UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Camile Frazão Bordini

Isomorfismo em Grafos

Curitiba-PR

2013

CAMILE FRAZÃO BORDINI

ISOMORFISMO EM GRAFOS

Monografia apresentada à disciplina de Trabalho de Conclusão de Curso do departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná, como requisito parcial à obtenção do título de Bacharel.

Orientador: Prof. Dr. Murilo V. G. da Silva

Curitiba-PR

2013

Sumário

Resumo

O presente trabalho tem como objetivo o estudo do problema do Isomorfismo de Grafos. Para tal, um estudo a respeito da Teoria de Grafos e suas propriedades, fazem-se necessários.

Há diversas discussões a respeito que são de interesse de áreas relacionadas com a ciência da computação. Desta forma, a contribuição à comunidade acadêmica que este trabalho pretende alcançar é a principal motivação para a sua realização.

O trabalho seguirá em duas frentes, primeiramente uma teórica e conceitual a respeito da teoria de grafos e complexidade algorítmica, para que tenhamos um ferramental matemático para tratar adequadamente o problema do isomorfismo em grafos.

Em um segundo momento será analisado o problema do isomorfismo e suas especificidades, e, por fim, a criação de um exemplo de como se comportaria este problema para uma classe específica de grafos em comparação com o caso geral.

Palavras-chave: Grafos, Complexidade, Isomorfismo.

Capítulo 1

Introdução

A Teoria dos Grafos tem sido utilizada em vários ramos da ciência pela sua capacidade de representar elementos de uma determinada natureza e suas relações, como por exemplo, o estudo de associações entre elementos em diferentes ramos da ciência como a genética e a química, dentre outros.

Muitas situações reais podem ser descritas por diagramas que consistem em um conjunto de pontos e linhas que agrupam alguns desses pontos. Desta forma os pontos poderiam representar, por exemplo, pessoas e as linhas representariam a união entre elas (BONDY; MURTY, 1976). Outro exemplo prático que podemos citar é a análise de características de uma rede, que permite o desenvolvimento de novas estratégias em diferentes áreas do conhecimento onde possa ser aplicada, como por exemplo, o desenvolvimento de novas estratégias de prevenção contra vírus na maior rede mundial de computadores, ou ainda o mapeamento de redes cerebrais e suas reações bioquímicas entre as células (BARABASI, 2003).

Diversos problemas práticos podem ser modelados como problemas em grafos, dentro os mais importantes destaca-se o Problema do Isomorfismo, que é o foco deste trabalho. O problema do isomorfismo tem importância não somente prática mas também teórica, que é o foco principal deste trabalho. Em particular, este é um raro problema para o qual não se conhece nenhum algoritmo polinomial para a sua resolução, mas por outro lado há fortes evidências de que o problema também não seja NP-Completo (ARORA e BARAK, 2009).

1.1 Justificativa

Neste trabalho será abordado um tema de natureza teórica e científica, e, portanto, o objetivo é produzir um texto científico abordando o assunto do isomorfismo de grafos.

A produção de um texto científico a respeito do problema do isomorfismo de grafos contribuirá, em cunho didático, com a comunidade científica, sendo esta a principal justificativa para o desenvolvimento deste trabalho.

1.2 Objetivos

1.2.1 Objetivo Geral

Analisar o problema do isomorfismo de grafos abordando suas especificidades e estado da arte, além do estudo de como se comportaria este problema para uma classe específica de grafos, no caso os grafos *Split*.

1.2.2 Objetivos Específicos

Este trabalho possui como objetivos específicos:

- Desenvolver um material teórico a respeito de dois temas principais, a Teoria de Grafos, tendo como foco de estudo o isomorfismo em grafos, e a Teoria de Complexidade Computacional;
- Realizar o embasamento teórico do estado da arte do problema do isomorfismo em grafos;
- Detalhar um exemplo encontrado pela autora deste trabalho juntamente com seu orientador de uma classe específica de grafos em que o problema do isomorfismo continua difícil de resolver assim como no caso geral.

1.3 Metodologia

Para uma melhor organização, o trabalho foi subdividido em duas grandes etapas. Em cada uma serão realizadas atividades interdependentes e que se relacionam para a obtenção do objetivo final.

Na primeira etapa será abordado a respeito da Teoria de Grafos, conceituando suas principais características, definições e as principais classes de grafos que influenciam o problema do isomorfismo. Também será estudado o tema relacionado à Teoria de Complexidade Computacional, seus conceitos, assim como a análise das principais classes de complexidade de algoritmos, como P, NP, e sua relação com o problema do isomorfismo.

Na segunda etapa, a ênfase estará no estudo do problema do isomorfismo propriamente dito, onde primeiramente será analisado seu estado da arte (definições, aplicações práticas e algoritmos existentes para o problema), e posteriormente será mostrado um exemplo encontrado para uma classe específica de grafos em que o problema do isomorfismo continua difícil de resolver assim como no caso geral.

Uma descrição ilustrada a respeito dessas atividades está representada na Figura 1.1.

1.4 Organização do Trabalho

Este trabalho está organizado da seguinte maneira: no capítulo 2 encontram-se conceitos sobre a teoria de grafos e a de complexidade computacional. No capítulo 3 é abordado o objeto deste trabalho, ou seja, primeiramente é analisado o estado da arte de isomorfismo em grafos e posteriormente a forma de operação de um dos algoritmos mais utilizados atualmente para a sua resolução, denominado *Nauty* (MCKAY, 2013). Por fim, o capítulo 4 apresenta um exemplo realizado para este trabalho mostrando

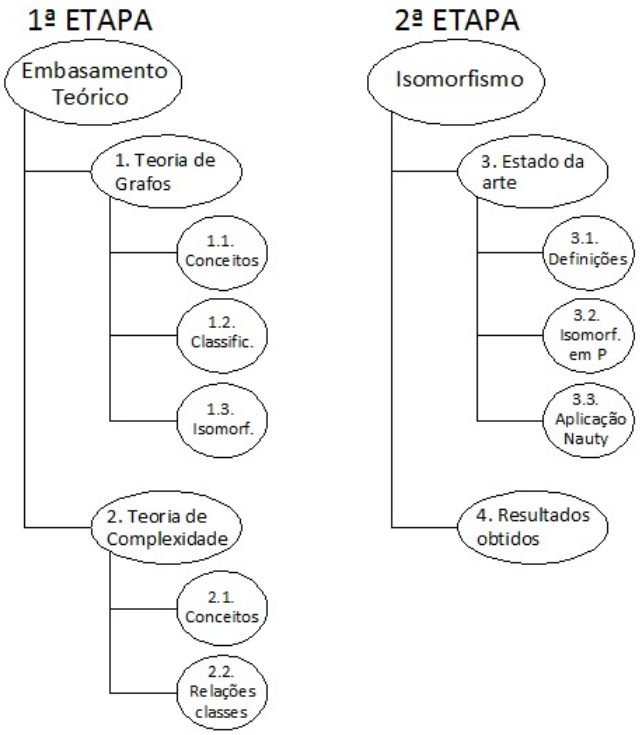


Figura 1.1: Estrutura analítica do projeto.

Fonte: Própria

uma classe específica de grafos escolhida em que o problema permanece difícil, ou seja, a resolução do problema, mesmo nesta classe restrita de grafos, é tão difícil como o problema em um grafo qualquer.

Capítulo 2

Referencial Teórico

Os conceitos necessários para tratar do problema de isomorfismo de grafos requer um estudo a respeito da Teoria de Grafos. Assim, definições clássicas, principais características e sua utilidade na resolução de problemas reais serão apresentadas neste Capítulo.

Além disso, faz-se necessário também um embasamento teórico a respeito de algoritmos que se comportam em tempo polinomial, ou seja, definições e características a respeito da Teoria de Complexidade serão abordadas num segundo momento.

Desta forma, nesta seção serão apresentados conceitos introdutórios dentro destes temas para que uma fundamentação do estado da arte possa ser apresentada.

2.1 Grafos

Grafos são abstrações matemáticas utilizadas para representar situações reais, onde os vértices representam alguma informação e as arestas os relacionamentos entre dois nós. Não há uma maneira única de desenhar um grafo, as posições absolutas dos vértices e arestas não possuem nenhum significado, desta forma, um grafo retrata meramente a relação de incidência entre esses vértices e arestas (BONDY; MURTY, 1976).

As definições e terminologias básicas de grafos apresentadas neste trabalho seguem as adotadas comumente na literatura, como nos livros de Bondy e Murty (1976), West (2002) e Golumbic (2004).

Formalmente, um grafo G é um par ordenado $G = (V, E)$ que consiste em um conjunto V não vazio de nós, ou vértices, um conjunto E de arestas. Normalmente denotamos uma aresta $\{v_1, v_2\}$ apenas por v_1v_2. Também costumamos denotar o conjunto de vértices de um grafo G por $V(G)$ e o conjunto de arestas por $E(G)$.

Se v_1v_2 é uma aresta de $E(G)$, dizemos que v_1 é adjacente a v_2, por exemplo: na Figura 2.1, os vértices 2 e 3 são adjacentes porém 2 e 4 não são, assim como dizemos que a aresta v_1v_2 é **incidente** a v_1 e v_2.

Um grafo é dito **não-direcionado** se sua relação de adjacência é simétrica, ou seja, se $E = E^{-1}$ (Figura 2.1).

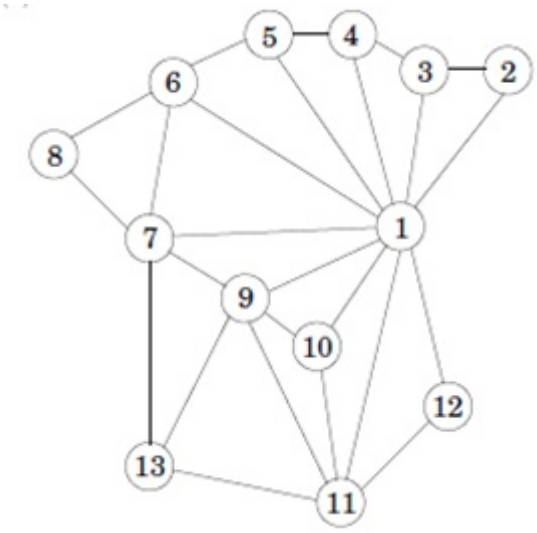


Figura 2.1: Exemplo de grafo não-direcionado.
Fonte: Dasgupta et al (2006, p.92).

Em grafos **direcionados** (ou dígrafos), arestas são pares ordenados $(v_1 v_2)$ onde cada aresta tem um sentido ou orientação (Figura 2.2). Assim, se $(u, v) \in E(G)$, dizemos que (u, v) é uma aresta de u para v. No caso de arestas direcionadas também usamos a notação uv.

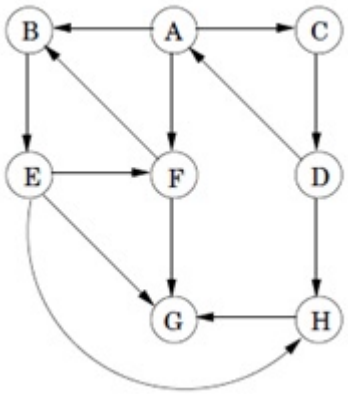


Figura 2.2: Exemplo de grafo direcionado.
Fonte: Dasgupta et al (2006, p.99).

Além do direcionamento, há diversas propriedades a respeito de grafos. Por exemplo, para alguns problemas pode ser necessário associar um peso (valor numérico) a cada aresta que liga um vértice a outro indicando alguma propriedade (peso, distância, etc.), neste caso diz-se que se trata de um **grafo ponderado**. O **grau** de um nó é o número de arestas incidentes a este nó.

Um grafo é dito **completo** se todo par de vértices distintos são adjacentes entre si, e um grafo completo com n vértices é denotado por K_n. Um **subgrafo** de um grafo $G = (V, E)$ é um grafo $H = (V', E')$ que satisfaz $V' \subseteq V$ e $E' \subseteq E$. Um tipo de subgrafo com particular importância é chamado de

"subgrafo induzido" por um conjunto de vértices, o qual será descrito melhor quando forem analisadas algumas classes de grafos específicas.

O **complemento** de G é um grafo H tais que dois vértices em H são adjacentes se e somente se eles não forem adjacentes em G. Um grafo é dito **conexo** se existe um caminho entre cada par de vértices, caso contrário, G é **desconexo**.

Um **caminho** em um grafo é uma sequência finita e não vazia de nós intercalados com arestas, onde um caminho deve obrigatoriamente iniciar e terminar em algum nó (não podendo iniciar ou terminar em uma aresta). Se há um caminho do nó v_0 à v_k, dizemos que v_0 é a origem e v_k é o destino, e v_1, v_2,..., v_{k-1} são nós internos. O inteiro k é o comprimento do caminho.

Um **caminho simples** em um grafo é quando nenhum vértice ocorre mais do que uma vez. Podemos definir ainda, a respeito do **comprimento** de um caminho, que é o número de arestas que um grafo não valorado contém. Caso haja repetição de uma aresta conta-se cada vez em que esta é usada. No caso de grafos ponderados é a soma dos pesos de percorrer cada aresta.

Um **ciclo** em um grafo é caracterizado através de um caminho fechado, ou seja, a origem e destino se tratam do mesmo vértice, e nenhum outro nó é repetido mais de uma vez. Um ciclo de tamanho k é chamado *k-ciclo*, por exemplo, um *3-ciclo* é chamado frequentemente de *triângulo*. Por último, um grafo sem ciclos é dito **acíclico**.

