UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE CIÊNCIA E TECNOLOGIA - ICT
CIÊNCIA DA COMPUTAÇÃO

PAULO SÉRGIO ALVES CARREIRA

ALGORITMOS CERTIFICADORES PARA PROBLEMAS CLÁSSICOS EM GRAFOS

Rio das Ostras
2018

PAULO SÉRGIO ALVES CARREIRA

ALGORITMOS CERTIFICADORES PARA PROBLEMAS CLÁSSICOS EM GRAFOS

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Ciência da Computação como requisito parcial para conclusão do curso.

Orientador:
Prof. Dr. Danilo Artigas da Rocha

Rio das Ostras
2018

Ficha catalográfica automática - SDC/BRO

C314a Carreira, Paulo Sérgio Alves
Algoritmos Certificadores Para Problemas Clássicos em Grafos / Paulo Sérgio Alves Carreira ; Danilo Artigas da Rocha, orientador. Rio das Ostras, 2018.
57 f. : il.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação)-Universidade Federal Fluminense, Instituto de Ciência e Tecnologia, Rio das Ostras, 2018.

1. Algoritmos. 2. Algoritmo em Grafos. 3. Convexidade. 4. Grafos. 5. Produção intelectual. I. Título II. Artigas da Rocha,Danilo, orientador. III. Universidade Federal Fluminense. Instituto de Ciência e Tecnologia. Departamento de Computação.

CDD -

Bibliotecária responsável: Maria Margareth Vieira Pacheco Rodrigues - CRB7/5874

PAULO SÉRGIO ALVES CARREIRA

ALGORITMOS CERTIFICADORES PARA PROBLEMAS CLÁSSICOS EM GRAFOS

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Ciência da Computação, como requisito parcial para conclusão do curso.

Aprovada em 12 de Julho de 2018.

BANCA EXAMINADORA

_____

Prof. Dr. Danilo Artigas da Rocha

_____

Prof. Dr. Marcos Ribeiro Quinet de Andrade

_____

Prof. Dr. André Renato Villela da Silva

Rio das Ostras
2018

A Paulo Alves, meu avô

AGRADECIMENTOS

Aos meus avós, Paulo Alves e Maria Aparecida, por todo investimento na minha educação e no meu caráter ao longo de toda minha vida.

Aos meus tios, Ligia e Edson, pelo apoio ao longo dessa jornada que foi minha graduação.

Ao Danilo, por toda paciência e tudo que me ensinou ao longo da minha Iniciação Científica e da produção deste trabalho.

A todos os professores que fizeram parte da minha formação e por tudo que me ensinaram.

RESUMO

Um Algoritmo Certificador é um algoritmo que produz, para cada saída, um certificado para esta. O usuário, de posse do certificado e da resposta do algoritmo, pode verificar se a resposta está correta. A finalidade do desenvolvimento de algoritmos certificadores é oferecer ao usuário a possibilidade de conferir a saída e, consequentemente, detectar se há ou não algum *bug* de implementação no programa. Dessa forma, ele não precisará confiar “cegamente” que o programa está correto, o que torna a saída mais confiável para uso posterior. A necessidade de algoritmos com saídas mais confiáveis e testes de software mais eficazes motiva o estudo mais detalhado desse tipo de algoritmo, que ainda não foi intensamente estudado na Computação. Neste trabalho, estudamos e desenvolvemos algoritmos certificadores para problemas clássicos de Convexidade em Grafos. Certificar tais algoritmos clássicos estabelece um grande passo para o estudo de certificados mais convincentes e de novos algoritmos certificadores.

Palavra-chave: Algoritmos certificadores; caminhos mínimos; caminhos induzidos; convexidade geodésica; convexidade monofônica.

ABSTRACT

A Certifying Algorithm is an algorithm that produces, for each output, a certificate for it. The user, in ownership of the certificate and the response of the algorithm, can verify if the answer is correct. The purpose of the development of certifying algorithms is to offer the user the possibility to check the output and, consequently, to detect if there is some implementation bug in the program, or not. Therefore, the user does not have to trust "blindly" that the program is correct which makes the output more reliable for later use. The need of algorithms with more reliable outputs and more effective software testing motivates further studies of this type of algorithm that has not been deeply studied in Computing yet. The approach taken in this work is around classical problems such as those of minimum paths in a simple graph. Certifying such classical algorithms sets a major step towards the study of more convincing certificates and new certifying algorithms.

Keywords: Certifying Algorithms, Minimal Paths, Induced Paths, Geodetic, Monophonic

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 **INTRODUÇÃO**

Um *programa* na sua visão tradicional nada mais é do que um conjunto de instruções que descrevem e executam uma determinada tarefa. O programa recebe uma entrada e produz uma saída. Tal programa é executado em algum dispositivo eletrônico como um computador ou celular e foi desenvolvido por um programador.

Deparar-se com programas com erros é algo comum e frustrante. Todo usuário de algum tipo de tecnologia já deve ter se deparado com alguma função de um programa com alguma falha de implementação.

