UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS SOBRAL
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO - PPGEEC

ALEXANDRE MARQUES ALBANO DA SILVEIRA

PROVA DE CONHECIMENTO NULO BASEADA EM ISOMORFISMO DE SUBGRAFOS

SOBRAL - CE
2015

ALEXANDRE MARQUES ALBANO DA SILVEIRA

PROVA DE CONHECIMENTO NULO BASEADA EM ISOMORFISMO DE SUBGRAFOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Engenharia da Computação da Universidade Federal do Ceará, campus Sobral, como requisito parcial para obtenção do título de Mestre.

SOBRAL - CE
2015

ALEXANDRE MARQUES ALBANO DA SILVEIRA

PROVA DE CONHECIMENTO NULO BASEADA EM ISOMORFISMO DE SUBGRAFOS

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia Elétrica e de Computação, da Universidade Federal do Ceará, Campus Sobral, como requisito parcial para a obtenção do grau de Mestre em Engenharia Elétrica e de Computação.
Área de concentração: Algoritmos e Computação Distribuída.

Aprovada em: 03/12/2015.

BANCA EXAMINADORA

Prof. Dr. José Cláudio do Nascimento (Orientador)
Universidade Federal do Ceará - UFC

Prof. Dr. Iális Cavalcante de Paula Júnior
Universidade Federal do Ceará - UFC

Prof. Dr. Rodrigo de Melo Souza Véras
Universidade Federal do Piauí – UFPI

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Pós-Graduação em Engenharia - BPGE

S586p Silveira, Alexandre Marques Albano da.
Prova de conhecimento nulo baseada em isomorfismo de subgrafos / Alexandre Marques Albano da Silveira. – 2015.
71 f. : il. color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Programa de Pós-Graduação em Engenharia Elétrica e de Computação, Sobral-CE, 2015.
Área de Concentração: Sistemas de informação.
Orientação: Prof. Dr. José Cláudio do Nascimento.

1. Engenharia elétrica. 2. Isomorfismos (Matemática). 3. Teoria dos grafos. 4. Gerador de subgrafos. I. Título.

CDD 621.3

A Deus, família, professor José Cláudio, demais professores, amigos e colegas.

AGRADECIMENTOS

Ao Prof. Dr. José Cláudio do Nascimento, pela excelente orientação.

Aos professores participantes da banca examinadora, Prof. Dr. Ialis Cavalcante de Paula Júnior e Prof. Dr. Rodrigo de Melo Souza Véras pelo tempo e valiosas sugestões.

À minha mãe pelo apoio e incentivo.

À minha família pela compreensão e apoio.

À Profa. Dr. Andréa Carneiro Linhares pela importante participação e ajuda prestada.

Aos demais professores que fazem parte do programa.

"Pouco conhecimento faz com que as pessoas se sintam orgulhosas. Muito conhecimento, que se sintam humildes."

Leonardo da Vinci

RESUMO

Sabe-se que o problema de isomorfismo de grafos tem um perfeito sistema de prova de conhecimento nulo, mas uma análise de segurança para saber se o problema de isomorfismo de subgrafo satisfaz as três condições de sistema de prova de conhecimento nulo ainda não foi proposta. Neste trabalho foi realizada uma análise de sistema de prova de conhecimento nulo baseado em isomorfismo de subgrafos, que é um problema mais árduo de se resolver computacionalmente por estar na classe $\mathcal{NP}$-completo.

Neste trabalho foi verificado que o sistema de prova de conhecimento nulo baseado em isomorfismo de subgrafos satisfaz as três condições: completude, validade e perfeita prova de conhecimento nulo. Também foi apresentado um gerador de subgrafos isomorfos que cria instâncias de grafos e subgrafos com alta probabilidade de não serem uma instância que possa ser resolvida em tempo polinomial pelos algoritmos conhecidos atualmente.

ABSTRACT

It is known that the problem of graph isomorphism has a perfect zero-knowledge proof system, but a security analysis to determine if the problem of subgraph isomorphism satisfies the three conditions of zero-knowledge proof system has not yet been proposed. In this work was done an analysis of zero knowledge proof system based on subgraphs isomorphism, which is a harder problem to solve computationally to be in class $\mathcal{NP}$-complete.

In this paper it's verified that the zero-knowledge proof system based on subgraphs isomorphism satisfies the three conditions: completeness, validity and perfect zero-knowledge proof. It was also submitted an isomorphic subgraphs generator that creates instances of graphs and subgraphs with high probability of not being an instance that can be solved in polynomial time by algorithms currently known.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

Capítulo 1

Introdução

Em 1976, os fundamentos da criptografia assimétrica foram lançados [1]. No lugar de a segurança dos sistemas criptográficos ser sustentada na forma de sigilo que o texto tinha quando cifrado [2], esta passou a ser fundamentada na dificuldade computacional de se decifrar a mensagem.

Atualmente o algoritmo de criptografia assimétrica mais popular da internet é o RSA. Basicamente esse algoritmo é construído sobre a dificuldade computacional de se fatorar números inteiros grandes em seus fatores primos [3]. O conceito elementar da criptografia assimétrica é a função alçapão. Nessa função é fácil de calcular num sentido, mas acredita-se ser difícil de se calcular em sentido oposto (encontrando o seu inverso) sem uma informação especial. O conceito de função alçapão usado na criptografia assimétrica não é aplicado somente para cifrar mensagens, mas para assinaturas digitais [4], identificação [5], distribuição de chaves [1], votação eletrônica [6] e dinheiro eletrônico [7]. Por isso, esse método é tão popular, pois tendo-se apenas um par de chaves, a pública e a secreta, pode-se realizar muitos tipos de comunicações e transações na rede com segurança. Mas, recentemente, foram testadas milhões de chaves públicas X.509 coletadas na web, e encontraram uma frequência assustadoramente alta de chaves privadas RSA duplicadas [8], mostrando que o processo de geração aleatória das chaves do RSA tem sido falho na prática. Enquanto a comunidade científica aguarda a solução desse problema, outros métodos que foram considerados inviáveis no passado merecem ser revistos.

Um algoritmo que é peça fundamental para muitos sistemas criptográficos é a prova de conhecimento nulo [5–7]. Uma prova de conhecimento é um método pelo qual uma das partes (o provador) pode provar à outra parte (o verificador) que uma declaração é verdadeira, sem transmitir qualquer informação adicional além da veracidade da afirmação. Através do trabalho [9] é bem conhecido que existem provas de conhecimento nulo para qualquer problema $\mathcal{NP}$, desde que as funções alçapão existam. Ainda em [9], são mostrados sistemas de prova de conhecimento nulo que não necessitam de mensagens cifradas. Estes sistemas são baseados

no problema de isomorfismo e não-isomorfismo de grafos. Como não se conhecia a mesma quantidade de propriedades nos problemas de isomorfismo de grafos que se conhecia nos problemas baseados em aritmética modular, com o passar do tempo poucos artigos foram surgindo e o problema de isomorfismo de grafos não chegou a ser empregado na prática.

Outro motivo que favoreceu para o não uso do problema de isomorfismo de grafos foi o desconhecimento da classe de complexidade computacional a que este pertenceria, pois o mesmo não estava na classe dos problemas polinomiais nem na classe $\mathcal{NP}$-completo. Por isso foi criada uma classe IG para investigar somente esse problema [10, 11]. Depois, surgiram algoritmos que reduziram significativamente a complexidade da solução, mas o problema não ainda não foi resolvido em tempo polinomial [12], [13]. Então, puseram sob suspeita os sistemas de prova de conhecimento nulo baseados no problema de isomorfismo de grafos.

Este trabalho contem duas contribuições. Como primeira contribuição foi proposto o uso do problema de isomorfismo de subgrafos (SGI) para a construção de um algoritmo de prova de conhecimento nulo [14] de forma a elevar o nível de segurança dos sistemas de prova de conhecimento nulo. Sabe-se que o problema de isomorfismo de grafos tem um perfeito sistema de prova de conhecimento, mas uma análise de segurança para saber se o problema de isomorfismo de subgrafo (SGI) satisfaz as três condições de sistema de prova de conhecimento nulo não tinha sido proposta. Se o problema de isomorfismo de grafos fosse mais geral que o SGI, esta análise não seria necessária, porém o problema SGI é mais geral. Assim, foi revisado o trabalho apresentado em [9] que propõe um sistema de prova de conhecimento nulo baseado em isomorfismo de grafos para checar se SGI também satisfaz as condições de completude, validade e perfeita prova de conhecimento nulo. A segunda contribuição foi a criação e análise de um gerador de instâncias de grafos e subgrafos. O gerador proposto encontra com alta probabilidade instâncias de pares grafo e subgrafos em que encontrar o isomorfismo entre eles é um problema árduo. Para comprovar isso, foram feitos testes experimentais da dificuldade de solucionar o isomorfismo dos grafos gerados usando o algoritmo mais eficiente já proposto para solucionar esse problema até então. Além disso uma análise estatística dos dados obtidos foi realizada para concluir que a medida que aumenta o número de vértices, então a probabilidade de sucesso para encontrar o isomorfismo entre os pares cai significativamente

Capítulo 2

Problema de Isomorfismo de Subgrafos

2.1 Preliminares sobre complexidade de problemas

Linguagens formais são mecanismos para a representação e especificação de linguagens. Elas podem ser representadas de maneira finita e precisa através de sistemas com sustentação matemática. O alfabeto é um conjunto finito e não vazio de símbolos, e suas combinações formam um conjunto infinito de palavras. Será considerado $\Sigma = \{0, 1\}$ como sendo o alfabeto binário. Uma palavra sobre o alfabeto Σ é uma sequência finita de símbolos de Σ. O comprimento de uma palavra w sobre Σ, descrito por $|w|$, é o número de símbolos presentes em w. Em particular, a palavra vazia, denotada por ϵ, não contém ocorrência de símbolos. Além disso, quando o comprimento de uma palavra w é indefinido, dizemos que $w \in \Sigma^*$. Um subconjunto $L \subseteq \Sigma^*$ é dito uma linguagem de Σ^* por possuir um conjunto de palavras de Σ^* que podem ser reconhecidas por uma dado algoritmo.x

Alan Turing (1912-1954) propôs um modelo de máquina conhecida como Máquina Universal, também conhecida como Máquina de Turing, considerando apenas os aspectos lógicos para a solução de problemas (memória, estados e transições). Turing construiu esse modelo para provar que não existe um algoritmo universal capaz de detectar proposições indecidíveis em um sistema axiomático [15].

Para medir a eficiência de um algoritmo é necessário usar o tempo teórico que o programa leva para encontrar uma resposta em função dos dados de entrada. Esse cálculo é feito associando-se uma unidade de tempo para cada transição que o algoritmo executa. Se a dependência do tempo com relação aos dados de entrada for polinomial com o comprimento da entrada $p(|x|)$, o programa será considerado rápido. Caso a dependência do tempo seja exponencial com o comprimento da entrada ($a^{|x|}$ para $a > 0$), então o programa é considerado lento. Verifica-se que

$$\lim_{|x|\to\infty} \frac{a^{|x|}}{|p(|x|)|} = \infty. \tag{2.1}$$

A noção de computação em tempo polinomial foi introduzida por Cobham [16] e Edmonds [17] como parte do desenvolvimento da teoria da complexidade computacional na década de 60. Embora, em 1953, Von Neumann tenha distinguido entre algoritmos de tempo polinomial e algoritmos de tempo exponencial [18]. Nessa época, foram encontrados muitos algoritmos que resistiam a uma simplificação polinomial. Então, Stephen Cook [19] observou um fato simples: se um problema pode ser resolvido em tempo polinomial, então pode-se também verificar se uma possível solução é correta em tempo polinomial (diz-se que o algoritmo pode ser certificado em tempo polinomial). Uma maneira mais formal de expressar essa ideia, é através da descrição do problema de decisão. Um problema de decisão pode ser visto como um problema de reconhecimento de linguagem. Considere Σ^* como sendo o conjunto de todas as possíveis entradas para um problema de decisão. Considere $L \subseteq \Sigma^*$ como sendo o conjunto de todas as entradas cuja resposta é sim. Esse conjunto é chamado de linguagem correspondente ao problema. Em [19], Cook definiu duas classes de problemas quanto à complexidade destes.

Definição 1. $(\textit{Classe de complexidade } \mathcal{P})$ *- Dizemos que uma linguagem L está na classe de complexidade $\mathcal{P}$, se L é reconhecível em tempo polinomial determinístico, e se existe uma máquina de Turing M e um polinômio $p(\cdot)$ tal que:*

- *Tendo uma string x como entrada, a máquina tem parada após $p(|x|)$ passos;*
- *$M(x) = 1$, se somente se, $x \in L$.*

Entretanto, existiam problemas que não admitiam uma simplificação polinomial no seu tempo de execução, mas que podiam ser certificados em tempo polinomial. Assim, Cook ainda introduziu a definição de algoritmos *não determinísticos em tempo polinomial*. A classe dos problemas $\mathcal{NP}$ é aquela para a qual apenas pode-se verificar, em tempo polinomial, se uma possível solução é correta. Formalmente, a classe $\mathcal{NP}$ é definida da seguinte maneira:

Definição 2. $(\textit{Classe de complexidade } \mathcal{NP})$ *- Uma linguagem L está em $\mathcal{NP}$ se existe uma relação Booleana $R_L \subseteq \Sigma^* \times \Sigma^*$ que pode ser reorganizada em tempo polinomial determinístico e um polinômio $p(\cdot)$ de forma que:*

- *$x \in L$, se somente se, existe um $y \in \Sigma^*$ tal que $(x, y) \in R_L$ e $|y| \leq p(|x|)$;*
- *$x \notin L$, se somente se, $(x, y) \notin R_L$ para todo $y \in \Sigma^*$.*

Enquanto uma máquina de Turing determinística possui um único "caminho de computação" a ser seguido, uma máquina de Turing não determinística possui uma "árvore" deles como opções de computação. Se qualquer ramo da árvore termina em uma condição de aceitação, diz-se que a máquina de Turing não determinística aceita a entrada. Uma indagação sobre o cálculo de tais máquinas é: como uma máquina não determinística sabe qual dessas

ações ela deve tomar? Há duas maneiras de olhar essa questão. Uma é supor que a máquina sempre escolherá uma transição que eventualmente leve a um estado de aceitação. Esse caso pode ser calculado deterministicamente por máquinas de Turing determinísticas, no entanto, ela precisa de uma entrada auxiliar que lhe indique o caminho. Por isso, a verificação de um problema pode ser feita em tempo polinomial determinístico. A outra maneira é imaginar que a máquina se ramifica em muitas cópias, na qual leva a diferentes possíveis transições. Obviamente $\mathcal{P} \subseteq \mathcal{NP}$, pois a classe $\mathcal{P}$ representa um caso particular de $\mathcal{NP}$. Mas o contrário, $\mathcal{NP} \subseteq \mathcal{P}$, no que implica que $\mathcal{P} = \mathcal{NP}$, não tem sua validade provada. Esse trabalho considera a conjectura de que $\mathcal{P} \neq \mathcal{NP}$.

Algoritmos para transportar de uma linguagem L a outra linguagem L' podem ser criados. Por isso, se existe uma função $f(x)$ computável em tempo polinomial no comprimento da entrada, $p(|x|)$, tal que $x \in L$ se somente se $f(x) \in L'$, então diz-se que L é *polinomialmente redutível* para uma linguagem L'. Nesse caso, simplesmente expressamos por $L \geq_p L'$ (lê-se L é p-redutível a L'). Leonid Levin [20] e Stephen Cook [19] observaram que dentre os problemas $\mathcal{NP}$ existem alguns que são mais difíceis do que outros, no sentido de que, resolvendo um desses problemas em tempo polinomial, então todos os problemas em $\mathcal{NP}$ também podem ser resolvidos em tempo polinomial. Assim, a classe dos problemas $\mathcal{NP}$-completos é o subconjunto dos mais difíceis problemas não-determinísticos polinomiais.

Definição 3. $(\textit{Classe de problemas } \mathcal{NP}\textit{-completo})$ *- Uma linguagem L é $\mathcal{NP}$-completo se:*

- *L está em $\mathcal{NP}$, e*
- *cada linguagem $L' \in \mathcal{NP}$ é $L' \geq_p L$.*

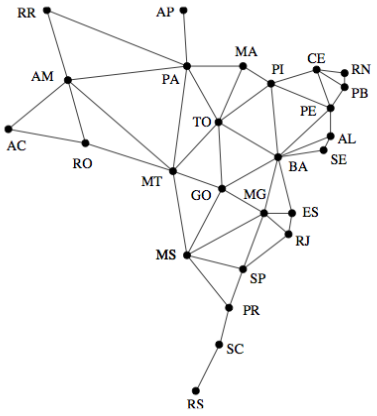
2.2 Grafo

É uma forma de representar e modelar abordagens matemáticas do mundo real, sendo muitos dos problemas computacionais representados por grafos. São usados para representar organogramas, mapas, trajetos, etc. Modelar com grafos permite uma melhor visualização do problema e uma representação matemática possível de ser tratada por computador. Um grafo é uma estrutura $G = (V, A)$ em que V é um conjunto discreto e A é uma família cujos elementos (não vazios) são definidos em função dos elementos de V. Os elementos de V são chamados vértices, nós ou pontos e o valor n = |V| é a ordem do grafo. Uma família A pode ser entendida como uma relação ou conjunto de relações de adjacência, cujos elementos são chamados em geral de ligações; em particular, nas estruturas não orientadas, os $e \in A$ são conhecidos como arestas e, nas orientadas, como arcos. Dois vértices que participam de uma ligação são ditos adjacentes. Um grafo $G = (V, A)$ consiste em um conjunto de elementos

chamados vértices $V = \{v_1, v_2, \cdots, v_n\}$ e um conjunto de pares de vértices chamados arestas, $(v_i, v_j) \in A$ para $i, j = 1, 2, \cdots, n$. O valor $m = |A|$ é considerado por alguns autores como sendo o tamanho do grafo. Se $m = 0$, o grafo é dito trivial. Neste trabalho será abordado apenas grafos finitos e simples, ou seja, com numero finito de arcos e arestas, cada aresta não pode ter como par o mesmo vértice e não existe duas arestas que liguem o mesmo par de vértices.