Um grafo é chamado de **esparso** se a quantidade arestas for da ordem de $O(|V|)$, e no caso contrário, quando o número de arestas for da ordem de $|V|^2$, diz-se que esse grafo é **denso**.

Por fim, alguns conceitos a respeito de multigrafo. Um **multigrafo** é um grafo não orientado que possui **arestas paralelas** (com as mesmas extremidades) e **laços** (arestas cujas extremidades são o mesmo nó). Assim, grafos que não possuem laços nem arestas paralelas são denominados **grafos simples**.

2.1.1 Classificação de Grafos

Muitos problemas podem ser tratados por algoritmos mais eficientes se reduzirmos o problema a uma classe de grafos mais específica. Por isso uma breve introdução a algumas classes particulares de grafos faz-se necessário.

- Grafo Bipartido

Um grafo não direcionado $G = (V, E)$ é dito *bipartido* se seus vértices puderem ser divididos em dois conjuntos disjuntos não vazios (S_1 e S_2), onde apenas nós pertencentes a conjuntos diferentes são adjacentes, em outras palavras, toda aresta de G possui uma extremidade em S_1 e outra em S_2. Costumamos utilizar a notação $G = (S_1, S_2, E)$ para enfatizar a partição.

Um grafo bipartido $G = (S_1, S_2, E)$ é dito "completo" se possui uma aresta para cada par de vértices $v_i \in S_1$ e $v_j \in S_2$, ou seja, todas as possibilidades de ligações entre os nós são satisfeitas.

Se a quantidade de nós de um conjunto for a e do outro b, este é denotado por $K_{a,b}$ (Figura 2.3).

- Grafo Planar

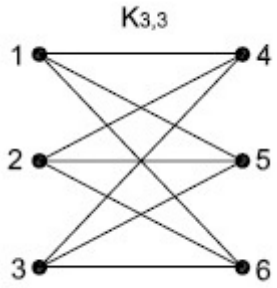


Figura 2.3: Exemplo de grafo bipartido.

Fonte: Própria.

Um grafo planar é aquele que pode ser desenhado no plano sem que duas arestas se cruzem. Uma representação planar divide o plano em regiões chamadas faces, onde arestas limitam e determinam uma face (Figura 2.4).

De acordo com a teoria de *Kuratowski* (BONDY; MURTY, 1976), um grafo é planar se e somente se ele não contém um K_5 ou um $K_{3,3}$ como *minor* (ou seja, quaisquer subdivisões suas). Outro fato sobre grafos planares, diz respeito à descoberta de *Euler*, sobre a relação entre o número de vértices (n), o de arestas (a) e o de regiões (r) em um grafo planar, em que nomeou essa relação de **fórmula de Euler**: $\mathbf{n - a + r = 2}$.

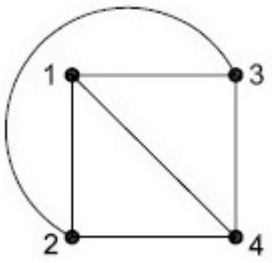


Figura 2.4: Exemplo de grafo planar.

Fonte: Própria.

- Grafos Perfeitos

Esta classe é importante por diversos motivos como, por exemplo, há muitos problemas de interesse prático mas intratáveis, que podem ser resolvidos eficientemente quando restritos à classe de grafos perfeitos.

Para a caracterização de grafos perfeitos, primeiramente é preciso entender sobre uma clique em um grafo, sobre coloração de grafos e o que vem a ser um subgrafo induzido.

Uma clique em um grafo é um subgrafo completo, ou seja, um conjunto de nós em que todos estejam conectados entre si. Podemos denotar como K_n como sendo um grafo completo com n vértices, ou ainda uma *n-clique*.

A coloração de grafos trata da atribuição de rótulos (ou cores) a elementos de um grafo (que podem ser seus vértices, suas arestas ou ainda suas faces) de forma que não haja dois elementos adjacentes com a mesma cor. No caso da coloração de vértices, que é o tipo de coloração ligado aos grafos perfeitos, a coloração deve seguir uma lógica tal que dois vértices adjacentes não compartilhem a mesma cor. Da mesma forma ocorre para a coloração de arestas e de faces.

Uma observação importante é que no estudo da coloração de grafos onde se procura obter o menor número possível de cores ("número cromático"), observa-se que dado um grafo G, o seu número cromático $\chi(G)$ deve ser maior ou igual ao tamanho da maior clique desse grafo, $\omega(G)$, pois todos os vértices de sua clique máxima devem possuir cores diferentes, uma vez que estão todos ligados entre si, (pré-requisito para se ter uma clique).

Por último, temos o conceito de subgrafo induzido. Um subgrafo induzido de um grafo G é um grafo cujo conjunto de vértices é um subconjunto do conjunto de vértices de G e o mesmo ocorre com suas arestas, ou seja, se para qualquer par de vértices do subgrafo, uma aresta entre esses vértices deve ocorrer se, e somente se, ela também ocorre em G. Em outras palavras, um subgrafo é dito induzido de G se ele tem todas as arestas que ocorrem em G referentes ao mesmo conjunto de vértices.

Assim, finalmente será possível definir um grafo perfeito. Todo grafo G é dito perfeito se o tamanho da sua maior clique é igual ao seu número cromático, e isso permanece verdade para todos os seus subgrafos induzidos. Em outras palavras: $\omega(H) = \chi(H)$ para todo subgrafo induzido H de G (Figura 2.5).

Desta forma, como dito anteriormente, como o número da clique fornece um limite inferior para o número cromático, um grafo perfeito é aquele em que este limite inferior é bastante rígido, não só no grafo como em todos os subgrafos induzidos. Os grafos perfeitos incluem várias famílias de grafos e unificam os resultados relativos a colorações e cliques nesses grupos. Alguns exemplos de grafos perfeitos são os grafos bipartidos - já mencionados -, as árvores, os grafos cordais e os grafos split. Estas classes serão definidas na sequência.

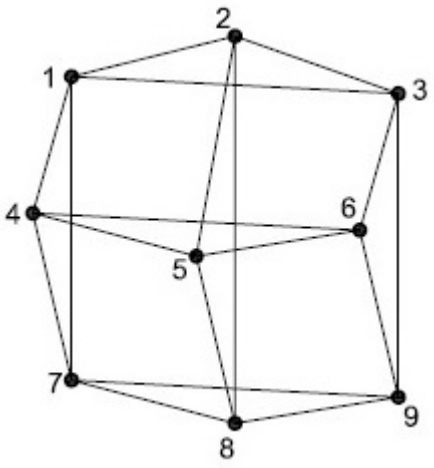


Figura 2.5: Exemplo de grafo perfeito.

Fonte: Própria.

- Árvore

Foi dito anteriormente que a restrição de um problema difícil a uma classe de grafos em particular pode fazer com que o problema seja resolvido em tempo polinomial, e isso é particularmente interessante quando o grafo apresenta uma estrutura de árvore. Desta forma, algumas definições básicas são necessárias. Estes seguem o adotado por West (2002).

Uma árvore é um grafo conectado e acíclico, portanto, dois vértices quaisquer são conectados por um único caminho. Uma **folha** é dito como sendo um vértice de grau 1 e os demais nós de uma árvore possuem grau maior ou igual a 1.

Uma árvore **enraizada** é um tipo especial de árvore que apresenta um vértice (raiz) que se distingue dos demais. Assim, uma árvore enraizada possui uma ramificação ordenada dos demais nós a partir da raiz. Árvores enraizadas possuem diversas aplicações, como por exemplo, em armazenagem de dados e buscas.

Uma árvore **binária** é uma árvore enraizada onde cada vértice possui no máximo dois filhos, e cada um dos seus filhos são designados como "filho à esquerda" ou "filho à direita". As subárvores enraizadas nos filhos do nó raiz são a "subárvore esquerda" e a "subárvore direita" da árvore (Figura 2.6). Uma **árvore binária completa** é quando todos os nós internos possuem dois filhos e todas as folhas estão na mesma profundidade.

Em especial, estruturas em árvore são bastante utilizadas em diversas aplicações que vão desde problemas de buscas e localização rápida de registros até representações algébricas envolvendo operações binárias, por isso esta classe de grafos é de especial interesse dentro da área de estruturas de dados, porém maiores detalhes a esse respeito não fazem parte deste trabalho.

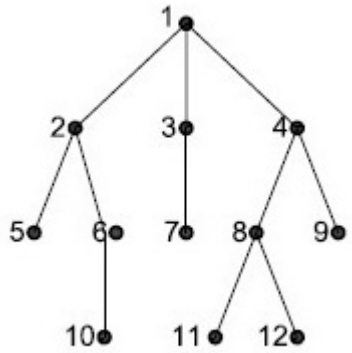


Figura 2.6: Exemplo de árvore.
Fonte: Própria.

- Grafo Cordal

Seja um grafo G não direcionado, G é dito cordal (ou triangularizado) quando em todo ciclo com comprimento maior do que 3 existe uma aresta (corda) não pertencente ao ciclo mas que liga dois vértices desse ciclo (GOLUMBIC, 2004) (Figura 2.7). Esse tipo de grafo torna-se importante pois certos

problemas que seriam NP-difíceis (explicação encontra-se na próxima seção) em grafos genéricos passam a executarem em tempo polinomial em grafos cordais.

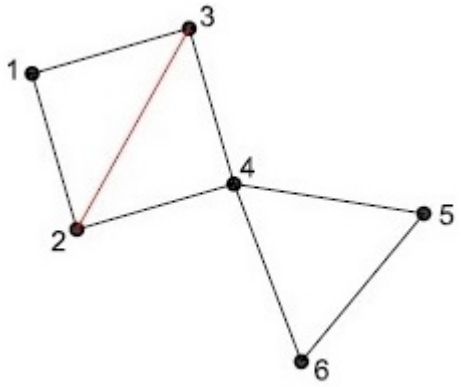


Figura 2.7: Exemplo de grafo cordal.
Fonte: Própria.

- Grafo Split

Para a caracterização de um grafo split, primeiramente é preciso entender o que vem a ser um conjunto independente. Um conjunto independente é um conjunto S de vértices em G tal que não existem dois vértices adjacentes em S, ou em outras palavras, é um grafo tal que seu complemento é uma clique.

Assim, um grafo split é aquele no qual o seu conjunto vértices $V(G)$ pode ser subdividido em dois subconjuntos, tal que um é uma clique - um conjunto de nós em que todos estejam conectados entre si - e outro é um conjunto independente. Uma definição alternativa seria: grafo split é um grafo cordal em que o seu complemento também é cordal (Figura 2.8).

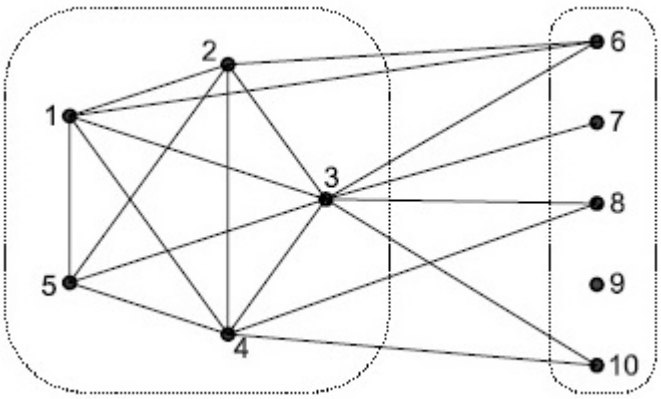


Figura 2.8: Exemplo de grafo split.
Fonte: Própria.

2.1.2 O Problema do Isomorfismo de Grafos

Não necessariamente dois grafos que diferem em sua representação visual são realmente grafos diferentes entre si, ou seja, podem ser diferentes visualmente, mas serem estruturalmente equivalentes (mesmos nós, mesmas arestas e mesma relação de adjacência entre vértices). Desta forma, estruturas que são iguais excetuando-se suas mudanças de nomes, são ditas isomorfas.

O problema do isomorfismo não é tão simples como aparenta, portanto, como este tema será o objeto de estudo deste trabalho, uma definição formal a respeito de isomorfismo de grafos se faz necessário.

Dois grafos $G = (V_1, E_1)$ e $H = (V_2, E_2)$ são isomorfos se existem funções bijetoras entre vértices de G para vértices de H, tal que preservem suas adjacências. Em outras palavras: f_1: $V_1 \rightarrow V_2$ e f_2: $E_1 \rightarrow E_2$ onde as funções f_1 e f_2 são bijeções entre os vértices e as arestas de um grafo para outro, respectivamente. Neste caso, dizemos que G é isomorfo a H.

Se H é isomorfo à G, então G também é isomorfo a H, mas frequentemente dizemos que "G e H são isomorfos", assim, o adjetivo "isomórfico" aplica-se a pares de grafos.

