UNIVERSIDADE FEDERAL FLUMINENSE

RODRIGO LAMBLET MAFORT

Propagação em Redes: Dominação Vetorial e Seleção de Alvos

NITERÓI

2020

UNIVERSIDADE FEDERAL FLUMINENSE

RODRIGO LAMBLET MAFORT

Propagação em Redes: Dominação Vetorial e Seleção de Alvos

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização

Orientador:
Fábio Protti

NITERÓI

2020

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

M187p Mafort, Rodrigo Lamblet
Propagação em Redes: Dominação Vetorial e Seleção de Alvos / Rodrigo Lamblet Mafort ; Fábio Protti, orientador. Niterói, 2020.
205 f. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2020.

DOI: http://dx.doi.org/10.22409/PGC.2020.d.10673549720

1. Algoritmos em Grafos. 2. Complexidade Computacional. 3. Teoria dos Grafos. 4. Metaheurísticas. 5. Produção intelectual. I. Protti, Fábio, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecário responsável: Sandra Lopes Coelho - CRB7/3389

RODRIGO LAMBLET MAFORT

Propagação em Redes:
Dominação Vetorial e Seleção de Alvos

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Doutor em Computação. Área de concentração: Algoritmos e Otimização

Aprovada em Novembro de 2020.

BANCA EXAMINADORA

Prof. Fábio Protti - Orientador, UFF

Profª. Lilian Markenzon, UFRJ

Prof. Ricardo Cordeiro Côrrea, UFRRJ

Profª. Isabel Cristina Mello Rosseti, UFF

Prof. Uéverton dos Santos Souza, UFF

Prof. Bruno Lopes Vieira, UFF

Niterói

2020

Aos meus pais, Antonio Jose e Rita de Cassia.

Ao querido tio Neilton (in memoriam).

Agradecimentos

À minha família, em especial aos meus pais Antonio Jose e Rita de Cassia, por acreditarem nesse sonho e pelo apoio irrestrito ao longo destes anos.

Ao meu orientador, o prof. Fábio Protti, pela motivação e pelo trabalho incansável. Sua atenção, seus ensinamentos e ponderações foram fundamentais ao longo de todo o percurso.

Ao Instituto de Computação da Universidade Federal Fluminense, seus professores e funcionários pela dedicação e presteza.

À Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro que possibilitou este trabalho.

Finalmente, à Deus por colocar todas essas pessoas em minha vida, pois nada disso seria possível sem elas.

Resumo

Este trabalho aborda dois problemas de propagação em redes, onde a principal ideia é identificar um conjunto de vértices capaz de dominar ou contaminar/influenciar os demais ao longo de sucessivas iterações. Nos modelos de propagação, um vértice é dito contaminado quando o número de seus vizinhos já contaminados supera um valor previamente definido, conhecido como o requisito do vértice. Os problemas de propagação em redes têm várias aplicações práticas, como, por exemplo, modelar computacionalmente a disseminação de informações ou influência em redes sociais, ou a propagação de uma epidemia em uma sociedade. Um exemplo bastante atual de uma aplicação do problema é a busca pelo menor número de usuários da rede social que devem emitir inicialmente uma informação para que ela se dissemine por toda a rede. O primeiro problema considerado neste trabalho é o Problema da Dominação Vetorial, onde a dominação de todos os vértices deve ser realizada em uma única iteração. Isto é, o problema busca um conjunto mínimo de vértices tal que todo vértice ou está contido neste conjunto, ou é imediatamente contaminado pelos vértices do conjunto. Para este problema, o foco do trabalho é identificar os limiares da complexidade computacional. Entretanto, dado o vasto universo de famílias de grafos, o escopo do trabalho foi delimitado aos grafos cordais e suas subclasses. Este estudo produziu um algoritmo linear para os grafos *split*-indiferença, entre outros resultados. O segundo problema abordado é o Problema da Seleção de Alvos, que pode ser visto como uma generalização do anterior, onde o contágio do grafo ocorre ao longo de várias iterações. Ou seja, um vértice contaminado em uma iteração colabora para que seus vizinhos também sejam contaminados nas iterações seguintes. Por ser um tema de grande interesse prático, o problema foi tratado através de algoritmos aproximativos. Um dos resultados obtidos neste trabalho é um algoritmo bioinspirado paralelizado, capaz de identificar soluções melhores do que as previamente conhecidas na literatura para grandes redes sociais.

Palavras-chave: Propagação em Redes. Problema da Dominação Vetorial. Problema da Seleção de Alvos. Complexidade Computacional. Algoritmos em Grafos.

Abstract

This work deals with two propagation problems in networks, where the main idea is to identify a set of vertices capable of dominating or contaminating/influencing the remaining ones over successive iterations. In the propagation models, a vertex is said to be contaminated when the number of its neighbors already dominated is greater than or equal to a previously defined threshold, called the vertex's requirement. Propagation problems in networks have several practical applications, such as computationally modeling the diffusion of information or influence in social networks, or the dissemination of epidemics in a society. An example of application in propagation problems is the search of the smallest number of users of the social media that must broadcast an information to ensure that it reaches the entire network. The first problem considered in this work is the Vector Domination Problem, in which the domination of all vertices must be accomplished in just one iteration. That is, the problem searches for a minimum set of vertices such that every vertex is contained in this set or is immediately contaminated by it. For this problem, the focus of this work is to identify the frontiers of the computational complexity. However, considering the broad universe of graph classes, the focus is restricted to chordal graphs and its subclasses. This study produced a linear time algorithm for split-indifference graphs, among other results. The second problem addressed is the Target Set Selection Problem, which may be seen as a generalization of the previous one, where the domination of the entire graph may take several iterations. That is, a contaminated vertex in one iteration collaborates in the domination of its neighbors in subsequent iterations. Since this problem presents a practical interest, it was approached via approximation algorithms. One of the results obtained in this work is a parallelized genetic algorithm capable of identifying better solutions than the previously known for large social networks.

Keywords: Propagation in Networks. Vector Domination Problem. Target Set Selection Problem. Computational Complexity. Algorithms in Graphs.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

Este trabalho abordará dois problemas de propagação em redes. O objetivo desta classe de problemas é identificar um conjunto de vértices capaz de dominar ou contaminar os demais após um determinado número de iterações. A ideia de contaminar vértices vem da atribuição de dois rótulos (ou estados): contaminado (ou dominado) e não-contaminado (ou não-dominado). Os vértices contaminados são capazes de propagar seu estado, contaminando os demais. Um vértice muda de rótulo ao satisfazer uma condição atrelada a cada problema em específico. O estudo dos problemas de propagação visa avaliar como a dominação se propaga pela rede, o número mínimo de vértices contaminados que são necessários para que toda a rede seja eventualmente dominada, e o impacto das regras dos problemas nesse processo.

Nos problemas estudados neste trabalho, um vértice passa do estado não-dominado para o estado dominado se ele possuir um número pré-determinado de vizinhos já marcados como dominados. Neste modelo baseado em vizinhança, o número de vizinhos contaminados necessário para que um vértice também se contamine é chamado de requisito do vértice. Considerando que a contaminação deve ser viável para todo vértice, assume-se que o requisito de cada vértice será menor ou igual do que o seu número de vizinhos.

A Figura 1.1 apresenta um caso da propagação baseada em vizinhança. O vértice a foi inicialmente marcado como contaminado. Na iteração seguinte, a propaga seu estado para os vértices b, c, d, pois seus requisitos foram satisfeitos. Na terceira iteração, os vértices e, f e g também são atingidos pela propagação iniciada por a. Um conjunto capaz de contaminar todo o grafo é chamado de conjunto de propagação. Na figura, o conjunto $\{a\}$ é um conjunto de propagação ótimo, pois sua cardinalidade é mínima. Conjuntos de propagação são conhecidos na literatura como *conversion sets*, *hull sets* ou *target sets*.

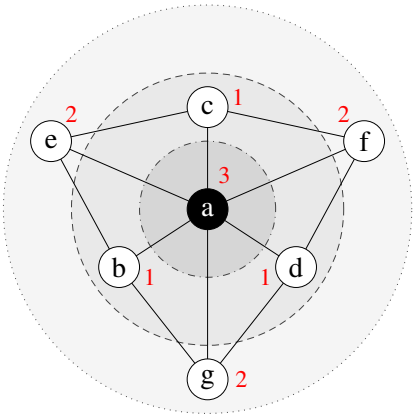


Figura 1.1: Exemplo de um problema de propagação baseado em vizinhança.

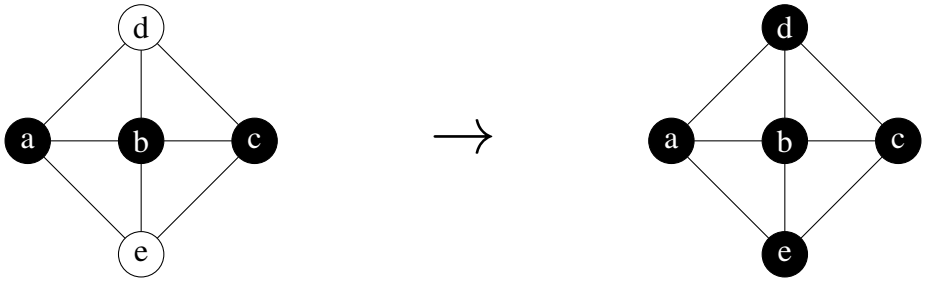
Existem outras regras para propagação que não são baseadas na vizinhança dos vértices. Um exemplo é o estudo da propagação baseada em caminhos de vértices. Neste tópico, um dos problemas mais conhecidos envolve a convexidade P_3, cuja regra de dominação pode ser definida da seguinte forma: um vértice v_2 passa a ser contaminado se estiver em um caminho (v_1, v_2, v_3) tal que v_1 e v_3 estão marcados como contaminados. É interessante notar que a convexidade P_3 também pode ser definida através da dominação por vizinhança, onde um vértice é considerado dominado se dois de seus vizinhos também estiverem dominados.

Vale destacar também que o processo de dominação pode ocorrer ao longo de diversas iterações, isto é, dado o conjunto inicial S, os novos vértices dominados por S também passam a influenciar os demais. Nos modelos apresentados até o momento, nota-se que é impossível que um vértice dominado retorne ao rótulo não-dominado, implicando que o processo da dominação é finito. Entretanto, existem problemas que fogem a essa regra, ao definir condições especiais para que vértices deixem de ser dominados. Por exemplo, ao definir que um vértice v é contaminado e se mantém neste estado enquanto possuir dois ou três vizinhos contaminados, o problema se torna reversível e possivelmente preso em um ciclo de repetição de configurações. É interessante observar que esta regra é a base do jogo da vida (*Conway's Game of Life*), apresentado por Gardner [44]. O jogo da vida é composto por quatro instruções bastante simples: Um organismo vivo morre de solidão se possuir menos do que dois vizinhos vivos; Um organismo vivo com dois ou três vizinhos vivos se mantém vivo na próxima geração; Um organismo vivo com mais de três vizinhos vivos morre por superpopulação; Um espaço vazio no tabuleiro será ocupado por um novo organismo vivo se este espaço possuir exatamente três vizinhos vivos (reprodução). Como pode ser verificado, a regra da dominação proposta contempla as quatro instruções. A

literatura do jogo da vida demonstra que algumas configurações iniciais geram processos que se estabilizam em uma repetição de configurações, o que permite concluir que o mesmo pode ocorrer com os problemas de propagação reversíveis. Os trabalhos de Dourado *et al.* [36] e Dourado *et al.* [34] podem ser apontados como referências no estudo de problemas reversíveis. Já sobre problemas irreversíveis, destacam-se os trabalhos de Centeno *et al.* [13] e Dreyer e Roberts [38].

A Figura 1.2 apresenta a comparação de dois problemas de propagação, tais que o contágio do primeiro exemplo é irreversível, enquanto o do segundo, reversível. Os vértices hachurados em preto estão contaminados e para ambos os casos, o conjunto inicial é $S = \{a, b, c\}$. Vale observar que no segundo exemplo, o conjunto de vértices dominados retornou ao ponto inicial após duas iterações, demonstrando que o processo é cíclico.

Irreversível: Um vértice v está contaminado quando $v \in S \ \vee \ |N(v) \cap S| \geq 2$



Reversível: Um vértice v está contaminado quando $2 \leq |N(v) \cap S| \leq 3$

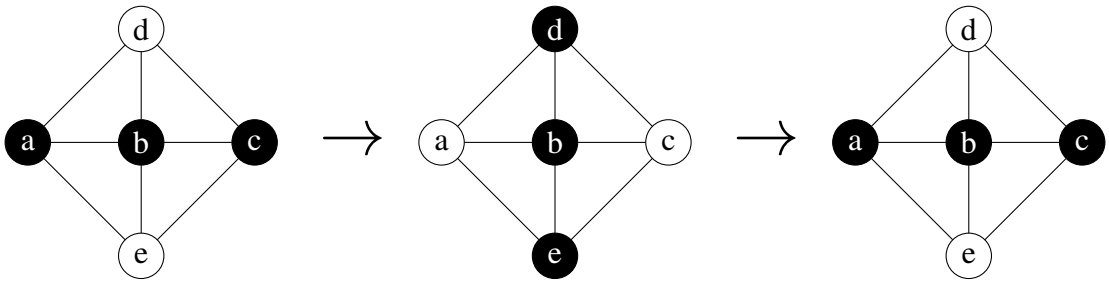


Figura 1.2: Comparação entre Dominação Irreversível e Reversível.

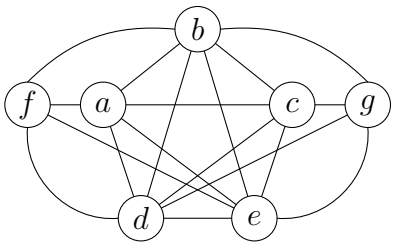
Os problemas de propagação que serão estudados nesta tese são o PROBLEMA DA SELEÇÃO DE ALVOS e o PROBLEMA DA DOMINAÇÃO VETORIAL. Ambos os problemas são irreversíveis, finitos e baseados no modelo de dominação por requisito de vizinhança. Entretanto, eles diferem ao apresentarem restrições diferentes a respeito do número máximo de iterações em que a dominação de todo o grafo deve ocorrer.

O PROBLEMA DA SELEÇÃO DE ALVOS (do inglês, *Target Set Selection Problem*), em sua versão de decisão, consiste em, dados um grafo $G = (V, E)$, um vetor de inteiros posi-

tivos $R = (R[v] \in \{0, \ldots, d(v)\} : v \in V)$, onde $d(v)$ equivale ao grau do vértice v[1], e um número inteiro k, determinar se existe um conjunto $S \subseteq V$ de cardinalidade k tal que todo vértice $v \in V$ está condido em S_n, onde $S_0 = S$ e $S_{i+1} = S_i \cup \{w \in V \mid |N(w) \cap S_i| \geq R[w]\}$. Esta regra estabelece uma sequência de ativações (ou dominações) na qual os vértices dominados em iterações anteriores colaboram para a dominação de seus vizinhos. A forma com que um conjunto influencia o próximo estabelece que o processo é irreversível, isto é, um vértice dominado continuará dominado ao longo de todo processo. Nota-se também que a contaminação do grafo estabilizará em alguma iteração $i \leq n$ (supondo que em cada iteração um novo vértice é dominado, todos os vértices estarão dominados após n iterações). A Figura 1.3 ilustra um exemplo do problema da seleção de alvos, onde a dominação do grafo ocorre após três iterações dado o conjunto inicial $S = \{d\}$: $S_1 = \{d, b\}$, $S_2 = \{d, b, e\}$ e $S_n = S_3 = \{d, b, e, a, c, f, g\}$.

Entradas:

Grafo $G = (V, E)$:



Vetor de Requisitos R:

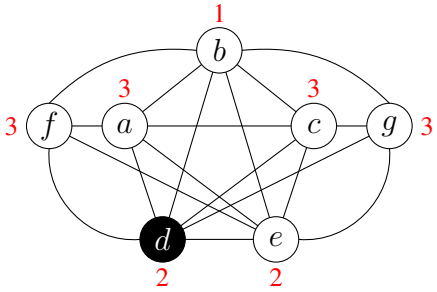
a	b	c	d	e	f	g
3	1	3	2	2	3	3

Inteiro k: 3

Saída:

Sim, existe um conjunto S de cardinalidade $k = 1$ capaz de R-dominar G.

Certificado:



$S = \{d\}$

Figura 1.3: Exemplo de uma instância do problema da seleção de alvos e uma solução.

O problema da seleção de alvos também é conhecido na literatura como *Irreversible Conversion Set Problem* e dentre suas referências destacam-se os algoritmos propostos por Centeno *et al.* [13] e por Dreyer e Roberts [38]. Além disso, os trabalhos de Dourado *et al.* [35] e Dourado *et al.* [37] são referências do estudo do problema sob a ótica da convexidade de conjuntos.

O problema da seleção de alvos é frequentemente utilizado para modelar computaci-

[1]Os conceitos de Teoria dos Grafos utilizados neste trabalho podem ser encontrados no Capítulo 2

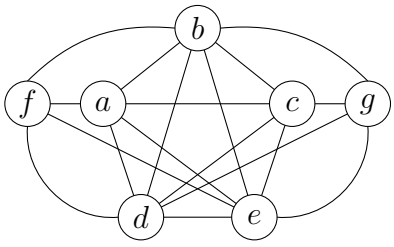
onalmente a propagação de doenças, influência e opiniões (Kempe *et al.* [57] e Friedkin [42]). Recentemente, alguns trabalhos abordaram este problema em instâncias baseadas em redes sociais (Kempe *et al.* [57], Dreyer e Roberts [38], Chen [16], Cicalese *et al.* [18] e Cordasco *et al.* [20, 21]). Um problema interessante que pode ser destacado é a propagação de *fake news* em redes sociais, que pode ser visto como o conjunto de usuários que devem ser inicialmente contaminados com uma notícia falsa ou uma opinião de modo que que ela seja propagada por toda a rede social.

Outro problema interessante neste universo é o problema da vacinação (Richardson e Domingos [75]), que busca identificar o conjunto mínimo capaz de impedir a propagação da dominação e o contágio de todo o grafo. Estes conjuntos são chamados de vacinas pois são capazes de interromper ou inibir a propagação de doenças e influência pela rede.

Além do problema da seleção de alvos, este trabalho abordará também o PROBLEMA DA DOMINAÇÃO VETORIAL (do inglês, *Vector Domination Problem*). Este problema pode ser visto como uma restrição do primeiro na qual todos os vértices devem ser dominados em uma única iteração. Isto é, determinar se existe um conjunto $S \subseteq V$ de cardinalidade k tal que todo vértice v do grafo ou está contido em S, ou é imediatamente contaminado por seus vizinhos em S ($|N(v) \cap S| \geq R[v]$). A Figura 1.4 ilustra um exemplo do problema.

Entradas:

Grafo $G = (V, E)$:



Vetor de Requisitos R:

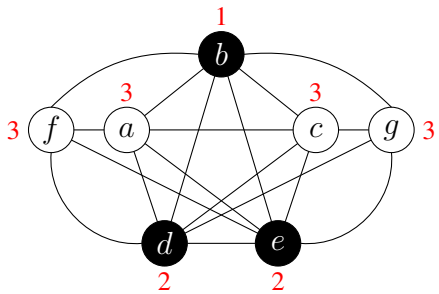
a	b	c	d	e	f	g
3	1	3	2	2	3	3

Inteiro k: 3

Saída:

Sim, existe um conjunto S de cardinalidade $k = 3$ capaz de R-dominar G.

Certificado:



$S = \{b, d, e\}$

Figura 1.4: Exemplo de uma instância do problema da dominação vetorial e uma solução.

É interessante notar nas Figuras 1.3 e 1.4 a diferença entre os resultados obtidos pelo problema da seleção de alvos e o problema da dominação vetorial para a mesma combina-

ção de grafo e vetor de requisitos. Ao considerar as definições dos problemas, é possível observar que uma solução do problema da dominação vetorial para uma instância constitui um limite superior para o problema da seleção de alvos para esta mesma instância.

O problema da dominação vetorial apresenta algumas variantes obtidas pela restrição dos requisitos ou mesmo por uma mudança na regra da dominação. Uma das variações mais conhecidas é o PROBLEMA DO CONJUNTO DOMINANTE, onde todos os vértices possuem requisito unitário, isto é, são dominados quando pelo menos um de seus vizinhos está contido no conjunto dominante. Para grafos em geral, este problema é reconhecidamente $\mathcal{NP}$-Completo [45]. Ao longo deste trabalho, serão apresentadas algumas demonstrações e referências sobre a complexidade computacional do problema do conjunto dominante para algumas classes de grafos.

Quando os requisitos são iguais a uma constante k, o problema da dominação vetorial é denominado k-DOMINAÇÃO. Há ainda uma terceira variação em relação aos requisitos, onde, para todo vértice v do grafo, $R[v] = \lceil d(v)/2 \rceil$. Esta variação é conhecida como PROBLEMA DO MONOPÓLIO [ou *Monopoly* — 58], sendo particularmente interessante para modelar computacionalmente problemas de votação ou consenso, uma vez que cada vértice é dominado quando a maioria de seus vizinhos está contida no conjunto R-dominante.

Existem também variações obtidas por alterações nas regras da dominação. A primeira variação deste tipo é denominada DOMINAÇÃO VETORIAL TOTAL (ou *Total Vector Domination*) e estabelece que todos os vértices do grafo, mesmo os contidos no conjunto dominante, devem ser dominados por seus vizinhos ($\forall v \in V,\ |N(v) \cap S| \geq R[v]$).

Tanto o problema da seleção de alvos quanto o problema da dominação vetorial podem ser vistos como restrições de um problema mais amplo, conhecido como PROBLEMA DA SELEÇÃO DE ALVOS COM LATÊNCIA LIMITADA (do inglês, *Latency-bounded Target Set Selection*) ou (λ, β, α)-SELEÇÃO DE ALVOS, que pode ser definido como o seguinte:

> **Problema da Seleção de Alvos com Latência Limitada:**
> **Instância:** Um grafo $G = (V, E)$, vetor de requisitos $R = (R[v] \in \{0, \ldots, d(v)\} : v \in V)$, um limite para a latência $\lambda \in \mathbb{N}$, um custo máximo $\beta \in \mathbb{N}$ e um número mínimo de vértices dominados $\alpha \in \mathbb{N}$.
> **Problema:** Determinar se existe um conjunto $S \subseteq V$, sujeito à $|S| \leq \beta$ e $|S_\lambda| \geq \alpha$, onde S_λ corresponde ao conjunto S após λ iterações.

Nota-se que o problema da seleção de alvos com latência limitada generaliza os pro-

blemas da seleção de alvos e da dominação vetorial ao considerar a possibilidade de estabelecer um número máximo de iterações λ, um número mínimo de vértices dominados α e o tamanho máximo da solução β.

Sendo assim, pode-se interpretar o problema da seleção de alvos como uma restrição do problema da (λ, β, α)-seleção de alvos onde $\alpha = n$, $\beta = k$ e $\lambda = n$, implicando que todos os vértices devem ser dominados em no máximo n iterações. Por outro lado, o problema da dominação vetorial, também pode ser definido como uma restrição onde $\alpha = n$, $\beta = k$ e $\lambda = 1$, estabelecendo que todos os vértices devem ser dominados em uma única iteração.

Existem na literatura diversos trabalhos sobre a complexidade computacional do problema da seleção de alvos com latência limitada. Chen [16] demonstra que não existe um algoritmo aproximativo para determinar um conjunto que domine todo o grafo (ou uma parte dele) após n iterações melhor do que $O(2^{log^{1-\epsilon}n})$, para qualquer constante $\epsilon > 0$. Além disso, o trabalho de Ben-Zwi *et al.* [2] prova que que o problema da (n, β, α)-seleção de alvos pode ser resolvido em tempo $O(t^w n)$, onde t é o requisito máximo dentre os vértices do grafo e w é o valor do *treewidth* do grafo, provando que o problema é tratável por parâmetro fixo se parametrizado por t e w.

Além destes trabalhos, um outro deve ser destacado. O artigo de Cicalese *et al.* [18] descreve um algoritmo de tempo polinomial para o problema da seleção de alvos com latência limitada em grafos com *clique-width* limitada. O algoritmo proposto tem tempo da ordem de $O(\lambda k|\sigma|(n+1)^{(3\lambda+2)k})$, onde k e σ correspondem respectivamente ao parâmetro *clique-width* do grafo e a uma k-expressão para o grafo. Desta forma, para valores fixos de k e λ, o algoritmo apresenta tempo polinomial no tamanho da entrada. Os conceitos de *clique-width* e k-expressão serão introduzidos no Capítulo 2.

Existem na literatura trabalhos que analisam os problemas de dominação quando são restritos a famílias de grafos em específico. Neste tópico, destacam-se os trabalhos de Farber [40], Farber [41], Haynes *et al.* [52], Chang [14] e Cicalese *et al.* [19].

Para melhor compreensão da extensão dos problemas de propagação, a Figura 1.5 ilustra algumas variações de acordo com suas características. Vale observar que o problema do monopólio foi incluído na categoria "requisitos variáveis", pois o requisito de cada vértice depende de seu grau, que constitui uma entrada do problema.

Tendo em vista que o termo "conjunto dominante" apresenta um significado bastante

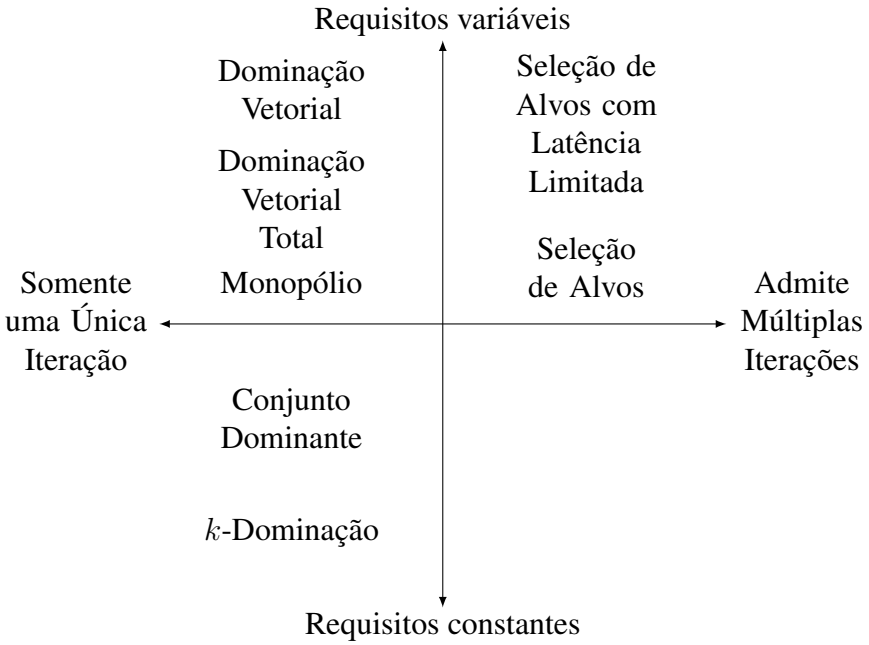


Figura 1.5: Variações de Problemas de Propagação.

atrelado a um problema em específico, seu uso neste texto pode induzir alguma confusão em relação aos problemas cujas instâncias são formadas por um grafo e um vetor de requisitos R. Desta forma, para referenciar conjuntos capazes de dominar todo o grafo em uma única iteração, respeitando o vetor de requisitos R, a expressão "conjunto R-dominante" será preferida. Quando um conjunto R-dominante possuir a menor cardinalidade possível, este será denominado conjunto R-dominante mínimo ou uma solução ótima.

1.1 Objetivos

Este trabalho aborda os problemas de dominação em duas frentes: A primeira tem como principal objetivo determinar as fronteiras da $\mathcal{NP}$-Completude do problema da dominação vetorial. Entretanto, dada a vasta amplitude do problema, o escopo foi delimitado à família dos grafos cordais, isto é, o foco desta frente é demonstrar para quais subfamílias de grafos cordais o problema permanece $\mathcal{NP}$-Completo ou passa a admitir um algoritmo de tempo polinomial. O estudo do problema da dominação vetorial em grafos cordais busca contribuir com os resultados da literatura, definindo novas fronteiras da intratabilidade para algumas classes de grafos que permanecem em aberto na literatura e apresentar novos algoritmos para algumas subclasses de grafos cordais.

A segunda frente aborda o problema da seleção de alvos. Considerando que este é

um problema de grande interesse prático, dado o forte apelo atrelado as redes sociais e a propagação de doenças, opiniões e influência, optou-se por uma abordagem experimental, baseada em algoritmos aproximativos. Desta forma, espera-se obter um algoritmo capaz de lidar com grandes redes sociais, estudando formas de minimizar o conjunto R-dominante. Este estudo envolve o uso de computação paralela, permitindo lidar com grandes redes em tempo hábil, além do uso de algoritmos bioinspirados.

1.2 Estrutura do Trabalho

O Capítulo 2 apresenta os conceitos de teoria dos grafos utilizados neste trabalho. Este capítulo define também os conceitos de *clique-width*, k-expressão e de problemas expressáveis em $MSOL$. Tendo em vista que um dos objetivos do trabalho é explorar a complexidade computacional do problema da dominação vetorial na família dos grafos cordais, o Capítulo 2 apresenta também as classes abordadas, suas definições e algumas ferramentas estruturais exploradas nos Capítulos 3 e 4.

Após a apresentação dos conceitos necessários, é possível prosseguir para o estudo do problema da dominação vetorial. O Capítulo 3 apresenta o estado da arte da complexidade computacional do problema da dominação vetorial na família dos grafos cordais e também em grafos com *clique-width* limitada.

Já o Capítulo 4 apresenta novos algoritmos para o problema da dominação vetorial desenvolvidos ao longo deste trabalho, com destaque para os algoritmos de tempo linear para grafos *split*-indiferença e grafos com exatamente duas cliques maximais.

Em seguida, no Capítulo 5, a complexidade computacional do problema da seleção de alvos será abordada. Este capítulo apresenta também um algoritmo bioinspirado paralelizado para o problema da seleção de alvos, além dos resultados de sua aplicação em grafos que modelam redes sociais de variadas dimensões.

Finalmente, no Capítulo 6, um apanhado dos resultados obtidos na tese é apresentado, incluindo também as publicações decorrentes da pesquisa. Este capítulo também expõe alguns problemas em aberto que foram identificados, além de propor temas para estudos futuros.

Capítulo 2

Definições Introdutórias

Este capítulo apresentará algumas definições básicas de Teoria dos Grafos e algumas ferramentas estruturais necessárias para o estudo do problema da dominação vetorial e do problema da seleção de alvos. Este capítulo detalhará também as definições e caracterizações das famílias de grafos cordais que serão abordadas nos Capítulos 3 e 4.

2.1 Definições importantes

Um grafo $G = (V, E)$ é definido por dois conjuntos V e E, onde o primeiro corresponde aos vértices, e o segundo, às arestas. Cada aresta $e \in E$ do grafo é representada por um par não ordenado $\{u, v\}$, onde $u, v \in V$, e indica que existe um relacionamento entre estes vértices no grafo. Neste caso, os vértices u e v são os extremos de e e, por possuírem esse relacionamento no grafo, tais vértices são ditos adjacentes ou vizinhos. Outra notação importante é que a aresta e é dita incidente aos vértices u e v, uma vez que estes correspondem aos seus extremos. Existem duas medidas em relação ao tamanho de um grafo $G = (V, E)$: n e m, que correspondem, respectivamente, ao número de vértices e arestas do grafo ($n = |V|$ e $m = |E|$).

Este trabalho abordará grafos simples e não orientados. Um grafo é dito simples se ele não contém multiarestas ou laços. Uma multiaresta corresponde a duas ou mais arestas com os mesmos extremos. Já um laço retrata uma aresta cujos extremos são iguais, isto é, incide duas vezes no mesmo vértice. Um grafo é chamado de não orientado quando a existência de uma aresta e entre dois vértices u e v do grafo implica uma relação bilateral entre eles, isto é, u é vizinho de v, assim como v é vizinho de u.

O conjunto dos vizinhos ou vizinhança aberta de um vértice v, $N(v)$, corresponde a todos os vértices u de G tais que $\{v, u\} \in E$[1], isto é, $N(v) = \{u \in V \mid \{v, u\} \in E\}$. A vizinhança fechada de um vértice v, $N[v]$, equivale à vizinhança aberta acrescida do próprio vértice, ou seja, $N[v] = N(v) \cup \{v\}$. Dois vértices u e v são chamados de gêmeos falsos se $N(v) = N(u)$ e de gêmeos verdadeiros quando $N[u] = N[v]$. O grau de um vértice v, denominado $d(v)$, equivale ao seu número de vizinhos: $d(v) = |N(v)|$. Um vértice é universal se $N[v] = V$. Finalmente, o grau máximo $\Delta(G)$ de um grafo $G = (V, E)$ corresponde ao maior grau de um vértice $u \in V$: $\Delta(G) = \max_{u \in V(G)} d(u)$.

Um grafo é dito conexo se, e somente se, existir um caminho entre quaisquer dois vértices do grafo. Um caminho corresponde a uma sequência de vértices distintos $(v_1, \ldots, v_k)$, onde $k \geq 2$ em um grafo $G = (V, E)$, tal que existe uma aresta $(v_i, v_{i+1}) \in E, \forall\ 1 \leq i \leq k - 1$. Se existir uma aresta entre o primeiro e o último vértice de um caminho $(v_1, v_2, \ldots, v_k)$, onde $k \geq 3$, então esse caminho é chamado de ciclo.

Uma clique C é um conjunto de vértices tal que para todo par de vértices distintos $u, v \in C$, existe a aresta $\{u, v\} \in E$. Um grafo $G' = (V', E')$ é chamado de subgrafo de um grafo $G = (V, E)$ quando $V' \subseteq V$ e $E' \subseteq E$. Além disso, $G' = (V', E')$ é um subgrafo induzido de $G = (V, E)$ quando G' é subgrafo de G e para toda aresta $e = \{u, v\} \in E$, $e \in E'$ se, e somente se, $u \in V'$ e $v \in V'$.

Existem diversas formas de representar um grafo. A mais utilizada neste trabalho é a representação geométrica do grafo: Os vértices equivalem a pontos distintos no plano, enquanto cada aresta $e \in E$ corresponde a uma linha ligando os vértices incidentes a e. Existem outras representações, como, por exemplo, a matriz de adjacências. Essa representação equivale a uma matriz M de ordem $n \times n$, onde cada linha e coluna são associadas a um vértice. Cada posição $M[i, j]$ da matriz pode conter valores 0 ou 1 de acordo com a seguinte regra:

$$M[i, j] = \begin{cases} 1, & \text{se } \{i, j\} \in E \\ 0, & \text{caso contrário} \end{cases}$$

Como referências na literatura em português de Teoria dos Grafos, destacam-se Szwarcfiter [79], Markenzon e Waga [70], Markenzon e Vernet [67] e Boaventura Netto [5]. Em inglês, destacam-se: Bondy e Murty [6], Golumbic [49], Brandstädt *et al.* [9] e Diestel [29].

[1] As arestas são descritas neste trabalho como pares não ordenados $\{v_1, v_2\}$, uma vez que constituem relações bilateral entre os vértices, reforçando a ideia de que os grafos são não-direcionados.

Quanto aos conceitos relacionados aos algoritmos e à teoria da complexidade computacional, destacam-se na literatura: Garey e Johnson [45], Dasgupta *et al.* [28] e Cormen *et al.* [22].

2.2 *Clique-width*, k-Expressão e Problemas Expressáveis em $MSOL$

O conceito de k-expressão foi apresentado por Courcelle *et al.* [25] e estabelece que qualquer grafo pode ser construído utilizando um conjunto de quatro operações, baseados na aplicação de rótulos aos vértices do grafo. A sequência com que as operações e os rótulos são utilizados para construir um grafo define uma k-expressão para o mesmo, onde k é o número de rótulos utilizados. Essa construção deve ser realizada através das seguintes operações:

$\boldsymbol{L(v)}$ Criação de um novo vértice v de rótulo L;

$\boldsymbol{G \oplus H}$ União disjunta de dois conjuntos (ou grafos) G e H;

$\boldsymbol{\eta_{i,j}(G)}$ Adição de todas as possíveis arestas com um extremo de rótulo i e outro de rótulo j contidos em G;

$\boldsymbol{\rho_{i \to j}(G)}$ Alteração dos rótulos de todos os vértices rotulados com i para o rótulo j.

Deve-se notar que o conjunto de operações não envolve a adição de uma aresta em específico, apenas arestas entre vértices que possuam os rótulos especificados pela operação.

A Figura 2.1 exibe um exemplo apresentado por Courcelle em Courcelle *et al.* [26], ilustrando a construção de um grafo completo com quatro vértices $\{u, v, w, x\}$. Pode-se notar que a construção deste grafo demanda apenas dois rótulos. Nessa figura, a ordem na qual as arestas foram adicionadas ao grafo foram indicadas, assim como os rótulos aplicados durante o processo de construção.

Uma 3-expressão para o grafo apresentado nas Figuras 1.3 e 1.4 pode ser definida como $\eta_{2,3}(\eta_{1,2}(\oplus(\rho_{2\to1}\ (\eta_{1,2}(\oplus(1(f), \rho_{1\to3}(\eta_{1,3}(\eta_{2,3}(\oplus(\rho_{3\to2}\ (\eta_{1,2}\ (\eta_{2,3}(\oplus(\oplus(2(b), 3(a)), 1(g))))), 3(c)))))))), \rho_{1\to2}(\eta_{1,2}(\oplus(2(d), 1(e)))))))$. É interessante observar que uma k-expressão pode ser visualizada como uma árvore onde os nós internos correspondem às operações $\oplus$, η e ρ e as folhas são os vértices do grafo, no momento em que são adicionados utilizando um rótulo em específico. Neste tópico, destaca-se a ferramenta TRAG [39],

Operação	Efeito no grafo
$1(u)$	Criação do vértice u com rótulo 1
$2(v)$	Criação do vértice v com rótulo 2
$\oplus\{u, v\}$	Adição dos vértices u e v no grafo
$\eta_{1,2}(G)$	Adição da aresta $\{u, v\}$
$\rho_{2\to1}(G)$	Alteração dos rótulos dos vértices de 2 para 1
$2(w)$	Criação do vértice w com rótulo 2
$\oplus\{w\}$	Adição do vértice w no grafo
$\eta_{1,2}(G)$	Adição das arestas $\{u, w\}$ e $\{v, w\}$
$\rho_{2\to1}(G)$	Alteração dos rótulos dos vértices de 2 para 1
$2(x)$	Criação do vértice x com rótulo 2
$\oplus\{x\}$	Adição do vértice x no grafo
$\eta_{1,2}(G)$	Adição das arestas $\{u, x\}$, $\{v, x\}$ e $\{w, x\}$
$\rho_{2\to1}(G)$	Alteração dos rótulos dos vértices de 2 para 1

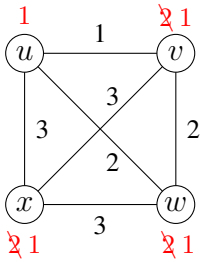


Figura 2.1: Exemplo do uso das operações de construção para obter o grafo.

que permite obter k-expressões ótimas para grafos com até 20 vértices (uma k-expressão ótima é uma k-expressão com o menor valor de k possível).

O parâmetro *clique-width* (cw) de um grafo corresponde ao menor k para o qual existe uma k-expressão. No exemplo da Figura 2.1 nota-se que o grafo foi construído utilizando dois rótulos. Como não é possível construir o mesmo grafo utilizando somente um rótulo, então o grafo apresentado possui $cw = 2$.

Ao observar o exemplo apresentado (um grafo completo), é possível observar que novos vértices podem ser adicionados ao grafo sem a necessidade de novos rótulos. Seja a um novo vértice. A adição de a pode ser realizada pelas seguintes operações: $2(a)$, $\oplus\{a\}$, $\eta_{1,2}(G)$ e $\rho_{2\to1}(G)$). Considerando que novos vértices podem ser incorporados ao grafo sem implicar no uso de novos rótulos, é possível afirmar que grafos completos possuem $cw \leq 2$, isto é, o parâmetro *clique-width* dessa família é limitado por uma constante.

Desta forma, o parâmetro *clique-width* de uma família de grafos corresponde ao menor número de rótulos utilizados para construir qualquer grafo desta família, independentemente do número de vértices e arestas. Neste trabalho serão apresentados outros exemplos de famílias de grafos com *clique-width* limitado, como, por exemplo, os grafos split-indiferença e os grafos ptolemaicos.

Uma k-expressão irredundante é uma k-expressão onde o grafo antes de qualquer operação $\eta_{a,b}$ (adição das arestras entre todos os vértices de rótulo a e todos os vértices de rótulo b) não contém nenhuma aresta entre vértices rotulados como a e vértices rotulados

como b. A Figura 2.2 apresenta um exemplo de uma árvore *clique-width* obtida para uma 3-expressão irredundante para um grafo *split*-indiferença.

Em Teoria dos Grafos, uma expressão $MSOL$ corresponde a uma expressão lógica de segunda ordem com operadores para a quantificação de subconjuntos de vértices, mas não de arestas. Um exemplo de fórmula $MSOL$ foi apresentada em Courcelle *et al.* [26] para o Problema da 3-Coloração de vértices (É possível particionar o conjunto de vértices de um grafo $G = (V, E)$ em três partições X_1, X_2 e X_3 de forma que X_1, X_2 e X_3 sejam conjuntos independentes?).

O Problema da 3-Coloração de vértices é um problema $MSOL$, uma vez que é possível expressá-lo da seguinte forma:

$$\varphi \equiv \exists X_1, X_2, X_3 (Part(X_1, X_2, X_3) \wedge CIndep(X_1) \wedge CIndep(X_2) \wedge CIndep(X_3))$$

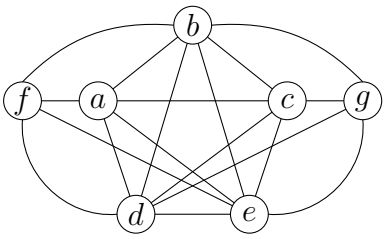
onde:

$$\begin{aligned} Part(X_1, X_2, X_3) \equiv \forall v \ & (X_1(v) \vee X_2(v) \vee X_3(v)) \\ & \wedge \nexists u \ ((X_1(u) \wedge X_2(u)) \\ & \quad \vee (X_1(u) \wedge X_3(u)) \\ & \quad \vee (X_2(u) \wedge X_3(u))) \end{aligned}$$

$$CIndep(X) \equiv \forall u, v (X(u) \wedge X(v) \implies \neg E(u, v))$$

Uma vez que os conceitos necessários para o desenvolvimento do trabalho foram apresentados, é possível prosseguir para as famílias de grafos abordadas nos Capítulos 3 e 4.

Grafo G:



3-expressão:

$\eta_{2,3}(\eta_{1,2}(\oplus(\rho_{2\to1}(\eta_{1,2}(\oplus(1(f), \rho_{1\to3}($
$\eta_{1,3}(\eta_{2,3}(\oplus(\rho_{3\to2}(\eta_{1,2}(\eta_{2,3}(\oplus(\oplus(2(b),$
$3(a)), 1(g))))), 3(c))))))), \rho_{1\to2}($
$\eta_{1,2}(\oplus(2(d), 1(e)))))))$

Árvore *clique-width*:

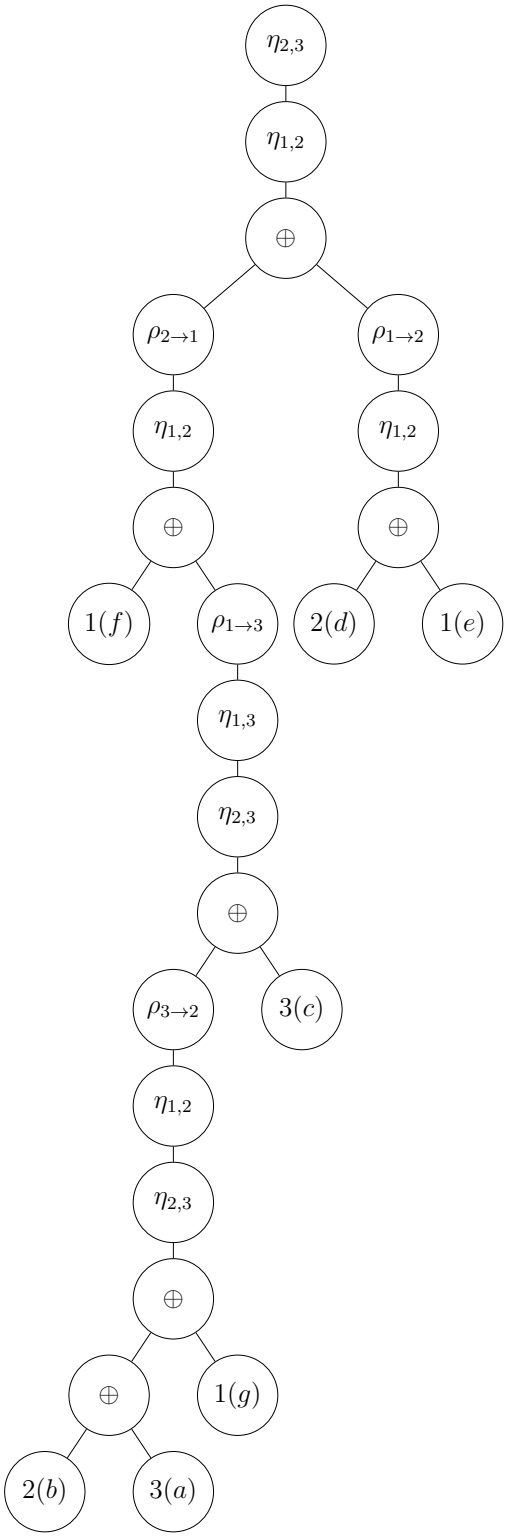


Figura 2.2: Exemplo de Grafo G, k-expressão para G e Árvore *clique-width* T para G.

2.3 A Família dos Grafos Cordais

Esta seção apresentará as caracterizações de algumas famílias de grafos cordais, que serão abordadas nos Capítulos 3 e 4. Como será visto ao longo desta seção, algumas famílias são caracterizadas por restrições de adicionais em relação a outras classes. Um exemplo são os *undirected path graphs* e *directed path graphs*. Na primeira classe, adota-se um modelo de interseções baseados em caminhos não direcionados em uma árvore não direcionada. Já a segunda, ao assumir que a árvore subjacente é direcionada, implementa restrições adicionais que estabelecem essa nova classe.

As relações de continência e interseção entre as classes definem um modelo de hierarquia, de forma que todas as propriedades definidas para um classe são herdadas pelas classes contidas na primeira. É interessante observar que essas relações foram exploradas para definir a complexidade computacional do problema da dominação vetorial no Capítulo 3.

Considerando que um dos objetivos deste trabalho é a exploração da família dos grafos cordais, as Subseções 2.3.1 a 2.3.12 apresentam as definições e ferramentas estruturais de algumas classes de grafos contidas na família dos grafos cordais.

2.3.1 Grafos Cordais

Um grafo não direcionado $G = (V, E)$ é um *grafo cordal* quando todo ciclo de tamanho maior que três possui uma corda. Uma corda corresponde a uma aresta conectando dois vértices não consecutivos de um ciclo. Os grafos cordais também são conhecidos na literatura como grafos triangularizados, devido à inexistência de subgrafos induzidos que correspondam a C_n para $n > 3$ (C_n representa um grafo que é formado por um único ciclo de tamanho n). Como referências na literatura sobre grafos cordais destacam-se os trabalhos de Golumbic [49] e Markenzon e Waga [70].

A Figura 2.3 apresenta um exemplo de um grafo cordal com 6 vértices. A aresta entre os vértices 2 e 4 (destacada como uma linha tracejada) é um exemplo de uma corda.

A inexistência de ciclos com mais de três vértices sem cordas é uma propriedade hereditária por subgrafos induzidos. Isto é, todo subgrafo induzido de um grafo cordal também é cordal. A verificação desta afirmativa decorre da impossibilidade da remoção de um vértice criar ciclos com mais de três vértices sem cordas.

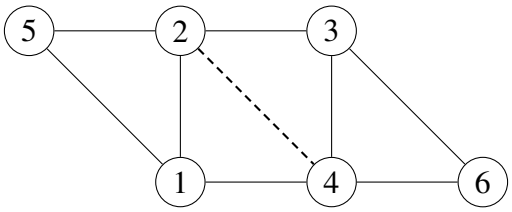


Figura 2.3: Exemplo de grafo cordal.

Um vértice v de um grafo é dito simplicial se todos os seus vizinhos também são vizinhos entre si, isto é, o subgrafo induzido por $N[v]$ corresponde a um grafo completo (grafos completos são detalhados na Subseção 2.3.6). Na Figura 2.3, apenas os vértices 5 e 6 são simpliciais. Os demais possuem, dentre os seus vizinhos, dois vértices não adjacentes.

Dirac [33] demonstrou que todo grafo cordal possui pelo menos dois vértices simpliciais. Além disso, se o grafo não for completo, ele deve possuir dois vértices simpliciais que não são adjacentes entre si. Fulkerson e Gross [43] apresentaram um método iterativo para reconhecer se um grafo é cordal ou não baseado na identificação de vértices simpliciais e na propriedade hereditária dos grafos cordais. O método proposto consiste em identificar e remover vértices simpliciais do grafo iterativamente até que o conjunto de vértices se torne vazio, implicando que o grafo é cordal. Entretanto, se em um dado momento o grafo não possuir nenhum vértice simplicial, o subgrafo induzido pelo conjunto de vértices restante não é um grafo cordal, implicando que o grafo inicial também não é, dada a propriedade hereditária desta classe.

A ordem em que os vértices são removidos é denominada um esquema de eliminação perfeita. Uma ordenação $\langle v_1, v_2, \ldots, v_n \rangle$ dos vértices do grafo é um esquema de eliminação perfeita (EEP) se cada vértice v_i da ordenação é simplicial no subgrafo induzido por $\{v_i, v_{i+1}, \ldots, v_n\}$. Um dos possíveis esquemas de eliminação perfeita para o grafo da Figura 2.3 é $\langle\ 5, 6, 3, 1, 2, 4\ \rangle$. Considerando que um esquema de eliminação perfeita atende ao método de reconhecimento proposto por Fulkerson e Gross [43], pode-se concluir que todo grafo cordal admite pelo menos um esquema de eliminação perfeita.

Uma das formas de reconhecer se um grafo é cordal é a obtenção de uma ordenação dos vértices e, em seguida, verificar se a ordenação obtida corresponde a um esquema de eliminação perfeita. Uma das formas de se obter uma ordenação dos vértices compatível com um esquema de eliminação perfeita é o algoritmo de percurso em largura lexicográfica. Este algoritmo difere dos algoritmos de percurso habituais ao se basear na aplicação de um

rótulo a cada vértice. A ordem na qual os vértices serão explorados depende deste rótulo. O Algoritmo 2.1 apresenta o algoritmo de percurso em largura lexicográfica: a linha 6 estabelece a ordenação dos vértices compatível com um EEP. Sempre que um vértice é escolhido, ele é removido do grafo e os rótulos de seus vizinhos são atualizados.

Algoritmo 2.1: Obtenção de EEP para grafos cordais (Rose *et al.* [77])

Entrada: Grafo Cordal $G = (V, E)$
Saída: Esquema de Eliminação Perfeita σ

1 **para cada** $v \in V$ **faça**
2 $L(v) \leftarrow \emptyset$
3 $V' \leftarrow V$; // V' contém os vértices que ainda não entraram no EEP
4 **para** $(i \leftarrow n; \quad i \geq 1; \quad i \leftarrow i - 1)$ **faça**
  /* Seleção: */
5 Selecione um vértice $v \in V'$ de maior rótulo L
6 $\sigma[i] \leftarrow v \qquad V' \leftarrow V' \setminus \{v\}$
  /* Atualização: */
7 **para cada** $w \in N(v) \cap V'$ **faça**
8 Concatene i ao rótulo $L[w]$
9 **retorne** σ

O Algoritmo 2.1 retorna uma ordenação para qualquer grafo. Entretanto, para determinar se a ordenação corresponde a um esquema de eliminação perfeita, implicando que o grafo é cordal, é necessário verificar se cada vértice é simplicial no subgrafo induzido pelos vértices posteriores a ele na sequência, incluindo ele próprio. O Algoritmo 2.2 detalha a verificação de uma ordenação, finalizando o reconhecimento dos grafos cordais.

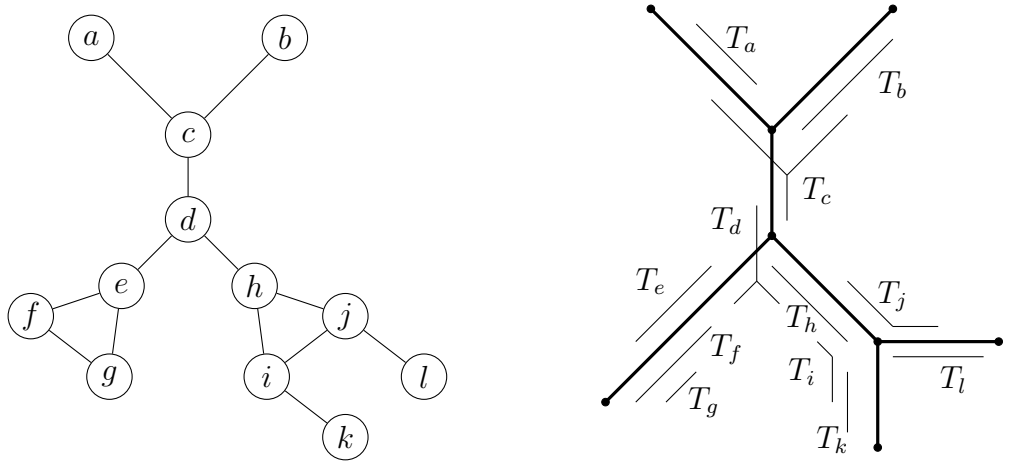
De acordo com Gavril [46], os grafos cordais também podem ser caracterizados como uma família de subárvores de uma árvore não direcionada. A Figura 2.4 ilustra um exemplo da representação de um grafo cordal como uma família de subárvores.

Dado um grafo $G = (V, E)$, um conjunto de vértices $S \subseteq V$ é um separador de vértices se existem dois vértices distintos $v_1, v_2 \in V \setminus S$ tais que existe um caminho entre v_1 e v_2 em G e não existe nenhum caminho entre v_1 e v_2 no subgrafo induzido por $V \setminus S$. Isto é, v_1 e v_2 estão em componentes conexas diferentes em $G[V \setminus S]$. Um conjunto separador S é dito minimal se é capaz de separar o grafo ao mesmo tempo em que não existe nenhum subconjunto próprio $S' \subset S$ capaz de fazer o mesmo. Dirac [33] propôs uma caracterização adicional dos grafos cordais: um grafo é cordal se, e somente se, todo separador minimal de vértices é uma clique. Na Figura 2.4, o vértice d é um separador minimal, assim como os vértices $\{h, i, j\}$.

Algoritmo 2.2: Reconhecimento se um grafo é cordal (Golumbic [49])

Entrada: Grafo $G = (V, E)$ e Ordenação de vértices σ
Saída: G é cordal?

para cada $v \in V$ **faça**
  $A(v) \leftarrow \emptyset$
para $(i \leftarrow i; \quad i < n; \quad i \leftarrow i + 1)$ **faça**
  $v \leftarrow \sigma(i)$
  $X \leftarrow \{x \in N(v) \mid \sigma^{-1}(v) < \sigma^{-1}(x)\}$; `// ` $\sigma^{-1}(v)$ `equivale a posição de` v `na sequência` σ
  se $X \neq \emptyset$ **então**
    $u \leftarrow \sigma(\min(\{\sigma^{-1}(x) \colon x \in X\}))$
    $A(v) \leftarrow A(v) \cup (X \setminus \{u\})$
  se $A(v) \setminus N(v) \neq \emptyset$ **então**
    retorne Falso
retorne Verdadeiro



Grafo cordal G Representação de G como subárvores

Figura 2.4: Exemplo de grafo cordal e sua representação como subárvores [49].

Uma árvore de cliques de um grafo cordal G corresponde a uma árvore T tal que cada vértice de T representa uma clique maximal de G. A construção das arestas de T deve respeitar uma restrição que assegura que, para todo par de cliques distintas K' e K'', o conjunto $K' \cap K''$ esteja contido em todas as cliques situadas no caminho entre K' e K'' em T. É interessante notar também que a árvore de cliques evidencia os separadores minimais do grafo: um conjunto S é um separador minimal de vértices se, e somente se, $S = K' \cap S''$ para uma aresta $\{K', K''\}$ da árvore de cliques do grafo.

A Figura 2.5 ilustra uma árvore de cliques obtida a partir do grafo cordal apresentado na Figura 2.4. A árvore apresentada na figura foi obtida através do algoritmo apresentado

por Markenzon e Waga [70], utilizando como esquema de eliminação perfeita a sequência de vértices $\langle a, b, c, k, l, f, g, e, i, j, h, d\rangle$.

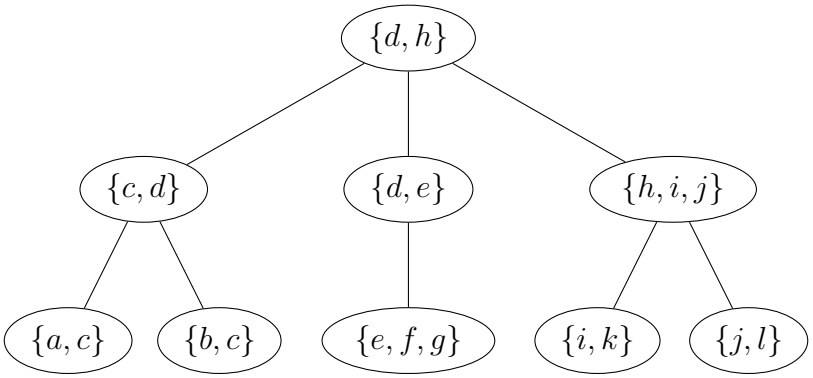


Figura 2.5: Exemplo de Árvore de Cliques.

Os trabalhos de Dirac [33], Gavril [46], Buneman [12] e Spinrad [78] são referências do estudo da árvore de cliques dos grafos cordais. A obtenção da árvore de cliques de um grafo cordal parte de um esquema de eliminação perfeita e pode ser realizada em tempo $O(n + m)$. O Algoritmo 2.3 demonstra como obter uma árvore de cliques para um grafo cordal, partindo de um esquema de eliminação perfeita.

A árvore de cliques é uma importante ferramenta no estudo da dominação vetorial nas classes de grafos, pois pode sugerir um ponto de início para a dominação dos vértices, uma vez que o algoritmo para dominação vetorial em grafos completos é bastante simples e direto, conforme será visto na Subseção 3.2.1.

Uma vez que a classe dos grafos cordais foi definida, é possível apresentar quais subclasses de grafos cordais serão estudadas e as relações de continência entre essas classes. A Figura 2.6 retrata algumas subclasses da hierarquia da família dos grafos cordais. É interessante observar que essas relações serão exploradas para definir a complexidade computacional do problema da dominação vetorial no Capítulo 3.

Uma vez que a hierarquia foi apresentada, é possível prossegui para a definição de cada classe de grafo. A primeira decorre de uma restrição ao modelo de interseção de subárvores, onde não são mais admitidas subárvores, mas apenas caminhos na árvore subjacente.

2.3.2 Undirected Path Graphs

A literatura apresenta diversas famílias de grafos que são definidas por modelos de interseção de caminhos (ou mesmo subárvores) em uma estrutura subjacente. Cada família

Algoritmo 2.3: Obtenção de uma árvore de cliques

Entrada: Grafo cordal $G = (V, E)$ e EEP σ
Saída: Árvore de Cliques $T = (V_T, E_T)$

$q \leftarrow 1$
$Q_1 \leftarrow \{v_n\}$ $\quad \eta(v_n) \leftarrow 1$
$V_T \leftarrow \{Q_1\}$ $\quad E_T \leftarrow \emptyset$
para $(i \leftarrow n-1; \quad i \geq 1; \quad i \leftarrow i-1)$ **faça**
  $X_\sigma(v_i) \leftarrow \{u \in N(v_i) \colon \sigma^{-1}(u) > \sigma^{-1}(v_i)\}$
  $w \leftarrow \sigma(\min(\{\sigma^{-1}(z) \colon z \in X_\sigma(v_i)\})$
  $k \leftarrow \eta(w)$
  se $|X_\sigma(v_i)| < |Q_k|$ **então**
    /* Criar nova clique */
    $q \leftarrow q + 1$
    $Q_q \leftarrow \{v_i\} \cup X_\sigma(v_i)$; /* Nova clique */
    $V_T \leftarrow V_T \cup \{Q_q\}$; /* Adicionar o nó da árvore */
    $E_T \leftarrow E_T \cup \{Q_q, Q_k\}$; /* Adicionar a aresta da árvore */
    $\eta(v_n) \leftarrow q$
  senão
    /* Aumentar a clique Q_k */
    $Q_k \leftarrow \{v_1\} \cup Q_k$
    $\eta(n_i) \leftarrow k$
retorne $T = (V_T, E_T)$

difere das demais por alguma restrição inerente aos caminhos ou mesmo a essa estrutura. Conforme apresentado anteriormente, os grafos cordais podem ser vistos como grafos de interseção de subárvores de uma árvore. Ao restringir ainda mais esse modelo, considerando não subárvores, mas caminhos, uma nova família é caracterizada: *Undirected Path Graphs*. Esta família de grafos é definida por admitir um modelo de interseção de caminhos não direcionados em uma árvore não direcionada e não enraizada (Monma e Wei [72]). A Figura 2.7 ilustra um grafo desta família e um possível modelo de interseção.

