

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Hoechst Cornélio Da Silva

**Grafos mergulhados em superfícies orientáveis:
Emparelhamento e Reconstrução**

Belo Horizonte - MG

2025

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE MATEMÁTICA

Hoechst Cornélio Da Silva

**Grafos mergulhados em superfícies orientáveis:
Emparelhamento e Reconstrução**

Tese apresentada ao Programa de Pós-Graduação em Matemática do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como parte dos requisitos para a obtenção do título de Doutor em Matemática.

Orientador: Bhalchandra Digambar Thatte
Coorientadora: Pouya Mehdipour (UFV)

Belo Horizonte - MG

2025

2025 Hoechst Cornélio da Silva.
Todos os direitos reservados

Silva, Hoechst Cornélio da.

S586g Grafos mergulhados em superfícies orientáveis:
[recurso eletrônico] emparelhamento e reconstrução /
Hoechst Cornélio da Silva. Belo Horizonte — 2024.
1 recurso online (144 f. il.): pdf.

Orientador: Bhalchandra Digambar Thatte.
Coorientadora: Pouya Mehdipour.

Tese (doutorado) - Universidade Federal de Minas
Gerais, Instituto de Ciências Exatas, Departamento de
Matemática.

Referências: f. 83-86

1. Matemática – Teses. 2. Topologia – Teses.
3. Superfícies (Matemática)– Teses. 4. Teoria dos
grafos – Teses. I. Thatte, Bhalchandra Digambar.
II. Mehdipour, Pouya. III. Universidade Federal de Minas
Gerais, Instituto de Ciências Exatas, Departamento de
Matemática. IV. Título.

CDU 51(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg
Lucas Cruz - CRB 6ª Região - nº 819.



FOLHA DE APROVAÇÃO

*Grafos mergulhados em superfícies orientáveis:
Emparelhamento e Reconstrução*

HOECHST CORNÉLIO DA SILVA

Tese defendida e aprovada pela banca examinadora constituída por:

Prof. Bhalchandra Digambar Thatte
Orientador - UFMG

Profª. Pouya Mehdipour
Coorientadora - UFV

Prof. Aldo Procacci
UFMG

Prof. Carlos Maria Carballo
UFMG

Profª. Catarina Mendes de Jesus Sánchez
UFJF

Profª. Esther Sanabria Codesal
(Univ. Politécnica de Valência, Espanha)

Belo Horizonte, 04 de agosto de 2025.

Agradecimentos

Primeiramente, agradeço a minha mãe, Rita de Cássia Cornélio, por dedicar muitos anos de sua vida a fazer de tudo, mesmo em situações em que seria natural desistir, para garantir que eu e meus irmãos entrássemos em uma Universidade Federal. Sem todo o seu esforço e incentivo, eu não teria chegado até aqui.

Aos meus irmãos, namorada e amigos pelo apoio durante essa longa jornada acadêmica, que por vezes não se desvencilhava da vivência pessoal.

Ao Bhalchandra pelo aceite da orientação, à Catarina pelas ricas discussões e trocas de ideias que tivemos, elas foram fundamentais para o desenvolvimento desse trabalho. E à Pouya pela coorientação, pela disposição, ricas sugestões, longas reuniões que tivemos e por todo apoio durante a trajetória do doutorado. Seu apoio foi fundamental para que eu pudesse concluir essa etapa com sucesso.

À Máisa Duarte por incentivar a pesquisa e desenvolvimento acadêmico, por flexibilizar as agendas para que a conclusão das disciplinas fosse feita da melhor forma possível. Possibilitando, dessa forma, a permanência no doutorado durante o período que trabalhamos juntos.

Por fim, mas não menos importante, agradeço aos professores que fizeram parte da minha formação e às pessoas que direta ou indiretamente contribuíram para a finalização do curso de doutorado.

"I have a dream!"
Martin Luther King Jr

Resumo

Neste trabalho investigamos grafos mergulhados em superfícies fechadas e orientáveis sob duas perspectivas: o emparelhamento de arestas e a reconstrução de mergulhos. No âmbito do emparelhamento de arestas, implementamos um algoritmo capaz de gerar explicitamente mergulhos canônicos em superfícies orientáveis. Mostramos que, para superfícies de gênero 1, 2, 3 e 4, existem exatamente 1, 4, 82 e 7258 emparelhamentos canônicos, respectivamente. Já no contexto da reconstrução topológica de mergulhos, demonstramos que todo mergulho de árvore é reconstrutível, desde que não contenha vértices de grau 1 acoplados ao centro, quando este for único. Além disso, no estudo da reconstrução de mergulhos celulares, provamos que os grafos de emparelhamento provenientes da cirurgia S_1 são reconstrutíveis como mergulhos e estabelecemos condições suficientes para garantir a reconstrutibilidade de mergulhos celulares em geral.

Palavras-chave: grafos; grafos em superfícies; emparelhamento de arestas; reconstrução; reconstrução de mergulhos; reconstrução topológica.

Abstract

In this work, we investigate graphs embedded in closed orientable surfaces from two complementary perspectives: edge pairings and embedding reconstruction. Regarding edge pairings, we implemented an algorithm that explicitly generates canonical embeddings on orientable surfaces. We show that for surfaces of genus 1, 2, 3, and 4, there exist exactly 1, 4, 82, and 7258 canonical pairings, respectively. Concerning the topological reconstruction of embeddings, we prove that every tree embedding is reconstructible, provided it does not contain degree-1 vertices attached to the center when the center is unique. Furthermore, in the study of cellular embedding reconstruction, we demonstrate that pairing graphs arising from the S_1 surgery are reconstructible as embeddings, and we establish sufficient conditions to ensure the reconstructibility of cellular embeddings in general.

Keywords: graphs; graphs on surfaces; edge pairing; reconstruction; embedding reconstruction; topological reconstruction.

Lista de Figuras

2.1	Representação local de grafo de emparelhamento sobre uma superfície	22
2.2	Grafo mergulhado sobre o toro e Bitoro	22
2.3	Grafo mergulhado sobre o toro e Bitoro - Imagem retirada de [9] . .	24
3.1	Grafo paralelo de uma árvore mergulhada em $\mathbb{S}^2$	32
3.2	Figura retirada de [8] - (a) Disposição de vértices associados em um grafo; (b) Representação, no <i>grafito</i> , de vértices associados.	33
3.3	Mergulho do grafo de emparelhamento com 3 vértices e 6 arestas no Bitoro.	40
3.4	Exemplo de emparelhamento de 3 vértices e 6 arestas sobre o Bitoro	41
3.5	Emparelhamento de I_2 sobre o Toro	42
3.6	Emparelhamentos de I_4 sobre o Bitoro	43
4.1	Subgrafos suficientes para a reconstrução abstrata - Figura reproduzida de [41]	53
4.2	Grafo com 4 arestas, 3 vértices e seu grafo paralelo em linhas pontilhadas.	56
4.3	Algoritmo da <i>Chuva de Vértices</i> para $\mathcal{W}_0(\mathcal{G})$	57
4.4	Grafo abstrato 3-regular	58
4.5	Dois mergulhos diferentes do mesmo grafo 3-regular	58
4.6	Visão local: padrão na palavra com vértice de grau 2	60
4.7	Visão local: padrão na palavra com vértice de grau 3	60
4.8	Visão local: padrão na palavra com vértice de grau 4	61
4.9	Visão local: padrão na palavra com vértice de grau ζ	61
4.10	Visão local do mergulho $\mathcal{G}$, com faces de tamanho δ adjacentes . . .	65

5.1	Representações contraídas gerais.	70
5.2	Representação contraída de uma árvore.	70
5.3	Multiconjunto $\mathcal{D}' _{(1)}(\mathcal{T})$	71
5.4	Multiconjunto $\mathcal{D}' _{(1)}(\mathcal{T})$ na forma contraída.	71
5.5	Representação contraída de mergulhos não reconstrutíveis com um centro e três caminhos conectados a ele.	73
5.6	Elementos do deck de mergulhos não equivalentes.	73
5.7	Representação contraída de uma árvore com um centro e quatro caminhos conectados a ele.	74
5.8	Classes de permutações de tamanho 4 conectadas aos elementos do deck.	75
5.9	Representação contraída de mergulhos não reconstrutíveis com exatamente dois vértices de grau 3 e dois caminhos conectados a cada um deles.	76
6.1	Árvores analisadas para o Lema 5.1.14 e os seus decks (1-2)	88
6.2	Árvores analisadas para o Lema 5.1.14 e os seus decks (3-4)	89
6.3	Árvores analisadas para o Lema 5.1.14 e os seus decks (5-6)	90
6.4	Árvores analisadas para o Lema 5.1.14 e os seus decks (7-8)	91
6.5	Árvores analisadas para o Lema 5.1.14 e os seus decks (9-10)	92
6.6	Árvores analisadas para o Lema 5.1.14 e os seus decks (11-12)	93
6.7	Árvores analisadas para o Lema 5.1.14 e os seus decks (13-14)	94
6.8	Árvores analisadas para o Lema 5.1.14 e os seus decks (15-16)	95
6.9	Árvores analisadas para o Lema 5.1.14 e os seus decks (17-18)	96
6.10	Árvores analisadas para o Lema 5.1.14 e os seus decks (19-20)	97
6.11	Árvores analisadas para o Lema 5.1.14 e os seus decks (21-22)	98
6.12	Árvores analisadas para o Lema 5.1.14 e os seus decks (23-24)	99
6.13	Árvores analisadas para o Lema 5.1.14 e os seus decks (25-26)	100
6.14	Árvores analisadas para o Lema 5.1.14 e os seus decks (27-28)	101
6.15	Árvores analisadas para o Lema 5.1.14 e os seus decks (29-30)	102
6.16	Árvores analisadas para o Lema 5.1.14 e os seus decks (31-32)	103
6.17	Árvores analisadas para o Lema 5.1.14 e os seus decks (33-34)	104

6.18	Árvores analisadas para o Lema 5.1.14 e os seus decks (35-36)	105
6.19	Árvores analisadas para o Lema 5.1.14 e os seus decks (37-38)	106
6.20	Árvores analisadas para o Lema 5.1.14 e os seus decks (39)	107
7.1	Emparelhamentos (1–9) de I_6 sobre o 3-Toro.	133
7.2	Emparelhamentos (10–18) de I_6 sobre o 3-Toro.	134
7.3	Emparelhamentos (19–27) de I_6 sobre o 3-Toro.	135
7.4	Emparelhamentos (28–36) de I_6 sobre o 3-Toro.	136
7.5	Emparelhamentos (37–45) de I_6 sobre o 3-Toro.	137
7.6	Emparelhamentos (46–54) de I_6 sobre o 3-Toro.	138
7.7	Emparelhamentos (55–63) de I_6 sobre o 3-Toro.	139
7.8	Emparelhamentos (64–72) de I_6 sobre o 3-Toro.	140
7.9	Emparelhamentos (73–81) de I_6 sobre o 3-Toro.	141
7.10	Emparelhamento (82-82) de I_6 sobre o tritoro	142

Sumário

1	Introdução	13
2	Notação e terminologia	15
2.1	Conceitos Básicos de Topologia	16
2.2	Grafos e Complexos Simpliciais	20
2.3	Grafos de emparelhamento de arestas	22
2.4	Cirurgias verticais e horizontais de grafos.	23
2.5	Grafitos associados a mergulhos	25
3	Grafos e emparelhamentos canônicos	28
3.1	Grafos associados a emparelhamento de arestas	28
3.1.1	Grafo Paralelo e Algoritmo Chuva de Vértices	30
3.1.2	Construção da implementação	35
3.2	Emparelhamentos canônicos para o Toro e Bitoro	41
3.3	Emparelhamentos canônicos para o Tritoro	43
3.4	Emparelhamentos canônicos para o 4 – <i>Toro</i>	44
4	Reconstrução topológica de mergulhos	47
4.1	Reconstrução de grafos sobre superfícies	47
4.1.1	Reconstrução abstrata	48
4.1.2	Reconstrução topológica	50
4.1.3	Propriedades topológicas para a reconstrução abstrata	51
4.2	Generalização da palavra associada a um mergulho: relação entre Grafito e o grafo paralelo	54

4.3	Reconstrução de mergulhos celulares	57
4.3.1	O problema de reconstrução de mergulhos	57
4.3.2	Reconstrução de grafos de emparelhamento	61
4.3.3	Reconstrução de mergulhos com restrições sobre o tamanho da face	63
5	Reconstrução de mergulhos de árvores	68
5.1	Reconstrução de mergulhos de árvores a partir das suas subárvores maximais	68
	Referências	83
6	Apêndice A	87
6.1	Verificação exaustiva: reconstrução de árvores mergulhadas	87
7	Apêndice B	121
7.1	Blocos de código	121
7.2	Mergulhos do I_6 sobre o 3-Toro	133
7.3	Mergulhos do I_8 sobre o 4-Toro	142

Capítulo 1

Introdução

O presente trabalho tem como objetivo, numa primeira parte, apresentar resultados sobre grafos associados a emparelhamentos de arestas em superfícies fechadas e orientadas de gênero g , denotadas por M_g . Na sequência, abordamos a reconstrução topológica de mergulhos nessas mesmas superfícies. Os mergulhos considerados são todos celulares, isto é, se $\mathcal{G}$ é um mergulho celular de G em M_g , então o complemento $M_g \setminus \mathcal{G}$ é um multiconjunto de regiões simplesmente conexas (homeomorfas a discos).

No Capítulo 2, apresentamos as notações e os conceitos básicos necessários para a compreensão dos resultados posteriores.

No Capítulo 3, implementamos um algoritmo que gera explicitamente todos os mergulhos canônicos não equivalentes em superfícies de gêneros 1, 2, 3 e 4. Demonstramos que, nesses casos, existem exatamente 1, 4, 82 e 7258 emparelhamentos canônicos nos toros de gêneros 1, 2, 3 e 4, respectivamente.

No Capítulo 4, tratamos da reconstrução topológica de mergulhos em superfície. Dados um mergulho celular $\mathcal{G}$ de um grafo G em M_g e seu deck

$$\mathcal{D}(\mathcal{G}) = \{\mathcal{G} - e : e \in E(\mathcal{G})\},$$

nos ocupamos de identificar condições sob as quais $\mathcal{G}$ é recuperável de modo único (a menos de homeomorfismos que preservem a orientabilidade). Mostramos, por exemplo, que todo emparelhamento obtido pela cirurgia S_1 possui essa propriedade. Além disso, se $\mathcal{G}$ apresenta duas faces de comprimento δ adjacentes que compartilham ao menos um vértice de grau 3 — sendo δ o tamanho da maior face —, então $\mathcal{G}$ também é reconstruível unicamente a partir de $\mathcal{D}(\mathcal{G})$. Além disso, no Teorema 4.3.13, fizemos a generalização para reconstrução de mergulhos, do resultado central de [40].

Por fim, no Capítulo 5, dedicamo-nos à reconstrução topológica de mergu-

lhos de árvores em M_g . Essa parte divide-se em dois casos:

- **Árvores com no máximo 4 folhas:** mostramos que, à exceção de um número finito de exemplos, todos os mergulhos são reconstrutíveis.
- **Árvores com pelo menos 5 folhas:** provamos que, sempre que a árvore não possui vértices de grau 1 acoplados ao seu centro, o mergulho pode ser recuperado unicamente a partir de seu deck.

Assim, este trabalho oferece tanto uma contagem explícita de emparelhamentos canônicos em baixos gêneros, com uma implementação expansível para outros gêneros, quanto critérios gerais de reconstrução topológica para mergulhos de grafos e de árvores.

Capítulo 2

Notação e terminologia

Neste capítulo, apresentamos uma revisão abrangente dos principais conceitos de topologia diferencial, topologia algébrica e topologia combinatória que serão fundamentais ao longo deste trabalho.

Iniciamos com uma abordagem da topologia diferencial, estabelecendo as definições de variedades topológicas, pontos regulares, imersões e mergulhos. Essa base permite a formalização de superfícies orientadas e a caracterização dos mergulhos de grafos sobre essas superfícies.

Na sequência, são apresentados os principais conceitos da topologia algébrica que serão utilizados, incluindo dominação, equivalência homotópica, característica de Euler e classificação de superfícies via gênero. Esses tópicos são fundamentais para o entendimento da estrutura global dos mergulhos e de suas propriedades topológicas invariantes.

A seguir, introduzimos os grafos como 1-complexos e os associamos a complexos celulares, o que permite a aplicação de ferramentas topológicas combinatórias. Esse ponto de vista será essencial para definirmos e estudarmos os grafos de emparelhamento de arestas, objetos centrais nesta tese. Exploramos as propriedades desses grafos e discutimos métodos para sua obtenção por meio de extensões, contrações e cirurgias topológicas, com base em resultados recentes da literatura.

Por fim, apresentamos o formalismo dos *grafitos*, que estabelece uma correspondência combinatória entre mergulhos de grafos e palavras em superfícies. Essa abordagem será central no desenvolvimento do algoritmo de reconstrução topológica que propomos nos capítulos posteriores. Os resultados clássicos de Hoffman são revisitados, servindo como estrutura para a generalização de técnicas já conhecidas no caso de grafos de emparelhamento.

Com essa fundamentação, o capítulo prepara o terreno teórico necessário

para a formulação e análise dos problemas de reconstrução topológica que serão tratados adiante.

2.1 Conceitos Básicos de Topologia

Dois espaços topológicos que possuem exatamente a mesma estrutura (no sentido topológico) são relacionados por homeomorfismos, conforme a Definição 2.1.1.

Definição 2.1.1 (Homeomorfismo, [30]). Uma função $f : X \rightarrow Y$ é um homeomorfismo se é uma bijeção contínua com inversa também contínua. Neste caso, dizemos que X e Y são homeomorfos.

Em muitas situações, é útil considerar funções que são homeomorfismos apenas localmente. A Definição 2.1.2 captura esse conceito.

Definição 2.1.2 (Homeomorfismo Local, [30]). $f : X \rightarrow Y$ é um homeomorfismo local se, para todo $x \in X$, existe um aberto $U \ni x$ tal que $f(U)$ é aberto em Y e $f : U \rightarrow f(U)$ é um homeomorfismo.

Definição 2.1.3 (Espaço Localmente Euclidiano, [30]). Um espaço topológico X é localmente euclidiano se para cada ponto $x \in X$ existe um aberto U contendo x tal que U é homeomorfo a um aberto de $\mathbb{R}^n$.

Prosseguindo, apresentamos a noção de espaços dominados, frequentemente utilizada em topologia algébrica e teoria de homotopia. Veja a Definição 2.1.4.

Definição 2.1.4 (Espaço dominado, [17], p.528). Um espaço Y é dominado por um espaço X se existem aplicações $I : Y \rightarrow X$ e $R : X \rightarrow Y$, tais que $R \circ I(y) = y$ para todo $y \in Y$.

Definição 2.1.5 (Equivalência homotópica, [17], p.3). Dois espaços X e Y são ditos homotopicamente equivalentes se existem aplicações $F : X \rightarrow Y$ e $G : Y \rightarrow X$, tais que $F \circ G \sim \text{id}_Y$ e $G \circ F \sim \text{id}_X$.

A noção de homotopia entre funções contínuas é central na topologia algébrica, pois permite identificar quando duas aplicações podem ser deformadas uma na outra de forma contínua. Formalizamos esse conceito na Definição 2.1.6.

Definição 2.1.6 (Homotopia entre funções, [17], p.2). Sejam $f, g : X \rightarrow Y$ funções contínuas entre espaços topológicos. Dizemos que f é homotópica a g , e escrevemos $f \simeq g$, se existe uma função contínua $H : X \times [0, 1] \rightarrow Y$ tal que:

$$H(x, 0) = f(x) \quad \text{e} \quad H(x, 1) = g(x), \quad \text{para todo } x \in X.$$

Tal função H é chamada de uma homotopia de f para g .

Uma aplicação importante da homotopia ocorre quando se deseja deformar todo um espaço topológico sobre um subespaço, preservando as posições relativas dos pontos desse subespaço. Esse processo é conhecido como retração por deformação, como apresentado a seguir.

Definição 2.1.7 (Retração por deformação, [17], p.4). Seja $A \subseteq X$ um subespaço. Dizemos que A é uma retração por deformação de X se existe uma homotopia $H : X \times [0, 1] \rightarrow X$ tal que:

- (i) $H(x, 0) = x$ para todo $x \in X$;
- (ii) $H(x, 1) \in A$ para todo $x \in X$;
- (iii) $H(a, t) = a$ para todo $a \in A$ e $t \in [0, 1]$.

No contexto do grupo fundamental, um conceito recorrente é o de curva nula-homotópica. Tal curva representa, em certo sentido, uma trajetória que pode ser "encolhida" continuamente até um ponto base, dentro do próprio espaço. A definição formal é dada a seguir.

Definição 2.1.8 (Curva nula-homotópica, [17], p.26). Seja $\gamma : [0, 1] \rightarrow X$ uma curva fechada baseada em x_0 , isto é, $\gamma(0) = \gamma(1) = x_0$. Dizemos que γ é *nula-homotópica* (ou *null-homotopic*) se ela é homotópica, com base fixa, à curva constante em x_0 . Isso significa que existe uma homotopia $H : [0, 1] \times [0, 1] \rightarrow X$ tal que:

- (i) $H(s, 0) = \gamma(s)$ para todo $s \in [0, 1]$;
- (ii) $H(s, 1) = x_0$ para todo $s \in [0, 1]$;
- (iii) $H(0, t) = H(1, t) = x_0$ para todo $t \in [0, 1]$.

Na topologia diferencial, estudamos aplicações suaves entre variedades.

Definição 2.1.9 (n-Variedade, [26] p.3). O conjunto X é dito uma variedade topológica de dimensão n (ou uma n -variedade) se é um espaço de Hausdorff, com base enumerável, e localmente euclidiano, isto é, localmente homeomorfo a $\mathbb{R}^n$.

Dentre as variedades topológicas de dimensão dois, temos um caso especial que é de particular interesse: as superfícies. A definição a seguir formaliza esse conceito. Veja a Definição 2.1.10.

Definição 2.1.10 (Superfície, [26]). Uma superfície é uma variedade conexa bidimensional (ou seja, de dimensão 2). A superfície é dita orientável se todo caminho fechado preserva a orientação. Caso exista algum caminho fechado que inverta a orientação, a superfície é dita não orientável.

Definição 2.1.11 (Ponto regular, [26]). Seja $f : M \rightarrow N$ uma aplicação diferenciável, onde M e N são variedades. Um ponto $p \in M$ é dito ponto regular de f quando a derivada $f'(p) : T_p M \rightarrow T_{f(p)} N$ é injetiva. Caso contrário, p é um ponto singular.

Observação 2.1.12.

A seguir, introduzimos o conceito de imersão, que se baseia na regularidade de todos os pontos da aplicação.

Definição 2.1.13 (Imersão, [26]). Uma aplicação diferenciável entre variedades $f : M \rightarrow N$ é dita uma imersão se todo ponto $p \in M$ é um ponto regular de f .

Por fim, temos a Definição 2.1.14, que caracteriza os mergulhos como imersões que são também homeomorfismos sobre a imagem.

Definição 2.1.14 (Mergulho, [26]). Seja U_0 um subconjunto aberto de $\mathbb{R}^m$. Uma imersão de classe C^k , $\psi : U_0 \rightarrow \mathbb{R}^m$, diz-se um mergulho de classe C^k de U_0 em $\mathbb{R}^m$, quando ψ é um homeomorfismo de U_0 sobre $\psi(U_0)$.

Agora que já definimos superfícies, apresentamos a noção de célula, que será a base para a construção de complexos celulares. Veja a Definição 2.1.15.

Definição 2.1.15 (n-Célula, [22]). Uma n -célula é um conjunto cujo interior é homeomorfo ao disco $D^n = \{x \in \mathbb{R}^n; \|x\| < 1\}$, com a propriedade adicional de que seu bordo pode ser subdividido em um número finito de células de dimensão inferior, chamadas faces da n -célula. Escrevemos $\sigma < \tau$ se σ é face de τ .

A estrutura composta por essas células, respeitando certas condições de regularidade e interseção, é chamada de complexo celular, como estabelece a Definição 2.1.16.

Definição 2.1.16 (Complexo celular, [22]). Um complexo celular K é um conjunto finito de células,

$$K = \bigcup \{\sigma : \sigma \text{ é uma célula}\},$$

tal que:

1. Se σ é uma célula de K , então todas as faces de σ pertencem a K .
2. Se σ e τ são células de K , então $\text{Int}(\sigma) \cap \text{Int}(\tau) = \emptyset$.

A dimensão de K é a dimensão da célula de maior dimensão em K .

Subconjuntos desses complexos que também formam complexos celulares são denominados subcomplexos, conforme a Definição 2.1.17.

Definição 2.1.17 (Subcomplexo celular, [17] p.520). Um subcomplexo de um complexo celular X é um subespaço $A \subset X$ que é união de células de X , tal que o fecho de cada célula de A está contido em X .

Com os conceitos anteriores, podemos definir a característica de Euler de um complexo celular, uma invariante topológica essencial. Veja a Definição 2.1.18.

Definição 2.1.18 (Característica de Euler, [22]). A característica de Euler de um n -complexo K , denotada por $\chi(K)$, é dada pela soma alternada:

$$\chi(K) = \#(0\text{-célula}) - \#(1\text{-célula}) + \#(2\text{-célula}) - \cdots + (-1)^n \#(n\text{-célula}),$$

onde $\#(r\text{-célula})$ representa o número de células de dimensão r em K . Para 2-complexos, é comum escrever:

$$\chi(K) = V - A + F,$$

onde V , A e F denotam respectivamente o número de vértices, arestas e faces.

Por fim, relacionamos essa característica ao conceito de gênero de uma superfície, como indicado na Definição 2.1.19.

Definição 2.1.19 (Gênero, [27]). Seja S uma superfície compacta. O gênero g de S se relaciona com sua característica de Euler $\chi(S)$ da seguinte forma:

$$\chi(S) = \begin{cases} 2 - 2g, & \text{se } S \text{ é orientável;} \\ 2 - g, & \text{caso contrário.} \end{cases}$$

2.2 Grafos e Complexos Simpliciais

Nesta seção, introduzimos os conceitos fundamentais de grafos e sua relação com complexos simpliciais. Começamos com a definição de um grafo.

Definição 2.2.1 (Grafo). Um grafo $G = (V, E)$ é composto por um conjunto finito de vértices V e um conjunto de arestas E , onde cada aresta é um subconjunto com dois elementos de V .

Podemos restringir o tipo de grafos considerados, como descrito a seguir.

Definição 2.2.2 (Grafo simples). Um grafo é dito simples se não possui laços (arestas que conectam um vértice a si mesmo) nem múltiplas arestas entre os mesmos dois vértices.

Definição 2.2.3 (Subgrafo). Um subgrafo de um grafo $G = (V, E)$ é um grafo $G' = (V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$, onde toda aresta em E' conecta apenas vértices de V' .

Definição 2.2.4 (Grafo conexo). Um grafo é dito conexo se, para todo par de vértices $u, v \in V$, existe um caminho em G que liga u a v .

Definição 2.2.5 (Componente conexa). Uma componente conexa de um grafo é um subgrafo conexo maximal, ou seja, um subgrafo conexo que não está contido propriamente em nenhum outro subgrafo conexo.

Definição 2.2.6 (K -conexidade, [38]). Seja $G = (V, E)$ um grafo e $S \subset E$ tal que $|S| = K$. O grafo G é dito K -conexo se $G - S$ tem mais de uma componente conexa e K é o menor inteiro com essa propriedade.

Definição 2.2.7 (Excentricidade, [38]). Sejam G um grafo e $u, v \in V(G)$. A excentricidade de um vértice u , denotada por $e(u)$, é definida como o valor máximo de $d(u, v)$ para $v \in V(G)$, ou seja, a maior distância de u até outro vértice da árvore.

Definição 2.2.8 (Grau de vértice e Valência mínima de grafo, [38]). Dados um grafo G e $u \in V(G)$, o grau de u , denotado por $\deg_G(u)$ ou simplesmente $\deg(u)$, quando não houver confusão, é o número de arestas incidentes à u . Diremos que G tem valência mínima ζ quando o menor grau dentre os vértices de G for ζ .

Definição 2.2.9 (Caminho). Um caminho em um grafo é uma sequência de vértices $v_0, v_1, \dots, v_k$ tal que cada par consecutivo $\{v_{i-1}, v_i\}$ pertence a E .

Definição 2.2.10 (Ciclo). Um ciclo é um caminho fechado de comprimento pelo menos 3, no qual os vértices (exceto o primeiro e o último, que coincidem) são distintos.

Definição 2.2.11 (Árvore). Uma árvore é um grafo conexo que não possui ciclos.

Definição 2.2.12 (Floresta). Uma floresta é um grafo acíclico, ou seja, um grafo que não possui ciclos. Equivalentemente, é uma união disjunta de árvores.

Na modelagem topológica, muitas vezes consideramos grafos como complexos simpliciais de dimensão 1, como ilustrado a seguir.

Definição 2.2.13 (Complexo celular de um grafo). Um grafo $G = (V, E)$ pode ser considerado como um complexo celular K de dimensão 1, onde:

- Os vértices de G são os 0-células;
- As arestas de G são os 1-células.

Com essa interpretação, é possível aplicar técnicas de topologia algébrica para estudar propriedades de grafos, como conexão, ciclos e homologia.

Definição 2.2.14 (Mergulho celular de um grafo em uma superfície). Seja G um grafo finito, considerado como um complexo celular 1-dimensional, cujas 0-células são os vértices e cujas 1-células são as arestas. Um *mergulho celular* de G em uma superfície compacta orientável M é uma aplicação contínua injetiva

$$\mathcal{G} : G \hookrightarrow M,$$

tal que:

- (i) Cada célula k -dimensional de G é mapeada homeomorficamente sobre sua imagem em M (isto é, o mergulho é compatível com a estrutura celular de G);
- (ii) A imagem $\mathcal{G}(G)$ é um subcomplexo 1-dimensional de um complexo celular 2-dimensional K cuja realização é homeomorfa à superfície M ;
- (iii) As componentes conexas da complementar $M \setminus \mathcal{G}(G)$ são abertas e homeomorfas a discos abertos, ou seja, são 2-células do complexo celular K .

Neste caso, dizemos que $\mathcal{G}(G)$ é um subcomplexo 1-dimensional de um *complexo celular 2-dimensional* que representa a superfície M , e que o mergulho é *2-celular*.

Observação 2.2.15. De agora em diante, sempre que falarmos de homeomorfismos, neste trabalho, estaremos considerando homeomorfismos que preservam a orientação.

2.3 Grafos de emparelhamento de arestas

Sabendo que o objeto de estudo do presente trabalho são os **mergulhos celulares de grafos** Definição 2.2.14 sobre superfície, primeiramente, vamos apresentar uma classe especial de mergulhos, à saber os grafos de emparelhamento de arestas. Esses grafos são mergulhos celulares aos quais o complemento do grafo à superfície consiste de apenas uma componente e esta é simplesmente conexa.

Definição 2.3.1 (Grafo de emparelhamento, [19] p.452). Um grafo $\mathcal{G}$, mergulhado numa superfície M_g , é dito **grafo de emparelhamento** de tal superfície, se o complemento $M_g \setminus \mathcal{G}$ é simplesmente conexo, ou seja, homeomorfo a um disco.

Exemplo 2.3.2. A Figura 2.1 ilustra a representação local de um grafo de emparelhamento. Cujo vértice em questão tem grau 4.

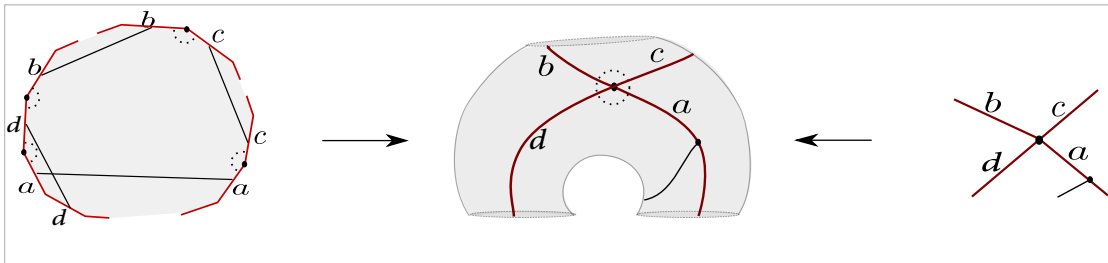


Figura 2.1: Representação local de grafo de emparelhamento sobre uma superfície

Exemplo 2.3.3. Note que um grafo pode ser um grafo de emparelhamento para uma dada superfície e não ser para uma outra superfície. A Figura 2.3 mostra, na esquerda, um grafo cujo o mergulho no Toro é composto por três regiões simplesmente conexas, enquanto o mesmo grafo, na direita, mergulhado no Bitoro, tem apenas uma componente simplesmente conexa, ou seja, é um grafo de emparelhamento para o Bitoro.

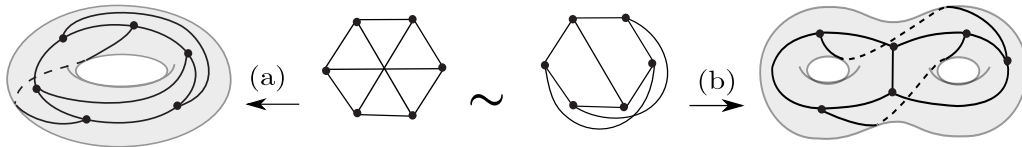


Figura 2.2: Grafo mergulhado sobre o toro e Bitoro

A seguir serão descritas e demonstradas algumas propriedades dos grafos de emparelhamento, apresentadas em [7] e as demonstrações foram retiradas de [35].

Proposição 2.3.4 ([7] p.5). Se $\mathcal{G}$ é um grafo de emparelhamento sobre M_g , então $V - A = 1 - 2g$.

Demonstração. Se $\mathcal{G}$ é um grafo de emparelhamento sobre M_g , o complemento de $\mathcal{G}$ sobre M_g é simplesmente conexo, portanto $F = 1$. Dessa forma:

$$A - V = 2g - 2 + F = 2g - 1.$$

□

Corolário 2.3.5 ([7] p.6). *O grafo de emparelhamento $\mathcal{G}$ é uma árvore se, e somente se, $M_g = \mathbb{S}^2$.*

Proposição 2.3.6 ([7] p.6). *Seja $\mathcal{G}$ um emparelhamento sobre a superfície M_g . Se $\mathcal{G}_1$ é uma extensão ou uma contração sobre M_g , do grafo $\mathcal{G}$ sobre M_g , então $\mathcal{G}_1$ é um grafo de emparelhamento de M_g .*

Em [35] o autor mostrou a relação das palavras (Definido na Seção 3.1.1) com a extensão de vértices do grafo mergulhado. No trabalho em questão, foi mostrado que é possível aplicar a extensão apenas na palavra associada ao grafo de emparelhamento de arestas e obter outro grafo de emparelhamento de arestas. Além disso, em [8] os autores mostraram que essa palavra, associada ao grafo de emparelhamento carrega informações sobre os vértices e essas podem ser decodificadas usando o algoritmo chamado *Chuva de vértices*. Todas as definições e resultados associados às palavras e ao algoritmo serão trazidos e tratados com mais detalhes na Seção 3.1.1.

Na próxima Seção vamos falar sobre uma outra técnica para obter novos grafos de emparelhamento, introduzida por [9] *Faria, Mendes de Jesus e Romero Sánchez*, as cirurgias. Mergulhos obtidos a partir dessas técnicas formarão uma classe especial no Capítulo 4, onde tratamos da reconstrução topológica de mergulhos.

2.4 Cirurgias verticais e horizontais de grafos.

Em [9] *Faria, Mendes de Jesus e Romero Sánchez*, trouxeram um grande avanço, onde foram definidas as cirurgias verticais e horizontais de grafos. Esse avanço se dá no sentido de possibilitar encontrar outros grafos de emparelhamento de arestas a partir de grafos de emparelhamento já identificados. Vale observar que nem todos os grafos de emparelhamento são determinados pelas cirurgias definidas.

Para declarar as cirurgias definidas pelos autores, primeiro precisamos falar sobre a soma conexa de superfícies. A soma conexa de duas (ou mais) superfícies consiste em identificar de forma biunívoca (no caso de superfícies orientáveis) os bordos das superfícies em questão. No caso em que estamos trabalhando, com superfícies fechadas e orientadas, isso é, sem bordo também é possível fazer a soma

conexa dessas superfícies, mas para isso devemos criar os bordos - pelo menos temporariamente - que serão colados.

Para fazermos a operação de soma conexa para superfícies orientadas, devemos nos atentar em dois pontos. O primeiro e já citado é a relação biunívoca da identificação dos bordos, para manter a orientabilidade da superfície resultante (necessária em nosso caso) e a coerência da característica de Euler, isso é, a característica de Euler da superfície resultante deve ser igual a soma das características das superfícies usadas para fazer a soma conexa. Portanto, para fazer a soma conexa de duas superfícies, devemos retirar um disco de cada uma delas e colar os bordos gerados na superfície usando um cilindro (esfera S^2 menos dois discos $\mathbb{D}^2$). Note que, todos os elementos retirados ou colocados na operação tem característica de Euler nula, portanto eles fazem manter a coerência da operação como citado.

Agora para falarmos da cirurgia S_1 e S_2 definida pelos autores, basta considerarmos que sobre as superfícies a serem somadas, agora temos grafos mergulhados e os discos a serem retirados são tomadas em uma região da superfície onde tem uma única aresta do grafo, mas não tem vértices. O que vai diferenciar a cirurgia S_1 da cirurgia S_2 é o grafo que vamos mergulhar na esfera S^2 antes de retirar os dois discos. Para S_1 vamos mergulhar o grafo com três arestas, dois loops e separável pela deleção de uma aresta, indicado na Figura 2.3 (a), chamado pelos autores de H_1 . Enquanto para S_2 vamos usar o grafo com três arestas, dois vértices e dois ciclos de tamanho 2, indicado na Figura 2.3 (b), chamado pelos autores de H_2 .

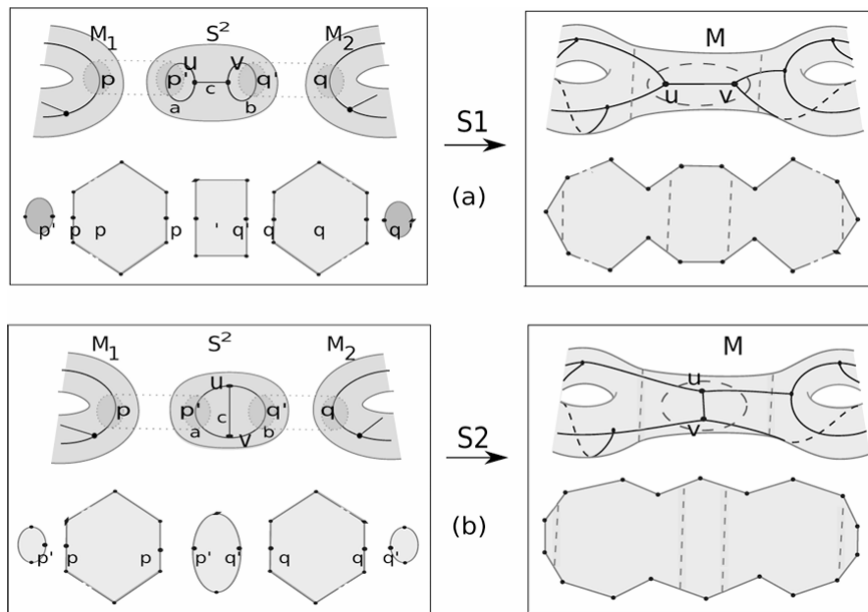


Figura 2.3: Grafo mergulhado sobre o toro e Bitoro - Imagem retirada de [9]

2.5 Grafitos associados a mergulhos

Na Seção 3.1.1, veremos uma técnica definida por *H.C. Silva* em [35] para associar um mergulho de um grafo a uma palavra. No entanto, na seção referenciada, as definições serão feitas sob uma restrição, à saber, o complemento do mergulho do grafo é homeomorfo a um disco. Mergulhos com essa restrição serão também abordados no presente trabalho, porém antes disso, nessa seção vamos apresentar os resultados gerais como em [18], fazendo as devidas referências os resultados mostrados. No contexto em que o complemento do grafo é homeomorfo a um disco, o autor chamou o código associado ao mergulho de "palavra", vamos continuar usando essa nomenclatura para esse contexto. Mas para o contexto geral, vamos adotar a nomenclatura "*Grafito*", que será definido nessa seção. Essa seção é importante, pois os conceitos inseridos por *Hoffman* em [18] serão muito relevantes para generalizarmos, para mergulhos celulares, o algoritmo *Chuva de vértices*, que será visto na Seção 3.1.1 e com isso poder usar as propriedades do *grafito* na reconstrução topológica de mergulhos. Portanto, o objetivo da presente seção é simplesmente expor os resultados introduzidos por *Hoffman*.

Esta seção é baseada no *Journal of Combinatorial Theory, Series B* 36, 65-84 (1984): *Embedding graphs on surfaces* [18].

Teorema 2.5.1 ([18] - Scissors Theorem). *Seja K um grafo contido em uma superfície M . Se K não possui vértices isolados, então existe uma superfície com borda B' e uma aplicação $\theta : B' \rightarrow M$ tal que:*

1. θ é sobrejetora e U é aberto em M se, e somente se, $\theta^{-1}(U)$ é aberto em B' .
2. θ mapeia $B' - \partial B'$ homeomorficamente em $M - K$.
3. θ mapeia $\partial B'$ em K e, para todo $b \in \partial B'$ tal que $val(\theta(b)) > 1$, existe uma vizinhança N de b em $\partial B'$ tal que θ mergulha N em M .
4. Para cada x em K , $|\theta^{-1}(x)| = val(x)$.

A seguir, o autor não impõe restrições quanto à conectividade de grafos ou superfícies, nem quanto à orientabilidade ou à topologia das faces do grafo mergulhado. Apenas por conveniência, o autor assume que o grafo não possui vértices isolados e que cada componente da superfície contém pelo menos uma componente do grafo, no caso da superfície ser desconexa.

Definição 2.5.2 ([18]). Um *Grafito* é um conjunto de palavras escritas em uma superfície M (sem borda e possivelmente desconectada), tal que:

- (i). Em cada componente de M , há pelo menos uma palavra.
- (ii). Contando inversas também, cada letra ocorre exatamente duas vezes no total.
- (iii). Para nenhum par de símbolos distintos a, b (incluindo suas inversas), é possível substituir cada ocorrência de ab por uma letra c (não previamente utilizada) e cada ocorrência de $b^{-1}a^{-1}$ por c^{-1} , eliminando assim todas as ocorrências de a e b .

Definição 2.5.3 ([18]). Um isomorfismo entre dois grafitos $G_1 \rightarrow G_2$ é uma correspondência biunívoca $\theta : L_1 \rightarrow L_2$, onde L_i é o conjunto de letras e suas inversas de G_i , tal que:

- (i) θ comuta com a inversão.
- (ii) θ determina uma correspondência biunívoca entre palavras, no sentido de que, para cada palavra $a_1a_2 \dots a_k$ de G_1 , $\theta(a_1)\theta(a_2) \dots \theta(a_k)$, quando ciclicamente permutada e possivelmente invertida, torna-se uma palavra de G_2 .
- (iii) As componentes das superfícies S_1 e S_2 associadas a G_1 e G_2 , respectivamente, devem corresponder homeomorficamente.
- (iv) Para cada par de componentes correspondentes em (iii) que sejam orientáveis e possuam mais de uma palavra inscrita, ou todas essas palavras devem ser invertidas ou nenhuma.

Teorema 2.5.4 ([18]). *Existe uma correspondência biunívoca entre classes de isomorfismo de grafitos e classes de equivalência de mergulhos de grafos.*

O Scissors Theorem é usado para provar o Teorema 2.5.4.

Definição 2.5.5 ([18]). Dados dois Grafitos com o mesmo alfabeto, um isomorfismo estrito $G \rightarrow G'$ é uma correspondência biunívoca entre conjuntos de palavras tal que as condições (ii), (iii) e (iv) (na Definição 2.5.3) são satisfeitas, onde θ deve mapear cada letra para si mesma ou para sua inversa.