A Figura 2.9 ilustra um exemplo de isomorfismo em grafos, onde as bijeções que tornam os grafos isomorfos são:

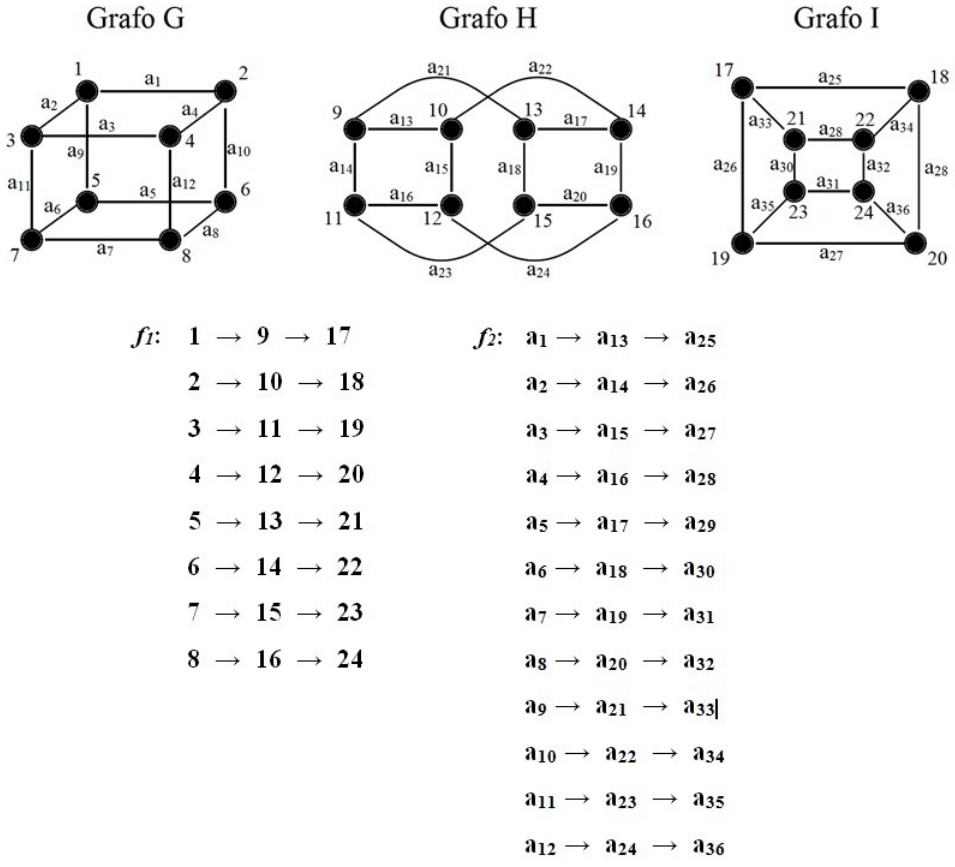


Figura 2.9: Grafos isomorfos.

Fonte: Própria, baseada em Fortin (1996, p.2).

A noção de isomorfismo pode também ser aplicada a grafos direcionados. Neste caso apenas é

preciso tomar cuidado para considerar a orientação das arestas, que não devem diferir entre os grafos.

Contudo, uma constatação importante a esse respeito é que de acordo com Miller (1978), o problema do isomorfismo de grafos não orientados é tão difícil quanto o problema do isomorfismo geral.

O problema do isomorfismo em grafos possui bastante importância tanto teórica quanto prática e é bastante estudado na ciência da computação. Suspeita-se que não seja um problema "NP-Completo" (definição encontra-se na próxima seção), no entanto, apesar de muitos esforços ainda não foi encontrado nenhum algoritmo de tempo polinomial para o problema.

Ao passo que a análise do problema do isomorfismo quanto à sua complexidade computacional já seria o suficiente para justificar seu estudo, existe muitas aplicações práticas que necessitam de um algoritmo eficiente para este problema, por exemplo, na área de visão computacional, para analisar a disposição de objetos em uma cena ou na detecção de um objeto embutido em um padrão, ou ainda na química orgânica, onde cientistas lidam frequentemente com grafos que representam links entre moléculas (FORTIN, 1996).

A principal estratégia utilizada na resolução do problema de isomorfismo em grafos é a identificação de vértices equivalentes para mapeamento, isto é, vértices que possuam as mesmas características como sequências de graus, distâncias entre os vértices, entre outros.

Desta forma, para se provar que dois grafos não são isomorfos, basta constatar que a(s) bijeção(ões) necessária(s) não existe(m). Assim, algumas condições triviais sob as quais dois grafos não são isomorfos são os casos em que os grafos diferem quanto:

1. À quantidade de nós;
2. À quantidade de arestas;
3. A existência de aresta(s) paralela(s) em um grafo e ausência em outro;
4. A existência de laço(s) em um grafo e ausência em outro;
5. A existência de uma granularidade de um nó em um grafo e não no outro;
6. À conectividade dos grafos;
7. A existência de ciclo(s) em um grafo e ausência em outro.

Entretanto, mesmo que o par de grafos satisfaça essas condições, eles podem não ser isomorfos.

2.1.3 Respresentação de Grafos

Para a resolução de problemas em grafos é importante a análise da melhor maneira de se representar um grafo computacionalmente, o que envolve duas estruturas de dados usuais: uma matriz de adjacências ou uma lista de adjacências.

Na Matriz de adjacências um grafo é representado por uma matriz $n \times n$, onde o elemento $M_{i,j}$ é o número de arestas entre os nós n_i e n_j. Para um grafo não-direcionado, a matriz de adjacência será sempre simétrica, porém o contrário não ocorre em grafos direcionados. Ainda, considerando um grafo

simples com pesos em suas arestas, é possível representar a matriz de adjacência com os valores destes pesos ao invés de apenas a indicação de presença ou ausência de uma aresta entre dois nós.

Suas vantagens podem ser descritas como: uma maior facilidade e rapidez de implementação e também em checar se uma certa aresta existe ou não. Suas desvantagens são que a necessidade de alocar uma matriz n^2 pode demandar muito espaço de memória.

Na Lista de Adjacências, o grafo é representado por um vetor de vértices e por várias listas, onde cada vértice possui uma lista de todos os vértices adjacentes a ele. Para percorrer de um item para outro na lista de um determinado vértice são usados ponteiros e essa estrutura chama-se lista encadeada. Para casos de grafos rotulados ou com pesos é possível incluir dados adicionais junto com o nó na lista de adjacência.

As vantagens dessa forma de representação de grafos são: uma melhor representação para grafos esparsos, um menor consumo de memória e a não necessidade de representação de arestas inexistentes no grafo original, diminuindo então a complexidade de buscas no grafo. Já suas desvantagens podem ser citadas como: necessidade de uma codificação mais demorada e maior dificuldade na verificação da existência de certa aresta no grafo.

2.2 Complexidade

Como este trabalho refere-se ao estudo de algoritmos computacionais, é de grande importância o estudo dos recursos computacionais (como o tempo) que são exigidos por esses algoritmos. Assim, é importante ressaltar algumas diferenças entre certas definições e de acordo com o enfoque do que está sendo analisado.

De acordo com Goldreich (2010), quando o estudo incide sobre os recursos que são necessários para que um algoritmo resolva uma tarefa específica ele é visto como pertencente à teoria da complexidade computacional ou "Teoria da Complexidade". Já quando o foco é sobre a concepção e análise de algoritmos específicos (em vez de sobre a complexidade intrínseca da tarefa, ou problema), o estudo é visto como pertencendo a outro domínio relacionado – o "Projeto e Análise de Algoritmo".

Além disso, Projeto e Análise de Algoritmos tende a ser subdividido de acordo com o domínio da matemática, ciência, engenharia e em que se concentram os problemas computacionais. Já a Teoria da Complexidade normalmente mantém uma unidade de estudo de problemas computacionais que são resolvidos com certos recursos (independentemente das origens desses problemas).

Portanto, o que será tratado neste trabalho se referirá ao estudo da teoria de complexidade.

A teoria de complexidade de algoritmos é de fundamental importância não só na área da ciência da computação como também diversas áreas a qual a computação está relacionada, como por exemplo, engenharia e segurança. A sua importância decorre do fato de que a análise e classificação correta de um problema dentro de classes de problemas conhecidos se torna fundamental na busca por uma solução.

Primeiramente é necessário definir o que é um problema. Segundo Cormen et al (2001), um problema pode ser de dois tipos: abstratos e concretos. Um problema abstrato é uma relação binária (R) entre um conjunto de instâncias (I) e um conjunto de soluções (S), por exemplo, o problema abstrato pode ser a soma de dois números, então a instância são os dois números reais e a solução será um número

real. A relação binária R é um subconjunto do produto cartesiano de $S \times I$.

Dentre os problemas abstratos encontram-se os problemas de otimização (quando algum valor deve ser maximizado ou minimizado) e os de decisão (quando o conjunto de soluções resume-se em "sim" ou "não").

Contudo, ainda de acordo com os autores, para que um computador possa resolver um problema, as suas instâncias devem ser codificadas de maneira que o computador possa interpretá-las, e para isso é necessário a definição de uma codificação de objetos abstratos (as instâncias de I) em cadeias binárias (C) de qualquer natureza. Assim, ao se mapear (codificar) de I para C estamos transformando um problema abstrato em um problema concreto. Por exemplo, I pode ser o conjunto $I = \{0, 1, 2, 3\}$ e seu mapeamento é $C = \{00, 01, 10, 11\}$.

2.2.1 Classes de Complexidade

As classes de complexidade de algoritmos agrupam problemas com características semelhantes, assim, as classificações e definições apresentadas na sequência seguem o padrão adotado por Cormen et al (2001).

- Classe P:

A classe de complexidade P (*Polynomial Time*) é definida como sendo a classe de problemas concretos de decisão que podem ser **resolvidos** em tempo polinomial (admitem algoritmo polinomial), ou, mais precisamente, problemas que podem ser resolvidos em tempo $O(n^k)$, para alguma constante k, onde n é o tamanho da entrada do problema.

Em seu aspecto prático trata-se de problemas que são solúveis por uma máquina de Turing determinística (dado o estado atual da máquina e todas as entradas, há apenas uma ação possível que ela pode realizar), sequencial (executa ações uma após a outra) e de tempo polinomial em relação ao tamanho da entrada.

- Classe NP:

A classe de complexidade NP (*Nondeterministic Polynomial Time*) é a classe de problemas que podem ser **verificados** em tempo polinomial (admitem um algoritmo de certificação que é polinomial). Para um melhor entendimento do que vem a ser o termo "verificado", podemos pensar da seguinte maneira: se especificarmos de alguma forma um "certificado" a uma solução, então podemos verificar que este certificado é correto em tempo polinomial em relação ao tamanho de entrada do problema. Por exemplo: em um problema que se deseja obter um caminho específico em um grafo, um certificado poderia ser uma sequência específica de vértices em que podemos checar facilmente em tempo polinomial se esta dada sequência satisfaz o problema.

Por fim, analogamente à classe P, soluções para problemas em NP podem ser encontrados em tempo polinomial em uma máquina de Turing não-determinística.

- NP-Completo:

Considerando um problema pertencente à classe NP, por exemplo, nomeando-o de A, se todos os problemas da classe NP podem ser reduzidos ao problema A, então A pertence aos NP-Completos. Diz-se que os NP-Completos são os problemas mais difíceis em NP.

- NP-Difícil:

Da mesma forma que na classificação anterior, considerando um problema B (que pode ser pertencente à classe NP ou não), se todos os problemas de NP e NP-Completos podem ser reduzidos ao problema B, então B pertence aos NP-Difíceis. Diz-se que essa classe (NP-Difíceis) são os problemas que são pelo menos tão difíceis quanto os NP-Completos, ou seja, com um grau de dificuldade no mínimo igual aos NP-Completos.

Problemas NP-Difíceis não precisam estar em NP, ou seja, não precisam ter soluções verificáveis em tempo polinomial.

A Figura 2.10 mostra um esquema da intersecção das classes apresentadas mais aceito atualmente.

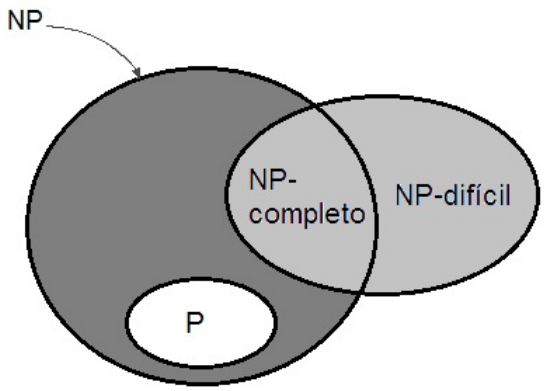


Figura 2.10: Classes de complexidade.
Fonte: Própria baseada em Cormen et al (2001).

Portanto, de acordo com o exposto acima, verificamos que como as classes NP-Completo e NP-Difícil "derivam" de NP, podemos resumir em duas as classes de complexidade: a classe P e a NP, onde, de uma maneira informal, a classe P é a dos problemas resolvidos "rapidamente" e a classe NP é a de problemas verificados "rapidamente".

Podemos verificar também que qualquer problema em P também está em NP, pois uma vez que um problema está em P, então podemos resolvê-lo em tempo polinomial sem mesmo ter fornecido um "certificado" como solução, e se conhecemos um algoritmo que resolva o problema em tempo polinomial, certamente podemos deduzir que um algoritmo de verificação rodará também em tempo polinomial.

Desta forma, sabemos que $P \subseteq NP$, contudo esta relação entre as classes introduz uma das maiores questões em aberto da ciência da computação: se P é um subconjunto próprio de NP. A famosa pergunta: $P = NP$? gira em torno dos NP-Completos, pois se é possível resolver em tempo polinomial um NP-Completo é possível resolver todos os problemas em NP em tempo polinomial devido à definição de NP-Completo.

Cabe destacar aqui ainda a relação entre o Problema do Isomorfismo de Grafos e as classes de complexidade de algoritmos. Primeiramente, o problema de isomorfismo de grafos é de grande interesse para a ciência da computação, pois sua classificação é considerada indefinida entre as classes de complexidade P e NP-Completo. Alguns pesquisadores conjecturam que existe um algoritmo polinomial para o problema, o que comprovaria que o mesmo está em P, contudo, devido a não existência ainda de um algoritmo com essa característica, muitos acreditam que o problema não pertença à classe P e nem à classe NP-Completo (ARORA e BARAK, 2009).