Em [21], no exemplo 1.5, é mostrado um grafo que representa os estados do Brasil. Esse exemplo é ilustrado na Figura 2.1, na qual cada vértice representa um dos estados da República Federativa do Brasil; dois estados são adjacentes se têm uma fronteira em comum.

Figura 2.1 – Exemplo de grafo



Fonte: http://www.ime.usp.br/ pf/teoriadosgrafos/texto/TeoriaDosGrafos.pdf. Acessado em: 22/06/2013

2.3 Permutação

Uma permutação é uma modificação, uma troca de lugar. Uma função de permutação possibilita a criação e verificação de um grafo isomorfo. Neste trabalho a permutação é representada como um vetor de N valores, sendo N a quantidade de vértices do grafo. Também pode ser entendido como uma matriz $2 \times N$, ou uma matriz de permutação binária $N \times N$.

Cada elemento do vetor de permutação fornece duas informações, o índice do vetor e o valor. Através desses dois valores pode-se modificar a matriz de adjacência que representa o grafo, trocando a linha da matriz correspondente ao valor do índice pela linha do valor do elemento do vetor. A mesma troca deve ser feita com a coluna, de forma que essa permuta dos valores da matriz gere uma nova matriz e consequentemente um novo grafo, que é isomorfo ao anterior.

Para a obtenção da matriz G_1 através de G_0 e $Perm$ remapeiam-se as linhas e colunas da matriz G_0 seguindo a nova orientação representada em $Perm$. Por exemplo, [1,3], a

linha e coluna 1 da matriz G_1 recebe os valores da linha e coluna 3 da matriz G_0. Dessa forma pode-se obter uma matriz através da outra sabendo a permutação que foi aplicada. Note que é perfeitamente reversível. Esse processo de permutação também pode ser feito por multiplicação de matriz. Pode-se representar as permutações de linhas e colunas como matrizes e multiplicar pelo grafo (G_0), de forma que ao multiplicar a matriz de permutação de linhas (P) pela esquerda e a matriz de permutação de colunas, que é a transposta (P^T), pela direita, tem-se o grafo permutado (G_1). Aplicando no exemplo anterior $P \times G_0 \times P^T = G_1$

2.4 Isomorfismo

Dois grafos são isomorfos quando estes tem a mesma forma, ou seja, mesma quantidade de vértices, arestas e mesma estrutura. Para grafos isomorfos é possível ter uma função que transforma um grafo no seu isomorfo. Essa função é denominada isomorfismo ou permutação. Dois grafos $G_0 = (V_0, A_0)$ e $G_1 = (V_1, A_1)$ são iguais quando se tem $V_0 = V_1$ e $A_0 = A_1$. Há isomorfismo entre estes quando existir uma bijeção f, tal que, para todo $v \in V_0$ e para todo $w \in V_1, w = f(v)$ haja preservação das relações de adjacência, logo $(v_k, v_r) \in A_0$ se, e somente se, $(w_p, w_q) \in A_1$, com $w_p = f(v_k)$ e $w_q = f(v_r)$ [22].

Diz-se que um par de grafos $G_0 = (V_0, A_0)$ e $G_1 = (V_1, A_1)$ é isomorfo quando existe um mapeamento dos vértices do grafo G_0 para os vértices do grafo G_1, $\sigma : V_0 \rightarrow V_1$, tal que $(v_i, v_j) \in A_0$ se, e somente se, $(\sigma(v_i), \sigma(v_j)) \in A_1$. Na prática, a função que realiza tal mapeamento de forma que o isomorfismo é sempre preservado é a função de permutação dos vértices. Portanto, dado um grafo G com n vértices é possível gerar $n!$ grafos isomorfos a G, pois sempre existirá uma permutação inversa dos vértices que retorna para o grafo G, já que a permutação é uma função bijetiva. Encontrar um isomorfismo entre dois grafos, G_0 e G_1, é um problema difícil, mas fornecida a permutação dos vértices de G_0 que o torna igual a G_1, fica fácil verificar o isomorfismo entre estes grafos. O isomorfismo entre grafos é um problema que se acredita não estar em $\mathcal{P}$ [23].

2.5 Subgrafos

Dizemos que um grafo $G_1 = (V_1, A_1)$ é um subgrafo, ou sub-estrutura, de um grafo $G_0 = (V_0, A_0)$, quando $V_1 \subseteq V_0$ e $A_1 \subseteq A_0$, ou seja, um subgrafo é uma parte de um grafo. Por exemplo, tendo-se um grafo $G_0 = (V_0, A_0)$, um subgrafo seria um grafo $G_1 = (V_1, A_1)$ contido em G_0 que mantêm a mesma orientação de vértices e arestas. Como exemplo, pode-se representar as cidades brasileiras por um grafo. Um subgrafo da representação das cidades brasileiras pode ser a representação das cidades do Ceará, uma vez que o Grafo das cidades do Brasil tem todas as cidades de todos os estados inclusive as do Ceará. Outro exemplo bem

ilustrativo pode ser a representação de todas as estradas do Brasil. Um subgrafo poderia ser a representação das estradas estaduais.

Definição 1. *Dado um grafo* $G_0 = (V_0, A_0)$*, um grafo* $G_1 = (V_1, A_1)$ *é dito ser subgrafo de* G_0 *se* $V_1 \subseteq V_0, A_1 \subseteq A_0$.

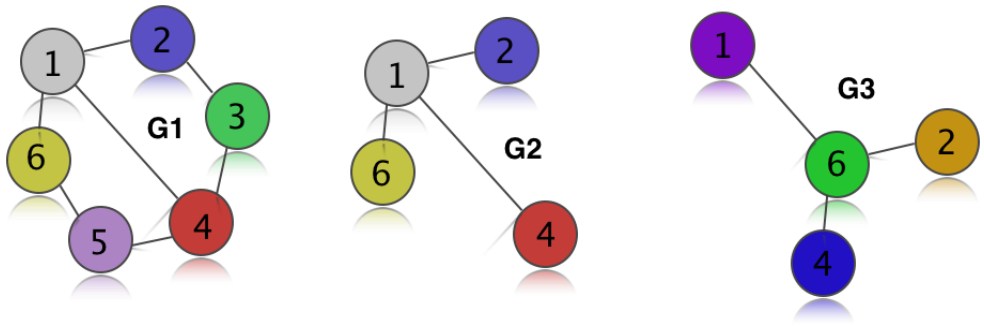
2.6 Isomorfismo de Subgrafos

Unindo os conceitos de isomorfismo de grafos e a noção de subgrafos tem-se uma outra definição, o isomorfismo de subgrafos.

Considere dois grafos $G_0 = (V_0, E_0)$ e $G_1 = (V_1, E_1)$, se G_0 contém um subgrafo, H, isomorfo a $G1$, ou seja, um subconjunto $V \subseteq V_0$ e $E \subseteq E_0$, tal que $|V| = |V_1|$, $|E| = |E_1|$, se existe uma função um-para-um $f : V_1 \rightarrow V$ satisfazendo $\{u, v\} \in E_1$ se, e somente se, $\{f(u), f(v)\} \in E$ [24].

Para uma melhor visualização a Figura 2.2 ilustra um exemplo de grafo G_1, subgrafo G_2 e subgrafo isomorfo G_3.

Figura 2.2 – Exemplo de subgrafo isomorfo



Fonte: Elaborada pelo autor.

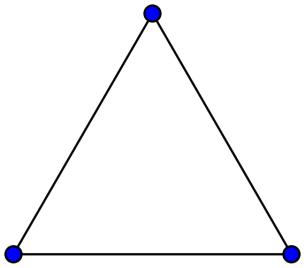
2.7 Problema de isomorfismo de subgrafo pertence à classe $\mathcal{NP}$-Completo

O problema de isomorfismo de subgrafos é classificado como $\mathcal{NP}$-Completo [24]. Nesta sessão será mostrado como ele pode ser reduzido ao problema do clique que também é $\mathcal{NP}$-Completo.

2.7.1 Problema do clique

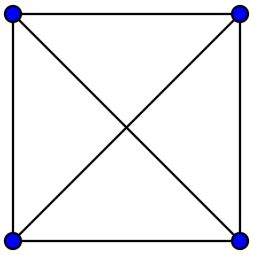
Para entender o problema do clique, primeiramente deve-se conhecer o conceito de grafo completo. Um grafo completo é um simples grafo sem orientação em que cada par de diferentes vértices estão ligados por uma única aresta, no qual todos os vértices tem ligações com todos os demais vértices do grafo. Desta forma, um grafo completo com n vértices, possui $\frac{n(n-1)}{2}$ arestas. Ver Figuras 2.3, 2.4, 2.5, as quais mostram, respectivamente, exemplos de grafos completos com 3, 4 e 5 vértices.

Figura 2.3 – Exemplo de grafo completo com 3 vértices



Fonte: http://pt.wikipedia.org/wiki/Grafo_completo. Acessado em: 22/06/2013

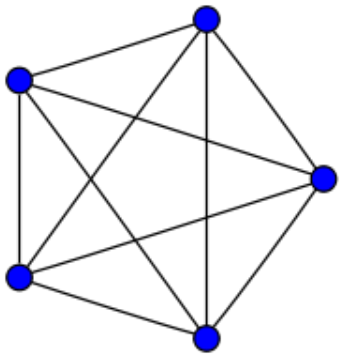
Figura 2.4 – Exemplo de grafo completo com 4 vértices



Fonte: http://pt.wikipedia.org/wiki/Grafo_completo. Acessado em: 22/06/2013

O problema do clique consiste em encontrar subgrafos completos dentro de um grafo. Esse problema envolve algumas abordagens de difícil resolução ($\mathcal{NP}$-Completo), como por exemplo: encontrar o maior ou os maiores cliques; encontrar um clique com maior valor; testar se um grafo possui um clique maior que um tamanho determinado; listar todos os cliques do grafo; dentre outros.

Figura 2.5 – Exemplo de grafo completo com 5 vértices



Fonte: http://pt.wikipedia.org/wiki/Grafo_completo. Acessado em: 22/06/2013

2.7.2 Prova que o problema de isomorfismo de subgrafos está em $\mathcal{NP}$-completo

A prova de que um problema é $\mathcal{NP}$-completo pode ser feita criando um algoritmo que converta este problema a um outro que é classificado como $\mathcal{NP}$-completo. A conversão deve ser feita em tempo polinomial. O inverso é verdadeiro, portanto, a prova também pode ser feita criando um algoritmo que instancie qualquer outro problema $\mathcal{NP}$-completo, em tempo polinomial, no algoritmo de resolução do problema proposto. Um algoritmo que resolva esse problema também resolverá o outro $\mathcal{NP}$-completo, pois o mesmo pode ser instanciado nesse algoritmo. Os problemas $\mathcal{NP}$-completos compartilham da propriedade de que se um for resolvido em tempo polinomial todos os outros também serão. Nessa demostração foi escolhido o problema do clique, que já é conhecido ser $\mathcal{NP}$-completo, para ser instanciado como problema de isomorfismo de subgrafos.

Definimos problema de isomorfismo de subgrafos como SGI. Inicialmente tem-se dois grafos isomorfos: SGI = $\{< G_1, G_2 > | G_1$ é isomorfo a um subgrafo de $G_2\}$. O primeiro passo é mostrar que $SGI \in \mathcal{NP}$. Tem-se que M é um mapeamento isomórfico dos vértices de G_1 para um subconjunto dos vértices de G_2. O *Algoritmo* 2.1 mostra como pode ser verificado o isomorfismo entre os grafos.

Para fazer essa verificação, a função V do *Algoritmo* 2.1 leva o tempo $O(|V|^2)$.

A segunda parte é mostrar que qualquer outro problema $\mathcal{NP}$-Completo, no caso o clique, pode ser reduzido em tempo polinomial ao problema proposto. Clique $\leqslant_p SGI$. Essa redução, *Algoritmo* 2.2, consiste em fazer o problema do clique se tornar um problema de isomorfismo de subgrafos. Ao solucionar o problema SGI também será solucionado o clique, e, consequentemente, os demais problemas $\mathcal{NP}$-completo, pois este pode ser uma instância de SGI.

O tempo da função R do *Algoritmo* 2.2 é polinomial em $E + V$. No algoritmo

Algoritmo 2.1 *Algoritmo* V de verificação

```
1: função V(G_1, G_2, M)
2:     enquanto v em G_1 faça
3:         enquanto u em G_1 faça
4:             se (u, v) está em G_1 e (f(u), f(v)) não está em G_2 então
5:                 devolve False
6:             fim se
7:         fim enquanto
8:     fim enquanto
9:     devolve True
10: fim função
```

Algoritmo 2.2 *Algoritmo* de redução $R : (G, K) \rightarrow (G_1, G_2)$

```
1: função R(G, K)
2:     G_2 = G
3:     G_1 = Grafo completo com K vértices
4: fim função
```

$R :< G, K >\in$ CLIQUE $\Leftrightarrow < G_1, G_2 >\in SGI$. Assumindo que $< G, K >\in$ CLIQUE, G tem um clique de tamanho K, isso é equivalente a dizer G tem um subgrafo completo de tamanho K. Assumindo que G_2 é o grafo G, G_2 tem um subgrafo de tamanho K isomorfo a G_1, tendo-se então $< G1, G2 >\in SGI$. Essa demonstração pode ser vista em [25].

2.8 Algoritmos de resolução do PISG

Existe uma grande quantidade de algoritmos e métodos para resolver o problema de isomorfismo de subgrafos. Nesta seção são mostrados os principais métodos, algoritmos, filtros e otimizações que visam resolver o problema de isomorfismo de subgrafos.

A primeira, e mais comum, resolução para o problema de isomorfismo de subgrafos é através do método de força bruta, usando backtrack [26]. Nessa abordagem testa-se todas as possibilidades em uma árvore de busca. Ullmann propôs, em 1976, [27] um algoritmo que elimina nós sucessores na árvore de busca, poupando algumas verificações desnecessárias, esse algoritmo ficou conhecido como algoritmo de Ullmann, e é a primeira resolução prática do problema de isomorfismo de subgrafos. No trabalho de Ullmann também foram mostrados experimentos sobre a eficiência do algoritmo, bem como o aumento na velocidade ganha para encontrar o isomorfismo. Apesar da melhora no método força bruta, o algoritmo de Ullmann favorece a resolução de apenas algumas classes de grafos.

Em 1999 Cordella apresentou o algoritmo VF, baseado numa estratégia de busca em profundidade, com um conjunto de regras para podar a árvore de busca [28]. Uma segunda versão do algoritmo, denominado VF2 [29], foi mostrada por Cordella em 2001. Essa ultima

versão otimiza o uso de memória, aumentando a performance para resolução de grafos maiores. Em 2004 Cordella fez uma análise mais geral do algoritmo VF2 [30].

No ano de 2008 foi proposto pelo professor Haichuan Shang o algoritmo QuickSI com o propósito de ser mais eficiente para grafos grandes [31]. O algoritmo usa um método diferente dos anteriores, que eram baseados no algoritmo de Ullmann. O algoritmo mostrado escolhe uma ordem de busca mais eficaz. Também é possível aplicar uma fase de filtragem no grafos para uma diminuição das possibilidades. Na publicação é mostrado a eficiência e escalabilidade do método proposto.

Em 2008 foi apresentada por Huahai He e Ambuj K. Singh uma linguagem formal para banco de dados de grafos, a *Graph Query Language* conhecida como GraphQL [32]. Também foi feito um estudo comparativo justificando a criação de uma linguagem de consulta para grafos e as vantagens em usar uma linguagem otimizada comparada ao SQL.

Em 2009 e 2010 foram apresentados mais dois algoritmos, o GADDI e o SPath, ambos com foco em grafos com grande quantidade de vértices e arestas. GADDI, com foco em resolver isomorfismo de subgrafos em redes biológicas [33]. SPath [34], que tenta reduzir a profundidade da árvore usando um método recursivo baseado em caminho.

Um novo filtro baseado em *local all different constraints* foi apresentado por Christine Solnon [35] no ano de 2010 e mostrado que poda mais ramos e é mais eficiente que os demais filtros existentes até o momento. Solnon disponibilizou uma implementação em C que pode ser encontrada em: `http://liris.cnrs.fr/csolnon/LAD`(Acessado em abril de 2015). Em Junho de 2013 ela atualizou a implementação e adicionou opção para grafos direcionados e com rótulos.

Também são encontrados na literatura trabalhos comparando performances [36], [30]. Essas comparações são importantes para saber em qual caso é melhor para utilizar cada algoritmo. Uma análise mais recente e completa pode ser encontrada em [37].

2.9 Classes de subgrafos resolvidas em tempo polinomial

O problema de isomorfismo de subgrafos é um problema NP-Completo, com resolução exponencial. Entretanto, é possível garantir uma solução eficiente para um determinado grupo restrito. Foi feito uma pesquisa bibliográfica para elencar esses grupos que tem uma resolução com complexidades diferentes.