Corolário 2.5.6 ([18]). *Dado um grafo K , escolha letras para nomear suas arestas direcionadas, com inversas correspondendo a direções opostas. Então, as classes de equivalência estrita de mergulhos de K em superfícies estão em correspondência biunívoca com as classes de isomorfismo estrito de grafitos cujo alfabeto são as arestas direcionadas de K , que satisfazem:*

- (A) Se $l_1l_2 \dots l_n$ é uma palavra com $n > 1$, então o vértice final de l_i é o inicial de l_{i+1} .

(B) Define-se $l_i^{-1} \longleftrightarrow l_{i+1}$, criando uma ordenação cíclica das arestas direcionadas a partir de qualquer vértice.

Proposição 2.5.7 ([18]). Mergulhos em superfícies orientáveis correspondem a grafitos onde todas as componentes são orientáveis e as palavras podem ser escolhidas de modo que cada letra e sua inversa ocorram exatamente uma vez.

Proposição 2.5.8 ([18]). Mergulhos celulares correspondem a grafitos cujas componentes são esferas e exatamente uma palavra é escrita em cada esfera.

Corolário 2.5.9 ([18]). Classes de equivalência estrita de mergulhos celulares de um grafo K estão em correspondência biunívoca com conjuntos de palavras nas arestas direcionadas de K , definidos até permutação cíclica e inversão, e que satisfazem (A), (B) (do Corolário 2.5.6), e:

(C) Cada aresta não direcionada ocorre exatamente duas vezes entre as palavras.

Corolário 2.5.10. [[18]] Classes de equivalência de mergulhos celulares de todos os grafos em todas as superfícies estão em correspondência biunívoca com classes de isomorfismo de conjuntos de palavras de algum alfabeto infinito de geradores, satisfazendo:

(A₁) As palavras são "palavras cíclicas" do grupo livre sobre o alfabeto (ou seja, definidas até permutação cíclica e inversão), exceto que inversas adjacentes não são canceladas.

(B₁) Um gerador ocorre exatamente duas vezes ou não ocorre (contando também suas inversas).

(C₁) Para nenhum par distinto a, b do alfabeto e suas inversas é possível substituir cada ocorrência de ab por um gerador c (não previamente utilizado) e cada ocorrência de $b^{-1}a^{-1}$ por c^{-1} , eliminando assim todas as ocorrências de a e b .

Proposição 2.5.11 ([18]). Seja $K \hookrightarrow M$ correspondente a um grafito, com K um grafo mergulhado em uma superfície M .

(a) K é conexo se, e somente se, nenhum subconjunto próprio de palavras satisfaz a condição (B₁) do Corolário 2.5.10.

(b) M é conexa se, e somente se, a condição (B₁) do Corolário 2.5.10 nunca é satisfeita por um subconjunto próprio de componentes.

Capítulo 3

Grafos e emparelhamentos canônicos

O objetivo desse capítulo é fazer uma implementação do algoritmo proposto por *H. C. Silva, et al* em [8] e explicitar todos os mergulhos canônicos, não equivalentes, possíveis para superfícies fechadas e orientadas de gênero 1, 2, 3 e 4. Fazendo isso, baseado no Teorema 3.1.1 de *C. M. de Jesus, et al* [7], estaremos fornecendo todos os grafos de emparelhamentos de arestas para essas superfícies, a menos de extensão de vértices.

3.1 Grafos associados a emparelhamento de arestas

Um mergulho $\mathcal{G} : G \rightarrow M_g$, de um grafo G com A arestas, em uma superfície fechada e orientada de gênero g , M_g , é dito grafo de **emparelhamento de arestas** sobre M_g ([7]) se o seu complemento $M_g \setminus \mathcal{G}$ é simplesmente conexa, isto é, homeomorfa a um disco. Vários trabalhos e técnicas foram desenvolvidas nesta direção:

- Dado um mergulho, descobrir se esse é um grafo de emparelhamento de arestas [8,10,19]
- Dado um grafo, descobrir se ele pode ser mergulhado em alguma superfície fechada e orientada de gênero g , de modo que o seu complemento seja homeomorfo a um disco. [8,32]
- Dados dois (ou mais) grafos de emparelhamento de arestas, descobrir se é possível fazer uma operação entre eles a fim de obter outro grafo de emparelhamento. [7,9]

À seguir vamos revisar alguns desses trabalhos e suas contribuições.

Em [19] Jorgensen-Naatanen fizeram um estudo para o caso de grafos regula-

res de valência 3 (**trivalentes**), sobre o Bitoro, mostrando que, a menos de reflexão, existem 8 formas diferentes de emparelhamento de P_{18} , que foram generalizado em [10] por Faria e Palazzo. Vale observar que esses grafos de emparelhamento trivalentes, estão associados com a construção de códigos corretores ótimos, ou seja códigos cuja probabilidade de erro é mínima [6], via empacotamento de esferas com densidade máxima [2, 10]. E as tesselações mais gerais, tem aplicações no sentido de códigos corretores de erros quânticos topológicos [1].

Já o caso dos grafos de emparelhamentos trivalentes sobre o Tritoro foi tratado em [29] e [32], usando métodos que envolvem o estudo dos sistemas de rotação dos mergulhos e dos grupos Fuchianos. Nakamura, em [32], mostrou que existem 927 padrões de emparelhamento dos grafos trivalentes para o Tritoro.

Até esse momento, não era conhecida uma técnica que fornecesse grafos de emparelhamento. Até então a busca era por padrões de emparelhamento, sempre partindo "do zero" em uma busca, quando se terminava outra. Na contramão do que se vinha fazendo, C. M. de Jesus *at all*, em [9], introduziram dois tipos de cirurgias (operações com os grafos sobre superfícies) para determinar novas famílias de grafos de emparelhamentos trivalentes associados a superfície com gênero $g = \sum_{i=1}^m g_i$, a partir de conjuntos de m emparelhamentos já conhecidos, associados à superfícies com gênero g_i .

Posteriormente, C. M. de Jesus *at all* introduziu ao contexto dos grafos de emparelhamento de arestas, os conceitos já existentes de extensão e contração de grafos. Essa técnica permitiu a construção de grafos de emparelhamento de arestas a partir de outros grafos de emparelhamentos, aplicando essas operações sobre a superfície. Em seu celebre trabalho [7], onde foram introduzidas essas operações no contexto citado, sabendo que o **emparelhamento canônico** I_{2g} é o mergulho, em M_g do grafo com um único vértice e $2g$ loops, de modo que $M_g \setminus I_{2g}$ seja homeomorfo a um disco.

Teorema 3.1.1 ([7]). *Todo grafos de emparelhamento de arestas $\mathcal{G}$ sobre M_g , com $g > 0$, pode ser obtido por extensão do emparelhamento canônico I_{2g} .*

H. C. Silva, *at all* em [8], introduziu uma técnica para associar uma palavra ao grafo $\mathcal{G}$, com v vértices e e arestas, mergulhado em M_g , essa técnica se dá via a definição do **grafo paralelo**, um grafo com $2e$ arestas, $\sum_{u \in G} \deg(u)$ vértices, e estabelece uma ponte entre o mergulho do grafo e uma palavra. Além disso, o autor introduziu um algoritmo, chamado **chuva de vértices**, onde dada uma palavra é possível descobrir se essa palavra está associada a um grafo de emparelhamento sobre alguma superfície, pois dada a palavra, o algoritmo fornece como saída o número de vértices do mergulho associado àquela palavra e os graus de cada um desses vértices. Sabendo que o número de sílabas na palavra é exatamente o dobro do número de

arestas do mergulho, como o algoritmo nos fornece o número de vértices e queremos que o número de faces seja igual a 1, conseguimos identificar se para alguma superfície de gênero g_1 o grafo associado à palavra satisfaz $\beta_1(G) = 2g_1$, condição necessária, que se torna suficiente se a palavra tem única componente (Teorema 3.1 - [8]), para que o grafo seja de emparelhamento.

3.1.1 Grafo Paralelo e Algoritmo Chuva de Vértices

Definição 3.1.2 (Entorno Tubular de um Mergulho, [34]). Seja $\mathcal{G} : G \hookrightarrow M$ um mergulho de um grafo finito G (ou seja, um 1-complexo finito) em uma superfície topológica M . Um *entorno tubular* de $\mathcal{G}$ em M é um aberto $U \subset M$ que satisfaz as seguintes propriedades:

- (i) $\mathcal{G} \subset U$;
- (ii) existe um homeomorfismo $\varphi : U \rightarrow \mathcal{G} \times (-\varepsilon, \varepsilon)$, para algum $\varepsilon > 0$, tal que o diagrama

$$\begin{array}{ccc} U & \xrightarrow{\varphi} & \mathcal{G} \times (-\varepsilon, \varepsilon) \\ & \searrow & \downarrow \pi \\ & & \mathcal{G} \end{array}$$

comuta, onde π é a projeção natural sobre o primeiro fator. Em outras palavras, U é um feixe normal topológico trivial sobre $\mathcal{G}$.

Tal vizinhança é localmente homeomorfa a faixas ao redor das arestas e a discos ao redor dos vértices do grafo, preservando sua estrutura combinatória e a topologia da superfície.

Seja $\mathcal{G} : G \rightarrow M_g$ um mergulho. O entorno tubular de $\mathcal{G}$ em M_g será denotado por $\nu(\mathcal{G})$. Se $M_g \setminus \mathcal{G}$ tiver $\mathcal{C}(G)$ componentes conexas, então a fronteira de $\nu(\mathcal{G})$ terá $\mathcal{C}(\mathcal{G})$ componentes de curvas simples e fechadas em $M_g \setminus \mathcal{G}$.

Definição 3.1.3 ([8]). O conjunto das curvas de fronteira do entorno tubular $\nu(\mathcal{G})$ em M_g será chamado de **curvas paralelas** de $\mathcal{G}$ e será denotado por $\gamma(\mathcal{G})$

Definição 3.1.4 ([8]). O grafo orientado e rotulado construído a seguir é chamado de **Grafo Paralelo** de $\mathcal{G}$ e será denotado por $\mathcal{B}(\gamma)$.

Construção 3.1.5 ([8]). Seja $\mathcal{G}$ um grafo mergulhado em M_g , tal que $M_g \setminus \mathcal{G}$ seja homeomorfo a um disco, e denote por $\gamma(\mathcal{G})$ as curvas paralelas deste mergulho. Considere um novo grafo orientado e rotulado, construído da seguinte maneira:

1. O conjunto de arestas é definido por:

$$E(\mathcal{B}(\gamma)) = \{\text{Todos os pares de segmentos de } \gamma(\mathcal{G}) \text{ que são paralelos a uma aresta de } \mathcal{G}\}.$$

2. O conjunto de vértices é definido por:

$$V(\mathcal{B}(\gamma)) = \{v; v \text{ conecta segmentos consecutivos de } \gamma(\mathcal{G})\}.$$

3. Para atribuir uma direção às arestas de $\mathcal{B}(\gamma)$, use a seguinte regra: "a orientação natural das arestas do grafo paralelo ocorre quando as arestas de $\mathcal{G}$ estão no lado esquerdo das arestas do grafo paralelo."

4. Para rotular as arestas do grafo paralelo, basta rotular, de forma arbitrária, por $a_1 \dots a_n$, as arestas de $\mathcal{G}$, e, se duas arestas de $\mathcal{B}(\gamma)$ forem paralelas a $a_i \in E(\mathcal{G})$, então rotule-as por A_i e A_i^{-1} , sem perda de generalidade.

O seguinte exemplo não consta em [8], mas ajudará a esclarecer a construção exibida e a definição de grafo paralelo.

Exemplo 3.1.6. A Figura 3.1 mostra uma árvore mergulhada em $M_0 = \mathbb{S}^2$ em linhas sólidas, no primeiro quadrante. No segundo quadrante, está exibido em linhas pontilhadas a interseção da vizinhança tubular do mergulho com a esfera $\mathbb{S}^2$. Esse conjunto de curvas, nesse caso unitário, é chamado de curva paralela. No terceiro quadrante, está representada uma orientação natural para o conjunto de curvas paralelas, onde o grafo sempre está à esquerda da curva. No quarto quadrante, divide-se cada componente do conjunto de curvas paralelas em segmentos, na vizinhança dos vértices do grafo original e coloca-se vértices conectando segmentos consecutivos, formando assim o grafo direcionado, orientado sobre a superfície $\mathbb{S}^2$ chamado grafo paralelo.

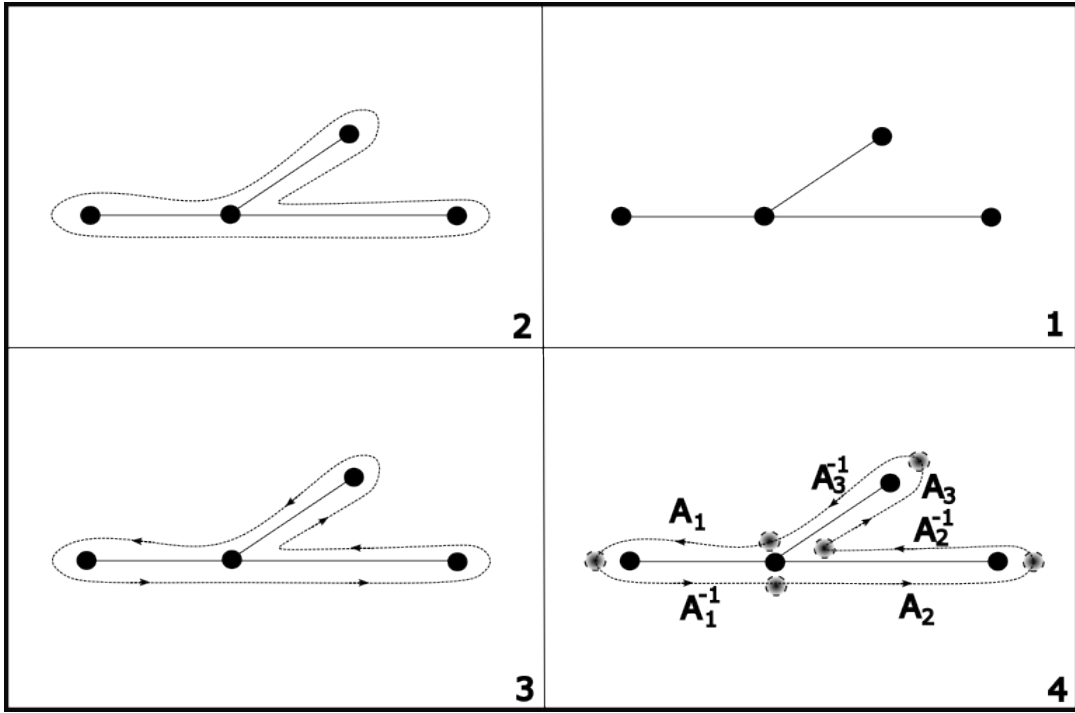


Figura 3.1: Grafo paralelo de uma árvore mergulhada em S^2

O grafo paralelo induz uma **palavra associada** ao mergulho do grafo, a partir de um percurso pelas arestas do grafo paralelo, além de apresentar as propriedades das palavras e as condições para equivalência entre palavras. Essa palavra contém informações sobre o mergulho do grafo, e cada elemento A_i (ou A_i^{-1}) é chamado de **sílabo da palavra**. Uma palavra associada a $\mathcal{G}$, mergulhada em M_g , é denotada por $\mathcal{W}_g(\mathcal{G})$. No caso do Exemplo 3.1.6, a palavra associada ao grafo é $\mathcal{W}_0(\mathcal{G}) = A_1 A_1^{-1} A_2 A_2^{-1} A_3 A_3^{-1}$.

Observação 3.1.7. De modo geral, note que, se $\mathcal{G}$ é um grafo mergulhado em M_g , tal que $M_g \setminus \mathcal{G}$ seja homeomorfo a um disco. Considere que $|E(\mathcal{G})| = n$ e $|V(\mathcal{G})| = m$. Dessa forma, por definição e construção, $\mathcal{B}(\gamma)$ possui exatamente $2n$ arestas e $\sum_{i=1}^m v_i$ vértices, onde $v_i \in V(\mathcal{G})$.

Observação 3.1.8. Se $v \in V(\mathcal{G})$ e $\deg(v) = p$, por definição e construção de $\mathcal{B}(\gamma)$, existem $U_1, \dots, U_p \in E(\gamma(\mathcal{G}))$ no entorno de v . Nesse caso, quando dizemos " U no entorno de V ", queremos dizer que " U é extremidade de uma aresta $A_i, A_j \in E(\gamma(\mathcal{G}))$, onde A_i é paralelo a $a_i \in E(\mathcal{G})$ e A_j é paralelo a $a_j \in E(\mathcal{G})$, e V é uma extremidade de a_i e a_j (i pode ser igual a j)".

Em seguida vamos explicitar quando duas palavras são equivalentes.

Propriedade 3.1.9 ([8]). Se α, β e Γ representam quaisquer blocos de uma palavra dada $\mathcal{W}_g$, ou seja, os outros termos da mesma, algumas transformações que não alteram a topologia da superfície serão apresentadas abaixo:

T1 A permutação cíclica de uma palavra não altera a topologia da superfície à qual ela está associada, ou seja, $\alpha\beta = \beta\alpha$.

T2 A remoção ou introdução de um par de arestas consecutivas $A A^{-1}$ (ou $A^{-1} A$), no polígono e, consequentemente, na palavra, não altera a topologia da superfície. Ou seja, $\alpha A A^{-1} \beta = \alpha \beta$.

T3 A inversão de todas as sílabas não altera a palavra, isto é:

$$\mathcal{W} = A_i \dots A_j A_k^{-1} \dots A_l = A_i^{-1} \dots A_j^{-1} A_k \dots A_l^{-1} = \mathcal{W}^{-1}$$

T4 A permutação de quaisquer dois blocos, entre duas arestas do tipo A e A^{-1} , não altera a topologia da superfície, ou seja, $\alpha A \beta \Gamma A^{-1} = \alpha A \Gamma \beta A^{-1}$.

T5 A permutação de termos do tipo $A A^{-1}$ não altera a topologia da superfície, ou seja, $\alpha A \beta A^{-1} \Gamma = \alpha A^{-1} \beta A \Gamma$.

Para mais detalhes sobre os itens anteriores, veja [13].

Observação 3.1.10. [8] Seja a_i uma aresta qualquer de $\mathcal{G}$, de modo que v_1 e v_2 constituam suas extremidades. Ao construir $\gamma(\mathcal{G})$, pode-se observar que as arestas A_i e A_i^{-1} de $\mathcal{B}(\gamma)$ são paralelas a a_i e que os vértices que estão no início da aresta A_i e no final da aresta A_i^{-1} (ou vice-versa) estão associados a v_1 (ou, no outro caso, a v_2). (veja Figura 3.2-(a)).

Para representar o conjunto de vértices de $\mathcal{B}(\gamma)$ associados a v_1 e v_2 no *grafito*, veja a Figura 3.2-(b).

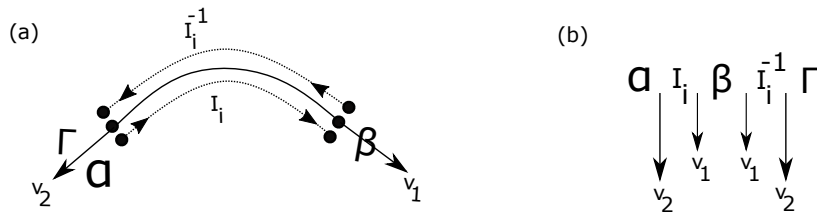


Figura 3.2: Figura retirada de [8] - (a) Disposição de vértices associados em um grafo; (b) Representação, no *grafito*, de vértices associados.

Baseado na Observação 3.1.10 e na definição, a seguir, de vértices associados, enunciaremos o algoritmo desejado. A definição de vértices associados é útil para definir o algoritmo que nos fornece a relação entre os vértices de $\mathcal{G}$ e a palavra associada pelo grafo paralelo.

Definição 3.1.11. [8] Se $U_1, U_2 \in V(\gamma(\mathcal{G}))$ estão no entorno de $V \in V(\mathcal{G})$, dizemos que U_1 e U_2 são **associados** por V , ou simplesmente, que U_1 e U_2 são vértices associados.

Algoritmo "chuva de vértices"

1. **Entrada:** a palavra (chamada de *grafito*) $\mathcal{W}_g$ e a seleção de quaisquer duas sílabas consecutivas do *grafito*, digamos A_i e A_j , respeitando a ordem.
2. **Entre as sílabas selecionadas, coloque uma seta (com comprimento de uma unidade) apontando para baixo e, no final dessa seta, coloque a notação V_i .**
(Essa notação indica que o vértice do grafo paralelo que conecta as arestas consecutivas A_i e A_j é V_i do grafo original).
3. **Coloque uma cópia da seta mencionada (com comprimento de uma unidade, apontando para baixo e com a notação V_i em sua extremidade), imediatamente antes da sílaba A_i^{-1} e imediatamente após a sílaba A_j^{-1} .** (Note que, conforme a Observação 3.1.10, o vértice do grafo paralelo que aparece imediatamente após A_k está associado ao vértice que aparece imediatamente antes de A_k^{-1} , para todo k existente. Assim, esta notação indica que os vértices antes de A_i^{-1} e após A_j^{-1} também estão associados ao vértice V_i do grafo original).
4. (Se a sílaba A_l estiver imediatamente antes da sílaba A_i^{-1} e a sílaba A_m estiver imediatamente após a sílaba A_j^{-1} , conforme o item anterior, há uma seta entre A_l e A_i^{-1} e entre A_j^{-1} e A_m , necessariamente nessa ordem).
Coloque uma cópia da seta imediatamente antes de A_l^{-1} e imediatamente após A_m^{-1} . (Novamente, estamos usando a Observação 3.1.10, e os passos 3 e 4 foram unicamente determinados pela escolha dos passos 1 e 2).
5. **Repita o procedimento até que não seja mais possível colocar, pela regra dada pela Observação 3.1.10, a seta em questão.**
6. **Coloque uma seta para baixo, com duas unidades de comprimento e notação V_2 em sua extremidade, entre duas sílabas consecutivas não rotuladas, se existirem. Digamos A_n e A_o , exatamente nessa ordem.** (Com esta notação, estamos dizendo que o vértice do grafo paralelo que conecta as arestas A_n e A_o está associado ao vértice V_2 do grafo original).
7. **Repita os passos 3, 4 e 5 para a nova seta, agora com as sílabas A_n e A_o como referência.** (Posteriormente, repita o passo 6, apenas adicionando uma unidade no tamanho da seta e também uma unidade no índice da notação da extremidade da seta).
8. **Saída:** Conjuntos de todos os vértices associados por $V_1, V_2, \dots, V_n$, respectivamente, onde n é a cardinalidade do conjunto de vértices do grafo original.

Observação 3.1.12 ([8]). A chuva de vértices é independente da escolha aleatória de duas sílabas consecutivas A_i e A_j feita no passo 1 e da escolha de colocar a notação V_1 para denotar que o vértice do grafo paralelo que conecta as arestas A_i e A_j está associado ao vértice V_1 (ou seja, está no entorno do vértice V_1). Para isso, basta notar que o vértice do grafo paralelo que conecta as arestas consecutivas A_i e A_j , por construção, está no entorno de algum vértice do grafo original. Com tais escolhas, o que estamos fazendo é fornecer um rótulo para o vértice do grafo original em questão.

Observação 3.1.13 ([8]). Uma palavra $\mathcal{W}_{\mathcal{G}}$ representa um grafo K -regular se, e somente se, o conjunto de vértices de $\mathcal{B}(\gamma)$ associados por V , de $\mathcal{G}$, tiver exatamente K elementos, para todo V em $\mathcal{G}$.

Para verificar esta afirmação, basta notar que, pela definição do grafo paralelo, o número de vértices de $\mathcal{B}(\gamma)$ que estão no entorno de um vértice V de $\mathcal{G}$ coincide com $\deg(V)$.

3.1.2 Construção da implementação

A fim de implementar o algoritmo chuva de vértices, foram adotadas duas abordagens, uma abordagem de *força bruta* e outra *segmentada*. Na abordagem de força bruta, calculamos todas as possibilidades de forma indiscriminada e mais simples possível, incluindo todas as equivalências por rotação, reindexação e reflexão válida. Na abordagem segmentada, dividimos o processo em subprocessos menores, computacionalmente falando, isto é, que ocupassem menos memória RAM.

Primeira abordagem - Força bruta

Nessa primeira abordagem, primeiramente, foi necessário construir um código *Python* que dada a quantidade de arestas do grafo original, retornasse todas as palavras possíveis, sabendo que todas as sílabas deveriam aparecer exatamente duas vezes e não deveríamos ter A_i e A_i^{-1} de forma consecutiva. Sabendo que n no código é o número de arestas do grafo original, o código feito para tal etapa pode ser visto no Bloco Python 7.1. Esse código contém uma função Python tem a seguinte disposição:

Função `generate_words(n)`

Entrada

- Um inteiro n : número de sílabas distintas $A_1, A_2, \dots, A_n$.

Saída

- Um gerador (`generator`) que produz tuplas de comprimento $2n$, cada tupla representando uma “palavra” onde cada A_i aparece exatamente duas vezes sem repetições consecutivas (incluindo entre o último e o primeiro elemento).

Uma vez geradas todas as possibilidades, precisávamos verificar as equivalências por rotação, reindexação e reflexão válida das palavras, para isso foi feita a uma função, disponível no Bloco Python 7.2. Essa função tem a seguinte descrição:

Função `canonical_full(word, n)`

Entrada

- `word`: tupla de sílabas (cada elemento A_i) que representa a palavra a ser canonicalizada.
- `n`: inteiro indicando o número de sílabas distintas $A_1, A_2, \dots, A_n$.

Saída

- Uma tupla que corresponde à forma canônica mínima de `word`, obtida ao aplicar todas as permutações de índices, rotações e reflexões possíveis.

Após a determinação das palavras possíveis e não equivalentes, podemos aplicar a chuva de vértices em cada uma delas, a fim de segmentar somente aquelas associadas a mergulhos canônicos, ou seja, com um vértice e $2g$ loops, onde g é o gênero de uma superfície fechada e orientada. A função retorna uma lista, onde cada elemento da lista é uma lista de vértices associados, no sentido da Definição 3.1.11. O código pode ser visto no Bloco Python 7.3 e tem a seguinte descrição:

Função `chuva_de_vertices(word)`

Entrada

- `word`: tupla de sílabas $\{A_1, \dots, A_n\}$ de comprimento $2n$, cada sílaba aparecendo exatamente duas vezes.

Saída

- Uma lista de `clusters`, onde cada cluster é uma lista de posições (índices de 0 a $2n - 1$) conectadas pelo algoritmo chuva de vértices.

As funções que plotam os desenhos foram omitidas do texto, mas estão disponíveis no Notebook do Google Colab [Notebook Python - Força bruta](#) (link interativo).

Visando melhorar a performance e diminuir a quantidade de palavras desnecessárias, leia desnecessárias como àquelas que seriam descartadas por rotação, fizemos um pequeno ajuste, mas que elimina diversas possibilidades ao gerar as palavras, dado um n . O ajuste foi exigir que todas as palavras geradas começassem com A_1 , isso elimina a busca em algum tipo de fator $n! - y$, para $(n - 1)! - y$, onde y é o número de palavras já excluídas pela restrição de não poder aparecer $A_i A_i^{-1}$ de forma consecutiva. Além disso, ao invés de gerar todas as palavras e depois aplicar a função que tira as duplicidades, aplicamos os dois métodos em uma mesma iteração e colocamos a parte de desenhar as saídas como opcional. Dessa forma, a versão otimizada, mas ainda em forma de *força bruta* pode ser vista no Bloco Python 7.4. Vale observar que essa função não foi adotada como padrão e sim opcional, pois apesar de melhorar a performance, ela demanda de mais poder computacional o que poderia restringir a sua replicação em outros hardwares. A descrição dessa função opcional, é a seguinte:

Função `find_single_cluster_words(n, v=1, draw=False)`

Entrada

- n : inteiro, número de sílabas distintas $A_1, \dots, A_n$ a serem usadas.
- v : inteiro (padrão 1), número de clusters desejados na palavra.
- `draw`: booleano (padrão `False`), indica se os clusters encontrados devem ser desenhados.

Saída

- Uma lista de palavras (cada palavra é uma lista de strings) que possuem exatamente v clusters, conforme identificado pelo algoritmo chuva de vértices.

A seguir veremos a abordagem segmentada. Essa abordagem foi necessária para termos resultados parciais, ao fazermos as buscas por grupamentos e para possibilitar a execução em partes que, individualmente, demandam menos poder

computacional.

Abordagem segmentada

Nesta abordagem a principal mudança feita é que ao invés de fixar simplesmente a primeira entrada como A_1 , decidimos dar a opção do usuário fixar as m primeiras entradas, sendo $m \leq n$. Dessa forma, o número de combinações diminui e com isso o tempo de execução e também o poder computacional necessário. Como em nosso caso ainda desejamos todas as combinações possíveis, o primeiro passo foi adaptar o código de geração das palavras para gerar os prefixos. Dado um número n , ainda número de arestas, a função retorna todas as sequências de tamanho q , preenchidas com elementos de $A_1, \dots, A_n$, sendo que não necessariamente todos precisam aparecer em uma sequência, um elemento pode repetir no máximo duas vezes e não queremos $A_i A_i^{-1}$ lado a lado. Note que essa abordagem nos fornece todos os prefixos de tamanho q possíveis. No nosso caso, colocamos $q = n$ e o código para essa primeira etapa pode ser visto no Bloco Python 7.5. E a sua descrição é a seguinte:

Função `gerar_prefixos(q: int, n: int) → list[tuple[int, ...]]`

Entrada

- q : inteiro, valor máximo (inclusivo) do alfabeto $\{1, \dots, q\}$.
- n : inteiro, comprimento desejado dos prefixos.

Saída

- Uma lista de tuplas de inteiros, cada tupla de tamanho n representando um prefixo válido em que cada valor de 1 a q aparece no máximo duas vezes e não há repetições adjacentes.

Agora o próximo passo foi criar uma função que, dadas duas palavras, ela deve verificar se existe uma equivalência por *swap*. Essa função retorna verdadeiro ou falso. Ela foi aplicada na lista de prefixos gerada anteriormente, eliminando as equivalências por swap e reduzindo a lista de prefixos que foi usada. Como esse pode ser um processo longo, note que a cada porcentagem do processo concluído, salvamos o resultado parcial em uma pasta local, para evitar retrabalho no caso de algum imprevisto ou caso seja necessário usar esses resultados para outros testes. Note que, a aplicação desse swap elimina prefixos que tem o mesmo padrão e que gerariam a mesma palavra ou em termos dos diagramas de emparelhamento, que teriam o mesmo padrão nos diagramas. Esse código pode ser encontrado no Bloco

Phyton 7.6 e a descrição dela é a seguinte:

Função `is_swap_equivalent_fast(s1, s2) → bool`

Entrada

- `s1`: tupla de inteiros, sequência original.
- `s2`: tupla de inteiros, sequência a ser comparada.

Saída

- Valor booleano: `True` se `s1` e `s2` diferem exatamente por uma troca (“swap”) de dois valores, `False` caso contrário.

Como próximo passo, fizemos uma pequena mudança na função *find single cluster words*, para que agora, além de receber o número de arestas n , o número de vértices v e se é para exibir o desenho ou não, ela também pode receber uma lista de prefixos. Caso não receba uma lista de prefixos, o único prefixo será A_1 . Quando ela recebe essa lista de prefixos, o objetivo é completar cada prefixo e já verificar a equivalência por rotação. Igual anteriormente, a saída dessa função é uma lista de palavras não equivalentes por rotação, que estão associadas a grafos de emparelhamentos de v vértices e n arestas. Essa nova versão da função pode ser vista no Bloco Python 7.7 e a descrição dela é a seguinte:

Função `find_single_cluster_words_prefixed(n, v=1, prefix=None, draw=False)`

Entrada

- `n`: inteiro, número de símbolos $A_1, \dots, A_n$ (palavra terá comprimento $2n$).
- `v`: inteiro (padrão 1), número exato de clusters que a palavra deve formar.
- `prefix`: lista de strings ou `None` (padrão `None`); prefixo inicial da palavra (fixa `["A1"]` se `None`).
- `draw`: booleano (padrão `False`); se `True`, plota cada palavra válida com `draw_chuva` e `draw_arrows`.

Saída

- Lista de palavras (cada palavra é uma lista de `str`) de comprimento $2n$ que formam exatamente v clusters, conforme identificado pelo algoritmo

de chuva de vértices.

As demais funções dessa abordagem foram omitidas do texto, mas podem ser encontradas no Notebook do Google colab que se encontra no seguinte link [Notebook Python - Segmentada](#) (link interativo).

Vamos replicar o exemplo com um grafo de 6 arestas e 3 vértices exibido detalhadamente pelo autor em [8]. Assim, também podemos ver que os códigos feitos são genéricos o suficiente para tratar casos em que existem mais vértices ou arestas.

Exemplo 3.1.14. Usando a palavra: $\mathcal{W} = A_1E_1C_1F_1D_1E_1C_1B_1A_1F_1B_1D_1$ associada a um emparelhamento de um grafo de 3 vértices e 6 arestas sobre o Bitoro (vide Figura 3.3) e o trecho de código:

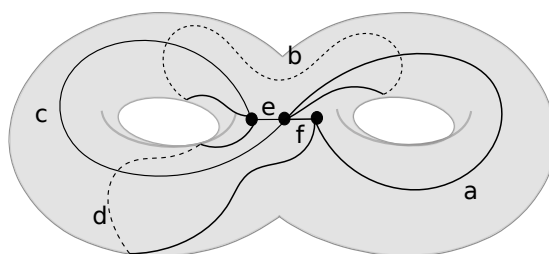


Figura 3.3: Mergulho do grafo de emparelhamento com 3 vértices e 6 arestas no Bitoro.

```

1 w = ("A1", "E1", "C1", "F1", "D1", "E1", "C1", "B1", "A1", "F1", "B1", "D1")
2
3 chord_indices = list(range(1, 7))
4 cl = chuva_de_vertices(list(w))
5 cl
6 draw_chuva(list(w), cl, chord_indices)
7 draw_arrows(list(w), cl)
8

```

Obtemos o seguinte output da Figura 3.4, assim como obtido no trabalho [8].

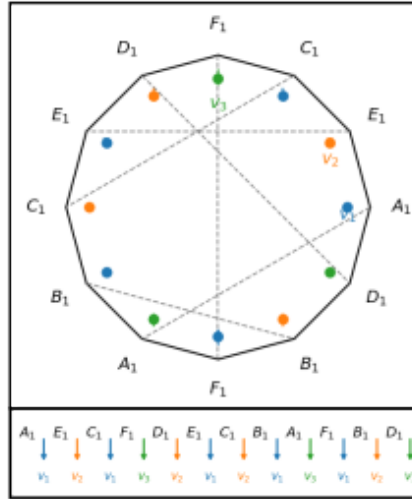


Figura 3.4: Exemplo de emparelhamento de 3 vértices e 6 arestas sobre o Bitoro

3.2 Emparelhamentos canônicos para o Toro e Bitoro

Ambas as abordagens de códigos podem ser aplicadas para o caso do Toro e Bitoro, onde ambos grafos canônicos tem um vértice, duas arestas (I_2), para o caso do Toro e quatro arestas (I_4) para o caso do Bitoro. Nesses casos não se faz necessário a inclusão de prefixos e nem segmentação dos processos.

Por simplicidade, os desenhos dos diagramas de emparelhamento foram feitos com as cordas nos vértices do polígono, mas para sair dessa representação e chegar na representação usual dos diagramas basta desenhar um polígono regular de mesma quantidade de lados, externamente, associando os vértices do polígono atual com as arestas do que será desenhado. Por esse motivo, essa ligeira diferença nas representações não são problemáticas. Além disso, nessas figuras, não fizemos distinção de A_i e A_i^{-1} , pois como o autor mostrou em [8], essas sílabas são intercambiáveis e para não incluir essas verificações no código, essas distinções não foram inclusas. Para o Toro, como já é conhecido, temos apenas um emparelhamento para o I_2 e ao executar o código para procurar esses emparelhamentos, obtemos a resposta representada na Figura 3.5:

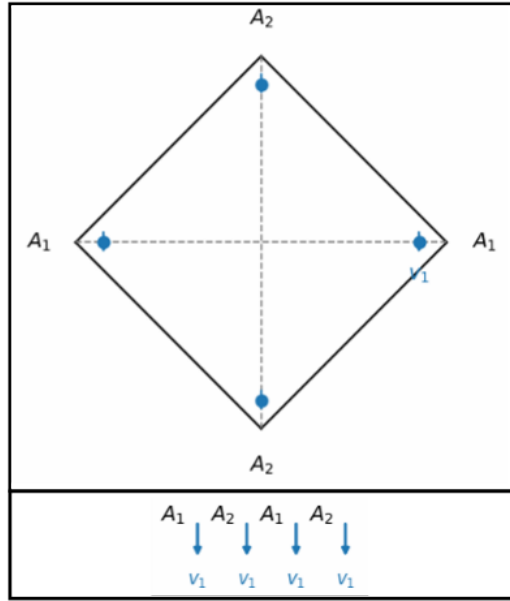


Figura 3.5: Emparelhamento de I_2 sobre o Toro

Teorema 3.2.1. *Existe um emparelhamento associado ao grafo canônico I_2 sobre o Toro. E ele está representado na Figura 3.5*

Teorema 3.2.2. *Existem 4 emparelhamentos associados ao grafo canônico I_4 sobre o 2 – Toro. Estes estão representados na Figura 3.6.*

Já para o caso do Bitoro, em [33], a autora mostra explicitamente 4 emparelhamentos possíveis para o I_4 sobre o Bitoro. Após a execução dos códigos, foi possível constatar que esses 4 emparelhamentos obtidos pela autora são os únicos possíveis, a menos de equivalência. A Figura 3.6 mostra os diagramas de emparelhamento e as palavras associadas aos mesmos, com o output da chuva de vértices.

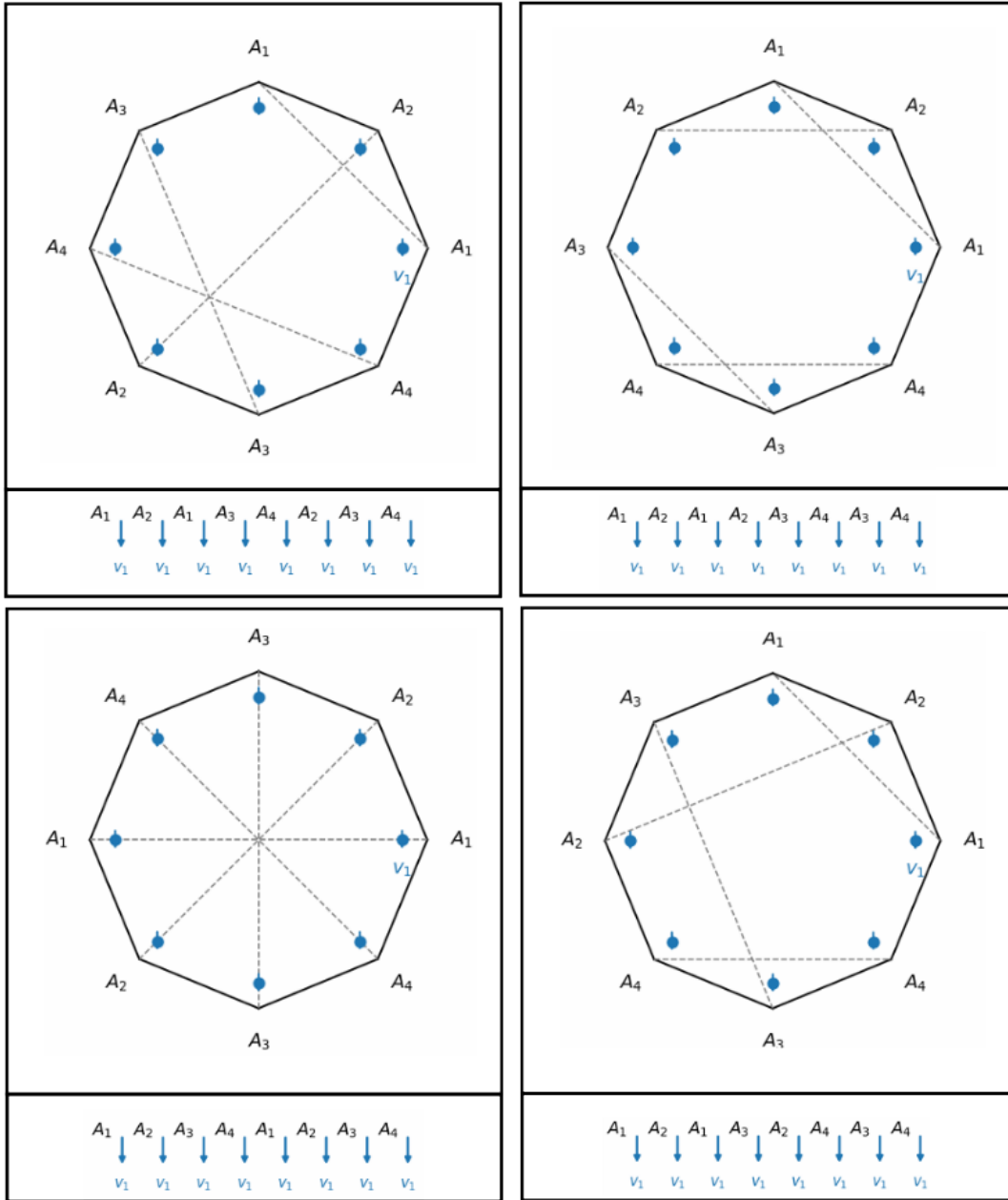


Figura 3.6: Emparelhamentos de I_4 sobre o Bitoro

3.3 Emparelhamentos canônicos para o Tritoro

O caso dos grafos de emparelhamento de arestas sobre o 3 – *Toro* foi explorado em [32], por *Nakamura*. Nesse trabalho o autor explorou os emparelhamentos 3 – *regulares* e exibiu todos os emparelhamentos associados a grafos 3–regulares possíveis, a menos de equivalência. No trabalho referido, Nakamura mostrou que existem 65 grafos trivalentes com 10 vértices e 15 arestas que podem ser mergulhados no tritoro, de modo que o complemento seja simplesmente conexo e existem

927 emparelhamentos associados a esses grafos. Usando o Teorema 3.1.1 de [7], poderíamos imaginar que a partir dos 927 emparelhamentos de Nakamura é possível contraí-los de forma a determinar todos os possíveis mergulhos de I_6 sobre o tritoro. De fato é possível, mas é garantido que cada mergulho pode ser contraído de forma única? A resposta é, não! Por exemplo, como vimos na Figura 3.6, existem 4 emparelhamentos associados ao I_4 , sobre o bitoro. Mas em [35], o autor mostrou que os 8 emparelhamentos trivalentes existentes, garantidos por [19], podem ser obtidos de apenas dois dos mergulhos da Figura 3.6, à saber os presentes no 1 e 4 quadrantes. Portanto, a fim de determinar todos os emparelhamentos associados a I_6 sobre o tritoro, as palavras e o algoritmo de chuva de vértices, pode nos ajudar. Aplicando os códigos para esse caso, obtivemos o Teorema 3.3.1. Note que, com o Teorema 3.1.1 de [7] e o Teorema 3.3.1, a menos de extensão de grafos, estão exibidos todos os emparelhamentos para o tritoro.

Teorema 3.3.1. *Existem 82 emparelhamentos associados ao grafo canônico I_6 sobre o tritoro. Estes estão explicitados na Seção 7.2 do Apêndice B, nas Figuras 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10.*

3.4 Emparelhamentos canônicos para o 4 – Toro

Para o caso do 4–Toro, em relação aos grafos de emparelhamento, não foram encontrados estudos com esse conteúdo. Até o momento, o único estudo que permitia gerar de forma sistemática emparelhamentos para o n –Toro, era [9], com a introdução das cirurgias. Mas agora, usando as palavras associadas aos mergulhos e o algoritmo chuva de vértices, podemos aplicar os códigos gerados e exibir, explicitamente, todos os emparelhamentos associados mergulho de I_8 sobre o 4–Toro. Para se gerar esses resultados, foram aplicados os códigos exibidos na Seção 3.1.2, mais especificamente a versão segmentada, devido ao alto poder de processamento necessário para executar o processo de uma só vez.