Como o relacionamento dessas duas classes é de grande importância para este trabalho, maiores informações a respeito serão tratadas na seção seguinte.

2.2.2 P vs NP

Como introduzido anteriormente, este famoso questionamento dentro da teoria de complexidade pode ser considerado como se perguntássemos se encontrar soluções para alguns problemas é mais difícil do que verificar a validade de certas soluções para os mesmos.

Segundo Goldreich (2010), acredita-se que a resposta a estas formulações é que encontrar soluções é mais difícil do que verificar se certa solução satisfaz o problema, ou seja, crê-se que P é diferente de NP. Assim, os conceitos e argumentos apresentados nesta seção se referem às ideias segundo este autor.

Como já visto, um problema em NP é chamado NP-Completo se um algoritmo eficiente para este problema pode ser convertido em outro algoritmo eficiente para qualquer outro problema em NP.

Quando somos confrontados com um problema aparentemente difícil em NP, só podemos provar que ele não está em P, assumindo que NP é diferente de P. Assim, a busca atual se refere em encontrar meios de se provar que, se o problema em questão está em P, então NP é igual a P, o que significa que todos os problemas em NP estão em P.

É aqui que a teoria da NP-Completude entra em cena. Segue-se que, se algum problema NP-Completo estiver em P, então todos os problemas em NP estão em P. Já por outro lado, se NP for diferente de P, então nenhum problema NP-Completo pode estar em P. Consequentemente, a questão P vs NP gira em torno de se um problema específico NP-Completo pode ou não ser resolvido de forma eficiente.

Na verdade, o objetivo da classe NP inteira (incluindo a P) está centrada em cada problema NP-Completo individualmente. Em particular, mostra que um problema que é NP-Completo implica que este problema não está em P, ao menos que P seja igual a NP.

Problemas NP-Completos existem, e, além disso, centenas de problemas computacionais provenientes de diversas áreas da matemática e das ciências são classificados como NP-Completos.

Exemplos de problemas NP-Completos conhecidos podem ser: encontrar uma atribuição satisfatória para certa fórmula booleana (ou decidir se tal atribuição existe), encontrar uma 3-coloração de vértices de um grafo (ou decidir se tal coloração existe). Tabela 1 mostra um quadro com alguns outros exemplos de problemas pertencentes a cada uma das classes mencionadas.

Assim, na coluna à direita temos problemas que podem ser resolvidos eficientemente por diversos algoritmos específicos (programação dinâmica, fluxo em redes, busca em grafos, algoritmos gulosos, etc.). Estes problemas são ditos fáceis por diferentes razões. Já os problemas à esquerda são difíceis pela mesma

Problemas Difíceis (NP-Completos)	**Problemas Fáceis (em P)**
3SAT	2SAT, HORN SAT
Problema do Caixeiro Viajante	Árvore Geradora Mínima
Caminho Mais Longo (Longest Path)	Caminho Mais Curto (Shortest Path)
3D Matching	Bipartite Matching
Problema da Mochila (Knapsack)	Unary Knapsack
Conjunto Indepentente (Independent Set)	Conjunto Independente em Árvores
Programação Linear Inteira	Programaçã Linear
Caminho de Rudrata (Rudrata Path)	Caminho Euleriano (Euler Path)
Corte Balanceado (Balanced Cut)	Corte Mínimo (Minimum Cut)

Tabela 2.1: Classes de complexidade.

Fonte: Dasgupta et al, (2006, p.257).

razão, em sua essência todos se referem ao mesmo problema apenas se apresentam em diferentes formatos, ou seja, são equivalentes e cada um pode ser reduzido a qualquer um dos outros.

Esta questão entre equivalência de problemas gira em torno da teoria da NP-Completude, que está centrada na ideia de uma "redução eficiente", baseada em uma relação entre problemas computacionais. Genericamente falando, um problema computacional é eficientemente redutível a outro problema se for possível resolver de forma eficiente o primeiro problema quando fornecido algum algoritmo (eficiente) para a solução do segundo. Assim, o primeiro problema não é mais difícil de resolver do que o segundo. Como já visto, um problema em NP é dito NP-Completo se qualquer problema em NP é redutível a ele, o que significa que o primeiro problema "codifica" todos os problemas em NP (e assim, em certo sentido, é o mais difícil entre eles).

Os problemas da coluna esquerda da tabela são de certa forma, o mesmo problema, mas apresentados em formatos diferentes. Portanto, se um deles possui um algoritmo que o resolve em tempo polinomial, então todos os problemas em NP possuem um algoritmo de tempo polinomial.

Na sequencia será melhor explicado a questão de redução de problemas computacionais. Mas para isto é necessário primeiramente o conceito de problemas de busca e decisão.

2.2.3 Problemas Computacionais

Em matemática assim como em diversas outras ciências, é habitual discutir objetos sem especificar sua representação. Isto não é possível na teoria da computação, em que a representação de objectos desempenha um papel central no seu estudo. Sob certo sentido, uma computação meramente transforma uma representação de um objeto em outra representação do mesmo objeto. Assim, um cálculo concebido para resolver um problema apenas transforma a instância do problema em sua solução, que pode ser pensada como sendo uma representação da instância. Na verdade, a resposta a qualquer pergunta específica está implícita na própria pergunta, e a computação é encarregada de tornar esta resposta explícita (Goldreich, 2010).

Problemas computacionais se referem a objetos que são representados de alguma forma, tal que forneça uma descrição "explícita" e "completa" do objeto correspondente.

Há dois tipos fundamentais de problemas computacionais, os chamados problemas de busca e problemas de decisão. Em ambos os casos, as noções-chave são instâncias do problema e especificação do problema. As definições a seguir estão baseadas em Goldreich (2010).

- Problemas de Busca: Um problema de busca consiste em uma especificação de um conjunto de soluções válidas (inclusive a solução vazia) para cada instância possível. Dado uma instância, é necessário encontrar uma solução correspondente (ou determinar que nenhuma solução existe).

Grande parte da ciência da computação baseia-se na resolução de problemas de busca, como por exemplo, encontrar caminhos mais curtos em um grafo, encontrar uma ocorrência de certo padrão em uma dada string, encontrar o valor médio de uma determinada lista de números, etc.

Além disso, os problemas de busca correspondem ao conceito comum de "resolução de um problema" (por exemplo, a procura de um caminho entre dois locais). Assim, de acordo com Dasgupta et al (2006), a discussão sobre a complexidade de problemas como esse corresponde a muitas necessidades práticas. Outros exemplos bastante comuns são a ordenação de uma sequência de números, a multiplicação de números inteiros e a fatoração prima de um número composto.

- Problemas de Decisão: Um problema de decisão consiste em uma especificação de um subconjunto de instâncias possíveis. Dado uma instância, é necessário determinar se a instância está no conjunto especificado.

Por exemplo, considerando o problema em que é dado um número natural e necessita-se saber se o número é primo ou não (isto é, se o número está no conjunto de números primos). Assim, problemas de decisão geralmente procuram decidir se um determinado objeto tem uma certa propriedade, que em geral se referem a tarefas comuns (como por exemplo, determinar se o semáforo está vermelho).

Um importante tipo de problema de decisão se refere àqueles derivados de problemas de busca considerando que o conjunto de instâncias contém uma solução. De fato, ser capaz de determinar se existe ou não uma solução é um pré-requisito para resolver o problema de busca correspondente.

Especificamente para este tipo de problema, a função que o resolve é uma função booleana, que indica membros a um certo conjunto predeterminado, ou seja, é necessário uma função f para indicar se uma instância x reside ou não no conjunto predeterminado S. Esta indicação é representada por um valor binário (1 corresponde a uma resposta positiva e 0 a uma resposta negativa). Assim, dado x, a função tem como saída 1 se $x \in S$, e 0 caso contrário.

Outra observação é que a função f que resolve um problema de decisão é exclusivamente determinada pelo próprio problema de decisão, isto é, se f resolve o problema de decisão de S, então f é igual à função característica de S.

2.2.4 Redução

Segundo Goldreich (2010), podemos dizer grosseiramente, que a computação é um processo que modifica um ambiente relativamente grande através de aplicações repetidas de uma regra simples e predeterminada.

Estamos interessados em um processo computacional (cálculo), em que uma regra está concebida para atingir um efeito desejado. O estado inicial ao qual a computação é aplicada codifica uma sequência de entrada, e o estado final (ao final da computação) codifica uma sequência de saída. Assim, a computação define um mapeamento de entradas para saídas, e tal mapeamento pode ser visto como a solução de um problema de busca (dado uma instância x encontre uma solução y que se relacione a x de alguma forma predeterminada) ou como um problema de decisão (dado uma instância x, determine se x possui alguma propriedade específica ou não).

De um modo geral, um problema computacional A é (eficientemente) redutível a um problema B se este segundo consegue resolver (eficientemente) o primeiro quando se fornece um algoritmo (eficiente) para a solução de B.

Para exemplificar melhor, podemos ver uma redução de um problema de busca A para um problema de busca B como sendo um algoritmo de tempo polinomial f que transforma qualquer instância I de A em uma instância $f(I)$ em B, junto a outro algoritmo de tempo polinomial h que mapeia qualquer solução S de $f(I)$ de volta a uma solução $h(S)$ de I, (conforme Figura 2.11). Se $f(I)$ não possui solução, então I também não possui. Esses dois procedimentos, f e h, implica que qualquer algoritmo que atue no problema B pode ser convertido em um algoritmo para o problema A por escalonamento entre f e h (DASGUPTA et al, 2006).

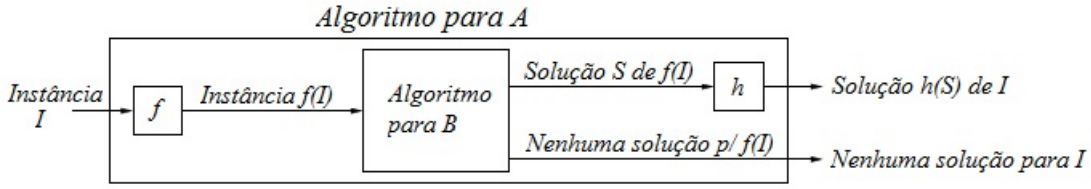


Figura 2.11: Redução de um problema A para B.
Fonte: Própria, baseada em Dasgupta et al (2006, p.259).

Em outras palavras, se dois problemas X e Y em NP, dizemos que Y é redutível a um problema de decisão X se existe um algoritmo que transforma qualquer instância "j" de Y em uma instância "i" de X tal que "j" é positiva se e somente se "i" for positiva. Pode-se dizer, então que Y é *redutível* a X se Y for um "subproblema", ou um "caso particular" de X.

São considerados aqui apenas os algoritmos de reduções *polinomiais*. E se existe uma redução polinomial de Y a X, escrevemos $Y \leq_p X$.

- 3SAT $\rightarrow$ Conjunto Independente

Para explicitar um exemplo de redução de um problema a outro, podemos demonstrar a redução do problema 3SAT ao do Conjunto Independente de acordo com Dasgupta et al, (2006).

A entrada do problema 3SAT é um conjunto de cláusulas, cada uma com três ou menos literais, por exemplo:

$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$

Onde se pretende encontrar uma atribuição verdade que satisfaça o conjunto. No Conjunto Independente a entrada é um grafo e um número "g", em que o objetivo está em encontrar um conjunto com "g" vértices não adjacentes.

Desta forma, iremos relacionar a lógica booleana com grafos para isto. O grafo produzido é formado a partir do momento que escolhemos um literal de cada cláusula como sendo o valor verdade (ou *true*), onde se escolhermos $\neg x$ em uma cláusula, não podemos escolher x em outra. Qualquer escolha de literais, um de cada cláusula, especifica uma atribuição verdade (onde os literais escolhidos não podem receber ambos os valores).

Assim, representamos uma cláusula, por exemplo: $(x \vee \neg y \vee z)$, por um triângulo com cada vértice indicado por x, $\neg y$, z, pois ao ter seus três vértices ligados entre si, teremos que escolher apenas um deles para o conjunto independente. No caso de cláusulas com apenas dois literais, a representação se dá por meio de uma aresta ligando os dois literais, e o caso de uma cláusula com apenas um literal, seu valor já estaria automaticamente determinado, não necessitando maiores análises.

No grafo resultante, o conjunto independente deve ter no máximo um literal de cada cláusula. Para garantir que será escolhido exatamente um de cada cláusula, podemos partir do número k como sendo o número de cláusulas, que no exemplo, $k = 4$.

O próximo passo é escolher uma maneira que nos impeça de escolher dois literais opostos (exemplo: x e $\neg x$) em diferentes cláusulas. Para isto, colocamos uma aresta entre dois vértices correspondendo a literais opostos. O grafo obtido com isso é mostrado na Figura 2.12.

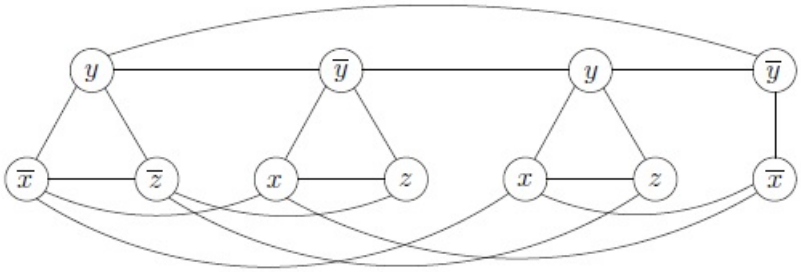


Figura 2.12: Grafo resultante.
Fonte: Dasgupta et al (2006, p.264).