Lingas, em 1988 mostrou um algoritmo polinomial para isomorfismo de subgrafos *two-connected series-parallel* [38] e em 1989 apresentou um algoritmo de tempo cúbico para grafos outerplanar biconectados [39].

Desmark mostrou um algoritmo com complexidade $O(n^{k+2})$ para isomorfismo de

subgrafos *k-connected partial k-trees* [40].

Em 1991 Damaschke mostrou que isomorfismo de subgrafos induzidos para co-grafos e grafos que incluem uniões disjuntas de caminho são NP-Completo [41].

Em 1995 Eppstein publicou uma solução $k^{O(k)}n$ para subgrafos planares [42], sendo k o número de vértices do subgrafo e n a quantidade de vértices do grafo base. Em 2009 Dorn melhorou o tempo para $2^{O(k)}n$ [43] [44].

Gupta, em 1996 apresentou uma análise para grafos da classe *k-trees* parciais de grau limitado (*partial k-trees of bounded degree*) em [45]. O trabalho mostra a complexidade $O(n^k)$ para a classe citada.

Em 2009 Damiand publicou um algoritmo em tempo polinomial que encontra subgrafos em grafos planares [46]. No trabalho é feito um experimento com imagens, aplicação em que pode ser representada por grafos planares, desde que as imagens sejam bi-dimensionais.

Heggernes em 2010 mostrou que isomorfismo de subgrafos induzidos em gráficos de intervalo apropriados é NP-completo, mesmo se o gráfico base for conexo, enquanto que o problema pode ser resolvido no tempo polinomial se o subgrafo for conexo [47]. Além disso,foi mostrado que isomorfismo de subgrafos induzidos é tratável com parâmetro fixo quando parametrizado pelo número de componentes ligadas ao subgrafo, se o grafo base for um grafo de intervalo e o subgrafo for um *proper interval graph*.

Marx e Schlotter, em 2012, mostraram que isomorfismo de subgrafos induzidos em grafos de intervalo é W[1]-Hard quando parametrizado pelo número de vértices do subgrafo, mas parâmetro fixo tratável quando parametrizado pelo número de vértices a ser removido do grafo base [48].

Em 2012, Kijima publicou uma análise de isomorfismo de subgrafos para grafos conexos perfeitos [44]. No trabalho foi mostrado que as classes *chain graphs, cochain graphs* e *threshold graphs* são problemas NP-Completos mas podem ser resolvidas em tempo polinomial desde que os subgrafos sejam restritos a mesma classe dos grafos.

Em 2013, Heggernes [49] publicou uma análise para isomorfismo de subgrafos em classes de subárvores induzidas em grafos de intervalo. Foi mostrado que quando o G for um grafo de intervalo e H(subgrafo) for uma arvore, pode-se ser resolvido o isomorfismo em tempo polinomial. Se G for um *proper interval graph* e H uma arvore o problema é NP-Completo, mas se H for um caminho, pode ser resolvido em tempo polinomial.

Matsuo no ano de 2014 apresentou uma solução polinomial para o caso em que o grafo base é *chordal* e os subgrafos isomorfos são *co-chain*. Também mostrou um algoritmo com crescimento de tempo linear para grafo base trivialmente perfeito e subgrafo *threshold* [50].

2.10 Classes de subgrafos não resolvidas em tempo polinomial

É desconhecida uma classe de grafos e subgrafos que não tenha ao menos uma instância que possa ser resolvida em tempo polinomial. O problema de isomorfismo de subgrafos são não polinomiais para o caso geral, ou seja, pode-se aplicar métodos para garantir algumas instâncias mas não pode-se garantir que para qualquer instância, existe um algoritmo que resolva o problema em tempo polinomial. Tem-se como evitar tais instâncias, pois elas possuem características bem definidas.

Para o propósito deste trabalho, os grafos devem ser o mais genérico possível. De forma que, apenas um algoritmo que garanta a resolução do caso geral para o problema de isomorfismo de subgrafos possa encontrar a função de permutação entre os pares grafos e subgrafos em tempo polinomial para um número considerável de instâncias.

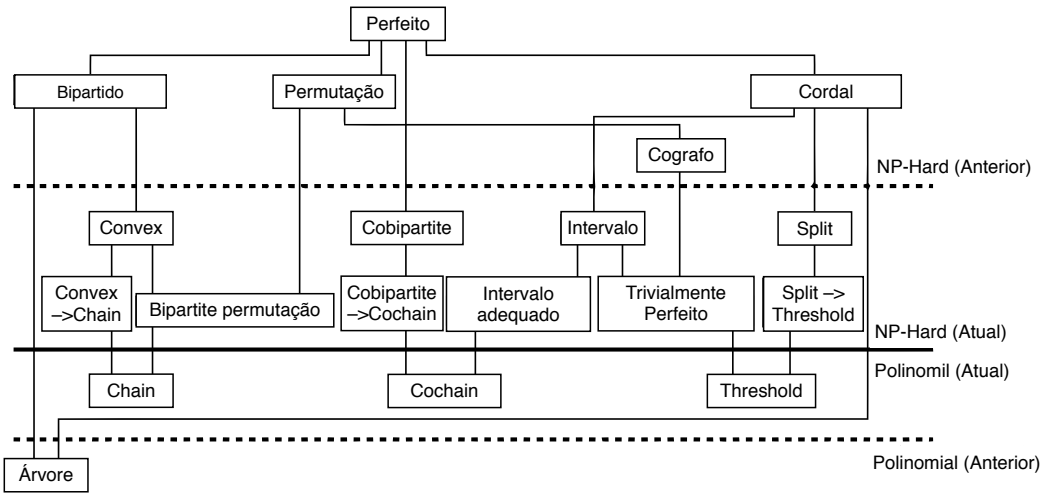
Recentemente, em 2012, Kijima [44] publicou uma análise bem completa sobre o problema de isomorfismo de subgrafos, apresentou uma imagem que ilustra bem as classes de subgrafos perfeitos. Como pode ser observado na Figura 2.6, todas as classes decaem em casos que podem ser resolvidos em tempo polinomial. Para garantir instâncias sempre não resolvidas em tempo polinomial, deve-se criar um gerador que não produz grafos que caiam nas classes resolvidas em tempo polinomial, seja evitando determinadas características ou verificando após a geração.

2.10.1 Grafos perfeitos

O número cromático de um grafo $G(V, A)$, denotado por $X(G)$, é o número mínimo de cores necessárias para colorir os vértices de um grafo de forma que não tenha vértices adjacentes com a mesma cor. Define-se $\omega(G)$ o tamanho do maior clique de um grafo $G(V, A)$. Todos os vértices de um clique tem ligações com os demais, de forma a delimitar o número cromático. Berge definiu como condição para um grafo ser perfeito se para todo subgrafo induzido H de $G, X(H) = \omega(H)$.

Um grafo é perfeito se e somente se, o mesmo ou seu complemento, não contem um buraco ímpar. Um buraco é um ciclo induzido de comprimento maior que quatro. Define-se como buraco ímpar um ciclo C_k com k sendo um valor ímpar maior ou igual a 5. Para qualquer subgrafo induzido de um grafo perfeito o número cromático é igual ao tamanho do maior clique. O conceito de grafo perfeito foi introduzido por Berge no ano de 1960. Na conjectura ele mostrou que os únicos grafos não perfeitos ou imperfeitos são os que contem buracos ímpares e seus complementos. A prova da conjectura foi anunciada em 2002 por Chudnovsky, Robertson, Seymour e Thomas e publicada em 2006 [51]. É mostrado em [52] que grafos perfeitos podem

Figura 2.6 – Classificação de Grafos Perfeitos



Fonte: [44]

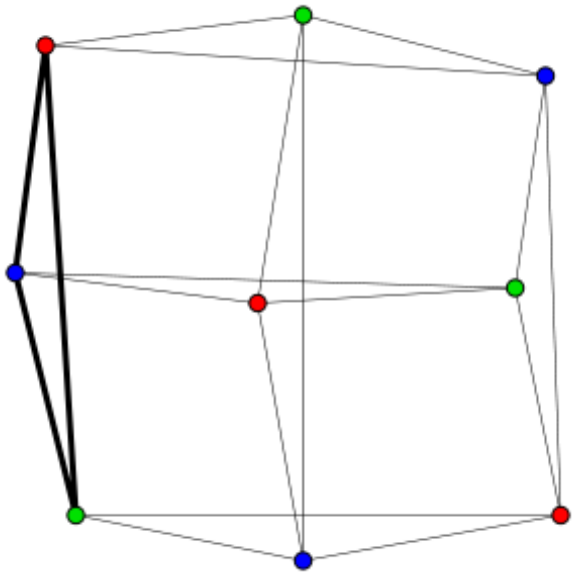
ser reconhecidos em tempo polinomial. Na Figura 2.7 é mostrado um grafo perfeito $G(V, A)$, com número cromático $X(G) = 3$ e, em negrito, um clique $\omega(G) = 3$. Para qualquer subgrafo H de G tem-se $X(H) = \omega(H)$

Abaixo uma descrição das classes listadas por Kijima [44] como classes pertencentes ao grupo resolvível em tempo polinomial.

Grafos threshold

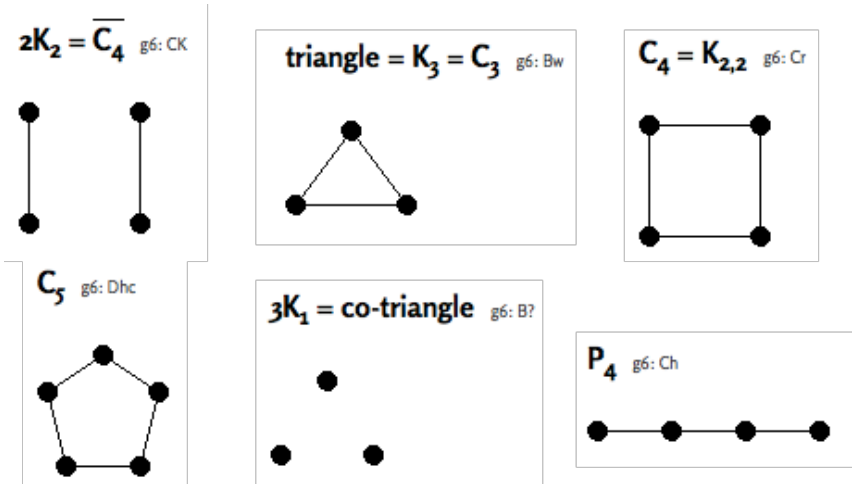
A classe grafos threshold, também conhecido como limiar, foi introduzidos nos anos 70 [53]. Podem ser caracterizados como grafos livres $2k_2, C_4, P_4$ ver Figura 2.8. Uma outra caracterização é via grafos *split nested*. Dizemos que um grafo é split se o eu conjunto de vértices V pode ser particionado em um clique K e um conjunto estável S. Os grafos *thresholds* são grafos *splits* com a propriedade *nested*(vizinhança). Além dos vértices formarem um clique K e um conjunto estável S, pode-se particionar K e S em conjuntos disjuntos tais que vértices pertencem a K_i e S_i, admitem a mesma quantidade de vizinhos. Na Figura 2.9 pode ser visto um exemplo de grafo *threshold*. Os vértices em vermelho são ligados a todos os demais e os vértices preto são ligados apenas aos vermelhos.

Figura 2.7 – Exemplo de um grafo perfeito



Fonte: `http://pt.wikipedia.org/wiki/Grafo_perfeito`. Acessado em novembro de 2014.

Figura 2.8 – Exemplo de alguns pequenos grafos



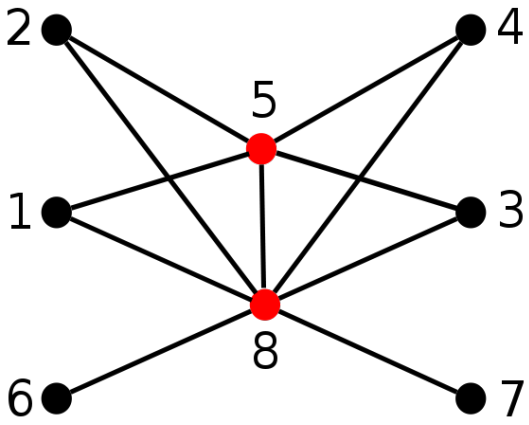
Fonte: `http://www.graphclasses.org/smallgraphs.html`Acessado em novembro de 2014.

Exemplo de grafos chain/co-chain

Um grafo é *chain* se, e somente se, não contiver nenhum subconjunto de vértices que induz $2K_2, C_3$ ou C_5 [54] ver Figura 2.8. Também conhecido por *bsplit*, podem ser divididos em um caminho independente e um clique.

Grafo co-chain é o complemento de um grafo *chain*. Um grafo é *Co-chain* se, e somente se, não contiver nenhum subconjunto de vértices que induz $3K_1, C_4$ ou C_5 [54] ver

Figura 2.9 – Exemplo de grafo *Threshold*



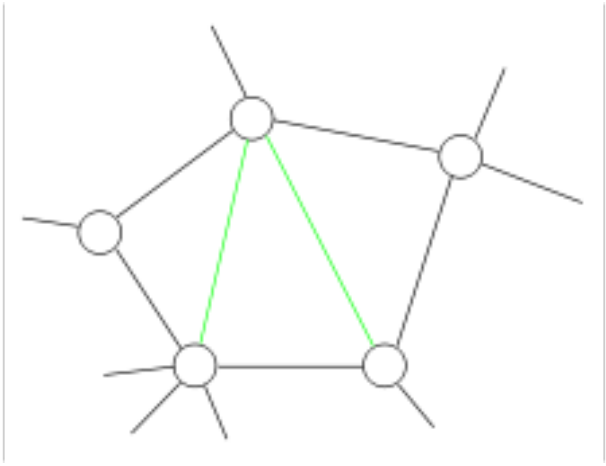
Fonte: `http://es.wikipedia.org/wiki/Grafo_umbral`Acessado em novembro de 2014

Figura 2.8.

Exemplo de grafo cordal

Um grafo é cordal quando todo ciclo de comprimento maior que quatro tem uma corda, isto é, uma aresta ligando dois vértices não consecutivos no ciclo. Uma descrição das propriedades pode ser encontrada em [55]. Os grafos cordais tem reconhecimento polinomial [56]. Na Figura 2.10 pode ser visto um exemplo de grafo cordal. A remoção de uma das arestas em verde resulta em um grafo não cordal.

Figura 2.10 – Exemplo de Grafo cordal



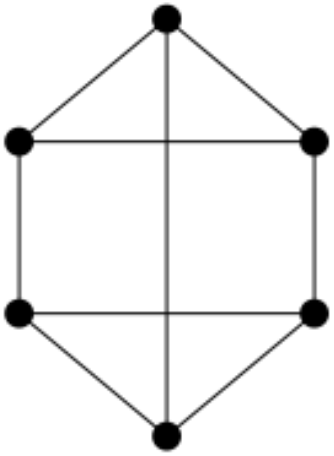
Fonte: `http://en.wikipedia.org/wiki/Chordal_graph`Acessado em novembro de 2014

2.10.2 Exemplo de grafos regulares

Através dos estudos na literatura e dos algoritmos existentes, viu-se uma maior dificuldade computacional em mapear grafos e subgrafos com graus semelhantes. Grafos densos, com alta regularidade nos graus dos vértices, "fogem" naturalmente de classes que existem uma solução polinomial, como por exemplo grafos bipartidos ou árvores.

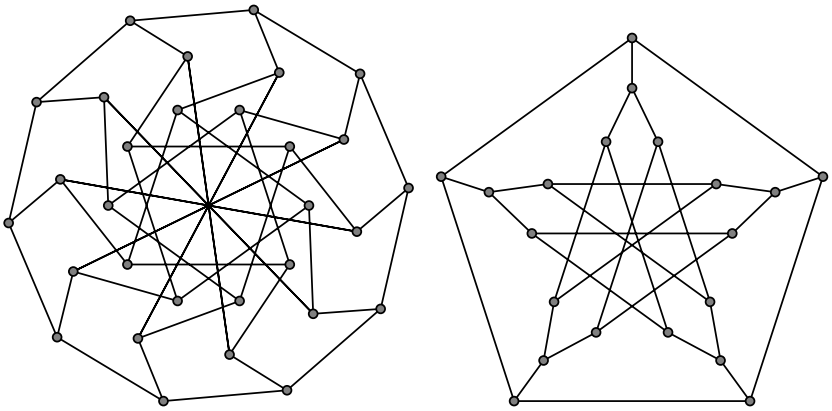
Um grafo é dito regular se a quantidade de ligações de cada vértice é igual para todos os vértices. Se um vértices tem k ligações, todos os demais devem ter k ligações, esse grafo é dito k-regular. As Figuras 2.11 e 2.12 mostram exemplos de grafos 3-regulares.

Figura 2.11 – Exemplo de grafo regular



Fonte: `http://pt.wikipedia.org/wiki/Grafo_regular` Acessado em novembro de 2014

Figura 2.12 – Exemplo de grafos regulares



Fonte: `http://www.texample.net/tikz/examples/combinatorial-graphs` Acessado em novembro de 2014

2.11 Explicação do LAD-filtro (*Local All Different*)

O problema de isomorfismos de subgrafos pode ser resolvidos por uma exploração sistemática do espaço de busca composto por todos as possíveis combinações injetivas do conjunto padrão de nós ao conjunto de nós de destino. Basicamente, o processo ocorre da seguinte maneira: iniciando de uma correspondência vazia, se estende de forma incremental até uma correspondência parcial, combinando um nó não-combinado padrão para um nó de destino incomparável até que:

1) Algumas arestas não sejam acompanhadas por um correspondente atual (nesse caso a pesquisa deve recuar para um ponto de escolha anterior e continuar com outra extensão);

2) Todos os nós padrões foram pareados (foi encontrada uma solução).

