ser empregado em diversas ocasiões. É, inclusive, peça fundamental na demonstração dos Teoremas 5.4 (p. 69) e 5.5 (p. 70), que constituem os principais resultados deste trabalho. Serve também de base para um teorema muito importante, o *Teorema de Vizing* (Teorema 3.2), que estabelece um limite superior para o índice cromático de qualquer grafo.

Teorema 3.2 (*Teorema de Vizing* [22])**.** *Para colorir as arestas de um grafo G são suficientes $\Delta(G)+1$ cores.*

Demonstração. Demonstraremos por indução.

Base de indução: grafos sem arestas atendem trivialmente à premissa.

Hipótese de indução: todo grafo G' com $|E(G')| < m$ pode ser colorido com $\Delta(G')+1$ cores.

Passo de indução: vamos provar que todo grafo G com m arestas pode ser colorido com $\Delta(G)+1$ cores. Denotamos $G-uv$ o grafo obtido mediante a remoção da aresta uv de G. Logicamente esse grafo é contemplado pela hipótese de indução. Podemos então aplicar em G a mesma coloração utilizada em $G-uv$, faltando colorir apenas a aresta uv. Temos então duas possibilidades:

(i) Se $\Delta(G) = \Delta(G-uv)+1$, então podemos utilizar essa cor extra (que obtivemos com o aumento do grau máximo ocorrido na passagem de $G-uv$ para G) para colorir a aresta uv. Finalizamos.

(ii) Se $\Delta(G) = \Delta(G-uv)$, então temos uma *cor livre* em cada um dos vértices de G. Aplicamos a *Recoloração de Vizing* em uv e finalizamos. ♦

3.2 O Problema da Classificação

O *Teorema de Vizing* (Teorema 3.2) estabelece limites justos para o índice cromático de um grafo, já que $\Delta(G) \leqslant \chi'(G) \leqslant \Delta(G)+1$. Assim, podemos classificar qualquer grafo G da seguinte forma:

(i) se $\chi'(G) = \Delta(G)$, dizemos que G é *Classe 1*;

(ii) se $\chi'(G) = \Delta(G)+1$, dizemos que G é *Classe 2*.

Em razão da existência de apenas duas classes, ARESTA-COLORAÇÃO pode ser apresentado de uma forma diferente:

Problema da Classificação (CLASSIFICAÇÃO). *O número de cores necessárias para colorir as arestas de um grafo é igual ao seu grau máximo?*

Muito embora haja apenas duas possíveis classes, CLASSIFICAÇÃO é um problema muito difícil. Na verdade, quando restrito a grafos 3-regulares (grafos nos quais todos os vértices têm grau 3), CLASSIFICAÇÃO já é $\mathcal{NP}$-completo. Isso ocorre por dois motivos:

(i) Há uma redução em tempo polinomial de 3-SAT para CLASSIFICAÇÃO restrito a grafos 3-regulares, o que faz com que esse problema seja $\mathcal{NP}$-difícil [13].

(ii) CLASSIFICAÇÃO restrito a grafos 3-regulares está em $\mathcal{NP}$. É fácil constatar tal pertinência mesmo sem a restrição, pois uma Δ-coloração de arestas é um certificado polinomial para uma instância positiva do problema.

Apesar de haver duas classes, *quase todos* os grafos são *Classe 1*, pois à medida que se tomam grafos com número maior de vértices, tal classe predomina. Isso ocorre porque um grafo *Classe 2* necessariamente deve ter no mínimo três vértices com grau igual a $\Delta(G)$, os quais se tornam proporcionalmente escassos com o aumento do número de vértices [19].

Não obstante seja bastante difícil encontrar o índice cromático de um grafo, esse problema é de simples solução em alguns casos. Ora, é fácil verificar que, pelo fato de corresponder a um emparelhamento, cada cor pode ser atribuída a no máximo $\lfloor n/2 \rfloor$ arestas, caso contrário a coloração seria inválida (haveria necessariamente duas arestas incidentes no mesmo vértice com a mesma cor, como ocorre na Figura 1.4b, mostrada na p. 30). Portanto, se o número de arestas for superior a $\Delta\lfloor n/2 \rfloor$, não será possível obter uma coloração válida com $\Delta(G)$ cores, e o grafo será invariavelmente *Classe 2*. Grafos com essa característica são chamados de grafos *overfull* (sobrecarregados) ou simplesmente *O*. Algumas características interessantes que se observam em um grafo *overfull* são [1]:

(i) possui no mínimo três Δ-vértices;

(ii) possui no máximo $\delta(G)$ vértices que não são Δ-vértices;

(iii) n é sempre ímpar.

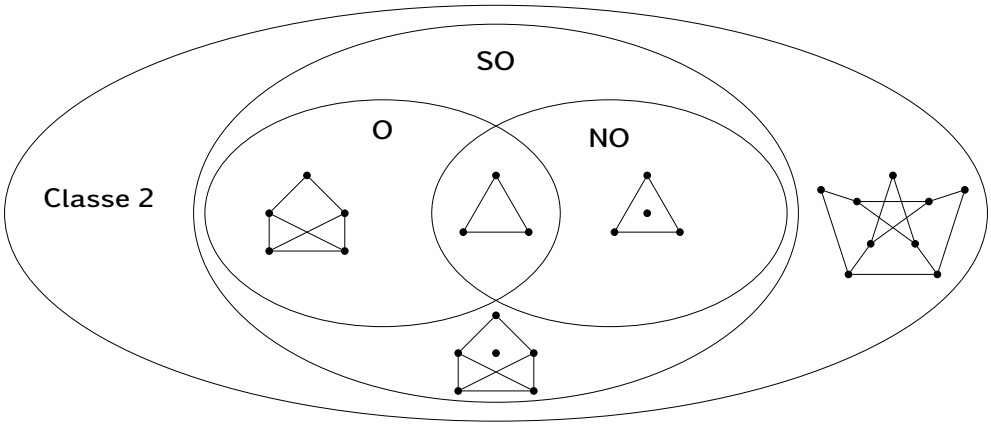


Figura 3.3: Relação entre *Classe 2*, *O*, *SO* e *NO*

Por outro lado, caso um subgrafo induzido H do grafo G seja *overfull* e $\Delta(H) = \Delta(G)$, então dizemos que G é subgrafo-*overfull* ou *SO* (também será *Classe 2*). Além disso, caso H seja induzido por um Δ-vértice e todos os seus vizinhos, dizemos que G é vizinhança-*overfull* (do inglês *neighbourhood-overfull*) ou *NO*. Para melhor ilustrar o exposto, a Figura 3.3 mostra as inclusões de *Classe 2*, *SO*, *O* e *NO*.

Muito embora todo subgrafo-*overfull* seja *Classe 2*, a recíproca não é verdadeira. É o caso do grafo de Petersen, esboçado na Figura 3.4. Esse grafo é *Classe 2* mas não é

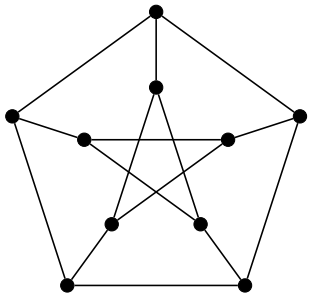


Figura 3.4: O grafo de Petersen

overfull, pois possui número par de vértices. Por outro lado, Chetwynd e Hilton enunciaram uma conjectura, conhecida como Conjectura *Overfull*, a qual sugere que, para todo grafo G com $\Delta(G) > n/3$, G é subgrafo-*overfull* se e somente se G é *Classe 2* [5]. O grafo P^*, que é obtido pela remoção de um vértice qualquer do grafo de Petersen e também é *Classe 2* (mostrado na Figura 3.3), sugere que esse limite inferior da Conjectura *Overfull* é bastante justo, já que possui $n = 9$, $m = 12$, $\Delta = 3$ e portanto não é *overfull*, mas também não se enquadra na premissa da conjectura. É importante lembrar

ainda que, quando $\Delta(G) > n/3$, o grafo possui no máximo três subgrafos induzidos que são Δ-*overfull* (ou seja, são *overfull* e possuem grau máximo igual ao do supergrafo), e há um algoritmo que pode encontrá-los em tempo polinomial [18]. Além disso, se $\Delta(G) \geqslant n/2$, há no máximo um subgrafo induzido Δ-*overfull*, e existe um algoritmo capaz de encontrá-lo em tempo linear [18].

A Conjectura *Overfull* já foi provada para diversas classes de grafos [18], dentre elas a dos multipartidos completos, fato que merece destaque por configurarem uma subclasse dos cografos. Para tanto, empregou-se um artifício chamado de técnica das cores livres, e obteve-se para essa classe a equivalência entre *O* e *Classe 2* [12]. Ou seja, além de serem subgrafo-*overfull*, todos os grafos multipartidos completos *Classe 2* são também *overfull*.

3.3 Coloração de grafos *quasi-threshold*

Tal como é o caso dos cografos, os grafos *quasi-threshold* também podem ser definidos de diversas formas. Sua definição construtiva é a seguinte [14]:

(i) K_1 é um grafo *quasi-threshold*;

(ii) se G_1 e G_2 são grafos *quasi-threshold*, $G_1 \cup G_2$ também é um grafo *quasi-threshold*;

(iii) se G é um grafo *quasi-threshold*, $G * K_1$ também é um grafo *quasi-threshold*.

Essa definição evidencia que os grafos *quasi-threshold* formam uma subclasse dos cografos. Outras duas formas equivalentes de defini-los são [14]:

1. Grafos que não têm P_4 nem C_4 induzidos.
2. Grafos que são simultaneamente grafos cordais (grafos nos quais cada ciclo C_n com $n > 3$ tem uma corda) e cografos.

Uma característica interessante e muito útil dos grafos *quasi-threshold* é que, quando conexos, eles sempre têm *vértice universal* [14]. De fato, se tomarmos a coárvore de um grafo *quasi-threshold* conexo, perceberemos que, sendo x o último vértice acrescido pela operação de junção e y qualquer outro vértice, $\mathrm{lca}(x, y) = 1$. Ou seja, x é um Δ-vértice e *vértice universal*. Essa propriedade traz à baila o resultado apresentado por Plantholt [20], segundo o qual, para grafos com *vértice universal*, vale *Classe 2* = *O*. *Quasi-thresholds* formam, portanto, uma família de cografos que sabemos classificar.

4 COLORAÇÃO DE ARESTAS DE COGRAFOS

Uma das razões pelas quais se acredita que CLASSIFICAÇÃO restrito a cografos admita solução em tempo linear é o fato de que eles se enquadram na premissa da Conjectura *Overfull* [8, 12], conforme evidencia o Teorema 4.1.

Teorema 4.1 ([8, 12])**.** *Se G é um cografo conexo, então $\Delta(G) \geqslant n/2$.*

Demonstração. Seja T a coárvore associada a G e r o número de ramos de G. Portanto, $V(G) = \alpha(1) \cup \ldots \cup \alpha(r)$. Sem perda de generalidade, suponha-se que $1 \leqslant a(1) \leqslant \ldots \leqslant a(r)$. Vamos construir um grafo G' com as seguintes características:

(i) $V(G') = V(G)$, e

(ii) $E(G') = E(G) \setminus \{uw : u, w \in \alpha(i)$, para todo $i, 1 \leqslant i \leqslant r\}$.