Aplicando os algoritmos, obtivemos o Teorema 3.4.1. Devido ao grande número de soluções, optamos por exibir as imagens no apêndice, a fim de deixar a leitura do texto fluida.

Como dito, os códigos exibidos na Seção 3.1.2, são mais que suficientes para executar a tarefa, para $n = 8$. Mas visando otimizar o tempo de execução e talvez permitir que esse algoritmo seja usado para valores maiores de n , construímos ainda uma versão alternativa de aplicação da função *find single cluster words*. A função continua exatamente como apresentada anteriormente, mas aqui colocamos uma implementação em paralelo, que consome mais recursos, à saber 196GB de memória RAM, mas permite que em algumas horas obtenhamos um *output*. O

código pode ser visto no Bloco Python 7.8, e a descrição é a seguinte:

Função `minimal_rotation(s)`

- **Entrada:** tupla `s` de inteiros.
- **Saída:** tupla contendo a rotação minimamente lexicográfica de `s`.

Função `process_prefix(prefix)`

- **Entrada:** tupla `prefix` de inteiros.
- **Saída:** lista de pares (can, w) onde `can` é a rotação mínima de cada palavra `w` gerada a partir de `prefix`.

Script principal

- **Entrada implícita:** `prefixos`, lista de tuplas a serem processadas.
- **Processo:** submete cada prefixo a `process_prefix` em paralelo, remove duplicatas canônicas via conjunto `canonical_set` e salva progresso em lotes de 1%.
- **Saída:** lista `Obtidas` de palavras únicas canônicas e logs de progresso impressos no console.

Vale comentar, que usamos as funções *gerar sequencias* e *is swap equivalent fast*, obtivemos 327.355 prefixos que seriam usados, posteriormente, na paralelização exibida da função *find single cluster words prefixed*. Em algumas horas obtivemos 237.020.420 de palavras, com sobreposições devido à iteração segmentada. Após retirarmos as sobreposições, usando a função *canonical pattern*, chegamos nas 7.258 que compõem o seguinte resultado:

Teorema 3.4.1. *Existem 7.258 emparelhamentos associados ao grafo canônico I_8 sobre o $4 - \text{Toro}$.*

Como a exibição dos resultados no texto ou mesmo em um apêndice ocuparia muito espaço, optamos por fornecer duas formas de visualização dos resultados. O primeiro deles é para os usuários que desejam apenas ver os resultados, neste caso podem acessar o PDF com os 7.258 emparelhamentos no seguinte link [Emparelhamentos do \$I_8\$ no \$4 - \text{Toro}\$](#) .

De forma alternativa, para o leitor que deseja trabalhar com os emparelhamentos disponibilizamos a base de dados em formato *.pkl* em [BaseEmparelhamentosI8.pkl](#). Após fazer o download do arquivo, o usuário deve executar os Blocos python 7.9 e 7.10.

Acreditamos que para a fim de replicar o estudo para o 5–Toro, com $n = 10$, no caso dos mergulhos do I_{10} , o código apresentado para o I_8 também seja executável. Para valores maiores de n é possível que o balanceamento ótimo entre poder de processamento necessário e tempo viável não seja bom. Portanto, os próximos passos dessa linha envolvem melhorar a implementação do algoritmo e pensar na possibilidade de usar algoritmos de *machine learning* para gerar (ou classificar) as palavras para um n dado, visto que agora conseguimos uma quantidade razoável de exemplos.

Capítulo 4

Reconstrução topológica de mergulhos

4.1 Reconstrução de grafos sobre superfícies

O problema de reconstrução de grafos a partir de subestruturas parciais ou informações incompletas tem sido amplamente estudado na teoria dos grafos clássica. Entretanto, quando consideramos grafos mergulhados em superfícies, surgem desafios adicionais relacionados à topologia envolvida e à estrutura celular induzida pelo mergulho. Esse cenário é particularmente relevante em aplicações de topologia combinatória, geometria algébrica e até mesmo em física matemática, onde a superfície de suporte carrega informações adicionais sobre o grafo.

Neste capítulo, exploramos a **reconstrução topológica de grafos sobre superfícies**, com foco especial em mergulhos 2-celulares em superfícies fechadas e orientáveis, cuja abordagem detalhada será apresentada na Seção 4.3. Inicialmente, retomamos a discussão sobre o *grafo paralelo*, já introduzido no contexto de grafos de emparelhamento, e avançamos para a generalização formal dos *Grafitos*, que neste caso coincidem com as palavras associadas aos mergulhos celulares. Tais Grafitos são definidos a partir do *Scissors Theorem* (Teorema 2.5.1) na Seção 2.5, constituindo uma ferramenta essencial para codificar percursos topológicos ao longo do grafo e as adjacências induzidas pela superfície.

Em seguida, ainda na Seção 4.2, desenvolvemos de forma detalhada a construção dos Grafitos, oferecendo um processo preciso e operacional para superfícies orientáveis sem fronteira, ao contrário de *Hoffman* [18], onde essa construção não é explicitada. Essa formalização é fundamental para o estudo das condições de reconstrução de mergulhos, uma vez que permite analisar combinatoriamente a estrutura das faces 2-celulares e estabelecer correspondências topológicas a partir de informações parciais.

Avançamos para o problema central de reconstrução de mergulhos celulares, abordando condições sob as quais um mergulho pode ser unicamente determinado a partir de subconjuntos ou palavras associadas parciais. Em um primeiro momento, apresentamos resultados inéditos e construtivos relativos à reconstrução de grafos de emparelhamento (Teorema 4.3.5), destacando como a estrutura da face 2-celular, via padrão na palavra associada, pode ser explorada para inferir a topologia original do mergulho.

Por fim, discutimos abordagens baseadas na análise do *tamanho das faces 2-celulares*, conectando esses aspectos combinatoriais com invariantes topológicos da superfície de suporte (Teoremas 4.3.9, 4.3.11, 4.3.13). Esses resultados abrem caminho para novas aplicações em reconstrução de superfícies, classificação de mergulhos e problemas inversos em topologia algébrica.

4.1.1 Reconstrução abstrata

A Conjectura da Reconstrução de Grafos é um dos problemas abertos mais antigos e importantes da Teoria dos Grafos. Sua origem remonta à tese de doutorado de P. J. Kelly, defendida em 1942 sob orientação de S. M. Ulam, onde investigava transformações isométricas em conjuntos finitos. Ulam reformulou a questão no contexto da teoria de conjuntos com distância binária, no livro *A Collection of Mathematical Problems* de 1960 [20,37].

Em 1957, Kelly traduziu o problema para a linguagem de grafos, provando que árvores e grafos desconexos com até seis vértices são reconstrutíveis [21]. A versão moderna da conjectura foi proposta por Harary, afirmando que todo grafo simples com pelo menos três vértices é determinado, a menos de isomorfismo, pelo multiconjunto de subgrafos induzidos obtidos pela exclusão de um vértice – o chamado *Deck* do grafo [15].

Diversos avanços foram registrados nas décadas seguintes. O Teorema de Contagem, formulado por Kelly e refinado por Greenwell e Hemminger, garante que o número de subgrafos isomorfos a um dado grafo H é reconstrutível a partir do *Deck* de G [28]. Além disso, diversas propriedades são hoje conhecidas como reconstrutíveis, incluindo grau dos vértices, sequência de graus, o número de vértices e arestas, regularidade, conectividade, entre outras [4,38].

Em 1977, Stockmeyer mostrou que a versão da conjectura para grafos direcionados é falsa, construindo contraexemplos entre os *tournaments* [36]. Isso motivou um foco maior nas propriedades específicas dos grafos não direcionados. Nessa linha, Harary e Robinson (1985) investigaram propriedades métricas como o diâmetro do grafo e de seu complemento, consideradas significativas e úteis para reduções da conjectura.

Na tentativa de restringir a classe de grafos necessários para provar a conjectura, Yang Yongzhi (1988) introduziu os chamados *P-graphs* e demonstrou que, se todos os grafos 2-conexos (vide Definição 2.2.6) forem reconstrutíveis, então todos os *P-graphs* também o são [39]. Isso implicou uma importante redução: a Conjectura da Reconstrução seria verdadeira se todos os grafos 2-conexos forem reconstrutíveis.

Seguindo esse caminho, Gupta, Mangal e Paliwal (2003) mostraram que a conjectura se reduz a três famílias específicas de grafos 2-conexos com diâmetro 2 ou 3 (i.e. tal que a distância máxima entre qualquer par de vértices seja 2 ou 3, denotado por $diam(G)$), que eles definem como $\mathcal{F}_1$, $\mathcal{F}_2$ e $\mathcal{F}_3$ [14]. Em 2009, Ramachandran e Monikandan reforçaram esse resultado ao mostrar que essas famílias são reconhecíveis e reconstrutíveis sob certas condições que combinam conectividade, diâmetro e propriedades do complemento [28].

Assim, os seguintes resultados equivalentes foram estabelecidos:

- A conjectura é verdadeira se todos os grafos 2-conexos G com $diam(G) = 2$ ou $diam(G) = diam(\overline{G}) = 3$ forem reconstrutíveis.
- Alternativamente, a conjectura também é verdadeira se todos os grafos 2-conexos com raio 2 forem reconstrutíveis. (Isto é, a conjectura é verdadeira se for possível reconstruir grafos 2-conexos cujo a menor excentricidade [Definição 2.2.7], dentre todos os vértices, é 2.)

Apesar de décadas de progresso e da verificação da conjectura para várias classes especiais de grafos (regulares, produtos cartesianos, lexicográficos, separáveis sem vértices-folha, árvores básicas, etc.), a Conjectura da Reconstrução continua aberta. No entanto, as reduções atuais restringem o problema a subclasses bem específicas, oferecendo um caminho promissor para uma resolução definitiva. A conjectura da reconstrução de grafos postula que todo grafo com pelo menos 3 vértices é unicamente determinado, a menos de isomorfismo, por sua coleção de subgrafos obtidos pela deleção de um único vértice (o *Deck*). Uma variante desse problema, formulada por Harary, também um problema em aberto até os dias de hoje, diz que todo grafo simples com pelo menos 4 arestas é unicamente determinado, a menos de isomorfismos, pelo seu *Deck de arestas*, isto é pela sua coleção de subgrafos obtidos pela deleção de uma única aresta.

Neste trabalho, estamos interessados na **reconstrução topológica**, ou seja, na possibilidade de recuperar o mergulho de um grafo em uma superfície fechada e orientável, a menos de homeomorfismo, a partir de informações parciais. Entretanto, a maior parte da literatura existente — especialmente os trabalhos clássicos

de Zhao e colaboradores — não trata diretamente da reconstrução do *mergulho*, mas sim da reconstrução **combinatória** do grafo abstrato, utilizando propriedades topológicas do mergulho como ferramenta técnica. Em particular, nos artigos de Zhao, os grafos são mergulhados em superfícies de gênero positivo, e são investigadas condições (como grau mínimo e representatividade) que garantem que o grafo abstrato é reconstrutível a partir do *deck* de deleções de arestas.

4.1.2 Reconstrução topológica

No início da década de 1980, casos fundamentais já haviam sido resolvidos. Notadamente, os *grafos planares maximais* (aqueles que triangulam a esfera) foram provados como reconstrutíveis [11, 25]. De fato, todos os grafos planares com grau mínimo suficientemente alto são reconstrutíveis: por exemplo, todo grafo planar com grau mínimo 5 é reconstrutível por arestas [24]. (A reconstrução por arestas é a variante em que o *deck* é composto por subgrafos com uma aresta removida; para muitas famílias de grafos, os problemas de reconstrução por vértices e por arestas são equivalentes [12].)

A partir da década de 1990, Zhao iniciou uma série de artigos em que investigou a reconstrução por arestas de grafos mergulhados em superfícies orientáveis [40–43]. Ainda que tais artigos considerem grafos mergulhados, seu foco é sempre a reconstrução do grafo abstrato, e não do mergulho em si. A topologia entra na forma de condições auxiliares: por exemplo, Zhao mostra que, se G é 3-conexo, tem grau mínimo $\delta \geq 6|\chi(\Sigma)| + 5$, e representatividade do mergulho ao menos 3, então G é reconstrutível por arestas [40].

Nos artigos seguintes, essas condições foram relaxadas. Em [41], Zhao provou que qualquer grafo com grau mínimo pelo menos 5 satisfazendo $(\delta - 5)|V(G)| + 6\chi(\Sigma) \geq 0$ é reconstrutível por arestas, bem como todo grafo com grau mínimo ao menos 4 que triangula uma superfície. Em [42], provou-se que basta ter grau mínimo 4 e número de vértices maior que $-\frac{4}{3}\chi(\Sigma)$. Finalmente, em [43], Zhao estendeu os resultados para grafos de grau mínimo 3 e largura mínima 5, usando o método do descarregamento.

Todos esses resultados são relevantes para a conjectura da reconstrução, mas sua ênfase é puramente combinatória. O mergulho é uma ferramenta — e não o objeto de reconstrução.

Complementarmente, houve avanços no problema original de reconstrução por vértices para grafos planares. Por exemplo, foi mostrado que a planaridade é uma propriedade reconstrutível [3], e que certas subclasses de grafos planares com conectividade suficiente são de fato completamente reconstrutíveis.

Neste trabalho, tomamos um caminho distinto: investigamos a possibilidade de reconstruir *a estrutura topológica do mergulho* de grafos em superfícies fechadas e orientáveis, a partir de seus minors ou subgrafos parcialmente observados. Ou seja, abordamos diretamente a **reconstrução topológica**, um problema ainda muito pouco explorado na literatura.

4.1.3 Propriedades topológicas para a reconstrução abstrata

Em Zhao, Y. [40], o objetivo do autor é provar o Teorema 4.1.3 e para isso é inicia o artigo com uma série de definições e lemas básicas:

Definição 4.1.1 ([40]). Seja M_g uma superfície fechada e orientada, de gênero $g > 0$. Uma curva fechada Γ de M_g é dita **essencial** se não é *null-homotopic* (Definição 2.1.8).

Definição 4.1.2 ([40]). Sejam M_g uma superfície fechada e orientada, de gênero $g > 0$ e $\mathcal{G}$ um mergulho de um grafo G em M_g . A representatividade de $\mathcal{G}$ é definida como:

$$\rho(\mathcal{G}) = \min\{\Gamma \cap \mathcal{G}, \Gamma \text{ é uma curva essencial de } M_g\}$$

Por simplicidade, o autor considera apenas curvas essenciais que passam apenas por vértices do grafo e por faces (2-células) e que não passe duas vezes pelo mesmo vértice ou face.

O seguinte Teorema é o resultado central do trabalho referido

Teorema 4.1.3 ([40]). *Seja G um grafo 3- Conexo de grau mínimo $6\alpha + 5$ onde α um inteiro não negativo. Se existe um mergulho de G em uma superfície M_g com representatividade pelo menos 3, com $\chi(M_g) = -\alpha|V(G)| + \beta$ onde $\beta \in \{0, 1\}$, então G é reconstrutível.*

Para provar o teorema central, o autor enuncia e demonstra uma sequência de lemas. O primeiro e o segundo deles, Lemmas 4.1.4, é comum em todos os artigos da sequência.

Lema 4.1.4 ([40]). *Seja G um grafo de valência mínima (Definição 2.2.8) mínima ζ . Suponha que, para algum $K \geq 0$, existe um vértice em G de grau $\zeta + K$ adjacente a $K + 1$ ou mais vértices de grau ζ*

Em todos os artigos dessa sequência, o autor tem resultados que afirmam que se um grafo tem determinadas estruturas como subgrafos, então é possível garantir a reconstrutibilidade. No artigo referido nessa seção, o Lema 4.1.5 figura como esse resultado característico.

Lema 4.1.5 ([40]). *Se um grafo G de valência mínima ζ contém um triângulo que é incidente a um vértice de grau ζ e dois vértices de grau $\zeta + 1$, então G é aresta reconstrutível.*

Depois do Lema 4.1.5, o autor faz restrições sobre os elementos com os quais ele vai trabalhar, se restringindo apenas a grafos, G , 3 – conexos, de valência mínima $6\alpha + 5$, com α um inteiro não negativo. As superfícies terão característica de Euler $-\alpha + \beta$, com $\beta \in \{0, 1\}$ e os mergulhos de G sobre uma superfície terão representatividade pelo menos 3. Além disso, ele se restringe aos grafos que não são regulares e nem *bidegreed* (i.e grafos cujo vértice se não tem grau a , tem grau b), pois para essas classes de grafos já existiam demonstrações de reconstrutibilidade abstrata por arestas, vide [5,31]. Após essas restrições ele demonstra dois resultados complementares e os usa para demonstrar o resultado central, acrescidos de um lema técnico.

Lema 4.1.6 ([40]). *Seja H um grafo conexo, de valência mínima ζ_H , mergulhado em uma superfície com característica de Euler $-\alpha|V(G)| + \beta$, onde α é um inteiro não negativo e $\beta \in \{0, 1\}$. Se H não é um grafo regular, então $\zeta_H \leq 6 + 5$.*

Lema 4.1.7 ([40]). *Existe pelo menos um vértice de grau ζ em G que é incidente a ζ 3-faces (faces 2-celulares) em M_g .*

Observação 4.1.8. O Lema 4.1.6 é usado para demonstrar o Lema 4.1.7. Além disso, na demonstração do Lema 4.1.7 o autor usa os fatos que o grafo é 3 – Conexo e que a representatividade é pelo menos 3 para provar que a vizinhança do vértice referido é um *Wheel-Neighborhood embedding*. Essa observação será usada para demonstrar o Teorema 4.3.13, onde no presente trabalho buscamos generalizar o Teorema 4.1.3 para a reconstrução de mergulhos.

Em Zhao, Y. [41], o autor tem três objetivos, à saber demonstrar os Teoremas 4.1.11, 4.1.18 e Corolário 4.1.19. No trabalho em questão o autor considera superfícies fechadas, limitadas e sem bordos. O artigo está dividido em duas seções, à saber "*Reconstructibility, surface embedding and minimum degree*", onde o autor demonstra o Teorema 4.1.11 e os lemas necessários para isso e "*Triangulations of surfaces*", onde o autor demonstra o Teorema 4.1.18 e o Corolário 4.1.19.

Além dos lemas comuns na sequência de papers do autor, um resultado importante para a demonstração do Teorema 4.1.11 é o Lema 4.1.9.

Lema 4.1.9 (4.1.11). *Se um grafo G de valência mínima ζ contém um dos seguintes grafos da Figura 4.1, onde em (a) $d(v) = \zeta$, $d(u) = d(w) = \zeta + 1$; em (b) $d(t) = \zeta$, $d(v) = d(u) = d(w) = \zeta + 1$; em (c) $d(t) = d(v) = \zeta$, $d(u) = d(w) = \zeta + 2$ e em (d) $d(t) = \zeta$, $d(v) = \zeta + 1$, $d(u) = d(w) = \zeta + 2$, então G é reconstrutível.*

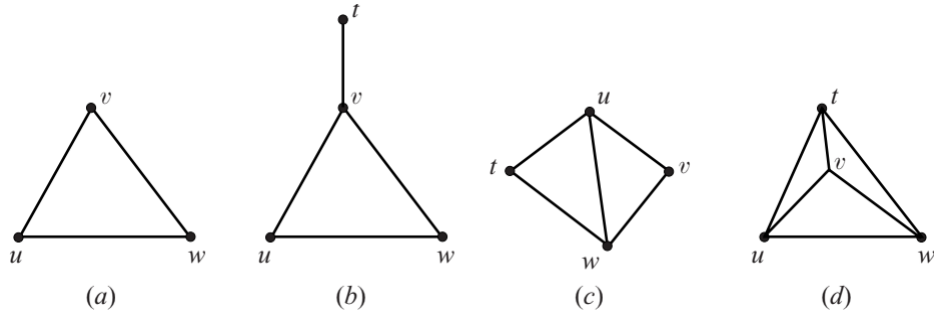


Figura 4.1: Subgrafos suficientes para a reconstrução abstrata - Figura reproduzida de [41]

Observação 4.1.10. Observe que o Lema 4.1.9 é o resultado do artigo em questão que fornece uma série de estruturas do grafo que são suficientes para a reconstrução abstrata. Ideias como essa também serão usadas quando formos tratar de reconstrução de mergulhos no trabalho presente, porém quando formos para esse contexto, somente os graus dos vértices e suas conexões não serão suficientes, teremos que fazer restrições também sobre o número e tamanho das faces 2 – *celulares* do mergulho. Tal abordagem será tratada na Seção 4.3.

Unindo o Lema 4.1.9 com o Lema 4.1.4, Zhao demonstra o seguinte teorema:

Teorema 4.1.11 ([41]). *Seja G um grafo com valência mínima $\zeta \geq 5$. Se G tem um mergulho em uma superfície Σ satisfazendo $(\zeta - 5)|V(G)| + 6\chi(\Sigma) \geq 0$, então G é aresta reconstrutível.*

Observação 4.1.12. Vale observar que um grafo que satisfaz o Teorema 4.1.3 também satisfaz o Teorema 4.1.11, pois se G é um grafo que satisfaz as hipóteses do Teorema 4.1.3, ele tem um mergulho em uma superfície Σ com representatividade pelo menos 3, grau mínimo $\zeta = 6\alpha + 5$ e a superfície em questão deve satisfazer $\chi(\Sigma) = \alpha n + \beta$, onde α é um inteiro não negativo e $\beta \in \{0, 1\}$. Dessa forma, temos:

$$\zeta = 6\alpha + 5 \Rightarrow \alpha = \frac{\zeta - 5}{6}.$$

Substituindo α na característica de Euler, temos:

$$\begin{aligned} \chi(\Sigma) &= -\alpha|V(G)| + \beta = -\left(\frac{\zeta - 5}{6}\right)|V(G)| + \beta \Rightarrow \\ &\Rightarrow 6\chi(\Sigma) + (\zeta - 5)|V(G)| = 6\beta \geq 0 \end{aligned}$$

Mas o Teorema 4.1.11 é mais geral, pois ele não faz nenhuma restrição sobre a disposição das arestas do grafo em algum mergulho existente, como por exemplo a representatividade do mergulho.

Na segunda parte do artigo o autor demonstra resultados referentes à grafos que triangulam superfícies, nessa parte ele demonstra o Teorema 4.1.18, seguido do Corolário 4.1.19 e para isso demonstram os seguintes resultados.

Lema 4.1.13 ([41]). *Seja G um grafo de grau mínimo 4 que triangula uma superfície Σ . Se H é uma reconstrução por arestas de G , isto é, tem o mesmo deck de arestas, então para qualquer 4-vértice v de H , segue que $\langle N_H(v) \rangle$ é hamiltoniano, ou seja, existe um ciclo em H que contém todos os vértices adjacentes a v .*

Lema 4.1.14 ([41]). *Seja G um grafo com valência mínima 4 que triangula uma superfície Σ . Se G não é aresta reconstrutível, então para qualquer 4-vértice $v \in V(G)$ com $\langle N_H(v) \rangle = C_4$, tem-se que v é adjacente a dois 5-vértices não adjacentes.*

Lema 4.1.15 ([41]). *Seja G um grafo de valência mínima 4 que triangula uma superfície Σ . Se G não é aresta reconstrutível, então para qualquer 4-vértice $v \in V(G)$ com $\langle N_H(v) \rangle = K_4 - e$, v é adjacente a dois 5-vértices não adjacentes.*

Lema 4.1.16 ([41]). *Seja G um grafo de grau mínimo 4 que triangula uma superfície Σ . Se existe um 4-vértice $v \in V(G)$ com $\langle N_H(v) \rangle = C_4$, então G é aresta reconstrutível.*

Lema 4.1.17 ([41]). *Seja G um grafo de grau mínimo 4 que triangula uma superfície Σ . Se existe um 4-vértice $v \in V(G)$ com $\langle N_H(v) \rangle = K_4 - e$, então G é aresta reconstrutível.*

Teorema 4.1.18 ([41]). *Seja G um grafo com valência mínima 4 que triangula alguma superfície Σ . Então G é aresta reconstrutível.*

Corolário 4.1.19 ([41]). *Se G triangula uma superfície de característica $\chi \geq 0$, então G é aresta reconstrutível.*

4.2 Generalização da palavra associada a um mergulho: relação entre Grafito e o grafo paralelo

A Seção 2.5 apresenta o *Scissors Theorem* (Teorema 2.5.1), que é utilizado posteriormente na mesma seção para definir as palavras (Grafito - Definição 2.5.2) associadas a qualquer grafo, mergulhado em qualquer superfície. No entanto, a forma de construir os Grafitos não é clara em Hoffman [18]. No nosso caso, o foco é em superfícies orientáveis sem fronteira.

Considere um mergulho $\mathcal{G} : G \hookrightarrow M$ de um grafo G em uma superfície orientável M . Em termos topológicos, existe sempre o *entorno tubular* de $\mathcal{G}$ em M — isto é, um conjunto aberto $\nu(\mathcal{G}) \subset M$ (Definição 3.1.2) que contém $\mathcal{G}$ e se parece localmente com $\mathcal{G} \times (-\varepsilon, \varepsilon)$ (um pequeno “tubo” ou faixa em torno de $\mathcal{G}$).

Formalmente, pode-se definir uma vizinhança tubular como um feixe normal trivial sobre $\mathcal{G}$, isto é, um par $(\nu(\mathcal{G}), p)$ onde $\nu(\mathcal{G})$ é um aberto de M contendo $\mathcal{G}$ e $p : \nu(\mathcal{G}) \rightarrow \mathcal{G}$ é uma projeção contínua que restringe à identidade em $\mathcal{G}$ (ou seja, uma retração por deformação). Intuitivamente, construímos $\nu(\mathcal{G})$ “engrossando” cada aresta em uma fita estreita dentro de M e cada vértice em um pequeno disco, de modo que $\mathcal{G}$ se deforma por retração dentro dessa vizinhança. Este resultado é análogo ao clássico Teorema da Vizinhança Tubular para subvariedades diferenciáveis, e vale também no contexto topológico e *piecewise-linear* (qualquer 1-complexo embutido em uma 2-variedade admite uma vizinhança tubular apropriada). Tais construções e definições podem ser encontradas no Capítulo 3 de [34].

Como consequência, $\nu(\mathcal{G})$ tem a estrutura de uma *superfície com bordo* (orientável, já que M é orientável) cujo retraente é $\mathcal{G}$. Os componentes de bordo de $\nu(\mathcal{G})$ formam curvas simples fechadas em M que “circundam” o grafo ao longo de suas arestas — as chamadas *curvas paralelas* (Definição 3.1.3) a $\mathcal{G}$. Mais precisamente, se denotarmos por $C(\mathcal{G})$ o número de componentes conexas do complemento $M \setminus \mathcal{G}$, então o bordo da vizinhança tubular $\nu(\mathcal{G})$ será uma união de $C(\mathcal{G})$ componentes de curvas simples fechadas em M . Cada uma dessas curvas de contorno corre paralelamente às arestas de $\mathcal{G}$ (a uma pequena distância constante), delineando os limites da faixa tubular em torno do grafo. Esse conjunto de curvas fechadas é o que chamamos de curvas paralelas, denotadas por $\gamma(\mathcal{G})$ e que está apresentado na Definição 3.1.3.

Como visto, a definição de *Curvas Paralelas* é válida para qualquer grafo mergulhado em qualquer superfície orientável, é claro (ao menos para o caso orientável) que o *Grafo Paralelo* (Definição 3.1.4) existe para o mergulho genérico referenciado.

Observação 4.2.1. Se $\mathcal{G}$ é um mergulho celular em uma superfície orientável M , pela definição de curvas paralelas, portanto pela definição de grafo paralelo (Definição 3.1.4), o número de componentes de $M \setminus \mathcal{G}$ é igual ao número de palavras no *Grafito*.

Observação 4.2.2. Se $\mathcal{G}$ é um grafo mergulhado em uma superfície orientável M , pela definição de grafo paralelo, cada componente de $\mathcal{B}(\gamma)$ é um caminho fechado e, como cada componente de $\mathcal{B}(\gamma)$ induz uma palavra do *Grafito*, cada palavra do *Grafito* é ciclicamente independente das demais. E, no caso de $\mathcal{G}$ ser um mergulho celular, cada componente de $\mathcal{B}(\gamma)$ é a borda de exatamente uma componente de $M \setminus \mathcal{G}$.

Observação 4.2.3. Uma pergunta natural após saber que a definição de grafo paralelo é válida para qualquer grafo mergulhado em uma superfície orientável é: "O

algoritmo da *Chuva de Vértices* funciona no caso genérico (qualquer grafo mergulhado, de forma celular, em qualquer superfície orientável) também?" E a resposta é positiva. A *Chuva de Vértices* funciona neste caso sob exatamente as mesmas regras mostradas na Seção 3.1.1, pois o algoritmo é baseado na regra da Observação 3.1.10 e essa regra também funciona no caso genérico. Basta notar que a inversa de uma sílaba pode ser colocada em uma palavra diferente do *Grafito*, mas essa informação não altera a execução do algoritmo.

Observação 4.2.4. Vale notar que a relação do padrão de distribuição das sílabas, em relação ao grau do vértice, como visto no Exemplo 3.1.14, se mantém para o caso geral. O que pode naturalmente acontecer e deve ser observado é que as sílabas referentes aos padrões podem estar em palavras distintas na composição do *Grafito*.

Notação 4.2.5. Para representar explicitamente um *Grafito*, separaremos com contrabarra (\) as palavras do *Grafito*. E, se um *Grafito* está associado a um mergulho de grafo em uma superfície de gênero g , possivelmente o nomearemos por $\mathcal{W}_g(\mathcal{G})$.

Exemplo 4.2.6. Seja $\mathcal{G}$ o seguinte grafo (Figura 4.2) mergulhado em $M_0 = \mathbb{S}^2$:

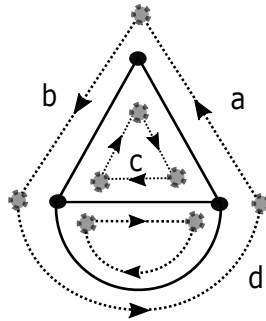


Figura 4.2: Grafo com 4 arestas, 3 vértices e seu grafo paralelo em linhas pontilhadas.

Note que, $\mathcal{B}(\gamma)$ induz o seguinte *Grafito*:

$$\mathcal{W}_0(\mathcal{G}) = A_1 \ B_1 \ D_1 \ \backslash \ D_1^{-1} \ C_1 \ \backslash \ B_1^{-1} \ A_1^{-1} \ C_1^{-1}$$

Como $\mathcal{G}$ é um mergulho celular em $\mathbb{S}^2$, o número de componentes de $\gamma(\mathcal{G})$ é igual ao número de palavras no *Grafito*.

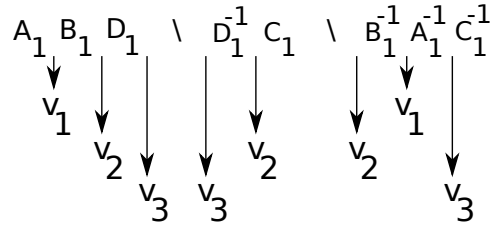


Figura 4.3: Algoritmo da *Chuva de Vértices* para $\mathcal{W}_0(\mathcal{G})$.

Note que, como o vértice que conecta as arestas a e b tem grau 2, a seta V_1 aparece duas vezes, e as outras aparecem três vezes.

4.3 Reconstrução de mergulhos celulares

Seja $\mathcal{G}$ um mergulho de um grafo conexo sobre uma superfície fechada e orientada de gênero g M_g . Já sabemos que por se tratar de um mergulho celular, esse mergulho pode ser associado, a menos de equivalência, a um único *Grafito*. Nessa Seção vamos abordar o tema das propriedades do *Grafito* em termos da representação dos subgrafos $\mathcal{G}$, sobre a superfície e como essas propriedades podem ser usadas para reconstruir o mergulho dado. Antes disso, vamos ver algumas ideias e definições importantes no contexto de reconstrução topológica de mergulhos.

4.3.1 O problema de reconstrução de mergulhos

Após observar o problema de reconstrução de arestas, surgiu uma variação deste problema. Seja M_g uma superfície fechada e orientável de gênero g e $\mathcal{G}$ um grafo mergulhado em M_g . Denotemos por $\mathcal{D}(\mathcal{G})$ o seguinte multiconjunto $\{\mathcal{G} - a; a \in E(\mathcal{G})\}$. Chamaremos $\mathcal{D}(\mathcal{G})$ de deck de arestas do mergulho do grafo $\mathcal{G}$. A questão central é: “Quais são as condições que $\mathcal{G}$ deve satisfazer para ser reconstrutível, a partir de $\mathcal{D}(\mathcal{G})$, como um mergulho de grafo?”

Adicionamos apenas um elemento no problema inicial, mas isso traz algumas complicações. A primeira delas é a seguinte. No problema tradicional de reconstrução de arestas (para grafos abstratos), dado o deck de um grafo e dois vértices não adjacentes em algum grafo do deck, a questão é: “A aresta ausente conecta ou não esses dois vértices?” Mas, em nosso problema, dado os mesmos objetos, a questão muda para: “A aresta ausente conecta ou não esses dois vértices? E, se conecta, como essa conexão ocorre?” Essa mudança torna a resposta mais difícil e, talvez, não seja trivial replicar os resultados simples de grafos abstratos para mergulhos de grafos.

Por exemplo, se G é um grafo abstrato (simples e conexo) com grau mínimo δ , e existem dois vértices adjacentes com grau δ , então G é reconstrutível em termos de arestas abstratas. Mas esse resultado não é trivial quando se trata de mergulhos de grafos.

Exemplo 4.3.1. O grafo a seguir, na Figura 4.4, quando considerado abstratamente, é reconstrutível em termos de arestas. Pois é simples, conexo e possui dois vértices adjacentes com grau mínimo.

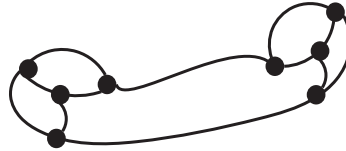


Figura 4.4: Grafo abstrato 3-regular

Mas não é claro no caso desse grafo mergulhado no toro. Pois temos, pelo menos, dois modos de fazer esse mergulho, alterando apenas uma aresta, veja a Figura 4.5.

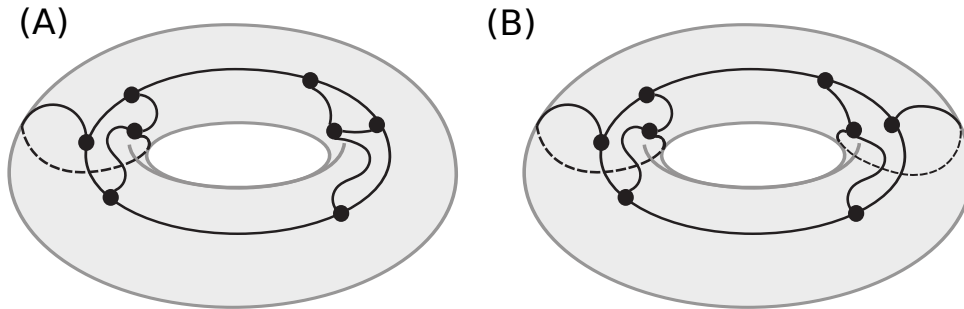


Figura 4.5: Dois mergulhos diferentes do mesmo grafo 3-regular

A segunda complicação, talvez menos complexa de resolver, é: "*Os mesmos invariantes serão reconstrutíveis a partir do deck de mergulho?*" Se denotarmos por G o grafo abstrato, e por $\mathcal{G}$ o mergulho de G em uma superfície M_g , é claro que conhecer $\mathcal{D}(\mathcal{G})$ implica conhecer $\mathcal{D}(G)$, mas a recíproca não é necessariamente verdadeira. Assim, os invariantes intrínsecos ao grafo, como a sequência de graus e o menor grau, são naturalmente reconstrutíveis a partir de $\mathcal{D}(\mathcal{G})$. Porém, invariantes envolvendo o número de subgrafos com alguma propriedade precisam ser estudados com mais detalhes.

Definição 4.3.2 (Mergulho reconstrutível). Um mergulho $\mathcal{G}$ de um grafo G , em uma superfície fechada e orientada de gênero g , M_g é dito reconstrutível (topologi-

camente) se pode ser obtido unicamente, a menos de homeomorfismo, a partir do seguinte multiconjunto, chamado de *Deck* do mergulho:

$$\mathcal{D}(\mathcal{G}) = \{\mathcal{G} - e; e \in E(\mathcal{G})\}.$$

Observe que dado o *Deck* do mergulho, é possível obter o *Deck* do grafo abstrato. Portanto, para as características intrínsecas ao grafo, ou seja, aquelas que são independentes do mergulho, os resultados que conhecemos sobre reconstrução seguem de forma natural. Por exemplo, a sequência de graus do mergulho, o número de arestas, o número de vértices e a planaridade do grafo são reconstrutíveis, quando estamos tratando dos mergulhos. Além disso, grafos 3–Conexos, mergulhados na esfera são reconstrutíveis como mergulho, visto que para esses grafos o mergulho é único.

Já sabemos que um Grafito associado ao mergulho carrega características intrínsecas do grafo, como o número de arestas do grafo mergulhado e que a relação é que o número de sílabas totais do Grafito é o dobro do número de arestas do grafo mergulhado. E o número de vértices podemos obter do Grafito a partir do algoritmo "chuva de vértices". Também sabemos que o deck de arestas de um mergulho $\mathcal{G}$ sobre uma superfície M_g é um multiconjunto cujos elementos são compostos pela superfície M_g e um subgrafo de $\mathcal{G}$ a menos de uma aresta. Portanto, a partir do deck de arestas do mergulho, sabemos o número de vértices, V , do grafo original, pois esse valor não se altera no deck, sabemos o número de arestas, pois por definição é exatamente uma a menos que o grafo original. E por fim, como conhecemos a superfície M_g , então sabemos também a sua característica de Euler. Portanto, a partir do deck, é possível saber o número de faces que o mergulho original deve ter, visto que $\chi(M_g) = V - A + F$, onde V é o número de vértices do grafo, A o número de arestas, F o número de faces 2-celulares e $\chi(M_g)$ a característica de Euler da superfície fechada e orientável de gênero g . O que nos permite introduzir o seguinte resultado.

Proposição 4.3.3. *Seja $\mathcal{G}$ um mergulho celular sobre M_g . Dado o deck de arestas do mergulho, o número de faces do mergulho é reconstrutível.*

Antes de seguirmos para os próximos resultados, observe o Exemplo 4.3.4 de grafos mergulhados na esfera S^2 .

Exemplo 4.3.4 (Relação entre padrões nas palavras e grau do vértice). Agora já sabemos como uma palavra pode representar um mergulho de um grafo em uma superfície fechada e orientada de gênero g . O algoritmo *Chuva de vértices* nos dá uma relação entre as palavras e o número de vértices do grafo e também sobre

os graus desses vértices. Mas além dessa relação, é possível identificar padrões de disposição das sílabas nas palavras a medida que variamos o grau do vértice observado.

Quando o mergulho tem um vértice de grau 2, podemos observar que ao caminhar sobre as arestas do grafo paralelo, se em um momento passamos ao lado das arestas a_1 e a_2 , nessa ordem, seguindo a caminhada a segunda e última vez que passarmos ao lado dessas arestas será na ordem a_2 e a_1 , na Figura 4.6 é possível ver à esquerda a visão local do vértice de grau 2 e à direita a palavra obtida após a construção do grafo paralelo.

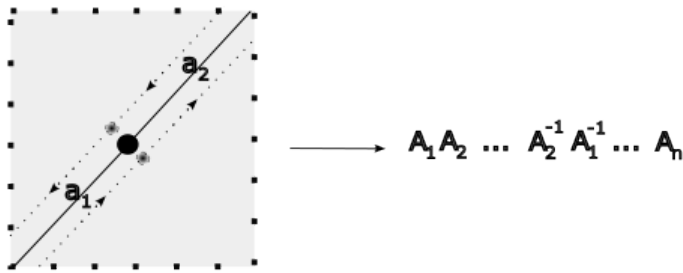


Figura 4.6: Visão local: padrão na palavra com vértice de grau 2

A primeira relação, vista na Figura 4.6, ocorre em superfícies gerais, de gênero $g \geq 0$. De agora em diante, nesse exemplo, vamos considerar apenas $g = 0$, isto é, mergulhos celulares na esfera. Assim, quando temos um vértice de grau 3, temos a seguinte distribuição, apresentada na Figura 4.7:

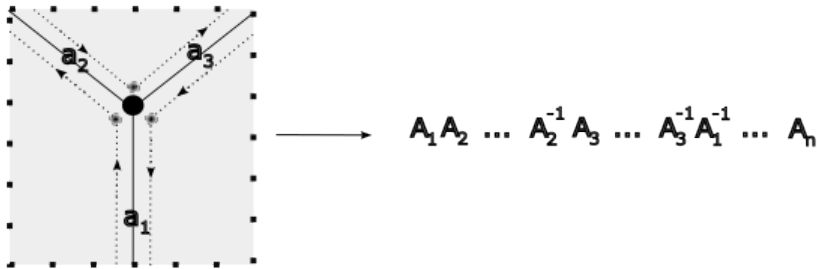


Figura 4.7: Visão local: padrão na palavra com vértice de grau 3

Para grau 4, temos a distribuição apresentada na Figura 4.8:

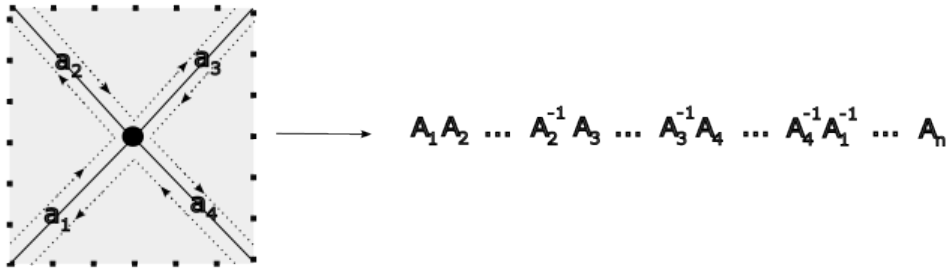


Figura 4.8: Visão local: padrão na palavra com vértice de grau 4

Agora, se o vértice tem um grau genérico ζ , temos a distribuição apresentada na Figura 4.9. Nesse caso, as reticências na figura representam as $\zeta - 5$ faces omitidas para o caso de um grau genérico, enquanto as reticências na palavra representam um conjunto de sílabas omitidas, lembrando que a ordem em que os pares aparecem pode ser alterada, sem nenhum problema para o padrão.

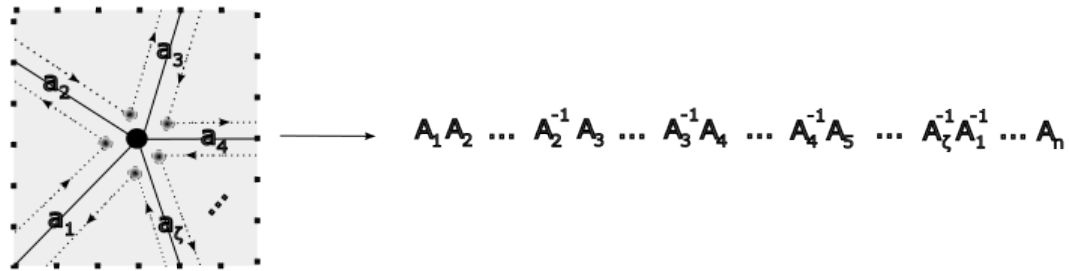


Figura 4.9: Visão local: padrão na palavra com vértice de grau ζ

4.3.2 Reconstrução de grafos de emparelhamento

O grafo de emparelhamento de arestas é o tipo mais simples de mergulho celular e para atacar o problema de reconstrução de mergulhos, essa será a primeira classe que abordaremos. Já vimos a definição dos grafos de emparelhamento de arestas, suas principais propriedades, que o mesmo pode ser representado por um *Grafo* composto por uma única componente, ao qual, sem o risco de confusão, vamos chamar de palavra. Agora provaremos um resultado, com condições suficientes para que alguns desses grafos sejam reconstrutíveis como mergulhos sobre a superfície a qual estão alocados.

