UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Gustavo Tarciso da Silva

VERIFICAÇÃO DE ISOMORFISMO DE GRAFOS VIA FORMAS NORMAIS PRIMÁRIAS

FLORIANÓPOLIS
2020

GUSTAVO TARCISO DA SILVA

VERIFICAÇÃO DE ISOMORFISMO VIA FORMAS NORMAIS PRIMÁRIAS

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Ciências da Computação

Florianópolis, novembro de 2020

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

da Silva, Gustavo Tarciso
Verificação de Isomorfismo de Grafos via Formas Normais Primárias / Gustavo Tarciso da Silva ; orientadora, Jerusa Marchi, 2020.
98 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2020.

Inclui referências.

1. Ciências da Computação. 2. Formas Normais Primárias. 3. Isomorfismo de Grafos. 4. Famílias de Teorias Proposicionais. 5. Satisfação Booleana. I. Marchi, Jerusa. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

UNIVERSIDADE FEDERAL DE SANTA CATARINA

GUSTAVO TARCISO DA SILVA

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Ciências da Computação, sendo aprovada em sua forma final pela banca examinadora:

Orientador(a): Profa. Dra. Jerusa Marchi
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Alvaro Junio Pereira Franco
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Rafael de Santiago
Universidade Federal de Santa Catarina - UFSC

Florianópolis, 01 de novembro de 2020

Resumo

O problema de isomorfismo entre grafos é um dos problemas cuja complexidade computacional ainda não foi definida [Skiena 2008], portanto, a investigação de sua classe de complexidade e de algoritmos eficientes ainda está em aberto. Desta forma, o objetivo deste trabalho foi produzir um método algorítmico para detecção de isomorfismo entre dois grafos. Através de uma modelagem dos grafos em teorias proposicionais na forma normal conjuntiva, e então, aplicar um algoritmo de transformação dual para encontrar uma teoria equivalente de tamanho reduzido na forma normal disjuntiva, composta apenas por implicantes primários. Com o conjunto de implicantes primários, o próximo passo foi contar a quantidade de modelos desta nova teoria reduzida, cujo objetivo é categorizar as teorias em famílias. Estas teorias foram categorizadas no trabalho de [Polya 1940] e de [Bittencourt, Marchi e Padilha 2003], e concluem que se duas teorias pertencem a mesma famílias, logo elas são congruentes. A motivação deste trabalho vem desta propriedade de congruência entre as teorias. O resultado obtido foi o desenvolvimento de um algoritmo que é capaz de identificar o isomorfismo entre dois grafos. Entretanto, durante as análises alguns casos de não-isomorfismo foram identificados erroneamente como isomorfos. Estes grafos aparentemente possuem uma característica em comum, tornando os resultados, mesmo que incorretos, promissores. Com isto conclui-se que este trabalho cumpriu com os objetivos definidos, entretanto, há a necessidade de investigar se o método de fato só falha para o conjunto de grafos com a característica identificada.

Palavras-chave: Teoria de Grafos, Isomorfismo, Algoritmos, Lógica Proposicional, Teoria da Computação, Formas Normais Primárias, Implicantes Primários, Congruência.

Abstract

The Graph Isomorphism Problem is one of the problems whose the computational complexity was not defined yet [Skiena 2008], so, an investigation of its complexity, and efficient algorithms, is still open. Thus, the objective of this study was to produce an algorithmic method to detect isomorphism between two graphs. Through modeling the graphs in propositional theories in the normal conjunctive form, and then applying a dual-transform algorithm to find an equivalent theory of reduced size in the normal disjunctive form, composed only of prime implicants. With the set of prime implicants, the next step was to count the number of models of this new reduced theory, whose objective is to categorize theories in families. These theories were categorized in the work of [Polya 1940] and [Bittencourt, Marchi e Padilha 2003], and both conclued that if two theories belong to the same families, they are therefore congruent. The motivation for this study comes from this property of congruence between theories. The result obtained was the development of an algorithm that is able to identify the isomorphism between two graphs. However, during the analysis some cases of non-isomorphism were mistakenly identified as isomorphs. These graphs apparently have a common characteristic, making the results, even if incorrect, promising. Therewith it is concluded that this work fulfilled the defined objectives, however, there is a need to investigate if the method in fact only fails for the set of graphs with the identified characteristic.

Keywords: Graph Theory, Isomorphism, Algorithm, Propositional Logic, Theory of Computation, Prime Normal Forms, Prime Implicants, Congruency.

Lista de ilustrações

Figura 1 – Exemplo de isomorfismo entre dois grafos 20
Figura 2 – Grafo representando a relação entre os estados gerados do Exemplo 1 . 31
Figura 3 – Grafo com a quantidade de modelos associados a cada estado do Exemplo 1 32
Figura 4 – Grafo com a quantidade de modelos associados exclusivamente a cada estado do Exemplo 1 33
Figura 5 – Exemplo de dois grafos não-isomorfos com quantidades diferentes de vértices 36
Figura 6 – Exemplo de dois grafos não-isomorfos com quantidades diferentes de arestas 36
Figura 7 – Exemplo de dois grafos não-isomorfos com vértices de graus diferentes . 37
Figura 8 – Diagrama de classes da implementação 39
Figura 9 – Grafos G_1 e G_2 utilizados como exemplo na execução do programa . . 52
Figura 10 – Execução do programa utilizando G_1 como entrada 52
Figura 11 – Execução do programa utilizando G_2 como entrada 53
Figura 12 – Grafo G_3 utilizado para exemplificar não-isomorfismo com G_1 e G_2 . . 53
Figura 13 – Execução do programa utilizando G_3 como entrada 54
Figura 14 – Resultado incorreto observado nos testes do conjunto de grafos com 6 vértices 56
Figura 15 – Resultado incorreto observado nos testes do conjunto de grafos com 7 vértices 57
Figura 16 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices 57
Figura 17 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices 58
Figura 18 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices 58

Sumário

1 INTRODUÇÃO 13
1.1 Objetivo 13
1.1.1 Objetivo Geral 13
1.1.2 Objetivos Específicos 14
1.2 Estrutura do Documento 14

2 *FUNDAMENTAÇÃO TEÓRICA* 17
2.1 Grafos 17
2.1.1 Definição 17
2.1.2 Representação de Grafos 17
2.1.3 Caminhos e Ciclos 18
2.1.4 Grafo Conexo 19
2.1.5 Grafo Completo 19
2.1.6 Isomorfismo de Grafos 19
2.1.7 Determinando se Dois Grafos são Isomorfos 20
2.2 Formas Normais Primárias 21
2.2.1 Lógica Proposicional 21
2.2.2 Satisfação Booleana 23
2.2.3 Equivalência Lógica 23
2.2.4 Formas Normais Canônicas 24
2.2.5 Implicantes Primários 25
2.2.6 Transformação Dual 26
2.2.6.1 Representação da Teoria 27
2.2.7 Algoritmo de Transformação Dual 28
2.2.8 Propriedades das Formas Normais Primárias 30
2.2.9 Contagem de Modelos 30

3 MÉTODO PARA DETECÇÃO DE ISOMORFISMO ENTRE GRAFOS 35
3.1 Pré-processamento dos Grafos para Verificação de Isomorfismo 35
3.2 Modelagem dos Grafos 37
3.3 Entrada de Dados 38
3.4 Implementação 39
3.5 Transformação Dual 41
3.5.1 Armazenamento da Teoria 41
3.5.2 Geração do Gap 42
3.5.3 Ordenação dos literais 43

3.5.4 Preparação do algoritmo de Transformação Dual 44
3.5.5 Transformação Dual - Obtenção dos Implicantes Primários 45
3.6 Contagem de Modelos **49**
3.7 Verificação do Isomorfismo **51**
3.8 Execução do Programa **51**

4 RESULTADOS E DISCUSSÃO **55**
4.1 Execução dos Testes **55**

5 CONCLUSÃO **59**
5.1 Considerações Finais **59**
5.2 Trabalhos Futuros **59**

REFERÊNCIAS **61**

APÊNDICES **63**

APÊNDICE A – ARTIGO **65**

APÊNDICE B – CÓDIGO FONTE **77**

1 Introdução

Muitos problemas do mundo real podem ser modelados em forma de grafos, dessa forma, diversos algoritmos em grafo podem ser utilizados para resolução desses problemas. Tais algoritmos possuem complexidades diferentes. Alguns podem ser da ordem de complexidade de tempo determinístico polinomial, já outros podem apresentar solução em uma ordem de complexidade de tempo não-determinístico polinomial. Há também alguns cuja ordem de complexidade ainda não é certa, e que é um dos nossos objetos de estudo, o problema de isomorfismo entre grafos, reside nesse conjunto de problemas que não se sabe ainda a qual classe de complexidade pertence [Skiena 2008].

O outro objeto de estudo é a lógica proposicional, que por sua vez, pode ser utilizada para modelar a resolução de problemas em grafos através do método da redução. Um problema em grafos, que pode estar na classe NP pode ser reduzido, ou seja, transformado, em um problema do campo da lógica já conhecido: a satisfação booleana (SAT). SAT foi o primeiro problema a ser provado pertencer a classe NP-completo [Cook 1971].

Já que não se sabe ainda a qual classe de complexidade computacional o problema de isomorfismo entre grafos pertence, e os algoritmos para verificação do isomorfismo são da ordem de complexidade fatorial no pior caso [Skiena 2008], serão estudadas as propriedades desses dois campos citados acima, afim de buscar relações entre eles, e tentar criar uma ponte entre essas duas formas de representar o problema, buscando assim, uma forma alternativa de resolver o problema de isomorfismo entre grafos.

1.1 Objetivo

1.1.1 Objetivo Geral

O objetivo deste projeto é tentar desenvolver um método algorítmico de checagem de isomorfismo entre dois grafos.

Um par de grafos é considerado isomorfo quando há uma função que mapeie os vértices de um grafo para outro, de forma que as propriedades do grafo sejam as mesmas, tornando os grafos equivalente. As sentenças lógicas, por sua vez, possuem uma característica semelhante a esta, pois dada uma sentença, é possível obter uma sentença lógica equivalente apenas trocando seus símbolos, de forma que uma função mapeie os símbolos trocados mantendo suas propriedades.

Para saber se duas sentenças lógicas, que serão apresentadas neste trabalho como teorias proposicionais, podem ser consideradas equivalentes e ter a mesma representação

simbólica, é necessário obter os conjuntos de Implicantes e Implicados primários. Uma vez que temos o conjunto de Implicantes e Implicados primários, podemos utilizá-los em conjunto com a quantidade de símbolos da teoria proposicional para descobrir de qual família de teorias proposicionais ela faz parte.

Desta forma, será representando grafos através de teorias proposicionais, e encontrando seus conjuntos de implicantes primários, que será investigado se é possível decidir se um par de grafos é ou não isomorfo, de modo que seja possível verificar qualquer relação entre a existência ou não de isomorfismo entre os dois grafos avaliados, a partir das famílias que as teorias proposicionais geradas a partir dos grafos pertencem.

Portanto, será investigado se existe uma relação entre isomorfismo de grafos e a família na qual as teorias proposicionais, geradas a partir dos grafos, pertencem. Caso haja relação, será investigado também se esta relação é válida para todo caso.

Caso não seja observada relação entre a quantidade de modelos nas teorias proposicionais obtidas através dos grafos e a presença de isomorfismo entre estes grafos, será avaliado também se, para algum conjunto específico de grafos, essa verificação funciona.

1.1.2 Objetivos Específicos

1. Estudar o problema do isomorfismo entre grafos;
2. Modelar grafos em forma de teorias proposicionais;
3. Obter os implicantes primários das teorias proposicionais obtidas;
4. Contar o número de modelos das teorias proposicionais a partir do seu conjunto de implicantes primários;
5. Verificar se dois grafos isomorfos são de uma mesma família de teorias proposicionais segundo os trabalhos de [Polya 1940] e [Bittencourt, Marchi e Padilha 2003];

1.2 Estrutura do Documento

Este trabalho está dividido em cinco capítulos, sendo este o capítulo introdutório. No capítulo 2 serão abordados conceitos de grafos e lógica proposicional, além dos Implicantes e Implicados primários e como obtê-los. Estes conceitos serão fundamentais para o desenvolvimento do trabalho. No capítulo 2 também serão abordados dois algoritmos que foram utilizados como base para o desenvolvimento do presente trabalho.

No capítulo 3 será abordado a implementação do método para identificação do isomorfismo entre grafos, onde serão discutidos alguns detalhes de implementação e a execução de alguns exemplos.

No capítulo 4 serão discutidos os resultados dos testes feitos a partir de benchmarks criados, e também a forma como os benchmarks foram gerados. O capítulo 5 apresenta a conclusão do trabalho, e possibilidades de trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo serão tratadas as definições fundamentais sobre grafos, problema de isomorfismo entre grafos, lógica proposicional e formas normais primárias, que servirão como base para o desenvolvimento deste trabalho.

2.1 Grafos

A seção a seguir busca definir conceitos básicos sobre grafos, focando em definições, propriedades de grafos, o problema de isomorfismo entre dois grafos, e as formas de decidir se há isomorfismo entre os grafos, baseando-se nos livros [Nievergelt 2002], [Johnsonbaugh 1986] e [Skiena 2008].

2.1.1 Definição

Diversos problemas do mundo real como cidades conectadas por rodovias, computadores conectados em uma rede, casas conectadas em uma rede elétrica, e outros problemas com essa mesma tipologia podem ser representados, modelados, e até mesmo resolvidos, utilizando teoria de grafos.

Um grafo G é constituído por um conjunto não nulo de vértices V e um conjunto de arestas E, ambos finitos, sendo denotado da forma $G = (V, E)$. Um grafo pode ser classificado como dirigido ou não dirigido.

Em grafos não dirigidos, cada aresta $e \in E$ está associada a um par *não ordenado* de vértices v e w, representado por $e = \{v, w\}$, onde a ordem que os vértices estão dispostos não importa, indicando somente que há uma relação entre aqueles dois vértices, sem indicar o sentido da relação. Portanto, $e = (v, w) \equiv e = (w, v)$.

Já em um grafo dirigido, cada aresta $e \in E$ está associada a um par *ordenado* de vértices v e w, representado por $e = (v, w)$, onde a ordem que os vértices estão dispostos informa de qual vértice a aresta está partindo, e para qual vértice aquela aresta está se dirigindo. Portanto $e = (v, w) \not\equiv e = (w, v)$.

Dois vértices v e w, cuja aresta e incide sobre eles, são ditos adjacentes. A quantidade de arestas que incidem em um vértice v é denominada grau do vértice v.

2.1.2 Representação de Grafos

A forma mais intuitiva e de fácil visualização de se representar um grafo é através de um diagrama, onde os vértices são representados por círculos, e as arestas são representadas

por linhas que ligam esses círculos. Em um grafo dirigido, é utilizada uma seta indicando de qual vértice partiu a aresta e para qual vértice ela está indo.

Uma forma de representar um grafo em um computador é através da matriz de adjacência. Para obter a matriz de adjacência é necessário rotular os vértices, e os índices das linhas e das colunas da matriz devem considerar tais rótulos. A entrada da matriz é 1 se os vértices são adjacentes, caso contrário, será 0.

Como por exemplo no grafo *G = (V, E)*, onde *V = {v, w, y}* e *E = {{v, w}, {w, y}}* as matrizes de adjacência dos grafos dirigidos e não dirigidos, onde a primeira linha e primeira coluna correspondem ao vértice v, a segunda linha e segunda coluna correspondem ao vértice w, e a terceira linha e terceira coluna correspondem ao vértice y, seriam:

Grafo não dirigido: Grafo dirigido:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Exemplo de matrizes de adjacência de grafos dirigidos e não dirigidos [Nievergelt 2002]

Uma forma alternativa de representação de grafos em um programa de computador, e que pode se mostrar mais eficiente para grafos esparsos, é utilizando listas de adjacência, onde estruturas de dados em listas encadeadas são usadas para guardar os vértices adjacentes de cada um dos vértices do grafo.

Para cada grafo, deve-se guardar o número de vértices, e rotular cada vértice com um identificador único, como um número de 1 a v_n, onde v_n é o número total de vértices do grafo, e as arestas são depois representadas em um conjunto de listas encadeadas. Caso o grafo seja não dirigido, uma aresta *(x, y)* aparece em duas listas de adjacência, onde x aparece na lista de adjacência de y, e y aparece na lista de adjacência de x. No grafo, é possível ainda guardar o grau de cada vértice.

2.1.3 Caminhos e Ciclos

Imaginando que os vértices de um grafo são cidades, e as arestas são as rodovias que ligam essas cidades, um caminho corresponde a uma cidade de partida, passando por algumas cidades pelas rodovias que as ligam, e terminando em uma cidade destino.

Considerando os vértices v_0 e v_n em um grafo, um caminho de v_0 até v_n de tamanho n é uma sequência de $n + 1$ vértices e n arestas, começando no vértice v_0 e terminando no

vértice v_n, sem repetição de vértices. Em resumo, um caminho começa em um vértice v_0, que vai para o vértice v_1 através da aresta e_1, que vai para o vértice v_2 pela aresta e_2, e assim por diante, até alcançar v_n.

Sendo um caminho definido por (v_0, e_1, v_1, e_2, v_2, ..., v_{n-1}, e_n, v_n), onde cada aresta e_i incide nos vértices v_{i-1} e v_i para $i = 1, \ldots, n$.

Um ciclo ocorre quando, em um caminho, um ou mais vértices podem ser percorridos mais de uma vez.

2.1.4 Grafo Conexo

Um grafo é considerado conexo quando cada vértice está conectado a qualquer outro vértice através de um caminho, como por exemplo, *G = (V, E)*, cujo *V = {v, w, y}* e *E = {{v, w}, {w, y}}*. Esse grafo é considerado conexo, pois mesmo não possuindo uma aresta que liga diretamente os vértices v e y, pode-se caminhar de v para w, e depois de w para y, portanto, há um caminho que liga indiretamente v e y. Quando não se consegue alcançar um vértice a partir de outro, o grafo é considerado desconexo.

2.1.5 Grafo Completo

Um grafo é considerado completo se cada par de vértices está conectado diretamente entre si, ou seja, em um conjunto de vértices de tamanho n, cada vértice tem que estar conectado com os outros *n-1* vértices através de uma aresta. Um grafo completo é representado por K_n, onde n é o número de vértices.

2.1.6 Isomorfismo de Grafos

O problema de isomorfismo entre grafos pode ser exemplificado da seguinte forma. É dito para pessoas desenharem um diagrama. Nenhuma delas pode ver o desenho da outra. O diagrama deve possuir 5 círculos *a, b, c, d, e*, e 4 conexões *(a, b), (b, c), (c, d), (d, e)*. Os diagramas (ou grafos) produzidos pelas duas pessoas podem ter aparências diferentes, mas elas definem o mesmo grafo, implicando que os dois grafos são isomorfos.

Grafos G_1 e G_2 são isomorfos se existe uma função f que mapeie cada um dos vértices de G_1 em G_2, e uma função g que mapeie cada uma das arestas de G_1 em G_2. Portanto, só haverá uma aresta e que incide em *(v, w)* em G_1 se e somente se a aresta $g(e)$ incide nos vértices $f(v)$ e $f(w)$ de G_2. O par de funções f e g é chamado de isomorfismo de G_1 em G_2.

Um isomorfismo para os grafos G_1 e G_2 acima, é definido por:

$$f(a) = A, \quad f(b) = B, \quad f(c) = C, \quad f(d) = D, \quad f(e) = E$$

$$g(x_i) = y_i, i = 1, \ldots, 5.$$

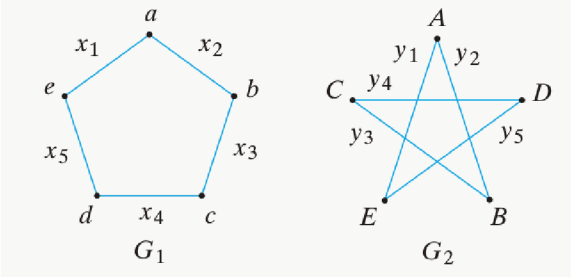

Figura 1 – Exemplo de isomorfismo entre dois grafos

[Johnsonbaugh 1986]

Remetendo a representação de grafos através de matrizes, os dois grafos G_1 e G_2 são isomorfos se e somente se há um ordenamento dos vértices em que suas matrizes de adjacência sejam iguais.

Como no exemplo da figura anterior, ambos os grafos terão a seguinte matriz de adjacência, respeitando a ordem alfabética do rótulo dos vértices no posicionamento das linhas e das colunas, evidenciando que eles são isomorfos.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Matriz de adjacência de G_1 e G_2 [Nievergelt 2002]

Se definirmos uma relação R em um conjunto de grafos pela regra G_1RG_2 e se G_1 e G_2 são isomorfos, R é uma relação de equivalência, onde cada classe de equivalência é constituída de um conjunto de grafos isomorfos entre si.

2.1.7 Determinando se Dois Grafos são Isomorfos

Determinar se dois grafos são isomorfos pode se tornar uma tarefa complicada, uma vez que não existe algoritmo conhecido para isomorfismo de grafos que seja de tempo polinomial. A ordem de complexidade do problema do isomorfismo de grafos é até então desconhecida. Por convenção, acredita-se que o problema resida entre as classes de complexidade P e NP-completo caso P $\neq$ NP [Skiena 2008]. O algoritmo básico faz uso de *backtrack* por todas as $n!$ permutações possíveis de vértices de G_1, e verifica se alguma dessas permutações é equivalente a G_2.