Resumindo os passos seguidos até aqui, tivemos que, a partir de uma instância I do problema 3SAT, criamos uma instância (G, k) do problema do Conjunto Independente, onde:

- **G**: é o grafo com um triângulo para cada cláusula (ou uma aresta, caso a cláusula possua apenas dois literais), com seus vértices indicados por cada literal, e arestas entre dois vértices quaisquer ligando literais opostos;
- **k**: é o objetivo, ou o número de cláusulas.

Esta construção pode ser feita em tempo polinomial. Contudo, em uma redução não basta apenas mapearmos instâncias do primeiro problema a instâncias do segundo (ou seja, a função f da Figura 2.11),

mas também uma maneira de reconstruir uma solução para o primeiro exemplo a partir de uma solução do segundo (a função h). Assim:

1. Dado um conjunto independente S com k vértices em G, é possível recuperar de forma eficiente a atribuição verdade satisfatória para I.

Como o conjunto S não pode conter vértices rotulados, por exemplo, com x e $\neg x$, então atribuímos a x o valor *true* se S contém o vértice x, e *false* se S contém o valor $\neg x$ (se não contiver nenhum, atribuir qualquer valor para x). Uma vez que S possui k vértices, ele deve ter um vértice por cláusula e esta atribuição verdade satisfaz cada literal específico e, portanto, satisfaz também todas as cláusulas.

2. Se o grafo G não possuir nenhum conjunto independente de tamanho k, então a fórmula booleana I é insatisfatível.

Geralmente é mais fácil provar o contrário, onde se I tiver uma atribuição satisfatível então G tem um conjunto independente de tamanho k. Isto é feito da seguinte forma: para cada cláusula, escolhe-se algum literal cujo valor sob uma atribuição satisfatória seja *true* (deve haver pelo menos um literal), e adiciona-se o vértice correspondente a S. Ao final, S conterá um conjunto independente.

2.2.5 Isomorfismo e Teoria de Complexidade

Segundo Arora e Barak (2009), um dos aspectos notáveis da NP-Completude é que um número surpreendente de problemas NP - incluindo alguns que foram estudados por muitas décadas - se tornaram NP-Completos.

Este fenômeno sugere uma conjectura interessante: para todos os problemas em NP, ou ele é P ou NP-Completo (Figura 2.13).

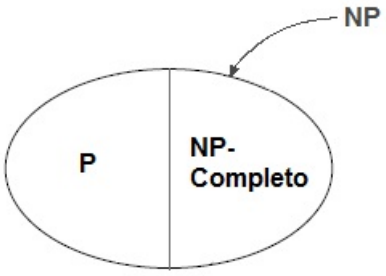


Figura 2.13: Classes de complexidade.
Fonte: Própria baseada em Dasgupta et al, (2006, p.260).

Há também, segundo o autor, um teorema chamado de **Teorema de Ladner** baseado nessa ideia. Maiores detalhes a respeito da prova deste teorema não fazem parte do escopo deste trabalho.

Teorema de Ladner (Linguagem "NP-Intermediária"): suponha que $P \neq NP$. Então existe uma linguagem $L \in NP \backslash P$ que não é NP-Completa.

Se P=NP, então a conjectura de que para todos os problemas em NP ou eles são P ou NP-

Completo é realmente verdadeira mas não traz maiores interesses. Mas caso $P \neq NP$ (como se acredita), há uma linguagem $L \in NP \backslash P$ que não é NP-Completa, conforme o teorema acima.

Em outras palavras, se for provado que $P \neq NP$, então existem problemas em NP que não estão nem em P nem em NP-Completo e tais problemas são chamados de problemas "NP-Intermediários".

Na verdade, existem muito poucos problemas candidatos para essa linguagem, pois a maioria foi resolvido por algum algoritmo criado ou alguma redução realizada.

Duas exceções interessantes neste contexto são o problema da Fatoração e do Isomorfismo em Grafos. Nenhum algoritmo de tempo polinomial é conhecido atualmente para esses problemas, e há fortes indícios de que eles não sejam NP-Completos.

Desta forma, como este trabalho tratará sobre o isomorfismo de grafos, este é um caso especial de problemas pois é um dos poucos que se encontra em NP mas que não se sabe se ele está em P, NP-Completo ou NP-Intermediário. Esta resposta não é conhecida, porém há fortes evidências de que ele não esteja, pelo menos, em NP-Completo (Figura 2.14).

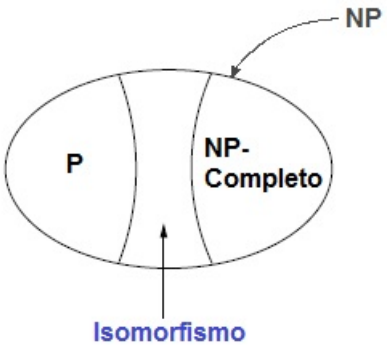


Figura 2.14: Teorema de Ladner.
Fonte: Própria baseada em Dasgupta et al, (2006, p.260).

2.2.6 A Classe coNP

Como já visto, a classe NP refere-se aos problemas que podem ser verificados em tempo polinomial, ou seja, existe um algoritmo verificador que recebe uma instância do problema e também um "certificado" da solução. Assim, o verificador checará em tempo polinomial se o certificado satisfaz ou não o problema, ou seja, responderá "sim" ou "não" ao final. No caso da resposta positiva dizemos que o verificador aceitou o certificado.

Desta forma, a classe NP é o conjunto de todos os problemas de decisão para os quais existem verificadores polinomiais. Contudo, a pertinência de um problema de decisão à esta classe nem sempre é identificado facilmente.

Para um melhor entendimento quanto a esta questão é preciso incluir alguns conceitos sobre as classes de complexidade já vistas anteriormente. Os problemas de decisão ocorrem em pares complementares. O *complemento* de um problema de decisão X é o problema de decisão Y, se Y possuir as mesmas

instâncias que X, além de toda instância positiva de X ser negativa em Y e vice-versa.

Assim, há classes complementares às classes de complexidade (de problemas de decisão) já analisadas, onde ao precedermos seu nome com o prefixo "co" queremos dizer que é a classe de conjuntos complementares, ou seja:

$coC = \{\{0,1\} * \backslash S : S \in C\}$.

Por exemplo, **coNP** é uma classe de complexidade que contém os complementos dos problemas encontrados na classe NP. Um problema de decisão Y pertence à coNP se, e somente se, o seu complemento X estiver em NP. Em outras palavras, coNP é a classe de problemas para o qual existam provas eficientemente verificáveis para a não existência de instância, os chamados contra-exemplos.

Por exemplo, SAT pertence à classe NP, isso implica que o conjunto de todas as fórmulas insatisfazíveis está em coNP. Da mesma forma, se o problema do caminho Halmiltoniano está em NP, então o problema co-Halmiltoniano está em coNP, por definição.

Se $E \subseteq \{0,1\}*$ é uma linguagem, denota-se por $\overline{E} : \{0,1\} * \backslash E$.

Uma definição mais formal esse respeito seria:

coNP : $\{E : \overline{E} \in NP\}$

Da mesma forma, um problema coNP-Completo é o complemento de um problema NP-Completo.

Um problema de decisão X é dito **coNP-Completo** se está em coNP e se qualquer problema Y, em coNP, pode ser reduzido em tempo polinomial à X, ou seja, existe um algoritmo polinomial que pode transformar qualquer instancia de Y em uma instancia de X com o mesmo valor verdade.

A partir das definições acima a respeito das classes complementares mencionadas, podemos fazer primeiramente uma análise a respeito de conjecturas existentes e posteriormente como elas se relacionam com o problema do isomorfismo de grafos.

- ***Conjecturas***:

Dizemos que uma classe é fechada à complementação caso ela seja igual ao seu complemento. Assim, foi provada que a classe P é fechada a complementação, ou seja, P = coP. Além disso, acredita-se que $NP \neq coNP$, ou seja, que a classe NP não é fechada à complementação. Desta forma, podemos verificar que esta conjectura está intimamente ligada ao famoso questionamento P vs NP, pois: sabemos que P é fechada à complementação, e além disso, caso fosse provado que P = NP, isso implicaria na classe NP também ser fechada à complementação, contrariando a ideia inicial.

Proposição 1: Supondo que **NP** $\neq$ **coNP** e que $A \in NP$ tal que todo conjunto em NP é redutível a A (A é NP-Completo). Portanto, pela definição, $\overline{A}$ não está em NP.

De acordo com um Teorema de Garey (1979): "Se existe um problema NP-Completo A, e seu complementar $\overline{A}$ está em NP, então NP = coNP".

Como consequência desse teorema, um problema A - para o qual tanto A quanto $\overline{A}$ pertencem a NP - não pode ser NP-Completo a não ser que NP = coNP, contrariando nossa proposição. Assim, não se poderia esperar que tal problema A seja NP-Completo realmente. Tudo isso mostra que há uma forte ligação entre os problemas NP-Completos e a conjectura de que $NP \neq coNP$.

Outro esclarecimento a respeito é que os conjuntos das classes NP e coNP possuem uma interseção não vazia, e toda linguagem em P encontra-se nesta intersecção, $NP \cap coNP$ (ARORA e BARAK, 2009).

Com tudo isso, se assumirmos que $NP \neq coNP$, a *Proposição 1* implica que conjuntos em $NP \cap coNP$ não podem ser NP-Completos.

O autor ainda ressalta se não haveria a possibilidade de ao utilizarmos certos tipos de reduções mais limitantes não poderíamos estar excluindo a classe de problemas NP-Completos da intersecção $NP \neq coNP$ erroneamente. Há um teorema que afirma que esta situação não é possível. O teorema afirma que alguns conjuntos em NP não podem ser reduzidos a conjuntos que se encontram na intersecção $NP \cap coNP$ mesmo sob reduções gerais, e caso isso fosse possível então teríamos que: $NP = coNP$.

Em particular, assumindo que $NP \neq coNP$, nenhum conjunto em $NP \cap coNP$ pode ser NP-Completo. Uma vez que $NP \cap coNP$ é conjecturado a ser um superconjunto próprio de P, segue-se que existem problemas de decisão em NP que não estão nem em P, ou seja, são especificamente os problemas de decisão que estão na intersecção de: $(NP \cap coNP) \neq P$ (assumindo que $NP \neq coNP$).

Por fim, podemos chegar à seguinte conclusão de acordo com tudo o que foi exposto acima: assumindo que $NP \neq coNP$, e considerando o fato de que existe uma classe intermediária que se encontra na intersecção $NP \cap coNP$, problemas NP-Completos não podem estar nesta intersecção, e, consequentemente, não podem pertencer ao conjunto coNP (apenas à NP, conforme já visto em seções anteriores). Pois caso problemas NP-Completos estivessem em coNP também, todos esses conjuntos estariam em P, ou seja: $NP = coNP = P$ (ARORA e BARAK, 2009), e por fim teríamos provado a famosa questão em aberto da ciência da computação: $P = NP$.

Lembrando, por fim, que esta conclusão parte-se do pressuposto de que $NP \neq coNP$. Assim, isto ajuda reforçar a conjectura de que realmente P é diferente de NP, pois acredita-se que a classe NP não é fechada à complementação. Portanto, caso seja provado que $NP \neq coNP$ realmente, então teremos automaticamente provado que $P \neq NP$.

A Figura 2.15 exemplifica a relação entre essas classes conforme tudo o que foi exposto até então, presumindo-se que $NP \neq coNP$.

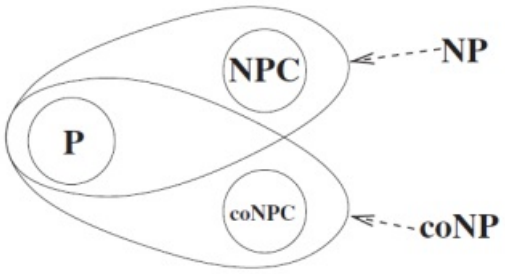


Figura 2.15: Visão geral das classes considerando-se que: $P \neq (NP \cap coNP) \neq NP$.
Fonte: Goldreich (2010, p.158).

- ***O problema do Isomorfismo de Grafos***:

Conforme já exposto anteriormente neste trabalho, não há nenhum algoritmo de tempo polinomial

conhecido para determinar se dois grafos são isomorfos, porém, há fortes evidências de que o problema também não é NP-Completo (ARORA e BARAK, 2009).

De acordo com Goldreich (2010), existem problemas conhecidos que se encontram na intersecção $NP \cap coNP$, porém, não se conhece nenhum algoritmo que os resolva em tempo polinomial (ou seja, não estão em P). Exemplos de problemas neste conjunto são a Fatoração e o Isomorfismo de Grafos. Acredita-se que o problema da fatoração de números inteiros não seja NP-Completo, pois caso contrário, NP = coNP. Quanto ao isomorfismo, acredita-se que o ele esteja em coNP (o que faz esta questão ser outro problema em aberto na ciência da computação).

É fácil verificar que o problema do isomorfismo está em NP. O problema maior é se ele está em coNP. Onde, caso esteja, então isto se torna mais uma prova de que o problema do isomorfismo não é NP-Completo, ao menos que: NP = coNP = P.

Assim, podemos verificar que há uma forte ligação entre o Isomorfismo em Grafos e as conjecturas entre as classes NP e coNP explicitadas na seção anterior.