O fato de o ser humano desenvolver *softwares* abre margem para erros e falhas. A própria teoria de Engenharia de Software assume que não existem *softwares* totalmente livre de falhas ou erros. A grande questão em torno deste fato é: Como confiar então na saída apresentada por um software?

Refletir sobre esta pergunta nos leva a duas ideias para sanar o problema: testar exaustivamente o *software* de maneira a detectar falhas e corrigi-las ou fazer com que o próprio programa apresente uma “prova” de que está correto.

A ideia de um programa que apresenta “provas” descreve o que são em si os algoritmos certificadores. Por mais que seja uma tarefa difícil criar tal “prova”, conhecida como certificado, esta traz diversos benefícios quanto a confiabilidade do usuário com o *software*.

Dentre os desafios de se criar um certificado, estão a simplicidade do certificado e facilidade em verificar (verificabilidade) que tal certificado realmente comprova que a saída do programa é confiável. Deve ser claro para o usuário que o certificado gerado comprova que a saída é correta. É desejável comprovar que o certificado é válido por meio de demonstrações matemáticas, para garantir a sua eficácia. Em alguns casos, pode ser desejável até mesmo criar um *software* a parte que, com o certificado, faz a verificação automaticamente, facilitando ainda mais o processo por parte de quem usa o programa. Parte da teoria por trás desse tipo de algoritmo, além de exemplos e novos algoritmos certificadores criados, será apresentada neste trabalho.

Alguns resultados em algoritmos certificadores podem ser encontrados em (KRATSCH, Dieter et al., 2006) e (KAPLAN, Haim; NUSSBAUM, Yahav., 2009).

O foco do nosso trabalho é criar algoritmos certificadores para problemas clássicos em grafos, em particular, os problemas de convexidade geodésica e monofônica e suas variações.

Dado um grafo G, uma *convexidade em grafo* é um par $(G, \mathcal{C})$ tal que: (i) $\mathcal{C}$ é uma família de subconjuntos de $V(G)$ fechada para interseção; (ii) $V(G)$ e $\emptyset$ pertencem a $\mathcal{C}$. Os conjuntos de $\mathcal{C}$ são denominados *convexos*.

Uma convexidade em grafo é denominada uma *convexidade de intervalo* se admite uma função de intervalo $I: 2^{V(G)} \rightarrow 2^{V(G)}$, tal que, $S \subseteq I[S]$. Neste trabalho estudamos duas convexidades intituladas *geodésica* e *monofônica*, a diferença entre uma e outra é a escolha da função de intervalo.

A convexidade geodésica se resume à convexidade baseada em caminhos mínimos, isto é, para um conjunto $X \subseteq V(G)$, o intervalo fechado $I[X]$ (ou *fecho geodésico*) é o conjunto de todos os vértices que se encontram em algum caminho mínimo entre algum par de vértices de X. Um conjunto X é denominado convexo se $I[X] = X$. Por sua vez, a convexidade monofônica é a convexidade baseada em caminhos induzidos, ou seja, para um conjunto $X \subseteq V(G)$, o intervalo fechado $I[X]$ (ou *fecho monofônico*) é o conjunto de todos os vértices que se encontram em algum caminho induzido entre algum par de vértices de X. Dentre os problemas clássicos em convexidade geodésica temos o de:

a) verificar se um conjunto X é *convexo*, ou seja, se o intervalo fechado de X contém apenas vértices de X.

b) verificar se um conjunto é *geodésico*, isto é, se o intervalo fechado de X, contém todos os vértices do grafo.

c) verificar se um conjunto é *envoltório*, ou seja, se sob recursivas e finitas iterações de cálculos de intervalos fechados sobre o intervalo fechado de X, todos os vértices do grafo serão incluídos.

Dentre os problemas clássicos em convexidade monofônica, que são bastante semelhantes aos de convexidade geodésica, temos:

a) verificar se um conjunto *X* é *m-convexo*, ou seja, se o fecho monofônico de *X* contém apenas vértices de *X*.

b) verificar se um conjunto é *monofônico*, isto é, se o fecho monofônico de *X* contém todos os vértices do grafo.

c) verificar se um conjunto é *m-envoltório*, ou seja, se sob recursivas e finitas iterações de cálculos de fechos monofônicos sobre o fecho monofônico de *X*, todos os vértices do grafo serão incluídos.

Nenhum dos seis problemas citados apresentam algoritmos certificadores para a sua resolução. Apresentaremos neste trabalho algoritmos certificadores para cinco destes problemas. Resultados em convexidade geodésica podem ser encontrados em (PELAYO, Ignacio M., 2013) e sobre convexidade monofônica em (DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L., 2010).

Quanto a organização deste trabalho esta se encontra da seguinte forma: no Capítulo 2, apresentamos todos os conceitos de grafos necessários para o entendimento dos algoritmos certificadores criados nesse trabalho e as teorias por trás destes. No Capítulo 3, apresentamos o ferramental formalizado por McConnell (MCCONNELL, R. M., 2011), um dos grandes pesquisadores neste tema, sobre algoritmos certificadores. Nos Capítulos 4 e 5 apresentamos as teorias de convexidade geodésica e monofônica, e os nossos algoritmos certificadores, criados para problemas clássicos nestes dois assuntos.