Para reduzir o espaço de busca, esta exploração exaustiva é combinada com técnicas de filtragem que visam remover pares candidatas de nós não correspondidos do alvo padrão. Diferentes níveis de filtragem podem ser considerados; alguns são mais fortes do que outros (eles removem mais nós), mas também tem maior complexidade temporal.

Em [35] Christine Solnon introduz um novo algoritmo de filtragem com base em restrições locais todas diferentes. Solnon mostrou que essa filtragem é mais forte do que outras filtragens existentes -ou seja, ele poda mais ramos- e que também é mais eficiente, isto é, permite resolver mais instâncias em menos tempo. Dessa forma, o LAD é nesse momento o algoritmo ideal para testar experimentalmente uma proposta de gerador aleatório do problema de isomorfismo de subgrafos por dois motivos:

1) Terá mais sucesso no tempo de ataque ao gerador aleatório comparado com outros algoritmos;

2) Não possui rotinas de escolha aleatória de ramos a serem podados, usando apenas os critérios de filtragem que escolhem de forma otimizada que ramos devem ser podados, deixando constante o tempo de espera quando o algoritmo é executado mais de uma vez para a mesma instância. Por exemplo, para uma instância com n=30, k=24 e r=10 o tempo de resposta do LAD foi t=4078 após 10 execuções sob a mesma instância. Quando muda a instância, o tempo também muda.

Nesta seção, será apresentado alguns elementos para a construção do LAD. Para cada função de subisomorfismo $f : N_p \rightarrow N_t$, para cada nó padrão $u \in N_p$, tem-se:

1. $\forall u' \in adj(u), f(u') \in adj(f(u))$
2. $\forall (u', u") \in adj(u) \times adj(u), u' \neq u" \Rightarrow f(u') \neq f(u")$

A primeira propriedade é consequência direta do fato de que as arestas são preservados pela função de subisomorfismo ao passo que a segunda propriedade é consequência direta do fato que as funções de subisomorfismo são injetoras.

Ao associar CSP (*constraint satisfaction problem*) ao problema de isomorfismo de subgrafos essas duas propriedades podem ser expressadas pela seguinte restrição na vizinhança de u:

$$x(u) = v \Rightarrow \forall u' \in adj(u), x_{u'} \in adj(v) \wedge \text{allDiff}(\{x_{u'} | u' \in adj(u)\})$$

A arco consistência generalizada de um restrição de vizinhança pode ser assegurada pela procura de uma correspondência de cobertura em um grafo bipartido, como proposto por Régin em [57] para a restrição global *AllDifferent*. Uma correspondência de um grafo $G = (N, E)$ é um subconjunto de arestas $m \subseteq E$ de tal forma que não há duas arestas de m compartilhando um mesmo ponto final. Uma correspondência $m \subseteq E$ abrange um conjunto de nós N_i se cada nó de N_i é um ponto final de uma aresta de m. Neste caso, diremos que m é uma correspondência de N_i-cobertura de G.

Para cada par de nós (u, v) de tal modo que $V \in D_u$, um grafo bipartido que associa um nó a cada nó adjacente a u, v e uma aresta com cada par (u', v') de tal modo que $v' \in D_{u'}$. Dado dois nós $(u, v) \in N_p \times N_t$ tal que $v \in D_u$, um grafo bipartido $G_{(u,v)} = (N_{(u,v)}, E_{(u,v)})$ tal que:

- $N(u, v) = adj(u) \bigcup adj(v)$;
- $E(u, v) = \{(u', v') \in adj(u) \times adj(v) | v' \in D_{u'}\}$

Se não existe uma correspondência do grafo $G_{(u,v)}$ que cobre $adj(u)$, então os nodos ajacente a u não podem ser associados a todos os nós diferentes, portanto v pode ser removido de D_u.

Essa filtragem deve ser iterativa. De fato, quando v é removido de D_u a aresta (u, v) é removida dos outros grafos bipartidos de modo que alguns grafos bipartidos podem não ter mais associação de cobertura. Um ponto chave para uma implementação incremental desses filtros reside no fato que as arestas (u, v) apenas pertencem a grafos bipartidos $G_{(u',v')}$ tal que $u' \in adj(u)$ e $v' \in adj(v) \bigcap D(u')$. A filtragem é iterativa até que o domínio se torne vazio - detectando, assim, uma inconsistência - ou chegar a um ponto fixo tal que arc-consistência generalizada tenha sido cumprida, ou seja, de tal forma que para cada par (u, v) existe uma $adj(u)$-cobertura correspondente de $G_{(u,v)}$.

O procedimento chamado LAD (*Local All Different*), recebe como entrada um conjunto S de pares de nós padrão/alvo a ser filtrado. Na raiz da árvore de busca, este conjunto deve conter todos os pares de nós padrão/alvo, ou seja, $S = \{(u, v) | u \in N_p, v \in D_u\}$. Em seguida, a cada ponto de escolha da árvore de pesquisa, S deve ser iniciado com o conjunto de todos os pares (u, v) tais que $v \in D_u$ e um nó adjacente ao v tiver sido removido a partir do domínio de um nó adjacente a u desde a última chamada para LAD-filtro.

Para cada par de nós (u, v) pertencentes ao conjunto S, LAD-filtro checa se existe uma correspondência de $G(u, v)$ que abrange $adj(u)$. Se este não for o caso, então v é removido

de D_u e todos os pares (u', v') de tal modo que u' é adjacente a u, e v' é adjacente a v e pertence a $D_{u'}$ são adicionados a S.

O ponto chave é a implementação eficaz do processo que verifica se existe uma correspondência de cobertura de $G(u, v)$. Regin mostrou em [Reg94] que se pode usar o algoritmo de Hopcroft e Karp [HK73] para encontrar uma correspondência. A complexidade de tempo deste algoritmo é $O(a\sqrt{b})$ onde a e b são respectivamente o número de arestas e nós no grafo bipartido. Como o grafo bipartido $G(u, v)$ tem $\#adj(u) + \#adj(v)$ nós e, no pior caso (se não houver redução do domínio), $\#adj(u) \cdot \#adj(v)$ arestas, e como $d_t \geq d_p$ (caso contrário a instância do problema de isomorfismo de subgrafos é trivialmente inconsistente), a complexidade de verificar se existe uma correspondência de cobertura de $G_{(u,v)}$ é $O(d_p \cdot d_t \cdot \sqrt{d_t})$

A complexidade pode ser melhorada através da exploração do fato de o algoritmo de Hopcroft e Karp ser incremental: a partir de um correspondência vazia, iterativamente calcula-se novos emparelhamentos que contêm mais arestas que o correspondente anterior, até que a correspondência seja máxima. Cada iteração consiste basicamente de uma busca em largura $O(d_p \cdot d_t)$ enquanto o número de iterações é delimitada por $2 \cdot \sqrt{d_t + d_p}$. No entanto, se inicia o algoritmo com uma correspondência com k arestas, e se a correspondência máxima for l arestas, então o número de iterações também é delimitada por $l - k$.

Foi usada essa propriedade para melhorar a complexidade de tempo de LAD-filtro. Mais precisamente, para cada nó padrão $u \in N_p$ e cada nó de alvo $v \in D_u$, memoriza-se a última correspondência computada de $G(u, v)$. A complexidade de espaço para armazenar todos os emparelhamentos de cobertura de todos os grafos bipartidos é $O(n_p \cdot n_t \cdot d_p)$(existe no máximo $n_p \cdot n_t$ grafos bipartidos, e a correspondência de cobertura de $G(u, v)$ é composta de $\#adj(u)$ arestas). Como seria muito caro, tanto no tempo como em memória, para criar uma cópia de todas as correspondências em cada ponto de escolha, simplesmente atualiza-se as correspondências quando é necessário. Mais precisamente, a cada passo é preciso verificar se existe uma correspondência de um grafo bipartido $G(u, v)$, da seguinte forma:

1. Escaneia-se a última correspondência gravada de $adj(u)$ e remove todos os pares (u', v') de tal forma que v' não pertence a $D(u')$;
2. Se um ou mais pares forem removidos, chama-se *Hopcroft Karp* para completá-lo;
3. Se *Hopcroft Karp* conseguir completá-lo, então é armazenado a correspondência completa.

A complexidade de tempo do LAD-filtro é $O(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$. Deve-se notar também que:

- A complexidade para encontrar a primeira correspondência é $O(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$, esse passo é feito uma vez no inicio do processo de busca;

- Cada vez que um valor de v é removido de um domínio D_u, tem-se que atualizar as correspondências de todos os grafos bipartidos $G(u', v')$ de tal modo que $u' \in adj(u)$ e $v' \in D_{u'} \cap adj(v)$, isto é, $d_p \cdot d_t$ grafos bipartidos no pior caso, e cada atualização é feita de forma incremental em $O(d_p \cdot d_t)$.

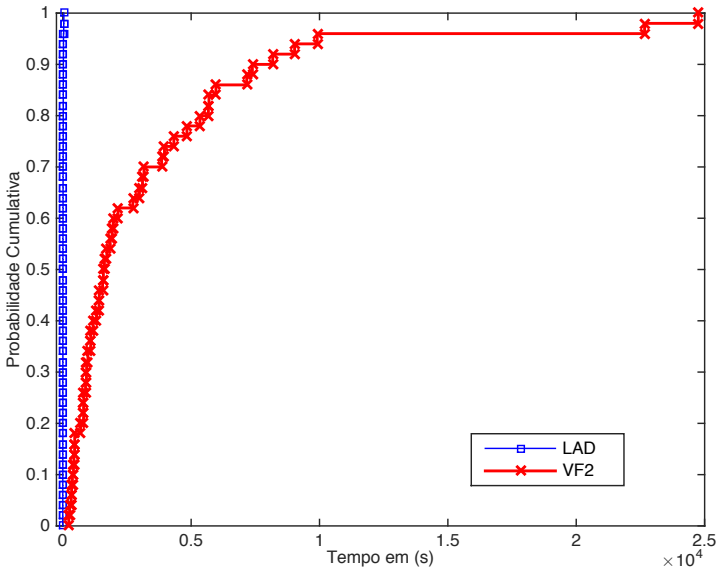
- No pior caso, apenas um valor é removido na atualização da correspondência de todos os vizinhos e eles são $n_p \cdot n_t$ valores para remover.

2.12 Comparação LAD com VF2

Foi feito um experimento de comparação com os métodos conhecidos de resolução de isomorfismo de subgrafos mais eficiente no momento [35], LAD e VF2. Nos testes foram gerados grafos regulares com 30 vértices e grau 20. Para gerar o subgrafo, foram retirados 10 vértices. O teste foi repetido 50 vezes e dado como entrada a mesma instância para os dois algoritmos. Os resultados podem ser visto na Figura 2.13. Foi feito uma comparação com todos os dados do vetor e visto que o algoritmo LAD obteve resultados melhores que o VF2 para todos os casos.

O método LAD obteve melhor resultado para todos os testes feitos e tempo de resolução consideravelmente inferior (melhor). Como é algoritmo mais eficiente conhecido no momento, foi utilizado o método LAD para resolver os demais testes de isomorfismo de grafos e subgrafos.

Figura 2.13 – Gráfico de comparação LAD e VF2



Capítulo 3

Gerador pseudo aleatório de PISGs

Para a aplicação num algoritmo criptográfico é importante que o gerador aleatório de instâncias seja confiável, evitando que chaves do algoritmo sejam forjadas. Neste trabalho apresentamos um gerador pseudo aleatório do problema de isomorfismo de subgrafos e uma análise estatística do tempo de espera necessário para um "ataque" bem sucedido.

3.1 Introdução

Problemas computacionalmente árduos são a base para construção de algoritmos criptográficos baseados no paradigma de segurança demonstrável. Nesse paradigma, conhecer a solução de um problema com alta complexidade, cuja solução é desconhecida a outra máquina, garante a segurança de muitos algoritmos. O exemplo mais popular, a cifra RSA, é construído sobre a dificuldade de fatoração de números semiprimos grandes [3].

Outros problemas computacionais podem ser usados na construção de algoritmos criptográficos. O problema de isomorfismo de grafos pode ser usado para construção de prova de conhecimento nulo. Além disso, provas de conhecimento nulo podem ser construídas a partir de problemas na classe NP-completo, desde que exista um cifra segura [9]. Uma proposta simples de prova de conhecimento nulo com base no problema de isomorfismo de subgrafos, um problema NP-completo, foi apresentada em [14]. Assinaturas digitais com base no problema de isomorfismo de subgrafos foi proposto em [58].

Para a aplicação de um algoritmo criptográfico é importante que o gerador aleatório de instâncias seja confiável. Por exemplo, em [8] foi mostrado que algumas chaves RSA, que são frequentemente utilizadas em protocolos da internet, são geradas por um mau gerador pseudo aleatório de números e podem ser facilmente recuperados, fornecendo nenhuma segurança. Também, na proposta de assinaturas digitais com o problema de isomorfismo de grafos, em [58], mais tarde alguns dos autores, numa comunicação pessoal com Kutylowski,

como apresentado em [59], descobriram que o algoritmo de geração de chaves permitia que assinaturas falsas fossem forjadas, tendo que propor uma correção.

Neste trabalho nós propomos uma gerador pseudo aleatório do problema de isomorfismo de grafos para aplicações criptográficas. Foi realizado um estudo estatístico do ataque usando o algoritmo LAD (*Local AllDifferent*) [35], algoritmo de busca de isomorfismo de subgrafos. Notamos que para grafos pequenos, até 30 vértices, a distribuição é inclinada a esquerda, a qual aproximamos por função densidade de Fisher-Snedecor. Verificamos que, além da quantidade de vértices, a escolha do grau do grafo base e o número de retiradas de vértices para formação do subgrafo isomorfo são determinantes no tempo de espera para o sucesso do ataque com o LAD.

3.2 Gerador pseudo aleatório do problema de isomorfismo de subgrafos

Um subgrafo é uma parte de um grafo. Tendo-se um grafo $G_0 = (V_0, A_0)$, um subgrafo seria um grafo $G_1 = (V_1, A_1)$ contido em G_0 que mantêm a mesma orientação de vértices e arestas. Quanto ao isomorfismo, dizemos que dois grafos são isomorfos quando estes tem a mesma forma, ou seja, mesma quantidade de vértices, arestas e mesma estrutura. O problema de isomorfismo de subgrafo é uma tarefa computacional no qual dois grafos, G_0 e G_1, são dados como entrada, e é preciso determinar se G_0 contém um subgrafo H que é isomorfo a G_1. A função bijetiva que opera o isomorfismo, nada mais é, do que uma permutação sobre a organização dos vértices. Escrevendo de maneira mais formal, existe um isomorfismo de subgrafos entre os grafos $G_0 = (V_0, A_0)$ e $G_1 = (V_1, A_1)$, se G_0 contém um subgrafo $H = (V, A)$, ou seja, um subconjunto $V \subseteq V_0$ e $A \subseteq A_0$, tal que $|V| = |V_1|$, $|A| = |A_1|$, e existe uma função de mapeamento um-para-um $f : V_1 \rightarrow V$ satisfazendo $\{u, v\} \in A_1$ se, e somente se, $\{f(u), f(v)\} \in A$ [24].

O problema de isomorfismo de subgrafos é classificado como $\mathcal{NP}$-completo [24], mas nem todo par de grafos contidos no problema de isomorfismo de subgrafos é um problema árduo. Por exemplo, grafos de gênero limitado [60, 61], como os grafos planares, grafos de valência limitada [62] e grafos cuja matriz de adjacência possui multiplicidade limitada [63], têm solução polinomial e devem ser evitados. Também sabemos que árvores de grau limitado também podem ser resolvidas em tempo polinomial [40, 64], pois a complexidade é polinomial em n e exponencial em k, $O(n^k)$.

Foi decidido fazer um gerador do problema de isomorfismo de subgrafo a partir de um gerador aleatório de grafos regulares, cujo grau é uma fração de n. Dessa forma, mesmo que o grafo gerado seja uma árvore, ele ainda será exponencial em n, $O(n^{0,8n})$. Grafos de

alto grau, possuem alta multiplicidade na sua matriz adjacência e alto gênero, o que evita os casos [60–64]. Foi proposto o seguinte algoritmo para geração do problema de isomorfismo de subgrafos:

Algoritmo 3.1 Gerador pseudo aleatório de pares de grafos dentro do problema de isomorfismo de subgrafos

---

1. Gera-se aleatoriamente um grafo k-regular G_0 com n vértices;
2. Uma permutação π é escolhida aleatoriamente e aplicada no grafo G_0, $\pi : G_0 \rightarrow H_0$;
3. Realiza-se r retiradas de vértices em H_0, escolhidos aleatoriamente, e o grafo resultante é G_1. O procedimento de retiradas de vértices será denotador por ρ_r, $\rho_r : H_0 \rightarrow G_1$. O isomorfismo será $\sigma = \rho_r \circ \pi$, pois:

$$\sigma : G_0 = \rho_r \circ \pi : G_0 = \rho_r : H_0 \rightarrow G_1.$$

4. A tupla (G_0, G_1, σ) é retornada como resultado.

Os parâmetros n, r e k definirão o tempo de resolução do problema de isomorfismo entre os grafos G_0 e G_1. Eles serão os parâmetros de segurança do algoritmo. Usaremos o termo pseudo aleatório por causa da algoritmo de construção do grafo regular G_0 e do sorteio de números de 1 até n para a construção da permutação π. O G_0 é construído com a função GraphBase.K_Regular(n,k) da biblioteca Python_igraph [65].