Teorema 4.3.5. *Seja $\mathcal{G}$ um grafo de emparelhamento sobre a superfície M_g , com $\deg(v) \geq 3 \forall v \in V(\mathcal{G})$. Se existe uma aresta que conecta dois vértices de grau 3, que divide o mergulho, então $\mathcal{G}$ é reconstrutível sobre M_g .*

Demonstração. Seja $\mathcal{G}$ um mergulho de um grafo de conexo G , com $\deg(v) \geq 3$, $\forall v \in V(G)$, sobre a superfície M_g . Considere que a aresta $e \in E(\mathcal{G})$ é uma aresta que conecta dois vértices de grau 3 e $\mathcal{G} - e$ tem duas componentes. Dado o deck do mergulho $\mathcal{D}(\mathcal{G})$ sabemos, pela Proposição 4.3.3, que é possível saber o número de faces do mergulho original. Portanto, sabemos que $M_g - \mathcal{G} = 1$, pois por hipótese $\mathcal{G}$ é um grafo de emparelhamento. Dessa forma, a palavra $\mathcal{W}_{\mathcal{G}}$ associada ao mergulho $\mathcal{G}$ deve ter única componente.

Agora, ao fazer os grafitos associados aos elementos de $\mathcal{D}(\mathcal{G})$, existe pelo menos uma, à saber $\mathcal{W} - e$, que possui duas componentes, de modo que as duas peças não compartilhem sílabas, isto é, se uma sílaba A_i está em uma componente, a sua inversa A_i^{-1} também está. Isso se dá pelo fato de que uma aresta separa o mergulho. Além disso, como $e \in E(\mathcal{G})$ conecta dois vértices de grau 3, temos que em cada componente existe um vértice de grau 2. Com isso, em uma componente do grafito, existe um padrão do tipo $A_i A_j \dots A_j^{-1} A_i^{-1} \dots$ e na outra um padrão do tipo $A_k A_l \dots A_l^{-1} A_k^{-1} \dots$, a menos de equivalência, em ambos os casos. Como a sequência de graus é reconstrutível a partir de $\mathcal{D}(\mathcal{G})$, sabemos que o grau mínimo é 3, então não pode ocorrer esses tipos de padrão em nenhuma das arestas. Portanto, a fim de eliminar esse padrão em cada componente do grafito colocando A_e entre $A_i A_j$ ou entre $A_j^{-1} A_i^{-1}$ e A_e^{-1} entre $A_k A_l$ ou $A_l^{-1} A_k^{-1}$. Para juntar as componentes, basta fazer um *shift* em cada uma delas de modo que A_e e A_e^{-1} estejam no início das componentes e posteriormente concatenar as duas peças em uma só. Note que é indiferente quais das opções vamos escolher quando dizemos "colocando A_e entre $A_i A_j$ ou entre $A_j^{-1} A_i^{-1}$ e A_e^{-1} entre $A_k A_l$ ou $A_l^{-1} A_k^{-1}$ ", pois ao fazermos o *shift* e concatenar as palavras, A_e e A_e^{-1} ainda separa o grafito, consequentemente o mergulho, e além disso, cada componente não altera o seu padrão de relação no diagrama de emparelhamento. E essa manutenção do padrão e equivalência é garantida pela propriedade T1, T4 e T5 da Propriedade 3.1.9 $\square$

Na Seção 2.4 apresentamos as cirurgias de grafos de emparelhamento introduzidas por [9]. Na cirurgia S_1 , pode-se observar que o grafo obtido está exatamente nas condições do Teorema 4.3.5. Portanto, de forma natural, podemos enunciar o seguinte Corolário.

Corolário 4.3.6. *Todo grafo de emparelhamento de arestas que é fruto da cirurgia S_1 , de outros dois grafos de emparelhamento, é reconstrutível como mergulho.*

4.3.3 Reconstrução de mergulhos com restrições sobre o tamanho da face

Como já mencionado, o problema de reconstrução de mergulhos gerais é um problema amplo e com muitas dificuldades. Na Subseção 4.3.2, atacamos um problema com uma restrição grande sobre o número de faces, mergulhos com apenas uma face 2-celular. Neste capítulo, a menos que se diga o contrário, quando dissermos "faces" estamos nos referindo à 2-células.

Agora não vamos fazer restrições sobre o número de faces e sim sobre o tamanho das faces, em especial vamos nos concentrar no tamanho da maior face. Nessa seção, ao considerarmos as maiores 2-faces do mergulho e estudarmos o deck do mergulho, teremos elementos no deck que não existem no mergulho original e esses elementos nos darão pistas sobre a possibilidade de reconstrução do mergulho, à saber esses elementos serão 2-faces maiores que as existentes no mergulho original. Vamos explorar quais características adicionais devemos ter para garantir a reconstrução dos mergulhos.

Até então, já vimos que um código pode ser associado a um mergulho celular de modo a carregar características intrínsecas do grafo e também características topológicas do mergulho. Já vimos também, na Proposição 4.3.3 que o número de faces 2-celulares pode ser reconstruído a partir do deck do mergulho. Agora, a fim de concluir o objetivo dessa seção, veremos como um grafito se comporta ao deletarmos arestas do mergulho associado.

A seguir vamos definir dois tipos de decomposição do grafito. Seja $\mathcal{W}$ um Grafito associado a um mergulho 2-celular $\mathcal{G}$ em M_g , uma superfície fechada e orientável, de gênero g . Suponha que $\mathcal{W}$ é composto por $\mathcal{W}_1, \dots, \mathcal{W}_n$. Quando estamos tentando o mergulho $\mathcal{G}$, temos as informações do deck do mergulho:

$$D(\mathcal{G}) = \{\mathcal{G} - e; e \in E(\mathcal{G})\}.$$

Como podemos ver, esse multiconjunto é obtido pela deleção de arestas do mergulho original. Ao deletar arestas de um mergulho celular, temos duas opções resultantes, a aresta deletada está na interseção de duas 2-faces e a deleção dessa aresta resulta em um mergulho celular ou a aresta deletada está apenas em uma 2-face e a deleção dessa aresta não resulta em um mergulho 2-celular. Essas duas opções podem ser descritas em termos do grafito e a Definição 4.3.7 traz as opções de decomposição do grafito.

Definição 4.3.7 (Decomposição de grafitos). Dado um Grafito $\mathcal{W}$, composto pelas palavras $\mathcal{W}$ é composto por $\mathcal{W}_1, \dots, \mathcal{W}_n$ e associado ao mergulho 2-celular $\mathcal{G}$, ao deletar uma aresta do mergulho tem-se os seguintes casos de decomposição:

1. Se $a_i \in E(\mathcal{G})$ pertence a somente uma face do mergulho, ou seja, $A_i, A_i^{-1} \in \mathcal{W}_k$, temos que:

$$\mathcal{W}_{a_i} = \mathcal{W}_k \setminus \dots \setminus \mathcal{W}_{k1} \setminus \mathcal{W}_{k2} \setminus \mathcal{W}_{k+1} \setminus \dots \setminus \mathcal{W}_n,$$

onde $\mathcal{W}_{k1} \setminus \mathcal{W}_{k2} = \mathcal{W}_k - \{A_i, A_i^{-1}\}$.

2. Se $a_i \in E(\mathcal{G})$ está na interseção de duas 2-faces, ou seja, $A_i \in \mathcal{W}_k$ e $A_i^{-1} \in \mathcal{W}_l$, devemos reordenar $\mathcal{W}_k$ e $\mathcal{W}_l$ de modo que A_i e A_i^{-1} ocupem posições extremas e idênticas e justapor $\mathcal{W}_k$ e $\mathcal{W}_l$, obtendo o que vamos chamar de $\mathcal{W}_x$. Dessa forma, temos:

$$\mathcal{W}_{a_i} = \mathcal{W}_1 \setminus \dots \setminus \mathcal{W}_x \setminus \dots \setminus \mathcal{W}_n.$$

Exemplo 4.3.8. Na Figura 4.3, podemos observar que todas as arestas pertencem a exatamente duas faces do mergulho, portanto, qualquer que seja a aresta retirada, estaremos no Caso 2 da Definição 4.3.7. A palavra associada ao mergulho da Figura 4.3 é:

$$\mathcal{W}_0(\mathcal{G}) = A_1 \ B_1 \ D_1 \ \setminus \ D_1^{-1} \ C_1 \ \setminus \ B_1^{-1} \ A_1^{-1} \ C_1^{-1}.$$

Ao retirarmos a aresta c , temos o seguinte grafito associado:

$$A_1 \ B_1 \ D_1 \ \setminus \ D_1^{-1} B_1^{-1} A_1^{-1}.$$

Agora, com base na Definição 4.3.7 e na Proposição 4.3.3 dado um Grafito que representa um elemento do deck é possível saber se esse grafito é proveniente da decomposição 1 ou 2, de acordo com o número de faces (ou palavras no grafito), que deveria ter em comparação com tanto que de fato possui. Agora, observe que se um mergulho com pelo menos 3 faces tem duas faces de tamanho máximo δ adjacentes, retirando a aresta que as conecta, temos uma decomposição do tipo 2. Além disso, nesse caso, tomando o máximo das palavras que compõem os grafitos obtidos por decomposições do tipo 2, conseguimos obter o tamanho da maior face δ . Esse raciocínio será usado implicitamente nos resultados dessa seção.

Nessa seção vamos nos concentrar no caso onde a decomposição de um mergulho celular, ainda é um mergulho celular, isto é, no Caso 2 da Definição 4.3.7. O Seguinte Teorema:

Teorema 4.3.9. *Seja $\mathcal{G}$ um mergulho celular sobre a superfície M_g , com grau mínimo 3 e pelo menos três faces, tal que a maior face tenha tamanho δ . Se o mergulho tem duas faces de tamanho δ adjacentes, que compartilham pelo menos um vértice de grau 3, então o mergulho é reconstrutível.*

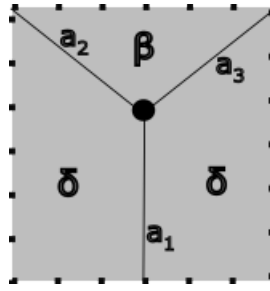


Figura 4.10: Visão local do mergulho $\mathcal{G}$, com faces de tamanho δ adjacentes

Demonstração. Seja $\mathcal{G}$ um mergulho celular sobre a superfície M_g , com grau mínimo 3 e pelo menos 3 faces, tal que o tamanho da maior face seja δ . Nesse caso, se $\mathcal{W}$ é o grafito associado ao mergulho $\mathcal{G}$, o máximo de sílabas que uma palavra que compõe $\mathcal{W}$ pode ter é δ .

Como na hipótese, suponha que o mergulho tenha duas faces de tamanho δ adjacentes e que essas compartilham pelo menos um vértice de grau 3. Suponha que a aresta a_1 seja a aresta comum entre as faces de tamanho δ , a_2 seja a aresta da esquerda, entre a face de tamanho δ e a face de tamanho β e a_3 a aresta restante, representadas na Figura 4.10. Note que, em $\mathcal{W}$, existe uma palavra, à saber representando a face de tamanho β , que possui a seguinte configuração $\alpha_i \dots A_3 A_2 \dots \Gamma_i$. Com as nomenclaturas das arestas fixadas, temos $\mathcal{G} - a_1$ pertence ao deck do mergulho e o Grafito $\mathcal{W}_{a_1}$, associado ao subgrafo $\mathcal{G} - a_1$ mergulhado em M_g é proveniente da Decomposição 2 da Definição 4.3.7, isso é, ele é da forma

$$\mathcal{W}_{a_1} = \mathcal{W}_1 \setminus \dots \setminus \mathcal{W}_x \setminus \dots \setminus \mathcal{W}_n.$$

Além disso, vale observar que esse grafito contém a mesma palavra $\alpha_i \dots A_3 A_2 \dots \Gamma_i$, digamos $\mathcal{W}_1$, do grafito original $\mathcal{W}$, pois as arestas da face de tamanho β não foram alteradas e a palavra de $2\delta - 2$ sílabas, que representa a nova face criada é $\mathcal{W}_x$, dada por:

$$\mathcal{W}_x = \alpha_j A_2^{-1} A_3^{-1} \Gamma_j.$$

Como o grau mínimo do mergulho é 3, sabemos que não pode existir nenhuma ocorrência do tipo $A_k A_l$ e $A_l^{-1} A_k^{-1}$ no grafito original. Portanto, devemos retirar a combinação $A_3 A_2$ de $\mathcal{W}_1$ ou $A_2^{-1} A_3^{-1}$ de $\mathcal{W}_x$. Uma vez que sabemos, por hipótese, que o tamanho da maior face é δ , devemos desfazer a palavra $\mathcal{W}_x$, que compõe o grafito $\mathcal{W}$, pois essa tem tamanho 2δ e portanto vamos desfazer a ocorrência $A_2^{-1} A_3^{-1}$. Dessa forma, o primeiro passo da reconstrução é colocar A_1 entre A_2^{-1} e A_3^{-1} :

$$\mathcal{W} = \mathcal{W}_1 \dots \setminus \alpha_j A_2^{-1} \textcolor{red}{A_1} A_3^{-1} \Gamma_j \setminus \dots \mathcal{W}_n.$$

Reescreva o grafito com A_1 figurando como a primeira sílaba de $\mathcal{W}_x$

$$\mathcal{W} = \mathcal{W}_1 \dots \setminus \textcolor{red}{A_1} A_3^{-1} \Gamma_j \alpha_j A_2^{-1} \setminus \dots \mathcal{W}_n.$$

Conte δ sílabas a partir de A_1 e posicione A_1^{-1} :

$$\mathcal{W} = \mathcal{W}_1 \dots \backslash A_1 A_3^{-1} \Gamma_j A_1^{-1} \alpha_j A_2^{-1} \backslash \dots \mathcal{W}_n.$$

Por fim, faça a separação da palavra em duas, deixando A_1 e A_1^{-1} como as primeiras sílabas de cada nova palavra:

$$\mathcal{W} = \mathcal{W}_1 \dots \backslash \mathcal{W}_{x1} \backslash \mathcal{W}_{x2} \backslash \dots \mathcal{W}_n,$$

onde $\mathcal{W}_{x1} = A_1 A_3^{-1} \Gamma_j$, $\mathcal{W}_{x2} = A_1^{-1} \alpha_j A_2^{-1}$ e cada uma dessas novas palavras tem tamanho δ . Reconstruindo assim o grafito original e portanto o mergulho original. $\square$

Uma vez demonstrado o Teorema 4.3.9, podemos observar que as triangulações de superfícies satisfazem as condições do Teorema supracitado. Portanto, podemos introduzir de forma natural o seguinte resultado.

Corolário 4.3.10. *Se uma triangulação de M_g possui pelo menos um vértice de grau 3, então essa triangulação é reconstrutível como mergulho.*

Agora, a fim de expandir um pouco mais a abrangência do Teorema 4.3.9, como já sabemos o que ocorre com o grafito associado a um mergulho, caso o mergulho tenha um vértice de grau ζ , visto no Exemplo 4.3.4. Usando essas relações, podemos demonstrar o seguinte resultado.

Teorema 4.3.11. *Seja $\mathcal{G}$ um mergulho celular sobre a superfície S^2 com grau mínimo ζ e pelo menos três faces, tal que a maior face tenha tamanho δ . Se o mergulho tem duas faces de tamanho δ adjacentes, que compartilham pelo menos um vértice de grau ζ , então o mergulho é reconstrutível.*

Demonstração. A demonstração é análoga à feita para o Teorema 4.3.9. Basta tomar um elemento do deck, que tem uma face de tamanho 2δ e compartilham um vértice de grau $\zeta - 1$. O padrão a ser evitado no grafito será distinto do padrão usado no Teorema citado, mas será idêntico ao visto no Exemplo 4.3.4. $\square$

O Teorema 4.3.11 aumenta a cobertura de exemplos que satisfazem o Teorema 4.3.9. Agora, com essa expansão nas hipóteses com relação ao grau, temos uma consequência direta que é a expansão do Corolário 4.3.10, para o Corolário 4.3.12.

Corolário 4.3.12. *Seja $\mathcal{G}$ um mergulho celular ζ – regular sobre a superfície S^2 , com pelo menos três faces, tal que o tamanho da maior face seja δ . Se o mergulho tem duas faces de tamanho δ adjacentes, então o mergulho $\mathcal{G}$ é reconstrutível.*

Como já vimos neste trabalho, mais especificamente na Seção 4.1.2, para provar a reconstrutibilidade abstrata por arestas de uma família de grafos, Zhao em seu trabalho "On the Edge Reconstruction of Graphs Embedded on Surfaces" fez restrições sobre a valência mínima de grafo que pode ser mergulhado sobre uma superfície de gênero $g > 0$ e característica de Euler $-\alpha|V(G)| + \beta$, com $\beta \in \{0, 1\}$, onde o grau mínimo de um vértice do grafo é $6\delta + 5$, sendo α um inteiro não negativo e a representatividade (apresentada na Definição 4.1.2) pelo menos 3. O autor aproveitou as características topológicas dos grafos que satisfazem essas condições para provar a reconstrutibilidade do objeto combinatório. Mas uma pergunta que podemos fazer é, "Dado um mergulho $\mathcal{G}$ de um grafo G , sobre uma superfície orientada e fechada, ambos satisfazendo essas condições, esse mergulho é reconstrutível?" A resposta é sim! Com essas condições, aproveitando da reconstrutibilidade abstrata do grafo e lemas provados pelo autor no trabalho citado, podemos provar o seguinte teorema:

Teorema 4.3.13. *Seja $\mathcal{G}$ o mergulho em M_g de um grafo G , 3-conexo, de valência mínima $6\alpha + 5$, com α um inteiro não negativo. Considere que a superfície M_g é tal que $\chi(M_g) = -\alpha|V(G)| + \beta$, com $\beta \in \{0, 1\}$. Se a representatividade de $\mathcal{G}$ é no mínimo 3 e G não é bidegreed ou regular, então o mergulho é reconstrutível.*

Demonstração. Considere $\mathcal{G}$ o mergulho em M_g de um grafo G , 3-conexo, com valência mínima $\zeta = 6\alpha + 5$; $\zeta \in \mathbb{N}$. Além disso, considere que M_g satisfaz as hipóteses. Neste caso, em Zhao, Y. [40], no Lema 4.1.7 o autor demonstra que em um grafo satisfazendo as condições dadas existe pelo menos um ζ -vértice que é incidente a ζ (quantidade Zeta de) 3-faces que são 2-celulares em M_g . Além disso, para demonstrar esse resultado ele demonstra que esse vértice existente tem uma vizinhança do tipo *wheel-neighborhood*, pelo fato de G ser 3-conexo e ter representatividade pelo menos 3, ou seja, as ζ 3-faces as quais o ζ -vértice é incidente são adjacentes. Agora note que, como o grafo é reconstrutível abstratamente, sabemos exatamente quantos ζ -vértices estão em uma vizinhança *wheel-neighborhood*. Portanto, se tomarmos o elemento do deck de arestas do mergulho tal que é retirada uma aresta incidente ao ζ -vértice, que está em uma vizinhança desse tipo, existe uma única forma de recolocar a aresta faltante, visto que não existe vértice de grau $\zeta - 1$ e este é incidente a uma 4-face, sendo que o grafo é simples. Logo, $\mathcal{G}$ é reconstrutível como mergulho. $\square$

Capítulo 5

Reconstrução de mergulhos de árvores

5.1 Reconstrução de mergulhos de árvores a partir das suas subárvores maximais

O objetivo dessa seção é verificar sob quais condições uma árvore T , mergulhada em uma superfície S , pode ser reconstruída a partir de suas subárvores maximais. Essa seção será uma generalização do artigo de *The reconstruction of a tree from its maximal subtrees* [16], onde os autores mostram que toda árvore (abstrata) pode ser reconstruída a partir de suas subárvores maximais. Como o mergulho de uma árvore, a menos de retrações e rotações, não se modifica ao modificarmos a superfície, a partir de agora, a menos que se diga o contrário, $\mathcal{T}$ representará um mergulho de uma T no disco D . Isso é suficiente para representar o mergulho em qualquer superfície que se queira.

As seguintes definições seguem a apresentação de West [38]. Um grafo T é dito uma *árvore* se for conexo e livre de ciclos. Dado um vértice $u_i \in V(T)$ de uma árvore T , o subgrafo $T_i = T - u_i$ é chamado de *subárvore maximal* se for conexo, ou seja, se a remoção de u_i não desconectar o grafo. Então se um subgrafo $T - u$ de uma árvore T é conexo, necessariamente u é um vértice de grau 1. Para quaisquer vértices $u, v \in V(T)$, a *distância* $d_T(u, v)$, ou simplesmente $d(u, v)$, é definida como o menor comprimento de um caminho ligando u a v . Como em uma árvore qualquer par de vértices está conectado por exatamente um caminho, essa distância é sempre finita. O *diâmetro* de T , denotado por $\text{diam}(T)$, é o máximo das distâncias entre todos os pares de vértices do grafo. O *raio* de T , denotado por $\text{rad}(T)$, é o menor valor de excentricidade (Definição 2.2.7) entre todos os vértices do grafo. Por fim, o *centro* de uma árvore é o subgrafo induzido pelos vértices que possuem excentricidade mínima.

Definição 5.1.1. Uma árvore é dita centrada se o centro é um único vértice e bicentrada caso o centro seja K_2

Em Koenig D. [23] (p. 84) o autor mostra que uma árvore ou é centrada ou bicentrada.

Os Teoremas 5.1.2, 5.1.4 e Corolários 5.1.3, 5.1.5, 5.1.6 enunciados na sequência foram enunciados e demonstrados em Harary F. and Palmer E. [16]. Observe que, quando estamos tratando de mergulhos de árvores, podemos naturalmente usar os mesmos resultados.

Teorema 5.1.2 ([16]). *Se uma árvore T é centrada, então no máximo duas árvores T_i são bicentradas.*

Corolário 5.1.3 ([16]). *Se pelo menos três das subárvores T_i , de uma árvore T , são bicentradas, então T é bicentrada.*

Teorema 5.1.4 ([16]). *Se T é bicentrada, então no máximo duas de suas subárvores T_i são centradas.*

Corolário 5.1.5 ([16]). *Se pelo menos três subárvores T_i de uma árvore T são centradas, então T é centrada.*

Corolário 5.1.6 ([16]). *Se uma árvore T tem no mínimo 5 vértices de grau 1 (end points), então T é centrada ou bicentrada, de acordo com a maioria das T_i se são centradas ou bicentradas, respectivamente.*

O Teorema 5.1.15 é o primeiro onde temos diferenças entre o caso de Harary e Palmer [16] (caso abstrato), para o caso de mergulhos. No caso dos mergulhos, temos mais exemplos de mergulhos que não são reconstrutíveis, a partir das subárvores maximais.

Para demonstrar o referido teorema, vamos demonstrar uma sequência de lemas, que tornarão natural a demonstração do Teorema 5.1.15. Para isso, somente para esses lemas, vamos fazer a seguinte convenção, a qual vamos chamar de representação contraída da árvore. Representaremos um mergulho de uma árvore com 3 ou 4 folhas pela contração dos vértices de grau 2 e rotulagem da árvore resultante com o tamanho do caminho original. Isto é, se $u_0, \dots, u_k \in \mathcal{T}$ é um caminho maximal tal que todos os vértices internos são de grau 2, então iremos substituí-lo por uma única aresta $u_0 u_k$ com peso k . Além disso, representaremos os três tipos de árvores contraídas mostradas na Figura 5.1 simbolicamente como (abc) , $(abcd)$ e $(ab|e|cd)$, para os casos em que a árvore tem único vértice de grau 3, único vértice de grau 4 e exatamente dois vértices de grau 3, respectivamente. Observe que a correspondência entre as árvores mergulhadas com no máximo 4

folhas e suas representações contraídas é de um para um. Observe que " a, b, c " em (abc) são escritos no sentido horário, assim como " a, b, c, d " em $(abcd)$. Já para " e, a, b " e " e, c, d " em $(a, b|e|c, d)$ temos " e, a, b " no sentido anti-horário e " e, c, d " no sentido horário.

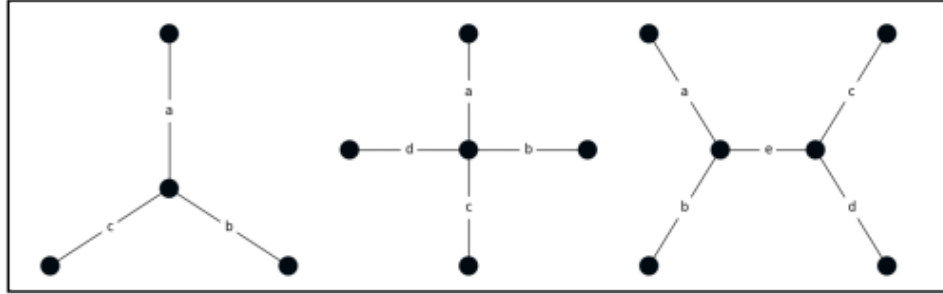


Figura 5.1: Representações contraídas gerais.

Exemplo 5.1.7. Na Figura 5.2, é possível observar a representação contraída de uma árvore com um caminho de tamanho 3 e dois caminhos de tamanho 2. Forma essa que será utilizada nos lemas a seguir, que antecedem o Teorema 5.1.15.

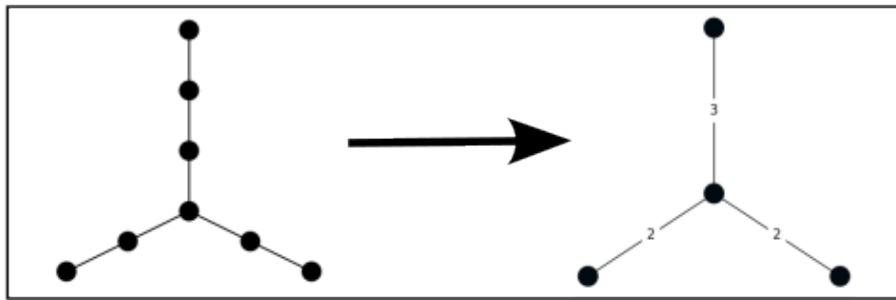


Figura 5.2: Representação contraída de uma árvore.

Além disso, para uma árvore mergulhada na esfera, vamos denotar, somente neste exemplo, por $\mathcal{D}'_{|(1)}(\mathcal{T})$ o seguinte multiconjunto:

$$\mathcal{D}'_{|(1)}(\mathcal{T}) = \{\mathcal{T} - v; v \in V(\mathcal{T}) \text{ e } \deg_{\mathcal{T}}(v) = 1\}.$$

Dessa forma, podemos representar $\mathcal{D}'_{|(1)}(\mathcal{T})$, usando a forma contraída da árvore, vide o Exemplo 5.1.8

Exemplo 5.1.8. Usando o mergulho da Figura 5.2, chamando-a de $\mathcal{T}$, note que o multiconjunto $\mathcal{D}'_{|(1)}(\mathcal{T})$ desse mergulho está representado na Figura 5.3:

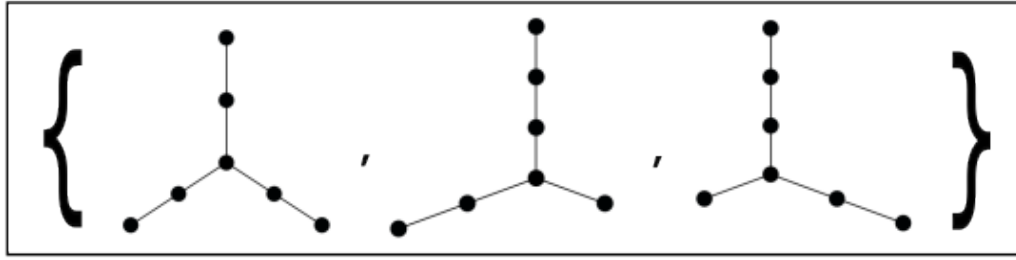


Figura 5.3: Multiconjunto $\mathcal{D}'_{(1)}(\mathcal{T})$.

Usando a forma contraída da árvore, esse multiconjunto pode ser representado como na Figura 5.4.

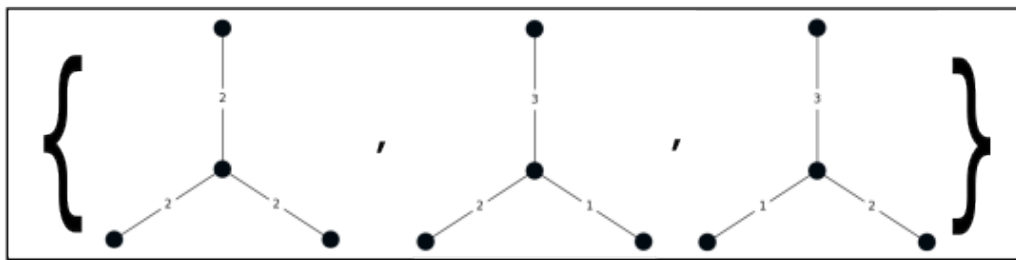


Figura 5.4: Multiconjunto $\mathcal{D}'_{(1)}(\mathcal{T})$ na forma contraída.

Por fim, por simplicidade, observe que o multiconjunto $\mathcal{D}'_{(1)}(\mathcal{T})$, quando escrito na forma contraída das subárvores que o compõem, pode ser reescrito em termos de permutações cíclicas $\{(221), (312), (321)\}$. Essa forma será útil nas demonstrações dos Lemas que antecedem o Teorema 5.1.15. Note que, essa representação em permutação cíclica só pode ser usada em árvores que contém exatamente um vértice de grau maior que 2, isso é, quando a árvore é subdivisão de um star graph $K_{1,m}$, com $m > 2$. No caso em que existem exatamente dois vértices de grau maior que 3, vamos representar por $(C_1|e|C_2)$, onde C_1 e C_2 são permutações cíclicas e e é o tamanho do caminho central.

Lema 5.1.9. *Um mergulho $\mathcal{T}$ de uma árvore com exatamente dois vértices de grau 1 (um caminho) é reconstrutível como mergulho.*

Demonstração. Neste caso, a reconstrução é possível, visto que a menos de homeomorfismos, o mergulho é único. $\square$

Agora vamos demonstrar um resultado auxiliar, à saber 5.1.10 que é um fato comum aos Lemas 5.1.11, 5.1.12, 5.1.14 e assim irão diminuir consideravelmente o tamanho e a complexidade da demonstração dos mesmos.

Lema 5.1.10. *Seja $\mathcal{T}$ um mergulho de uma árvore T com 3 ou 4 folhas, cuja representação contraída é dada por um dos formatos $(a\ b\ c)$, $(a\ b\ c\ d)$ ou $(a\ b\ |e|\ c\ d)$. Suponha que um dos pesos, a , b , c ou d , presentes na forma contraída seja $k > 1$ e que o valor $k - 1$ não ocorra como peso em nenhuma aresta da representação. Então o mergulho $\mathcal{T}$ é reconstrutível a partir do seu deck de submergulhos obtidos pela remoção de folhas.*

Demonstração. Sabemos que, do ponto de vista abstrato (isto é, considerando apenas a estrutura da árvore sem o mergulho), a árvore T é reconstrutível a partir de suas subárvores maximais, conforme garantido por Harary e Palmer [16]. Isso nos fornece a estrutura combinatória de T , incluindo os comprimentos dos caminhos que partem dos centros (ou do centro) da árvore. Em particular, sabemos exatamente quais são os pesos que devem aparecer na forma contraída correspondente ao mergulho original $\mathcal{T}$.

Suponha agora que algum dos pesos esperados seja $k > 1$, mas que na representação contraída (derivada da coleção de submergulhos) o que aparece seja $k - 1$. Isso indica que houve, em algum dos submergulhos, a contração de um caminho com comprimento incorreto — ou seja, há um elemento no deck cuja forma contraída sugere a existência de uma aresta de peso $k - 1$, a qual sabemos que não deveria existir, dado o conhecimento da árvore original.

Como o valor $k - 1$ não pode corresponder a nenhum dos caminhos da árvore original (pela suposição do lema), a única explicação possível é que esse valor foi reduzido indevidamente durante a formação da representação contraída. Assim, podemos corrigir esse erro restaurando o valor esperado k na representação contraída. Como a correspondência entre mergulhos de árvores com até 4 folhas e suas representações contraídas é bijetiva (como estabelecido na convenção), isso implica que o mergulho original $\mathcal{T}$ é unicamente determinado — isto é, reconstrutível — a partir do deck do mergulho. $\square$

A partir de agora, denotaremos por $\Delta(T)$ o maior grau de uma árvore T e por r o raio.

Lema 5.1.11. *Seja $\mathcal{T}$ um mergulho de uma árvore T com $\Delta(T) = 3$ e um único vértice de grau 3. O mergulho é reconstrutível, exceto nos casos exibidos na Figura 5.5.*

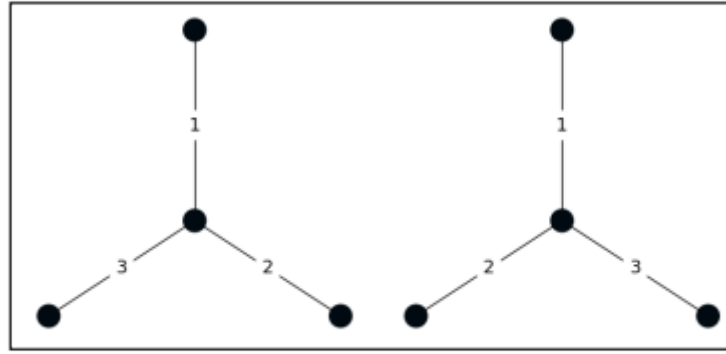


Figura 5.5: Representação contraída de mergulhos não reconsntrutíveis com um centro e três caminhos conectados a ele.

Demonstração. Usando o Lema 5.1.10, precisamos trabalhar apenas com a, b, c satisfazendo $1 = a \leq b \leq c$, tal que $b - a \leq 1$ e $c - b \leq 1$, pois os outros casos já provamos ser reconstrutíveis. Agora, exclusivamente para o caso da árvore mergulhada ter exatamente um vértice de grau 3, temos que se dois pesos são repetidos, então o mergulho é reconstrutível, pois o mergulho de T é único. Isso se dá pelo fato do mergulho ser representado por uma permutação cíclica de tamanho 3, (abc) e da árvore abstrata ser reconstrutível. Portanto, nos resta trabalhar somente com $1 = a < b < c$, tal que $b - a < 1$ e $c - b < 1$, isto é, $a = 1, b = 2$ e $c = 3$. E para os dois mergulhos possíveis com esses elementos, é fácil ver que ambos tem o mesmo deck, à saber elementos da Figura 5.6, mas os mergulhos não são equivalentes (vide Figura 5.5). Sendo esses os únicos casos que o mergulho não é reconstrutível. $\square$

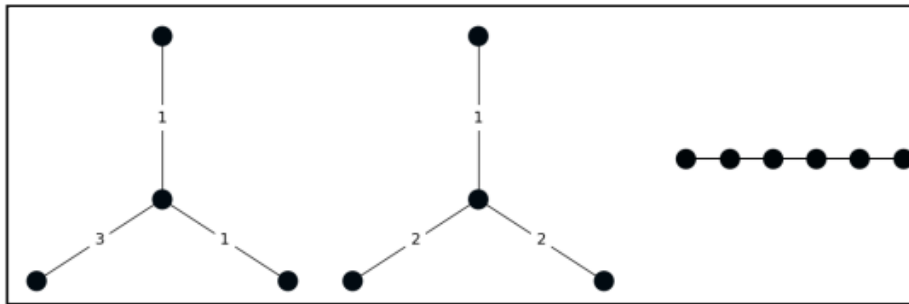


Figura 5.6: Elementos do deck de mergulhos não equivalentes.

Lema 5.1.12. *Seja $\mathcal{T}$ um mergulho de uma árvore T com $\Delta(T) = 4$, um único vértice de grau 4 e nenhum vértice de grau 3. O mergulho é reconstrutível, exceto nos casos exibidos na Figura 5.7.*

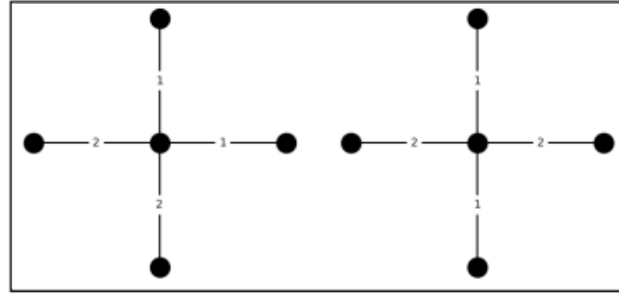


Figura 5.7: Representação contraída de uma árvore com um centro e quatro caminhos conectados a ele.

Demonstração. Usando o Lema 5.1.10, precisamos trabalhar apenas com a, b, c, d satisfazendo $1 = a \leq b \leq c \leq d$, tal que $b - a \leq 1$, $c - b \leq 1$ e $d - c \leq 1$, pois os demais casos já provamos ser reconstrutíveis. Exclusivamente para o caso em questão, se três pesos são repetidos, então o mergulho é reconstrutível. Isso se dá pelo fato do mergulho ser representado por uma permutação cíclica de tamanho 4, $(abcd)$ e da árvore abstrata ser reconstrutível. Portanto, nos resta verificar os casos em que os pesos tem no máximo duas repetições. Para esses casos, observe o grafo da Figura 5.8. Nele é possível observar na parte superior um representante de cada classe das permutações que cabem nas condições restantes e na parte inferior as sequências obtidas por substituição de elementos, isto é, elementos do deck. E podemos observar, na comparação dois a dois, que as únicas permutações que compartilham todos os elementos do deck são (1122) e (1212) . Todos os demais possuem em seu deck, pelo menos um elemento que os distingue.

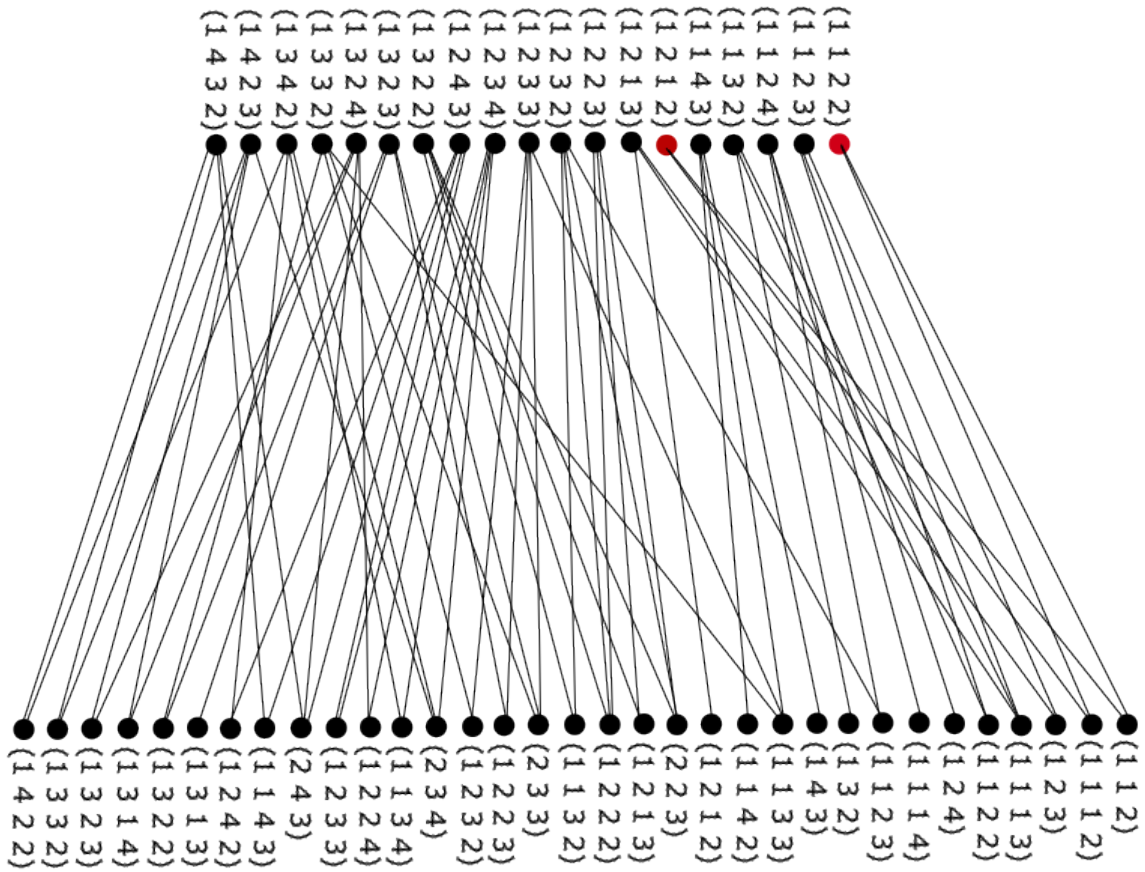


Figura 5.8: Classes de permutações de tamanho 4 conectadas aos elementos do deck.

□

Observação 5.1.13. Note que na Figura 5.8, uma vez que temos a sequência que representa a árvore, é possível saber a multiplicidade de cada elemento do seu deck. Dessa forma, a ilustração é suficiente para verificar a reconstrução, pois vemos os elementos não reconstrutíveis, além de compartilharem todos os elementos do deck, se igualam também pelas multiplicidades desses elementos. Já os elementos reconstrutíveis, não dependem da multiplicidade para a reconstrução, pois tem pelo menos um elemento que os distinguem.

Lema 5.1.14. Seja $\mathcal{T}$ um mergulho de uma árvore T com $\Delta(T) = 3$ e exatamente dois vértices de grau 3. O mergulho é reconstrutível, exceto nos casos exibidos na Figura 5.9

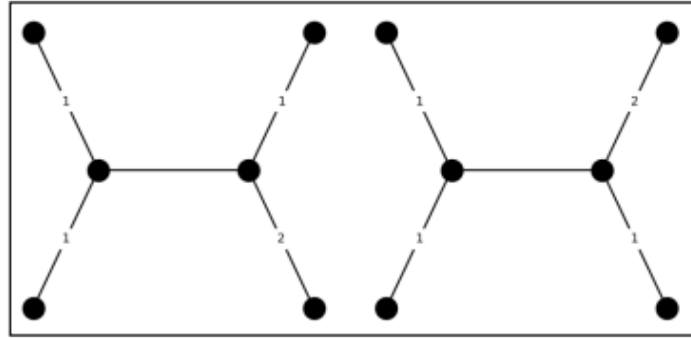


Figura 5.9: Representação contraída de mergulhos não reconstrutíveis com exatamente dois vértices de grau 3 e dois caminhos conectados a cada um deles.

Demonstração. Usando o Lema 5.1.10, precisamos trabalhar apenas com a, b, c, d satisfazendo $1 = a \leq b \leq c \leq d$, tal que $b - a \leq 1$, $c - b \leq 1$ e $d - c \leq 1$, pois os demais casos já provamos ser reconstrutíveis. Para os casos restantes, fizemos uma busca exaustiva sobre os mergulhos possíveis e seus decks, os detalhes estão exibidos na Seção 6.1, do Apêndice A. De todos os mergulhos testados, os únicos não reconstrutíveis foram os exibidos na Figura 5.9. Onde os mergulhos não são equivalentes, mas os seus decks são iguais. Vale notar que se $e > 1$, isto é, se a aresta central for um caminho de tamanho $e > 1$, então existe pelo menos uma subárvore que distingue as árvores originais - similares à Figura 5.9, incluindo um caminho no centro, à saber $(2, 1, 1 + e)$ e $(1, 2, 1 + e)$, vide Árvore 1 da Figura 6.1 e Árvore 2 da Figura 6.2. $\square$

Os Lemas 5.1.11, 5.1.12, 5.1.14 implicam o Teorema 5.1.15. A partir de agora, vamos usar a notação $\mathcal{D}_M(\mathcal{T}) = \{\mathcal{T} - u \mid \mathcal{T} \text{ é conexo}\}$.

