Alessandra Silva Anyzewski

Estudo Experimental da Aplicação do Algoritmo IVL na Etapa de Detecção de Isomorfismos do GROOVE

Vitória, Espírito Santo

2016

Alessandra Silva Anyzewski

Estudo Experimental da Aplicação do Algoritmo IVL na Etapa de Detecção de Isomorfismos do GROOVE

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do título de Mestre em Informática.

Universidade Federal do Espírito Santo

Programa de Pós-Graduação em Informática

Mestrado em Informática

Orientador: Prof. Dr. Eduardo Zambon

Co-orientadora: Prof[a]. Dr[a]. Maria Claudia da Silva Boeres

Vitória, Espírito Santo

2016

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial Tecnológica,
Universidade Federal do Espírito Santo, ES, Brasil)

A637e

Anyzewski, Alessandra Silva, 1987-

Estudo experimental da aplicação do algoritmo IVL na etapa de detecção de isomorfismos do GROOVE / Alessandra Silva Anyzewski. – 2016.

69 f. : il.

Orientador: Eduardo Zambon.

Dissertação (Mestrado em Informática) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Isomorfismos (Matemática). 2. Teoria dos grafos. 3. Verificação de modelos. I. Zambon, Eduardo. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título

CDU: 004

Agradecimentos

Agradeço primeiramente à minha família: papai, mamãe, Dedê, tia Laurença e minha linda Clarice, vocês são tudo pra mim.

Aos meus queridos professores, orientador Dr. Eduardo Zambon e co-orientadora Drª. Maria Claudia da Silva Boeres. Muito obrigada pela dedicação e pela acolhida ao aceitarem me orientar a distância com todo empenho.

Ao Rodrigo por ter entrado no segundo tempo como um grande apoiador, e à Lulu pelo companheirismo nos estudos e na vida.

Ao colega Marcos Baroni pela a ajuda.

À Glaydis Martins, da secretaria do PPGI, uma importantíssima incentivadora. Espero um dia poder retribuir.

Ao meu amigo e "sub-orientador" André Machado.

Ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo pela oportunidade.

Resumo

Um dos problemas clássicos da Teoria de Grafos é o problema de isomorfismo de grafos. Esse problema trata de determinar se, dado dois grafos, é possível definir um mapeamento entre seus vértices de forma que sejam respeitadas as conexões definidas por suas arestas. Um algoritmo proposto recentemente para resolver esse problema é o IVL (*Iterated Vertex Labelling*) [Baroni (2012)].

O GROOVE (GRaph-based Object-Oriented VErification) é uma ferramenta de verificação de modelos baseados em grafos que faz uso de algoritmos de isomorfismo. No contexto do GROOVE, o problema de isomorfismo de grafos se apresenta de uma maneira diferente do problema clássico: não se deseja determinar se dois grafos são isomorfos, e sim se, dado um grafo, ele é isomorfo a algum dos elementos de um conjunto de grafos.

Neste trabalho, propõe-se a adaptação do IVL para o GROOVE e a realização de experimentos computacionais com o objetivo de determinar se essa adaptação traz ganhos de performance para a ferramenta. Os resultados levam à conclusão de que o IVL tem desempenho análogo ao algoritmo de isomorfismos que já está implementado no GROOVE.

Além desses resultados, foi investigado em um cenário similar o uso de filtros de não-isomorfismo, com a intenção de determinar o não-isomorfismo entre dois grafos a um custo computacional baixo. Os resultados dos testes indicam que essa abordagem é bastante promissora, sendo capaz de detectar não-isomorfismos com eficiência de quase 100% , com tempos de execução bem mais baixos que os performados pelo algoritmo atual do GROOVE quando executado nesse cenário adaptado.

Palavras-chaves: Isomorfismo de Grafos, Verificação de Modelos, GROOVE, IVL, Sistema de Transição de Grafos.

Abstract

The graph isomorphism is a classical problem in Graph Theory, which consists of determining if, given two graphs, it is possible to define a mapping between their vertexes in a way so that the connection defined by their edges are respected. An algorithm proposed recently to solve this problem is the IVL (Iterated Vertex Labelling) [Baroni (2012)].

GROOVE (GRaph-based Object-Oriented VErification) is a graph-based model checking tool which makes use of isomorphism algorithms. In GROOVE's context, the graph isomorphism problem is set differently from the classical problem: they are not interested on determining if two graphs are isomorphic, instead, they want to determine if, given a graph, it is isomorphic to one of the elements of a graph set.

In this work, it's proposed the IVL adaptation to GROOVE and computational experiments in order to test if this new adapted algorithm brings performance gains to the tool. It can be concluded from the results that IVL has a similar performance compared to the current implementation in GROOVE.

Beyond those results, it was investigated in a similar framework the use of non-isomorphism filters, intending to determine the non-isomorphism between two graphs in a low computational cost. The test results point out that this is a promising approach, being able to detect non-isomorphisms with almost 100% efficiency, with a much lower running time when compared to current GROOVE algorithm when executed in this framework.

Keywords: Graph Isomorphism, Model Checking, GROOVE, IVL, Graph Transition System.

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Sumário

1 Introdução

Com o advento da tecnologia, a sociedade vem fazendo cada vez mais uso de sistemas automatizados, nas mais diversas tarefas do cotidiano. Dependendo do sistema, um comportamento inesperado pode trazer graves consequências, como, por exemplo, acidentes envolvendo sistemas que atuam em equipamentos pesados, de alta pressão ou de alta temperatura. Analisar os possíveis comportamentos de um sistema para garantir que ele se mantenha em suas especificações independente do contexto é o objetivo da *verificação de sistemas*. Realizar essa tarefa de forma sistemática, com rigor matemático, caracteriza a *verificação formal* de um sistema.

Uma maneira de se fazer a verificação formal de um sistema é representá-lo por um modelo, e a partir dele, inferir seus possíveis comportamentos. A isso chamamos *verificação de modelos* (*model checking*)[Baier, Katoen et al. (2008)]. Uma das formas mais flexíveis de representar um sistema e as transições entre seus comportamentos é a modelagem por grafos.

O GROOVE (GRaph-based Object-Oriented VErification) [Rensink (2004)] é destaque no meio científico como uma das principais ferramentas disponíveis para verificação de modelos baseados em grafos, e é objeto de pesquisa corrente do grupo que o desenvolve, o *Formal Methods and Tools* (FMT) da Universidade de Twente. O GROOVE tem sido capaz de modelar e verificar modelos de forma automatizada, com resultados cada vez melhores ao longo dos anos.

O isomorfismo de grafos [Fortin (1996)] é um problema clássico da Teoria de Grafos, que consiste em determinar se é possível definir um mapeamento entre seus vértices de forma que sejam respeitadas as conexões definidas por suas arestas. Esse problema dispõe de diversos algoritmos na literatura que o solucionam com bons desempenhos, como o *Nauty* [McKay et al. (1981)], o *Traces* [McKay e Piperno (2014)] e o *Saucy* [Darga, Sakallah e Markov (2008)] por exemplo. Mais recentemente, foi proposto o IVL [Baroni (2012)] que mostrou bons resultados frente aos principais algoritmos disponíveis na literatura.

O GROOVE contém uma etapa de verificação de isomorfismos, composta por um dos algoritmos mais importantes entre os processos presentes no GROOVE, de forma que, ganhos de performance nessa etapa impactam o desempenho da ferramenta como um todo. O objetivo deste trabalho é investigar experimentalmente o comportamento de novas implementações nessa etapa.

Introduziremos nas próximas seções os conceitos de verificação de modelos, como são operados esses modelos, o que é o GROOVE, o que é o problema de isomorfismo e qual o algoritmo de detecção de isomorfismos que está sendo implementado neste trabalho.

1.1 Verificação de Modelos

O modelo de um sistema é uma representação desse sistema como uma linguagem formal, matematicamente consistente, o que permite descrever os elementos dessa linguagem de uma maneira precisa. A verificação de um modelo consiste em enumerar os elementos de sua linguagem (ou estados do sistema), a fim de checar à exaustão se todos os estados vão ao encontro das especificações definidas. Essa forma de verificação formal é conhecida como abordagem enumerativa.

Nesta dissertação, a ferramenta de verificação de modelos em estudo (GROOVE) utiliza a abordagem enumerativa sobre uma linguagem de grafos, e tem como base do seu funcionamento o conceito de transformação de grafos.

1.2 Transformação de Grafos

A transformação de grafos [Rozenberg (1997)] é uma forma de modelar sistemas que tem se mostrado bastante apropriada para as mais variadas aplicações, graças ao seu formalismo, flexibilidade e à capacidade do grafo de representar variados sistemas. Em um grafo nós representam entidades do sistema e as arestas, suas relações. A mudança dessas relações e a criação/exclusão de entidades são os preceitos da transformação de grafos.

A transformação de grafos é baseada em um conjunto de regras. Cada regra é composta basicamente por dois grafos (lado esquerdo e lado direito). Ao identificar o lado esquerdo como um subgrafo do grafo hospedeiro (isto é, o grafo que sofrerá a transformação) sua ocorrência é substituída pelo lado direito, gerando um novo grafo.

Neste trabalho a transformação de grafos é usada para modelar a dinâmica do sistema a ser verificado. O estado inicial do sistema, representado por um *grafo inicial*, associado ao conjunto de regras de transformação compõe uma *gramática de grafos*, ou simplesmente gramática. As aplicações sucessivas das regras de transformação sobre os estados gerando todos os grafos contidos na linguagem associada a uma gramática caracteriza o processo de *caminhamento sobre uma gramática de grafos* ou *exploração de um espaço de estados do sistema*. Há diversas formas de percorrer uma gramática, a depender da estratégia de exploração (por exemplo, busca em largura ou busca em profundidade). Quando todas as possibilidades de aplicação de regras são aplicadas no caminhamento dizemos que foi feita uma *exploração exaustiva*. Mais detalhes sobre transformação de grafos serão apresentados na seção 2.1.

1.3 O GROOVE

O GROOVE é uma ferramenta para verificação de sistemas baseada em transformação de grafos, que faz a exploração do espaço de estados de uma gramática de grafos. Estados representados por grafos isomorfos são considerados equivalentes no contexto de verificação de modelos baseados em grafos, e são reduzidos a uma única representação, o que reduz significativamente o tamanho do espaço de estados, conforme experimentado em Ghamarian e Zambon (2009).

O desafio do GROOVE é ser capaz de representar e verificar a robustez de sistemas cada vez maiores e mais complexos. Analisar todas as possibilidades de estados de um sistema exige uma grande capacidade de memória e de processamento. Para não armazenar estados redundantes, ele faz uso de uma etapa de isomorfismo de grafos.

1.4 O Problema de Isomorfismo de Grafos

Classicamente, o problema do isomorfismo de grafos consiste em, dado dois grafos, definir se existe uma função bijetora que relacione cada um dos nós do primeiro grafo com os nós do segundo grafo, respeitando as conexões definidas pelas arestas. Esse problema tem várias aplicações reais, como por exemplo, o reconhecimento de imagens [Conte et al. (2004)] e segurança de redes de computadores [Pedarsani e Grossglauser (2011)]. Recentemente foi demonstrado que ele é um problema de dificuldade quasi-polinomial [Babai (2015)]. Uma das soluções para o problema de isomorfismo de grafos mais disseminadas dentre as disponíveis na literatura é a que faz uso do *Nauty*, proposto por McKay et al. (1981) e usado como inspiração para a solução implementada atualmente no GROOVE.

No GROOVE, o problema de isomorfismo se estabelece de uma maneira um pouco diferente: ao invés de comparar se dois grafos são isomorfos entre si, a ferramenta busca descobrir se um novo grafo (novo estado), ele é isomorfo a algum dos estados explorados previamente (sabidamente não-isomorfos entre si). Parte dos conceitos do *Nauty* foram adaptado para essa abordagem, mas certamente outros algoritmos propostos na literatura poderiam servir de inspiração para a solução do problema de isomorfismo no GROOVE. Neste trabalho propomos adaptar para o GROOVE um algoritmo desenvolvido recentemente, o chamado IVL.

1.5 O Novo Algoritmo de Isomorfismo: Iterative Vertex Labeling - IVL

O algoritmo para detecção de isomorfismos chamado IVL foi proposto por Baroni (2012). O IVL soluciona o problema clássico de isomorfismo, comparando diretamente dois grafos e respondendo se são isomorfos ou não. Para isso, o IVL se inspira no método das potências proposto por Santos, Rangel e Boeres (2010) para construir um algoritmo simples de implementar e eficiente para diferentes classes de grafos.

Os resultados apresentados por testes com o IVL [Baroni (2012)] mostram que esse algoritmo é competitivo frente a alguns algoritmos consagrados na literatura (serão detalhados no Capítulo 2). Para usar o IVL no GROOVE é necessário adaptá-lo, pois o GROOVE trabalha com grafos direcionados e rotulados, enquanto o IVL trabalha com grafos não-direcionados e não-rotulados (conceitos a serem detalhados no Capítulo 2).

1.6 Questões-chave da Pesquisa

O objetivo principal deste trabalho é adaptar o IVL para o GROOVE e verificar como essa nova implementação se compara à implementação atual. Adicionalmente, deseja-se determinar qual a melhor forma de discriminar quanto a isomorfismo os grafos das gramáticas de teste (descritas na seção 2.6) usando-se critérios menos custosos computacionalmente. Por último, testamos os comportamentos do IVL adaptado e do algoritmo do GROOVE quando usados na verificação direta de isomorfismo entre dois grafos (fora do contexto de comparar um grafo com um conjunto de grafos). Para orientar o desenvolvimento da pesquisa, foram propostas as seguintes questões, que serão respondidas ao longo da dissertação:

1. É possível adaptar o IVL para o GROOVE?
2. De que formas isso pode ser feito?
3. Qual a performance das adaptações do IVL em relação ao algoritmo original do GROOVE?
4. Considerando os grafos gerados pelas gramáticas de teste, quais atributos de grafos ajudam a diferenciá-los quanto a isomorfismo de maneira eficiente?
5. Como os IVLs adaptados se comparam ao GROOVE na detecção de isomorfismos entre dois grafos (fora do contexto do GROOVE)?

1.7 Organização da Dissertação

Capítulo 2 apresenta mais detalhadamente os conceitos, as metodologias e as ferramentas utilizadas no trabalho juntamente com a revisão bibliográfica. Define melhor o problema tratado, cujos resultados serão apresentados nos capítulos seguintes.

Capítulo 3 mostra as possíveis adaptações do algoritmo IVL para grafos orientados e rotulados, ressaltando as particularidades de se adaptar para o GROOVE. Nesse capítulo são apresentados e discutidos os resultados computacionais dessas adaptações ao executarmos elas para algumas gramáticas de teste.

Capítulo 4 contém algumas estratégias de *filtros* para os algoritmos de isomorfismo, ou seja, critérios capazes de detectar não-isomorfismos a um custo computacional mais baixo do que a execução completa do algoritmo de isomorfismo. Aqui é apresentado o motivo de usar essa abordagem, as possibilidades de filtros e os resultados computacionais quando executados sobre algumas gramáticas de teste. Além disso, o capítulo mostra o desempenho do algoritmo original do GROOVE e dos IVL adaptados quando executados fora do ambiente GROOVE para as mesmas gramáticas de teste, comparando os grafos dois a dois.

Capítulo 6 conclui a dissertação e sugere trabalhos futuros.

2 Definição do Problema e Revisão Bibliográfica

Conforme a seção 1.6, podemos estabelecer os problemas principais a serem abordados nesta dissertação:

- Verificar se é possível aplicar, e de quais maneiras, um novo método de detecção de isomorfismo de grafos (IVL) no GROOVE (seções 3.1 a 3.5).
- Checar os resultados das adaptações do IVL frente à implementação atual do GROOVE (seção 3.6).
- Propor estratégias de detecção de não-isomorfismos a baixo custo computacional, que podem ser executadas previamente aos algoritmos de detecção de isomorfismo, e checar suas eficiências em relação às gramáticas de teste (seções 4.1 a 4.5).
- Verificar ainda, como se comparam os algoritmos de isomorfismo fora do GROOVE (seção 4.6).

Neste capítulo, introduzimos com mais detalhes cada um dos conceitos que são utilizados ao longo da dissertação para a apresentação dos resultados.

2.1 Conceitos de Transformação de Grafos

Nesta seção são apresentados os principais conceitos e definições de Transformação de Grafos.

2.1.1 Grafos e suas Relações

O GROOVE utiliza grafos do tipo direcionados e rotulados.

Definição 2.1.1 (Grafo Direcionado e Rotulado)**.** *Seja Rot um universo finito de rótulos. Um* grafo *direcionado e rotulado é uma tupla* $G = \langle V_G, A_G \rangle$ *em que*

- V_G *é um conjunto finito de vértices (ou nós);*
- $A_G \subseteq V_G \times Rot \times V_G$ *é um conjunto finito de arestas em que, para cada aresta* $a \in A_G = \langle n_0, rot, n_1 \rangle$*, está associado um nó-fonte* $n_0 \in V_G$*, um rótulo* $rot \in Rot$ *e um nó-sumidouro* $n_1 \in V_G$*.*

A diferenciação dos dois nós associados a uma aresta em fonte/sumidouro (definindo uma direção para a aresta) caracteriza o grafo como sendo *direcionado*. A associação de um rótulo à aresta caracteriza o grafo como sendo *rotulado*. As funções $fonte : A_G \rightarrow V_G$, $sumid : A_G \rightarrow V_G$ e $rot : A_G \rightarrow Rot$ serão usadas para recuperar os elementos da tupla que forma uma aresta.

Nesta seção e na seção seguinte estaremos nos referenciando a grafos direcionados e rotulados ao usarmos o termo *grafo*. Grafos são relacionados por morfismos.

Definição 2.1.2 (Morfismo de Grafos)**.** *Um morfismo entre dois grafos G, H é uma função $m : (V_G \cup A_G) \rightarrow (V_H \cup A_H)$, tal que*

- $m(V_G) \subseteq V_H$;
- $m(A_G) \subseteq A_H$;
- $rot_H \circ m = rot_G$

Ou seja, m mantém coerentes as associações entre nós-fonte, nós-sumidouro e rotulações. Um isomorfismo é um tipo especial de morfismo de grafos.