Primeiramente, observamos que G' é subgrafo de G. Portanto,

$$\Delta(G) \geqslant \Delta(G'). \tag{4.2}$$

Além disso, notamos que, pela forma como construímos G', todos os Δ-vértices estão em $\alpha(1)$. Logo,

$$\Delta(G') = n - a(1). \tag{4.3}$$

Em razão da ordem que estabelecemos para os vértices da partição, podemos escrever

$$\begin{aligned} a(1) &\leqslant a(2) + a(3) + \ldots + a(r) \\ a(1) + a(1) &\leqslant a(1) + a(2) + \ldots + a(r) \\ 2a(1) &\leqslant n \\ a(1) &\leqslant n/2 \\ a(1) + n/2 &\leqslant n \\ n/2 &\leqslant n - a(1) \end{aligned} \tag{4.4}$$

Aplicando (4.2) e (4.3) em (4.4), temos que

$$\begin{aligned} n - a(1) &\geqslant n/2 \\ \Delta(G') &\geqslant n/2 \\ \Delta(G) &\geqslant \Delta(G') \geqslant n/2 \\ \Delta(G) &\geqslant n/2 \end{aligned} \tag{4.5}$$

♦

Por si só, o Teorema 4.1 (p. 4.1) estabelece que apenas cografos conexos têm $\Delta(G) \geqslant n/2$, ficando de fora, pois, os cografos desconexos (cuja coárvore leva o rótulo 0 na raiz). Entretanto, para efeitos de CLASSIFICAÇÃO, grafos podem ser considerados conexos sem perda de generalidade [17], fato para o qual apresentamos uma demonstração:

Proposição 4.6 ([17]). *O índice cromático de um grafo G é o máximo dentre os índices cromáticos das componentes de G.*

Demonstração. Por indução no número de componentes de G. Se G tem uma só componente, G é conexo e a asserção se verifica trivialmente. Suponhamos, então, que G tenha $t > 1$ componentes $C_1, \ldots, C_t$ e que, para $G' = C_1 \cup \cdots \cup C_{t-1}$, $\chi'(G') = \max\{\chi'(C_1), \ldots, \chi'(C_{t-1})\}$ por hipótese de indução. Podemos colorir as arestas de C_t com as cores $1, \ldots, \chi'(C_t)$ e as arestas de G' com $1, \ldots, \chi'(G')$, já que nenhuma aresta de C_t é adjacente a alguma aresta de G'. Logo, $\chi'(G) = \max\{\chi'(G'), \chi'(C_t)\} = \max\{\chi'(C_1), \cdots, \chi'(C_t)\}$. ♦

A propósito, já foi demonstrado que, para cografos com coárvore de um ou dois níveis, $SO = O =$ *Classe 2*, ao passo que, para coárvores de três níveis completos, $SO = (O \cup NO) \backslash (O \cap NO)$ [8]. Tem-se, portanto, indícios muito fortes de que a coloração de arestas de cografos é um problema que admite solução não apenas polinomial, mas linear, já que o Algoritmo 4.1 decide se um grafo G é subgrafo-*overfull* em tempo linear [18].

Teste-SO(G):
Entrada: *Um grafo G com $\Delta(G) \geqslant n/2$.*
Saída: **V** *se G é subgrafo-overfull ou* **F** *caso contrário.*
1 $S \leftarrow \{u \in V(G) : d^{**}(u) \leqslant 1\}$;
2 $G' \leftarrow G[V(G) \setminus S]$;
3 se $\Delta(G') < \Delta(G)$, **devolva F**;
4 $r \leftarrow |V(G')|$;
5 **ordene** $V(G')$ de forma que $d^*(v_1) \geqslant d^*(v_2) \geqslant \ldots \geqslant d^*(v_r)$;
6 **para** cada j ímpar tal que $|V(G)| - \Delta(G) < j \leqslant |V(G')|$:
7 **se** $d^*(v_j) \geqslant d^*(v_j + 1) + 2$ ou $j = r$:
8 **se** $G[\{v_1, v_2, \ldots, v_j\}]$ for Δ-*overfull*, **devolva V**;
9 **devolva F**.

Algoritmo 4.1: Teste de subgrafo-*overfull*

4.1 Coloração de arestas de grafos-junção

A Proposição 4.6 (p. 62), aliada à Conjectura *Overfull*, permite que se concentre o foco de estudo na junção de cografos. De fato, cografos conexos constituem uma subclasse dos grafos-junção, que são os grafos que podem ser construídos a partir da junção de dois grafos quaisquer (não necessariamente cografos). A Observação 4.7 apresenta resultados já conhecidos em grafos-junção (e portanto em cografos), alguns dos quais serão explicados com maiores detalhes em seguida.

Observação 4.7. *Vamos assumir G_1 e G_2 grafos com ordem respectivamente $n_1 \leqslant n_2$ e $G = G_1 * G_2$. Vamos também assumir $\Delta = \Delta(G)$, $\Delta_1 = \Delta(G_1)$ e $\Delta_2 = \Delta(G_2)$. São condições suficientes para que G seja Classe 1:*

(i) $\Delta_1 > \Delta_2$ *[10];*

(ii) $\Delta_1 < \Delta_2$ *e* $n_1 = n_2$ *[10];*

(iii) $\Delta_1 = \Delta_2$ *e ainda:*

- *(a)* G_1 *e* G_2 *serem Classe 1 [10];*
- *(b)* G_1 *ser subgrafo de* G_2 *[10];*
- *(c)* G_1 *e* G_2 *serem uniões disjuntas de grafos completos (ou seja, G é um cografo cuja coárvore possui profundidade* 3 *em cada folha e raiz com exatos dois filhos) [10];*
- *(d) G ser regular [10];*

(iv) $\Delta_2 < n_2 - n_1$ *[17];*

(v) G_1 *e* G_2 *serem regulares e* $n_1 + n_2$ *ser par [21].*

Adiante são apresentadas as demonstrações de alguns resultados da Observação 4.7, que utilizam as definições seguintes.

Definição 4.8. Sendo G o grafo obtido pela junção dos grafos G_1 e G_2, consideramos que:

(i) B_G é o grafo bipartido obtido a partir da remoção de G de todas as arestas que possuem ambos os extremos em G_1 ou G_2. Ou seja, é o grafo $B_G(V(G), E(G) \setminus (E(G_1) \cup E(G_2)))$.

(ii) Dado um emparelhamento M de B_G, G_M é o grafo que se obtém quando são removidas de G as arestas de $B_G \setminus M$. Ou seja, é o grafo $G_M(V(G), E(G_1) \cup E(G_2) \cup M)$.

Lema 4.9 ([10]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$, $\Delta(G_1) \geqslant \Delta(G_2)$ e houver um emparelhamento máximo M em B_G tal que G_M seja Classe 1, então G será Classe 1.*

Demonstração. Se $|V(G_1)| \leqslant |V(G_2)|$ e $\Delta(G_1) \geqslant \Delta(G_2)$, então $\Delta(G) = \Delta(G_1) + |V(G_2)|$. Assim, algum Δ-vértice de G pertence a G_1. Por sua vez, $\Delta(B_G) = |V(G_2)|$. Seja B' o grafo bipartido que se obtém quando se remove de B_G todas as arestas que estão em M. Sabemos que $\chi'(G) \leqslant \chi'(G_M) + \chi'(B')$, pois podemos colorir as arestas de G_M com as cores de 1 a $\chi'(G_M)$ e as arestas de B' com as cores de $\chi'(G_M)+1$ a $\chi'(G_M)+\chi'(B')$, obtendo assim uma coloração válida de G. Ora, B' é bipartido e, pelo Teorema de König [15], $\chi'(B') = \Delta(B')$. Uma vez que B' foi obtido a partir de B_G com a remoção de um emparelhamento, $\Delta(B') = \Delta(B_G) - 1$. Assim, $\chi'(B') = \Delta(B_G) - 1$. Por sua vez, como G_M é *Classe 1*, $\chi'(G_M) = \Delta(G_M)$. Temos então que $\chi'(G) \leqslant \chi'(G_M) + \chi'(B') = \Delta(G_M) + \Delta(B_G) - 1$. Sabemos que $\Delta(G_M) = \Delta(G_1) + 1$, pois, com a adição do emparelhamento M, os Δ-vértices originários de G_1 tiveram seu grau acrescido de um. Portanto, $\chi'(G) \leqslant \Delta(G_M)+\Delta(B_G)-1 = \Delta(G_1)+1+\Delta(B_G)-1 = \Delta(G_1)+|V(G_2)| = \Delta(G)$, ou seja, G é *Classe 1*. ♦

Teorema 4.10 ([10]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$ e $\Delta(G_1) > \Delta(G_2)$, então G é Classe 1.*

Demonstração. Em razão do Lema 4.9, basta provar que existe ao menos um emparelhamento máximo M em B_G tal que G_M seja *Classe 1*. Seja $\Delta = \Delta(G_1)$. Vamos provar que é possível colorir G_M com $\Delta + 1$ cores. Na pior das hipóteses, G_1 e G_2 são *Classe 2*. Colorimos as arestas de G_1 com as cores 1 a $\Delta + 1$ e as arestas de G_2 com as cores 1 a Δ. Faltam ser coloridas apenas as arestas de M. Tomamos tais arestas uma a uma e chamamos de u o extremo em G_1 e de v o extremo em G_2. Passamos a analisar a vizinhança de v: u tem *cor livre* em virtude da própria aresta do emparelhamento ainda não colorida. Quanto aos vizinhos de v em G_2, todos têm ao menos uma *cor livre* porque são extremos de no máximo Δ arestas. Logo, pelo Lema 3.1 (p. 53), cada aresta uv pode ser colorida com uma das $\Delta + 1$ cores, e o grafo G_M é *Classe 1*. ♦

Teorema 4.11 ([10]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$ e tanto G_1 quanto G_2 forem Classe 1, então G é Classe 1.*

Demonstração. Em razão do Lema 4.9 (p. 64), basta provar que existe ao menos um emparelhamento máximo M em B_G tal que G_M seja *Classe 1*. Sabemos que podemos colorir G_1 e G_2 com $\Delta_1 = \Delta_2$ cores. Assim, de um conjunto de $\Delta + 1$ cores, sempre teremos uma *cor livre*, que podemos utilizar para colorir M. ⧫

Teorema 4.12 ([10]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$ e G_1 for subgrafo de G_2, então G é Classe 1.*

Demonstração. Em razão do Lema 4.9 (p. 64), basta provar que existe ao menos um emparelhamento máximo M em B_G tal que G_M seja *Classe 1*. Sejam $V_1 = \{u_1, u_2, \ldots, u_{n_1}\}$ e $V_2 = \{v_1, v_2, \ldots, v_{n_2}\}$. Pinte as arestas de G_2 com $\Delta_2 + 1$ cores. Já que G_1 é subgrafo de G_2, para cada aresta $u_i u_j$ de G_1 existe uma aresta $v_i v_j$ em G_2. Utilizamos a cor de cada $v_i v_j$ para colorir a aresta $u_i u_j$. Sabemos que cada vértice v_i tem uma *cor livre* dentre as $\Delta_2 + 1$ cores. Nesse caso, cada u_i também terá a mesma *cor livre*, que pode ser utilizada para colorir a aresta $u_i v_i$ de M. ⧫

Teorema 4.13 ([10]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$ e tanto G_1 quanto G_2 forem uniões disjuntas de grafos completos, então G é Classe 1.*

Demonstração. Em razão do Lema 4.9 (p. 64), basta provar que existe ao menos um emparelhamento máximo M em B_G tal que G_M seja *Classe 1*. Primeiramente vamos ordenar os vértices de G_1 de forma a obter uma sequência $u_1, u_2, \ldots, u_{n_1}$ conforme as seguintes regras:

(i) Vértices pertencentes à mesma clique são dispostos consecutivamente;

(ii) Se a ordem s do grafo completo maximal ao qual pertence u_i é menor que a ordem t do grafo completo maximal ao qual pertence u_j, então $i < j$.