Muitas vezes, para evitar um gasto computacional desnecessário verificando se um par de grafos é isomorfo, são realizados alguns testes para não isomorfismo, com uma ordem

de complexidade inferior aos algoritmos clássicos. Caso esses testes consigam verificar que o par de grafos não é isomorfo, evita-se um gasto de recurso computacional desnecessário. Portanto, para realizar essa etapa de verificação, e demonstrar que dois grafos G_1 e G_2 *não são isomorfos* basta encontrar uma propriedade de G_1 que não exista em G_2, que ambos teriam em comum caso fossem realmente isomorfos.

Como dois grafos, para serem isomorfos, precisam ter funções que mapeiam os vértices e arestas de um grafo em outro, duas propriedades que precisam ser verificadas quando se busca determinar o *não isomorfismo* são o número de vértices e o número de arestas. Caso os grafos G_1 e G_2 tenham uma quantidade diferente de vértices ou arestas, pode-se descartar o isomorfismo entre eles.

Mesmo que a quantidade de vértices e arestas de G_1 e G_2 sejam iguais, para que esses dois grafos sejam considerados *não isomorfos* basta que o grau de algum vértice seja diferente. Para verificar isso, é utilizado a função f de mapeamento dos vértices, onde, dado um vértice v de grau k no grafo G_1, seu respectivo vértice $f(v)$ em G_2 também tem que ter grau k. Caso exista um vértice em G_1 com um grau x, e nenhum vértice em G_2 possua grau x, então G_1 e G_2 *não são isomorfos*.

Outra propriedade que deve ser testada para descartar o isomorfismo entre dois grafos, é verificar, caso haja ciclos nos grafos, se o número de ciclos, e comprimento dos ciclos é o mesmo. Caso exista em G_1 um ciclo com k arestas, e não exista em G_2 um ciclo com k arestas, então G_1 e G_2 *não são isomorfos*.

Nessa seção foram abordados conceitos referentes a grafos, com foco no problema de isomorfismo entre grafos, e as formas de resolver tal problema. O objetivo então é tentar tratar o problema de isomorfismo utilizando formas normais primárias, e para isso é necessário compreender os conceitos fundamentais sobre lógica proposicional e transformação dual que serão tratados na seção a seguir.

2.2 Formas Normais Primárias

A seção a seguir busca definir conceitos básicos sobre lógica proposicional, focando em satisfação booleana, formas normais conjuntivas e disjuntivas, um algoritmo para conversão de fórmulas lógicas em formas normais conjuntivas e disjuntivas, implicantes primários e implicados primários. A base para essa seção serão os livros [Nievergelt 2002], [Johnsonbaugh 1986], [Quine 1959] e [Russell e Norvig 2016].

2.2.1 Lógica Proposicional

Lógica pode ser expressa como o uso do raciocínio correto nas relações entre sentenças, onde não há como saber se uma sentença é verdadeira ou não utilizando a

lógica, mas sim se a relação entre determinadas sentenças é válida. Como por exemplo:

Sempre que chove, faz frio.

Quando faz frio, pessoas usam casacos.

Sempre que chove, pessoas usam casacos.

No exemplo, a lógica não é usada pra saber se as duas primeiras sentenças são verdadeiras, mas é utilizando a lógica que podemos deduzir que a terceira sentença é verdadeira, caso as duas primeiras sejam verdadeiras.

Na lógica proposicional as sentenças permitidas são sentenças atômicas, ou seja, elementos sintáticos indivisíveis, chamadas de literais, onde a sentença consiste de um único símbolo proposicional. Esse símbolo proposicional nada mais é que um rótulo arbitrário para uma proposição que pode ser verdadeira ou falsa. No exemplo da seção anterior podemos representar a proposição *sempre que chove* com o símbolo P e *faz frio* com o símbolo Q. Com essas senteças atômicas é possível criar sentenças lógicas mais complexas, com o uso de conectivos lógicos. Existem 5 conectivos de uso comum que serão descritos a seguir.

$\neg$ (não): É a negação de uma sentença, ou seja, sempre que P for verdadeiro, $\neg P$ será falso, e vice-versa.

$\wedge$ (e): Uma sentença na forma de $P \wedge Q$ é chamada de *conjunção*. Quando P for verdadeiro, e Q for verdadeiro, $P \wedge Q$ será verdadeiro, caso algum dos literais seja falso, a sentença será falsa.

$\vee$ (ou): Uma sentença na forma de $P \vee Q$ é chamada de *disjunção*. Para a sentença de uma disjunção ser considerada verdadeira, basta que um de seus literais seja verdadeiro. Se ambos os literais forem falsos, a sentença será falsa.

$\rightarrow$ (implica): Uma sentença na forma $P \rightarrow Q$ é chamada de implicação, onde P é a premissa, e Q é a consequência. Em uma implicação lógica, a única forma da sentença ser falsa é se a premissa for verdadeira, e a conclusão for falsa. Caso a premissa seja falsa, a sentença é considerada verdadeira, pois partindo de uma premissa falsa pode-se assumir qualquer coisa.

$\leftrightarrow$ (se e somente se): Chamada de bicondicional, uma sentença na forma $P \leftrightarrow Q$ só será verdadeira se P e Q forem ambos verdadeiros, ou ambos falsos. Caso um seja verdadeiro e o outro falso, a senteça é falsa.

Há também em uma sentença lógica, assim como na álgebra, a precedência de operadores, onde a ordem de precedência é, da mais alta para a mais baixa, $\neg$, $\wedge$, $\vee$, $\rightarrow$ e $\leftrightarrow$.

Seguindo a ordem de precedencia, a sentença $\neg P \vee Q \wedge R \rightarrow S$ é equivalente a

$((\neg P) \vee (Q \wedge R)) \rightarrow S$.

Uma forma de representar os possíveis valores-verdade para cada uma das combinações de valores dos literais para uma dada proposição, é através da tabela verdade, onde é usado F para falso e V para verdadeiro. Um exemplo de tabela verdade:

P	Q	$P \wedge Q$
V	V	V
V	F	F
F	V	F
F	F	F

Sabendo que computadores possuem uma certa facilidade inerente para simular conceitos de verdade, é possível também simular esses valores de verdade da tabela utilizando álgebra, onde o valor verdadeiro seria representado por 1, e o falso por 0, e os operadores $\wedge$ representado por * e $\vee$ representado por $+$, mantendo as características de cada operação. Para ilustrar esse comportamento, as tabelas verdade dos operadores $\wedge$ e $\vee$ são mostradas a seguir em fórmulas algébricas.

$$0 + 0 = 0, \qquad 0 * 0 = 0,$$
$$0 + 1 = 1, \qquad 0 * 1 = 0,$$
$$1 + 0 = 1, \qquad 1 * 0 = 0,$$
$$1 + 1 = 1, \qquad 1 * 1 = 1,$$

2.2.2 Satisfação Booleana

Para uma sentença ser considerada satisfatível, ela precisa ser verdadeira para pelo menos uma combinação de valores de seus literais, por exemplo, a sentença $p_1 \wedge p_2 \wedge p_3$ é satisfatível, pois quando p_1, p_2 e p_3 são verdadeiros, a sentença é verdadeira. Já a sentença $p_1 \wedge \neg p_1$ não possui um valor para p_1 que torne a sentença verdadeira, nesse caso, ela é chamada de contradição. Determinar se uma fórmula lógica proposicional é satisfatível foi o primeiro problema a ser provado NP-completo [Cook 1971].

2.2.3 Equivalência Lógica

Em lógica, duas sentenças P e Q são consideradas equivalentes se são verdadeiras nas mesmas combinações de valores de seus literais, ou seja, se em P um p_1 verdadeiro e um p_2 falso resultam em *verdadeiro*, em Q, com p_1 e p_2 assumindo os mesmos valores também deve resultar em verdadeiro. É possível verificar se duas sentenças são equivalentes comparando suas tabelas verdade.

Algumas equivalências lógicas comuns são listadas abaixo, considerando P, Q e R sentenças arbitrárias. [Russell e Norvig 2016]

$(P \wedge Q) \equiv (Q \wedge P)$, comutatividade de $\wedge$

$(P \vee Q) \equiv (Q \vee P)$, comutatividade de $\vee$

$((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$, associatividade de $\wedge$

$((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$, associatividade de $\wedge$

$\neg(\neg P) \equiv P$, eliminação de dupla negação

$(P \rightarrow Q) \equiv (P \wedge (\neg Q \rightarrow \neg P))$, contraposição

$(P \rightarrow Q) \equiv (\neg P \vee Q)$, eliminação da implicação

$(P \leftrightarrow Q) \equiv ((P \rightarrow Q) \wedge (Q \rightarrow P))$, eliminação de bicondicional

$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$, De Morgan

$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$, De Morgan

$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$, distributividade de $\wedge$ sobre $\vee$

$(P \vee (Q \wedge R)) \equiv ((P \vee R) \wedge (P \vee R))$, distributividade de $\vee$ sobre $\wedge$

2.2.4 Formas Normais Canônicas

As formas normais canônicas são sentenças formadas apenas por conjunções de disjunções ou disjunções de conjunções, podendo ser classificadas em *Forma Normal Conjuntiva* (*FNC*) e *Forma Normal Disjuntiva* (*FND*).

A FNC é uma sentença expressa como conjunções ($\wedge$) de fórmulas menores denominadas cláusulas, compostas apenas por disjunções ($\vee$). Um exemplo de FNC é $(p_1 \vee p_2 \vee p_3) \wedge (p_4 \vee p_5)$. Há também uma família restrita de sentenças k-FNC, onde uma sentença k-FNC possui k literais por cláusula, onde a fórmula seria expressa por $(p_{1,1} \vee \ldots \vee p_{1,k}) \wedge \ldots \wedge (p_{n,1} \vee \ldots \vee p_{n,k})$, onde $p_{i,k}$ é um literal.

Já a FND é uma sentença expressa como disjunções ($\vee$) de cláusulas compostas apenas por conjunções ($\wedge$). Um exemplo de FND é $(p_1 \wedge p_2) \vee (p_3 \wedge p_4 \wedge p_5)$. Assim como as FNCs, existe uma família restrita de sentenças k-FND, onde cada sentença k-FND possui k literais por cláusula, onde a fórmula seria expressa por $(p_{1,1} \wedge \ldots \wedge p_{1,k}) \vee \ldots \vee (p_{n,1} \wedge \ldots \wedge p_{n,k})$, onde $p_{i,k}$ é um literal.

Uma senteça composta apenas por literais e conjunções ou disjunções de literais é chamada de fórmula fundamental, onde é garantido que um literal não se repete nessa fórmula fundamental. As formas normais são compostas por uma combinação de cláusulas, que nada mais são do que fórmulas fundamentais, que no contexto de uma forma normal recebem o nome de cláusula.

Após conhecer as formas normais canônicas e as equivalencias lógicas descritas anteriormente, *é possível concluir que toda sentença da lógica proposicional é lógicamente equivalente a uma conjunção de disjunções de literais* [Russell e Norvig 2016]. Dito isso, será descrito um procedimento simples para converter qualquer sentença em uma FNC.

As etapas do algoritmo de conversão consistem em eliminar toda operação *bicondicional*, substituindo $p_i \leftrightarrow p_j$ por $(p_i \rightarrow p_j) \wedge (p_j \rightarrow p_i)$, em seguida deve-se eliminar toda operação de *implicação*, substituindo $p_i \rightarrow p_j$ por $\neg p_i \vee p_j$, feito isso, é necessário que a sentença apareça somente em literais, e para isso é aplicado repetidamente as equivalências $\neg(\neg p_i) \equiv p_i$; $\neg(p_i \vee p_j) \equiv (\neg p_i \wedge \neg p_j)$; e $\neg(p_i \wedge p_j) \equiv (\neg p_i \vee \neg p_j)$. Por fim, têm-se uma sentença com operadores $\wedge$ e $\vee$ aninhados, aplicados a literais, e esse último passo de conversão consiste de aplicar a lei de distributividade para eliminar esse aninhamento, distribuindo $\wedge$ sobre $\vee$ sempre que possível.

Um exemplo do funcionamento deste algoritmo, aplicado a fórmula $p_1 \leftrightarrow (p_2 \vee p_3)$:

1. $p_1 \leftrightarrow (p_2 \vee p_3)$
2. $(p_1 \rightarrow (p_2 \vee p_3)) \wedge ((p_2 \vee p_3) \rightarrow p_1)$, eliminação do bicondicional.
3. $(\neg p_1 \vee p_2 \vee p_3) \wedge (\neg(p_2 \vee p_3) \vee p_1)$, eliminação da implicação.
4. $(\neg p_1 \vee p_2 \vee p_3) \wedge ((\neg p_2 \wedge \neg p_3) \vee p_1)$, De Morgan.
5. $(\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_3 \vee p_1)$, distributividade de $\vee$ sobre $\wedge$.

Analogamente, é possível converter qualquer sentença em uma FND utilizando o algoritmo anterior com uma modificação, que consiste em inverter a distributividade, ou seja, onde antes havia distributividade de $\vee$ sobre $\wedge$, para a FND é necessário fazer a distributividade de $\wedge$ sobre $\vee$.

2.2.5 Implicantes Primários

Essa seção se baseia em [Quine 1959] e tem como objetivo estudar uma forma de reduzir sentenças lógicas arbitrárias em uma forma normal primária equivalente que seja de tamanho reduzido.

Fórmulas Fundamentais são fórmulas em lógica proposicional, compostas apenas por conjunções de literais, onde esses literais só podem aparecer uma vez na sentença, ou seja, como a fórmula fundamental é uma sequência de $\wedge$ (Operador *e* lógico), se o mesmo literal aparecer mais de uma vez na fórmula fundamental, ela será redundante. Se aparecer um determinado literal na fórmula fundamental, e o mesmo literal negado, será uma contradição.

Como foi visto ateriormente em Formas Normais Canônicas, as formas normais são formadas por conjunções de disjunções, ou disjunções de conjunções. Sabendo isso, podemos

definir que a Forma Normal Disjuntiva é uma disjunção de Fórmulas Fundamentais, ou seja, as Fórmulas Fundamentais são as cláusulas da Forma Normal Disjuntiva.

Sendo P uma sentença lógica, um Implicante Primário de P é uma Fórmula Fundamental que implica P, e que caso remova um de seus literais, não mais implica P, ou seja, a Fórmula Fundamental necessita estar reduzida, e para isso, sempre que a remoção de um literal da Fórmula Fundamental não influenciar na implicação de P, esse literal é removido, e quando isso ocorre, a Fórmula Fundamental é considerada uniliteralmente redundante. O conjunto de Implicantes Primários de uma sentença P será chamado PI_P.

Considerando como exemplo, que os literais de P sejam p_1, p_2, p_3, p_4, se $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ resulta em verdadeiro, e que $p_1 \wedge p_2 \wedge p_3 \wedge p_4 \rightarrow P$, então $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ é uma Fórmula Fundamental, mas se essa sentença for equivalente a, por exemplo, $p_1 \vee p_3 \rightarrow P$, então $p_1 \vee p_3$ é um Implicante Primário de P.

Como um implicante primário é uma Fórmula Fundamental mínima que implica P, uma sentença que seja composta por disjunções de implicantes primários, será uma Forma Normal mínima equivalente a P.

O número de Implicantes Primários que uma sentença tem é finito, uma vez que o número de literais de uma sentença é finito, o número de todas as combinações possíveis de literais também é finito.

Para uma sentença lógica P ser comprimida em uma forma normal equivalente, é necessário transformar a sentença em uma disjunção de *implicantes primários*, e depois remover o maior número possível de cláusulas supérfluas. Para a remoção de cláusulas supérfluas, se tivermos duas cláusulas C_1 e C_2, que são implicantes primários de P, e elas tiverem apenas um literal contraditório, por exemplo, se C_1 possui um literal p_1, e C_2 possui um literal $\neg p_1$, então exclui-se o literal p_1 a partir da conjunção $C_1 \wedge C_2$, resultando em uma única cláusula, contendo todos os literais de C_1 e C_2 exceto a contradição, e excluí-se as cláusulas C_1 e C_2. Essa operação é chamada de Resolução

Por exemplo, se C_1: $(p_1 \wedge p_2 \wedge \neg p_3 \wedge p_4)$ é implicante primário de P, e C_2: $(p_1 \wedge p_2 \wedge p_3 \wedge p_4)$ também é implicante primário de P, fazendo a conjunção de C_1 e C_2, remove-se o literal p_3, e resulta em uma nova clausula C_3 composta pelos literais de C_1 e C_2, onde o novo implicante primário seria C_3: $p_1 \wedge p_2 \wedge p_3$ que implica P. Se todos os literais de um implicante primário C_1 estão em outro implicante primário C_2, C_1 subsume C_2. Tomando como exemplo C_1 sendo $p_1 \wedge p_2$ e C_2 sendo $p_1 \wedge p_2 \wedge p_3 \wedge p_4$, como todos os literais de C_1 então contidos em C_2, C_1 subsume C_2, portanto, C_2 pode ser excluído.

2.2.6 Transformação Dual

Algoritmos para resolver o problema da satisfação booleana (SAT) como o algoritmo de Davis-Putnam-Logemann-Loveland (DPLL) [Davis e Putnam 1960], geralmente utilizam

uma abordagem semântica, de forma que é possível atribuir valor de verdade aos símbolos proposicionais. Por outro lado, neste trabalho será utilizado um método baseado em uma abordagem sintática, explorando propriedades das formas normais de um conjunto de cláusulas proposicionais, que será chamado de teoria proposicional, ou seja, esta teoria proposicional é representada por formas normais conjuntivas (FNC) ou disjuntivas (FND).

Dada uma teoria proposicional P, P pode ser transformada em uma FNC P_c ou em uma FND P_d, P_c, por sua vez, pode ser transformada em P_d, e vice-versa, utilizando distributividade dos operadores lógicos $\wedge$ e $\vee$.

O algoritmo que será utilizado calcula a representação mínima de uma teoria P_d. A representação mínima de P_d é a P_c gerada a partir de P_d. Essa representação mínima é o conjunto de cláusulas não contraditórias, e que não podem ser subsumidas. Como visto anteriormente, essas cláusulas são os implicantes primários da teoria P.

Cada cláusula mínima representa mínimamente o conjunto de valores verdade atribuídos aos símbolos da teoria P, para que P seja satisfatível. Gerando todas as cláusulas duais mínimas, obtêm-se todas as combinações de valores verdade para os símbolos da teoria P que tornem ela satisfatível. A ideia principal é que a quantidade de cláusulas duais mínimas geradas seja sempre menor ou igual a quantidade de literais.

2.2.6.1 Representação da Teoria

Considerando tudo o que foi abordado anteriormente, outro elemento precisa ser inserido para que o algoritmo funcione. O elemento fundamental do algoritmo é o conceito de *quantum*, definido como um par (p, F), onde p é um literal, e $F \subseteq P_c$ é o conjunto de coordenadas que representam a posição de cada cláusula em que p pertence. Para exemplificar, em uma teoria: $(p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee \neg p_3)$, as cláusulas seriam $(p_1 \vee p_2 \vee p_3)$, $(\neg p_1 \vee p_2 \vee p_3)$ e $(p_1 \vee p_2 \vee \neg p_3)$, e o conjunto de coordenadas para representar essas cláusulas seria $P_c = \{0, 1, 2\}$, sendo os números contidos nos conjuntos a posição da cláusula, onde 0 representa $(p_1 \vee p_2 \vee p_3)$, 1 representa $(\neg p_1 \vee p_2 \vee p_3)$ e 2 representa $(p_1 \vee p_2 \vee \neg p_3)$. A forma utilizada para representar as coordenadas de um literal é p^F, sendo p o literal, e F o conjunto de coordenadas. Portanto, as coordenadas são representadas por $p_1^F = \{0, 2\}$, $\neg p_1^F = \{1\}$, $p_2^F = \{0, 1, 2\}$, $p_3^F = \{0,1\}$ e $\neg p_3^F = \{2\}$.

Para gerar uma DNF P_d mínima, novas cláusulas são formadas de modo que, a união das coordenadas dos *quanta* dos literais tem que cobrir todas as cláusulas de P_c, ou seja, seguindo o exemplo anterior, para uma nova cláusula ser formada, a união de *quanta* dos literais tem que resultar no conjunto $\{0, 1, 2\}$. Como o quantum de p_1 é $\{0, 2\}$, e o de p_3 é $\{0, 1\}$, é possível criar a cláusula C_0: $(p_1 \wedge \mathrm{p}_3)$ da teoria P_d.

Cada literal da nova cláusula deve também representar pelo menos uma cláusula de P_c, ou seja, dado um literal p, em uma nova cláusula gerada para P_d, ele deve estar

contido em uma cláusula de P_c sem que os outros literais da nova cláusula gerada estejam nessa mesma cláusula de P_c. Aplicando no exemplo anterior, onde a nova cláusula $(p_1 \wedge p_3)$ é criada, p_1 está contido em $(p_1 \vee p_2 \vee \neg p_3)$, onde p_3 não está, e p_3 está contido em $(\neg p_1 \vee p_2 \vee p_3)$, onde p_1 não está.

Caso o literal adicionado na nova cláusula não represente sozinho nenhuma cláusula de P_c ele é redundante, portanto, deve ser eliminado.