3.3 Ataque

Neste trabalho decidimos gerar os pares de grafos como descritos no Algoritmo 1 e descobrir π através dos grafos G_0 e G_1. Muitos algoritmos para decidir se dois grafos (grafo e subgrafo) são isomorfos já foram propostos [27,29–32] para o ataque do gerador proposto nesse artigo, decidimos usar um algoritmos de busca de isomorfismo de subgrafos, conhecido como LAD, proposto em [35], devido este ter apresentado menor tempo de execução em comparação com muitas propostas anteriores.

Neste momento, queremos apenas apresentar um estudo da variação do tempo médio de teste de isomorfismo usando o LAD, variando o grau do grafo, k. Este é um parâmetro de segurança do gerador, além dos tamanhos dos grafos, definidos por n e r. A razão de sua importância se deve ao fato que ele pode maximizar o tempo de determinação do isomorfismo com relação aos tamanhos dos grafos. Isso pode evitar grafos excessivamente grandes que inviabilizam a troca de grafos na rede.

Na simulação apresentada a seguir a máquina utilizada foi um Desktop HP Elite Desk com processador Intel core i5 4570 3.2 GHz, com 16GB de memória RAM, Sistema

operacional Ubuntu 14.04 LTS de 64bits. Os grafos gerados G_0 e G_1 inseridos no LAD, LAD(G_0, G_1), sempre retornarão verdadeiro, pois todos os testes são de grafos isomorfos, e os tempos de resposta foram registrados.

Os tempos são diferentes para cada par de grafos G_0 e G_1 mesmo com n, k e r constantes. De forma que, para uma análise clara dos tempos, foi feito um teste com 100 tentativas. Os parâmetros n, k e r são fixados e 100 pares de grafos são gerados aleatoriamente. Cada um dos pares é inserido no LAD e o tempo para a solução do isomorfismo é armazenado. Os resultados apresentados a seguir foram gerados para $n = 30$, $r = 10$ e k variando de 5 a 29. Para cada par de grafos, observamos um crescimento na média e na variância semelhante a curva de uma exponencial. Assim, levantamos a hipótese de que a média dos tempos com relação a k, $\mu(k)$, é uma curva exponencial em k. Para testar a hipótese procedemos da seguinte maneira:

- calculamos $\log \mu(k)$ e observamos uma função ainda crescente com uma leve curvatura para cima. Então concluímos se tratar de uma exponencial;
- comparamos o polinômio $p(k) = a(k-b)(k-c)(k-d)$ com $\log \mu(k)$ através de uma função fitness $f(k) = |p(k) - \log \mu(k)|$. Obtemos $a = 0,0006$, $b = -9,6097$, $c = 17,0825$ e $d = -27,7897$;
- como resultado obtemos

$$\mu(k) \approx 1,0006^{k^3+20,3169k^2-371,8246k-4561,9} \tag{3.1}$$

Na Figura 3.1 temos a curva do tempo médio dada em segundos. Percebemos que $k < 20$ o tempo médio é quase nulo. Mas para $k > 20$ começa um crescimento acentuado do tempo médio de solução. A curva de ajuste, obtida através de algoritmo genético, está bem próximo dos resultados experimentais com o k até 24. Mostrando assim, que a média do tempo esperado realmente tem um crescimento exponencial. Após o ponto máximo a função tem um rápido decaimento por conta do crescimento do grau do grafo, de forma que se aproximar de um grafo completo facilita a resolução do problema, pois encontrar um subgrafo em um grafo completo é um problema polinomial.

O mesmo ocorre com a variância. Usando um precedimento semelhante ao que foi usado para a média, testamos se a variância também apresenta um crescimento exponencial em k. Para testar a hipótese procedemos da seguinte maneira:

- calculamos $\log \mu(k)$ e observamos uma função ainda crescente com uma leve curvatura para cima. Então concluímos se tratar de uma exponencial;

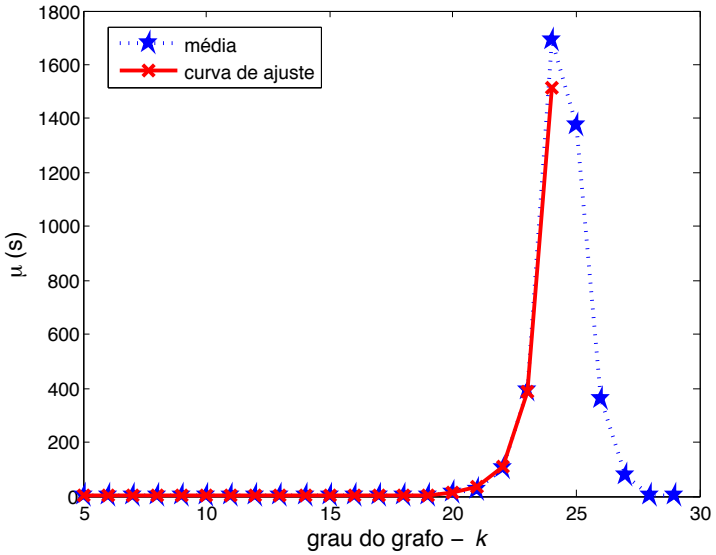


Figura 3.1 – Gráfico da média do tempo esperado para solução do isomorfismo usando o algoritmo LAD – a curva azul marcada com estrelas representa a variância calculada para $n = 30$, $r = 10$ e variando o k de 5 a 29. Em cada média calculada foram realizados 100 testes. A linha vermelha marcada com $\times$ é a curva de ajuste mostrada na equação 3.1.

- comparamos o polinômio $p(k) = a(k-b)(k-c)(k-d)$ com $\log var(k)$ através de uma função fitness $f(k) = |p(k) - \log var(k)|$. Obtemos $a = 0,0009$ $b = -23,8506$ $c = -20,0426$ $d = 16,4885$;

- A curva aproximada para a variância, através do algoritmo genético, foi

$$var(k) \approx 1,0009^{k^3+27,4047k^2-245,7050k-7882}. \tag{3.2}$$

Na Figura 3.2 a exponencial aproximada obtida através do algoritmo genético é inferior ao resultado experimental. Dando indícios de que a variância dos tempos medidos também apresenta um crescimento exponencial.

Para observar o formato da distribuição de probabilidade decidimos fazer o histograma e uma curva de ajuste para a função densidade de probabilidade (pdf). Muitas *pdf*'s com inclinação acentuada a esquerda da mediana podem ser usadas no ajuste para descrever a frequência dos tempos obtidos, mas optamos pela distribuição de Fisher-Snedecor por se ajustar melhor ao histograma (ver Figura 3.3). A F-pdf como é conhecida essa distribuição é dada a abaixo:

$$F(t, d_1, d_2) = \frac{1}{\mathrm{B}\left(\frac{d_1}{2}, \frac{d_2}{2}\right)} \left(\frac{d_1}{d_2}\right)^{\frac{d_1}{2}} t^{\frac{d_1}{2}-1} \left(1 + \frac{d_1}{d_2}\, t\right)^{-\frac{d_1+d_2}{2}}.$$

Os parâmetros $d_1 = 2,5056$ e $d_2 = 0,1536$ são para ajuste da curva e B é a função beta.

A distribuição apresentada é dita ter inclinação positiva. A cauda direita é mais longo, enquanto que a massa da distribuição fica concentrada no lado esquerdo da figura. Isso

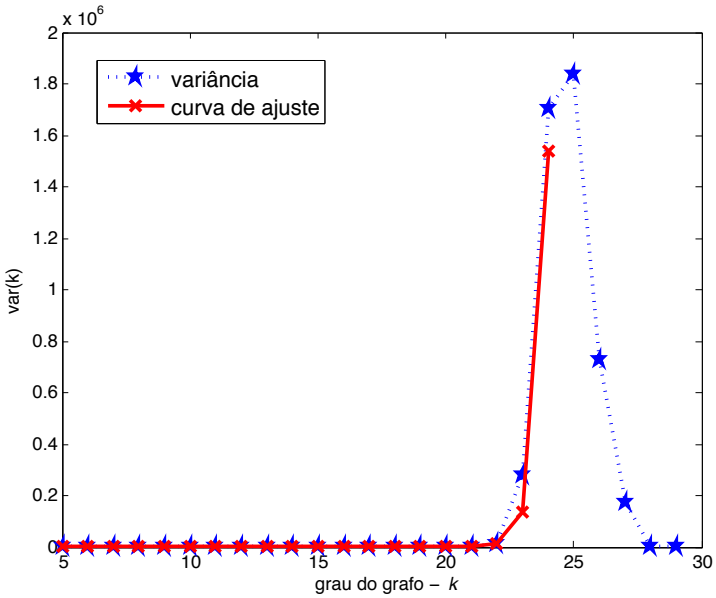


Figura 3.2 – Gráfico da variância do tempo esperado para solução do isomorfismo usando o algoritmo LAD – a curva azul marcada com estrelas representa a média calculada para $n = 30$, $r = 10$ e variando o k de 5 a 29. Em cada média calculada foram realizados 100 testes. A linha vermelha marcada com $\times$ é a curva de ajuste mostrada na equação 3.2.

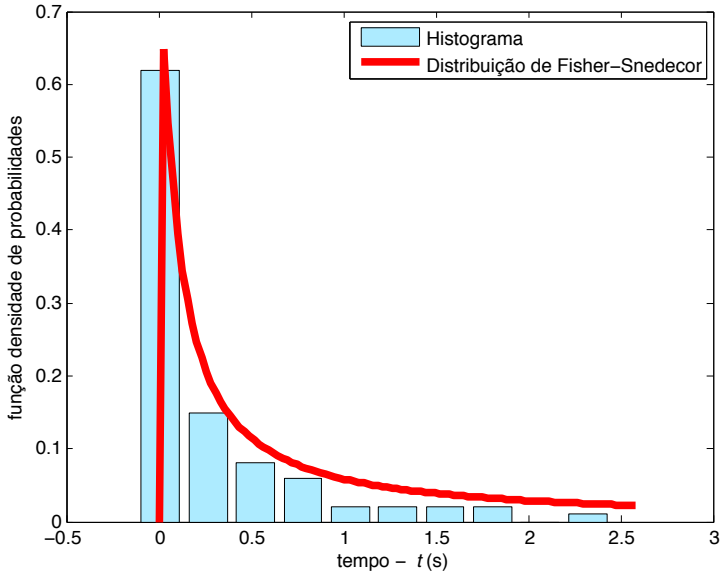


Figura 3.3 – Histograma e função densidade de probabilidade - Simulação do tempo de espera para $n = 30$, $k = 15$ e $r = 10$ com 100 testes.

nos mostra que para o tamanho do grafo escolhido, a maioria dos tempos observados são menores do que a mediana. Essa é a razão da distribuição apresentar uma inclinação para a esquerda.

A função distribuição cumulativa (cdf), ou probabilidade cumulativa, poderá nos dizer mais sobre as chances de sucesso do LAD descobrir o isomorfismo oculto. Pois ele nos dirá qual a probabilidade de o isomorfismo oculto ser descoberto até um determinado tempo t_0, Pr$(t \leq t_0)$. Na Figura 3.4 nós podemos ver a probabilidade de sucesso até $t_0 = 3.600s$, quando $n = 30$ e $r = 10$ para $k = 20,\ 23,\ 24,\ 25,\ 29$. Quando $k = 29$, temos que G_0 é um grafo

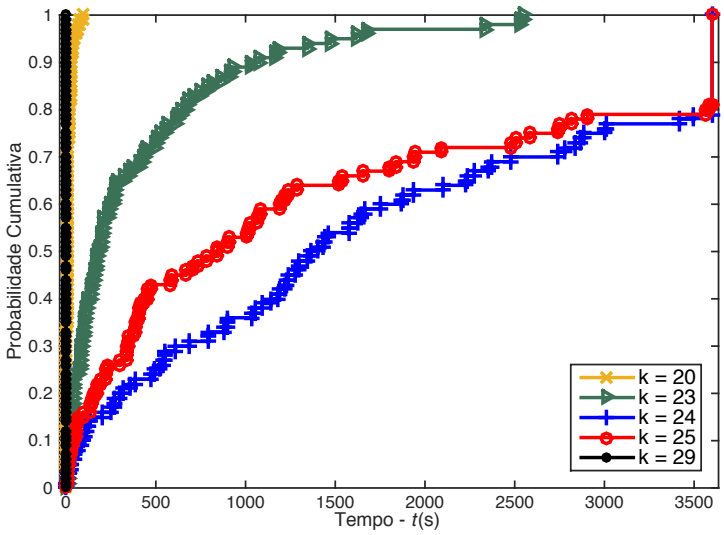


Figura 3.4 – Probabilidade cumulativa variando k.

completo, também conhecido como *clique*. Como qualquer isomorfismo aplicado a G_0 é o próprio G_0 e quaisquer 10 vértices retirados de G_0 fará que G_1 também seja um grafo completo de grau 19, então encontrar um isomorfismo neste caso é um problema trivial. Observamos também, que para $k \leq 20$, o LAD também resolve rapidamente qualquer dos 100 pares de grafos gerados. Para $k = 23$ já notamos o crescimento da dificuldade para o LAD. Somente após $t_0 \approx 2.500s$ é que o LAD poderá resolver descobrir qualquer isomorfismo oculto. Mas para $k = 24$ ou $k = 25$ não se pode garantir que o LAD descobrirá qualquer isomorfismo oculto com menos de uma hora (3.600s). Nestes dois casos a probabilidade de sucesso é 0.8, Pr$(t_0 \leq 3.600s) = 0.8$.

Um outro fator que altera a probabilidade de sucesso do LAD é o número de retiradas em H_0 (ver Algoritmo 1). O número de retiradas r determina o número de vértices do grafo G_1, $n - r$, e esse é um fator que também determinará o tempo esperado pelo LAD para encontrar qualquer isomorfismo oculto. Na Figura 3.5 apresentamos uma comparação das cdf's para $r = 0,\ 9,\ 10,\ 11$ fixando $n = 30$ e $k = 24$. Lembrando que para $r = 0$ tem-se um problema de isomorfismo de grafos. Para $r = 11$ temos que o tempo aguardado pelo LAD para encontra qualquer isomorfismo foi aproximadamente $4.300s$. Notamos que $r = 10$ diminui o sucesso do LAD ao máximo, pois em $r = 9$ as chances de sucesso melhoram um pouco por a

cdf para este caso possui valores superiores para a maiorias dos tempos com relação a $r = 10$.

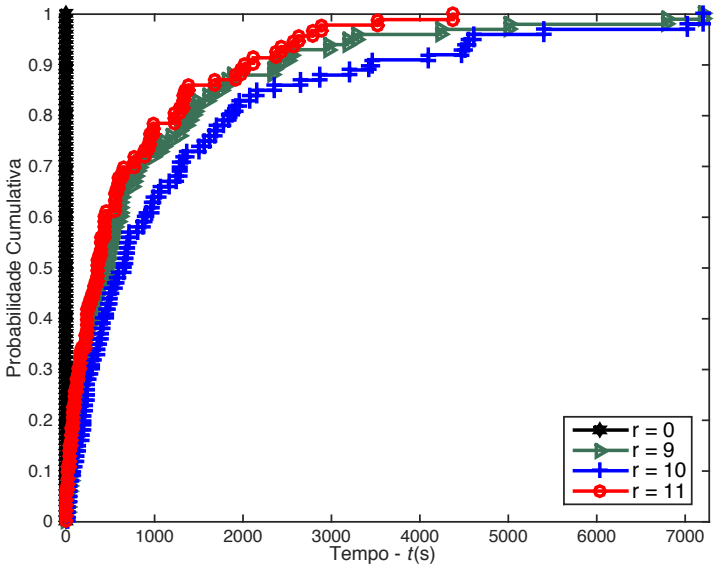


Figura 3.5 – Probabilidade cumulativa variando r

Foi feito um experimento com tempo máximo de resolução limitado a 7200 segundos. Aumentou-se o valor de da quantidade de vértices, do grau e do tamanho do subgrafo e foi medido a quantidade de vezes que o isomorfismo foi encontrado. A Figura 3.6 ilustra o decaimento da quantidade de vezes que o algoritmo consegue encontrar o isomorfismo. Para os valores que obedecem a proporção $k = 0,8 * n$ e $m = n/3$ a dificuldade é visivelmente maior, sendo os pontos de minimo da quantidade de resolução. Na Figura 3.6, é possível visualizar que para $n \geq 55$, o algoritmo não encontra nenhum isomorfismo no tempo de 7200 segundos.