Sobre algoritmos que resolvem o problema do isomorfismo, o melhor até hoje conhecido é executado em tempo de $\epsilon^{O(\sqrt{nlogn})}$ para grafos com n vértices (MCKAY, 2013). Vários casos especiais são conhecidos por estarem em P. Assim como vários problemas são conhecidos por serem NP-Difíceis. Assim, o isomorfismo é amplamente considerado ser da classe NP-Intermediária. A Figura 2.16 ilustra essa situação.

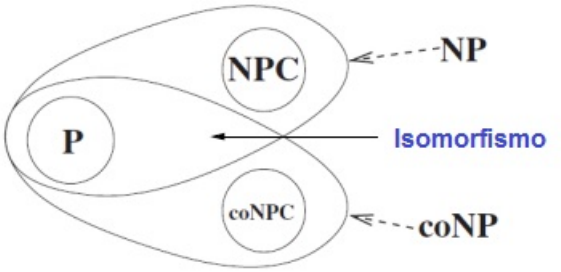


Figura 2.16: Isomorfismo.
Fonte: Própria, baseada em Goldreich (2010, p.158).

Assim, o objetivo até aqui foi situar o problema do isomorfismo no atual cenário dentro da teoria de complexidade, bem como introduzir os principais conceitos e relações existentes entre as diversas classes e como isso interfere na resolução de problemas e análises de algoritmos computacionais.

Na próxima seção será abordado o problema do isomorfismo de grafos propriamente dito, seu estado da arte e os algoritmos que serão analisados para resolução.

Capítulo 3

Isomorfismo de Grafos

Nesta seção será abordado o tema do isomorfismo em grafos propriamente dito.

Como já citado anteriormente, o problema do isomorfismo é de fundamental importância, mas de difícil tratamento. Sabemos também que ele se encontra em NP, porém, ainda não é conhecido nenhum algoritmo determinístico que o resolva em tempo polinomial e nem se sabe se este pertence à coNP também, além de haver fortes evidências de que não é NP-Completo.

Esta ambiguidade do exato local em que se encontra o isomorfismo dentro das classes de complexidade influenciou a existência de muitas pesquisas na área.

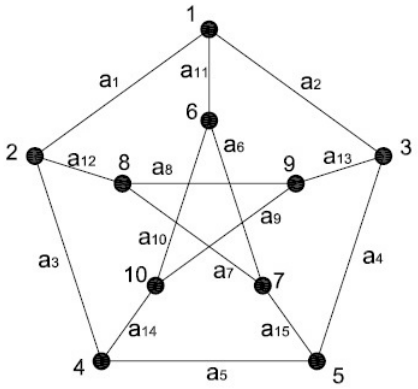
Enquanto que a posição do isomorfismo dentro da análise de complexidade já é suficiente para se justificar seu estudo, também existem muitas aplicações práticas para as quais seria interessante a obtenção de um algoritmo de execução eficiente para o problema. Assim muitos artigos discutem como construir verificadores rápidos e práticos. Algumas dessas rotinas são originadas de algoritmos especiais que executam em tempo polinomial para classes bastante específicas de grafos. Contudo, a maioria dos algoritmos práticos não possui um pior caso polinomial (FORTIN, 1996).

Mais adiante retornaremos a esta questão dos algoritmos que executam em tempo polinomiais para certas classes especiais de grafos. Primeiramente é preciso definir alguns conceitos gerais que permeiam o problema do isomorfismo. Os termos abaixo seguem o adotado por Fortin (1996).

- **Automorfismo**

O *automorfismo* de um grafo G pode ser definido como um isomorfismo entre duas cópias de G, ou em outras palavras, é um mapeamento f entre pares de vértices e arestas tais que $(a, b) \in E \iff (f(a), f(b)) \in E$. Um *Automorfismo Trivial* é um automorfismo onde $f(a) = a$ para todo $a \in V$. Uma *Partição de Automorfismo* de um grafo G é uma sequencia de subconjuntos disjuntos $V_1, \ldots, V_k$, para os quais todos os pares de vértices $a, b \in V_i$, ocorre um automorfismo onde $f(a) = b$. Assim, a *Partição de Automorfismo* divide V em conjuntos que consistem de todos os vértices que podem ser mapeados uns aos outros.

A Figura 3.1 ilustra um exemplo de automorfismo, onde o mapeamento f_1 entre vértices e f_2 entre arestas se dá da seguinte maneira:



f_1: $1 \to 6$ $6 \to 1$ f_2: $a_1 \to a_{10}$ $a_6 \to a_1$ $a_{11} \to a_{11}$
$2 \to 7$ $7 \to 3$ $a_2 \to a_6$ $a_7 \to a_3$ $a_{12} \to a_{14}$
$3 \to 10$ $8 \to 5$ $a_3 \to a_9$ $a_8 \to a_5$ $a_{13} \to a_{15}$
$4 \to 8$ $9 \to 4$ $a_4 \to a_7$ $a_9 \to a_4$ $a_{14} \to a_{13}$
$5 \to 9$ $10 \to 2$ $a_5 \to a_8$ $a_{10} \to a_2$ $a_{15} \to a_{12}$

Figura 3.1: Exemplo de automorfismo.
Fonte: Própria.

- **Isomorfismo-Completo**

Uma vez que o problema do isomorfismo não é classificado nem em P nem em NP-Completo, pesquisadores têm considerado outras abordagens na determinação de sua complexidade. Uma dessas abordagens é defini-lo como estando em sua própria classe de complexidade, a *Isomorfismo-Completo*, e ao mesmo tempo procurar por quais outros problemas possam estar inseridos nesta classe em particular. Outra abordagem está na generalização da noção de NP em noções mais sutis de complexidade, e no estudo do local exato do problema do isomorfismo nestas novas classes. Uma terceira abordagem ainda é tentar determinar quais classes de grafos em que o problema do isomorfismo se situe em P quando estiver restrito a estes grafos.

Quanto à classe própria do problema, a *Isomorfismo-Completo*, Mathon (1979 apud FORTIN, 1996, p.3) mostrou que o problema de contar o número de isomorfismos existentes entre dois grafos pertence a esta classe. Este resultado evidencia ainda mais a conjectura de que o problema do isomorfismo não é NP-Completo, uma vez que a busca por versões de problemas NP-Completos, segundo o autor, parecem ser muito mais difíceis que seus problemas de decisão equivalentes.

Mathon também demonstrou os estritos laços entre o problema do isomorfismo e do automorfismo, onde o isomorfismo é equivalente ao problema de contar o número de automorfismos não triviais existentes em um grafo.

O autor também mostrou que há muitos problemas conhecidos por serem polinomialmente equivalentes ao problema do isomorfismo.

Alguns exemplos de grafos em que o isomorfismo se encontra em Isomorfismo-Completo são: *gra-*

fos bipartidos, grafos linha ("line graphs"), digrafos acíclicos enraizados, grafos cordais, grafos split, grafos orientados transitivamente e grafos regulares. É de se esperar estes resultados, pois muitas dessas classes citadas apresentam certa quantidade de regularidade fazendo com que a obtenção de um isomorfismo se torne mais difícil.

No capítulo 4 será mostrada uma análise criada pela autora deste trabalho juntamente com o orientador, a respeito da equivalência polinomial entre classes de grafos e o problema do isomorfismo. No referido capítulo foi analisado sobre grafos bipartidos e grafos splits, e a relação entre eles com as reduções de NP-Completude.

3.1 Classes de Grafos Onde o Problema de Testar o Isomorfismo Está em P

Retornando à questão dos algoritmos em tempo polinomiais conhecidos para classes especiais de grafos, o primeiro maior resultado neste campo foi com o artigo de Luks (1982 apud FORTIN, 1996, p.6). Ele mostrou que para grafos que possuem graus limitados, por exemplo, com grau máximo menor ou igual a alguma constante K, existe um algoritmo de tempo polinomial para checar o isomorfismo.

Outras propriedades também têm sido estudadas e uma delas, segundo Fortin (1996), se refere ao *genus* de um grafo, onde se quiséssemos fazer a imersão de um grafo em uma superfície tal que duas linhas não se cruzem, pode ser necessária uma superfície com uma estrutura bastante complexa dependendo do grafo. O *genus* é a medida de quão complicado tal superfície pode ser, e a distribuição em *genus* de um grafo é definida como uma sequencia de números g_1, g_2,... tal que g_i é o número de maneiras para se desenhar um grafo em uma superfície de *genus* i.

Filotti e Mayer (1980 apud FORTIN, 1996, p.6) mostraram que se fixarmos um valor K, então os grafos que têm $g_k > 0$ podem ser checados quanto ao isomorfismo em tempo polinomial. Este algoritmo não é muito prático, pois seu tempo de execução é $O(n^{ck})$ onde $c > 300$ é uma constante. *Grafos planares* são um caso especial deste resultado, pois eles podem ser imersos no plano (ou numa esfera) e por isso tem $g_k > 0$ para todos $K \geq 1$. Isso foi primeiramente mostrado por Hopcroft e Wong (1974 apud FORTIN, 1996, p.7), em que grafos planares podem ser checados quanto ao isomorfismo em tempo linear, embora seu algoritmo possua uma grande constante o que faz com que ele seja inapropriado na prática.

Segundo McKay (2013), grafos que possuem grau ou *genus* limitados admitem algoritmos de tempo polinomiais. Porém, os algoritmos resultantes de todas essas classes são na maioria das vezes poucos usados na prática, com exceção de apenas algumas classes de grafos, tais como as árvores e grafos planares. Contudo, existem ainda abordagens práticas que vem sendo tratados de forma satisfatória por meio de heurísticas, que superam os métodos gerais, onde as mais eficientes são baseadas em procedimentos que agrupam vértices de acordo com suas similaridades como as que serão estudados nesta seção.

Ainda, para citar mais exemplos, outra classe de grafos onde o isomorfismo pode ser resolvido eficientemente é a de *grafos de arcos-circulares.* Grafos de arcos circulares são aqueles que podem ser criados ao inserir certo número de arcos em um círculo, criando um vértice para cada arco, e juntando dois vértices caso seus arcos se sobreponham. Uma subclasse de grafos de arcos circulares é a de *grafos*

de intervalo, onde os "arcos" são restritos a um número real em linha. Hsu (1995 apud FORTIN, 1996, p.7) mostrou que existe um algoritmo $O(mn)$ para testar o isomorfismo desses grafos, onde m é o número de arestas, e n é o número de vértices. Este resultado é interessante, segundo o autor, pelo fato de não requerer nenhum parâmetro como constante, além disso, aparenta ser aplicado a um grande conjunto prático de grafos.

Há muitas outras classes específicas de grafos para as quais o problema do isomorfismo pode ser resolvido em tempo polinomial. Por exemplo, Cogis e Guinaldo (1995 apud FORTIN, 1996, p.7) mostram que o isomorfismo para *grafos conceituais* que possuem uma função de ordenação linear podem ser resolvidos em tempo polinomial. Esta classe específica de grafos é de interesse especial à comunidade de inteligência artificial, onde grafos conceituais são usados como técnicas para a representação do conhecimento. Outras classes de grafo tais como *árvores, grafos de permutação* e *partial k-trees*, também possuem problemas de isomorfismo que estão em P.

Como vimos, há muitas classes de grafos em que o problema do isomorfismo pode ser resolvido em tempo polinomial, porém como muitas vezes se tratam de grafos bastante específicos, tais considerações não são muito comuns em situações viáveis reais. Desta forma, na sequencia iremos abordar algoritmos de isomorfismo de grafos com um enfoque prático, baseando-se em Fortin (1996).

3.2 Algoritmos de Isomorfismo - Aplicações Práticas

Um dos fatores a contribuir para essa grande quantidade de esforços a respeito do problema do isomorfismo de grafos são sem dúvidas as muitas aplicações práticas da técnica. Há duas amplas abordagens para resolver o problema do isomorfismo. A primeira abordagem é a partir dos dois grafos a serem comparados procurar um isomorfismo diretamente entre eles. Esta abordagem possui a vantagem de que mesmo que existam vários isomorfismos entre os grafos, somente o primeiro precisa ser considerado.

A segunda abordagem se trata de, a partir de um grafo simples G, computar algumas funções $C(G)$ que retorne uma **rotulagem canônica**; onde um rótulo canônico significa que $C(G) = C(H)$ se G e H forem isomorfos. Tal rotulação não só resolve o problema do isomorfismo, mas possui também um interesse particular em aplicações práticas, como na química orgânica.

A área de processamento de imagens se classifica junto com a química como um grande domínio de aplicação na busca de um isomorfismo eficiente. Os grafos que surgem neste domínio de aplicação são grafos planares que representam as inter-relações entre objetos em uma cena, por isso, como citado na seção anterior, a verificação de um isomorfismo planar pode ser resolvido em tempo linear.

Uma importante ferramenta prática na resolução de problemas de isomorfismo em grafos é o chamado *Invariante de Vértices*, que será tratado a seguir.

- **Invariante de Vértices**

Uma técnica bastante prática, que é independente do algoritmo usado para efetivamente resolver o problema do isomorfismo é o uso da *Invariante de Vértices*. A invariante de vértice é algum número $i(v)$ atribuído ao vértice v tal que caso exista algum isomorfismo que mapeie v para v' então $i(v) = i(v')$.