De modo análogo ordenamos os vértices de G_2, obtendo assim a sequência $v_1, v_2, \ldots, v_{n_2}$. Seja $f : E(G) \to \{f_0, f_1, \ldots, f_{\Delta_1}\}$ uma coloração de G_1 feita do seguinte modo: cada aresta $u_i u_j$ recebe a cor f_h, sendo $h = (i + j) \bmod (\Delta_1 + 1)$.

Para mostrar que tal coloração é válida, basta provar que cada par de arestas adjacentes possui cores distintas. Por contradição, vamos assumir que as arestas $u_i u_j$

e $u_j u_k$ (com $i \neq k$) possuem a mesma cor f_h. Sabemos que $h = (i+j) \bmod (\Delta_1 + 1)$ e $h = (j+k) \bmod (\Delta_1 + 1)$. Isso significa que, para algum inteiro $t_1 \geqslant 0$, $h = i + j - t_1(\Delta_1 + 1)$ e, para algum inteiro $t_2 \geqslant 0$, $h = j + k - t_2(\Delta_1 + 1)$. Ademais, já que $i \neq k$, $t_1 \neq t_2$. Mas nesse caso temos que $(\Delta_1 + 1)(t_2 - t_1) - (k - i) = 0$, e assim $|k - i| = |t_2 - t_1|(\Delta_1 + 1)$, e assim $|k - i| \geqslant \Delta_1 + 1$. Por outro lado, a ordenação dos vértices de G_1 impõe que $|k - i| \leqslant \Delta_1$, pois u_i e u_j pertencem ao mesmo grafo completo maximal cuja ordem é no máximo $\Delta_1 + 1$. Contradição.

Ora, é fácil perceber que cada vértice u_i ($1 \leqslant i \leqslant n_1$) tem $f_{(2i) \bmod (\Delta_1+1)}$ como *cor livre*. De modo análogo, cada vértice v_i ($1 \leqslant i \leqslant n_2$) também tem $f_{(2i) \bmod (\Delta_1+1)}$ como *cor livre*. Assim, podemos colorir cada aresta $u_i v_i$ de M com tal cor. ♦

A demonstração do Teorema 4.16 necessita do clássico lema de Fournier referente à classificação de grafos com núcleo acíclico (Lema 4.14) e do Lema 4.15.

Lema 4.14 ([11]). *Se G for um grafo tal que $\Lambda[G]$ é acíclico, então G é Classe 1.*

Lema 4.15 ([17]). *Seja G o grafo obtido pela união dos grafos $G_1(V, E_1)$ e $G_2(V, E_2)$ (ou seja, com o mesmo conjunto de vértices). Se $\Delta(G) = \Delta(G_1) + \Delta(G_2)$ e nenhuma aresta de E_1 tiver ambos os extremos em $N[\Lambda[G_2]]$, então G é Classe 1.*

Teorema 4.16 ([17]). *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $\Delta_2 < |V(G_2)| - |V(G_1)|$, então G é Classe 1.*

Demonstração. Seja $G' = G_2 \cup B_G$. O grau de qualquer vértice $v \in V(G_1)$ em G' é $|V(G_2)|$. Portanto, uma vez que $|V(G_2)| > \Delta_2 + |V(G_1)|$, os Δ-vértices de G' são vértices de G_1. Eles induzem um núcleo sem arestas, portanto acíclico, e pelo Lema 4.14 concluímos que G' é *Classe 1*. Por outro lado, $G = G' \cup G_1$ e $\Delta(G) = |V(G_2)| + \Delta(G_1) = \Delta(G') + \Delta(G_1)$. Além disso, G' é *Classe 1* e todas as suas arestas possuem ao menos um extremo que não está $N[\Lambda[G_1]]$, o qual é vértice de G_2. Assim, pelo Lema 4.15, G é *Classe 1*. ♦

5 RESULTADOS OBTIDOS

O objetivo proposto, qual seja, *encontrar algum resultado útil acerca da Coloração de Arestas de Cografos*, foi alcançado com a demonstração de dois teoremas relativos à classificação de grafos-junção, superclasse dos cografos conexos (Teorema 5.4, p. 69). Para se chegar a tal resultado, além da leitura e compreensão dos artigos mencionados nos capítulos anteriores, foi desenvolvido um programa de análise de índice cromático de cografos, chamado Cograph (descrito na Seção 5.1). Após diversas consultas à base de dados gerada pelo programa (que construiu todos os 202.939 cografos com $n \leqslant 13$), constatou-se o seguinte:

Observação 5.1. *Sejam G o grafo obtido pela junção dos cografos G_1 e G_2, $n_1 = |V(G_1)|$ e $n_2 = |V(G_2)|$. Se $n_1 + n_2 \leqslant 13$, $n_1 \leqslant n_2$, $\Delta(G_1) = \Delta(G_2)$ e $\Lambda[G_1]$ for acíclico, então G será Classe 1.*

Tal constatação despertou interesse para uma análise mais atenta desse caso. Contudo, não foi possível comprová-la para todos os casos (tampouco demonstrar que é incorreta). Ainda assim, obtiveram-se dois resultados relacionados, demonstrados nos Teoremas 5.4 (p. 69) e 5.5 (p. 70).

5.1 O programa Cograph

O programa Cograph [7] (listado no Apêndice C) foi desenvolvido na linguagem *python* e roda na plataforma *GNU/Linux*. Durante a execução, na primeira etapa, constrói uma base de dados (utilizando o SGBD *sqlite*) contendo diversas características de todos os cografos com ordem no máximo *MAX_ORDER* (constante definida na linha 22 do arquivo *cograph.py* como sendo 13). Todos os módulos do programa são invocados por meio de um *Makefile*. A construção da base de dados é realizada com o seguinte comando:

```
$ make build
```

Com a base de dados construída, pode-se executar o módulo de consulta por meio da interface *sqlite*, que é acionada da seguinte maneira:

[7] Código-fonte disponível no endereço `https://github.com/cunhalima/cograph` sob a licença *GNU GPL v3*

```
$ make run
```

Finalmente, há também um módulo que executa algumas consultas no banco de dados para fim de testes. O comando a ser utilizado é o seguinte:

```
$ make test
```

A base de dados consiste de duas tabelas: *OP* e *GR*. A tabela *GR* possui uma entrada para cada cografo construído. Cada grafo é identificado de forma única por meio de um *ID* (número sequencial) ou pela sua coárvore. As coárvores são codificadas por uma cadeia de caracteres com três possíveis símbolos: o ponto final ".", que representa um vértice, ou os dígitos "0" e "1", que representam, respectivamente, as operações de união disjunta e junção. Apenas as coárvores canônicas são válidas — portanto, há sequências de símbolos que não são permitidas, pois codificam coárvores não-canônicas. As demais colunas registram características de cada cografo. As informações que elas armazenam podem ser deduzidas intuitivamente a partir de seus nomes (linhas 599 a 626 do arquivo *cograph.py*).

A outra tabela, *OP*, limita-se a armazenar as operações que produzem cada cografo. A coluna *OP* indica qual é a operação (0 é união disjunta e 1 é junção), ao passo que as colunas *D*, *A* e *B* indicam, respectivamente, os grafos da operação $G = G_1 * G_2$ (considerando sempre $|V(G_1)| \leqslant |V(G_2)|$). É digno de nota que a coluna *D* não pode servir como chave primária da tabela, pois um mesmo cografo pode ser resultante de diferentes operações. Desse fato decorre a seguinte observação:

Observação 5.2. *Dados dois cografos G_1 e G_2 e sua junção $G = G_1 * G_2$, é possível que não se saiba de antemão se G será Classe 1 ou Classe 2, simplesmente pelo fato de que as propriedades de G_1 e G_2 não são abarcadas por nenhum caso já demonstrado. No entanto, é possível que G possa também ser obtido pela junção de outros dois cografos G_1' e G_2' que se enquadrem em algum caso já conhecido.*

O programa é orientado a objetos e possui apenas duas classes:

(i) *CotreeNode* é utilizada para modelar uma coárvore canônica. As restrições são impostas pelo método *sort*, que ordena as subárvores de modo único.

(ii) *CoNode* é utilizada para modelar um cografo (que traz em seu bojo a codificação da respectiva coárvore canônica). Todas as propriedades do grafo são calculadas no momento de sua criação (método *build*).

A função *makeOrder* é responsável pela geração de todos os cografos de determinada ordem. No decorrer da criação, seus dados são gravados no banco de dados pelas funções *writeCGData* e *updateGraphs*.

5.2 Uma conjectura e dois teoremas

A observação feita com a utilização do programa Cograph permitiu que se encontrassem alguns padrões, traçando-se assim a seguinte conjectura:

Conjectura 5.3. *Seja G o grafo obtido pela junção dos cografos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$, $\Delta(G_1) = \Delta(G_2)$ e $\Lambda[G_1]$ for acíclico, então G será Classe 1.*

Em contrapartida, ao limitar a casos em que G_1 tem núcleo sem arestas, chegamos ao Teorema 5.4:

Teorema 5.4. *Seja G o grafo obtido pela junção dos grafos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)|$, $\Delta(G_1) = \Delta(G_2)$ e o núcleo de G_1 não possuir arestas, então G será Classe 1.*

Demonstração. Em razão do fato de que grafos *Classe 2* devem possuir ciclos no núcleo (Lema 4.14, p. 66), G_1 é *Classe 1*. Sejam $\Delta = \Delta(G_1) = \Delta(G_2)$, B o grafo bipartido $G - (E(G_1) \cup E(G_2))$ e $M \subseteq E(B)$ um emparelhamento que cobre todos os vértices de G_1. A Figura 5.1 mostra um exemplo de um grafo G_M construído com base nessa descrição e na Definição 4.8 (p. 63).

Figura 5.1: Exemplo do grafo G_M

Vamos demonstrar que as arestas em $(V(G), E(G_1) \cup E(G_2) \cup M)$ podem ser coloridas com $\Delta + 1$ cores, ou seja, com as cores de $\mathscr{C} = \{1, \ldots, \Delta + 1\}$.

Passo 1 Pinte as arestas de G_1 com as cores $1, \ldots, \Delta$ e as arestas de G_2 com as cores $1, \ldots, \Delta + 1$.

Passo 2 Considere, uma de cada vez, cada aresta uv de M não incidente a um Δ-vértice de G_1, com $u \in V(G_1)$. Pelo fato de que cada vizinho de u (inclusive v) tem uma *cor livre* em $\mathscr{C}$, uv satisfaz as condições do Lema 3.1 (p. 53) e pode ser colorida.

Passo 3 Por final, considere cada aresta uv de M com $u \in V(\Lambda[G_1])$. Cada w vizinho de u em G_1 tem uma *cor livre* em $\mathscr{C}$, mesmo que a aresta M incidente a w já tenha sido colorida. Uma vez que v também possui uma *cor livre* em $\mathscr{C}$, uv satisfaz as condições do Lema 3.1 (p. 53) e pode ser colorida.

Finalmente, pelo Lema 4.9 (p. 64), G é *Classe 1*. ♦

Além disso, chegamos um resultado similar, mas aplicável apenas em cografos, apresentado no Teorema 5.5.