A classe dos *undirected path graphs* está contida na família dos grafos cordais. Uma evidência clara dessa relação pode ser vista na Figura 2.8. Nesta figura, a adição da aresta entre os vértices 1 e 4 no grafo implica a necessidade de que o caminho P_4 (correspondente ao vértice 4 no grafo) apresente uma interseção com o caminho P_1. Entretanto, pode-se observar que é impossível que esta interseção exista sem que P_4 intersecte também o caminho P_2 (o que criaria a corda entre os vértice 2 e 4). Desta forma, fica evidente a impossibilidade da existência de ciclos com quatro ou mais vértices sem cordas, demonstrando a relação de continência entre as classes.

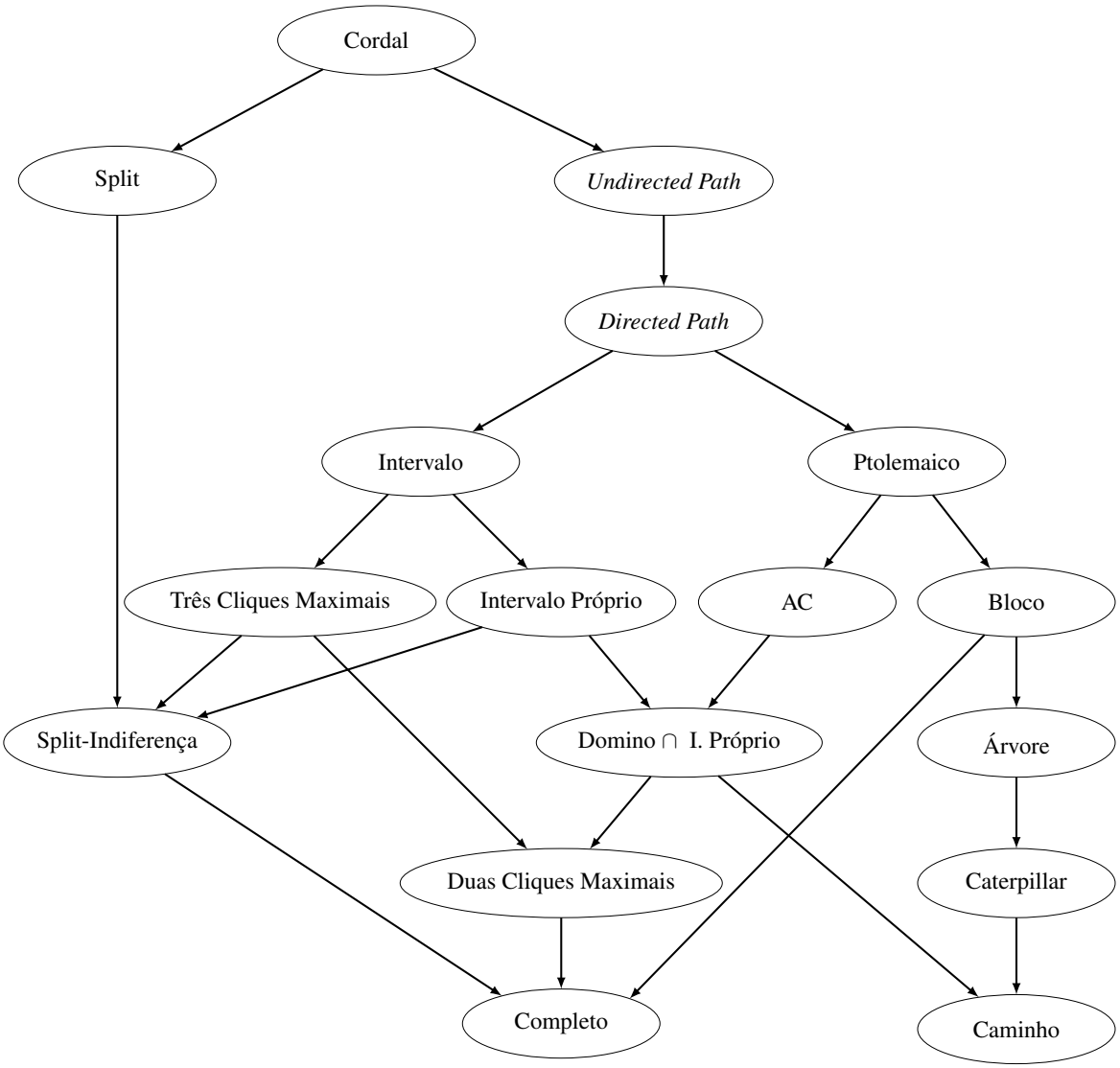
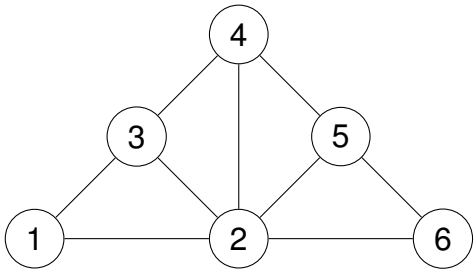


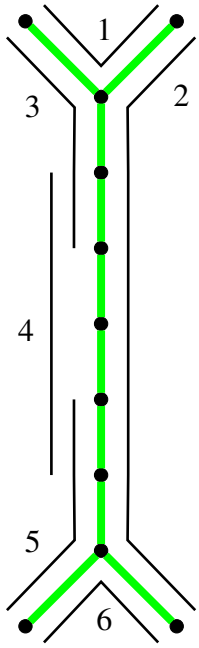
Figura 2.6: Hierarquia de subclasses dos grafos cordais estudados.

2.3.3 Directed Path Graphs

Uma vez que a classe dos *undirected path graphs* foi demonstrada, pode-se considerar uma segunda restrição, na qual a estrutura subjacente deve ser uma árvore enraizada direcionada. Isto é, o grafo deve admitir um modelo de interseção de caminhos direcionados em uma árvore direcionada e enraizada. Esta restrição é baseada na adição de direcionamento nas arestas da árvore subjacente, o que implica que os caminhos também devem ser direcionados. Esta restrição dá origem aos *Directed Path Graphs* (ou Grafos de Interseção de Caminhos Sobre uma Árvore Direcionada Enraizada), objeto de estudo desta seção. Outra questão que deve ser destacada é a continência entre essas classes: os grafos cordais contêm os *undirected path graphs* que, por sua vez, contêm os *directed path graphs*.

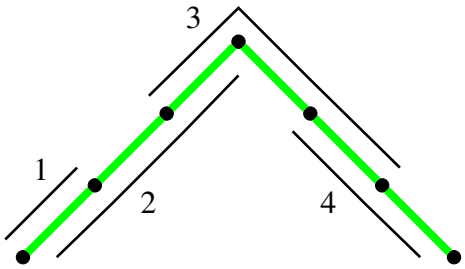


Undirected Path Graph G.

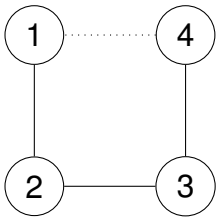


Árvore Característica de G.

Figura 2.7: Exemplo de um *undirected path graph* e sua árvore característica.



Modelo de Interseção para G.



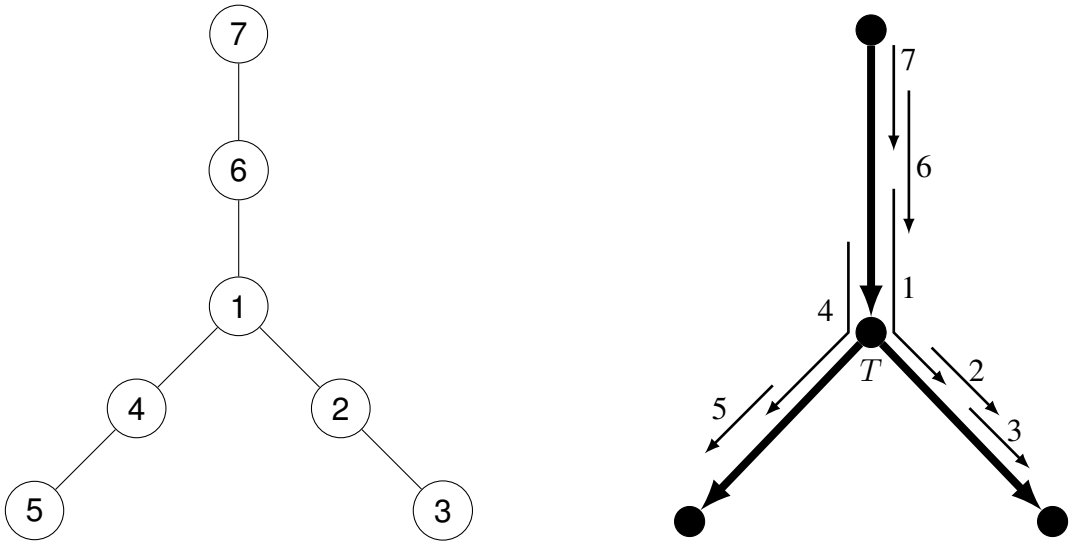
Undirected Path Graph G.

Figura 2.8: Relação entre *Directed Path Graphs* e Grafos Cordais.

Uma definição formal da família dos *directed path graph* foi apresentada por Gavril [47] no Teorema 2.3.1.

Teorema 2.3.1 (Gavril [47])**.** *Sejam $G = (V, E)$ um grafo e C o conjunto das cliques de G. A partir de C, um segundo conjunto pode ser construído: Para todo $v \in V$, seja C_v o conjunto de todas as cliques de G que contém v. Para caracterizar G, uma árvore enraizada direcionada T deve ser construída: Seja $\tilde{C}$ um novo conjunto tal que cada clique $c \in C$ corresponde a um vértice $\tilde{c}$ em $\tilde{C}$. Além disso, para cada vértice $v \in V$, seja $\tilde{C_v} = \{\tilde{c} : c \in C_v\}$. O conjunto $\tilde{C}$ corresponde ao conjunto de vértices de T. O grafo G é um directed path graph se, e somente se, existe uma árvore enraizada direcionada T tal que, para todo vértice $v \in V$, $T(\tilde{C_v})$ (subárvore induzida) corresponde a um caminho direcionado em T. A árvore T é chamada de árvore característica de G.*

A Figura 2.9 apresenta um exemplo de um grafo desta família e sua representação como caminhos direcionados em uma árvore direcionada. Nota-se que dois vértices são adjacentes no grafo se, e somente se, os caminhos associados aos dois vértices possuem uma interseção.



Directed Path Graph G. Árvore Característica de G.

Figura 2.9: Exemplo de um *directed path graph* e sua árvore característica.

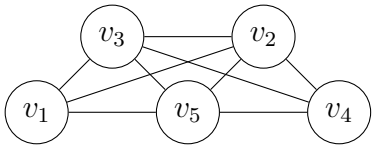
Uma caracterização adicional, também proposta por Gavril [47], evidencia uma interessante característica da árvore de cliques dos *directed path graphs*.

Teorema 2.3.2 (Gavril [47])**.** *Um grafo $G = (V, E)$ é um directed path graph se, e somente se, existe uma árvore direcionada T cujos vértices são as cliques maximais de G, tal que, para todo vértice $x \in V$, o conjunto das cliques maximais que contém x induzem um caminho direcionado em T.*

Existem na literatura diversas referências adicionais sobre os *directed path graphs*, Dentre elas, destacam-se os trabalhos de Dietz [30], Monma e Wei [72] e Chaplick *et al.* [15]. Estes trabalhos apresentam caracterizações adicionais e propõem algoritmos de reconhecimento para esta classe.

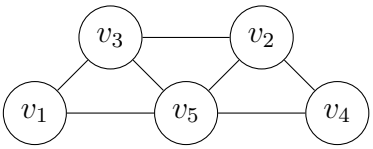
A classe dos *directed path graphs* contém diversas outras classes de grafos, como, por exemplo, os grafos ptolemaicos e os grafos de intervalo. É interessante observar que os *directed path graphs* são uma das últimas classes na hierarquia que contém propriamente estas duas subclasses.

Grafo ptolemaico G



A desigualdade ptolemaica é respeitada para quaisquer quatro vértices.

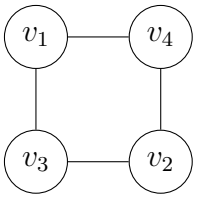
Grafo não-ptolemaico G' (gema)



$$d(v_1, v_2) \cdot d(v_3, v_4) \leq$$
$$d(v_1, v_3) \cdot d(v_2, v_4) + d(v_1, v_4) \cdot d(v_2, v_3)$$
$$2 \cdot 2 \leq 1 \cdot 1 + 2 \cdot 1 \qquad 4 \leq 3$$

A desigualdade ptolemaica foi violada.

Figura 2.10: Exemplo de grafo ptolemaico e não-ptolemaico.



$$d(v_1, v_2) \cdot d(v_3, v_4) \leq d(v_1, v_3) \cdot d(v_2, v_4) + d(v_1, v_4) \cdot d(v_2, v_3)$$
$$2 \cdot 2 \leq 1 \cdot 1 + 1 \cdot 1$$
$$4 \leq 2$$

A desigualdade ptolemaica foi violada.

Figura 2.11: Exemplo da violação da desigualdade ptolemaica em ciclos sem cordas.

2.3.4 Grafos Ptolemaicos

O trabalho de Kay e Chartrand [56] define uma nova subclasse de grafos cordais: os grafos ptolemaicos. Um grafo $G = (V, E)$ conexo é dito ptolemaico se quaisquer quatro vértices $v_1, v_2, v_3, v_4 \in V$ satisfazem a seguinte expressão: $d(v_1, v_2) \cdot d(v_3, v_4) \leq d(v_1, v_3) \cdot d(v_2, v_4) + d(v_1, v_4) \cdot d(v_2, v_3)$, onde $d(u, w)$ equivale a distância entre u e w no grafo. Essa expressão é chamada de desigualdade ptolemaica, explicando a origem do nome da família.

A Figura 2.10 ilustra dois grafos grafos com cinco vértices. O grafo da direita atende a indiferença ptolemaica, e, portanto, é um grafo ptolemaico. Já o da esquerda evidencia uma contradição da indiferença ptolemaica, suficiente para demonstrar que tal grafo não é ptolemaico. O grafo da direita é denominado na literatura como grafo gema.

Em relação aos grafos cordais, a desigualdade ptolemaica também impede que grafos ptolemaicos contenham ciclos com mais de quatro vértices sem cordas. A Figura 2.11 ilustra como a desigualdade ptolemaica é violada nestes casos. Como consequência dessa impossibilidade, pode-se afirmar que todo grafo ptolemaico é também um grafo cordal.

O trabalho de Howorka [54] apresenta duas importantes caracterizações sobre os grafos ptolemaicos.

Teorema 2.3.3 (Howorka [54])**.** *Um grafo G é ptolemaico se G é cordal e não contém um grafo gema como subgrafo induzido.*

Teorema 2.3.4 (Howorka [54])**.** *Um grafo G é ptolemaico se G é cordal e é um grafo de distância hereditária.*

Em relação ao primeiro teorema, é interessante notar que o trabalho de Howorka [54] demonstra que a adição da aresta $\{v_1, v_2\}$ ou da aresta $\{v_3, v_4\}$ no grafo gema ilustrado na Figura 2.10 é suficiente para tornar o grafo ptolemaico. Já o segundo teorema apresenta uma importante caracterização sobre os grafos ptolemaicos, utilizada para o estudo da complexidade computacional do problema da dominação vetorial nesta classe.

Um grafo conexo $G = (V, E)$ é dito de distância hereditária se a distância entre quaisquer dois vértices $v, u \in V$ permanece a mesma em qualquer subgrafo conexo induzido $G' = (V', E')$ de G tal que $v, u \in V'$.

Os grafos de distância hereditária apresentam ainda uma segunda caracterização relacionada a construção do grafo. Um grafo G é de distância hereditária se G pode ser construído a partir de um único vértice isolado utilizando três operações:

$\boldsymbol{P(u, v)}$ Adicionar um vértice pendente u, isto é, adjacente somente um único vértice v do grafo: $N(u) = \{v\}$;

$\boldsymbol{G_V(u, v)}$ Adicionar um vértice gêmeo verdadeiro u de um vértice v do grafo: $N[u] = N[v]$;

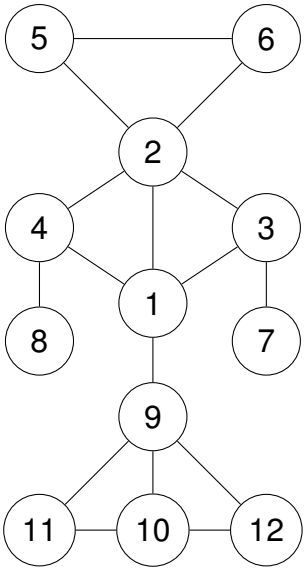
$\boldsymbol{G_F(u, v)}$ Adicionar um vértice gêmeo falso u de um vértice v do grafo: $N(u) = N(v)$.

Deve-se notar que a diferença entre a segunda e a terceira operação é a aresta entre v e u: na segunda operação essa aresta é adicionada, enquanto na terceira, não. Vale destacar que a classe dos grafos de distância hereditária não está propriamente contida na classe dos grafos cordais. Essa afirmação pode ser facilmente confirmada ao se observar que um ciclo de tamanho quatro sem nenhuma corda pode ser construído da seguinte forma:

1. Ponto de Partida - Um grafo com um vértice isolado v_1;
2. $P(v_2, v_1)$ - Adicionar um vértice pendente v_2, conectado a v_1;
3. $P(v_3, v_1)$ - Adicionar um vértice pendente v_3, conectado a v_1;
4. $G_F(v_4, v_1)$ - Adicionar um gêmeo falso v_4 de v_1.

Os grafos ptolemaicos também podem ser construídos utilizando essas mesmas operações. Contudo, dada as restrições adicionais atreladas a caracterização da classe, a adição de gêmeos falsos é restrita aos vértices simpliciais. Essa restrição é suficiente para impedir a existência do exemplo anterior, onde um ciclo sem cordas foi construído: Não é possível adicionar v_4 como gêmeo falso de v_1, pois v_1 não é simplicial (v_2 e v_3 não são adjacentes).

A Figura 2.12 ilustra a construção de um grafo ptolemaico utilizando os operações definidas.



1	Vértice isolado 1
2	$P(4,1)$
3	$P(3,1)$
4	$G_V(2,1)$
5	$P(5,2)$
6	$G_V(6,5)$
7	$P(7,3)$
8	$P(8,4)$
9	$P(9,1)$
10	$P(10,9)$
11	$G_V(11,10)$
12	$G_F(12,11)$

Figura 2.12: Exemplo de grafo ptolemaico e as operações de construção utilizadas.

2.3.5 Grafos Bloco

Um grafo bloco pode ser definido como um grafo conexo onde cada componente biconexa é um grafo completo, isto é, o grafo é completo ou é formado por cliques conectadas por articulações. Uma articulação é um vértice cuja remoção desconecta o grafo. Uma componente biconexa é um subgrafo sem articulações, isto é, a remoção de um único vértice qualquer (que não seja uma articulação do grafo) não torna a componente desconexa. O Teorema 2.3.5, apresentado em Markenzon e Waga [70], apresenta uma compilação das caracterizações encontradas na literatura para esta família.

Teorema 2.3.5. *São equivalentes:*

1. *G é um grafo bloco.*

2. *G é um grafo conexo onde cada componente biconexa é um grafo completo.*

3. G é um grafo cordal livre de diamante (K_4 sem uma aresta).

4. G é cordal e qualquer separador minimal de vértices é um conjunto unitário.

A Figura 2.13 apresenta um exemplo de grafo bloco, composto por dezoito vértices, oito articulações e onze cliques maximais.

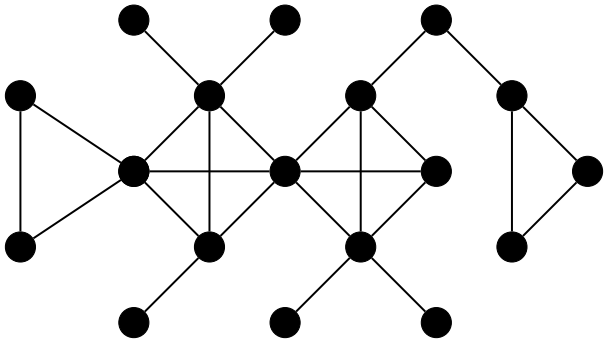


Figura 2.13: Exemplo de grafo bloco.

2.3.6 Grafos Completos

O grafo $G = (V, E)$ com n vértices é caracterizado como completo se cada vértice $v \in V$ for adjacente a todos os demais vértices de G, isto é, todos os vértices possuem grau $n-1$. Grafos pertencentes a essa classe são denominados K_n, onde n representa o número de vértices do grafo.

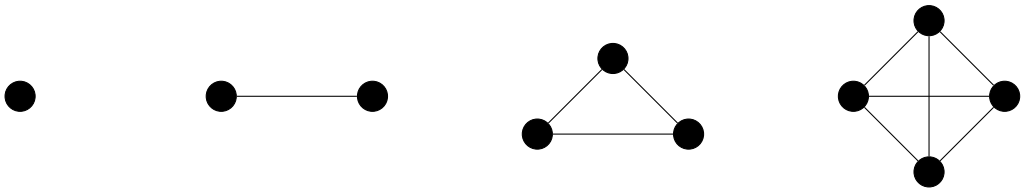


Figura 2.14: Os grafos K_1, K_2, K_3 e K_4, respectivamente.

Considerando que, em um grafo completo, todos os vértices devem ser adjacentes entre si, pode-se concluir que não existe a possibilidade de ciclos de tamanho maior do que 3 sem cordas, o que implica que todo grafo completo é cordal.

2.3.7 Árvores

Uma árvore é um grafo conexo e acíclico. Em uma árvore, todo vértice v tal que $d(v) \leq 1$ é chamado folha e os demais, interiores. Uma árvore pode possuir um vértice,

denominado raiz, que é ancestral de todos os demais vértices. Quando este vértice não existe, é possível escolher um vértice qualquer como tal. O Teorema 2.3.6 fornece quatro caracterizações adicionais para esta família.

Teorema 2.3.6. *As cinco afirmativas seguintes são equivalentes:*

- *O grafo $T = (V, E)$ é uma árvore.*
- *T é conexo e $|E|$ é mínima.*
- *T é conexo e $|E| = |V| - 1$.*
- *T é acíclico e $|E| = |V| - 1$.*
- *T é acíclico e para todo $v, w \in V$, a adição da aresta $\{v, w\}$ produz um grafo contendo exatamente um ciclo.*

Existem três outras classes de grafos que se definem por restrições adicionais às árvores: a primeira são os caminhos. Um grafo é dito um caminho se, além de conexo e acíclico, todos os seus vértices internos possuem grau 2. Já a segunda, denominada estrela, contém os grafos que são acíclicos, conexos e possuem um vértice universal, isto é, conectado a todos os outros vértices. A restrição de ser acíclico somada à necessidade de um vértice universal implica que todos os demais vértices são folhas. A terceira classe é conhecida como *Caterpillar* e contém árvores que, uma vez removidas todas as suas folhas, se resumem a um caminho com pelo menos um vértice.

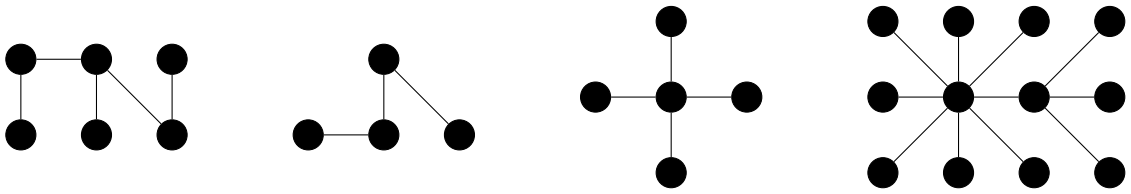


Figura 2.15: Exemplo de Árvore, Caminho, Estrela e *Caterpillar*, respectivamente.

2.3.8 Grafos AC

Antes de definir a classes dos grafos AC é necessário apresentar uma outra: os grafos dominó. Um grafo é dito dominó (Kloks *et al.* [59]) se cada vértice pertence a no máximo duas cliques maximais. É interessante observar que esta família não está propriamente contida na família dos grafos cordais: um ciclo sem cordas com n ($n \geq 4$) vértices contém

n arestas; cada aresta induz uma clique maximal do grafo; cada vértice é extremo de duas arestas, implicando que cada vértice está contido em exatamente duas cliques e que todo C_k é um grafo dominó.

A família dos grafos AC (Blair [4]) é definida pela interseção dos grafos dominó e dos grafos cordais. A necessidade de ser cordal implica que o caminho formado pelas cliques maximais seja aberto, tornando evidente a existência de vértices simpliciais.

Além dessa caracterização baseada na interseção entre as classes, existe uma outra baseada em subgrafos induzidos: grafos AC são os grafos cordais livres de gemas e garras como subgrafos induzidos. Vale notar que a gema e a garra violam a definição dos grafos AC uma vez que existem vértices contidos em três cliques maximais.

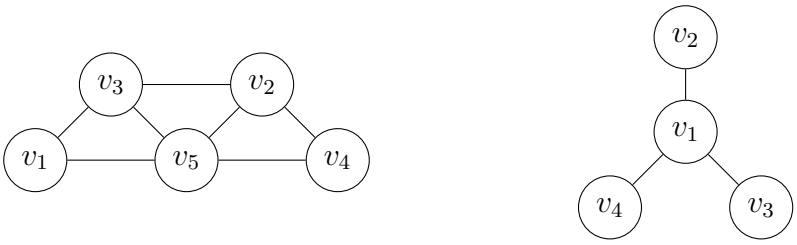


Figura 2.16: Exemplo de um Gema e de uma Garra.

Antes de prosseguir é interessante definir a localização desta classe de grafos na hierarquia dos grafos cordais. O artigo de Markenzon e Waga [69] demonstrou que todo grafo AC é também um grafo ptolemaico. Essa caracterização permite estabelecer que os grafos AC também possuem *clique-width* limitado.

2.3.9 Intervalo e Intervalo Próprio

Um grafo $G = (V, E)$ é dito grafo de intervalo se ele é o grafo de interseção de uma família finita de intervalos em uma reta real. A cada intervalo corresponde um vértice e dois vértices são adjacentes se, e somente se, os intervalos correspondentes possuem uma interseção. A Figura 2.17 apresenta um exemplo de um conjunto de intervalos e o grafo que o representa.

Além da definição fornecida, os grafos de intervalo também podem ser caracterizados como grafos cordais sem triplas asteroidais. Uma tripla asteroidal é constituída por um trio de vértices não adjacentes entre si tal que existem caminhos conectando cada par de vértices evitando a vizinhança do terceiro. A Figura 2.18 apresenta três grafos onde os vértices a, b e c definem triplas asteroidais. Como exemplo desta afirmação, pode-se observar que

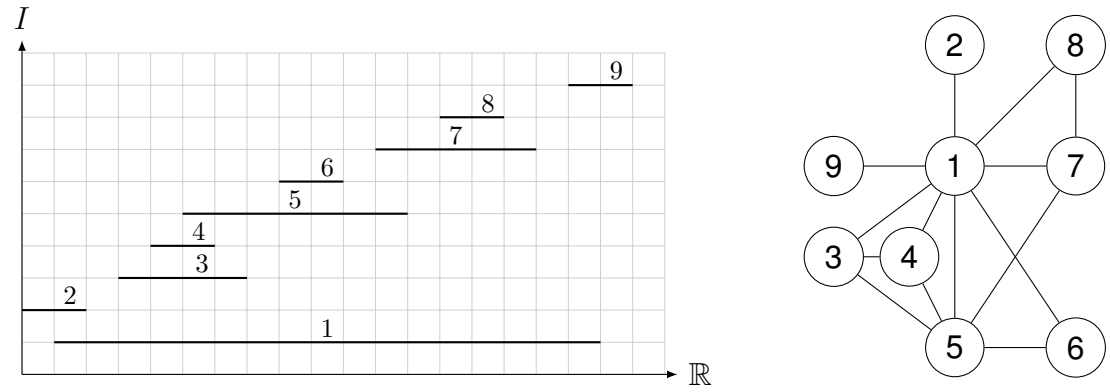


Figura 2.17: Exemplo de grafo de intervalo e sua representação na reta real.

no primeiro grafo (garra dupla), é possível ir do vértice a para o vértice b sem passar por nenhum vizinho de c. Como essa observação vale para qualquer combinação entre a, b e c, fica claro que estes três vértices formam uma tripla asteroidal.

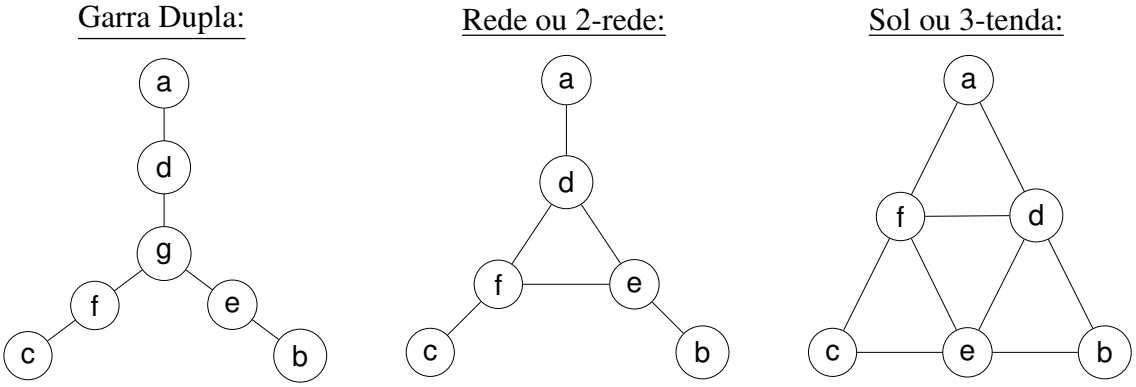


Figura 2.18: Exemplos de Triplas Asteroidais.

Grafos de intervalo próprio são definidos por uma restrição aos grafos de intervalo onde nenhum intervalo pode conter propriamente um outro. A Figura 2.19 apresenta um exemplo de grafo de intervalo próprio.

O trabalho de Roberts [76] apresenta caracterizações adicionais para os grafos de intervalo próprio:

Teorema 2.3.7 (Roberts [76])**.** *Seja $G = (V, E)$ um grafo não direcionado. As seguintes afirmações são equivalentes:*

- *G é um grafo de intervalo próprio;*
- *G é um grafo de intervalo unitário;*
- *G é um grafo de indiferença;*

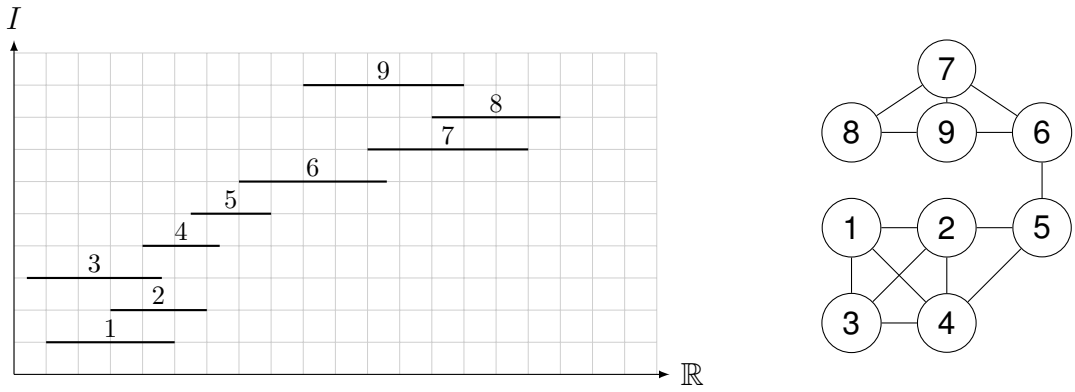


Figura 2.19: Exemplo de grafo de intervalo próprio.

- *G é um grafo de intervalo que não contém uma garra como subgrafo induzido.*

O Teorema 2.3.7 introduz duas novas classes: Grafos de Intervalo Unitário e Grafos de Indiferença. Um grafo é dito grafo de intervalo unitário se ele admite uma representação na forma de intervalos na qual todos os intervalos da reta real possuem comprimentos unitários (ou iguais). Essa caracterização já é suficiente para verificar a equivalência entre as classes: se todos os intervalos possuem o mesmo comprimento então não é possível que um intervalo contenha propriamente um outro.

Um grafo é dito de indiferença se ele admite uma função $f : V \to \mathbb{R}$ e um número real k tal que para quaisquer dois vértices $u, v \in V$, $\{u, v\} \in E$ se, e somente se, $|f(u) - f(v)| \leq k$. Roberts [76] também estabelece que um grafo é de indiferença se ele não contém como subgrafos induzidos uma garra, ciclos com mais de quatro vértices sem cordas, uma rede ou um sol (ver Figura 2.18). Um grafo é chamado de Grafo Garra se ele é constituído por quatro vértices, tais que um vértice é universal e os três demais tem grau 1. Esse grafo também é chamado na literatura de $K_{1,3}$, isto é, um grafo bipartido conexo onde a primeira partição é um conjunto unitário e a outra contém os três vértices remanescentes. É interessante observar que este grafo, embora simples, não admite uma representação por intervalos unitários ou próprios. A Figura 2.20 apresenta o grafo garra, com uma tentativa de o representar como um modelo de interseção com intervalos de mesmo tamanho. No modelo de interseção de intervalos é possível observar que não é possível representar a interseção entre os intervalos correspondentes aos vértices c e d sem que c apresente também uma interseção com o vértice b (marcada pela linha vermelha no modelo de interseção). No segundo modelo é possível visualizar que este grafo pode ser facilmente representado como intervalos na reta real se o intervalo d contiver propriamente o intervalo b.

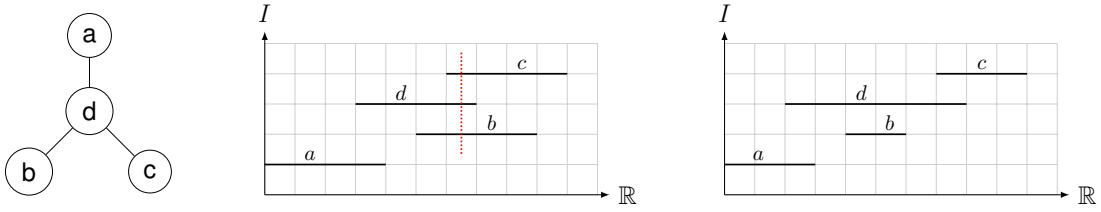


Figura 2.20: Grafo garra e representações como modelos de intervalos.

Os grafos de intervalo e de intervalo próprio apresentam uma propriedade interessante relacionada à ordenação linear das cliques maximais. Segundo Gilmore e Hoffman [48], as cliques maximais de um grafo de intervalo podem ser linearmente ordenadas de modo que, para cada vértice do grafo, as cliques maximais que o contém ocorrem consecutivamente na representação de intervalos deste grafo. Além disso, a identificação das cliques maximais também pode ser trivialmente realizada, percorrendo a reta real e identificando as interseções entre os intervalos. A Figura 2.21 ilustra um exemplo da ordenação linear das cliques de um grafo de intervalo próprio. Em relação a árvore de cliques, outra característica dos grafos de intervalo é que existe uma árvore de cliques que corresponde a um caminho, isto é, possui exatamente duas folhas.

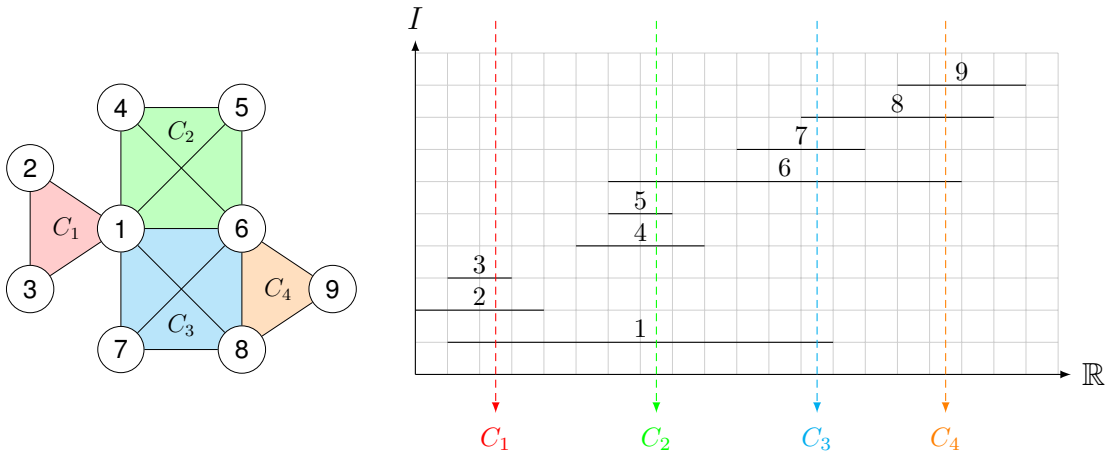


Figura 2.21: Identificação e ordenação das cliques maximais em um grafo de intervalo.

2.3.10 Grafos Dominó ∩ Intervalo Próprio

Além dos grafos AC, apresentados na Subseção 2.3.8, existe outra classe definida por uma interseção com os grafos dominó: os grafos dominó ∩ intervalo próprio.

É interessante observar que a interseção entre grafos de intervalo e grafos dominó equivale a família dominó ∩ intervalo próprio. Essa afirmação pode ser facilmente verificada:

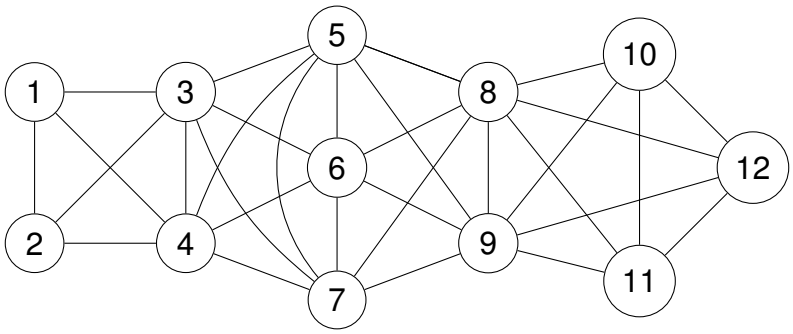


Figura 2.22: Exemplo de Grafo Dominó ∩ Intervalo Próprio.

(i) um grafo é de intervalo próprio se ele é um grafo de intervalo que não contém uma garra (grafo $K_{1,3}$) como subgrafo induzido; (ii) o vértice universal do grafo garra está contido em três cliques maximais (cada clique maximal contém um vértice folha e o vértice universal), violando a caracterização de grafo domino; (iii) como o único subgrafo induzido proibido pela definição dos grafos de intervalo próprio é inibido pela caracterização dos grafos dominó, fica evidente que as duas classes resultantes equivalem.

Além das relações já apresentadas, resta demonstrar que a família dos grafo dominó ∩ intervalo próprio está propriamente contida nos classe de grafo AC: como todo grafo dominó ∩ intervalo próprio é um grafo cordal no qual cada vértice está contido em no máximo duas cliques maximais, fica evidente a continência entre as famílias.

Ao restringir que todo vértice em um grafo de intervalo próprio esteja contido em no máximo duas cliques maximais, obtém-se uma restrição da classe dos grafos de intervalo próprio, cuja estrutura ainda mais simplificada, favorece a exploração dessas classes e de suas características estruturais. Um exemplo dessa afirmativa é a identificação das cliques maximais e de suas interseções: como todo vértice ou é simplicial ou está contido em no máximo duas cliques, é possível estabelecer de forma simples uma ordenação das cliques que corresponde a um caminho das cliques maximais.

A Figura 2.22 ilustra um exemplo de grafo dominó ∩ intervalo próprio. A figura apresenta quatro cliques maximais $Q_1 = \{1, 2, 3, 4\}$, $Q_2 = \{3, 4, 5, 6, 7\}$, $Q_3 = \{5, 6, 7, 8, 9\}$ e $Q_4 = \{8, 9, 10, 11, 12\}$, onde se pode identificar que nenhum vértice está presente em mais de duas cliques.

2.3.11 Grafos *Split*

Um grafo $G = (V, E)$ é caracterizado como *split* se seu conjunto de vértices V admitir uma partição em dois conjuntos disjuntos, tais que o primeiro constitui uma clique, isto é, todos os vértices são adjacentes entre si, e o segundo, um conjunto independente (o conjunto de arestas do subgrafo induzido por estes vértices é vazio). A Figura 2.23 apresenta um exemplo de grafo *split*: Os vértices da clique estão hachurados em preto, enquanto os do conjunto independente, em branco.

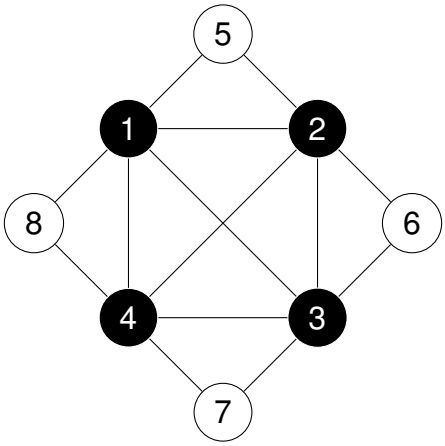


Figura 2.23: Exemplo de Grafo *Split*.

2.3.12 Grafos *Split*-Indiferença

Os grafos *split*-indiferença consistem da interseção entre a classe dos grafos *split* e os grafos de intervalo próprio, ou seja, atendem em simultâneo as restrições inerentes a ambas as classes. A primeira definição formal desta classe foi apresentada por Ortiz *et al.* [74] no seguinte teorema:

Teorema 2.3.8 (Ortiz *et al.* [74])**.** *Seja $G = (V, E)$ um grafo conexo, G é um grafo split-indiferença se e somente se:*

1. *G é um grafo completo (Caso 1), ou*
2. *G possui duas cliques maximais C_1 e C_2, tais que $|C_1 \setminus C_2| = 1$ (Caso 2) ou*
3. *G possui três cliques maximais C_1, C_2 e C_3, tais que*
   - *$|C_1 \setminus C_2| = |C_3 \setminus C_2| = 1$*
   - *$C_1 \cap C_3 = \emptyset$ (Caso 3a) ou $C_1 \cup C_3 = V$ (Caso 3b).*

No Teorema 2.3.8 podem ser observados quatro casos distintos de grafos *split*-

indiferença. Essa observação é baseada no número de cliques e em suas interseções. A Figura 2.24 apresenta um exemplo para cada caso.

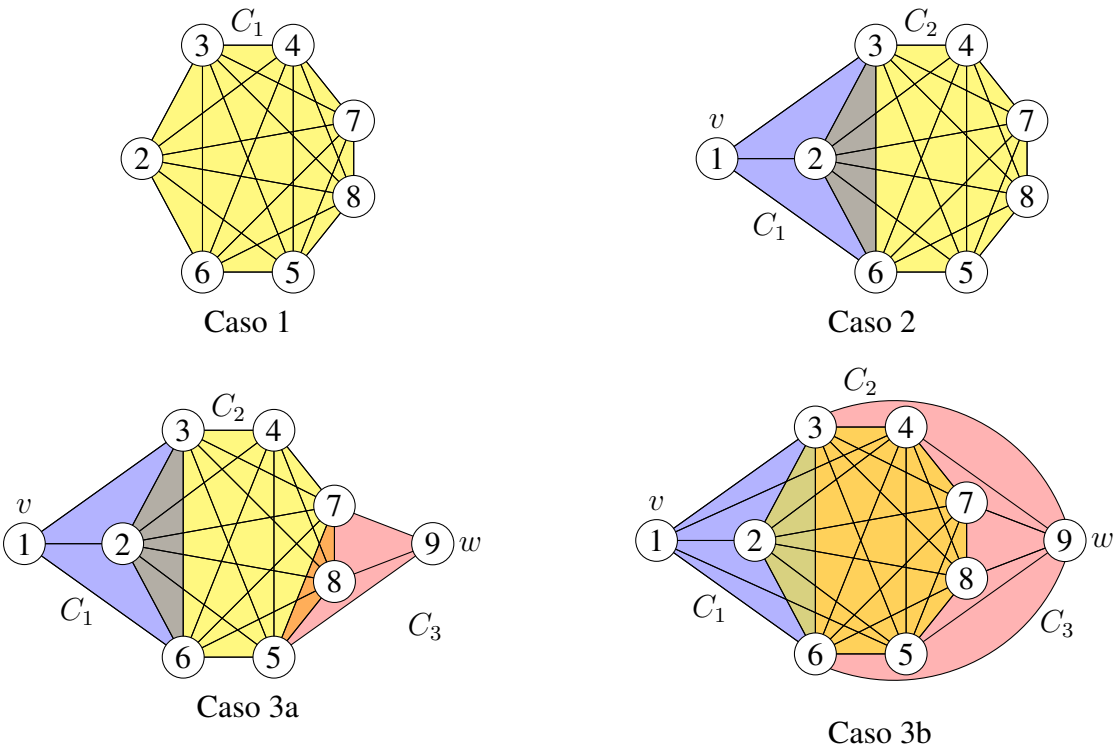
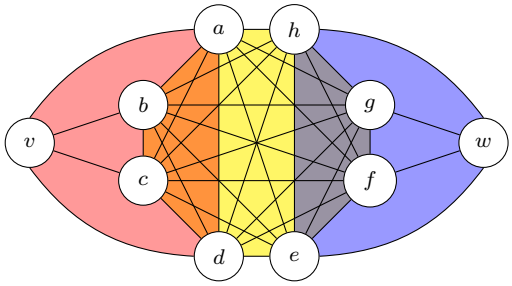


Figura 2.24: Exemplos dos quatro casos de grafos *split*-indiferença.

Vale destacar que a definição proposta por Ortiz *et al.* [74] permite que um mesmo grafo seja caracterizado como Caso 3a e 3b, isto é, o grafo satisfaz $C_1 \cap C_3 = \emptyset$ e $C_1 \cup C_3 = V$ simultaneamente. A Figura 2.25 ilustra um exemplo de grafo que admite essa dubiedade quanto a caracterização.



$C_1 = \{v, a, b, c, d\}$
$C_2 = \{a, b, c, d, e, f, g, h\}$
$C_3 = \{e, f, g, h, w\}$
$C_1 \setminus C_2 = \{v\}$ e $C_3 \setminus C_2 = \{w\}$
$C_1 \cap C_3 = \emptyset$
$C_1 \cup C_3 = \{v, a, b, c, d, e, f, g, h, w\}$

Figura 2.25: Exemplo de grafo *split*-indiferença que satisfaz as restrições dos casos 3a e 3b simultaneamente.

O trabalho de Markenzon e Waga [68] propõe uma caracterização alternativa que restringe a possibilidade de um grafo atender em simultâneo os Caso 3a e 3b do Teorema 2.3.8. Essa nova caracterização é baseada nos separadores minimais de vértices. Um conjunto S

é um separador minimal de vértices para um grafo $G = (V, E)$ se sua remoção aumenta o número de componentes conexas do grafo, isto é, existe um par de vértices $v_1, v_2 \in V \setminus S$, tal que G admite um caminho entre v_1 e v_2, porém em $G - S$ não existe nenhum caminho possível entre v_1 e v_2.

Teorema 2.3.9 (Markenzon e Waga [68])**.** *Um grafo $G = (V, E)$ é um grafo split-indiferença se e somente se:*

1. *G é um grafo completo (Caso 1), ou*
2. *G possui duas cliques maximais C_1 e C_2, tais que $|C_1 \setminus C_2| = 1$ (Caso 2) ou*
3. *G possui três cliques maximais C_1, C_2 e C_3, tais que*
   - *$|C_1 \setminus C_2| = |C_3 \setminus C_2| = 1$*
   - *$S_{1,2} \cap S_{3,2} = \emptyset$ (Caso 3a) ou*
   - *$S_{1,2} \cap S_{3,2} \neq \emptyset$ e $|S_{1,2} \cup S_{3,2}| = |C_2|$ (Caso 3b)*

   Onde $S_{1,2} = C_1 \cap C_2$ e $S_{3,2} = C_3 \cap C_2$ são separadores minimais para G

É interessante observar que o grafo apresentado na Figura 2.25 é caracterizado pelo Teorema 2.3.9 como Caso 3a, pois $S_{1,2} = \{a, b, c, d\}$, $S_{3,2} = \{e, f, g, h\}$ e $S_{1,2} \cap S_{3,2} = \emptyset$.

Capítulo 3

Estado da Arte da Complexidade Computacional do Problema da Dominação Vetorial

Este capítulo apresentará o estado da arte da complexidade computacional do problema da dominação vetorial em grafos com *clique-width* limitada e em algumas famílias de grafos cordais. Entretanto, antes de avançar neste tópico, é necessário demonstrar que a versão de decisão do problema da dominação vetorial está em $\mathcal{NP}$.

Problema da Dominação Vetorial - Certificação de um conjunto S:

Entradas: Um grafo $G = (V, E)$ e um vetor de requisitos R tal que $R[v] \in \{0, 1, \ldots, d(v)\}\ \forall v \in V$.

Questão: O conjunto $S \subseteq V$ é capaz de R-dominar G?

Verificação: Para cada vértice $v \in V$ do grafo: Avaliar se S satisfaz $v \in S$ (v está em S) ou $|N(v) \cap S| > R[v]$ (v possui pelo menos $R[v]$ vizinhos em S). Caso a condição seja verdadeira para todo vértice do grafo, então S é um conjunto R-dominante para G e R. Do contrário o conjunto dos vértices não dominados demonstram que S não é um conjunto R-dominante. Esta verificação pode ser realizada em tempo polinomial, pois cada vértice é analisado uma única vez.

Em relação à complexidade computacional do problema em grafos em geral, é interessante observar que o problema do conjunto dominante pode ser analisado como uma restrição do problema da dominação vetorial onde todo vértice $v \in V$ possui $R[v] = 1$.

Desta forma, como o problema do conjunto dominante é reconhecidamente $\mathcal{NP}$-Completo em grafos em geral (Garey e Johnson [45]), é possível concluir que o problema da dominação vetorial também é $\mathcal{NP}$-Completo em grafos em geral.

Entretanto, dada as restrições implícitas à algumas famílias de grafos em específico, surge a dúvida se o problema permanece ou não intratável computacionalmente nestas famílias (supondo $\mathcal{P} \neq \mathcal{NP}$). Sendo assim, o foco deste trabalho é determinar para quais famílias de grafos o problema continua $\mathcal{NP}$-Completo ou passa a admitir algoritmos de tempo polinomial. O escopo do estudo foi reduzido a alguns grafos da família dos cordais, tendo em vista o vasto número de famílias de grafos conhecidas.

A Seção 3.3 é dedicada ao estudo dos grafos cordais, incluindo algumas definições, caracterizações e ferramentas estruturais. Esta seção também aborda a complexidade computacional do problema da dominação vetorial nesta família de grafos, incluindo um diagrama que expõe a hierarquia da família dos grafos cordais e o estado da arte da complexidade computacional.

3.1 Grafos com *Clique-width* limitada

Esta seção abordará a complexidade do problema da dominação vetorial em grafos com *clique-width* limitada. Como mencionado anteriormente, o parâmetro *clique-width* (cw) de um grafo é um parâmetro que representa o menor número de rótulos necessários para construir um grafo utilizando apenas quatro operações: L, $\oplus$, η e ρ. A definição de *clique-width* e as operações de construção foram apresentadas no Capítulo 2. O parâmetro *clique-width* também pode se aplicado a uma família de grafos, significando o menor número de rótulos necessários para construir qualquer grafo desta família. É importante notar que o valor do *clique-width* nestes casos independe do número de vértices do grafo.

Como o valor do parâmetro *clique-width* representa a complexidade estrutural de cada família de grafos, nota-se que este valor está intimamente ligado a complexidade computacional de alguns problemas em grafos, como, por exemplo, os problemas de dominação. Desta forma, esta seção apresentará um estudo da complexidade computacional do problema da dominação vetorial em famílias de grafos com *clique-width* limitada por uma constante k.

O estudo da complexidade computacional apresentará dois resultados da literatura sobre grafos com *clique-width* limitada. O primeiro é o algoritmo proposto por Cicalese *et al.*

[18] para o problema da (λ, β, α)-SELEÇÃO DE ALVOS. Enquanto o segundo é baseado no Teorema de Courcelle (Courcelle e Olariu [27], Courcelle [23] e Courcelle *et al.* [26]).

Como mencionado anteriormente, o problema da (λ, β, α)-SELEÇÃO DE ALVOS busca encontrar um conjunto R-dominante S de cardinalidade β suficiente para dominar pelo menos α vértices do grafo em no máximo λ iterações. O problema da dominação vetorial é um caso particular deste problema onde $\lambda = 1$ e $\alpha = n$.

O algoritmo proposto por Cicalese *et al.* [18] é baseado em programação dinâmica e parte de uma árvore T definida pela k-expressão irredundante σ fornecida para o grafo de entrada.

O algoritmo percorre a árvore T partindo das folhas e seguindo em direção a raiz. Para o problema da dominação vetorial, o algoritmo apresenta complexidade $O(k|\sigma|(n+1)^{5k})$, onde σ é uma k-expressão para o grafo.

Entretanto, é possível estabelecer o problema da dominação vetorial com requisitos limitados em grafos com *clique-width* limitada apresenta admite um algoritmo de tempo linear. O Teorema de Courcelle [27] estabelece que problemas que podem ser definidos como formulações expressas em lógica monádica de segunda ordem, quando restritos a grafos com *clique-width* limitados, admitem algoritmos de tempo linear.

Problema da dominação vetorial com requisitos limitados
Instância: Um grafo $G = (V, E)$, constantes r e t, vetor de requisitos $R = (R[v] \in \{1, \ldots, \min(d(v), r)\} : v \in V\}$
Problema: Determinar se existe ou não $S \subseteq V$, sujeito à $|S| \leq t$ tal que S seja suficiente para R-dominar G.

Resta estabelecer se o problema da dominação vetorial com requisitos limitados a uma constante r pode ser definido como uma expressão ***MSOL***, mais propriamente, $LinMSOL$ [26], pois o objeto neste caso é minimizar o tamanho do conjunto resultante.

Teorema 3.1.1. *O problema da dominação vetorial com requisitos limitados a uma constante r é um problema $LinMSOL$ expressável.*

Demonstração. Sejam $G = (V, E)$ e R, um grafo e um vetor de requisitos R, tal que $R[v] \in \{0, \ldots, \min(d(v), r)\}$ $\forall v \in V$. A demonstração de que o problema admite uma formulação $MSOL$ será realizada analisando o valor de r: O primeiro caso que deve ser

considerado é $r \leq 1$ ($r = 0$ é trivial). Se todo requisito está limitado a $r \in \{0, 1\}$, então existem dois casos para um vértice v:

$\boldsymbol{R[v] = 0}$ Neste caso, v é auto resolvido e a regra da dominação é trivialmente satisfeita;

$\boldsymbol{R[v] = 1}$ Para dominar v é necessário que $v \in S$ ou que exista um vértice a, adjacente de v, tal que $a \in S$.

Sendo assim, seja φ_1 a formulação $LinMSOL$ equivalente ao problema da dominação vetorial com requisitos de no máximo 1:

$$\begin{aligned}\varphi_1 \equiv \min \mid S \mid\ ((S \subseteq V) \wedge (\forall v\ (S(v) \vee \\ (R(v) = 0) \vee \\ ((R(v) = 1) \wedge (\exists a\ (S(a) \wedge E(v, a)))))))\end{aligned}$$

Na formulação, $S(v)$ retorna verdadeiro se v estiver contido em S, e falso do contrário, $E(v, a)$ retorna verdeiro se, e somente se, existir uma aresta entre os vértices v e a, e $R(v)$ corresponde a uma função que retorna o requisito do vértice v. Considerando que os grafos abordados são simples e sem laços, pode-se afirmar que se $E(v, a)$ é verdadeiro, então v é diferente de a (do contrário, a aresta seria um laço).

Nota-se que existem duas condições aplicadas a todos os vértices de v, unidas por um operador lógico OU ($\vee$), implicando que v é considerado dominado se pelo menos uma condição for avaliada como verdadeira (embora em alguns casos ambas podem ser verdadeiras). A primeira analisa se v está contido em S ($S(v)$), o que é suficiente para dominar v. Já a segunda avalia dois casos relacionados ao requisito do vértice v: Se $R(v) = 0$, então v está trivialmente dominado, satisfazendo a fórmula. Por outro lado, se $R(v) = 1$, então, para dominar v, é necessário que exista um vértice a tal que $S(a)$ e $E(a, v)$ sejam verdadeiros, isto é, v possui um vizinho a em S.

Antes de passar para a demonstração de um caso geral, é necessário avaliar o cenário onde $r \leq 2$. Essa análise adicional é necessária, pois o primeiro caso não é suficientemente significativo para evidenciar todas as peculiaridades do problema. Desta forma, seja φ_2 a formulação $LinMSOL$ para os casos em que $r \leq 2$:

$$\begin{aligned}\varphi_2 \equiv \min \mid S \mid\ & ((S \subseteq V) \wedge (\forall v\ (S(v) \vee \\ & (R(v) = 0) \vee \\ & ((R(v) = 1) \wedge (\exists a\ (S(a) \wedge E(v,a)))) \vee \\ & ((R(v) = 2) \wedge (\exists a_1, a_2\ (S(a_1) \wedge S(a_2) \wedge E(v,a_1) \wedge E(v,a_2) \wedge Dif(a_1,a_2)))))))\end{aligned}$$

Nota-se que a expressão associada ao caso em que $R(v) = 2$ demandou a aplicação de uma validação adicional para assegurar que os vértices a_1 e a_2 são diferentes. Sem essa condição, bastaria um vizinho de v para validar a dominação de v ($a_1 = a_2$). Desta forma, seja $Dif(a_1, a_2)$ um operador adicional capaz de distinguir se dois vértices são diferentes.

Uma vez que foi estabelecida uma expressão $LinMSOL$ para os casos onde $R[v] \leq 2$, pode-se demonstrar uma formulação para o caso geral. Desta forma, seja $\varphi(r)$ a expressão em $LinMSOL$ para o problema da dominação vetorial com requisitos limitados a r.

$$\begin{aligned}\varphi_r \equiv \min \mid S \mid\ & ((S \subseteq V) \wedge (\forall v\ (S(v) \vee \\ & (R(v) = 0) \vee \\ & ((R(v) = 1) \wedge (\exists a \\ & \quad (S(a) \wedge E(v,a)))) \vee \\ & ((R(v) = 2) \wedge (\exists a_1, a_2 \\ & \quad ((S(a_1) \wedge E(v,a_1) \wedge Dif(a_1,a_2)) \wedge \\ & \quad (S(a_2) \wedge E(v,a_2))))) \vee \\ & \vdots \\ & ((R(v) = r) \wedge (\exists a_1, a_2, \ldots, a_r \\ & \quad ((S(a_1) \wedge E(v,a_1) \wedge Dif(a_1,a_2) \wedge \ \ldots \ \wedge Dif(a_1,a_{r-1}) \wedge Dif(a_1,a_r)) \wedge \\ & \quad (S(a_2) \wedge E(v,a_2) \wedge Dif(a_2,a_3) \wedge \ \ldots \ \wedge Dif(a_2,a_{r-1}) \wedge Dif(a_2,a_r)) \wedge \\ & \quad \vdots \\ & \quad (S(a_{r-1}) \wedge E(v,a_{r-1}) \wedge Dif(a_{r-1},a_r)) \wedge \\ & \quad (S(a_r) \wedge E(v,a_r)))))))\end{aligned}$$

□

Uma vez que foi demonstrado que o problema da dominação vetorial com requisitos limitados a uma constante r pode ser definido por uma expressão em $LinMSOL$, pode-se estabelecer que:

Corolário 3.1.1. *O problema da dominação vetorial com requisitos limitados a uma constante r admite um algoritmo de tempo linear quando restrito a famílias de grafos com clique-width limitada por uma constante k.*

Demonstração. Segue como um resultado imediato do Teorema 3.1.1 e do Teorema de Courcelle. □

Embora o Corolário 3.1.1 demonstre que existe um algoritmo de tempo linear, ainda não foi possível projetá-lo. Desta forma, consta como trabalho futuro apresentar tal algoritmo.

Uma vez que o estudo da complexidade computacional do problema da dominação vetorial em famílias de grafos com *clique-width* limitada foi concluído, é possível retomar a exploração de famílias de grafos.

Considerando que o parâmetro *clique-width* apresenta uma relação como número de cliques maximais do grafo, uma continuidade natural do estudo foram grafos com número limitado de cliques maximais. Desta forma, a Seção 3.2 abordará essa classe de grafos.

3.2 Grafos com Número Limitado de Cliques Maximais

Esta seção abordará alguns casos de grafos com um número limitado de cliques maximais. Esse limite permite estabelecer com maior evidência quais características dos grafos tornam o problema da dominação vetorial aparentemente intratável computacionalmente (supondo $\mathcal{P} \neq \mathcal{NP}$) ou colaboram para possibilitar o desenvolvimento de algoritmos polinomiais.

O primeiro caso abordado nesta seção são os grafos com uma única clique maximal, isto é, os grafos completos. Em seguida, serão abordados grafos com duas, três e ℓ cliques maximais. Vale destacar que grafos com até três cliques maximais são grafos cordais (inexiste a possibilidade de ciclos sem cordas no interior das cliques ou mesmo na interseção entre elas). Entretanto, ao extrapolar esse limite (grafos com quatro ou mais cliques maximais), não é mais possível assegurar que tais grafos sejam cordais. Isto implica que

esta seção excede em parte o escopo deste trabalho. Contudo, os resultados obtidos permitem visualizar com mais clareza quais características colaboram para tornar o problema da dominação vetorial tratável em algumas classes de grafos.