Definição 2.1.3 (Isomorfismo de Grafos)**.** *Dado dois grafos G e H, um isomorfismo entre G e H, denotado $G \simeq H$, é um morfismo $m : G \rightarrow H$ tal que m é uma função bijetora.*

As formas de encontrar a relação de isomorfismo entre dois grafos é o foco deste trabalho e será melhor descrito na seção 2.3.

2.1.2 Transformação de Grafos

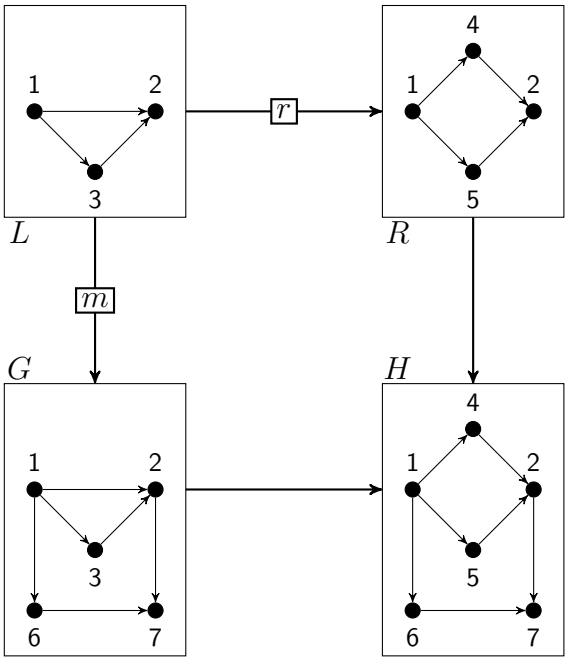
Alterações nas estruturas de um grafo levam à sua transformação.

Definição 2.1.4 (Transformação de Grafos)**.** *A transformação de grafos é uma técnica baseada em regras do tipo $r : L \rightharpoonup R$, onde L e R são grafos diretamente relacionados por um morfismo de grafos* parcial *(isto é, um morfismo entre subconjuntos dos elementos de L e R). Transformar um grafo consiste em aplicar um regra r sobre um* grafo hospedeiro *G (grafo que sofrerá a transformação). Ao encontrar um subgrafo isomorfo a L em G, de acordo com a regra r, ele será substituído por R, formando um novo grafo H.*

Um exemplo de transformação de grafo pode ser visto na figura 1 adaptada de Zambon (2013). Na figura, o grafo L é isomorfo aos vértices $1, 2$ e 3 de G (*matchm*). Esse *match* é substituído então pelo grafo R conforme a regra r, formando o grafo H.

Se G pode ser transformado em H através da aplicação da regra r, escrevemos $G \xrightarrow{r} H$.

Figura 1: Exemplo de uma transformação de grafos.



Fonte: adaptado de Zambon (2013)

Podemos resumir então os passos seguidos pelo GROOVE para transformar um grafo. Dada a regra $r\colon L \rightharpoonup R$ e o grafo hospedeiro G:

1. Achar os subgrafos isomorfos a L em G (*matches* de L em G), operação realizada pela função $match : r, G \rightarrow M$ que retorna um conjunto de *matches* M.

2. Para cada *match* $m \in M$, gerar novo grafo H conforme descrito pela regra r, ou seja, H é produto da aplicação da regra r sobre o *match* m presente em G. Essa operação é realizada pela função $transformar(r, m, G)$.

O foco deste trabalho não é a transformação de grafos em si, e sim, o isomorfismo entre os grafos que são produtos das transformações. No caso do GROOVE, o método de transformação de grafos utilizado é a dita *Transformação Algébrica*, cujos detalhes podem ser vistos em Rozenberg (1997). Aqui, basta saber que existe um processo de transformação de grafos que gerará os grafos que serão analisados entre si quanto a isomorfismo, mas a maneira em que isso é feito não é objeto deste trabalho. Portanto, não serão apresentadas as implementaões das funções *match* e *transformar*.

O conjunto de todos os estados gerados pelas operações de transformação de grafos e o conjunto de todas as transições aplicadas sobre os estados formam juntos o que chamamos de *Sistema de Transição de Grafos* (STG).

2.2 Sistema de Transição de Grafos - *STG*

Nesta seção apresentamos em detalhes os processos que dão origem à exploração de um espaço de estados, representada no GROOVE por um STG. O ponto de partida para a exploração é a gramática de grafos a ser explorada, conceito a ser definido a seguir.

Definição 2.2.1 (Gramática de Grafos)**.** *Uma* gramática de grafos $\mathcal{G} = \langle \mathcal{R}, G_0 \rangle$ *é uma tupla composta por um conjunto de regras de transformação de grafos* $\mathcal{R}$ *e um grafo inicial* G_0*.*

A exploração de uma gramática de grafos gera um sistema de transição de grafos.

Definição 2.2.2 (Sistema de Transição de Grafos)**.** *Um sistema de transição de grafos* $STG = \langle \mathcal{S}, \rightarrow, S_0 \rangle$ *é uma tupla composta por:*

- *um conjunto* $\mathcal{S}$ *de estados;*
- *um conjunto* $\rightarrow$ *de transições de grafos a partir do conjunto de regras; e*
- *um estado inicial* S_0*.*

Uma gramática $\mathcal{G}$ *gera um STG onde:*

- $S_0 = G_0$*; e*
- *se* $G \xrightarrow{r} H$ *para algum* $G \in \mathcal{S}$ *e* $r \in \mathcal{R}$*, então existe* $H' \in \mathcal{S}$ *tal que* $H \simeq H'$*, e* G *pode ser transformado para* H' *pela mesma regra* r *(*$G \xrightarrow{r} H'$*).*

De acordo com o último item da definição acima, uma gramática de grafos gera um STG que representa todas as possibilidades de estados e transições do sistema a partir de um estado inicial, desde que esteja condicionada a ter em seu conjunto $\mathcal{S}$ apenas um representante de cada conjunto de grafos isomorfos. Como o STG representa todas as possibilidades de aplicações das regras na exploração de sua respectiva gramática, dizemos que a gramática foi explorada de forma *exaustiva*. Todos os STG's gerados por uma gramática são equivalentes. Eles variam entre si apenas na ordem em que os estados são explorados, o que depende da estratégia de exploração adotada.

O princípio de colapsar todos os estados que sejam suficientemente similares na exploração do espaço de estados é chamado de *redução por simetria*. Aqui, o critério de similaridade adotado é o isomorfismo, ou seja, dois estados isomorfos são considerados equivalentes. Portanto, na exploração de um espaço de estados, uma etapa importante é checar, para cada novo estado gerado, se ele é ou não isomorfo a cada um dos estados gerados previamente. Comparar o novo grafo com cada um dos anteriores geraria um

custo computacional da ordem de $O(|\mathcal{S}|^2)$. Uma vez que $\mathcal{S}$ é, em geral, um conjunto muito grande, esse custo pode ser muito alto e qualquer melhora na performance tem um grande impacto no desempenho do GROOVE. Daí a importância de escolher bem a estratégia de detecção de isomorfismos durante essa exploração.

A exploração do espaço de estados no STG a partir de uma gramática de grafos pode ser traduzida no algoritmo 1, onde $\mathcal{N}$ representa o conjunto de estados *novos*, ainda a serem explorados.

Algoritmo 1: Algoritmo para exploração de espaço de estados.

Entrada: gramática $\mathcal{G} = \langle G_0, \mathcal{R} \rangle$
Saída: $STG = \langle \mathcal{S}, \rightarrow, S_0 \rangle$

```
1  S := ∅, → := ∅, N := {G0};
2  enquanto N ≠ ∅ faça
3      para cada G ∈ N faça
           // a ordem de escolha de G depende da estratégia de
              exploração
4          N := N \ {G};
5          para r ∈ R, m ∈ match(r, G) faça
6              H := transformar(r, m, G);
7              se novo(H, S) então
8                  S := S ∪ {H}, N := N ∪ {H};
9              fim
10             → := → ∪ {G -r-> H};
11         fim
12     fim
13 fim
```

A linha 1 inicializa o sistema de transformação com conjuntos vazios, e inicializa o conjunto de estados *novos* $\mathcal{N}$ com o grafo inicial. Enquanto houver estados a serem explorados, o algoritmo escolhe um deles na linha 3 e o remove do conjunto $\mathcal{N}$ na linha 4, de acordo com a estratégia de exploração. A estratégia adotada (por exemplo, busca em largura ou busca em profundidade) não tem qualquer impacto sobre quais estados serão encontrados, já que a busca é exaustiva.

A linha 5 trata do *matching* das regras de transição com o estado que está sendo explorado (G). Ao aplicar a regra r em G é gerado o estado H (linha 6), o qual é checado quanto a isomorfismo em relação ao conjunto $\mathcal{S}$ na linha 7, pelo método $novo(H, \mathcal{S})$. Na linha 10, a transição executada na linha 6 é armazenada no conjunto de transições $\rightarrow$.

O algoritmo 2 descreve o método $novo(H, \mathcal{S})$.

Algoritmo 2: ALGORITMO PARA O MÉTODO $novo(H, \mathcal{S})$.

Entrada: grafo H, conjunto de estados explorados $\mathcal{S}$

Saída: H é um grafo novo?

1 $C := certificado(H)$, $\bar{\mathcal{S}} := \{H' \in \mathcal{S} \mid certificado(H') = C\}$
2 **para** *cada* $H' \in \bar{\mathcal{S}}$ **faça**
3 **se** $H \simeq H'$ **então**
4 **retorna** falso
5 **fim**
6 **fim**
 // todos os candidatos foram checados, mas nenhum era isomorfo a H, então H é novo
7 **retorna** verdadeiro

O algoritmo 2 faz uso de *certificados de grafos*, que é um *hash* definido para os grafos (a definição formal de certificado de grafo e suas propriedades serão apresentadas na seção 3.1.1). O cálculo dos certificados é definido de tal forma que dois grafos isomorfos tenham o mesmo certificado. É essencial notar que o contrário não é verdade, dois grafos com mesmo certificado não necessariamente são isomorfos. Portanto, é necessário fazer o teste de isomorfismo entre H e todos os grafos contidos no conjunto $\bar{\mathcal{S}} \subseteq \mathcal{S}$, onde $\bar{\mathcal{S}}$ é o conjunto que contém todos os grafos de $\mathcal{S}$ que possuem certificados iguais aos do grafo H. O algoritmo 2 engloba o clássico problema de isomorfismo de grafos (teste da linha 3), a ser apresentado na próxima seção.

2.3 O Problema de Isomorfismo de Grafos

Desde o estado da arte levantado por Fortin (1996), o problema de isomorfismo de grafos, conforme a definição 2.1.2, era classificado como NP-difícil, mas não se sabia precisar se era P ou NP-completo. Recentemente, Babai (2015) demonstrou um grande resultado para a computação teórica: o problema de isomorfismo de grafos é quasi-polinomial. Esse resultado teórico, segundo o próprio autor, não interfere nas soluções práticas providas até então na literatura, soluções essas nas quais embasamos os algoritmos adotados neste trabalho.

Nos algoritmos de isomorfismo de grafos utiliza-se alguns conceitos que serão introduzidos a seguir. As definições apresentadas aqui são adaptadas de McKay e Piperno (2014).

Definição 2.3.1 (*Coloração*)**.** *Uma* coloração[1] π *de um grafo* G *é uma função sobrejetiva tal que, para cada nó é atribuida uma cor. Um par* (G, π) *é chamado um* grafo colorido*. Um conjunto de nós mapeados para a mesma cor constituem uma* partição*. Se cada nó possuir uma cor diferente, o grafo é dito ter* partições discretas*. Para duas colorações distintas referentes a um mesmo grafo, aquela que define um maior número de partições é dita* mais refinada*. O processo de tornar a coloração mais refinada é chamado de* refinamento*.*

Definição 2.3.2 (Automorfismo)**.** *É denominado automorfismo a relação de isomorfismo estabelecida entre um grafo e ele mesmo.*

Definição 2.3.3 (Isomorfismo de Grafos Coloridos)**.** *Dois grafos coloridos* (G, π) *e* (G', π') *são ditos isomorfos se for possível mapear os vértices de algum dos automorfos de* G *para os vértices de* G*' e mapear suas colorações* $\pi \to \pi'$ *ambas por uma mesma função* g*, ou seja,* $(G', \pi') = (G, \pi)^g$ *(o mapeamento* g *é indicado sobrescrito).*

Definição 2.3.4 (Representação Canônica de um Grafo)**.** *A representação (ou rotulação/coloração) canônica de um grafo é uma função* C *tal que:*

- $C(G, \pi) \simeq (G, \pi)$
- $C(G^g, \pi^g) = C(G, \pi)$

Essas condições definem que, para cada grafo colorido, teremos um grafo colorido isomorfo representando toda a sua classe de isomorfos. Portanto, $(G, \pi) \simeq (G', \pi') \Leftrightarrow C(G, \pi) = C(G', \pi')$*.*

McKay justifica o uso da rotulação canônica dizendo que [2] "Testar o isomorfismo entre dois grafos diretamente pode ter a vantagem de que um isomorfismo pode ser encontrado muito antes de completar a busca exaustiva. Por outro lado, isso não se adequa bem ao problema comum de rejeitar isomorfos ante uma coleção de grafos ou identificar um grafo em um banco de dados de grafos. Por essa razão, a abordagem mais comum é a 'rotulação canônica', um processo em que um grafo é rotulado de forma que grafos isomorfos são idênticos após a rotulação."[McKay e Piperno (2014), tradução nossa].

O problema dessa abordagem é que encontrar a rotulação canônica é um processo bastante custoso. Daí o uso de *certificados de grafos* na busca de isomorfismo.

[1] Na Teoria dos Grafos, essa definição é mais conhecida como *rotulação de grafos*. Baroni (2012) usa essa terminologia, porém, neste trabalho usaremos mais o termo coloração, já que usamos o termo *grafo rotulado* para designar grafos cujas arestas são rotuladas.

[2] "Testing two graphs for isomorphism directly can have the advantage that an isomorphism might be found long before an exhaustive search is complete. On the other hand, it is poorly suited for the common problems of rejecting isomorphs from a collection of graphs or identifying a graph in a database of graphs. For this reason, the most common practical approach is "canonical labelling", a process in which a graph is relabelled in such a way that isomorphic graphs are identical after relabelling."

O algoritmo 2 foi introduzido o conceito de certificado de grafos, sendo que, naquele caso, *não* se tratava de uma representação canônica. Apesar de dois grafos isomorfos terem o mesmo certificado, o contrário nem sempre é verdade, violando a segunda condição da definição 2.3.4. Esse ponto será mais detalhado na seção 2.4. Um certificado bem escolhido funciona quase com a mesma eficiência de uma rotulação canônica [Rensink (2010)], exigindo que a busca direta por isomorfismo (linha 3 do algoritmo 2) seja executada poucas vezes.

Os principais algoritmos de verificação de isomorfismos disponíveis na literatura são:

Nauty foi introduzido por McKay et al. (1981). Utiliza uma árvore de busca para encontrar a representação canônica de um grafo. É a ferramenta mais difundida até hoje para o problema de isomorfismos de grafos. Inspirou o algoritmo de detecção de isomorfismos utilizado no GROOVE.

Traces foi introduzido por McKay e Piperno (2014). Evolução do *nauty* no que diz respeito principalmente ao processo de busca em árvore. Se mostrou superior aos algoritmos concorrentes especialmente para grafos muito grandes.

Além desses, o *saucy* [Darga, Sakallah e Markov (2008)], o *bliss* [Junttila e Kaski (2011)], e o *conauto* [López-Presa, Anta e Chiroque (2011)] estão entre os mais utilizados. Os dois primeiros utilizam rotulações canônicas (ambos são inspirados no *nauty*), sendo o *saucy* especializado em grafos grandes e esparsos. O *bliss* tem uma performance diferenciada para grafos difíceis. O *conauto* compara diretamente dois grafos, sem utilizar rotulação canônica. Definir qual deles é o melhor depende da classe de grafos que se está tratando. Mas, considerando os resultados gerais, o *Traces* se mostra o melhor algoritmo disponível até então, enquanto o *nauty* continua sendo o mais difundido, com resultados satisfatórios especialmente para grafos pequenos.

Vale ressaltar a dificuldade de definir bons casos de teste para comparar o desempenho desses algoritmos. Em geral, segundo Babai (2015), é difícil se aproximar do pior caso, ou seja, do caso quasi-polinomial. Nas aplicações desta dissertação, em geral, os grafos são pequenos, no máximo atingindo a ordem de algumas dezenas de nós. Para inserir uma maior dificuldade, foram propostos testes em algumas gramáticas que contêm muitos automorfismos. As gramáticas de teste serão especificadas na seção 2.6.

2.4 O Isomorfismo no GROOVE

O GROOVE implementa os algoritmos 1 e 2 descritos anteriormente, sendo que o segundo é o núcleo que contém o algoritmo de detecção de isomorfismos. O código da

ferramenta é modularizado, de forma a permitir o acoplamento de um novo método de detecção de isomorfismos, principal objetivo deste trabalho. Ele é escrito em Java. Mais detalhes sobre a ferramenta, funcionalidades e implementação podem ser consultados no trabalho de Rensink (2004). No trabalho de Ghamarian et al. (2012) é possível encontrar vários exemplos de explorações feitas no GROOVE, mostrando com mais detalhes a forma de se modelar os sistemas nessa ferramenta.

Um exemplo de exploração de um espaço de estados com detecção de isomorfismos é representado na figura 2. Nela, as caixinhas representam estados, enquanto as setas, as transições. Cada estado é um grafo e as transições estão rotuladas com o nome da regra de transformação que foi aplicada.