O exemplo típico de uma invariante de vértice é o *grau* de um vértice: se v e v' são mapeados um ao outro sob algum isomorfismo, então certamente eles devem ter o mesmo grau. É possível notar que o oposto não é verdadeiro, ou seja, se dois vértices têm o mesmo grau, isso não significa que exista necessariamente um isomorfismo entre eles. De fato, se existe uma invariante de vértice computável polinomialmente para o qual o oposto se mantém (exista realmente um isomorfismo), então temos uma solução de tempo polinomial para o problema do isomorfismo.

Existem muitas invariantes de vértices que foram propostas na literatura. Abaixo estão listadas algumas invariantes que estão presentes em um algoritmo para o isomorfismo chamado *Nauty*, o qual será visto com mais detalhes na sequência.

- *twopaths*: atribui um número a v baseado nos vértices alcançáveis ao longo de um caminho de comprimento dois.

- *adjacency triangles*: atribui um número a v baseado no tamanho da vizinhança comum daqueles vizinhos adjacentes de v.

- *k-cliques*: atribui um número a v baseado no número de diferentes cliques de tamanho k que contém v.

- *independent k-sets*: atribui um número a v baseado no número de diferentes conjuntos independentes de tamanho k que contém v.

- *distances*: atribui um número a v baseado no número de vértices que cada distância de 1 à n a partir de v.

Ainda segundo Fortin (1996), fazendo uma análise prática, invariantes de vértices são tipicamente usadas para combinar os resultados de muitas invariantes. Por exemplo, se um vértice tem grau 5, pertence à 2 cliques de tamanho 4, e pode alcançar 9 vértices em caminhos de comprimento 2, então uma invariante composta poderia ser "529". Esta habilidade de compor invariantes levou ao caso em que algoritmos práticos sempre usam o grau como a invariante "base", e outras invariantes vão acrescentando-se a ela para compor o valor final.

Uma observação importante a respeito dessas outras invariantes é que o uso de invariantes de vértices no cálculo do problema do isomorfismo pode não resultar em uma diminuição do espaço de busca realmente, pois pode ser um pouco custoso calculá-las comparado ao tempo total de execução do algoritmo. Com isto verifica-se que muitos problemas de isomorfismo podem ser resolvidos de forma *direta* (quando busca-se diretamente descobrir um mapeamento entre os vértices) em menos tempo do que levaria para calcular as invariantes mais eficazes, além do que a determinação de qual (e se alguma) invariante será melhor para um grafo em particular não é uma tarefa trivial.

Assim, o que é feito realmente é deixar a decisão de se e quando usar uma invariante de vértice para o usuário (exceto para o caso do grau, que é sempre usado). Então se um usuário se depara com um número muito grande de grafos difíceis, é possível realizar uma análise a partir do tempo para determinar experimentalmente qual invariante usar. Outra justificativa para isto é que usar o grau como invariante é suficiente para a maioria dos grafos encontrados na prática, e, portanto, o usuário somente precisa intervir se o grafo possuir uma estrutura bastante regular.

- **Rotulagem Canônica**

Como já citado brevemente a respeito, uma das abordagens de resolução do problema do isomorfismo está em computar algumas funções $C(G)$ que retorne uma rotulagem canônica para o grafo G - onde, se $C(G) = C(H)$ então G e H são isomorfos.

Neste ponto nós iremos adentrar a respeito desta questão e de sua relação com a técnica de invariantes vista anteriormente e introduzir na sequencia o método preferido atualmente para resolução o problema do isomorfismo de grafos: o algoritmo *Nauty* (FORTIN, 1996). Os termos e definições a seguir segue o adotado por McKay (2013).

Para que um algoritmo determine se dois grafos com n vértices são isomorfos ou não, seria preciso testar todos os mapeamentos entre seus vértices, ou seja: $n!$. Porém, o tempo necessário para esta computação torna-o inviável, com exceção de casos em que os grafos sejam muito pequenos ou de classes especiais como já citado (árvores, grafos planares, etc.).

Segundo McKay (2013), testar diretamente se dois grafos são isomorfos possui a vantagem de que um isomorfismo pode ser encontrado muito antes de uma busca exaustiva (*Backtracking Search*) ser finalizada. Por outro lado, esta abordagem é pouco adequada para os problemas onde busca-se a rejeição de isomorfirsmos a partir de um conjunto de grafos ou de identificação de um grafo em uma base de dados. Por esta razão, a abordagem prática mais comum é a **rotulagem canônica**, um processo em que um grafo é renomeado (é encontrado um número que representa o grafo) de tal modo a que os grafos isomorfos são idênticos após a classificação (rotulação). A rotulagem canônica trabalha em um grafo por vez.

Na sequencia será tratado mais formalmente o algoritmo Nauty.

3.3 Nauty

Há diversos algoritmos publicados sobre o problema do isomorfismo. No entanto, ainda de acordo com McKay (2013), a história mostra que as abordagens mais bem sucedidas envolvem o desenvolvimento de heurísticas, onde, dentre elas, as mais eficientes se baseiam na identificação de vértices juntamente com o refinamento de partições do conjunto de vértices.

Isso se deve ao fato de que ao analisar cada grafo separadamente no intuito de identificar diferenças estruturais entre os vértices e agrupá-los de acordo com suas semelhanças, evitamos mapear vértices para os quais tenham sido identificadas certas diferenças (por exemplo, graus distintos). É exatamente esta a ideia que rege os algoritmos denominados *algoritmos de refinamento*.

Este paradigma de "refinamento individualizado" foi introduzido por Parris e Read (1969), no entanto, o primeiro programa que pode lidar tanto com grafos regulares com centenas de vértices quanto com grafos com grandes grupos de automorfismo foi o de McKay (1978, 1980), que mais tarde se tornou conhecido como **Nauty**. Sua principal vantagem sobre os programas anteriores foi o seu uso inovador de automorfismos para "podar" a pesquisa.

Portanto, a identificação de vértices juntamente com o refinamento de partições do conjunto de vértices ocorre da seguinte maneira, primeiramente identificamos vértices e suas características, as chamadas *invariantes*, como já visto. Com isso provocamos um particionamento inicial do conjunto

de vértices, e posteriormente, utilizamos procedimentos de refinamento com o objetivo de identificar características adicionais que possibilitem distinguirmos vértices não similares.

Assim, nesta seção serão vistas definições a respeito de colorações (partições) e invariantes. Será visto também conceitos a respeito da busca em árvore, que está no núcleo da maioria dos algoritmos de isomorfismo em grafos. As definições e conceitos utilizados na sequencia estão baseados em Fortin (1996) e McKay (2013).

- **Partições**

No centro de operações do Nauty está a ideia de *partição ordenada*, ou *coloração*. Uma partição, ou coloração, π divide os vértices do grafo em subconjuntos disjuntos não vazios de V (chamados *células*): $V_1, V_2, \ldots, V_k$. O número de cores, ou seja, k, é denotado por $|\pi|$. Portanto, uma célula de π é um conjunto de vértices com alguma determinada cor. Por fim, se a um grafo G está associada uma coloração π, diz-se que este é um *grafo colorido* denotado por G_π e dois vértices u e v têm a mesma cor se $\pi(u) = \pi(v)$.

Uma partição com somente conjuntos unitários vai ser chamada de *partição folha* (ou no caso específico do Nauty eles são referidos como *partições* ou *colorações discretas*, entretanto o termo *partição folha* destaca o seu papel como folhas em uma árvore de busca). Podemos notar também que uma coloração discreta é uma permutação em V. Portanto, o ponto chave desta questão é que quando encontramos uma partição folha, ela define uma re-rotulação do grafo onde o vértice na célula V_i fica rotulado como i.

Se π e π' são colorações, então π' é *mais fina ou igual* à π – escrito por: $\pi' \Upsilon \pi$ - se, para qualquer par de vértices, o fato de ambos pertencerem à mesma classe de coloração em π' implicar que estes também pertencem à mesma classe de coloração em π. Em termos formais: $\pi(v) < \pi(w) \rightarrow \pi'(v) < \pi'(w)$ para todo $v, w \in V$. Isto implica que cada célula de π' é um subconjunto de uma célula de π, mas o inverso não é verdadeiro.

Uma vez que uma coloração particiona V em células, ela é frequentemente chamada de *partição*, como já visto. Entretanto, é importante destacar que as cores aparecem em certa ordem específica por isso a necessidade de definições como “mais fino”.

Por exemplo, consideremos o grafo da Figura 3.2. São dadas três partições folha para os vértices no grafo, e associado a cada partição está a matriz de adjacência onde a ordem dos vértices nas linhas e colunas correspondem à ordem dos vértices nas partições. Automorfismos são conhecidos por encontrar duas partições folha distintas que (após a re-rotulação) dão origem ao mesmo grafo. No exemplo $|c|a|b|$ e $|a|c|b|$ proporcionam a mesma matriz e portanto definem um automorfismo.

Há duas operações principais que o *Nauty* realiza em partições: *refinar uma partição* e *gerar os filhos de uma partição*. O objetivo dos procedimentos de refinamento é tentar descobrir diferenças estruturais entre os vértices de uma partição e produzir, a cada iteração, uma coloração mais fina que a coloração anterior. É durante o refinamento que as *invariantes* dos vértices podem ser usadas, sendo calculadas para cada vértice em todo o grafo para criar uma partição inicial, e então uma nova invariante é calculada limitado às células da partição para tentar distingui-las melhor.

Assim, se para alguns vértices de uma classe de coloração forem detectadas características que os diferenciam dos demais, a eles atribui-se uma nova cor. Este processo, denominado refinamento, termina

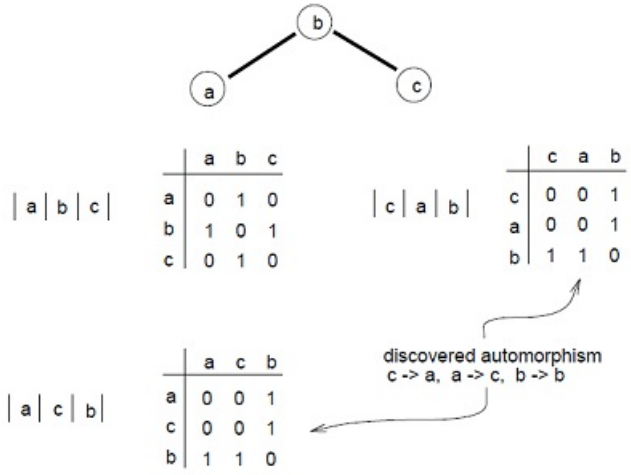


Figura 3.2: Descobrindo um automorfismo pela comparação de partições folha.
Fonte: Fortin (1996, p.13).

quando nenhum vértice pode ser mais distinguido. Diz-se então que a coloração tornou-se *estável*.

Para exemplificar o exposto, assuma que $d(v, S)$ retorna o número de vizinhos de v que estão no conjunto S. Nós podemos então definir uma operação de refinamento como:

1. $\pi \leftarrow$ a partição inicial $V_1, V_2, \ldots V_k$ onde para todos os vértices $x, y \in V_i$ temos que $d(x, V) = d(y, V)$.
2. Selecione um $V_i \in \pi$ tal que V_i tenha mais que um elemento.
3. Para cada $v \in V_i$, calcule a seguinte sequência: $a_v = [d(v, V_1), \ldots, d(v, V_k)]$
4. Divida V_i em um número de subconjuntos, de modo que os vértices em cada subconjunto tenham o mesmo valor para a_v.

Na verdade, nos passos acima iniciamos selecionando V_1 e percorremos todos os V_i (incluindo aqueles criados depois da divisão) até que não seja possível quebrar mais em nenhuma célula.

Nós geramos os filhos de uma partição ao escolher a primeira célula V_i que tenha mais do que um membro, e então para cada vértice $v \in V_i$ criamos uma partição-filha que é: $V_1, \ldots, V_{i-1}, \{v\}, V_i/\{v\}, V_{i+1}, \ldots, V_k$. Ou seja, criamos um novo filho para o vértice v quebrando a célula V_i em duas células, uma que contém somente v e outra que contém V_i sem v.

- **Conclusão**

Como conclusão, podemos observar o sucesso de um programa como o *Nauty* sobre uma abordagem mais direta para o problema do isomorfismo de grafos. Não é difícil especular porque *Nauty* se comporta melhor. Apesar de os dois programas fazerem uso de invariantes de vértices, o *Nauty* usa-os em seu processo de refinamento, enquanto que o algoritmo direto usa-os quando decide se expande um isomorfismo encontrado ou não.

A principal diferença entre as duas técnicas parece ser seus usos dos resultados parciais. Sempre que *Nauty* explora uma porção da árvore de busca e encontra um automorfismo, ele irá trabalhar

com a informação encontrada, descartando partições da árvore de busca. Já a busca direta não utiliza informações obtidas previamente, “desperdiçando” assim um conhecimento que poderia ser aproveitado (FORTIN, 1996).

Capítulo 4

Resultados Obtidos

Neste capítulo será apresentado um resultado original obtido pela autora deste trabalho juntamente com o seu orientador, mais especificamente, é mostrado que o problema do isomorfismo de grafos split é isomorfismo-completo, ou seja, tão difícil quanto o problema do isomorfismo no caso geral. Para tal, faremos uma redução do problema do isomorfismo em grafos bipartidos.

É importante observar entretanto, que embora a redução mostrada neste trabalho tenha sido obtida de maneira independente, os autores encontraram outra redução que assemelha-se ao que Lueker (1977) também obteve. A diferença é que neste trabalho foi analisada uma redução de grafo bipartido para grafo split, já Lueker realizou uma redução a partir de um grafo qualquer para grafo cordal. Porém, Stewart (1978) observou que esse mapeamento realizado por Lueker também demonstra o isomorfismo completude para grafos split, uma vez que ela é uma subclasse de grafos cordais, por isso a semelhança com o obtido neste trabalho.