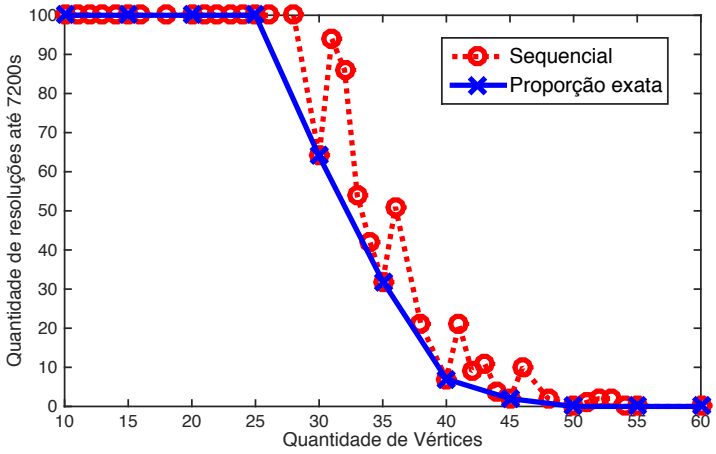


Figura 3.6 – Decaimento da quantidade de resoluções para o tempo de 7200s

3.4 Conclusões

Para uma real aplicação, deve-se ter um gerador de pares de grafos dentro do problema de isomorfismo de subgrafos cuja tarefa de descoberta do isomorfismo seja árdua. Nesse trabalho, propomos um gerador e apresentamos um estudo estatístico da segurança, atacando com o algoritmo LAD. Notamos que para grafos pequenos, até 30 vértices, a distribuição é inclinada a esquerda, a qual aproximamos por uma função densidade de Fisher-Snedecor. Também considerando $k = 0,8 * n$ e $m = n/3$, e aumentando a quantidade de vértices, podemos aumentar o tempo limite mantendo a probabilidade de sucesso do ataque próxima de zero, $Pr(t \leq T_{limite}) \leq P_{sucesso}$. Assim, podemos definir um gerador seguro em que as chances de sucesso de quebrar o isomorfismo se tornam quase nulas. Uma outra característica importante a ser destacada é o fato de que o gerador usa algorítimos pseudo aleatórios eficientes, ou seja, todos executáveis em tempo factíveis.

Nesse trabalho, não há uma conclusão sobre a forma da distribuição de probabilidade em função do tempo de espera para quebrar o isomorfismo quando grafos de muitos vértices são gerados. A razão para isso é que experimentos para testar tal hipótese não são praticáveis devido ao longo tempo de espera em cada tentativa. Mas conjecturamos que uma inclinação suave a esquerda deva aparecer nessa situação, pelo fato de que os casos fáceis começam a se tornar raros.

Capítulo 4

Sistema de prova de conhecimento nulo com PISG

Um algoritmo que é peça fundamental para muitos sistemas criptográficos é a prova de conhecimento nulo [5–7]. Uma prova de conhecimento é um método pelo qual uma das partes (o provador) pode provar a outra parte (o verificador) que uma declaração é verdadeira, sem transmitir qualquer informação adicional além do fato de que a afirmação é verdade. Através do trabalho [9] é bem conhecido que existem provas de conhecimento nulo para qualquer problema NP, desde que as funções alçapão existam. Ainda em [9] são mostrados sistemas de prova de conhecimento nulo que não necessitam de mensagens cifradas. Estes sistemas são baseados no problema de isomorfismo e não-isomorfismo de grafos. Como não se conheciam a mesma quantidade de propriedades nos problemas de isomorfismo de grafos que se conheciam nos problemas baseados em aritmética modular, com o passar do tempo poucos artigos foram surgindo e o problema de isomorfismo de grafos não chegou a ser empregado na prática.

Outro motivo que favoreceu o não uso do problema de isomorfismo de grafos foi o desconhecimento da classe de complexidade computacional a que ele pertenceria, pois ele não estava na classe dos problemas polinomiais, mas também não estava na classe NP-Completo. Por isso foi criada uma classe IG para investigar somente esse problema [10, 11]. Depois surgiram algoritmos que reduzem significativamente a complexidade da solução, embora o problema ainda não possa ser resolvido em tempo polinomial [12], [13]. Assim, puseram sob suspeita os sistemas de prova de conhecimento nulo baseados no problema de isomorfismo de grafos.

Uma vantagem que os problemas com grafos têm sobre os problemas com aritmética modular é o tempo de execução. Isso desperta o interesse para aplicações em hardware mais simples [58]. No artigo [14] foi relatado o uso de um algoritmo de prova de conhecimento nulo baseado em isomorfismo de subgrafos. Este é um problema NP-Completo e o tempo de espera para encontrar o isomorfismo oculto é alto no caso árduo [27], [66], [35]. No entanto, não foi fundamentada a segurança nem o método de geração de grafos. O que exige uma pesquisa que

melhor fundamente a sua aplicação. Embora seja conhecido que o problema SGI apresente um excelente método para assinatura de mensagens que pode ser implementado em microcontroladores [58], nem todo problema de isomorfismo de subgrafos é NP-Completo. Existem problemas em P como podemos encontrar em [67], [68], [69], [70]. Se o problema GI fosse mais geral que o SGI, esta análise não seria necessária, pois seria apenas um caso particular a aplicação do problema. Mas isso não é verdade. O SGI é um problema mais geral que o problema GI.

Neste Capítulo é mostrado uma análise de segurança do perfeito sistema de prova de conhecimento nulo para o problema de isomorfismo de subgrafos. A análise de segurança feita segue o caminho apresentado em [9] para o isomorfismo de grafos.

4.1 Complexidade de problemas e prova de conhecimento nulo

Para medir a eficiência de um algoritmo é necessário usar o tempo teórico que o programa leva para encontrar uma resposta em função dos dados de entrada. Se a dependência do tempo com relação aos dados de entrada for polinomial com o comprimento da entrada p($|x|$), o programa será considerado rápido. Caso a dependência do tempo seja exponencial com o comprimento da entrada ($a^{|x|}$ para a > 0), então o programa é considerado lento.

Stephen Cook [19] observou um fato simples: se um problema pode ser resolvido em tempo polinomial, então, pode-se também verificar se uma possível solução é correta em tempo polinomial. Uma maneira mais formal de expressar essa ideia é através da descrição do problema de decisão. Um problema de decisão pode ser visto como um problema de reconhecimento de linguagem. Considere $\sum^*$ como sendo o conjunto de todas as possíveis entradas para um problema de decisão. Considere $L \subseteq \sum^*$ como sendo o conjunto de todas as entradas cuja resposta é sim. Esse conjunto é chamado de linguagem correspondente ao problema. Em [19], Cook definiu classes de problemas quanto à complexidade destes e percebeu que alguns problemas não admitiam uma simplificação polinomial no seu tempo de execução, mas podiam ser certificados em tempo polinomial. Então, Cook definiu a classe dos problemas NP ou classe de complexidade NP como aquela para a qual apenas pode-se verificar, em tempo polinomial, se uma possível solução é correta.

Algoritmos eficientes para transportar ("traduzir") de uma linguagem L a outra linguagem L' podem ser criados. Assim, diz-se que L é polinomialmente redutível para uma linguagem L'. Nesse caso, simplesmente expressamos por $L \subseteq_p L'$ (lê-se L é p-redutível a L'). Leonid Levin [20] e Stephen Cook [19] observaram que dentre os problemas NP existem alguns que são mais difíceis do que outros, no sentido de que, resolvendo um desses problemas em tempo polinomial, então, todos os problemas em NP também podem ser resolvidos em tempo polinomial. Assim, a classe dos problemas NP-completos é o subconjunto dos mais

difíceis problemas não-determinísticos polinomiais.

A assimetria entre a complexidade da tarefa de verificação e a complexidade da tarefa do ato de provar faz com que a classe NP seja vista como um sistema de prova. Basta notar que o processo de verificação é fácil, enquanto que chegar até a prova é uma tarefa difícil [23]. Duas propriedades de um sistema de prova são a sua validade e a completude. A propriedade de validade afirma que *o verificador deve possuir a habilidade de se proteger contra argumentos falsos para não ser convencido por eles*. Por outro lado, a propriedade de completude afirma à *capacidade de um provador convencer sempre com sentenças verdadeiras*.

Definição 2: Sistema de Prova Interativo. *Um par de máquinas interativas (**P,V**) é chamado um sistema de prova interativo para uma linguagem L, se a máquina **V** tem eficiência de tempo polinomial e satisfaz as seguintes condições:*

1. (*Completude*) *Para todo* $x \in L$,

$$Pr[(\boldsymbol{P,V})(x) = 1] > \frac{2}{3};$$

2. (*Validade*) *Para todo* $x \notin L$ *e um provador interativo* $\boldsymbol{P'}$, então

$$Pr[(\boldsymbol{P',V})(x) = 1] < \frac{1}{3}.$$

É evidente que a definição dos limites para a satisfação das propriedades de completude e validade podem ser modificados. De forma geral, o sistema de prova interativo tem completude limitada por probabilidade p_c e tem validade limitada por p_v.

Definição 2: Perfeito Conhecimento Nulo. *Seja (**P,V**) um sistema de prova interativa para alguma linguagem L. Diz-se que (**P,V**) ou de fato a máquina do provador **P** tem perfeito conhecimento nulo se para toda máquina interativa probabilística de tempo polinomial **V**, existe um algoritmo S, tal que para toda entrada* $x \in L$*, as seguintes duas variáveis aleatórias são identicamente distribuídas:*

1. (***P,V***)(*x*)*, ou seja, a saída da máquina interativa **V** após interagir com a máquina **P** na entrada x;*

2. *S*(*x*)*, ou seja, a saída da máquina S na entrada x. A máquina S é chamada de simulador da interação de **V** com **P**.*

O real ganho de conhecimento de habilidade computacional é caracterizado quando uma máquina, após interagir com outra parte, é capaz de computar algo que antes ela não era capaz. No caso de nenhum ganho de conhecimento computacional, tem-se que, com o algoritmo

$S(x)$, após a interação entre as máquinas ***V*** e ***P***, a máquina ***V*** não é capaz de fazer computações além do que ela era capaz de fazer sozinha antes da interação.

4.2 Sistema interativo de prova de conhecimento nulo para subgrafos isomorfos

Agora suponha que Paulo, um provador, que denotaremos por **P** conheça o isomorfismo entre um grafo G_0 com n vértices e um subgrafo, G_1, com m vértices, tal que $n > m$. Assim, temos uma função de correspondência de G_0 a G_1, $\sigma : G_0 \rightarrow G_1$. Sabendo que o isomorfismo de subgrafos é um problema NP-Completo, considere n e m suficientemente grandes, tal que, qualquer outra máquina probabilística de tempo polinomial não encontre o isomorfismo entre G_0 e G_1 em tempo hábil. Assim, Paulo guardará como segredo a transformação $\sigma : G_0 \rightarrow G_1$ e dirá a Victor, um verificador, denotado pela letra **V**, que conhece esse segredo. Para provar a **V** de que ela está falando a verdade sem revelar o segredo, ela terá que provar em tempo hábil desafios lançados por **V** como descrito no seguinte algoritmo:

Protocolo 1 (Sistema de prova de conhecimento nulo para isomorfismos de subgrafos) - O seguintes passos são repetidos n vezes:

1. **P** gera a função de correspondência $\lambda : G_0 \rightarrow H$ e envia H para **V** (λ é uma permutação e H é um grafo isomorfo a G_0, que é, consequentemente, subgrafo isomorfo a G_1);

2. **V** gera aleatoriamente um bit b e envia o bit a **P**;

3. **P** envia a função de correspondência $\xi_b = \sigma^b \circ \lambda$ para **V**;

4. Se o valor de $b = 0$, então o verificador usa a função de correspondência $\xi_0 = \lambda$ para testar se H é isomorfo a G_0. Se $b = 1$, então o verificador usa a função de correspondência $\xi_1 = \sigma \circ \lambda$ para testar se H é um subgrafo isomorfo de G_1.

Quando se olha o protocolo acima como um sistema de prova interativo, percebe-se que a entrada desse sistema é o par de grafos isomorfos G_0 e G_1. Então, a sentença dessa linguagem é $(G_0, G_1) \in SGI$. Notamos que a verificação dessa sentença sempre é feita indiretamente através de $\sigma^b \circ \lambda : H \rightarrow G_b$. Portanto, a função de correspondência σ sempre será um segredo mantido pelo provador.

Quanto a eficiência de execução do Protocolo 1, o programa do provador é executado por uma máquina probabilística de tempo polinomial, ou seja, apenas uma permutação escolhida aleatoriamente é realizada no Passo 1. O programa do verificador pode ser executado

em tempo polinomial determinístico no passo 4. Denotaremos esse programa por:

$$V(H, G_b, \xi) = \begin{cases} True & \text{se} \quad \xi : H \to G_b \\ False & \text{se} \quad \xi : H \to \tilde{H} \neq G_b \end{cases}$$

A escolha do bit de desafio é uma simples computação probabilística no Passo 2 e a verificação é $O(n^2)$ independente do bit escolhido. Assim, queremos verificar se a linguagem SGI tem um perfeito sistema de prova de conhecimento nulo.

Proposição 1 A linguagem SGI tem um perfeito sistema de prova interativa com conhecimento nulo. Para verificadores limitados por máquinas de tempo polinomial (determinística ou probabilística), o Protocolo 1 satisfaz as seguintes afirmativas:

1. Se G_1 é um grafo isomorfo a um subgrafo contido em G_0, então o verificador sempre aceita quando interage com $\mathbf{P}$;

2. Se G_1 não é um grafo isomorfo a um subgrafo contido em G_0, então a entrada será rejeitada com probabilidade menor do que $\frac{1}{2}$;

3. $\mathbf{P}$ realiza provas com perfeito conhecimento nulo.

Quando n interações entre o verificador e o provador são realizadas a probabilidade de erro para a validade é limitada por $\frac{1}{2^n}$. Deve ser enfatizado que todas as computações probabilísticas são completamente independentes para cada iteração das máquinas probabilísticas, consequentemente, isto é válido para interações entre as máquinas. Portanto, segue a prova da Proposição 1:

1. **Prova:** Claramente, se o grafo G_1 é um grafo isomorfo a um subgrafo de G_0, então o grafo H construído por $\mathbf{P}$ no passo 1 sempre retornará True na no passo 4, $V(H, G_0, \xi_0) = V(H, G_1, \xi_1) = True$. Consequentemente se cada parte segue o que está prescrito no protocolo, então $\mathbf{V}$ sempre aceita os argumentos do provador.

2. **Prova:** Já sabemos que um provador desonesto não pode fazer provas. Mas o verificador pode vir a ser um futuro provador desonesto após uma real interação com o provador? A resposta é não se o verificador, sem qualquer interação com o provador, conseguir apenas tuplas que podem ser obtidas através de uma simulação com baixo esforço computacional. O baixo esforço computacional a que nos referimos é um algoritmo de tempo polinomial em máquina probabilística. Neste caso, considerando um verificador desonesto, o que deve ser demonstrado é que o provador realiza provas sem que o verificador tenha qualquer aprendizado (ganho de conhecimento) sobre suas habilidades computacionais [71].

3. **Prova:** Já sabemos que um provador desonesto não pode fazer provas. Mas o verificador pode vir a ser um futuro provador desonesto após uma real interação com o provador? A resposta é não, se o verificador, sem qualquer interação com o provador, conseguir apenas tuplas que podem ser obtidas através de uma simulação com baixo esforço computacional. O baixo esforço computacional a que nos referimos é um algoritmo de tempo polinomial em máquina probabilística. Neste caso, considerando um verificador desonesto, o que deve ser demonstrado é que o provador realiza provas sem que o verificador tenha qualquer aprendizado (ganho de conhecimento) sobre suas habilidades computacionais [71]. Assim, de posse de um simulador, este terá que produzir n tuplas (H, b, ξ) onde H é gerado uniformemente (pois o provador é honesto) e b é gerado de acordo com $\mathbf{V}$. Então uma possível simulação pode ser realizada da seguinte forma:

   (a) O simulador escolhe $b \in \{0, 1\}$ de forma uniformemente aleatória;

   (b) O simulador escolhe uma permutação $\xi : G_b \rightarrow H$ de maneira uniformemente aleatória. Se $b = 0$, então ξ será uma permutação sobre n vértices. Caso contrário, ξ será uma permutação sobre m vértices;

   (c) O simulador aplica um algoritmo probabilístico de tempo polinomial, $V_1(H, w_1)$, em que $w_1 = \epsilon$ significa sem símbolo inicial, e verifica se o bit c calculado por V_1 é igual a b;

      i. Caso $c = b$, então a simulação foi feita com sucesso e a tupla nesta interação deve ser (H, c, ξ). A computação auxiliar feita por $\mathbf{V}$ com o fim de ser utilizada em interações futuras é gravada em w_1;

      ii. Se $c \neq b$, então o simulador volta para o Passo (a).

   Após a primeira iteração $(k > 1)$, no Passo (c), o simulador aplica $V_k(H, w_k)$, um algoritmo probabilístico de tempo polinomial para escolha do bit c de desafio na k-ésima iteração que usa dados auxiliares $w_k \in \{0, 1\}^*$ obtidos em computações anteriores, e verifica se o bit c calculado por V_k é igual a b;

      i. Caso $c = b$, então a simulação foi feita com sucesso e a tupla nesta interação deve ser (H, c, ξ). A computação auxiliar feita por $\mathbf{V}$ com o fim de ser utilizada em interações futuras é gravada em w_{k+1};

      ii. Se $c \neq b$, então o simulador volta para o Passo (a).

A principal diferença no algoritmo de simulação para gerar as tuplas do protocolo de prova de conhecimento nulo baseado SGI para o de GI em [71] é o passo b). A escolha do bit b no passo a) altera a entrada do gerador aleatório da permutação, definindo o tamanho sobre qual conjunto de vértices a permutação ocorrerá. Deixando-o dependente do passo anterior. Essa alteração não aumenta a complexidade da simulação. No demais, as consequências

são as mesmas. O verificador não ganha nenhum conhecimento adicional e o provador realiza um perfeita prova de conhecimento nulo. O objetivo de V_k e w_k é escolher o bit c de forma a minimizar as chances de fracasso no passo ii). Podemos imaginar que o algoritmo V_k sempre terá mais chances do que V_{k-1} e que o dado w_k sempre será mais útil do que w_{k-1}. Mas para ter sucesso nessa simulação, não serão necessários tantos melhoramentos nesses algoritmos e dados. Pois no pior caso, em que os melhoramentos de V_k e w_k em cada iteração não acontecem, a probabilidade de $c = b$ é $\frac{1}{2}$, dado que b foi escolhido com distribuição de probabilidade uniforme e c poderá ser escolhido da mesma maneira. Mesmo nesta situação, o simulador terá sucesso, em média, após duas tentativas. Para finalizar, basta verificar que a sequência de tuplas (H, c, ξ) gerada pelo simulador tem exatamente a mesma distribuição que as tuplas produzidas por uma interação real com o provador. Por esta razão, estas sequências são computacionalmente indistinguíveis.