Teorema 5.1.15. *Se $\mathcal{T}$ tem menos de 5 end points, então pode ser determinada unicamente, a partir de $\mathcal{D}_M(\mathcal{T})$, exceto para os casos exibidos na Figura 5.5, Figura 5.7 e Figura 5.9.*

A partir de agora, o nosso esforço será no sentido de demonstrar o Teorema 5.1.21, onde afirmamos que mergulhos de árvores com pelo menos 5 folhas são reconstrutíveis a partir dos mergulhos das subárvores maximais, caso não possuam um branch homeomorfo a um caminho de tamanho 1. Agora para as árvores com 5 folhas ou mais, temos duas possibilidades T é centrada (Teorema 5.1.19) ou T é bicentrada (5.1.20).

Vamos demonstrar cada um dos teoremas citados, para a demonstração do Teorema 5.1.21.

Antes de demonstrarmos os Teoremas 5.1.19 e Teorema 5.1.20, precisamos demonstrar os Lemas 5.1.16, 5.1.17 e 5.1.18. A partir de agora, seja $\mathcal{T}$ um mergulho de uma árvore T com pelo menos 5 folhas. Por Harary e Palmer [16], sabemos que

as árvores abstratas são reconstrutíveis e já vimos que o deck do mergulho, implica no deck abstrato, portanto é possível identificar se a árvore é centrada ou bicentrada e o(s) grau(s) do(s) centro(s). Vamos usar a notação (B_i, c) quando quisermos nos referir a um branch qualquer conectado ao centro c , sem nos apegarmos a quaisquer outras características desse branch.

Lema 5.1.16. *Seja $\mathcal{T}$ um mergulho de uma árvore T centrada em c , com grau de c igual a 2 e pelo menos 5 folhas. Sejam (B_1, c) e (B_2, c) os dois branches de $\mathcal{T}$ em c , onde $|V(B_1)| = p_1$ e $|V(B_2)| = p_2$. Então $\mathcal{T}$ é reconstrutível a partir de $\mathcal{D}_M(\mathcal{T})$.*

Demonstração. Como T é centrada, no máximo duas das subárvores obtidas pela remoção de uma única folha (denotadas por $\mathcal{T}_i$) são bicentradas (Teorema 5.1.2). Sem perda de generalidade, suponha que, se houver bicentradas, sejam $\mathcal{T}_1$ e $\mathcal{T}_2$.

Caso 1. Para cada $i \geq 3$, $\mathcal{T}_i$ tem um dos branches em c que é um caminho (isto é, com exatamente um vértice de grau 1).

Nesse caso, seja r o raio de $\mathcal{T}$. Como há pelo menos cinco folhas, um dos branches em c , digamos (B_1, c) , deve ser um caminho de comprimento r , garantindo que pelo menos uma $\mathcal{T}_i$ seja bicentrada. Defina em cada $\mathcal{T}_i$ (para $i \geq 3$) o ponto w_i de grau > 2 mais próximo de c , e seja $d = \min_{i \geq 3} d(c, w_i)$. Então $\mathcal{T}_1$ foi obtida apagando justamente a folha em (B_1, c) , e o outro branch de T em c coincide com o branch de $\mathcal{T}_1$ em c_1 que contém c_2 . Desse modo, identificamos (B_1, c) e (B_2, c) e reconstruímos $\mathcal{T}$.

Caso 2. Para algum $i \geq 3$, $\mathcal{T}_i$ tem mais de um vértice de grau 1 em cada um dos dois branches em c (nenhum é caminho).

Denote os dois branches de $\mathcal{T}_i$ por (B_1^i, c) e (B_2^i, c) . Pelo ajuste de notação, temos

$$|V(B_1)| = p_1, \quad |V(B_2)| = p_2,$$

e em cada $\mathcal{T}_i$ os tamanhos satisfazem

$$\{|V(B_1^i)|, |V(B_2^i)|\} = \{p_1 - 1, p_2\} \quad \text{ou} \quad \{p_1, p_2 - 1\},$$

$$p_1 \geq p_2.$$

Caso 2.1: Se $p_1 > p_2 + 1$, então qualquer branch de $\mathcal{T}_i$ com p_1 vértices é necessariamente um branch de $\mathcal{T}$. O outro branch de $\mathcal{T}$ em c é um branch de algum $\mathcal{T}_j$ com p_2 vértices em c .

Caso 2.2: Se $p_1 < p_2 + 1$, então, como $p_1 \geq p_2$, segue $p_1 = p_2$. Assim, em cada $\mathcal{T}_i$ podemos assumir $|V(B_1^i)| = p_1$ e $|V(B_2^i)| = p_1 - 1$. Se todos os (B_1^i, c) são homeomorfos, cada branch de $\mathcal{T}$ em c é igual a (B_1^i, c) . Caso haja i, j tais que

(B_1^i, c) e (B_1^j, c) não são homeomorfos, essas duas escolhas fornecem exatamente os branches de $\mathcal{T}$.

Caso 2.3: Se $p_1 = p_2 + 1$, então em algum $\mathcal{T}_i$ temos $|V(B_1^i)| = p_1$ e $|V(B_2^i)| = p_2 - 1$.

Caso 2.3.1: Se alguma $\mathcal{T}_i$ é bicentrada, proceda como no Caso 1 para identificar qual folha foi deletada no branch maior, recuperando (B_1, c) e (B_2, c) .

Caso 2.3.2: Se nenhuma $\mathcal{T}_i$ é bicentrada, basta observar as classes de homeomorfismo dos branches de tamanho p_2 :

- Se houver apenas uma classe, qualquer representante é (B_2, c) .
- Se houver duas classes, exatamente uma delas será um menor de (B_1, c) , e a outra definirá (B_2, c) .
- Se houver mais de duas classes, (B_2, c) é o branch de tamanho p_2 que aparece com maior frequência em $\mathcal{D}_M(\mathcal{T})$.

Em todos os casos, recuperamos (B_1, c) e (B_2, c) , concluindo que $\mathcal{T}$ é reconstrutível a partir de seu deck. $\square$

Seja $\mathcal{T}$ um mergulho de T centrada, com S branches. A partir de agora, vamos adotar a seguinte notação:

- P_r : Branch que é homeomorfo a um caminho de tamanho r .
- B_i , $i = 1, \dots, q$: Todos os possíveis branches, distintos por homeomorfismo, de um mergulho de T , que não são caminhos.
- $\sim$: Homeomorfismo.
- $\simeq$: Isomorfismo, para árvores abstratas
- A_i : Árvore abstrata de B_i . Note que podemos ter $A_i \simeq A_j$, mas $B_i \not\sim B_j$.
- $m(B_i, \mathcal{T})$: O número de branches de $\mathcal{T}$ homeomorfos a B_i .
- α_{ij} : O número de vértices $u \in B_i$ tais que $B_i - u \simeq B_j$ e a remoção de u não altera o centro.
- s_i : O número de vértices $u \in B_i$ tais que $\mathcal{T} - u$ tem o mesmo centro de $\mathcal{T}$. Se $s_i = 0$, então $B_i \simeq P_r$ e existe apenas outro branch que contém vértices à distância r do centro.
- $\mathcal{D}_1(\mathcal{T}) = \{\mathcal{T} - u \in \mathcal{D}_M(\mathcal{T}) \mid c(\mathcal{T} - u) = c(\mathcal{T})\}$, onde $c(\mathcal{T})$ denota o centro do mergulho.

- p_i : O número de vértices de B_i (exceto o centro).

Lema 5.1.17 (Lema da contagem). *Se $\mathcal{T}$ é uma árvore centrada, então é possível reconstruir $m(B_i, \mathcal{T})$.*

Demonstração. A árvore centrada T , pode ter branches que são caminhos $P_1, \dots, P_r$, onde r é o raio da árvore. Neste caso, já sabemos a multiplicidade desses caminhos, pela própria árvore abstrata, que é reconstrutível, em particular, a partir do deck do mergulho. Portanto, vamos tomar B_i como definido na notação anterior. Suponha que

$$B_1, B_2, \dots, B_L$$

são ordenadas tais que:

- $p_i \geq p_j$, se $i \leq j$.
- $p_1 = P_i$, $\forall i \leq K$ e $p_i < p_1$, $\forall i > K$,

onde, sem perda de generalidade, K é o número de branches da lista que tem o mesmo tamanho que B_1 , podendo ou não ter sido gerados pelo mesmo A_1 que gerou B_1 . Note que vamos tomar $K > 0$, visto que se todo branch de T é caminho, já sabemos as suas multiplicidades pela árvore abstrata. Por simplicidade, vamos denotar $m_i = m(B_i, \mathcal{T})$.

Caso 1: Contando m_i para $1 \leq i \leq K$:

Vamos separar esse caso em duas partes.

Parte 1: Nenhum branch de T é isomorfo a B_i e nem a P_r .

Seja A um branch de T , então A contém um vértice v tal que $\deg(v) = 1$ e $c(\mathcal{T} - v) = c(\mathcal{T})$ e $m(A_i, \mathcal{T} - v) = m_i$.

Por outro lado, B_i contém um vértice u tal que $c(\mathcal{T} - u) = c(\mathcal{T})$ e $\deg(u) = 1$. Portanto, $m_i = \max_v (m(B_i, \mathcal{T} - v))$.

Parte 2: Se todo branch de T é isomorfo a A_i ou a P_r .

Seja A um branch de T , se A tem mais de dois mergulhos observados em $\mathcal{D}_1$, digamos $B_1, \dots, B_q$, então:

$$m_i = \max\{m(B_i, \mathcal{T} - v) \mid \mathcal{T} - v \in \mathcal{D}_1(\mathcal{T}), \forall i \ 1 \leq i \leq q\}.$$

Sejam l e t , respectivamente, o número de cópias de A em T e o número de caminhos de tamanho r em T . Se exatamente um mergulho de A é visto em $\mathcal{D}_1(\mathcal{T})$, digamos B_1 , então temos duas opções.

Se $|\mathcal{D}_1(\mathcal{T})| = ls_1$, então $m_1 = m(B_1, \mathcal{T} - v) + 1$, para qualquer $\mathcal{T} - v$ em

$\mathcal{D}_1(\mathcal{T})$. Porque todo $\mathcal{T} - v$ em $\mathcal{D}_1(\mathcal{T})$ é obtido pela deleção de um vértice v de uma cópia de B_1 e $l > 1$.

Se $|\mathcal{D}_1(\mathcal{T})| > ls_1$, então isso significa que a remoção de uma folha de um caminho de tamanho r também não altera o centro. Portanto, $|\mathcal{D}_1(\mathcal{T})| = ls_1 + t$. Dessa forma, tome $\mathcal{T} - v \in \mathcal{D}_1(\mathcal{T})$ tal que essa subárvore tenha l cópias de B_1 , e conecte um vértice a uma cópia de P_{r-1} em $\mathcal{T} - v$. Reconstruindo assim o mergulho $\mathcal{T}$ e consequentemente, contando as multiplicidades desejadas.

Por fim, caso nenhum mergulho de A é visto em $\mathcal{D}_1(\mathcal{T})$, isso implica que $l = 1$ e a deleção de uma folha de P_r altera o centro e $|\mathcal{D}_1(\mathcal{T})| = s_1$ e $t = 2$.

Neste caso, temos dois caminhos de tamanho r e sabemos que B_1 tem mais de uma folha. Portanto, basta observar em uma das árvores bicentradas, que em um centro, digamos c_1 , temos um caminho de tamanho $r - 1$ e um outro branch e em c_2 , temos apenas um caminho de tamanho $r - 1$. Isso significa que o branch que acompanha o caminho de tamanho $r - 1$, em c_1 é B_1 .

Caso 2: Contando m_i para $i > K$ e $K \neq 0$.

Iremos dividir esse caso em duas partes.

Parte 1: Se existe $j < i$, tal que $\alpha_{ji} \neq 0$:

Neste caso, temos que $m_i = \max\{m(B_i, \mathcal{T} - v)\} - 1$, pois existe v em B_j tal que $\mathcal{T} - v$ contém $m_i + 1$ cópias de B_i .

Parte 2: Para todo $j < i$, tem-se que $\alpha_{ji} = 0$:

Para todo $j < i$, temos que $s_j \neq 0$. Portanto,

$$m_i = \max_{\mathcal{T} - v \in \mathcal{D}_1(\mathcal{T})} \{m(B, \mathcal{T} - v)\}. \quad \square$$

Lema 5.1.18. *Sejam $B_1, \dots, B_L$ os branches distintos de $\mathcal{T}$, com multiplicidades $m(B_i, \mathcal{T}) = m_i$, agora com B_i podendo ser caminhos também. Se para algum B_i , existe um vértice u tal que $B_i - u \not\subseteq B_j \forall j$ e a retirada de u não altera o centro de $\mathcal{T}$, então $\mathcal{T}$ é reconstrutível.*

Demonstração. Em $\mathcal{T} - u$ existe um único branch B que não está na lista $B_1, \dots, B_L$ e existe um único B_i tal que $m(B_i, \mathcal{T} - u) = m_i - 1$. Portanto, B deve ser substituído por B_i para obter $\mathcal{T}$. $\square$

Teorema 5.1.19. *Seja $\mathcal{T}$ um mergulho de uma árvore T centrada, que não contém branches triviais, com pelo menos 5 folhas. O mergulho $\mathcal{T}$ pode ser reconstruído a partir de $\mathcal{D}_M(\mathcal{T})$.*

Demonstração. O caso em que o grau do centro c é 2 foi demonstrado no Lema 5.1.16 e para o caso em que o grau do centro c é maior que 2, sem branches triviais, vale notar que uma consequência direta do Lema 5.1.18 é que se uma árvore centrada não contém vértices de grau 1 conectados ao centro, isto é, branches triviais, então ela pode ser reconstruída a partir de suas subárvores maximais. Portanto, o segundo caso segue diretamente do Lema 5.1.18. $\square$

Teorema 5.1.20. *Seja $\mathcal{T}$ um mergulho de uma árvore T bicentrada, com pelo menos 5 folhas. O mergulho $\mathcal{T}$ pode ser reconstruído a partir de $\mathcal{D}_M(\mathcal{T})$.*

Demonstração. Vamos denotar por c_1 e c_2 os centros de T , n_1 o número total de folhas, que não mudam os centros, nos branches conectados à c_1 , n_2 o número total de folhas, que não mudam os centros, dos branches conectados à c_2 . Por fim, denote por X_1 o número total de vértices dos branches conectados à c_1 e por X_2 o número total de vértices dos branches conectados à c_2 .

Como a árvore abstrata T é reconstrutível, sabemos n_1, n_2, X_1 e X_2 . Para fazer a demonstração, vamos separar em dois casos. Suponhamos, sem perda de generalidade que $n_1 \geq n_2$.

Caso 1. Se $n_1 > n_2 + 1$:

Então ao observar o multiconjunto das subárvores maximais $\mathcal{T}_i$, restrito à retirada das folhas que não mudam os centros, c_2 e a sua sequência de branches irão aparecer n_1 vezes. Enquanto c_1 e a sua sequência de branches irão aparecer n_2 vezes. Assim, é possível reconstruir o mergulho $\mathcal{T}$ original.

Caso 2. Se $n_1 = n_2 + 1$:

Então para reconstruir o branch que possui n_1 folhas, é suficiente tomar algum elemento de $\mathcal{D}_M(\mathcal{T})$, restrito aos vértices que não alteram o centro, tal que algum lado da árvore tenha n_1 folhas. Já para o branch com menos folhas, vamos observar os elementos de $\mathcal{D}_M(\mathcal{T})$ tal que os dois lados tenham n_2 folhas.

- Se existe apenas uma classe de homeomorfismo entre as sequências cíclicas de branches conectados aos centros, então a reconstrução é possível.
- Caso existam mais de duas classes de homeomorfismos entre as sequências cíclicas de branches conectados aos centros, então aquela sequência cíclica de branches com n_2 que aparece mais vezes é a original.
- Se existem exatamente duas classes de homeomorfismo, a sequência cíclica original será aquela que não for subárvore da sequência cíclica com n_1 folhas que não alteram o centro.

Caso 3. Se $n_1 = n_2$:

Vamos analisar os casos sobre os números X_1 e X_2 .

Caso 3.1. Se $X_1 > X_2 + 1$:

Neste caso, no multiconjunto $\mathcal{D}_M(\mathcal{T})$, quando restrito aos vértices que não alteram os centros, é possível observar algumas $\mathcal{T}_i$ tais que de um lado tem X_1 vértices e do outro $X_2 - 1$, enquanto em outras ocorrem $X_1 - 1$ e X_2 . Mas pela reconstrutibilidade abstrata, sabemos que não existem conjuntos de branches em c_1 e c_2 que somam $X_1 - 1$ ou $X_2 - 1$ vértices. Portanto, a reconstrução é possível.

Caso 3.2. Se $X_1 = X_2 + 1$:

Existe alguma árvore de $\mathcal{D}_M(\mathcal{T})$, restrito aos vértices que não alteram os centros, em que os branches conectados a um dos centros somam X_1 vértices. Portanto já sabemos um lado do mergulho original.

A fim de descobrir o outro lado, é suficiente considerar todos os elementos de $\mathcal{D}_M(\mathcal{T})$, tais que tanto os branches conectados a um centro, quanto ao outro somam $X_1 - 1 = X_2$ vértices. Se existe só uma classe de homeomorfismo entre esses, sabemos os branches e suas sequências cíclicas que deve entrar do outro lado. Caso exista mais de duas classe de homeomorfismo, aquele que aparecer mais vezes no multiconjunto é o elemento faltante. E por fim, se existem somente duas classes de homeomorfismo, uma delas não é subárvore do lado que soma X_1 vértices e essa é a parte que procuramos.

Caso 3.3. Se $X_1 = X = X_2$:

Todas as árvores de $\mathcal{D}_M(\mathcal{T})$, restrito aos vértices que não alteram os centros, terão X vértices nos branches de um centro e $X - 1$ vértices nos branches do outro. Se em todas as árvores os branches de tamanho X são homeomorfos, então em $\mathcal{T}$ esses branches se repetem conectados aos dois centros. Caso contrário, existem exatamente duas classes de homeomorfismo, entre os branches de tamanho X , e também é possível reconstruir $\mathcal{T}$. $\square$

Tendo demonstrado todos os teoremas deste capítulo, podemos enunciar o teorema central, que fica imediatamente demonstrado.

Teorema 5.1.21. *Se $\mathcal{T}$ é um mergulho de uma árvore T , que não contém branches triviais, com pelo menos 5 folhas, então $\mathcal{T}$ pode ser reconstruída a partir de $\mathcal{D}_M(\mathcal{T})$.*

Uma vez demonstrado o Teorema 5.1.21, para trabalhos futuros ainda nos resta verificar a reconstrutibilidade de árvores centradas, que possuem um vértice de grau 1 acoplados ao centro.

Referências

- [1] CD Albuquerque, R Palazzo, and EB Silva. Topological quantum codes on compact surfaces with genus $g \geq 2$. Journal of mathematical physics, 50(2), 2009.
- [2] Christophe Bavard. Disques extrémaux et surfaces modulaire. In Annales de la Faculté des sciences de Toulouse: Mathématiques, volume 5, pages 191–202, 1996.
- [3] Mark Bilinski, Young Soo Kwon, and Xingxing Yu. On the reconstruction of planar graphs. J. Combin. Theory Ser. B, 97(5):745–756, 2007.
- [4] J. A. Bondy and R. Hemminger. Graph reconstruction-a survey. Journal of Graph Theory, 1:227 – 268, 10 1977.
- [5] J Adrian Bondy and Robert L Hemminger. Graph reconstruction—a survey. Journal of Graph Theory, 1(3):227–268, 1977.
- [6] Rodrigo Gusmao Cavalcante, Henrique Lazari, Joao de Deus Lima, and Reginaldo Palazzo Jr. A new approach to the design of digital communication. In Algebraic Coding Theory and Information Theory: DIMACS Workshop, Algebraic Coding Theory and Information Theory, December 15-18, 2003, Rutgers University, Piscataway, New Jersey, volume 68, page 145. American Mathematical Soc., 2005.
- [7] C Mendes de Jesus and Pantaleón D Romero. Graphs and closed surfaces associated with a pairing of edges for regular polygons. Bulletin of the Brazilian Mathematical Society, New Series, 51(2):527–541, 2020.
- [8] H. C. SILVA e C. MENDES DE JESUS e J. R. P. COELHO. Grafos de emparelhamento de arestas. Trends in Computational and Applied Mathematics, 24, (2):379–395, aceito 2023.
- [9] MB Faria, C Mendes de Jesus, and PDR Sanchez. Surgeries of pairing of edges associated to trivalent graphs. Bulletin of the Brazilian Mathematical Society, New Series, 47(4):1085–1095, 2016.

- [10] MB Faria and R Palazzo Jr. Emparelhamentos generalizados associados à tesselação $\{12g-6, 3\}$. TEMA-Tendências em Matemática Aplicada e Computacional, 11(1):59–67, 2010.
- [11] Stefano Fiorini and Josef Lauri. The reconstruction of maximal planar graphs, i. recognition. J. Combin. Theory Ser. B, 30(2):188–195, 1981.
- [12] Stefano Fiorini and Bernard Manvel. Edge-reconstruction of 4-connected planar graphs. Journal of Graph Theory, 6(1):33–42, 1982.
- [13] Peter A Firby and Cyril F Gardiner. Surface topology. Elsevier, 2001.
- [14] S. K. Gupta, P. Mangal, and V. Paliwal. Some work towards the proof of the reconstruction conjecture. Discrete Mathematics, 272:291–296, 2003.
- [15] F. Harary. On the reconstruction of a graph from a collection of subgraphs. Theory of Graphs and its applications (M. Fiedler,ed), pages 47–52., 1964.
- [16] Frank Harary and Ed Palmer. The reconstruction of a tree from its maximal subtrees. Canadian Journal of Mathematics, 18:803–810, 1966.
- [17] Allen Hatcher. Algebraic topology. , 2005.
- [18] Peter Hoffman and Bruce Richter. Embedding graphs in surfaces. Journal of Combinatorial Theory, Series B, 36(1):65–84, 1984.
- [19] Troels Jørgensen and Marjatta Näätänen. Surfaces of genus 2: generic fundamental polygons. The Quarterly Journal of Mathematics, 33(4):451–461, 1982.
- [20] P. J. Kelly. On isometric transformations. Ph.D. thesis, University of Wisconsin., 1942.
- [21] Paul J. Kelly. A congruence theorem for trees. Pacific J. Math., 7(1):961–968, 1957.
- [22] L Christine Kinsey. Topology of surfaces. Springer Science & Business Media, 1997.
- [23] Dénes König. Theorie der endlichen und unendlichen Graphen, volume 72. American Mathematical Soc., 2001.
- [24] Josef Lauri. Edge-reconstruction of planar graphs with minimum valency 5. Journal of Graph Theory, 3(3):269–286, 1979.
- [25] Josef Lauri. The reconstruction of maximal planar graphs, ii. reconstruction. J. Combin. Theory Ser. B, 30(2):195–214, 1981.

- [26] M Lee John. Introduction to smooth manifolds. Graduate Texts in Mathematics, 218, 2003.
- [27] Elon Lages Lima. Variedades diferenciáveis. Number 15. Instituto Matemática Puro e Aplicada, Conselho Nacional de Pesquisas, 1973.
- [28] Sivaramakrishnan Monikandan. Reconstruction of graphs. Recent Applications in Graph Theory, 2019.
- [29] Lee Mosher. A user's guide to the mapping class group: once punctured surfaces. Proc. of a Comput. Sci, 25:101–174, 1996.
- [30] James R Munkres. Topology; a First Course [By] James R. Munkres. Prentice-Hall, 1974.
- [31] Wendy J Myrvold, Mark N Ellingham, and DG Hoffman. Bidegreed graphs are edge reconstructible. Journal of graph theory, 11(3):281–302, 1987.
- [32] Gou Nakamura. Generic fundamental polygons for surfaces of genus three. Kodai Mathematical Journal, 27(1):88–104, 2004.
- [33] Francielle Alves Rodrigues. Construções de emparelhamentos de arestas de polígonos regulares. 2024.
- [34] Colin P. Rourke and Brian J. Sanderson. Introduction to Piecewise-Linear Topology. Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer-Verlag, 1972.
- [35] Hoechst Cornélio da Silva. Extensão de grafos sobre superfícies orientadas. 2020.
- [36] L. Stockmeyer. The falsity of the reconstruction conjecture for tournaments. Journal of Graph Theory, 1:19–25, 1977.
- [37] Stanislaw M Ulam. A collection of mathematical problems. (No Title), 1960.
- [38] D. B. West. Introduction to Graph Theory. Prentice-Hall, 2nd edition, 2005.
- [39] Yang Yongzhi. The reconstruction conjecture is true if all 2-connected graphs are reconstructible. Journal of Graph Theory, 12(2):237–243, 1988.
- [40] Yue Zhao. On the edge reconstruction of graphs embedded on surfaces. Graphs and Combinatorics, 9(2-4):391–395, 1993.
- [41] Yue Zhao. On the edge reconstruction of graphs embedded in surfaces ii. Journal of the London Mathematical Society, 57(2):268–274, 1998.

- [42] Yue Zhao. On the edge reconstruction of graphs embedded in surfaces, iii. Journal of Combinatorial Theory, Series B, 74(2):302–310, 1998.
- [43] Yue Zhao. On the edge-reconstruction of graphs embedded in surfaces iv. Discrete Mathematics, 263(1-3):331–338, 2003.

Capítulo 6

Apêndice A

6.1 Verificação exaustiva: reconstrução de árvores mergulhadas

Nesta seção apresentamos os detalhes dos testes feitos para a demonstração do Lemma 5.1.14. Para a demonstração referida, todas as árvores testadas estão apresentadas nas Figuras 6.1 a 6.20.

Nos blocos de código Python, usados para gerar essas árvores e os seus decks, os trechos em verde indicam os comentários e dizem exatamente a finalidade de cada trecho. Para gerar todas as possibilidades de árvores que precisávamos, foram usados os Blocos Python 6.1, 6.2 e 6.3. Já para calcular e exibir os decks foram usados os Blocos Python 6.4, 6.5 e 6.6. Para a execução dos blocos referidos, usamos Python 3.12.6 e as seguintes versões das bibliotecas: `itertools` - Nativa, `networkx`==3.4.2, `matplotlib`==3.10.1, `math` - Nativa, `os` - Nativa.

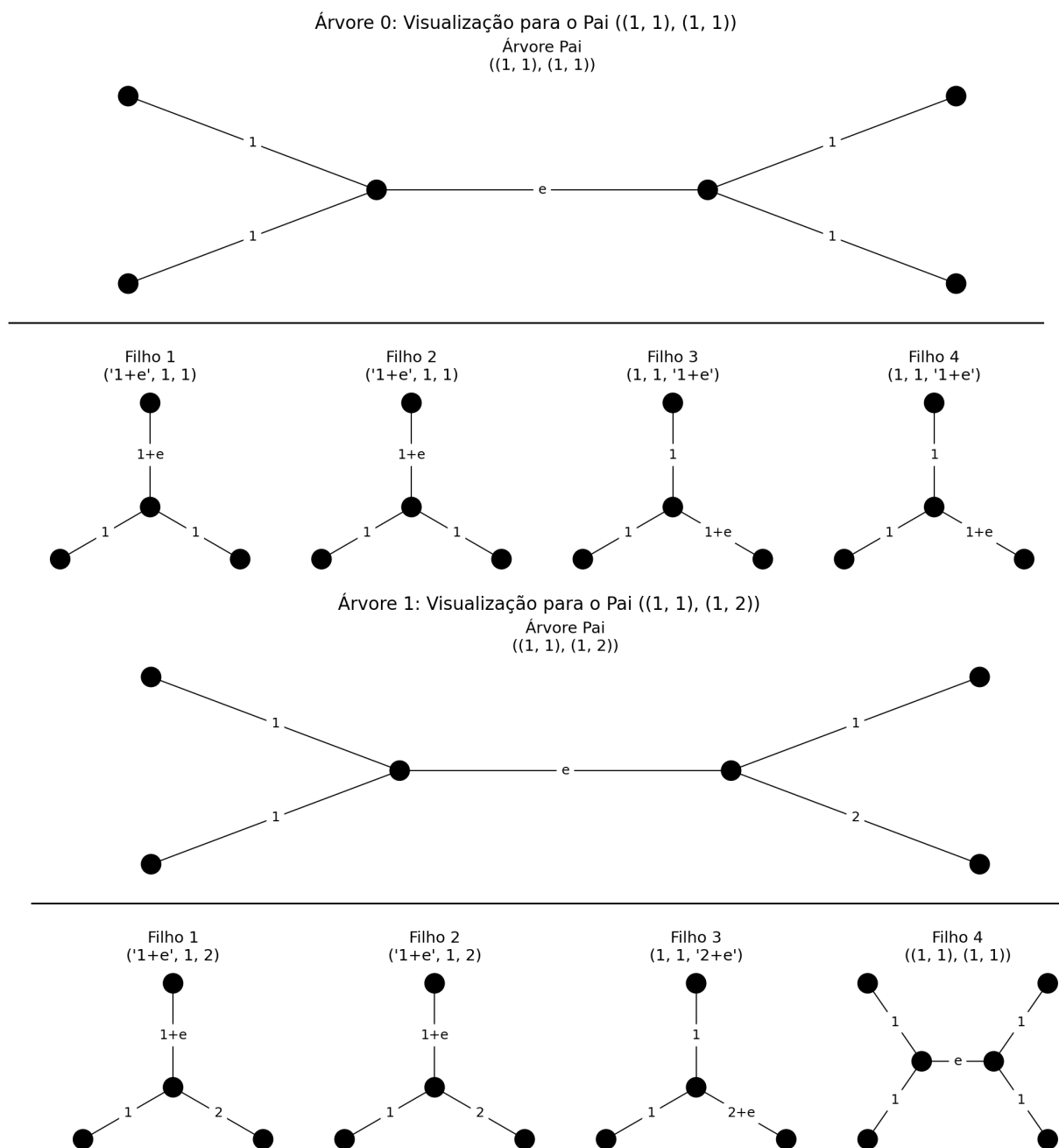


Figura 6.1: Árvores analisadas para o Lema 5.1.14 e os seus decks (1-2)

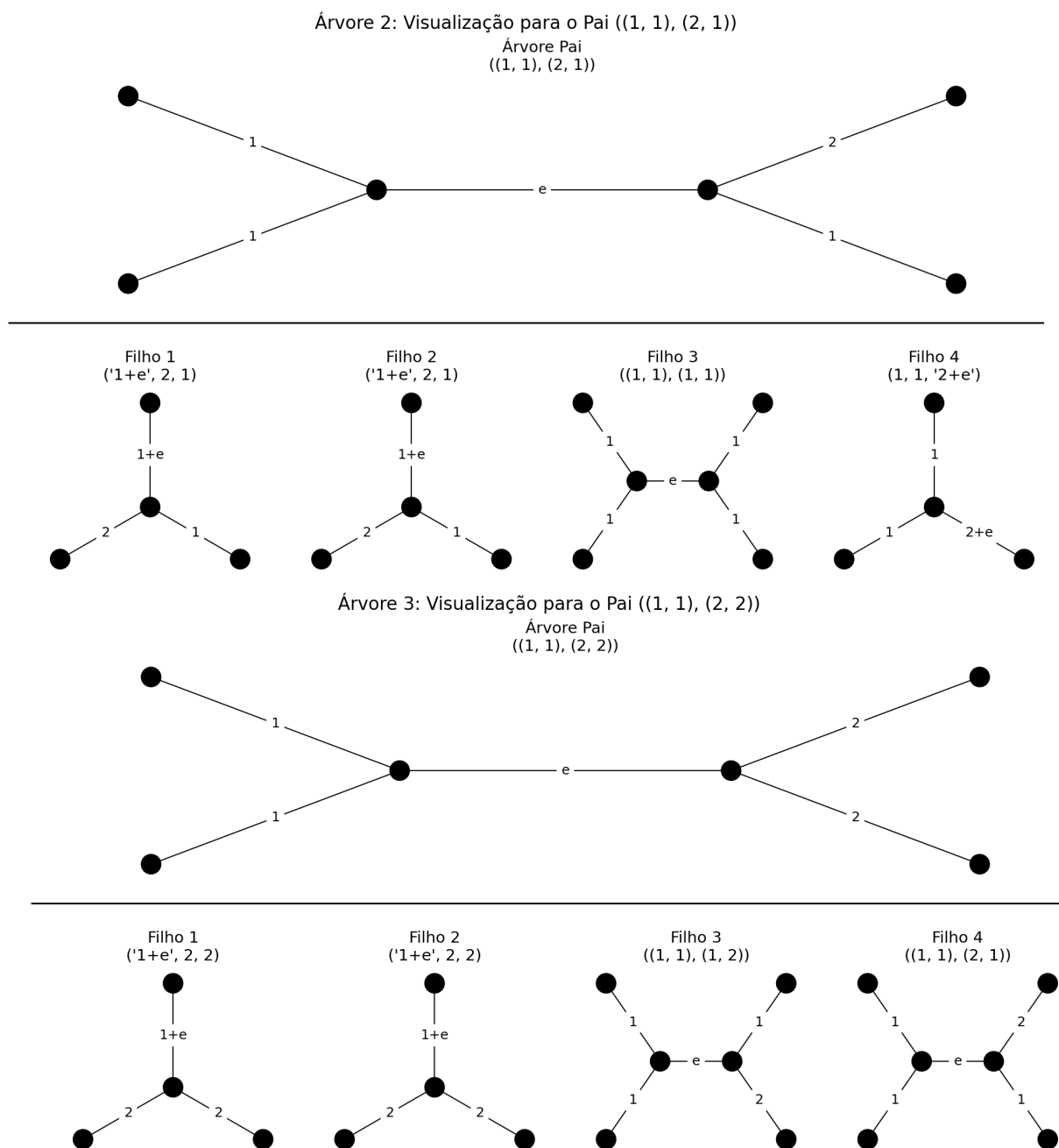


Figura 6.2: Árvores analisadas para o Lema 5.1.14 e os seus decks (3-4)

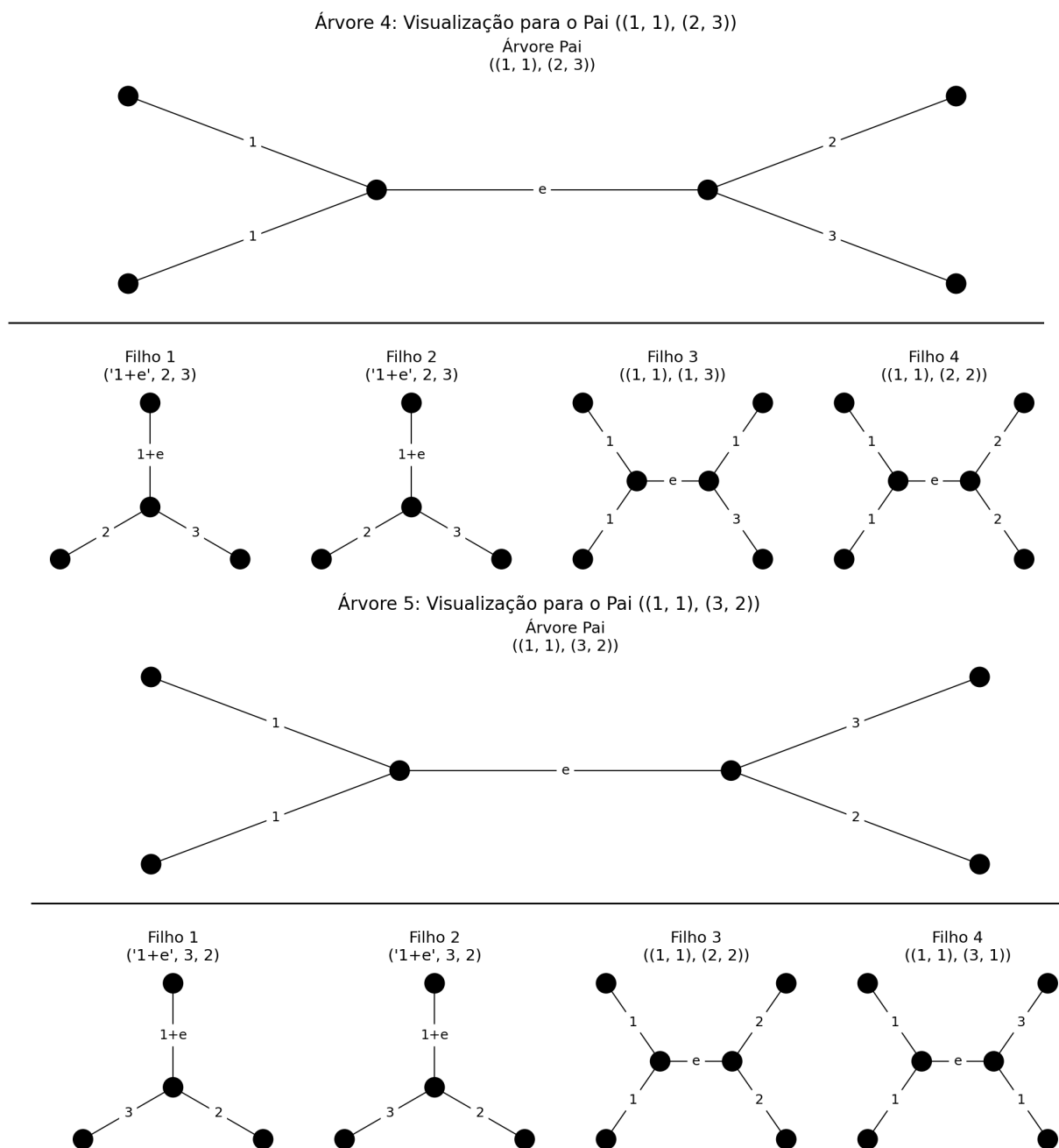


Figura 6.3: Árvores analisadas para o Lema 5.1.14 e os seus decks (5-6)

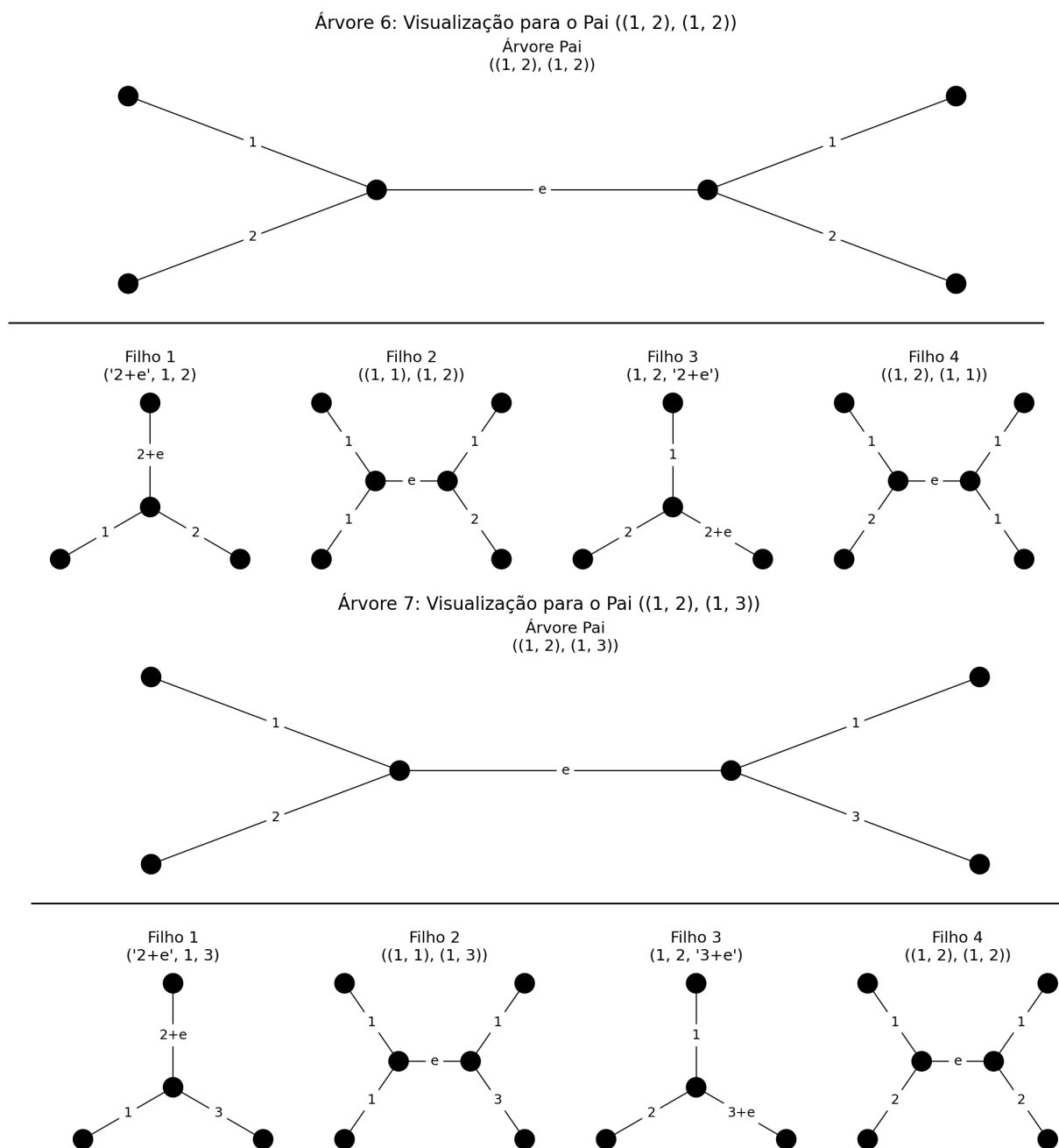


Figura 6.4: Árvores analisadas para o Lema 5.1.14 e os seus decks (7-8)

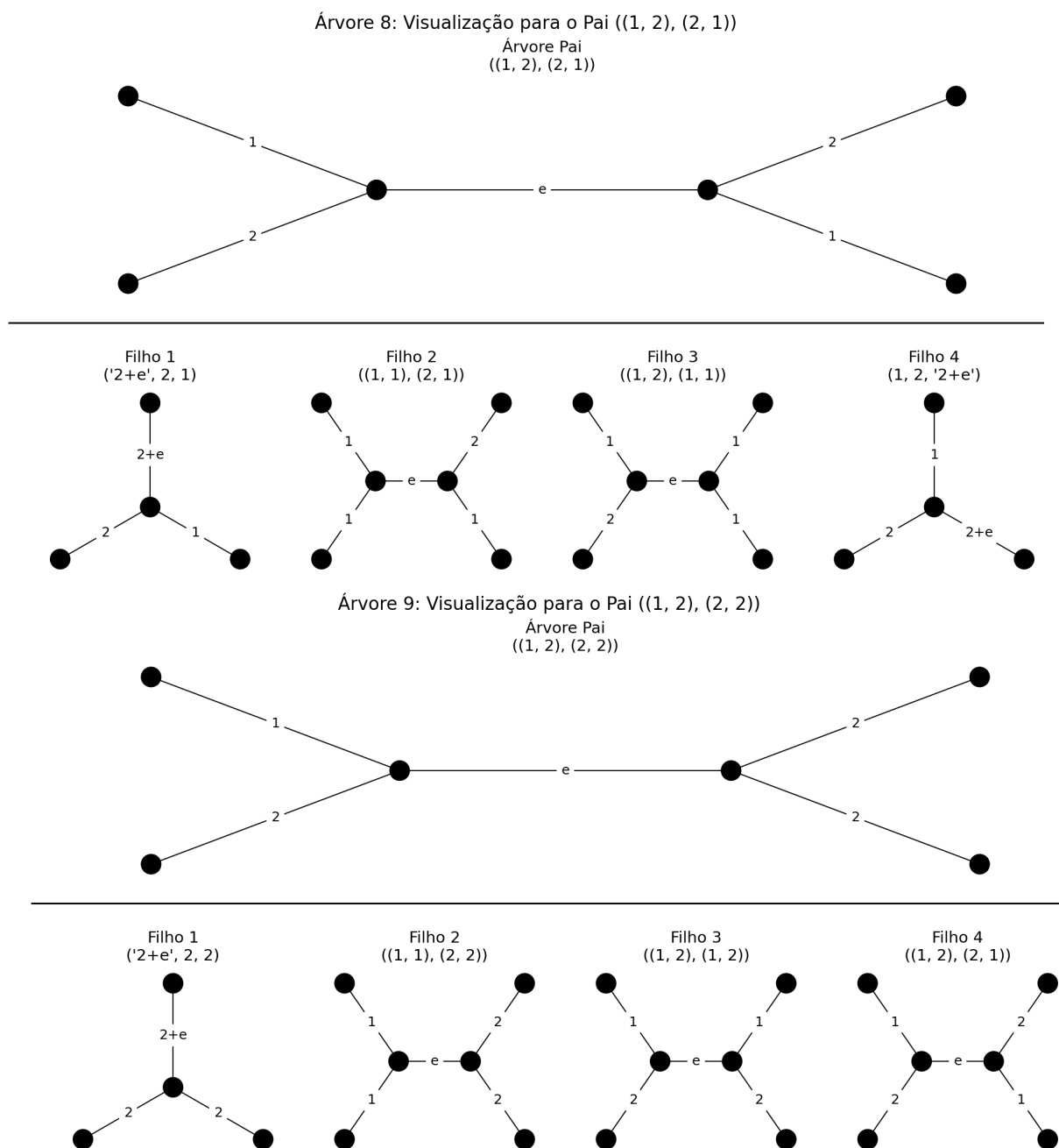


Figura 6.5: Árvores analisadas para o Lema 5.1.14 e os seus decks (9-10)