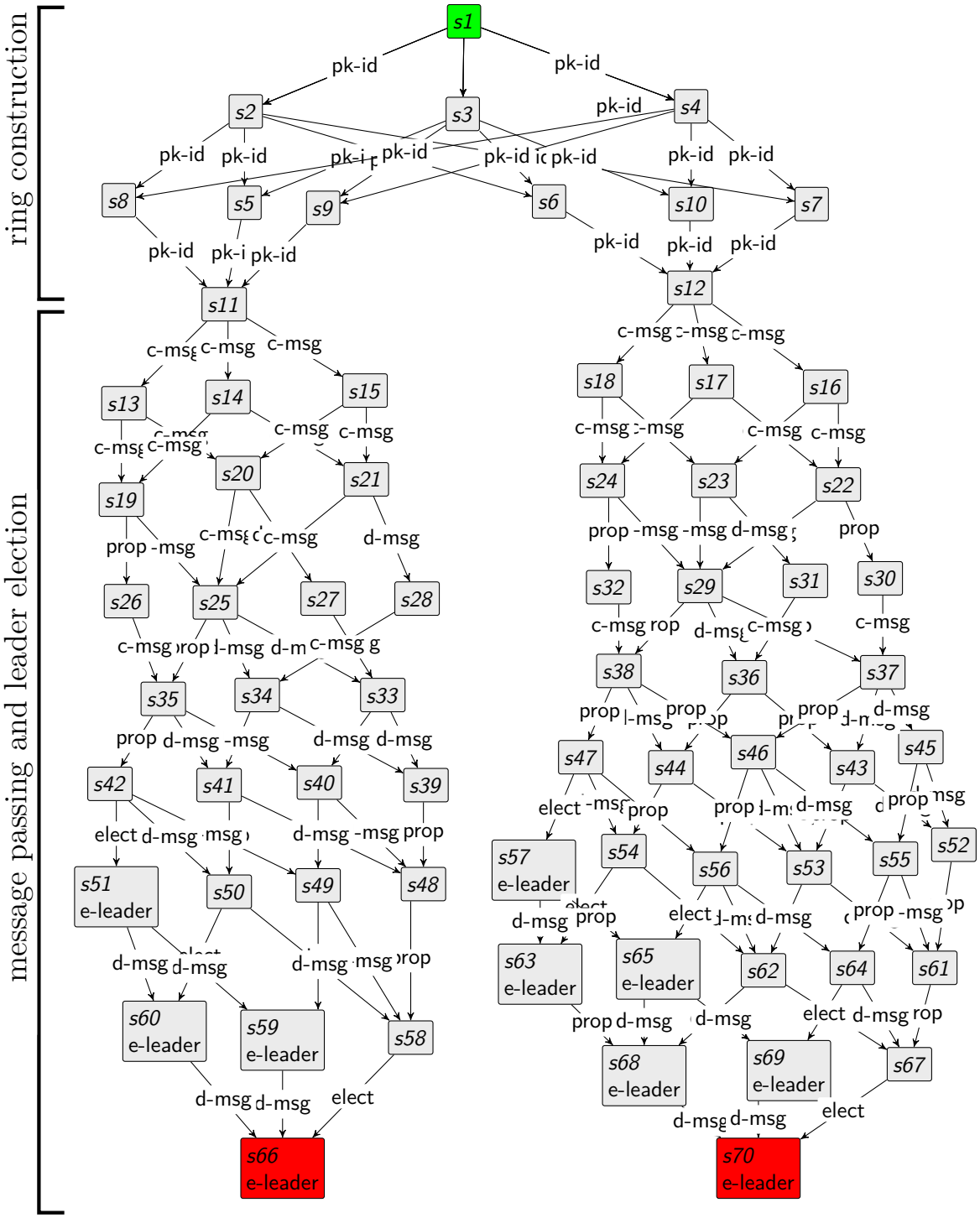
Podemos ver que o sistema converge para dois possíveis estados finais, rotulados como $s66$ e $s70$. Sem a detecção de isomorfismos, o caminhamento sobre a gramática levaria a uma ramificação cada vez maior do STG conforme as regras fossem aplicadas. Assim, seriam gerados vários estados finais isomorfos entre si, quando na verdade eles podem ser reduzidos aos estados $s66$ ou pelo $s70$.

Conforme definido na seção 2.2, um ponto crítico de um STG é justamente a detecção de isomorfismo entre o novo estado gerado e os já computados previamente. Se isomorfo, ele não é armazenado, enquanto se não-isomorfo, ele é incorporado ao conjunto $\mathcal{S}$ de estados explorados.

Para implementar a checagem de isomorfismo no GROOVE, os autores se inspiraram em McKay et al. (1981) para calcular os certificados de grafos, adaptando dois pontos principais: (i) suportar grafos do tipo orientado e rotulado; (ii) utilizar em um contexto em que a pergunta não é se dois grafos são isomorfos, e sim, se dado um grafo, ele é isomorfo a algum dos elementos de um conjunto de grafos.

Rensink (2007) apresenta uma primeira solução para essa implementação, baseada no cálculo de certificados, conforme definido na seção 2.2. Apesar de ser uma boa proposta, ele constata que o algoritmo de isomorfismo, para algumas gramáticas, representa boa parte do custo computacional da exploração de estados, chegando a 75% do tempo total. A maior parte desse tempo era associada ao cálculo dos certificados. Já Rensink (2010) traz outra solução, que se inspira novamente no *nauty*, e no trabalho de Paige e Tarjan (1987). Ela melhora o primeiro algoritmo, de forma a reduzir significantemente o custo associado à etapa de isomorfimo. Esse segundo trabalho é o que tomamos como referência de implementação do GROOVE em nossos estudos.

Figura 2: Exemplo do resultado final de uma verificação no GROOVE.



Fonte: Zambon (2013)

2.5 O Método *Iterative Vertex Labeling* - IVL

O *IVL* é um algoritmo de verificação de isomorfismo de grafos do tipo não-direcionados e não-rotulados, proposto por Baroni (2012) originalmente com o nome de *ARIMC*. Em trabalho ainda não publicado [Baroni et al. (2013)], o mesmo algoritmo foi

chamado de *IVL*, e é assim que o referenciamos ao longo do texto.

O IVL é um algoritmo de abordagem direta, ou seja, que compara diretamente dois grafos a fim de determinar se são isomorfos ou não. O desenvolvimento desse algoritmo foi bastante focado em alcançar boa eficiência para grafos com alta regularidade (alto número de automorfismos). Na prática, nas aplicações de grafos na modelagem de sistemas reais, é bastante incomum encontrar grafos com tão alta regularidade. Dessa forma, ao trabalhar no GROOVE, por estarmos tratando de uma aplicação real, a vantagem de mais destaque do IVL é desperdiçada.

Os testes com o IVL [Baroni (2012)] mostram que ele é competitivo em relação ao *bliss*, *saucy* e *nauty* para várias classes de grafos, obtendo performances consideravelmente melhores para os grafos com alta densidade de arestas e com alto grau de regularidade. Esses resultados mostram que, apesar de tratar um tipo de grafo diferente, e estar principalmente focado nos grafos com alta densidade de arestas e alta regularidade, testar a adaptação do IVL ao GROOVE é bastante apropriado.

A ideia geral do IVL é que cada nó tenha um rótulo que agregue suas informações e as de seus vizinhos (análogo ao que chamamos no GROOVE de certificados). Primeiramente, o nó só contém informações referentes a ele mesmo. Em seguida, serão incorporadas informações dos vizinhos imediatos, e depois dos vizinhos dos vizinhos, e assim sucessivamente, até que essas informações incorporadas não estejam mais diferenciando os nós entre si.

Após efetuar essas iterações sobre os dois grafos que serão comparados, deve-se averiguar se os dois grafos tem conjuntos de rótulos idênticos e se é possível estabelecer um morfismo entre os dois grafos com base nesses rótulos. Se ambas as respostas são positivas, podemos afirmar que eles são grafos isomorfos. A forma de calcular esses rótulos é mostrada em detalhes no capítulo 3.

Apesar do IVL ser classificado como um algoritmo de abordagem direta (compara diretamente um par de grafos), o cálculo dos rótulos é iterado independentemente para cada grafo, estabelecendo-se se há o isomorfismo ou não somente quando ambos já tem todos os rótulos de nós calculados. Essa abordagem faz com que facilmente esses rótulos de nós, ao serem combinados, se transformem em um *hash* do grafo, o que remete à abordagem de rotulação canônica do *nauty*. Uma boa estratégia para o cálculo dos rótulos pode alcançar uma efetividade muito próxima de uma rotulação canônica. Em linhas gerais, podemos notar que o IVL se estabelece como uma abordagem muito análoga ao algoritmo já implementado no GROOVE. Essa comparação será exposta de forma esquemática no próximo capítulo.

2.6 Gramáticas de Teste

As gramáticas de teste utilizadas neste estudo são as mesmas testadas em Rensink (2007) e Rensink (2010), além da *Bad tic-tac-toe*. Essas gramáticas são testadas com tamanhos de grafo inicial diferentes ao longo do estudo. Conforme os resultados forem apresentados, esses tamanhos serão especificados.

Append modela a inserção de um elemento no fim de uma fila, quando vários métodos são chamados ao mesmo tempo para realizar essa tarefa. A gramática é caracterizada pelo tamanho da fila e o número de invocações concorrentes.

Ad-hoc é o modelo de uma rede *ad-hoc*, que consiste de um conjunto de nós conectados, mas sem a presença de um servidor central. Cada nó mantem uma visão limitada dos seus vizinhos mais próximos, conhecendo então apenas um pedaço da rede. Os parâmetros deste problema são quantos nós tem essa rede, e qual o alcance da visão de cada um dos nós.

Gossip modela o problema "*gossiping girls*" descrito em Hurkens (2000). Todos os grafos gerados ao longo da exploração tem o mesmo número de nós, correspondente ao número de garotas a fazer fofocas.

Filósofos é o modelo para o conhecido problema do jantar dos filósofos. O número de nós dos grafos gerados é fixo (o dobro do número de filósofos), o que gera uma grande simetria nesse problema.

Bad tic-tac-toe é o jogo da velha. O número de nós é fixo mas o problema não tem simetria, já que os campos de marcação do jogo da velha são diferenciados entre si.

A gramática *Append* é a única em que variam o número de nós, e, dentre as gramáticas, é a que contem os maiores grafos (na ordem de 20-30 nós nos exemplos testados ao longo do trabalho). As outras tem um número de nós fixo, e foram escolhidas pelo alto número de automorfismos em seus estados, o que dificulta a diferenciação de grafos não-isomorfos. A *Bad tic-tac-toe* é dita *bad* por ter um número pequeno de possíveis estruturas de conexões dos grafos, mas rótulos de arestas que os diferenciam entre si, de forma que a gramática não gera estados isomorfos. Com as estruturas dos grafos sendo iguais, os algoritmos de isomorfismo que analisem principalmente a estrutura de conexões dos grafos podem tomar muito tempo até que consigam diferenciar esses estados. A *Bad tic-tac-toe* é o único cenário a ser testado em que os algoritmos de isomorfismos só oneram o processo de exploração da gramática, sem trazer nenhum benefício em termos de memória utilizada na execução nem em performance. .

3 Adaptação do IVL para o GROOVE

Neste capítulo apresentamos o IVL e o algoritmo de isomorfismo do GROOVE em detalhes, vislumbramos quais são as possibilidades de adaptação do IVL ao GROOVE, e discutimos os resultados obtidos quando eles são executados na exploração de espaço de estados do GROOVE.

3.1 Aspectos Gerais de Ambos os Algoritmos

Nesta seção são apresentados:

1. O que são os certificados de grafos de uma maneira mais formal, com alguns exemplos.
2. O contexto em que se encaixam os algoritmos de isomorfismo.
3. A sequência de processos presentes nos algoritmos de isomorfismo.

Uma definição comum de atributos de grafos que estaremos utilizando bastante neste capítulo é o conceito de *vizinhos n-dist*.

Definição 3.1.1 (Vizinhos N-dist de um Nó)**.** *São considerados vizinhos de um determinado nó aqueles nós que podem ser alcançados a partir de suas arestas. Vizinhos* 1-dist *são aqueles que estão a uma aresta de distância. Os vizinhos* n-dist *são aqueles a n arestas de distância.*

3.1.1 Cálculo de Certificados

Tanto o IVL quanto o GROOVE usam o cálculo de certificados de nós (conforme definido na seção 2.2) como principal indicador de isomorfismos.

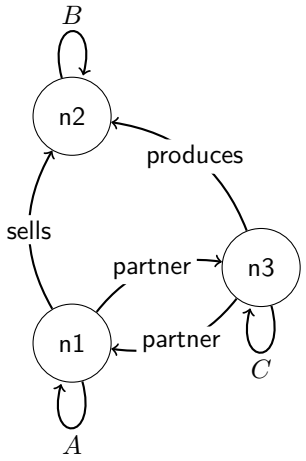
Definição 3.1.2 (Certificados de Nós)**.** *Os certificados de nós c_V de um grafo G são uma coloração π (definição 2.3.1) em que para cada nó é associado um número* inteiro *de acordo com uma função sobrejetiva.*

$c_V : G \to \pi$, *tal que* $\pi = \{(V_G, \mathbb{Z})\}$.

Um exemplo de coloração de nós pode ser visto na figura 3. Em (a) temos um grafo direcionado e rotulado G, e em (b) seus certificados de nós $c_V(G)$, que são definidos como o número de arestas saindo do nó. Assim $c_V(G) = \{(n1, 3), (n2, 1), (n3, 3)\}$.

Figura 3: Exemplo de certificados de nós.

(a) Grafo G direcionado e rotulado.



(b) Certificado de nós correspondente.

Nós	c_V
n1	3
n2	1
n3	3

Fonte: da autora

Figura 4: Exemplo de certificados de arestas para o grafo G.

Arestas	c_A
(n1, A, n1)	1
(n2, B, n2)	1
(n3, C, n3)	1
(n1, sells, n2)	5
(n1, partner, n3)	7
(n3, partner, n1)	7
(n3, produces, n2)	8

Fonte: da autora

Definição 3.1.3 (Certificados de Arestas)**.** *Analogamente ao certificado de nós, o certificado de arestas é uma função do tipo* $c_A : G \to \{(A_G, \mathbb{Z})\}$.

Para o mesmo grafo da figura 3(a), podemos definir, por exemplo, os certificados de arestas como sendo $c_A(a \in A_G) =$ número de caracteres do rótulo de a. Assim, temos os certificados para as arestas de G conforme a figura 4.

Definição 3.1.4 (Certificado de Grafo)**.** *Um certificado de grafo* $C : G \to \mathbb{Z}$ *é um* hash *de G.*

Normalmente $C(G)$ é calculado em função dos certificados de nós e de arestas. Considerando os exemplos das figuras 3 e 4, se, por exemplo, definirmos $C(G) = \sum_{v \in V_G} c_V(v) + \sum_{a \in A_G} c_A(a)$ então $C(G) = 37$.

Ambos os certificados de nós e de grafo são calculados com base em atributos dos grafos, ou seja, características invariantes do grafo em questão. Por esse motivo, tanto

Rensink (2010) quanto Baroni (2012) se referem ao certificados de nós como *invariantes de nós* e ao certificado de grafo como *invariante de grafo*.

Os certificados de nós, de arestas e de grafo são definidos de forma que, se definimos os certificados elementares $c(G) = c_V(G) \cup c_A(G)$ são válidas as seguintes propriedades:

1. se $G \simeq H \rightarrow c(G) = c(H)$ e $C(G) = C(H)$
2. se $c(G) \neq c(H)$ ou $C(G) \neq C(H) \rightarrow G \not\simeq H$

É importante notar que se G e H tem os mesmos certificados, não podemos concluir nada em relação a isomorfismo. Se $c(G) = c(H)$ ou $C(G) = C(H)$ e $G \not\simeq H$ dizemos que o cálculo de certificados gerou um *falso positivo*. O número de falsos positivos gerados por um algoritmo é um indicador de qualidade: quanto maior esse número, mais distantes estão os certificados gerados por ele de se comportarem como uma rotulação canônica. Na rotulação canônica, $c(G) = c(H)$ ou $C(G) = C(H) \rightarrow G \simeq H$, o que torna a detecção de isomorfismos um problema muito mais fácil, e o número de falsos positivos é *zero*.

Se $G \simeq H$ e $c(G) \neq c(H)$ ou $C(G) \neq C(H)$ chamamos o resultado dos certificados de *falso negativo*. Essa situação *não é possível* nos cálculos de certificados de nenhum dos algoritmos usados neste trabalho. O cálculo é feito usando-se *atributos*, características invariantes dos nós e do grafo (por exemplo, número de arestas, número de vizinhos de um nó, etc), portanto, na pior das situações, teremos um certificado ruim, que não discrimina bem grafos não-isomorfos. Assim, a busca direta do morfismo entre os dois grafos, que é uma parte custosa dos algoritmos, será chamada mais vezes. Como se dá a busca direta e o porquê de ela ser custosa são pontos abordados nas subseções 3.2.4 e 3.3.4, separadamente para ambos os algoritmos, já que eles executam essa etapa de maneiras um pouco diferentes. A propriedade 2, se bem explorada, pode economizar bastante tempo computacional na detecção de isomorfismo. Essa ideia é o foco de estudo do capítulo 4.

Usaremos esses certificados para executar o algoritmo 3 na subseção 3.1.2, que descreve a inserção de um novo estado no espaço de estados explorado.

3.1.2 Contexto do Algoritmo de Isomorfismo

No capítulo 2 foi apresentado o algoritmo 2, que detecta se o estado recém-gerado é novo ou não e retorna um valor booleano. Essa é essencialmente a parte do GROOVE que utiliza o algoritmo de isomorfismo. Aqui mostramos mais detalhes desse mesmo algoritmo, com a diferença que ele retorna a função isomorfismo entre os dois grafos, ao invés da

variável booleana. Se ele retorna *null*, significa que não há isomorfismo entre os grafos.

Algoritmo 3: Algoritmo para inserção de um novo estado no STG.

Entrada: grafo G, conjunto de grafos que já foram explorados $\mathcal{S}$
Saída: função isomorfismo f

```
1 B := {H ∈ S | C(H) = C(G)};
2 para todo H ∈ B tal que certificado de nós c_V(H) = c_V(G) faça
3     se ∃ função isomorfismo f : G → H tal que f é um isomorfismo de G em H e
        f ⊆ c(H)^{-1} ∘ c(G) então
4         retorna f
5     fim
6 fim
7 S := S ∪ G
  retorna nulo
```

No algoritmo 3 desejamos inserir o grafo G no conjunto de grafos $\mathcal{S}$ que já foram explorados. Se há um grafo $H \in \mathcal{S}$ tal que $H \simeq G$, então G já está representado em $\mathcal{S}$, e não será inserido. Caso contrário, inserimos G em $\mathcal{S}$.