Teorema 5.5. *Seja G o cografo obtido pela junção dos cografos G_1 e G_2. Se $|V(G_1)| \leqslant |V(G_2)| - |\Lambda[G_2]|$, $\Delta(G_1) = \Delta(G_2)$ e $\Lambda[G_1]$ for acíclico, então B_G é Classe 1.*

Demonstração. Denominamos *estrela* o cografo obtido pela junção do K_1 com um cografo sem arestas. *Centro da estrela* é o vértice advindo do K_1 (único vértice com grau maior do que um quando a estrela tiver mais do que dois vértices) e os demais vértices são *folhas*. No caso de estrelas com dois vértices, um será folha e o outro será o centro, de forma arbitrária. Quanto às estrelas com apenas um vértice, este será considerado folha. Novamente, pelo Lema 4.14 (p. 66), G_1 é *Classe 1*. Pelo fato de G_1 ser cografo, todas as componentes de $\Lambda[G_1]$ devem ser *estrelas*. Sejam $\Delta = \Delta(G_1) = \Delta(G_2)$, B o grafo bipartido $G-(E(G_1)\cup E(G_2))$ e $M \subseteq E(B)$ um emparelhamento que cobre todos os vértices de G_1 mas nenhum Δ-vértice de G_2. Vamos demonstrar que as arestas em $(V(G), E(G_1)\cup E(G_2)\cup M)$ podem ser coloridas com $\Delta+1$ cores, ou seja, com as cores de $\mathscr{C} = \{1,\ldots,\Delta+1\}$.

Passo 1 Pinte as arestas de G_1 com as cores $1,\ldots,\Delta$ e as arestas de G_2 com as cores $1,\ldots,\Delta+1$.

Passo 2 Considere, uma de cada vez, cada aresta uv de M não incidente a um Δ-vértice de G_1, com $u \in V(G_1)$. Pelo fato de que cada vizinho de u (inclusive v) tem uma *cor livre* em $\mathscr{C}$, uv satisfaz as condições do Lema 3.1 (p. 53) e pode ser colorida.

Passo 3 Em seguida, considere cada aresta uv de M com u sendo folha de alguma estrela de G_1. Cada w vizinho de u em G_1 tem uma *cor livre* em $\mathscr{C}$, mesmo que a aresta M incidente a w já tenha sido colorida. Uma vez que v também possui uma *cor livre* em $\mathscr{C}$, uv satisfaz as condições do Lema 3.1 (p. 53) e pode ser colorida.

Passo 4 Finalmente, considere cada aresta xv de M que ainda não foi colorida, ou seja, quando x for centro de alguma estrela S de G_1. Se x e v tiverem a mesma *cor livre* em $\mathscr{C}$, podemos colorir xv com essa cor e finalizamos. Senão, sejam $\alpha \in \mathscr{C}$ uma *cor livre* de v, $\beta \in \mathscr{C}$ a única *cor livre* de x e xu a aresta de G_1 colorida com α. Pinte xv com α e remova a cor de xu. Perceba que β é ainda *cor livre* de x e agora α é *cor livre* de u. Temos dois casos:

1 Se $u \in \Lambda[G_1]$, sabemos que todo w vizinho de u tem uma *cor livre* em $\mathscr{C}$, mesmo que $w \in V(G_2)$. Logo, xu satisfaz a condição do Lema 3.1 (p. 53).

2 Se $u \notin \Lambda[G_1]$, remova a cor de todas as arestas incidentes em u cujo extremo oposto seja um Δ-vértice. Assim, cada vizinho de u tem uma *cor livre* em $\mathscr{C}$ e podemos colorir xu em razão do Lema 3.1 (p. 53). Considere agora, uma por vez, cada aresta wu que foi descolorida. Temos três casos:

   (i) Se w é adjacente a x então o único vizinho de w que não possui uma *cor livre* é x. Após descolorir xw, podemos aplicar o Lema 3.1 (p. 53) para colorir wu. Voltamos ao Caso 1, com w no lugar de u, e portanto podemos colorir também xw.

   (ii) Se w for uma folha de uma estrela $S' \neq S$ de $\Lambda[G_1]$, seja y o centro de S' e remova a cor de yw. Assim, todo vizinho de w terá uma *cor livre* e podemos aplicar o Lema 3.1 (p. 53) em wu, retornando ao Caso 1 com y no lugar de x e w no lugar de u, o que possibilita que yw seja colorida.

   (iii) Se w for o centro de uma estrela S', remova a cor de todas as arestas de S', obtendo assim uma *cor livre* em todos os vizinhos de w. Então aplique o Lema 3.1 (p. 53) e pinte wu. Agora considere, uma por uma, cada aresta wr que foi descolorida. Já que r é folha de S', voltamos ao Caso 1 com w no lugar de x e r no lugar de u, e assim podemos pintar

wr.

Finalmente, pelo Lema 4.9 (p. 64), G é *Classe 1*. ♦

6 CONCLUSÃO

A Coloração de Arestas é um problema intrigante: há apenas duas classes no que tange ao índice cromático de um grafo, e ainda assim CLASSIFICAÇÃO é um problema $\mathcal{NP}$-completo. Por outro lado, quando restritos a cografos, problemas considerados intratáveis costumam admitir soluções muito mais eficientes. Com essa motivação, apresentamos diversas evidências de que CLASSIFICAÇÃO restrito a cografos é um problema que está em $\mathcal{P}$ ou até mesmo em $\mathcal{TIME}(n+m)$: a Conjectura *Overfull* [5], o caso dos multipartidos completos [12], dos *quasi-threshold* [14, 20], além dos casos que já haviam sido demonstrados para os grafos-junção [17, 10]. Serviu também como motivação o fato de existir um algoritmo de tempo linear para reconhecimento da família dos cografos que constrói a coárvore para as instâncias positivas [2], o que pode facilitar a análise de sua estrutura para a pretendida coloração.

Com o objetivo de encontrar novos resultados relacionados a CLASSIFICAÇÃO em cografos, optamos por desenvolver um programa que apontasse indícios e também servisse para encontrar contraexemplos. Nessa esteira, o desenvolvimento do programa Cograph foi muito importante, pois foi a partir da consulta a sua base de dados que se observou um padrão de obtenção de cografos *Classe 1* (enunciado na forma de conjectura) sempre que ocorria uma junção de cografos de mesmo Δ e aquele de menor ordem tivesse núcleo acíclico. Diante dos resultados já conhecidos relativos aos grafos-junção, optamos por adaptar a *Recoloração de Vizing* aos cografos, e obtivemos sucesso em dois casos:

1. Trocando o núcleo acíclico da conjectura por núcleo sem arestas (e assim estendemos o resultado para grafos-junção).

2. Estabelecendo uma quantidade mínima de vértices com grau menor do que Δ no cografo de maior ordem, para obter um emparelhamento que atendesse à premissa da *Recoloração de Vizing*.

O trabalho aqui desenvolvido sinaliza diretrizes para trabalhos futuros, afinal, muitos são os vieses de estudo de que se pode lançar mão para a abordagem do problema, como já detalhado ao longo deste texto. Seguem abaixo algumas das possibilidades que se vislumbram:

(i) Explorar o fato descrito na Observação 5.2 (p. 68) para tentar obter um leque maior de casos conhecidos.

(ii) Demonstrar a Conjectura 5.3 (p. 69) ou encontrar um contraexemplo.

(iii) Estender o programa Cograph para que utilize o Algoritmo 4.1 (p. 62) a fim de encontrar subgrafos Δ-*overfull* em tempo linear, e assim não depender da geração de subgrafos próprios. Será possível, portanto, gerar cografos aleatórios de ordens mais elevadas, e com isso avaliar com mais segurança possíveis conjecturas.

Enfim, não bastasse o fato de servir como referência para trabalhos futuros e também como orientação para possíveis abordagens do problema da *classificação de cografos quanto ao índice cromático*, o presente trabalho encerra em si resultados inéditos que certamente contribuem para o enriquecimento do conhecimento da área.

REFERÊNCIAS

[1] K. BONGARD, A. HOFFMANN, e L. VOLKMANN. Minimum degree conditions for the overfull conjecture for odd order graphs. *Australas. J. Combin.*, 28:121–129, 2003.

[2] A. BRETSCHER, D. G. CORNEIL, M. HABIB, e C. PAUL. A simple linear time LexBFS cograph recognition algorithm. *SIAM J. Discrete Math.*, 22(4):1277–1296, 2008.

[3] C. CAPELLE, A. COURNIER, e M. HABIB. Cograph recognition algorithm revisited and online induced P_4 search, 1994.

[4] G. CHARTRAND e P. ZHANG. *Chromatic graph theory*. Discrete mathematics and its applications. Chapman & Hall/CRC/Taylor & Francis Group, Boca Raton, 2009.

[5] A. G. CHETWYND e A. J. W. HILTON. Star multigraphs with three vertices of maximum degree. *Math. Proc. Camb. Phil. Soc.*, 100:303–317, 1986.

[6] D. G. CORNEIL, H. LERCHS, e L. STEWART BURLINGHAM. Complement reducible graphs. *Discr. Appl. Math.*, 3:163–174, 1981.

[7] D. G. CORNEIL, Y. PERL, e L. K. STEWART. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14(4):926–934, 1985.

[8] C. P. de MELLO, M. M. BARBOSA, e J. MEIDANIS. Local conditions for edge-colouring of cographs. *Congressus Numerantium*, 133:45–55, 1998.

[9] H. N. de RIDDER et al. Information System on Graph Classes and their Inclusions (ISGCI). `http://www.graphclasses.org`, acesso em 2 de dezembro de 2014.

[10] C. de SIMONE e C. P. de MELLO. Edge-colouring of join graphs. *Theoretical Computer Science*, 355:364–370, 2006.

[11] J.-C. FOURNIER. Méthode et théorème générale de coloration des arêtes. *J. Math. Pures Appl.*, 56:437–453, 1977.

[12] D. G. HOFFMAN e C. A. RODGER. The chromatic index of complete multipartite graphs. *Journal of Graph Theory*, 16(2):159–163, 1992.

[13] I. HOLYER. The $\mathcal{NP}$-completeness of edge-colouring. *SIAM J. Comput.*, 10(4):718–720, 1981.

[14] Y. Jing-Ho, C. Jer-Jeong, e G. J. Chang. Quasi-threshold graphs. *Discr. Appl. Math.*, 69:247–255, 1996.

[15] D. KÖNIG. Über graphen ihre anwendung auf determinantentheorie und mengenlehre. *Math. Ann.*, 77:453–465, 1916.

[16] D. LOKSHTANOV, F. MANCINI, e C. PAPADOPOULOS. Characterizing and computing minimal cograph completions. *Discr. Appl. Math.*, 158(7):755–764, 2010.

[17] R. C. MACHADO e C. M. de FIGUEIREDO. Decompositions for edge-coloring join graphs and cobipartite graphs. *Discr. Appl. Math.*, 158:1336–1342, 2010.

[18] T. NIESSEN. How to find overfull subgraphs in graphs with large maximum degree, II. *Electr. J. Comb.*, 8(1), 2001.

[19] P.ERDŐS e R. J. WILSON. On the chromatic index of almost all graphs. *Journal of Combinatorial Theory, Série B*, 23(2–3):255–257, 1977.

[20] M. J. Plantholt. The chromatic index of graphs with a spanning star. *Journal of Graph Theory*, 5:45–53, 1981.