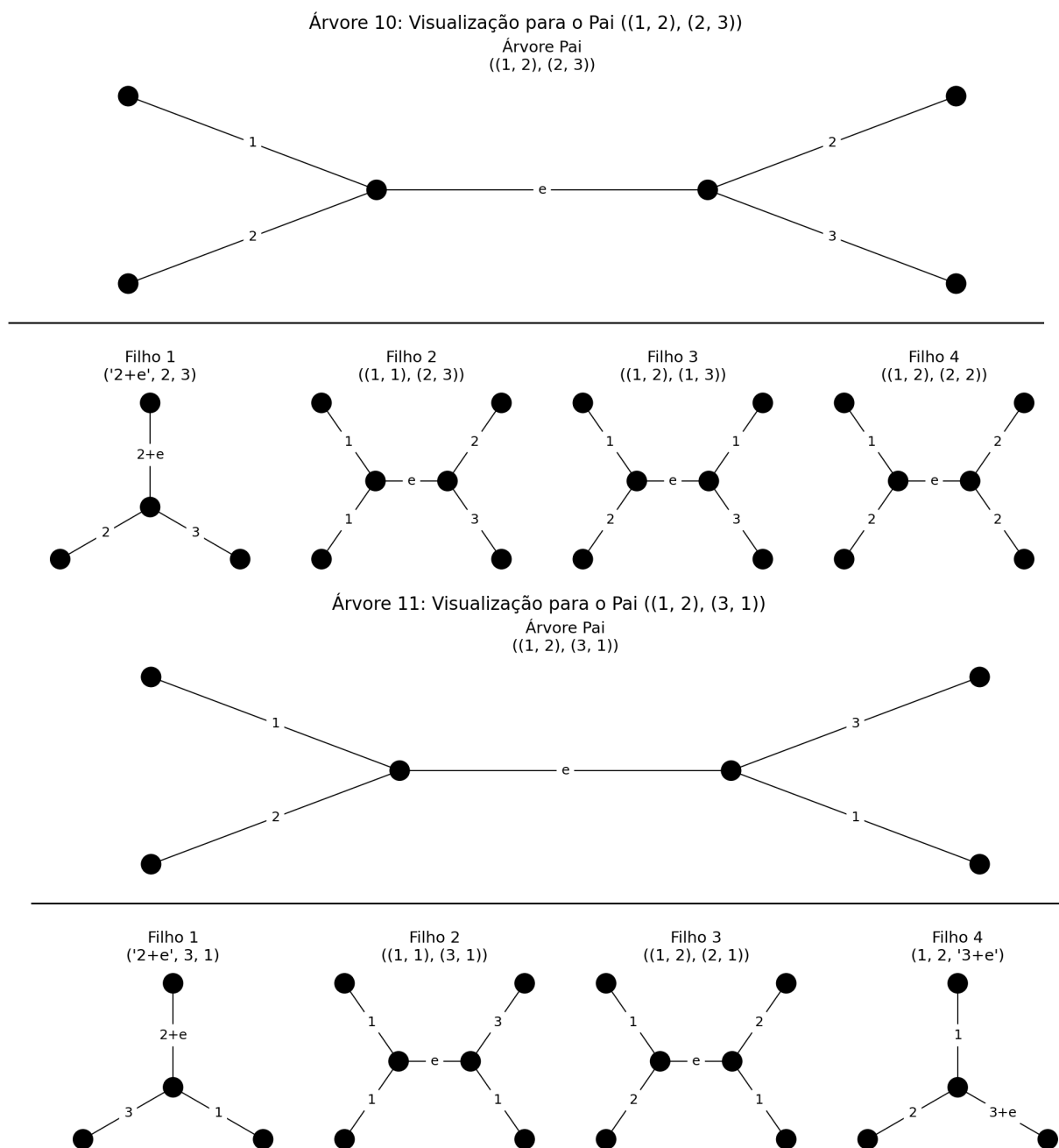


Figura 6.6: Árvores analisadas para o Lema 5.1.14 e os seus decks (11-12)

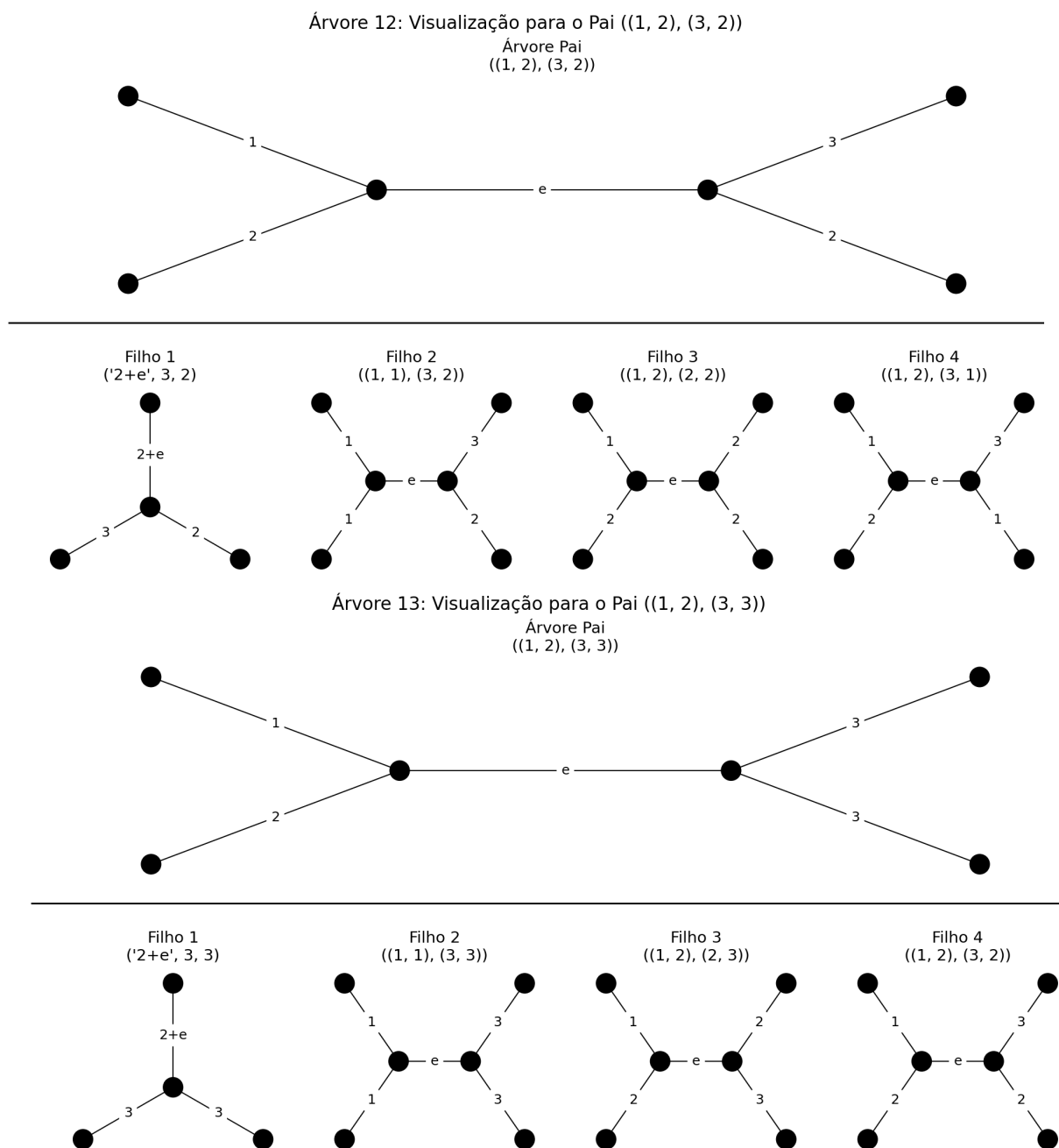


Figura 6.7: Árvores analisadas para o Lema 5.1.14 e os seus decks (13-14)

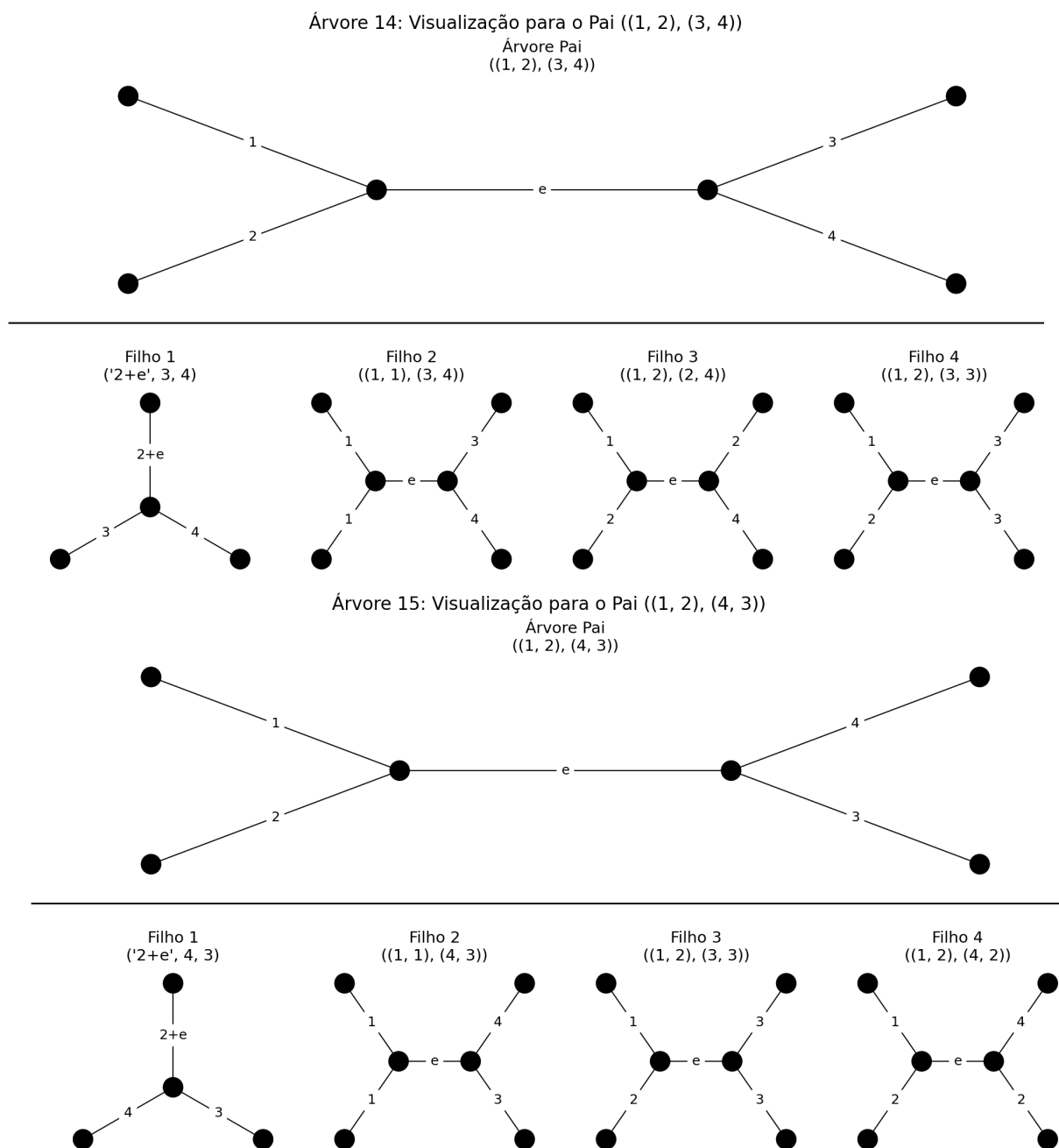


Figura 6.8: Árvores analisadas para o Lema 5.1.14 e os seus decks (15-16)

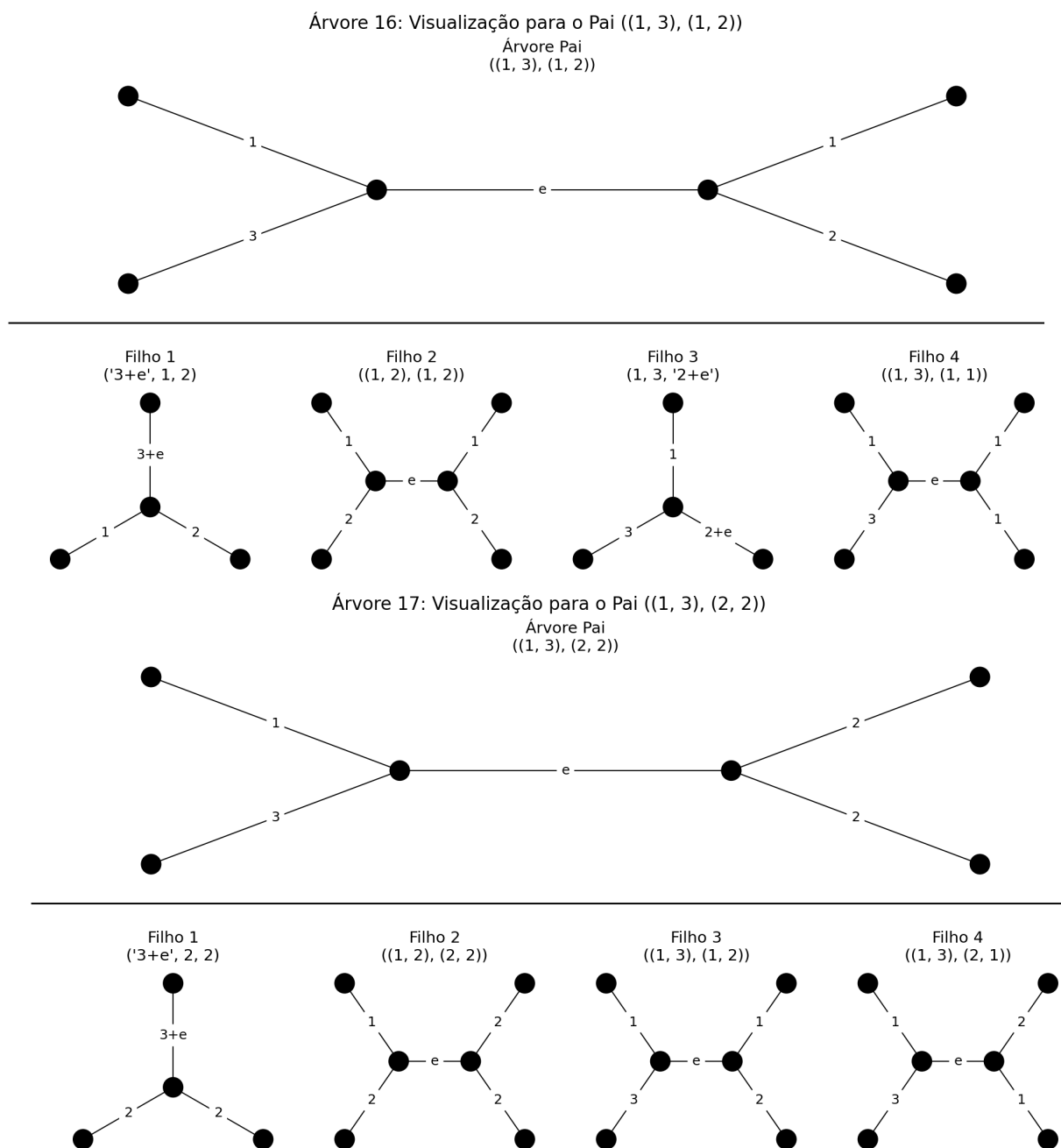


Figura 6.9: Árvores analisadas para o Lema 5.1.14 e os seus decks (17-18)

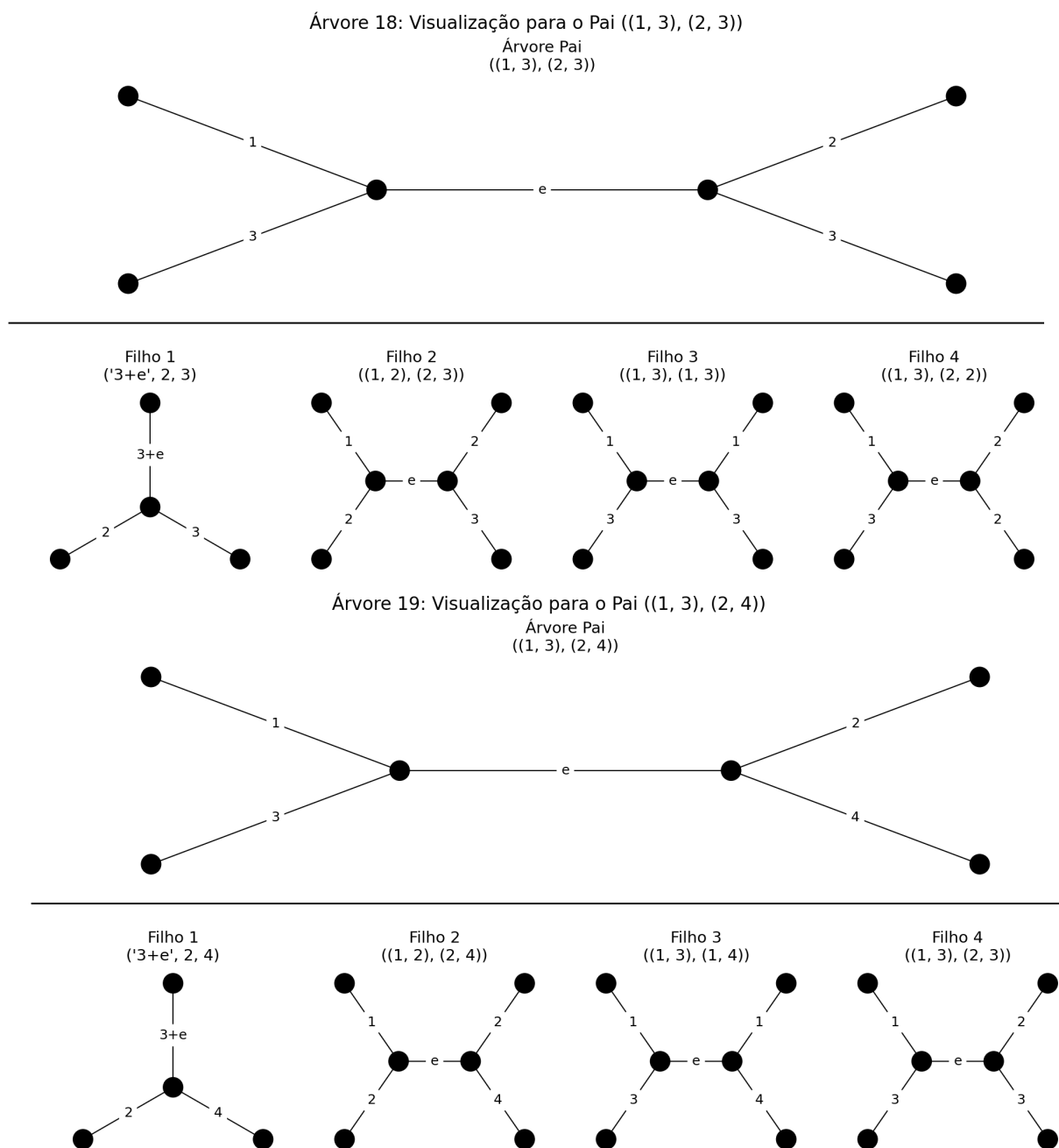


Figura 6.10: Árvores analisadas para o Lema 5.1.14 e os seus decks (19-20)

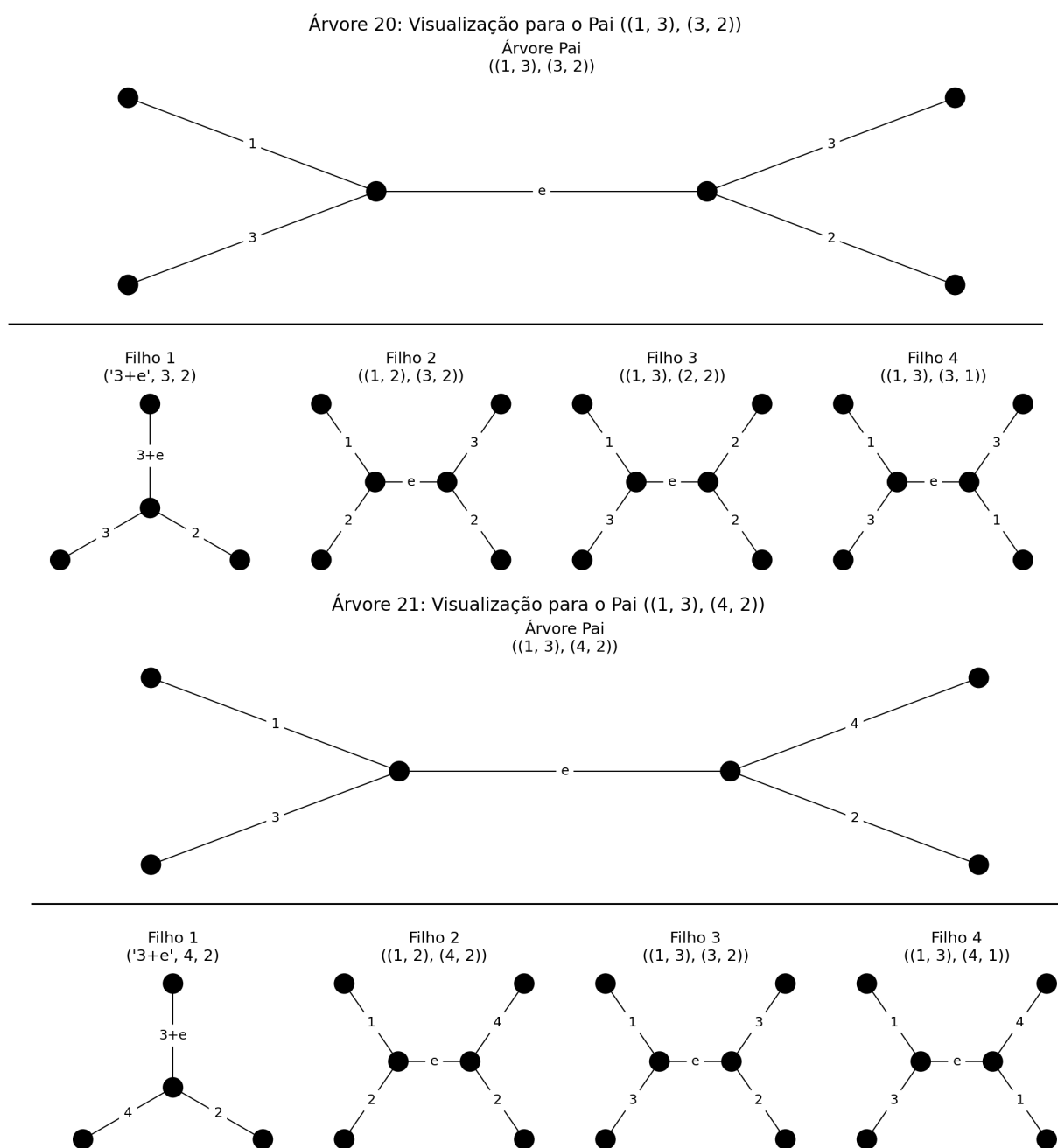


Figura 6.11: Árvores analisadas para o Lema 5.1.14 e os seus decks (21-22)

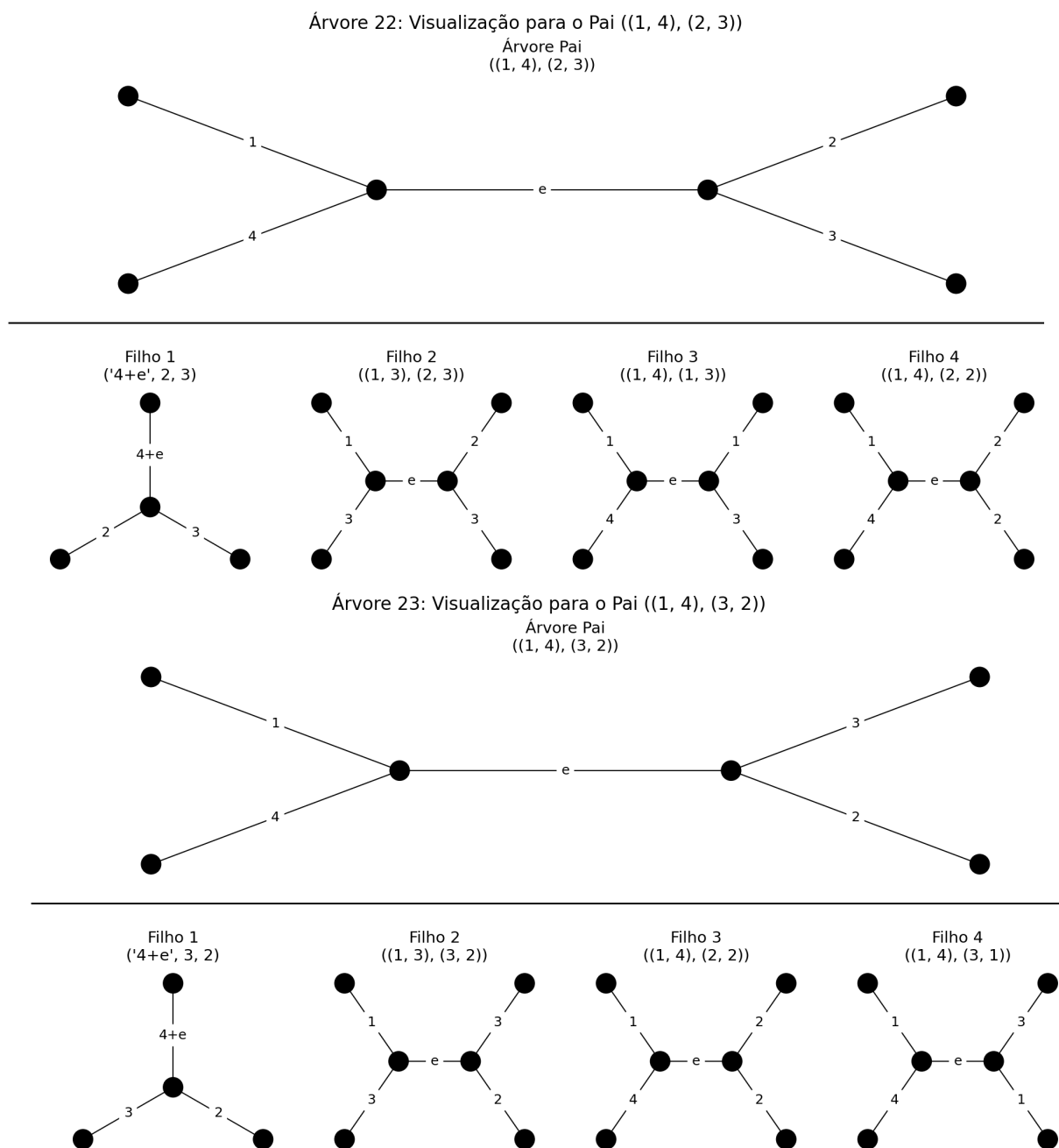


Figura 6.12: Árvores analisadas para o Lema 5.1.14 e os seus decks (23-24)

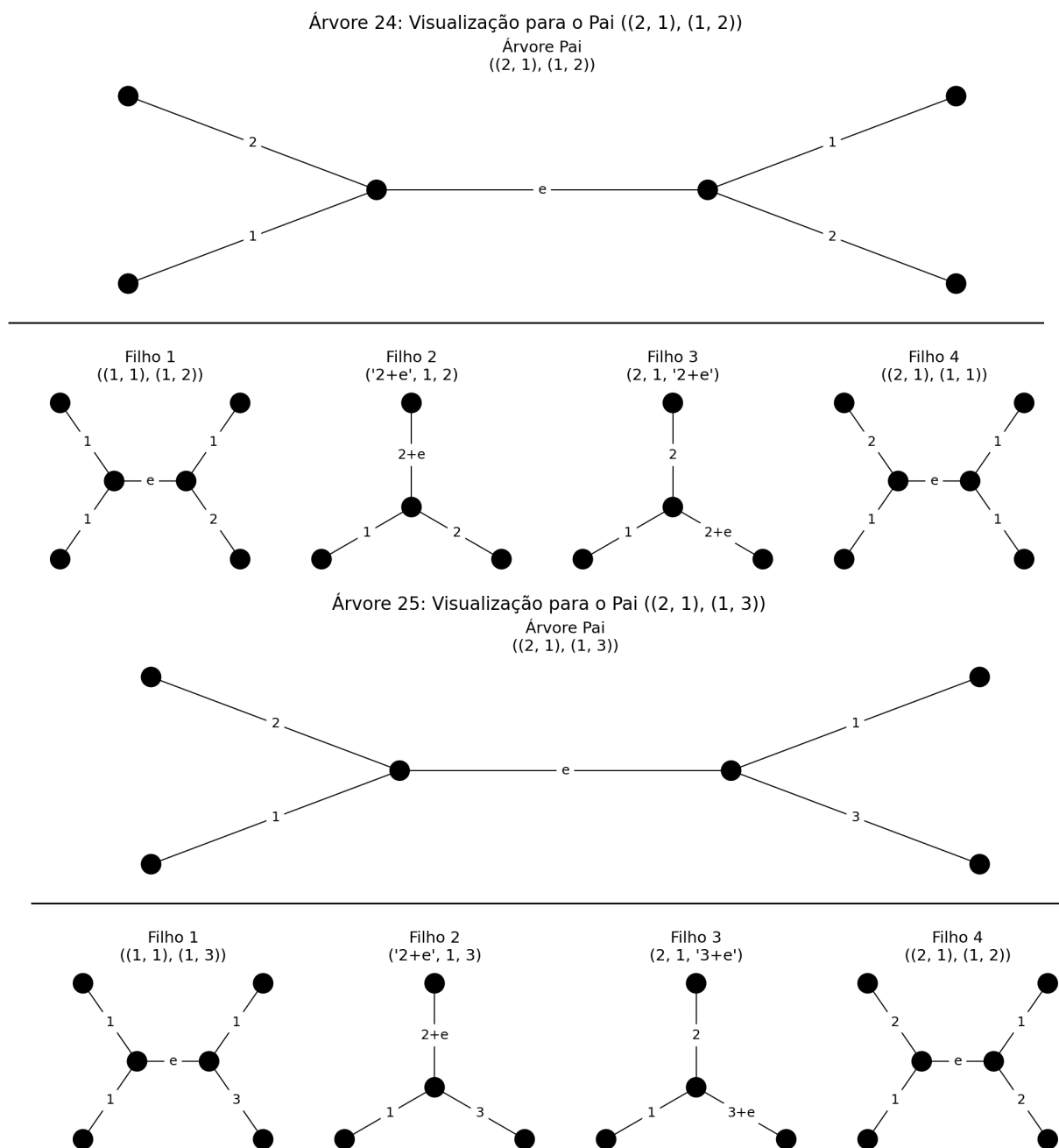


Figura 6.13: Árvores analisadas para o Lema 5.1.14 e os seus decks (25-26)

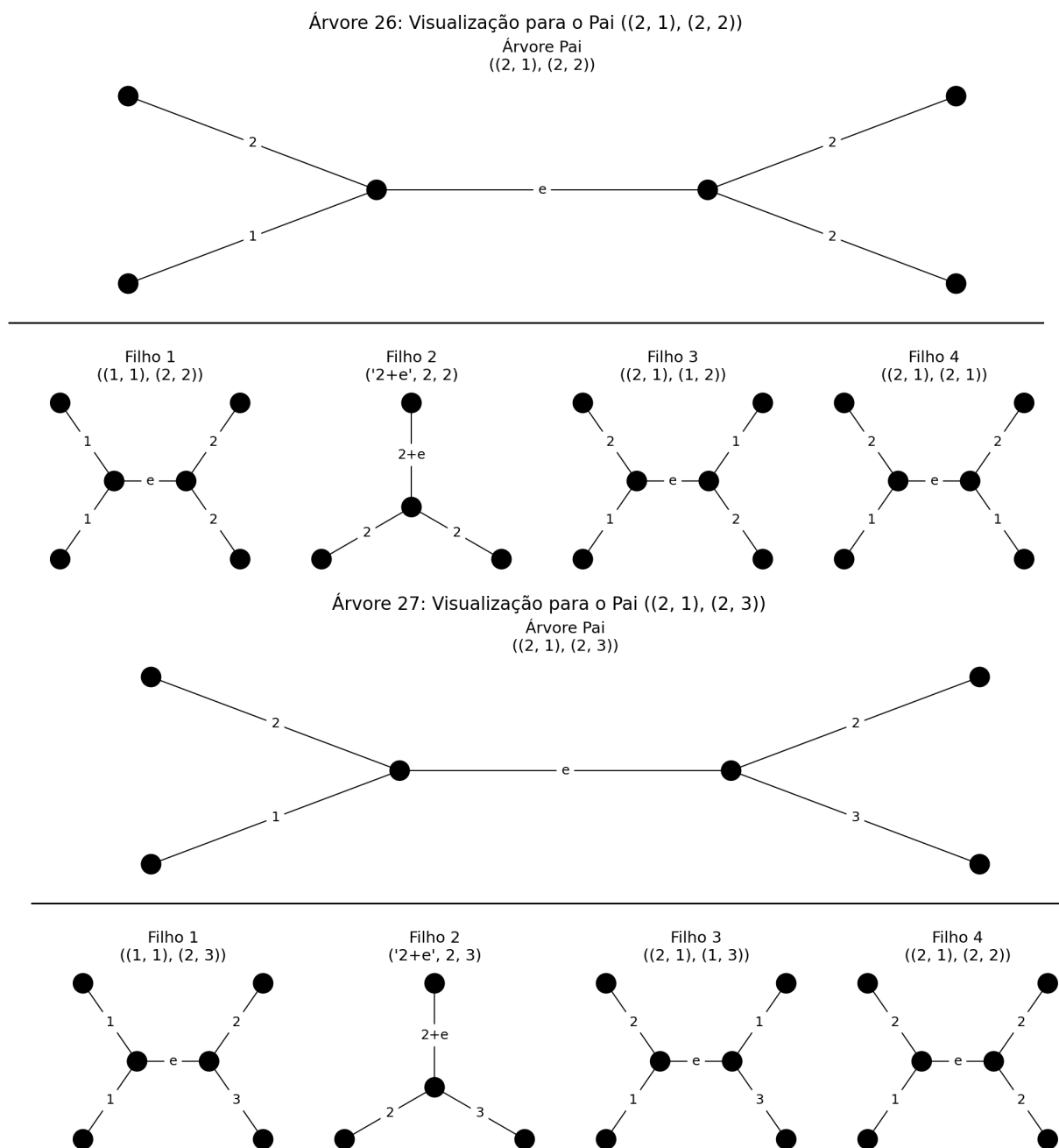


Figura 6.14: Árvores analisadas para o Lema 5.1.14 e os seus decks (27-28)

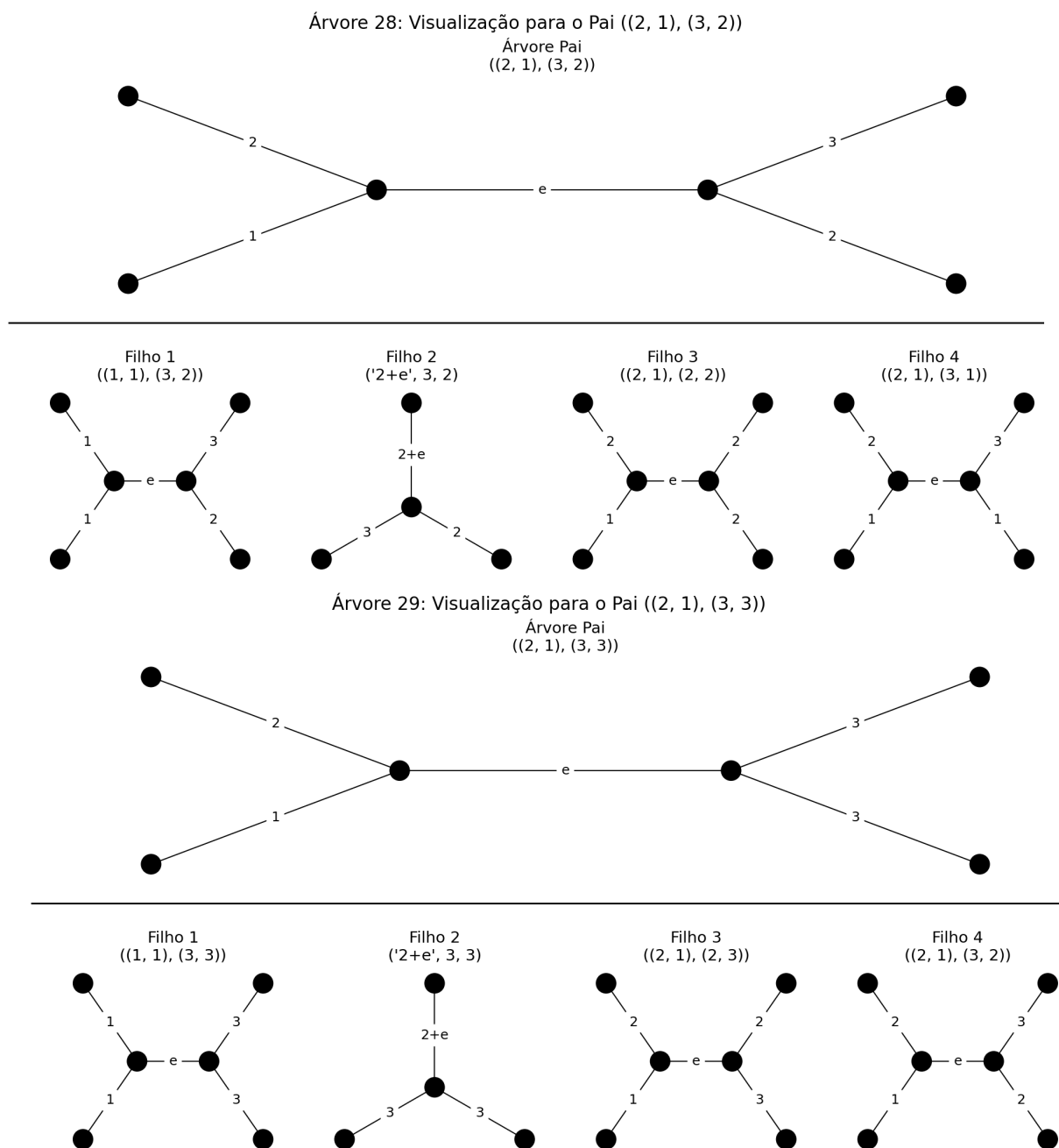


Figura 6.15: Árvores analisadas para o Lema 5.1.14 e os seus decks (29-30)

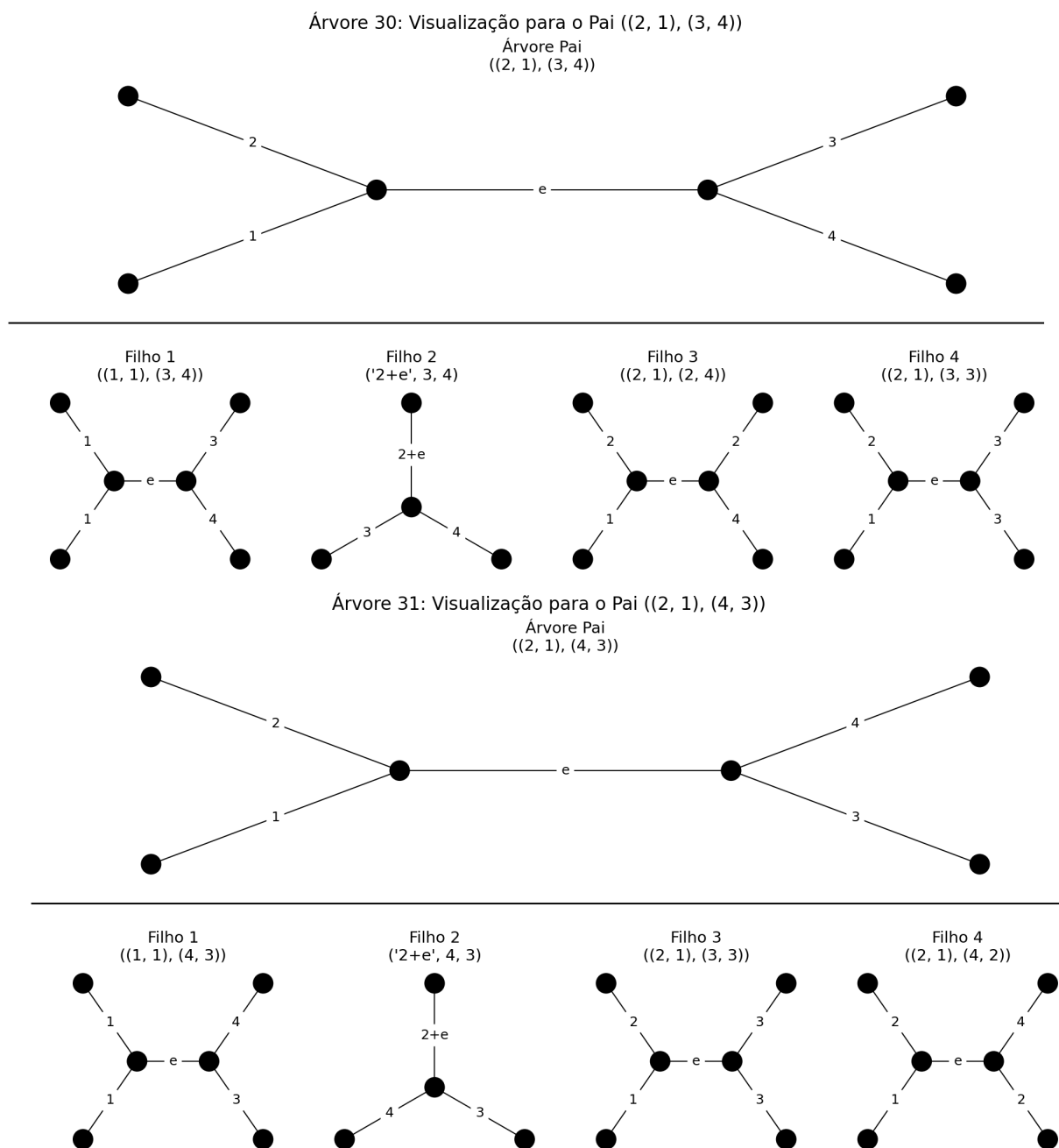


Figura 6.16: Árvores analisadas para o Lema 5.1.14 e os seus decks (31-32)