Na linha 1, o algoritmo seleciona todos os grafos que tem o mesmo *hash* de G e compõe com esses grafos o conjunto B. A linha 2 checa para cada grafo de B se os certificados de nós também são iguais e, nesse caso, busca na linha 3 se há uma função isomorfismo que mapeie G em H, de forma a respeitar os certificados de nós e de arestas calculados para os dois grafos. Se for achado algum grafo que tenha isomorfismo com G, é retornada a função isomorfismo f na linha 4. Se não encontrar, nenhuma f, é constatado então que G não é isomorfo a nenhum grafo de $\mathcal{S}$ e retorna nulo.

Uma vez definido o contexto em que o algoritmo de isomorfismo é chamado, podemos notar que os dois algoritmos vão apresentar vários processos em comum.

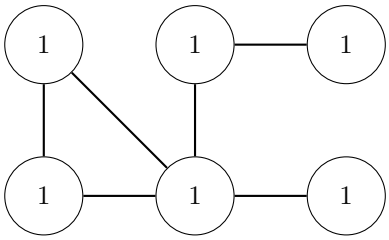
3.1.3 Sequência Geral dos Algoritmos de Isomorfismo

Podemos esquematizar a busca por isomorfismos como sendo constituída de quatro etapas, sendo elas:

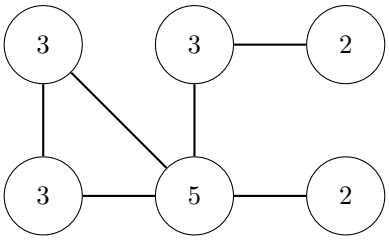
Pré-inicializacão Tenta estabelecer um critério simplificado para identificar não-isomorfismos (por exemplo, se os dois grafos tem o mesmo número de nós). Com isso, evita-se o cálculo de certificados, que pode ser uma etapa lenta. A pré-inicialização faz mais sentido na checagem direta de isomorfismo entre dois grafos, já que na determinação de isomorfismo em relação a um conjunto de grafos, o *hash* é sempre calculado e é muito mais discriminante que os critérios simples que normalmente

Figura 5: Exemplo de iteração de certificados.

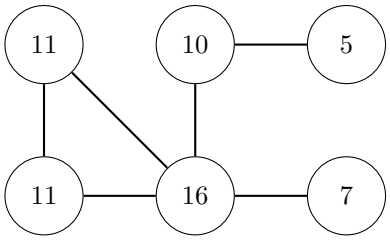
(a) Grafo G e seus certificados de nós.



(b) Certificados de nós após a 1ª iteração.



(c) Certificados de nós após a 2ª iteração.



Fonte: da autora

são usados nesta etapa.

Inicialização dos valores de certificados Determinar um valor inicial para os certificados. São exemplos os certificados definidos nas figuras 3 e 4. Essa etapa pode ter um custo muito baixo (se por exemplo todos os nós receberem um único valor) ou mais alto (se for usado um critério do tipo *soma dos graus de saída dos nós da vizinhança 2-dist*). Um bom critério economiza o número de iterações a serem realizadas na etapa seguinte, mas, em contrapartida, iniciar o cálculo de certificados já com uma etapa custosa pode não ser interessante. Há aqui uma solução de compromisso entre essa etapa e a seguinte.

Iteração do cálculo dos certificados Cálculo com base nos valores dos certificados de nós (ou de arestas) adjacentes, até que se atinja o critério de parada. Por exemplo, seja G um grafo não-direcionado e não-rotulado, conforme a figura 5. Em (a) temos os valores iniciais arbitrários dos certificados representados em cada um dos nós. Supondo que a iteração dos certificados para $v \in V_G$ se dê como $c_V(v) = c_V(v) + \sum_{v' \in V'} c_V$, onde V' é o conjunto dos nós vizinhos a v. Em (b) estão apresentados os novos valores de certificados, após a primeira iteração. Em (c), os certificados após a segunda iteração.

O critério de parada para os dois algoritmos é quando se estabiliza o número de partições de nós. Caso o domínio dos valores de certificados seja ilimitado $(] - \infty, +\infty[)$, a iteração do cálculo dos certificados leva a duas situações: uma operação de refinamento, ou estabilização do número de partições. Ou seja, temos a garantia

de que a iteração dos certificados não leva a um número de partições menor que o da iteração anterior. Apesar de não ser possível garantir isso na prática (pois o domínio dos números inteiros é finito), o domínio dos valores de certificados ainda é grande e portanto essa propriedade é válida. Baroni (2012) constatou essa propriedade empiricamente para o IVL.

Busca direta Checa primeiro se os *hashs* são diferentes. Depois, se os certificados de nós são diferentes. As duas situações levam a conclusão de que não há isomorfismo. Porém, se os certificados de nós são iguais, deve-se fazer a busca direta, tentando estabelecer um isomorfismo entre os grafos que respeite os certificados de nós e de arestas. A busca direta tem um custo que depende bastante da qualidade dos certificados calculados, ou seja, a capacidade de discriminar os nós em partições distintas. Quanto mais refinadas as colorações, mais rapidamente é estabelecido o isomorfismo entre os dois grafos.

Com essa visão geral em mente, podemos então apresentar as características específicas de cada um dos algoritmos de uma forma esquemática.

3.2 Funcionamento do GROOVE

Nesta seção será apresentado com detalhes cada uma das etapas do algoritmo de detecção de isomorfismos do GROOVE descritas na subseção anterior.

3.2.1 Pré-inicialização

A ideia principal da pré-inicialização é evitar passar pelo cálculo de certificados, que pode ser uma etapa bastante custosa. Porém, no GROOVE essa etapa é quase suprimida. Ao gerar um novo grafo G, a primeira providência é calcular os certificados de nós, a fim de gerar o *hash* de G. Isso acontece porque o *hash* do grafo, além de servir para a determinação de isomorfismos, serve também como *hash* para a estrutura de armazenamento dos grafos gerados no GROOVE. Ou seja, *antes* de fazer-se a pergunta "G é isomorfo a algum dos estados que já conheço?", o *hash* de G já está calculado. Essa lógica fica evidente na linha 2 do algoritmo 3.

A pré-inicialização é executada assim que é feita a pergunta de isomorfismo, checando o número de nós e de arestas de G comparativamente aos grafos que possuem o mesmo *hash*. Muito dificilmente os grafos serão diferenciados nessa etapa, ou seja, a pré-inicialização é praticamente sem efeito. Portanto, uma vez que os certificados de nós de todos os grafos já estão calculados, responder a pergunta de isomorfismo no GROOVE faz com que a próxima etapa já seja a busca direta.

3.2.2 Inicialização

Os certificados de nós são inicializados como os *hashcodes* dos rótulos das arestas do tipo laço sobre o nó, ou, se não há arestas desse tipo, como um valor *default*. No GROOVE são calculados também certificados para as arestas (a ser explicado na próxima subseção), e eles são inicializados também com os *hashcodes* dos rótulos da aresta.

3.2.3 Iteração

A iteração dos certificados no GROOVE é também chamada de etapa de *refinamento*, já que divide os nós do grafo em questão em um número cada vez maior de partições, em subdivisões cada vez mais finas, até que essas partições se estabilizem. O número de partições definidas para um dado certificado de nós $c_V(G)$ de um grafo G será denotado por $|c_V(G)|$. No GROOVE, além dos certificados de vértices, são calculados também certificados de arestas, denotados por $c_A(G)$. Esses últimos ajudam a iterar os certificados de nós graças à estrutura de dados do GROOVE. Nessa estrutura, são armazenadas as arestas e seus nós fonte e sumidouro. Listar quais são os nós vizinhos de um determinado nó é uma consulta *mais custosa e indireta* pois passa por consultar todas as arestas que entram/saem desse nó, e quais são seus respectivos nós fonte/sumidouro. Essa estrutura de dados é um aspecto importante do GROOVE que será revisitado na seção 3.5.

No GROOVE a iteração dos certificados (ou refinamento) é dividida em duas etapas, sendo a primeira o cálculo de certificados em si, e a segunda, uma mudança de estratégia do cálculo de certificados a fim de diversificar os certificados e chegar nas partições unitárias. Essa segunda etapa é chamada de *quebra de simetria*. Descreveremos a primeira etapa de refinamento de acordo com o pseudocódigo apresentado no algoritmo 4.

Algoritmo 4: Algoritmo do GROOVE para primeira etapa de iteração de certificados de nós.

Entrada: certificados elementares iniciais (c_V^0, c_A^0)

Saída: certificados elementares finais $c^1 = (c_V^1, c_A^1)$

1 $(c_V, c_A) = (c_V^0, c_A^0)$;
2 **repita**
3 $\quad (c_V^1, c_A^1) = (c_V, c_A)$;
4 $\quad$ **para** *todo* $a \in A_G$ **faça**
5 $\quad\quad c_A(a) \coloneqq f_a(c_V^1(fonte(a)), c_V^1(sumid(a)), c_A^1(a))$ `// certificado de arestas iterado`
6 $\quad$ **fim**
7 $\quad$ **para** *todo* $v \in V_G$ **faça**
8 $\quad\quad c_V(v) \coloneqq g_v(\{c_A(a)|v = fonte(a)\}, \{c_A(a)|v = sumid(a)\})$ `// certificado de nós iterado`
9 $\quad$ **fim**
10 **até** $|c_V^1| = |c_V|$;
`// iterar até que o número de partições estabilize`
11 **retorna** (c_V, c_A)

No laço das linhas 2 a 10 do algoritmo 4 vemos que ele será iterado enquanto o novo cálculo (c_V, c_A) tiver mais partições que o cálculo anterior (c_V^1, c_A^1). Os cálculos se dão nas linhas 5 e 8. A linha 5 calcula os novos certificados de cada aresta $c_A(a)$ como uma função f_a dos certificados atuais do nó-fonte, do nó-sumidouro e da aresta em questão. A linha 8 calcula os novos certificados de nós c_V como uma função g_v aplicada sobre os certificados de arestas em que ele está contido como nó-fonte ou nó-sumidouro.

No algoritmo implementado atualmente no GROOVE as linhas 5 e 8 são executadas paralelamente, conforme se computa o novo certificado de cada uma das arestas. Isso torna o código eficiente já que as operações executadas não dependem da ordem em que se percorre os nós e as arestas (graças às operações comutativas). Esse é um aspecto interessante desse algoritmo, já que consultar os vizinhos de um nó é relativamente caro. Podemos ver abaixo a parte do código referente ao algoritmo 4.

```
1 this.initValue = this.label.hashCode();
2 protected int computeNewValue() {
3             int targetShift = (this.initValue & 0xf) + 1;
4             int sourceHashCode = this.source.value;
5             int targetHashCode = this.target.value;
6             int result =
7                 ((sourceHashCode << 8) | (sourceHashCode >>> 24))
8                     + ((targetHashCode << targetShift) | (targetHashCode >>> targetShift))
```

```
9                   + this.value;
10              this.source.nextValue += 2 * result;
11              this.target.nextValue -= 3 * result;
12              return result;
13          }
```

No código, é calculado o novo certificado de uma aresta. Define-se o *targetShift* como função do *hashcode* do rótulo da aresta (*initValue*). Ele será o tamanho do *shift* binário a ser feito no valor do certificado. A ideia do *shift* é dispersar os bits do valor do certificado a fim de diferenciá-los melhor, aproveitando o fato de que uma variável do tipo *int* tem 32 bits no código Java. O novo valor do certificado de aresta (*result*) é uma função f_a (linhas 6 a 12) do certificado de seu nó-fonte (*sourceHashCode*) e do seu nó-sumidouro (*targetHashCode*). Os novos valores de certificados para os nós fonte e sumidouro (*this.source.nextValue* e *this.target.nextValue* respectivamente) incorporam o valor de (*result*). As operações são comutativas (*or*, *xor*, soma, subtração) e por isso os valores finais dos certificados independem da ordem em que as arestas foram percorridas. Aparentemente esse código foi escolhido empiricamente, mas Rensink (2010) não deixa claro os meios em que se chegou a essa implementação.

Um outro aspecto das iterações de certificados de nós é que, ao invés de o novo valor de certificado calculado substituir o antigo, eles são combinados por uma operação *xor*. Isso preserva a informação obtida previamente, incorporando as novas informações sem descartar as antigas. Podemos ver isso no trecho de código a seguir, referente à iteração dos certificados de nós, executado após o cálculo de certificado de todas as arestas.

```
1 protected int computeNewValue() {
2           int result = this.nextValue ^ this.value;
3           this.nextValue = 0;
4           return result;
5       }
```

No código, o novo valor de certificado de nó (*result*) é a combinação do valor atual (*this.value*) com o novo valor (*this.nextValue*), obtido após as iterações sobre todas as arestas. A segunda etapa de iteração dos certificados, a quebra de simetria, tem o objetivo de discriminar nós que estejam em partições que não sejam unitárias. A estratégia é variar o modo de calcular os certificados, a fim de distinguir esses nós. A seguir é apresentado o pseudocódigo da quebra de simetria.

Algoritmo 5: Algoritmo do GROOVE para segunda etapa de iteração de certificados de nós: quebra de simetria

Entrada: certificado de nós $c_V(G)$

Saída: novo certificado de nós $c'_V(G)$

1 $c = F(c)$;

2 $V' = \{v \in V_G | \exists v' \in V_G : v' \neq v \text{ e } c(v) = c(v')\}$;

3 **para** *todo* $v' \in V'$ **faça**

4 $\quad c'(v') \coloneqq c(v') + F(c(v'))$; `// para os nós com certificados repetidos, mudar suas colorações`

5 **fim**

6 **retorna** $c'_V(G)$

Na linha 1 é operada a transformação F sobre toda a coloração c. Se houverem nós com mesma coloração (conjunto V' na linha 2), eles (e somente eles) terão suas colorações alteradas novamente pela função F na linha 4. Essa função de operações binárias F que atua sobre V' é mostrada no código abaixo.

```
1 public void breakSymmetry() {
2             this.value ^= this.value << 5 ^ this.value >> 3;
3   }
```

Na função *breakSymmetry* o valor do certificado (*this.value*) é alterado com operações binárias. No GROOVE, o certificado de grafo é calculado como sendo o somatório de todos os certificados de nós e arestas, ou seja, para um grafo G, $C(G) = \sum c_V(G) + \sum c_A(G)$.

3.2.4 Busca Direta

A busca direta no GROOVE é a etapa de encontrar a função de isomorfismo f da (linha 5 do algoritmo 3). Ou seja, é a etapa em que se estabelece a correspondência entre os dois grafos que estão sendo comparados, caso eles sejam isomorfos. Antes de fazer essa correspondência o GROOVE checa primeiro se os grafos possuem o mesmo certificado de grafos e os mesmos certificados de nós. Caso contrário, eles não podem ser isomorfos. Se ambos os grafos possuem partições discretas, o mapeamento de nós é direto (nós com os mesmos certificados são correspondentes). O GROOVE, nessa etapa, além de analisar os certificados de nós, checa se os certificados de arestas também são discriminantes entre si.

Para partições não discretas, o mapeamento entre os dois grafos é estabelecido primeiro fazendo as correspondências entre as *arestas* dos dois grafos que tenham os mesmos certificados, e seguidamente, mapeando seus correspondentes nós fonte e sumidouro.

A cada mapeamento de arestas é checado se o mapeamento dos nós continua válido. Por armazenar a estrutura de grafos como um conjunto de arestas (com seus nós fonte e sumidouro), é natural que o GROOVE percorra aresta a aresta para estabelecer esse mapa. Porém, essa busca, para ser eficiente, exige que haja uma boa discriminação entre os certificados de arestas, e o que estivemos analisando o tempo todo como qualidade do certificado é o número de *partições de nós*. Esse é um ponto importante já que a busca direta é o processo mais oneroso entre as etapas enumeradas, e não há uma medida direta de quão bom está o cálculo dos certificados de arestas (quão distinguíveis elas estão entre si).

Na próxima seção veremos como funciona o algoritmo IVL quando subdividido nessas mesmas quatro etapas.

3.3 Funcionamento do IVL

O algoritmo proposto por Baroni (2012) também pode ser apresentado esquematicamente como sendo dividido nas quatro etapas descritas na subseção 3.1.3. Ele também faz uso de certificados do tipo inteiro, definidos para cada um dos nós, mas não trabalha com certificados de arestas nem certificado de grafo. Apesar de implementado para comparações diretas entre dois grafos não-orientados e não-rotulados, o IVL tem essencialmente características bastante parecidas com o que está implementado no GROOVE, como veremos na descrição das etapas a seguir.

Importante notar que, no trabalho original, Baroni (2012) chama as partições de *grupos*, e as informações dos vizinhos de um nó (que irão compor seu certificado) são reunidos em um atributo chamado *multiconjunto*.

3.3.1 Pré-inicialização do IVL

Antes de calcular os certificados (ou invariantes), o IVL checa se os grafos têm o mesmo número de nós, de arestas e se seus nós têm os mesmos graus. Essa última checagem é feita reunindo as informações de graus dos nós de cada grafo em um vetor, e comparando esses vetores ordenados. Importante notar que a implementação original do IVL trabalha com uma estrutura de dados de lista de adjacências para armazenar os grafos. Sendo assim, recuperar a informação de graus dos nós é ler diretamente o tamanho da lista associada ao nó.

3.3.2 Inicialização dos Certificados do IVL

Os certificados de nós são inicializados de acordo com o seguinte algoritmo.