3.2.1 Grafos Completos

Existem na literatura diversas versões de algoritmos para o problema da dominação vetorial em grafos completos, como, por exemplo, os algoritmos propostos por Cicalese *et al.* [19] e Lan e Chang [60]. A versão apresentada neste trabalho corresponde a uma simplificação destes algoritmos.

O Algoritmo 3.1 encontra um conjunto R-dominante para grafos completos. Seu funcionamento é baseado na seleção de um determinado número de vértices que seja suficiente para R-dominar os demais. O critério de seleção destes vértices é guloso, isto é, os vértices de maiores requisitos são selecionados primeiro, pois claramente são os mais difíceis de se dominar.

Algoritmo 3.1: Grafos completos: Dominação Vetorial

Entrada: Grafo completo $K = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

```
1  S ← ∅
2  Seja L uma lista dos vértices de K
3  Ordenar L em ordem decrescente de requisitos
4  enquanto L ≠ ∅ faça
5      Seja v o primeiro vértice de L
6      se R[v] > |S| então
7          S ← S ∪ {v}
8      senão
9          Interromper o laço de repetição
10     L ← L \ {v}
11 retorne S
```

Teorema 3.2.1. *O Algoritmo 3.1 encontra um conjunto R-dominante para grafos completos.*

Demonstração. O algoritmo é baseado em sucessivas adições ao conjunto em construção, até que todo o grafo seja R-dominado. O critério utilizado para selecionar qual vértice deve ser adicionado ao conjunto garante que os vértices de maiores requisitos são selecionados

primeiro, pois obviamente são os de mais difícil dominação. Assim que um vértice é R-dominado, todos os demais também serão, pois possuem requisitos menores ou iguais e, portanto, o algoritmo pode ser interrompido.

Seja v o último vértice adicionado ao conjunto construído S. Considerando que v possui o menor requisito em S e que nenhum conjunto de menor cardinalidade pode R-dominar v (do contrário v não teria sido adicionado), pode-se concluir que o conjunto resultante S é mínimo. □

A complexidade deste algoritmo é definida pela etapa de ordenação dos vértices. Considerando que os requisitos dos vértices são limitados a n, pode-se utilizar um algoritmo de ordenação em tempo linear, tal como o *Bucket Sort* [22]. Além disso, é evidente que cada vértice é tratado no máximo uma única vez. Logo, é possível concluir que a complexidade deste algoritmo é $O(n)$.

O Algoritmo 3.1 será utilizado como base para diversas famílias de grafos, dada a sua capacidade de encontrar conjuntos R-dominantes mínimos para cliques.

Dentre as classes cuja solução é baseada no algoritmo para grafos completos destacam-se os grafos bloco e as árvores. O primeiro, pois cada componente do grafo é por definição um grafo completo, como será visto na Subseção 3.3.4. Já as árvores podem ser vistas como cliques de tamanho dois conectadas entre si. As árvores serão abordadas na Subseção 3.3.5.

3.2.2 Grafos Conexos com Duas Cliques Maximais

Este subseção constitui uma evolução natural aos grafos apresentados na Subseção 3.2.1: grafos conexos com exatamente duas cliques maximais. É interessante observar que a restrição no número de cliques maximais, implicou na continência desta classe em uma outra: os grafos AC.

Tendo em vista que os grafos AC são definidos pela interseção entre os grafos dominó e os grafos cordais, essa relação entre as classes pode ser facilmente verificada: como existem apenas duas cliques maximais, inexiste a possibilidade de um vértice estar contido em mais de duas cliques e também de ciclos com mais de três vértices sem cordas. Essa caracterização permite definir que todo grafo com duas cliques maximais possui *clique-width* limitada. Como visto na Seção 3.1, os grafos com *clique-width* limitada admitem um algoritmo de tempo polinomial (Cicalese *et al.* [18]).

Entretanto, a complexidade do problema da dominação vetorial nesta classe pode ser melhorada. A Seção 4.1 apresenta um algoritmo linear desenvolvido durante este trabalho para o problema da dominação vetorial em grafos com exatamente duas cliques maximais.

3.2.3 Grafos Conexos com Três Cliques Maximais

Nesta seção serão abordados grafos conexos com exatamente três cliques maximais. Diferentemente do caso anterior, existem alguns casos distintos de grafos nesta família. Para identificar estes casos, rotulou-se as diferentes regiões que podem compor tais grafos. A Figura 3.1 apresenta estas regiões.

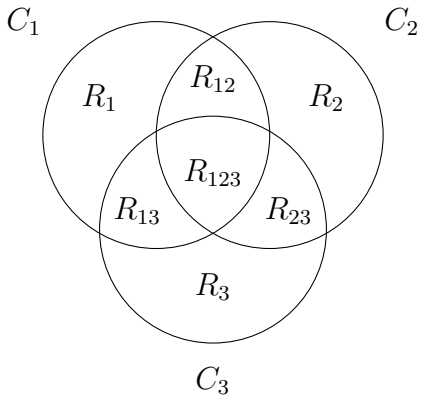
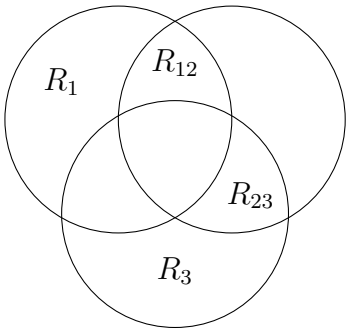


Figura 3.1: Regiões que podem compor um grafo com três cliques maximais.

Para reconstruir o grafo a partir dessa representação de regiões, basta adicionar todas as arestas possíveis entre os vértices situados em uma mesma região e entre os vértices situados em regiões contidas em uma mesma clique. É interessante notar que, em alguns casos, a adição de vértices em todas as regiões leva a grafos com mais de três cliques maximais, o que extrapola o alvo desta seção. Sendo assim, foi necessário estabelecer todos os possíveis casos de grafos com exatamente três cliques maximais e suas representações como regiões.

Foram obtidos cinco casos de grafos com três cliques maximais. As Figuras 3.2 a 3.6 apresentam os casos enumerados. Deve-se notar que existem outras possibilidades de grafos com três cliques maximais, mas todas são simétricas às apresentadas. Nas figuras, as regiões marcadas em negrito correspondem aos vértices simpliciais das cliques.

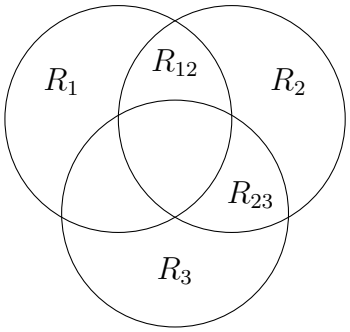
De acordo com a representação do grafo como regiões no diagrama, existe um sexto caso. Contudo, o grafo correspondente a este caso não é distinto do terceiro caso apresentado. Considerando, sem perda de generalidade, que a região R_2 é vazia ($R_2 = \emptyset$),

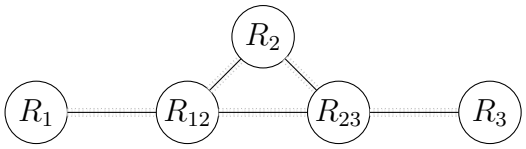


$R_2 = \emptyset$
$R_{13} = \emptyset$
$R_{123} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{12}\}$
$C_2 = \{R_{12}, R_{23}\}$
$C_3 = \{\boldsymbol{R_3}, R_{23}\}$

Figura 3.2: Três cliques maximais - caso 1.

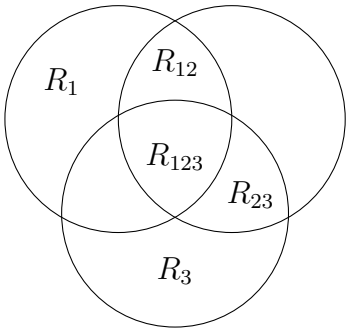


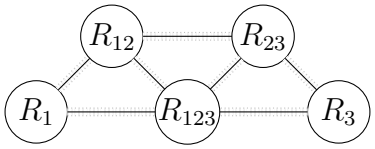


$R_{13} = \emptyset$
$R_{123} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{12}\}$
$C_2 = \{\boldsymbol{R_2}, R_{12}, R_{23}\}$
$C_3 = \{\boldsymbol{R_3}, R_{23}\}$

Figura 3.3: Três cliques maximais - caso 2.





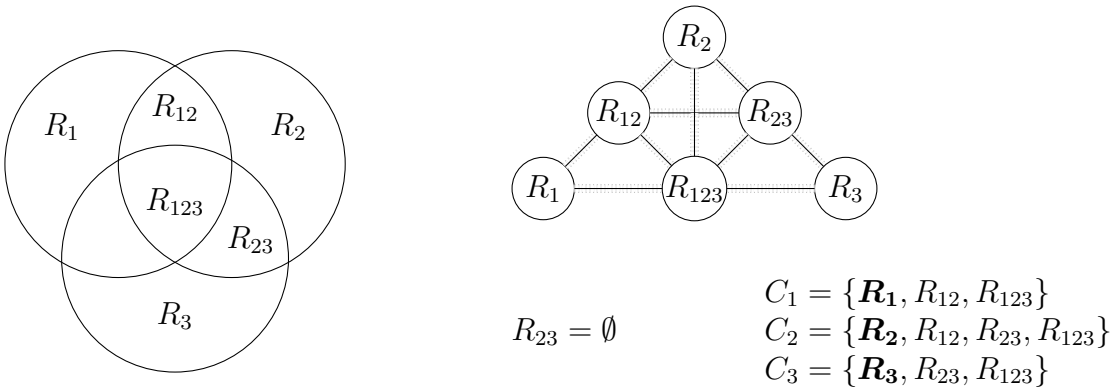
$R_2 = \emptyset$
$R_{13} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{12}, R_{123}\}$
$C_2 = \{R_{12}, R_{23}, R_{123}\}$
$C_3 = \{\boldsymbol{R_3}, R_{23}, R_{123}\}$

Figura 3.4: Três cliques maximais - caso 3.

não existe a possibilidade de que as regiões R_{13} e R_{123} sejam distintas, isto é, os vértices pertencentes a R_{13} são gêmeos verdadeiros dos contidos em R_{123}, e portanto, devem ser agrupados em uma mesma região (sem perda de generalidade, R_{123}). Isto implica que este quinto caso se reduziu ao terceiro caso apresentado. A Figura 3.7 ilustra esse caso.

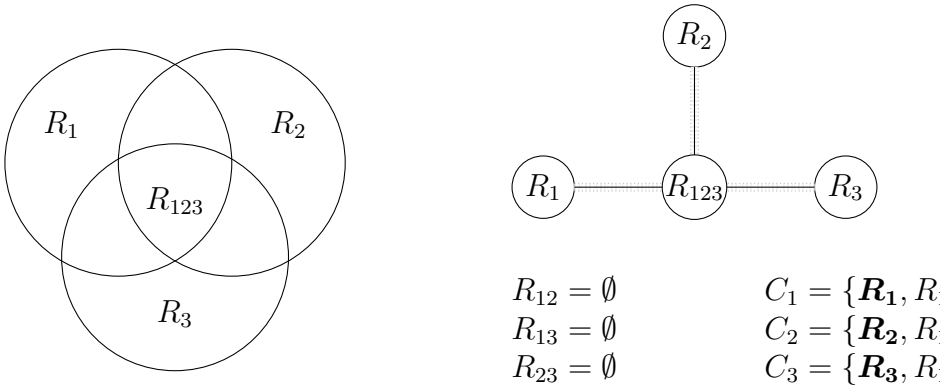
Além dos casos anteriores, existem mais dois formados pela atribuição de vértices as diferentes regiões do grafo. Entretanto, como pode ser visto nas Figuras 3.8 e 3.9, estas atribuições de vértices induzem uma quarta clique maximal, extrapolando o escopo desta subseção. No primeiro caso todas as regiões do grafo contém pelo menos um vértice.



$R_{23} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{12}, R_{123}\}$
$C_2 = \{\boldsymbol{R_2}, R_{12}, R_{23}, R_{123}\}$
$C_3 = \{\boldsymbol{R_3}, R_{23}, R_{123}\}$

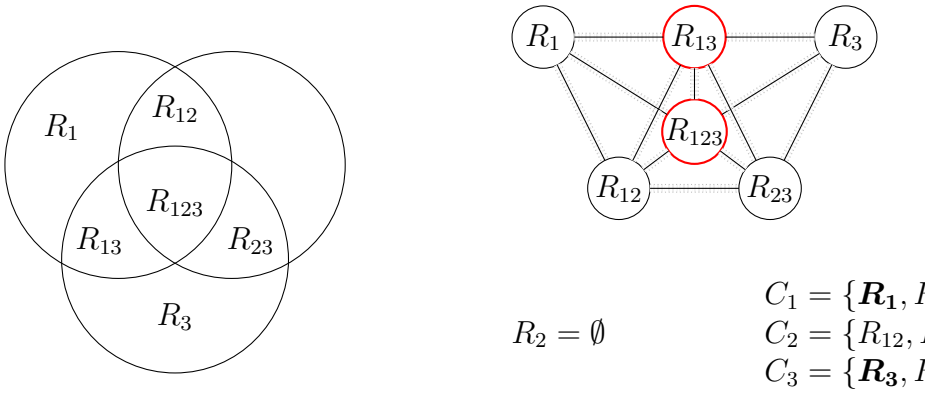
Figura 3.5: Três cliques maximais - caso 4.



$R_{12} = \emptyset$
$R_{13} = \emptyset$
$R_{23} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{123}\}$
$C_2 = \{\boldsymbol{R_2}, R_{123}\}$
$C_3 = \{\boldsymbol{R_3}, R_{123}\}$

Figura 3.6: Três cliques maximais - caso 5.



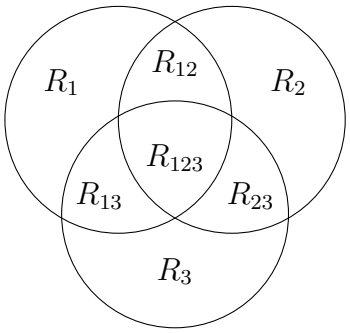
$R_2 = \emptyset$

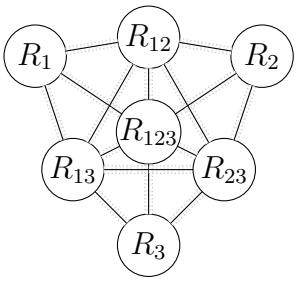
$C_1 = \{\boldsymbol{R_1}, R_{12}, R_{13}, R_{123}\}$
$C_2 = \{R_{12}, R_{23}, R_{23}, R_{123}\}$
$C_3 = \{\boldsymbol{R_3}, R_{13}, R_{23}, R_{123}\}$

Figura 3.7: Três cliques maximais - caso inexistente.

Como pode ser visto na figura, os vértices contidos em R_{12}, R_{13}, R_{23} e R_{123} induzem uma quarta clique maximal, implicando que grafos que atendem esta distribuição de vértices não estão contidos no escopo desta subseção.

No segundo caso (Figura 3.9), os vértices contidos nas regiões R_{12}, R_{23} e R_{13} formam

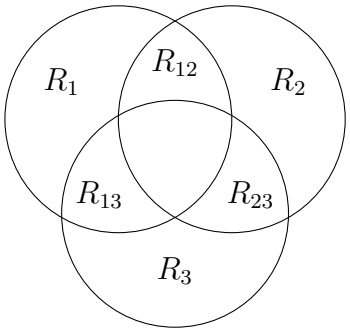


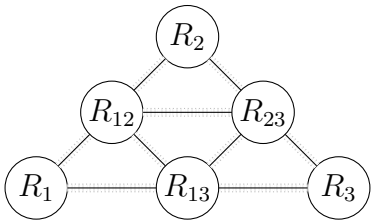


$C_1 = \{\boldsymbol{R_1}, R_{12}, R_{13}, R_{123}\}$
$C_2 = \{\boldsymbol{R_2}, R_{12}, R_{23}, R_{123}\}$
$C_3 = \{\boldsymbol{R_3}, R_{13}, R_{23}, R_{123}\}$
$C_4 = \{R_{12}, R_{13}, R_{23}, R_{123}\}$

Figura 3.8: Exemplos de casos excluídos: Quatro cliques maximais - Caso 1.

uma quarta clique maximal que não está propriamente contida em C_1, C_2 ou C_3. Assim como no caso anterior, tais grafos não estão contidos no escopo desta subseção. Vale destacar que o grafo formado pela atribuição de um único vértice a cada região forma o grafo 3-Tenda, apresentado na Figura 2.18.





$R_{123} = \emptyset$

$C_1 = \{\boldsymbol{R_1}, R_{12}, R_{13}\}$
$C_2 = \{\boldsymbol{R_2}, R_{12}, R_{23}\}$
$C_3 = \{\boldsymbol{R_3}, R_{13}, R_{23}\}$
$C_4 = \{R_{12}, R_{13}, R_{23}\}$

Figura 3.9: Exemplos de casos excluídos: Quatro cliques maximais - Caso 2.

Uma vez que todos os casos possíveis de grafos com três cliques maximais foram apresentados, pode-se estabelecer algumas caracterizações adicionais para essa família de grafos. Essas caracterizações permitem afirmar, conforme será demonstrado no Teorema 3.2.2, que todo grafo com três cliques maximais é um grafo de intervalo. O Lema 3.2.1 demonstra que todo grafo com três cliques maximais é um grafo cordal, enquanto o Lema 3.2.2 estabelece que essa mesma família não admite triplas asteroidais. Uma tripla asteroidal é um tripla de vértices não adjacentes entre si, tal que, existe um caminho entre cada par de vértices que evita a vizinhança do terceiro.

Lema 3.2.1. *Todo grafo com no máximo três cliques maximais é um grafo cordal.*

Demonstração. Conforme demonstrado por Gavril [46], se um grafo admite uma representação como um modelo de interseções de subárvores de uma árvore então esse grafo é cordal. Desta forma, para demonstrar que todo grafo conexo G com no máximo três cliques maximais é cordal é suficiente fornecer um modelo de interseção de subárvores para essa família grafos.

Conforme apresentado na Figura 3.1, um grafo com até três cliques maximais pode ser decomposto em sete regiões, formadas pelas interseções entre as clique maximais. Além disso, é trivial notar que todos os vértices de uma mesma região são gêmeos verdadeiros, o que permite representar todos estes vértices como cópias de uma mesma subárvore no modelo de interseção. Para representar o modelo com maior clareza, todos esses vértices serão agrupados e grafados como uma única subárvore correspondente a cada região.

A Figura 3.10 apresenta um modelo de interseções de subárvores de uma árvore, onde cada região da Figura 3.1 corresponde a uma subárvore.

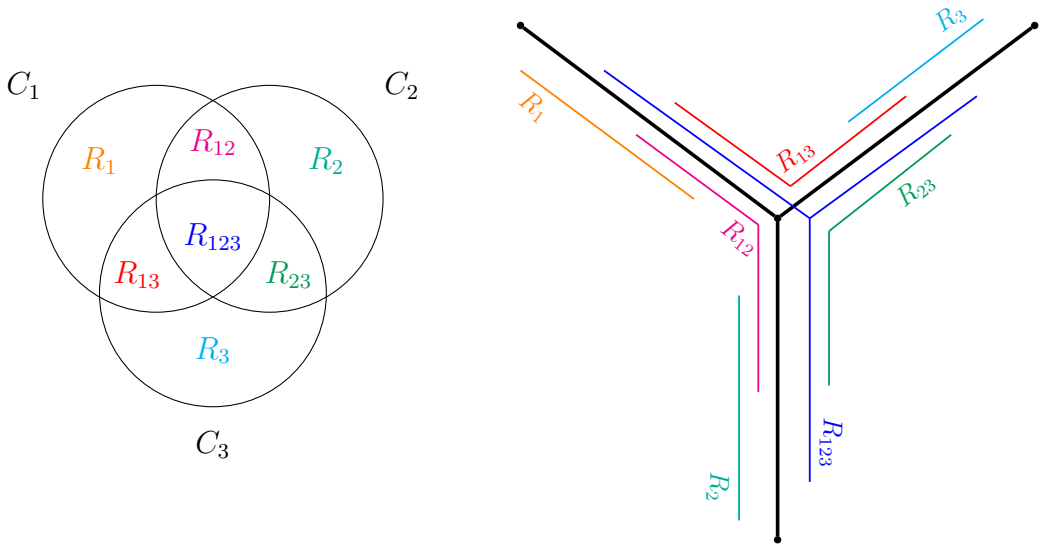


Figura 3.10: Modelo de interseção de subárvores para grafos com até três cliques maximais.

Considerando que a Figura 3.10 apresenta uma representação para qualquer grafo com até três cliques maximais como interseções de subárvores de uma árvore então pode-se concluir que todo grafo com até três cliques maximais é um grafo cordal. □

Lema 3.2.2. *Grafos com três cliques maximais não admitem triplas asteroidais.*

Demonstração. Supondo, por absurdo, que existe um grafo G com três cliques maximais tal que G possui uma tripla asteroidal. Sejam a,b e c os três vértices que constituem uma tripla asteroidal. Como estes três vértices não são vizinhos entre si, eles devem estar condidos em R_1, R_2 e R_3, uma vez que, em qualquer outra região, haveria pelo menos dois vértices adjacentes entre si. Supondo, sem perda de generalidade, $a \in R_1$, $b \in R_2$ e $c \in R_3$. Para estabelecer a tripla asteroidal, são necessários pelo menos mais três vértices:

$\boldsymbol{x \in R_{12}}$ Estabelece o caminho entre a e b, evitando a vizinhança de c ($x \notin N(c)$);

$\boldsymbol{y \in R_{13}}$ Estabelece o caminho entre a e c, evitando a vizinhança de b ($y \notin N(b)$);

$\boldsymbol{z \in R_{23}}$ Estabelece o caminho entre b e c, evitando a vizinhança de a ($z \notin N(a)$).

Contudo, dada a localização destes três vértices nas cliques maximais de G, existem outras restrições que devem ser consideradas:

- Todo vértice em R_{12}, por pertencer à C_1 e à C_2, deve ser vizinho de todos os vértices contidos em R_{13} e em R_{23} ($x \in N(y)$ e $x \in N(z)$);
- Todo vértice em R_{13}, por pertencer à C_1 e à C_3, deve ser vizinho de todos os vértices contidos em R_{12} e em R_{23} ($y \in N(x)$ e $y \in N(z)$);
- Todo vértice em R_{23}, por pertencer à C_2 e à C_3, deve ser vizinho de todos os vértices contidos em R_{12} e em R_{13} ($z \in N(x)$ e $x \in N(y)$).

Essas três condições estabelecem que os vértices que formam os caminhos entre a, b e c, necessários para a caracterização de uma tripla asteroidal, formam uma nova clique maximal, diferente e não propriamente contida em C_1, C_2 ou C_3. Desta forma, fica caracterizada uma contradição: a construção de um grafo com três cliques maximais com a presença de uma tripla asteroidal induz uma quarta clique, contrariando a afirmação inicial de que o grafo possui somente três cliques maximais. A Figura 3.11 ilustra o resultado do processo construtivo empregado e a quarta clique formada, grafada em azul. Sendo assim, fica evidenciado que grafos com somente três cliques maximais não podem conter triplas asteroidais.

□

Os resultados dos Lemas 3.2.1 e 3.2.2 permitem concluir que todo grafo com três cliques maximais é um grafo de intervalo, conforme demonstrado no Teorema 3.2.2.

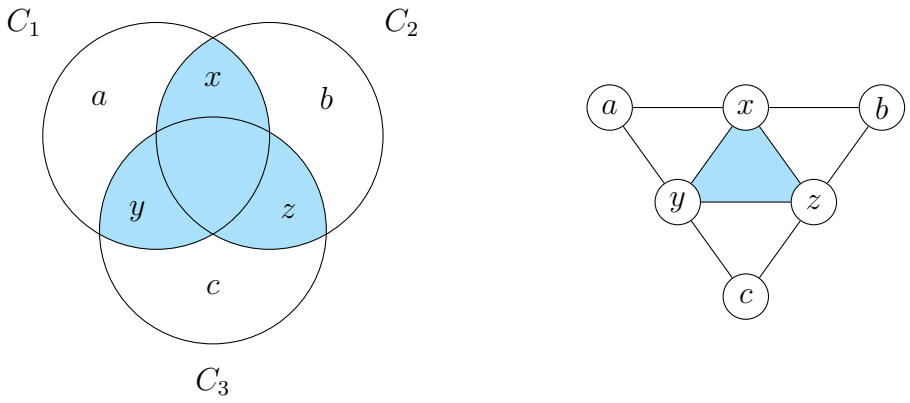


Figura 3.11: Exemplo da construção de uma tripla asteroidal a partir de um grafo com três cliques maximais.

Teorema 3.2.2. *Grafos com três cliques maximais são grafos de intervalo.*

Demonstração. Considerando que todo grafo cordal livre de triplas asteroidais é um grafo de intervalo, a prova segue como resultado imediato dos Lemas 3.2.1 e 3.2.2. □

É interessante notar que, apesar de todo grafo com três cliques maximais ser um grafo de intervalo, não é possível estender esse resultado para a classe dos grafos de intervalo próprio. A Figura 3.6 apresenta um caso em que o grafo construído contém como subgrafo induzido uma garra (grafo com um vértice universal e três vértices simpliciais, tais que os simpliciais são adjacentes somente ao vértice universal), subgrafo este que não admite uma representação como interseções de intervalos sem que um intervalo contenha propriamente um outro.

Em relação a complexidade computacional dos problemas de dominação, o problema do conjunto dominante pode ser trivialmente resolvido quando restrito a classe de grafos conexos com três cliques maximais: nos Casos 3, 4 e 5, um único vértice é suficiente para dominar o grafo, pois os vértices de R_{123} são universais. Nos demais casos, são necessários dois vértices. Sem perda de generalidade, um conjunto composto de um vértice da região R_{12} e um outro da região R_{23} é suficiente para dominar G.

Ao abordar o problema considerando requisitos variáveis, isto é, o problema da dominação vetorial, vale observar os resultados apontados na Seção 3.1. Como todo grafo com três cliques maximais possui *clique-width* constante ($cw = 3$), o algoritmo de tempo $O(|\sigma|(n+1)^{15})$ (σ equivale a uma k-expressão para o grafo) proposto por Cicalese *et al.* [18] pode ser aplicado, resolvendo o problema nesta classe de grafos. Além deste resultado,

vale destacar que o trabalho de Courcelle *et al.* [26] garante a existência de um algoritmo de tempo linear quando os requisitos são limitados por uma constante.

Resta como um problema para estudos futuros uma melhoria na complexidade computacional para grafos com três cliques maximais, baseada no estudo realizado em grafos dominó $\cap$ intervalo próprio. Além disso, outro ponto interessante é a análise do parâmetro *boolean-width* e a árvore de decomposição do grafo, nos moldes do que foi proposto no trabalho proposto em Chiarelli *et al.* [17] para grafos de intervalo próprio.

O estudo de grafos com exatamente três cliques maximais motivou uma análise mais abrangente, onde o grafo possui um número constante ℓ de cliques maximais.

3.2.4 Grafos com ℓ Cliques Maximais

Nesta subseção, serão apresentados resultados a respeito da complexidade computacional do problema da dominação vetorial em grafos conexos com um número ℓ cliques maximais. Deve-se destacar que a classe dos grafos com $\ell > 3$ cliques maximais não está propriamente contida na família dos grafos cordais. Contudo, considerando que um dos objetivos deste trabalho é determinar as fronteiras da $\mathcal{NP}$-Completude do problema da dominação vetorial e que, aparentemente, um dos fatores que pode influenciar na complexidade computacional é justamente o número de cliques, optou-se por incluir no escopo deste trabalho essa classe de grafos.

Como visto nas Subseções 3.2.1, 3.2.2 e 3.2.3, os grafos com uma, duas e três cliques maximais possuem *clique-width* limitada. Considerando que o caso em que o grafo possui ℓ cliques maximais é uma progressão do estudo, uma questão que deve ser analisada é o relacionamento entre o número de cliques e o parâmetro *clique-width*. Para isso, é necessário apresentar uma forma de construir tais grafos usando as operações *clique-width*.

Lema 3.2.3. *Grafos conexos com ℓ cliques maximais podem ser particionados em $2^\ell - 1$ cliques disjuntas.*

Demonstração. Seja $G = (V, E)$ um grafo com ℓ cliques maximais. Então, é possível estabelecer que o número de cliques formadas pelas interseções destas cliques maximais é no máximo:

$$\begin{aligned} ncliques &\leq \sum_{i=1}^{\ell} \binom{\ell}{i} \\ &\leq \sum_{i=1}^{\ell} \frac{\ell!}{i! \cdot (\ell - i)!} \\ &\leq 2^{\ell} - 1 \end{aligned}$$

Entretanto, é importante destacar que algumas destas cliques são conjuntos vazios (algumas devem ser vazias, pois, do contrário, G teria mais de ℓ cliques maximais). Essa observação não impacta o resultado final, pois se trata de um limite máximo que desconsidera o conteúdo de cada conjunto definido. □

Uma vez estabelecido como um grafo desta família pode ser particionado, pode-se demonstrar o parâmetro *clique-width* desta família.

Teorema 3.2.3. *A família dos grafos conexos com ℓ cliques maximais possui clique-width limitada.*

Demonstração. Considerando que:

- Grafos conexos com ℓ cliques maximais podem ser particionados em $2^{\ell} - 1$ cliques disjuntas, de acordo com o Lema 3.2.3;
- Cada clique pode ser construída utilizando dois rótulos, sendo um deles individual para cada clique e o outro, temporário e compartilhado entre as cliques.

Pode-se estabelecer que é possível construir qualquer grafo desta família utilizando no máximo 2^{ℓ} rótulos.

A construção pode ser realizada da seguinte forma:

1. Inicialmente, cada clique C_i é construída usando dois rótulos: L_i e L_{aux}. Ao final, todos os vértices da clique apresentam rótulo L_i (logo L_{aux} pode ser reaproveitado);
2. Em seguida, se duas cliques C_i e C_j estão contidas em uma mesma clique maximal, então todas as arestas entre as cliques são adicionadas, utilizando a operação $\eta_{L_i,L_j}(G)$.

Como foi possível construir o grafo utilizando no máximo 2^{ℓ} rótulos, então o parâmetro

clique-width desta família de grafos é limitado por uma função do número de cliques: $cw \leq 2^{\ell}$. □

Vale observar que o limite estabelecido oferece uma grande margem de folga, pois, como visto para algumas classes de grafos, é possível construir os grafos usando menos rótulos.

Ao demonstrar que os grafos com número limitado de cliques maximais apresentam *clique-width* limitada, é possível aplicar o algoritmo proposto por Cicalese *et al.* [18]. Para grafos com ℓ cliques maximais a complexidade computacional do algoritmo é $O(2^{\ell}\,|\sigma|\,(n+1)^{5\cdot 2^{\ell}})$, onde σ é uma k-expressão.

Além desta observação, ao definir que o parâmetro *clique-width* é limitado em função do número de cliques maximais, pode-se propor outro resultado, que segue como consequência imediata do Teorema de Courcelle e do Teorema 3.1.1 ($MSOL$): O problema da dominação vetorial com requisitos limitados a uma constante k, quando restrito a grafos conexos com ℓ cliques maximais, está em FPT quando parametrizado por ℓ e por k.

Uma vez que a complexidade do problema da dominação vetorial foi estabelecida em grafos com *clique-width* limitada e com número limitado de cliques maximais, é possível prosseguir para o estudo da família dos grafos cordais.

3.3 Família dos Grafos Cordais

Conforme apresentado anteriormente, sabe-se que o problema do conjunto dominante é uma restrição do problema da dominação vetorial no qual todos os vértices possuem requisitos unitários. Desta forma, pode-se demonstrar que o problema da dominação vetorial é possivelmente intratável computacionalmente (supondo $\mathcal{P} \neq \mathcal{NP}$) em uma classe de grafos ao provar que o problema do conjunto dominante é $\mathcal{NP}$-Completo nesta mesma classe.

O Teorema 3.3.1 demonstra que o problema do conjunto dominante permanece $\mathcal{NP}$-Completo, mesmo quando restrito à classe dos grafos cordais.

Teorema 3.3.1 (Booth [7])**.** *O problema do conjunto dominante em grafos cordais é $\mathcal{NP}$-Completo.*

Demonstração. Seja $G = (V, E)$ um grafo arbitrário e seja k um inteiro positivo qualquer.

Será demonstrado que o Problema da Cobertura de Vértices para G e k pode ser polinomialmente reduzido para o problema do conjunto dominante para G' e k, onde G' é um grafo cordal construído a partir de G.

Seja $G' = (V', E')$ o grafo que será construído, tendo como base o grafo $G = (V, E)$. O conjunto V' é formado pelo conjunto de vértices V, acrescido de um vértice para cada aresta de G. Os vértices contidos em V são denominados V-vértices e aqueles associados as arestas de G, E-vértices. A construção do conjunto de arestas E' de G' é feito em dois momentos: inicialmente, são adicionadas arestas entre cada par de V-vértices, formando uma clique; Em seguida, para cada E-vértice (atrelado a uma aresta de G) são adicionadas duas arestas, ligando o E-vértice aos dois V-vértices correspondentes aos extremos desta aresta em G. A Figura 3.12 apresenta o resultado da construção do grafo G' a partir do grafo G.

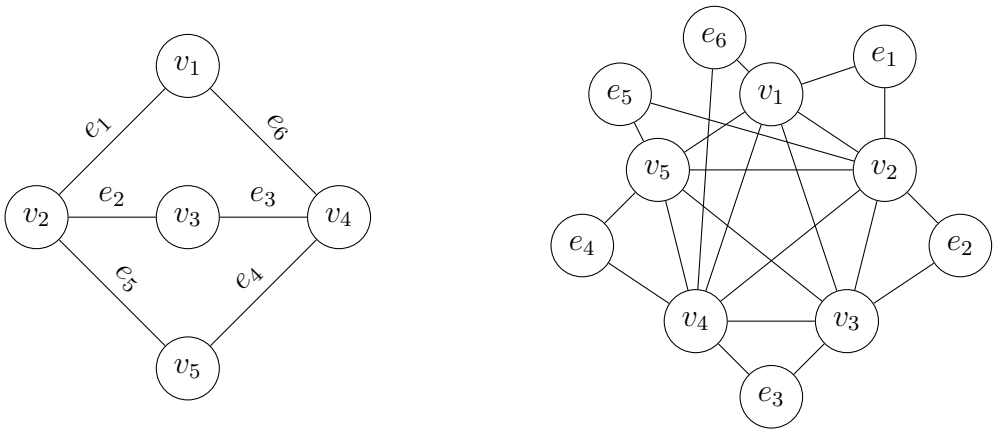


Figura 3.12: O grafo G e o cordal resultante G'.

Será provado que G possui uma cobertura de vértices de tamanho k se, e somente se, G' possuir um conjunto dominante de cardinalidade k. O problema da cobertura de vértices busca determinar se existe um conjunto de vértices de tamanho k tal que toda aresta do grafo possua pelo menos um extremo neste conjunto. O problema da cobertura de vértices é reconhecidamente $\mathcal{NP}$-Completo para grafos em geral ([45]).

Dada a forma com que G' foi construído, pode-se assegurar que toda cobertura de vértices para G é um conjunto dominante para G'. Isso ocorre pois os V-vértices em G' formam uma clique e, desta forma, são trivialmente dominados por qualquer vértice da cobertura. Além disso, considerando que cada E-vértice de G' corresponde a uma aresta de G e que todas as arestas em G possuem um extremo na cobertura de vértices, nota-se que todo E-vértice possui pelo menos um vizinho neste conjunto, implicando que a cobertura

de vértices é suficiente para dominar os E-vértices.

No sentido contrário, dado um conjunto dominante para G' pode-se assumir, sem perda de generalidade, que ele não contém E-vértices, pois bastaria trocar cada vértice deste tipo por outro V-vértice adjacente a ele, mantendo o conjunto dominante. Sendo assim, seja S um conjunto capaz de dominar todo os vértices do grafo tal que S não contém E-vértices. Como todo E-vértice é dominado por S, pode-se concluir que toda aresta do grafo G possui pelo menos um extremo em S: cada E-vértice possui pelo menos um vizinho em S. Desta forma, S é uma cobertura de vértices para G.

A prova é finalizada ao se notar que G' é cordal, uma vez que os V-vértices constituem uma clique e cada E-vértice é adjacente a apenas dois V-vértices, também adjacentes entre si. Esta configuração impossibilita a criação de ciclos com mais de três vértices sem cordas, pois a clique contém todas as arestas possíveis e os E-vértices formam ciclos de tamanho três. Adicionalmente, um esquema de eliminação perfeito para G' pode ser trivialmente obtido, removendo-se, primeiramente, os E-vértices, pois são simpliciais (cada E-vértice pertence a apenas uma clique formada por ele e dois V-vértices) e, em seguida, os V-vértices. □

Uma vez que foi provado que o problema do conjunto dominante é $\mathcal{NP}$-Completo para grafos cordais, pode-se afirmar que o problema da dominação vetorial também é intratável computacionalmente (supondo $\mathcal{P} \neq \mathcal{NP}$).

É interessante notar que a prova apresentada para grafos cordais também demonstra a $\mathcal{NP}$-Completude do problema para grafos *split*, uma vez que os V-vértices constituem uma clique e os E-vértices, um conjunto independente. A Subseção 3.3.7 retomará essa demonstração, além de apresentar ainda um outra prova a respeito deste tema.

Nas próximas subseções, algumas subfamílias de grafos cordais serão analisadas para identificar os limiares da complexidade computacional, isto é, para quais o problema permanece $\mathcal{NP}$-Completo e para quais outras ele passa a admitir algoritmos polinomiais.

A Figura 3.13 ilustra as relações entre as famílias de grafos abordadas neste trabalho. Esta figura permite visualizar uma relação de continência entre os grafos da família dos cordais. A hierarquia apresentada foi baseada no modelo apresentado por Markenzon e Waga [70]. Nesta figura, o problema da dominação vetorial é indicado por quatro cores: as classes onde o problema é $\mathcal{NP}$-Completo são hachuradas em vermelho, enquanto as classes em amarelo admitem algoritmos de tempo polinomial. Já as classes marcadas em

verde admitem algoritmos de tempo linear. Para os casos onde a complexidade ainda é desconhecida, as classes são hachuradas na cor azul. Existem classes cuja complexidade computacional depende dos requisitos, como, por exemplo, os *directed path graphs* onde o problema do conjunto dominante ($R[v] = 1\ \forall v \in V$) admite algoritmo de tempo linear enquanto a complexidade do problema da dominação vetorial permanece em aberto. Para estes casos, as classes foram apresentadas como um degradê de cores, onde a cor do lado esquerdo representa a complexidade do problema do conjunto dominante, enquanto a cor do lado direito, a complexidade do problema da dominação vetorial. É importante notar que todas as classes que apresentam *clique-width* limitada por uma constante k admitem algoritmos de tempo polinomial [18], conforme apresentado na Seção 3.1. Os resultados obtidos por este trabalho foram destacados na figura através de bordas duplas e serão apresentados no Capítulo 4.

A Figura 3.13 permite uma visualização rápida do estado da arte da complexidade computacional de algumas subclasses de grafos cordais, pois esta decorre da continência entre as famílias. Isto é, ao demonstrar que o problema é $\mathcal{NP}$-Completo para uma determinada família de grafos, implica-se também que o problema segue $\mathcal{NP}$-Completo para todas as famílias que a contém propriamente. Entretanto, a relação inversa não é necessariamente verdadeira, pois as restrições adicionais que definem as subfamílias podem tornar o problema tratável computacionalmente. Por outro lado, a identificação de um algoritmo capaz de resolver o problema para uma família em tempo polinomial, implica necessariamente que o problema segue tratável para todas as subfamílias contidas nesta primeira. Um dos objetivos deste trabalho é identificar os limiares da complexidade computacional na família dos grafos cordais, o que é representado neste diagrama como uma aresta partindo de uma classe onde o problema é $\mathcal{NP}$-Completo para outra onde ele admite um algoritmo de tempo polinomial (ou mesmo linear), como, por exemplo, os grafos *split* e *split*-indiferença.

Uma vez que a complexidade computacional do problema da dominação vetorial em grafos cordais foi estabelecida, é possível avançar o estudo em subclasses de grafos cordais. A primeira classe estudada será a dos *undirected path graphs*. Como visto no Capítulo 2, essa classe é definida por uma restrição no modelo de interseções: enquanto grafos cordais admitem um modelo de subárvores em uma árvore não direcionada, os *undirected path graphs* restringem o modelo a interseção de caminhos não direcionados em uma árvore não direcionada.

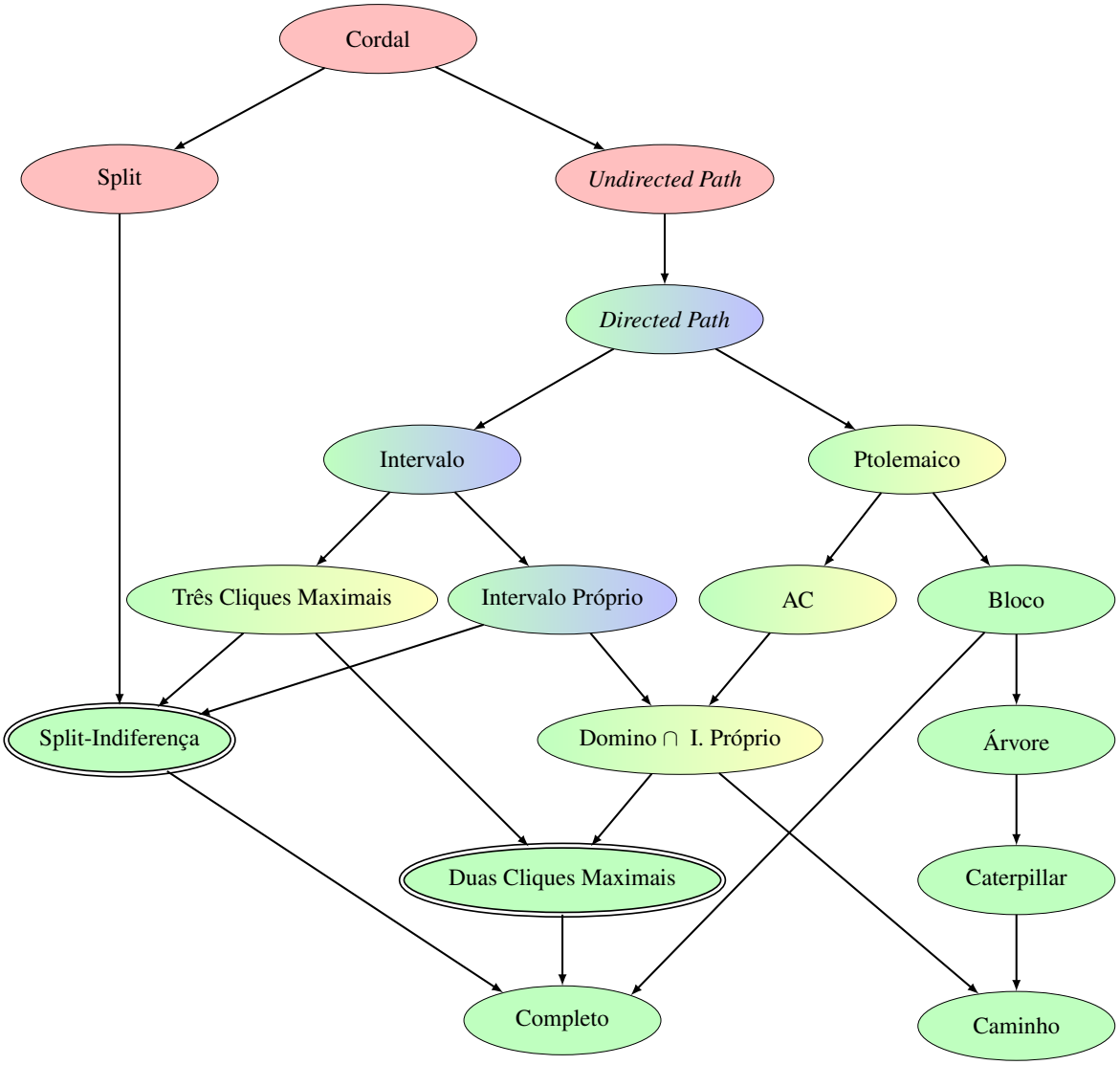


Figura 3.13: Estado da arte da complexidade do problemas de dominação em grafos cordais.

3.3.1 *Undirected Path Graphs*

O trabalho de Booth e Johnson [8] também demonstra que o problema do conjunto dominante continua $\mathcal{NP}$-Completo nesta família de grafos. Esta demonstração consiste em estabelecer uma redução polinomial do problema *3-dimensional matching* em uma instância do problema do conjunto dominante em *undirected path graphs*. Como o problema *3-dimensional matching* é reconhecidamente $\mathcal{NP}$-Completo Garey e Johnson [45], pode-se concluir que o problema do conjunto dominante nesta família de grafos também é intratável computacionalmente (supondo $\mathcal{P} \neq \mathcal{NP}$).

A próxima classe abordada é definida por uma segunda restrição no modelo de interseção. Enquanto nos *undirected path graphs* os caminhos não eram direcionados, nos *directed path graphs* os caminhos são direcionados. Além disso, a árvore subjacente passa a ter uma raiz e arestas direcionadas. Como será visto na Subseção 3.3.2, essa restrição possibilitou que o problema do conjunto dominante se tornasse tratável nessa nova classe.

3.3.2 *Directed Path Graphs*

Nesta seção serão apresentados o algoritmo proposto por Booth e Johnson [8] para o problema do conjunto dominante em *directed path graphs* e uma modificação deste mesmo algoritmo para solucionar o problema da dominação vetorial com requisitos 0 e 1.

O Algoritmo 3.2, destinado ao problema do conjunto dominante, foi proposto por Booth e Johnson [8]. Seu funcionamento está baseado em primeiramente obter a árvore de cliques do grafo e, em seguida, percorrê-la em pós-ordem, processando, a cada iteração, um nó desta árvore. Além disso, o algoritmo utiliza dois rótulos para guiar a escolha dos vértices que farão parte do conjunto dominante: $depth[C]$ e $high[v]$. O primeiro rótulo representa o tamanho do caminho entre a raiz da árvore de cliques e um nó C. Já o segundo, aplicado a cada vértice v do grafo, retrata o menor de valor de $depth$ dentre todos os nós da árvore que contém v.

Após a iteração sobre os nós filhos de um nó C da árvore de cliques (comportamento padrão do percurso em pós-ordem), o nó C deve ser explorado. Para isso, o algoritmo busca dentre todos os vértices $v \in C$, um vértice v não dominado tal que $high[v] = depth[C]$. Para um vértice $v \in C$, a condição $high[v] = depth[C]$ implica que C é a última clique no percurso em pós-ordem pela árvore de cliques em que v está contido. Isto é, v não pertence a nenhuma clique situada acima de C na árvore de cliques, o que também implica que esta

iteração é o último momento possível para assegurar a dominação de v durante a execução do algoritmo, uma vez que v não será processado novamente.

A dominação do vértice v é assegurada ao adicionar um vizinho u de v ao conjunto R-dominante. Considerando que o vértice u deve estar obrigatoriamente contido em todas as cliques situadas no caminho entre quaisquer duas cliques que o contenham na árvore de cliques, ao se escolher um vértice u tal que o valor de $high[u]$ seja o menor possível, assegura-se que a inclusão de u ao conjunto R-dominante maximiza o número de vértices dominados (todos os vértices situados em cliques que contenham u serão dominados pela inclusão de u no conjunto R-dominante). Sendo assim, o vértice u é adicionado ao conjunto R-dominante e todo vértice $w \in N[u]$ é marcado como dominado, assegurando a dominação de v e também de todos os vértices em C.

É interessante observar que no algoritmo descrito por Booth e Johnson [8], a iteração pela árvore de cliques é descrita como um percurso em pré-ordem. Entretanto, este percurso não atende ao comportamento retratado, uma vez que o correto funcionamento do algoritmo depende de uma exploração partindo das folhas da árvore em direção a raiz. Desta forma, o algoritmo correto é um percurso em pós-ordem.

Algoritmo 3.2: *Directed Path Graphs*: Conjunto Dominante

Entrada: *Directed path graph* $G = (V, E)$
Saída: Conjunto dominante mínimo S

```
1  S ← ∅
2  Obter árvore de cliques T de G
3  Seja r a raiz de T
4  Calcular depth de cada nó da árvore T
5  Calcular high para cada vértice v ∈ G
6  para cada v ∈ V faça Dom[v] ← Falso ;
7  PosOrdem (T, r, depth, high, dom, S)
8  retorne S

1  Função PosOrdem (T, C, depth, high, dom, S)
2      para cada C' ∈ Filhos(C) faça
3          PosOrdem (T, C', depth, high, S)
4      se ∃ v ∈ C | Dom[v] = Falso ∧ high[v] = depth[C] então
5          Seja u o vértice contido em C de menor high[u]
6          S ← S ∪ {u}
7          para cada w ∈ N[u] faça
8              Dom[w] ← Verdadeiro
9      retorne S
```

A Figura 3.14 ilustra um exemplo da aplicação deste algoritmo. No exemplo, as linhas marcadas com "Resta dominar" correspondem aos vértices que atendem as condições da linha 4 da função `PosOrdem`. Os vértices que não possuem $high$ igual ao $depth$ da clique são omitidos nos exemplos.

O algoritmo de percurso em pós-ordem não determina nenhum critério de desempate entre os filhos de uma clique. Desta forma, o algoritmo pode ser iniciado pelas cliques C_5, C_3, C_4 e C_8. Entretanto, como o resultado é o mesmo em todas as opções, a clique C_8 foi escolhida para ser processada primeiro.

Durante o processamento de C_8, observa-e que o vértices 12 apresenta $high(12) = depth(C_8)$, implicando que deve ser dominado nesta iteração. Dado o critério de seleção, o vértice de menor $high$ dentre os adjacentes é o vértice 9, que é adicionado ao conjunto em construção. As próximas cliques quem deve ser processada são C_7 e C_6, entretanto todos os vértices destas cliques já estão dominados.

Em seguida, o algoritmo processará a clique C_4: o vértice 7 possui $high(7) = depth(C_4)$ e, portanto, deve ser dominado nesta iteração. Para dominar o vértice 7, o algoritmo selecionará o vizinho de menor $high$, isto é, o vértice 3.

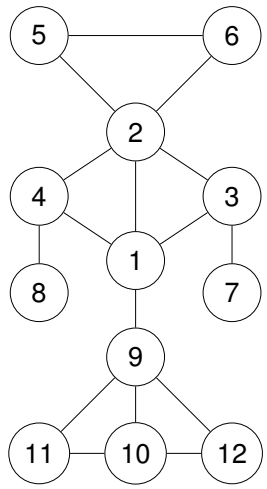
A próxima clique é C_3, onde se nota que os vértices $v \in \{5, 6\}$ possuem $high(v) = depth(C_3)$. Para dominar estes vértices, dado o critério empregado, o algoritmo selecionará o vértice 2.

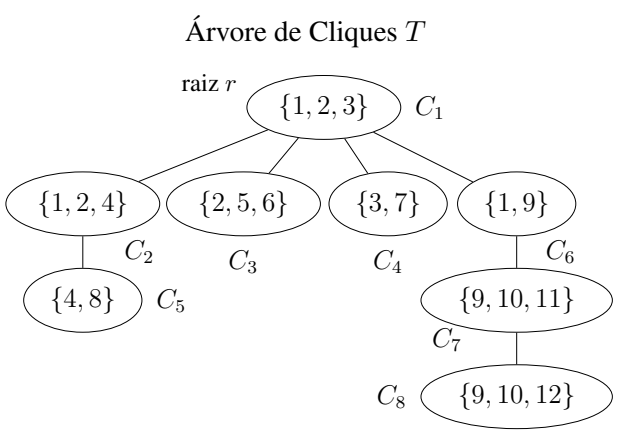
Resta processar as cliques C_5, C_2 e C_3. Em C_5, nota-se que o vértice 8 não está dominado e possui $high(8) = depth(C_5)$. Novamente, o algoritmo procederá com a adição de um vértice para assegurar a dominação deste vértice: o vértice 4 é escolhido e adicionado ao conjunto. Finalmente, considerando que todos os vértices do grafo já foram dominados, é seguro afirmar que o processamento de C_2 e C_1 é trivial. O algoritmo retornará o conjunto dominante $\{2, 3, 4, 9\}$ para G.

A corretude do Algoritmo 3.2, formulada como o Teorema 3.3.2, foi demonstrada por Booth e Johnson [8].

Teorema 3.3.2 (Booth e Johnson [8])**.** *O Algoritmo 3.2 encontra um conjunto dominante mínimo para directed path graphs.*

Considerando que a obtenção de uma árvore de cliques pode ser realizada em tempo linear através de modificações no algoritmo proposto por Dietz *et al.* [31] e que cada nó dessa árvore é percorrido apenas uma vez, pode-se afirmar que o Algoritmo 3.2 é linear.





$EEP(G) = (12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$

v	$high$
1	0
2	0
3	0
4	1
5	1
6	1
7	1
8	2
9	1
10	2
11	2
12	3

$C_8 = \{9, 10, 12\}$
$high(12) = depth(C_8)$
Resta dominar: 12
$u = 9$
$S = \{9\}$

$C_7 = \{9, 10, 11\}$
10 e 11 já dominados

$C_6 = \{1, 9\}$
9 já dominado

$C_4 = \{3, 7\}$
$high(7) = depth(C_4)$
Resta dominar: 7
$u = 3$
$S = \{9, 3\}$

$C_3 = \{2, 5, 6\}$
$high(5) = depth(C_3)$
$high(6) = depth(C_3)$
Resta dominar: 5 e 6
$u = 2$
$S = \{9, 3, 2\}$

$C_5 = \{4, 8\}$
$high(8) = depth(C_5)$
$u = 4$
$S = \{9, 3, 2, 4\}$

$C_2 = \{1, 2, 4\}$
4 já dominado

$C_1 = \{1, 2, 3\}$
1, 2 e 3 já dominados
$\mathbf{S} = \{\mathbf{9}, \mathbf{3}, \mathbf{2}, \mathbf{4}\}$

Figura 3.14: Exemplo da aplicação do Algoritmo 3.2.

Vale destacar a dicotomia encontrada ao comparar a complexidade computacional do problema do conjunto dominante nas classes *undirected path graphs* e *directed path graphs*, uma vez que na primeira o problema é $\mathcal{NP}$-Completo, enquanto na segunda ele passa a admitir um algoritmo de tempo linear. Como pode ser observado, a corretude deste algoritmo decorre da propriedade apresentada no Teorema 2.3.2, onde foi demonstrado que todo que, para todo vértice $x \in V(G)$, o conjunto das cliques maximais que contém x induzem um caminho direcionado na árvore de cliques. Essa propriedade assegura que a escolha do vértice u na linha 5 da função POSORDEM no Algoritmo 3.2 é ótima e, portanto, conduz a um conjunto dominante mínimo. Finalmente, considerando que esta propriedade existe apenas na família dos *directed path graphs* (e suas subfamílias), pode-se estabelecer esta como a primeira dicotomia justificada neste estudo.

O algoritmo proposto para o problema do conjunto dominante pode ser modificado para resolver também uma versão mais restrita do problema da dominação vetorial onde os requisitos dos vértices estão limitados a 0 e 1, isto é, $R[v] \in \{0, 1\}, \forall v \in V$. O Algoritmo 3.3 apresenta o algoritmo modificado.

Algoritmo 3.3: *Directed Path Graphs*: Dominação Vetorial com Requisitos 0 e 1

Entrada: *Directed Path Graph* $G = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

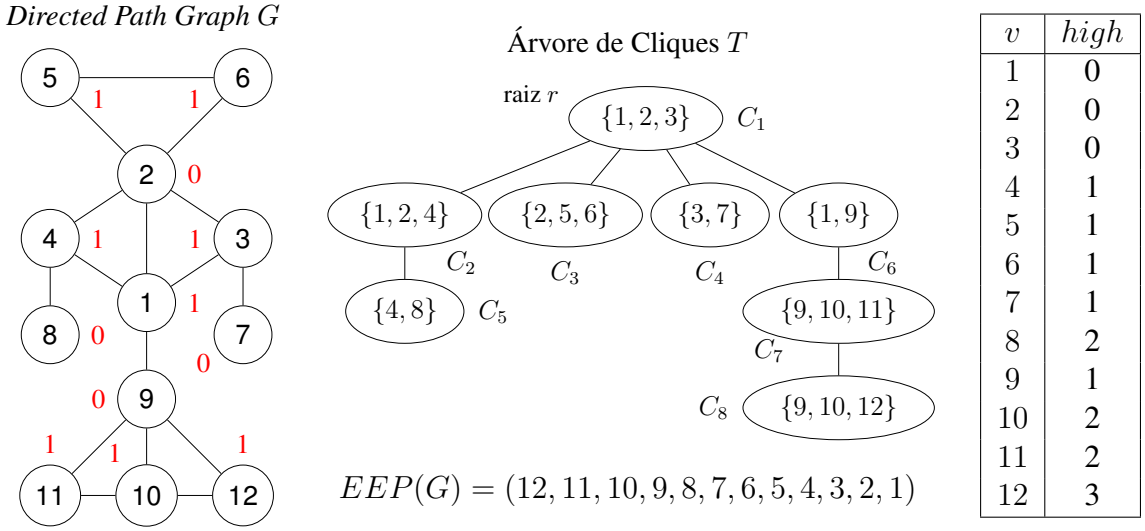
```
1  S ← ∅
2  Obter árvore de cliques T de G
3  Seja r a raiz de T
4  Calcular depth de cada nó da árvore T
5  Calcular high para cada vértice v ∈ G
6  para cada v ∈ V faça
7      se R[v] = 0 então Dom[v] ← Verdadeiro ;
8      Dom[v] ← Falso
9  PosOrdem (T, r, depth, high, dom, S)
10 retorne S

1  Função PosOrdem (T, C, depth, high, dom, S)
2      para cada C' ∈ Filhos(C) faça
3          PosOrdem (T, C', depth, high, S)
4      se ∃ v ∈ C | dom[v] = Falso ∧ high[v] = depth[C] então
5          Seja u o vértice contido em C de menor high[u]
6          S ← S ∪ {u}
7          para cada w ∈ N[u] faça
8              Dom[w] ← Verdadeiro
9      retorne S
```

Teorema 3.3.3. *O Algoritmo 3.3 encontra um conjunto R-dominante quando os requisitos são restritos a 0 e 1 e o grafo pertence à classe dos directed path graphs.*

Demonstração. Vértices que possuem requisito 0 são peculiares, pois não necessitam de nenhum vizinho para serem dominados, contudo podem ser utilizados para assegurar a dominação de seus adjacentes quando são adicionados ao conjunto R-dominante. Considerando que os vértices de requisitos 0 são marcados como dominados (linhas 7-9 do algoritmo) eles não demandarão que qualquer vizinho seja adicionado ao conjunto em construção para sua dominação. Além disso, qualquer vértice de requisito 0 pode ser utilizado para dominar um vértice, caso este seja o vizinho de menor *high*, o que não altera a corretude do método original.

Considerando que o Algoritmo 3.2 é correto de acordo com o Teorema 3.3.2 e que os vértices de requisito 0 podem ser escolhidos para dominar outros vértices, ao mesmo tempo, em que não demandarão que nenhum vértice seja adicionado ao conjunto em construção para sua dominação, pode-se concluir que o Algoritmo 3.3 também está correto. □

A Figura 3.15 ilustra um exemplo da execução do Algoritmo 3.3. Este exemplo apresenta o mesmo grafo da Figura 3.14 para permitir uma comparação do comportamento do algoritmo ao encontrar vértices de requisito 0, que, por definição, não demandam nenhum vizinho para sua dominação. Entretanto, como pode ser visto, estes vértices foram utilizados para compor uma solução mínima. Os requisitos de cada vértice são representados pelos números em vermelho ao lado de cada vértice.



v	$high$
1	0
2	0
3	0
4	1
5	1
6	1
7	1
8	2
9	1
10	2
11	2
12	3

$C_8 = \{9, 10, 12\}$
$high(12) = depth(C_8)$
Resta dominar: 12
$u = 9$
$S = \{9\}$

$C_7 = \{9, 10, 11\}$
10 e 11 já dominados

$C_6 = \{1, 9\}$
9 já dominado

$C_4 = \{3, 7\}$
$R[7] = 0$
7 já dominado

$C_3 = \{2, 5, 6\}$
$high(5) = depth(C_3)$
$high(6) = depth(C_3)$
Resta dominar: 5 e 6
$u = 2$
$S = \{9, 2\}$

$C_5 = \{4, 8\}$
$R[8] = 0$
8 já dominado

$C_2 = \{1, 2, 4\}$
4 já dominado

$C_1 = \{1, 2, 3\}$
1, 2 e 3 já dominados
$\mathbf{S = \{9, 2\}}$

Figura 3.15: Exemplo da aplicação do Algoritmo 3.3.