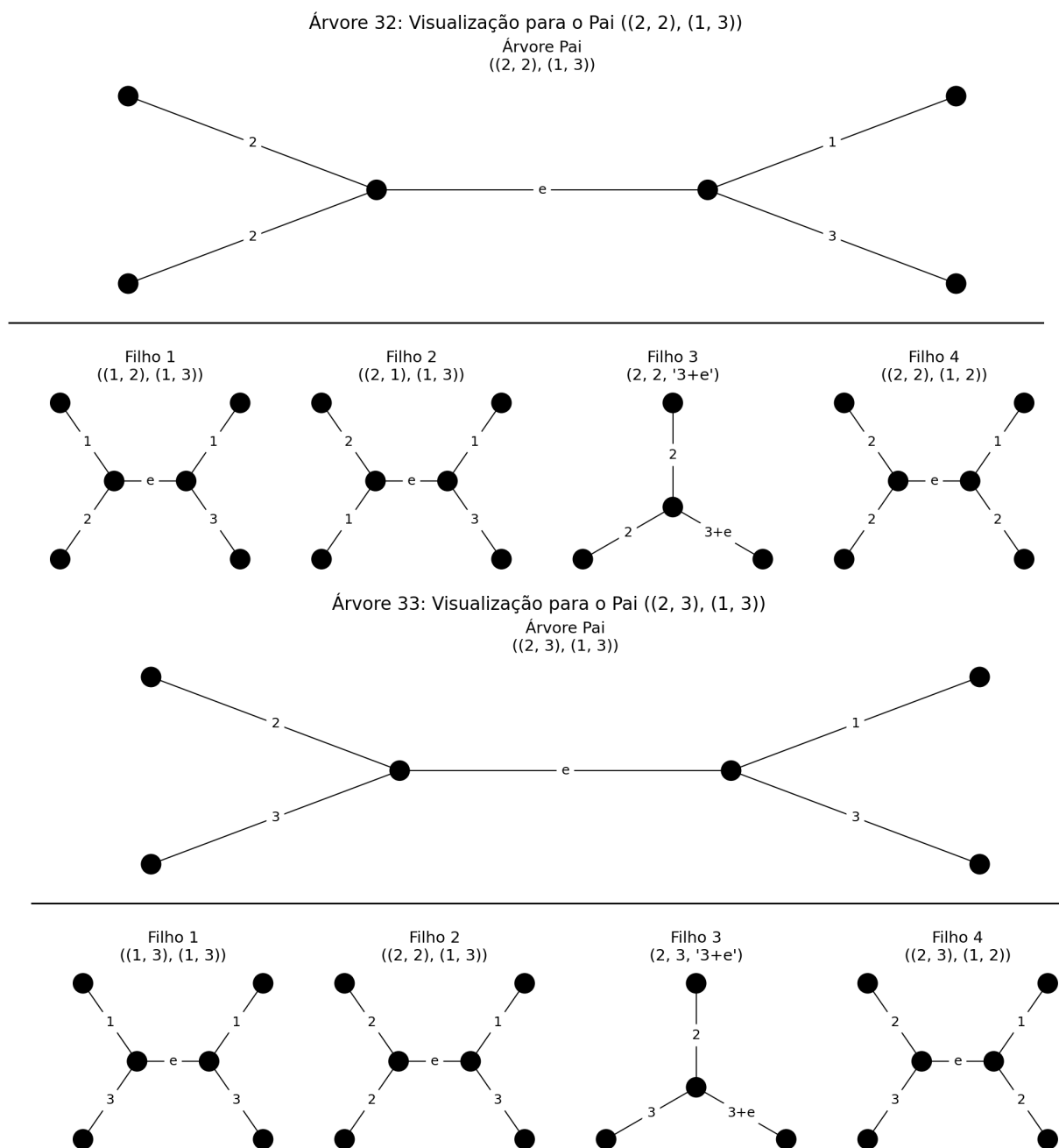


Figura 6.17: Árvores analisadas para o Lema 5.1.14 e os seus decks (33-34)

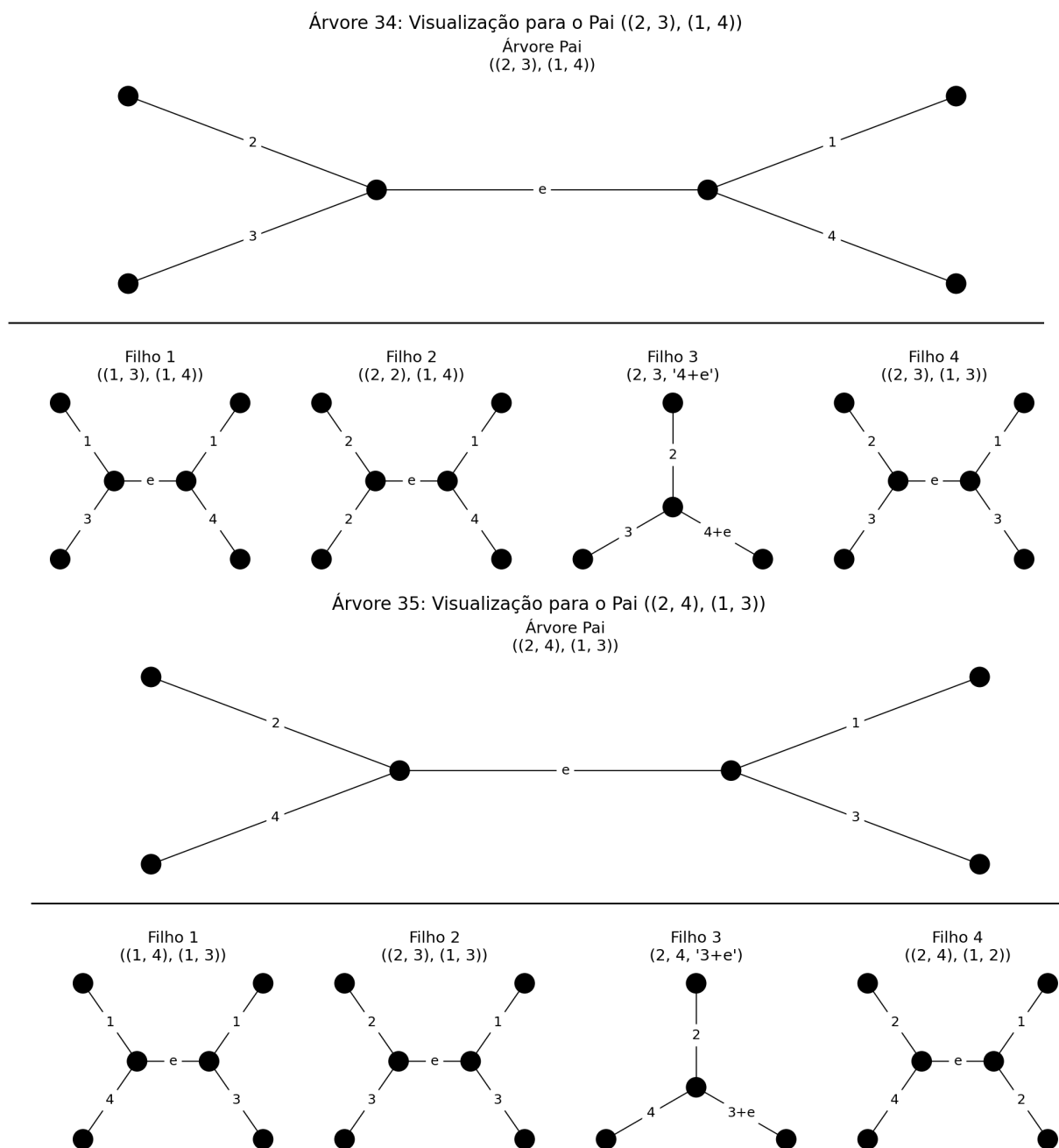


Figura 6.18: Árvores analisadas para o Lema 5.1.14 e os seus decks (35-36)

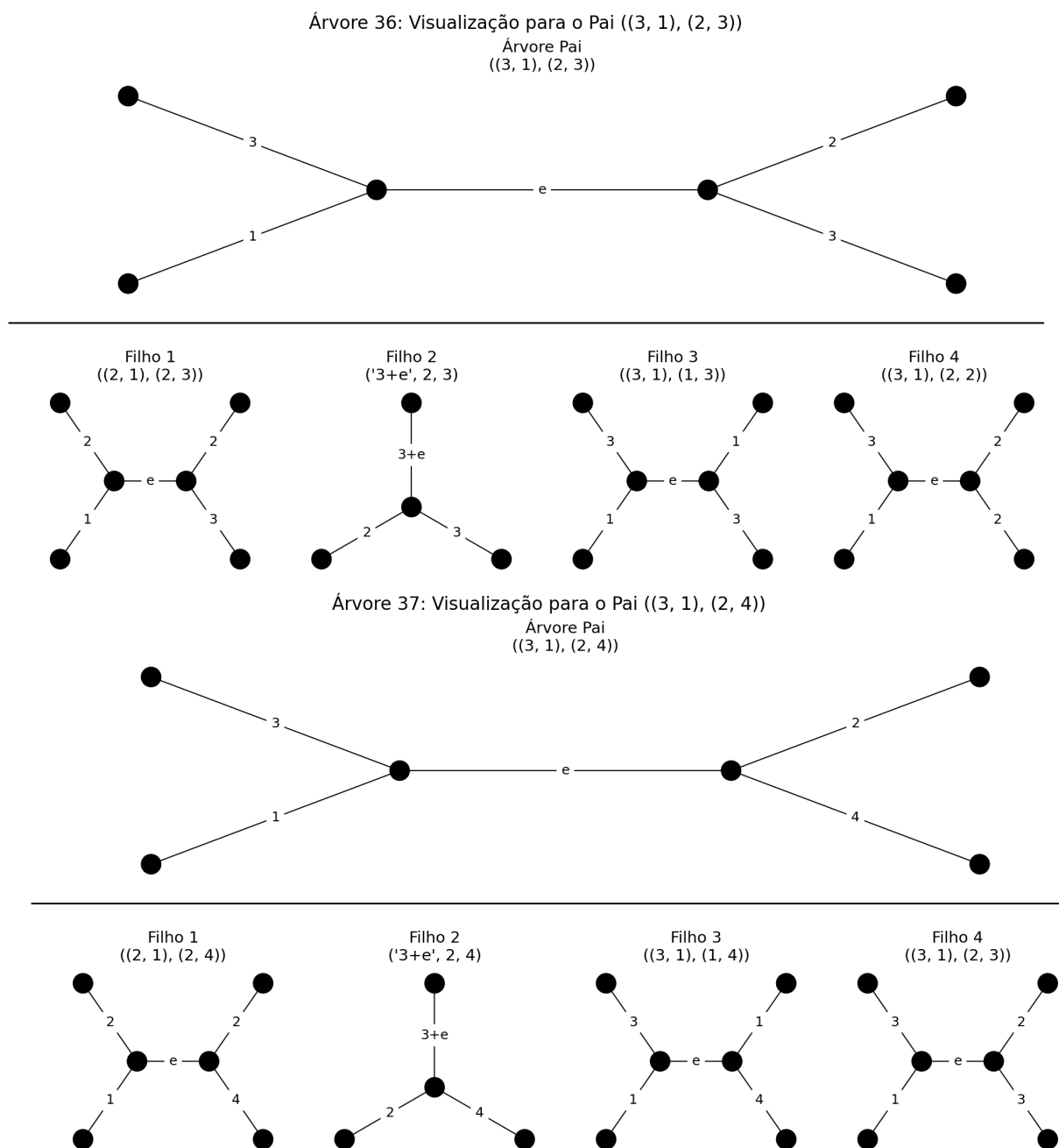


Figura 6.19: Árvores analisadas para o Lema 5.1.14 e os seus decks (37-38)

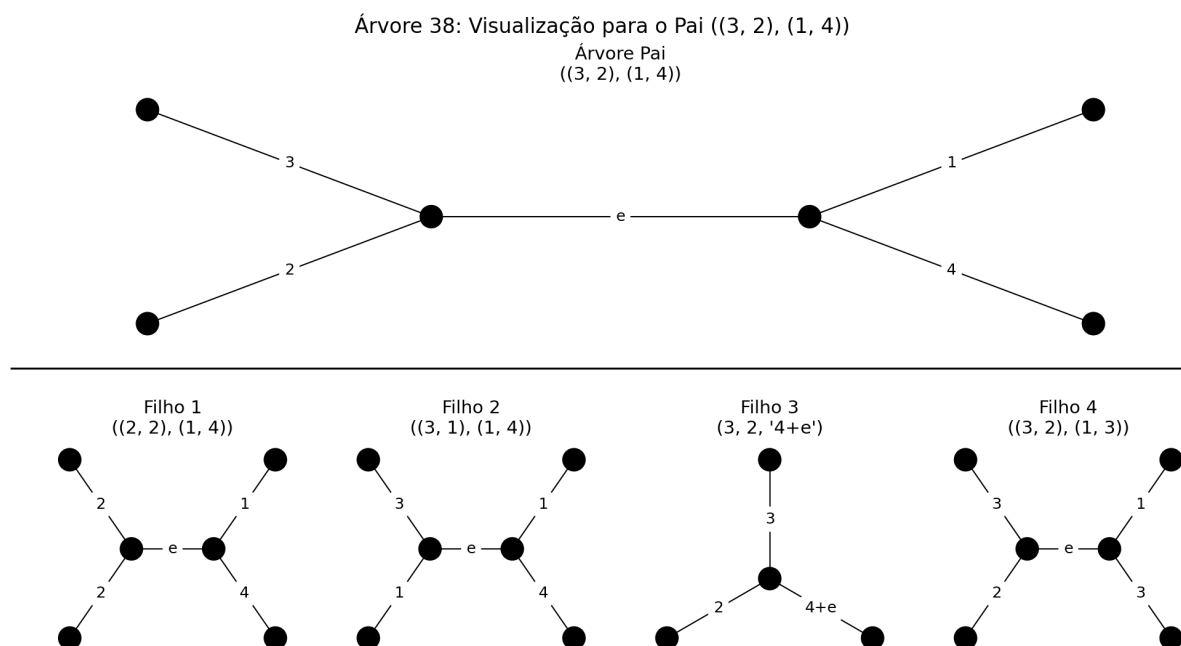


Figura 6.20: Árvores analisadas para o Lema 5.1.14 e os seus decks (39)

Bloco Python 6.1: Geração de todas as sequências de tamanho n com valores em $\{1, \dots, q\}$ usando itertools, permitindo repetições

```

1 import itertools
2
3 def gerar_sequencias(q, n):
4     """
5     Gera todas as sequências de tamanho n, compostas por valores
6     entre 1 e q,
7     com repetições permitidas.
8
9     Parâmetros:
10    q (int): o valor máximo para os elementos da sequência (
11    inclusivo), ou seja,
12    os valores variam de 1 até q.
13    n (int): o tamanho das sequências.
14
15    Retorna:
16    list: uma lista contendo todas as sequências válidas (q^n
17    sequências ao todo).
18    """
19    if q <= 0 or n < 0:
20        raise ValueError("q deve ser > 0 e n deve ser >= 0.")
21
22    # Gera todas as combinações de tamanho n com repetição,

```

```

21     # usando valores de 1 a q em vez de 0 a q-1
22     sequencias = list(itertools.product(range(1, q + 1), repeat=n))
23     return sequencias
24
25 # Exemplo de uso
26 if __name__ == "__main__":
27     q = 4 # Valores possíveis: 1, 2 e 3
28     n = 4 # Tamanho das sequências
29
30     sequencias = gerar_sequencias(q, n)
31     print(f"Total de sequências geradas: {len(sequencias)}") #
32     deve ser 3^4 = 81
33     for seq in sequencias:
34         print(seq)
35 Permut = [i for i in sequencias if 1 in i]

```

Bloco Python 6.2: Filtragem de tuplas removendo aquelas que contêm algum $x > 1$ sem o predecessor $x - 1$

```

1 def filter_tuples(list_of_tuples):
2     """
3     Filtra uma lista de tuplas, removendo todas aquelas em que
4     existe
5     um elemento  $x > 1$  tal que  $(x - 1)$  não está na mesma tupla.
6
7     Parâmetros:
8         list_of_tuples (list[tuple[int, ...]]): lista de tuplas de
9         inteiros.
10
11     Retorna:
12         list[tuple[int, ...]]: nova lista contendo apenas as tuplas
13         válidas.
14     """
15     filtered = []
16     for tpl in list_of_tuples:
17         weights = set(tpl)
18         # Verifica, para cada  $x > 1$  em weights, se  $(x-1)$  também est
19         á em weights
20         if all((x == 1 or (x - 1) in weights) for x in weights):
21             filtered.append(tpl)
22     return filtered
23
24 # Exemplo de uso

```

```

21 if __name__ == "__main__":
22
23
24     resultado_permut = filter_tuples(Permut)
25     print("Antes:", Permut)
26     print("Depois:", resultado_permut)

```

Bloco Python 6.3: Criação, layout e visualização de árvores bicentradas a partir de configurações únicas de branches

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import itertools
4 import math
5
6 # Variável global para garantir que os vértices tenham nomes únicos
7   em cada árvore
8 node_counter = 0
9
10 def get_unique_node_id():
11     """Retorna um ID numérico único para um vértice."""
12     global node_counter
13     node_counter += 1
14     return node_counter
15
16 def layout_bicentered_tree(G, c1='c1', c2='c2'):
17     """
18     Posiciona a árvore bicentrada com:
19     - c1 em (-0.2, 0), c2 em (0.2, 0)
20     - Vizinhos de cada centro dispostos simetricamente em ( 0 .5,
21       0 .4)
22     """
23     pos = {}
24     # Posições centrais fixas
25     pos[c1] = (-0.2, 0)
26     pos[c2] = ( 0.2, 0)
27
28     # Vizinhos de c1
29     neighbors_c1 = [n for n in G.neighbors(c1) if n != c2]
30     neighbors_c1 = sorted(neighbors_c1)
31     if len(neighbors_c1) >= 1:
32         pos[neighbors_c1[0]] = (-0.5, 0.4)
33     if len(neighbors_c1) >= 2:

```

```

32         pos[neighbors_c1[1]] = (-0.5, -0.4)
33
34     # Vizinhos de c2
35     neighbors_c2 = [n for n in G.neighbors(c2) if n != c1]
36     neighbors_c2 = sorted(neighbors_c2)
37     if len(neighbors_c2) >= 1:
38         pos[neighbors_c2[0]] = (0.5, 0.4)
39     if len(neighbors_c2) >= 2:
40         pos[neighbors_c2[1]] = (0.5, -0.4)
41
42     return pos
43
44
45 def create_bicentered_tree(branches_c1, branches_c2):
46     """
47     Cria uma árvore bicentrada. A aresta central tem peso 1 para o
48     layout
49     e rótulo 'e' para visualização.
50     """
51     global node_counter
52     node_counter = 0 # Reseta o contador para cada nova árvore
53
54     G = nx.Graph()
55     center_v1, center_v2 = 'c1', 'c2'
56     G.add_edge(center_v1, center_v2, weight=1, label='e')
57
58     def create_weighted_branches(center_node, branch_weights):
59         for w in branch_weights:
60             if w <= 0:
61                 continue
62             leaf = get_unique_node_id()
63             G.add_edge(center_node, leaf, weight=w, label=str(w))
64
65     create_weighted_branches(center_v1, branches_c1)
66     create_weighted_branches(center_v2, branches_c2)
67     return G
68
69 def visualize_trees(trees, configs, sample_size=75):
70     """
71     Desenha uma amostra das árvores geradas, exibindo os rótulos
72     das arestas.
73     """

```

```

72     if not trees:
73         print("Nenhuma árvore para visualizar.")
74         return
75
76     N = len(trees)
77     # Amostragem uniforme
78     if N > sample_size:
79         indices = [int(i*(N-1)/(sample_size-1)) for i in range(
sample_size)]
80     else:
81         indices = list(range(N))
82
83     sample = [(trees[i], configs[i], i) for i in indices]
84     cols = min(4, len(sample))
85     rows = math.ceil(len(sample)/cols)
86     plt.figure(figsize=(cols*4, rows*3))
87     for ax_i, (G, (c1, c2), orig_idx) in enumerate(sample, start=1)
:
88         ax = plt.subplot(rows, cols, ax_i)
89         pos = layout_bicentered_tree(G, c1='c1', c2='c2')#nx.
spring_layout(G, seed=42,)
90         node_colors = ['#CC092F' if n in ('c1', 'c2') else '#1f77b4'
for n in G.nodes()]
91         nx.draw(G, pos, with_labels=False, node_color=node_colors,
node_size=200, ax=ax)
92         el = nx.get_edge_attributes(G, 'label')
93         nx.draw_networkx_edge_labels(G, pos, edge_labels=el,
font_size=12, ax=ax, rotate=False) #, bbox=dict(facecolor='white
', alpha=0.6)
94         ax.set_title(f"Árvore {orig_idx} \n c1={c1} c2={c2}")
95         ax.axis('off')
96     plt.tight_layout()
97     plt.show()
98
99 if __name__ == "__main__":
100
101     # 1) Converta cada tupla (a, b, c, d) em ( [a, b], [c, d] )
102     all_configs = [[(t[0], t[1]), (t[2], t[3])] for t in
resultado_permut]
103
104
105     # # 2) Remove equivalências do tipo ([x,y],[z,w]) == ([z,w],[x,

```

```

y]), mantendo a ordem interna
106
107     seen = set()
108     unique_configs = []
109
110     for pair in all_configs:
111         a, b = pair
112         # Cria chave canônica considerando reversão interna + troca
de posição entre as listas
113         key1 = tuple(a), tuple(b)
114         key2 = tuple(reversed(b)), tuple(reversed(a))
115
116         # Usamos a menor (lexicograficamente) das duas chaves possí
veis para garantir unicidade
117         canonical_key = min(key1, key2)
118
119         if canonical_key not in seen:
120             seen.add(canonical_key)
121             unique_configs.append(pair)
122
123     # 3) Gere as árvores
124     all_trees = [create_bicentered_tree(c1, c2) for c1, c2 in
unique_configs]
125
126     print(f"Total de árvores geradas: {len(all_trees)}")
127
128     # 4) Visualize (opcional)
129     viz = visualize_trees(all_trees, unique_configs, sample_size=
39)

```

Bloco Python 6.4: Construção de dicionário de minors subtraindo 1 de cada elemento das configurações únicas

```

1 minors_dict = {}
2
3 for pair in unique_configs:
4     a, b = pair # [a1, a2], [b1, b2]
5     minors = []
6
7     # Subtrair 1 de cada elemento de a individualmente
8     for i in range(2):
9         new_a = a.copy()
10        if new_a[i] > 1:

```

```

11         new_a[i] -= 1
12     else:
13         new_a.pop(i)
14     minors.append((new_a, b))
15
16     # Subtrair 1 de cada elemento de b individualmente
17     for i in range(2):
18         new_b = b.copy()
19         if new_b[i] > 1:
20             new_b[i] -= 1
21         else:
22             new_b.pop(i)
23         minors.append((a, new_b))
24
25     minors_dict[(tuple(a), tuple(b))] = minors
26
27 minors_dict

```

Bloco Python 6.5: Aplicação de regras de “flatten and shift” e normalização ao dicionário de minors

```

1 # ----- Função de transformação -----
2 def flatten_and_shift(pair):
3     """
4     Converte:
5         ([x], [y, z]) -> (x+1, y, z)
6         ([x,y], [z] ) -> (x, y, z+1)
7     Mantém qualquer outro par inalterado.
8     """
9     a, b = pair
10
11     # Regra ( [x], [y,z] )
12     if len(a) == 1 and len(b) == 2:
13         x, = a
14         y, z = b
15         return (f"{str(x)}+e", y, z)
16
17     # Regra ( [x,y], [z] )
18     elif len(a) == 2 and len(b) == 1:
19         x, y = a
20         z, = b
21         return (x, y, f"{str(z)}+e")
22

```

```

23     # Demais casos: devolve idêntico
24     return pair
25
26
27 # ----- Aplica a transformação ao dicionário inteiro -----
28 def transform_minors_dict(minors_dict):
29     """
30     Retorna novo dicionário no mesmo formato, mas com todas
31     as chaves e valores transformados conforme as duas regras.
32     """
33     new_dict = {}
34
35     for key, minor_list in minors_dict.items():
36         new_key = flatten_and_shift(key)
37         new_minors = [flatten_and_shift(m) for m in minor_list]
38         new_dict[new_key] = new_minors
39
40     return new_dict
41
42 transformed = transform_minors_dict(minors_dict)
43
44 normalized_dict = normalize_minors_dict(transformed)
45
46 normalized_dict = {
47     k: [to_tuple(v) for v in vals]
48     for k, vals in normalized_dict.items()
49 }

```

Bloco Python 6.6: Visualização hierárquica de árvores bicentradas e centradas a partir de estrutura de dados, com layout fixo em grids e salvamento opcional de imagens

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import matplotlib.gridspec as gridspec
4 import math
5 import os
6
7 # Variável global para garantir que os vértices tenham nomes únicos
8   em cada diagrama de árvore.
9 node_counter = 0
10
11 def get_unique_node_id():

```

```

11     """Retorna um ID numérico único para um vértice, começando em
12     1."""
13     global node_counter
14     node_counter += 1
15     return node_counter
16
17 def create_bicentered_tree(branches_c1, branches_c2):
18     """
19     Cria uma árvore bicentrada. A aresta central tem um peso de 1
20     para o layout
21     e um rótulo 'e' para a visualização.
22     """
23     global node_counter
24     node_counter = 0 # Reinicia o contador para cada nova árvore
25
26     G = nx.Graph()
27     center_v1, center_v2 = 'c1', 'c2'
28     G.add_edge(center_v1, center_v2, weight=1, label='e')
29
30     def create_weighted_branches(center_node, branch_weights):
31         """Função auxiliar para adicionar nós folha a um centro."""
32         for w in branch_weights:
33             # Lida com segurança com tipos não numéricos
34             if isinstance(w, (int, float)) and w <= 0:
35                 continue
36             leaf = get_unique_node_id()
37             G.add_edge(center_node, leaf, weight=1, label=str(w))
38
39     create_weighted_branches(center_v1, branches_c1)
40     create_weighted_branches(center_v2, branches_c2)
41     return G
42
43 def create_centered_tree(branch_weights):
44     """
45     Cria uma árvore centrada com um único nó central.
46     """
47     global node_counter
48     node_counter = 0 # Reinicia o contador para cada nova árvore
49
50     G = nx.Graph()
51     center_node = 'c0'
52     G.add_node(center_node)

```

```

51
52     for w in branch_weights:
53         # Lida com segurança com tipos não numéricos
54         if isinstance(w, (int, float)) and w <= 0:
55             continue
56         leaf = get_unique_node_id()
57         # Atribui o valor como rótulo, seja ele string ou int
58         G.add_edge(center_node, leaf, weight=1, label=str(w))
59
60     return G
61
62 def layout_bicentered_tree(G, c1='c1', c2='c2'):
63     """
64     Posiciona uma árvore bicentrada com coordenadas fixas para os
65     centros e os seus vizinhos
66     para um layout consistente e simétrico.
67     """
68     pos = {c1: (-0.2, 0), c2: (0.2, 0)}
69
70     # Posiciona os vizinhos de c1
71     neighbors_c1 = sorted([n for n in G.neighbors(c1) if n != c2])
72     if len(neighbors_c1) >= 1:
73         pos[neighbors_c1[0]] = (-0.5, 0.4)
74     if len(neighbors_c1) >= 2:
75         pos[neighbors_c1[1]] = (-0.5, -0.4)
76
77     # Posiciona os vizinhos de c2
78     neighbors_c2 = sorted([n for n in G.neighbors(c2) if n != c1])
79     if len(neighbors_c2) >= 1:
80         pos[neighbors_c2[0]] = (0.5, 0.4)
81     if len(neighbors_c2) >= 2:
82         pos[neighbors_c2[1]] = (0.5, -0.4)
83
84     return pos
85
86 def layout_centered_tree(G, c0='c0'):
87     """
88     Posiciona uma árvore centrada, organizando as suas folhas num c
89     írculo ao redor do centro.
90     """
91     pos = {c0: (0, 0)}
92     neighbors = sorted(list(G.neighbors(c0)))

```

```

91     num_neighbors = len(neighbors)
92
93     if num_neighbors == 0:
94         return pos
95
96     # Organiza as folhas num círculo para um layout limpo
97     angle_step = 2 * math.pi / num_neighbors
98     for i, node in enumerate(neighbors):
99         angle = angle_step * i + (math.pi / 2) # Começa do topo
100         pos[node] = (math.cos(angle), math.sin(angle))
101
102     return pos
103
104 def draw_tree_on_axis(ax, G, pos, title):
105     """
106     Função auxiliar para desenhar uma única árvore num determinado
107     eixo Matplotlib.
108     """
109     # Diferencia os nós centrais com uma cor distinta
110     center_nodes = [node for node in G.nodes() if isinstance(node,
111 str) and 'c' in node]
112     node_colors = ['#CC092F' if n in center_nodes else '#1f77b4'
113 for n in G.nodes()]
114
115     nx.draw(G, pos, ax=ax, with_labels=False, node_color=
116 node_colors, node_size=300)
117
118     # Desenha os rótulos das arestas (pesos)
119     edge_labels = nx.get_edge_attributes(G, 'label')
120     nx.draw_networkx_edge_labels(G, pos, ax=ax, edge_labels=
121 edge_labels, font_size=12, rotate=False)
122
123     ax.set_title(title, fontsize=14)
124     ax.axis('off')
125
126 def visualize_data_structure(data, output_path=None):
127     """
128     Visualiza a estrutura de dados hierárquica e, opcionalmente,
129     salva as figuras.
130
131     :param data: O dicionário contendo a estrutura de árvores.
132     :param output_path: O caminho da pasta onde as imagens serão

```

```

salvas.
    """
    if not data:
        print("O dicionário de entrada está vazio. Nada para
visualizar.")
        return

    # Cria o diretório de saída se ele for especificado e não
existir
    if output_path:
        os.makedirs(output_path, exist_ok=True)
        print(f"Salvando figuras em: {os.path.abspath(output_path)}")

    for i, (parent_config, children_configs) in enumerate(data.
items()):

        num_children = len(children_configs) if children_configs
else 1
        fig = plt.figure(figsize=(max(4 * num_children, 8), 8))
        fig.suptitle(f"Árvore {i}: Visualização para o Pai {
parent_config}", fontsize=16, y=0.98)

        gs = gridspec.GridSpec(2, num_children, height_ratios=[1.2,
1], hspace=0.4)

        # --- Desenha a Árvore Pai ---
        ax_parent = fig.add_subplot(gs[0, :])
        parent_tree = create_bicentered_tree(parent_config[0],
parent_config[1])
        parent_pos = layout_bicentered_tree(parent_tree)

        parent_title = f"Árvore Pai\n{parent_config}"

        draw_tree_on_axis(ax_parent, parent_tree, parent_pos,
parent_title)

        # --- Desenha a Linha Separadora Horizontal ---
        line = plt.Line2D([0.1, 0.9], [0.5], transform=fig.
transFigure, color='black', lw=1.5)
        fig.add_artist(line)

```

```

158     # --- Desenha as Árvores Filhas ---
159     if not children_configs:
160         ax_child = fig.add_subplot(gs[1, :])
161         ax_child.text(0.5, 0.5, "Sem filhos para exibir", ha='
center', va='center', fontsize=14)
162         ax_child.axis('off')
163     else:
164         for j, child_config in enumerate(children_configs):
165             ax_child = fig.add_subplot(gs[1, j])
166
167             if isinstance(child_config, tuple) and len(
child_config) == 2 and \
168                 isinstance(child_config[0], tuple) and
isinstance(child_config[1], tuple):
169                 child_tree = create_bicentered_tree(
child_config[0], child_config[1])
170                 child_pos = layout_bicentered_tree(child_tree)
171                 title = f"Filho {j+1}\n{child_config}"
172             else:
173                 child_tree = create_centered_tree(child_config)
174                 child_pos = layout_centered_tree(child_tree)
175                 title = f"Filho {j+1}\n{child_config}"
176
177             draw_tree_on_axis(ax_child, child_tree, child_pos,
title)
178
179     plt.tight_layout(rect=[0, 0, 1, 0.95])
180
181     # --- Salva a figura se um caminho foi fornecido ---
182     if output_path:
183         filename = f"Arvore_{i+1}.png"
184         filepath = os.path.join(output_path, filename)
185         plt.savefig(filepath, dpi=150, bbox_inches='tight')
186
187     # Exibe todos os gráficos na tela após o loop
188     plt.show()
189
190
191 if __name__ == "__main__":
192     input_data = normalized_dict
193
194     # Gera as visualizações e as salva na pasta especificada.

```

195

```
visualize_data_structure(input_data)
```

Capítulo 7

Apêndice B

Este capítulo será dedicado a exibir, explicitamente, os emparelhamentos possíveis associados ao grafo I_8 sobre o 4 – *Toro*, I_6 sobre o 3 – *Toro* e os códigos python usados para a obtenção desses emparelhamentos. Na Subseção 7.1 estão indicados os blocos de código usados no Capítulo 3. Na subseção 7.2 estão os possíveis emparelhamentos do I_6 sobre o 3 – *Toro* e em cada página teremos no máximo 9 pares, contendo os diagramas de emparelhamento e palavras associadas, como as indicações do algoritmo chuva de vértices. Já na Subseção 7.3 estão os possíveis emparelhamentos do I_8 sobre o 4 – *Toro* na mesma disposição apresentada na subseção 7.2.

Para os códigos desse capítulo, usamos a versão 3.12.6 do python e as seguintes versões das bibliotecas: collections, os, pickle, concurrent.futures - Nativas, typing-inspection==0.4.1, typing-extensions==4.13.2, matplotlib==3.10.1.

7.1 Blocos de código

Bloco Python 7.1: Gera todas as palavras W de $2n$ sílabas sem sílabas consecutivas

```

1 def generate_words(n):
2     """
3     Gera todas as palavras W de 2n sílabas, cada Ai aparecendo
4     exatamente duas vezes,
5     sem permitir Ai Ai consecutivos (nem do fim para o início).
6     """
7     counts = Counter({f"A{i}": 2 for i in range(1, n+1)})
8     def backtrack(prefix):
9         if len(prefix) == 2*n:
10             if prefix[0] != prefix[-1]:

```

```

10         yield tuple(prefix)
11     return
12     for lab in list(counts):
13         if counts[lab] == 0: continue
14         if prefix and prefix[-1] == lab: continue
15         counts[lab] -= 1
16         prefix.append(lab)
17         yield from backtrack(prefix)
18         prefix.pop()
19         counts[lab] += 1
20     return backtrack([])

```

Bloco Python 7.2: Canonicalização completa de uma palavra sob permutações de índices, rotações e reflexões

```

1 def canonical_full(word, n):
2     """
3     Canonicaliza 'word' sob:
4     1) todas as permutações de índices 1..n,
5     2) rotações (T1) e reflexões (parte do diedral).
6     Retorna a tupla mínima.
7     """
8     best = None
9     labels = list(word)
10    for perm in itertools.permutations(range(1, n+1)):
11        # mapeia A i Aperm[i-1]
12        mapping = {f"A{i}": f"A{perm[i-1]}" for i in range(1, n+1)}
13        permuted = [mapping[x] for x in labels]
14        # diedral: rotações de permuted e de rev
15        rev = permuted[::-1]
16        L = len(permuted)
17        for i in range(L):
18            r1 = tuple(permuted[i:]+permuted[:i])
19            r2 = tuple(rev[i:]+rev[:i])
20            for cand in (r1, r2):
21                if best is None or cand < best:
22                    best = cand
23    return best

```

Bloco Python 7.3: Algoritmo chuva de vértices adaptado para palavras com duas ocorrências de cada sílaba

```

1 def chuva_de_vertices(word):
2     """

```

```

3     Algoritmo chuva de vértices adaptado para palavras onde cada Ai
4     aparece DUAS vezes.
5     Retorna uma lista de clusters (cada cluster = lista de posições
6     0..2n-1).
7     """
8     L = len(word)
9     used = [False]*L
10    clusters = []
11    for start in range(L):
12        if used[start]:
13            continue
14        stack = [start]
15        used[start] = True
16        cluster = []
17        while stack:
18            p = stack.pop()
19            cluster.append(p)
20
21            # 1)  X  = word[p]          encontre a OUTRA ocorrência
22            de X
23            X = word[p]
24            pos = [i for i,x in enumerate(word) if x==X]
25            other = pos[1] if pos[0]==p else pos[0]
26            p1 = (other - 1) % L
27            if not used[p1]:
28                used[p1] = True
29                stack.append(p1)
30
31            # 2)  Y  = word[p+1]        encontre a OUTRA ocorrê
32            ncia de Y
33            Ypos = (p+1) % L
34            Y = word[Ypos]
35            posY = [i for i,x in enumerate(word) if x==Y]
36            otherY = posY[1] if posY[0]==Ypos else posY[0]
37            p2 = otherY
38            if not used[p2]:
39                used[p2] = True
40                stack.append(p2)
41
42        clusters.append(sorted(cluster))
43    return clusters

```

Bloco Python 7.4: Geração e identificação de palavras com um único cluster, com monitoramento de progresso

```

1 def find_single_cluster_words(n: int, v: int = 1, draw: bool=False)
  -> list[list[str]]:
2     """
3     Gera palavras e identifica as com 1 cluster, monitorando
    progresso.
4     """
5     print(f"=== Iniciando busca para n={n} ===")
6     counts = Counter({f"A{i}":2 for i in range(1,n+1)})
7     counts["A1"] -= 1
8     prefix=["A1"]
9     seen=set(); singles=[]
10    total_checked=0
11
12    def backtrack():
13        nonlocal total_checked
14        if len(prefix)==2*n:
15            if prefix[-1]==prefix[0]: return
16            total_checked+=1
17            if total_checked%100000==0:
18                print(f" ~ {total_checked} prefix completados...")
19            pat=canonical_pattern(prefix)
20            if pat in seen: return
21            seen.add(pat)
22            if len(chuva_de_vertices(prefix))== v:
23                singles.append(prefix.copy())
24            return
25        prev=prefix[-1]
26        for lab in counts:
27            if counts[lab]==0 or lab==prev: continue
28            counts[lab]-=1; prefix.append(lab)
29            backtrack()
30            prefix.pop(); counts[lab]+=1
31
32    backtrack()
33    print(f"=== Buscando concluída: {total_checked} sequências
    testadas, {len(singles)} válidas ===")
34
35    if draw:
36        chord_indices=list(range(1,n+1))
37        for w in singles:

```

```

38         cl=chuva_de_vertices(w)
39         draw_chuva(w, cl, chord_indices)
40         draw_arrows(w, cl)
41     return singles

```

Bloco Python 7.5: Geração de todos os prefixos de comprimento n com valores em $\{1, \dots, q\}$, sem repetições adjacentes e com no máximo duas ocorrências de cada elemento.

```

1
2 from collections import Counter
3
4 def gerar_prefixos(q: int, n: int) -> list[tuple[int, ...]]:
5     """
6     Gera todas as sequências de tamanho n, compostas por valores
7     entre 1 e q,
8     de modo que cada elemento apareça no máximo duas vezes e não
9     haja repetições
10    adjacentes.
11
12    Parâmetros:
13    -----
14    q (int): valor máximo (inclusivo) do alfabeto, elementos de 1 a
15    q.
16    n (int): comprimento das sequências.
17
18    Retorna:
19    -----
20    List[tuple[int, ...]]: cada tupla é uma sequência válida.
21    """
22    # validações iniciais
23    if n < 0:
24        raise ValueError("n deve ser não-negativo.")
25    if n > 2 * q:
26        raise ValueError(f"n não pode ser maior que 2*q={2*q}, pois
27        excederia o limite de 2 ocorrências por elemento.")
28
29    # contador de quantas vezes cada valor ainda pode aparecer (má
30    ximo 2)
31    counts = Counter({i: 2 for i in range(1, q + 1)})
32    resultado: list[tuple[int, ...]] = []
33    prefix: list[int] = []

```

```

30     def backtrack():
31         # se já alcançamos o comprimento desejado, salvamos a
sequencia
32         if len(prefix) == n:
33             resultado.append(tuple(prefix))
34             return
35
36         # tenta inserir cada valor de 1 a q
37         for val in range(1, q + 1):
38             # não há vagas para este valor?
39             if counts[val] == 0:
40                 continue
41             # evita repetição adjacente
42             if prefix and prefix[-1] == val:
43                 continue
44
45             # faz a escolha
46             counts[val] -= 1
47             prefix.append(val)
48             backtrack()
49             # desfaz a escolha
50             prefix.pop()
51             counts[val] += 1
52
53     backtrack()
54     return resultado
55
56 # Exemplo de uso
57 if __name__ == "__main__":
58     q, n = 8, 8
59     prefixos = []
60     for seq in gerar_prefixos(q, n):
61         prefixos.append(tuple(["A" + str(i) for i in seq]))
62         print(tuple(["A" + str(i) for i in seq]))
63

```

Bloco Python 7.6: Filtragem de sequências únicas sob equivalência por swap, com salvamento periódico de progresso

```

1 import os
2 import pickle
3 def is_swap_equivalent_fast(s1: tuple[int, ...], s2: tuple[int,
...]) -> bool:

```

```

4      # 1) Se são idênticas, não contam como swap
5      if s1 == s2:
6          return False
7
8      # 2) Coletar apenas as diferenças (a = s1[k], b = s2[k])
9      diffs: list[tuple[int, int]] = []
10     for a, b in zip(s1, s2):
11         if a != b:
12             diffs.append((a, b))
13
14     # 3) Se não houver diferença, não é swap
15     if not diffs:
16         return False
17
18     # 4) Defino i = rótulo em s2, j = correspondente em s1 na
19     primeira diferença
20     j, i = diffs[0] # note: diffs[0] = (a,b) onde a=s1, b=s2, então
21     o i=b, j=a
22
23     # Se por acaso i==j (não devia), logo não é válido
24     if i == j:
25         return False
26
27     # 5) Verifico se todas as diferenças batem com o swap i<->j
28     for a, b in diffs:
29         # único caso permitido é b==i e a==j, ou b==j e a==i
30         if (b == i and a == j) or (b == j and a == i):
31             continue
32         return False
33
34     # Se passou em tudo, é equivalente por exatamente esse swap
35     return True
36
37 #Exemplo de uso
38 save_dir = os.path.join(drive_base, 'progresso_swap')
39 os.makedirs(save_dir, exist_ok=True)
40
41 #Inicialização
42 filtradas: list[tuple[int, ...]] = []
43 append = filtradas.append
44 is_swap = is_swap_equivalent_fast # sua função de comparação

```

```

44 total = len(prefixos)
45 interval = max(1, total // 100)
46
47 # 4) Loop de filtragem com salvamento periódico
48 for idx, seq in enumerate(prefixos, start=1):
49     # aplica filtro de swap
50     if not any(is_swap(seq, ex) for ex in filtradas):
51         append(seq)
52
53     # a cada 1% (ou no final), salva o estado atual de 'filtradas'
54     if idx % interval == 0 or idx == total:
55         pct = (idx * 100) // total
56         print(f"[{idx}/{total}] ({pct}%) processados, {len(
57             filtradas)} únicos até agora")
58         pkl_path = os.path.join(save_dir, f'filtradas_{pct}.pkl')
59         with open(pkl_path, 'wb') as f:
60             pickle.dump(filtradas, f)
61
62 print("Processamento concluído.")
63

```

Bloco Python 7.7: Geração de palavras de tamanho $2n$ com exatamente v clusters, iniciando por um prefixo opcional e com opção de plotagem

```

1 def find_single_cluster_words_prefixed(
2     n: int,
3     v: int = 1,
4     prefix: list[str] | None = None,
5     draw: bool = False
6 ) -> list[list[str]]:
7     """
8     Gera palavras de tamanho 2n com exatamente v clusters de 'chuva
9     de vertices',
10    iniciando por um prefixo opcional.
11
12    Parâmetros:
13    -----
14
15    n          : número de símbolos A1...An (palavra terá comprimento 2
16    n)
17    prefix     : lista inicial, ex: ["A1","A2","A1"]; se None, usa ["A1
18    "]
19    draw      : se True, plota cada palavra válida com draw_chuva e