Primeiramente cabe destacar que o resultado obtido baseia-se na ideia de que a partir de um par de grafos quaisquer, podemos encontrar uma transformação tal que o par de grafos gerados serão isomorfos se e somente se os originais também forem, ou seja, se o par de grafos originais são isomorfos os novos grafos gerados também serão.

Assim, tomando como base a Figura 4.1 podemos ver três grafos bipartidos onde a partir deles foram gerados grafos splits correspondentes da seguinte maneira:

Dado um grafo bipartido $G = (V, E)$, onde V é particionado em dois conjuntos A e B, sendo: $\{a_1,\ a_2,\ \ldots,\ a_k\} \in A$, e $\{b_1,\ b_2,\ \ldots,\ b_k\} \in B$, o grafo split correspondente ao grafo bipartido será $G' = (V', E')$, onde:

$V' = V \cup X$, onde: $X = \{u_{ij} | a_i b_j \in E(G)\}$

$E' = A \cup B \cup X$, onde: $A = \{a_i u_{ij} \mid a_i b_j \in E(G)\}$, $B = \{b_i u_{ji} \mid b_i a_j \in E(G)\}$ e $X' = \{$todas as arestas possíveis entre os elementos de $X\}$

A ideia é que dois grafos bipartidos são isomorfos se e somente se seus grafos split correspondentes forem isomorfos entre si.

Isto é possível notar na Figura 4.1, onde os Grafos Bipartidos 1 e 2 apesar de aparentarem serem diferentes eles são isomorfos. É possível notar que o Grafo Split gerado por eles é o mesmo.

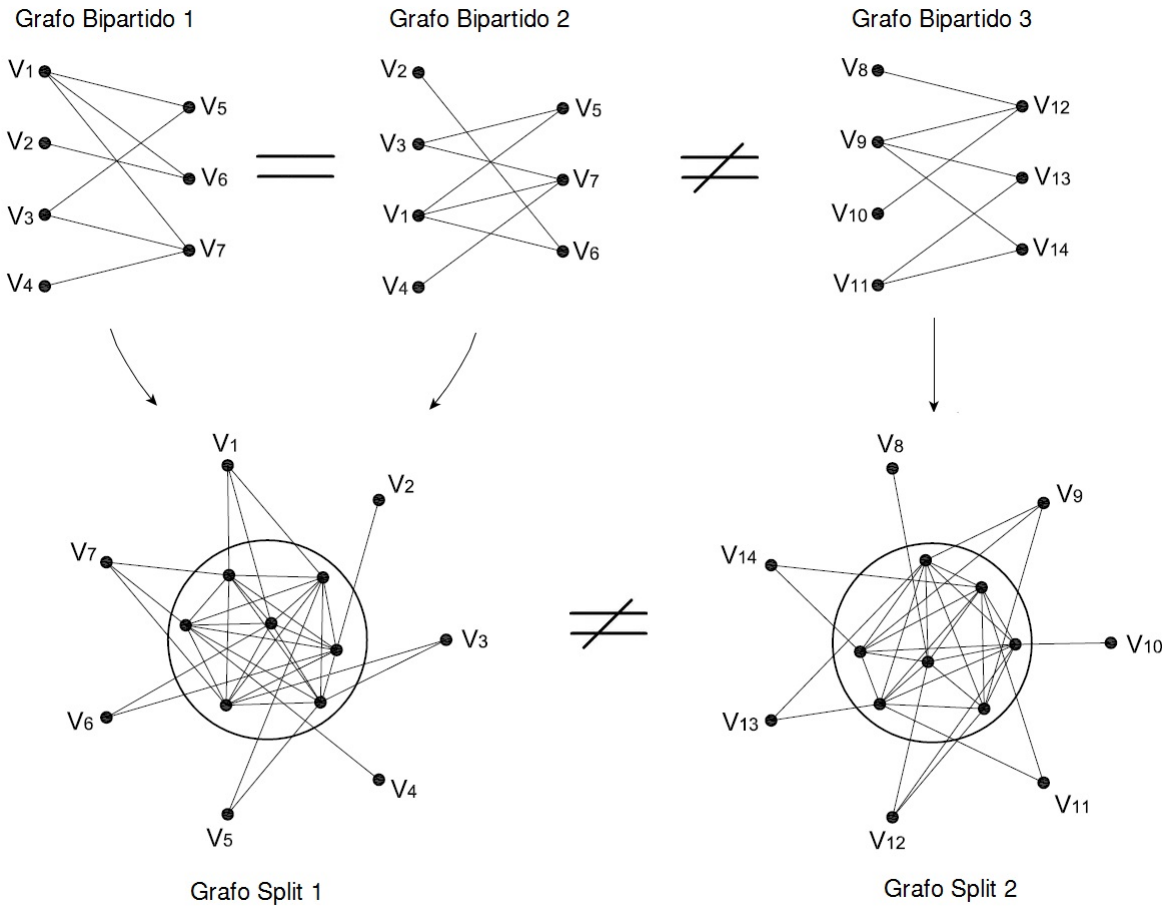


Figura 4.1: Relação entre Grafos Bipartidos e Splits correspondentes.
Fonte: Própria.

Já no caso do Grafo Bipartido 3, ele não é isomorfo aos outros dois, mesmo sendo igual a quantidade de vértices e de grau de cada um dos conjuntos independentes aos dos outros grafos Bipartidos. Por exemplo, em todos os grafos bipartidos existem sete vértices divididos em dois conjuntos independentes, um com quatro e outro com três vértices. No conjunto com quatro, há um vértice de grau 3, um de grau 2 e dois vértices com grau 1. No conjunto menor há um vértice de grau 3 e dois com grau 2. Isso ocorre nos três grafos bipartidos, porém apenas o terceiro não é isomorfo aos outros dois, devido às relações de incidências entre arestas e vértices neste grafo diferir das incidências nos demais grafos. Por consequência, o Grafo Split gerado pelo Grafo Bipartido 3 também não corresponde ao grafo Split anterior.

Podemos agora analisar com mais detalhes uma relação citada no capítulo 3, em que, segundo Mathon (1979), há muitos problemas conhecidos por serem polinomialmente equivalentes ao problema do isomorfismo. Podemos notar que isto é semelhante às reduções de NP-Completude já analisadas anteriormente também. Assim, um problema pode ser redutível a outro quando existe um algoritmo que transforma qualquer instância de um problema geral em uma instância de outro mais específico. Da mesma forma ocorre com problemas Isomorfismo-Completo, onde ao transformarmos um grafo qualquer em algum com certas propriedades mais específicas, podemos analisar seu comportamento e procurar

por uma solução, que pode ser tão difícil quanto no caso geral, ou neste caso tornar-se polinomial a sua resolução.

Para exemplificar, uma estratégia para provarmos que o problema do isomorfismo continua difícil em uma certa classe de grafos mais específica (por exemplo, os grafos cordais) é feita uma redução da mesma forma que é feita para NP-Completude da seguinte maneira: a partir de um par de grafos gerais os transformamos em grafos cordais e analisamos o isomorfismo nestes novos grafos. O resultado deve ser tão difícil quanto em problemas NP-Completo, pois conforme já visto anteriormente, grafos cordais se encontram na classe Isomorfismo-Completo.

Outro exemplo seria a transformação de um grafo geral em um bipartido. Sabemos que grafos bipartidos também se encontram em Isomorfismo-Completo e portanto é difícil testar sua resolução.

Desta transformação de grafos gerais para bipartidos e da relação entre estes e grafos splits analisados anteriormente é que foram concluídos alguns aspectos neste trabalho e que estão relatados a seguir.

Para resolver o problema do isomorfismo entre dois grafos gerais, nós podemos da mesma forma que na abordagem anterior (em que trabalhamos com reduções para mostrar que um problema continua difícil em uma classe de grafos mais específica), transformá-los primeiramente em grafos bipartidos e a partir destes, em grafos split conforme descrito anteriormente. Assim, se os grafos originais forem isomorfos os splits finais gerados também serão isomorfos.

Com isso, concluímos que uma vez sabendo que é difícil testar o isomorfismo em grafos bipartidos, a verificação do isomorfismo em grafos split também deve ser difícil, pois, caso contrário, teríamos uma solução polinomial para grafos bipartidos. Por consequência, e paralelamente ao estudo teórico realizado anteriormente sobre o problema do isomorfismo, podemos dizer que grafos split também pertencem à classe mais difícil de resolução do problema do isomorfismo, a Isomorfismo-Completo.

4.1 Conclusões

É possível verificar que um dos resultados interessantes deste trabalho foi o embasamento teórico a respeito dos conceitos necessários para tratar o problema do isomorfismo de grafos, como por exemplo, o estudo da teoria de grafos, da teoria de complexidade e do próprio estado da arte do problema.

Foi utilizada uma abordagem que tratasse cada tema em questão sob um enfoque abrangente, procurando analisar conceitos, definições e os principais embasamentos teóricos que auxiliassem o desenvolvimento do problema do isomorfismo e sua posterior análise.

Como já citado, o problema do isomorfismo em grafos possui bastante importância tanto teórica quanto prática. Há também muitos esforços na busca por um algoritmo que o resolva em tempo polinomial. Sendo assim, este trabalho procurou analisar os principais aspectos que influenciam e caracterizam o problema do isomorfismo em diferentes tipos de grafos de uma maneira global, sem se ater em um único tema ou assunto específico ao longo de seu desenvolvimento.

Por fim, o resultado original obtido com este trabalho - que se deu através da transformação de grafos bipartidos para grafos split e das conclusões inferidas por essa transformação - também contribuiu

para produção de um material científico a respeito do problema em questão, colaborando conjuntamente com a comunidade científica, sendo esta a principal motivação para o desenvolvimento deste trabalho.

Referências Bibliográficas

[1] ARORA, Sanjeev, BARAK, Boaz. **Computational Complexity: A Modern Approach**. Princeton University. 2009.

[2] BARABASI, Albert László, BONABEAU, Eric. ***Scale-Free Networks***. Scientific American. University of Notre Dame, p.50-59, mai, 2003.

[3] BONDY, J. A. and MURTY, U, S, R. **Graph Theory with Applications**. Department of Combinatorics and Optimization. University of Waterloo. Ontario, Canadá. 1976.

[4] COGIS, O. and GUINALDO, O. *A linear descriptor for conceptual graphs and a class for polynomial isomorphism test.* In **Proceedings of the 3rd International Conference on Conceptual Structures**, p. 263-277, Santa Cruz, USA, August 1995.

[5] CORMEN, Thomas H., LEISERSON, Charles E., RIVEST, Ronald L., STEIN, Clifford. **Introduction to Algorithms**. The MIT Press. 2001.

[6] DASGUPTA, Sanjoy, PAPADIMITRIOU, Christos H, VAZIRANI, Umesh V. **Algorithms**. McGraw-Hill, September, 2006.

[7] FILLOTTI, I. and MAYER, J. *A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus.* In **Proceedings of the 12th ACM Symposium on the Theory of Computing**, p. 235-243, 1980.

[8] FORTIN, Scott. **The Graph Isomorphism Problem**. Technical Report TR 96-20. University of Alberta. Department of Computing Science. Edmonton, Alberta, Canadá. 1996.

[9] GOLDREICH, Oded. **P, NP, and NP-Completeness**. The Basics of Computational Complexity. Cambridge. 2010.

[10] HOPCROFT, J. and WONK, J. *A linear time algorithm for isomorphism of planar graphs.* **Proceedings of the 6th ACM Symposium on the Theory of Computing**, p. 172-184, 1974.

[11] HSU, W. *O(mn) algorithms for the recognition and isomorphism problem on circular-arc graphs.* **SIAM Journal of Computing**, 24(3):411-439, June 1995.

[12] LUEKER, G. S. and BOOTH, K. S. *A Linear Time Algorithm for Deciding Interval Graph Isomorphism.* **Journal of the Association for Computing Machinery**. 1977.

[13] LUKS, E. *Isomosphism of graphs of bounded valence can be testes in polynomial time.* **Journal of Computer and System Sciences**, 25:42-65, 1982.

[14] MATHON, R. **A note on the graph isomorphism counting problem**. Information Processing Letters, 8:131-132. 1979.

[15] McKAY, B. D. *Computing automorphisms and canonical labellings of graphs.* In **Combinatorial Mathematics, Lecture Notes in Mathematics, 686**. Springer-Verlag, Berlin, 223-232. 1978.

[16] McKAY, B. D. **Practical graph isomorphism**. Congressus Numerantium, Vol. 30. Department of Computer Science. Australian National University. Canberra, Australia. pp. 45-87. 1980.

[17] McKAY, B. D. **Practical Graph Isomorphism, II**. Research School of Computer Science. Australian National University, Canberra. 2013.

[18] MILLER, Gary L. *Graph Isomorphism, General Remarks.* In **Journal of Computer and System Sciences**. Applied Mathematics Group, Department of Mathematics. Massachusetts Institute of Technology. Cambridge, Massachussets. 1978.

[19] PARRIS, R. and READ, R. C. **A coding procedure for graphs**. Scientific Report. UWI/CC 10. Univ. of West Indies Computer Centre. 1969.

[20] STEWART, L. **Cographs – a class of tree representable graphs**. M.Sc. thesis. University of Toronto. 1978; also Technical Report 126/78, Department of Computer Science, University of Toronto, 1978.

[21] WEST, Douglas B. **Introduction to Graph Theory**. Second Edition. University of Illinois. 2002.