2.2.7 Algoritmo de Transformação Dual

O algoritmo para encontrar as novas cláusulas da FND P_d é uma busca no espaço de estados, sendo aplicado uma busca A* para encontrar tais estados. Como um algoritmo de busca, temos o estado inicial e o estado final que queremos alcançar, que no caso é uma cláusula que cubra todo o conjunto de coordenadas de P_c, então primeiro a lista de literais é ordenada de modo que os literais com mais coordenadas sejam escolhidos primeiro, de modo a otimizar a busca, já que cobrem uma quantidade maior de cláusulas. Depois disso é criado um conjunto chamado *Gap*, onde ficam guardadas todas as coordenadas que não foram cobertas ainda, e para cada literal candidato a entrar na cláusula conforme é realizada a busca, se em seu *quantum* houver uma ou mais coordenadas que estejam no *Gap*, ou seja, coordenadas que ainda não foram cobertas, esse literal é adicionado à cláusula e seu *quantum* é removido do *Gap*. Um estado é considerado válido se em algum momento o conjunto *Gap* for vazio, e então é adicionado em um conjunto de estados válidos, que representarão as cláusulas de P_d [Bittencourt, Marchi e Padilha 2003].

Algoritmo:

0. Inicializa a lista de estados: states $\leftarrow \emptyset$;

1. Ordena a lista de literais conforme o tamanho do conjunto de coordenadas;

2. $\forall p_i$ em literais:

   Inicializa e popula o conjunto Gap com as coordenadas não cobertas por p_i;

   Inicializa um novo estado ω com p_i;

   $\forall p_j^F$ em literais:

   se $p_j^F{}_k$ está em Gap:

   novo estado $\leftarrow p_j$;

   remove p_j^F de Gap;

   se Gap $= \emptyset$ e states não contém ω:

   states $\leftarrow \omega$

3. retorna states

(2.1)

Exemplo 1: Usando como exemplo a seguinte teoria a Forma Normal Conjuntiva com 4 símbolos proposicionais, e 4 cláusulas:

Observação: Para facilidade na representação, os operadores lógicos serão omitidos, sendo apenas listados os literais que compõe a cláusula, separados por vírgula.

$C_0 : [\neg p_2, \neg p_4]$

$C_1 : [\neg p_3, p_4]$

$C_2 : [\neg p_2, \neg p_3]$

$C_3 : [\neg p_1]$

O conjunto de coordenadas dos literais será:

$\neg p_1^F = \{3\}$;

$\neg p_2^F = \{0, 2\}$;

$\neg p_3^F = \{1, 2\}$;

$p_4^F = \{1\}$;

$\neg p_4^F = \{0\}$;

A lista de literais é então ordenada de acordo com o critério de ordenação definido, sendo escolhido primeiro os literais com maior conjunto de coordenadas. O resultado é a lista de literais na seguinte ordem: $\neg p_2^{\{0,2\}}$, $\neg p_3^{\{1,2\}}$, $\neg p_1^{\{3\}}$, $p_4^{\{1\}}$, $\neg p_4^{\{0\}}$.

Escolhendo $\neg p_2$ como primeiro literal, já que ele cobre o maior número de cláusulas, $\neg p_2$ é inserido no novo estado ω_0, o conjunto *Gap* então é populado com o que não é coberto por $\neg p_2$, portanto $Gap = \{1, 3\}$.

Iterando a lista de literais a fim de cobrir o *Gap*, o próximo literal da lista que possui uma coordenada no seu conjunto p^F que está contida em *Gap* é $\neg p_3^{\{1,2\}}$. O literal $\neg p_3$ é então inserido no estado ω_0 e o conjunto $\{1, 2\}$ é então subtraído de *Gap*, deixando $Gap = \{3\}$;

Continuando a iteração, o próximo literal a ser verificado é $\neg p_1^{\{3\}}$, cujo conjunto de coordenadas está contido em *Gap*. O literal $\neg p_3$ é inserido em ω_0 e o conjunto $\{3\}$ é subtraído de *Gap*, resultando em $Gap = \{\emptyset\}$. Como *Gap* agora é um conjunto vazio, $\omega_0 = [\neg p_2, \neg p_3, \neg p_1]$ é considerado um estado válido e é inserido na lista de estados.

Seguindo o mesmo procedimento para os outros ramos da árvore de estados possíveis, escolhendo outros literais para compor os estados, encontramos ainda os seguintes estados

válidos: $\omega_1 = [\neg p_2, \neg p_1, p_4]$ e $\omega_2 = [\neg p_3, \neg p_1, \neg p_4]$.

Interpretando os estados como cláusulas, onde $\omega_i = C_i$, temos então uma nova teoria proposicional na Forma Normal Disjuntiva:

$C_0 : [\neg p_2, \neg p_3, \neg p_1]$

$C_1 : [\neg p_2, \neg p_1, p_4]$

$C_2 : [\neg p_3, \neg p_1, \neg p_4]$

2.2.8 Propriedades das Formas Normais Primárias

Formas Normais Primárias são os pares de implicantes e implicados primários de uma teoria proposicional. Uma característica importante dessa representação é que, dado uma teoria proposicional, esta teoria gerará apenas um único par de implicantes e implicados primários, entretanto, este par de implicantes e implicados primários representa uma família de teorias proposicionais congruentes, que são equivalentes em relação ao grupo de permutações e complementações. [Bittencourt 2008].

Famílias de teorias proposicionais podem ser classificadas de acordo com o número de símbolos proposicionais e pela sua quantidade de modelos proposicionais. Cada família possui ainda uma população, sendo a população a quantidade de teorias proposicionais congruentes dentro da família [Bittencourt 2007] [Polya 1940].

2.2.9 Contagem de Modelos

Modelos proposicionais são combinações de valores verdade em uma fórmula proposicional que resulte em *Verdadeiro* para a fórmula, e a contagem de modelos nada mais é do que a quantidade dessas combinações de valores que resultem em *Verdadeiro* para uma fórmula proposicional dada. [Gomes, Sabharwal e Selman 2006]

Um algoritmo bem comum para resolver o problema de contagem de modelos é o algoritmo de Davis-Putnam-Logemann-Loveland [Davis e Putnam 1960], porém, outro algoritmo será utilizado, uma vez que ele calcula o número de modelos de uma teoria proposicional baseado nos implicantes e implicados primários daquela teoria, podendo se mostrar mais eficiente em algumas classes de teorias. Portanto, a ideia do algoritmo é calcular os implicantes e implicados primários de uma fórmula de entrada, para então determinar o número de modelos. [Bittencourt 2007]

O algoritmo recebe uma *FNC* de entrada, que representa uma fórmula P, e então calcula os implicantes primários de P (IP_P), utilizando o algoritmo descrito em Algoritmo de Transformação Dual.

Utilizando a teoria proposicional do Exemplo 1 para ilustrar o método de contagem de modelos, com a aplicação do Algoritmo de Transformação Dual já temos IP_P. Sendo

IP_P o conjunto de cláusulas resultante do algoritmo, então IP_P:

$C_0 : [\neg p_2, \neg p_3, \neg p_1]$

$C_1 : [\neg p_2, \neg p_1, p_4]$

$C_2 : [\neg p_3, \neg p_1, \neg p_4]$

O próximo passo é determinar as coordenadas disjuntivas dos literais em IP_P, onde essas coordenadas são apenas a representação indicando a quais termos de IP_P cada literal pertence.

No *Exemplo 1* as coordenadas disjuntivas de cada literal são : $\neg p_1^{\{0,1,2\}}$, $\neg p_2^{\{1,2\}}$, $\neg p_3^{\{0,2\}}$, $p_4^{\{1\}}$, $\neg p_4^{\{0\}}$

O terceiro passo é encontrar o conjunto de termos compatíveis em IP_P, onde dois termos são compatíveis se eles não possuem os mesmos literais com sinais opostos. Esse conjunto pode ser determinado de forma eficiente utilizando as coordenadas disjuntivas.

No *Exemplo 1*, C_0 forma termos compatíveis com C_1 e C_2, mas C_1 e C_2 não formam termos compatíveis, uma vez que C_1 possui o literal p_4 e C_2 possui o literal $\neg p_4$, causando uma contradição.

O quarto passo é uma busca no espaço de estados, onde cada estado é representado por um termo de IP_P, e/ou pela combinação de dois ou mais termos compatíveis em IP_P. Os estados iniciais são os termos em IP_P, e para calcular os sucessores basta determinar os termos compatíveis, usando os conjuntos obtidos no passo 3. Também é necessário assegurar que um termo não esteja contido na união de outros termos. Os estados então são modelados em um grafo, onde os termos são ligados às combinações que eles são compatíveis.

Os estados criados a partir do *Exemplo 1* são as próprias cláusulas, e os termos compatíveis C_0,C_1 e C_0,C_2, formando um grafo de estados ilustrado abaixo.

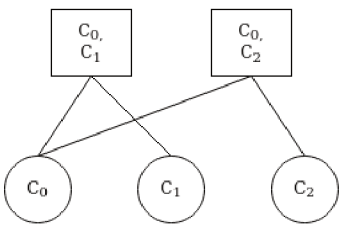

Figura 2 – Grafo representando a relação entre os estados gerados do Exemplo 1

No quinto passo, o número de modelos associado a cada estado é calculado. Se a fórmula possui n símbolos proposicionais, e os termos associados ao estado possui k literais, então o número de modelos associados aquele estado será 2^{n-k}.

Como no *Exemplo 1* a quantidade de símbolos proposicionais da teoria é 4, os conjuntos de literais dos estados e a quantidade de modelos associados a cada estado são:

$C_0, C_1 = \{\neg p_1, \neg p_2, \neg p_3, \neg p_4\}$, como o tamanho de C_0, C_1 é 4, a quantidade de modelos associados será $2^{4-4} \equiv 2^0 \equiv 1$.

$C_0, C_2 = \{\neg p_1, \neg p_2, \neg p_3, \neg p_4\}$, como o tamanho de C_0, C_2 é 4, a quantidade de modelos associados será $2^{4-4} \equiv 2^0 \equiv 1$.

$C_0 = \{\neg p_1, \neg p_2, \neg p_3\}$, como o tamanho de C_0 é 3, a quantidade de modelos associados será $2^{4-3} \equiv 2^1 \equiv 2$.

$C_1 = \{\neg p_1, \neg p_2, p_4\}$, como o tamanho de C_1 é 3, a quantidade de modelos associados será $2^{4-3} \equiv 2^1 \equiv 2$.

$C_2 = \{\neg p_1, \neg p_3, \neg p_4\}$, como o tamanho de C_2 é 3, a quantidade de modelos associados será $2^{4-3} \equiv 2^1 \equiv 2$.

Com a quantidade de modelos associados a cada estado calculada, o grafo é atualizado, onde cada estado recebe um rótulo com a sua quantidade de modelos. A imagem abaixo ilustra o grafo atualizado.

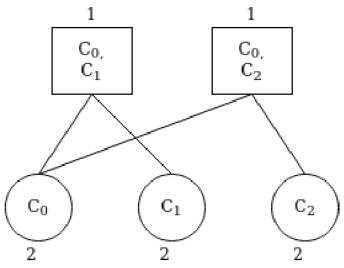

Figura 3 – Grafo com a quantidade de modelos associados a cada estado do Exemplo 1

O sexto passo consiste em calcular a quantidade de modelos que estão associados exclusivamente a cada estado. Para fazer isso, é subtraído da soma de modelos dos estados que representam um termo a quantidade de modelos dos outros estados formados por aquele termo. Por exemplo: Se existe uma cláusula C_0 com 2 modelos, e uma cláusula C_1, também com 2 modelos, e ela está associada a um estado que representa o par de cláusulas $C_{0,1}$: (C_0, C_1), e a quantidade de modelos desse par é 1, após o calculo a quantidade de modelos associada a C_0 será 1. Se houver também um par de cláusulas $C_{1,2}$: (C_1, C_2) e a quantidade de modelos desse par é 1, a quantidade de modelos associados a C_1 será 0, pois ele tinha 2 modelos, mas como estava associado a $C_{0,1}$ e $C_{1,2}$, a quantidade de modelos desses pares é deduzido de C_1.

Atualizando a quantidade de modelos de cada estado do *Exemplo 1* de acordo com o sexto passo, como C_0 está associado a C_0, C_1 e C_0, C_2, a quantidade de modelos associados exclusivamente a C_0 será seu número de modelos (2) subtraindo o número de

modelos de C_0, C_1 (1) e C_0, C_2 (1), resultando em 0 modelos associados a C_0. Seguindo a mesma regra, como C_1 só está associado a C_0, C_1 seu número de modelos será 1, e como C_2 só está associado a C_0, C_2 seu número de modelos também será 1.

Após atualizar a quantidade de modelos associadas a cada estado, teremos o seguinte grafo:

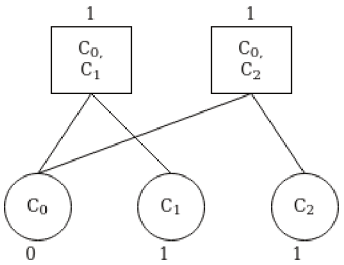

Figura 4 – Grafo com a quantidade de modelos associados exclusivamente a cada estado do Exemplo 1

O ultimo passo é somar o total de modelos de cada cláusula e dos estados formados pela combinação de cláusulas compatíveis, resultando na quantidade de modelos da teoria.

Após somar a quantidade de modelos associados a cada estado do *Exemplo 1*, verifica-se que a teoria proposicional possuía 4 modelos.

3 Método Para Detecção de Isomorfismo Entre Grafos

Neste capítulo será desenvolvido um método para detecção de isomorfismo entre grafos utilizando o que foi abordado no capítulo anterior, onde os grafos de entrada serão primeiro avaliados em um pré-processamento, após esse pré-processamento, se o isomorfismo não for descartado logo no início, os grafos passam por uma remodelagem, onde são transformados em fórmulas lógicas. As fórmulas lógicas formadas a partir dos grafos de entrada passam então pelo algoritmo de transformação dual, gerando seus respectivos conjuntos de implicantes primários, por fim, serão contados os modelos de cada um dos conjuntos de implicantes primários, e a partir dessa quantidade de modelos, será interpretado um resultado. Neste capítulo esse processo será descrito passo a passo.

3.1 Pré-processamento dos Grafos para Verificação de Isomorfismo

Certas características são necessárias para que haja isomorfismo entre os grafos, conforme já foi abordado no capítulo anterior na seção Determinando se Dois Grafos são Isomorfos. Considerando que ambos os grafos a serem avaliados devem ter algumas características em comum, um pré-processamento é realizado a fim de evitar processamento desnecessário.

Considerando que o método proposto receba como entrada dois arquivos, cada um contendo informações sobre um grafo, as etapas a seguir são realizadas como pré-processamento.

(1): Inicialmente, são verificados se ambos os grafos possuem o mesmo número de vértices. Caso os grafos não possuam o mesmo número de vértices eles são considerados não isomorfos.

Na entrada da implementação realizada, a informação da quantidade de vértices já é fornecida, portanto, nessa parte do pré-processamento é apenas verificado se os dois arquivos possuem o mesmo valor nesse campo.

Para exemplificar um caso de não-isomorfismo devido a uma quantidade diferente de vértices, temos dois grafos G_1 e G_2, conforme o diagrama apresentado na Figura 5, onde G_1 possui 5 vértices e G_2 possui 6 vértices, .

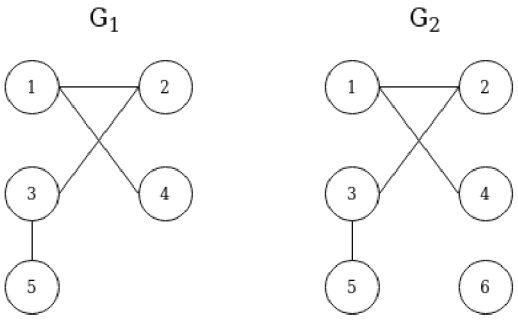

Figura 5 – Exemplo de dois grafos não-isomorfos com quantidades diferentes de vértices

(2): A próxima etapa de pré-processamento é verificar se ambos os grafos possuem o mesmo número de arestas. Caso os grafos não possuam o mesmo número de arestas, eles são considerados não-isomorfos.

Na entrada da implementação também é esperado a quantidade de arestas, portanto, nessa etapa do pré-processamento basta verificar se os dois arquivos de entrada possuem o mesmo valor nesse campo.

Para exemplificar um caso de não-isomorfismo devido a uma quantidade diferente de arestas, temos novamente dois grafos G_1 e G_2, representados na Figura 6, onde G_1 possui 5 vértices e 5 arestas, e G_2 possui 5 vértices e 4 arestas.

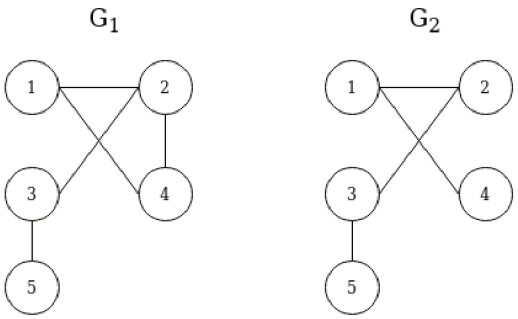

Figura 6 – Exemplo de dois grafos não-isomorfos com quantidades diferentes de arestas

(3): A terceira e última etapa de pré-processamento é verificar se os vértices possuem um mapeamento de grau, onde cada vértice v_i de um grafo G_1 deve possuir o mesmo grau de algum vértice v_j em G_2.

Para exemplificar um caso de não-isomorfismo devido a diferença de graus dos vértices, temos na Figura 7 dois grafos G_1 e G_2, onde ambos possuem 5 vértices e 5 arestas, porém, com vértices de graus diferentes. Em G_2 todos os vértices possuem grau 2, porém, em G_1 apenas os vértices 1, 3 e 4 possuem grau 2, o vertice 2 possui grau 3, e o vértice 5 possui grau 1.

O mapeamento dos graus dos vértices pode ser verificado utilizando os conjuntos

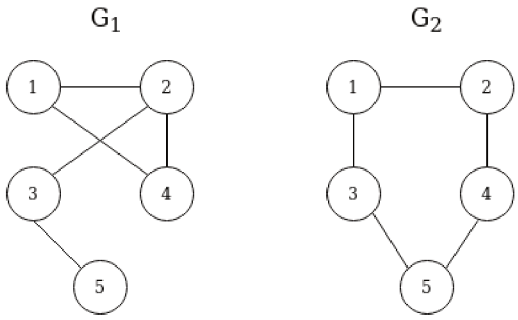

Figura 7 – Exemplo de dois grafos não-isomorfos com vértices de graus diferentes

de coordenadas dos literais das teorias proposicionais fornecidas na entrada do programa. O conjunto de coordenadas nada mais é do que o conjunto de arestas incidentes em um vértice. Para esta verificação basta ordenar a lista de literais de acordo com o tamanho do conjunto de coordenadas de cada literal, e depois verificar se o tamanho do conjunto de coordenadas de um literal, em uma posição da lista de literais de uma teoria, é igual ao tamanho do conjunto de coordenadas de um literal na mesma posição da lista de literais da outra teoria.

3.2 Modelagem dos Grafos

É necessário encontrar uma maneira de modelar os grafos na forma de teorias proposicionais, de modo que sua semântica, ou seu significado, não se altere. Neste trabalho, apenas grafos não-direcionados, e cujas arestas não possuam peso, serão tratados.

Primeiramente, foi proposto modelar as teorias lógicas de acordo com a matriz de adjacência do grafo, onde cada cláusula representaria uma linha da matriz, cada coluna representaria um literal, e os valores 0 e 1 da matriz de adjacência representariam o valor do literal, se ele é um literal puro ou negado.

Logo no início é possível notar que essa modelagem poderia não ser eficaz, já que uma posição da matriz de adjacência poderia ter significados ambíguos, já que tanto as linhas, quanto as colunas da matriz de adjacência representam vértices, e na modelagem proposta inicialmente, as linhas representariam cláusulas e as colunas representariam os literais. Desta forma, um vértice poderia ser representado em um momento como cláusula, e em outro momento como literal.

Para evitar significados ambíguos para os vértices, a modelagem adotada cria teorias proposicionais onde os vértices seriam os símbolos proposicionais, e as arestas seriam as cláusulas.

Porém, essa modelagem não considera literais negados, apenas literais puros, e isto implicará em algumas modificações nos algoritmos descritos no capítulo anterior, conforme

será visto nas seções referentes à implementação.

A modelagem adotada ainda possui um problema, que pode gerar resultados falsos, para isso algumas medidas serão tomadas, conforme descrito a seguir, a fim de corrigir esses possíveis problemas.

O principal problema encontrado na modelagem, ocorre por levar em consideração apenas as arestas como cláusulas, de modo que, se houverem dois grafos G_1 e G_2, como na Figura 5 deste capítulo, se G_1 e G_2 possuírem as mesmas arestas, mas G_2 não seja conexo, e possua vértices sem nenhuma ligação com outros vértices, a modelagem atual representaria ambos os grafos como uma mesma teoria.

Seguindo o exemplo da Figura 5, ambos os grafos são modelados na seguinte teoria proposicional:

$C_0 : [p_1, p_2]$

$C_1 : [p_1, p_4]$

$C_2 : [p_2, p_3]$

$C_3 : [p_3, p_5]$

Para resolver esse problema, é necessário informar na entrada a quantidade de vértices, que representariam os símbolos proposicionais da teoria, e guardar esse valor em uma variável do programa, para que casos como este possam ser identificados.

O número de vértices ainda é necessário para o cálculo da quantidade de modelos da teoria proposicional, conforme será descrito na seção Contagem de Modelos, de modo que a variável criada para armazenar a quantidade de vértices não seja apenas para corrigir uma possível falha na modelagem, mas também ser importante para o funcionamento do método em si.