4.3 Conclusões

Foi verificado que sistemas de prova de conhecimento nulo baseado no problema de isomorfismo de subgrafos garante três importantes pontos de segurança: o verificador sempre aceita os argumentos do provador quando interage com ele, falsos isomorfismos não serão aceitos após n interações e que $\mathbf{P}$ realiza provas com perfeito conhecimento nulo. Além disso, temos um sistema de prova de conhecimento nulo baseado num problema NP-Completo sem a necessidade de cifra segura. Isso viabiliza a proposta [14], cuja motivação foi a facilidade de implementação desse algoritmo em máquinas de baixo poder computacional. Foi verificado que a impossibilidade de transferência de prova ou perfeito conhecimento nulo é uma propriedade muito bem preservada com problemas de isomorfismo de subgrafos, pois as modificações feitas no simulador não alteraram a sua complexidade. Além disso, fato de um problema ser verificável em tempo polinomial, mas cuja demonstração exige a solução de um problema NP-Completo, fortalece a completude e a validade dos argumentos apresentados.

Conclusão e Trabalhos Futuros

Pode ser verificado que é possível utilizar o problema de isomorfismo de subgrafos como prova para sistemas de conhecimento nulo. Foi proposto um gerador que para os métodos de resolução do problema de isomorfismo de subgrafos atuais, garante as instâncias que não serão resolvidas em tempo polinomial com uma probabilidade confiável. Aplicando os devidos parâmetros, tem-se uma confiabilidade de tempo para encontrar o isomorfismo que pode ser ajustada de acordo com o nível de segurança desejado.

Como trabalho futuro, deve-se fazer uma análise mais aprofundada do gerador e estimar a confiabilidade para valores que possam ser aplicados em autenticações usuais, como por exemplo grafos com 512, 1024, 2048 vértices. Também deve-se verificar qual a estimativa de tempo que um supercomputador atual poderia resolver o problema para os parâmetros citados ou algum outro que seja desejado. Além disso, deve ser avaliado o espaço em memória que será necessário para armazenar os grafos e quanto de banda é gasto na transmissão de dados do algoritmo.

REFERÊNCIAS

[1] DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. *IEEE Trans. Inf. Theor.*, Piscataway, NJ, USA, v. 22, n. 6, p. 644–654, Nov. 1976.

[2] SHANNON, C. E. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, v. 28, p. 656–715, 1949.

[3] RIVEST, R.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, New York, NY, USA, v. 21, n. 2, p. 120–126, 1978.

[4] GOLDWASSER, S.; MICALI, S.; RIVEST, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, Philadelphia, PA, USA, v. 17, n. 2, p. 281–308, Abril. 1988.

[5] FIAT, A.; SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. *Advances in Cryptology - CRYPTO'86*, v. 163, p. 186–197, 1987.

[6] CRAMER, R.; GENNARO, R.; SCHOENMAKERS, B. A secure and optimally efficient multi-authority election scheme. In: . Springer-Verlag, c1997. p. 103–118.

[7] CHAUM, D.; FIAT, A.; NAOR, M. Untraceable electronic cash. In: . CRYPTO '88. New York, NY, USA: Springer-Verlag New York, Inc., c1990. p. 319–327.

[8] LENSTRA, A. K.; HUGHES, J. P.; AUGIER, M.; BOS, J. W.; KLEINJUNG, T.; WACHTER, C. Ron was wrong, whit is right. *IACR Cryptology ePrint Archive*, v. 2012, p. 64, 2012. informal publication.

[9] GOLDREICH, O.; MICALI, S.; WIGDERSON, A. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, New York, NY, USA, v. 38, n. 3, p. 690–728, 1991.

[10] BOOTH, K. S.; COLBOURN, C. S. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, Computer Science Department, University of Waterloo, Waterloo, Ontario, 1977.

[11] KöBLER, J.; SCHöNING, U.; TORáN, J. *The graph isomorphism problem: its structural complexity*. Basel, Switzerland, Switzerland: Birkhauser Verlag, 1993.

[12] CORDELLA, L. P.; FOGGIA, P.; SANSONE, C.; VENTO, M. Performance evaluation of the vf graph matching algorithm. In: . ICIAP '99. Washington, DC, USA: IEEE Computer Society, c1999. p. 1172–.

[13] MCKAY, B. D. Practical Graph Isomorphism. *Congressus Numerantium*, v. 30, p. 45–87, 1981.

[14] J. M. KIZZA, L. B. E. E. M. Using Subgraph Isomorphism as a Zero Knowledge Proof Authentication in Timed Wireless Mobile Networks. *International Journal of Computing and ICT Research*, v. 4, 2010.

[15] TURING, A. M. On computable numbers: With an application to the entscheidungsproblem. In: . Mathematical Society, c1936. p. 230–265.

[16] COBHAM, A. The intrinsic computational difficulty of functions. In: . Elsevier, c1964. p. 24–30.

[17] EDMONDS, J. Minimum partition of a matroid into independent subsets. In: . c1965. v. 69. p. 67–72.

[18] NEUMANN, J. V. *A certain zero-sum two-person game equivalent to the optimal assignment problem*. Princenton University Press, 1953. v. II.

[19] COOK, S. A. The complexity of theorem-proving procedures. In: . New York, NY, USA: ACM, c1971. p. 151–158.

[20] LEVIN, L. A. Universal sequential search problems. *Jour Probl. Peredachi Inf.*, v. 9, p. 115–116, 1973.

[21] P. FEOFILOFF, Y. KOHAYAKAWA, Y. W. Uma introdução sucinta à teoria dos grafos, jun 2011.

[22] NETO, P. O. B. *Grafos: Teoria, modelos, algoritmos*. São Paulo, SP, Brasil: Editora Edgard Blücher, 2004.

[23] GOLDREICH, O. *Foundations of cryptography: Basic tools*. Cambridge University Press, 2001. v. 1.

[24] GAREY, M. R.; JOHNSON, D. S. *Computers and intractability: A guide to the theory of np-completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. GT48.

[25] KULCZYCKI, D. G. Computation and formal languages. Technical report, Department of Computer Science Virginia Tech, 2006.

[26] KNUTH, D. E. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, Stanford, CA, USA, v. 29, 1975.

[27] ULLMANN, J. R. An algorithm for subgraph isomorphism. *J. ACM*, New York, NY, USA, v. 23, n. 1, p. 31–42, Jan. 1976.

[28] CORDELLA, L. P.; FOGGIA, P.; SANSONE, C.; VENTO, M. Performance evaluation of the vf graph matching algorithm. In: . ICIAP '99. Washington, DC, USA: IEEE Computer Society, c1999. p. 1172–.

[29] CORDELLA, L. P.; FOGGIA, P.; SANSONE, C.; VENTO, M. An improved algorithm for matching large graphs. In: . c2001. p. 149–159.

[30] P. CORDELLA, L.; FOGGIA, P.; SANSONE, C.; VENTO, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, Washington, DC, USA, v. 26, n. 10, p. 1367–1372, Oct. 2004.

[31] SHANG, H.; ZHANG, Y.; LIN, X.; YU, J. X. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, v. 1, n. 1, p. 364–375, Aug. 2008.

[32] HE, H.; SINGH, A. K. Graphs-at-a-time: Query language and access methods for graph databases. In: . SIGMOD '08. New York, NY, USA: ACM, c2008. p. 405–418.

[33] ZHANG, S.; LI, S.; YANG, J. Gaddi: Distance index based subgraph matching in biological networks. In: . EDBT '09. New York, NY, USA: ACM, c2009. p. 192–203.

[34] ZHAO, P.; HAN, J. On graph query optimization in large networks. *Proc. VLDB Endow.*, v. 3, n. 1-2, p. 340–351, Sept. 2010.

[35] SOLNON, C. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, Essex, UK, v. 174, n. 12-13, p. 850–864, Aug. 2010.

[36] FOGGIA, P.; SANSONE, C.; VENTO, M. A performance comparison of five algorithms for graph isomorphism. In: . c2001. p. 188–199.

[37] LEE, J.; HAN, W.-S.; KASPEROVICS, R.; LEE, J.-H. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: . PVLDB'13. VLDB Endowment, c2013. p. 133–144.

[38] LINGAS, A.; SYSLO, M. M. A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs. In: . Editors LEPISTö, T.; SALOMAA, A. Springer, c1988. v. 317 of *Lecture Notes in Computer Science*. p. 394–409.

[39] LINGAS, A. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. *Theoretical Computer Science*, v. 63, n. 3, p. 295 – 302, 1989.

[40] DESSMARK, A.; LINGAS, A.; PROSKUROWSKI, A. Faster algorithms for subgraph isomorphism of k-connected partial k-trees. *Algorithmica*, v. 27, p. 501–513, 1996.

[41] DAMASCHKE, P. Induced subraph isomorphism for cographs is np-complete. In: MöHRING, R. (Ed.) *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, 1991. v. 484 of *Lecture Notes in Computer Science*, p. 72–78.

[42] EPPSTEIN, D. Subgraph isomorphism in planar graphs and related problems. In: . c1995. p. 632–640.

[43] DORN, F. Planar subgraph isomorphism revisited. *CoRR*, v. abs/0909.4692, 2009.

[44] KIJIMA, S.; OTACHI, Y.; SAITOH, T.; UNO, T. Subgraph isomorphism in graph classes. *Discrete Mathematics*, v. 312, n. 21, p. 3164 – 3173, 2012.

[45] GUPTA, A.; NISHIMURA, N. The complexity of subgraph isomorphism for classes of partial k-trees. *Theoretical Computer Science*, v. 164, n. 1–2, p. 287 – 298, 1996.

[46] DAMIAND, G.; DE LA HIGUERA, C.; JANODET, J.-C.; SAMUEL, E.; SOLNON, C. A polynomial algorithm for submap isomorphism. In: TORSELLO, A.; ESCOLANO, F.; BRUN, L. (Eds.) *Graph-Based Representations in Pattern Recognition*. Springer Berlin Heidelberg, 2009. v. 5534 of *Lecture Notes in Computer Science*, p. 102–112.

[47] HEGGERNES, P.; MEISTER, D.; VILLANGER, Y. Induced subgraph isomorphism on interval and proper interval graphs. In: CHEONG, O.; CHWA, K.-Y.; PARK, K. (Eds.) *Algorithms and Computation*. Springer Berlin Heidelberg, 2010. v. 6507 of *Lecture Notes in Computer Science*, p. 399–409.

[48] MARX, D.; SCHLOTTER, I. Cleaning interval graphs. *Algorithmica*, v. 65, n. 2, p. 275–316, 2013.

[49] HEGGERNES, P.; VAN'T HOF, P.; MILANIČ, M. Induced subtrees in interval graphs. In: *Combinatorial Algorithms*. Springer, 2013. p. 230–243.

[50] KONAGAYA, M.; OTACHI, Y.; UEHARA, R. Polynomial-time algorithms for subgraph isomorphism in small graph classes of perfect graphs. In: GOPAL, T.; AGRAWAL, M.;

LI, A.; COOPER, S. (Eds.) *Theory and Applications of Models of Computation*. Springer International Publishing, 2014. v. 8402 of *Lecture Notes in Computer Science*, p. 216–228.

[51] CHUDNOVSKY, M.; ROBERTSON, N.; SEYMOUR, P.; THOMAS, R. The strong perfect graph theorem. *ANNALS OF MATHEMATICS*, v. 164, p. 51–229, 2006.

[52] CORNUEJOLS, G.; LIU, X.; VUšKOVIć, K. A polynomial algorithm for recognizing perfect graphs. In: . c2003. p. 20–27.

[53] CHVáTAL, V.; HAMMER, P. L. Aggregation of inequalities in integer programming. In: P.L. HAMMER, E.L. JOHNSON, B. K.; NEMHAUSER, G. (Eds.) *Studies in Integer Programming*. Elsevier, 1977. v. 1 of *Annals of Discrete Mathematics*, p. 145 – 162.

[54] HEGGERNES, P.; KRATSCH, D. Linear-time certifying recognition algorithms and forbidden induced subgraphs. *Nord. J. Comput.*, v. 14, n. 1-2, p. 87–108, 2007.

[55] GOLUMBIC, M. C. *Algorithmic graph theory and perfect graphs (annals of discrete mathematics, vol 57)*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 2004.

[56] ROSE, D.; TARJAN, R.; LUEKER, G. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, v. 5, n. 2, p. 266–283, 1976.

[57] RÉGIN, J.-C. A filtering algorithm for constraints of difference in csps. In: . c1994. v. 94. p. 362–367.

[58] SZöLLöSI, L.; MAROSITS, T.; FEHÉR, G.; RECSKI, A. Fast digital signature algorithm based on subgraph isomorphism. In: . CANS'07. Berlin, Heidelberg: Springer-Verlag, c2007. p. 34–46.

[59] SZöLLöSI, FEHéR, G.; MAROSITS, T. Proper key generation for the izosign algorithm. In: . Editors FERNáNDEZ-MEDINA, E.; MALEK, M.; HERNANDO, J. INSTICC Press, c2008. p. 368–372.

[60] FILOTTI, I. S.; MAYER, J. N. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In: . c1980. p. 236–243.

[61] MILLER, G. L. Isomorphism testing for graphs of bounded genus. In: . c1980. p. 225–235.

[62] LUKS, E. Isomorphism of bounded valence can be tested in polynomial time. v. 25, p. 42–65, 1982.

[63] BABAI, L.; YU, D.; GRIGORYEV, Y.; MOUNT, D. Isomorphism of graphs with bounded eigenvalue multiplicity. In: . c1982. p. 310–324.

[64] GUPTA, A.; NISHIMURA, N. The complexity of subgraph isomorphism for classes of partial k-tress. *Theoretical Computer Science*, v. 164, n. 1-2, p. 287–298, 1996.

[65] CSÁRDI, G.; NEPUSZ, T. The igraph software package for complex network research. *InterJournal Complex Systems*, v. 1695, 2006.

[66] ULLMANN, J. R. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *J. Exp. Algorithmics*, New York, NY, USA, v. 15, p. 1.6:1.1–1.6:1.64, Feb. 2011.

[67] RAVIV, D.; KIMMEL, R.; BRUCKSTEIN, A. M. Graph isomorphisms and automorphisms via spectral signatures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Los Alamitos, CA, USA, v. 35, n. 8, p. 1985–1993, 2013.

[68] Editors PUECH, C.; REISCHUK, R. Springer, c1996. v. 1046 of *Lecture Notes in Computer Science*.

[69] GUPTA, A.; NISHIMURA, N. Characterizing the complexity of subgraph isomorphism for graphs of bounded path-width. In: . Editors PUECH, C.; REISCHUK, R. Springer, c1996. v. 1046 of *Lecture Notes in Computer Science*. p. 453–464.

[70] EPPSTEIN, D. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms & Applications*, Irvine, CA, 92697-3425, USA, v. 3, p. 1–27, 1999.

[71] MATEUS, P. Análise de sistemas de prova de conhecimento nulo. *Portuguese IBM Scientific Prize*, 2005.