Algoritmo 6: Inicialização dos certificados no IVL

Entrada: grafo $G = (V, A)$
Saída: certificado de nós inicial $c_V^0(G)$

```
se G é regular então
    se G é 2-dist regular então
        para todo v ∈ V faça
            c_V^0(v) = |Γ_3(v)|;
        fim
    fim
    senão
        para todo v ∈ V faça
            c_V^0(v) = |Γ_2(v)|;
        fim
    fim
fim
senão
    para todo v ∈ V faça
        c_V^0(v) = graus(v);
    fim
fim
```

No algoritmo $|\Gamma_i(v)|$ denota o tamanho da vizinhança $i - dist$ em relação ao nó v. Um grafo é dito $i-dist$ regular se $|\Gamma_i(v)|$ tem o mesmo valor para todos os vértices $v \in V$. Dadas essas notações, é trivial entender a inicialização do IVL. Ele busca inicializar os certificados de nós com os valores dos tamanhos de vizinhanças, de forma a discriminar os nós. Vale observar novamente que o IVL não trabalha com certificados de arestas.

3.3.3 Iteração dos Certificados do IVL

O IVL itera os certificados de nós de acordo com o algoritmo 7.

Algoritmo 7: Algoritmo do IVL para iteração de certificados de nós.

Entrada: certificados de nós inicial c_V^0

Saída: certificados de nós final c_V^1

1 $c_V^1 = c_V^0$;
2 $c_V = c_V^0$ xor $f_{disp}(c_V^0)$;
3 **enquanto** $|c_V^1| < |c_V|$ **faça**
4 $\quad c_V^1 = c_V$;
5 $\quad c_V : v \to f_v(c_V^1)$ `// certificado de nós iterado`
6 **fim**
7 **retorna** c_V

Na linha 2, o valor dos certificados iniciais c^0 são atualizados com auxílio da função f_{disp}, chamada *função dispersão*. Essa função é na verdade um vetor de inteiros com 10000 posições, gerados aleatoriamente, e usados para levar os valores iniciais dos certificados (geralmente pequenos) para valores de inteiro maiores. Isso faz com que a representação binária dos certificados faça uso de mais bits, e tire maior proveito do fato de um inteiro ser representado por 32 bits. Assim, ao operarmos a função *xor* em números maiores, a chance de conseguirmos discriminar os nós também é maior. No GROOVE existe essa mesma ideia de dispersar os bits, mas foi feito uso de operações de *shift* binário ao invés de um mapeamento por tabela (função f_a do algoritmo 4).

As linhas 3 a 5 mostram as iterações que serão executadas sobre os certificados de nós, e que serão executadas aplicando a função f_v enquanto o número de partições estiver aumentando. A função f_v da linha 5 é a operação xor feita sobre os certificados de todos os vizinhos de v. A escolha da função *xor*, analogamente às operações escolhidas no GROOVE, tem a ver com o fato de essa ser uma operação comutativa, que independe da ordem dos fatores, ou melhor, da ordem dos elementos do multiconjunto que irá compor o valor do certificado.

Comparando esse algoritmo com o algoritmo 4 podemos notar que eles têm estruturas idênticas. O que os diferencia, em resumo, é o fato de que o GROOVE tem certificados de arestas, de que há uma alteração inicial dos valores de certificados via a função dispersão, e a função utilizada na iteração do certificado de nós.

3.3.4 Busca Direta do IVL

A busca direta após o cálculo dos certificados é feita primeiro checando se os certificados de nós dos dois grafos são iguais, caso contrário eles não podem ser isomorfos. Se são iguais, procura-se o mapeamento entre os dois conjuntos de nós com base em suas partições. A medida que os nós são mapeados com relação aos seus certificados, o algoritmo

checa se as adjacências estão sendo preservadas corretamente. Essa checagem é direta já que as listas de adjacências indicam prontamente quais são as conexões que devem ser mantidas após o mapeamento. No GROOVE, por exemplo, checar as adjacências dessa mesma forma seria muito mais caro, já que a estrutura de dados é bastante diferente.

3.4 Comparação IVL x GROOVE

Foram detalhados nas seções anteriores os dois algoritmos em questão, ressaltando as particularidades de implementação em cada uma das quatro etapas de sua construção. Podemos ressaltar portanto as principais diferenças entre eles:

- O GROOVE é feito para grafos direcionados e rotulados, enquanto o IVL é para não-direcionados e não-rotulados;
- O GROOVE define certificados de arestas, que transmitem as informações contidas nos nós para seus nós vizinhos indiretamente, e que determinam a eficácia na busca direta;
- O GROOVE calcula os certificados assim que os grafos são gerados, já que ele faz uso desse *hash* para o armazenamento dos grafos, enquanto o IVL só calcula os certificados se eles forem necessários na determinação de isomorfismo;
- O IVL faz uso da função dispersão em forma de tabela enquanto o GROOVE faz uso de operações binárias;
- O IVL não calcula certificado de grafos.

Há outras diferenças menores, referentes às etapas de pré-inicialização e de inicialização por exemplo, mas os pontos acima são os principais e que serão os maiores pontos de atenção para a adaptação do IVL ao GROOVE. As diferenças entre cada uma das etapas estão descritas na tabela 1.

Com as semelhanças e diferenças entre os dois algoritmos descritas, podemos na próxima seção listar as possibilidades de adaptação do IVL para o GROOVE.

3.5 Adaptação do IVL para o GROOVE

Com a lista de diferenças entre os algoritmos detalhada na seção anterior, foi possível organizar as ideias para a adaptação. Cada um dos pontos-chave está descrito abaixo, e os algoritmos propostos estão resumidos na tabela 2.

Tabela 1: Tabela de comparação: GROOVE vs IVL

Etapas	**GROOVE**	**IVL**
Pré-inicialização	Checa número de arestas e nós. (Realizado depois do cálculo dos certificados)	Checa número de arestas, nós e graus dos vértices.
Inicialização	nós: hash do rótulo do nó ou valor default; arestas: hash do rótulo da aresta;	nós: tamanho da vizinhança 1, 2, ou 3-dist.
Iteração	nós: XOR entre certificado atual e uma função dos certificados dos nós e arestas adjacentes; arestas: função dos certificados dos nós fonte e sumidouro; (Itera até número de partições estabilizar)	nós: XOR entre certificado atual e certificados dos nós vizinhos; (Itera até número de partições estabilizar)
Busca Direta	Constrói o mapeamento entre os dois grafos comparando primeiramente os certificados de arestas, e validando os certificados de nós.	Constrói o mapeamento entre os dois grafos consultando os certificados de nós, e validando as adjacências determinadas pelas arestas.

Fonte: da autora

1. Adaptar o IVL para grafos orientados exige dividir os vizinhos em duas categorias: vizinhos-fonte e vizinhos-sumidouro. Eles são tratados diferentemente ao longo do cálculo de certificados (inclusive com tabelas de dispersão diferentes, com 1000 inteiros ao invés dos 10000 do IVL original para cada uma das tabelas).

2. Adaptar para grafos rotulados requer usar essas informações de rótulo no cálculo dos certificados. Elas são usadas no cálculo do certificado de arestas, que são incorporados ao valor do certificado do grafo.

3. É necessário estabelecer um cálculo para certificados de arestas, de forma que esses certificados são iterados, mas incorporados somente à informação de certificado de grafo (diferente do GROOVE, em que esses valores são transmitidos aos nós).

4. As tabelas de dispersão são utilizadas aqui em todas as iterações do cálculo de certificados (diferente do IVL em que só eram utilizadas na primeira iteração). A ideia é substituir de fato as operações binárias que funcionam como função de dispersão no GROOVE.

5. O cálculo de certificados pelo IVL também é executado assim que o grafo é gerado. O *hash* gerado pelo IVL é utilizado para o armazenamento de grafos, assim como o *hash* gerado originalmente no GROOVE.

6. Como no GROOVE o número de graus de cada nó é uma informação indireta, testes mostraram ser muito lenta a inicialização conforme o IVL. Optou-se portanto por

inicializar os certificados da mesma forma que o GROOVE, usando como informação base os rótulos de nós e arestas.

7. A busca direta, pela estrutura de dados do GROOVE, não pode ser alterada. Assim, os certificados de arestas são os primeiros a serem consultados ao estabelecer o mapeamento entre os nós dos dois grafos (em detrimento dos certificados de nós, usados na busca direta do IVL).

8. É necessário definir uma fórmula de cálculo de certificados de grafo para o IVL. Na maioria das propostas usa-se a mesma fórmula do GROOVE.

9. Na iteração dos valores dos certificados, aqui optamos por utilizar o certificado atual em conjunto com as informações dos vizinhos para compor o valor dos certificados iterado. No IVL original, o certificado é dado somente em função dos certificados dos nós vizinhos, desprezando-se o valor atual do certificado do nó que está sendo iterado.

As propostas de adaptação do IVL ao GROOVE estão resumidas na tabela 2. Nela, os campos indicados com "GROOVE" indicam que foi mantida a sua implementação atual. Foram propostas quatro adaptações a serem testadas com as gramáticas de teste descritas na seção 2.6. A etapa de pré-inicialização é realmente suprimida nessas implementações. Os certificados de arestas nos IVLs adaptados são iterados da seguinte forma:

$$cert(aresta) = cert(aresta) \text{ xor } cert(nó - fonte) \text{ xor } cert(nó - sumidouro);$$

Os certificados de nós nos IVLs adaptados foram iterados da seguinte forma:

$$cert(nó - fonte) = cert(nó - fonte) \text{ xor } tabela - fonte(cert(nó - sumidouro));$$
$$cert(nó - sumidouro) = cert(nó - sumidouro) \text{ xor } tabela - sumidouro(cert(nó - fonte));$$

Essas formas de calcular os certificados serão referenciadas na tabela como "IVL". A quebra de simetria foi testada também com os IVLs, tal como implementada no GROOVE. Os resultados obtidos nos testes estão discriminados na próxima seção.

3.6 Resultados Computacionais

As gramáticas utilizadas nos testes estão descritas na seção 2.6. Foram usadas todas elas, exceto a *Bad tic-tac-toe*, que no tamanho adequado aos testes (4x4) não pode ser executada com a memória disponível na máquina. A máquina utilizada foi uma estação de trabalho HP Z800, com processador Intel com 12 núcleos,e memória RAM de 32GB. O código foi feito em Java, compilado e executado com o com o JDK 1.7.0_79.

Tabela 2: IVLs adaptados

	IVL 1	**IVL 2**	**IVL3**	**IVL 4**
Inicialização	Nós: GROOVE Arestas: GROOVE	Nós: GROOVE Arestas: GROOVE	Nós: • fonte: Troca o $\sum$ do GROOVE por XOR • sumidouro: Troca o $\sum$ do GROOVE por XOR (XOR com certificado $<<$ 1) Arestas: GROOVE	Nós: GROOVE Arestas: GROOVE
Iteração	IVL	IVL	IVL	Nós: IVL Arestas: cert(aresta) += cert(nó-fonte) - cert(nó-sumidouro)
Busca Direta	GROOVE	GROOVE	GROOVE	GROOVE
Certificado de Grafo	GROOVE	XOR(certs(nós)) ^XOR(certs(arestas))	GROOVE	GROOVE

Fonte: da autora

As gramáticas testadas utilizaram as seguintes configurações:

- Append: fila de tamanho 8, com 4 invocações concorrentes;
- Ad-hoc: rede de 7 nós com visão de amplitude 2;
- Gossip: 8 garotas;
- Filósofos: 10 filósofos.

Foram obtidos os resultados computacionais da tabela 3.

Tabela 3: Resultados Computacionais dos IVLs Adaptados

	Algoritmo	GROOVE		IVL1		IVL2		IVL3		IVL4	
	Quebra de Simetria	Não	Sim	Não	Sim	Não	Sim	Não	Sim	Não	Sim
Append	Isos verificados	708541	708541	708547	708541	708569	708571	708570	708828	708552	708542
	Isos em busca direta	27777	990	27777	1254	28508	990	27777	28335	28968	1254
	Falsos Positivos $C(G)$	0	0	0	0	23	25	9	5	4	5
	% dos detectados	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	Falsos Positivos $c_V(G)$	0	**0**	0	**0**	0	0	0	**0**	0	**0**
	% dos detectados	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	Tempo total (ms)	83380	**82933**	81130	**80820**	82636	84176	84394	**84184**	80841	**82736**
	Tempo iso (% do total)	23.4%	29.1%	26.6%	32.5%	25.7%	33.5%	26.8%	29.2%	31.0%	30.2%
Gossip	Isos verificados	9889107	9887247	9888524	9887311	94254628	94217443	9888887	9910713	9888943	9887608
	Isos em busca direta	6585856	6562890	6562984	6562904	6590494	6559479	6562832	22080	6562745	6562844
	Falsos Positivos $C(G)$	1734	99	1646	56	84367304	84330373	1648	32	1708	23
	% dos detectados	0.0%	0.0%	0.0%	0.0%	89.5%	89.5%	0.0%	0.0%	0.0%	0.0%
	Falsos Positivos $c_V(G)$	1601	**6**	1547	**2**	23699360	23869023	1586	**0**	1558	0
	% dos detectados	0.0%	0.0%	0.0%	0.0%	25.1%	25.3%	0.0%	0.0%	0.0%	0.0%
	Tempo total (ms)	2406041	**2596037**	2531423	**835359**	2932855	3178944	2621603	2513458	2792145	2706682
	Tempo iso (% do total)	18.2%	29.3%	17.5%	30.2%	17.7%	34.7%	19.5%	41.4%	17.2%	30.5%
Filósofos	Isos verificados	9762334	9762335	9766945	9766873	10289392	10280948	9766932	9781019	9783321	9782974
	Isos em busca direta	3646891	5360	3657558	15680	3514718	16306	3710850	3730167	3518434	16691
	Falsos Positivos $C(G)$	23	23	4496	4417	527033	518586	4525	0	20916	20562
	% dos detectados	0.0%	0.0%	0.0%	0.0%	5.1%	5.0%	0.0%	0.0%	0.0%	0.2%
	Falsos Positivos $c_V(G)$	23	**0**	4496	**4417**	443582	374030	4525	**0**	20916	**4453**
	% dos detectados	0.0%	0.0%	0.0%	0.0%	4.3%	3.6%	0.0%	0.0%	0.0%	0.0%
	Tempo total (ms)	517419	**479961**	536826	**486970**	548683	496191	537994	**539988**	530554	**494092**
	Tempo iso (% do total)	26.0%	31.0%	29.3%	33.4%	27.5%	32.2%	30.1%	27.7%	31.4%	34.5%
no-hops	Isos verificados	1508888	1395793	1509176	1395728	1512136	1397410	1508402	1401044	1508311	1395810
	Isos em busca direta	860230	44901	838637	44734	835923	44951	843251	847442	843623	46281
	Falsos Positivos $C(G)$	113120	30	113464	24	116244	1372	112577	45	112606	83
	% dos detectados	7.5%	0.0%	7.5%	0.0%	7.7%	0.1%	7.5%	0.0%	7.5%	0.0%
	Falsos Positivos $c_V(G)$	113119	**2**	**113463**	**0**	113648	903	112574	**44**	112599	**0**
	% dos detectados	7.5%	0.0%	7.5%	0.0%	7.5%	0.1%	7.5%	0.0%	7.5%	0.0%
	Tempo total (ms)	253486	**238983**	259912	**239596**	263188	244388	258406	**244825**	259531	**242541**
	Tempo iso (% do total)	9.3%	13.3%	10.9%	8.7%	10.3%	9.5%	9.3%	16.0%	11.6%	11.8%

Fonte: da autora

Na tabela 3, encontramos nas colunas os algoritmos de cálculo de certificados, o original do GROOVE e os IVLs adaptados, com ou sem o uso da etapa de quebra de simetria. Nas linhas temos as gramáticas testadas, e oito estatísticas observadas ao executarmos essas gramáticas:

Isos verificados: número de vezes em que a detecção de isomorfismos foi acionada;

Isos em busca direta: número de isomorfismos que foram detectados na busca direta;

Falsos Positivos $C(G)$: número de vezes em que $C(G)$ indicou isomorfismo mas na verdade os grafos não eram isomorfos;

% dos detectados: porcentagem de falsos positivos $C(G)$ em relação ao número de isomorfismos verificados;

Falsos Positivos $c_V(G)$: número de vezes em que $c_V(G)$ indicou isomorfismo mas na verdade os grafos não eram isomorfos;

% dos detectados: porcentagem de falsos positivos $c_V(G)$ em relação ao número de isomorfismos verificados;

Tempo total (ms): Tempo total de execução do GROOVE em milissegundos;

Tempo iso (% do total): porcentagem do tempo total que foi gasto com a detecção de isomorfismos;

A tabela é bastante densa mas apresenta os resultados dos algoritmos basicamente em termos de eficácia (número de falsos positivos) e de performance (tempo computacional). A partir dela, podemos fazer as seguintes constatações (os dados em negrito na tabela 3 são os dados destacados nas conclusões a seguir):

- As adaptações do IVL, com exceção do IVL2, tem eficácias e tempos computacionais parecidos com o que já está implementado no GROOVE.
- Eleger o melhor algoritmo é uma tarefa difícil, pois os resultados dependem bastante das gramáticas testadas. Por exemplo, o IVL1 tem menos falsos positivos $c_V(G)$ e tempo computacional bem menor que o GROOVE para a gramática *Gossip*. Esses mesmos índices são piores para o IVL1 quando olhamos a gramática *Filósofos*.
- A quebra de simetria é uma etapa que melhora muito a eficácia de todos os algoritmos. Por exemplo, para o IVL1, a quebra de simetria reduz os falsos positivos de $c_V(G)$ de 113463 a 0 para a gramática *no-hops*.
- O IVL1 com quebra de simetria só não é melhor que o GROOVE em eficácia para a gramática *Filósofos*.

- O IVL 3 com quebra de simetria mostra boas eficácias mas performances piores para todas as gramáticas.
- Comparando a performance do IVL1 com quebra de simetria com os outros IVLs com quebra de simetria, podemos constatar que o IVL1 é mais rápido para todas as gramáticas. O IVL1 é em média 4% mais rápido que os outros IVLs para a gramática *Append*, 70% para a *Gossip*, 5% para a *Filósofos* e 2% para a *no-hops*.
- Em relação ao GROOVE o IVL1 chega a ser 78% mais rápido para a gramática *Gossip* e, no outro extremo, é 1,5% mais lento para a gramática *Filósofos*.