2 **NOÇÕES SOBRE GRAFOS**

As definições apresentadas neste capítulo foram retiradas de (BONDY, John Adrian; MURTY, Uppaluri Siva Ramachandra, 2008).

Um *grafo* é um par ordenado $G=(V,E)$, onde $V = V(G)$ é um conjunto finito e não vazio de objetos conhecidos como *vértices* e $E = E(G)$ é um conjunto de pares de vértices distintos, isto é, subconjuntos compostos de dois elementos de V chamados *arestas*. Se $e = (u, v)$ pertence a $E(G)$, então podemos dizer que: u e v são *adjacentes*, ou e une u e v, ou e é *incidente* a u e a v ou ainda que u e v são *vizinhos*. A *vizinhança de um vértice*, denotada por $N(v)$, é o conjunto de vizinhos de v no grafo G, isto é, $N(v) = \{ u \in V(G) : (u, v) \in E(G)\}$.

Um *passeio* é uma sequência de vértices tal que se v e w são consecutivos na sequência, então (v,w) é uma aresta do grafo G. Uma *trilha* é um passeio que não repete arestas. Um *caminho* é um passeio que não apresenta vértices repetidos. Um *caminho* é *fechado* se sua origem é igual ao seu término, ou seja, o caminho começa e termina no mesmo vértice. Um *ciclo* é um caminho fechado de comprimento maior que 1. Um *caminho* C de um vértice u a um vértice v é *mínimo* se não existe outro caminho de u a v que tenha menos arestas que C. A *distância* de um vértice u a um vértice v é o número de arestas de um caminho mínimo de u a v e é denotado por $d(u,v)$.

Um *subgrafo* H de um grafo G é um grafo tal que $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, e neste caso podemos escrever que $H \subseteq G$. H é dito um *subgrafo induzido* de um grafo G se é um subgrafo tal que todo par de vértices u, v de H, se eles são adjacentes em G, então eles também são adjacentes em H. Se H é um subgrafo induzido de G e $S = V(H)$, podemos dizer que H é o subgrafo induzido por S em G, podendo ser denotado por $H = G[S]$. Dado um conjunto $S \subset V(G)$, o subgrafo $G - S$, obtido ao se remover todos os vértices de S de G, é o subgrafo induzido $G[V(G) - S]$. Em caso particular, dado qualquer $v \in V(G)$, o subgrafo $G – v$, obtido removendo-se v de G, é o subgrafo induzido $G[V(G) – v]$.

Um *grafo completo* K_n é aquele que quaisquer de seus dois vértices são adjacentes. Uma *clique* é um conjunto de vértices dois a dois adjacentes. Em outras

palavras, um conjunto $C \subseteq V(G)$ de vértices é uma clique tal que para todo par de vértices distintos *v, w* em *C*, há uma aresta entre *v* e *w*.

Um par de conjuntos de vértices *(X, Y)* é dita uma *partição* de *V(G)* se $X \cup Y = V$ e $X \cap Y = \emptyset$. Um grafo *G* é dito *bipartido* se o conjunto de vértices *V(G)* pode ser particionado em dois conjuntos de vértices tais que não exista aresta de entre vértices do mesmo conjunto.

Um grafo que não possui ciclos é dito um *grafo acíclico*. Uma *árvore* é um grafo conexo e acíclico. Uma árvore é denominada *enraizada* se um vértice é escolhido como especial; esse vértice é chamado *raiz*. Em uma árvore enraizada, o *nível de um vértice* é definido recursivamente: A raiz de uma árvore tem nível 0. Se um vértice tem nível *i*, seus filhos têm nível $i + 1$. Outra maneira de definir níveis é a seguinte: nível de um vértice *v* é a distância de v até o nó raiz em uma árvore enraizada. Em uma árvore enraizada, um vértice *v* é dito *pai* de *u* se há uma aresta de *u* para *v* e *v* está um nível acima de *u*, isto é $nível(v) = nível(u) - 1$. Neste caso, *u* é *filho* de *v* e $nível(u) = nível(v) + 1$. Um vértice *u* é dito *irmão* de *v* se possuem o mesmo vértice pai. Um vértice em uma árvore é dito uma *folha* se não possui filhos e um vértice é dito *interno* se não é uma folha.

2.1 **Busca em largura**

A *busca em largura* é um dos algoritmos mais simples para se percorrer um grafo (CORMEN, Thomas H. et al., 2009). Este algoritmo serve de base para uma grande quantidade de outros algoritmos como o algoritmo de Dijkstra (caminhos mais curtos de origem única) e o algoritmo de Prim (árvore geradora mínima).