[21] C. D. Simone e A. Galluccio. Edge-colouring of joins of regular graphs II. *Jornal of Combinatorial Optmization*, 25:78–90, 2011.

[22] V. G. VIZING. On an estimate of the chromatic class of a p-graph. *Diskretnyi Analiz 3*, p. 25–30, 1964 (em russo).

APÊNDICES

APÊNDICE A – Lemata

Todos as demonstrações apresentadas neste apêndice são oriundas de um artigo de Bretscher [2].

Lema A.1 (A Condição dos Quatro Pontos)**.** *Uma permutação σ é uma ordenação LexBFS de G se e somente se $\forall x, y, z \in \sigma$ tais que $x \prec_\sigma y \prec_\sigma z$, se $xy \notin E(G)$ e $xz \in E(G)$, então $\exists w \in \sigma$ tal que $w \prec_\sigma x$, $wy \in E(G)$ e $wz \notin E(G)$.*

Demonstração. No momento em que x foi tomado como pivô, y e z estavam em células diferentes, caso contrário teríamos $z \prec_\sigma y$. Por contradição, vamos assumir que, para todo vértice u tal que $u \prec_\sigma x$, em nenhum caso temos que $uy \in E(G)$ e $uz \notin E(G)$. Isso quer dizer que a quebra da célula em que y e z estavam foi ocasionada por um vértice u tal que $uy \notin E(G)$ e $uz \in E(G)$. Portanto, $z \prec_\sigma y$. Contradição. ⧫

Lema A.2. *Toda ordenação LexBFS de um cografo G é livre de guarda-chuvas (ou seja, não pode ocorrer o caso em que $\exists x, y, z \in \sigma$ tais que $x \prec_\sigma y \prec_\sigma z$, $xy \notin E(G)$, $yz \notin E(G)$ e $xy \in E(G)$).*

Demonstração. Faremos a demonstração por contradição e, para tanto, assumiremos que essa sequência de vértices (xyz) seja a primeira que forma um *guarda-chuva* em σ. Pelo Lema A.1, $\exists w \in \sigma$ tal que $w \prec_\sigma x$, $wy \in E(G)$ e $wz \notin E(G)$. Vamos analisar se w e x são vizinhos ou não:

(i) se $wx \in E(G)$, então $ywxz$ formam um P_4 e G não é um cografo. Contradição.

(ii) se $wx \notin E(G)$, então a sequência wxy também forma um *guarda-chuva*; assim, xyz não é a primeira sequência de vértices que forma um *guarda-chuva* em σ. Contradição. ⧫

Lema A.3. *Sejam G um cografo e σ=LexBFS(G, τ) para uma permutação τ qualquer de $V(G)$. Para todo $v \in V$ e todo $i < j$ vale que:*

$$\forall x \in S_i(v), \forall y \in S_j(v), xy \notin E(G) \tag{A.4}$$

$$N_<(S_j(v)) \subsetneq N_<(S_i(v)) \tag{A.5}$$

Demonstração. Para provar (A.4) (p. 79) por contradição, vamos assumir que $xy \in E(G)$. Se x e y estão em *fatias* diferentes, então há algum $u \in S^A(v)$ tal que $ux \in E(G)$ e $uy \notin E(G)$. Além disso, como $vu \in E(G)$, $vx \notin E(G)$, $vy \notin E(G)$ e $xy \in E(G)$, $vuxy$ formará um P_4. Contradição. No caso de (A.5) (p. 79), também por contradição, vamos assumir que $u \in N_<(S_j(v))$ e $u \notin N_<(S_i(v))$. Sabemos, em razão de (A.4) (p. 79), que $xy \notin E(G)$. Nesse caso, contudo, $\{u, x, y\}$ é um *guarda-chuva* e σ não atende ao Lema A.2 (p. 79). Contradição. ♦

Lema A.6. *Sejam* σ=LexBFS(G, τ) *para uma permutação* τ *qualquer de* $V(G)$, $\overline{\sigma}^-$=LexBFS$^-(G, \sigma)$ *e* M *um módulo de* G. *O primeiro vértice de* M *com relação a* σ *é* x *se e somente se* x *for o primeiro vértice de* M *com relação a* $\overline{\sigma}^-$. *Além disso,* $S(x)$ *e* $\overline{S}(x)$ *são as menores fatias de* σ *e* $\overline{\sigma}^-$, *respectivamente, que contêm* M.

Demonstração. ($\Rightarrow$) Seja x o primeiro vértice de M com relação a σ. Além disso, seja $\overline{S}(u)$ a *fatia* mínima de $\overline{\sigma}^-$ que contém M. Por contradição, vamos supor que $u \neq x$. Sabemos que, pelo fato de M ser um módulo, os pivôs externos a ele não conseguem dividi-lo em células. Portanto, dentro de $\overline{S}(u) \setminus \{u\}$ haverá uma outra *fatia* contendo M. Assim, $\overline{S}(u)$ não pode ser a *fatia* mínima de $\overline{\sigma}^-$ que contém M. Contradição.

($\Leftarrow$) Seja x o primeiro vértice de M com relação a $\overline{\sigma}^-$. No instante em que x se torna pivô em $\overline{\sigma}^-$, M está incluso na *fatia* $\overline{S}(x)$ (caso contrário, M não seria um módulo). Nesse caso, x é o primeiro vértice de $\overline{S}(x)$ visitado em $\overline{\sigma}^-$ porque ele também foi o primeiro vértice de $\overline{S}(x)$ visitado em σ (que é utilizada como permutação de desempate no algoritmo LexBFS$^-$). Logo, x também é o primeiro vértice de M visitado em σ. ♦

Observação A.7. *Sejam* σ=LexBFS(G, τ) *para uma permutação* τ *qualquer de* $V(G)$; $\overline{\sigma}^-$=LexBFS$^-(G, \sigma)$; $\overline{S}(v)$ *uma fatia que contém* w *em* $\overline{\sigma}^-$. *Se* $v \neq w$ *então* $v \prec_\sigma w$ *em* σ.

Demonstração. Por contradição, vamos supor que $w \prec_\sigma v$. Sabemos que, no instante em que v é selecionado como pivô em $\overline{\sigma}^-$, v e w pertencem à mesma célula (pois ambos pertencem a $\overline{S}(v)$), mas v é tomado no lugar de w em razão do critério de desempate fornecido pela permutação σ, passada como parâmetro ao algoritmo LexBFS$^-$. Ou seja, $v \prec_\sigma w$. Contradição. ♦

Lema A.8. *Sejam σ=LexBFS(G,τ) para uma permutação τ qualquer de $V(G)$ e $S(v)$ a fatia mínima de σ que contém o P_4 induzido p. Se v não for vértice de nenhum P_4 de G, então $V(p) \cap S^A(v)$ é o conjunto formado pelos dois vértices internos de p.*

Demonstração. Analisaremos três possibilidades:

(i) Os vértices de p estão todos em $S^A(v)$, que é também uma *fatia* de σ. Nesse caso $S(v)$ não é mínima. Contradição.

(ii) Os vértices de p estão todos em $S^N(v)$. Uma vez que $S(v)$ é mínima, algum vértice u de $S^A(v)$ deve ser vizinho de um subconjunto próprio dos vértices de p, dividindo-o em ao menos duas *fatias*. Há portanto uma aresta xy em p tal que $ux \in E(G)$ e $uy \notin E(G)$. Nesse caso, contudo, $vuxy$ forma um P_4. Contradição.

(iii) Resta o caso em que v é adjacente a um subconjunto próprio dos vértices de p. E para que v não faça parte de um P_4, a única alternativa é que v seja adjacente aos dois vértices internos de p e não seja adjacente aos extremos de p. ⧫

Corolário A.9. *Sejam σ=LexBFS(G,τ) para uma permutação τ qualquer de $V(G)$, $\overline{\sigma}^{-}$=LexBFS$^{-}(G,\sigma)$, w o primeiro vértice de σ que pertence a um P_4 e p um P_4 que cobre o vértice w. Se w for um extremo de p, então será um vértice interno de $\overline{p}$ em $\overline{\sigma}^{-}$.*

Demonstração. Seja $\overline{S}(v)$ a *fatia* mínima que contém $V(\overline{p})$ em $\overline{\sigma}^{-}$. Analisaremos duas possibilidades:

(i) Se v pertence a $\overline{p}$, então $v = w$ e, em razão da complementação feita em p, w passou a ser um vértice interno de $\overline{p}$. Finalizamos.

(ii) Se v não pertence a $\overline{p}$, então, pela Observação A.7 (p. 80), v ocorre antes de p em σ. Logo, v não pertence a nenhum P_4, e pelo Lema A.8 temos que os vértices internos de p surgem antes dos extremos em $\overline{\sigma}^{-}$. Finalizamos. ⧫

Lema A.10. *Sejam σ=LexBFS(G,τ) para uma permutação τ qualquer de $V(G)$ e $p = abcd$ um P_4 induzido de G (com extremos a e d). Se $S(a)$ contém $V(p)$, então existem dois inteiros distintos i e j tais que $S_i(a)$ e $S_j(a)$ não atendem à PSV.*

Demonstração. Uma vez que a é adjacente a b mas não é adjacente a c nem a d, b está em $S^A(a)$ e c e d estão ambos em $S^N(a)$. Além disso, sabemos que c e d não estão na

mesma célula, pois b é adjacente a c mas não é adjacente a d. Sejam então $S_i(a)$ e $S_j(a)$, $i < j$, as células em que estão c e d. Suponhamos que $c \in S_i(a)$ e $d \in S_j(a)$. A existência de uma aresta entre c e d faz com que $N^\ell(S_j(a)) \nsubseteq N^\ell(S_i(a))$. O mesmo ocorre se $d \in S_i(a)$ e $c \in S_j(a)$. Portanto, σ não atende à PSV. ⧫

Lema A.11. *Sejam σ=LexBFS(G, τ) para uma permutação τ qualquer de $V(G)$ e $p = abcd$ um P_4 induzido de G (com extremos a e d). Se $S(v)$ é a menor fatia que contém $V(p)$, de tal forma que v não pertence a nenhum P_4, então existem dois inteiros distintos i e j tais que $S_i(v)$ e $S_j(v)$ não atendem à PSV.*

Demonstração. Pelo Lema A.8 (p. 81), $b, c \in S^A(v)$ e $a, d \in S^N(v)$. Além disso, sabemos que a e d não estão na mesma célula, pois b é adjacente a a mas não é adjacente a d. Sejam então $S_i(v)$ e $S_j(v)$, $i < j$, as células em que estão a e d. Suponhamos que $a \in S_i(v)$ e $d \in S_j(v)$. Uma vez que c e d são vizinhos mas c e a não são, temos que $N^\ell(S_j(v)) \nsubseteq N^\ell(S_i(v))$. De maneira análoga, suponhamos que $d \in S_i(v)$ e $a \in S_j(v)$. Uma vez que b e a são vizinhos mas b e d não são, temos que $N^\ell(S_j(v)) \nsubseteq N^\ell(S_i(v))$. Em ambos os casos, σ não atende à PSV. ⧫

APÊNDICE B – Construção da coárvore

Todos as demonstrações apresentadas aqui são oriundas de um artigo de Bretscher [2].

Para a construção da coárvore, assumimos que todos os grafos deste apêndice são cografos. Tal construção é possível em razão dos lemas apresentados abaixo.