3.7 Conclusões

O IVL 1 é a melhor adaptação, já que tem a melhor performance entre os IVLs e só tem menor eficácia que o GROOVE para a gramática *Filósofos*. A única diferença entre o IVL1 e o GROOVE é a função de dispersão. Podemos notar que as tabelas de dispersão são eficazes na diferenciação de grafos, pois produzem menos falsos positivos que a implementação atual do GROOVE. Em termos de performance a diferença em geral é pequena, para mais ou para menos, dependendo da gramática.

Em linhas gerais as adaptações do IVL para o GROOVE não introduzem uma grande mudança no esquema que já está implementado. O GROOVE tem uma estrutura de processos bem definida e adaptar o IVL a essa estrutura "amarra" as possibilidades de adaptação. Um trabalho futuro decorrente desses testes é implementar uma nova forma de armazenar os grafos no GROOVE, que não dependa do cálculo de certificados.

Os processos no GROOVE são dependentes de sua estrutura de dados, que por sua vez fazem uso do cálculo de certificados. É importante notar que os resultados obtidos são a interferência do cálculo de certificados sobre vários processos do GROOVE, não somente a detecção de isomorfismos, já que um certificado que discrimine bem os grafos ajuda a construir um armazenamento dos grafos mais eficiente, e, consequentemente, mais agilidade ao recuperarmos na memória os grafos armazenados na estrutura de dados do GROOVE.

No próximo capítulo o problema de isomorfismo foi isolado, e o cálculo de certificados ficou independente do armazenamento dos grafos em memória. Vamos analisar como se comportam estratégias diversas aplicadas à etapa de pré-inicialização, suprimida no contexto deste capítulo.

4 Determinação de Filtros para Identificar Não-Isomorfismos

Neste capítulo, vamos propor uma forma diferente da usada no GROOVE para detectar isomorfismos. No GROOVE, o processo de detecção de isomorfismos "se mistura" com outros processos da exploração do espaço de estados, como por exemplo, o armazenamento dos novos grafos, o que dificulta isolar a análise do problema que estamos focando neste estudo.

A primeira seção é de introdução deste capítulo, explicitando principalmente as motivações. A partir daí, apresentamos os *filtros* (a serem definidos na próxima seção) e seus resultados computacionais.

4.1 Introdução

No capítulo 3 vimos que os algoritmos de isomorfismo aplicados neste trabalho podem ter seus processos divididos esquematicamente em quatro etapas: pré-inicialização, inicialização, iteração dos certificados e busca direta (conforme a seção 3.1.3). Ao apresentar a estrutura do GROOVE na seção 3.2, foi ressaltado que o cálculo de certificados no GROOVE tem o papel não somente de detectar estados isomorfos, mas também, determinar um *hash* que será utilizado na estrutura de armazenamento de grafos. Dessa forma, o *hash* é calculado para *todos* os grafos que são gerados, sendo armazenados somente aqueles que ainda não estejam representados por seus isomorfos no espaço de estados. Esse "duplo papel" do cálculo de certificados acaba influenciando a performance do GROOVE como um todo pois, um bom *hash* não só ajuda a detectar rapidamente isomorfismo como também a recuperar rapidamente um grafo que esteja armazenado na memória. Naturalmente surge então a necessidade de isolar esses efeitos. A pergunta que se busca responder neste capítulo então é: como se comportam os certificados em se tratando somente da determinação de isomorfismos?

Outro aspecto a ser considerado e que foi comentado na seção 3.2 é que, uma vez que os certificados são calculados para todos os grafos, a primeira etapa (de pré-inicialização) fica suprimida. Ela acaba perdendo o sentido, pois se torna um evento muito raro dois grafos que tenham os mesmos *hashs* terem número de nós/arestas diferentes, por exemplo, já que esses são critérios muito mais "grosseiros". Assim, propõe-se neste capítulo partir para um outro cenário, inspirado por, mas fora do GROOVE. Vamos buscar isomorfismos fazendo uso da etapa de pré-inicialização, com o objetivo de encontrar

rapidamente *não-isomorfismos*, ou seja, tentaremos evitar o cálculo de certificados a fim de melhorar a performance na detecção de isomorfismos. Utilizaremos critérios mais simples, mais baratos computacionalmente e deixaremos os certificados para os casos mais difíceis. Cada um desses critérios será chamado de *filtro* ao longo deste capítulo.

Os filtros são testados sobre instâncias das gramáticas de teste já apresentadas na seção 2.6, considerando todos os grafos gerados por elas com a detecção de isomorfismo desligada. Assim, todos os estados gerados ao longo da exploração, mesmo que sejam isomorfos, são mantidos no STG referente à exploração da gramática. Dessa forma podemos testar o banco de grafos dos STGs quanto a isomorfismo independente da ordem em que os estados foram gerados.

O objetivo deste capítulo é responder as seguintes perguntas:

1. Quais filtros propor?
2. Qual é o melhor dos filtros propostos quando cada um deles é testado individualmente?
3. Qual o melhor filtro que podemos compor a partir desses filtros individuais propostos?
4. Como se comportam os cálculos de certificados quando os utilizamos como se fossem filtros (eliminando seu uso para o armazenamento dos grafos)?

Nas seções seguintes são detalhados os filtros propostos, as instâncias das gramáticas testadas e os resultados.

4.2 Filtros detectores de não-isomorfismos

Nesta seção são apresentadas as propostas de filtros que foram testadas, respondendo assim a questão 1 da introdução deste capítulo. Foram propostos seis tipos de filtros, a serem detalhados a seguir, inclusive em termos de implementação. Todos os filtros recebem como entrada um par de grafos a serem comparados quanto a isomorfismo.

1. **Contagem de nós** Compara quantos nós tem os grafos. Se o número de nós é diferente, com certeza os grafos não são isomorfos. Para a instância de um grafo, há um método que permite recuperar essa informação sem esforço computacional adicional.
2. **Contagem de arestas** Análogo à contagem de nós. Também existe método para buscar essa informação sem esforço computacional adicional.

3. **Contagem de rótulos** Conta quantas vezes aparece cada um dos rótulos do grafo. Não há um método que permita recuperar essa informação diretamente, então a implementação não é tão direta quanto nos filtros acima. O pseudocódigo deste filtro é apresentado no algoritmo 8.

Algoritmo 8: Filtro de contagem de rótulos.

Entrada: grafos $G1$, $G2$
Saída: $G1$ e $G2$ podem ser isomorfos?

```
1  CR = conjunto de rótulos nulo;
2  para cada aresta a ∈ A_G1 faça
3  |  adicionarRótulo(CR, rot(a))
4  fim
5  para cada rótulo r ∈ CR faça
6  |  se contaArestas(G1, r)! = contaArestas(G2, r) então
7  |  |  retorna falso
8  |  fim
9  fim
10 retorna verdadeiro
```

O algoritmo 8 funciona da seguinte forma: primeiramente, na linha 2, ele busca no grafo $G1$ todos os rótulos relativos a suas arestas. Uma vez definidos quais são os rótulos, o filtro conta quantas arestas têm esse rótulo em $G2$ e quantas em $G2$ (linha 5). Se para algum rótulo esses valores forem diferentes, os grafos não podem ser isomorfos.

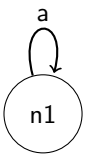
4. **Graus dos vizinhos n-dist** Este filtro percorre os vizinhos de cada um dos nós, armazenando o somatório de todos os graus de saída e dos graus de entrada dos nós percorridos. A implementação é iterativa, percorrendo a distância n em relação aos vizinhos. Nos nossos testes, iteramos até a distância n=1, 2, 3 e 4. Isso significa que o algoritmo buscará vizinhos do tipo fonte/sumidouro até a quarta iteração.

   Nessa implementação, não são considerados vizinhos de um nó o próprio nó (nas arestas do tipo laço) e nem o nó que o precedeu no caminho até ele (na situação de arestas paralelas, com diferença somente de sentido). Essas situações estão ilustradas na figura 6.

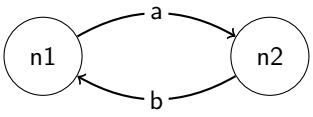
   O pseudocódigo deste filtro é apresentado no algoritmo 9.

Figura 6: Situações em que os nós não são considerados vizinhos no filtro de graus.

(a) Arestas do tipo laço.



(b) Se $n2$ já foi percorrido como vizinho de $n1$, $n1$ não será considerado novamente como vizinho em relação a $n2$.



Fonte: da autora

Algoritmo 9: FILTRO DE GRAUS

Entrada: grafos $G1$, $G2$

Saída: $G1$ e $G2$ podem ser isomorfos?

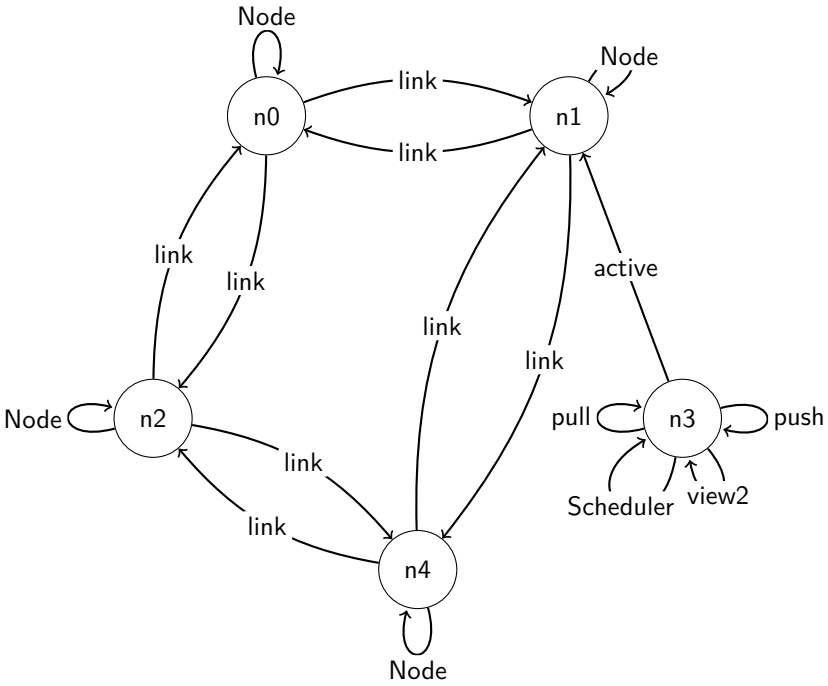
```
1  graus1 = calculaGraus(G1, n); // lista de vetores com as somas dos
       graus para cada nó e para cada dist
2  graus2 = calculaGraus(G2, n);
3  se tamanho(graus1) == tamanho(graus2) então
4  |  se número de vetores distintos em graus1 == número de vetores distintos em
   |   graus2 então
5  |  |  se graus1 == graus2 então
6  |  |  |  retorna verdadeiro
7  |  |  fim
8  |  fim
9  fim
10 retorna falso
```

No algoritmo 9 as informações de graus são extraídas nas linhas 1 e 2. As listas retornadas pela função *calculaGraus* tem seus tamanhos comparados na linha 3 e as variedades de seus conteúdos analisados na linha 4. Uma vez que as listas não tenham sido discriminadas por esses critérios, elas são comparadas diretamente na linha 5. Para que dois grafos possam ser isomorfos essas listas devem ter o mesmo conteúdo.

A função *calculaGraus*(*GrafoG*, *distância*) soma os graus de entrada e soma os graus de saída para cada nó, para cada dist, até o n-dist. Ela retorna uma lista de vetores correspondente a cada um dos nós contendo os valores inteiros das somas concatenados. As tabelas 4 e 5 ilustram o funcionamento da função *calculaGraus* com uma instância de um dos grafos da gramática *Ad-hoc* com uma rede de 4 nós com visão de amplitude 2, representado na figura 7.

Executando para o exemplo a função *calculaGraus* em duas iterações, temos os

Figura 7: Exemplo de grafo da gramática *Ad-hoc*.



Fonte: da autora

Tabela 4: Vizinhos dos nós do grafo exemplo considerando as exceções.

Nós	vizinhos-fonte	vizinhos-sumidouro
n0	n1, n2	n1, n2
n1	n0, n3, n4	n0, n4
n2	n0, n4	n0, n4
n3	∅	n1
n4	n1, n2	n1, n2

Fonte: da autora

resultados da tabela 5. Na primeira iteração, a função retorna os graus de entrada e de saída para cada nó. Na segunda, retorna a soma dos graus de entrada de seus vizinhos-fonte e dos graus de saída de seus vizinhos-sumidouro (vizinhos 1-dist), respeitando as exceções da figura 6. A tabela 4 mostra os nós que foram considerados vizinhos na segunda iteração.

Cada linha da tabela 5 é representada como um vetor pela função *calculaGraus*, e esses vetores são comparados entre si para determinar o número de partições, e com os vetores calculados para os outros grafos, a fim de determinar se há isomorfismo ou não.

5. **Método das potências adaptado** Esse método foi apresentado por Baroni (2012). Ele consiste em atribuir um certificado inicial para cada nó (por exemplo, atribuir o valor 1 para todos os nós) e iterar sobre esses valores de forma que o novo certificado

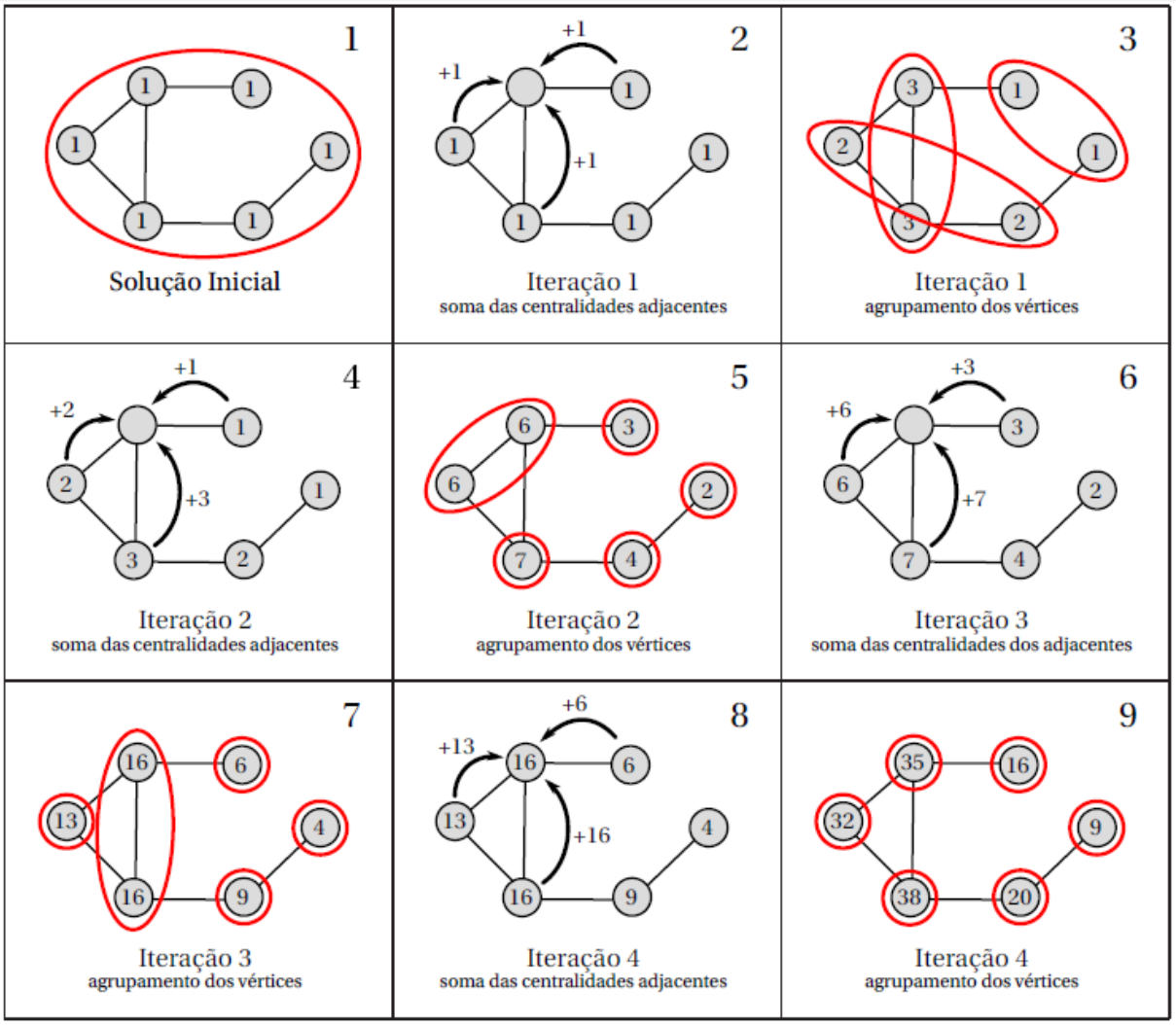
Tabela 5: Exemplo função *calculaGraus*

	1ª iteração		2ª iteração	
Nós	entrada	saída	entrada	saída
n0	2	2	5	4
n1	3	2	4	5
n2	2	2	4	4
n3	0	1	0	2
n4	2	2	5	4

Fonte: da autora

de nó seja a soma dos certificados de seus vizinhos. A ideia apresentada por Baroni (2012) é iterar esses certificados (ou centralidades) até que o número de partições (ou grupos) estabilize. A figura 8 extraída do trabalho original ilustra esse processo.

Figura 8: Método das potências adaptado em Baroni (2012).



Fonte: Baroni (2012)

Na figura, os números nos vértices indicam os valores dos certificados e as elipses

indicam as partições da iteração.

Aqui, como estamos tratando de grafos direcionados, iteramos os certificados separadamente para os vizinhos-fonte e os vizinhos-sumidouro. Os certificados de nós são inicializados com os valores dos graus de entrada e de saída. Os certificados de vizinhos-fonte/sumidouro são somados compondo o que chamaremos aqui de *potência de entrada* e *potência de saída*. O algoritmo 10 mostra o funcionamento do filtro de potências.

Algoritmo 10: FILTRO DE POTÊNCIAS.