3.3 Entrada de Dados

O programa gerado a partir do algoritmo proposto deve receber como entrada a quantidade de vértices, que também representará a quantidade de símbolos proposicionais, e a quantidade de arestas, que também representará a quantidade de cláusulas da teoria fornecida na entrada do programa. A entrada deve também fornecer o conjunto de cláusulas, ou arestas, sendo cada cláusula composta por um par de símbolos proposicionais. Caso a entrada fornecida seja em forma de pares de vértices que estejam conectados, é necessário fazer uma modificação no programa para gerar símbolos proposicionais representando os vértices, ou utilizar o vértice como símbolo proposicional guardando seu valor em uma string.

3.4 Implementação

Para implementação da metodologia descrita neste trabalho, a linguagem escolhida foi ANSI/ISO C++11. O programa será apresentado em forma de um diagrama de classes, porém, detalhes importantes da implementação serão apresentados com trechos do código fonte. A Figura 8 apresenta o modelo de classes. São quatro classes principais descritas a seguir.

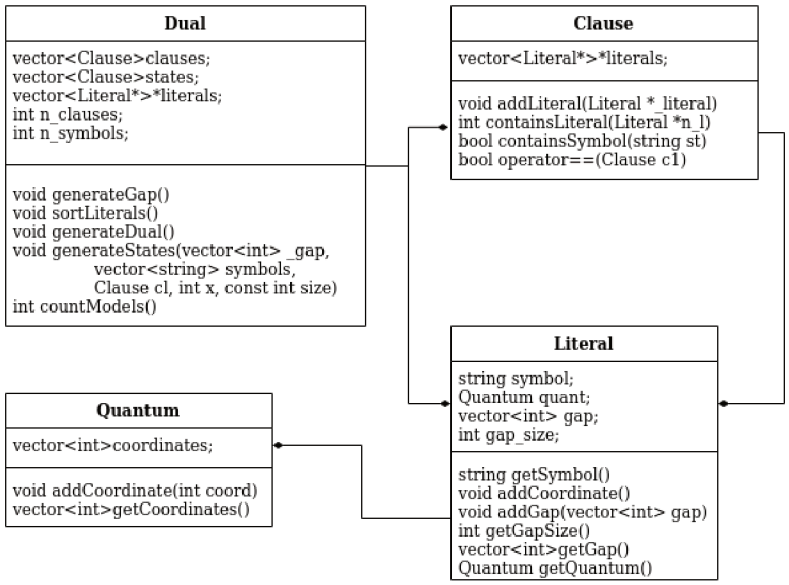

Figura 8 – Diagrama de classes da implementação

A classe *Quantum* é apenas uma abstração do conjunto de coordenadas de um literal, que poderia inclusive ser apenas um atributo da classe *Literal*, porém, por questões de organização do código, a modelagem escolhida visa separar as coordenadas em uma classe distinta.

A classe *Literal* possui informações como o rótulo do literal, ou seja, seu símbolo proposicional na teoria, um objeto da classe *Quantum*, que é o conjunto de coordenadas do literal, e a classe *Literal* também é responsável por guardar seu Gap, que mais tarde, no algoritmo de transformação dual, será utilizado. Guardar o gap no objeto Literal foi uma escolha no *design*, pois evitaria ficar recalculando o Gap inicial.

É na classe *Literal* que é necessário fazer a primeira alteração do código devido a forma como os grafos foram modelados em teorias proposicionais. Como as arestas foram modeladas como cláusulas, que ligam dois vértices modelados como literais, nunca existirão literais negados, portanto, não é necessário à classe *Literal* um atributo que identifique se um literal é ou não negado.

Os métodos da classe *Literal* são utilizados apenas para definir ou obter o valor de algum atributo de sua classe, ou da classe *Quantum* atribuída como variável.

A característica principal dos objetos da classe *Literal*, é que todos os literais da teoria proposicional foram modelados como ponteiros, sendo assim, os objetos são únicos, não podendo haver dois literais iguais na implementação. Essa característica da implementação foi escolhida de forma que, quando um literal seja modificado em uma cláusula, suas modificações sejam garantidas em todas as cláusulas que o literal pertence, sem correr risco de haver algum defeito no código devido a um atributo não atualizado em algum objeto. Outra vantagem de modelar objetos da classe *Literal* como ponteiros é a verificação de igualdade entre os objetos, pois, como são ponteiros, para serem o mesmo objeto basta estarem alocados no mesmo endereço de memória.

Conforme descrito anteriormente, a classe *Clause* possui um ponteiro para uma lista de ponteiros de objetos da classe *Literal*. Os métodos da classe *Clause*, exceto pelo método addLiteral(Literal *_literal), são majoritariamente para verificação de algumas características da cláusula. O método addLiteral, como seu nome sugere, é utilizado apenas para inserir um literal na cláusula, já os outros métodos são métodos de checagem utilizados no algoritmo de transformação dual. O método containsLiteral(Literal *n_l) verifica se o literal pertence à cláusula. Da mesma forma, o método containsSymbol(string sy) apenas verifica se o símbolo proposicional está presente na cláusula. O último método é uma sobrecarga do operador de igualdade (==), implementado para facilitar a verificação se duas cláusulas são iguais. A sobrecarga recebe como parâmetro um objeto da classe *Clause*, e verifica se o objeto da classe *Clause* contido no ponteiro *this* é igual ao objeto passado por parâmetro.

Ao contrário dos objetos da classe *Literal*, os objetos da classe *Clause* não foram modelados como ponteiros, devido a complexidade da classe, uma vez que para que literais sejam iguais, na forma como a implementação foi projetada, eles necessitam apenas do símbolo proposicional, enquanto um objeto da classe *Clause* nescessitaria de listas de literais iguais.

A classe *Dual*, por sua vez, possui uma lista de ponteiros de objetos da classe *Literal*. Essa lista possui todos os literais contidos na teoria proposicional. A classe *Dual* possui ainda, duas listas de objetos da classe *Clause*, a primeira lista, rotulada como clauses, contém todas as cláusulas da teoria proposicional fornecida na entrada do programa. Já a lista *states*, é uma lista vazia, que representa os estados válidos alcançados pelo algoritmo de transformação dual, sendo preenchida posteriormente, e que representará o conjunto de implicantes primários da teoria proposicional fornecida na entrada do programa.

Nas próximas seções os métodos da classe *Dual* serão abordados de forma mais aprofundada, incluindo detalhes da implementação e excertos do código fonte.

3.5 Transformação Dual

Essa seção aborda apenas a implementação da classe *Dual*, as demais classes estão descritas na seção anterior e são, em sua maioria, implementadas de forma a tornar o ambiente para implementar o algoritmo de transformação dual mais organizado e legível.

3.5.1 Armazenamento da Teoria

O primeiro processo na classe *Dual* é modelar a teoria proposicional dentro do programa. Para isto, foi implementado um método na classe *Dual* chamado readFile(), apresentado no *Listing 3.1*, que é responsável por tratar a entrada do programa. O método readFile() armazena primeiramente a quantidade de arestas e de vértices (linhas 6 e 7), e depois lê a sequência de arestas, modeladas como cláusulas proposicionais.

Conforme o arquivo é processado, para cada linha lida através do comando getline() do C++, os literais são separados e é feita uma verificação para cada literal, se ele já está presente na lista de literais ele é apenas inserido na cláusula, caso ele não exista ainda é criado um novo literal, e este novo literal é inserido na cláusula e na lista de literais. A posição da cláusula também é registrada, e inserida no conjunto de coordenadas dos literais lidos, no seu objeto da classe Quantum (linha 39).

Listing 3.1 – Leitura da teoria

```
1  void Dual::readFile() {
2
3    int n_c, n_s, i;
4    cin >> n_c;
5    cin >> n_s;
6    this->n_clauses = n_c; // number of clauses / edges
7    this->n_symbols = n_s; // number of symbols / vertex
8
9    for(i = 0; i < this->n_clauses; i++) {
10     size_t pos = 0; // var to separate the literals in line
11     getline(cin, aux);
12     if(aux.empty()) { // in case the line is blank
13       i--; // don't count as clause read
14       continue;
15     }
16     string token; // auxiliar var to keep the token of the literal
17     Clause new_clau; // New clause to be inserted in theory
18
19     while((pos = aux.find(delimiter)) != string::npos) { // ends when all
           literals were read
20       token = aux.substr(0, pos); // using token to keep the symbol of
             literal
21
```

```
22        Literal *n_lit; // possible new Literal
23
24        bool control = true; // flag to control if Literal n_lit already
              exists in our theory
25
26        for(vector<Literal*>::iterator it = this->literals->begin(); it !=
              this->literals->end(); ++it) {
27          if (token == (*it)->getSymbol()) { // if Literal already exists
28            n_lit = (*it); // set the n_lit to the literal found in list of
                  literals
29            control = false; // set flag to false, so we don't keep the same
                  literal twice in the list
30            break;
31          }
32        }
33
34        if(control) { // if control is true, creates new literal and add to
              theory's list of Literals
35          n_lit = new Literal(token, negate);
36          this->literals->push_back(n_lit);
37        }
38
39        n_lit->addCoordinate(i); // even if literal already exists, insert
              the position of the clause
40                    // in the coordinates of the literal
41        new_clau.addLiteral(n_lit); // add the literal in the clause
42
43        aux.erase(0, pos+delimiter.length()); // set the "pointer" to the
              next literal in the line read
44      }
45      this->clauses.push_back(new_clau); // add the new clause created to the
            list of clauses
46    }
47  }
```

3.5.2 Geração do Gap

Após construir a teoria proposicional no programa, o próximo passo é gerar o conjunto de Gap para cada literal. Nesta implementação apresentada no *Listing 3.2*, a abordagem foi, para cada literal da teoria, criar dois vetores de inteiros, um contendo a lista de coordenadas do literal, e o outro contendo a lista de todas as coordenadas da teoria (linhas 6 e 8). Após isso foi aplicada a operação de diferença de conjuntos, onde todas as coordenadas contidas nos dois conjuntos eram removidas, desta forma, restando apenas as coordenadas de Gap (linha 15). Este conjunto resultante então é estabelecido como Gap do literal.

Listing 3.2 – Geração dos conjuntos Gap dos literais da teoria

```
void Dual::generateGap() {

  // iterates through literals list, generating Gap for every literal in
     our theory
  for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
     literals->end(); ++it) {
    // auxiliar int vector to keep track of each coordinate of literal
    vector<int> aux1 = (*it)->getQuantum().getCoordinates();
    // auxiliar int vector to add gap coordinates of literal
    vector<int> aux2;
    // add every coordinates in theory
    for(int j = 0; j < this->n_clauses; j++) {
      aux2.push_back(j);
    }
    // removes all coordinates from literal of literal's gap
    for(int j = 0; j < aux1.size(); j++) {
      aux2.erase(std::remove(aux2.begin(), aux2.end(), aux1[j]), aux2.end()
         );
    }
    (*it)->addGap(aux2); // set the literal's gap
  }
}
```

3.5.3 Ordenação dos literais

O próximo passo é ordenar a lista de literais de acordo com o tamanho do Gap dos literais, os literais com menor Gap, ou seja, cobrem a maior quantidade de cláusulas, são escolhidos primeiro. Para ordenar o conjunto, foi utilizado o método std::sort da biblioteca <algorithm>, presente no standard do C++. Para ordenar, foi passado por parâmetro do método std::sort 3 argumentos: o começo da lista de literais, o final da lista de literais, e uma função que, dados dois literais, retorna qual é menor. O trecho de código abaixo apresenta este passo.

Listing 3.3 – Ordenação da lista de literais

```
void Dual::sortLiterals() {
  // calling sort from algorithm.h
  // this lambda iterates through literals and sort according
  // to which literal has smaller gap
  std::sort(this->literals->begin(), this->literals->end(),
    [](Literal *lit1, Literal *lit2) {
    if(lit1->getGapSize() == lit2->getGapSize())
      return lit1->getGapSize() < lit2->getGapSize();

    return lit1->getGapSize() < lit2->getGapSize();
```

```
11    });
12 }
```

3.5.4 Preparação do algoritmo de Transformação Dual

Após ordenar a lista de literais, é necessário gerar a árvore de estados, a fim de encontrar estados válidos, que representam os Implicantes Primários da teoria proposicional. O primeiro passo para gerar essa árvore de estados é configurar os estados iniciais. Os estados iniciais são compostos por uma lista de símbolos proposicionais, essa lista no estado inicial possui apenas o símbolo do literal do estado inicial. Além da lista de símbolos proposicionais, os estados iniciais possuem uma cláusula, representando o próprio estado. Essa cláusula é composta apenas pelo literal do estado inicial. O último componente dos estados iniciais é o conjunto Gap, obtido através do literal do estado inicial.

Após a configuração dos estados iniciais, é necessário chamar o metodo *generateStates*, que leva como parâmetro, além das configurações iniciais listadas acima, o índice do próximo literal da lista, que no código abaixo é representado pela variável i, e o tamanho do conjunto de literais.

Listing 3.4 – Preparação do algoritmo de Transformação Dual

```
1  void Dual::generateDual() {
2
3    const int size = this->literals->size(); // Save the amount of literals
4    int i = 1; // Iterator for generateStates()
5
6    // Iterate through literals list
7    for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
       literals->end(); ++it) {
8      vector<string> symbols; // list of symbols already used in the new
         clause
9      symbols.push_back((*it)->getSymbol()); // add the symbol of the literal
          iterated to the list of symbols
10     vector<int> new_gap = (*it)->getGap(); // get gap of literal iterated
11     Clause n_clause; // create a new blank clause
12     n_clause.addLiteral((*it)); // add the literal iterated to the clause
13     this->generateStates(new_gap, symbols, n_clause, i, size); // calls the
          method that generate the prime implicants
14     i++; // this iterator is added because we only check literals forward
         in the list, to avoid check literals already checked
15   }
16
17 }
```

3.5.5 Transformação Dual - Obtenção dos Implicantes Primários

Essa subseção irá abordar a implementação do método *generateStates*. A assinatura do método foi omitida nos trechos do código fonte, mas já foi especificada ao final da subseção anterior.

No prólogo do algoritmo, no *Listing 3.5*, primeiro é feito uma verificação se o último literal da lista de literais foi alcançado. Esse passo é apenas uma garantia de que elementos fora da lista não sejam lidos, garantindo o bom funcionamento do algoritmo. Outro passo na verificação inicial, é a verificação do Gap, se o Gap já é um conjunto vazio, não há necessidade de prosseguir com o algoritmo, pois uma solução já foi encontrada.

Após essa verificação inicial, os elementos recebidos por parâmetro são armazenados em variáveis auxiliares. O método *generateStates* é então chamado de forma recursiva no início para evitar que ramos da árvore de estados deixem de ser gerados. Isto ocorreria no seguinte caso: em uma teoria P, ao gerar um nodo raíz contendo o literal p_1, e uma lista de literais $\{p_2, p_3, ...\}$, caso existam nodos válidos, com as seguintes cláusulas intermediárias $C_0 : \{p_1, p_2\}$ e $C_1 : \{p_1, p_3\}$, a chamada de *generateStates* permite que C_1 seja alcançado, caso contrário, p_2 sempre seria inserido no estado, impossibilitando a criação de C_1.

Listing 3.5 – Prólogo do algoritmo

```
1  // x is the "i" from the method that call generateStates()
2  // x is the index of the literal in the list
3  if(x == size) return; // check if literal reach end of the list
4  if(n_gap.size() == 0) return; // check if Gap is already null
5
6  // prologue
7  // in here, all parameters are copied in new variables
8  vector<int> new_gap = n_gap; // actual gap of the state
9  vector<string> n_symbols = symbols; // symbols in state
10
11 Clause new_clause = n_clause; // state
12 // this part is tricky, is used to reach branchs of the states tree
13 // that may not be covered if we only call generateStates() at the end of
       the algorithm
14 this->generateStates(new_gap, n_symbols, new_clause, x+1, size);
```

Para o prosseguimento do algoritmo, é criada uma lista de coordenadas, onde serão armazenadas as coordenadas que serão removidas do Gap caso o literal seja inserido no estado. É necessário utilizar esta lista auxiliar.

Em seguida, no *Listing 3.6*, o conjunto de coordenadas do literal candidato a entrar no estado é iterado, nesta etapa são criadas algumas flags de controle (linhas 4, 13 e 15), para facilitar as checagens de inserção do literal no estado. Para cada coordenada do literal, é verificado se esta coordenada está contida no conjunto Gap, caso ela esteja contida, a

flag_gap recebe o valor True. Com o valor de *flag_gap* sendo true, é verificado se o literal será inserido no estado. Para o literal ser inserido na lista é feito uma verificação se seu símbolo está contido na lista de símbolos já presentes no estado. Após esta verificação, o literal pode então ser inserido no estado, e a *flag_remove* recebe o valor True (linha 19). A *flag_remove* garante que o literal não seja inserido repetidas vezes no estado e também é utilizada para remover as coordenadas de Gap.

Listing 3.6 – Gerando um candidato a novo estado

```
1
2  vector<int> removed_coord; // auxiliar vector of removed coordinates, will
       help forward in removing coordinates from gap
3  vector<Literal*>::iterator it; // iterator to get literal from list
4  bool flag_remove = false; // flag to check if coordinate need to be removed
       from gap
5
6  it = this->literals->begin(); // set iterator to begin of literal's list
7  advance(it, x); // move the pointer to the index of the next literal
       candidate to be in new state
8
9  vector<int> coordinates = (*it)->getQuantum().getCoordinates(); // get
       coordinates from literal
10
11 for(int i = 0; i < coordinates.size(); i++) { // for each coordinate of the
       literal
12   // this flag verify if coordinate is in gap set
13   bool flag_gap = (find(new_gap.begin(), new_gap.end(), coordinates[i]) !=
         new_gap.end());
14   if(flag_gap) { // if coordinate is in gap then check if symbol is not in
         states symbol list and keep it in a flag
15     bool flag_symbol = (find(n_symbols.begin(), n_symbols.end(), (*it)->
           getSymbol()) != n_symbols.end());
16     if(!flag_symbol && !flag_remove) { // if symbol is not in the states
           symbol list and flag_remove was not setted to true yet then
17       new_clause.addLiteral((*it)); // add literal to new clause
18       n_symbols.push_back((*it)->getSymbol()); // add symbol to states
             symbol list
19       flag_remove = true; // set flag_remove to true, so it can skip this
             code above if new coordinates are found in gap and only remove the
              coordinate from gap
20     }
21     if(flag_remove) { // if flag remove is true, put coordinate in list to
           remove later from gap
22       removed_coord.push_back(coordinates[i]);
23     }
24   }
25 }
```

```
26
27 // removing coordinates from gap
28 for(int i = 0; i < removed_coord.size(); i++) {
29   new_gap.erase(std::remove(new_gap.begin(), new_gap.end(), removed_coord[i
       ]), new_gap.end());
30 }
```

Caso Gap seja um conjunto vazio, então é feita uma verificação se o estado irá virar uma cláusula da nova teoria, no *Listing 3.7*. Neste passo, há uma modificação em relação ao algoritmo original, pois, como a modelagem adotada não possui literais com símbolos negados, não é necessário aplicar resolução, sendo apenas aplicada a subsunção quando necessário.

São contados quantos literais há na nova cláusula (linha 12), e quantos literais desta nova cláusula estão contidos em cada uma das cláusulas geradas na nova teoria (linha 16). No caso de todos literais da nova cláusula estarem contidos em uma outra cláusula C_i da nova teoria, e a nova cláusula possui mais literais, a nova cláusula então é descartada, pois ela seria subsumida por C_i (linha 26). Caso todos os literais da nova cláusula estejam contidos em uma outra cláusula C_i da nova teoria, e a nova cláusula possua menos literais, a cláusula C_i é descartada e a nova cláusula é inserida na teoria, pois a nova cláusula subsume C_i (linha 40).

Listing 3.7 – Adicionando o novo estado na lista de estados

```
1  // if gap is null, then a new state can be added in list
2  if(new_gap.size() == 0) {
3      if(this->states.empty()) { // if states list is empty, then the state
           can be added
4        this->states.push_back(new_clause);
5        return;
6      }
7      bool check_clause = true; // flag to check if clause is not already in
           state
8      int new_clause_size = new_clause.getLiterals().size(); // get the
           number of literals from state
9
10     for(int i = 0; i < this->states.size(); i++) { // for each state
           already found
11       int flag_nc = 0; // this flag will count how many literals from that
             state are contained in each state in the list
12       int size_cl = this->states[i].getLiterals().size(); // get the number
              of states
13
14       vector<Literal*> aux_literal = this->states[i].getLiterals(); // get
             literals from each state
15       for(int j = 0; j < aux_literal.size(); j++) {
```

```
16          flag_nc += new_clause.containsLiteral(aux_literal[j]); // check if
                state contains this literal, if it contains then add to the
                flag_nc counter
17        }
18        if(flag_nc == size_cl) { // if it's true, then every literals in
             state are present in another state of the list, and this state is
             repeated, so it not will be include on the list
19          if(size_cl < new_clause_size) { // if size of the new state is
                bigger, then it will not subsume other state
20            check_clause = false;
21          }
22        }
23
24        if(new_clause_size == flag_nc) { // if it's true, then every literals
             in state are present in another state of the list
25          if(flag_nc >= size_cl) { // if it's true, the state is bigger then
                other states, and will not subsume other state
26            check_clause = false;
27            break;
28          }
29          Clause aux_clause;
30          if(flag_nc < size_cl) { // if it's true, new state will subsume
                other state
31
32            /*
33               * subsuming the clause in list
34               * if the checked clause contains the new clause, and new
                    clause size < checked clause size
35               * then subsume checked clause
36            */
37
38            aux_clause = this->states[i];
39            // here the actual state is subsuming other states
40            this->states.erase(this->states.begin()+i);
41            this->states.push_back(new_clause);
42
43            return;
44          }
45        }
46      }
47      // if check clause is true, then the state is added to states list
48      if(check_clause) {
49        this->states.push_back(new_clause);
50      }
51      return;
52    }
```

Ao final do algoritmo, ocorre uma chamada recursiva do método *generateStates* para cada literal restante na lista.