Lema B.1. *Sejam* σ=LexBFS(G,τ) *para uma permutação* τ *qualquer de* $V(G)$ *e* x *o primeiro vértice de* σ. *Cada* $S_i(x)$, *para* $i>0$, *corresponde aos vértices da subárvore* T^x_{0i}.

Demonstração. Designamos o conjunto de vértices da árvore T por $V(T)$.

($\Rightarrow$) Vamos provar por indução que todo vértice de $S_i(x) \subseteq V(T^x_{0i})$, para $i>0$. Primeiramente, façamos a demonstração da base de indução, qual seja, $S_1(x) \subseteq V(T^x_{01})$:

Tomemos um vértice $y \in S_1(x)$. Por contradição, vamos assumir que $y \notin V(T^x_{01})$. Nesse caso, deve haver um vértice $z \in V(T^x_{1k})$ (para algum $k>0$), tal que $wz \in E(G)$ para algum $w \in V(T^x_{01})$ mas $yz \notin E(G)$. Isso significa que w e y não estão na mesma *fatia*, pois possuem vizinhança à esquerda diferente. Além do mais, y está numa *fatia* à direita da *fatia* de w, já que a vizinhança local de y está contida na vizinhança local de w. Logo, $y \notin S_1(x)$. Contradição.

Para a indução pretendida, aplicaremos o mesmo raciocínio:

Base de indução: $S_1(x) \subseteq V(T^x_{01})$.

Hipótese de indução: Para todo $i' < i$, $S_{i'}(x) \subseteq V(T^x_{0i'})$.

Passo de indução: Queremos demonstrar que, em razão da hipótese de indução, $S_i(x) \subseteq V(T^x_{0i})$. Tomemos, portanto, um vértice $y \in S_i(x)$. Por contradição, vamos assumir que $y \notin V(T^x_{0i})$. Pela hipótese de indução, $y \in V(T^x_{0j})$ para algum $j > i$. Nesse caso, deve haver um vértice $z \in V(T^x_{1i})$ tal que $wz \in E(G)$ para algum $w \in V(T^x_{0i})$ mas $yz \notin E(G)$. Isso significa que w e y não estão na mesma *fatia*, pois possuem vizinhança à esquerda diferente. Além do mais, y está numa *fatia* à direita da *fatia* de w, já que a vizinhança local de y está contida na vizinhança local de w. Entretanto, sabemos pela hipótese de indução que w não está em nenhuma das *fatias* à esquerda de $S_i(x)$ (não pode estar em nenhuma $S_{i'}(x)$). Logo, $y \notin S_i(x)$. Contradição.

($\Leftarrow$) Vamos provar por indução que $V(T^x_{0i}) \subseteq S_i(x)$, para $i>0$. Primeiramente, façamos a demonstração da base de indução, qual seja, $V(T^x_{01}) \subseteq S_1(x)$:

Tomemos um vértice $y \in V(T^x_{01})$. Por contradição, vamos assumir que $y \notin S_1(x)$. Nesse caso, deve haver um vértice $z \in S^A(x)$ tal que $wz \in E(G)$ para algum $w \in S_1(x)$ mas $yz \notin E(G)$. Isso significa que w e y não estão na mesma subárvore com raiz em P_{xR}, pois $\text{lca}(w,z) \neq \text{lca}(y,z)$. Logo, $y \notin V(T^x_{01})$. Contradição.

Para a indução pretendida, aplicaremos o mesmo raciocínio:

Base de indução: $V(T^x_{01}) \subseteq S_1(x)$.

Hipótese de indução: Para todo $i' < i$, $V(T^x_{0i'}) \subseteq S'_i(x)$.

Passo de indução: Queremos demonstrar que, em razão da hipótese de indução, $(T^x_{0i}) \subseteq S_i(x)$. Tomemos um vértice $y \in V(T^x_{0i})$. Por contradição, vamos assumir que $y \notin S_i(x)$. Pela hipótese de indução, $y \in S_j(x)$ para algum $j > i$. Nesse caso, deve haver um vértice $z \in S^A(x)$ tal que $wz \in E(G)$ para algum $w \in S_i(x)$ mas $yz \notin E(G)$. Isso significa que w e y não estão na mesma subárvore com raiz em P_{xR}, pois $\text{lca}(w,z) \neq \text{lca}(y,z)$. Além do mais, a raiz da subárvore de y está mais distante de x do que a raiz da subárvore de y. Entretanto, sabemos pela hipótese de indução que w não está em nenhuma das subárvores $T^x_{0i'}$. Logo, $y \notin V(T^x_{0i})$. Contradição. ♦

Corolário B.2. *Sejam* $\overline{\sigma}^{-}$=LexBFS$^{-}(G,\tau)$ *para uma permutação* τ *qualquer de* $V(G)$ *e* x *o primeiro vértice de* $\overline{\sigma}^{-}$. *Cada* $\overline{S}_i(x)$, *para* $i > 0$, *corresponde aos vértices da subárvore* T^x_{1i}.

Demonstração. Basta aplicar o Lema B.1 (p. 83) em $\overline{\sigma}^{-}$ e $\overline{G}$, já que a coárvore de $\overline{G}$ corresponde à coárvore de G com os nós 0 e 1 trocados. ♦

Corolário B.3. *Sejam* σ=LexBFS(G,τ) *para uma permutação* τ *qualquer de* $V(G)$ *e* $\overline{\sigma}^{-}$=LexBFS$^{-}(G,\pi)$ *para uma permutação* π *qualquer de* $V(G)$. *Se* x *for o primeiro vértice de* σ *e* $\overline{\sigma}^{-}$, *então o caminho* P_{xR} *da coárvore de* G *pode ser construído com as fatias em* $S^N(x)$ *e* $\overline{S^N}(x)$.

Demonstração. A demonstração decorre do Lema B.1 (p. 83) e do Corolário B.2, restando apenas saber qual dentre as subárvores T^x_{01} e T^x_{11} tem como raiz o nó pai de x. Isso é fácil de constatar porque, sendo $y \in T^x_{01}$ e $z \in T^x_{11}$, o $\text{lca}(y,z)$ é 0 se a raiz de T^x_{11} for o pai de x e 1 caso contrário. Basta, portanto, que se saiba a adjacência entre y e z, já que $yz \in E(G) \iff \text{lca}(y,z) = 1$. ♦

Em que pese a possibilidade de aplicar recursivamente a construção dos caminhos P_{xR}, conforme sugerido no Corolário B.3 (p. 84), até que se construa a coárvore,

tal modo de construção não seria linear porque demandaria duas buscas (σ e $\overline{\sigma^{-}}$) para cada um desses caminhos. Queremos aproveitar as buscas que já foram feitas na etapa de reconhecimento, e para tanto precisaremos do seguinte resultado.

Lema B.4. *Se os vértices de T_{0i}^{x} forem vértices de $S_i(x) = S(y)$ para algum vértice x da LexBFS σ e para algum $i > 0$, então, para todo $j > 0$, $S_j(x)$ corresponde aos vértices da subárvore T_{0j}^{y}.*

Demonstração. Basta demonstrar o caráter recursivo do Lema B.1 (p. 83). Fácil de verificar, pois, pela própria definição, *fatias* de uma permutação LexBFS também são permutações LexBFS. Além disso, $S_i(x) = S(y)$ induz um módulo, que também é um cografo. ⧫

A construção da coárvore é realizada pelo Algoritmo B.1, a qual utiliza a função Nó para criar uma estrutura de dados que representa um nó interno da árvore. Essa função recebe como parâmetros a operação (0 para união e 1 para junção) e uma subárvore. A coárvore final é representada pelo nó raiz.

Construa-coárvore(v):
Entrada: *Um vértice v de um cografo G.*
Saída: *A coárvore de $G[v \cup S^N(v) \cup \overline{S^N}(v)]$.*
1 **se** $S_1(v) = \varnothing$ e $\overline{S}_1(v) = \varnothing$:
2 **devolva** v;
3 **se** $S_1(v) = \varnothing$:
4 **devolva** Nó$(1, v,$Construa-coárvore$(\overline{v}[1])$;
5 **se** $\overline{S}_1(v) = \varnothing$:
6 **devolva** Nó$(0, v,$Construa-coárvore$(v[1])$;
7 **se** $v[1]\overline{v}[1] \in E(G)$: $\omega \leftarrow 0$; **senão**: $\omega \leftarrow 1$;
8 $T \leftarrow v$;
9 $k \leftarrow 1$;
10 $i \leftarrow 1$;
11 **enquanto** ($\omega = 0$ e $S_i(v) \neq \varnothing$) ou ($\omega = 1$ e $\overline{S}_i(v) \neq \varnothing$):
12 **se** $\omega = 0$: $u \leftarrow v[i]$; **senão**: $u \leftarrow \overline{v}[i]$;
13 $T \leftarrow$ Nó$(\omega, T,$Construa-coárvore$(u))$;
14 $\omega \leftarrow 1 - \omega$;
15 $k \leftarrow k + 1$;
16 $i \leftarrow \lceil k/2 \rceil$;
17 **devolva** T.

Algoritmo B.1: Construção da coárvore

A execução do Algoritmo B.1 no cografo da Figura 2.1a (p. 39) resulta no ani-

nhamento de chamadas à função Nó exibido em (B.5):

$$\mathbf{1}(\mathbf{0}(\mathbf{1}(\mathbf{0}(z,y),x),w),\mathbf{0}(d,\mathbf{0}(e,\mathbf{0}(\mathbf{1}(u,v),\mathbf{0}(\mathbf{1}(b,a),c))))) \tag{B.5}$$

Em razão do Lema B.4 (p. 85), concluímos que o Algoritmo B.1 (p. 85), ao receber como parâmetro o primeiro vértice de uma permutação LexBFS de um cografo G, constrói a respectiva coárvore em tempo linear.