Entrada: grafos $G1$, $G2$
Saída: $G1$ e $G2$ podem ser isomorfos?

```
1  pots1 = calculaPotencias(G1, n);
2  pots2 = calculaPotencias(G2, n);
3  se tamanho(pots1) == tamanho(pots2) então
4  |  ordena(pots1);
5  |  ordena(pots2);
6  |  se pots1 = pots2 então
7  |  |  retorna verdadeiro
8  |  fim
9  |  senão
10 |  |  retorna falso
11 |  fim
12 fim
13 retorna falso
```

O algoritmo 10 extrai as informações de potência nas linhas 1 e 2, no caso, um vetor de inteiros para cada grafo. Na linha 3 eles são comparados quanto ao tamanho e nas linhas 4 e 5 os conteúdos dos vetores são ordenados. Na linha 6 os dois vetores ordenados são comparados. Para que dois grafos possam ser isomorfos esses vetores devem ter o mesmo conteúdo.

A função *calculaPotencias* retorna um vetor de inteiros para cada grafo. Cada elemento do vetor corresponde ao *hashcode* de um objeto contendo o par ordenado de valores *(potência de entrada, potência de saída)* calculado em cada iteração. Portanto, para 4 iterações, por exemplo, teremos um vetor de tamanho 4. Se os vetores tem o mesmo tamanho (linha 3), eles são ordenados nas linhas 4 e 5 e comparados na linha 6. Os grafos são isomorfos se os vetores de potência ordenados são idênticos.

Para o grafo da figura 7, temos os valores das potências calculados pela função *calculaPotencias* na tabela 7 com base nas vizinhanças discriminadas na tabela 6.

Tabela 6: Vizinhos dos nós do grafo exemplo.

Nós	vizinhos-fonte	vizinhos-sumidouro
n0	n0, n1, n2	n0, n1, n2
n1	n0, n1, n3, n4	n0, n1, n4
n2	n0, n2, n4	n0, n2, n4
n3	n3	n1,n3
n4	n1, n2, n4	n1, n2, n4

Fonte: da autora

Tabela 7: Valores de potências de entrada e saída para o grafo exemplo.

	Inicialização		1ª iteração		2ª iteração	
Nós	entrada	saída	entrada	saída	entrada	saída
n0	3	3	10	9	33	27
n1	4	3	14	9	38	27
n2	3	3	9	9	29	27
n3	4	5	4	8	4	17
n4	3	3	10	9	34	27

Fonte: da autora

A principal diferença entre o filtro de graus e o de potências é que o de graus, ao longo das iterações, busca os vizinhos mais distantes (2-dist, 3-dist, etc), enquanto o filtro de potências sempre consulta seus vizinhos 1-dist. Além disso o filtro de graus compõe um vetor para cada nó ao longo das iterações enquanto o filtro de potências sempre tem somente um inteiro associado ao nó pela função *calculaPotencia*. Outro ponto de diferenciação, é que no filtro de potências não se aplica as exceções da figura 6.

6. **Sequência de rótulos dos vizinhos n-dist** Esse filtro é análogo ao fitro de graus do item 4, porém, ao invés de percorrer os vizinhos armazenando as somas dos seus graus, ele percorre as arestas que levam aos vizinhos e armazena seus rótulos. Os rótulos a uma certa distância dos nós iniciais são ordenados lexicograficamente em vetores correspondentes a cada nó. Se os dois grafos não possuem o mesmo conjuto de vetores de rótulos, eles não podem ser isomorfos.

   A sequência de rótulos para o grafo da figura 7 com o algoritmo executado em duas iterações pode ser visto na tabela 8. Cada entrada dessa tabela apresenta a sequência de rótulos das arestas adjacentes aos nós ao longo das iterações. As vizinhanças consideradas são as mesmas da tabela 4.

Tabela 8: Sequência de rótulos para o grafo exemplo.

	1ª iteração		2ª iteração	
Nós	entrada	saída	entrada	saída
n0	[link, link]	[link, link]	[active, link, link, link, link]	[link, link, link, link]
n1	[active, link, link]	[link, link]	[link, link, link, link]	[active, link, link, link, link]
n2	[link, link]	[link, link]	[link, link, link, link]	[link, link, link, link]
n3	∅	[active]	∅	[link, link]
n4	[link, link]	[link, link]	[active, link, link, link, link]	[link, link, link, link]

Fonte: da autora

4.3 Gramáticas de teste

A base de grafos para testes foi gerada com as mesmas gramáticas da seção 2.6, executando o GROOVE com a detecção de isomorfismo desligada e salvando cada um dos estados gerados em um arquivo texto. Guardar esses grafos nesse formato demanda bastante memória mesmo para gramáticas pequenas. Buscou-se escolher os tamanhos das gramáticas o maior possível de forma que ainda fosse viável armazenar o espaço de estados gerado por elas. Foram utilizadas então as seguintes gramáticas de teste:

- Append: fila de tamanho 5, com 3 invocações concorrentes;
- Ad-hoc: rede de 4 nós com visão de amplitude 2;
- Gossip: 4 garotas;
- Filósofos: 4 filósofos;
- Bad tic-tac-toe: tabuleiro de tamanho 3x3.

As características de número médio de arestas, número médio de nós, número total de grafos gerados pela exploração do espaço de estados e número de grafos não-isomorfos da linguagem são apresentadas na tabela 9 para cada uma das gramáticas. Podemos notar que a gramática *Gossip* é proporcionalmente a que tem mais grafos isomorfos, enquanto a *Bad tic-tac-toe* não contém nenhum isomorfo (graças à diferenciação dos rótulos referentes a cada uma das posições do jogo da velha). Essa diferença de proporção de não-isomorfos na base de grafos na gramática não chega a ter impacto na eficácia esperada para o filtro. São muito mais impactantes as características de estrutura das gramáticas, discutidas na seção 2.6.

Conhecidos os filtros e as instâncias de teste, temos na próxima seção seus resultados para os testes individuais.

Tabela 9: Características das Gramáticas de Teste.

Gramáticas	Média arestas	Média nós	Total de Grafos	Número de Grafos Não-Isomorfos
Append	22.4	25.4	4325	1720
Ad-hoc network	10.9	4.9	2448	117
Gossip priorities	2.8	8	1923	55
Phil	3.7	8	972	247
Tic_tac	8.2	19	6046	6046

Fonte: da autora

4.4 Resultados Computacionais dos Filtros Individuais

Os testes foram feitos de forma que cada conjunto de grafos relativo a uma gramática foi testado executando-se o filtro sobre as combinações de grafos dois a dois, ou seja, todos os grafos foram testados quanto a isomorfismo com relação a todos os outros grafos do conjunto de grafos da gramática. Isso significa que, para um conjunto de dados contendo n grafos, foram feitos $\frac{n\times(n-1)}{2}$ testes (combinação de n grafos 2 a 2).

Essa situação de comparação não é a mesma encontrada no GROOVE já que uma vez que um grafo é tido como isomorfo a algum dos outros já armazenados, ele é descartado, e não comparado com o restante do conjunto.

A principal justificativa de mantermos todas as comparações é que o desempenho dos algoritmos seria muito dependente da ordem em que fossem feitos os testes, ou mais especificamente, de quão grande é o número de grafos "descartados" logo no começo das comparações. Na aplicação real do GROOVE é importante notar que a exploração de um espaço de estados não é uma operação determinística, ou seja, a sequência em que as regras são aplicadas não é sempre a mesma.

No GROOVE não há como controlar a sequência em que os novos grafos são gerados, o que implica em diferentes resultados para diferentes execuções do simulador. Ou seja, não faria sentido testar uma única sequência de grafos, simulando a ordem em que eles foram gerados no GROOVE. Neste capítulo a intenção é isolar o problema do isomorfismo de grafos, e sendo assim, manter todas as comparações dois a dois ajuda a melhor avaliar os desempenhos dos algoritmos quanto a esse problema. Posto isso, não há como fazer uma comparação direta entre os resultados obtidos neste capítulo e os resultados da simulação dessas gramáticas no GROOVE.

Os filtros descritos na seção 4.2 foram executados para cada uma das gramáticas de teste em uma máquina Intel core i7, com 8GB de RAM, gerando os resultados das tabelas 10 e 11. Os filtros foram implementados em Java, compilados e executados com o com o JDK 1.7.0_79. Os desempenhos foram comparados em termos de eficácia e de tempo computacional em milissegundos. O termo eficácia (expressa em percentagem) se

Tabela 10: Resultados dos filtros individuais (parte 1)

Gramáticas	Resultados	NÓS	ARESTAS	RÓTS	GR 1	GR 2	GR 3	GR 4
Append	tempo	363	325	54116	271600	826953	1475782	9350650
	eficácia	91.19	97.49	97.90	99.92	99.94	99.95	99.97
Ad-hoc	tempo	108	113	7023	18555	55564	109727	167817
	eficácia	6.46	76.94	61.35	96.61	99.09	99.46	99.46
Gossip	tempo	66	69	3590	17764	51862	82574	102648
	eficácia	0	86.74	87.00	99.93	100	100	100
Filósofos	tempo	10	13	991	4217	12256	17143	20189
	eficácia	0	74.55	98.37	84.39	84.39	84.39	84.39
Tic_tac	tempo	630	594	87383	246548	500747	649258	802128
	eficácia	0	80.73	80.73	80.73	80.73	80.73	80.73

Fonte: da autora

Tabela 11: Resultados dos filtros individuais (parte 2)

Gramáticas	Resultados	SROT1	SROT2	SROT3	SROT4	POT1	POT2	POT3	POT4
Append	tempo	49433	1189288	1967869	2786872	343373	474513	620336	759188
	eficácia	99.93	100	100	100	99.94	99.95	99.97	99.98
Ad-hoc	tempo	39132	103429	190004	281929	24385	32922	44110	54445
	eficácia	99.25	99.97	100	100	99.07	99.44	99.44	99.44
Gossip	tempo	33117	76715	105769	133131	21420	30746	40467	50344
	eficácia	99.93	100	100	100	100	100	100	100
Filósofos	tempo	7920	16098	19373	22080	4658	7606	10251	12866
	eficácia	98.37	99.93	99.93	99.93	84.39	84.39	84.39	84.39
Tic_tac	tempo	399462	552541	682062	816702	269071	374513	483534	601721
	eficácia	100	100	100	100	80.73	80.73	80.73	80.73

Fonte: da autora

refere ao número de pares não-isomorfos que foi detectado, em relação ao total de pares não-isomorfos presentes no conjunto de dados. Os nomes dos filtros na primeira linha da tabela estão abreviados.

Observando as tabelas 10 e 11 , podemos concluir que:

- Mesmo sendo simples, os filtros podem atingir eficácias percentuais muito altas;
- Os três primeiros filtros (contagem de nós, arestas e rótulos) tem uma, duas, ou mais ordens de grandeza a menos no custo computacional em relação aos outros filtros;
- O filtro de contagem de arestas é o que tem melhor relação eficácia/tempo computacional entre os filtros de contagem;
- Os filtros de graus e de potências tem eficácias parecidas, sendo o de potências um pouco mais eficiente. Porém em relação aos custos computacionais eles se comportam diferentemente. O filtro de graus 1-dist tem custo menor em relação ao de potências

1-dist. Porém, o filtro de graus tem um custo crescendo a uma taxa aproximadamente exponencial conforme aumentamos a distância, enquanto o de potências tem taxa aproximadamente linear;

- O filtro mais eficiente entre eles é o de sequência de rótulos, mas ele é também o mais caro;
- Para os filtros que vão aumentando o alcance da análise (os que usam os vizinhos n-dist), podemos perceber que o 1-dist já resolve bem o problema. Utilizar distâncias maiores aumenta bastante o custo computacional mas não chega a compensar em termos de eficácia.

Com essas observações, concluimos que o filtro de sequência de rótulos 1-dist é o melhor filtro individual proposto, já que ele atinge quase 100% de eficácia para as variadas gramáticas de teste sem penalizar muito o tempo de execução. Essa constatação é a resposta da questão 2 levantada na seção 4.1.

É interessante notar que não necessariamente os não-isomorfismos detectados por cada um dos filtros tem interseção. Ou seja, possivelmente, se combinados, eles podem se complementar nessa detecção gerando filtros ainda melhores. Essa é a proposta da próxima seção.

4.5 Resultados Computacionais dos Filtros Compostos

Nesta seção deseja-se propor combinações dos filtros testados individualmente na seção anterior, a fim de melhorar seus desempenhos. Propomos aqui sete composições de filtros. Todos eles contém os três primeiros filtros dentre os listados na seção 4.2, já que esses filtros são muito mais rápidos de executar que os demais. Os filtros são executados na ordem em que são listados.

COMP 1: contagem de arestas + contagem de nós + contagem de rótulos;

COMP 2: COMP 1 + graus 1-dist;

COMP 3: COMP 1 + potências 1-dist;

COMP 4: COMP 1 + sequência de rótulos 1-dist;

COMP 5: COMP 1 + graus 1-dist + sequência de rótulos 1-dist;

COMP 6: COMP 1 + potências 1-dist + sequência de rótulos 1-dist;

COMP 7: COMP 1 + graus 1-dist + potências 1-dist + sequência de rótulos 1-dist;

Tabela 12: Resultados dos filtros compostos

Gramáticas	Resultados	COMP1	COMP2	COMP3	COMP4	COMP5	COMP6	COMP7
Append	tempo (ms)	2041	9200	10385	13865	9633	10920	10207
	eficácia (%)	97.97	99.92	99.94	99.93	99.93	99.95	99.95
Ad-hoc	tempo (ms)	1783	6214	7528	11207	8163	8151	8116
	eficácia (%)	76.94	96.61	99.07	99.26	99.26	99.87	99.87
Gossip	tempo (ms)	771	3266	3658	5670	4510	4865	5140
	eficácia (%)	86.74	99.93	100	99.93	99.93	100	100
Filósofos	tempo (ms)	360	945	997	1444	1674	1699	2055
	eficácia (%)	87.13	92.43	92.43	98.37	98.37	98.37	98.37
Tic_tac	tempo (ms)	20982	74048	79377	101695	145208	139769	187797
	eficácia (%)	80.73	80.73	80.73	100	100	100	100

Fonte: da autora

Os resultados dos testes desses filtros compostos estão apresentados na tabela 12. Na primeira da linha os nomes dos filtros estão abreviados. Observando a tabela podemos concluir que:

- O Composto 1 é aproximadamente cinco vezes mais rápido que os outros, e responde bem para o caso em que os grafos gerados pela gramática são mais variados (gramática *Append*).
- Os filtros compostos do 4 em diante detectam quase 100% dos não-isomorfismos em todas as gramáticas. A gramática *Tic_tac* é decisiva na escolha do melhor filtro, pois, diferenciar seus não-isomorfismos está muito ligado a mapear seus rótulos. Os filtros do 4 em diante contem o filtro sequência de rótulos, o que faz eles atingirem 100% de diferenciação para essa gramática.
- Dos últimos quatro filtros compostos, os mais rápidos são o 5 e o 6. A diferença entre eles é que o 5 usa o filtro de graus 1-dist e o 6 usa o de potências 1-dist. Já vimos nos testes individuais que, para o caso 1-dist, o critério de graus é em média mais rápido que o de potências.
- Importante notar que os tempos de execução para todos os filtros compostos são bastante menores que os dos filtros individuais apresentados nas tabelas 10 e 11 (exceto os filtros de contagem).

Com base nessas observações, elegemos o filtro composto 5 como a melhor combinação para resolver o problema de não-isomorfismo entre todos os filtros propostos. Essa constatação responde a pergunta número 3 proposta na seção 4.1.

Uma questão que surge naturalmente ao testarmos os filtros compostos é quanto os filtros que os compõe estão sendo requisitados, e quanto tempo de processamento é associado a cada um deles. Essa discriminação por componente da eficácia e do tempo

gastos em cada filtro composto é apresentada na tabela 13. Nela os campos com o símbolo "–" indicam que aquele componente não foi usado no filtro composto. Os acertos estão expressos em porcentagem dos não-isomorfismos detectados e a coluna "tempo" indica o tempo de execução em milissegundos.