Dado um grafo $G=(V, E)$, tome um vértice de origem *s*. A busca em largura é um procedimento que percorre as arestas do grafo *G* até ter visitado cada vértice que seja acessível partindo de *s*.

Este algoritmo naturalmente calcula a distância do vértice inicial *s* até todos os vértices que são alcançáveis a partir dele. Nele, o grafo é particionado em camadas: o próprio *s* é uma camada; os vértices que estão a distância 1 de *s* são outra camada; os vértices que estão a distância 2 de *s* constituem uma outra

camada e assim por diante. A busca em largura é apresentada inteiramente no Algoritmo 1.

É bom ressaltar que a busca em largura iniciada em *s* calcula apenas as distâncias de *s* para os outros vértices, isto é, a partir do procedimento de busca em largura partindo de *s*, apenas obtemos informações da distância de *s* para os outros vértices.

2.2 **Árvores da busca em largura**

O procedimento de busca em largura induz a uma estrutura de árvore, a árvore de caminhos mínimos. A estrutura de árvore indica as distâncias da raiz a qualquer outro vértice a partir do nível dele na árvore, isto é, *dist(r, v) = nível(v)* na árvore de busca em largura enraizada em *r*.

Na Figura 1, apresentamos um grafo e a árvore gerada na busca em largura iniciada no vértice 3.

Algoritmo 1: Busca em Largura (PAPADIMITRIOU, C. H.; DASGUPTA, C. H.; VAZIRANI, Umesh V., 2015)

Entrada: Grafo simples *G = (V, E)* e vértice inicial s pertencente ao conjunto *V(G)*.
Saída: Para todos os vértices *u* alcançáveis a partir de *s*, *distancia(u)* é atribuído à distância de *s* até *u*.

```
1. para todo u ∈ V faça:
2.       distância(u) ← ∞
3. distância(s) ← 0
4. adicione s à fila Q
5. enquanto Q não estiver vazia faça:
6.       u ← início da fila Q
7.       remova o vértice do início da fila
8.       para toda aresta (u,v) ∈ E faça:
9.                se distância(v) = ∞ então:
10.                       insira v em Q
11.                       distância(v) ← distância(u) + 1
```

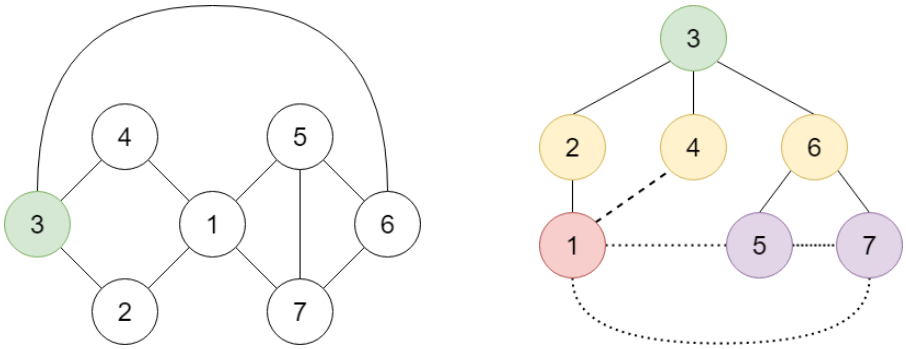

Figura 1: Grafo simples e árvore de busca em largura

Quando um vértice é visitado pela primeira vez na busca, ele é colocado na fila como filho do vértice que o fez ser “encontrado”. Por exemplo, o vértice 3 é vizinho direto do vértice 2. Assim, quando a busca pelos vizinhos de 3 é feita, o vértice 2 é encontrado pela primeira vez. Entre esses vértices há uma das arestas-pai na árvore, que nada mais são do que as arestas da árvore propriamente ditas (arestas em linha contínua na Figura 1).

Existem três outros tipos de arestas na árvore: *aresta-tio, aresta-irmão, aresta-primo*. A aresta-irmão é uma aresta entre dois vértices de mesmo nível na árvore e mesmo pai (linha pontilhada entre 5 e 7 na Figura 1), enquanto a aresta-primo é uma aresta entre dois vértices de mesmo nível na árvore e pais diferentes (linha pontilhada entre 1 e 5 e entre 1 e 7 na Figura 1). Isso ocorre quando há dois vértices adjacentes no grafo que estão a uma mesma distância do vértice raiz. A aresta-tio é uma aresta entre dois vértices de níveis diferentes, mais especificamente vértices de níveis consecutivos (linha tracejada entre 1 e 4 na Figura 1). Esta aresta indica um caminho mínimo alternativo entre a raiz e tal vértice. Note que, na Figura 1, o caminho de 3 a 1 passando por 2 e o caminho de 3 a 1 passando por 4 possuem o mesmo comprimento e, além disso, são caminhos mínimos. É bom ressaltar que a aresta-irmão não apresenta um novo caminho mínimo, apenas uma relação de vizinhança no grafo *G* entre vértices de mesmo nível na árvore.

Para nosso trabalho, serão importantes as arestas-pai e arestas-tio, indicadores de caminhos mínimos na árvore de busca em largura.