É importante destacar que não foi possível modificar esse algoritmo para atender re-

quisitos maiores ou iguais a 2, pois o critério de selecionar vértices situados mais próximos à raiz da árvore de cliques (menor $high$ possível dentro da clique) pode resultar em escolhas ruins. A Figura 3.16 ilustra um exemplo onde esse critério induz uma escolha ruim. Na figura, ao processar a clique $C_4 = \{1, 2, 3, 4\}$, o algoritmo deve escolher dois vizinhos do vértice 1 de menor $high$ ($R[1] = 2$), o que implica na escolha dos vértices 3 e 4. O processamento da clique $C_3 = \{3, 4, 5, 6\}$ é trivial, pois todos os vértice já estão dominados. A próxima clique $C_2 = \{5, 6, 7\}$ não contém vértices não dominados com $high(v) = depth(C_2)$ ($high(7) = 0$ e $depth(C_2) = 1$). Finalmente, ao processar a clique $C_1 = \{7, 8\}$ nota-se que o vértice 7 ainda não foi dominado e atende ao critério $high = depth$, implicando na adição do vértice 7 para dominar a clique. Desta forma, o algoritmo retornará o conjunto $\{3, 4, 7\}$. Este conjunto, embora seja suficiente para dominar o grafo, não é ótimo, uma vez que a solução ótima para esta instância é o conjunto $\{1, 5\}$.

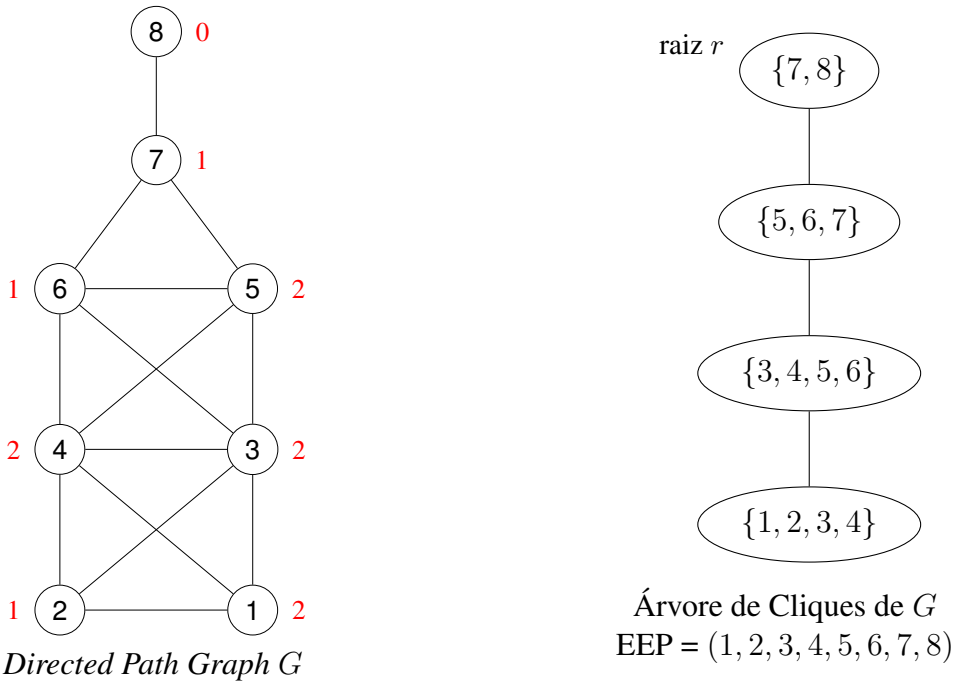


Figura 3.16: Contra-exemplo do algoritmo para *directed path graphs* com requisitos $\{0, 1, 2\}$.

É interessante destacar que o contra-exemplo apresentado na Figura 3.16 foi obtido pelo ambiente de pré-validação, apresentado no Apêndice A. O ambiente foi utilizado para construir grafos de intervalo próprio (que por definição são *directed path graphs*) e buscar por combinações de requisitos onde o algoritmo não foi capaz de encontrar a solução ótima.

Em relação ao problema da dominação vetorial sem restrições quanto aos requisitos, até o momento não se conhece nenhum algoritmo polinomial ao mesmo tempo em que

não existem provas que demonstrem a sua $\mathcal{NP}$-Completude. Desta forma, este problema permanece em aberto para estudos futuros.

Na Subseção 3.3.3, uma outra classe de grafos será abordada: grafos ptolemaicos. Esta família de grafos apresenta um ponto de interesse em relação ao problema da dominação vetorial, pois, como será visto, ela introduz uma fronteira na complexidade computacional.

3.3.3 Grafos Ptolemaicos e Grafos AC

Como mencionado anteriormente, a caracterização de que todo grafo ptolemaico também é um grafo de distância hereditária é uma importante ferramenta para estabelecer a complexidade computacional do problema da dominação vetorial nesta família de grafos. O trabalho de Golumbic e Rotics [50] demonstrou que todo grafo de distância hereditária possui *clique-width* menor ou igual a três ($cw \leq 3$). Desta forma, é correto afirmar que todo grafo ptolemaico também possui *clique-width* limitado.

Tendo em vista que o trabalho de Cicalese *et al.* [18] descreve um algoritmo de tempo polinomial para o problema da (λ, β, α)-SELEÇÃO DE ALVOS em grafos com *clique-width* limitada e que o problema da dominação vetorial pode ser interpretado como uma restrição deste problema onde $\alpha = n$, $\beta = k$ e $\lambda = 1$, pode-se concluir que o algoritmo proposto por Cicalese *et al.* [18] pode ser aplicado nesta família de grafos. Este algoritmo tem complexidade de tempo da ordem de $O(|\sigma|(n+1)^{15})$, onde σ equivale a uma 3-expressão (k-expressão onde $k \leq 3$) para o grafo. Como o trabalho de Golumbic e Rotics [50] demonstra que uma 3-expressão pode ser obtida em tempo $O(n+m)$ para os grafos de distância hereditária, é correto afirmar que para todo grafo ptolemaico o problema da dominação vetorial admite algoritmo de tempo polinomial (a complexidade da obtenção da k-expressão é inferior a complexidade do algoritmo). Entretanto, como será visto ao longo desta trabalho, existem algumas subfamílias de grafos ptolemaicos que admitem algoritmos de tempo linear.

Vale destacar que ao definir a complexidade computacional do problema da dominação vetorial na classe dos grafos ptolemaicos, uma fronteira da intratabilidade do problema também foi identificada: até o momento a complexidade do problema na classe dos *directed path graphs* segue em aberto, enquanto para uma de suas subclasses (grafos ptolemaicos) o problema admite um algoritmo de tempo polinomial. Conforme demonstrado, essa dicotomia está intrinsecamente relacionada ao parâmetro *clique-width* da família, pois, conforme demonstrado pelo trabalho de Cicalese *et al.* [18], toda família de grafo com

clique-width limitado admite um algoritmo de tempo polinomial.

Como demonstrado no Capítulo 2, os grafos AC são definidos pela interseção dos grafos dominó e dos grafos cordais. O trabalho de Markenzon e Waga [69] demostrou que a classe dos grafos AC está propriamente contida na classe dos grafos ptolemaicos. Sendo assim, é possível estender os resultados apontados nesta seção sobre a complexidade computacional do problema da dominação vetorial em grafos ptolemaicos para os grafos AC, ou seja, o problema da dominação vetorial admite um algoritmo polinomial nesta classe.

Ainda neste sentido, é interessante destacar que para uma segunda subfamília de grafos também contida na classe dos *directed path graphs*, os grafos de intervalo, a solução apresentada não é válida, uma vez que, segundo Golumbic e Rotics [50], os grafos de intervalo não possuem *clique-width* limitada (o trabalho de Golumbic e Rotics [50] também demonstra esta afirmação para os grafos de intervalo próprio, entretanto, dada a relação entre as classes, é possível extrapolar essa mesma conclusão para os grafos de intervalo). Os grafos de intervalo e de intervalo próprio serão abordados na Subseção 3.3.6.

3.3.4 Grafos Bloco

O trabalho de Lan e Chang [60] propõe um algoritmo baseado na aplicação de dois rótulos associados aos vértices. O primeiro rótulo, $L_1(v)$, determina se o vértice v é requerido obrigatoriamente (R) ou não (B) para se obter um conjunto R-dominante mínimo. Já o segundo rótulo, $L_2(v)$, é utilizado para indicar quantos vizinhos de v devem ser incluídos neste conjunto R-dominante para que v seja dominado. Inicialmente, este segundo rótulo equivale ao vetor de requisitos. Estes dois rótulos, para melhor representação, serão agrupados em uma tupla $L(v) = (L_1(v), L_2(v))$.

O primeiro passo do algoritmo consiste em inicializar os rótulos de cada vértice v: $L(v) = (B, R[v])$. Em seguida, a cada iteração, um bloco folha do grafo é analisado. Um bloco folha B de um grafo bloco G consiste de uma clique, tal que no máximo um de seus vértices é uma articulação em G.

Sempre que um bloco folha é tratado pelo algoritmo, dois casos distintos devem ser considerados. Seja u a única articulação deste bloco:

Caso 1 – $\mathbf{L_1(u) = B}$: Neste caso, a articulação não foi marcada como requerida. Logo ela pode participar ou não da solução do bloco. Sendo assim, este caso deve analisar

duas hipóteses distintas: $L'_2(u) = 0$ e $L'_1(u) = R$. Na primeira, o algoritmo já considerará a articulação dominada, implicando que ela não será incluída no conjunto R-dominante deste bloco. Na segunda hipótese, a articulação será considerada como requerida e obrigatoriamente participará da solução do bloco. Para cada hipótese, o algoritmo obterá um conjunto R-dominante para o bloco e analisará qual dos conjuntos possui a menor cardinalidade. Em seguida, o algoritmo selecionará o conjunto escolhido para compor a solução final para o grafo. Uma vez que um conjunto foi escolhido, o algoritmo atualizará os rótulos da articulação de acordo com a melhor hipótese. Na primeira, o rótulo $L_2(u)$ deve ser decrementado do número de vizinhos da articulação que pertencem ao conjunto R-dominante do bloco. Já na segunda, o rótulo $L_1(u)$ deve ser marcado como requerido para as demais iterações.

Caso 2 – $\mathbf{L_1(u) = R}$: Já neste caso, o algoritmo detectou que a articulação foi marcada como requerida em alguma iteração anterior. Este bloco é resolvido e a solução obtida (que deve incluir u) é incorporada ao conjunto R-dominante para o grafo.

Sempre que um bloco folha é solucionado, ele é removido do grafo. Este processo deve ser repetido até que o grafo se torne biconexo. Uma vez que isso ocorra, o grafo foi reduzido a apenas uma clique, que pode ser facilmente solucionada. O conjunto R-dominante consiste das soluções obtidas para os blocos folha unidas à solução encontrada para esta última clique.

Deve-se destacar que nas linhas 13 e 22 do Algoritmo 3.4 o vértice u, que corresponde à articulação do bloco folha, foi removido do conjunto R-dominante deste bloco no momento de sua incorporação ao conjunto R-dominante do grafo, pois esse vértice será incluído no momento do processamento de outro bloco, onde u não é mais uma articulação. Essa adição é assegurada, pois $L_1(u)$ foi marcado como requerido ($L_1(u) = R$).

O algoritmo apresentado faz uso de uma função denominada *DominarClique*, responsável por solucionar cada bloco folha (clique). Seu funcionamento é baseado no Algoritmo 3.1, diferindo apenas por uma etapa de pré-processamento, onde os vértices marcados como requeridos são imediatamente incluídos no conjunto em construção.

Considerando que é possível determinar se o grafo é biconexo e obter blocos folha em tempo linear, o algoritmo apresentado possui complexidade $O(n)$, uma vez que, a cada iteração, um bloco folha é removido.

A próxima seção abordará um caso especial de grafos bloco: as árvores. Uma ár-

Algoritmo 3.4: Grafos bloco: Dominação Vetorial

Entrada: Grafo bloco $G = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

$S \leftarrow \emptyset$ $\quad G' \leftarrow G$ $\quad L \leftarrow \emptyset$
para cada $v \in V(G')$ **faça**
  $L(v) \leftarrow (B, R[v])$ $\quad L \leftarrow L \cup L(v)$
enquanto $V(G') \neq \emptyset$ **faça**
  se G' *é biconexo* **então**
    $S_B \leftarrow$ `DominarClique` (G', L)
    $S \leftarrow S \cup S_B$
    $G' \leftarrow \emptyset$
  senão
    Seja B um bloco folha de G' e seja u a única articulação de B
    se $L_1(u) = R$ **então**
      $S_B \leftarrow$ `DominarClique` (B, L)
      $S \leftarrow S \cup (S_B \setminus \{u\})$
    senão
      $L' \leftarrow L'' \leftarrow L$
      $L''_1(u) \leftarrow R$ $\quad S_R \leftarrow$ `DominarClique` (B, L'')
      $L'_2(u) \leftarrow 0$ $\quad S_0 \leftarrow$ `DominarClique` (B, L')
      se $|S_0| < |S_R|$ **então**
        $S \leftarrow S \cup S_0$
        $L_2(u) \leftarrow max\{L_2(u) - |S_0|,\ 0\}$
      senão
        $S \leftarrow S \cup (S_R \setminus \{u\})$
        $L_1(u) \leftarrow R$
      $G' \leftarrow G' - B$
retorne S

Função `DominarClique` (K, L)
  $S = \{v \in K \mid L_1(v) = R\}$ $\quad K' = K \setminus S$
  enquanto $K' \neq \emptyset$ **faça**
    Seja v o vértice de maior L_2 em K'
    se $L_2(v) > |S|$ **então** $S \leftarrow S \cup \{v\}$ $\quad K' \leftarrow K' \setminus \{v\}$;
    senão Interromper o laço de repetição ;
  retorne S

vore pode ser vista como um grafo bloco onde cada componente biconexa é formada por dois vértices. Os blocos folha podem ser trivialmente obtidos: uma folha da árvore e seu ancestral direto. Esta observação é a base do Algoritmo Algoritmo 3.5, apresentado na Subseção 3.3.5.

3.3.5 Árvores

O Algoritmo 3.5 é uma variação do Algoritmo 3.4, na qual todos os blocos possuem no máximo dois vértices. Essa restrição permite a criação de uma solução mais simples. Assim como o Algoritmo 3.4, seu princípio de funcionamento também está baseado na utilização de rótulos.

Em um primeiro momento, os rótulos de todos os vértices v da árvore são inicializados como $(B, R[v])$. A partir deste momento, cada bloco folha é analisado e resolvido. Tendo em vista a estrutura da árvore, pode-se perceber que o algoritmo, a cada iteração, resolve um bloco folha, constituído de uma folha da árvore e seu ancestral, sendo esse a articulação do bloco. Isto implica que, a cada iteração, do algoritmo uma folha é removida.

Para determinar uma ordenação dos vértices $v_1, v_2, \ldots, v_n$ na qual cada vértice v_i é uma folha na subárvore induzida por $\{v_i, \ldots, v_n\}$ o algoritmo pode definir rótulos para cada vértice utilizando um algoritmo de percurso em pós-ordem. Esse procedimento é suficiente para assegurar a ordenação requirida, uma vez que todos os filhos de um nó possuem rótulos menores do que o seu. Seja $P(v)$ o rótulo de cada vértice v definido pelo percurso.

Vale destacar que durante o processamento de vértice folha v, v só é adicionado ao conjunto R-dominante se foi ele previamente marcado como requerido ou se demanda mais de um vizinho para sua dominação. Em caso contrário, seu ancestral na árvore é preferido, pois claramente possui maior influência sobre outros nós da árvore (a influência de v neste momento é limitada ao seu ancestral). Neste caso, seu ancestral é marcado como requerido.

Em relação à complexidade computacional, nota-se que o Algoritmo 3.5 é de tempo linear, uma vez que cada vértice é processado somente uma vez (no momento em que se torna folha) e o percurso em pós-ordem em árvores tem complexidade $O(n)$.

A Figura 2.15 ilustra a execução deste algoritmo. Nesta figura, as legendas associadas aos vértices (situadas no interior do círculo de cada vértice) correspondem ao rotulo P obtido pelo percurso em pós-ordem. Quando um vértice é removido da árvore, a aresta que o conectava é grafada como uma linha tracejada. Além disso, sempre que um vértice é incluído no conjunto R-dominante sua borda é hachurada. Deve-se destacar que foram omitidas algumas iterações do algoritmo, situadas entre as duas árvores apresentadas na terceira linha da figura.

Algoritmo 3.5: Árvores: Dominação Vetorial

Entrada: Árvore $T = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

Seja T' a árvore T enraizada em um vértice r qualquer
Percorrer a árvore T' em pós-ordem, rotulando os vértices
Seja $P(v)$ o rótulo de cada vértice v
$S \leftarrow \emptyset$
$L \leftarrow \emptyset$
para cada $v \in V(T')$ **faça**
    $L(v) \leftarrow (B, R[v])$
    $L \leftarrow L \cup \{L(v)\}$
para $(i \leftarrow 1; \quad i < n; \quad i++)$ **faça**
    Seja v o vértice de rótulo $P(v) = i$
    Seja u o ancestral direto de v
    se $L_1(v) = R \vee L_2(v) > 1$ **então**
        $L_2(u) \leftarrow max\{L_2(u) - 1, \ 0\}$
        $S \leftarrow S \cup \{v\}$
    se $L_1(v) = B \wedge L_2(v) = 1$ **então**
        $L_1(u) \leftarrow R$
se $L_1(r) = R \vee L_2(r) > 0$ **então**
    $S \leftarrow S \cup \{r\}$
retorne S

Um ponto interessante é que a apresentação de um algoritmo linear para o problema de encontrar um conjunto R-dominante para árvores também demonstra uma solução para o problema em caminhos, estrelas e *caterpillars*, uma vez que também são árvores.

Uma vez terminada a abordagem deste ramo da hierarquia dos grafos cordais (grafos ptolemaicos, blocos, árvores e caminhos), é possível iniciar o estudo dos grafos de intervalo. Como será visto na Subseção 3.3.6, esses dois ramos ainda apresentam algumas interseções como, por exemplo, os grafos dominó ∩ intervalo próprio.

3.3.6 Grafos de Intervalo e de Intervalo Próprio

Considerando que a classe dos grafos de intervalo está contida na classe dos *directed path graphs*, pode-se afirmar que o problema do conjunto dominante admite um algoritmo polinomial quando restrito a grafos de intervalo.

Em relação ao problema da k-dominação, o trabalho de Chiarelli *et al.* [17] apresenta um algoritmo de tempo $O(n^{3k})$ para grafos de intervalo próprio. O algoritmo apresentado

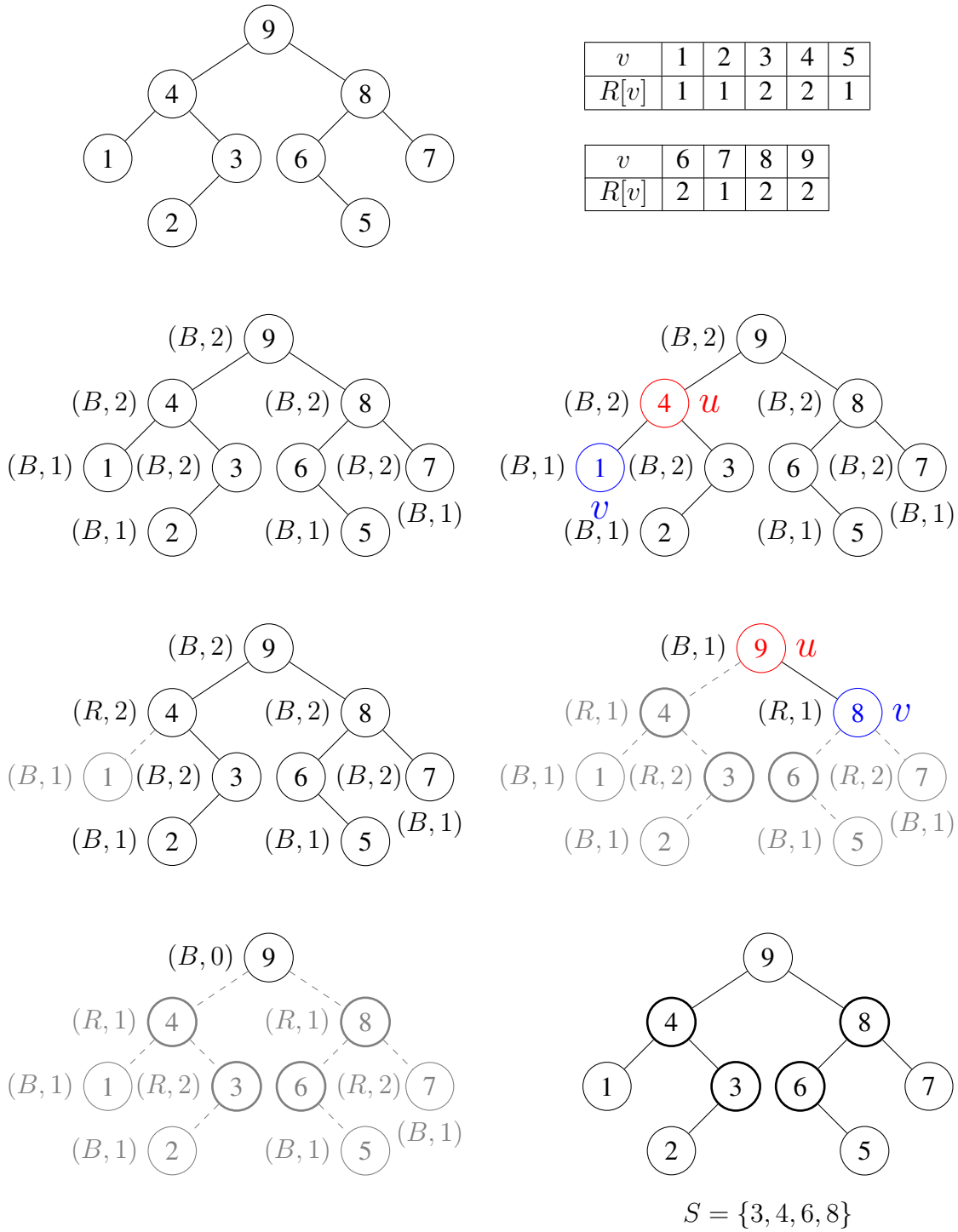


Figura 3.17: Exemplo da aplicação do Algoritmo 3.5 em uma árvore.

é baseado em programação dinâmica e utiliza o parâmetro *boolean-width* e a árvore de decomposição do grafo (*decompositon-tree*) (Bui-Xuan *et al.* [10, 11] e Belmonte e Vatshelle [1]).

Entretanto, o estudo da dominação vetorial nesta classe de grafos ainda não demons-

trou resultados conclusivos, restando como um problema em aberto. Considerando a amplitude da classe dos grafos de intervalo (e de intervalo próprio), optou-se por um estudo baseado em subclasse de grafos de intervalo próprio ainda mais restritas: os grafos dominó $\cap$ intervalo próprio, abordados na Seção 4.2 e os grafos *split-indiferença*, estudados na Seção 4.3.

3.3.7 Grafos *Split*

A primeira abordagem para identificar a complexidade computacional do problema da dominação vetorial em grafos *split* consistiu em buscar resultados na literatura sobre o problema do conjunto dominante. Existem dois trabalhos que demonstram que o problema do conjunto dominante é $\mathcal{NP}$-Completo em grafos *split*.

A primeira demonstração a respeito da $\mathcal{NP}$-Completude decorre do Teorema 3.3.1 destinado aos grafos cordais. A redução polinomial proposta por Booth [7], constrói a partir de um dado grafo $G = (V, E)$, um novo grafo H de forma que:

1. O conjunto $V(G)$ corresponda a uma clique em H;
2. Para cada aresta $e \in E$, um novo vértice v_e é criado em H, tal que v_e é adjacente aos dois vértices correspondentes aos extremos da aresta e em G.

. Em seguida, é demonstrado que G possui uma cobertura de vértices de tamanho k se, e somente, se H possuir um conjunto dominante de cardinalidade k.

Ao analisar o grafo H construído pode-se observar que H, além de ser um grafo cordal, como demonstrado por Booth [7], também é um grafo *split*. A primeira partição, correspondente a uma clique, é formada pelos vértices de G. Já o segundo conjunto construído é constituído pelas arestas de G, transformadas em vértices de grau dois em H. Já que cada elemento é adjacente somente aos vértices da clique então este conjunto corresponde a um conjunto independente.

A Figura 3.18 apresenta uma modificação da Figura 3.12, apresentada na Seção 3.3, na qual os vértices da clique estão hachurados em preto, enquanto os vértices do conjunto independente, em branco.

Sendo assim, pode-se concluir que a demonstração proposta por Booth [7] pode ser aplicada também para grafos *split*.

A segunda demonstração encontrada a respeito da $\mathcal{NP}$-Completude do problema do

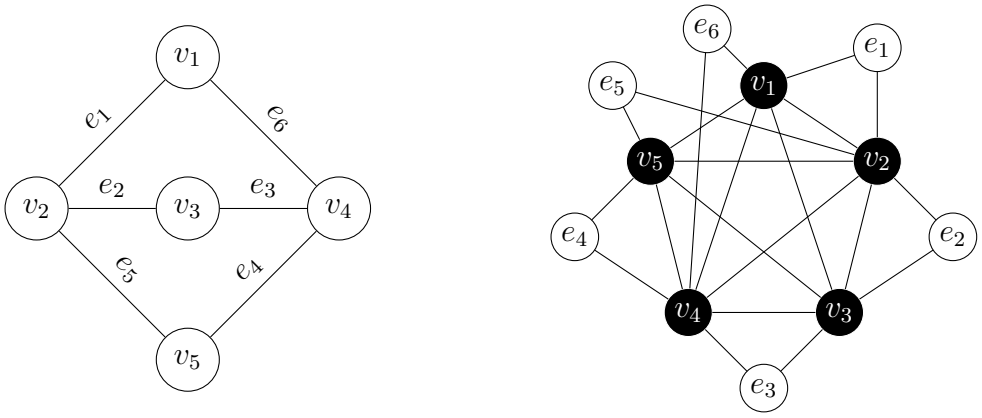


Figura 3.18: O grafo G e o grafo *split* resultante G'.

conjunto dominante em grafos *split* foi apresentada por Bertossi [3].

Teorema 3.3.4 (Bertossi [3])**.** *O Problema do Conjunto Dominante para grafos split é $\mathcal{NP}$-Completo.*

A demonstração do Teorema 3.3.4 consiste em uma redução polinomial para o Problema da Cobertura Mínima de Conjuntos. Dados um conjunto finito e não vazio $A = \{a_1, \ldots, a_m\}$, um conjunto $C = \{C_1, \ldots, C_n\}$ de subconjuntos de A, tal que $C_i \neq \emptyset$ e $\bigcup_{i=1}^{n} C_i = A$, e um inteiro positivo h, o problema da cobertura mínima de conjuntos busca determinar se existe uma cobertura $\{C_{i_1}, \ldots, C_{i_h}\}$ para A, isto é, um subconjunto de C tal que $\bigcup\{C_{i_1}, \ldots, C_{i_h}\} = A$, de tamanho h.

A partir de uma entrada qualquer para o problema da cobertura mínima de conjuntos, um grafo split é construído: A clique base K é constituída dos n elementos de C e o conjunto independente I é formado pelos m elementos de A. Seja $S = (V, E)$ o grafo split construído: $V = K \cup I$ e $E = \{\{C_i, C_j\} : C_i, C_j \in C, i \neq j\} \cup \{\{a_i, C_k\} : a_i \in C_k\}$. Como $\bigcup_{i=1}^{n} C_i = A$ e $C_i \neq \emptyset$, então S é conexo e todo vértice de K é adjacente a pelo menos um vértice de I. A Figura 3.19 ilustra um exemplo desta construção.

Uma vez que o grafo split foi construído, Bertossi [3] demonstra que existe uma solução de tamanho h para o problema da cobertura mínima de conjuntos se, e somente se, existir um conjunto dominante de tamanho h em S. No exemplo apresentado pela Figura 3.19, S possui um conjunto dominante de tamanho 2: $\{C_3, C_4\}$. Esse mesmo conjunto é a solução para o problema da cobertura mínima de conjuntos.

Um ponto interessante do trabalho de Bertossi [3] é que a demonstração da $\mathcal{NP}$-Completude do problema do conjunto dominante em grafos *split* foi utilizada como base para provar que esse mesmo problema também é $\mathcal{NP}$-Completo em grafos bipartidos.

$A = \{a_1, a_2, a_3, a_4, a_5\}$
$C = \{C_1, C_2, C_3, C_4\}$
$C_1 = \{a_2, a_3\}$
$C_2 = \{a_2, a_4\}$
$C_3 = \{a_1, a_2, a_5\}$
$C_4 = \{a_3, a_4, a_5\}$

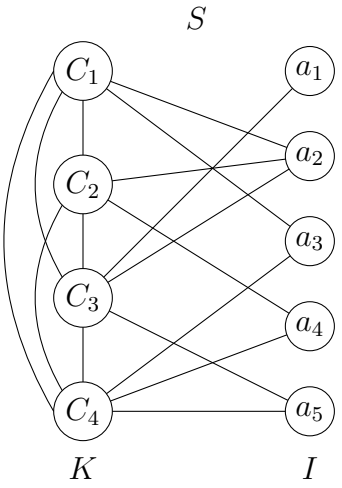


Figura 3.19: Exemplo da construção do grafo split a partir de uma instância do problema da cobertura mínima de conjuntos.

O resultado obtido a respeito da complexidade computacional do problema do conjunto dominante em grafos split permite afirmar que o problema da dominação vetorial também é $\mathcal{NP}$-Completo nesta família.

A Seção 4.3 apresenta uma classe de grafos contida na família dos grafos *split*, cuja complexidade computacional difere da classe que a contém, isto é, constitui um caso especial tratável computacionalmente.

3.4 Conclusões

Este capítulo explorou o estado da arte da complexidade computacional do problema da dominação vetorial em algumas subfamílias de grafos cordais. Este estudo possibilitou um estudo mais aprofundado das características e ferramentas estruturais das classes abordadas.

Uma vez estabelecido o estado da arte, o Capítulo 4 apresenta novos resultados obtidos para algumas classes de grafos cordais, como grafos com exatamente duas cliques maximais e grafos *split*-indiferença. Além desses dois algoritmos lineares, o capítulo expõe também um algoritmo aproximativo para os grafos dominó $\cap$ intervalo próprio.

Capítulo 4

Novas Contribuições para o Problema da Dominação Vetorial

Este capítulo apresenta as contribuições deste trabalho em relação à complexidade computacional do problema da dominação vetorial em grafos cordais. A primeira é um algoritmo linear para grafos com exatamente duas cliques maximais. Como visto na Subseção 3.2.2, o método anteriormente conhecido na literatura para resolver o problema neste classe tem complexidade $O(|\sigma|(n+1)^{15})$, o σ é uma k-expressão para o grafo [18].

Já a segunda contribuição é uma continuação do estudo da classe dos grafos de intervalo próprio. O método proposto é um algoritmo aproximativo para grafos dominó $\cap$ intervalo próprio. Como será visto na Seção 4.2, este estudo possibilitou um melhor entendimento do problema da dominação vetorial em relação à decomposição do grafo em suas cliques maximais.

Finalmente, a terceira contribuição visa avaliar a complexidade de uma classe formada pela interseção dos grafos *split* e os grafos de intervalo próprio. Como visto nas Subseções 3.3.6 e 3.3.7, a complexidade do problema da dominação vetorial é desconhecida em grafos de intervalo próprio e intratável em grafos *split* (assumindo que $\mathcal{P} \neq \mathcal{NP}$). Contudo, como será demonstrado na Seção 4.3, o problema admite um algoritmo de tempo linear na interseção das duas classes, os grafos *split*-indiferença.

4.1 Grafos com duas cliques maximais

Nesta seção serão apresentados os resultados obtidos por este trabalho para grafos com duas cliques maximais, isto é, grafos constituídos por exatamente duas cliques maximais

conectadas por pelo menos um vértice em comum. Como visto anteriormente, nota-se que estes grafos pertencem à família dos cordais, uma vez que inexiste a possibilidade de um ciclo com mais de três vértices sem uma corda. Como demonstração adicional, um esquema de eliminação perfeita pode ser facilmente obtido: sejam C_1 e C_2 as duas cliques; Primeiro, os vértices contidos em $(C_1 \setminus C_2) \cup (C_2 \setminus C_1)$ são removidos em qualquer ordem; Em seguida, os vértices de $C_1 \cap C_2$ também podem ser removidos.

O algoritmo desenvolvido para encontrar conjuntos R-dominantes mínimos em grafos com duas cliques maximais é constituído de duas fases: na primeira, um conjunto capaz de R-dominar todos os vértices é construído para o grafo. Já na segunda fase, este conjunto é aprimorado de forma que se torne mínimo.

Inicialmente, o algoritmo identifica as duas cliques que compõem o grafo e, para isso, basta identificar um vértice de grau mínimo. Seja v esse um desses vértices. Sabe-se que v não pertence à interseção entre as cliques, pois tais vértices são universais e, portanto, possuem grau máximo. Desta forma, a primeira clique é determinada pela vizinhança fechada deste vértice v. Para identificar a segunda clique, basta localizar um vértice u que não seja vizinho de v. A segunda clique é constituída pela vizinhança fechada deste vértice u. Sejam C_1 e C_2 as duas cliques identificadas e seja I a interseção entre elas. Esta etapa pode ser realizada em tempo $O(n)$. Uma vez que as cliques foram identificadas, o algoritmo pode construir um conjunto capaz de R-dominar o grafo para, em seguida, aprimorá-lo.

Algoritmo 4.1: Duas cliques maximais: Dominação vetorial

Entrada: Grafo $G = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

1 Seja v o vértice de grau mínimo em V
2 $C_1 \leftarrow N[v]$
3 Seja u um vértice qualquer em $V \setminus C_1$
4 $C_2 \leftarrow N[u]$
5 $I \leftarrow C_1 \cap C_2$
6 $S_{aux} \leftarrow$ `ConstruirSoluçãoDuasCliques` (C_1, C_2, I, R)
7 $S \leftarrow$ `AprimorarSolução` (C_1, C_2, I, R, S_{aux})
8 **retorne** S

A construção de um conjunto viável é imediata: todos os vértices de $C_1 \setminus I$ e $C_2 \setminus I$ são incluídos no conjunto em construção S_{aux}; Em seguida, são incorporados os vértices de I não dominados por S_{aux} em ordem decrescente de requisitos. O Algoritmo 4.2 detalha este procedimento.

Algoritmo 4.2: Duas cliques maximais: Construir solução

Entrada: Cliques C_1 e C_2, Interseção entre as cliques I e Vetor de requisitos R
Saída: Conjunto S_{aux} capaz de R-dominar G

```
1 Função ConstruirSoluçãoDuasCliques(C1, C2, I, R)
2     S_aux ← (C1 \ I) ∪ (C2 \ I)
3     Seja C uma lista dos vértices de I ordenada de forma decrescente por
        requisitos
4     enquanto C ≠ ∅ faça
5         Remover o primeiro elemento de C. Seja v esse vértice
6         se R[v] > |S| então
7             S_aux ← S_aux ∪ {v}
8     retorne S_aux
```

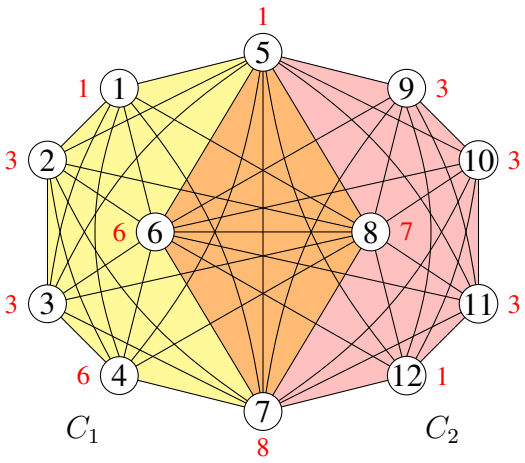
Em relação ao conjunto construído, pode-se assegurar sua capacidade de dominar todos os vértices. Para isso, basta observar que todos os vértices de $(C_1 \setminus I) \cup (C_2 \setminus I)$ pertencem ao conjunto e os demais (contidos em I) são analisados de forma que os não dominados pelo primeiro conjunto sejam incorporados ao conjunto resultante, seguindo o mesmo procedimento já demonstrado para cliques.

Nota-se que o Algoritmo 4.2 possui complexidade $O(n)$ [1], pois cada vértice é processado apenas uma vez e a ordenação empregada pode ser realizada em tempo linear, uma vez que os requisitos são limitados a n.

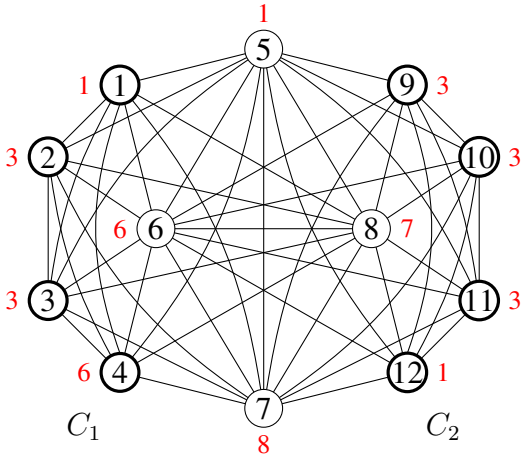
A Figura 4.1 apresenta um exemplo do comportamento do Algoritmo 4.2. Pode-se notar claramente as duas etapas do algoritmo: Na primeira, todos os vértices que não pertencem à interseção são adicionados. Em seguida, apenas os vértices não dominados pelo conjunto construído anteriormente são adicionados.

Uma vez que um conjunto capaz de R-dominar o grafo foi construído, o algoritmo inicia a fase de aprimoramento. Para isso, é necessário demonstrar quais operações podem ser realizadas para transformar um conjunto S construído pelo Algoritmo 4.2 em um conjunto R-dominante mínimo para G. O Lema 4.1.1 restringe as operações que podem ser realizadas para aprimorar tal conjunto.

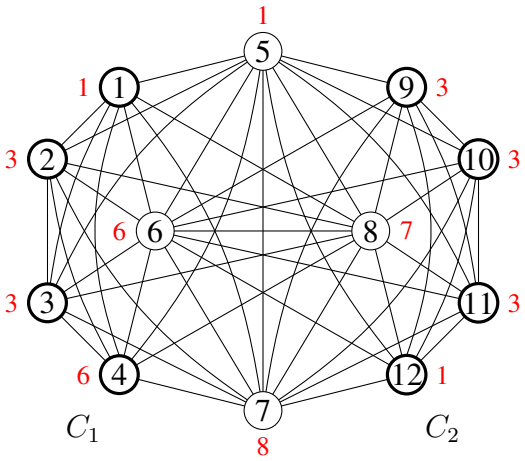
[1] Considerando que o grafo é representado como uma lista de adjacências, é possível obter o grau de cada vértice em tempo constante.



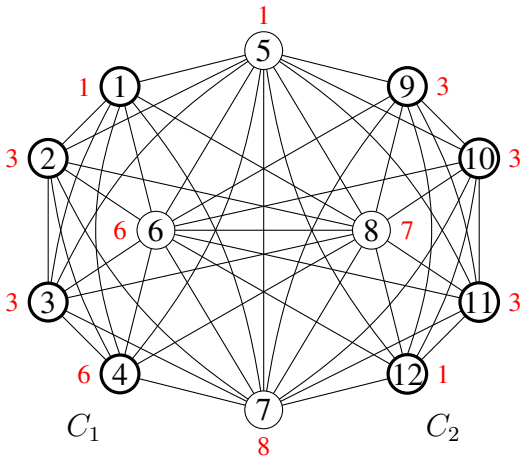
Início do Algoritmo:
Grafo G e suas duas cliques C_1 e C_2



Etapa 1: Adição indiscriminada dos vértices situados fora da interseção entre C_1 e C_2



Etapa 2:
$C = \{7, 8, 6, 5\}$



Etapa 2: Primeiro vértice de C já é dominado por S ($R[7] = 8 = |S|$).
Fim do algoritmo.

Resultado final: $S = \{1, 2, 3, 4, 9, 10, 11, 12\}$

Figura 4.1: Exemplo do Algoritmo 4.2.

Lema 4.1.1. *Seja S' uma solução mínima para G e R que contenha os vértices de maior requisitos de cada região que compõem o grafo ($C_1 \setminus C_2$, $C_2 \setminus C_1$, $I = C_1 \cap C_2$). Além disso, seja S um conjunto capaz de R-dominar G tal que $(S \setminus I) \supseteq (S' \setminus I)$ e $(S \cap I) \subseteq (S' \cap I)$. A Figura 4.2 apresenta visualmente as restrições impostas ao conjunto S em relação à S'.*

Existem apenas duas modificações que podem ser realizadas em S para aprimorá-lo:

- *Adicionar um vértice v, caso $v \in I \setminus S$;*
- *Remover um vértice v, caso $v \in (V \setminus I) \cap S$.*

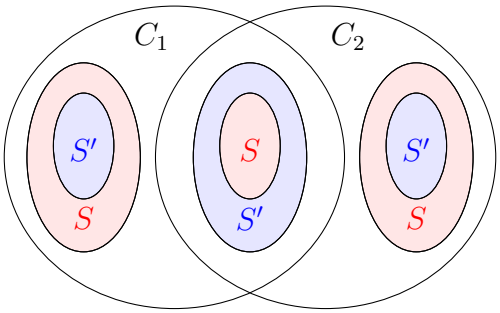


Figura 4.2: Apresentação visual das restrições impostas ao conjunto S em relação à S'.

Demonstração. Sejam C_1 e C_2 as duas cliques maximais de G, tais que $I = C_1 \cap C_2$. Seja v um vértice qualquer do grafo. Existem oito possíveis casos, considerando a localização de v no grafo e nos conjuntos S e S':

Supondo que $v \in I$:

1. $v \in S'$, mas $v \in S$
2. $v \in S'$, mas $v \notin S$
3. $v \notin S'$ e $v \in S$
4. $v \notin S'$ e $v \notin S$

Supondo que $v \notin I$:

5. $v \in S'$, mas $v \in S$
6. $v \in S'$, mas $v \notin S$
7. $v \notin S'$ e $v \in S$
8. $v \notin S'$ e $v \notin S$

Inicialmente, dentre as proposições descritas, percebe-se que duas delas resultam em absurdos: A proposição 3 é absurda, pois $(S' \cap I) \supseteq (S \cap I)$. A proposição 6 também é absurda, uma vez que $(S \setminus I) \supseteq (S' \setminus I)$.

Considerando os casos restantes, os conjuntos diferem em apenas dois:

- Existe um vértice $v \in I$ contido em S', mas não em S (Proposição 2). Para tornar S ótimo, é necessário adicionar v;
- Existe um vértice v em $S \setminus I$, mas não em S' (Proposição 7). Para tornar S ótimo, é necessário remover v.

Logo, conclui-se que de fato só existem duas modificações que podem ser realizadas em S para aprimorá-lo. □

O Algoritmo 4.3 busca aprimorar o conjunto construído de forma que este se torne mínimo, através das adições e remoções demonstradas no Lema 4.1.1. A cada iteração, dois vértices de requisitos mínimos são identificados: o primeiro pertencente ao conjunto $(C_1 \setminus C_2) \cap S$ e o segundo, a $(C2 \setminus C_1) \cap S$. Caso existam tais vértices, o algoritmo avalia se ambos (ou apenas um deles) podem ser removidos do conjunto S. Caso isso seja possível, a remoção é executada.

Quando nenhum vértice puder ser removido sob risco de deixar algum vértice não R-dominado, o algoritmo acrescenta um único vértice do conjunto $(C_1 \cap C_2) \setminus S$ a S e recomeça o processo de identificar possíveis remoções. No momento em que nenhuma operação puder ser realizada (adição ou remoção), o algoritmo é encerrado. A cada etapa, o conjunto obtido é comparado ao menor já encontrado: caso um conjunto de menor cardinalidade seja localizado, ele é salvo, sobrescrevendo o anterior. Ao final, o menor conjunto identificado é retornado. Vale destacar que, em todas as iterações do algoritmo, o conjunto pode R-dominar o grafo, uma vez que as remoções são realizadas somente quando são viáveis (o grafo seguirá R-dominado). Desta forma, qualquer que seja o conjunto retornado, ele é suficiente para R-dominar o grafo. Como será demonstrado no Lema 4.1.2, o conjunto retornado é também uma solução (mínima) para o grafo.

Para verificar se é possível remover os vértices com os menores requisitos de cada clique, o algoritmo deve considerar estes dois vértices e mais um terceiro, pertencente à interseção. Sejam v e u os dois vértices de menores requisitos dos seguintes conjuntos $(C_1 \setminus C_2) \cap S$ e $(C_2 \setminus C_1) \cap S$ e seja w o vértice de maior requisito em $(C_1 \cap C_2) \setminus S$. Desta forma, para verificar se a remoção de v (simétrico para u) é possível, o algoritmo deve validar as seguintes condições:

- $R[v] \leq |N(v) \cap S|$;
- $R[w] < |N(w) \cap S|$;
- $\nexists x \in (N(v) \setminus S) \mid R[x] \geq |N(x) \cap S|$.

Além disso, se for possível remover ambos os vértices v e u, um outro caso deve ser analisado: $\nexists z \in ((C_1 \cap C_2) \setminus S) \mid R[z] \geq (N(z) \cap S) - 2$. Se este caso for violado, apenas um dos vértices pode ser removido. Neste caso, o vértice de menor requisito é removido.

Pode-se notar que a execução direta destes testes implicará em um percurso na vizinhança dos vértices v, u e w. Para evitar o custo desses percursos, o algoritmo armazena os requisitos dos últimos vértices removidos de cada clique ($maxRem_{C_1}$ e $maxRem_{C_2}$). É suficiente analisar estes dois vértices, pois a remoção é executada em ordem crescente de requisitos. Para complementar o teste, o vértice de maior requisito pertencente a $((C_1 \cap C_2) \setminus S)$ é analisado (no algoritmo, equivale ao primeiro elemento do conjunto $Inter$). Vale observar que o conjunto inicial contém $C_1 \setminus C_2$ e $C_2 \setminus C_1$ e, portanto, não existem impedimentos nesses conjuntos na primeira remoção Desta forma, são os valores são inicializados com 0.

No algoritmo, vale destacar que as operações de remoção e adição acontecem interca-

ladas. Esse comportamento garante que apenas soluções minimais são salvas (caso sejam as menores já localizadas).

Algoritmo 4.3: Duas cliques maximais: Aprimorar solução

Entrada: Cliques C_1 e C_2, Interseção I, Vetor de requisitos R e Conjunto S
Saída: Conjunto R-dominante S_{Min}

```
1  Função AprimorarSolução(C1, C2, I, R, S)
2      S1 ← S ∩ (C1 \ I)      S2 ← S ∩ (C2 \ I)      SI ← S ∩ I
       Inter ← I \ S
3      Ordenar S1 e S2 em ordem crescente de requisitos
4      Ordenar Inter em ordem decrescente de requisitos
5      S_Min ← S      maxRem_C1 ← 0      maxRem_C2 ← 0
6      faça
7          impS1 ← Verdadeiro      impS2 ← Verdadeiro
8          se S1 ≠ ∅ então
9              Seja v1 o primeiro elemento de S1
10             impS1 ← (R[v1] ≥ |S1| + |SI|) ∨ (maxRem_C1 ≥ |S1| + |SI|)
11         se S2 ≠ ∅ então
12             Seja v2 o primeiro elemento de S2
13             impS2 ← (R[v2] ≥ |S2| + |SI|) ∨ (maxRem_C2 ≥ |S2| + |SI|)
14         se Inter ≠ ∅ então
15             Seja vi o primeiro elemento de Inter
16             impS1 ← impS1 ∨ R[vi] ≥ |S|      impS2 ← impS2 ∨ R[vi] ≥ |S|
17         se impS1 = Falso ∧ impS2 = Falso ∧ Inter ≠ ∅ então
18             Seja vi o primeiro elemento de Inter
19             se R[vi] > |S| − 2 então
20                 se R[v1] > R[v2] então impS1 ← Verdadeiro
21                 senão impS2 ← Verdadeiro
22         se impS1 = Falso então
23             S ← S \ {v1}      S1 ← S1 \ {v1}      maxRem_C1 ← R[v1]
24         se impS2 = Falso então
25             S ← S \ {v2}      S2 ← S2 \ {v2}      maxRem_C2 ← R[v2]
26         se impS1 = Verdadeiro ∧ impS2 = Verdadeiro ∧ Inter ≠ ∅ então
27             Seja vi o primeiro elemento de Inter
28             S ← S ∪ {vi}      SI ← SI ∪ {vi}      Inter ← Inter \ {vi}
29             impS1 ← Falso      impS2 ← Falso
30         se |S| < |S_Min| então S_Min ← S
31     enquanto impS1 = Falso ∨ impS2 = Falso
32     retorne S_Min
```

A Figura 4.3 apresenta um exemplo da execução do Algoritmo 4.3. O primeiro quadro, demonstrado na figura, corresponde à saída apresentada na Figura 4.1 após as seguintes

alterações (já executadas): Na primeira iteração, o vértice 7 é adicionado, pois nenhuma remoção era viável; Em seguida, os vértices 1 e 12 são removidos. Na terceira iteração, o vértice 8 é adicionado, pois novamente nenhuma remoção era viável. Já na quarta iteração, (última antes do início da Figura 4.3), os vértice 2 e 9 são removidos.

Conforme pode ser observado no último quadro da figura, S_{Min} indica um novo conjunto mínimo localizado e aponta ainda que o algoritmo fará uma nova (e última) iteração, que corresponde à adição do vértice 5. Após essa adição, o algoritmo encerrará sua execução, pois nenhuma operação poderá ser realizada. Desta forma, o algoritmo retornará o conjunto $S_{Min} = \{3, 6, 7, 8\}$, que corresponde a um conjunto R-dominante mínimo para as entradas apresentadas na Figura 4.1.

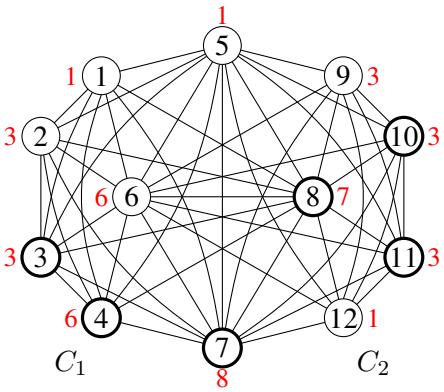
A corretude do Algoritmo 4.3 é demonstrada pelo Lema 4.1.2.

Lema 4.1.2. *O Algoritmo 4.3 é capaz de encontrar um conjunto R-dominante mínimo, partindo de um conjunto S_{aux} capaz de R-dominar G, construído pelo Algoritmo 4.2, para um grafo $G = (V, E)$, constituído de exatamente duas cliques maximais C_1 e C_2, e um vetor de requisitos R.*

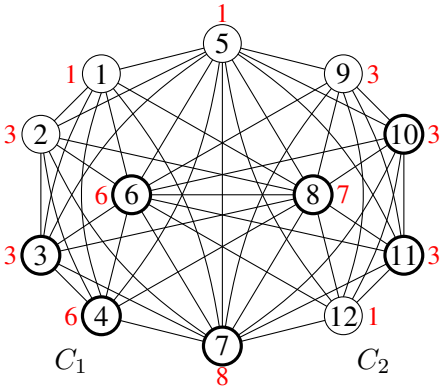
Demonstração. A demonstração será feita em duas partes. A primeira busca estabelecer que, a cada iteração, o conjunto S satisfaz os requisitos do Lema 4.1.1. Já a segunda demonstra que, ao final do processo, o conjunto S é mínimo e, portanto, uma solução para as entradas fornecidas.

A primeira parte da demonstração será realizada por indução em S. A base da indução corresponde ao primeiro conjunto abordado, isto é, o conjunto construído pelo Algoritmo 4.2. Seja I a interseção entre as cliques C_1 e C_2. Pelo processo empregado no algoritmo de construção, sabe-se que S contém todo o conjunto $((C_1 \setminus I) \cup (C_2 \setminus I))$, satisfazendo a primeira restrição. Em relação à segunda, sabe-se que todo vértice $v \in S \cap I$ não é dominado por nenhum conjunto de cardinalidade inferior à $|S|$, pois, do contrário, v não teria sido adicionado durante a construção de S. Isto implica que todo vértice $v \in S \cap I$ deve pertencer a pelo menos uma solução mínima. (Caso exista um vértice u tal que $u \in S'$, $v \notin S'$ e $R[u] \leq R[v]$, pode-se trocar os vértice u e v sem que a solução deixe de dominar todos os vértices ou tenha seu tamanho alterado.)

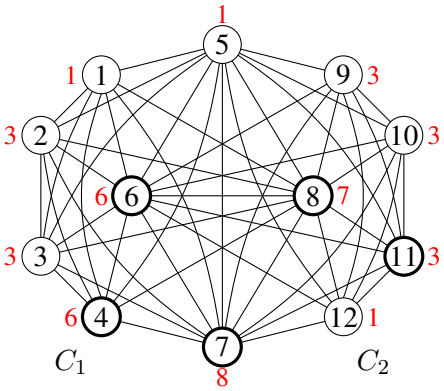
Seja S_x o conjunto S após x iterações, tal que S_x não corresponde a uma solução (conjunto mínimo), e seja C_S o conjunto de todas as soluções mínimas para um grafo $G = (V, E)$ e o vetor de requisitos R. Além disso, seja S' uma solução dentre C_S, tal



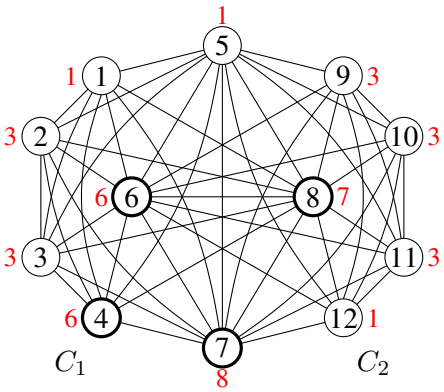
Variável	C_1	C_2
S	$\{3,4\}$	$\{10,11\}$
v	3	10
imp	Verdadeiro	Verdadeiro
$maxRem$	3	3
S_I	$\{7,8\}$	
$Inter$	$\{6,5\}$	
S	$\{3,4,7,8,10,11\}$	
S_{min}	$\{3,4,7,8,10,11\}$	
Consequência	Adição do vértice 6	



Variável	C_1	C_2
S	$\{3,4\}$	$\{10,11\}$
v	3	10
imp	Falso	Falso
$maxRem$	3	3
S_I	$\{7,8,6\}$	
$Inter$	$\{5\}$	
S	$\{3,4,6,7,8,10,11\}$	
S_{min}	$\{3,4,7,8,10,11\}$	
Consequência	Remoção dos Vértices 3 e 10	



Variável	C_1	C_2
S	$\{4\}$	$\{11\}$
v	4	11
imp	Falso	Verdadeiro
$maxRem$	3	3
S_I	$\{6,7,8\}$	
$Inter$	$\{5\}$	
S	$\{4,6,7,8,11\}$	
S_{min}	$\{4,6,7,8,11\}$	
Consequência	Remoção do vértice 11	



Variável	C_1	C_2
S	$\{4\}$	$\{\}$
v	4	
imp	Verdadeiro	Verdadeiro
$maxRem$	3	3
S_I	$\{6,7,8\}$	
$Inter$	$\{5\}$	
S	$\{\mathbf{4},\mathbf{6},\mathbf{7},\mathbf{8}\}$	
S_{min}	$\{\mathbf{4},\mathbf{6},\mathbf{7},\mathbf{8}\}$	
Consequência	Adição do vértice 5	

Figura 4.3: Ilustração de alguns passos do Algoritmo 4.3.

que S' contém os vértices de maiores requisitos de cada região que compõem cada solução mínima. Pela hipótese da indução, sabe-se que S_x atende aos requisitos. Para provar o passo da indução, basta demonstrar que o conjunto S_{x+1}, obtido ao final de uma nova iteração, também mantém os requisitos satisfeitos.

O primeiro caso que deve ser demonstrado corresponde à remoção de um vértice (supondo que exista um vértice v que pode ser removido). Pelo Lema 4.1.1, v deve pertencer a $((C_1 \setminus I) \cap S)$ ou a $((C_2 \setminus I) \cap S)$. Seja $S_{x+1} = S_x \setminus \{v\}$. Nota-se que $S_{x+1} \subset S_x$. Portanto, para demonstrar que S_{x+1} atende aos requisitos, é necessário provar que $S' \setminus I \subseteq S_{x+1}$ (Restrição 1 do Lema 4.1.1). Supondo que v pode ser removido de S_x, mas $v \in S'$ e ainda assim $(S' \setminus I) \subseteq S_x$ e S' é mínimo, conclui-se que existe um vértice qualquer u tal que $u \in S_x \setminus S'$. Como S' possui os vértices de maiores requisitos, conclui-se que $R[v] \geq R[u]$. O caso em que $R[v] > R[u]$ resulta em um absurdo, pois v foi escolhido por possuir o menor requisito dentre os candidatos. Isso implica que $R[v] = R[u]$ e, neste caso, basta trocar v por u (ou seja, u seria removido, mantendo v em S_{x+1}). Considerando que essa troca não altera a cardinalidade do resultado final, conclui-se que $(S' \setminus I) \subseteq S_{x+1}$. A segunda restrição segue inalterada, pois $S_x \cap I$ não foi alterado.

O segundo caso analisado considera a adição de um vértice. É importante destacar que o algoritmo adiciona um vértice somente quando nenhuma remoção é possível, implicando que S_x é minimal. Além disso, como S_x não é mínimo conclui-se que: $(S' \setminus I) \subset S_x$, $(S_x \cap I) \subset (S' \cap I)$ e que, para reduzir a cardinalidade de S_x, uma adição é necessária.

Antes de iniciar a demonstração relativa à adição de um vértice, é necessário provar que existe um vértice v que pode ser adicionado. Para isso, supõe-se, por absurdo, que não existe tal vértice v ($v \in I \setminus S_x$). Considerando que S_x atende às restrições impostas pelo Lema 4.1.1, S_x não é um conjunto R-dominante mínimo para G e que nenhum vértice pode ser removido de S_x sem impactar sua capacidade de dominar todos os vértices, conclui-se que $S_x \supseteq I$ e, portanto, $S' \supseteq I$, implicando que $S_x \cap I = S' \cap I$. Considerando que S_x não é uma solução, percebe-se que deve existir um vértice que pertence a $S_x \setminus I$, mas não a $S' \setminus I$, passível de remoção. Desta forma, nota-se um absurdo, um vez que S_x é minimal e possui os vértices de maiores requisitos, implicando que existe de fato um vértice v que pode ser adicionado.

A demonstração de que o conjunto S_{x+1}, obtido pela adição do vértice v ao conjunto S_x, atende as restrições é imediata, pois v possui o maior requisito dentre os candidatos e, conforme demonstrado anteriormente, $(S_x \cap I) \subsetneq S'$.

Uma vez demonstrado o passo da indução, a primeira parte da prova está concluída. Resta provar a segunda parte, onde se deseja estabelecer que uma solução pode ser encontrada a partir de uma solução S obtida pelo Algoritmo 4.2. Considerando que todas as remoções e adições são testadas pelo algoritmo, percebe-se que alguma solução S_x chegará ao limite das restrições impostas pelo Lema 4.1.1, isto é, $(S_x \cap I) = (S' \cap I)$ e $(S' \setminus I)) = (S_x \setminus I)$. Além disso, nota-se que as duas restrições combinadas implicam que $S_x = S'$, demonstrando que S_x é uma solução. Como S_x é mínimo, conclui-se que este conjunto será salvo na variável S_{min} e retornado pelo algoritmo. □

O Algoritmo 4.3 possui complexidade linear em relação ao número de vértices do grafo, pois existem no máximo n operações que podem ser realizadas. Além disso, todas as ordenações podem ser realizadas também em tempo linear.

Teorema 4.1.1. *O Algoritmo 4.1 é capaz de localizar uma solução mínima para grafos constituídos de exatamente duas cliques maximais.*

Demonstração. Considerando que o conjunto S_{aux}, construído pelo Algoritmo 4.2, consegue dominar todos os vértices e que o Algoritmo 4.3 pode, a partir deste conjunto, encontrar uma solução mínima para o problema, conforme demonstrado pelo Lema 4.1.2, conclui-se que o Algoritmo 4.1 está correto. □

Tendo em vista que a identificação das cliques maximais, a construção de uma solução e o aprimoramento da solução possuem complexidade $O(n)$, pode-se estabelecer que o Algoritmo 4.1 encontra uma solução em tempo linear em relação ao número de vértices do grafo.

A Seção 4.2 apresenta outro resultado deste trabalho: um método aproximativo para o problema da dominação vetorial em grafos dominó ∩ intervalo próprio. Como visto na Subseção 3.3.6, a ideia de estudar esta classe veio da necessidade de restringir ainda mais a estrutura dos grafos de intervalo próprio, visando determinar a complexidade computacional do problema da dominação vetorial.

4.2 Grafos Dominó ∩ Intervalo Próprio

Como visto na Subseção 2.3.10, a complexidade computacional do problema da dominação vetorial em grafos dominó ∩ intervalo próprio já é conhecida: o algoritmo de

Cicalese *et al.* [18] resolve o problema da dominação vetorial nesta família de grafos em tempo $O(|\sigma|(n+1)^{15})$, onde σ equivale a uma 3-expressão (k-expressão onde $k \leq 3$) para o grafo.

Embora a complexidade já seja conhecida, optou-se por estudar esta família de grafos por simplificar a estrutura dos grafos de intervalo próprio: como todo vértice está contido em no máximo duas cliques, cada vértice de uma clique ou é simplicial, ou está contido na interseção de somente duas cliques. Essa restrição garante que a remoção de uma clique folha (apresenta interseção com somente uma clique maximal do grafo) não desconecta o grafo, além de estabelecer um percurso fácil pelas pelas cliques maximais do grafo. Outra propriedade observada nesta família é a obtenção trivial de um caminho de cliques (uma árvore de cliques que induz um caminho). Espera-se que os algoritmos desenvolvidos para esta família também sejam válidos para os grafos de intervalo próprio (ou até mesmo para grafos de intervalo). A Figura 4.4 ilustra um exemplo de grafo desta família.