Listing 3.8 – Iterando o algoritmo para gerar todos os estados

```
1 // call generateStates for the remaining literals in list after the current
      literal
2 for(; it != this->literals->end(); ++it) {
3   x++;
4   this->generateStates(new_gap, n_symbols, new_clause, x, size);
5 }
```

3.6 Contagem de Modelos

Para a contagem de modelos, foi utilizado o método descrito em Contagem de Modelos com algumas modificações que o tornam mais simples. Como na modelagem dos grafos não é possível existir literais negados, nesta implementação do método de contagem de modelos não foi necessário aplicar resolução na combinação de cláusulas para contagem de modelos, implicando algumas facilitações na implementação do algoritmo, que serão descritas com mais detalhes no decorrer do texto.

Como não há literais negados, há uma grande diferença na combinação de cláusulas utilizadas na contagem de modelos, e que visam remover propagação no número de modelos, pois existirá apenas uma combinação contendo todas as cláusulas, onde a quantidade de modelos desta combinação propagará por todas as cláusulas.

Nesta implementação apresentada no *Listing 3.9*, um array é criado para armazenar a quantidade de modelos associados a cada cláusula. Como foi visto no capítulo anterior, para cada cláusula é associado um número de modelos calculado na forma 2^{n-k}, onde n é o número de símbolos da teoria, e k é o número de símbolos presentes na cláusula.

Para a contagem da quantidade de modelos da combinação de todas as cláusulas, o mesmo cálculo foi efetuado, sendo o número de modelos igual a 2^{n-k}, onde n é o número de símbolos da teoria, e k é o número de símbolos presentes em todas as cláusulas da teoria. Este valor será armazenado em uma variável chamada *value_ combination*(linha 34).

Listing 3.9 – Contando a quantidade de modelos de cada cláusula e da combinação de todas as cláusulas

```
1 int n_st = this->states.size();
2 int cls[n_st] = {0}; // vector containing number of models in each clause
3 int value_combination = 0; // count the number of models of combination of
    clauses
4
5 for(int i = 0; i < n_st; i++) {
```

```
6   int st_size = this->states[i].getLiterals().size();
7   cls[i] = pow(2, this->n_symbols - st_size); // set the clause models to
        2^(n-k), where n is the number of symbols in theory and k is the
        number of symbols in clause
8 }
9
10 vector<Literal *> aux = this->states[i].getLiterals(); // combination of
     clauses, we will use it for model counting, according to the number of
     symbols contained in here
11 for(int j = 1; j < n_st; j++) {
12   vector<Literal *> temp;
13
14   temp = this->states[j].getLiterals(); // other clauses to be added
15
16   int add[temp.size()];
17   for(int k = 0; k < temp.size(); k++) {
18     add[k] = 1; // populate with 1 | all the symbols to be added in aux for
          further model counting
19   }
20   for(int k = 0; k < aux.size(); k++) {
21     for(int m = 0; m < temp.size(); m++) {
22       if(aux[k]->getSymbol() == temp[m]->getSymbol()) { // if symbol
            already exists
23         add[m] = 0; // set to 0 and symbol is not added to aux
24       }
25     }
26   }
27
28   for(int k = 0; k < temp.size(); k++) {
29     if(add[k]) aux.push_back(temp[k]); // adding symbols to aux for
          counting models
30   }
31 }
32 int result = pow(2, this->n_symbols - aux.size()); // calculate the number
     of models in this clauses combination
33
34 value_combination = result;
```

Para remoção da propagação na quantidade de modelos, é necessário subtrair o valor de *value_combination* de cada uma das cláusulas associadas à esta combinação de cláusulas. Como todas as cláusulas da teoria estão presentes na combinação, então *value_combination* é subtraído da quantidade de modelos associados a cada cláusula da teoria. O trecho de código abaixo apresenta este passo.

Listing 3.10 – Removendo propagação na contagem de modelos

```
1 for(int j = 0; j < n_st; j++) {
2   cls[j] -= value_combination;
```

```
}
```

Após remover a propagação da quantidade de modelos, é então somada a quantidade total de modelos associados à teoria. Este passo é apresentado no *Listing 3.11*, e consiste apenas da soma da quantidade de modelos associados a cada cláusula, e a soma do valor armazenado em *value_combination*.

Listing 3.11 – Somando os modelos de cada cláusula e da combinação de cláusulas para obter a quantidade total de modelos da teoria

```
int n_models = value_combination;
for(int i = 0; i < n_st; i++) {
  n_models += cls[i];
}
return n_models;
```

3.7 Verificação do Isomorfismo

O último passo da metodologia é a verificação de isomorfismo entre dois grafos. Caso as teorias geradas a partir da transformação dual de ambos os grafos possuam a mesma quantidade de modelos, os grafos podem ser isomorfos.

Conforme foi descrito na seção Propriedades da Formas Normais Primárias, no capítulo anterior, duas teorias proposicionais podem ser consideradas da mesma família de teorias se possuírem o mesmo número de símbolos proposicionais, e a mesma quantidade de modelos proposicionais.

Caso duas teorias proposicionais sejam da mesma família de teorias, elas são congruentes. Se as duas teorias que representam dois grafos forem de uma mesma família de teorias, os dois grafos são então considerados isomorfos.

Na seção de Resultados, os resultados obtidos na execução do programa serão avaliados e uma conclusão acerca destes resultados será obtida.

3.8 Execução do Programa

Nesta seção será demonstrada a execução da metodologia implementada, onde dois grafos G_1 e G_2, sabidamente isomorfos, serão avaliados pelo programa. Será mostrado cada passo contendo resultados intermediários, ilustrando cada etapa descrita anteriormente.

Os grafos do exemplo possuem 5 vértices e 5 arestas. G_1 possui o conjunto de arestas = {(1, 3), (1, 5), (3, 5), (3, 4), (4, 2)}. G_2 possui o conjunto de arestas = {(3, 1), (3, 5), (1, 5), (1, 2), (2, 4)}

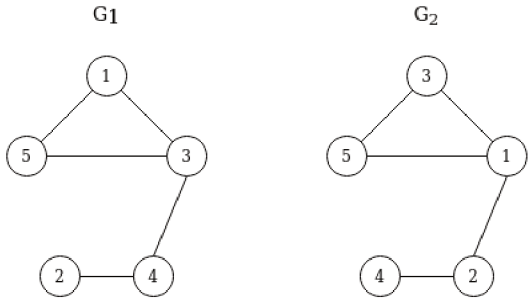

Figura 9 – Grafos G_1 e G_2 utilizados como exemplo na execução do programa

Neste exemplo, os grafos não serão diretamente comparados ao final do programa, apenas será executado o programa para obter informações de cada etapa da metodologia, para cada grafo separadamente, de acordo com o descrito na seção anterior.

```
archive read
Gap generated
Literals sorted, read to run dual transform
printing clauses
C0: {p1, p3}
C1: {p1, p5}
C2: {p2, p4}
C3: {p3, p5}
C4: {p3, p4}

printing gap
Gap of p3:
{1, 2}

Gap of p1:
{2, 3, 4}

Gap of p5:
{0, 2, 4}

Gap of p4:
{0, 1, 3}

Gap of p2:
{0, 1, 3, 4}

Generating dual clauses
Dual clauses created
printing states
C0: {p3, p5, p2}
C1: {p3, p5, p4}
C2: {p3, p1, p2}
C3: {p3, p1, p4}
C4: {p1, p5, p4}

Number of models: 16
```

Figura 10 – Execução do programa utilizando G_1 como entrada

A execução do programa utilizando G_1 como entrada resultou em uma teoria com 16 modelos associados.

A execução do programa utilizando G_2 como entrada também resultou em uma teoria com 16 modelos associados.

A saída de ambos os grafos, que são isomorfos, resultou em 16 modelos. Considerando a metodologia utilizada, o programa acusaria que os grafos G_1 e G_2 são isomorfos.

```
archive read
Gap generated
Literals sorted, read to run dual transform
printing clauses
C0: {p1, p2}
C1: {p1, p3}
C2: {p1, p5}
C3: {p2, p4}
C4: {p3, p5}

printing gap
Gap of p1:
{3, 4}

Gap of p2:
{1, 2, 4}

Gap of p3:
{0, 2, 3}

Gap of p5:
{0, 1, 3}

Gap of p4:
{0, 1, 2, 4}

Generating dual clauses
Dual clauses created
printing states
C0: {p1, p5, p4}
C1: {p1, p3, p4}
C2: {p1, p2, p5}
C3: {p1, p2, p3}
C4: {p2, p3, p5}

Number of models: 16
```

Figura 11 – Execução do programa utilizando G_2 como entrada

Para demonstrar que o programa identifica grafos não isomórficos, será utilizado como entrada um terceiro grafo G_3, que também possui 5 vértices e 5 arestas, seus vértices possuem os mesmos graus que os vértices de G_1 e G_2. O conjunto de arestas de G_3 é = {(1, 2), (1, 3), (2, 5), (3, 5), (4, 5)}

Como é possível perceber no diagrama abaixo, G_3 não é isomorfo à G_1 e G_2, uma vez que G_1 e G_2 possuem um ciclo formado por três vértices, e G_3 possui um ciclo formado por quatro vértices.

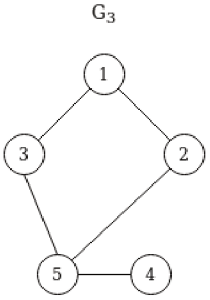

Figura 12 – Grafo G_3 utilizado para exemplificar não-isomorfismo com G_1 e G_2

```
archive read
Gap generated
Literals sorted, read to run dual transform
printing clauses
C0: {p1, p2}
C1: {p1, p3}
C2: {p2, p5}
C3: {p3, p5}
C4: {p4, p5}

printing gap
Gap of p5:
{0, 1}

Gap of p1:
{2, 3, 4}

Gap of p2:
{1, 3, 4}

Gap of p3:
{0, 2, 4}

Gap of p4:
{0, 1, 2, 3}

Generating dual clauses
Dual clauses created
printing states
C0: {p5, p2, p3}
C1: {p5, p1}
C2: {p2, p3, p4}

Number of models: 14
```

Figura 13 – Execução do programa utilizando G_3 como entrada

A execução do programa utilizando G_3 como entrada resultou em uma teoria com 14 modelos associados, resultando em um valor diferente dos valores obtidos executando com G_1 e G_2 como entrada, indicando que G_3 não é isomorfo a G_1 e G_2.

4 Resultados e Discussão

4.1 Execução dos Testes

Para realização de testes do método de verificação de isomorfismo entre grafos implementado neste trabalho, foi gerado um *benchmark* para validação de pares de grafos isomorfos, gerando um grafo aleatoriamente, e gerando um grafo isomorfo a partir deste primeiro grafo aplicando uma função que mapeia os vértices do grafo, fazendo com que o segundo grafo seja isomorfo ao primeiro.

Foi produzido um programa em C++ para geração de pares de grafos, e um *script* utilizando Python 3.6.9 que automatizasse esta criação de grafos.

No programa que gera grafos isomorfos são sorteados uma quantidade aleatória de vértices e de arestas, utilizando a função *rand()*, de acordo com um limite definido por meio de dois *#define* do C++, um para o limite superior, e outro para o limite inferior de vértices. Na geração do benchmark o limite superior utilizado foi de 25 vértices, e o limite inferior foi de 5 vértices. A quantidade de arestas também é definida através da função *rand()*, de modo que seu valor máximo poderia ser igual a $(v-1)^2$, sendo v o número de vértices. Após isso, as arestas são geradas também aleatoriamente, e é feito um embaralhamento da ordem dos vértices em um *vector* auxiliar, e a partir deste *vector* as arestas são mapeadas para o segundo grafo.

A saída do programa é um arquivo contendo os dois grafos, de modo que seja possível manter um registro facilmente organizado de quais são os pares isomorfos. Todos os arquivos contendo pares de grafos foram então armazenados em um diretório separado, que será lido por um *script* que executa o *benchmark*.

Para executar o benchmark foi criado um programa em C++, que lê o arquivo contendo os dois grafos, aplica o método implementado, e retorna 1 se os grafos forem isomorfos, ou 0 se não forem isomorfos. Foi criado também um script para automatizar a execução destes testes, e fazer uma contagem de quantos testes retornaram valor 1, e assim ter um resultado da eficácia do algoritmo para identificação de isomorfismo.

Ao todo foram gerados 105 pares de grafos isomorfos, e ao serem submetidos ao programa, todos os 105 pares de grafos isomorfos foram identificados como isomorfos pelo programa.

Para a verificação do não isomorfismo, a geração do benchmark foi feita de forma aleatória. Foi escrito um programa em C++ que gerasse grafos com um tamanho definido, onde vários grafos de mesma quantidade de vértices e arestas foram gerados.

Há uma grande dificuldade em avaliar se os grafos que foram gerados aleatóriamente são isomorfos ou não, ao contrário da geração de pares isomorfos, onde era possível ter a certeza que os grafos gerados eram isomorfos. Devido a esta dificuldade, foram gerados conjuntos de grafos com uma quantidade de vértices não muito grande, possibilitando a avaliação manual dos resultados.

Foram gerados grafos com quantidades de vértices e arestas arbitrárias, sendo gerados no total 10 grafos com 6 vértices e 7 arestas, 25 grafos com 7 vértices e 10 arestas, e 36 grafos com 8 vértices e 12 arestas. Para a execução dos testes, os grafos foram separados em diretórios de acordo com a quantidade de vértices, onde cada grafo foi comparado com o restante dos grafos contidos no diretório, totalizando 45 execuções para o conjunto de grafos com 6 vértices, 300 execuções para o conjunto de grafos com 7 vértices e 630 execuções para o conjunto de grafos com 8 vértices.

A execução do primeiro conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 39 testes, identificou não-isomorfismo utilizando o método desenvolvido em 5 testes, e identificou erroneamente o par de grafos da Figura 14 como isomorfos.

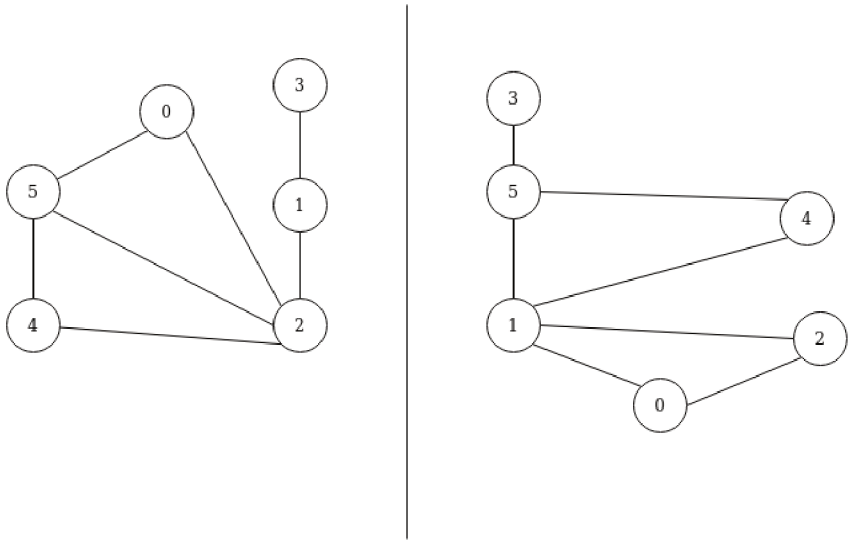

Figura 14 – Resultado incorreto observado nos testes do conjunto de grafos com 6 vértices

A execução do segundo conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 283 testes, identificou não-isomorfismo utilizando o método desenvolvido em 16 testes, e identificou erroneamente o par de grafos da Figura 15 como isomorfos.

A execução do terceiro conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 601 testes, identificou não-isomorfismo utilizando o método desenvolvido em 26 testes, e identificou erroneamente os pares de grafos da Figura 16, Figura 17 e Figura 18 como isomorfos.

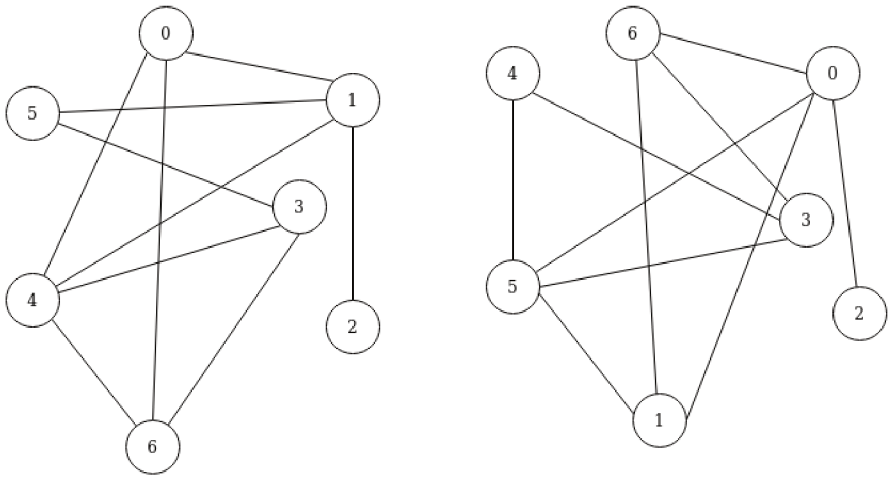

Figura 15 – Resultado incorreto observado nos testes do conjunto de grafos com 7 vértices

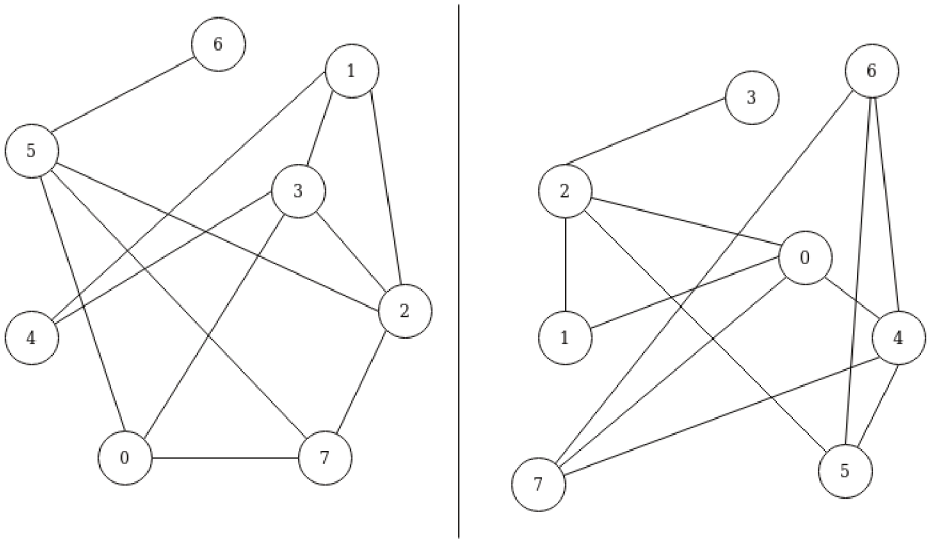

Figura 16 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices

Nos pares que acusaram de forma incorreta o isomorfismo, é possível notar que em todos havia um vértice com apenas uma aresta, indicando que grafos com esta característica podem ser problemáticos para o método desenvolvido. Como as falhas apareceram em grafos com a mesma característica, pode ser que este método apresente problemas somente nas classes de grafos com esta característica, sendo necessário uma avaliação melhor e outros testes para averiguar se é somente esta classe de grafos que é excluída do conjunto de respostas corretas geradas pelo método.