3 **NOÇÕES SOBRE ALGORITMOS CERTIFICADORES**

A visão tradicional de programas de computador, ou seja, um programa que recebe uma entrada, a processa e gera apenas uma saída, sempre foi a mais estudada e aprofundada. Por mais que, durante muito tempo, tenha se falado de “programas que verificam a resposta de outros programas” ou “algoritmos que mostram que estão certos”, levou-se bastante tempo até formalizar-se o que são algoritmos certificadores.

Um grande avanço nesse tema foi dado em 2011, após a divulgação do artigo “Certifying Algorithms” (MCCONNELL, R. M. et al., 2011), onde o autor desenvolve, além de uma motivação para investir-se mais nesse tipo de algoritmo, uma série de aplicações para algoritmos certificadores.

Um algoritmo certificador é um tipo de algoritmo que, dada uma determinada entrada *x*, produz uma saída *y* e, além dela, um certificado *w* que tenta “provar” que a saída *y* não foi comprometida por um erro, ou seja, tenta demonstrar que *y* é a saída correta para a entrada *x*. Espera-se que tal certificado seja de fácil verificação. Isto é, dada uma entrada *x*, o usuário receberá a saída *y* e um certificado *w*, e então poderá verificar manualmente ou por meio de outro *software* (o verificador), se a saída é confiável. O comportamento de um algoritmo comum comparado ao de um algoritmo certificador é mostrado na Figura 2.

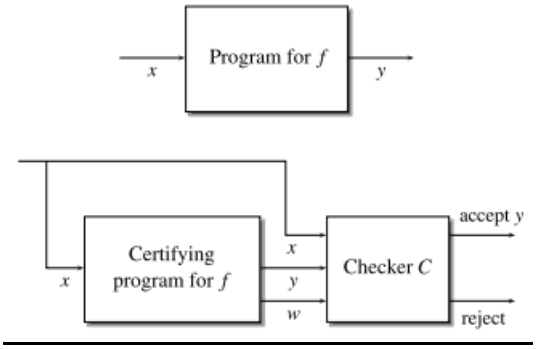

Figura 2: Reproduzida de (MCCONNELL, Ross M. et al, 2011). Comportamento de um algoritmo comum e de um algoritmo certificador.

O processo de verificação comentado anteriormente também é bastante importante para o estudo da certificação de algoritmos. Este processo pode ser feito por um outro *software* que toma como entrada *x*, *y* e *w* e prova que *y* é correto para *x* baseando-se em *w*. Tal *software* de verificação desejavelmente é de implementação simples, o que não diminui a confiabilidade na checagem. Uma prova de corretude do algoritmo verificador é desejável, porém não obrigatória, devido à simplicidade da tarefa executada por este.

Apresentaremos o caso do problema de decidir se um grafo é bipartido.

Em resumo, o problema de decisão de grafos bipartidos é: dado um grafo simples *G=(V(G), E(G)),* decidir se *G* é ou não é bipartido.

Um algoritmo não-certificador para este problema simplesmente responde SIM (ele é bipartido) ou NÃO (este grafo não é bipartido) para uma dada entrada. A versão certificadora de tal algoritmo vai um pouco mais longe e tenta mostrar de maneira convincente o porquê desta saída. O teorema a seguir apresenta uma maneira de se saber se um grafo é bipartido.

Teorema (BONDY, John Adrian; MURTY, Uppaluri Siva Ramachandra., 2008)**:** Um grafo é bipartido se, e somente se, não contém ciclos ímpares.

Tal teorema e sua demonstração podem ser encontrados em (BONDY, John Adrian; MURTY, Uppaluri Siva Ramachandra., 2008). A Figura 3 apresenta um exemplo de grafo bipartido e outro não bipartido.

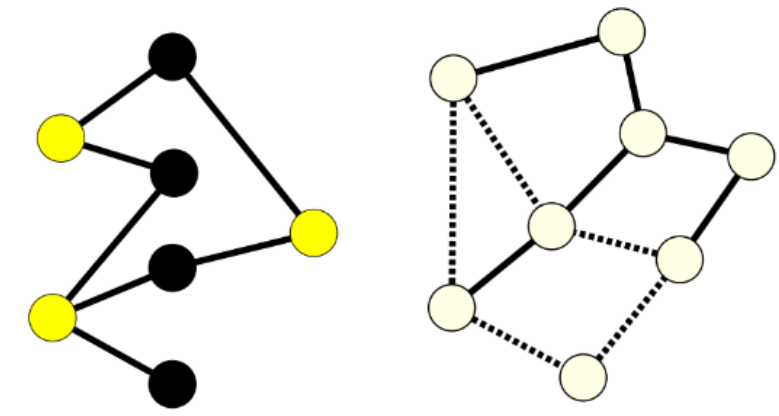

Figura 3: Reproduzida de (MCCONNELL, Ross M. et al, 2011). Grafo bipartido (à esquerda) apresentando a bipartição dos vértices (conjunto dos vértices amarelos e conjunto dos vértices pretos) e grafo não bipartido (à direita) com ciclo ímpar destacado.