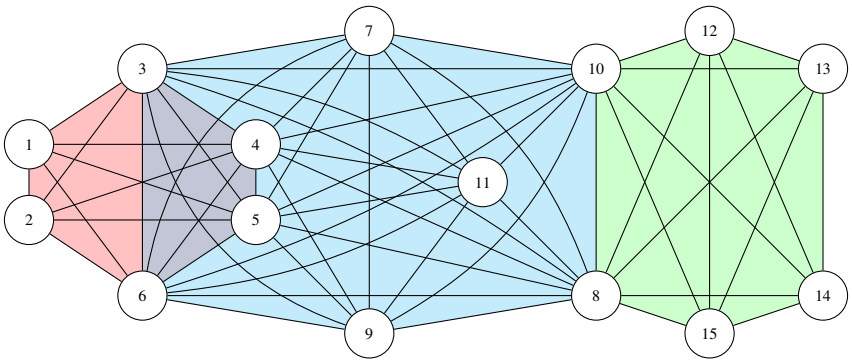


Figura 4.4: Exemplo de grafo dominó ∩ intervalo próprio.

Esperava-se também que estrutura mais restrita desta família de grafos permitisse estabelecer uma decomposição do grafo em cliques maximais de forma a viabilizar um algoritmo baseado em divisão e conquista ou até mesmo um algoritmo guloso, como os algoritmos destinados aos grafos bloco e aos *directed path graphs*. Entretanto, foi observado que o problema da dominação vetorial não apresenta subestrutura ótima quando particionado por cliques maximais nesta família de grafos, o que inviabiliza a construção de um algoritmo divisão e conquista (como, por exemplo, um algoritmo baseado em programação dinâmica). Esta conclusão foi obtida após submeter vários algoritmos desenvolvidos ao ambiente de pré-validação. Em todos os resultados, foram identificados casos em que um único vértice adicionado na primeira clique é capaz de impactar a solução da última clique maximal do grafo. Além disso, também foram observados casos em que a solução ótima de cada clique maximal não leva a uma solução ótima global.

Entretanto, dentre os métodos implementados, o Algoritmo 4.4 se mostrou um bom método aproximativo, que geralmente encontra soluções ótimas (conclusão experimental utilizando o ambiente de pré-validação). O caminho das cliques é percorrido partindo de uma clique folha em direção a outra (uma das duas é a raiz da árvore de cliques, entretanto isso não altera o funcionamento do algoritmo) e, a cada iteração, os vértices simpliciais da cliques maximal são dominados. Após o processamento, os vértices simpliciais são removidos, implicando que os vértices da interseção se tornam simpliciais na próxima iteração. Quando restar somente uma única clique, todos os vértices são simpliciais e o algoritmo destinado a cliques (Algoritmo 3.1) pode ser aplicado.

Para cada clique dois casos são analisados. O primeiro conjunto é construído de forma gulosa utilizando obrigatoriamente vértices da interseção entre a clique em processamento e a próxima no caminho até que todos os vértices simpliciais sejam dominados (o algoritmo assume que os vértices da interseção serão dominados durante o processamento da próxima clique). Caso ainda reste algum vértice não dominado após a inclusão de toda a interseção, vértices simpliciais são adicionados até que não reste nenhum vértice não dominado. O segundo conjunto é construído também de forma gulosa, utilizando apenas vértices simpliciais. A dominação dos vértices simpliciais é garantida, pois no pior caso, todos os vértices são adicionados ao conjunto. Portanto, sejam S_1 e S_2 os dois conjuntos construídos.

Antes de decidir a melhor opção, o algoritmo avalia o impacto de cada conjunto na próxima clique. Para isso, o algoritmo guloso apresentado para cliques é aplicado considerando os casos em que S_1 e S_2 são adicionados ao conjunto R-dominante. Sejam X_1 e X_2 os dois conjuntos obtidos para a próxima clique considerando respectivamente a inclusão de S_1 e S_2 ao conjunto em construção. O algoritmo pode então optar pela combinação que encontrou o menor conjunto $|S_1 \cup X_1|$ e $|S_2 \cup X_2|$. Entretanto, um caso demanda tratamento especial: quando S_1 contém S_2 não faz sentido utilizar S_1, uma vez que a ideia é minimizar o conjunto resultante. Neste caso, S_2 é escolhido, implicando que os vértices da interseção devem ser dominados pela próxima clique maximal.

A última clique demanda um processamento diferente: não existe uma próxima clique, logo todos os vértices são simpliciais e, portanto, devem ser resolvidos nesta iteração. Para isso, o algoritmo para cliques é aplicado. A solução para o grafo consiste da união dos conjuntos obtidos para cada clique maximal. O Algoritmo 4.4 detalha este procedimento.

A corretude do algoritmo pode ser facilmente verificada: cada conjunto S_{Ki} domina

os vértices simpliciais de cada clique K_i; todos os vértices serão simpliciais em algum momento, logo serão dominados por algum S_{Ki} (mesmo que seja S_{Kk}). Como todos os vértices são dominados, o conjunto resultante é suficiente para R-dominar o grafo. A complexidade deste algoritmo é $O(n^3)$, dada a necessidade de percorer a vizinhança dos vértices em cada iteração para ajustar o vetor de requisitos.

Embora o algoritmo retorne um conjunto R-dominante, os resultados são apenas aproximativos. O ambiente de pré-validação, apresentado no Apêndice A, foi utilizado para testar este algoritmo. Os contraexemplos retornados pelo ambiente permitiram identificar as falhas do algoritmo, além de terem contribuído para um entendimento mais aprofundado do problema nesta classe de grafos. Foram identificados casos em que a solução ótima era formada por conjuntos subótimos para cada clique maximal, demonstrando que o problema não admite um algoritmo baseado na decomposição do grafo em cliques maximais. A Figura 4.5 ilustra um exemplo dessas afirmativas. Como simplificação, todos os vértices de cada grafo apresentam os mesmos requisitos.

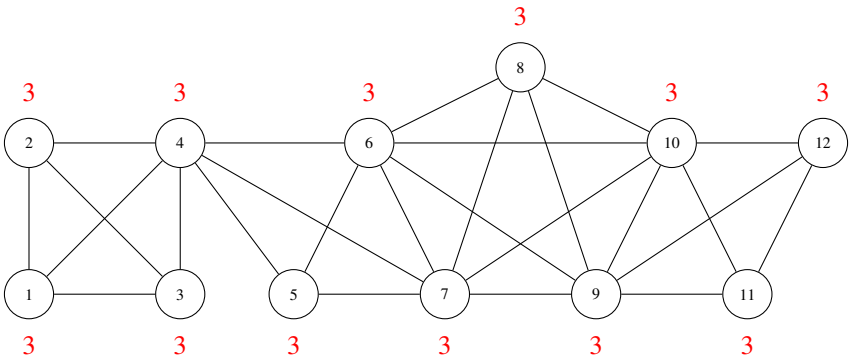


Figura 4.5: Exemplo de caso onde uma decisão ótima local implica que a solução global é subótima.

Neste exemplo, a execução do Algoritmo 4.4 retorna o conjunto $\{1, 2, 4, 6, 7, 9, 11, 12\}$, entretanto a solução ótima para esta entrada é o conjunto $\{1, 2, 4, 5, 8, 11, 12\}$. Para a clique $\{1, 2, 3, 4\}$, o algoritmo identifica $S_1 = \{4, 1, 2\}$, $S_2 = \{1, 2, 3\}$, $X_1 = \{5, 6\}$ e $X_2 = \{5, 6, 7\}$ e opta por S_1 ($|S_1 \cup X_1| < |S_2 \cup X_2|$). Em seguida, ao processar a clique maximal $\{4, 5, 6, 7\}$, o algoritmo identifica que o vértice 4 já está dominado e que apenas o vértice 5 precisa ser dominado: $S_1 = \{6, 7\}$, $X_1 = \{8\}$, $S_2 = \{5\}$ e $X_2 = \{8, 9, 10\}$. O algoritmo opta novamente pelo conjunto S_1. Na próxima clique maximal, apenas o vértice 8 deve ser dominado: $S_1 = \{9\}$, $X_1 = \{11, 12\}$, $S_2 = \{8\}$ e $X_2 = \{11, 12\}$. Como existe um empate, o conjunto S_1 é escolhido. A solução para a última clique é o conjunto $\{11, 12\}$.

Pode-se observar que as soluções para as cliques são ótimas localmente, embora a solução global seja subótima. Ao observar o processamento da clique $\{4, 5, 6, 7\}$, nota-se que o vértice 4 já está contido no conjunto em construção (adicionado pelo processamento da clique anterior $\{1, 2, 3, 4\}$) e que os demais vértices necessitam de dois vizinhos para sua dominação. Os dois casos considerados são $S_1 = \{6, 7\}$, $X_1 = \{10\}$, $S_2 = \{5\}$ e $X_2 = \{8, 9, 10\}$. Como $|S_1 \cup X_1| < |S_2 \cup X_2|$, o algoritmo escolhe o ótimo local $\{6, 7\}$. Como pode ser observado ao continuar a execução do algoritmo, esta escolha embora localmente adequada, implica que a solução resultante não será ótima.

Vale destacar que esta mesma conclusão a respeito de decisões locais ótimas implicarem que a solução global não seja ótima foi observada em diversos outros algoritmos, como, por exemplo, um algoritmo cuja decisão não considera a próxima clique, mas somente a melhor solução para a clique maximal em processamento. A análise do impacto de cada decisão em todas as cliques subsequentes implica em um algoritmo de complexidade exponencial no número de cliques maximais. Em relação à aplicação de programação dinâmica, observou-se que o processamento de uma clique é fortemente dependente dos resultados obtidos nas demais cliques, uma vez que os requisitos dos vértices podem ser alterados em virtude das cliques adjacentes.

Como nenhum método desenvolvido até o momento se demonstrou exato e correto, resta como um problema em aberto continuar o estudo e o desenvolvimento de algoritmos para esta família de grafos. Como outro ponto em aberto neste tópico consta a análise do parâmetro *boolean-width* e a árvore de decomposição do grafo, nos moldes do que foi proposto no trabalho proposto em Chiarelli *et al.* [17] para o problema da k-dominação em grafos de intervalo próprio. Uma suposição é a possibilidade de que a árvore de decomposição do grafo induza uma subestrutura ótima que viabilize a aplicação de algoritmos de divisão e conquista ou até mesmo métodos gulosos (como os propostos para os *directed path graphs* e os grafos bloco).

É interessante salientar que a definição da complexidade computacional do problema da dominação vetorial nas classes de grafos de intervalo e de intervalo próprio apresenta um interesse em particular, dado que pode indicar uma fronteira da complexidade computacional do problema: os grafos *split*-indiferença.

Algoritmo 4.4: Domino ∩ Intervalo Próprio: Método Aproximativo

Entrada: Grafo $G = (V, E)$, Vetor de Requisitos R
Saída: Conjunto S capaz de R-dominar G

$S \leftarrow \emptyset$
Seja $T = (K_1, K_2, \ldots, K_k)$ um caminho de cliques para G
para $(i \leftarrow 1; \quad i \leq k-1; \quad i++)$ **faça**
  $I \leftarrow K_i \cap K_{i+1} \qquad Simp_1 \leftarrow Simp_2 \leftarrow (K_i \setminus K_{i+1}) \setminus S$
  se $Simp_1 \neq \emptyset$ **então**
    Ordenar I, $Simp_1$, $Simp_2$ em ordem decrescente de requisitos
    $S_1 \leftarrow \emptyset \qquad S_2 \leftarrow \emptyset$
    enquanto $R[Simp_1[0]] > |S_1|$ **faça**
      se $I \neq \emptyset$ **então** $S_1 \leftarrow S_1 \cup \{I[0]\} \qquad I \leftarrow I \setminus \{I[0]\}$;
      senão $S_1 \leftarrow S_1 \cup \{Simp_1[0]\} \qquad Simp_1 \leftarrow Simp_1 \setminus \{Simp_1[0]\}$;
    enquanto $R[Simp_2[0]] > |S_2|$ **faça**
      $S_2 \leftarrow S_2 \cup \{Simp_2[0]\} \qquad Simp_2 \leftarrow Simp_2 \setminus \{Simp_2[0]\}$
    $X_1 \leftarrow \texttt{DomClique}(K_{i+1}, S_1) \quad X_2 \leftarrow \texttt{DomClique}(C_{i+1}, S_2)$
    se $S_1 \not\supseteq S_2 \wedge |S_1 \cup X_1| \leq |S_2 \cup X_2|$ **então** $S_{Ki} \leftarrow S_1$ **senão** $S_{Ki} \leftarrow S_2$;
    para cada $v \in S_{Ki}$ **faça**
      para cada $u \in N(v)$ **faça**
        $R[u] \leftarrow \max(R[u] - 1, 0)$
    $S \leftarrow S \cup S_{Ki}$
$DomReq \leftarrow K_k \setminus S$
Ordernar $DomReq$ em ordem decrescente de requisitos
enquanto $R[DomReq[0]] > |S_{Kk}|$ **faça**
  $S_{Kk} \leftarrow S_{Kk} \cup \{DomReq[0]\} \quad DomReq \leftarrow DomReq \setminus \{DomReq[0]\}$
$S \leftarrow S \cup S_{Kk}$
retorne S

Função `DomClique(`K, S`)`
  $R_2 \leftarrow R \qquad DomReq \leftarrow K \setminus S \qquad X \leftarrow \emptyset$
  para cada $v \in S$ **faça**
    para cada $u \in N(v)$ **faça**
      $R_2[u] \leftarrow \max(R_2[u] - 1, 0)$
  Ordernar $DomReq$ em ordem decrescente de R_2
  enquanto $R[DomReq[0]] > |X|$ **faça**
    $X \leftarrow X \cup \{DomReq[0]\} \quad DomReq \leftarrow DomReq \setminus \{DomReq[0]\}$
  retorne X

4.3 Grafos *Split*-Indiferença

Os resultados obtidos para grafos *split*-indiferença constituem a terceira contribuição deste trabalho. É interessante notar que esta classe apresenta muito claramente uma das dicotomias buscadas por este estudo: como visto nas Subseções 3.3.6 e 3.3.7, a complexidade do problema da dominação vetorial é desconhecida em grafos de intervalo próprio e intratável em grafos *split* (assumindo que $\mathcal{P} \neq \mathcal{NP}$), enquanto admite um algoritmo linear em grafos *split*-indiferença.

O algoritmo desenvolvido por este trabalho para a classe dos grafos *split*-indiferença é baseado nos casos apresentados pelos Teoremas 2.3.8 e 2.3.9. Para cada caso apresentado nos teoremas, um método distinto é aplicado. Desta forma, o algoritmo em si consiste em reconhecer o caso correspondente ao grafo fornecido como entrada, identificar as cliques maximais e aplicar a função apropriada. Estes passos são apresentados no Algoritmo 4.5.

Algoritmo 4.5: *Split*-indiferença: Dominação Vetorial

Entrada: Grafo *split*-indiferença $G = (V, E)$ e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

```
1  C ←IdentificarCliques(G)
2  se |C| = 1 então
3  |  S ←ResolverClique(V,R)
4  senão
5  |  se |C| = 2 então
6  |  |  Sejam C1 e C2 duas cliques tais que |C1 \ C2| = 1
7  |  |  S ← ResolverCaso2 (C1, C2, R)
8  |  senão
9  |  |  Sejam C1, C2 e C3 três cliques tais que |C1 \ C2| = |C3 \ C2| = 1
10 |  |  S1,2 ← C1 ∩ C2        S3,2 ← C3 ∩ C2
11 |  |  se S1,2 ∩ S3,2 = ∅ então
12 |  |  |  S ←ResolverCaso3a (C1, C2, C3, R)
13 |  |  senão
14 |  |  |  S ←ResolverCaso3b (C1, C2, C3, R)
15 retorne S
```

4.3.1 Caso 1: Uma clique maximal

O primeiro caso do Teorema 2.3.9 aborda grafos com uma única clique. O Algoritmo 4.6 é baseado no Algoritmo 3.1, destinado aos grafos completos. Optou-se por apresentar novamente este algoritmo uma vez que este é um importante componente para a

solução dos demais casos. Inicialmente, os vértices são ordenados de forma decrescente por seus requisitos. Em seguida, essa ordenação é percorrida e, a cada iteração, um vértice é analisado. Se o conjunto em construção possuir menos elementos que os necessários para R-dominar este vértice, o algoritmo o adicionará ao conjunto. Do contrário, este vértice já foi dominado e o algoritmo pode ser encerrado.

Algoritmo 4.6: *Split*-indiferença - Caso 1: Dominação Vetorial

Entrada: Vértices da clique V_C, Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S para a clique

```
1 Função ResolverClique(V_C, R)
2     S ← ∅        NC = {v ∈ V_C | R[v] > 0}
3     enquanto |NC| > 0 faça
4         Remover o vértice de maior requisito de NC. Seja u esse vértice.
5         se R[u] > |S| então S ← S ∪ {u}
6     retorne S;
```

O Algoritmo 4.6 possui complexidade $O(n)$, pois cada vértice é explorado no máximo uma única vez e a ordenação utilizada pode ser executada em tempo linear no número de vértices (como os requisitos estão contidos no intervalo $[0, n]$ então é possível aplicar um algoritmo de ordenação de tempo $O(n)$, como, por exemplo, a ordenação por caixas). A corretude deste algoritmo foi demonstrada no Teorema 3.2.1, apresentado na Subseção 3.2.1.

4.3.2 Caso 2: Duas Cliques Maximais

No segundo caso apresentado pelo Teorema 2.3.9, o grafo é constituído de duas cliques maximais C_1 e C_2, sendo que uma delas possui exatamente um vértice simplicial v. Sem perda de generalidade, seja $v \in C_1$. Considerando que v pode ou não estar contido no conjunto R-dominante S, os dois casos são analisados separadamente:

Caso 1 – $v \in S$: O vértice simplicial é incluído no conjunto, debitando em uma unidade o número de vértices requeridos para dominar seus vizinhos. Seja

$$R_1[u] = \begin{cases} \max\{R[u] - 1, 0\}, & \text{se } u \in N(v), \\ R[u], & \text{do contrário} \end{cases} \quad \forall\, u \in C_2$$

o vetor de requisitos atualizado. Para dominar C_2 e R_1, aplica-se o método apresentado para o Caso 1. Seja $S^1_{C_2}$ o conjunto retornado pela função. O resultado deste caso é dado

por $S_1 = \{v\} \cup S^1_{C_2}$.

Caso 2 – $v \notin S$: O vértice simplicial deve ser dominado por seus vizinhos, o que implica na inclusão de $R[v]$ vizinhos de v no conjunto. Desta forma, seja A o conjunto dos $R[v]$ vizinhos de v de maiores requisitos. Uma vez que A domina v e influencia os vértices de C_2, é necessário atualizar os requisitos dos demais vértices em C_2. Seja $R_2[u] = \max\{R[u] - |A|, 0\}\ \forall\ u \in C_2 \setminus A$ o vetor de requisitos atualizado. Novamente, aplica-se a função descrita para o Caso 1 considerando $C_2 \setminus A$ e R_2 como entradas. Seja $S^2_{C_2}$ o conjunto retornado pela função. Neste caso, o conjunto retornado é $S_2 = A \cup S^2_{C_2}$.

O conjunto R-dominante mínimo para G e R corresponde ao conjunto de menor cardinalidade dentre S_1 e S_2. O Algoritmo 4.7 apresenta os passos descritos anteriormente.

Algoritmo 4.7: *Split*-indiferença - Caso 2: Dominação Vetorial

Entrada: Cliques C_1 e C_2 e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

Função `ResolverCaso2` (C_1, C_2, R)
  $v \leftarrow C_1 \setminus C_2$
  $R_1 \leftarrow R_2 \leftarrow R$
  `/* Caso 1:` v `está no conjunto` R`-dominante */`
  para $u \in N(v)$ **faça** $R_1[u] \leftarrow \max(R_1[u] - 1, 0)$
  $S_1 \leftarrow \{v\} \cup$ `ResolverCaso1` (C_2, R_1)
  `/* Caso 2:` v `é dominado por seus vizinhos */`
  $A \leftarrow \emptyset$
  enquanto $R[v] > |A|$ **faça**
    Seja u o vértice de maior requisito em $N(v) \setminus A$
    $A \leftarrow A \cup \{u\}$
  para $u \in C_2$ **faça** $R_2[u] \leftarrow \max(R_2[u] - |A|, 0)$
  $S_2 \leftarrow A \cup$ `ResolverCaso1` $(C_2 \setminus A, R_2)$
  `/* Retornar o menor conjunto dentre` S_1 `e` S_2 `*/`
  se $|S_1| \leq |S_2|$ **então** $S \leftarrow S_1$ **senão** $S \leftarrow S_2$
  retorne S

Lema 4.3.1. *O Algoritmo 4.7 obtém um conjunto R-dominante mínimo para grafos split-indiferença que pertençam ao Caso 2 do Teorema 2.3.9 em tempo linear.*

Demonstração. Seja S uma solução ótima de cardinalidade k para um grafo $G = (V, E)$ e um vetor de requisitos R, tal que G é um grafo *split*-indiferença que pertence ao Caso 2.

Se $v \in S$ então $S \setminus \{v\} \subseteq C_2$ e $|S \setminus \{v\}| = k - 1$. Desta forma, todo vértice

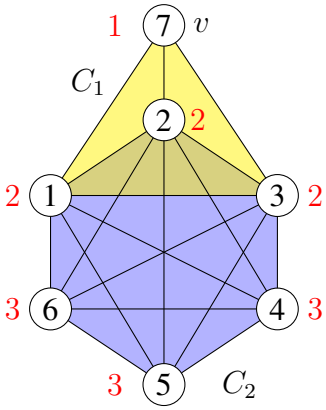
$x \in C_2 \setminus S$ satisfaz $R[x] \leq \begin{cases} k, & \text{se } x \in N(v) \\ k-1, & \text{do contrário} \end{cases}$, implicando que a estratégia de reduzir os requisitos dos vizinhos v em uma unidade e, em seguida, selecionar os $k-1$ vértices de maiores requisitos (atualizados para contemplar a presença de v no conjunto) está correta.

Se $v \notin S$ então $S \subseteq C_2$ e $k \geq R[v]$. Além disso, é possível assumir que os $R[v]$ vértices de maiores requisitos de C_2 estão em S (do contrário, seria possível trocar qualquer vértice por um de maior requisito sem alterar a cardinalidade de S) e que os demais $k-R[v]$ vértices de S podem ser selecionados de forma gulosa, como descrito no algoritmo.

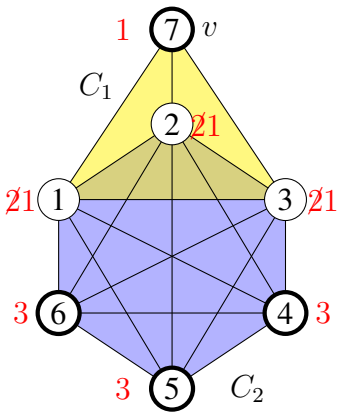
Como demonstrado, o procedimento utilizado para construir o conjunto R-dominante utilizado pelo Algoritmo 4.7 é capaz de identificar um conjunto mínimo, e, portanto, está correto.

Em relação à complexidade computacional, as ordenações utilizadas podem ser realizadas em tempo linear no número de vértices e as atualizações dos requisitos dos vértices também podem ser realizadas em $O(n)$. Desta forma, fica demonstrado que o Algoritmo 4.7 tem complexidade $O(n)$. □

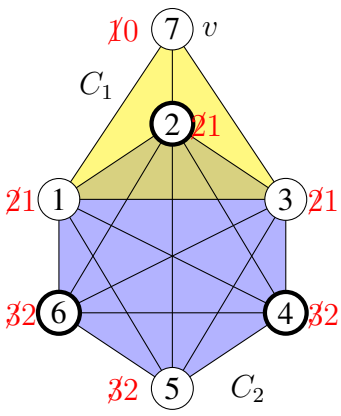
A Figura 4.6 apresenta um exemplo da aplicação do Algoritmo 4.7 em um grafo *split*-indiferença. Pode-se identificar que o grafo é constituído de duas cliques C_1 e C_2 e existe exatamente um vértice simplicial em C_1. Logo, este grafo pertence ao caso 2 do Teorema 2.3.9.



Grafo G e os requisitos dos vértices
$C_1 = \{1, 2, 3, 7\}$
$C_2 = \{1, 2, 3, 4, 5, 6\}$



$v \in S$
$S^1_{C_2} = \{4, 5, 6\}$
$S_1 = \{4, 5, 6, 7\}$



$v \notin S$
$A = \{2\}$ e $S^2_{C_2} = \{4, 6\}$
$\mathbf{S_2 = \{2, 4, 6\}}$

Figura 4.6: Exemplo de conjunto R-dominante para *split*-indiferença - Caso 2.

4.3.3 Caso 3a: Três cliques maximais tais que $S_{1,3} \cap S_{3,2} = \emptyset$

No terceiro caso descrito pelo Teorema 2.3.9, existem três cliques C_1, C_2 e C_3 tais que $|C_1 \setminus C_2| = |C_3 \setminus C_2| = 1$ e $S_{1,3} \cap S_{3,2} = \emptyset$. Tendo em vista que C_1 e C_3 não possuem vértices em comum, é possível utilizar o mesmo procedimento aplicado ao caso anterior, considerando não apenas um vértice simplicial, mas dois: $v = C_1 \setminus C_2$ e $w = C_3 \setminus C_2$.

É interessante observar que a remoção do vértice v (ou do vértice w) de um grafo do Caso 3a o transforma em um grafo que pertence ao Caso 2. Além disso, como $N(v) \cap N(w) = \emptyset$, os vértices necessários para dominar v são diferentes dos requeridos para dominar w, a inclusão de v (w) no conjunto não altera o conjunto necessário para dominar w (v) ou mesmo os requisitos dos vizinhos de w (v).

Essa relação possibilita que o algoritmo proposto para o Caso 2 seja aplicado ao Caso 3a, considerando sem perda de generalidade os casos em que o vértice v está ou não presente no conjunto. Desta forma, a solução para um grafo *split*-indiferença $G = (V, E)$ do Caso 3a e um vetor de requisitos R é o menor conjunto dentre dois casos:

$v \in S$ O vértice v é incluído no conjunto e os requisitos de seus vizinhos são decrementados de uma unidade. Seja R_1 o vetor de requisitos atualizados. Em seguida, o algoritmo para o Caso 2 é aplicado ao grafo $G[V \setminus \{v\}]$ (subgrafo induzido por $V \setminus \{v\}$), retornando o conjunto $S^1_{C_2 \cup C_3}$. O conjunto resultante neste caso é $S_1 = \{v\} \cup S^1_{C_2 \cup C_3}$.

$v \notin S$ A dominação de v demanda a inclusão de $R[v]$ vizinhos de v, logo seja A o conjunto dos $R[v]$ vizinhos de v de maiores requisitos. Vale ressaltar que, por definição do Caso 3a, $A \cap N(w) = \emptyset$. Em seguida, os requisitos dos vértices em $C_2 \setminus A$ são debitados em $|A|$ unidades. Seja R_2 o vetor de requisitos atualizado. Assim como no caso anterior, o algoritmo para o Caso 2 é aplicado ao grafo $G[V \setminus (\{v\} \cup A)]$, retornando o conjunto $S^2_{C_2 \cup C_3}$. O resultado deste caso é o conjunto $S_2 = A \cup S^2_{C_2 \cup C_3}$.

É interessante observar que o método descrito anteriormente avalia quatro possibilidades - as duas primeiras são consideradas pelo primeiro caso ($v \in S$), enquanto as últimas pelo caso $w \notin S$:

(a) $v \in S$ e $w \in S$;
(b) $v \in S$ e $w \notin S$;
(c) $v \notin S$ e $w \in S$;
(d) $v \notin S$ e $w \notin S$.

O Algoritmo 4.8 apresenta o procedimento destinado a este caso. A corretude e a complexidade computacional deste algoritmo estão demonstrados no Lema 4.3.2.

Algoritmo 4.8: *Split*-indiferença - Caso 3a: Dominação Vetorial

Entrada: Cliques C_1, C_2 e C_3 e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo S

Função `ResolverCaso3a` (C_1, C_2, C_3, R)
  Sejam v e w os vértices pertencentes respectivamente a $C_1 \setminus C_2$ e $C_3 \setminus C_2$
  $R_1 \leftarrow R_2 \leftarrow R$
  `/* Caso 1:` v `está no conjunto` R`-dominante */`
  para $u \in N(v)$ **faça** $R_1[u] \leftarrow \max(R_1[u] - 1, 0)$
  $S_1 \leftarrow \{v\} \cup$ `ResolverCaso2` (C_3, C_2, R_1)
  `/* Caso 2:` v `é dominado por seus vizinhos */`
  $A \leftarrow \emptyset$
  enquanto $R[v] > |A|$ **faça**
    Seja u o vértice de maior requisito em $N(v) \setminus A$
    $A \leftarrow A \cup \{u\}$
  para $u \in C_2$ **faça** $R_2[u] \leftarrow \max(R_2[u] - |A|, 0)$
  $S_2 \leftarrow A \cup$ `ResolverCaso2` $(C_3, C_2 \setminus A, R_2)$
  `/* Retornar o menor conjunto dentre` S_1 `e` S_2 `*/`
  se $|S_1| \leq |S_2|$ **então** $S \leftarrow S_1$ **senão** $S \leftarrow S_2$
  retorne S

Lema 4.3.2. *O Algoritmo 4.8 obtém um conjunto R-dominante para grafos split-indiferença que pertençam ao Caso 3a do Teorema 2.3.9 em tempo linear.*

Demonstração. Considerando que o método apresentado é baseado no Algoritmo 4.7, cuja corretude e complexidade já foram demonstrados, é possível afirmar que o Algoritmo 4.8 também está correto.

Assim como o Algoritmo 4.7, o método proposto também tem complexidade $O(n)$, pois a ordenação e as atualizações de requisitos podem ser realizadas em tempo linear. □

4.3.4 Caso 3b: $S_{1,3} \cap S_{3,2} \neq \emptyset$ e $|S_{1,2} \cup S_{3,2}| = |C_2|$

O quarto e último caso apresentado pelo Teorema 2.3.9 também é constituído por três cliques C_1, C_2 e C_3, tais que $|C_1 \setminus C_2| = |C_3 \setminus C_2| = 1$. A diferença do caso anterior é evidenciada pela existência de vértices comuns às cliques C_1 e C_3, isto é, $S_{1,2} \cap S_{3,2} \neq \emptyset$. A Figura 4.7 apresenta o Caso 3b. Nesta figura são apresentadas as diferentes regiões do

grafo. Além disso, os rótulos apresentados na figura serão utilizados para garantir uma padronização na forma com que cada região é referenciada ao longo desta subseção.

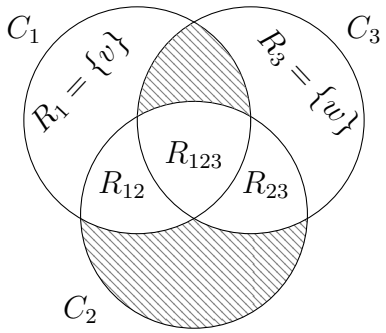
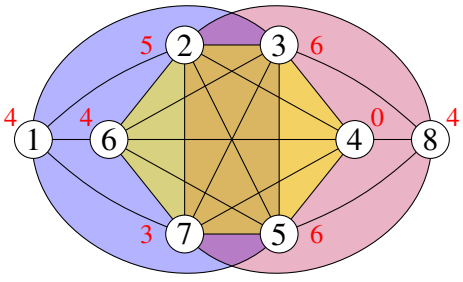


Figura 4.7: Representação do Caso 3b, onde $R_{12} = (C_1 \cap C_2) \setminus C_3$, $R_{23} = (C_2 \cap C_3) \setminus C_1$ e $R_{123} = C_1 \cap C_2 \cap C_3$.

Assim como o algoritmo anterior, a busca por uma solução ótima para o grafo consiste do estudo de quatro casos distintos que contemplam a presença de v e w no conjunto. Os casos em que v ou w (ou ambos) estão contidos no conjunto R-dominante podem ser tratados pelo Algoritmo 4.8, uma vez que v e w não são adjacentes. Contudo, o caso em que v e w não participam do conjunto R-dominante demanda um esforço adicional.

A necessidade do tratamento diferenciado para este caso está relacionada à presença de vértices adjacentes comuns a v e a w, uma vez que ao adicionar vizinhos de v no conjunto em construção, o requisito de w também é impactado. A Figura 4.8 demonstra um caso onde o Algoritmo 4.8, destinado ao caso 3a de grafos split-indiferença, retorna um resultado incorreto para um grafo relativo ao caso 3b. Na figura, pode-se observar que o Algoritmo 4.8 retornou o conjunto $\{2, 3, 5, 6, 7\}$ para o caso onde v e w não participam do conjunto R-dominante, uma solução sub-ótima. Já o algoritmo proposto para o caso 3b retorna o conjunto $\{2, 3, 5, 7\}$, uma solução ótima.



Algoritmo Caso 3a		
Caso 1:	$v \in S, w \in S$	$\{1, 2, 3, 5, 8\}$
	$v \in S, w \notin S$	$\{1, 2, 3, 5, 7\}$
Caso 2:	$v \notin S, w \in S$	$\{2, 3, 5, 6, 8\}$
	$v \notin S, w \notin S$	$\{2, 3, 5, 6, 7\}$

Algoritmo Caso 3b	
$v \notin S, w \notin S$	$\{2, 3, 5, 7\}$
$\vdots$	$\vdots$

Figura 4.8: Exemplo onde os resultados dos Algoritmos 4.8 e 4.13 divergem.

Sendo assim, uma nova abordagem foi desenvolvida especialmente para este caso. É importante destacar que a existência de adjacentes comuns aos vértices v e w não implica em consequências para os casos em que v ou w estão contidos no conjunto R-dominante, pois ao atualizar os requisitos dos vizinhos de v antes de selecionar o conjunto de vizinhos de w que será utilizado para dominar w, se garante a validade da ordenação empregada para selecionar os vértices de maiores requisitos (o mesmo vale para o caso oposto).

O algoritmo desenvolvido para este caso é formado por duas etapas: construção e aprimoramento. A fase de construção estabelece um conjunto capaz de R-dominar o grafo sem nenhuma garantia em relação a sua otimalidade, isto é, o conjunto construído pode não ser mínimo. Já a fase de aprimoramento, ao receber o conjunto construído, o torna mínimo executando remoções e trocas de vértices.

A construção de um conjunto R-dominante para o grafo está embasada na restrição inerente ao caso 3b do Teorema 2.3.9: Como $C_1 \cup C_3 = V$, então a união de conjuntos R-dominante para C_1 e C_3 é suficiente para R-dominar o grafo. Tendo em vista que os conjuntos R-dominante para C_1 e C_3 devem ser construídos sem permitir a adição de v ou de w, foi necessário estabelecer uma modificação em relação ao método exposto no Algoritmo 4.6 (para cliques e grafos completos). Essa modificação visa garantir que v (e w) será dominado, ao mesmo tempo em que é impedido de participar do conjunto retornado. Desta forma, o Algoritmo 4.9 apresenta o método de dominação de cliques atualizado. Dois pontos importantes devem ser destacados: O conjunto retornado pela função de construção descrita no Algoritmo 4.9 pode não ser mínimo e o vértice simplicial sempre será dominado, pois $R[v] \in \{0, \ldots d(v)\}$.

Algoritmo 4.9: *Split*-indiferença - Caso 3b: dominação de cliques modificado

Entrada: Vértices da clique V_C, Vetor de requisitos R e Vértice simplicial v
Saída: Conjunto R-dominante S para a clique

```
1 Função ResolverCliqueMod(V_C, R, v)
2     Seja L ← V \ {v} ordenada decrescentemente por requisitos
3     S ← ∅
4     enquanto L ≠ ∅ faça
5         Remover o primeiro elemento de L. Seja u esse vértice.
6         se (R[u] > |S|) ∨ (R[v] > |S|) então S ← S ∪ {u}
7     retorne S
```

Uma vez que o algoritmo responsável por definir um conjunto R-dominante para as cliques, respeitando as restrições apresentadas, foi demonstrado, o próximo passo é sim-

plesmente unir os conjuntos R-dominantes para C_1 e C_3. Sejam S_{C_1} e S_{C_3} os conjuntos R-dominantes para C_1 e C_3, respectivamente. O conjunto R-dominante S, retornado para o grafo, consiste da união entre S_{C_1} e S_{C_2}. O método empregado para construir S está detalhado na Função `Construir`, definida no Algoritmo 4.10.

Algoritmo 4.10: *Split*-Indiferença - Caso 3b: Fase 1 - Construção do conjunto S

Entrada: Cliques C_1, C_2 e C_3 e Vetor de Requisitos R
Saída: Conjunto R-dominante S

```
1 Função Construção(C1, C2, C3, R)
2     Sejam v e w os vértices pertencentes respectivamente a C1 \ C2 e C3 \ C2
3     S_C1 ←ResolverCliqueMod(C1, R, v)
4     S_C3 ←ResolverCliqueMod(C3, R, w)
5     retorne S ← S_C1 ∪ S_C3
```

A próxima fase do algoritmo é aprimorar S para torná-lo mínimo. Para isso, são empregados dois métodos distintos: remoções e trocas. Esses métodos são aplicados em passos distintos, ou seja, primeiro são removidos vértices desnecessários de S e, em seguida, quando nenhuma remoção é possível, as trocas são executadas. Tendo em vista essa aplicação em passos distintos, cada método será detalhado separadamente.

Na fase de remoção, cada vértice u pertencente ao conjunto S é analisado, seguindo uma ordenação crescente de requisitos, para avaliar se $S \setminus \{u\}$ ainda é um conjunto R-dominante. Em caso afirmativo, u é removido de S. Ao iterar sobre a ordenação estabelecida para os vértices de S, em caso de empate, o algoritmo opta por um vértice adjacente a v e a w. Essa opção visa minimizar o número de vértices em R_{123} (adjacentes a v e a w), pois essa área será explorada pela próxima fase (trocas). O Algoritmo 4.11 apresenta esse procedimento. Seja S' o conjunto resultante da fase de remoção. É trivial perceber que S' é minimal, uma vez que o término do algoritmo implica que nenhuma outra remoção é viável.

Uma vez que nenhuma remoção seja possível, o conjunto minimal retornado, S', é submetido a uma nova fase, que busca formas de trocar vértices. Uma troca é um movimento que substitui dois vértices em S' por um outro vértice não contido em S'. Para que uma troca seja viável, três condições devem ser satisfeitas:

1. São necessários dois vértices a_v e a_w tais que $a_v \in R_{12} \cap S'$ e $a_w \in R_{23} \cap S'$;
2. Todos os vértices de $(C_2 \setminus (S \setminus \{a_v, a_w\}))$ devem possuir requisitos estritamente menores que o tamanho de S';

Algoritmo 4.11: *Split*-indiferença - Caso 3b: Fase 2 - Remoção

Entrada: Grafo G, Vértices v e w, Vetor de requisitos R e Conjunto R-dominante S

Saída: Conjunto R-dominante minimal S'

```
1 Função Remoção(G, v, w, R, S)
2     Seja u_1, ..., u_k uma ordenação dos vértices de S tal que R[u_i] ≤ ... ≤ R[u_k]
        e u_i precede u_{i+1} se R[u_i] = R[u_{i+1}], u_i ∈ R_123 e u_{i+1} ∉ R_123
3     S' ← S
4     para (i ← 1;  i ≤ k;  i++) faça
5         se S' \ {u_i} é um conjunto R-dominante para G então
6             S' ← S' \ {u_i}
7     retorne S'
```

3. Existência de um vértice a_c tal que $a_c \notin R_{123} \setminus S'$, isto é, a_c deve ser um vizinho de v e de w não contido em S'.

Se essas três condições forem satisfeitas, a troca é executada: a_v e a_w são removidos e a_c é adicionado ao conjunto. Dentre todas as triplas de vértices (a_v, a_w, a_c) que satisfazem as condições apresentadas, a_v, a_w e a_c são escolhidos de forma a minimizar $R[a_v]$ e $R[a_w]$ e maximizar $R[a_c]$. A Figura 4.9 apresenta um exemplo do movimento de troca, situando todos os vértices participantes. Na figura, os vértices circulados estão contidos no conjunto em construção. É interessante observar que os vértices de C_2 perdem um vizinho em S', enquanto $|N(v) \cap S'|$ e $|N(w) \cap S'|$ permanecem inalterados.

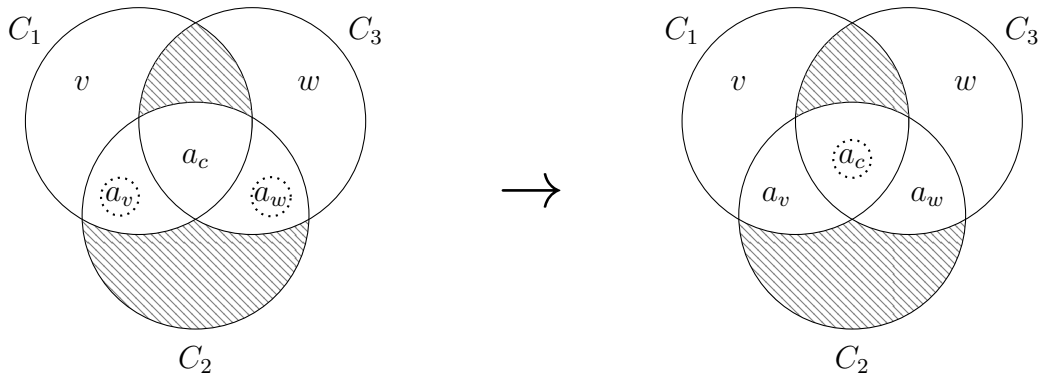


Figura 4.9: Movimento de Troca.

O processo de trocas é repetido até que nenhuma outra troca seja possível. O resultado deste processo é um conjunto R-dominante mínimo para G. O Algoritmo 4.12 apresenta o procedimento descrito.

Conforme afirmado anteriormente, o algoritmo é constituído de duas etapas: Constru-

Algoritmo 4.12: *Split*-indiferença - Caso 3b: Fase 2 - Trocas

Entrada: Grafo G particionado em cliques e regiões, Vetor de Requisitos R e Conjunto R-dominante minimal S'
Saída: Conjunto R-dominante mínimo S''

```
1  Função Trocas(C1, C2, C3, R, S)
2      Sejam v e w os vértices pertencentes respectivamente a C1 \ C2 e C3 \ C2
3      S'' ← S'
4      adj_v ← R12 ∩ S''; adj_w ← R23 ∩ S''  adj_vw ← R123 \ S'';
5      enquanto (|adj_v| > 0) ∧ (|adj_w| > 0) ∧ (|adj_vw| > 0) faça
6          remover de adj_v um vértice a_v tal que R[a_v] seja mínimo
7          remover de adj_w um vértice a_w tal que R[a_w] seja mínimo
8          se R[u] < |S''| para todo u ∈ C2 \ (S'' \ {a_v, a_w}) então
9              remover de adj_vw um vértice a_c tal que R[a_c] seja máximo
10             S'' ← (S'' \ {a_v, a_w}) ∪ {a_c}
11     retorne S''
```

ção de um conjunto R-dominante para o grafo (sem garantias de ser mínimo) e otimização do conjunto obtido através de remoções de vértices desnecessários e trocas de vértices. Para melhor visualização, o algoritmo foi dividido em três funções: `Construção`, `Remoção` e `Trocas`, apresentadas, respectivamente, nos Algoritmos 4.10, 4.11 e 4.12. Desta forma, o algoritmo destinado à obtenção de conjuntos R-dominantes mínimos para o Caso 3b de grafos split indiferença consiste em executar ordenadamente as funções apresentadas para o Caso a, onde v e w não participam do conjunto construído, e de executar o método descrito pelo Algoritmo 4.8 para os demais casos (onde v ou w participam do conjunto R-dominante). O conjunto R-dominante mínimo corresponde ao conjunto de menor cardinalidade obtido. O Algoritmo 4.13 apresenta esse procedimento detalhadamente. A corretude e a complexidade computacional deste método é demonstrada no Lema 4.3.3.

Lema 4.3.3. *O Algoritmo 4.13 obtém um conjunto R-dominante mínimo para grafos split-indiferença que pertençam ao Caso 3b do Teorema 2.3.9 em tempo linear.*

Demonstração. Esta demonstração será dividida em duas partes: os casos em que v, w ou ambos participam do conjunto R-dominante e o caso em que nenhum dos dois está incluído no conjunto. Para o primeiro caso, a corretude já foi estabelecida: o Algoritmo 4.8 utilizado nestes casos já foi provado correto pelo Lema 4.3.2. Desta forma, pode-se concluir que, para estes casos, o algoritmo está correto, isto é, se existe um conjunto R-dominante mínimo S que contenha v ou w (ou ambos) então S será retornado pelo Algoritmo 4.13 em tempo linear.

Algoritmo 4.13: *Split*-indiferença - Caso 3b: Dominação Vetorial

Entrada: Cliques C_1, C_2 e C_3 e Vetor de requisitos R
Saída: Conjunto R-dominante mínimo

Função `ResolverCaso3b` (C_1, C_2, C_3, R)
  Sejam v e w os vértices pertencentes respectivamente a $C_1 \setminus C_2$ e $C_3 \setminus C_2$
  $R_1 \leftarrow R_2 \leftarrow R$
  /* Construir conjunto R-dominante S_v com v */
  para $u \in N(v)$ **faça**
    $R_1[u] \leftarrow \max(R_1[u] - 1, 0)$
  $S_v \leftarrow \{v\} \cup$ `ResolverCaso2` (C_3, C_2, R_1)
  /* Construir conjunto R-dominante S_w com w */
  para $u \in N(w)$ **faça**
    $R_2[u] \leftarrow \max(R_2[u] - 1, 0)$
  $S_w \leftarrow \{w\} \cup$ `ResolverCaso2` (C_1, C_2, R_2)
  /* Construir conjunto R-dominante $\tilde{S}$ tal que $v \notin \tilde{S}$ e $w \notin \tilde{S}$ */
  $S \leftarrow$`Construção` (C_1, C_2, C_3, R)
  $S' \leftarrow$`Remoção` (G, v, w, R, S)
  $\tilde{S} \leftarrow$`Trocas` (C_1, C_2, C_3, R, S')
  retorne o conjunto de menor cardinalidade dentre S_v, S_w e $\tilde{S}$

Já o caso em que v e w não participam do conjunto R-dominante ainda precisa ser demonstrado, isto é, caso existam conjuntos R-dominante mínimos sem v e sem w, um destes será retornado pelo método descrito.

Seja S' o conjunto retornado pela Função *Remoção*, apresentada no Algoritmo 4.11, tendo como entrada o conjunto S, construído pela Função *Construção* (Algoritmo 4.10). Pode-se observar que S' satisfaz três propriedades:

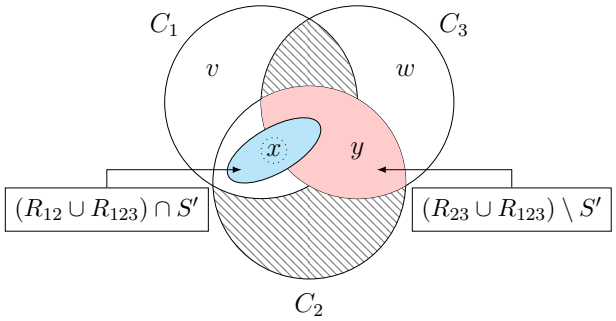
(i) S' é um conjunto R-dominante minimal (por inclusão), isto é, nenhum subconjunto próprio de S' pode R-dominar G;

(ii) Para cada par x, y tal que $x \in (R_{12} \cup R_{123}) \cap S'$ (respectivamente $(R_{23} \cup R_{123}) \cap S'$) e $y \in (R_{12} \cup R_{123}) \setminus S'$ (respectivamente $(R_{23} \cup R_{123}) \setminus S'$), a condição $R[x] \geq R[y]$ é satisfeita;

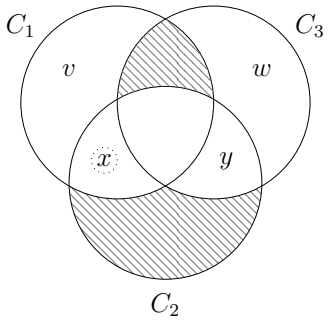
(iii) Se existir um par x, y tal que $x \in R_{12} \cap S'$ (respectivamente $R_{23} \cap S'$)) e $y \in R_{23} \setminus S'$ (respectivamente $R_{12} \setminus S'$) que satisfaça $R[x] < R[y]$, então $R[v] = |N(v) \cap S'|$ (respectivamente, $R[w] = |N(w) \cap S'|$).

A propriedade (i) segue trivialmente pela forma com que o Algoritmo 4.11 constrói S' a partir de S. A propriedade (ii) segue pela forma com que os conjuntos S_{C_1} e S_{C_3} são construídos e pela ordenação crescente seguida pelo algoritmo de remoções.

A ordenação empregada também garante a propriedade (iii): se um vértice de maior requisito é removido enquanto um vértice x de menor requisito é mantido em S' então é claro que existe um vértice $y \notin S'$ que impede a remoção de x, isto é, se x for removido de S' então y deixa de R-dominado por S'. Essa condição implica que $x \in N(y)$ e que $R[y] = |N(y) \cap S'|$. Entretanto, considerando que todos os vértices de C_2 tem os mesmos vizinhos em S' a única conclusão possível é que $y = v$ ou $y = w$, implicando que $R[v] = |N(v) \cap S'|$ ou que $R[w] = |N(w) \cap S'|$. A Figura 4.10 ilustra as propriedades (ii) e (iii), com a localização dos vértices x e y no grafo (Os casos simétricos foram omitidos).



Para cada par $x \in (R_{12} \cup R_{123}) \cap S'$ e $y \in (R_{23} \cup R_{123}) \setminus S'$ a condição $R[x] \geq R[y]$ é satisfeita.



Se existir um par $x \in R_{12} \cap S'$ e $y \in R_{23} \setminus S'$ tal que $R[x] < R[y]$ então $R[v] = |N(v) \cap S'|$.

Figura 4.10: Propriedades (ii) e (iii) satisfeitas pelo conjunto R-dominante minimal S'.

Para continuar a demonstração é necessário analisar os motivos que impedem novas remoções de S'. Seja a o vértice de menor requisito em S'. Duas possibilidades devem ser consideradas. Se $R[a] \geq |S'|$ então S' é mínimo: supondo um conjunto Y tal que $|Y| < |S'|$ e um vértice $y \in S' \setminus Y$; Como todo vértice em S' possui requisito maior ou igual à $|S'|$ então $R[y] \geq S' > Y$, implicando que $R[y] > |Y|$ e que y não é R-dominado por Y. Logo, não existe um conjunto R-dominante com menos elementos que S', provando que S' é de fato mínimo.

Por outro lado, se $R[a] < |S'|$ então deve existir um vértice $b \notin S'$ que impede a remoção de a, isto é, $R[b] = |N(b) \cap S'|$. Para continuar a demonstração algumas possíveis localizações de b no grafo devem ser analisadas.

Caso A - $b \in C_2$: neste caso, $S' \subseteq N(b)$, implicando que $R[b] = |S'|$. Ao analisar a localização do vértice a no grafo existem duas alternativas: $a \in R_{12}$ (o caso $a \in R_{23}$ é

análogo) ou $a \in R_{123}$. Como $R[b] = |S'|$ e $R[a] < |S'|$, o caso em que $a \in R_{123}$ viola a propriedade ii, pois $R[a] < R[b]$, $a \in S'$ e $b \notin S'$. Sendo assim, sem perda de generalidade, somente o caso $a \in R_{12}$ é válido. Neste caso, a propriedade (ii), implica que $b \in R_{23}$. Além disso, a propriedade (iii) estabelece que, nestas condições, $R[v] = |N(v) \cap S'|$. Como $b \notin S$ e $R[b] = |S'|$, a propriedade (ii) também garante que todo vértice $x \in (R_{23} \cup R_{123}) \cap S'$ satisfaz $R[x] \geq |S'|$.

Neste ponto da demonstração, as observações anteriores implicam que v impede a remoção de todos os vértices de $(R_{12} \cup R_{123}) \cap S'$ e que nenhum vértice em $(R_{23} \cup R_{123}) \cap S'$ pode ser dominado por um conjunto com menos elementos do que S'. Essa última conclusão pode ser justificada da seguinte forma: seja Y um conjunto R-dominante com menos elementos do que S'. Supondo que Y é um conjunto R-dominante mínimo (diferente de S') então existe um vértice $z \notin Y$ tal que $z \in (R_{23} \cup R_{123}) \cap S'$ ou $z \in R_{12} \cap S'$. Na primeira alternativa, como Y deve conter $R[z]$ vizinhos de z e $R[z] \geq |S'|$ então $|Y| \geq R[z] \geq |S'|$, garantindo que neste caso S' é mínimo. Por outro lado, como $z \in R_{12} \cap S'$ então Y deve conter $R[v]$ vizinhos de v. Contudo, como $R[v] = |N(v) \cap S'|$, novamente fica estabelecido que $|Y| \geq |S'|$. Implicando que se $b \in C_2$ então S' é de fato um conjunto R-dominante mínimo e, portanto, uma solução ótima.

Caso B - $b \notin C_2$: essa observação implica que $b = v$ ou $b = w$. Sem perda de generalidade, assume-se que $b = v$. Desta forma, como v impede a remoção de a então $a \in N(v) \cap S'$, implicando que $R[v] = |N(v) \cap S'|$ e que $a \in R_{12}$ ou $a \in R_{123}$.

Se neste ponto $adj_w = R_{23} \cap S' = \emptyset$ então $S' \subseteq N(v)$, o que implica que $R[v] = |S'|$ e que S' é uma solução ótima, pois não existe nenhum conjunto com menos elementos do que S' capaz de R-dominar v. Por outro lado, se $adj_w \neq \emptyset$, então seja a_w o vértice de menor requisito em adj_w. Existes duas alternativas sobre o requisito de a_w:

Caso B.1 - $R[a_w] \geq |S'|$: neste caso S' é uma solução ótima. Para demonstrar essa conclusão, supondo que S' não é mínimo, seja um conjunto R-dominante mínimo X tal que $v \notin X$ e $w \notin X$. Em seguida, particiona-se os conjuntos X e S' em subconjuntos distintos: $X \cap N(v)$, $X \cap R_{23}$, $S' \cap N(v)$ e $S' \cap R_{23}$. Como $|X| < |S'|$ então $|X \cap N(v)| + |X \cap R_{23}| < |S' \cap N(v)| + |S' \cap R_{23}|$. Além disso, como $R[a_w] \geq |S'|$ e $|S'| > |X|$, pode-se concluir que $R[a_w] > |X|$ e que todo vértice de adj_w deve estar contido em X, implicando que $|X \cap R_{23}| = |S' \cap R_{23}|$. Consequentemente, a expressão $|X \cap N(v)| + |X \cap R_{23}| < |S' \cap N(v)| + |S' \cap R_{23}|$ pode ser simplificada para $|X \cap N(v)| < |S' \cap N(v)|$. Contudo, conforme demonstrado anteriormente, $R[v] = |S' \cap N(v)|$, implicando que $|X \cap N(v)| < R[v]$ e

que, portanto, X não domina v. Desta forma, fica demonstrado que nenhum conjunto com menos elementos do que S' pode R-dominar G e que S' é uma solução ótima.

Caso B.2 - $R[a_w] < |S'|$: como $a_w \in S'$ e $a_w \notin N(v)$ então deve existir um vértice $y \notin S'$ tal que $S' \setminus \{a_w\}$ não é suficiente para R-dominar y (a remoção de a_w foi impedida pelo vértice y). É trivial notar que $y \neq v$, pois $a_w \notin N(v)$. Além disso, $y \notin C_2$ uma vez que: (a) se $y \in R_{23}$ ou $y \in R_{123}$ então a propriedade (ii) seria violada por a_w e y, já que neste caso $a_w \in R_{23} \cap S' \subseteq (R_{23} \cup R_{123}) \cap S'$, $y \in (R_{23} \cup R_{123}) \setminus S'$ e $R[a_w] < |S'| = R[y]$; (b) se $y \in R_{12}$ então a propriedade (ii) também seria violada por a e y, pois $a \in (R_{12} \cup R_{123}) \cap S'$ (a é o vértice cuja remoção foi impedida por v), $y \in (R_{12} \cup R_{123}) \setminus S'$ e $R[a] < |S'| = R[y]$. Portanto, pode-se concluir que a única possibilidade restante é $y = w$, o que implica que $R[w] = |N(w) \cap S'|$.

Vale destacar que até este ponto da demonstração foi concluído que $R[v] = |N(v) \cap S'|$ e $R[w] = |N(w) \cap S'|$. Para continuar, deve-se analisar mais precisamente a localização do vértice a (vértice de menor requisito contido no conjunto S', cuja remoção foi impedida pelo vértice v) no grafo. Como $a \in N(v)$, restam apenas duas possibilidades: $a \in R_{12} \cap S'$ e $a \in R_{123} \cap S'$.

Caso B.2.1 - $a \in R_{12} \cap S'$: a é candidato ao movimento de troca. Além de a, a_w também é candidato, pois é o vértice de menor requisito em $R_{23} \cap S'$. Se $R_{123} \setminus S' = \emptyset$ então nenhuma troca é possível e S' é uma solução ótima: $R_{123} \subseteq S'$ e não existe nenhum conjunto de menor cardinalidade capaz de R-dominar v ou w. Do contrário, seja a_c o vértice de menor requisito em $R_{123} \setminus S'$ e seja $S'' = (S' \setminus \{a, a_w\}) \cup \{a_c\}$. Como $R[a] < |S'|$ e $R[a_w] < |S'|$ então S'' é suficiente para R-dominar a e a_w. Além disso, é possível observar que S'' R-domina v e w: cada um perdeu um vizinho em R_{12} e R_{23} e ganhou um outro vizinho contido em R_{123}, logo seguem R-dominados. Desta forma, S'' é uma conjunto R-dominante, obtido por um movimento de troca, com menos elementos do que S'.

Caso B.2.2 - $a \in R_{123} \cap S'$: a não pode participar do movimento de troca. Contudo a_w continua sendo um candidato e, portanto, é necessário buscar por um candidato em $R_{12} \cap S'$. Caso não exista tal vértice ($R_{12} \cap S' = \emptyset$) então S' é ótimo, pois $S' \subseteq N(w)$ e $R[w] = |S'|$. Caso contrário, seja a_v o vértice de menor requisito em $R_{12} \cap S'$. O caso em que $R[a_v] \geq |S'|$ é análogo ao caso abordado anteriormente onde $R[a_w] \geq |S'|$, implicando na mesma conclusão: S' é mínimo. Por outro lado, se $R[a_v] < |S'|$ então a_v é candidato e o mesmo raciocínio demonstrado no caso anterior pode ser aplicado considerando $a = a_v$.

Como em ambos os casos (B.2.1 e B.2.2) o conjunto S'' obtido através de um movimento de troca satisfaz as propriedades (i), (ii) e (iii): A propriedade (i) continua satisfeita, uma vez que $R[v] = |N(v) \cap S''|$ e $R[w] = |N(w) \cap S''|$, implicando que qualquer remoção torna v ou w não-dominado. Considerando que o movimento de troca substitui os vértices de menores requisitos de $R_{12} \cap S'$ e $R_{23} \cap S'$ por um vértice de requisito máximo em $R_{123} \setminus S'$, a propriedade (ii) segue satisfeita. A propriedade (iii) também continua satisfeita, pois, conforme demonstrado anteriormente, $R[v] = |N(v) \cap S''|$ e $R[w] = |N(w) \cap S''|$.

Desta forma, como as propriedades (i), (ii) e (iii) seguem mantidas em S'', novos movimentos de troca podem ser aplicados iterativamente (linhas 5 - 10 do Algoritmo 4.12), preservando este invariante. Além disso, como o número de trocas é finito e limitado por $\min\{|R_{12} \cap S'|, |R_{23} \cap S'|, |R_{123} \setminus S'|\}$, então é seguro afirmar que a execução do Algoritmo 4.12 será finalizada em algum momento.

É essencial observar o ponto-chave desta demonstração: *ou existe pelo menos um movimento de troca, ou a solução é ótima, pois, em todos os possíveis casos onde o movimento de troca se mostrou impossível, o conjunto corrente se mostrou mínimo.* Portanto, pode-se concluir que o Algoritmo 4.13 está correto.

Complexidade computacional: A construção de S_v e S_w (linhas 4 - 9 do Algoritmo 4.13) pode ser realizada em tempo $O(n)$, conforme demonstrado no Lema 4.3.2. O método de construção do conjunto S, definido na linha 10 do Algoritmo 4.10, pode ser realizado em tempo $O(n)$, pois a ordenação pode ser realizada em tempo linear a cada vértice é processado uma única vez.