```

```

draw_arrows

16
17     Retorna:
18     -----
19     Lista de palavras completas (listas de str) com v clusters.
20     """
21     # Inicializa contagem: cada Ai deve aparecer 2 vezes
22     counts = Counter({f"A{i}": 2 for i in range(1, n + 1)})
23
24     # Configura ou valida o prefixo
25     if prefix is None:
26         prefix = ["A1"] # Simetria: fixa A1 no
início
27     else:
28         if len(prefix) > 2 * n:
29             raise ValueError(f"Prefixo muito longo (máx {2*n}).")
30         # Garante que não haja pares adjacentes iguais no prefixo
31         for i in range(1, len(prefix)):
32             if prefix[i] == prefix[i - 1]:
33                 raise ValueError("Prefixo contém símbolos
adjacentes iguais.")
34
35         # Desconta ocorrências do prefixo no Counter
36         for lab in prefix:
37             if lab not in counts:
38                 raise ValueError(f"Rótulo inválido no prefixo: {lab}")
39             counts[lab] -= 1
40             if counts[lab] < 0:
41                 raise ValueError(f"Prefixo usa {lab} mais vezes do que
permitido (2).")
42
43         seen = set() # Para deduplicar padrão
es canônicos
44         singles: list[list[str]] = [] # Acumula palavras vá
lidas
45         total_checked = 0 # Contador de sequências
testadas
46
47     def backtrack():
48         nonlocal total_checked
49
50         # Caso base: prefixo já alcançou tamanho 2n

```

```

51     if len(prefix) == 2 * n:
52         # Filtra ciclos: primeiro      último
53         if prefix[-1] == prefix[0]:
54             return
55         total_checked += 1
56         # Log de progresso a cada 100k iterações
57         if total_checked % 100000 == 0:
58             print(f" ~ {total_checked} sequências testadas..."
59 )
60         # Canonicaliza e verifica duplicatas
61         pat = canonical_pattern(prefix)
62         if pat in seen:
63             return
64         seen.add(pat)
65         # Se formar exatamente 1 cluster, salva cópia
66         if len(chuva_de_vertices(prefix)) == v:
67             singles.append(prefix.copy())
68         return
69
70     prev = prefix[-1]                # Símbolo anterior,
71     para evitar repetições
72     # Tenta cada símbolo disponível no Counter
73     for lab, cnt in counts.items():
74         if cnt == 0 or lab == prev:
75             continue                # Pula se não houver
76     ocorrências ou se repetir o anterior
77     counts[lab] -= 1                # Escolhe lab
78     prefix.append(lab)
79     backtrack()                    # Recurssão
80     prefix.pop()                  # Desfaz escolha
81     counts[lab] += 1                # Restaura contador
82
83     # Início da busca
84     print(f"=== Iniciando busca para n={n}, prefix={prefix} ===")
85     backtrack()
86     print(f"=== Busca concluída: {total_checked} sequências
87     testadas, {len(singles)} válidas ===")
88
89     # Se solicitado, plota resultados
90     if draw:
91         chord_indices = list(range(1, n + 1))
92         for w in singles:

```

```

89         cl = chuva_de_vertices(w)
90         draw_chuva(w, cl, chord_indices)
91         draw_arrows(w, cl)
92
93     return singles
94

```

Bloco Python 7.8: Processamento paralelo de prefixes com ProcessPoolExecutor para cálculo de rotações mínimas e agregação de palavras canônicas

```

1  from typing import Tuple, List
2  from concurrent.futures import ProcessPoolExecutor, as_completed
3
4  def minimal_rotation(s: Tuple[int, ...]) -> Tuple[int, ...]:
5      s2 = s + s
6      n = len(s)
7      i, j, k = 0, 1, 0
8      while i < n and j < n and k < n:
9          a, b = s2[i+k], s2[j+k]
10         if a == b:
11             k += 1
12             continue
13         if a > b:
14             i += k + 1
15         else:
16             j += k + 1
17         if i == j:
18             j += 1
19         k = 0
20     start = min(i, j)
21     return tuple(s2[start:start+n])
22
23 def process_prefix(prefix: Tuple[int, ...]) -> List[Tuple[Tuple[int, ...], ...], Tuple[int, ...]]:
24     singles = find_single_cluster_words(
25         8,
26         v=1,
27         prefix=list(prefix),
28         draw=False
29     )
30     out: List[Tuple[Tuple[int, ...], Tuple[int, ...]]] = []
31     for w in singles:
32         w_tup = tuple(w)

```

```

33         can = minimal_rotation(w_tup)
34         out.append((can, w_tup))
35     return out
36
37 if __name__ == '__main__':
38     # prefixos já definido como lista de tuplas
39     total = len(prefixos)
40     interval = max(1, total // 100)      # 1% do total, no mínimo 1
41     processed = 0
42
43     Obtidas: List[Tuple[int, ...]] = []
44     canonical_set: set[Tuple[int, ...]] = set()
45
46     print(f"=== Iniciando {total} tasks em paralelo ===")
47     with ProcessPoolExecutor() as executor:
48         futures = {executor.submit(process_prefix, p): p for p in
49             prefixos}
50
51         for future in as_completed(futures):
52             processed += 1
53             # print a cada 1% (ou no fim)
54             if processed % interval == 0 or processed == total:
55                 print(f"[{processed}/{total}] processados, faltam {
56                     total-processed}")
57                 try:
58                     results = future.result()
59                 except Exception as e:
60                     print(f" Erro em {futures[future]}: {e!r}")
61                     continue
62
63                 for can, w in results:
64                     if can not in canonical_set:
65                         canonical_set.add(can)
66                         Obtidas.append(w)
67
68     print(f"=== Concluído: {len(Obtidas)} classes únicas geradas
69     ===")

```

7.2 Mergulhos do I_6 sobre o 3-Toro

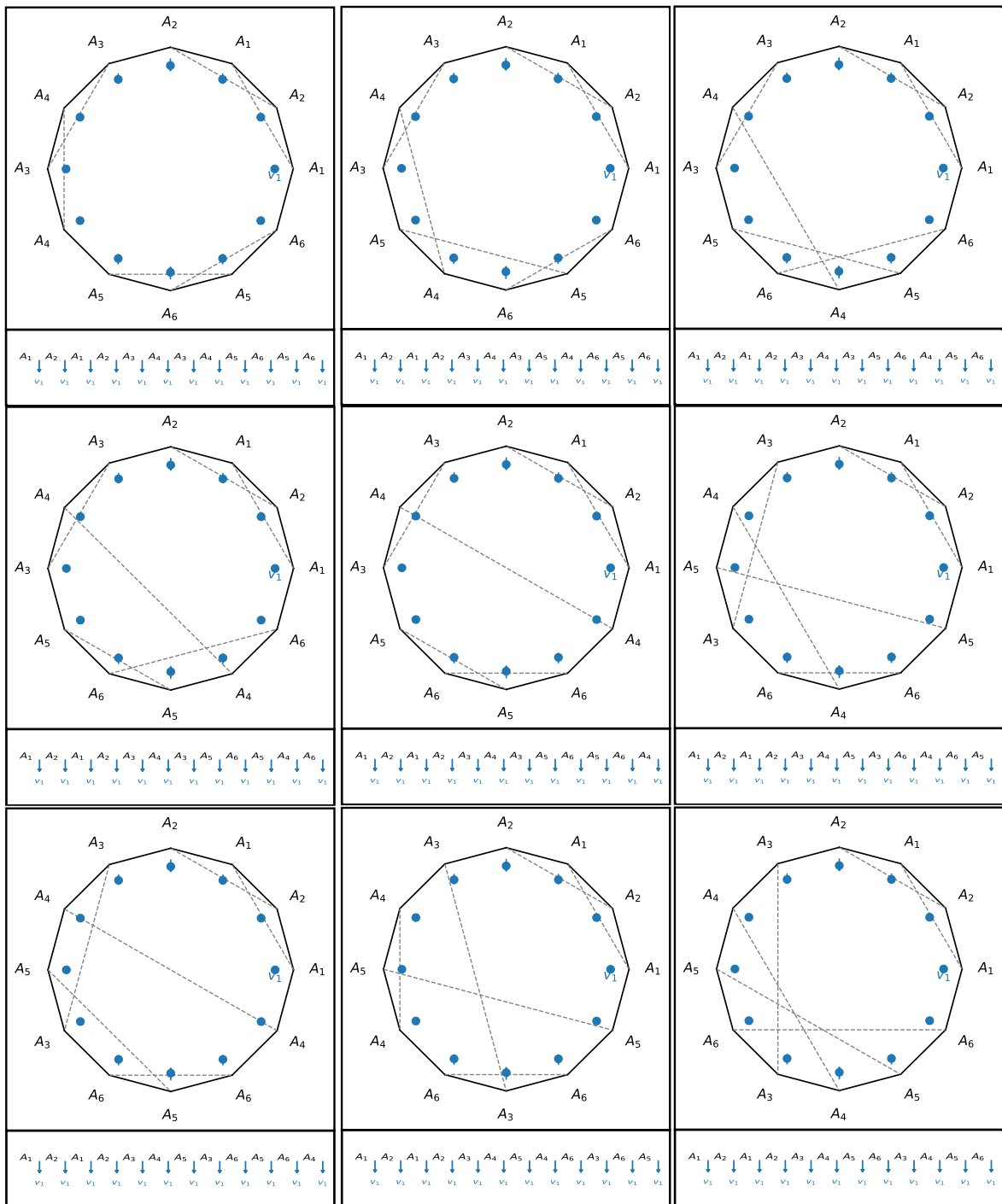


Figura 7.1: Emparelhamentos (1–9) de I_6 sobre o 3-Toro.

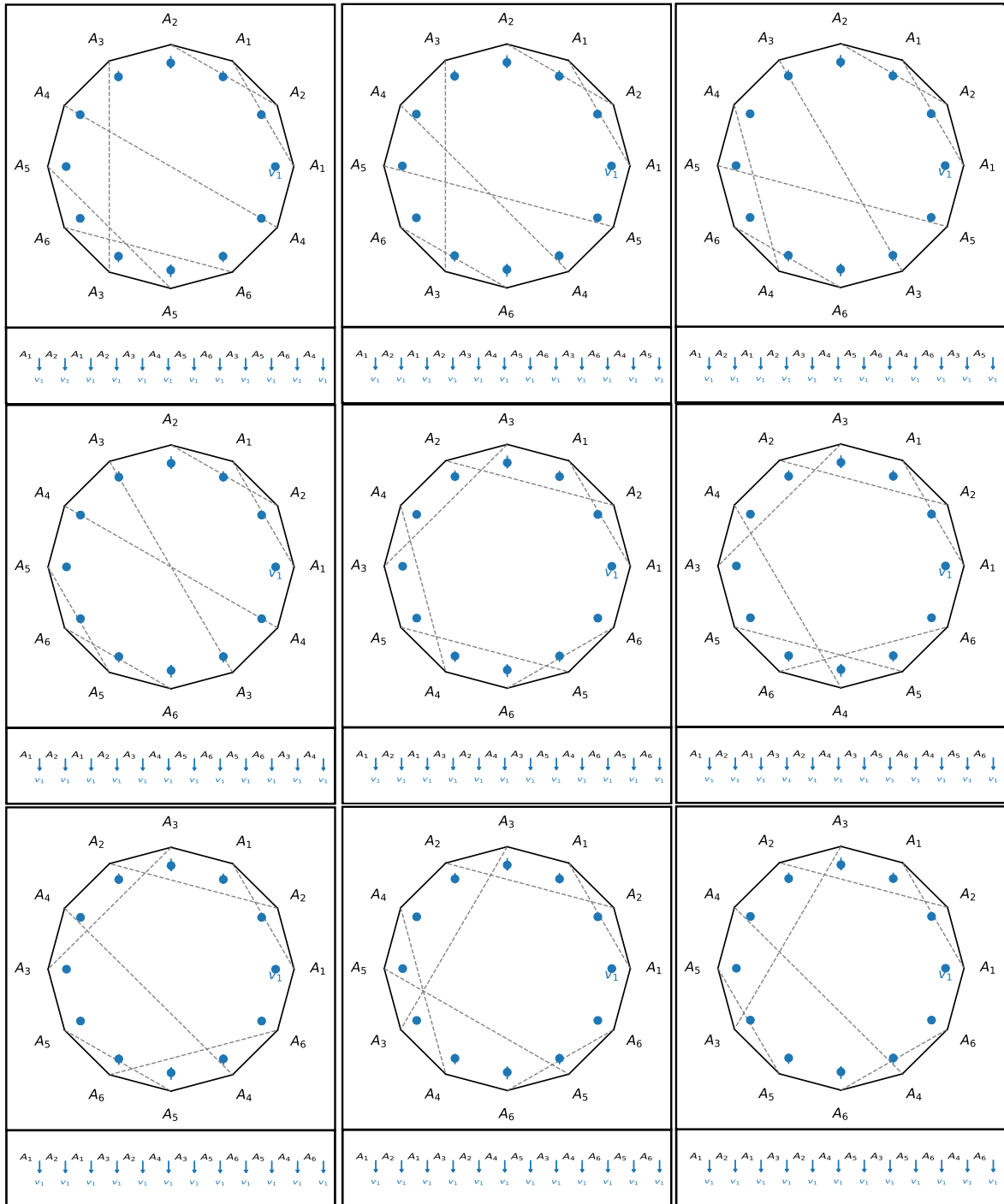


Figura 7.2: Emparelhamentos (10–18) de I_6 sobre o 3-Toro.

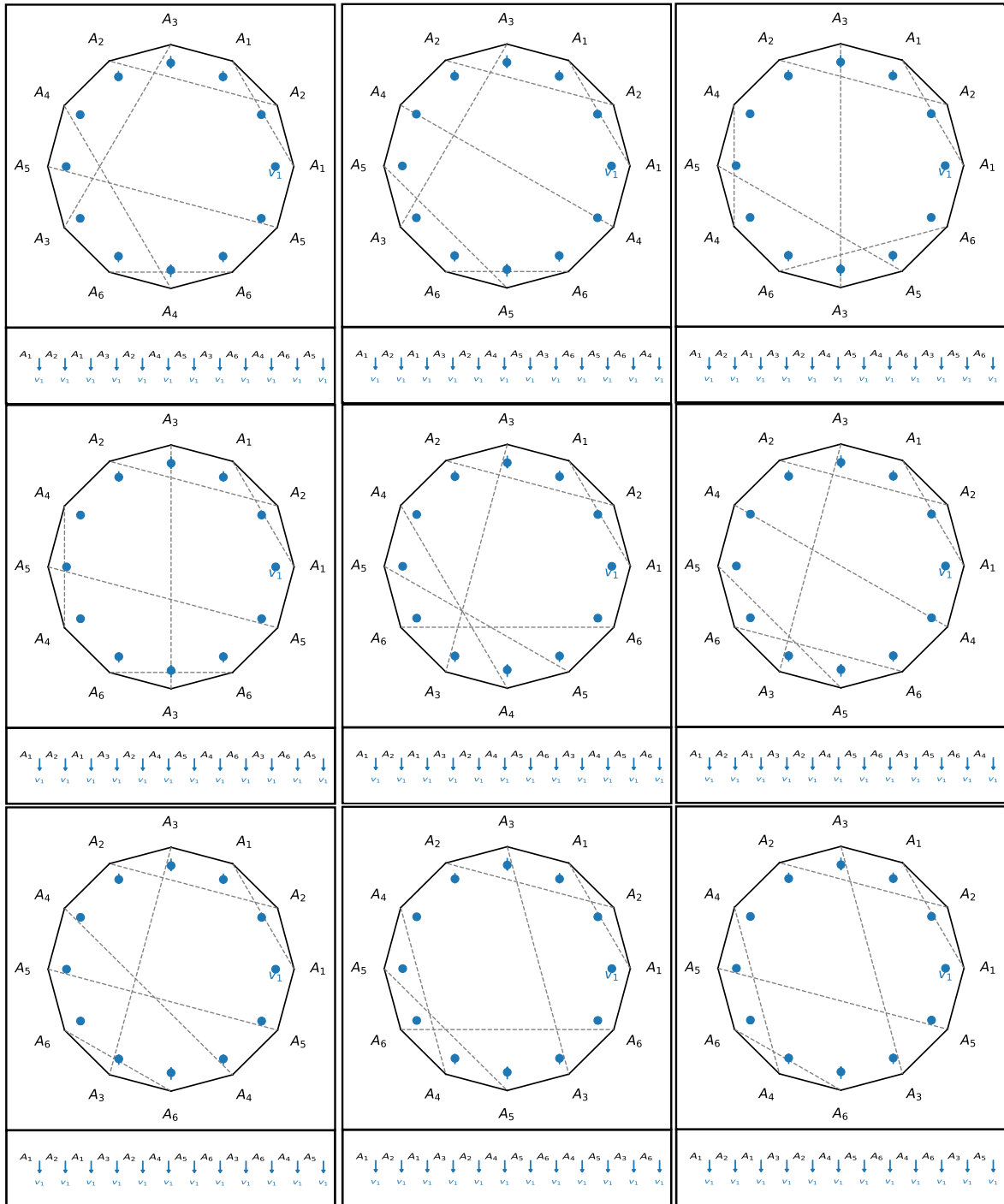


Figura 7.3: Emparelhamentos (19–27) de I_6 sobre o 3-Toro.

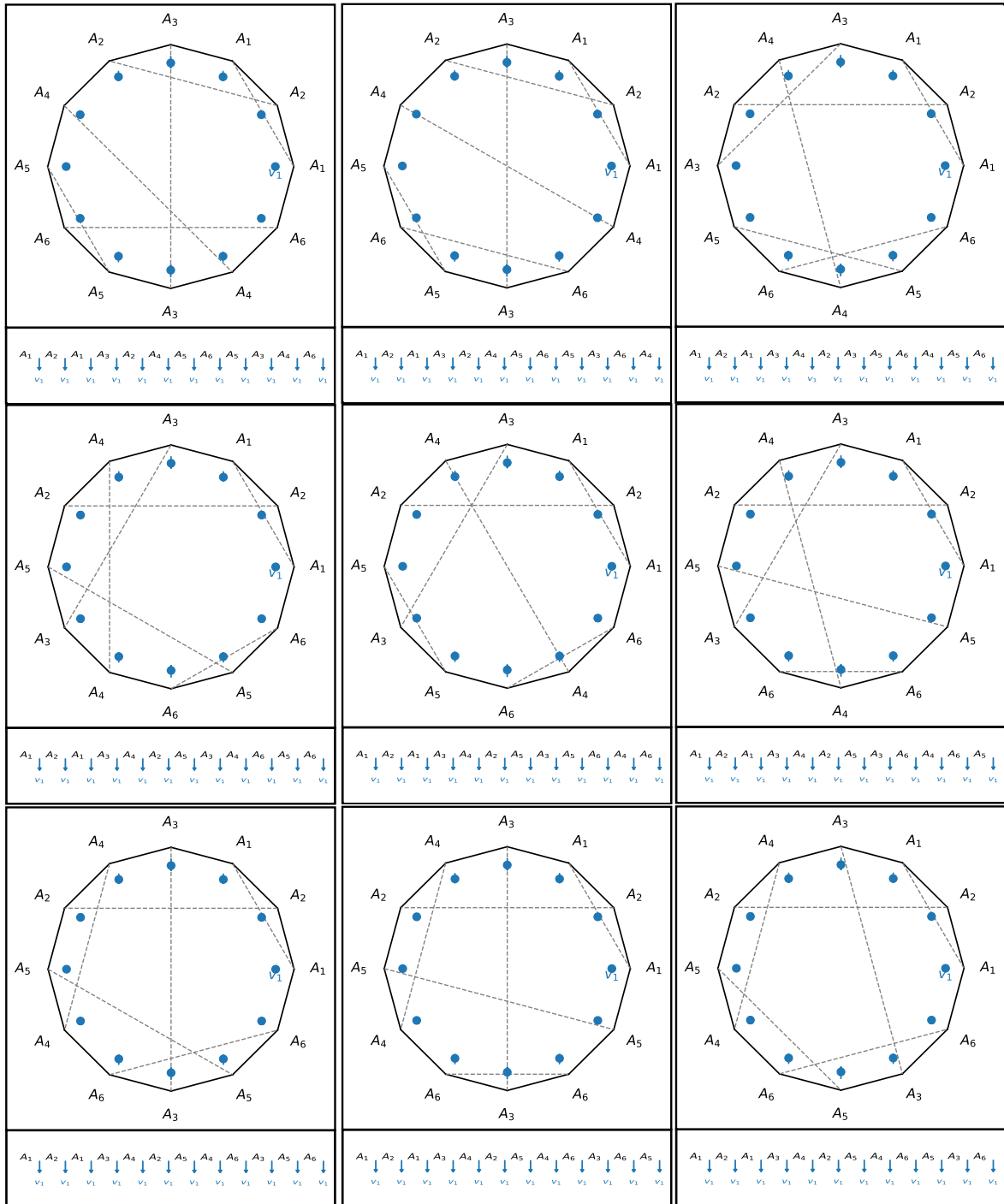


Figura 7.4: Emparelhamentos (28–36) de I_6 sobre o 3-Toro.

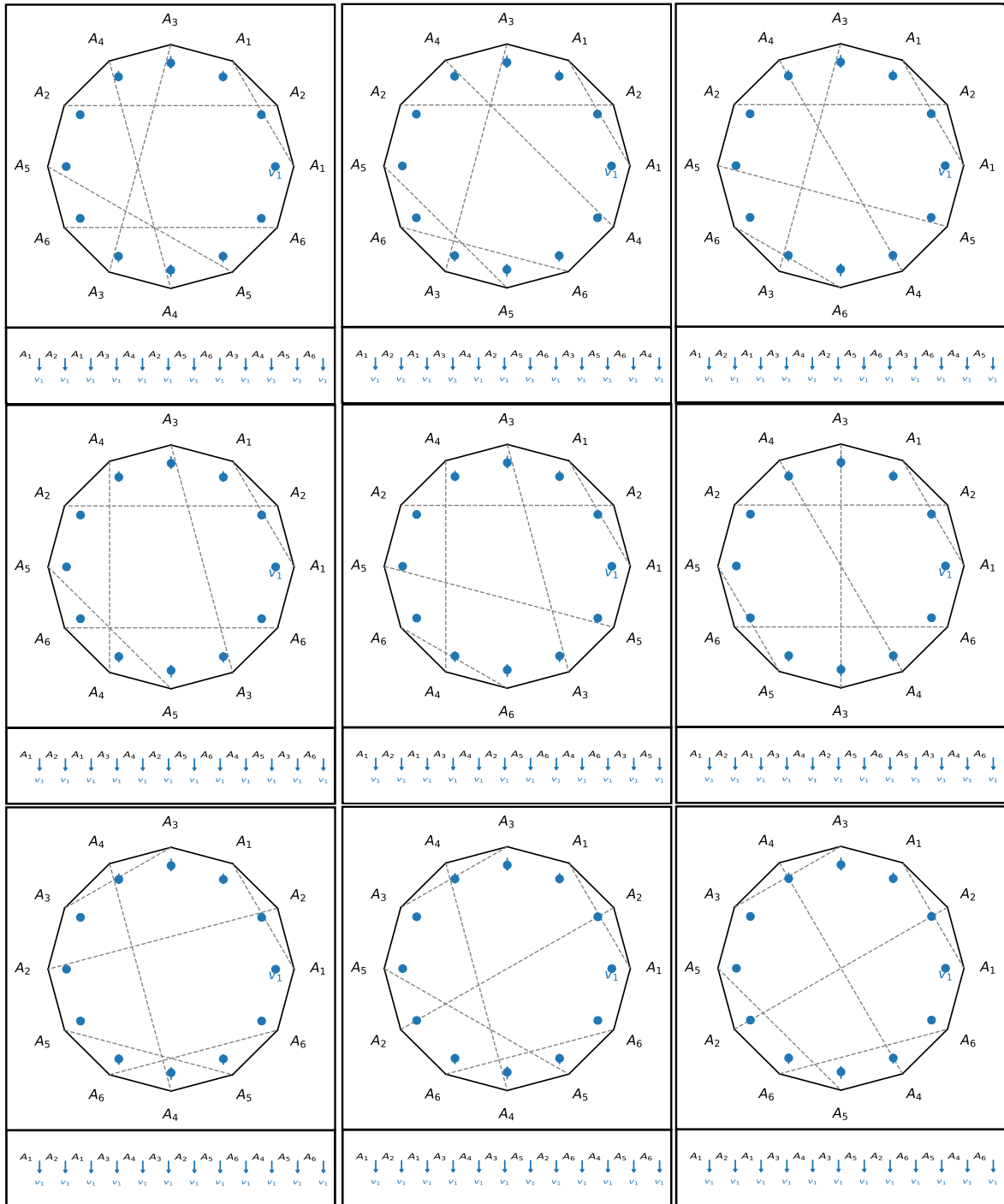


Figura 7.5: Emparelhamentos (37–45) de I_6 sobre o 3-Toro.

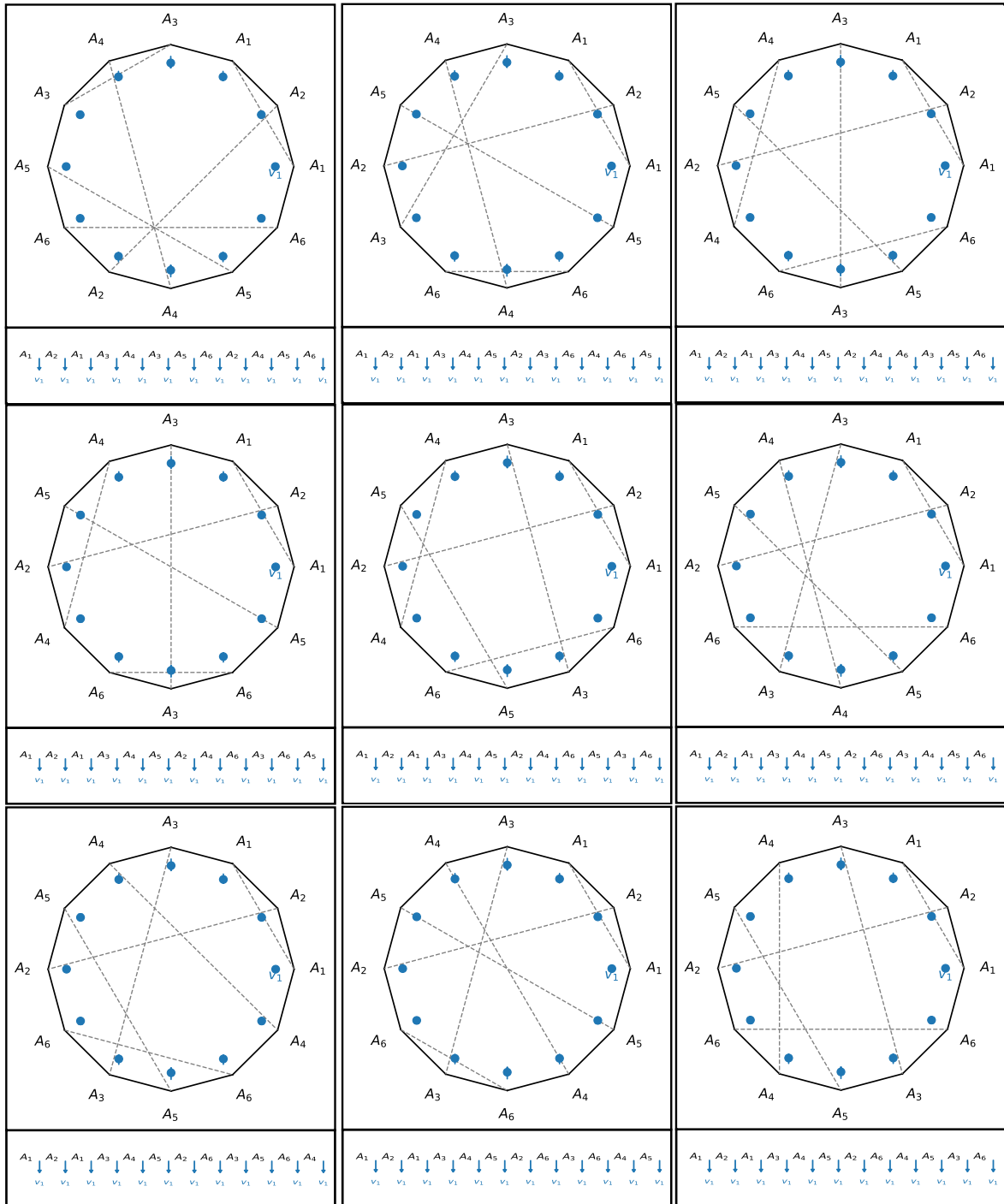


Figura 7.6: Emparelhamentos (46–54) de I_6 sobre o 3-Toro.

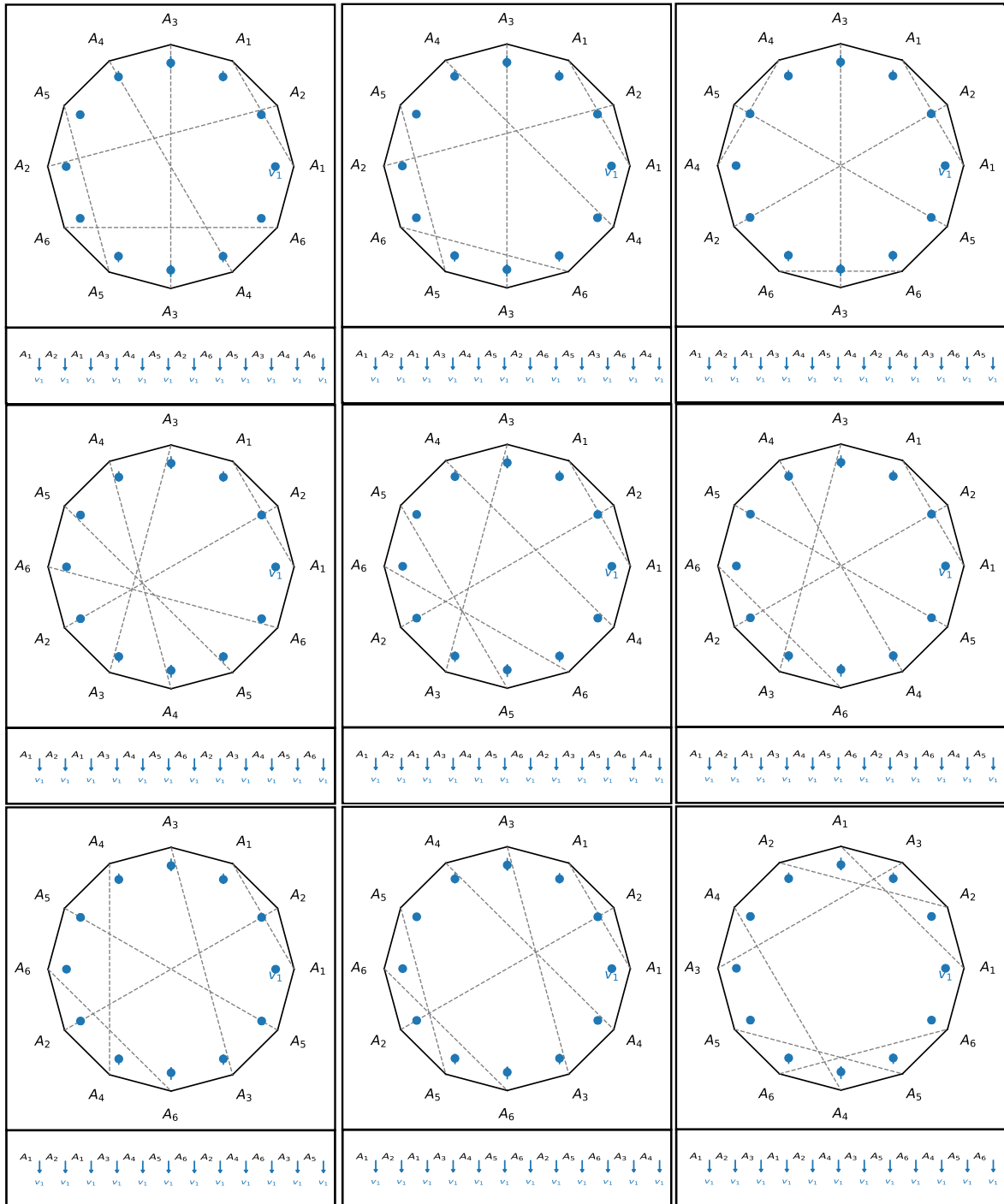


Figura 7.7: Emparelhamentos (55–63) de I_6 sobre o 3-Toro.

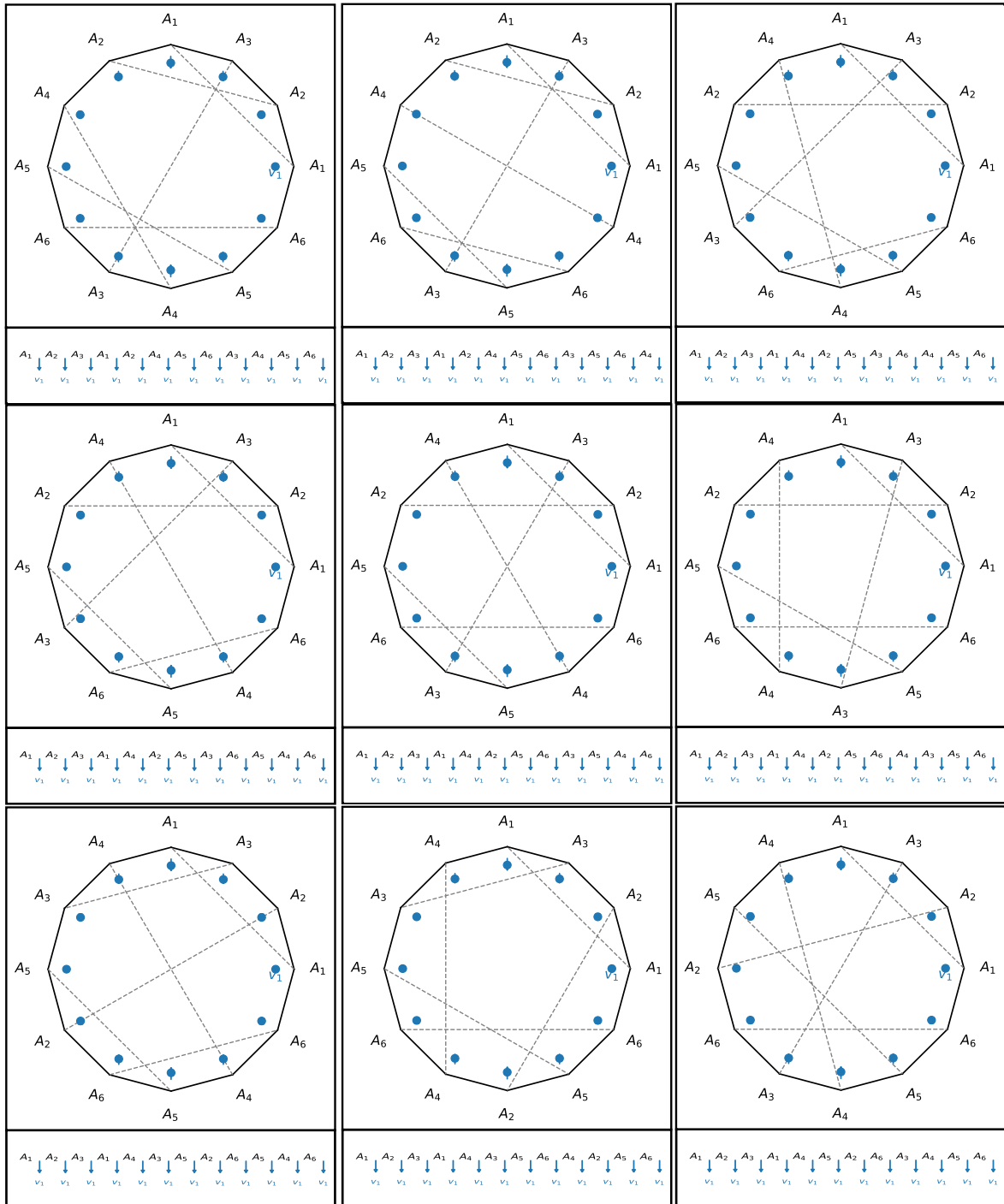


Figura 7.8: Emparelhamentos (64–72) de I_6 sobre o 3-Toro.

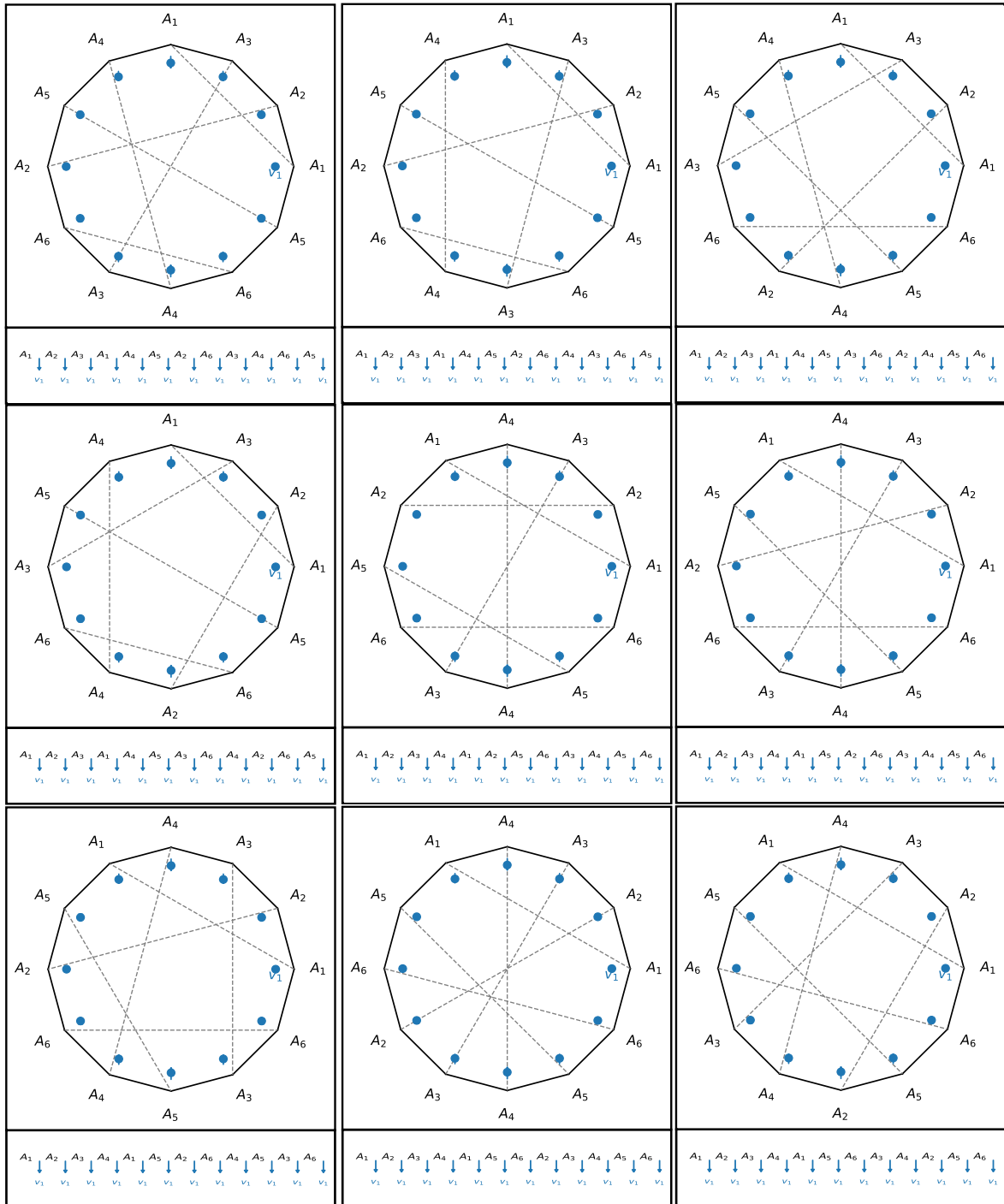


Figura 7.9: Emparelhamentos (73–81) de I_6 sobre o 3-Toro.

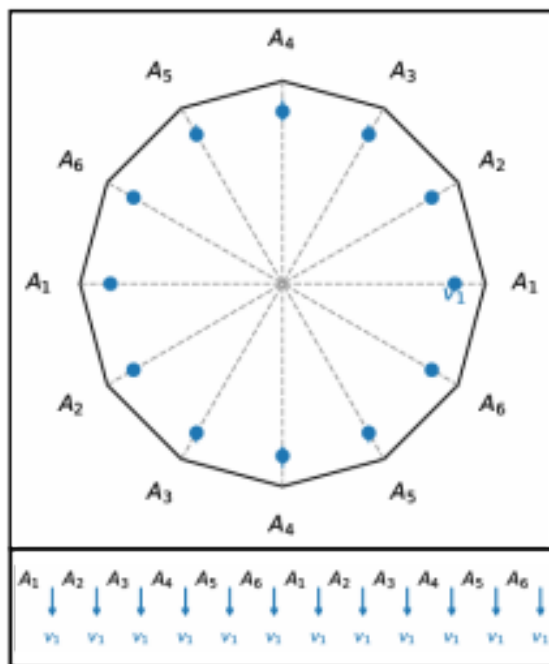


Figura 7.10: Emparelhamento (82-82) de I_6 sobre o tritoro

7.3 Mergulhos do I_8 sobre o 4-Toro

Bloco Python 7.9: Carregamento de sequências processadas de arquivo pickle e conversão para notação A_i - Modifique a linha 5 com o caminho do seu arquivo

```

1 import pickle
2 import os
3
4 # Ajuste para o mesmo diretório onde salvou o pickle --> Arquivo
  baixado do link fornecido
5 pickle_path = r"INCLUA O CAMINHO DO SEU ARQUIVO"
6
7 if not os.path.isfile(pickle_path):
8     raise FileNotFoundError(f"Arquivo não encontrado: {pickle_path}"
9                             ")
10
11 with open(pickle_path, 'rb') as f:
12     visto = pickle.load(f)
13
14 print(f"{len(visto)} sequências carregadas de {pickle_path}")
15
16 visto0 = []

```

```

17 for j in visto:
18     visto0.append(tuple(["A"+ str(i) for i in j]))
19
20 visto0

```

Bloco Python 7.10: Geração de diagramas de chuva e de setas para cada sequência e salvamento em arquivos PDF

```

1 import os
2 import matplotlib.pyplot as plt
3
4 #
5
6 # Parâmetros principais (ajuste conforme necessário):
7 n = 8 # valor de n
8 chord_indices = list(range(1, n+1))
9
10 output_dir = r"imagens_n8" # onde salvar PDFs
11
12 # cria pasta de saída se faltar
13 os.makedirs(output_dir, exist_ok=True)
14
15 # evita travar em plt.show()
16 _orig_show = plt.show
17 plt.show = lambda *a, **k: None
18
19 # 1) gerar e salvar cada par de figuras
20 filenames: list[tuple[str, str]] = []
21 for idx, w in enumerate(visto0, start=1):
22     w_list = list(w)
23     cl = chuva_de_vertices(w_list)
24
25     # diagrama circular
26     draw_chuva(w_list, cl, chord_indices)
27     ch_path = os.path.join(output_dir, f"ch_{idx:03d}.pdf")
28     plt.savefig(ch_path, bbox_inches="tight")
29     plt.close()
30
31     # diagrama de setas
32     draw_arrows(w_list, cl)
33     arr_path = os.path.join(output_dir, f"arr_{idx:03d}.pdf")
34     plt.savefig(arr_path, bbox_inches="tight")

```

```
34     plt.close()
35
36     filenames.append((ch_path, arr_path))
37
38 plt.show = _orig_show # restaura show
39
40 print(f"PDFs gerados em '{output_dir}")
```