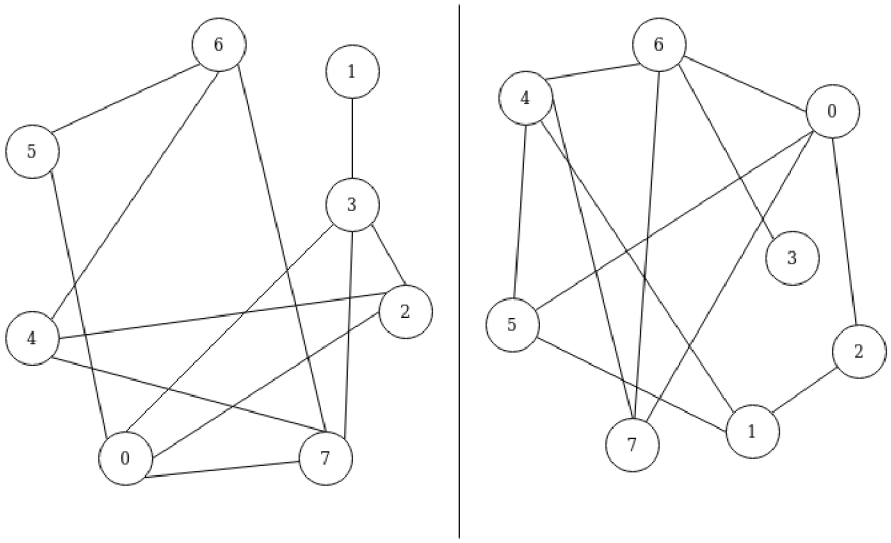

Figura 17 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices

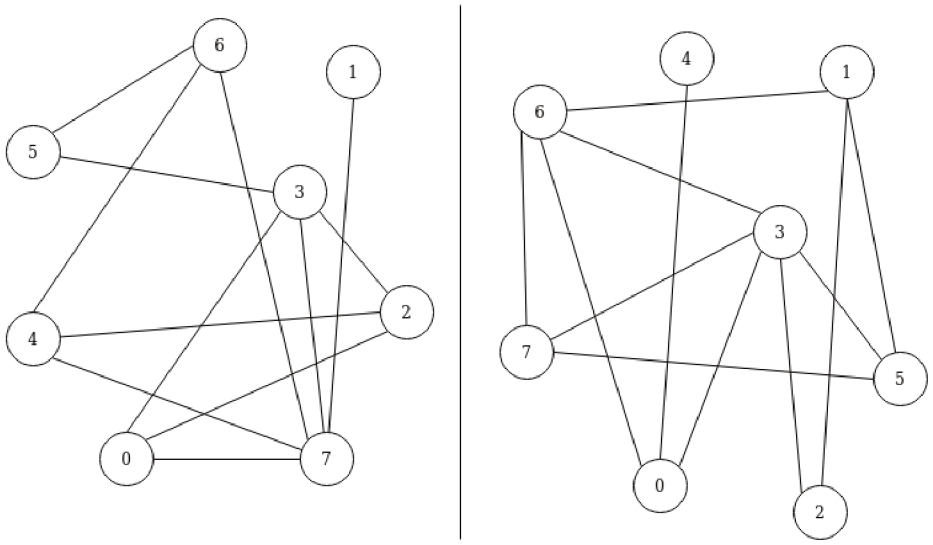

Figura 18 – Resultado incorreto observado nos testes do conjunto de grafos com 8 vértices

5 Conclusão

5.1 Considerações Finais

O presente trabalho teve como objetivo investigar a relação entre os pares de Implicantes e Implicados primários de teorias proposicionais, geradas a partir de uma modelagem de grafos, com o problema de isomorfismo entre grafos.

A partir disto, foi implementado um método utilizando a linguagem de programação C++, em que o algoritmo de transformação dual é utilizado para obtenção dos Implicantes primários dos grafos que foram modelados em teorias proposicionais. Com os conjuntos de Implicantes primários, foi possível obter a quantidade de modelos de cada teoria proposicional, e com a quantidade de símbolos proposicionais de cada teoria, foi possível categorizá-las e indicar se elas pertenciam a uma mesma família de teorias.

Com o método implementado, foi necessário atestar sua eficácia. O programa obtido foi submetido a testes de isomorfismo, onde eram gerados pares de grafos isomórficos, e o programa identificava corretamente o isomorfismo entre esses pares de grafos utilizando a categorização descrita anteriormente, onde grafos isomorfos pertenciam a mesma família de teorias proposicionais.

O programa foi submetido também a testes de não-isomorfismo, onde foram gerados vários grafos com uma mesma quantidade de vértices e arestas, e foi efetuado um teste comparando todos os grafos em pares. Neste teste, houveram resultados promissores, em que a grande maioria dos testes identificou o isomorfismo e classificou os grafos em famílias de teorias proposicionais diferentes, porém, houveram alguns testes que falharam, e em todos os testes que apresentaram falha os grafos possuíam a mesma característica de ter um vértice com apenas uma aresta, sendo necessário uma investigação mais aprofundada do motivo pelo qual somente estes grafos foram classificados na mesma família de teorias proposicionais.

5.2 Trabalhos Futuros

Este trabalho gerou um método de identificação de isomorfismo entre grafos que possui experimentação do seu funcionamento correto para a maioria dos grafos testados, sendo ineficaz nos testes apenas para um conjunto de grafos com uma mesma característica, portanto, a sugestão para um trabalho futuro é a investigação se o método falha apenas para o conjunto de grafos identificado, ou se falha para mais conjuntos de grafos, tornando possível ou não a adição de condições de exclusão ao método.

Referências

BITTENCOURT, G. *Counting Models using Prime Forms*. 2007. Unpublished. Citado na página 30.

BITTENCOURT, G. *Boolean Concepts*. 2008. Unpublished. Citado na página 30.

BITTENCOURT, G.; MARCHI, J.; PADILHA, R. A syntactic approach to satisfaction. 09 2003. Citado 3 vezes nas páginas 5, 7 e 14.

BITTENCOURT, G.; MARCHI, J.; PADILHA, R. S. A syntactic approach to satisfaction. In: KONEV, B.; SCHIMIDT, R. (Ed.). *4^{th} Inter. Workshop on the Implementation of Logic (LPAR03)*. [S.l.]: Univ. of Liverpool and Univ. of Manchester, 2003. p. 18–32. Citado na página 28.

COOK, S. A. The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1971. (STOC '71), p. 151–158. Disponível em: <http://doi.acm.org/10.1145/800157.805047>. Citado 2 vezes nas páginas 13 e 23.

DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 7, n. 3, p. 201–215, jul. 1960. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321033.321034>. Citado 2 vezes nas páginas 26 e 30.

GOMES, C.; SABHARWAL, A.; SELMAN, B. Model counting: A new strategy for obtaining good bounds. In: . [S.l.: s.n.], 2006. v. 1. Citado na página 30.

JOHNSONBAUGH, R. *Discrete Mathematics revised edition*. [S.l.]: New York: Macmillian Publishing Company, 1986. Citado 3 vezes nas páginas 17, 20 e 21.

NIEVERGELT, Y. *Foundations of Logic and Mathematics*. [S.l.]: Springer, 2002. Citado 4 vezes nas páginas 17, 18, 20 e 21.

POLYA, G. Sur les types des propositions composées. *The Journal of Symbolic Logic*, Cambridge University Press, v. 5, n. 3, p. 98–103, 1940. Citado 4 vezes nas páginas 5, 7, 14 e 30.

QUINE, W. V. On cores and prime implicants of truth functions. *The American Mathematical Monthly*, Taylor & Francis, v. 66, n. 9, p. 755–760, 1959. Citado 2 vezes nas páginas 21 e 25.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Malaysia; Pearson Education Limited,, 2016. Citado 3 vezes nas páginas 21, 24 e 25.

SKIENA, S. S. *The algorithm design manual: Second edition*. 2nd. ed. [S.l.]: Springer Science & Business Media, 2008. v. 1. 1–730 p. ISBN 9781848000698. Citado 5 vezes nas páginas 5, 7, 13, 17 e 20.

Apêndices

APÊNDICE A – Artigo

Verificação de Isomorfismo de Grafos Via Formas Normais Primárias

Gustavo Tarciso da Silva

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina

Resumo. *O objetivo deste trabalho foi produzir um método algorítmico para detecção de isomorfismo entre dois grafos. Através de uma modelagem dos grafos em teorias proposicionais na forma normal conjuntiva, e então, aplicar um algoritmo de transformação dual para encontrar uma teoria equivalente de tamanho reduzido na forma normal disjuntiva, composta apenas por implicantes primários. Com o conjunto de implicantes primários, o próximo passo foi contar a quantidade de modelos desta nova teoria reduzida, cujo objetivo é categorizar as teorias em famílias. Estas teorias foram categorizadas no trabalho de [Polya 1940] e de [Bittencourt et al. 2003a], e concluem que se duas teorias pertencem a mesma famílias, logo elas são congruentes. A motivação deste trabalho vem desta propriedade de congruência entre as teorias. O resultado obtido foi o desenvolvimento de um algoritmo que é capaz de identificar o isomorfismo entre dois grafos. Entretanto, durante as análises alguns casos de não-isomorfismo foram identificados erroneamente como isomorfos. Estes grafos aparentemente possuem uma característica em comum, tornando os resultados, mesmo que incorretos, promissores. Com isto conclui-se que este trabalho cumpriu com os objetivos definidos, entretanto, há a necessidade de investigar se o método de fato só falha para o conjunto de grafos com a característica identificada.*

Abstract. *The objective of this study was to produce an algorithmic method to detect isomorphism between two graphs. Through modeling the graphs in propositional theories in the normal conjunctive form, and then applying a dual-transform algorithm to find an equivalent theory of reduced size in the normal disjunctive form, composed only of prime implicants. With the set of prime implicants, the next step was to count the number of models of this new reduced theory, whose objective is to categorize theories in families. These theories were categorized in the work of [Polya 1940] and [Bittencourt et al. 2003a], and both conclued that if two theories belong to the same families, they are therefore congruent. The motivation for this study comes from this property of congruence between theories. The result obtained was the development of an algorithm that is able to identify the isomorphism between two graphs. However, during the analysis some cases of non-isomorphism were mistakenly identified as isomorphs. These graphs apparently have a common characteristic, making the results, even if incorrect, promising. Therewith it is concluded that this work fulfilled the defined objectives, however, there is a need to investigate if the method in fact only fails for the set of graphs with the identified characteristic.*

1. Introdução

Muitos problemas do mundo real podem ser modelados em forma de grafos, dessa forma, diversos algoritmos em grafo podem ser utilizados para resolução desses problemas. Tais algoritmos possuem complexidades diferentes. Alguns podem ser da ordem de complexidade de tempo determinístico polinomial, já outros podem apresentar solução em uma ordem de complexidade de tempo não-determinístico polinomial. Há também alguns cuja ordem de complexidade ainda não é certa, e que é um dos nossos objetos de estudo, o problema de isomorfismo entre grafos, reside nesse conjunto de problemas que não se sabe ainda a qual classe de complexidade pertence [Skiena 2008].

O outro objeto de estudo é a lógica proposicional, que por sua vez, pode ser utilizada para modelar a resolução de problemas em grafos através do método da redução. Um problema em grafos, que pode estar na classe NP pode ser reduzido, ou seja, transformado, em um problema do campo da lógica já conhecido: a satisfação booleana (SAT). SAT foi o primeiro problema a ser provado pertencer a classe NP-completo [Cook 1971].

Já que não se sabe ainda a qual classe de complexidade computacional o problema de isomorfismo entre grafos pertence, e os algoritmos para verificação do isomorfismo são da ordem de complexidade fatorial no pior caso [Skiena 2008], serão estudadas as propriedades desses dois campos citados acima, afim de buscar relações entre eles, e tentar criar uma ponte entre essas duas formas de representar o problema, buscando assim, uma forma alternativa de resolver o problema de isomorfismo entre grafos.

1.1. Objetivo Geral

O objetivo deste projeto é tentar desenvolver um método algorítmico de checagem de isomorfismo entre dois grafos.

Um par de grafos é considerado isomorfo quando há uma função que mapeie os vértices de um grafo para outro, de forma que as propriedades do grafo sejam as mesmas, tornando os grafos equivalente. As sentenças lógicas, por sua vez, possuem uma característica semelhante a esta, pois dada uma sentença, é possível obter uma sentença lógica equivalente apenas trocando seus símbolos, de forma que uma função mapeie os símbolos trocados mantendo suas propriedades.

Para saber se duas sentenças lógicas, que serão apresentadas neste trabalho como teorias proposicionais, podem ser consideradas equivalentes e ter a mesma representação simbólica, é necessário obter os conjuntos de Implicantes e Implicados primários. Uma vez que temos o conjunto de Implicantes e Implicados primários, podemos utilizá-los em conjunto com a quantidade de símbolos da teoria proposicional para descobrir de qual família de teorias proposicionais ela faz parte.

Desta forma, será representando grafos através de teorias proposicionais, e encontrando seus conjuntos de implicantes primários, que será investigado se é possível decidir se um par de grafos é ou não isomorfo, de modo que seja possível verificar qualquer relação entre a existência ou não de isomorfismo entre os dois grafos avaliados, a partir das famílias que as teorias proposicionais geradas a partir dos grafos pertencem.

Portanto, será investigado se existe uma relação entre isomorfismo de grafos e a família na qual as teorias proposicionais, geradas a partir dos grafos, pertencem. Caso haja relação, será investigado também se esta relação é válida para todo caso.

Caso não seja observada relação entre a quantidade de modelos nas teorias proposicionais obtidas através dos grafos e a presença de isomorfismo entre estes grafos, será avaliado também se, para algum conjunto específico de grafos, essa verificação funciona.

2. Conceitos

2.1. Sentenças Lógicas e Satisfação Booleana

Lógica pode ser expressa como o uso do raciocínio correto nas relações entre sentenças, onde não há como saber se uma sentença é verdadeira ou não utilizando a lógica, mas sim se a relação entre determinadas sentenças é válida. Como por exemplo:

Sempre que chove, faz frio.

Quando faz frio, pessoas usam casacos.

Sempre que chove, pessoas usam casacos.

No exemplo, a lógica não é usada pra saber se as duas primeiras sentenças são verdadeiras, mas é utilizando a lógica que podemos deduzir que a terceira sentença é verdadeira, caso as duas primeiras sejam verdadeiras.

Na lógica proposicional as sentenças permitidas são sentenças atômicas, ou seja, elementos sintáticos indivisíveis, chamadas de literais, onde a sentença consiste de um único símbolo proposicional. Esse símbolo proposicional nada mais é que um rótulo arbitrário para uma proposição que pode ser verdadeira ou falsa. No exemplo da seção anterior podemos representar a proposição *sempre que chove* com o símbolo *P* e *faz frio* com o símbolo *Q*. Com essas senteças atômicas é possível criar sentenças lógicas mais complexas, com o uso de conectivos lógicos. [Nievergelt 2002]

Uma forma de representar os possíveis valores-verdade para cada uma das combinações de valores dos literais para uma dada proposição, é através da tabela verdade, onde é usado F para falso e V para verdadeiro. Um exemplo de tabela verdade:

P	Q	$P \wedge Q$
V	V	V
V	F	F
F	V	F
F	F	F

Sabendo que computadores possuem uma certa facilidade inerente para simular conceitos de verdade, é possível também simular esses valores de verdade da tabela utilizando álgebra, onde o valor verdadeiro seria representado por 1, e o falso por 0, e os operadores $\wedge$ representado por * e $\vee$ representado por +, mantendo as características de cada operação. Para ilustrar esse comportamento, as tabelas verdade dos operadores $\wedge$ e $\vee$ são mostradas a seguir em fórmulas algébricas.

Para uma sentença ser considerada satisfatível, ela precisa ser verdadeira para pelo menos uma combinação de valores de seus literais, por exemplo, a sentença $p_1 \wedge p_2 \wedge p_3$ é satisfatível, pois quando p_1, p_2 e p_3 são verdadeiros, a sentença é verdadeira. Já a sentença $p_1 \wedge \neg p_1$ não possui um valor para p_1 que torne a sentença verdadeira, nesse caso, ela é chamada de contradição. Determinar se uma fórmula lógica proposicional é satisfatível foi o primeiro problema a ser provado NP-completo [Cook 1971].

2.2. Formas Normais Canônicas

As formas normais canônicas são sentenças formadas apenas por conjunções de disjunções ou disjunções de conjunções, podendo ser classificadas em *Forma Normal Conjuntiva* (*FNC*) e *Forma Normal Disjuntiva* (*FND*).

A FNC é uma sentença expressa como conjunções ($\wedge$) de fórmulas menores denominadas cláusulas, compostas apenas por disjunções ($\vee$). Um exemplo de FNC é $(p_1 \vee p_2 \vee p_3) \wedge (p_4 \vee p_5)$. Há também uma família restrita de sentenças k-FNC, onde uma sentença k-FNC possui k literais por cláusula, onde a fórmula seria expressa por $(p_{1,1} \vee \ldots \vee p_{1,k}) \wedge \ldots \wedge (p_{n,1} \vee \ldots \vee p_{n,k})$, onde $p_{i,k}$ é um literal.

Já a FND é uma sentença expressa como disjunções ($\vee$) de cláusulas compostas apenas por conjunções ($\wedge$). Um exemplo de FND é $(p_1 \wedge p_2) \vee (p_3 \wedge p_4 \wedge p_5)$. Assim como as FNCs, existe uma família restrita de sentenças k-FND, onde cada sentença k-FND possui k literais por cláusula, onde a fórmula seria expressa por $(p_{1,1} \wedge \ldots \wedge p_{1,k}) \vee \ldots \vee (p_{n,1} \wedge \ldots \wedge p_{n,k})$, onde $p_{i,k}$ é um literal [Nievergelt 2002] [Johnsonbaugh 1986].

2.3. Formas Normais Primárias

Fórmulas Fundamentais são fórmulas em lógica proposicional, compostas apenas por conjunções de literais, onde esses literais só podem aparecer uma vez na sentença, ou seja, como a fórmula fundamental é uma sequência de $\wedge$ (Operador e lógico), se o mesmo literal aparecer mais de uma vez na fórmula fundamental, ela será redundante. Se aparecer um determinado literal na fórmula fundamental, e o mesmo literal negado, será uma contradição.

Sendo P uma sentença lógica, um Implicante Primário de P é uma Fórmula Fundamental que implica P, e que caso remova um de seus literais, não mais implica P, ou seja, a Fórmula Fundamental necessita estar reduzida, e para isso, sempre que a remoção de um literal da Fórmula Fundamental não influenciar na implicação de P, esse literal é removido, e quando isso ocorre, a Fórmula Fundamental é considerada uniliteralmente redundante. O conjunto de Implicantes Primários de uma sentença P será chamado PI_P.

Considerando como exemplo, que os literais de P sejam p_1, p_2, p_3, p_4, se $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ resulta em verdadeiro, e que $p_1 \wedge p_2 \wedge p_3 \wedge p_4 \rightarrow P$, então $p_1 \wedge p_2 \wedge p_3 \wedge p_4$ é uma Fórmula Fundamental, mas se essa sentença for equivalente a, por exemplo, $p_1 \vee p_3 \rightarrow P$, então $p_1 \vee p_3$ é um Implicante Primário de P. Como um implicante primário é uma Fórmula Fundamental mínima que implica P, uma sentença que seja composta por disjunções de implicantes primários, será uma Forma Normal mínima equivalente a P.

O número de Implicantes Primários que uma sentença tem é finito, uma vez que o número de literais de uma sentença é finito, o número de todas as combinações possíveis de literais também é finito.

Para uma sentença lógica P ser comprimida em uma forma normal equivalente, é necessário transformar a sentença em uma disjunção de *implicantes primários*, e depois remover o maior número possível de cláusulas supérfluas. Para a remoção de cláusulas supérfluas, se tivermos duas cláusulas C_1 e C_2, que são implicantes primários de P, e elas tiverem apenas um literal contraditório, por exemplo, se C_1 possui um literal p_1, e C_2 possui um literal $\neg p_1$, então exclui-se o literal p_1 a partir da conjunção $C_1 \wedge C_2$, resultando

em uma única cláusula, contendo todos os literais de C_1 e C_2 exceto a contradição, e excluí-se as cláusulas C_1 e C_2. Essa operação é chamada de Resolução

Por exemplo, se C_1: $(p_1 \wedge p_2 \wedge \neg p_3 \wedge p_4)$ é implicante primário de P, e C_2: $(p_1 \wedge p_2 \wedge p_3 \wedge p_4)$ também é implicante primário de P, fazendo a conjunção de C_1 e C_2, remove-se o literal p_3, e resulta em uma nova clausula C_3 composta pelos literais de C_1 e C_2, onde o novo implicante primário seria C_3: $p_1 \wedge p_2 \wedge p_3$ que implica P. Se todos os literais de um implicante primário C_1 estão em outro implicante primário C_2, C_1 subsume C_2. Tomando como exemplo C_1 sendo $p_1 \wedge p_2$ e C_2 sendo $p_1 \wedge p_2 \wedge p_3 \wedge p_4$, como todos os literais de C_1 então contidos em C_2, C_1 subsume C_2, portanto, C_2 pode ser excluído [Quine 1959].

2.3.1. Algoritmo de Transformação Dual

Algoritmos para resolver o problema da satisfação booleana (SAT) como o algoritmo de Davis-Putnam-Logemann-Loveland (DPLL)[Davis and Putnam 1960], geralmente utilizam uma abordagem semântica, de forma que é possível atribuir valor de verdade aos símbolos proposicionais. Por outro lado, neste trabalho será utilizado um método baseado em uma abordagem sintática, explorando propriedades das formas normais de um conjunto de cláusulas proposicionais, que será chamado de teoria proposicional, ou seja, esta teoria proposicional é representada por formas normais conjuntivas (FNC) ou disjuntivas (FND).

Dada uma teoria proposicional *P*, *P* pode ser transformada em uma FNC P_c ou em uma FND P_d, P_c, por sua vez, pode ser transformada em P_d, e vice-versa, utilizando distributividade dos operadores lógicos $\wedge$ e $\vee$.

O algoritmo que será utilizado calcula a representação mínima de uma teoria P_d. A representação mínima de P_d é a P_c gerada a partir de P_d. Essa representação mínima é o conjunto de cláusulas não contraditórias, e que não podem ser subsumidas. Como visto anteriormente, essas cláusulas são os implicantes primários da teoria *P*.

Cada cláusula mínima representa mínimamente o conjunto de valores verdade atribuídos aos símbolos da teoria P, para que P seja satisfatível. Gerando todas as cláusulas duais mínimas, obtêm-se todas as combinações de valores verdade para os símbolos da teoria P que tornem ela satisfatível. A ideia principal é que a quantidade de cláusulas duais mínimas geradas seja sempre menor ou igual a quantidade de literais.