A função Redução, implementada no Algoritmo 4.11 também apresenta complexidade $O(n)$, pois a ordenação empregada na linha 2 pode ser realizada em tempo linear e a avaliação sobre a viabilidade de cada remoção (linha 5) pode ser otimizada utilizando algumas estruturas auxiliares, atualizadas durante a execução do algoritmo: um vetor booleano A_v tal que $A_v[u_i] = Verdadeiro$ se e somente se $u_i \in N(v) \cap S'$; um vetor boolean A_w tal que $A_w[u_i] = Verdadeiro$ se e somente se $u_i \in N(w) \cap S'$; contadores dinâmicos c_v e c_w utilizados para armazenar o número de elementos em A_v e A_w, respectivamente; um inteiro R_{max} utilizado para manter o maior requisito dentre os vértices contidos em $C_2 \setminus S'$. Todas as variáveis podem ser inicializadas em $O(n)$, pois demandam apenas um percurso entre os vizinhos de v e w (para construir A_v, A_w, c_v, c_w) e entre os vértices de C_2 para identificar o maior requisito. O teste de viabilidade da remoção, aplicado na linha 5 do

algoritmo, se resume em avaliar a expressão:

$$(R[u_i] < |S'|) \ \wedge \ (R_{max} < |S'|) \ \wedge \ (A_v[u_i] = \mathit{falso} \ \vee \ R[v] < c_v) \\ \wedge \ (A_w[u_i] = \mathit{falso} \ \vee \ R[w] < c_w).$$

Para concluir a análise da complexidade computacional da função Remoção é necessário demonstrar as atualizações efetuadas nas variáveis auxiliares. Toda vez que o algoritmo efetua a remoção de um vértice u_i da solução S' as variáveis são atualizadas em tempo $O(1)$: se $R[u_i] > R_{max}$ então a variável é atualizada: $R_{max} \leftarrow R[u_i]$. Além disso, se $u_i \in N(v)$ ($A_v[u_i] = \mathit{verdadeiro}$) então as variáveis associadas ao vértice v são corrigidas para contemplar a remoção de u_i: $A_v[u_i] \leftarrow \mathit{falso}$ e $c_v \leftarrow c_v - 1$. Essa mesma lógica vale para A_w e c_w.

Resta avaliar a complexidade da função Trocas (linha 12), apresentada no Algoritmo 4.12. O método de trocas também pode ser otimizado utilizando estruturas de dados auxiliares: os conjuntos adj_v, adj_w e adj_{vw} são implementados como filas ordenadas, tal que adj_v e adj_w seguem uma ordenação crescente de requisitos, enquanto adj_{vw} segue uma ordem decrescente. A inicialização destas filas pode ser realizada em tempo $O(n)$ e as remoções executadas nas linhas 6, 7 e 9, em tempo $O(1)$. Além das filas, o algoritmo também utiliza uma variável inteira R''_{max} para armazenar o maior requisito dos vértices em $(R_{12} \cup R_{23}) \setminus S''$.

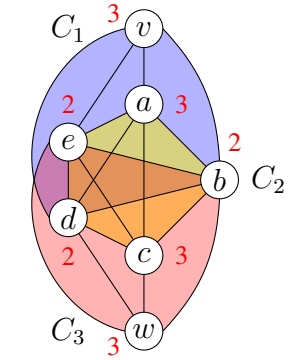
A cada iteração a_v e a_w são desenfileirados de adj_v e adj_w, respectivamente. Seja z o segundo elemento de adj_{vw} (caso adj_{vw} contenha apenas um elemento, z não existe e a última condição no teste de viabilidade pode ser ignorado). Sendo assim, o teste de viabilidade do movimento de troca (linha 6 do Algoritmo 4.12) pode ser avaliado em tempo $O(1)$ utilizando a seguinte expressão:

$$(R[a_v] < |S''|) \wedge (R[a_w] < |S''|) \wedge (R''_{max} < |S''|) \wedge (R[z] < |S''|).$$

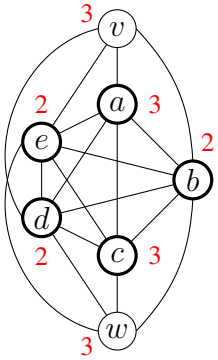
Após o movimento de trocas, as variáveis devem ser atualizadas: R''_{max} é atualizado em tempo $O(1)$ para $\max\{R''_{max}, R[a_v], R[a_w]\}$. Como o número de trocas é limitado por $\min\{\, |R_{123} \setminus S'|, |R_{12} \cap S'|, |R_{23} \cap S'| \,\} \leq n$, então o algoritmo de trocas pode ser executado em tempo $O(n)$.

Finalmente, como todas as etapas podem ser realizadas em tempo linear, pode-se concluir que a complexidade computacional do Algoritmo 4.13 é $O(n)$. □

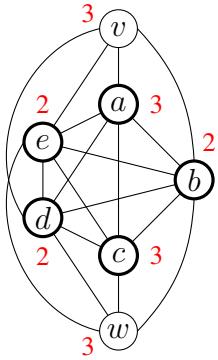
A Figura 4.11 apresenta um exemplo da aplicação do algoritmo para obtenção de um conjunto R-dominante. Deve-se notar que o grafo pertence ao quarto caso descrito no Teorema 2.3.9. Na figura, observa-se que inicialmente uma remoção foi realizada, pois o vértice b possui requisito inferior ao tamanho do conjunto, assim como todos os seus vizinhos. Em seguida, como nenhuma outra remoção é viável, a fase de trocas é iniciada. Como os vértices a, c e b atendem aos requisitos estabelecidos para execução de uma troca, este movimento é executado. Como não resta nenhum outro vértice que seja adjacente a v e w e não pertença ao conjunto, essa fase é terminada, retornando um conjunto R-dominante mínimo para os parâmetros fornecidos.



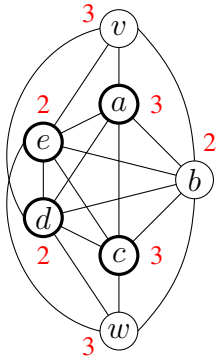
$C_1 = \{v, a, b, d, e\}$
$C_2 = \{a, b, c, d, e\}$
$C_3 = \{w, b, c, d, e\}$



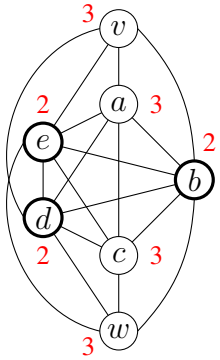
$S_{C_1} = \{a, b, e\}$
$S_{C_3} = \{b, c, d\}$
$S = \{a, b, c, d, e\}$



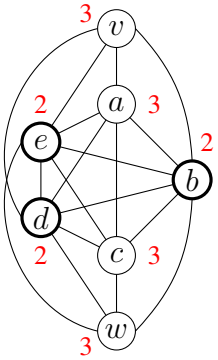
Redução:
b pode ser removido
$S' = \{a, \not{b}, c, d, e\}$



Fase de Trocas:
$S'' = \{a, c, d, e\}$
$a_v = a$ e $a_w = c$
$a_c = b$



Fase de Trocas:
$S'' = \{\not{a}, \not{c}, d, e\}$
Remover a e c
Adicionar b



Fim do Algoritmo:
$\tilde{S} = \{d, e, b\}$

Figura 4.11: Exemplo da aplicação do Algoritmo 4.13.

Teorema 4.3.1. *O Algoritmo 4.5 retorna um conjunto R-dominante para grafos split-indiferença em tempo linear.*

Demonstração. Tendo em vista que o algoritmo utiliza funções distintas para cada um

dos quatro casos apresentados pelos Teoremas 2.3.8 e 2.3.9 e que as corretudes destas funções foram demonstradas pelo Teorema 3.2.1 e pelos Lemas 4.3.1, 4.3.2 e 4.3.3, pode-se assegurar que o algoritmo está correto. Além disso, como todas as funções podem ser executadas em tempo linear, pode-se concluir que a complexidade computacional do Algoritmo 4.5 é $O(n)$. □

O algoritmo de tempo $O(n)$ para grafos *split*-indiferença foi uma das contribuições deste trabalho. Este resultado foi publicado na forma de um artigo completo na *Information Processing Letters* [65].

4.3.5 Conclusões

Este capítulo apresentou três contribuições deste trabalho. A primeira é um algoritmo de tempo linear para grafos *split*-indiferença, que explora a estrutura desta classe através da decomposição em casos baseados na presença ou não de vértices simpliciais. O segundo algoritmo foi destinado aos grafos conexos com exatamente duas cliques maximais. Além desses resultados, este capítulo propôs também um algoritmo aproximativo para o problema da dominação vetorial em grafos dominó $\cap$ intervalo próprio. O estudo desta classe possibilitou um melhor entendimento da dinâmica do problema em relação a decomposição do grafo em cliques maximais.

Vale observar que este capítulo abordou problemas de propagação onde todos os vértices devem ser contaminados em uma única iteração. Uma pergunta interessante que resultou desta análise é o impacto da flexibilização desta regra, isto é, a admissão de múltiplas iterações do processo de contágio. Desta forma, o Capítulo 5 abordará o problema da seleção de alvos, onde o grafo deve ser totalmente contaminado ao longo de múltiplas iterações, tal que cada vértice contaminado em uma iteração ajuda no contágio de seus vizinhos nas iterações seguintes. Entretanto, vale destacar que a abordagem do Capítulo 5 difere da atual, pois é baseada em métodos aproximativos e voltada a instâncias da literatura que mapeiam redes sociais.

Capítulo 5

Problema da Seleção de Alvos

Dado um grafo $G = (V, E)$ e um vetor de números inteiros naturais R, onde para todo vértice $v \in V$ vale $0 \leq R[v] \leq d(v)$, o problema da seleção de alvos consiste em localizar um conjunto mínimo $S \subseteq V$ que satisfaça a seguinte condição (regra da dominação): $S_n = V$, onde $S_0 = S$ e $S_{i+1} = S_i \cup \{w \in V \mid |N(w) \cap S_i| \geq R[w]\}$. Como pode ser observado, a regra da dominação estabelece que o processo da dominação é irreversível, pois quando um vértice v é adicionado a um conjunto S_i, v estará contido em todo conjunto S_k, onde $k \geq i$. Essa observação implica que o processo da dominação estabilizará em alguma iteração $i \leq n$: se um vértice for dominado a cada iteração, todos os vértices serão dominados em n iterações.

É interessante observar que o problema da seleção de alvos pode ser utilizado para modelar computacionalmente a propagação de doenças, influência e opiniões, como proposto por Kempe *et al.* [57], Friedkin [42]. Recentemente, os trabalhos de Kempe *et al.* [57], Dreyer e Roberts [38], Chen [16], Cicalese *et al.* [18], Cordasco *et al.* [20] e Cordasco *et al.* [21] abordaram o problema em instâncias que mapeiam redes sociais. Um problema interessante neste escopo é a propagação de *fake news* em redes sociais, que pode ser interpretado como estabelecer um conjunto mínimo de usuários de uma rede social que devem ser inicialmente contaminados (ou convencidos) por uma falsa notícia para que ela se propague (ou viralize) por toda a rede (ou uma grande parte dela).

Tal como no problema da dominação vetorial, existem variações do problema da seleção de alvos, onde, por exemplo, se limita o número máximo de iterações necessárias para que o grafo seja dominado. Vale destacar que se a dominação deve ocorrer em uma única iteração, o problema se resume ao problema da dominação vetorial, abordado nos Capítulos 3 e 4.

Outra variação assume que todos os requisitos são definidos por uma constante k, isto é, $R[v] = k, \forall v \in V$. O trabalho de Dreyer e Roberts [38] estabeleceu que para qualquer $k \geq 3$, o problema da seleção de alvos em sua versão de decisão é $\mathcal{NP}$-Completo. Em um trabalho mais recente, Centeno *et al.* [13] demonstrou que o problema da seleção de alvos continua $\mathcal{NP}$-Completo mesmo quando $k = 2$. Vale observar que se $k = 1$ então a solução para o problema é trivial: um único vértice do grafo (supondo que o grafo é conexo) é capaz de dominar os demais em no máximo n iterações. Neste mesmo trabalho, Centeno *et al.* [13] apresentou um algoritmo para solucionar o problema da seleção de alvos em algumas famílias de grafos, como por exemplo, grafos bloco e árvores.

Outra variação interessante é o Problema da Vacinação [75], que busca determinar um conjunto mínimo capaz de impedir que outros conjuntos dominem todo o grafo. Estes conjuntos são chamados de vacinas pois inibem a propagação da doença ou da influência no grafo.

Este capítulo é voltado para o problema da dominação vetorial em grafos em geral. Considerando que o problema é $\mathcal{NP}$-Completo [57], uma das alternativas para obter uma solução viável em uma quantidade aceitável de tempo é o uso de métodos aproximativos, como, por exemplo, meta-heurísticas. Desta forma, este capítulo apresentará um algoritmo genético para o problema da seleção de alvos.

5.1 Métodos Aproximativos da Literatura

No campo das soluções aproximadas, é possível destacar os trabalhos de Chen [16], Dinh *et al.* [32] e Cordasco *et al.* [20, 21]. Dentre os métodos apresentados o algoritmo proposto por Cordasco *et al.* [20, 21] será formalmente apresentado, pois, como será visto na Seção 5.2, é utilizado como ponto de partida para o algoritmo genético proposto neste capítulo.

O algoritmo guloso proposto por Cordasco *et al.* [20, 21] difere de todos os outros métodos gulosos já apresentados nesta tese. Enquanto os outros escolhem um vértices e o adicionam ao conjunto em construção, o método apresentado define quais vértices serão dominados por seus vizinhos e os remove do grafo. Esse processo é repetido até atingir o ponto em que um vértice deve ser obrigatoriamente incluído no conjunto, pois o conjunto de seus vizinhos que ainda permanecem no grafo não é mais suficiente para o dominar (seu grau se tornou menor do que seu requisito). Vale destacar que o critério para selecionar

quais vértices serão dominados por seus vizinhos é puramente guloso, embasado na razão entre o o requisito e o grau de cada vértice.

O método descrito utiliza três variáveis auxiliares para cada vértice:

$\boldsymbol{A[v]}$ Vizinhos de v que ainda permanecem no grafo;
$\boldsymbol{\delta[v]}$ Número de vizinhos que o vértice v ainda possui no grafo ($\delta[v] = |A[v]|$);
$\boldsymbol{k[v]}$ Requisito do vértice v.

A cada iteração, as variáveis são atualizadas para contemplar a ação executada. Existem três casos possíveis executados em cada passo do algoritmo. No primeiro caso, existe um vértice v tal que $k[v] = 0$, isto é, v não demanda mais nenhum vizinho para ser dominado. A dominação de v implica que os requisitos de seus vizinhos devem ser atualizados: $k[u] = \max(k[u] - 1, 0)\ \forall u \in A[v]$. Vale destacar que no problema da seleção de alvos, vértices dominados colaboram para a dominação de seus vizinhos, mesmo que não estejam contidos no conjunto solução. Por último, o vértice v é removido do grafo (linhas 15-18 do Algoritmo 5.1).

O segundo caso possível (dada a inexistência de um vértice que atenda ao caso anterior) considera que existe um vértice v que possui menos vizinhos no grafo do que os necessários para sua dominação ($\delta[v] < k[v]$). Neste caso, a única possibilidade para assegurar sua dominação é incluir o próprio vértice v no conjunto solução S. Assim como no caso anterior, os requisitos dos vizinhos são atualizados e v é removido do grafo.

Já no último caso (novamente, supondo que não existem vértices que atendam aos casos anteriores), um vértice v é escolhido de forma gulosa para ser dominado por seus vizinhos. Diferentemente dos casos anteriores, os requisitos dos vizinhos não são atualizados, pois v ainda não foi efetivamente dominado, apenas marcado como tal. Por fim, v é removido do grafo.

Como em todos os casos, o algoritmo remove um vértice, pode-se considerar que o algoritmo é finito. Além disso, vale destacar novamente que o algoritmo escolhe vértices que serão dominados por seus vizinhos e os remove do grafo até atingir o ponto em que um vértice não possua vizinhos o suficiente para ser dominado, isto é, atenda ao caso 2. Este vértice será então adicionado ao conjunto em construção. O Algoritmo 5.1 demonstra o método descrito. A corretude deste método é demonstrada por Cordasco *et al.* [20, 21].

A Figura 5.1 ilustra a execução do grafo em uma instância gerada aleatoriamente. O

Algoritmo 5.1: Algoritmo Cordasco *et al.* [20, 21] para o problema da seleção de alvos

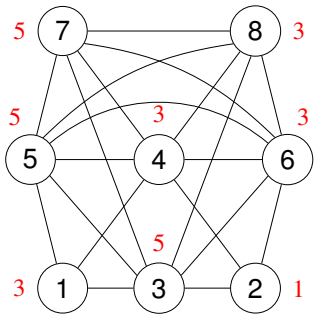
Entrada: Grafo $G = (V, E)$ e Vetor de requisitos R
Saída: Conjunto de alvos S

```
1  S ← ∅        U ← V
2  para cada v ∈ V faça
3      A[v] = N(v)        δ[v] = |A[v]|        k[v] = R[v]
4  enquanto U ≠ ∅ faça
5      se ∃v ∈ U | k[v] = 0 então
           /* Caso 1: O vértice v está dominado                    */
6          para cada u ∈ A[v] faça
7              k[u] ← max(k[u] − 1, 0) ;          /* v influencia seus
                adjacentes em U */
8      senão
9          se ∃v ∈ U | δ[v] < k[v] então
               /* Caso 2: O vértice v deve ser adicionado ao
                  conjunto S, pois seus vizinhos em U não são
                  suficientes para garantir sua dominação   */
10             S ← S ∪ {v}
11             para cada u ∈ A[v] faça
12                 k[u] ← max(k[u] − 1, 0) ;          /* v influencia seus
                    adjacentes em U */
13         senão
               /* Caso 3: O vértice v será dominado por seus
                  vizinhos                                          */
14             v ← argmax_{u∈U} { k[u] / ((δ[u])(δ[u]+1)) }
           /* Remover o vértice v do grafo                         */
15     para cada u ∈ A[v] faça
16         δ[u] ← δ[u] − 1
17         A[u] ← A[u] \ {v}
18     U ← U \ {v}
19 retorne S
```

valor ϕ da figura corresponde a expressão $\left\{\frac{k[u]}{(\delta[u])(\delta[u]+1)}\right\}$, utilizada no caso 3 do algoritmo para escolher o vértice que será dominado por seus vizinhos. Os valores de k, δ e A são apresentados graficamente na figura: k é o número vermelho ao lado de cada vértice, A e δ são expostos como os vizinhos de cada vértice. Cada alteração demandada pelo algoritmo é executada no grafo. Após cada iteração, os vértices que foram escolhidos e removidos pelo caso 3 foram grafados em verde, enquanto os adicionados ao conjunto em construção pelo caso 2 estão em azul. Para melhor visualização do caso 3, quando $k[v] > \delta[v]$, as arestas que compõe o valor $\delta[v]$ foram destacadas em vermelho.

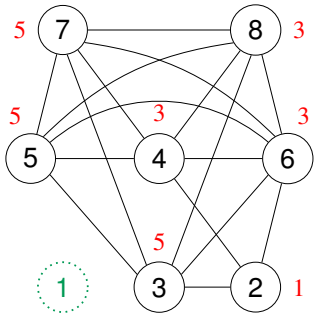
Como apontado anteriormente, o comportamento do algoritmo, baseado em selecionar vértices que serão dominados por seus vizinhos, pode ser facilmente identificado na Figura 5.1. Nas primeiras iterações, os vértices 1 e 3 foram removidos e marcados como dominados por seus vizinhos (Caso 3 do algoritmo) sem que nenhum vértice tenha sido efetivamente adicionado ao conjunto em construção. Na iteração seguinte, o vértice 7 é adicionado, já que possui $k > \delta$ ($5 > 4$). Esse comportamento difere de todos os algoritmos já apresentados, que selecionam quais vértices irão compor o conjunto solução. O Algoritmo 5.1 possui complexidade de tempo $O(n^2)$, pois demanda a atualização dos vizinhos do vértice processado em cada iteração. Outro ponto que deve ser destacado é a falta de critérios de desempate. Na terceira iteração, onde o vértice 7 foi adicionado, nota-se que o vértice 5 também atendia aos critérios do caso 2: $k > \delta$ ($5 > 4$). Nestes casos, a seleção segue critérios puramente randômicos, como visto no exemplo.

Vale observar que o resultado obtido pelo Algoritmo 5.1 ($S = \{5, 7, 8\}$) não é ótimo. A execução de um algoritmo *backtracking* nesta instância retornou o conjunto $S_{otm} = \{3, 5\}$. Embora o algoritmo não retorne soluções ótimas, seus resultados oferecem um bom ponto de partida para outros métodos, como, por exemplo, o algoritmo genético proposto na Seção 5.2.



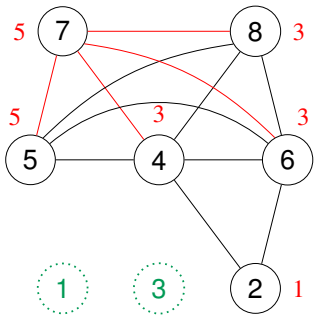
v	ϕ
1	**0,25**
2	0,08
3	0,12
4	0,07
5	0,12
6	0,07
7	0,17
8	0,1

↳ Caso 3: $v = 1$

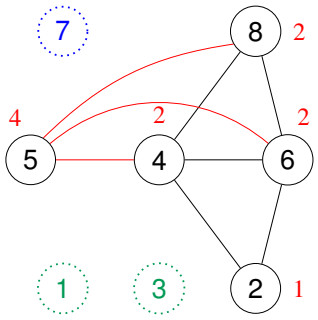


v	ϕ
2	0,08
3	**0,17**
4	0,1
5	0,17
6	0,07
7	0,17
8	0,1

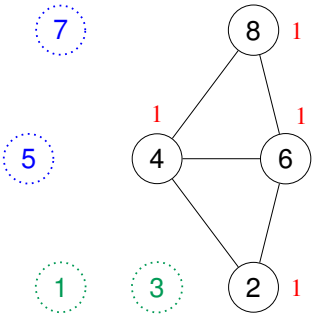
↳ Caso 3: $v = 3$



↳ Caso 2: $v = 7$

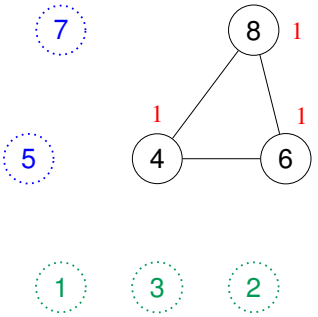


↳ Caso 2: $v = 5$



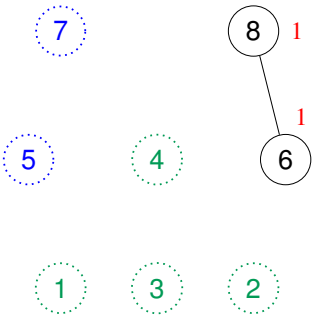
v	ϕ
2	**0,17**
4	0,08
6	0,08
8	0,17

↳ Caso 3: $v = 2$



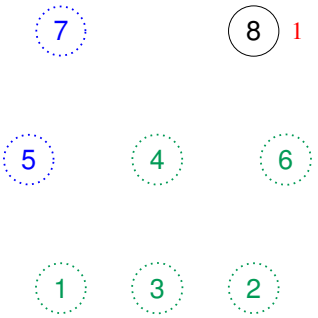
v	ϕ
4	**0,17**
6	0,17
8	0,17

↳ Caso 3: $v = 4$



v	ϕ
6	**0,5**
8	0,5

↳ Caso 3: $v = 6$



↳ Caso 2: $v = 8$

Fim: $S = \{5, 7, 8\}$

Figura 5.1: Exemplo de Aplicação do Algoritmo 5.1.

5.2 Algoritmo Genético

Um algoritmo genético é um algoritmo de inspiração biológica, que implementa a ideia de simular computacionalmente várias gerações de uma população, formada por indivíduos distintos. Cada indivíduo representa uma solução para o problema tratado. Por exemplo, no problema da seleção de alvos, os genes de um indivíduo mapeiam um conjunto de vértices do grafo. Um indivíduo é viável se seus genes corresponderem a um conjunto capaz de dominar o grafo. A reprodução desses indivíduos cria novas gerações, formadas por permutações dos genes da geração anterior.

O algoritmo proposto nesta seção usa as informações do censo dos indivíduos construídos (tal como o censo demográfico executado em um país ou região) para aperfeiçoar o algoritmo genético. Como no mundo real, a ideia de um censo é adquirir e armazenar informações sobre a população em um determinado momento. Os dados coletados são então usados pelo algoritmo como uma forma de assegurar uma maior diversidade de soluções e também para favorecer uma avaliação mais precisa da qualidade das soluções. Dessa forma, dois tipos diferentes de censo são armazenados: $\mathrm{S\text{-}Censo}$ e $\mathrm{V\text{-}Censo}$. O primeiro é usado para acompanhar quantas vezes um indivíduo foi detectado na população, enquanto o segundo armazena o número de vezes que cada vértice foi incluído em uma solução.

Além do uso de informações do censo, o algoritmo proposto também possui a capacidade de se auto adaptar, permitindo um ajuste na agressividade de cada operação, de modo a escapar de soluções sub-ótimas locais.

Um conjunto de vértices é chamado de uma solução viável quando todo o grafo é dominado após n iterações da regra da dominação. Essa característica do problema implica que a verificação da viabilidade de uma solução requer a execução de cada uma dessas iterações, resultando em um alto custo computacional, especialmente em grafos de grandes dimensões. Como esse procedimento deve ser executado em cada indivíduo construído pelo algoritmo, o custo computacional se torna um ponto crítico, especialmente ao se considerar que algoritmos genéticos geralmente requerem a simulação de muitos indivíduos ao longo de várias gerações. Para minimizar o custo acumulado das verificações de viabilidade, o algoritmo foi construído de forma a admitir sua execução em um ambiente paralelizado.

O principal objetivo da paralelização do algoritmo é executar cada verificação de viabilidade em um processador diferente, uma vez que isto dilui o tempo necessário para

concluir cada geração. Para usufruir ainda mais do ambiente multiprocessado, os métodos de reprodução e parte da avaliação do *fitness* de cada indivíduo também são executados em paralelo. O termo *fitness* é utilizado no jargão dos algoritmos genéticos como uma forma de expressar a qualidade de cada indivíduo, isto é, indivíduos de maior *fitness* correspondem a soluções melhores para o problema. Assim como na natureza, os indivíduos de maior aptidão possuem maior probabilidade de serem os ancestrais de uma nova geração. No caso do problema da seleção de alvos, pode-se dizer (de forma simplificada) que quanto menor o conjunto que um indivíduo representa, maior será seu *fitness*. A utilização de um ambiente paralelizado permite que o algoritmo seja executado de forma independente em cada processador, exceto quando é necessário consolidar uma nova geração ou avaliar a mesma.

O algoritmo genético proposto neste capítulo é aprimorado por três características principais:

- Uso de informações do censo para orientar o algoritmo a realizar melhores escolhas;
- Uso de um parâmetro auto-adaptável para ajustar a agressividade de cada método de reprodução e mutação;
- Uso de um ambiente paralelizado para minimizar o tempo gasto na reprodução, avaliação e, especialmente, no controle da viabilidade dos indivíduos.

O objetivo do uso das informações do censo é melhorar as escolhas que o algoritmo genético executa: quais indivíduos serão reproduzidos e quais vértices devem ser selecionados para minimizar o conjunto resultante ou elevar a diversidade de indivíduos na população. O algoritmo proposto usa duas informações distintas do censo.

O primeiro (S-Censo) está relacionado aos indivíduos, ou seja, conjuntos inteiros de vértices. Cada vez que um novo indivíduo (nunca identificado antes) é encontrado, ele é adicionado à tabela com o valor 1, representando que esse indivíduo foi localizado apenas uma vez. Se um indivíduo encontrado anteriormente for construído mais uma vez, o valor correspondente a ele na tabela é incrementado em uma unidade. Essas informações são utilizadas para reduzir a chance de um indivíduo recorrente ser selecionado como ancestral da próxima geração, fornecendo uma maior diversidade ao algoritmo. O uso do parâmetro S-Censo será detalhado na Subseção 5.2.3.

As informações do segundo censo (V-Censo) tratam de vértices e representam quantas vezes cada vértice foi incluído em um conjunto representado por um indivíduo. Essas informações são utilizadas pelo algoritmo para elevar a probabilidade de que vértices pouco

explorados sejam incluídos em novas soluções. Sem o uso desse artifício, o algoritmo seguiria um comportamento puramente guloso, ao incluir apenas vértices que são imediatamente boas escolhas.

Os dois censos são usados como componentes na tomada de decisão, mas não sozinhos. Para especificar os pesos de cada componente na decisão, quatro parâmetros orientam o algoritmo: $wSCenso$, $wTam$, $wGrau$, $wVCenso$. Estes parâmetros serão detalhados ao longo das Subseções 5.2.3 e 5.2.4.

Além desses parâmetros, um outro é utilizado para especificar a agressividade de cada operador de reprodução e mutação, representando o número esperado de alterações que cada operador deve executar em cada indivíduo. Este parâmetro é usado para conceder ao algoritmo a capacidade de escapar de soluções sub-ótimas que poderiam levar ao encerramento prematuro do algoritmo. A agressividade do algoritmo é ajustada pelo parâmetro δ, que é incrementado toda vez o algoritmo encerra uma iteração sem encontrar uma solução melhor do que a identificada nas iterações anteriores.

Para garantir que o algoritmo não perca boas soluções entre gerações, um terço dos indivíduos na última geração são copiados para a próxima. Os indivíduos copiados correspondem àqueles com maior *fitness*, o que adiciona um comportamento nitidamente elitista ao algoritmo. Os dois terços restantes são obtidos através da reprodução de indivíduos contidos na geração anterior. A Subseção 5.2.4 detalhará este procedimento, incluindo o valor inicial de δ para cada instância.

Como mencionado anteriormente, o algoritmo proposto foi projetado para usufruir dos benefícios de um ambiente paralelizado, executando algumas de suas funções de forma concomitante e distribuída por diversos processadores. A paralelização do algoritmo foi implementada utilizando o protocolo MPI [73]. Cada processador no ambiente é identificado exclusivamente por um número inteiro $P \in \mathbb{N}^*$. Vale destacar que no protocolo MPI, cada processador possui sua própria memória principal, isolada dos demais, e processa uma cópia local do algoritmo, executada em paralelo em todos os processadores. Para os processadores trocarem informações entre si são necessárias rotinas especiais de comunicação, que algumas vezes exigem uma sincronização entre eles. Essa sincronização é realizada forçando que cada processador aguarde em um ponto do algoritmo até que todos os demais alcancem esse mesmo ponto, acarretando um possível desperdício no tempo de processamento. Esse desperdício pode ser evitado reduzindo a comunicação entre os processadores e uniformizando o tempo necessário para executar as tarefas associadas a

cada processador.

No início do algoritmo, cada processador recebe a tarefa de construir três novos indivíduos viáveis de forma independente. Após a construção, a geração B_0 é formada, reunindo os indivíduos dos processadores. Em seguida, o algoritmo avaliará o *fitness* de cada indivíduo em B_0, concluindo o processamento da geração inicial.

A partir de uma geração B_i, o algoritmo construirá uma nova geração B_{i+1}: primeiro, cada processador seleciona dois indivíduos P_1 e P_2 de B_i. Em seguida, P_1 e P_2 se reproduzem, gerando dois novos indivíduos S_1 e S_2. Como o procedimento de reprodução contém fatores aleatórios, cada processador deve garantir de forma independente que S_1 e S_2 sejam soluções viáveis (isto é, capazes de dominar todo o grafo após algumas iterações). Em seguida, cada processador P copia o P-melhor indivíduo S_3 de B_i. Finalmente, a nova geração é formada pelos indivíduos S_1, S_2 e S_3 de cada processador, e o *fitness* de cada indivíduo é avaliado. Este processo é repetido até que as condições de parada sejam atendidas.

A Figura 5.2 ilustra como o algoritmo genético foi paralelizado. Na figura, as caixas azuis são executadas em paralelo por cada processador, enquanto as vermelhas representam tarefas que exigem comunicação e sincronização entre os processadores. Pode-se notar que as caixas correspondentes à avaliação dos indivíduos são grafadas nas duas cores, pois possuem dois momentos: no primeiro, o tamanho de cada solução é calculada em paralelo; em seguida, a comunicação entre os processadores é exigida para calcular a média e o desvio padrão dos tamanhos das soluções, necessários para o escalonamento sigma (*sigma scaling*). A avaliação do *fitness* de cada indivíduo e o escalonamento sigma serão detalhados na Subseção 5.2.3.

Como todos os principais recursos foram apresentados, os componentes do algoritmo podem ser detalhados separadamente. A Subseção 5.2.1 exibe como cada conjunto é representado como um indivíduo. A seguir, a Subseção 5.2.2 apresenta a construção da geração inicial. Na Subseção 5.2.3, a avaliação do *fitness* de cada indivíduo é detalhada, considerando o restante da população. Finalmente, na Subseção 5.2.4, a fase de reprodução, na qual novos indivíduos são construídos a partir da geração anterior, é apresentada. A Subseção 5.2.4 também apresenta o procedimento de otimização, no qual a viabilidade de cada indivíduo é assegurada.

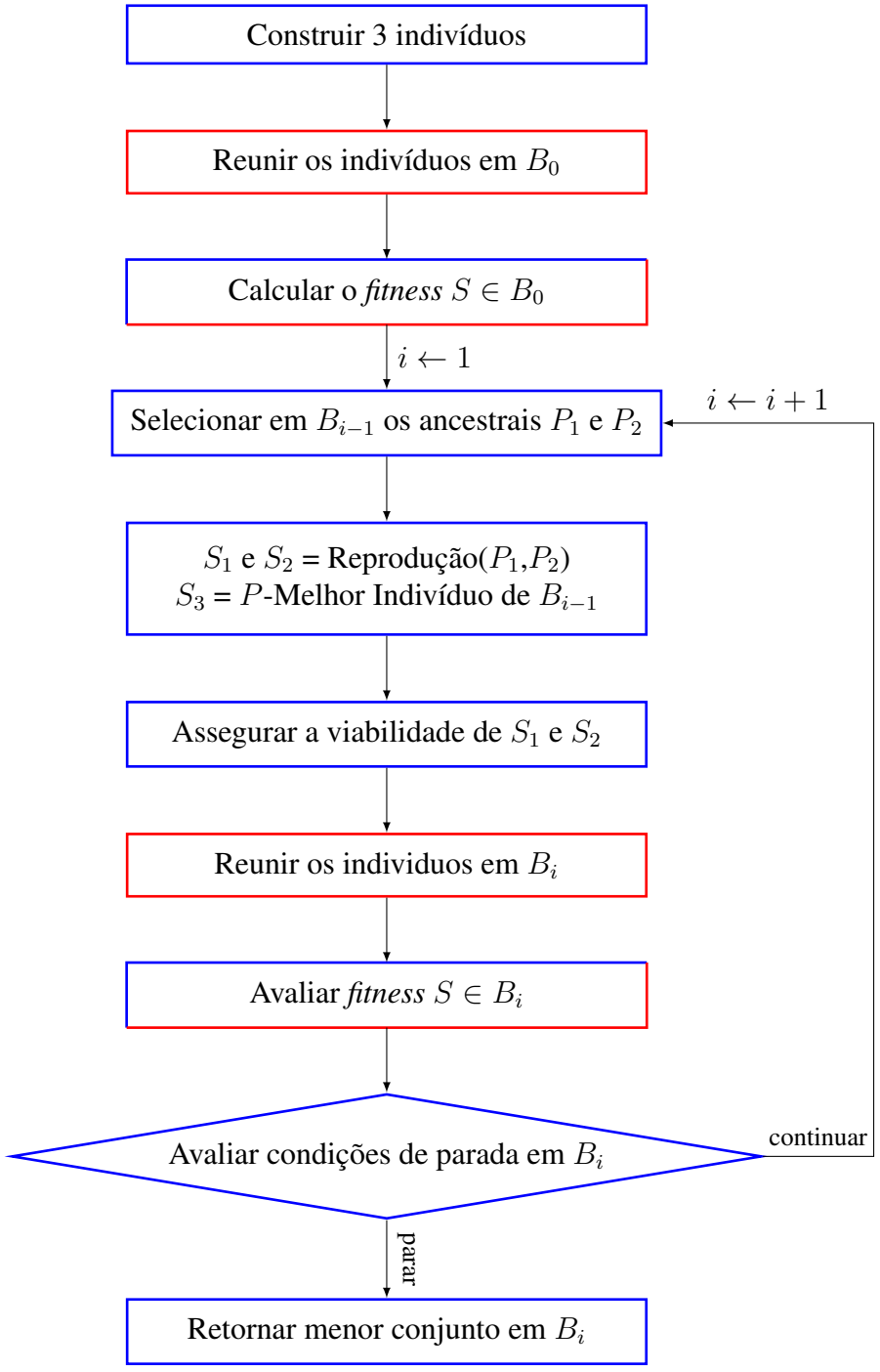


Figura 5.2: Arcabouço da paralelização aplicada ao algoritmo genético.

5.2.1 Representando Conjuntos como Indivíduos

Cada conjunto de vértices é representado como um vetor de booleanos com n posições. Cada posição do vetor é associada a um vértice v do grafo: o vértice v está incluído no conjunto se, e somente se, a posição correspondente a v no vetor contiver uma atribuição verdadeira.

Vale observar que essa representação usa ligeiramente mais memória do que outras opções, que contém apenas os vértices que estão no conjunto correspondente. No entanto, a representação implementada permite que o algoritmo identifique em tempo $O(1)$ se um vértice está ou não contido no conjunto, o que acelera a execução dos métodos de reprodução. A adição e a remoção de vértices ao conjunto também é trivial, pois consiste de simplesmente alterar a atribuição do valor na posição correspondente ao vértice. Uma vantagem adicional desta representação é a facilidade para transmitir os indivíduos entre diferentes processadores, uma vez que todos os vetores apresentam o mesmo tamanho. Para exemplificar o uso de memória decorrente desta representação, supondo o maior grafo abordado neste capítulo (1.138.499 vértices), cada individuo construído durante o processamento do grafo pode demandar de mais de um megabyte de memória principal, considerando que, na maioria das linguagens de programação, um valor booleano é armazenado como um byte.

5.2.2 Construção da Geração Inicial B_0

Cada processador P constrói três soluções válidas para o problema da seleção de alvos. Para garantir mais diversidade na população inicial, cada solução é construída por um método diferente.

A primeira solução é construída por uma versão aleatória do Algoritmo 5.1. Para isso foi necessário incorporar uma fonte de aleatoriedade no algoritmo, induzindo uma maior diversidade de soluções, já que a versão original é puramente determinística. O caso 3 do Algoritmo 5.1 seleciona o vértice v que maximiza a expressão $\phi(v)$, onde $\phi(v) = \dfrac{k[v]}{(\delta[v])(\delta[v]+1)}$, e $k[v]$ e $\delta[v]$ são as variáveis de controle utilizadas no Algoritmo 5.1.

Para tornar o algoritmo aleatório, a forma com que o vértice v é selecionada foi modificada: inicialmente, o algoritmo constrói um conjunto ordenado X utilizando a mesma expressão $\phi(v)$. Seja $X = \{(v, \phi(v))\}\ \forall v \in U$, onde U representa os vértices que ainda

permanecem no grafo; em seguida, cada processador P cria um subconjunto X' de X com apenas os P vértices com maiores valores de $\phi(v)$; o vértice v é então escolhido aleatoriamente deste subconjunto X'. Vale observar que o primeiro processador possui $P = 1$ e, portanto, corresponde à versão original. É fácil ver que mais processadores implicam em mais diversidade.

A segunda solução é construída por um método guloso randomizado, detalhado no Algoritmo 5.2. O método é baseado na adição de vértices ao conjunto solução S, seguida da propagação de cada adição. Os vértices ainda não dominados são representados como o conjunto X. No algoritmo, $N[v]$ e $D[v]$ armazenam respectivamente a vizinhança e o requisito de cada vértice $v \in X$.

Além destas variáveis, o algoritmo também utiliza um vetor auxiliar PD para rastrear a propagação da dominação no grafo: se $PD[v] =$ `verdadeiro` então v está dominado (mesmo que $v \notin S$). A seleção dos vértices que serão adicionados é baseada em um método conhecido na literatura como roleta viciada, onde cada vértice v possui uma probabilidade $w(v)$ de ser selecionado. Nesta técnica, os vértices de maior probabilidade são selecionados mais frequentemente, embora sua natureza randômica também possibilite que vértices de baixa probabilidade sejam escolhidos. No algoritmo proposto, $w(v) = |N[v]| \ / \ |X|$. Vale observar que essa formulação define o critério guloso do algoritmo: quanto mais vizinhos um vértice tiver, maior será sua probabilidade de ser selecionado. Esse critério visa maximizar o número de vértices impactados por cada adição. A cada iteração, um vértice x é selecionado da roleta, adicionado ao conjunto solução S. Em seguida, os efeitos dessa adição são propagados pelo procedimento `PropagarDominação`.

O procedimento responsável por propagar os efeitos de cada dominação (incluíndo a ocasionada pela adição de um vértice ao conjunto em construção) foi detalhado como o Algoritmo 5.3. Seu funcionamento é baseado em uma fila Q: primeiro, Q é inicializada com o vértice cuja dominação deve ser propagada, isto é, o vértice x; em seguida, enquanto $Q \neq \emptyset$, a cada iteração, um vértice v é removido de Q e os vizinhos de v são atualizados para considerar que v agora está dominado. Se a adição de v implicar na dominação de um vértice u ($u \in N(v)$), u é inserido em Q. Esse procedimento é repetido até que a filha se torne vazia, o que implica que todos os efeitos da adição do vértice x foram propagados.

Finalmente, a terceira solução S_3 é obtida combinando as duas primeiras: para cada vértice $v \in V$, $S_3[v] = S_1[v] \wedge S_3[v]$. Como S_3 pode não ser um indivíduo viável, o algoritmo precisa se assegurar que S_3 domine o grafo antes de prosseguir.

Algoritmo 5.2: Algoritmo Guloso Randomizado para Construção de Soluções

Entrada: Grafo $G = (V, E)$, Vetor de requisitos R
Saída: Solução S

$X \leftarrow V$
para $v \in X$ **faça**
    $S[v] \leftarrow$ **Falso** $PD[v] \leftarrow$ **Falso**
    $D[v] \leftarrow R[v]$
    $N[v] \leftarrow u \in V \mid uv \in E$
enquanto $X \neq \emptyset$ **faça**
    Seja BR uma roleta viciada com $(v, w(v))\ \forall v \in X$, tal que $w(v) = |N[v]| \ / \ |X|$
    Selecionar um vértice x de BR
    $S[x] \leftarrow$ **Verdadeiro**
    `PropagarDominação` (X, N, D, PD, x)
retorne S

Para garantir que S_3 seja uma solução para o problema da dominação vetorial, primeiro é necessário propagar iterativamente a dominação de cada vértice $v \in S_3$. Durante essa propagação, o algoritmo também remove vértices desnecessários de S_3. Um vértice v é dito desnecessário se, no momento em está sendo processado, seus vizinhos já o dominam. Neste caso, nota-se que v pode ser removido de S_3 sem interferir na capacidade do conjunto dominar o grafo. Quando cada vértice de S_3 é processado e a propagação termina, se ainda houver algum vértice não dominado, o algoritmo então adiciona novos vértices para garantir que S_3 seja viável. Como o Algoritmo 5.2 efetua exatamente a operação desejada, é possível executá-lo tomando como ponto de partida o resultado do processamento de S_3. Quando o procedimento termina, é possível assegurar que S_3 pode dominar todos os vértices.

Após cada processador construir três indivíduos (soluções), o algoritmo prossegue para a próxima fase: avaliar a qualidade (*fitness*) das soluções construídas.

5.2.3 Avalição do *Fitness* dos indivíduos

Para avaliar a qualidade (*fitness*) de cada indivíduo, primeiro é necessário calcular o tamanho do conjunto correspondente. Isso pode ser feito rapidamente contando quantas atribuições verdadeiras estão contidas no vetor. Portanto, seja $z(S)$ o tamanho de um indivíduo S, isto é, a cardinalidade do conjunto representado por S.

Durante o desenvolvimento do algoritmo, foi observado que usar apenas o tamanho

Algoritmo 5.3: Propagar Dominação: Vértice x adicionado ao conjunto

```
1  Function PropagarDominação(X, N, D, PD, x)
2      Q ← {x}
3      enquanto Q ≠ ∅ faça
4          v ← Desenfileirar(Q)
5          para u ∈ N[v] faça
6              D[u] ← max(D[u] − 1, 0)
7              se PD[u] = falso ∧ D[u] ≤ 0 então
8                  PD[u] ← Verdadeiro
9                  Enfileirar(Q,u)
10             N[u] ← N[u] \ {v}
11         N[v] ← ∅
12         X ← X \ {v}
```

do indivíduo como valor de *fitness* leva a uma convergência prematura do algoritmo, uma vez que os melhores indivíduos serão selecionados com maior frequência como ancestrais para a próxima geração. Portanto, para evitar isso, duas melhorias foram introduzidas na avaliação da qualidade de cada indivíduo (*fitness*): usar os dados do censo de soluções (S-Censo) e aplicar o escalonamento sigma (*sigma scaling*), proposto por Mitchell [71].

Como explicado anteriormente, a ideia do censo de soluções (S-Censo) é manter o registro de quantas vezes cada indivíduo foi encontrado pelo algoritmo. Para manter esses registros, sempre que o algoritmo termina uma geração B_i, todas as soluções $S \in B_i$ são analisadas: Se S é uma solução nunca identificada antes, S é adicionado à tabela de soluções; caso contrário, a tabela é atualizado para armazenar que S agora tem mais uma ocorrência.

O tabela S-Censo é usada para influenciar ligeiramente o *fitness* de cada solução: Seja S-Censo (S) o censo de um indivíduo S. Se durante a execução do algoritmo uma solução S for identificada frequentemente então o *fitness* de S será reduzido proporcionalmente a S-Censo (S). Dessa forma, se duas soluções S_1 e S_2 apresentarem $z(S_1) = z(S_2)$ e S_1 for identificado com mais frequência do que S_2 então o *fitness* de S_2 será ligeiramente maior que o de S_1, implicando que S_2 terá mais chances de ser selecionado como um ancestral para a próxima geração do que S_1. Para equilibrar o peso do censo versus o tamanho no cálculo do *fitness* de um indivíduo, dois parâmetros $wSCenso$ e $wTam$ são utilizados.

Uma vez demonstrados os fatores necessários para o cálculo da aptidão, é possível apresentar a expressão que retorna o "protofitness" de uma solução. Vale destacar que o

valor do "protofitness" não corresponde a avaliação definitiva do *fitness* de um indivíduo, pois ainda é necessário aplicar o escalonamento sigma. Portanto, seja

$$f'(S) = \frac{((n - z(S)) * wTam) + ((W - \text{S-Censo}\,(S)) * wSCenso)}{wTam + wSCenso}$$

o "protofitness" de um indivíduo S, onde W é o número total de indivíduos já construídos pelo algoritmo até o momento, incluindo todas as gerações anteriores.

A motivação para aplicar o escalonamento sigma é evitar uma convergência prematura do algoritmo causada por um elitismo demasiadamente acentuado. Para isso, o escalonamento sigma formula o *fitness* de um indivíduo como uma expressão, que depende não só da qualidade deste indivíduo mas também da média das avaliações na geração corrente. Ao expressar a avaliação de um indivíduo como uma função que considera toda a geração, o escalonamento sigma evita o domínio exacerbado de um indivíduo sobre os demais, normalizando as avaliações.

Desta forma, sejam B_i e $S \in B_i$ respectivamente uma geração construída pelo algoritmo genético e uma solução contida nessa geração. Além disso, sejam $\sigma(B_i)$ e $\overline{f'(B_i)}$ o desvio padrão e o valor médio obtido da "protofitness" de cada indivíduo em $S \in B_i$, respectivamente. O *fitness* de um indivíduo $S \in B_i$ é calculado pela seguinte expressão:

$$f(S) = \begin{cases} 1, & \sigma(B) = 0 \\ \max\left(1 + \frac{f'(S) - \overline{f'(B)}}{2\sigma(B)}, 0.01\right), & \sigma(B) \neq 0 \end{cases}$$

Duas observações devem ser feitas sobre a avaliação do *fitness* de um indivíduo $f(S)$:

- Se $\sigma(B_i) = 0$ então todo indivíduo $S \in B_i$ tem o messmo "protofitness" $f'(S)$, implicando que todo indivíduo deve ter a mesma probabilidade de ser escolhido como ancestral da próxima geração. Logo, para esses casos, a função sempre retorna o valor constante 1;
- Para garantir que todo indivíduo tenha uma pequena probabilidade de ser selecionado durante a seleção dos ancestrais da próxima geração, optou-se por atribuir um valor mínimo e arbitrário $0,01$ para o *fitness* de cada indivíduo.

A avaliação do "protofitness" de cada indivíduo de uma geração pode ser realizada em paralelo. No entanto, o escalonamento sigma requer o calculo do desvio padrão e da média das avaliações de todos os indivíduos da geração, implicando que o algoritmo deve sincronizar todos os valores calculados entre os processadores para permitir o cálculo do

fitness final. Desta forma, esse procedimento é iniciado como uma rotina paralela, porém só termina quando todos os processadores se comunicam, o que exige uma sincronização entre eles.

5.2.4 Construção de Novas Gerações

Uma vez que a geração inicial foi construída e avaliada, o algoritmo pode prosseguir para a construção de novas gerações. Vale observar que o algoritmo repete a construção e avaliações de gerações até que as condições de parada sejam atendidas.

Conforme apresentado anteriormente, o algoritmo foi desenvolvido para se beneficiar de um ambiente de processamento paralelo. Logo, para reduzir o tempo necessário para construir novas gerações, cada processador é responsável por gerar indivíduos de forma independente, utilizando diferentes operadores de reprodução. Quando todos os processadores terminam a construção, o algoritmo reúne os indivíduos em uma geração e a dissemina entre os processadores de forma que cada um tenha uma cópia local de toda a população. Isso também se aplica à população inicial.

Todo processador P (identificador de cada processador) constrói três indivíduos: os dois primeiros são obtidos por reprodução e mutação; O último é uma cópia da P melhor solução encontrada na geração anterior. Vale notar que se $P = 1$ então o algoritmo copia a melhor solução da geração anterior para a próxima.

Considerando que o algoritmo propaga os melhores indivíduos de uma geração para a próxima, é possível garantir que, uma vez que uma boa solução seja identificada, ela será propagada até que uma melhor seja encontrada. Esse comportamento elitista garante que o resultado final seja a melhor solução identificada durante a execução do algoritmo. Outro efeito desse comportamento é que dois terços de cada geração são descartados, sendo substituídos por novos indivíduos.

A construção das novas duas soluções S_1 e S_2 é realizada em três passos:

1. Escolher aleatoriamente da geração anterior dois ancestrais P_1 e P_2;
2. Construir dois novos indivíduos (S_1 e S_2) usando reprodução e mutação a partir dos ancestrais selecionados;
3. Assegurar que os novos indivíduos são soluções viáveis para o problema da seleção de alvos.

A seleção dos ancestrais é feita de forma randomizada: o algoritmo constrói uma roleta

viciada na qual cada indivíduo S da geração anterior tem uma probabilidade $f(S)$ de ser escolhido. Depois que a roleta é preparada, dois indivíduos, P_1 e P_2, são selecionados. Uma vez que os ancestrais foram selecionados, o algoritmo pode prosseguir para a fase de reprodução e mutação. Vale observar que cada processador seleciona dois ancestrais de forma aleatória. Esse comportamento visa que cada processador selecione dois ancestrais diferentes, embora não exista nenhuma restrição quanto a isso.

5.2.4.1 Operadores de Reprodução

Para aumentar a diversidade de soluções, o algoritmo genético proposto utiliza catorze métodos de reprodução:

1. *Crossover* de um ponto
2. *Crossover* de dois pontos
3. *Crossover* randomizado
4. *Crossover* uniforme
5. Operador lógico E ($\wedge$)
6. Operador lógico OU ($\vee$)
7. Operador lógico NÃO ($\neg$)
8. Operador lógico E ($\wedge$) randomizado
9. Operador lógico OU ($\vee$) randomizado
10. Média da geração anterior
11. Operador de consenso
12. Trocar um vértice por seus vizinhos
13. Construir dois novos indivíduos
14. Mutação forçada

Os dois primeiros operadores são bem conhecidos na literatura de algoritmos genéticos. Dado os dois pais P_1 e P_2, o Operador 1 define aleatoriamente um número inteiro s no intervalo $[2, n-1]$ e os dois novos filhos são $S_1 = P_1[1], \cdots, P_1[s], P_2[s+1], \cdots, P_2[n]$ e $S_2 = P_2[1], \cdots, P_2[s], P_1[s+1], \cdots, P_1[n]$. O operador 2 difere do primeiro, pois usa dois números aleatórios s_1 e s_2, tais que $s_1 \in \left[2, \frac{n}{2} - 1\right]$ e $s_2 \in \left[\frac{n}{2} + 1, n-1\right]$. Os filhos são definidos pela seguinte regra: $S_1 = P_1[1], \cdots, P_1[s_1], P_2[s_1+1], \cdots, P_2[s_2], P_1[s_2+1], \cdots, P_1[n]$ e $S_2 = P_2[1], \cdots, P_2[s_1], P_1[s_1+1], \cdots, P_1[s_2], P_2[s_2+1], \cdots, P_2[n]$. O terceiro operador segue a mesma ideia que os dois anteriores, mas usa um número aleatório de pontos de corte. O número de pontos de corte é determinado pelo parâmetro δ do algoritmo.

O quarto operador difere dos três primeiros, pois constrói os dois filhos trocando posições aleatórias dos vetores. O número de trocas é determinado por $pProbCross$, um parâmetro do algoritmo. Para cada posição dos vetores S_1 e S_2, um número real p é escolhido aleatoriamente no intervalo $[0, 1]$. Dado o valor de p, os valores de cada

posição de S_1 e S_2 são determinados por $S_1[i] = \begin{cases} P_2[i], & p < pProbCross \\ P_1[i], & \texttt{caso contrário} \end{cases}$ e $S_2[i] = \begin{cases} P_1[i], & p < pProbCross \\ P_2[i], & \texttt{caso contrário} \end{cases}$.

O quinto operador usa um operador lógico E ($\wedge$) para definir os valores booleanos de cada posição das matrizes associadas aos filhos S_1 e S_2. Isto é, o operador 5 retorna $S_1[i] = S_2[i] = P_1[i] \wedge P_2[i]$. É importante destacar que, após a construção, S_1 e S_2 são iguais, entretanto após possíveis mutações e após a fase de otimização, na qual a capacidade dos indivíduos dominarem o grafo é assegurada, os fatores aleatórios envolvidos podem alterar isso. O Operador 6 segue o mesmo procedimento, mas aplicando o operador lógico OU ($\vee$).

O Operador 7 também faz uso de um operador lógico: ele constrói dois filhos que são exatamente o oposto de seus pais. Esse comportamento é realizado pelo operador NÃO ($\neg$), isto é: $S_1[i] = \neg P_1[i]$ e $S_2[i] = \neg P_2[i]$, para cada vértice i do grafo.

Como os Operadores 5, 6 e 7 podem alterar as soluções de forma bastante radical, uma versão mais sutil foi desenvolvida. Nesta versão, o operador lógico é aplicado apenas a alguns vértices da solução, copiando os valores do ancestral nos demais casos. Portanto, o Operador 8 retorna para cada vértice i do grafo os valores de $S_1[i]$ e $S_2[i]$ determinados por $S_1[i] = \begin{cases} P_1[i] \wedge P_2[i], & p < \delta/n \\ P_1[i], & \texttt{caso contrário} \end{cases}$ e $S_2[i] = \begin{cases} P_1[i] \wedge P_2[i], & p < \delta/n \\ P_2[i], & \texttt{caso contrário} \end{cases}$, onde p é um número real selecionado aleatoriamente no intervalo $[0, 1]$ para cada vértice do grafo e δ é um parâmetro do algoritmo. O operador 9 segue o mesmo procedimento, mas aplica o operador lógico OU.

O Operador 10 constrói dois novos indivíduos com base na média das soluções contidas na geração anterior. Ou seja, se um vértice específico estiver contido na maioria dos indivíduos da geração anterior, então é possível supor que esse vértice pode ser necessário para uma boa solução e, desta forma, esse vértice é incluído no novo conjunto. Duas soluções diferentes S_1 e S_2 são construídas: S_1 exige que um vértice pertença a pelo menos 50% dos indivíduos da geração anterior para que este seja incluído no conjunto. Já S_2 requer 60%. Dessa forma, seja B_{i-1} a geração anterior. Pri-

meiro, a solução S_1 é construída através da seguinte regra [1] para cada vértice i do grafo: $S_1[i] = \begin{cases} \texttt{verdadeiro}, & (\sum_{S_a \in B_{i-1}} S_a[i]) > 0.5 * |B_{i-1}| \\ \texttt{falso}, & \texttt{caso contrário} \end{cases}$. Da mesma forma, S_2 é dado por $S_2[i] = \begin{cases} \texttt{verdadeiro}, & (\sum_{S_a \in B_{i-1}} S_a[i]) > 0.6 * |B_{i-1}| \\ \texttt{falso}, & \texttt{caso contrário} \end{cases}$.

O décimo primeiro operador usa o consenso construído ao longo de todas as gerações anteriores para determinar se um vértice deve ser incluído ou não na solução. O operador de consenso é aplicado apenas em alguns vértices aleatórios da solução, portanto, todas as outras posições são definidas como cópias dos pais selecionados. Os dados do censo de vértices (V-Censo) são utilizados para definir se existe ou não um consenso sobre um vértice da solução. Dado um valor real p selecionado aleatoriamente do intervalo $[0, 1]$ para cada vértice do grafo, o valor de cada posição $S_1[i]$ é definido pela seguinte expressão: $S_1[i] = \begin{cases} \texttt{verdadeiro}, & p < \delta/n \ \wedge \ \text{V-Censo}[i] > 0.5 * W \\ \texttt{falso}, & p < \delta/n \ \wedge \ \text{V-Censo}[i] \leq 0.5 * W \\ P_1[i], & \texttt{caso contrário} \end{cases}$, onde W corresponde ao número total de indivíduos já criados pelo algoritmo. O valor de $S_2[i]$ segue a mesma regra, exceto pelo uso de P_2 em vez de P_1.

O operador 12 usa as características do problema da seleção de alvos para construir novas soluções trocando alguns vértices. Cada novo indivíduo é inicializado como uma cópia de seu pai ($S_1 \leftarrow P_1$ e $S_2 \leftarrow P_2$) e, em seguida, são realizadas trocas de vértice em cada indivíduo. Como nos operadores anteriores, o parâmetro δ será utilizado para especificar o número de trocas que o algoritmo deve executar.

Tendo em vista que no problema da seleção de alvos um vértice v pode ser dominado de duas formas: $v \in S_0$ ou $|N(v) \cap S_x| \geq R[v]$, onde S_0 é o conjunto inicial e S_x é um conjunto obtido propagando a regra de dominação a partir de S_0 por x iterações, é possível remover com segurança um vértice v de S_0 e manter v dominado ao satisfazer a segunda expressão.

Para evitar o custo computacional de obter o conjunto S_x de S_0, apenas os efeitos imediatos da dominação são considerados, isto é, a dominação v deve ser assegurada por seus vizinhos que de fato estão incluídos em S_0. Portanto, todas as operações são executadas em S, que é o ponto inicial da dominação ($S_0 = S$).

[1] Assume-se que valores booleanos podem ser somados como números inteiros, onde uma atribuição verdadeira corresponde a uma unidade.

O Algoritmo 5.4 apresenta o esboço deste operador. Nesse algoritmo, as variáveis m_v e c_v armazenam, respectivamente, quantos vizinhos de v devem ser adicionados ao conjunto para assegurar a dominação de v (em uma iteração) e o conjunto dos vizinhos de v que podem ser incluídos em S para assegurar a dominação de v (um conjunto de vértices candidatos para inclusão). Os valores de m_v e c_v são inicializados como $R[v] - |N(v) \cap S|$ e $N(v) \setminus S$. Pode-se perceber que a adição de m_v vizinhos de v garantirá que v seja dominado, uma vez que $N(v) = m_v + |c_v|$ e $R[v] \leq N(v)$.

Vale destacar também que o Operador 12 não efetua a propagação de todas as alterações em S, mesmo que isso implique em menos inclusões para manter o vértice dominado. Essa propagação foi suprimida para evitar o tempo requerido pelo operador de reprodução, uma vez que a função de propagação (Algoritmo 5.3) tem um alto custo computacional. Essa redução no tempo da execução do método foi necessária para equilibrar o tempo necessário para executar os operadores de reprodução. Um desbalanceamento no consumo de tempo entre os operadores implica que os processadores que já terminaram o processamento de métodos mais rápidos fiquem aguardando para que juntos, todos prossigam para a avaliação da geração construída. Essa espera ocasiona um desperdício de recursos computacionais e, ao evitar os custos da propagação, foi possível manter um tempo de processamento mais uniforme entre os processadores. Além disso, vale observar que a solução será otimizada pelo procedimento que assegura que todos os conjuntos construídos R-dominem o grafo após n iterações.

Até o momento, todos os operadores selecionam indivíduos aleatórios (respeitando as probabilidades associadas as avaliações de cada) como ancestrais para construir dois novos indivíduos. No entanto, há mais uma fonte potencial para a construção de bons indivíduos: explorar as melhores soluções da geração anterior. Portanto, dois novos operadores foram implementados baseados nesta oportunidade: Operadores 13 e 14. O Operador 13 ainda processa indivíduos selecionados aleatoriamente como ancestrais, porém utiliza o tamanho do melhor indivíduo na geração anterior para orientar a construção de novas soluções baseadas nestes ancestrais. Por outro lado, o Operador 14 usa somente os melhores indivíduos como fonte de novas soluções.

O décimo terceiro operador constrói dois indivíduos por dois métodos distintos. O primeiro usa partes de cada pai para compor uma nova solução. Inicialmente, sejam P_1 e P_2 os dois ancestrais selecionados da geração anterior. O novo indivíduo S_1 é inicializado

Algoritmo 5.4: Operador 12: Trocar v por $R[v]$ vizinhos de v.