Repare que tanto a existência da uma bipartição dos vértices como a existência de um ciclo ímpar, constituem, respectivamente, uma “prova” de que o grafo *G* é ou não um grafo bipartido. Desta maneira, temos tanto o certificado para a resposta SIM quanto para a resposta NÃO. De maneira resumida, um algoritmo certificador para o problema é mostrado no Algoritmo 2.

O exemplo dos grafos bipartidos pode aparentar ao leitor que criar um algoritmo certificador é uma tarefa simples. Porém, para a grande maioria dos casos, achar um certificado convincente, válido e em um tempo computacional aceitável é uma tarefa extremamente complicada. Falaremos mais à frente sobre tais dificuldades.

Em seu artigo, McConnell *et al*. desenvolveu um ferramental formal para algoritmos certificadores: apresentou diversas definições básicas, teoremas, exemplos claros e aplicações, além de características desejáveis de um algoritmo certificador. Iremos agora comentar sobre algumas destas definições, alguns teoremas criados, além de motivações para se estudar o assunto, citadas pelo autor em seu trabalho.

Algoritmo 2: Algoritmo Certificador para Problema do Grafo Bipartido
Entrada: Grafo simples G = (V, E)

1. Tome o grafo G
2. Verifique a existência de um ciclo ímpar
3. Caso exista ciclo ímpar:
4. Responda NÃO e apresente o ciclo ímpar como certificado
5. Caso não exista:
6. Responda SIM e apresente como certificado qualquer bipartição de V

3.1 **Motivação**

Um dos problemas recorrentes em Engenharia de *Software*, e podemos dizer também que é um dos maiores, é a corretude de um programa. A etapa de

testes de *software* constitui a maneira mais amplamente utilizada de se tentar garantir a corretude destes. A técnica de testes gera um conjunto de pares (x_i, y_i) onde x_i é a entrada para o programa e y_i é a saída correta para tal entrada. Tais entradas são submetidas no *software* e as saídas geradas são comparadas às saídas corretas y_i. Caso alguma saída não seja igual à correta, constata-se uma falha no *software,* que deve ser corrigida. Caso nenhuma saída seja diferente da correta, ainda não se pode dizer que o *software* não apresenta falhas. Isto constitui um grande problema: o teste de *software* constata a presença de erros, mas não a inexistência deles; um programa pode ser testado para um grande conjunto de entradas, mas podem existir outras não testadas onde o programa não apresenta a saída correta.

Um algoritmo certificador, por sua vez, gera para toda entrada, além da saída, uma prova de que ela está correta e isto configura que o algoritmo ao mesmo tempo que executa sua funcionalidade e apresenta a saída, faz um teste de *software* por meio do certificado e é ainda melhor, pois o certificado é gerado para toda entrada do *software*.

A seguir, apresentaremos outras vantagens que motivam o uso de algoritmos certificadores:

a) Teste de Software para todo tipo de entrada, inclusive as mais complexas: Diferentes técnicas e ferramentas para teste já foram criadas de maneira a garantir uma menor chance de softwares com *bugs* serem lançados no mercado ou entregues a clientes. Por mais que essas técnicas e ferramentas sejam elaboradas, o teste do software ainda apresenta “lacunas” com relação a todas as possíveis entradas que um usuário pode submeter, principalmente quando se tratam de entradas não tão simples como grafos, por exemplo. Algoritmos certificadores fazem teste para qualquer entrada submetida pelo usuário e dizem se a saída associada a tal entrada é confiável ou não.

b) Corretude de instâncias: Existem tipos de algoritmos certificadores mais complexos e mais robustos que ainda testam se a entrada faz parte do conjunto de entradas possíveis e aceitáveis para as quais o software foi desenvolvido, ou seja, testam se a entrada segue uma precondição estabelecida.

Isso é interessante, pois identifica se é de fato uma falha do *software* ou da entrada fornecida.

c) Isolamento de erro: se considerarmos um sistema como partes de um sistema algoritmos certificadores, podemos perceber que dada a constatação de uma saída não confiável por parte deste elemento certificador, a saída incorreta pode ser isolada e não ser passada adiante para outros módulos do sistema, disseminando o erro.

d) Desenvolvimento de *software* mais efetivo: é demandado um tempo considerável na identificação de erros em um *software* – criar casos de testes, submetê-los, comparar saídas com as esperadas etc. – além da grande quantidade de tempo demandada em consertar tais erros. Com algoritmos certificadores, o tempo de realização desta tarefa tende a ser minimizado.

e) Confiança no software com o mínimo de “investimento intelectual”: seguindo-se os preceitos de um bom algoritmo certificador e assumindo que o usuário tenha um conhecimento mínimo e básico sobre o problema, podemos afirmar que não é necessário muito “investimento intelectual” nele para que este entenda o que o certificado significa e como ele prova que a saída é correta. Além disso, dadas as boas práticas na criação da certificação, será fácil para ele fazer a verificação da relação entre a saída e a certificação.