APÊNDICE C – Listagem do código-fonte do programa Cograph

C.1 Arquivo cograph.py

```
#
#    cograph - Cograph Generation and Chromatic Analysis
#    Copyright (C) 2014 Alex Reimann Cunha Lima
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
import sqlite3

# Constants
DB_FILE     = "cograph.db"
MAX_ORDER   = 13
VF_DELTA    = 1
INF_GIRTH   = 5
OP_VERT     = "."
OP_UNION    = "0"
OP_JOIN     = "1"
OPERATIONS = (OP_UNION, OP_JOIN, OP_VERT)

# Globals
g_nullGraph = None
g_cgset = {}
g_cglst = []
g_enc_set = {}
g_enc_lst = []
g_enc_nod = []

# Cotree encoding
# ------------------------------------------------------------------------------
class CotreeNode:
    def __init__(self, op, a, b):
        self.op = op
        self.a = a
        self.b = b
        self.string = self.op

    def __str__(self):
        return self.string

    def sort(self):
        if self.a != None and self.b != None:
```

```
52              if self.a.op == self.op or self.a.op != self.op:
53                  lst = []
54                  n = self.a
55                  while n.op == self.op:
56                      lst.append(n.a)
57                      n = n.b
58                  lst.append(n)
59                  n = self.b
60                  while n.op == self.op:
61                      lst.append(n.a)
62                      n = n.b
63                  lst.append(n)
64                  lst = sorted(lst, key = lambda x: x.string)
65                  bstr = ""
66                  for i in range(1, len(lst)):
67                      bstr += lst[i].string
68                  for i in range(len(lst) - 2):
69                      bstr += self.op
70                  self.a = lst[0]
71                  self.b = g_enc_nod[g_enc_set[bstr]]
72              elif self.b.op != self.op:
73                  if self.a.string > self.b.string:
74                      self.a, self.b = self.b, self.a
75              self.string = self.a.string + self.b.string + self.op
76          if self.string in g_enc_set:
77              return g_enc_nod[g_enc_set[self.string]]
78          addEncoding(self.string, self)
79          return self
80
81  def makeCotreeNode(op, a, b):
82      if op == "" or op == OP_VERT:
83          s = op
84      else:
85          s = a.string + b.string + op
86          if s in g_enc_set:
87              return g_enc_nod[g_enc_set[s]]
88          s = b.string + a.string + op
89      if s in g_enc_set:
90          return g_enc_nod[g_enc_set[s]]
91      node = CotreeNode(op, a, b)
92      return node.sort()
93
94  def validateCotree(s):
95      vnode = g_enc_nod[g_enc_set[OP_VERT]]
96      stack = []
97      for op in s:
98          if op == OP_VERT:
99              stack.append(vnode)
100         else:
101             if len(stack) < 2:
102                 return None
103             b = stack.pop()
104             a = stack.pop()
105             stack.append(makeCotreeNode(op, a, b))
106     if len(stack) != 1:
107         return None
108     n = stack.pop()
109     if n == None:
```

```
110         return None
111     return n.string
112 
113 def addEncoding(s, node):
114     n = len(g_enc_lst)
115     g_enc_set[s] = n
116     g_enc_lst.append(s)
117     g_enc_nod.append(node)
118     return n
119 
120 def genGraphs(cur = None):
121     for i in range(MAX_ORDER + 1):
122         makeOrder(i, cur)
123 
124 def initEncoding():
125     global orderBase
126     global currentOrder
127     orderBase = []
128     currentOrder = -1
129 
130 def makeOrder(order, cur):
131     global currentOrder
132     global orderBase
133     if (order == currentOrder + 1):
134         currentOrder += 1
135         orderBase.append(len(g_enc_lst))
136     elif order > currentOrder + 1:
137         return
138     orderMax = (order // 2) + 1
139     if order == 0:
140         n = makeCotreeNode("", None, None)
141         writeCGData(n, None, None, cur)
142         return
143     elif order == 1:
144         n = makeCotreeNode(OP_VERT, None, None)
145         writeCGData(n, None, None, cur)
146         return
147     for i in range(1, orderMax):
148         orderA = i
149         orderB = order - orderA
150 
151         startA = orderBase[orderA]
152         endA = orderBase[orderA + 1]
153         startB = orderBase[orderB]
154         endB = orderBase[orderB + 1]
155         for j in range(startA, endA):
156             for k in range(startB, endB):
157                 nodeA = g_enc_nod[j]
158                 nodeB = g_enc_nod[k]
159                 n = makeCotreeNode(OP_UNION, nodeA, nodeB)
160                 writeCGData(n, nodeA, nodeB, cur)
161                 n = makeCotreeNode(OP_JOIN, nodeA, nodeB)
162                 writeCGData(n, nodeA, nodeB, cur)
163 
164 def encodeOperation(op, a, b):
165     assert op in OPERATIONS
166     an = g_enc_nod[g_enc_set[a]]
167     bn = g_enc_nod[g_enc_set[b]]
```

```
    n = makeCotreeNode(op, an, bn)
    return n.string

def delCotreeVertex(enc, k):
    stack = []
    out = ""
    for i in range(len(enc)):
        c = enc[i]
        if c == OP_VERT:
            if k == 0:
                stack.append(0)
            else:
                stack.append(1)
                out += c
            k -= 1
        else:
            if len(stack) < 2:
                return None
            b = stack.pop()
            a = stack.pop()
            stack.append(a | b)
            if (a & b) == 1:
                out += c
    if len(stack) != 1:
        return None
    return validateCotree(out)

def induceCotree(enc, indset):
    if enc == "":
        return ""
    stack = []
    out = ""
    p = 0
    for i in range(len(enc)):
        c = enc[i]
        if c == OP_VERT:
            if not indset[p]:
                stack.append(0)
            else:
                stack.append(1)
                out += c
            p += 1
        else:
            if len(stack) < 2:
                return None
            b = stack.pop()
            a = stack.pop()
            stack.append(a | b)
            if (a & b) == 1:
                out += c

    if len(stack) != 1:
        return None
    return validateCotree(out)

# Cograph building
# ---------------------------------------------------------------------------
def loadGraph(encodedCotree, cl=0):
```

```
    if encodedCotree == "":
        return makeCoNode(None, None, None)
    stack = []
    forceClass = 0
    for i in range(len(encodedCotree)):
        c = encodedCotree[i]
        assert c in OPERATIONS
        if i == len(encodedCotree) - 1:
            forceClass = cl
        if c == OP_VERT:
            a = None
            b = None
        else:
            if len(stack) < 2:
                return None
            a = stack.pop()
            b = stack.pop()
        stack.append(makeCoNode(c, a, b, forceClass))
    if len(stack) != 1:
        return None
    return stack.pop()

def makeCoNode(op, a, b, cl=0):
    if op != None:
        assert op in OPERATIONS
        if op == OP_VERT:
            assert a == None and b == None
        else:
            assert a != None and b != None
        if a == None or b == None:
            cmd = op
        else:
            apos = len(a.cmd)
            bpos = len(b.cmd)
            while a.cmd[apos - 1:apos] == op: apos -= 1
            while b.cmd[bpos - 1:bpos] == op: bpos -= 1
            acmd = a.cmd[:apos]
            bcmd = b.cmd[:bpos]
            if acmd > bcmd:
                acmd, bcmd = bcmd, acmd
            cmd = acmd + bcmd + a.cmd[apos:] + b.cmd[bpos:] + op
    else:
        a = None
        b = None
        cmd = ""
    if cmd in g_cgset:
        o = g_cgset[cmd]
    else:
        o = CoNode(op, a, b, cmd, len(g_cglst), cl)
        g_cgset[cmd] = o
        o.build()
        g_cglst.append(o)
    return o

class CoNode:
    def __init__(self, op, a, b, cmd, index, cl=0):
        self.id = index
        # cotree data
```

```
284         self.op = op
285         self.a = a
286         self.b = b
287         self.cmd = cmd
288         self.height = 0             # cotree height
289         self.numchildren = 0        # number root's children
290         # cograph data
291         self.V = []
292         self.VF = []                # vertices flags
293         self.VDN = []               # number of delta neighbours
294         self.numV = 0
295         self.numE = 0
296         # cograph properties
297         self.cpClass = 1            # null graphs are class one
298         self.maxDegree = 0
299         self.maxEDegree = 0
300         self.minDegree = 0
301         self.connected = True       # null graphs are connected
302         self.overfull = False       # null graphs are not overfull
303         self.SO = False
304         self.NO = False
305         self.deltaSubgraphs = {}
306         self.complete = True        # null graphs are complete
307         self.cycle = False          # null graphs have no cycles
308         self.clique = 0             # clique number
309         self.combinations = 0       # num of diff combos for building this graph
310         self.complement = self
311         self.fullHeight = True
312         self.girth = INF_GIRTH
313         self.ovsub = None           # the overfull subgraph (if exists)
314         self.chromNum = 0
315         self.chromInd = 0
316         self.strangers = False      # every pair of vertices are neighbours
317         self.c4 = False
318         self.star = False
319
320     def addEdge(self, x1, x2):
321         self.V[x1].append(x2)
322         self.V[x2].append(x1)
323         self.numE += 1
324
325     def calcDeltaSubgraphs(self):
326         if self.numV <= 1:
327             return
328         if self.overfull:
329             return
330         if self.op == OP_UNION:
331             return
332         for i in range(self.numV):
333             enc = delCotreeVertex(self.cmd, i)
334             assert(enc in g_cgset)
335             g = g_cgset[enc]
336             if g.maxDegree == self.maxDegree:
337                 self.deltaSubgraphs[g.id] = g
338
339     def calcSO(self):
340         self.SO = False
341         self.ovsub = None
```

```
342         if self.overfull:
343             self.SO = True
344             self.ovsub = self
345             if self.numV == self.maxDegree + 1:
346                 self.NO = True
347             return
348         if self.op == OP_UNION:
349             self.SO = self.a.SO or self.b.SO
350             if self.a.NO:
351                 self.NO = True
352                 self.ovsub = self.a.ovsub
353             elif self.b.NO:
354                 self.NO = True
355                 self.ovsub = self.b.ovsub
356             elif self.SO:
357                 if self.a.SO:
358                     self.ovsub = self.a.ovsub
359                 else:
360                     self.ovsub = self.b.ovsub
361             return
362         elif self.op == OP_JOIN:
363             for gid, g in self.deltaSubgraphs.items():
364                 if g.SO:
365                     self.SO = True
366                     self.ovsub = g.ovsub
367                     self.NO = self.ovsub.NO
368                     break
369 
370     def calcOverfull(self):
371         if self.numE > self.maxDegree * (self.numV // 2):
372             self.overfull = True
373         else:
374             self.overfull = False
375 
376     def calcClass(self):
377         if self.SO:
378             self.cpClass = 2
379         else:
380             self.cpClass = 1
381         self.chromInd = self.maxDegree + self.cpClass - 1
382 
383     def calcCore(self):
384         if self.complete:
385             self.core = self
386         elif self.op == OP_VERT:
387             self.core = self
388         elif self.op == OP_UNION:
389             if self.a.maxDegree > self.b.maxDegree:
390                 self.core = self.a.core
391             elif self.b.maxDegree > self.a.maxDegree:
392                 self.core = self.b.core
393             else:
394                 en = encodeOperation(OP_UNION, self.a.core.cmd, self.b.core.cmd)
395                 self.core = g_cgset[en]
396         elif self.op == OP_JOIN:
397             adegree = self.a.maxDegree + self.b.numV
398             bdegree = self.b.maxDegree + self.a.numV
399             if adegree > bdegree:
```

```
400                 self.core = self.a.core
401             elif bdegree > adegree:
402                 self.core = self.b.core
403             else:
404                 en = encodeOperation(OP_JOIN, self.a.core.cmd, self.b.core.cmd)
405                 self.core = g_cgset[en]
406 
407     def calcFlags(self):
408         while len(self.VF) < self.numV:
409             self.VF.append(0)
410             self.VDN.append(0)
411 
412         for u in range(self.numV):
413             if len(self.V[u]) == self.maxDegree:
414                 self.VF[u] |= VF_DELTA
415 
416         for u in range(self.numV):
417             if self.VF[u] & VF_DELTA:
418                 self.VDN[u] += 1
419             for w in self.V[u]:
420                 if self.VF[w] & VF_DELTA:
421                     self.VDN[u] += 1
422 
423     def induce(self, inset):
424         enc = induceCotree(self.cmd, inset)
425         return g_cgset[enc]
426 
427     def calcSemiCore(self):
428         if self.complete:
429             self.semiCore = self
430             return
431         induceSet = []
432         for u in range(self.numV):
433             if self.VDN[u] > 0:
434                 induceSet.append(True)
435             else:
436                 induceSet.append(False)
437         self.semiCore = self.induce(induceSet)
438 
439     def calcChildren(self):
440         self.numchildren = 2
441         if self.a.op == self.op:
442             self.numchildren += self.a.numchildren - 1
443         if self.b.op == self.op:
444             self.numchildren += self.b.numchildren - 1
445 
446     def calcHeight(self):
447         ha = self.a.height
448         hb = self.b.height
449         if self.a.op != self.op:
450             ha += 1
451         if self.b.op != self.op:
452             hb += 1
453         self.height = max(ha, hb)
454         if (ha == hb) and (self.a.fullHeight and self.b.fullHeight):
455             self.fullHeight = True
456         else:
457             self.fullHeight = False
```

```
458
459     def unionRaw(self):
460         self.numV = self.a.numV + self.b.numV
461         self.numE = self.a.numE + self.b.numE
462         while len(self.V) < self.numV:
463             self.V.append([])
464         for x1 in range(self.a.numV):
465             edges = self.a.V[x1]
466             for x2 in edges:
467                 self.V[x1].append(x2)
468         splitpos = self.a.numV
469         for x1 in range(self.b.numV):
470             edges = self.b.V[x1]
471             for x2 in edges:
472                 self.V[splitpos + x1].append(splitpos + x2)
473         self.cycle = self.a.cycle or self.b.cycle
474         self.girth = min(self.a.girth, self.b.girth)
475         self.c4 = self.a.c4 or self.b.c4
476
477     def union(self):
478         self.unionRaw()
479         self.connected = False
480         self.maxDegree = max(self.a.maxDegree, self.b.maxDegree)
481         self.minDegree = min(self.a.minDegree, self.b.minDegree)
482         self.maxEDegree = max(self.a.maxEDegree, self.b.maxEDegree)
483         self.chromNum = max(self.a.chromNum, self.b.chromNum)
484         self.strangers = True
485         self.star = self.a.star or self.b.star
486
487     def join(self):
488         self.unionRaw()
489         for x1 in range(self.a.numV):
490             for y in range(self.b.numV):
491                 x2 = y + self.a.numV
492                 self.addEdge(x1, x2)
493         self.connected = True
494         adeg = self.a.maxDegree + self.b.numV
495         bdeg = self.b.maxDegree + self.a.numV
496         self.maxDegree = max(adeg, bdeg)
497         adeg = self.a.minDegree + self.b.numV
498         bdeg = self.b.minDegree + self.a.numV
499         self.minDegree = min(adeg, bdeg)
500         # cycle, girth, c4 already pre-calculated by unionRaw
501         if self.a.numE > 0 or self.b.numE > 0:
502             self.cycle = True
503             self.girth = 3
504         elif self.a.numV > 1 and self.b.numV > 1:
505             self.cycle = True
506             self.girth = min(self.girth, 4)
507         Mmax = self.a.maxDegree + self.b.maxDegree
508         Amax = self.a.maxEDegree
509         Bmax = self.b.maxEDegree
510         if self.a.numE > 0:
511             Amax += self.b.numV + self.b.numV
512         if self.b.numE > 0:
513             Bmax += self.a.numV + self.a.numV
514         self.maxEDegree = max(Mmax, Amax, Bmax)
515         self.chromNum = self.a.chromNum + self.b.chromNum
```

```
516         self.strangers = self.a.strangers or self.b.strangers
517         if self.a.strangers and self.b.strangers:
518             self.c4 = True
519         if self.numV >= 3 and not self.cycle:
520             self.star = True
521         else:
522             self.star = False
523 
524     def singleton(self):
525         self.numV = 1
526         self.numE = 0
527         self.V.append([])
528         self.maxDegree = 0
529         self.minDegree = 0
530         self.connected = True
531         self.overfull = False
532         self.SO = False
533         self.cpClass = 1
534         self.cycle = False
535         self.a = g_nullGraph
536         self.b = g_nullGraph
537         self.height = 0
538         self.numchildren = 0
539         self.clique = 1
540         self.chromNum = 1
541         self.chromInd = 0
542 
543     def calcComplete(self):
544         if self.numE == (self.numV * (self.numV - 1)) // 2:
545             self.complete = True
546         else:
547             self.complete = False
548 
549     def calcClique(self):
550         if self.op == OP_UNION:
551             self.clique = max(self.a.clique, self.b.clique)
552         elif self.op == OP_JOIN:
553             self.clique = self.a.clique + self.b.clique
554 
555     def build(self):
556         if self.op == None:             # null graph
557             self.a = self
558             self.b = self
559         else:
560             assert self.op in OPERATIONS
561             if self.op == OP_VERT:
562                 self.singleton()
563             else:
564                 if self.op == OP_UNION:
565                     self.union()
566                 elif self.op == OP_JOIN:
567                     self.join()
568                 self.calcChildren()
569                 self.calcHeight()
570                 self.calcOverfull()
571                 self.calcDeltaSubgraphs()
572                 self.calcSO()
573                 self.calcClass()
```

```
574         self.calcComplete()
575         self.calcCore()
576         self.calcFlags()
577         self.calcSemiCore()
578         self.calcClique()
579 
580 def updateGraphs(cur):
581     tr = str.maketrans("01", "10")
582     for g in g_cglst:
583         ccmd = validateCotree(g.cmd.translate(tr))
584         cid = -1
585         if ccmd != None:
586             cg = g_cgset[ccmd]
587             g.complement = cg
588             cid = g.complement.id
589         cur.execute("UPDATE gr SET combinations=%d,complement=%d WHERE id=%d" %
590             (g.combinations, cid, g.id))
591 
592 def writeGraphData():
593     con = sqlite3.connect(DB_FILE)
594     with con:
595         cur = con.cursor()
596         cur.execute("DROP TABLE IF EXISTS gr")
597         cur.execute("DROP TABLE IF EXISTS op")
598         cur.execute("CREATE TABLE gr("
599                     "id INT PRIMARY KEY, "
600                     "cotree TEXT, "
601                     "n INT, "
602                     "m INT, "
603                     "star INT, "
604                     "c4 INT, "
605                     "complete INT, "
606                     "clique INT, "
607                     "combinations INT, "
608                     "complement INT, "
609                     "connected INT, "
610                     "cycle INT, "
611                     "girth INT, "
612                     "maxdeg INT, "
613                     "mindeg INT, "
614                     "maxedeg INT, "
615                     "ov INT, "
616                     "so INT, "
617                     "no INT, "
618                     "ovsub INT, "
619                     "class INT, "
620                     "core INT, "
621                     "semicore INT, "
622                     "height INT, "
623                     "fullheight INT, "
624                     "numchildren INT, "
625                     "chromnum INT, "
626                     "chromind INT) WITHOUT ROWID")
627         cur.execute("CREATE TABLE op(op INT, g INT, a INT, b INT)")
628         genGraphs(cur)
629         updateGraphs(cur)
630 
631 def writeCGData(n, a, b, cur):
```

```
632     if cur == None:
633         return
634     creating = False
635     if n.string in g_cgset:
636         n = g_cgset[n.string]
637     else:
638         n = loadGraph(n.string)
639         creating = True
640     if a != None and b != None:
641         a = g_cgset[a.string]
642         b = g_cgset[b.string]
643         if not (a.numV <= b.numV):
644             a, b = b, a
645     if n.op != None:
646         op = OPERATIONS.index(n.op)
647     else:
648         op = len(OPERATIONS)
649     if a != None and b != None:
650         cur.execute("INSERT INTO op VALUES(%d, %d, %d, %d)" %
651             (op, n.id, a.id, b.id))
652     n.combinations += 1
653     if not creating: return
654     ovsub = 0
655     if n.ovsub != None:
656         ovsub = n.ovsub.id
657     cur.execute("INSERT INTO gr VALUES(" \
658         "%d, '%s', %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, "
659         "%d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d)" %
660         (n.id, n.cmd, n.numV, n.numE, n.star, n.c4, n.complete,
661         n.clique, n.combinations, n.complement.id, n.connected, n.cycle,
662         n.girth, n.maxDegree, n.minDegree, n.maxEDegree, n.overfull, n.SO,
663         n.NO, ovsub, n.cpClass, n.core.id, n.semiCore.id, n.height,
664         n.fullHeight, n.numchildren, n.chromNum, n.chromInd))
665
666 initEncoding()
667 writeGraphData()
```