Tabela 13: Resultados dos filtros compostos: número de acertos (%) e tempo computacional

		COMPOSED 1		**COMPOSED 2**		**COMPOSED 3**		**COMPOSED 4**		**COMPOSED 5**		**COMPOSED 6**		**COMPOSED 7**	
		acertos	tempo	acertos	tempo	acertos	tempo	acertos	tempo	acertos	tempo	acertos	tempo	acertos	tempo
Append	Edge	97,49	141	97,49	316	97,49	203	97,49	388	97,49	363	97,49	270	97,49	284
	Node	0,48	30	0,48	32	0,48	31	0,48	47	0,48	15	0,48	46	0,48	32
	Label	0	1917	0	1965	0	1714	0	1552	0	2052	0	1891	0	1806
	Deg1	–	–	**1,95**	**8234**	–	–	–	–	1,95	5733	–	–	1,95	5540
	Pot1	–	–	–	–	**1,97**	**7665**	–	–	–	–	1,97	7198	0,02	373
	LT1	–	–	–	–	–	–	1,96	11277	0,02	559	0,01	486	0,01	724
	Total	**97,97**	**2324**	**99,92**	**10905**	**99,94**	**9876**	**99,93**	**13529**	**99,93**	**9051**	**100**	**10312**	**99,95**	**9101**
Ad-hoc network	Edge	76,94	126	76,94	170	76,94	124	76,94	281	76,94	124	76,94	157	76,94	158
	Node	0	15	0	47	0	15	0	0	0	32	0	30	0	79
	Label	0	1608	0	2126	0	1892	0	1995	0	1959	0	1983	0	1995
	Deg1	–	–	19,68	3428	–	–	–	–	19,68	3777	–	–	19,68	3906
	Pot1	–	–	–	–	22,31	4868	–	–	–	–	22,31	4894	2,45	888
	LT1	–	–	–	–	–	–	22,32	7558	2,64	1611	0,81	660	0,81	684
	Total	**76,94**	**1827**	**96,61**	**5928**	**99,07**	**6961**	**99,25**	**9958**	**99,25**	**7707**	**99,87**	**7898**	**99,87**	**7835**
Gossip	Edge	86,74	46	86,74	79	86,74	62	86,74	79	86,74	62	86,74	94	86,74	123
	Node	0	0	0	0	0	0	0	0	0	16	0	0	0	15
	Label	0,35	701	0,35	689	0,35	701	0,35	777	0,35	699	0,35	689	0,35	706
	Deg1	–	–	19,68	2350	–	–	–	–	19,68	2386	–	–	19,68	2295
	Pot1	–	–	–	–	12,91	2951	–	–	–	–	12,91	2602	0,07	561
	LT1	–	–	–	–	–	–	19,68	4559	0	1238	0	1309	0	1307
	Total	**87,09**	**794**	**99,93**	**3180**	**100**	**3730**	**99,93**	**5478**	**99,93**	**4465**	**100**	**4742**	**100**	**5103**
Filósofos	Edge	**74,55**	0	74,55	16	74,55	32	74,55	0	74,55	0	74,55	17	74,55	0
	Node	0	16	0	0	0	0	0	0	0	16	0	0	0	0
	Label	12,58	359	12,58	361	12,58	329	12,58	390	12,58	330	12,58	454	12,58	456
	Deg1	–	–	**5,30**	**559**	–	–	–	–	**5,30**	**598**	–	–	5,30	544
	Pot1	–	–	–	–	**5,30**	**715**	–	–	–	–	**5,30**	**494**	0	343
	LT1	–	–	–	–	–	–	11,23	1171	5,94	698	5,94	735	5,94	716
	Total	**87,13**	**390**	**92,43**	**983**	**92,43**	**1076**	**98,37**	**1592**	**98,37**	**1688**	**98,37**	**1716**	**98,37**	**2059**
Tic_tac	Edge	80,73	734	80,73	1787	80,73	1836	80,73	1976	80,73	1970	80,73	1931	80,73	2016
	Node	0	203	0	297	0	280	0	343	0	265	0	251	0	422
	Label	0	17579	0	19522	0	19741	0	19137	0	19835	0	21131	0	20388
	Deg1	–	–	0	41701	–	–	–	–	0	44656	–	–	0	45722
	Pot1	–	–	–	–	0	46873	–	–	–	–	0	49068	0	43296
	LT1	–	–	–	–	–	–	19,27	68528	19,27	62798	19,27	59016	19,27	58663
	Total	**80,73**	**19328**	**80,73**	**63917**	**80,73**	**69257**	**100**	**90961**	**100**	**130241**	**100**	**132176**	**100**	**171209**

Fonte: da autora

Observando a tabela 13 podemos notar que:

- A contagem de arestas é realmente um bom primeiro filtro para os filtros compostos. No mínimo, esse critério sozinho já filtra perto de 75% dos não-isomorfismos (pior caso é a gramática *Filósofos*), com tempos de execução bem baixos.
- A contagem de rótulos tem um comportamento bimodal: ora detecta um bom percentual de não-isomorfismos (*Gossip* e *Filósofos*), ora não ajuda em absolutamente nada. Seu custo computacional não é tão baixo quanto as outras contagens.
- O filtro de potências tende a ter maior eficácia a um custo computacional mais elevado. Comparando os Composto 2 e 3 para a gramática *Append* e os Composto 5 e 6 para a gramática *Filósofos*, o filtro de potências é mais eficiente que o de graus e gasta menos tempo computacional. Já nos Composto 2 e 3 para a gramática *Filósofos* o filtro de potências é mais custoso. Investigar qual é o melhor entre o filtro de graus e o de potência é um trabalho futuro.
- O filtro de sequências de rótulos tem seu custo bastante reduzido ao ficar por último, e é bem eficiente em encontrar os não-isomorfismos que os outros critérios não conseguiram identificar (vide filtro Composto 7).

Na próxima seção comparamos os resultados dos filtros compostos com os algoritmos de isomorfismo do GROOVE e os IVLs adaptados quando esses algoritmos são usados como filtros.

4.6 Uso dos cálculos de certificados como filtros

Vimos que os filtros são bem eficientes, e que podem evitar a maior parte dos cálculos de certificados que seriam feitos para determinar isomorfismos. Mas o que temos no GROOVE é que todos os grafos gerados tem seus certificados calculados, já que o certificado de grafo é usado como *hash* na estrutura de armazenamento dos grafos. Nesta seção queremos checar como os filtros e os certificados se comparam caso os certificados fossem utilizados exclusivamente para a detecção de isomorfismo. Os resultados dos certificados do GROOVE e dos certificados implementados no capítulo 3 são mostrados na tabela 14. A sigla *PR* representa o algoritmo *Partition Refiner* que é o adotado atualmente no GROOVE [Rensink (2010)]. *QS* indica o uso de quebra de simetria, conforme explicado na seção 3.2.3. Assim, *cQS* significa "com quebra de simetria" e *sQS* significa "sem quebra de simetria". Todos os casos tiveram eficácia de 100% na detecção de não-isomorfismos, já que esses algoritmos tem a busca direta como sua última etapa. Dessa forma, as linhas contendo as eficácias foram omitidas na tabela 14.

Tabela 14: Resultados do uso dos certificados como filtros

Gramáticas	PR sQS	PR cQS	IVL1 sQS	IVL1 cQS	IVL3 sQS	IVL3 cQS
Append	139910	169770	149516	200828	149102	182993
Ad-hoc network	11984	12996	12559	15037	12311	13979
Gossip priorities	10469	28940	11142	32855	10766	29841
Filósofos	2029	2124	2076	2902	2046	2560
Tic_tac_toe	152807	126167	151034	135487	108187	111540

Fonte: da autora

Tabela 15: Relação entre os tempos computacionais do PR cQS e do COMP5

Gramáticas	tempo COMP5 / tempo PR cQS
Append	5,7%
Ad-hoc network	62,8%
Gossip priorities	15,6%
Filósofos	78,7%
Tic_tac_toe	117,4%

Fonte: da autora

Podemos notar, comparando as tabelas 12 e 14 que os filtros têm muitas vezes tempos de execução mais baixos que qualquer um dos certificados e em algumas situações atingem até 100% de eficácia.

Na tabela 14 podemos notar é que os algoritmos de IVL adaptados, quando executados para essas gramáticas de teste, são em média piores que os algoritmos originais do GROOVE. Isso pode se dever ao fato de que aqui estamos usando grafos ainda menores que os gerados pelas gramáticas de teste do capítulo 3, onde o IVL se mostrou aproximadamente equivalente em termos de performance. Os resultados de Baroni (2012) mostram que o IVL é especialmente competitivo para grafos grandes, da ordem de centenas de nós. Aqui estamos testando-o com grafos bem menores, o que poderia explicar essa piora na performance.

A tabela 15 relaciona diretamente o tempo de execução do filtro composto que foi eleito o melhor (COMP5) com o algoritmo PR cQS do GROOVE.

Na tabela 15 podemos notar que o filtro COMP5 tem performaces melhores que o PR cQS para todas as gramáticas exceto a *Tic_tac*. Isso ocorre justamente devido às particularidades dessa gramática, de possuir grafos com várias estruturas idênticas, diferenciáveis apenas pela disposição dos rótulos. Essas características fazem com que apenas dois filtros atuem na detecção de não-isomorfismos para a *Tic_tac*: contagem de arestas e sequência de rótulos (tabela 13). Assim, os outros filtros incorporados ao COMP5 não são úteis para essa gramática em particular, onerando o tempo computacional do filtro COMP5 além do tempo computacional da execução do cálculo de certificados. Mas

podemos ver que, por exemplo, o filtro COMP4 é mais rápido que o PR cQS para todas as gramáticas, incluindo a *Tic_tac*, o que valida a abordagem por filtros como sendo uma boa estratégia quando comparada à implementação atual do GROOVE.

Essas observações respondem a quarta e última pergunta proposta na seção 4.1. A próxima seção resume as conclusões dos resultados obtidos neste capítulo.

4.7 Conclusões

Dos experimentos realizados neste capítulo podemos concluir que os filtros podem ser um artefato interessante na detecção de isomorfismos, mesmo que seja uma comparação entre um grafo e um conjunto de grafos. Há critérios mais simples que os certificados capazes de eliminar a possibilidade de os grafos serem isomorfos com um tempo computacional muito mais reduzido. Há uma particularidade nesses casos de teste, já que as instâncias de grafos testadas aqui são muito pequenas, com poucos automorfismos, e em geral os certificados de nós estão preparados justamente para casos mais difíceis, onde os grafos sejam maiores e com muitos automorfismos. Por outro lado, sabemos que para as aplicações mais comuns do GROOVE, dificilmente são executadas gramáticas que tenham grafos com a ordem de centenas de nós, já que geralmente os modelos de sistemas reais têm grafos com no máximo algumas dezenas de nós. Ou seja, aqui nossos testes são com grafos com menos de uma dezena de nós, e os certificados estão preparados para grafos com a ordem de centenas de nós. Uma proposta de trabalho futuro é investigar o que acontece se pudermos testar no GROOVE as gramáticas que geram grafos com dezenas de nós, usando um sistema de armazenamento de grafos diferente do atual.

É importante notar que os resultados e as conclusões apresentados são fortemente dependentes das gramáticas de teste. Aqui buscamos diversificar os tipos de gramáticas testadas, na tentativa de sustentar a argumentação de que seria interessante o uso de filtros dentro do GROOVE, caso tivéssemos outra opção que não o uso do *hash* para armazenar os grafos. Armazenar as gramáticas de outra forma no GROOVE pode requerer um uso ainda maior de memória. Usar mais memória para aumentar o desempenho computacional, ou penalizar desempenho para usar menos memória, é uma conhecida dicotomia dos profissionais que trabalham desenvolvendo ferramentas computacionais. Outro trabalho futuro é determinar o ponto ótimo, experimentar essas soluções e, sob o critério de usabilidade da ferramenta, determinar a melhor implementação que atenda os objetivos da aplicação.

5 Conclusão

A proposta deste trabalho é mostrar os resultados experimentais da adaptação de um novo algoritmo de detecção de isomorfismos de grafos, o IVL, na etapa de verificação de isomorfismos de uma das principais ferramentas disponíveis para verificação de modelos baseados em sistemas de transformação de grafos, o GROOVE. Mais além, foram realizados experimentos com o uso de filtros, implementações capazes de encontrar não-isomorfismos de forma mais rápida que o processo de calcular os certificados iterativamente para cada um dos grafos.

Ao longo do trabalho são apresentados experimentos que nos permitem responder as questões-chave da pesquisa, propostas no Capítulo 1.

1. É possível adaptar o IVL para o GROOVE?

   Sim. No capítulo 3 foram detalhadas as semelhanças e diferenças entre os dois algoritmos, e foi mostrado quais eram as adaptações que eram necessárias serem feitas no IVL para utilizarmos ele no GROOVE (seção 3.4). Os pontos de adaptação levantados são viáveis de serem implementados.

2. De que formas isso pode ser feito?

   Há diversas maneiras de adaptar o IVL ao GROOVE, pois para cada ponto de adaptação temos diversas estratégias possíveis. Na seção 3.5 apresentamos quatro possibilidades de implementações distintas.

3. Qual a performance do IVL em relação ao algoritmo original do GROOVE?

   Na seção 3.6 são mostrados os resultados computacionais dos IVLs adaptados quando comparados ao GROOVE. Esses resultados mostram que os dois algoritmos tem respostas bastante parecidas para as gramáticas de teste. Isso deve ao fato de que os dois algoritmos tem estruturas bastante parecidas, juntamente com a dificuldade de implementar grandes variações nos processos, já que a arquitetura do GROOVE é bastante engessada, customizada para a solução adotada na ferramenta atualmente.

4. Considerando os grafos gerados pelas gramáticas de teste, quais atributos de grafos ajudam a diferenciá-los quanto a isomorfismo de maneira eficiente?

   No capítulo 4 são propostos diversos atributos de grafos a serem comparados com o objetivo de detectar rapidamente se dois grafos são não-isomorfos. A descrição dessas propostas (chamadas no capítulo de filtros) estão na seção 4.2. Elas foram testadas fora do GROOVE para que o problema de isomorfismo fosse isolado dos outros processos da ferramenta. Os testes de cada filtro individualmente estão presentes

na seção 4.4. Nela podemos notar que há filtros muito eficazes, checando a detectar 100% dos não-isomorfismos. Os filtros individuais foram combinados para formarem filtros compostos, e esses últimos foram testados nas mesmas condições dos filtros individuais, gerando resultados com eficácias e performances ainda melhores (seção 4.5). Os filtros mostraram que podem ser uma boa opção de implementação a fim de determinar não-isomorfismos de forma eficiente e evitar o esforço computacional de calcular os certificados para todos os grafos.

5. Como os IVLs adaptados se comparam ao GROOVE na detecção de isomorfismos entre dois grafos (fora do contexto do GROOVE)?

   Os certificados do GROOVE, quando usados exclusivamente para detectar não-isomorfismos, são mais eficientes que as adaptações do IVL propostas no capítulo 3. Porém, o GROOVE é mais lento na detecção de não-isomorfismos que os filtros compostos em muitas situações, chegando a relação de 5% entre o tempo computacional de um filtro com alta eficácia e o GROOVE.

Uma vez respondidas as questões-chave de pesquisa, podemos enumerar os trabalhos futuros.

5.1 Trabalhos Futuros

Com base nos resultados, propõe-se os seguintes trabalhos futuros:

- Estudar outros algoritmos de isomorfismo de grafos que possam ser promissores ao serem adaptados para o GROOVE.
- Implementar uma estrutura de dados que armazene de maneira eficiente os vizinhos de cada um dos nós de um grafo. Com isso, pode-se fazer a busca da função isomorfismo entre dois grafos estabelecendo o morfismo pelos nós, e não pelas arestas. Dessa forma, o certificado de arestas pode ser suprimido, o que diminui o tempo computacional gasto com o cálculo de certificados em relação a calcularmos somente os certificados de nós.
- Implementar no GROOVE outra estrutura de dados para o armazenamento de grafos, tornando viável o uso dos filtros apresentados no capítulo 4.
- Aprofundar os estudos sobre os filtros, especialmente para determinar qual entre os filtros de graus e de potências é mais eficaz e tem melhor performance.

Referências

BABAI, L. Graph isomorphism in quasipolynomial time. *arXiv preprint arXiv:1512.03547*, 2015. Citado 2 vezes nas páginas 13 e 21.

BAIER, C.; KATOEN, J.-P. et al. *Principles of model checking*. [S.l.]: MIT press Cambridge, 2008. Citado na página 11.

BARONI, M. D. V. *Um Estudo da Eficiência da Autocentralidade no Problema de Isomorfismo de Grafos*. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, Vitória - ES, 1 2012. Citado 14 vezes nas páginas 4, 5, 6, 11, 14, 22, 25, 26, 30, 33, 38, 52, 53 e 64.

BARONI, M. D. V. et al. *An Iterative Label-Based Algorithm for Graph Isomorphism*. [S.l.], 2013. 30 p. Citado na página 25.

CONTE, D. et al. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, World Scientific, v. 18, n. 03, p. 265–298, 2004. Citado na página 13.

DARGA, P. T.; SAKALLAH, K. A.; MARKOV, I. L. Faster symmetry discovery using sparsity of symmetries. In: ACM. *Proceedings of the 45th annual Design Automation Conference*. [S.l.], 2008. p. 149–154. Citado 2 vezes nas páginas 11 e 23.

FORTIN, S. *The graph isomorphism problem*. [S.l.], 1996. Citado 2 vezes nas páginas 11 e 21.

GHAMARIAN, A. H. et al. Modelling and analysis using groove. *International journal on software tools for technology transfer*, Springer, v. 14, n. 1, p. 15–40, 2012. Citado na página 24.

GHAMARIAN, A. H.; ZAMBON, E. Verifying the leader election algorithm in groove. 2009. Citado na página 13.

HURKENS, C. A. Spreading gossip efficiently. *Nieuw Archief voor Wiskunde*, Citeseer, v. 1, p. 208–210, 2000. Citado na página 27.

JUNTTILA, T.; KASKI, P. Conflict propagation and component recursion for canonical labeling. In: *Theory and Practice of Algorithms in (Computer) Systems*. [S.l.]: Springer, 2011. p. 151–162. Citado na página 23.

LÓPEZ-PRESA, J. L.; ANTA, A. F.; CHIROQUE, L. N. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *arXiv preprint arXiv:1108.1060*, 2011. Citado na página 23.

MCKAY, B. D. et al. *Practical graph isomorphism*. [S.l.]: Department of Computer Science, Vanderbilt University Tennessee, US, 1981. Citado 4 vezes nas páginas 11, 13, 23 e 24.

MCKAY, B. D.; PIPERNO, A. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, Elsevier, v. 60, p. 94–112, 2014. Citado 4 vezes nas páginas 11, 21, 22 e 23.

PAIGE, R.; TARJAN, R. E. Three partition refinement algorithms. *SIAM Journal on Computing*, SIAM, v. 16, n. 6, p. 973–989, 1987. Citado na página 24.

PEDARSANI, P.; GROSSGLAUSER, M. On the privacy of anonymized networks. In: ACM. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.], 2011. p. 1235–1243. Citado na página 13.

RENSINK, A. The groove simulator: A tool for state space generation. In: *Applications of Graph Transformations with Industrial Relevance*. [S.l.]: Springer, 2004. p. 479–485. Citado 2 vezes nas páginas 11 e 24.

RENSINK, A. Isomorphism checking in groove. *Electronic Communications of the EASST*, v. 1, 2007. Citado 2 vezes nas páginas 24 e 27.

RENSINK, A. Isomorphism checking for symmetry reduction. Centre for Telematics and Information Technology University of Twente, 2010. Citado 6 vezes nas páginas 23, 24, 27, 30, 36 e 63.

ROZENBERG, G. *Handbook of Graph Grammars and Comp.* [S.l.]: World scientific, 1997. Citado 2 vezes nas páginas 12 e 18.

SANTOS, P. L. F. dos; RANGEL, M. C.; BOERES, M. C. S. Teoria espectral de grafos aplicada ao problema de isomorfismo de grafos. *Proceedings of XLII Simpósio Brasileiro de Pesquisa Operacional (SBPO 2010)*, p. 1–12, 2010. Citado na página 14.

ZAMBON, E. *Abstract Graph Transformation Theory and Practice*. Tese (Doutorado) — Centre for Telematics and Information Technology, University of Twente, 2013. Citado 3 vezes nas páginas 17, 18 e 25.