f) Não é necessário conhecimento da implementação: sabendo-se apenas o tipo de entrada, as saídas e o problema resolvido pelo algoritmo é possível criar um algoritmo certificador, ou seja, o código-fonte pode ser mantido em segredo, sem violar a propriedade intelectual.

g) Verificação simples: dado um certificado simples ou um software que faz o processo de verificação – tal software é extremamente simples na maioria dos casos – é fácil constatar a confiabilidade da saída com uso do certificado. Além disso, os algoritmos verificadores são tão descomplicados que é quase que natural provar sua corretude.

h) Certificação e verificação podem ser feitas de forma remota

É bom lembrar que a maioria dos algoritmos produzidos e usados são não-certificadores e é um grande desafio encontrar algoritmos certificadores que sejam tão eficientes quanto as versões não-certificadores e que tragam tal testemunho que garante a confiabilidade da saída.

3.2 **Pré-condições e pós-condições**

Considere um algoritmo que toma uma entrada de um conjunto *X* e gera uma saída em um conjunto *Y.*

A entrada x ∈ X supostamente deverá satisfazer uma *pré-condição* dita φ *(x)* e, por sua vez, a saída y ∈ Y satisfazer uma *pós-condição ψ(x, y)*. Temos que φ é uma função: φ *: X → {V, F}* e *ψ : X × Y → {V, F}.* Chamamos esse par de funções φ e *ψ* de especificação de entrada e saída, respectivamente.

Como exemplo, vamos pensar no caso do problema de grafos bipartidos. Temos que o conjunto *X* é o conjunto de todos os grafos simples (não-direcionados) e finitos e que *Y* é o conjunto *{bipartido, não bipartido} ou {SIM, NÃO}*. A partir disso, podemos dizer que φ *(x) = T* para todo *x ∈ X* e *ψ (x,y) = T* se e somente se:

a) *x* é bipartido e *y* = bipartido ou *y* = SIM;

b) *x* é não bipartido e *y* = não bipartido ou *y* = NÃO.

A seguir apresentaremos os tipos de algoritmos certificadores e em seguida algumas questões relacionadas a eficiência e existência de certificações para todo tipo de problema.

3.3 **Tipos de algoritmos certificadores**

Nem sempre pode-se garantir todas as propriedades desejáveis de um algoritmo certificador: em alguns momentos não é possível verificar se a pré-condição foi estabelecida, por exemplo. A existência ou falta dessas propriedades nos algoritmos certificadores definem o seu tipo na classificação proposta por McConnell *et al.*.

3.3.1 Algoritmos fortemente certificadores

São o tipo de algoritmo certificador desejável. Este tipo de algoritmo vai além da certificação do algoritmo propriamente dito. Em outras palavras, um algoritmo fortemente certificador mostra também se o usuário forneceu ou não uma entrada

válida para o algoritmo, isto é, ele certifica se a entrada é correta ou não para este algoritmo.

Para cada entrada $x \in X$, um *algoritmo fortemente certificador* terá duas saídas possíveis:

a) Um certificado mostrando que a entrada *x* não satisfaz a pré-condição estabelecida para o algoritmo.

b) Uma saída *y* e um certificado *w* mostrando que o par *(x,y)* satisfaz a pós-condição.

Este tipo de algoritmo certificador é desejável, pois aponta com precisão se há algum problema no fornecimento da entrada, tornando o certificado mais preciso.

3.3.2 Algoritmos puramente certificadores

Um *algoritmo certificador* (nem forte, nem fraco) é aquele que simplesmente mostra um certificado para todos os casos de entrada, de forma que ele não aponta diretamente se a entrada viola a pré-condição estabelecida. Este é o tipo de algoritmo certificador mais amplamente utilizado e estudado por ser simples e útil ao mesmo tempo.

Tomando como exemplo a busca binária que consiste em dizer se um dado valor *x* está presente em um vetor *A* de valores reais.

Uma pré-condição estabelecida pelo algoritmo da busca binária é que o vetor de entrada deva estar ordenado em ordem crescente.

Um algoritmo fortemente certificador:

a) faria a análise da entrada e apresentaria um certificado caso a sequência de valores não estivesse ordenada: um par de valores a_i e a_j, tais que $i < j$ e $a_i > a_j$; ou

b) apresentaria um certificado para os resultados: o índice *i* onde o valor foi encontrado configura um certificado para o SIM e dois índices *i* e *j* consecutivos

tais que o valor buscado *x* não está entre eles, ou seja, $a_i < x < a_j$ (partindo do fato que a sequência está ordenada) configuram um certificado para o NÃO.

Um algoritmo puramente certificador para busca binária apenas apresentaria o certificado descrito no item *b*. Note que ele não faz a checagem da entrada, portanto não pode apontar se a entrada segue ou não as pré-condições estabelecidas para o algoritmo.