C.2 Arquivo test.py

```
#
#    cograph - Cograph Generation and Chromatic Analysis
#    Copyright (C) 2014 Alex Reimann Cunha Lima
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
import sqlite3

con = sqlite3.connect("cograph.db")
baseQuery = "SELECT COUNT(*) " \
            "FROM op INNER JOIN gr AS r ON op.g = r.id " \
            "INNER JOIN gr AS a ON op.a  = a.id " \
            "INNER JOIN gr AS b ON op.b  = b.id " \
            "INNER JOIN gr AS core ON r.core = core.id " \
            "INNER JOIN gr AS acore ON a.core = acore.id " \
            "INNER JOIN gr AS bcore ON b.core = bcore.id " \
            "WHERE op.op = 1 "

def BasicTest(name, prop):
    print("==============================")
    print(" ", name, "test:")
    print("==============================")
    cur = con.cursor()
    query = baseQuery + prop
    cur.execute(query)
    total = cur.fetchall()[0][0]
    if total == 0:
        print("Unable to test", name, "due to lack of valid instances\n")
        return
    cur.execute(query + "AND r.class = 1")
    class1 = cur.fetchall()[0][0]
    cur.execute(query + "AND r.class = 2")
    class2 = cur.fetchall()[0][0]
    print("Class 1: %7d cases out of %d" % (class1, total))
    print("Class 2: %7d cases out of %d" % (class2, total))
    if total == class1:
        print(name, "passed the test\n")
    else:
        print(name, "DID NOT pass the test\n")

# Test Conjecture A
def Test_1():
    BasicTest("Conjecture A", "AND a.maxdeg = b.maxdeg AND acore.cycle = 0 ")

```

```
# Test Theorem 5.2
def Test_2():
    BasicTest("Theorem 5.2", "AND a.maxdeg = b.maxdeg AND acore.m = 0 ")

# Test Theorem 4.7
def Test_3():
    BasicTest("Theorem 4.7[Simone & Mello]", "AND a.maxdeg = b.maxdeg " \
              "AND a.class = 1 AND b.class = 1 ")

# Test Theorem 4.12
def Test_4():
    BasicTest("Theorem 4.12[Machado & Figueiredo]",
              "AND b.maxdeg < (b.n - a.n) ")

def smallQuery(title, query):
    cur = con.cursor()
    cur.execute(query)
    res = cur.fetchall()
    print(title, res[0][0])

# Show basic SO cographs
def Test_5():
    print("==============================")
    print(" Smallest SO cographs:")
    print("==============================")
    queryHead = "SELECT cotree " \
                "FROM gr WHERE "
    queries = [ "ov=1 AND no=1 ",
                "ov=1 AND no=0 ",
                "ov=0 AND no=1 ",
                "so=1 AND ov=0 AND no=0 " ]
    titles = [ "SO, O, NO",
               "SO, O",
               "SO, NO",
               "SO" ]
    for i in range(len(queries)):
        smallQuery("A cograph %-9s: cotree =" % titles[i],
                   queryHead + queries[i] + "LIMIT 1")

Test_1()
Test_2()
Test_3()
Test_4()
Test_5()
```

C.3 Arquivo Makefile

```
#
#    cograph - Cograph Generation and Chromatic Analysis
#    Copyright (C) 2014 Alex Reimann Cunha Lima
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
PYTHON=/usr/bin/env python3
SQLITE=sqlite3
PROGRAM=cograph.py
TEST=test.py
DBASE=cograph.db

.PHONY: all clean run build test

all: build

clean:
    @rm -f $(DBASE)

build: $(DBASE)

run: $(DBASE)
    @$(SQLITE) $^

$(DBASE): $(PROGRAM)
    @$(PYTHON) $^

test: $(DBASE)
    @$(PYTHON) $(TEST)
```