Entrada: Ancestrais P_1 e P_2, Parâmetro δ, Grafo $G = (V, E)$ e Vetor de Requisitos R
Saída: Novos Indivíduos S_1 e S_2

Função `Operador12` (P_1, P_2, δ, G, R)
  $S_1 \leftarrow$ `Trocar`(P_1, δ, G, R)
  $S_2 \leftarrow$ `Trocar`(P_2, δ, G, R)
  retorne S_1 and S_2

Função `Trocar` (P, δ, G, R)
  $S \leftarrow P$
  para $v \in V$ **faça**
    se $S[v] =$ **Verdadeiro então**
      Seja r um número real aleatório no intervalo $[0, 1]$
      se $r < \delta/n$ **então**
        $S[v] \leftarrow$ **Falso**
        $m_v \leftarrow R[v]$ $\quad$ $c_v \leftarrow \emptyset$
        para $u \in N(v)$ **faça**
          se $S[u] =$ **Verdadeiro então**
            $m_v \leftarrow m_v - 1$
          senão
            $c_v \leftarrow c_v \cup \{u\}$
        para $i \in \{1 \dots m_v\}$ **faça**
          Seja u o vértice de maior grau em c_v
          $S[u] \leftarrow$ **Verdadeiro** $\quad$ $c_v \leftarrow c_v \setminus \{u\}$
  retorne S

através da seguinte expressão $S_1[v] = \begin{cases} P_1[v], & r = 0 \\ P_2[v], & \texttt{caso contrário} \end{cases}$, onde para cada $v \in V$, r é um número inteiro aleatório no intervalo $[0, 1]$. Este procedimento é semelhante ao aplicado pelo Operador 4, entretanto, o Operador 13 define um tamanho específico para a solução construída. Portanto, seja P_{melhor} o indivíduo com maior *fitness* na geração anterior e seja $t = \begin{cases} z(P_{melhor}) - \delta, & z(P_{melhor}) > \delta \\ z(P_{melhor}), & \texttt{caso contrário} \end{cases}$ o tamanho definido para S_1 ($z(P_{Melhor})$ corresponde ao número de vértices no conjunto representado por P_{melhor}). Dados S_1 e t, há três casos: $z(S_1) = t$, $z(S_1) < t$ e $z(S_1) > t$. No primeiro caso, o algoritmo retorna S_1 como está, uma vez que já possui o tamanho desejado. Do contrário, o algoritmo deve adicionar ou remover vértices aleatórios, respectivamente, para atingir o tamanho definido com o valor alvo. Após a correção do tamanho, S_1 é retornado e o algoritmo pode prosseguir para a construção do segundo indivíduo S_2.

A construção de S_2 segue um procedimento diferente: os valores do censo de vértices (V-Censo) são utilizados para construir um indivíduo inteiramente aleatório formado por t vértices, onde t corresponde ao valor descrito durante a construção de S_1. Primeiro, S_2 é inicializado como $S_2[v] = \texttt{Falso}$ para todo $v \in V$. Enquanto $z(S_2) < t$, o algoritmo prepara uma roleta viciada com os valores $(v, \text{V-Censo}(v))\ \forall v \in \{x \in V \mid S_2[x] = \texttt{falso}\}$ e, em seguida, adiciona um vértice selecionado aleatoriamente da roleta em S_2. Como todos os indivíduos construídos por um método de reprodução são submetidos a rotina de otimização, não é necessário nenhuma consideração neste momento sobre a viabilidade da solução, uma vez que ela será garantida posteriormente. Conforme explicado no Operador 12, a propagação da dominação também foi evitada para assegurar um melhor balanceamento no consumo de tempo entre os operadores de reprodução.

O funcionamento do Operador 14 é baseado em uma mutação genética forçada. Desta forma, inicialmente os dois indivíduos com maior *fitness* da geração anterior são isolados. Seja P_1 e P_2 esses indivíduos. Em seguida, o algoritmo executa uma mutação forçada em cada pai para construir novos indivíduos. Vale observar que toda solução construída tem uma pequena chance de ser submetida a uma mutação pelo algoritmo genético, contudo o Operador 14 obriga que os indivíduos sejam forçadamente submetidos ao procedimento de mutações, isto é, o fator aleatório é desconsiderado.

5.2.4.2 Mutação

Depois de construir novas soluções utilizando os operadores de reprodução apresentados, cada processador sujeita os indivíduos construídos a possíveis mutações aleatórias. Vale observar que o algoritmo executa o procedimento de mutação em apenas alguns indivíduos selecionados aleatoriamente. O parâmetro `pMutação` expressa a probabilidade de um indivíduo ser submetido ao processo de mutação.

O procedimento de mutação consiste em alterações aleatórias no vetor booleano para incluir e excluir vértices do conjunto representado pelo indivíduo corrente. O parâmetro δ é utilizado para expressar quantas alterações devem ser executadas pelo algoritmo. Vale observar que dada a natureza aleatória do método, o valor do parâmetro δ é apenas uma expectativa, não um limite rígido. O Algoritmo 5.5 apresenta o processo de mutação. Como o algoritmo busca construir soluções cada vez menores, a probabilidade de remover um vértice da solução foi aumentada, permitindo que um valor trocado de falso para verdadeiro (linha 6) se torne falso novamente (linha 7), e aplicando um segundo laço de repetição apenas para remoções aleatórias de vértices (linhas 8, 9 e 10 do algoritmo).

Algoritmo 5.5: Procedimento de Mutação

Entrada: Ancestral P e Parâmetro δ
Saída: Indivíduo mutado S

```
1  Procedure AplicarMutação(S, δ)
2      S ← P
3      prob_δ ← δ/n
4      para i = 1...n faça
5          Sejam r_1 e r_2 dois números reais aleatórios contidos no intervalo [0, 1]
6          se S[i] = Falso      ∧ r_1 < prob_δ então S[i] ← Verdadeiro ;
7          se S[i] = Verdadeiro ∧ r_2 < prob_δ então S[i] ← Falso ;
8      para i = 1...n faça
9          Seja r um número real aleatório contido no intervalo [0, 1]
10         se S[i] = Verdadeiro ∧ r < prob_δ então S[i] ← Falso ;
```

Uma vez que os dois indivíduos S_1 e S_2 foram construídos, a próxima etapa é garantir que eles representem soluções viáveis para o problema da seleção de alvos.

5.2.4.3 Assegurar a viabilidade dos indivíduos

Esse processo é um pouco diferente do aplicado na fase de construção, pois agora é possível aperfeiçoar os critérios de escolha de quais vértices serão adicionados e removi-

dos das soluções. O Algoritmo 5.6 apresenta o procedimento de otimização de soluções, responsável por garantir que cada indivíduo corresponda a uma solução viável para o problema da seleção de alvos.

O algoritmo verifica iterativamente cada vértice da solução. Para isso, a cada iteração um vértice é adicionado e os efeitos dessa adição são propagados. Para selecionar a ordem na qual os vértices serão processados, o algoritmo utiliza uma roleta viciada. Durante a etapa de propagação, caso um vértice u da solução que ainda não tenha sido verificado seja dominado, este vértice pode ser removido da solução sem que a capacidade de dominar o grafo da mesma seja alterada. Esta etapa foi marcada pelo algoritmo como a rotina de otimização.

Após o processamento da solução, caso ainda exista algum vértice não dominado, o algoritmo passa a avaliar a inserção de novos vértices de $V \setminus S$ em S para corrigir isso. Novos vértices são adicionados até que o grafo seja inteiramente dominado. Como o algoritmo para apenas quando todos os vértices são dominados, é possível assegurar que ao final do procedimento S é uma solução viável. O método de propagação segue como apresentado no Algoritmo 5.3.

Considerando que um algoritmo genético busca sempre mais diversidade na população, uma ideia interessante é usar as informações do censo de vértices (V-Censo) para orientar a ordem na qual os vértices serão adicionados durante o algoritmo de otimização, aumentando a probabilidade de que vértices que raramente são usados sejam escolhidos para integrar soluções. Essa ideia é aplicada primeiro a vértices que já estão incluídos no conjunto, mas quando esses vértices são insuficientes para dominar o grafo, o algoritmo passa a adicionar outros vértices não dominados usando essa regra. No Algoritmo 5.2, o valor de $w(v)$ (usado para construir a roleta viciada) foi formulado como uma divisão do grau de cada vértice pelo número de vértices não dominados restantes.

A ideia é aprimorar a formulação de $w(v)$ com as informações do censo de vértices, garantindo que vértice pouco utilizados tenham uma probabilidade ligeiramente maior de serem mantidos ou adicionados, aumentando a diversidade de soluções. Desta forma, a probabilidade de um vértice ser escolhido pela roleta foi definida como

$$w'(v) = \frac{\left(\frac{|N[v]|}{|X|} * wGrau\right) + \left(\frac{W - \text{V-Censo}\,(v)}{P} * wVCenso\right)}{wGrau + wVCenso}$$

, onde $N(v)$ e X são as variáveis de controle do algoritmo, W é o número total de in-

divíduos construídos pelo algoritmo genético em todas as gerações anteriores e $wGrau$ e $wVCenso$ são parâmetros do algoritmo. Os valores de $wGrau$ e $wVCenso$ definem os pesos de cada valor no cálculo da probabilidade. Um destaque importante é que esse procedimento busca uma melhoria na diversidade de soluções, o que pode não levar a melhorias imediatas no *fitness* dos indivíduos.

Algoritmo 5.6: Procedimento de Otimização

Entrada: Indivíduo S, Censo de Vértices V-Censo , Parâmetros $wGrau$ and $wVCenso$, Grafo $G = (V, E)$ e Vetor de requisitos R

Procedure `OtimizarIndivíduo` (S, G, R, V-Censo , $wGrau, wVCenso$)
- $S' = \{v \in V \mid S[v] =$ **Verdadeiro** $\}$
- $X \leftarrow V$
- **para** $v \in X$ **faça**
  - $S[v] \leftarrow$ **Falso** $\quad P[v] \leftarrow$ **Falso**
  - $D[v] \leftarrow R[v] \quad N[v] \leftarrow \{u \in X \mid uv \in E\}$
- **enquanto** $X \neq \emptyset$ **faça**
  - $S' \leftarrow S' \setminus \{x \in S' \mid D[x] =$ **Verdadeiro** $\}$; `// Otimizar` S
  - **se** $S' \neq \emptyset$ **então**
    - Construir uma roleta viciada BR com $(x,\ w'(x))\ \forall x \in S'$
    - Selecionar aleatoriamente um vértice v de BR
    - $S[v] \leftarrow$ **Verdadeiro** ; `// Manter` v `em` S
  - **senão**
    - Construir uma roleta viciada BR com $(x,\ w'(x))\ \forall x \in X$
    - Selecionar aleatoriamente um vértice v de BR
    - $S[v] \leftarrow$ **Verdadeiro** ; `// Adicionar vértice` v `em` S
  - `PropagarDominação` (X, N, D, P, v)

5.2.4.4 Esboço do Método de Reprodução

Como todos os componentes do algoritmo de reprodução foram apresentados, este procedimento pode ser formulado como o Algoritmo 5.7.

Depois que cada processador constrói três soluções viáveis (duas por reprodução e uma copiando uma boa solução anterior), o algoritmo reúne os novos indivíduos em uma nova geração B_i e a transmite para todos os processadores. Em seguida, o algoritmo efetua a avaliação dos indivíduos em B_i. Como cada processador executa o método independentemente, cada operador de reprodução foi implementado de forma a consumir uma quantidade semelhante de tempo, reduzindo o tempo de espera entre processadores. A Figura 5.3 apresenta um gráfico de consumo médio de tempo de cada operador, incluindo o tempo ne-

Algoritmo 5.7: Procedimento de Reprodução

Entrada: Geração anterior B_{i-1}, Parâmetros δ e `pMutação`, Identificador do processador P

Saída: Três indivíduos S_1, S_2 e S_3

Function `Reprodução` $(B_{i-1}, \delta, pMutação)$

Seja r um número inteiro aleatório contido no intervalo $[1, 14]$
Seja Φ_r o Operador de Reprodução r
se $r \leq 13$ **então**
  Seja BR uma roleta viciada com $(S, f(S))\ \forall S \in B_{i-1}$
  Selecionar aleatóriamente de BR dois indivíduos P_1 e P_2
  se $r < 13$ **então**
    $(S_1, S_2) \leftarrow \Phi_r(P_1, P_2)$
  senão
    Seja P_{melhor} o indíviduo com o maior $f(S)\ \forall\ S \in B_{i-1}$
    $t \leftarrow z(P_{melhor}) - \delta$
    se $t \leq 0$ **então** $t \leftarrow z(P_{melhor})$;
    $(S_1, S_2) \leftarrow \Phi_r(P_1, P_2, t)$
senão
  Sejam $P_{Melhor1}$ e $P_{Melhor2}$ os dois indivíduos com os maiores $f(S)\ \forall\ S \in B_{i-1}$
  $(S_1, S_2) \leftarrow \Phi_r(P_{Melhor1}, P_{Melhor2})$
Sejam r_1 e r_2 dois números reais aleatórios contidos no intervalo $[0, 1]$
se $r_1 < pMutação$ **então** `AplicarMutação` (S_1, δ);
se $r_2 < pMutação$ **então** `AplicarMutação` (S_2, δ);
`OtimizarIndivíduo` (S_1)
`OtimizarIndivíduo` (S_2)
Seja S_3 o P-melhor indivíduo em B_{i-1}
retorne S_1, S_2, e S_3

cessário para o procedimento de otimização, obtido durante a execução dos experimentos apresentados na Seção 5.3 [2].

[2] Os dados da instância "YouTube2" não foram incluídos no gráfico pois os resultados obtidos para esta instância não podem ser comparados às demais, uma vez que seu tamanho é aproximadamente quinze vezes maior que a média dos tamanhos de todas as outras instâncias.

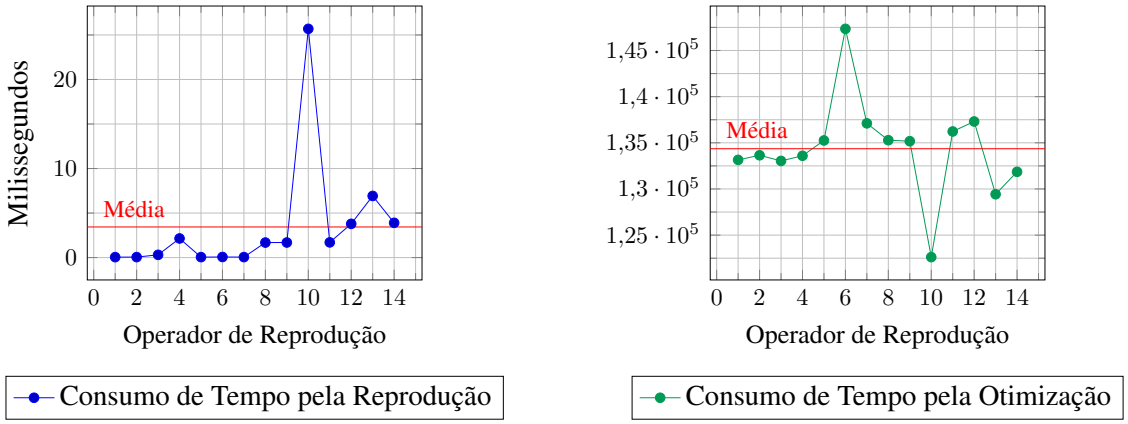


Figura 5.3: Consumo de Tempo pelos Operadores de Reprodução.

Uma vez que o método responsável pode construir novos indivíduos foi detalhado, é possível prosseguir para a análise das condições de parada do algoritmo.

5.2.5 Condições de Parada

O algoritmo constrói novas gerações até que as condições de parada sejam atendidas. Quando o algoritmo para, ele retorna a solução de menor cardinalidade na última geração. As condições de parada do algoritmo são definidas por três parâmetros: $gMin$, $gMax$ e $gSemMelhoria$. Os dois primeiros representam o número mínimo e máximo de gerações, respectivamente. O terceiro, $gSemMelhoria$, representa quantas gerações o algoritmo construirá sem identificar nenhuma melhoria na menor solução já encontrada.

Como mencionado anteriormente, o algoritmo genético proposto é capaz de se adaptar, aumentando ou reduzindo a amplitude das alterações realizadas na fase de reprodução e mutação. A agressividade do algoritmo é definida pelo parâmetro δ, que é ajustado proporcionalmente a quantas gerações foram construídas sem nenhuma melhoria no tamanho das soluções. O valor inicial de δ é calculado a partir do grafo e do vetor de requisitos fornecidos como entradas. Desta forma, o valor de δ é definido pela seguinte expressão:

$$\delta_0(G, R) = \min\left(\frac{\left(\max\left(R[v]\ \forall\ v \in V\right)\right)^2 * n}{\sum_{v \in V} R[v]}, \frac{n}{4}\right).$$

A ideia de inicializar δ através de uma função baseada nos dados de entrada visa definir um número razoável de alterações em um indivíduo S sem que ele seja transformado em um indivíduo completamente diferente. Para grafos pequenos, o valor de δ geralmente é limitado por $n/4$. A Tabela 5.1 apresenta alguns exemplos de valores para δ_0 calculados

para algumas instâncias apresentadas na Seção 5.3. Em cada instância, o requisito de cada vértice $v \in V$ é definido por $\min(d(v), R)$, onde R é o valor da coluna "Requisitos".

Grafo	Vértices	Arestas	Requisitos	δ_0
ca-GrQc	5.241	14.484	1	1
ca-GrQc	5.241	14.484	5	8
ca-GrQc	5.241	14.484	10	24
ca-HepTh	9.875	25.973	1	1
ca-HepTh	9.875	25.973	5	7
ca-HepTh	9.875	25.973	10	23
BlogCatalog	88.784	2.093.195	1	1
BlogCatalog	88.784	2.093.195	5	7
BlogCatalog	88.784	2.093.195	10	17
Douban	154.907	327.162	1	1
Douban	154.907	327.162	5	14
Douban	154.907	327.162	10	46
YouTube2	1.138.499	2.990.443	2	2
YouTube2	1.138.499	2.990.443	5	11
YouTube2	1.138.499	2.990.443	7	20

Tabela 5.1: Alguns exemplos dos valores de δ_0 para algumas instâncias.

5.2.6 O algoritmo genético proposto

Como todos os componentes do algoritmo genético (inicialização, construção de novas gerações, avaliação dos indivíduos e condições de parada) já foram apresentados, é possível formular o algoritmo proposto como o Algoritmo 5.8, demonstrando a integração de todos os componentes.

Algoritmo 5.8: Algoritmo Genético Proposto

Entrada: Grafo G, Vetor de requisitos R, Parâmetros $gMin$, $gMax$, $gSemMelhoria$, pMutação, $wTam$, $wSCenso$, $wGrau$ e $wVCenso$ e Identificador do processador P

Saída: Solução S para G e R

```
1  i ← 0        ctSemMelhorias ← 0
2  Calcular δ_0 a partir de G e R
3  δ_passo ← (3 * δ_0) / gSemMelhorias
4  Construir a geração inicial B_0
5  Avaliar fitness de cada indivíduo S ∈ B_0
6  Atualizar censos populacionais: S-Censo e V-Censo
7  faça
8      i ← i + 1
9      Construir uma nova geração B_i por reprodução partindo de B_{i-1}
10     Avaliar fitness de cada indivíduo S ∈ B_i
11     Atualizar censos populacionais: S-Censo e V-Censo
12     se min(z(S) ∀ S ∈ B_i) < min(z(S') ∀ S' ∈ B_{i-1}) então
13         ctSemMelhorias ← 0
14         δ ← δ_0
15     senão
16         ctSemMelhorias ← ctSemMelhorias + 1
17         δ ← δ_0 + ctSemMelhorias * δ_passo
18 enquanto (i ≤ gMin ∨ ctSemMelhorias ≤ gSemMelhorias) ∧ i ≤ gMax;
19 retorne S | z(S) ≤ z(S') ∀ S, S' ∈ B_i
```

5.3 Resultados

Esta seção apresentará uma revisão de cada parâmetro utilizado no algoritmo genético e os valores a eles atribuídos. Além disso, o ambiente computacional utilizado nos experimentos será detalhado. Em seguida, os resultados dos experimentos serão apresentados e comparados aos resultados de Cordasco *et al.* [20, 21]. Vale destacar que o formato utilizado nos resultados segue o modelo definido por Cordasco *et al.* [20, 21], incluindo a mesma divisão em duas partes: a primeira apresenta os resultados obtidos em grafos aleatórios com 30 e 50 vértices, enquanto a segunda exibe os resultados do processamento de catorze instâncias da literatura que mapeiam grandes redes sociais da vida real.

5.3.1 Parâmetros do Algoritmo e Ambiente Computacional

O valor atribuído a cada parâmetro foi definido através de uma série de comparações em várias instâncias. Os valores descritos correspondem ao melhor equilíbrio entre a qualidade dos resultados e consumo de tempo. Vale observar que o valor do parâmetro δ é obtido a partir dos dados da instância e, portanto, não é um parâmetro fixo.

Primeiro, sobre os parâmetros usados na avaliação do *fitness* de cada indivíduo: $wSCenso$ e $wTam$. Conforme apresentado anteriormente, esses parâmetros especificam os pesos associados aos dados do censo de soluções (S-Censo) e o tamanho da solução na equação responsável por calcular o "protofitness" do indivíduo. Empiricamente, os melhores resultados foram obtidos quando $wTam = 0,98$ e $wSCenso = 0,02$, isto é, o tamanho corresponde a 98% da avaliação, enquanto o censo, a 2%.

Os parâmetros $wGrau$ e $wVCenso$ são utilizados para definir os pesos associados aos dados do censo de vértices (V-Censo) e o grau de cada vértice no cálculo da probabilidade de um vértice ser selecionado para ser adicionado à solução, durante o procedimento de otimização. Além disso, foi observado que $wGrau = 0,98$ e $wVCenso = 0,02$ resultam em um bom equilíbrio de pesos.

A fase de reprodução utiliza mais dois parâmetros: $pProbCross$ e `pMutação`. O parâmetro $pProbCross$ é usado no Operador de Reprodução 4 para definir a probabilidade de trocar os valores associados a um vértice entre duas soluções, enquanto *pMutação* especifica a probabilidade de um indivíduo sofre uma mutação. Assim como os casos anteriores, foi observado empiricamente que o melhor valor para esses parâmetros é $pProbCross = 0,3$ e *pMutação* $= 0,025$, isto é, 30% dos vértices serão trocados entre os

ancestrais e 2,5% das soluções construídas sofrerão mutações.

Os últimos parâmetros estão relacionados às condições de parada do algoritmo: $gMin$, $gMax$ e $gSemMelhoria$. Os dois primeiros definem, respectivamente, o número mínimo e máximo de gerações que serão construídas pelo algoritmo; O parâmetro $gSemMelhoria$ define quantas gerações o algoritmo executará sem encontrar nenhuma melhoria no tamanho da menor solução até que ele pare. Considerando que esses parâmetros têm um impacto significativo no tempo de execução, foi observado que $gMin = 10$, $gMax = 500$ e $gSemMelhoria = 50$ resultam em um bom balanceamento entre o tempo e a qualidade das soluções obtidas. Vale observar que o algoritmo não contabiliza a geração inicial B_0 como uma iteração, pois seu método construtivo difere das demais gerações. Logo, quando $gMax = gMin = 1$ o algoritmo construirá duas gerações B_0 e B_1. O mesmo se aplica a $gSemMelhorias$.

Uma vez que os parâmetros foram definidos, é possível prosseguir para o ambiente computacional. Os experimentos foram executados utilizando o ambiente *Compute Engine Service*, fornecido pelo *Google Cloud Platform* [51]. Os experimentos utilizaram uma instância dedicada otimizada para computação de alto desempenho (*Compute-optimized* - C2) com 60 processadores e 240 GB de RAM. Além disso, a instância executa o Ubuntu Linux 2018.04 LTS. O algoritmo foi implementado em C++, sendo que a paralelização do algoritmo foi realizada utilizando o protocolo *MPI* [73].

5.3.2 Resultados para Grafos Randômicos

Os testes do algoritmo genético seguiram o mesmo procedimento utilizado por Cordasco *et al.* [20, 21]. Os primeiros testes consistem em instâncias formadas por grafos e requisitos aleatórios. Os grafos foram gerados utilizando um parâmetro p, que expressa a probabilidade de uma aresta existir ou não. Portanto, a construção de um grafo $G(n, p)$ com n vértices com probabilidade p consiste de iterar sobre cada aresta e dentre as $(n^2 - n)/2$ possíveis. Para cada aresta um número real aleatório r no intervalo $[0, 1]$ é selecionado: se $r < p$ então a aresta e é adicionada ao grafo, do contrário e é omitida. Uma vez que o grafo foi construído, sua conexidade é avaliada: caso o grafo seja desconexo ele é descartado, pois, representa duas instâncias distintas com menos vértices. Desta forma, novos grafos são construídos até que um grafo conexo seja encontrado. Uma vez que o grafo foi definido, resta estabelecer o vetor de requisitos. Para cada vértice $v \in V$, $R[v]$ corresponde a um número inteiro aleatório no intervalo $[1, d(v)]$, onde $d(v)$ equivale ao grau do vértice.

O algoritmo genético foi então aplicado a instâncias aleatórias com 30 e 50 vértices, com probabilidades $p \in \{0{,}1,\ 0{,}2,\ \ldots,\ 0{,}9\}$.

Considerando que, para instâncias menores, é possível obter a solução ótima em uma quantidade aceitável de tempo, a resposta obtida para cada instância foi confrontada com o resultado ótimo para a mesma. A Subseção A.2.2 apresenta o algoritmo *backtracking* paralelizado utilizado como uma fonte de soluções ótimas.

As Figuras 5.4 e 5.5 apresentam os resultados obtidos para grafos randômicos com 30 e 50 vértices, respectivamente. Nas duas figuras, a coluna azul representa os resultados obtidos pelo algoritmo *backtracking* (soluções ótimas), a coluna verde apresenta os resultados obtidos pelo algoritmo genético proposto e a amarela, os resultados do algoritmo Cordasco *et al.* [20, 21]. Vale destacar que para permitir uma comparação entre os resultados obtidos por cada um dos algoritmos para uma mesma instância, o método proposto por Cordasco *et al.* [20, 21] foi implementado em C++, conforme descrito nos artigos e executado no mesmo ambiente computacional. Para cada probabilidade p, os dois métodos foram executado apenas uma única vez. Sobre o consumo de tempo, o algoritmo proposto por Cordasco *et al.* [20, 21] foi mais rápido, conforme o esperado. Contudo, o algoritmo genético encontra a solução ótima para todas as instâncias, enquanto em sete casos, o melhor algoritmo anterior [20, 21] erra a solução ótima por um vértice.

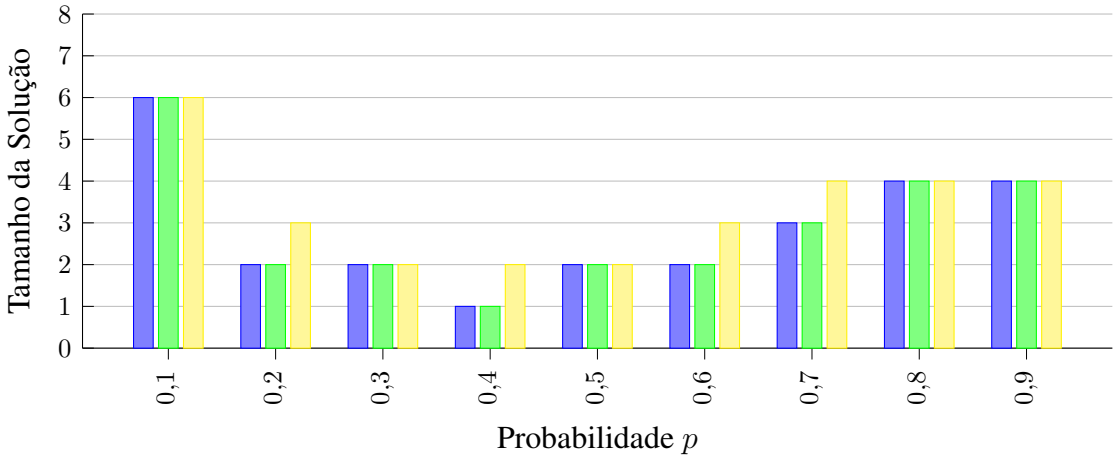


Figura 5.4: Experimentos com grafos randômicos com 30 vértices.

5.3.3 Resultados para Grandes Redes Sociais

Seguindo o formato dos experimentos realizados por Cordasco *et al.* [20, 21], o algoritmo genético proposto foi aplicado em grafos obtidos através do mapeamento de grandes

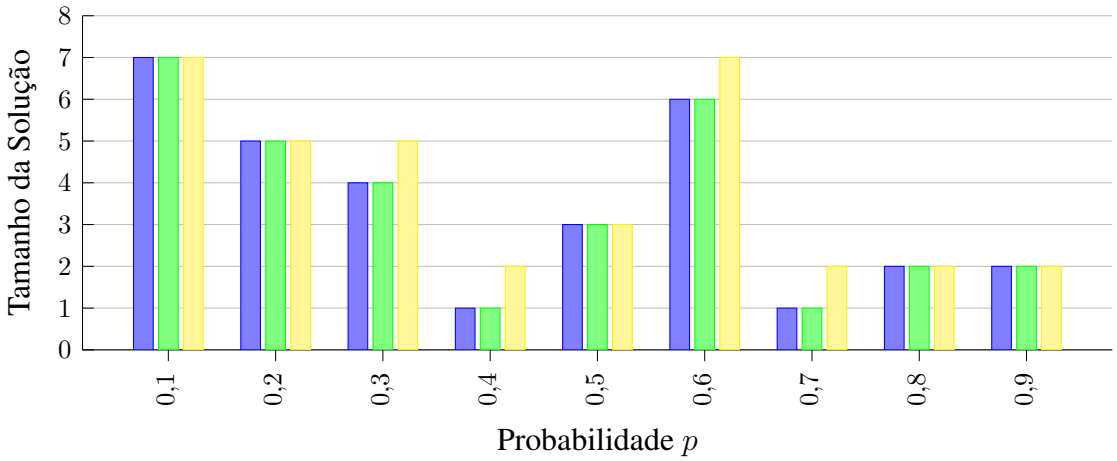


Figura 5.5: Experimentos com grafos randômicos com 50 vértices.

sociais. Foram analisados catorze grafos que modelam redes sociais, com tamanhos variando entre 5.241 e 1.138.499 vértices, obtidos de duas fontes diferentes.

A primeira fonte é o SNAP (*Stanford Large Network Dataset Collection*) [61, 62] e consiste de cinco grafos: ca-AstroPh, ca-CondMat, ca-GrQc, ca-HepPh e ca-HepTh. A segunda fonte explorada foi o repositório de dados de computação social da Universidade do Estado do Arizona (*Social Computing Data Repository from Arizona State University*) [80], com nove grafos: BlogCatalog, BlogCatalog2, BlogCatalog3, BuzzNet, Delicious, Douban, Last.fm, Livemocha e Youtube2.

O primeiro passo para executar o algoritmo genético nestas instâncias foi processá-las em arquivos binários, que serão posteriormente fornecidos como entradas para o algoritmo. Contudo, foram observadas algumas inconsistências nos grafos nesta etapa. Por exemplo, a instância Delicious é fornecida como dois arquivos que contêm os vértices e as arestas do grafo. O arquivo com as arestas contém 1.385.843 entradas, porém a descrição da instância diz que o grafo deveria possuir 1.419.519 arestas. Nota-se então que existem arestas ausentes, conforme declarado no site do conjunto de dados.

Portanto, para detectar e corrigir inconsistências, os dados das instâncias foram pré-processados antes de iniciar os testes com o algoritmo genético. Essa etapa de processamento removeu multiarestas (duas ou mais arestas com os mesmos vértices finais), laços (arestas que começam e terminam no mesmo vértice) e vértices isolados (com grau 0), resultando em um grafo simples e não direcionado (a conexidade não foi avaliada). A Tabela 5.2 apresenta os resultados do pré-processamento.

Depois do pré-processamento as instâncias foram armazenadas em arquivos binários,

	Dados da Instância		Resultados do Processamento		
Instância	Vértices	Arestas	Vértices	Aresta	Δ
BlogCatalog	88.784	4.186.390	88.784	2.093.195	9.444
BlogCatalog2	97.884	2.043.701	97.884	1.668.647	27.849
BlogCatalog3	10.312	333.983	10.312	333.983	3.992
BuzzNet	101.168	4.284.534	101.163	2.763.066	64.289
ca-AstroPh	18.772	396.160	18.771	198.050	504
ca-CondMat	23.133	186.936	23.133	93.439	279
ca-GrQc	5.242	28.980	5.241	14.484	81
ca-HepPh	12.008	237.010	12.006	118.489	491
ca-HepTh	9.877	51.971	9.875	25.973	65
Delicious	103.144	1.419.519	102.154	881.594	2.473
Douban	154.907	654.188	154.907	327.162	287
Last.fm	108.493	5.115.300	108.493	3.470.597	5.225
Livemocha	104.438	2.196.188	104.103	2.193.083	2.980
Youtube2	1.138.499	2.990.443	1.138.499	2.990.443	28.754

Tabela 5.2: Resultado do Pré-Processamento das Instâncias.

formados por $n + m$ tuplas de inteiros, onde as n primeiras tuplas contém a associação entre os rótulos dos vértices do grafo resultante aos rótulos originais. Essa associação foi necessária para permitir atribuir um rótulo interno aos vértices e ainda assim ser possível uma transposição para os rótulos originais. Esse rótulo interno é requerido pelo algoritmo genético, pois a forma de representar as soluções depende de uma ordenação dos vértices de 1 até n, utilizada para endereçar o vetor de booleanos. As demais m posições do arquivo contém as arestas do grafo, baseadas já nos rótulos ordenados de 1 até n. Esses arquivos binários foram então fornecidos como entradas para o algoritmo genético, com os parâmetros já apresentadas.

Antes de prosseguir para a tabulação dos resultados, vale destacar que o pre-processamento das instâncias pode ter modificado os grafos de alguma forma, implicando que os resultados apresentados por Cordasco *et al.* [20, 21] podem não ser mais coerentes com as novas instâncias. Desta forma, para permitir uma comparação justa entre os resultados, cada instância foi processada novamente. Portanto, cada instância foi processada duas vezes: primeiro pelo algoritmo proposto por Cordasco *et al.* [20, 21] e, em seguida, pelo algoritmo genético. Deve ser observado que o método guloso de Cordasco *et al.* [20, 21] não foi concebido para se beneficiar de um ambiente multiprocessado, portanto apenas um processador foi utilizado para sua execução, enquanto os demais ficaram ociosos.

Durante o processamento das instâncias da literatura, foi detectado um problema com as instâncias baseadas no grafo denominado “YouTube2”, cujo tamanho é aproximada-

mente quinze vezes maior que a média dos tamanhos de todos os outros grafos. Nas primeiras tentativas de executar o algoritmo genético nas instâncias "YouTube2" (especialmente o caso $R = 1$), observou-se que a execução do algoritmo não havia terminado após mais de 24 horas de processamento contínuo. Desta forma, foi necessário estabelecer uma condição de parada específica para estes casos, consistindo de um temporizador que interrompe o processamento após 20 horas de execução (descartando o tempo de leitura do arquivo binário). O limite de tempo de 20 horas foi definido ao observar a média de consumo das instâncias anteriores (tempo consumido divido pelo tamanho do grafo). Esse valor foi multiplicado pelo tamanho da instância do "YouTube2".

Desta forma, as Tabelas 5.3 a 5.16 apresentam os resultados do algoritmo genético para cada instância, comparados com os resultados de Cordasco *et al.* [20, 21]. A coluna R apresenta o requisito alvo, isto é, $R[v] = \min(R, d(v)) \; \forall v \in V$. Já as colunas "Tam. C." e "Tempo C." apresentam os resultados obtidos pelo algoritmo proposto por Cordasco *et al.* [20, 21], executado no mesmo ambiente computacional. As colunas "Tam. G." e "Tempo G." contêm os resultados obtidos pelo algoritmo genético proposto. Vale observar que as colunas "Tempo C." e "Tempo G." apresentam o tempo de execução em minutos, ignorando o tempo de leitura da instância. Finalmente, as colunas "Gerações" e "Melhorias" contêm, respectivamente, quantas gerações o algoritmo genético executou para cada instância e a diferença entre "Tamanho G.A." e "Tamanho C.". Valores positivos para a coluna "Melhorias" indicam um ganho na melhor solução conhecida. Isto é, para estes casos, o algoritmo genético encontrou uma solução de menor cardinalidade do que a retornada pelo método proposto por Cordasco *et al.* [20, 21].

BlogCatalog						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	2,7833	1	36,9833	51	0
2	73	0,6500	73	23,4333	51	0
3	168	0,9833	167	30,3167	52	1
4	288	1,2333	288	33,0500	51	0
5	439	1,0167	439	27,0167	51	0
6	603	1,0500	603	30,3333	51	0
7	810	0,7167	810	29,5333	51	0
8	999	0,9333	999	29,6500	51	0
9	1197	0,8000	1197	27,7167	51	0
10	1421	0,6167	1421	25,5333	51	0

Tabela 5.3: Resultados para a instância BlogCatalog.

A Tabela 5.17 apresenta as médias obtidas para cada instância. A colunas "Ganho"

BlogCatalog2						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	3,4833	1	34,7167	51	0
2	2	2,7167	2	36,6333	51	0
3	3	3,5167	3	41,4333	51	0
4	4	2,6333	4	42,5000	51	0
5	5	3,4500	5	38,5667	51	0
6	10	2,8000	10	36,3833	51	0
7	14	2,7833	14	40,8833	51	0
8	20	2,6333	20	37,9000	51	0
9	28	2,5500	28	40,1000	51	0
10	37	3,4333	37	34,7333	51	0

Tabela 5.4: Resultados para a instância BlogCatalog2.

BlogCatalog3						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	0,0333	1	3,0500	51	0
2	2	0,0167	2	3,9500	51	0
3	4	0,0333	3	3,2500	51	1
4	8	0,0333	7	3,5000	52	1
5	15	0,0167	14	4,0833	57	1
6	23	0,0167	23	4,2167	54	0
7	30	0,0167	29	5,2833	72	1
8	45	0,0167	43	5,4500	72	2
9	68	0,0167	67	3,7833	51	1
10	82	0,0167	81	3,2333	51	1

Tabela 5.5: Resultados para a instância BlogCatalog3.

BuzzNet						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	3,8000	1	49,6333	51	0
2	2	3,0833	2	44,6667	51	0
3	10	0,9000	10	44,8833	51	0
4	37	1,2833	37	38,3500	51	0
5	78	1,4833	78	33,4167	51	0
6	198	1,1333	192	33,6167	54	6
7	411	1,0500	402	49,3167	96	9
8	678	1,3667	676	60,5667	103	2
9	956	1,0000	949	43,4333	65	7
10	1199	1,0333	1196	44,8500	55	3

Tabela 5.6: Resultados para a instância BuzzNet.

ca-AstroPh						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	297	0,0833	289	4,1833	51	8
2	807	0,0167	807	3,2333	51	0
3	1370	0,0333	1369	3,5000	51	1
4	1904	0,0333	1902	6,6333	77	2
5	2407	0,0167	2407	4,5833	51	0
6	2887	0,0167	2883	10,0667	97	4
7	3311	0,0333	3311	9,2833	59	0
8	3716	0,0500	3709	9,2500	65	7
9	4112	0,0500	4101	13,9667	78	11
10	4476	0,0333	4465	32,3667	146	11

Tabela 5.7: Resultados para a instância ca-AstroPh.

ca-CondMat						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	594	0,1833	567	3,0833	51	27
2	1704	0,0167	1703	3,8500	53	1
3	3078	0,0333	3071	5,6333	53	7
4	4474	0,0500	4469	10,7000	56	5
5	5672	0,0500	5668	28,9500	98	4
6	6737	0,0500	6728	31,6667	67	9
7	7660	0,0500	7654	59,7833	78	6
8	8430	0,0667	8420	59,8667	102	10
9	9067	0,0833	9066	65,2500	76	1
10	9598	0,1333	9591	55,9000	73	7

Tabela 5.8: Resultados para a instância ca-CondMat.

ca-GrQc						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	379	0,0000	354	1,8000	51	25
2	810	0,0000	809	1,9667	52	1
3	1246	0,0000	1240	2,5833	62	6
4	1556	0,0000	1556	2,8833	62	0
5	1809	0,0000	1803	2,8000	57	6
6	1985	0,0000	1979	4,3500	71	6
7	2121	0,0000	2117	3,2667	65	4
8	2229	0,0000	2226	3,6833	71	3
9	2312	0,0000	2312	4,4500	75	0
10	2376	0,0000	2374	4,7333	81	2

Tabela 5.9: Resultados para a instância ca-GrQc.

ca-HepPh						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	289	0,0167	276	2,3667	51	13
2	778	0,0167	778	2,7500	51	0
3	1348	0,0333	1346	3,3000	53	2
4	1854	0,0000	1852	5,9833	91	2
5	2344	0,0000	2343	4,0667	62	1
6	2764	0,0000	2760	7,3000	84	4
7	3104	0,0000	3099	6,6333	71	5
8	3437	0,0167	3425	9,3500	75	12
9	3675	0,0167	3667	7,6333	58	8
10	3895	0,0167	3884	16,1667	116	11

Tabela 5.10: Resultados para a instância ca-HepPh.

ca-HepTh						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	451	0,0000	427	2,0500	51	24
2	1104	0,0000	1102	1,7167	52	2
3	1797	0,0000	1794	3,2667	72	3
4	2364	0,0000	2358	4,8000	74	6
5	2850	0,0000	2843	10,1667	151	7
6	3227	0,0000	3227	5,6333	62	0
7	3530	0,0167	3528	5,0833	51	2
8	3778	0,0000	3773	10,4000	96	5
9	3970	0,0000	3968	6,4333	51	2
10	4130	0,0000	4128	12,5667	88	2

Tabela 5.11: Resultados para a instância ca-HepTh.

Delicious						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	1,3000	1	11,8667	51	0
2	22	0,3333	22	5,1500	51	0
3	105	0,6667	101	7,9500	51	4
4	315	0,5833	306	8,1167	52	9
5	706	0,7333	672	10,8667	93	34
6	1241	0,8333	1203	11,4333	105	38
7	1862	0,6833	1839	10,0833	74	23
8	2572	0,6333	2541	9,8167	67	31
9	3314	0,7000	3278	12,8167	99	36
10	4082	0,6667	4031	12,3167	76	51

Tabela 5.12: Resultados para a instância Delicious.

Douban						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	5,1333	1	46,8667	51	0
2	241	0,1000	241	22,7333	51	0
3	639	2,8500	639	30,7167	51	0
4	1072	2,3500	1072	35,3833	51	0
5	1462	2,5667	1462	38,3667	51	0
6	1938	3,3667	1927	39,5167	51	11
7	2362	2,7000	2362	45,4167	51	0
8	2833	6,3167	2816	45,6500	59	17
9	3230	4,8833	3222	55,7667	96	8
10	3590	5,2500	3586	57,2500	96	4

Tabela 5.13: Resultados para a instância Douban.

Last.fm						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	5,4000	1	73,8500	51	0
2	3	5,2333	2	90,3667	74	1
3	6	5,0000	6	74,2500	51	0
4	11	1,5500	11	22,0667	51	0
5	17	1,6833	17	22,3333	51	0
6	46	1,6000	45	26,6167	70	1
7	91	1,6167	90	29,5667	84	1
8	162	1,7167	160	25,7833	66	2
9	267	1,7333	261	27,3333	73	6
10	440	1,7167	429	32,4333	97	11

Tabela 5.14: Resultados para a instância Last.fm.

Livemocha						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1	5,5167	1	57,1500	51	0
2	17	0,6500	17	29,0167	51	0
3	87	0,6833	87	32,7333	51	0
4	238	0,8667	238	30,0167	51	0
5	482	0,6500	482	32,3833	51	0
6	829	0,6333	828	31,9333	51	1
7	1272	1,1167	1271	33,4500	51	1
8	1793	1,0833	1787	72,3500	113	6
9	2425	0,8833	2424	35,7500	51	1
10	3097	0,9167	3091	57,2167	73	6

Tabela 5.15: Resultados para a instância Livemocha.

YouTube2						
R	Tam. C.	Tempo C. (min)	Tam. G.	Tempo G (min)	Gerações	Melhoria
1	1628	447,6500	930	1200,0000	0	698
2	74180	10,3667	74167	1200,0000	31	13
3	120877	54,7167	120873	1200,0000	9	4
4	151198	68,9667	151197	1200,0000	5	1
5	172064	78,9167	172064	1200,0000	3	0
6	187217	75,7667	187217	1200,0000	3	0
7	198824	94,0833	198824	1200,0000	2	0
8	207982	99,9000	207982	1200,0000	1	0
9	215325	106,0333	215325	1200,0000	1	0
10	221343	110,3333	221343	1200,0000	1	0

Tabela 5.16: Resultados para a instância YouTube2.

representa a redução média no número de vértices da solução do algoritmo guloso quando comparado o Algoritmo de Cordasco *et al.* [20, 21]. As demais colunas seguem o formato apresentado anteriormente.

Instância	Tempo C. (min)	Tempo G.A. (min)	Gerações	Ganho
BlogCatalog	1,0783	29,3567	51,1000	0,1000
BlogCatalog2	3,0000	38,3850	51,0000	0,0000
BlogCatalog3	0,0217	3,9800	56,2000	0,8000
BuzzNet	1,6133	44,2733	62,8000	2,7000
ca-AstroPh	0,0367	9,7067	72,6000	4,4000
ca-CondMat	0,0717	32,4683	70,7000	7,7000
ca-GrQc	0,0000	3,2517	64,7000	5,3000
ca-HepPh	0,0117	6,5550	71,2000	5,8000
ca-HepTh	0,0017	6,2117	74,8000	5,3000
Delicious	0,7133	10,0417	71,9000	22,6000
Douban	3,5517	41,7667	60,8000	4,0000
Last,fm	2,7250	42,4600	66,8000	2,2000
Livemocha	1,3000	41,2000	59,4000	1,5000
YouTube2	114,6733	1200,0000	5,6000	71,6000
Média	9,1999	107,8326	59,9714	9,5714

Tabela 5.17: Médias dos resultados obtidos pelo algoritmo genético para cada instância.

5.4 Conclusão

Este capítulo apresentou um algoritmo genético que utiliza as informações produzidas durante sua execução para melhorar suas tomadas de decisões. Os objetivos eram evitar uma possível convergência prematura, aumentar a diversidade das soluções e criar uma ma-

neira de escapar de soluções sub-ótimas locais. Considerando as melhorias nas soluções anteriormente conhecidas, pode-se considerar que o objetivo foi alcançado, estabelecendo o uso de informações decorrentes do censo como uma boa alternativa para algoritmos genéticos.

Vale observar que, em geral, os dados necessários para implementar o uso das informações do censo são habitualmente gerados pelos algoritmos genéticos. Desta forma, este trabalho demonstrou que a coleta e o uso de tais dados também são simples. No entanto, a memória usada para armazenar as informações do censo pode ser uma desvantagem, especialmente em problemas onde a representação de soluções como indivíduos implica em um consumo elevado de memória principal.

Durante este capítulo, também foram introduzidos novos operadores de reprodução, como, por exemplo, o operador que troca um vértice com seus vizinhos. A Figura 5.6 apresenta o impacto de cada operador na busca por novas melhores soluções. É importante observar que cada nova melhor solução pode ser resultado de várias melhorias sucessivas, cada uma alcançada por um operador diferente. Nesta figura é possível notar que o operador lógico E (operador 5) tem um impacto significativamente maior que o operador 6, que corresponde ao operador lógico OU. Essa conclusão pode ser explicada ao se observar que o operador E constrói soluções menores que o operador OU, uma vez que, no primeiro, um vértice só é adicionado à solução resultante se ambos os pais contém esse vértice, enquanto no segundo, um vértice é adicionado se apenas um dos ancestrais contiver esse vértice no conjunto correspondente.

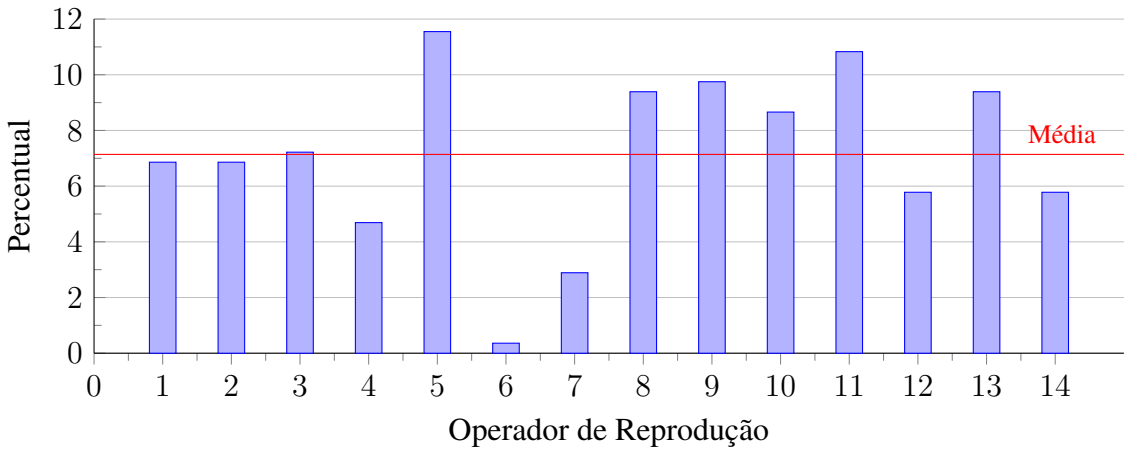


Figura 5.6: Impacto dos operadores de reprodução na busca por melhores soluções.

Vale destacar que o algoritmo genético proposto aprimorou a melhor solução anteriormente conhecida em $59,29\%$ dos casos, com uma melhoria média de $9,57$ vértices.

Pode-se considerar que o ganho obtido justifica o uso do poder computacional necessário para superar o algoritmo de Cordasco *et al.* [20, 21].

Além da análise sobre a qualidade das soluções encontradas, também foi possível efetuar uma comparação sobre o consumo de tempo dos algoritmos, uma vez que o procedimento proposto por Cordasco *et al.* [20, 21] foi implementado e executado no mesmo ambiente computacional. O algoritmo da literatura foi demorou em média 1,11 minutos para fornecer os resultados, enquanto o algoritmo genético gasta 23,82 minutos [3]. Essa diferença era esperada, já que um algoritmo genético requer a simulação de várias gerações formadas por diversos indivíduos, onde cada indivíduo representa uma solução diferente. Como demonstrado, o maior consumo de tempo também pose ser justificado, uma vez que, na maioria dos casos, o algoritmo genético retornou melhores soluções.

Uma proposta para trabalhos futuros, visando o aperfeiçoamento do algoritmo genético, é efetuar uma análise mais aprofundada sobre o valor atribuído para cada parâmetro do algoritmo. Esse estudo poderia ser realizado utilizando o Pacote *IRACE*, proposto por López-Ibáñez *et al.* [63], dada a sua habilidade de determinar a melhor configuração possível para cada parâmetro de forma muito mais detalhada e sensível do que as observações empíricas utilizadas até o momento.

Outra ideia para um estudo futuro é adequar o algoritmo genético para o problema da dominação vetorial, apresentado nos Capítulos 3 e 4. Um ponto interessante deste estudo é a avaliação do impacto da alteração da regra da dominação. Como todo vértice deve ser dominado em uma única iteração, as rotinas de avaliação e adequação se tornam mais simples, pois não existe mais a necessidade de propagar os efeitos da dominação de cada vértice. Essa análise poderia comprovar as observações de que o custo da propagação da dominação é um ponto crítico no consumo de tempo.

[3] As instâncias YouTube2 foram excluídas da comparação do tempo porque a execução foi interrompida após 20 horas de processamento.

Capítulo 6

Conclusão

Este trabalho abordou dois problemas de propagação em grafos através de métodos exatos e aproximativos. Para o problema da dominação vetorial, o foco do trabalho foi definir as fronteiras da complexidade computacional. Entretanto, considerando o vasto universo de famílias de grafos, optou-se por limitar o escopo aos grafos cordais e suas subfamílias.

Como visto no Capítulo 3, o problema da dominação vetorial é $\mathcal{NP}$-Completo para grafos em geral, assim como para grafos cordais. Porém, a definição dos limiares da complexidade computacional ainda é um problema em aberto para algumas subfamílias de grafos cordais. Este trabalho explorou como o problema da dominação vetorial se relaciona ao número de cliques maximais do grafo e à caracterização de cada família de grafos. O trabalho de Cicalese *et al.* [18] demonstrou que para famílias de grafos onde o parâmetro *clique-width* é limitado, o problema da dominação vetorial admite um algoritmo de tempo polinomial, ainda que com polinômio de grau alto em alguns casos.

Entretanto, como evidenciado neste trabalho, é possível melhorar esses resultados em algumas classes explorando características inerentes à estrutura de cada família. No Capítulo 4, a caracterização dos grafos *split*-indiferença foi explorada como base para um algoritmo de tempo linear para resolver o problema da dominação vetorial. Esse algoritmo foi baseado na existência de somente quatro casos possíveis de grafos *split*-indiferença e na limitação de vértices simpliciais nas cliques.

Além deste resultado, existe outro caso em que foi possível demonstrar um algoritmo de tempo linear: grafos com exatamente duas cliques maximais. Novamente, o algoritmo explora a estrutura mais restrita da família de grafos para encontrar um conjunto

R-dominante mínimo. Uma progressão natural deste estudo foram grafos com exatamente três cliques maximais. Contudo, não foi possível encontrar até o momento um algoritmo linear para este caso. Para grafos com *clique-width* limitada, vale o resultado de Cicalese *et al.* [18]: o problema da dominação vetorial nestes grafos admite um algoritmo de tempo $O(k|\sigma|(n+1)^{5k})$, onde k é o valor da *clique-width* e σ equivale a uma k-expressão para o grafo. Vale observar que, durante o estudo dos grafos com exatamente três cliques maximais, foi possível estabelecer que todo grafo desta família é também um grafo de intervalo.

Tendo em vista que uma das motivações deste trabalho foi estabelecer as fronteiras da complexidade computacional do problema da dominação vetorial em grafos cordais, vale relembrar a Figura 6.1. Nesta figura, a complexidade computacional do problema da dominação vetorial é indicada por quatro cores: as classes onde o problema é $\mathcal{NP}$-Completo são hachuradas em vermelho, enquanto as classes em amarelo admitem algoritmos de tempo polinomial. Já as classes marcadas em verde admitem algoritmos de tempo linear. Para os casos onde a complexidade ainda é desconhecida, as classes são hachuradas na cor azul. Existem classes cuja complexidade computacional depende dos requisitos, como, por exemplo, os *directed path graphs* onde o problema do conjunto dominante ($R[v] = 1 \; \forall v \in V$) admite algoritmo de tempo linear, enquanto a complexidade do problema da dominação vetorial permanece em aberto. Para estes casos, as classes foram apresentadas como um degradê de cores, onde a cor do lado esquerdo representa a complexidade do problema do conjunto dominante, enquanto a cor do lado direito, a complexidade do problema da dominação vetorial. As contribuições deste trabalho foram marcadas com linhas duplas.

O segundo problema abordado neste trabalho foi o problema da seleção de alvos, que pode ser visto como uma generalização do primeiro onde o contágio do grafo pode ser realizado ao longo de múltiplas iterações. Vale observar que o problema da dominação vetorial constitui um limite superior para o problema da seleção de alvos (se o conjunto R-domina o grafo em uma geração, trivialmente ele o contamina ao longo de múltiplas iterações).

Como visto no Capítulo 5, a abordagem empregada para o estudo do problema da seleção de alvos foi baseada em algoritmos aproximativos. Essa abordagem foi motivada pelo grande interesse prático no problema, especialmente como um modelo computacional para disseminação de doenças ou para propagação de influência em redes sociais.

Um dos resultados do estudo de métodos aproximativos para o problema da seleção de

Figura 6.1: Hierarquia de subclasses dos grafos cordais (reapresentação da Figura 3.13 - página 59)

alvos foi o algoritmo genético apresentado na Seção 5.2. Os resultados dos experimentos computacionais mostram que o algoritmo desenvolvido encontra soluções melhores do que as abordagens apresentadas na literatura. Uma segunda contribuição deste estudo foi propor o uso do censo populacional como uma ferramenta capaz de assessorar o algoritmo genético em suas tomadas de decisões, sem demandar grandes adaptações ou impactar o desempenho do algoritmo.

6.1 Questões em Aberto e Trabalhos Futuros

Ao longo deste trabalho, algumas questões em aberto foram identificadas e apontadas como trabalhos futuros. Em relação ao problema da dominação vetorial, a questão mais intrigante é a complexidade do problema quando restrito aos grafos de intervalo e de intervalo próprio. O interesse nestas classes pode ser justificado pela fronteira observada entre grafos *split*, intervalo próprio e *split*-indiferença. Essa fronteira motiva uma pergunta bastante pertinente: o problema passou a ser tratável computacionalmente exatamente nos grafos *split*-indiferença ou ele já era tratável em grafos de intervalo próprio (e em grafos de indiferença)?

Considerando a dificuldade encontrada nestas classes, optou-se por abordar o problema através de uma progressão no número de cliques maximais. Como apresentado durante o texto, a solução para grafos completos já era conhecida pela literatura e este trabalho apresentou um algoritmo linear para grafos com duas cliques maximais. A evolução natural foi o estudo de grafos com três cliques maximais. A solução conhecida para este caso é o algoritmo polinomial de Cicalese *et al.* [18], baseado no parâmetro *clique-width* e na k-expressão.

O próximo passo do estudo foi a classe dos grafos dominó $\cap$ intervalo próprio, uma vez que apresenta uma estrutura mais rígida e controlada do que os grafos de intervalo próprio. A motivação para estudar esta classe era a expectativa de que um algoritmo capaz de lidar com as interseções entre as cliques pudesse indicar uma metodologia para resolver o problema em grafos de intervalo próprio. Contudo, ainda não foi possível descrever um algoritmo linear, embora o método proposto por Cicalese *et al.* [18] possa resolver o problema da dominação vetorial nesta classe (todo grafo dominó $\cap$ intervalo próprio possui *clique-width* limitada). Ainda assim, o estudo desta classe pode ser apontado como uma questão em aberto, dada a possibilidade de explorar as limitações e caracterizações da classe em algoritmos mais eficientes ou mesmo como uma fonte de evidências sobre como resolver o problema da dominação vetorial em outras classes de grafos formadas pela interseção de cliques.

Outra questão em aberto é a complexidade computacional do problema da dominação vetorial quando restrito aos *directed path graphs*. Esta questão é um ponto de interesse em particular, pois pode indicar novas fronteiras na complexidade computacional já que o problema é tratável em grafos ptolemaicos e $\mathcal{NP}$-Completo em *undirected path graphs*.

Como a família dos grafos cordais é muito vasta, foi necessário delimitar o escopo deste trabalho. Desta forma, além das questões já apontadas, outra possibilidade para trabalhos futuros é continuar o estudo do problema da dominação vetorial em grafos cordais, incluindo outras subfamílias, como, por exemplo, os grafos fortemente cordais.

Em relação ao problema da seleção de alvos, uma questão que se mostrou interessante foi a complexidade inerente à propagação da regra do contágio. Durante os testes com redes sociais foi possível identificar que o método responsável pela otimização das soluções é um ponto bastante crítico no desempenho do algoritmo. Desta forma, um trabalho futuro neste tópico seria buscar formas de reduzir o custo computacional deste processo, como, por exemplo, a supressão desta etapa ao utilizar operadores randômicos que já asseguram de alguma forma a viabilidade da solução resultante.

Embora o algoritmo genético tenha se mostrado capaz de encontrar resultados melhores do que os anteriormente conhecidos, uma possibilidade para trabalhos futuros seria o seu aperfeiçoamento. Uma das ideias para isso seria encontrar um ajuste mais preciso para os parâmetros do algoritmo utilizando o pacote *IRACE*, proposto por López-Ibáñez *et al.* [63]. Como mencionado no Capítulo 5, uma questão também bastante interessante seria a adaptação do algoritmo genético para o problema da dominação vetorial.

Durante o estudo do problema da dominação vetorial e do problema da seleção de alvos foram identificadas algumas variações intrigantes dos problemas. Dentre todas as variações, três devem ser destacadas como questões pertinentes para trabalhos futuros. A primeira é o problema da vacinação, que busca um conjunto mínimo capaz de impedir o grafo de ser inteiramente dominado. Como um assunto bastante atual é a propagação de *fake news* em redes sociais, o problema da vacinação pode ser interpretado como um conjunto mínimo de usuários que devem ser esclarecidos sobre um assunto para evitar que uma notícia falsa se propague pela rede.

A segunda variação dos problemas citados envolve atribuir pesos aos vértices do grafo. Enquanto os problemas abordados buscam conjuntos de cardinalidade mínima, a variação com pesos associados aos vértices visa identificar um conjunto capaz de dominar o grafo, tal que a soma dos pesos dos vértices neste conjunto seja mínima.

A terceira variação também é baseada na atribuição de pesos, mas às arestas. Esta terceira variação permite, por exemplo, modelar o problema em redes sociais de forma mais detalhada, considerando a intensidade de cada relação na rede social. Uma relação mais próxima e intensa pode ter mais impacto na propagação da opinião do que uma mais

distante, por exemplo. Outra possibilidade é modelar a propagação de doenças com mais precisão, pois a proximidade entre as pessoas pode ser explorada como um fator que favorece o contágio.