Considerando tudo o que foi abordado anteriormente, outro elemento precisa ser inserido para que o algoritmo funcione. O elemento fundamental do algoritmo é o conceito de *quantum*, definido como um par (p, *F*), onde p é um literal, e $F \subseteq P_c$ é o conjunto de coordenadas que representam a posição de cada cláusula em que p pertence. Para exemplificar, em uma teoria: $(p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee \neg p_3)$, as cláusulas seriam $(p_1 \vee p_2 \vee p_3)$, $(\neg p_1 \vee p_2 \vee p_3)$ e $(p_1 \vee p_2 \vee \neg p_3)$, e o conjunto de coordenadas para representar essas cláusulas seria $P_c = \{0, 1, 2\}$, sendo os números contidos nos conjuntos a posição da cláusula, onde 0 representa $(p_1 \vee p_2 \vee p_3)$, 1 representa $(\neg p_1 \vee p_2 \vee p_3)$ e 2 representa $(p_1 \vee p_2 \vee \neg p_3)$. A forma utilizada para representar as coordenadas de um literal é p^F, sendo p o literal, e F o conjunto de coordenadas. Portanto, as coordenadas são representadas por $p_1^F = \{0, 2\}$, $\neg p_1^F = \{1\}$, $p_2^F = \{0, 1, 2\}$, $p_3^F = \{0,1\}$ e $\neg p_3^F = \{2\}$.

Para gerar uma DNF P_d mínima, novas cláusulas são formadas de modo que, a união das coordenadas dos *quanta* dos literais tem que cobrir todas as cláusulas de P_c, ou seja, seguindo o exemplo anterior, para uma nova cláusula ser formada, a união de *quanta* dos literais tem que resultar no conjunto $\{0, 1, 2\}$. Como o quantum de p_1 é $\{0, 2\}$, e o de p_3 é $\{0, 1\}$, é possível criar a cláusula C_0: $(p_1 \wedge \mathrm{p}_3)$ da teoria P_d.

Cada literal da nova cláusula deve também representar pelo menos uma cláusula de P_c, ou seja, dado um literal p, em uma nova cláusula gerada para P_d, ele deve estar contido em uma cláusula de P_c sem que os outros literais da nova cláusula gerada estejam nessa mesma cláusula de P_c. Aplicando no exemplo anterior, onde a nova cláusula $(p_1 \wedge p_3)$ é criada, p_1 está contido em $(p_1 \vee p_2 \vee \neg p_3)$, onde p_3 não está, e p_3 está contido em $(\neg p_1 \vee p_2 \vee p_3)$, onde p_1 não está.

Caso o literal adicionado na nova cláusula não represente sozinho nenhuma cláusula de P_c ele é redundante, portanto, deve ser eliminado [Bittencourt et al. 2003a] [Bittencourt 2007] [Bittencourt 2008][Bittencourt et al. 2003b].

O algoritmo para encontrar as novas cláusulas da FND P_d é uma busca no espaço de estados, sendo aplicado uma busca A* para encontrar tais estados. Como um algoritmo de busca, temos o estado inicial e o estado final que queremos alcançar, que no caso é uma cláusula que cubra todo o conjunto de coordenadas de P_c, então primeiro a lista de literais é ordenada de modo que os literais com mais coordenadas sejam escolhidos primeiro, de modo a otimizar a busca, já que cobrem uma quantidade maior de cláusulas. Depois disso é criado um conjunto chamado *Gap*, onde ficam guardadas todas as coordenadas que não foram cobertas ainda, e para cada literal candidato a entrar na cláusula conforme é realizada a busca, se em seu *quantum* houver uma ou mais coordenadas que estejam no *Gap*, ou seja, coordenadas que ainda não foram cobertas, esse literal é adicionado à cláusula e seu *quantum* é removido do *Gap*. Um estado é considerado válido se em algum momento o conjunto *Gap* for vazio, e então é adicionado em um conjunto de estados válidos, que representarão as cláusulas de P_d[Bittencourt et al. 2003b].

2.4. Contagem de Modelos

Formas Normais Primárias são os pares de implicantes e implicados primários de uma teoria proposicional. Uma característica importante dessa representação é que, dado uma teoria proposicional, esta teoria gerará apenas um único par de implicantes e implicados primários, entretanto, este par de implicantes e implicados primários representa uma família de teorias proposicionais congruentes, que são equivalentes em relação ao grupo de permutações e complementações.[Bittencourt 2008].

Famílias de teorias proposicionais podem ser classificadas de acordo com o número de símbolos proposicionais e pela sua quantidade de modelos proposicionais. Cada família possui ainda uma população, sendo a população a quantidade de teorias proposicionais congruentes dentro da família[Bittencourt 2007][Polya 1940].

Modelos proposicionais são combinações de valores verdade em uma fórmula proposicional que resulte em *Verdadeiro* para a fórmula, e a contagem de modelos nada mais é do que a quantidade dessas combinações de valores que resultem em *Verdadeiro* para uma fórmula proposicional dada. [Gomes et al. 2006]

O algoritmo recebe uma *FNC* de entrada, que representa uma fórmula P, e então

calcula os implicantes primários de P (IP_P), utilizando o Algoritmo de Transformação Dual.

O próximo passo é determinar as coordenadas disjuntivas dos literais em IP_P, onde essas coordenadas são apenas a representação indicando a quais termos de IP_P cada literal pertence.

O terceiro passo é encontrar o conjunto de termos compatíveis em IP_P, onde dois termos são compatíveis se eles não possuem os mesmos literais com sinais opostos. Esse conjunto pode ser determinado de forma eficiente utilizando as coordenadas disjuntivas.

O quarto passo é uma busca no espaço de estados, onde cada estado é representado por um termo de IP_P, e/ou pela combinação de dois ou mais termos compatíveis em IP_P. Os estados iniciais são os termos em IP_P, e para calcular os sucessores basta determinar os termos compatíveis, usando os conjuntos obtidos no passo 3. Também é necessário assegurar que um termo não esteja contido na união de outros termos. Os estados então são modelados em um grafo, onde os termos são ligados às combinações que eles são compatíveis.

No quinto passo, o número de modelos associado a cada estado é calculado. Se a fórmula possui n símbolos proposicionais, e os termos associados ao estado possui k literais, então o número de modelos associados aquele estado será 2^{n-k}.

O sexto passo consiste em calcular a quantidade de modelos que estão associados exclusivamente a cada estado. Para fazer isso, é subtraído da soma de modelos dos estados que representam um termo a quantidade de modelos dos outros estados formados por aquele termo.

O ultimo passo é somar o total de modelos de cada cláusula e dos estados formados pela combinação de cláusulas compatíveis, resultando na quantidade de modelos da teoria.

3. Desenvolvimento

3.1. Modelagem

Foi desenvolvido um método para detecção de isomorfismo entre grafos utilizando o que foi abordado anteriormente, onde os grafos de entrada serão primeiro avaliados em um pré-processamento, após esse pré-processamento, se o isomorfismo não for descartado logo no início, os grafos passam por uma remodelagem, onde são transformados em fórmulas lógicas. As fórmulas lógicas formadas a partir dos grafos de entrada passam então pelo algoritmo de transformação dual, gerando seus respectivos conjuntos de implicantes primários, por fim, serão contados os modelos de cada um dos conjuntos de implicantes primários, e a partir dessa quantidade de modelos, será interpretado um resultado.

A modelagem adotada cria teorias proposicionais onde os vértices seriam os símbolos proposicionais, e as arestas seriam as cláusulas.

Porém, essa modelagem não considera literais negados, apenas literais puros, e isto implicará em algumas modificações nos algoritmos descritos anteriormente, conforme será visto a seguir.

Inicialmente, são verificados se ambos os grafos possuem o mesmo número de vértices. Caso os grafos não possuam o mesmo número de vértices eles são considerados não isomorfos.

A próxima etapa de pré-processamento é verificar se ambos os grafos possuem o mesmo número de arestas. Caso os grafos não possuam o mesmo número de arestas, eles são considerados não-isomorfos.

A terceira e última etapa de pré-processamento é verificar se os vértices possuem um mapeamento de grau, onde cada vértice v_i de um grafo G_1 deve possuir o mesmo grau de algum vértice v_j em G_2.

O mapeamento dos graus dos vértices pode ser verificado utilizando os conjuntos de coordenadas dos literais das teorias proposicionais fornecidas na entrada do programa. O conjunto de coordenadas nada mais é do que o conjunto de arestas incidentes em um vértice. Para esta verificação basta ordenar a lista de literais de acordo com o tamanho do conjunto de coordenadas de cada literal, e depois verificar se o tamanho do conjunto de coordenadas de um literal, em uma posição da lista de literais de uma teoria, é igual ao tamanho do conjunto de coordenadas de um literal na mesma posição da lista de literais da outra teoria.

Após ser realizado o pré-processamento, a próxima etapa é modelar a teoria proposicional dentro do programa. As clausulas são geradas conforme a entrada de dados, armazenando o literal dentro da cláusula, e registrando no literal a coordenada da cláusula a qual ele pertence.

Após construir a teoria proposicional no programa, o próximo passo é gerar o conjunto de Gap para cada literal.

O próximo passo é ordenar a lista de literais de acordo com o tamanho do Gap dos literais, os literais com menor Gap, ou seja, cobrem a maior quantidade de cláusulas, são escolhidos primeiro.

Após ordenar a lista de literais, é necessário gerar a árvore de estados, a fim de encontrar estados válidos, que representam os Implicantes Primários da teoria proposicional. O primeiro passo para gerar essa árvore de estados é configurar os estados iniciais. Os estados iniciais são compostos por uma lista de símbolos proposicionais, essa lista no estado inicial possui apenas o símbolo do literal do estado inicial. Além da lista de símbolos proposicionais, os estados iniciais possuem uma cláusula, representando o próprio estado. Essa cláusula é composta apenas pelo literal do estado inicial. O último componente dos estados iniciais é o conjunto Gap, obtido através do literal do estado inicial.

Em seguida o conjunto de coordenadas do literal candidato a entrar no estado é iterado. Para cada coordenada do literal, é verificado se esta coordenada está contida no conjunto Gap. Se a coordenada estiver contida no conjunto Gap, é verificado se o literal será inserido no estado. Para o literal ser inserido na lista é feito uma verificação se seu símbolo está contido na lista de símbolos já presentes no estado. Após esta verificação, o literal pode então ser inserido no estado e suas coordenadas são removidas de Gap.

Caso Gap seja um conjunto vazio, então é feita uma verificação se o estado irá virar uma cláusula da nova teoria.

Ao final do algoritmo, ocorre uma chamada recursiva para cada literal restante na lista.

Após a obtenção dos implicantes primários das teorias proposicionais, é feito uma contagem da quantidade de modelos da teoria, a fim de categoriza-la em uma família de teorias, utilizando o algoritmo de contagem de modelos descrito anteriormente, porém, com algumas modificações devido a ausência de literais negados.

Com a quantidade de modelos e a quantidade de símbolos, é possível categorizar a teoria proposicional em sua família de teorias, e se um grafo candidato a par isomorfo passar pelo mesmo processo e for categorizado na mesma família, então os grafos são ditos isomorfos.

3.2. Modificação no Algoritmo de Transformação Dual

Como a modelagem adotada não possui literais com símbolos negados, não é necessário aplicar resolução, sendo apenas aplicada a subsunção quando necessário.

3.3. Modificação no Algoritmo de Cotagem de Modelos

Como na modelagem dos grafos não é possível existir literais negados, nesta implementação do método de contagem de modelos não foi necessário aplicar resolução na combinação de cláusulas para contagem de modelos, implicando algumas facilitações na implementação do algoritmo.

Como não há literais negados, há uma grande diferença na combinação de cláusulas utilizadas na contagem de modelos, e que visam remover propagação no número de modelos, pois existirá apenas uma combinação contendo todas as cláusulas, onde a quantidade de modelos desta combinação propagará por todas as cláusulas.

4. Considerações Finais

4.1. Resultados e Testes

Para realização de testes do método de verificação de isomorfismo entre grafos implementado neste trabalho, foi gerado um *benchmark* para validação de pares de grafos isomorfos, gerando um grafo aleatoriamente, e gerando um grafo isomorfo a partir deste primeiro grafo aplicando uma função que mapeia os vértices do grafo, fazendo com que o segundo grafo seja isomorfo ao primeiro.

Ao todo foram gerados 105 pares de grafos isomorfos, e ao serem submetidos ao programa, todos os 105 pares de grafos isomorfos foram identificados como isomorfos pelo programa.

Para a verificação do não isomorfismo, a geração do benchmark foi feita de forma aleatória. Foi escrito um programa em C++ que gerasse grafos com um tamanho definido, onde vários grafos de mesma quantidade de vértices e arestas foram gerados.

Foram gerados grafos com quantidades de vértices e arestas arbitrárias, sendo gerados no total 10 grafos com 6 vértices e 7 arestas, 25 grafos com 7 vértices e 10 arestas, e 36 grafos com 8 vértices e 12 arestas. Para a execução dos testes, os grafos foram separados em diretórios de acordo com a quantidade de vértices, onde cada grafo foi comparado com o restante dos grafos contidos no diretório, totalizando 45 execuções

para o conjunto de grafos com 6 vértices, 300 execuções para o conjunto de grafos com 7 vértices e 630 execuções para o conjunto de grafos com 8 vértices.

A execução do primeiro conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 39 testes, identificou não-isomorfismo utilizando o método desenvolvido em 5 testes, e identificou erroneamente um de grafos como isomorfos.

A execução do segundo conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 283 testes, identificou não-isomorfismo utilizando o método desenvolvido em 16 testes, e identificou erroneamente um par de grafos como isomorfos.

A execução do terceiro conjunto de grafos identificou não-isomorfismo utilizando os critérios de pré-processamento em 601 testes, identificou não-isomorfismo utilizando o método desenvolvido em 26 testes, e identificou erroneamente 3 pares de grafos como isomorfos.

Nos pares que acusaram de forma incorreta o isomorfismo, é possível notar que em todos havia um vértice com apenas uma aresta, indicando que grafos com esta característica podem ser problemáticos para o método desenvolvido. Como as falhas apareceram em grafos com a mesma característica, pode ser que este método apresente problemas somente nas classes de grafos com esta característica, sendo necessário uma avaliação melhor e outros testes para averiguar se é somente esta classe de grafos que é excluída do conjunto de respostas corretas geradas pelo método.

4.2. Conclusão

O presente trabalho teve como objetivo investigar a relação entre os pares de Implicantes e Implicados primários de teorias proposicionais, geradas a partir de uma modelagem de grafos, com o problema de isomorfismo entre grafos.

A partir disto, foi implementado um método utilizando a linguagem de programação C++, em que o algoritmo de transformação dual é utilizado para obtenção dos Implicantes primários dos grafos que foram modelados em teorias proposicionais. Com os conjuntos de Implicantes primários, foi possível obter a quantidade de modelos de cada teoria proposicional, e com a quantidade de símbolos proposicionais de cada teoria, foi possível categorizá-las e indicar se elas pertenciam a uma mesma família de teorias.

Com o método implementado, foi necessário atestar sua eficácia. O programa obtido foi submetido a testes de isomorfismo, onde eram gerados pares de grafos isomórficos, e o programa identificava corretamente o isomorfismo entre esses pares de grafos utilizando a categorização descrita anteriormente, onde grafos isomorfos pertenciam a mesma família de teorias proposicionais.

O programa foi submetido também a testes de não-isomorfismo, onde foram gerados vários grafos com uma mesma quantidade de vértices e arestas, e foi efetuado um teste comparando todos os grafos em pares. Neste teste, houveram resultados promissores, em que a grande maioria dos testes identificou o isomorfismo e classificou os grafos em famílias de teorias proposicionais diferentes, porém, houveram alguns testes que falharam, e em todos os testes que apresentaram falha os grafos possuíam a mesma ca-

racterística de ter um vértice com apenas uma aresta, sendo necessário uma investigação mais aprofundada do motivo pelo qual somente estes grafos foram classificados na mesma família de teorias proposicionais.

Referências

Bittencourt, G. (2007). Counting models using prime forms. Unpublished.

Bittencourt, G. (2008). Boolean concepts. Unpublished.

Bittencourt, G., Marchi, J., and Padilha, R. (2003a). A syntactic approach to satisfaction.

Bittencourt, G., Marchi, J., and Padilha, R. S. (2003b). A syntactic approach to satisfaction. In Konev, B. and Schimidt, R., editors, *4^{th} Inter. Workshop on the Implementation of Logic (LPAR03)*, pages 18–32. Univ. of Liverpool and Univ. of Manchester.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215.

Gomes, C., Sabharwal, A., and Selman, B. (2006). Model counting: A new strategy for obtaining good bounds. volume 1.

Johnsonbaugh, R. (1986). *Discrete Mathematics revised edition*. New York: Macmillian Publishing Company.

Nievergelt, Y. (2002). *Foundations of Logic and Mathematics*. Springer.

Polya, G. (1940). Sur les types des propositions composées. *The Journal of Symbolic Logic*, 5(3):98–103.

Quine, W. V. (1959). On cores and prime implicants of truth functions. *The American Mathematical Monthly*, 66(9):755–760.

Skiena, S. S. (2008). *The algorithm design manual: Second edition*, volume 1. Springer Science & Business Media, 2nd edition.

APÊNDICE B – Código Fonte

Este anexo contém o código fonte desenvolvido para este trabalho, com exceção dos scripts desenvolvidos para execução do benchmark. Neste anexo, a implementação da contagem de modelos e da transformação dual não sofreram as adaptações descritas no trabalho, tornando possível ser reutilizada para implementações que visem uma modelagem de grafos que utilize símbolos negados. Como não há nenhuma exclusão, apenas adições no código fonte, é possível ser facilmente alterada para a forma descrita no trabalho.

Listing B.1 – Implementação de main.cc

```
#pragma once

#include <iostream>
#include "../dual.h"
#include "../literal.h"

#define ISOMORPH 1
#define NON_ISOMORPH 0
#define DIFF_VERTEX 2
#define DIFF_EDGE 3
#define DIFF_DEGREE 4

using namespace std;

int main() {

  Dual dual1;
  Dual dual2;

  int v1, v2, e1, e2;

  v1 = dual1.getNSymbols();
  v2 = dual2.getNSymbols();

  e1 = dual1.getNClauses();
  e2 = dual2.getNClauses();

  cerr << v1 << "  " << v2 << endl;

  if(v1 != v2) {
    cout << DIFF_VERTEX << endl;
    return 0;
  }
```

```
34
35    if(e1 != e2) {
36      cout << DIFF_EDGE << endl;
37      return 0;
38    }
39    vector<Literal*> lit1 = *dual1.getLiterals();
40    vector<Literal*> lit2 = *dual2.getLiterals();
41
42    for(int i = 0; i < v1; i++) {
43      int n1 = lit1[i]->getGapSize();
44      int n2 = lit2[i]->getGapSize();
45
46      if(n1 != n2) {
47        cout << DIFF_DEGREE << endl;
48        return 0;
49      }
50    }
51    dual1.generateDual();
52    dual2.generateDual();
53
54    if(dual1.countModels() == dual2.countModels()) {
55      cout << ISOMORPH << endl;
56    } else {
57      cout << NON_ISOMORPH << endl;
58    }
59
60    return 0;
61
62  }
```

Listing B.2 – Header da classe Clause

```
1  #pragma once
2
3  #include <vector>
4  #include <iostream>
5  #include "literal.h"
6  #include <string>
7
8  using namespace std;
9
10 class Clause {
11 private:
12   vector<Literal*> literals;
13
14 public:
15   Clause();
16   Clause(vector<Literal*> _literals);
```

```
17    //~Clause();
18    void addLiteral(Literal* literal);
19    vector<Literal*> getLiterals();
20    void printClause();
21    int containsLiteral(Literal *n_l);
22    bool containsSymbol(string st);
23    bool containsContraditory(string st, bool n);
24
25    bool operator==(Clause cl1) {
26      int cont = 0;
27      vector<Literal*> aux = cl1.getLiterals();
28      for(int i = 0; i < aux.size(); i++) {
29
30        cont += this->containsLiteral(aux[i]);
31
32      }
33      if(cont == cl1.getLiterals().size() && cont == this->literals.size())
          return false;
34      return true;
35    }
36
37  };
```

Listing B.3 – Implementação da classe Clause

```
1   #pragma once
2
3   #include <string>
4   #include <vector>
5   #include <iostream>
6   #include <algorithm>
7   #include <tuple>
8   #include "literal.h"
9   #include "quantum.h"
10  #include "clause.h"
11
12  using namespace std;
13
14  Clause::Clause() {
15    //this->literals = new list<Literal*>();
16  }
17
18  Clause::Clause(vector<Literal*> _literals) {
19    this->literals = _literals;
20  }
21  /*
22  Clause::~Clause() {
23    this->literals->clear();
```

```
24   delete this->literals;
25 }*/
26
27 void Clause::addLiteral(Literal* _literal) {
28   this->literals.push_back(_literal);
29 }
30
31 void Clause::printClause() {
32   cout << "{";
33   for(int i = 0; i < this->literals.size(); i++) {
34     this->literals[i]->printLiteral();
35     if (i != this->literals.size()-1) cout << ", ";
36   }
37   cout << "}";
38 }
39
40 vector<Literal*> Clause::getLiterals() {
41   return this->literals;
42 }
43
44 int Clause::containsLiteral(Literal *n_l) {
45   for(int i = 0; i < this->literals.size(); i++) {
46     if(n_l == this->literals[i]) return 1;
47   }
48   return 0;
49 }
50
51 bool Clause::containsSymbol(string st) {
52   for(int i = 0; i < this->literals.size(); i++) {
53     if(st == this->literals[i]->getSymbol()) return true;
54   }
55   return false;
56 }
57
58 bool Clause::containsContraditory(string st, bool n) {
59   for(int i = 0; i < this->literals.size(); i++) {
60     if(st == this->literals[i]->getSymbol() && n != this->literals[i]->
        getNegate()) return true;
61   }
62   return false;
63
64 }
```