Apêndice A

Distribuição Exponencial

A distribuição exponencial possibilita analisar o intervalo de ocorrência do evento, o tempo necessário para completar uma tarefa ou a duração de um equipamento. A variável aleatória x, que é igual à distância entre contagens sucessivas de um processo de Poison, com média $\lambda > 0$, é uma variável aleatória exponencial com parâmetro λ. A distribuição exponencial tem esse nome por causa da função exponencial na função densidade de probabilidade. Gráficos da distribuição exponencial para valores selecionados de λ são mostrados nas Figura A.1 e Figura A.2. Para qualquer valor de λ a distribuição exponencial é bem distorcida.
A função densidade de probabilidade de x é:

$$f(x) = \lambda e^{-\lambda x}, 0 \leq x < \infty.$$

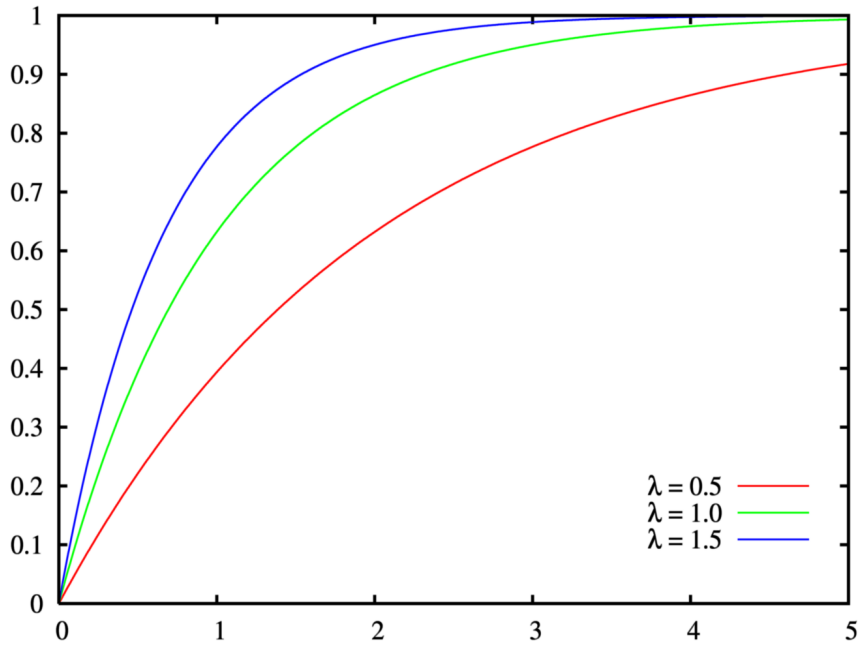
Valor esperado:

$$\mu = E(x) = \frac{1}{\lambda}.$$

Variância:

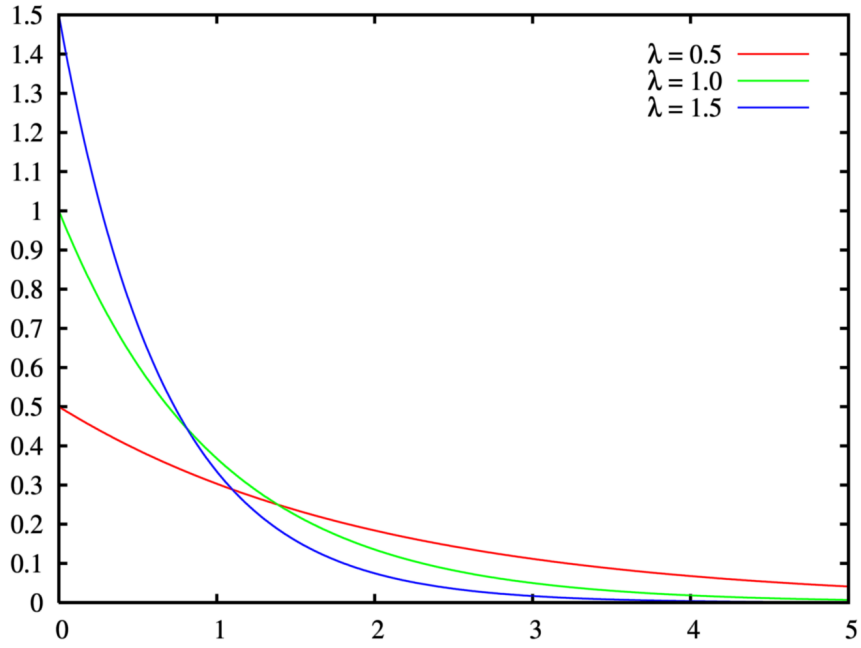
$$\sigma^2 = V(x) = \frac{1}{\lambda^2}.$$

Figura A.1 – Gráfico da distribuição Exponencial CDF



Fonte: `http://pt.wikipedia.org/wiki/Distribuicao_exponencial`

Figura A.2 – Gráfico da distribuição Exponencial PDF



Fonte: `http://pt.wikipedia.org/wiki/Distribuicao_exponencial`

Apêndice B

Função Densidade de Probabilidade - PDF

Para uma variável contínua X a função densidade de probabilidade é tal que:

$$F(x) = P(a \leq X \leq b) = \int_a^b f(x)dx$$

Uma função densidade de probabilidade de $f(x)$ satisfaz as seguintes propriedades:

- $F(x) \geq 0$
- $\int_{-\infty}^{\infty} f(x)dx = 1$
- $0 \leq P(a \leq X \leq b) \leq 1$

A probabilidade de um evento acontecer no intervalo a, b é dada pela área sob a curva da função densidade de probabilidade. A área total sob a curva é 1. Deve-se observar que $\int_a^a f(x)dx = 0$, ou seja, a probabilidade em um ponto específico é zero. Nessa interpretação tem-se que: $P(a \leq X \leq b) = P(a \leq X \leq b) = P(a \leq X \leq b) = P(a \leq X \leq b)$.

Apêndice C

Função de Distribuição Cumulativa - CDF

A função de distribuição cumulativa ou acumulada de uma variável X retorna a probabilidade acumulada até o valor de x. Definida por:

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(u)du$$

$F(x)$ satisfaz as propriedades:

- $0 \leq F(x) \leq 1$
- Se $x \leq y$, $F(x) \leq F(y)$
- $\lim_{x \to -\infty} F(x) = 0$
- $\lim_{x \to \infty} F(x) = 1$

Como a CDF é uma função que retorna o somatório das probabilidades até um valor de x, ela é sempre uma função crescente.

Apêndice D

Códigos

D.1 Grafo Regular e subgrafo não regular

```
#importar as bibliotecas necessarias
from igraph import *
import random
import time

#funcao para gerar um isomorfismo aleatoriamente
def gera_isomorfismo(vertices):
        #cria um vetor seguencial
        isomorfismo = range(vertices)
        #aplica trocas de posicoes nos elementos do vetor
        for i in range(vertices):
                valor1 = random.randint(0,vertices-1)
                valor2 = random.randint(0,vertices-1)
                temp = isomorfismo[valor1]
                isomorfismo[valor1] = isomorfismo[valor2]
                isomorfismo[valor2] = temp
        #retorna o vetor com valores aleatorios, de forma que a
            permutacao he repesentada pelo indice a o valor
        return isomorfismo

#quantidade de vertices
n = 28
#grau
k = 14
```

```
#quantidade de vertices para retirar
retirar = 14
#quantidade de testes
testes = 100
#dados do teste
dados = "v" + str(n) +"k" + str(k) + "r" + str(retirar) + "␣=␣"
#iniciando o vetor que vai armazenar os resultados
resultado = []
#faz para a quantidade de testes
for i in range(testes):
        #gera um isomorfismo
        Isomorfismo = gera_isomorfismo(n)
        #gera um grafo G1 K regular com n vertices
        G1 = GraphBase.K_Regular(n,k)
        #gera um grafo H permutando G1 com o isomorfismo gerado
        H = G1.permute_vertices(Isomorfismo)

        #Para gerar o subgrafo, inicialmente cria-se um vetor
           seguencial de vertices
        vertices_subgrafo = range(n)
        #r recebe a quantidade de vertices a ser retirado para
           gerar o subgrafo. Quantidade de vertices do subrafo
           he igual a quantidade de vertices de G1 menos r.
        r = retirar
        while r:
                try:
                        #Remove randomicamente valores do vetor
                            gerado. Esse valores correspondem
                           aos vertices que serao retirados do
                           grafo H para gerar G2
                        vertices_subgrafo.remove(random.randint
                           (0,n-1))
                        r-=1
                except:
                        pass
        #gera-se G2 como um subgrafo induzido de H recebendo os
            vertices que nao foram retirados
```

```
        G2 = H.induced_subgraph(vertices_subgrafo)

        #Gera-se outro isomorfismo para gerar o grafo G1
           permutado
        Isomorfismo = gera_isomorfismo(n)
        G1P = G1.permute_vertices(Isomorfismo)

        #verifica-se o isomorfismo e extrai o tempo
        try:
                ini = time.time()
                #aplica-se o algoritmo LAD para encontrar o
                   isomorfismo entre G1 permutafo e G2 e guarda
                   -se o tempo
                resp_isomorfismo = G1P.subisomorphic_lad(G2,
                   return_mapping=False)
                fim = time.time()
        except:
                resultado.append(t_max)
                print "erro no isomorfismo ou ultrapassou o
                   tempo limite"
                continue

        resultado.append(fim-ini)
        print resultado

print str(dados) + str(resultado) + ';'
```

D.2 Grafo Regular e subgrafo regular

```
# -*- coding: utf-8 -*-

#importar as bibliotecas necessarias
from igraph import *
import random
import time

#funcao para gerar um isomorfismo aleatoriamente
```

```
def gera_isomorfismo(vertices):
        #cria um vetor seguencial
    isomorfismo = range(vertices)
        #aplica trocas de posicoes nos elementos do vetor
    for i in range(vertices):
        valor1 = random.randint(0,vertices-1)
        valor2 = random.randint(0,vertices-1)
        temp = isomorfismo[valor1]
        isomorfismo[valor1] = isomorfismo[valor2]
        isomorfismo[valor2] = temp
        #retorna o vetor com valores aleatorios, de forma que a
            permutacao he repesentada pelo indice a o valor
    return isomorfismo

#quantidade de vertices
n = 20
#grau
k = 12
#quantidade de testes
testes = 100
#cria uma lista de tuplas para ligar ao grafo
m = []
for i in range(n):
    for j in range(n/2):
        m.append((i, ((j+n+i)%n+n)))
#dados do teste
dados = "v" + str(2*n) +"k" + str(2*k) + "␣=␣"
#iniciando o vetor que vai armazenar os resultados
resultado = []
#faz para a quantidade de testes
for i in range(testes):
        #gera um isomorfismo
    Isomorfismo = gera_isomorfismo(n)
    #gera um grafo G1 K regular com n vertices
    G1 = GraphBase.K_Regular(n,k)
    #gera G2 aplicando o isomorfismo em G1
    G2 = G1.permute_vertices(Isomorfismo)
```

```
#gera um grafo L K regular com n vertices
L = GraphBase.K_Regular(n,k)
#gera um grafo H K regular com n vertices
H = GraphBase.K_Regular(n,k)

#cria lista tuplas l,l1,l2 com os subgrafos H,L
l = []
for i in H.get_edgelist():
    l.append((i[0]+n,i[1]+n))
l1 = []
for i in L.get_edgelist():
    l1.append((i[0]+n,i[1]))
l2 = []
for i in L.get_edgelist():
    l2.append((i[0],i[1]+n))

    #Adiciona-se os vertices n e as arestas l,l1,l2 ao
       grafo G1
G1.add_vertices(n)
G1.add_edges(l)
G1.add_edges(l1)
G1.add_edges(l2)

    #Gera-se outro isomorfismo para gerar o grafo G1
       permutado
Isomorfismo = gera_isomorfismo(2*n)
G1P = G1.permute_vertices(Isomorfismo)

#verifica-se o isomorfismo e extrai o tempo
ini = time.time()
#aplica-se o algoritmo LAD para encontrar o isomorfismo
   entre G1 permutafo e G2 e guarda-se o tempo
resp_isomorfismo= G1P.subisomorphic_lad(G2,return_mapping=
   False)
fim = time.time()

print resultado
```

```
    resultado.append(fim-ini)

print str(dados) + str(resultado) + ';'
```

D.3 Grafo Regular e subgrafo perfeito

```
# -*- coding: utf-8 -*-

#importar as bibliotecas necessarias
from igraph import *
import random
import time

#funcao para gerar um isomorfismo aleatoriamente
def gera_isomorfismo(vertices):
    #cria um vetor seguencial
    isomorfismo = range(vertices)
    #aplica trocas de posicoes nos elementos do vetor
    for i in range(vertices):
        valor1 = random.randint(0,vertices-1)
        valor2 = random.randint(0,vertices-1)
        temp = isomorfismo[valor1]
        isomorfismo[valor1] = isomorfismo[valor2]
        isomorfismo[valor2] = temp
    #retorna o vetor com valores aleatorios, de forma que a
       permutacao he repesentada pelo indice a o valor
    return isomorfismo

#funcao para gerar grafos perfeitos. Recebe a quantidade de
   vertices e o grau desejado. Retorna um grafo perfeito
def gera_perfeito(vertices, grau):
    G = GraphBase.Full(vertices)
    R = GraphBase.K_Regular(vertices,grau)
    #conjunto de vertices do clique deslocado
    l = []
    for i in G.get_edgelist():
        l.append((i[0]+vertices,i[1]+vertices))
```

```
    #conjunto de vertices 1 do grafo regular
    l1 = []
    for i in R.get_edgelist():
        l1.append((i[0]+vertices,i[1]))
    #conjunto de vertices 2 do grafo regular
    l2 = []
    for i in R.get_edgelist():
        l2.append((i[0],i[1]+vertices))
    #Adiciona-se os vertices n e as arestas l,l1,l2 ao grafo G
    G.add_vertices(vertices)
    G.add_edges(l)
    G.add_edges(l1)
    G.add_edges(l2)
    #retorna o grafo G
    return G

#quantidade de vertices
n = 20
#grau
k = 12

testes = 100

kk = (k/2)+(k%2)

resultado = []
dados = "v" + str(2*n) +"k" + str(2*k) + "␣=␣"

for i in range(testes):
    #grafo perfeito gerado
    G1 = gera_perfeito(n/2,kk)

    P2 = GraphBase.K_Regular(n,k+kk-1)

    #desloca pro final
    l = []
    for i in P2.get_edgelist():
```

```
            l.append((i[0]+n,i[1]+n))

        #pegando o grafo pequeno
        Isomorfismo = gera_isomorfismo(n)
        G2 = G1.permute_vertices(Isomorfismo)
        #adiciona os vertices n e as arestas l ao grago G1
        G1.add_vertices(n)
        G1.add_edges(l)
        #cria um grafo r k regular com n vertices
        r = GraphBase.K_Regular(n,k)
        #adiciona em l1 a lista de arestas de r
        l1 = []
        for i in r.get_edgelist():
            l1.append((i[0]+n,i[1]))
        #adiciona em l2 a lista de arestas de r.
        #+n faz o deslocamento
        l2 = []
        for i in r.get_edgelist():
            l2.append((i[0],i[1]+n))
        #aplica a lista de arestas deslocadas em G1
        G1.add_edges(l1)
        G1.add_edges(l2)
        #gera um novo isomorfismo
        Isomorfismo = gera_isomorfismo(2*n)
        #gera o G1P com alicando o novo isomorfismo em G1
        G1P = G1.permute_vertices(Isomorfismo)

        #aplica o algoritmo e calcula o tempo
        ini = time.time()
        resp_isomorfismo= G1P.subisomorphic_lad(G2,return_mapping=
            False)
        fim = time.time()
        resultado.append(fim-ini)

print str(dados) + str(resultado) + ';'
```

D.4 Grafo perfeito e subgrafo perfeito

```
# -*- coding: utf-8 -*-

#importar as bibliotecas necessarias
from igraph import *
import random
import time

#funcao para gerar um isomorfismo aleatoriamente
def gera_isomorfismo(vertices):
    #cria um vetor seguencial
    isomorfismo = range(vertices)
    #aplica trocas de posicoes nos elementos do vetor
    for i in range(vertices):
        valor1 = random.randint(0,vertices-1)
        valor2 = random.randint(0,vertices-1)
        temp = isomorfismo[valor1]
        isomorfismo[valor1] = isomorfismo[valor2]
        isomorfismo[valor2] = temp
    #retorna o vetor com valores aleatorios, de forma que a
       permutacao he repesentada pelo indice a o valor
    return isomorfismo

#funcao para gerar grafos perfeitos. Recebe a quantidade de
   vertices e o grau desejado. Retorna um grafo perfeito
def gera_perfeito(vertices, grau):
    G = GraphBase.Full(vertices)
    R = GraphBase.K_Regular(vertices,grau)
    #conjunto de vertices do clique deslocado
    l = []
    for i in G.get_edgelist():
        l.append((i[0]+vertices,i[1]+vertices))
    #conjunto de vertices 1 do grafo regular
    l1 = []
    for i in R.get_edgelist():
        l1.append((i[0]+vertices,i[1]))
```

```
    #conjunto de vertices 2 do grafo regular
    l2 = []
    for i in R.get_edgelist():
        l2.append((i[0],i[1]+vertices))
    #Adiciona-se os vertices n e as arestas l,l1,l2 ao grafo G
    G.add_vertices(vertices)
    G.add_edges(l)
    G.add_edges(l1)
    G.add_edges(l2)
    #retorna o grafo G
    return G

#quantidade de vertices
n = 20
#grau
k = 12

testes = 100

kk = (k/2)+(k%2)

resultado = []
dados = "v" + str(2*n) +"k" + str(2*k) + "␣=␣"

for i in range(testes):
    #gera o grafo perfeito G1
    G1 = gera_perfeito(n/2,kk)
        #gera o grafo perfeito P2
    P2 = gera_perfeito(n/2,kk)

    #l recebe a lista de vertices de P2 acrescida de n
    l = []
    for i in P2.get_edgelist():
        l.append((i[0]+n,i[1]+n))

    #pegando o grafo pequeno
    Isomorfismo = gera_isomorfismo(n)
```

```
        G2 = G1.permute_vertices(Isomorfismo)
        #adiciona os vertices n e as arestas l ao grago G1
        G1.add_vertices(n)
        G1.add_edges(l)
        #cria um grafo r k regular com n vertices
        r = GraphBase.K_Regular(n,k)
        #adiciona em l1 a lista de arestas de r
        l1 = []
        for i in r.get_edgelist():
            l1.append((i[0]+n,i[1]))
        #adiciona em l2 a lista de arestas de r.
        #+n faz o deslocamento
        l2 = []
        for i in r.get_edgelist():
            l2.append((i[0],i[1]+n))
        #aplica a lista de arestas deslocadas em G1
        G1.add_edges(l1)
        G1.add_edges(l2)
        #gera um novo isomorfismo
        Isomorfismo = gera_isomorfismo(2*n)
        #gera o G1P com alicando o novo isomorfismo em G1
        G1P = G1.permute_vertices(Isomorfismo)

        #aplica o algoritmo e calcula o tempo
        ini = time.time()
        resp_isomorfismo= G1P.subisomorphic_lad(G2,return_mapping=
            False)
        fim = time.time()
        resultado.append(fim-ini)

print str(dados) + str(resultado) + ';'
```