6.2 Resultados e Publicações

Durante o desenvolvimento deste trabalho, alguns dos resultados obtidos foram publicados como artigos e apresentados em congressos. A primeira contribuição foi o artigo Mafort e Ochi, "An Efficient Ant Colony Algorithm for The Minimum Latency Problem", 2016 [64], que propôs uma meta-heurística bioinspirada para o Problema da Latência Mínima[1]. Em seguida, o resultado do levantamento do estado da arte do problema da dominação vetorial na família do grafos cordais foi apresentado no I Workshop Escola de Inverno em Teoria da Computação (2017, Niterói).

Com o desenvolvimento do trabalho, o algoritmo de tempo $O(n)$ obtido para o problema da dominação vetorial em grafos *split*-indiferença foi apresentado no 3º Encontro de Teoria da Computação (2018, Natal) e no VIII *Latin American Workshop on Cliques in Graphs* (2018, Rio de Janeiro). Este resultado foi publicado na forma de um artigo completo na *Information Processing Letters* (Mafort e Protti, "Vector Domination in split-indifference graphs", 2020 [65]).

Os resultados obtidos pelo algoritmo genético para o problema da seleção de alvos foram submetidos para publicação na *Applied Soft Computing* como um artigo completo Mafort e Protti, "A Census-Based Genetic Algorithm for the Target Set Selection Problem in Social Networks", 2020 [66]. Além do algoritmo genético proposto e dos resultados obtidos, este trabalho propôs também o uso do censo populacional como um componente promissor no aperfeiçoamento das tomadas de decisão dos algoritmos genéticos.

[1] O Problema da Latência Mínima é uma variação do Problema do Caixeiro Viajante, cujo objetivo é localizar em um grafo completo $G = (V, E)$ com pesos nas arestas um ciclo hamiltoniano que minimize o somatório das latências dos vértices. A latência de um vértice $v \in V$ é a soma dos pesos das arestas contidas no caminho entre v e o vértice inicial do ciclo.

Referências

[1] Rémy Belmonte e Martin Vatshelle**. "Graph classes with structured neighborhoods and algorithmic applications"**. Em: *Theoretical Computer Science* 511 (nov. de 2013), pp. 54–65. ISSN: 03043975. DOI: 10.1016/j.tcs.2013.01.011.

[2] Oren Ben-Zwi, Danny Hermelin, Daniel Lokshtanov e Ilan Newman**. "Treewidth governs the complexity of target set selection"**. Em: *Discrete Optimization* 8.1 (fev. de 2011), pp. 87–96. ISSN: 15725286. DOI: 10.1016/j.disopt.2010.09.007.

[3] Alan A. Bertossi**. "Dominating sets for split and bipartite graphs"**. Em: *Information Processing Letters* 19.1 (jul. de 1984), pp. 37–40. ISSN: 00200190. DOI: 10.1016/0020-0190(84)90126-1.

[4] Jean R.S. Blair**. "The efficiency of AC graphs"**. Em: *Discrete Applied Mathematics* 44.1-3 (jul. de 1993), pp. 119–138. ISSN: 0166218X. DOI: 10.1016/0166-218X(93)90227-F.

[5] Paulo Oswaldo Boaventura Netto**. *Grafos: Teoria, Modelos, Algoritmos***. 5ª ed. Edgard Blucher Ltda, 2012, p. 314. ISBN: 9788521206804.

[6] John Adrian Bondy e Uppaluri Siva Ramachandra Murty**. *Graph Theory***. 2008.

[7] Kellogg S. Booth**. *Dominating Sets in Chordal Graphs***. Rel. técn. University of Waterloo, 1980, p. 12.

[8] Kellogg S. Booth e J.Howard Johnson**. "Dominating sets in chordal graphs."** Em: *SIAM J. Comput.* 11.1 (1982), pp. 191–199. ISSN: 0097-5397; 1095-7111/e. DOI: 10.1137/0211015.

[9] Andreas Brandstädt, Van Bang Le e Jeremy P. Spinrad**. *Graph Classes: A Survey***. Society for Industrial e Applied Mathematics, jan. de 1999. ISBN: 978-0-89871-432-6. DOI: 10.1137/1.9780898719796.

[10] Binh-Minh Bui-Xuan, Jan Arne Telle e Martin Vatshelle**. "Boolean-width of graphs"**. Em: *Theoretical Computer Science* 412.39 (set. de 2011), pp. 5187–5204. ISSN: 03043975. DOI: 10.1016/j.tcs.2011.05.022.

[11] Binh-Minh Bui-Xuan, Jan Arne Telle e Martin Vatshelle**. "Fast dynamic programming for locally checkable vertex subset and vertex partitioning problems"**. Em: *Theoretical Computer Science* 511 (nov. de 2013), pp. 66–76. ISSN: 03043975. DOI: 10.1016/j.tcs.2013.01.009.

[12] Peter Buneman**. "A characterisation of rigid circuit graphs"**. Em: *Discrete Mathematics* 9.3 (set. de 1974), pp. 205–212. ISSN: 0012365X. DOI: 10.1016/0012-365X(74)90002-8.

[13] Carmen C. Centeno, Mitre Costa Dourado, Lucia Draque Penso, Dieter Rautenbach e Jayme Luiz Szwarcfiter**. "Irreversible conversion of graphs"**. Em: *Theoretical Computer Science* 412.29 (2011), pp. 3693–3700. ISSN: 03043975. DOI: 10.1016/j.tcs.2011.03.029.

[14] Gerard Jennhwa Chang**. "Algorithmic Aspects of Domination in Graphs"**. Em: *Handbook of Combinatorial Optimization*. New York, NY: Springer New York, 2013, pp. 221–282. DOI: 10.1007/978-1-4419-7997-1_26.

[15] Steven Chaplick, Marisa Gutierrez, Benjamin Lévêque e Silvia B. Tondato**. "From Path Graphs to Directed Path Graphs"**. Em: ed. por Dimitrios M. Thilikos. Vol. 6410. Lecture Notes in Computer Science January 2017. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 256–265. ISBN: 978-3-642-16925-0. DOI: 10.1007/978-3-642-16926-7_24.

[16] Ning Chen**. "On the Approximability of Influence in Social Networks"**. Em: *SIAM Journal on Discrete Mathematics* 23.3 (jan. de 2009), pp. 1400–1415. ISSN: 0895-4801. DOI: 10.1137/08073617X.

[17] Nina Chiarelli, Tatiana Romina Hartinger, Valeria Alejandra Leoni, Maria Inés Lopez Pujato e Martin Milanič**. "New algorithms for weighted k-domination and total k-domination problems in proper interval graphs"**. Em: *Theoretical Computer Science* 795 (2019), pp. 128–141. ISSN: 03043975. DOI: 10.1016/j.tcs.2019.06.007.

[18] Ferdinando Cicalese, Gennaro Cordasco, Luisa Gargano, Martin Milanič e Ugo Vaccaro**. "Latency-bounded target set selection in social networks"**. Em: *Theoretical*

Computer Science 535 (2014), pp. 1–15. ISSN: 03043975. DOI: 10.1016/j.tcs.2014.02.027.

[19] Ferdinando Cicalese, Martin Milanič e Ugo Vaccaro**. "On the approximability and exact algorithms for vector domination and related problems in graphs"**. Em: *Discrete Applied Mathematics* 161.6 (abr. de 2013), pp. 750–767. ISSN: 0166218X. DOI: 10.1016/j.dam.2012.10.007.

[20] Gennaro Cordasco, Luisa Gargano, Marco Mecchia, Adele A. Rescigno e Ugo Vaccaro**. "A Fast and Effective Heuristic for Discovering Small Target Sets in Social Networks"**. Em: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. por Zaixin Lu, Donghyun Kim, Weili Wu, Wei Li e Ding-Zhu Du. Vol. 9486. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 193–208. ISBN: 978-3-319-26625-1. DOI: 10.1007/978-3-319-26626-8_15.

[21] Gennaro Cordasco, Luisa Gargano, Marco Mecchia, Adele A. Rescigno e Ugo Vaccaro**. "Discovering Small Target Sets in Social Networks: A Fast and Effective Algorithm"**. Em: *Algorithmica* 80.6 (jun. de 2018), pp. 1804–1833. ISSN: 0178-4617. DOI: 10.1007/s00453-017-0390-5.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein**. "Algoritmos Teoria e Prática: Soluções"**. Em: (2012).

[23] Bruno Courcelle**. "The monadic second-order logic of graphs : Definable sets of finite graphs"**. Em: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 344 LNCS. 726. 1989, pp. 30–53. ISBN: 9783540507284. DOI: 10.1007/3-540-50728-0_34.

[24] Bruno Courcelle**. "The monadic second-order logic of graphs VIII: Orientations"**. Em: *Annals of Pure and Applied Logic* 72.2 (mar. de 1995), pp. 103–143. ISSN: 01680072. DOI: 10.1016/0168-0072(95)94698-V.

[25] Bruno Courcelle, Joost Engelfriet e Grzegorz Rozenberg**. "Handle-rewriting hypergraph grammars"**. Em: *Journal of Computer and System Sciences* 46.2 (1993), pp. 218–270. ISSN: 10902724. DOI: 10.1016/0022-0000(93)90004-G.

[26] Bruno Courcelle, Johann A. Makowsky e Udi Rotics**. "Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width"**. Em: *Theory of Computing Systems* 33.2 (mar. de 2000), pp. 125–150. ISSN: 1432-4350. DOI: 10.1007/s002249910009.

[27] Bruno Courcelle e Stephan Olariu**. "Upper bounds to the clique width of graphs"**. Em: *Discrete Applied Mathematics* 101.1-3 (abr. de 2000), pp. 77–114. ISSN: 0166218X. DOI: 10.1016/S0166-218X(99)00184-5.

[28] Sanjoy Dasgupta, Christos H. Papadimitriou e Umesh Vazirani**. *Algorithms***. 1ª ed. McGraw-Hill Education, 2006, p. 336. ISBN: 9780073523408.

[29] Reinhard Diestel**. *Graph Theory***. 5ª ed. Springer, 2017, p. 448. ISBN: 3662536218.

[30] Paul Dietz**. "Intersection Graph Algorithms"**. Tese de dout. Cornell University, 1984, p. 148.

[31] Paul Dietz, Merrick Furst e John Hopcroft**. *A linear time algorithm for the generalized consecutive retrieval problem***. Rel. técn. 1979, p. 54.

[32] Thang N. Dinh, Huiyuan Zhang, Dzung T. Nguyen e My T. Thai**. "Cost-Effective Viral Marketing for Time-Critical Campaigns in Large-Scale Social Networks"**. Em: *IEEE/ACM Transactions on Networking* 22.6 (dez. de 2014), pp. 2001–2011. ISSN: 1063-6692. DOI: 10.1109/TNET.2013.2290714.

[33] Gabriel Andrew Dirac**. "On rigid circuit graphs"**. Em: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 25.1 (1961), pp. 71–76. ISSN: 1865-8784. DOI: 10.1007/BF02992776.

[34] Mitre C. Dourado, Carlos Vinícius G C Lima e Jayme Luiz Szwarcfiter**. "On f-Reversible Processes on Graphs"**. Em: *Electronic Notes in Discrete Mathematics* 50 (2015), pp. 231–236. ISSN: 15710653. DOI: 10.1016/j.endm.2015.07.039.

[35] Mitre Costa Dourado, John G. Gimbel, Jan Kratochvíl, Fábio Protti e Jayme Luiz Szwarcfiter**. "On the computation of the hull number of a graph"**. Em: *Discrete Mathematics* 309.18 (2009), pp. 5668–5674. ISSN: 0012365X. DOI: 10.1016/j.disc.2008.04.020.

[36] Mitre Costa Dourado, Lucia Draque Penso, Dieter Rautenbach e Jayme Luiz Szwarcfiter**. "Reversible iterative graph processes"**. Em: *Theoretical Computer Science* 460.September 2010 (2012), pp. 16–25. ISSN: 03043975. DOI: 10.1016/j.tcs.2012.05.042.

[37] Mitre Costa Dourado, Fábio Protti, Dieter Rautenbach e Jayme Luiz Szwarcfiter**. "On the Convexity Number of Graphs"**. Em: *Graphs and Combinatorics* 28.3 (2012), pp. 333–345. DOI: 10.1007/s00373-011-1049-7.

[38] Paul A. Dreyer e Fred S. Roberts**. "Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion"**. Em: *Discrete Applied Mathematics* 157.7 (2009), pp. 1615–1627. ISSN: 0166218X. DOI: 10.1016/j.dam.2008.09.012.

[39] Irène Durand e Michael Raskin**. *TRAG System***. http://trag.labri.fr/. Accessed: 2019-06-21. 2018.

[40] Martin Farber**. "Independent domination in chordal graphs"**. Em: *Operations Research Letters* 1.4 (set. de 1982), pp. 134–138. ISSN: 01676377. DOI: 10.1016/0167-6377(82)90015-3.

[41] Martin Farber**. "Domination, independent domination, and duality in strongly chordal graphs"**. Em: *Discrete Applied Mathematics* 7.2 (fev. de 1984), pp. 115–130. ISSN: 0166218X. DOI: 10.1016/0166-218X(84)90061-1.

[42] Noah E. Friedkin**. "A formal theory of social power"**. Em: *The Journal of Mathematical Sociology* 12.2 (ago. de 1986), pp. 103–126. ISSN: 0022-250X. DOI: 10.1080/0022250X.1986.9990008.

[43] Delbert Fulkerson e Oliver Gross**. "Incidence matrices and interval graphs"**. Em: *Pacific Journal of Mathematics* 15.3 (set. de 1965), pp. 835–855. ISSN: 0030-8730. DOI: 10.2140/pjm.1965.15.835.

[44] Martin Gardner**. "The fantastic combinations of John Conway's new solitaire game "life""**. Em: *Scientific American* 223 (1970), pp. 120–123.

[45] Michael R. Garey e David S. Johnson**. *Computers and Intractability; A Guide to the Theory of NP-Completeness***. W. H. Freeman \& Co., 1990. ISBN: 0716710455.

[46] Fănică Gavril**. "The intersection graphs of subtrees in trees are exactly the chordal graphs"**. Em: *Journal of Combinatorial Theory, Series B* 16.1 (fev. de 1974), pp. 47–56. ISSN: 00958956. DOI: 10.1016/0095-8956(74)90094-X.

[47] Fănică Gavril**. "A recognition algorithm for the intersection graphs of directed paths in directed trees"**. Em: *Discrete Mathematics* 13.3 (1975), pp. 237–249. DOI: http://www.sciencedirect.com/science/article/pii/0012365X75900217.

[48] P. C. Gilmore e A. J. Hoffman**. "A characterization of comparability graphs and of interval graphs"**. Em: *Canadian Journal of Mathematics* 16 (jan. de 1964), pp. 539–548. ISSN: 1496-4279. DOI: 10.4153/CJM-1964-055-5.

[49] Martin Charles Golumbic**. *Algorithmic Graph Theory and Perfect Graphs***. 1980.

[50] Martin Charles Golumbic e Udi Rotics**. "On the clique-width of perfect graph classes (extended abstract)."** Em: *Graph theoretic concepts in computer science. 25th international workshop, WG '99, Ascona, Switzerland, June 17–19, 1999. Proceedings*. Berlin: Springer, 1999, pp. 135–147. ISBN: 3-540-66731-8.

[51] Google**. *Google Cloud Platform – Compute Engine***. 2008. URL: https://cloud.google.com/compute/?hl=en-us (acesso em 18/10/2019).

[52] Teresa. W. Haynes, Stephen. Hedetniemi e Peter. J. Slater**. *Fundamentals of Domination in Graphs***. 1998, p. 446. ISBN: 0-8247-0033-3/hbk.

[53] Pascal Van Hentenryck**. *The OPL Optimization Programming Language***. The MIT Press, 1999, p. 255. ISBN: 9780262720304.

[54] Edward Howorka**. "A characterization of ptolemaic graphs"**. Em: *Journal of Graph Theory* 5.3 (1981), pp. 323–331. ISSN: 03649024. DOI: 10.1002/jgt.3190050314.

[55] IBM**. *IBM ILOG CPLEX Optimization Studio***. 2009. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio (acesso em 18/10/2019).

[56] David C Kay e Gary Chartrand**. "A Characterization of Certain Ptolemaic Graphs"**. Em: *Canadian Journal of Mathematics* 17 (1965), pp. 342–346. DOI: 10.4153/CJM-1965-034-0.

[57] David Kempe, Jon Kleinberg e Éva Tardos**. "Maximizing the spread of influence through a social network"**. Em: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*. New York, New York, USA: ACM Press, 2003, p. 137. ISBN: 1581137370. DOI: 10.1145/956755.956769.

[58] K. Khoshkhah, M. Nemati, H. Soltani e M. Zaker**. "A Study of Monopolies in Graphs"**. Em: *Graphs and Combinatorics* 29.5 (set. de 2013), pp. 1417–1427. ISSN: 0911-0119. DOI: 10.1007/s00373-012-1214-7.

[59] A J J Kloks, D Kratsch e H Müller. ***Dominoes***. Computing science notes. Technische Universiteit Eindhoven, 1994.

[60] James K. Lan e Gerard Jennhwa Chang. **"Algorithmic aspects of the k-domination problem in graphs"**. Em: *Discrete Applied Mathematics* 161.10-11 (jul. de 2013), pp. 1513–1520. ISSN: 0166218X. DOI: 10.1016/j.dam.2013.01.015.

[61] Jure Leskovec e Andrej Krevl. ***SNAP Datasets: Stanford Large Network Dataset Collection***. Jun. de 2014. URL: http://snap.stanford.edu/data (acesso em 18/10/2019).

[62] Jure Leskovec e Rok Sosič. **"SNAP: A General-Purpose Network Analysis and Graph-Mining Library"**. Em: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), p. 1.

[63] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie [Pérez Cáceres], Mauro Birattari e Thomas Stützle. **"The irace package: Iterated racing for automatic algorithm configuration"**. Em: *Operations Research Perspectives* 3 (2016), pp. 43–58. ISSN: 2214-7160. DOI: https://doi.org/10.1016/j.orp.2016.09.002.

[64] Rodrigo Lamblet Mafort e Luiz Satoru Ochi. **"An Efficient Ant Colony Algorithm for The Minimum Latency Problem"**. Em: (2016), pp. 1752–1763.

[65] Rodrigo Lamblet Mafort e Fábio Protti. **"Vector Domination in split-indifference graphs"**. Em: *Information Processing Letters* 155 (mar. de 2020), p. 105899. ISSN: 00200190. DOI: 10.1016/j.ipl.2019.105899.

[66] Rodrigo Lamblet Mafort e Fábio Protti. **"A Census-Based Genetic Algorithm for the Target Set Selection Problem in Social Networks"**. Ago. de 2020.

[67] Lilian Markenzon e Oswaldo Vernet. **"Representações Computacionais de Grafos"**. Em: *Notas em Matemática Aplicada*. Vol. 24. 2006, p. 78. ISBN: 85-86883-28-X.

[68] Lilian Markenzon e Christina F. E. M. Waga. **"Counting and enumerating unlabeled split–indifference graphs"**. Em: *Discrete Mathematics, Algorithms and Applications* 09.04 (ago. de 2017), p. 1750055. ISSN: 1793-8309. DOI: 10.1142/s1793830917500550.

[69] Lilian Markenzon e Christina Fraga Esteves Maciel Waga. **"New results on ptolemaic graphs"**. Em: *Discrete Applied Mathematics* 196 (dez. de 2015), pp. 135–140. ISSN: 0166218X. DOI: 10.1016/j.dam.2014.03.024.

[70] Lilian Markenzon e Christina Fraga Esteves Maciel Waga**. "Ferramentas Estruturais Em Grafos Cordais"**. Em: *Notas em Matemática Aplicada*. Vol. 82. 2016, p. 106. ISBN: 978-85-8215-073-3.

[71] Melanie Mitchell**. *An Introduction to Genetic Algorithms***. Fifth Edit. MIT Press, 1999, p. 158. ISBN: 0262631857.

[72] Clyde L. Monma e Victor K. Wei**. "Intersection graphs of paths in a tree"**. Em: *Journal of Combinatorial Theory, Series B* 41.2 (out. de 1986), pp. 141–181. ISSN: 00958956. DOI: 10.1016/0095-8956(86)90042-0.

[73] ***Open MPI: Open Source High Performance Computing***. 2004. URL: https://www.open-mpi.org/ (acesso em 09/12/2020).

[74] Carmen Z. Ortiz, Nelson Maculan e Jayme Luiz Szwarcfiter**. "Characterizing and edge-colouring split-indifference graphs"**. Em: *Discrete Applied Mathematics* 82.1-3 (mar. de 1998), pp. 209–217. ISSN: 0166218X. DOI: 10.1016/S0166-218X(97)00128-5.

[75] Matthew Richardson e Pedro Domingos**. "Mining knowledge-sharing sites for viral marketing"**. Em: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02*. Vol. 472. New York, New York, USA: ACM Press, 2002, p. 61. ISBN: 158113567X. DOI: 10.1145/775047.775057.

[76] F. S. Roberts**. "Indifference graphs"**. Em: *Proof techniques in graph theory*. Ed. por Frank Harary. New York: Academic Press, 1969, pp. 139–146. ISBN: 0123242606.

[77] Donald J. Rose, R. Endre Tarjan e George S. Lueker**. "Algorithmic Aspects of Vertex Elimination on Graphs"**. Em: *SIAM Journal on Computing* 5.2 (jun. de 1976), pp. 266–283. ISSN: 0097-5397. DOI: 10.1137/0205021.

[78] Jeremy P Spinrad**. *Efficient graph representations***. Fields Institute monographs. American Mathematical Society, 2003. ISBN: 978-1-4704-3146-4.

[79] Jayme Luiz Szwarcfiter**. *Teoria Computacional de Grafos***. 1ª ed. Elsevier Editora Ltda, 2018, p. 352. ISBN: 9788535288841.

[80] Reza Zafarani e Huan Liu**. *Social Computing Data Repository at ASU***. 2009. URL: http://socialcomputing.asu.edu (acesso em 18/10/2019).

APÊNDICE A – Biblioteca de Algoritmos

Durante o desenvolvimento dos algoritmos apresentados no Capítulo 4 foi necessário implementar um sistema que permitisse acelerar a obtenção de contra-exemplos para os métodos em desenvolvimento. Esse sistema evoluiu para um ambiente que busca pré-validar os algoritmos através de uma extensa bateria de testes com instância randomizadas. O ambiente foi chamado de pré-validador pois a ausência de contra-exemplos após um grande número de testes com diferentes tamanhos de grafos é um indicativo de que o método em análise pode estar correto. Entretanto, vale ressaltar que a corretude do método só pode ser assegurada depois de uma demonstração formal, e, portanto, o ambiente não pode ser denominado um validador ou certificador de algoritmos.

A construção deste ambiente demandou a implementação de uma biblioteca para manipulação computacional de grafos. Tendo em vista que os Capítulos 3 e 4 abordaram subfamílias de grafos cordais, essa biblioteca contém também implementações de ferramentas e características estruturais inerentes a cada família de grafos. Destaca-se neste ponto o desenvolvimento de algoritmos para construção aleatória de grafos em geral e também para algumas famílias em específico. Vale observar que a biblioteca de manipulação de grafos foi utilizada também na implementação do algoritmo genético apresentado no Capítulo 5.

Este capítulo apresentará a biblioteca de algoritmos desenvolvida. Entretanto, visando permitir o uso de cada parte do trabalho em separado, a biblioteca foi dividida em quatro partes. A primeira, detalhada na Seção A.1, contém as ferramentas e a modelagem computacional dos grafos e algumas de suas características estruturais. Além da manipulação de grafos, essa primeira parte contém também as rotinas atreladas ao vetor de requisitos, utilizado tanto no problema da dominação vetorial quanto no problema da seleção de alvos. Dentre essas rotinas, destacam-se os algoritmos para verificar se um determinado conjunto é suficiente para R-dominar o grafo. Essa primeira parte é totalmente independente das demais e pode ser utilizada em outros projetos de algoritmos em grafos.

A segunda parte, abordada na Seção A.2 contém as rotinas do ambiente de pré-validação de algoritmos.Esta seção abrange também os algoritmos *backtracking* e os modelos de programação linear inteira para os problemas da dominação vetorial e da seleção de alvos.

A terceira parte contém as implementações dos algoritmos abordados nos Capítulos 3 e 4, enquanto a quarta e última apresenta os métodos descritos no Capítulo 5.

A.1 Ferramentas Estruturais e Modelagem Computacional de Grafos

A primeira questão que deve ser considerada quando se modela computacionalmente um grafo é qual representação será adotada para identificar as relações (arestas) entre os vértices. Dentre todas as estruturas descritas na literatura de teoria dos grafos, a que mais se adequou aos problemas de dominação de vértices foi a lista de adjacências, dada a possibilidade imediata de identificar e percorrer o conjunto de adjacentes de cada vértice.

Além das rotinas básicas de manipulação de vértices e arestas, foram implementadas rotinas para obtenção de subgrafos induzidos, verificação de vértices gêmeos verdadeiros e falsos.

O Capítulo 2 apresentou a família dos grafos cordais, suas características e ferramentas estruturais, como, por exemplo, a obtenção de árvores de cliques maximais e de esquemas de eliminação perfeita. Vale observar que a árvore de cliques foi utilizada neste trabalho como um meio confiável de se obter as cliques maximais do grafo, utilizadas como referência para a solução de algumas subclasses de grafos, como, por exemplo, o algoritmo para grafos *split*-indiferença.

Considerando que os algoritmos foram codificados em C++, foi uma escolha natural implementar as famílias de grafos como classes (programação orientada a objetos). Essa estrutura permitiu estabelecer uma relação de herança entre as classes implementadas. Desta forma, as subfamílias dos grafos cordais herdam as propriedades e as funções implementadas para grafos cordais, assim como as definidas para grafos em geral.

Como mencionado anteriormente, a biblioteca contempla também algoritmos para construção randomica de grafos de cada família abordada. Para grafos em geral, a construção de uma instância tem como entradas o número de vértices desejados e uma probabili-

dade p associada a existência de cada aresta. Este método de construção foi apresentado no Capítulo 5 e utilizado nos testes do algoritmo genético em instâncias aleatórias. O Algoritmo A.1 apresenta o procedimento aplicado.

Vale observar que em problemas de dominação, as componentes conexas de um grafo desconexo podem ser vistas como instâncias menores do problema original, de forma que a solução para o grafo original equivale à união das soluções para cada componente. Desta forma, para garantir que o tamanho da instância será observado, optou-se por considerar apenas grafos conexos.

Para avaliar se o grafo é resultante é conexo, o Algoritmo A.1 verifica se é possível atingir todos os vértices do grafo partindo de v_1. Caso exista algum vértice não atingido pelo processo de busca, o algoritmo descarta o grafo construído e reinicia o processo. O descarte do grafo visa evitar a manipulação de arestas necessária para estabelecer a conexidade do grafo, uma vez que qualquer alteração nas arestas do grafo pode ser interpretada como uma adulteração do método randômico.

Além deste algoritmo, também foram desenvolvidos métodos para a construção de grafos de intervalo, intervalo próprio, *split*-indiferença, grafos dominó $\cap$ intervalo próprio e grafos com duas e três cliques maximais. É importante destacar que a habilidade de construir grafos de forma randômica é uma parte fundamental para o ambiente de pré-validação de algoritmos, pois possibilita maior diversidade de entradas para os algoritmos em estudo.

Como apresentado anteriormente, essa parte da biblioteca contém também as rotinas para manipulação do vetor de requisitos. Tendo em vista a necessidade de criar requisitos de forma aleatória, foram implementados alguns métodos para inicializar randomicamente o vetor de requisitos, respeitando o grau de cada vértice. Dentre as rotinas destinadas à manipulação de requisitos, constam também métodos para verificar se um determinado conjunto domina o grafo, tanto no problema da dominação vetorial, quanto no problema da seleção de alvos.

Os algoritmos e a modelagem descrita nesta seção foram publicados na plataforma GitHub no endereço `https://github.com/rodrigomafort/BibGrafos`.

Uma vez que a representação computacional dos grafos foi detalhada, pode-se prosseguir para apresentação mais detalhada do ambiente de pré-validação.

Algoritmo A.1: Construtor de Grafos Randômicos

Entrada: Número de vértices n e Probabilidade p
Saída: Grafo conexo $G = (V, E)$

repita
  $V \leftarrow \{v_1, v_2, \ldots, v_n\}$
  $E \leftarrow \emptyset$
  para $(i \leftarrow 1; \quad i \leq n; \quad i++)$ **faça**
    para $(j \leftarrow i+1; \quad j \leq n; \quad j++)$ **faça**
      Seja r um número real obtido aleatoriamente do intervalo $[0, 1]$
      se $r \leq p$ **então**
        $E \leftarrow E \cup \{\{v_i, v_j\}\}$
até `TestarConexidade` *(* $G = (V, E)$ *)* = **Verdadeiro**;
retorne $G = (V, E)$

Função `TestarConexidade` $(G = (V, E))$
  para $v \in V$ **faça** $A[c] \leftarrow$ **Falso** ;
  $A[v_1] \leftarrow$ **Verdadeiro** $\quad ct_A \leftarrow 1$
  $Q \leftarrow \{v_1\}$
  enquanto $Q \neq \emptyset$ **faça**
    $v \leftarrow$ `Desenfileirar`(Q)
    para $u \in N(v)$ **faça**
      se $A[u] =$ **Falso então**
        $A[u] \leftarrow$ **Verdadeiro** $\quad ct_A \leftarrow ct_A + 1$
        `Enfileirar` (Q,u)
  retorne $ct_A = |V|$

A.2 Ambiente de Pré-Validação de Algoritmos

Durante a elaboração dos algoritmos propostos no Capítulo 4, houve a necessidade de criar uma ferramenta para acelerar os testes de pré-validação destes métodos. Essa etapa de pré-validação consiste em múltiplas execuções do algoritmo com diferentes entradas para, em seguida, comparar os resultados obtidos com um método certificador.

A automatização destes testes reduziu significativamente o tempo necessário para encontrar possíveis contra-exemplos para um algoritmo, permitindo seu contínuo aprimoramento até o ponto em que seja possível demonstrar formalmente sua corretude. A pré-validação de um algoritmo requer três componentes principais: a construção randomizada de entradas, o algoritmo alvo da validação e um método certificador. Esses componentes são orquestrados por um sistema automatizado.

Como visto anteriormente, a implementação da construção randômica e das ferramentas estruturais de cada família de grafos foram reunidos em uma biblioteca para grafos, que foi então utilizada pelo ambiente de pré-validação.

Uma vez que a construção randômica de grafos tenha sido assegurada, a próxima etapa abordada foi como obter conjuntos R-dominantes com garantia de corretude, isto é, um método capaz de avaliar se a saída do algoritmo em teste é ótima ou não.

Para cada problema foram desenvolvidos algoritmos *backtracking* e modelos de programação linear inteira. Em relação aos algoritmos *backtracking*, considerando o momento em que cada método foi implementado durante o desenvolvimento deste trabalho, existem algumas diferenças de abordagem entre os métodos apresentados. A mais evidente delas é a otimização empregada nos algoritmos.

A.2.1 Métodos Exatos - Problema da Dominação Vetorial

O algoritmo *backtracking* desenvolvido para o problema da dominação vetorial foi projetado para execução em um único processador e sua otimização foi implementada na forma de um limite na profundidade da árvore de possibilidades explorada pelo algoritmo. Esse limite é dado em relação ao conjunto obtido pelo algoritmo em validação. Desta forma, seja S o conjunto retornado pelo algoritmo em teste. A partir desse resultado, o algoritmo *backtracking* inicia a busca exaustiva por conjuntos com no máximo $|S| - 1$ vértices (limite superior incluído) capaz de R-dominar o grafo. Caso algum conjunto com essas características seja localizado, ele é retornado e apresentado como um contra-exemplo para o algoritmo, encerrando a busca. Uma vez que toda as possibilidades de conjuntos foram exauridas sem encontrar algum que atenda aos requisitos (de tamanho e capacidade de R-dominar o grafo), certifica-se que para as entradas fornecidas o algoritmo está correto. É importante destacar que a corretude do algoritmo para um grafo e um vetor de requisitos não implica na corretude para toda e qualquer entrada. O Algoritmo A.2 demonstra o método *backtracking* desenvolvido.

A complexidade do algoritmo desenvolvido é $O(2^n)$, pois, no pior caso, todas as combinações possíveis de vértices devem ser testadas ($|S| = n$). Apesar da limitação imposta ao algoritmo (o tamanho máximo do conjunto), sua execução se mostrou extremamente lenta, além de demandar quantidades significativas de memória principal. Essa conclusão motivou a busca por uma nova alternativa de como resolver o problema.

Algoritmo A.2: Dominação Vetorial: Algoritmo *backtracking*

Entrada: Grafo $G = (V, E)$, Vetor de requisitos R, Conjunto S capaz de R-dominar G
Saída: S é um conjunto R-dominante para G e R?

```
1 Seja Sol um vetor de booleanos, onde cada posição corresponde a um vértice de G
2 para cada v ∈ V faça
3     Sol[v] = Falso
4 retorne Recursão (G,R,|S|,Sol,0,1)

1 Função Recursão (G, R, tamViavel, Sol, tamSol, i)
2     se (tamSol ≥ tamViavel) ∨ (i > |V|) então
3         retorne Falso
4     senão
5         C_Sol = {x ∈ V | Sol[x] = Verdadeiro }
6         se AvaliarConjunto (G,R,C_Sol) então
7             retorne Verdadeiro
8         senão
9             Seja v o vértice de rótulo i
10            Sol[v] = Verdadeiro
11            se Recursão (G,R,tamViavel,Sol,tamSol + 1,i + 1) então
12                retorne Verdadeiro
13            senão
14                Sol[v] = Falso
15                retorne Recursão (G,R,tamViavel,Sol,tamSol,i + 1)

1 Função AvaliarConjunto (G, R, C)
2     Dom = V \ C
3     para cada v ∈ Dom faça
4         se R[v] > |N(v) ∩ C| então
5             retorne Falso
6     retorne Verdadeiro
```

Para estabelecer este novo método foi necessário estudar o problema da dominação vetorial sob a ótica da programação linear inteira. Desta forma, uma nova modelagem para o problema foi desenvolvida e implementada utilizando o ambiente *IBM® ILOG® CPLEX® Optimization Studio* [55].

Seja um grafo $G = (V, E)$ representado pela sua matriz de adjacências ADJ. Além disso, seja R o vetor de requisitos. No modelo construído, o conjunto R-dominante é representado como um vetor de inteiros S onde cada posição corresponde a um vértice do grafo. Se neste vetor, a posição de um determinado vértice v contiver o número 1, então v está contido no conjunto R-dominante. Analogamente, se a posição contiver 0, então esse vértice deve ser dominado por seus vizinhos. É trivial perceber que o tamanho do conjunto equivale ao somatório das posições do vetor.

A verificação da dominação de um vértice por um determinado conjunto (representado como um vetor de inteiros) pode ser descrita como uma expressão matemática. Seja v um vértice do grafo e seja $R[v]$ seu requisito. Para determinar se v está dominado, basta que a seguinte expressão seja avaliada como verdadeira:

$$(S[v] * R[v]) + \left(\sum_{u \in V} ADJ[v, u] * S[u] \right) \geq R[v]$$

.

Se o vértice v estiver no conjunto R-dominante então a primeira parte da expressão é suficiente para satisfazer a condição. Do contrário, são necessários $R[v]$ vizinhos de v para isso. Um vértice u do grafo só contribuirá no somatório se for vizinho de v ($ADJ[v, u] = 1$) e se u estiver no conjunto ($S[u] = 1$). Se a expressão for satisfeita para todo vértice do grafo, então S é capaz de R-dominar G.

Sendo assim, o modelo foi definido como:

Função Objetivo:

$$\min \sum_{v \in V} S[v]$$

Sujeito a:

$$(S[v] * R[v]) + \left(\sum_{u \in V} ADJ[v, u] * S[u] \right) \geq R[v], \forall v \in V$$

$$ADJ[v,u] \in \{0,1\}, \forall v \in V, \forall u \in V$$

$$S[v] \in \{0,1\}, \forall v \in V$$

$$R[v] \in \{0,1,\ldots,d(v)\}, \forall v \in V$$

Uma vez que o modelo foi desenvolvido, ele foi implementado utilizando a linguagem OPL (*Optimization Programming Language* - Hentenryck [53]) para execução no ambiente CPLEX. Quando comparado ao algoritmo *backtracking*, esse modelo se mostrou mais eficiente, contudo sua execução ainda é muito custosa computacionalmente, especialmente quando são fornecidos como entrada grafos com mais de cem vértices.

O sistema de validação foi originalmente desenvolvido para o problema da dominação vetorial, contudo ele pode ser facilmente estendido para o problema da seleção de alvos. Como mencionado anteriormente, o ambiente precisa de um método de construção randomizada de entradas, o algoritmo em teste e de um método certificador. Considerando que as entradas construídas para o primeiro problema são entradas viáveis para o segundo, restou apenas desenvolver métodos certificadores para o problema da seleção de alvos.

A.2.2 Métodos Exatos - Problema da Seleção de Alvos

Assim como o problema da dominação vetorial, foram implementados um algoritmo *backtracking* e um método baseado em programação linear inteira. Tendo em vista o ambiente de processamento paralelo utilizado no Capítulo 5, um segundo algoritmo *backtracking* foi desenvolvido para explorar os recursos computacionais disponibilizados.

O Algoritmo A.3 apresenta o primeiro algoritmo *backtracking* desenvolvido. Dada as características do problema, a cada recursão do algoritmo, somente os vértices ainda não dominados tem sua inclusão no conjunto solução avaliada (linha 6 da função Recursão). Vale destacar que nesta mesma linha existe uma condição adicional: $v > v_{min}$. Essa condição garante que, em uma recursão do algoritmo, apenas vértices com rótulos maiores do que o último adicionado sejam incluídos. Desta forma, todos os conjuntos obtidos pela permutação de elementos são desconsiderados (apenas conjuntos ordenados são avaliados), reduzindo o tempo necessário para o processamento do algoritmo.

Visando ainda uma redução de tempo, quando um vértice é dominado, os efeitos da dominação são propagados em sua vizinhança (linhas 12 a 14 da função `Recursão`) e o vértice é removido do grafo. Essa remoção busca reduzir o número de vizinhos que devem

ser percorridos a cada vértice dominado.

Vale observar ainda que para permitir o retorno imediato de cada recursão, o estado anterior das variáveis de controle é mantido. Para isso, cada chamada a função recursiva estabelece uma cópia local das variáveis e manipula apenas esta cópia (linhas 7 e 8 da função `Recursão`).

Como esperado, as modificações reduziram o tempo de processamento, contudo este ainda é consideravelmente alto, mesmo em grafos de dimensões reduzidas. Vale observar que o custo computacional atrelado a propagação dos efeitos da dominação de um vértice se tornou um ponto crítico do algoritmo. Esse custo adicional não existe no problema da dominação vetorial, uma vez que os efeitos são imediatos. Além disso, como notado no algoritmo para o problema da dominação vetorial, o consumo de memória principal ainda é bastante significativo.

Para explorar os recursos computacionais disponíveis durante o estudo do problema da seleção de alvos, uma versão paralelizada deste algoritmo também foi implementada. Nesta versão cada processador recebe um vértice como ponto de partida. Esse vértice é adicionado ao conjunto e sua dominação é propagada. Tomando como base as mesmas variáveis de controle do Algoritmo A.3, atualizadas pela adição do vértice inicial, a rotina recursiva é inicializada. A função `Recursão` utilizada na versão paralela é igual à apresentada no Algoritmo A.3.

Como cada processador considera apenas soluções ordenadas, ao iniciar um conjunto com vértices distintos, é possível garantir que cada conjunto possível é analisado por somente um processador e que as permutações serão ignoradas. Por exemplo, se um processador receber o vértice v_x como ponto de partida, ele somente avaliará conjuntos ordenados formados por vértices de rótulos maiores que v_x.

Como cada processador recebe um ponto de partida distinto, é natural que cada um retorne uma solução diferente para as entradas (ou não retorne nenhuma solução). Desta forma, o algoritmo necessita de um passo adicional: comparar as soluções obtidas e identificar a de menor cardinalidade, que será então retornada como a saída do algoritmo para as entradas fornecidas. Além disso, considerando que o número de processadores disponíveis pode ser diferente do número de vértices do grafo, pode ser necessário que alguns processadores recebam mais de um ponto de partida para processamento ou mesmo que alguns fiquem sem nenhuma tarefa. A versão apresentada no Algoritmo A.4 assume que existe um processador para cada vértice do grafo.

Algoritmo A.3: Seleção de Alvos: Algoritmo *backtracking*

Entrada: Grafo $G = V, E$, Vetor de Requisitos R, Tamanho Máximo $tamMax$
Saída: Solução Ótima S_{otm}

```
1  S_otm ← V
2  S ← ∅
3  NDom ← V
4  para cada v ∈ V faça
5      MapaD[v] ← Falso        N[v] ← N(v)        CtD[v] ← 0
6  Recursão(NDom, CtD, MapaD, N, S, 0, tamMax)
7  retorne S_otm

1  Função Recursão(NDom, CtD, MapaD, N, S, min_v, tamMax)
2      se |S| ≤ tamMax então /* O tamanho de S ainda é aceitável
       */
3          se NDom = ∅ então                    /* S dominou o grafo */
4              se |S| < |S_otm| então S_otm ← S;
5          senão        /* Necessário adicionar mais vértices */
6              para cada v ∈ NDom | v > min_v faça
                   /* Avaliar os desdobramentos da inclusão de
                      v na solução                                  */
7                  NDom_2 ← NDom    MapaD_2 ← MapaD    CtD_2 ← CtD
                    N_2 ← N ;       /* cópias locais para garantir a
                    volta da recursão */
8                  S_2 ← S ∪ {v}       Q ← v
9                  enquanto Q ≠ ∅ faça
10                     u ← Desenfileirar(Q)
11                     para cada w ∈ N_2[u] faça
12                         CtD_2[w] ← CtD_2[w] + 1
13                         se CtD_2[w] ≥ R[w] ∧ MapaD_2[w] = Falso então
                               /* w foi dominado                   */
14                             Enfileirar (Q,w)    MapaD_2[w] ← Verdadeiro
15                         N_2[w] ← N_2[w] \ {u}
16                     N_2[u] ← ∅       NDom_2 ← Dom_2 \ {u}
17                 Recursão (NDom_2, CtD_2, MapaD_2, N_2, S_2, v, tamMax)
```

Vale observar que a implementação foi baseada no protocolo MPI, onde cada processador executa uma cópia do algoritmo, recebe as mesmas entradas e possui sua própria memória principal. Cabe a cada processador determinar e executar as tarefas pertinentes a ele, de acordo com seu identificador P.

Algoritmo A.4: Seleção de Alvos: Algoritmo *backtracking* paralelo

Entrada: Grafo $G = V, E$, Vetor de requisitos R, Tamanho máximo $tamMax$, Identificador do processador P e Número de processadores P_{total}
Saída: Solução Ótima S_{otm}
Requisito: $n = P_{total}$

1 $S_{otm} \leftarrow V$
2 $NDom \leftarrow V$
3 **para cada** $v \in V$ **faça**
4 $\quad MapaD[v] \leftarrow$ **Falso** $\quad N[v] \leftarrow N(v) \quad CtD[v] \leftarrow 0$
/* Processador P toma o vértice v_P como ponto de partida */
5 $S \leftarrow \{v_P\}$
/* Os efeitos da dominação de v_P são propagados */
6 $Q \leftarrow v_P$
7 **enquanto** $Q \neq \emptyset$ **faça**
8 $\quad u \leftarrow$ Desenfileirar(Q)
9 $\quad$ **para cada** $w \in N[u]$ **faça**
10 $\quad\quad CtD[w] \leftarrow CtD[w] + 1$
11 $\quad\quad$ **se** $CtD[w] \geq R[w] \wedge MapaD[w] =$ **Falso então**
$\quad\quad\quad$ /* w foi dominado */
12 $\quad\quad\quad$ Enfileirar $(Q,w) \quad MapaD[w] \leftarrow$ **Verdadeiro**
13 $\quad\quad N[w] \leftarrow N[w] \setminus \{u\}$
14 $\quad N[u] \leftarrow \emptyset \quad NDom \leftarrow Dom \setminus \{u\}$
/* Exploração da árvore de possibilidades $\{v_p, \ldots\}$ */
15 Recursão($NDom$, CtD, $MapaD$, N, S, v_P, $tamMax$)
16 **Condição de Barreira**: Aguardar todos os processadores
17 Sejam $S_1 \ldots S_{P_{total}}$ os conjuntos retornados por cada processador P
18 Seja S_{otm} a solução de menor cardinalidade dentre $\{S_1 \ldots S_{P_{total}}\}$
19 **retorne** S_{otm}

Tendo em vista a necessidade de bibliotecas adicionais e de um sistema computacional com capacidade para processamento em paralelo, a versão paralelizada do algoritmo *backtracking* foi omitida da biblioteca referente ao ambiente de pré-validação. A implementação em C++ deste algoritmo foi incluída com os fontes referentes ao algoritmo genético.

Além dos algoritmos *backtracking*, um modelo de programação linear inteira também foi desenvolvido para o problema da seleção de vértices.

Assim como na versão apresentada para o problema da dominação vetorial, o grafo é representado pela matriz de adjacências ADJ. O resultado corresponde a um vetor S ($S[u] \in \{0,1\}, \forall u \in V$), onde um valor 1 em uma posição v do vetor indica que o vértice v pertence à solução. Nota-se que o objetivo é encontrar um vetor com o menor número possível de valores 1 capaz de dominar o grafo.

Entretanto, tendo em vista que no problema da seleção de alvos são admitidas múltiplas iterações da dominação, foi necessário aprimorar o modelo anterior. Desta forma, uma matriz D foi utilizada para identificar quais vértices estão dominados em cada iteração. Cada vértice corresponde a uma linha da matriz, enquanto cada iteração i é associada a uma coluna. Dado que existem n vértices e que a dominação pode demandar no máximo n iterações até sua estabilização, a matriz D possui n linhas e $n+1$ colunas: a coluna 0 corresponde a solução inicial, enquanto a coluna n, ao resultado final do processo. Como o conjunto deve dominar todos os vértices, o objetivo do modelo é identificar uma atribuição ótima de 0 e 1 em $S[v]$ ($D[v][0] = S[v]$) de forma que $D[v][n] = 1$ para todo $v \in V$.

Desta forma, a regra da dominação é modelada em duas regras:

- Na primeira, $D[v,k+1] \geq D[v,k], \forall k \in \{0,\ldots,n-1\}, \forall v \in V$, é estabelecido que a dominação é irreversível, proibindo que um vértice dominado (valor 1) retorne ao valor 0 nas iterações seguintes.
- A segunda, $D[v,k+1] \leq D[v,k] + \frac{\sum_{j \in V} D[j,k] * ADJ[v,j]}{R[v]}, \forall k \in \{0,\ldots,n-1\}, \forall v \in V$, inibe trocas espontâneas, isto é, um vértice só pode assumir valor 1 em uma iteração se na anterior ele já estiver dominado ou possuir pelo menos $R[v]$ vizinhos dominados.

Desta forma, o modelo de programação linear inteira pode ser definido como:

Função Objetivo:

$$\min \sum_{v \in V} S[v]$$

Sujeito a:

$$D[v][0] = S[v], \forall v \in V$$

$$D[v][n] = 1, \forall v \in V$$

$$D[v,k+1] \geq D[v,k], \forall k \in \{0 \ldots n-1\}, \forall v \in V$$

$$D[v,k+1] \leq D[v,k] + \frac{\sum_{j \in V} D[j,k] * ADJ[v,j]}{R[v]}, \forall k \in \{0,\ldots,n-1\}, \forall v \in V$$

$$ADJ[v,u] \in \{0,1\}, \forall v \in V, \forall u \in V$$

$$S[v] \in \{0,1\}, \forall v \in V$$

$$R[v] \in \{0,1,\ldots,d(v)\}, \forall v \in V$$

Assim como o anterior, este modelo foi implementado utilizando a linguagem OPL, visando sua execução no ambiente IBM CPLEX. Considerando que este modelo necessita analisar cada iteração possível do processo da dominação, seu tempo de execução é bastante significativo e muito superior ao modelo destinado ao problema da dominação vetorial.

Vale frisar que o elevado custo computacional do algoritmo *backtracking* motivou a modelagem do problema utilizando programação linear inteira. Contudo, diferentemente do problema da dominação vetorial, a insatisfação com o elevado consumo de tempo deste para o modelo estimulou a implementação paralelizada do algoritmo *backtracking*. Em todas as soluções propostas ficou evidente que a propagação da regra da dominação ao longo de várias iterações é um ponto crítico no estudo do problema da seleção de alvos. Umas das propostas para trabalhos futuros é encontrar formas de otimizar este processo, principalmente durante trocas de vértices no conjunto solução.

A.2.3 Ambiente de Pré-Validação

Uma vez que todos os requisitos do sistema foram satisfeitos, o ambiente de pré-validação foi então implementado. Para acelerar o processo de pré-validação de algoritmos, o ambiente foi criado visando a execução em paralelo dos testes. É importante destacar que o ambiente computacional para o qual o sistema de pré-validação foi desenvolvido difere do apresentado no Capítulo 5, uma vez que a premissa inicial era executar os testes em computadores domésticos, sem exigências em específico sobre o número de processadores.

A técnica de programação que melhor atendia a esse critério é o uso de múltiplas *threads*. Nesta técnica, existe uma única cópia do programa em execução, que distribui partes de seu processamento entre as *threads*. Cada *thread* tem uma área de memória particular, embora também possa acessar a área alocada ao processo principal. Nesta técnica de

programação paralela não existe uma associação rígida entre processadores e *threads*, o que pode ocasionar um desbalanceamento da carga entre os processadores. Desta forma, considerando que o número de processadores é limitado, foi necessário estabelecer um equilíbrio entre o número de *threads* e o número de processadores disponíveis na máquina. Vale observar que a utilização desta técnica permitiu acelerar significativamente os testes necessários para pré-validar um algoritmo, contudo as peculiaridades da técnica e a necessidade de um ganho ainda maior em desempenho e escala motivaram o estudo de outra técnica de processamento paralelo, aplicada no algoritmo genético apresentado no Capítulo 5.

O ambiente de pré-validação foi implementado com duas formas de processamento. Na primeira, o algoritmo gera randomicamente um grafo com número de vértices previamente estabelecido, que atenda as restrições inerentes à classe objetivo. Em seguida, todas as combinações possíveis de requisitos são analisadas individualmente. Para isso, são iniciados dois processos que sincronizam os testes. O primeiro processo alimenta uma fila de entradas com os dados referentes aos testes que deve ser executados. Cada entrada da fila corresponde a uma cópia do grafo e uma das combinações de requisitos. Em paralelo, um segundo processo é iniciado para obter os resultados referentes aos testes e transcrevê-los para um arquivo de resultados (evitando condições de corrida e *deadlocks*).

Quando os processos responsáveis por sincronizar o ambiente estão em execução, o programa inicia múltiplas *threads*. Cada *thread* retira uma instância da fila de entradas, a processa e transfere os resultados para a fila de saída. Vale observar que o processamento de uma instância contempla a execução do algoritmo em validação e do método certificador. Após finalizar o processamento de uma instância, a *thread* retorna a fila de entradas em busca de novas tarefas, reiniciando o processo.

Tendo em vista que o número de combinações possíveis de requisitos tende a ser muito grande, uma segunda forma de processamento foi desenvolvida. Nesta segunda forma, são analisados vetores de requisitos gerados randomicamente, respeitando os limites relacionados ao grau de cada vértice. Essa mudança na forma de processar os vetores de requisitos dispensa o primeiro processo de controle descrito na metodologia anterior. Sendo assim, cada *thread* recebe durante sua inicialização o número de testes que devem ser executados e uma cópia do grafo. Em seguida, cada *thread* gera combinações aleatórias de requisitos e efetua os testes já descritos. O segundo processo de controle, responsável pela saída dos resultados, permanece. É importante destacar que essa segunda forma de processar o grafo não garante que cada combinação de requisitos será testada, pois, desta forma, evita-se a

complexidade inerente à geração de vetores de requisitos sem repetições.

Para cada grafo testado, o ambiente retorna um arquivo de texto, onde cada instância avaliada corresponde a uma linha de texto. Cada linha contém o vetor de requisitos utilizado, o resultado retornado pelo algoritmo em validação e se foi localizada algum conjunto de cardinalidade inferior a este. Caso tenha sido constatado que o algoritmo não apresentou o resultado ótimo, a linha contém também o conjunto R-dominante encontrado (um contra-exemplo para a instância).

A Figura A.1 apresenta os dois ambientes desenvolvidos. No primeiro, existe um controlador que gera todas as combinações de requisitos para o grafo e as enfileira. A seguir, cada *thread* obtém uma tarefa, a processa e salva os resultados em uma fila de saída. Uma vez que a *thread* termina de processar uma combinação, ela busca na fila a próxima tarefa. Os dados da fila de saída são lidos por uma *thread* controladora que os salva em arquivo. No segundo ambiente apresentado, o controlador apenas informa inicialmente as *threads* qual o grafo e quantos testes que devem ser executados. Em seguida, as *threads* se tornam autônomas, gerando os requisitos aleatoriamente e os processando. A dinâmica da saída dos resultados permanece inalterada. Em ambos os ambientes, as saídas das *threads* contém o conjunto retornado pelo algoritmo em teste S_{alg} e o conjunto R-dominante obtido pelo módulo de programação inteira S_{otm}. A partir desses resultados é possível aferir se existe algum caso para o qual o algoritmo não foi capaz de encontrar um conjunto R-dominante. O número de *threads* e de testes (aplicável somente ao segundo ambiente) são parâmetros dependentes da extensão pretendida para o teste e do desempenho da máquina.

Embora o ambiente tenha sido originalmente projetado para ser executado em um computador doméstico, optou-se por executá-lo utilizando o ambiente de computação nas nuvens (*Cloud Computing*) disponibilidade pelo Google (*Google Cloud Platform* [51]. Foram utilizadas máquinas com até 16 núcleos de processamento e 64 GB de memória principal (RAM), disponibilizadas como avaliação gratuita do ambiente. A utilização deste ambiente virtual tornou possível a execução contínua e ininterrupta de alguns testes por vários dias, o que seria bastante complexo em um computador doméstico. Vale mencionar que este primeiro contato com o ambiente de computação nas nuvens motivou o uso ainda mais intenso, como apresentado no Capítulo 5.

Vale ressaltar que a ideia do ambiente de pré-validação em nenhum momento foi demonstrar a corretude de um algoritmo, mas apenas fornecer contra-exemplos, que podem ser utilizados para orientar o aperfeiçoamento do algoritmo. Entretanto, quando um algo-

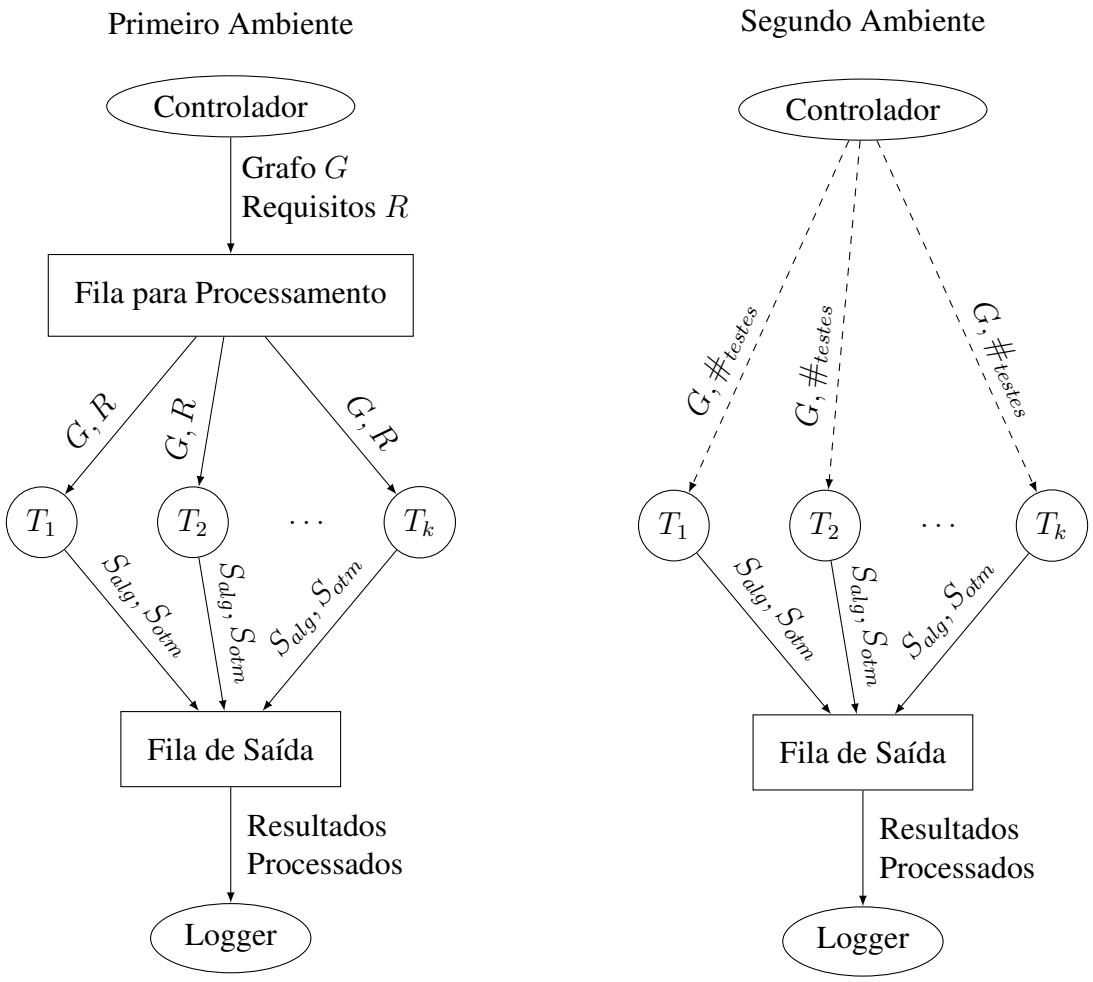


Figura A.1: Diagramas de funcionamento dos ambientes desenvolvidos.

ritmo passa com sucesso por uma grande bateria de testes, envolvendo várias instâncias de grafos de diferentes tamanhos, se tem um indício de sua corretude, o que motiva uma análise mais aprofundada. Deve ser esclarecido também que uma das etapas implícitas a análise dos algoritmos é avaliar se o conjunto retornado é de fato uma solução viável para o problema (o conjunto é suficiente para R-dominar o grafo).

Em relação à biblioteca disponibilizada, é importante notar que os modelos de programação linear inteira dependem do sistema IBM CPLEX [55] para sua execução, o que implica na dependência do mesmo para a compilação e utilização do sistema de pré-validação. Assim como a biblioteca de manipulação de grafos, o ambiente de pré-validação também foi disponibilizado na plataforma GitHub no endereço `https://github.com/rodrigomafort/BibValidador`.

A.3 Implementações dos Algoritmos - Capítulos 3, 4 e 5

O desenvolvimento dos Capítulos 3, 4 e 5 implicou na necessidade de implementar os algoritmos propostos e também alguns métodos da literatura. Tendo em vista a diferença entre as propostas dos Capítulos 3 e 4 em comparação com o Capítulo 5, cada problema de propagação foi formatado como uma parte distinta da biblioteca.

A primeira parte compreende os algoritmos e resultados apresentados nos Capítulos 3 e 4. Destacam-se as implementações dos algoritmos propostos para grafos *split*-indiferença e grafos com duas cliques maximais. Vale observar que métodos discutidos, mas ainda não comprovados, como, por exemplo, o algoritmo para grafos dominó $\cap$ intervalo próprio, foram omitidos por ainda carecerem de um maior desenvolvimento. Esta primeira parte está disponibilizada no GitHub através do endereço `https://github.com/rodrigomafort/VDCordais`.

Já a segunda parte é composta pelo algoritmo genético e por uma implementação em C++ do algoritmo proposto por Cordasco *et al.* [20, 21]. Além desses algoritmos, a biblioteca inclui também uma implementação da roleta viciada e a versão paralelizada do algoritmo *backtracking* para o problema da seleção de alvos.

Considerando o volume de dados gerados durante a execução do algoritmo genético (tais como tempos de execução, soluções obtidas e eficiência dos operadores) optou-se por utilizar um banco de dados *MySQL* como solução para o armazenamento. Desta forma, este projeto contém também o *script* SQL referente a implementação do banco de dados.

Além disso, tendo em vista que o pré-processamento realizado nas instâncias da literatura pode ter alterado de alguma forma a estrutura da instância e visando uma maior transparência nos resultados, os arquivos procedentes do pré-processamento também foram incluídos na biblioteca. Vale ressaltar que os arquivos binários resultantes possuem $n + m + 1$ entradas e que cada entrada contém dois números inteiros (*struct* em C++). A primeira entrada contém os valores de n e m. Em seguida, são apresentadas n entradas representando uma associação um para um entre os rótulos criados internamente, que são linearmente ordenados de 1 até n, e os rótulos originais dos vértices nas instâncias. As últimas m entradas contém as arestas do grafo, já tomando como base os rótulos internos como extremos de cada aresta.

Tendo em vista que o algoritmo genético e os resultados obtidos foram submetidos para publicação, uma parte do repositório referente ao Capítulo 5 ainda não foi tornada pública.

Entretanto, todo o repositório, incluindo o banco de dados com o resultado do processamento, será disponibilizado imediatamente após o parecer sobre a publicação. Uma parte da biblioteca referente ao problema da seleção de alvos foi disponibilizada no GitHub, no endereço `https://github.com/rodrigomafort/TSSGenetico`.

Finalmente, vale observar que as duas partes apresentadas nesta seção são independentes entre si, porém necessitam das bibliotecas de manipulação computacional de grafos e do ambiente de pré-validação.