Listing B.4 – Header da classe Dual

```
1 #pragma once
2
3 #include <string>
```

```
4  #include <vector>
5  #include "clause.h"
6  #include "literal.h"
7  #include "quantum.h"
8
9  class Dual{
10 private:
11   vector<Clause> clauses;
12   vector<Literal*> *literals;
13   vector<Clause> states;
14   int n_clauses;
15   int n_symbols;
16
17 public:
18   Dual();
19   ~Dual();
20
21   void readFile();
22
23   void setStatesAsClauses();
24
25   // just for testing
26   void printCoordinates();
27   void printClauses();
28   void printGap();
29   void printStates();
30
31   void generateDual();
32   void generateStates(vector<int> n_gap, vector<string> symbols, Clause cl,
         int x, const int size);
33   void generateGap();
34   void sortLiterals();
35
36   vector<Literal *>* getLiterals();
37   int getNSymbols();
38   int getNClauses();
39
40   int countModels();
41 };
```

Listing B.5 – Implementação da classe Dual

```
1  #pragma once
2
3  #include <string>
4  #include <vector>
5  #include <iostream>
6  #include <algorithm>
```

```
7  #include <cmath>
8  #include <cstring>
9  #include "clause.h"
10 #include "literal.h"
11 #include "quantum.h"
12 #include "dual.h"
13
14 using namespace std;
15
16 Dual::Dual() {
17
18   this->literals = new vector<Literal*>();
19   this->readFile();
20   //cout << "archive read" << endl;
21   this->generateGap();
22   //cout << "Gap generated" << endl;
23   this->sortLiterals();
24   //cout << "Literals sorted, read to run dual transform" << endl;
25
26 }
27
28 Dual::~Dual() {
29
30   while(!this->literals->empty()) {
31     delete this->literals->front();
32     this->literals->erase(this->literals->begin());
33   }
34   delete this->literals;
35
36 }
37
38 void Dual::setStatesAsClauses() {
39
40   for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
       literals->end(); ++it) {
41     (*it)->clearCoordinates();
42   }
43   this->clauses.clear();
44
45   for(int i = 0; i < this->states.size(); i++) {
46     vector<Literal *> aux = this->states[i].getLiterals();
47     for(int j = 0; j < aux.size(); j++) {
48       aux[j]->addCoordinate(i);
49     }
50
51     this->clauses.push_back(this->states[i]);
52   }
```

```
53
54    this->n_clauses = this->clauses.size();
55
56    this->states.clear();
57
58    this->generateGap();
59
60    this->sortLiterals();
61
62  }
63
64
65  vector<Literal *>* Dual::getLiterals() {
66    return this->literals;
67  }
68
69  int Dual::getNSymbols() {
70    return this->n_symbols;
71  }
72
73  int Dual::getNClauses() {
74    return this->n_clauses;
75  }
76
77  void Dual::readFile() {
78
79    int n_c, n_s, i;
80    cin >> n_c;
81    cin >> n_s;
82    this->n_clauses = n_c;
83    this->n_symbols = n_s;
84
85    string delimiter = " ";
86    string aux = "";
87    for(i = 0; i < this->n_clauses; i++) {
88      size_t pos = 0;
89      getline(cin, aux);
90      if(aux.empty()) {
91        i--;
92        continue;
93      }
94      string token;
95      Clause new_clau;
96
97      while((pos = aux.find(delimiter)) != string::npos) {
98        token = aux.substr(0, pos);
99
```

```
100       bool negate = false;
101       if(token[0] == '~') {
102         negate = true;
103         token = token.substr(1, pos);
104       }
105
106       Literal *n_lit;
107
108       bool control = true;
109
110       for(vector<Literal*>::iterator it = this->literals->begin(); it !=
              this->literals->end(); ++it) {
111         if ((*it)->getNegate() == negate && token == (*it)->getSymbol()) {
112           n_lit = (*it);
113           control = false;
114           break;
115         }
116       }
117
118
119       if(control) {
120         n_lit = new Literal(token, negate);
121         this->literals->push_back(n_lit);
122       }
123
124       n_lit->addCoordinate(i);
125       new_clau.addLiteral(n_lit);
126
127       aux.erase(0, pos+delimiter.length());
128     }
129     this->clauses.push_back(new_clau);
130   }
131 }
132
133 void Dual::printCoordinates() {
134
135   for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
          literals->end(); ++it) {
136     (*it)->printLiteral();
137     cout << endl;
138     (*it)->printCoordinates();
139   }
140
141 }
142
143 void Dual::printGap() {
144
```

```
145   cout << "printing gap" << endl;
146
147   for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
          literals->end(); ++it) {
148     cout << "Gap of ";
149     (*it)->printLiteral();
150     cout << ":" <<endl;
151     (*it)->printGap();
152     cout << endl;
153   }
154
155 }
156
157 void Dual::printClauses(){
158   cout << "printing clauses" << endl;
159   for(int i = 0; i < this->clauses.size(); i++) {
160     cout << "C" << i << ": ";
161     this->clauses[i].printClause();
162     cout << endl;
163   }
164   cout << endl;
165
166 }
167
168 void Dual::printStates() {
169
170   cout << "printing states" << endl;
171   for(int i = 0; i < this->states.size(); i++) {
172     cout << "C" << i << ": ";
173     this->states[i].printClause();
174     cout << endl;
175   }
176   cout << endl;
177
178 }
179
180 void Dual::generateGap() {
181
182   for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
          literals->end(); ++it) {
183     vector<int> aux1 = (*it)->getQuantum().getCoordinates();
184     vector<int> aux2;
185
186     for(int j = 0; j < this->n_clauses; j++) {
187       aux2.push_back(j);
188     }
189
```

```
190     for(int j = 0; j < aux1.size(); j++) {
191       aux2.erase(std::remove(aux2.begin(), aux2.end(), aux1[j]), aux2.end()
              );
192     }
193     (*it)->addGap(aux2);
194   }
195 }
196
197 void Dual::sortLiterals() {
198   std::sort(this->literals->begin(), this->literals->end(),
199     [](Literal *lit1, Literal *lit2) {
200     if(lit1->getGapSize() == lit2->getGapSize())
201       return lit1->getGapSize() < lit2->getGapSize();
202
203     return lit1->getGapSize() < lit2->getGapSize();
204   });
205 }
206
207
208 void Dual::generateDual() {
209
210   //cout << "Generating dual clauses" << endl;
211
212   const int size = this->literals->size();
213   int i = 1;
214
215   for(vector<Literal*>::iterator it = this->literals->begin(); it != this->
        literals->end(); ++it) {
216     vector<string> symbols;
217     symbols.push_back((*it)->getSymbol());
218     vector<int> new_gap = (*it)->getGap();
219     Clause n_clause;
220     n_clause.addLiteral((*it));
221     for(int j = i; j < this->literals->size(); j++) {
222       this->generateStates(new_gap, symbols, n_clause, j, size);
223     }
224     i++;
225   }
226
227
228   //cout << "Dual clauses created" << endl;
229
230 }
231
232 void Dual::generateStates(vector<int> n_gap, vector<string> symbols, Clause
      n_clause, int x, const int size) {
233
```

```
234   if(x == size) return;
235   if(n_gap.size() == 0) return;
236
237   //prologue
238
239   vector<int> new_gap = n_gap;
240   vector<string> n_symbols = symbols;
241
242   Clause new_clause = n_clause;
243   vector<int> removed_coord;
244
245   vector<Literal*>::iterator it;
246
247   bool flag_remove = false; // flag to check if coordinate need to be
          removed from gap
248   this->generateStates(new_gap, n_symbols, new_clause, x+1, size);
249
250   it = this->literals->begin();
251   advance(it, x);
252
253   vector<int> coordinates = (*it)->getQuantum().getCoordinates();
254
255   for(int i = 0; i < coordinates.size(); i++) {
256     bool flag_gap = (find(new_gap.begin(), new_gap.end(), coordinates[i])
          != new_gap.end());
257     if(flag_gap) {
258       bool flag_symbol = (find(n_symbols.begin(), n_symbols.end(), (*it)->
            getSymbol()) != n_symbols.end());
259       if(!flag_symbol && !flag_remove) {
260         new_clause.addLiteral((*it));
261         n_symbols.push_back((*it)->getSymbol());
262         flag_remove = true;
263       }
264       if(flag_remove) {
265         removed_coord.push_back(coordinates[i]);
266       }
267     }
268   }
269
270
271   for(int i = 0; i < removed_coord.size(); i++) {
272     new_gap.erase(std::remove(new_gap.begin(), new_gap.end(), removed_coord
          [i]), new_gap.end());
273   }
274
275
276   if(new_gap.size() == 0) {
```

```
277     if(this->states.empty()) {
278       this->states.push_back(new_clause);
279       return;
280     }
281     bool check_clause = true;
282     int new_clause_size = new_clause.getLiterals().size();
283
284     for(int i = 0; i < this->states.size(); i++) {
285       int flag_nc = 0;
286       int size_cl = this->states[i].getLiterals().size();
287
288       vector<Literal*> aux_literal = this->states[i].getLiterals();
289       for(int j = 0; j < aux_literal.size(); j++) {
290         flag_nc += new_clause.containsLiteral(aux_literal[j]);
291       }
292       if(flag_nc == size_cl) {
293         if(size_cl < new_clause_size) {
294           check_clause = false;
295         }
296       }
297
298       if(new_clause_size == flag_nc) {
299
300         if(flag_nc >= size_cl) {
301           check_clause = false;
302           break;
303         }
304         Clause aux_clause;
305
306         if(new_clause_size < size_cl) {
307
308           /*
309              * subsuming the clause in list
310              * if the checked clause contains the new clause, and new
                    clause size < checked clause size
311              * then subsume checked clause
312           */
313
314           aux_clause = this->states[i];
315           this->states.erase(this->states.begin()+i);
316           //this->states.erase(std::remove(this->states.begin(), this->
                states.end(), aux_clause), this->states.end());
317           this->states.push_back(new_clause);
318           return;
319         }
320       }
321     }
```

```
322
323       if(check_clause) {
324         this->states.push_back(new_clause);
325       }
326       return;
327     }
328     int k = x;
329     for(; it != this->literals->end(); ++it) {
330       k++;
331       this->generateStates(new_gap, n_symbols, new_clause, k, size);
332     }
333
334 }
335
336 /*
337   This algorithm to count models is a simplyfied version, once the way we
        are modeling graphs,
338   we could never have contraditory symbols in a pair of clauses.
339 */
340 int Dual::countModels() {
341
342   int n_st = this->states.size();
343   int models[n_st][n_st] = {};
344
345
346   int cls[n_st] = {0};
347   int values[n_st] = {0};
348
349   for(int i = 0; i < n_st; i++) {
350     int st_size = this->states[i].getLiterals().size();
351     cls[i] = pow(2, this->n_symbols - st_size);
352   }
353
354   for(int i = 0; i < n_st; i++) {
355     vector<Literal *> aux = this->states[i].getLiterals();
356     models[i][i] = 1;
357     for(int j = i+1; j < n_st; j++) {
358       vector<Literal *> temp;
359       bool flag_contraditory = false;
360       for(int k = 0; k < aux.size(); k++) {
361         if(this->states[j].containsContraditory(aux[k]->getSymbol(), aux[k
              ]->getNegate())) {
362           flag_contraditory = true;
363           break;
364         }
365       }
366       if(flag_contraditory) continue;
```

```
367
368       temp = this->states[j].getLiterals();
369
370       int add[temp.size()];
371       for(int k = 0; k < temp.size(); k++) {
372         add[k] = 1;
373       }
374       for(int k = 0; k < aux.size(); k++) {
375         for(int m = 0; m < temp.size(); m++) {
376           if(aux[k]->getSymbol() == temp[m]->getSymbol()) {
377             add[m] = 0;
378           }
379         }
380       }
381
382       for(int k = 0; k < temp.size(); k++) {
383         if(add[k]) aux.push_back(temp[k]);
384       }
385
386       models[i][j] = 1;
387     }
388     int result = pow(2, this->n_symbols - aux.size());
389
390     values[i] = result;
391   }
392   /*
393   for(int i = 0; i < n_st; i++) {
394     for(int j = 0; j < n_st; j++) {
395       cout << models[i][j] << " ";
396     }
397     cout << endl;
398   }*/
399
400   for(int i = 1; i < n_st; i++) {
401     for(int j = 0; j < i; j++) {
402       bool flag = true;
403       for(int k = 0; k < n_st; k++) {
404         int aux = models[i][k] && models[j][k];
405         if(aux != models[i][k]) {
406           flag = false;
407           break;
408         }
409       }
410       if(flag) {
411
412         for(int k = 0; k < n_st; k++) {
413           models[i][k] = 0;
```

```
414           }
415           values[i] = 0;
416         }
417       }
418     }
419     /*
420     for(int i = 0; i < n_st; i++) {
421       for(int j = 0; j < n_st; j++) {
422         cout << models[i][j] << " ";
423       }
424       cout << endl;
425     }*/
426
427     for(int i = 0; i < n_st; i++) {
428       for(int j = 0; j < n_st; j++) {
429         if(models[i][j]) {
430           cls[j] -= values[i];
431         }
432       }
433     }
434     int n_models = 0;
435     for(int i = 0; i < n_st; i++) {
436       //cout << "values: " << values[i] << endl;
437       n_models += cls[i];
438       n_models += values[i];
439     }
440
441     //cout << "Number of models: " << n_models << endl;
442     return n_models;
443
444 }
```

Listing B.6 – Header da classe Literal

```
1  #pragma once
2
3  #include <string>
4  #include <vector>
5  #include "quantum.h"
6
7  using namespace std;
8
9  class Literal {
10 private:
11   string symbol;
12   bool negate;
13   Quantum quant;
14   vector<int> gap;
```

```
15    int gap_size;
16
17  public:
18    Literal();
19    Literal(string _symbol, bool _negate);
20    void changeLiteral(string _symbol);
21    void changeNegate(bool _negate);
22    string getSymbol();
23    bool getNegate();
24    void printLiteral();
25    void addCoordinate(int coord);
26    void printCoordinates();
27    void addGap(vector<int> gap);
28    void printGap();
29    int getGapSize();
30    void clearCoordinates();
31    vector<int> getGap();
32    Quantum getQuantum();
33  };
```

Listing B.7 – Implementação da classe Literal

```
1   #pragma once
2
3   #include <string>
4   #include <iostream>
5   #include <vector>
6   #include "literal.h"
7   #include "quantum.h"
8
9   using namespace std;
10
11  Literal::Literal() {
12    this->symbol = "";
13    this->negate = false;
14    this->gap_size = 0;
15  }
16
17  Literal::Literal(string _symbol, bool _negate) {
18    this->symbol = _symbol;
19    this->negate = _negate;
20  }
21
22  void Literal::changeLiteral(string _symbol) {
23    this->symbol = _symbol;
24  }
25
26  void Literal::changeNegate(bool _negate) {
```

```
27    this->negate = _negate;
28  }
29
30  string Literal::getSymbol() {
31    return this->symbol;
32  }
33
34  bool Literal::getNegate() {
35    return this->negate;
36  }
37
38  void Literal::printLiteral() {
39    if(this->negate) {
40      cout << "~" ;
41    }
42    cout << this->symbol;
43  }
44
45  void Literal::addCoordinate(int coord) {
46    this->quant.addCoordinate(coord);
47  }
48
49  void Literal::printCoordinates() {
50    this->quant.printCoordinates();
51  }
52
53  void Literal::addGap(vector<int> _gap) {
54    this->gap = _gap;
55    this->gap_size = this->gap.size();
56  }
57
58  void Literal::printGap() {
59    cout << "{";
60    for(int i = 0; i < this->gap.size(); i++) {
61      cout << this->gap[i];
62      if(i != this->gap.size()-1) cout << ", ";
63    }
64    cout << "}"<<endl;
65  }
66
67  int Literal::getGapSize() {
68    return this->gap_size;
69  }
70
71  vector<int> Literal::getGap() {
72    return this->gap;
73  }
```

```
74
75 Quantum Literal::getQuantum() {
76   return this->quant;
77 }
78
79 void Literal::clearCoordinates() {
80   this->quant.clearCoordinates();
81   this->gap.clear();
82 }
```

Listing B.8 – Header da classe Quantum

```
1  #pragma once
2
3  #include <vector>
4
5  using namespace std;
6
7  class Quantum {
8    private:
9      vector<int> coordinates;
10
11   public:
12     Quantum();
13     Quantum(vector<int> coord);
14     void addCoordinate(int coord);
15     //void removeCoordinate(int coord);
16     vector<int> getCoordinates();
17     void printCoordinates();
18     void clearCoordinates();
19 };
```

Listing B.9 – Implementação da classe Quantum

```
1  #pragma once
2
3  #include "quantum.h"
4  #include <vector>
5  #include <iostream>
6
7  using namespace std;
8
9  Quantum::Quantum() {}
10
11 Quantum::Quantum(vector <int> coord) {
12   this->coordinates = coord;
13 }
14
```

```
15 void Quantum::addCoordinate(int coord) {
16   this->coordinates.push_back(coord);
17 }
18
19 /*
20 void Quantum::removeCoordinate(int coord) {
21   this->coordinates.remove(coord);
22 }*/
23
24 vector<int> Quantum::getCoordinates() {
25   return this->coordinates;
26 }
27
28 void Quantum::printCoordinates() {
29   for(int i = 0; i < this->coordinates.size(); i++) {
30     cout << this->coordinates[i] << " ";
31   }
32   cout << endl;
33 }
34
35 void Quantum::clearCoordinates() {
36   this->coordinates.clear();
37 }
```

Listing B.10 – Implementação do gerador de grafos isomorfos

```
1  #pragma once
2
3  #include <iostream>
4  #include <cstdlib>
5  #include <vector>
6  #include <ctime>
7
8  #define MAXV 11
9  #define MINV 4
10
11 using namespace std;
12
13 int main() {
14   srand(time(0));
15   int v, e;
16   int i, f;
17   //cout << "informe o n mero do primeiro arquivo" << endl;
18   //cin >> i;
19   //cout << "informe o n mero do arquivo final" << endl;
20   //cin >> f;
21
22   v = rand()%MAXV+MINV;
```

```
23
24  int graph1[v][v] = {0};
25  int graph2[v][v] = {0};
26
27  e = rand()%((v-1)*(v-1));
28
29  int cont = 0;
30  while(cont < e) {
31    int i = rand()%v;
32    int j = rand()%v;
33
34    if(i != j) {
35      if(graph1[i][j] == 0) {
36        graph1[i][j] = 1;
37        graph1[j][i] = 1;
38        cont++;
39      }
40    }
41  }
42
43  vector<int> vec1;
44  for(int i = 0; i < v; i++) {
45    vec1.push_back(i);
46  }
47  vector<int> vec2;
48  vector<int> f_index;
49
50  while(!vec1.empty()) {
51    int aux = rand()%vec1.size();
52    vec2.push_back(vec1[aux]);
53    vec1.erase(vec1.begin()+aux);
54  }
55
56  for(int i = 0; i < v; i++) {
57    for(int j = i+1; j < v; j++) {
58      int a, b;
59      a = vec2[i];
60      b = vec2[j];
61      graph2[a][b] = graph1[i][j];
62      graph2[b][a] = graph1[j][i];
63    }
64  }
65
66  // printing graphs
67
68  cout << e  << " " << v;
69  for(int i = 0; i < v; i++) {
```

```
70      for(int j = i+1; j < v; j++) {
71        if(graph1[i][j] == 1) {
72          cout << endl << "p" << i << " p" << j << " |";
73        }
74      }
75    }
76    cout << endl << e << " " << v;
77
78    for(int i = 0; i < v; i++) {
79      for(int j = i+1; j < v; j++) {
80        if(graph2[i][j] == 1) {
81          cout << endl << "p" << i << " p" << j << " |";
82        }
83      }
84    }
85  }
```

Listing B.11 – Implementação do gerador de grafos

```
1  #pragma once
2
3  #include <iostream>
4  #include <cstdlib>
5  #include <vector>
6  #include <ctime>
7
8  #define MAXV 11
9  #define MINV 4
10
11 #define VERTEX 6
12 #define EDGES 7
13
14 using namespace std;
15
16 int main() {
17   srand((unsigned int)time(NULL));
18   int v, e;
19
20   v = VERTEX;
21   e = EDGES;
22
23   int graph1[v][v];
24
25   for(int i = 0; i < v; i++) {
26     for(int j = 0; j < v; j++) {
27       graph1[i][j] = 0;
28     }
29   }
```

```
30
31    int cont = 0;
32    while(cont < e) {
33      int i = rand()%v;
34      int j = rand()%v;
35
36      if(i != j) {
37        if(graph1[i][j] == 0) {
38          graph1[i][j] = 1;
39          graph1[j][i] = 1;
40          cont++;
41        }
42      }
43    }
44
45
46
47    cout << e  << " " << v;
48    for(int i = 0; i < v; i++) {
49      for(int j = i+1; j < v; j++) {
50        if(graph1[i][j] == 1) {
51          cout << endl << "p" << i << " p" << j << " |";
52        }
53      }
54    }
55
56  }
```