3.3.3 Algoritmos fracamente certificadores

Um *algoritmo fracamente certificador* é um tipo de algoritmo certificador que não apresenta certificado para todo tipo possível de entrada, isto é, ele não apresenta certificado para entradas que não satisfazem a pré-condição estabelecida para o algoritmo (o algoritmo pode até não parar de rodar neste caso).

Um exemplo de algoritmo fracamente certificador é o de SAT-randomizado, onde, dada uma fórmula *x*, quer provar-se que ela é satisfatível usando-se valores aleatórios para as variáveis que compõem *x.* O certificado para o SIM é um conjunto de valores booleanos para *x* que tornem a fórmula verdadeira; porém, para o NÃO, não se tem um certificado que possa ser verificável e gerado em tempo aceitável – pois geraria aleatoriamente diversos conjuntos de valores que não tornariam a fórmula verdadeira, assim o algoritmo não pararia. Logo, o algoritmo descrito anteriormente é fracamente certificador.

Dificilmente trabalha-se com algoritmos fracamente certificadores. Estes não são tão aplicáveis e úteis para a maioria dos problemas.

3.4 **Questões sobre eficiência**

Um detalhe importante a se citar é a respeito da eficiência de um algoritmo certificador. Definimos um algoritmo certificador *P* como eficiente se existir um verificador *C* para ele, tal que as complexidades assintóticas de *P* e *C* sejam no

máximo o tempo do melhor algoritmo conhecido satisfazendo as especificações de entrada e saída.

Esta definição traz as seguintes questões: Quando vale a pena se usar um algoritmo certificador? E se ele não for tão eficiente quanto a versão não-certificadora, vale a pena usá-lo?

A resposta para essas perguntas são complexas e envolvem diversos fatores, mas podemos resumir da seguinte maneira:

a) Deve-se ter um maior esforço e estudo em criar algoritmos certificadores que sejam, ao mesmo tempo, fortes e eficientes;

b) Um algoritmo um pouco mais lento, porém certificador, pode ser mais útil e confiável do que um na versão não-certificadora (principalmente para aplicações onde *bugs* são fatais, porém o tempo de execução não é um fator crucial);

c) O verificador na maioria das vezes é simples e eficiente, inclusive, pode ser embutido no próprio algoritmo sem alterar a complexidade assintótica;

d) Em alguns casos, criar um certificado mais forte pagando-se o preço de um maior tempo de execução pode ser interessante em nome da confiabilidade no *software*;

e) Algoritmos certificadores mais lentos podem ser usados para criar casos de testes para as suas versões não-certificadoras;

f) Pode ocorrer que, para entradas pequenas ou tipos especiais de entradas, a versão certificadora seja ser tão rápida quanto a não-certificadora.

O ponto principal é: o ideal é que a versão certificadora seja eficiente ao mesmo tempo que produza um bom e convincente certificado em um tempo computacional aceitável. Entretanto, caso a versão não-certificadora não seja eficiente, vale a pena avaliar seu uso e seu impacto.

3.5 **Questões sobre simplicidade e verificabilidade**

O leitor deste trabalho já pode ter percebido que estamos citando diversas propriedades desejáveis de um algoritmo certificador. Algumas delas são definidas matematicamente e de forma exata como a eficiência. Entretanto, quando falamos de “certificados simples” ou “certificados fáceis de se verificar”, tratamos de termos não tão exatos.

Quando tentamos garantir em um algoritmo certificador as propriedades de simplicidade e verificabilidade, estamos tentando garantir que o usuário de posse do certificado possa facilmente avaliar se a saída é confiável sem dispender muito tempo ou dar a ele muito trabalho. Por mais que definir tais propriedades matematicamente seja algo complicado, McConnell apresenta certas formalizações para tais termos.

O autor cita em seu trabalho a trivialidade em definir se *y* é confiável dada o terno *W(x,y,w)* formado por entrada, saída e certificado, respectivamente. Uma das maneiras que o autor utiliza para definir tal trivialidade é interessante: O autor afirma que podemos dizer que é trivial determinar a validade de *W(x,y,w)* se houver um algoritmo de decisão em tempo linear que valida *W*. Desta maneira, o autor vincula os conceitos de simplicidade do certificado e verificabilidade a um fator matematicamente definível que é a eficiência do verificador.

3.6 **Questões sobre a existência de algoritmos certificadores para todo tipo de problema**

Uma das questões que pode passar na mente de alguém que se depara com este assunto é: Todo algoritmo tem uma versão certificadora?

Como vimos anteriormente, existem diversos tipos de algoritmos certificadores e os fortemente certificadores são os mais desejáveis. Porém, há complicadores em criá-los: as pré-condições e pós-condições, principalmente a primeira. Para alguns problemas, não é possível prever na totalidade os comportamentos de entrada e saída, muito menos defini-los, como, por exemplo, no problema da parada (*halting problem*). Agora, se avaliarmos a existência de

algoritmos certificadores sob a visão de algoritmos fracamente certificadores, podemos afirmar que existem versões certificadoras para todo problema. Isso ocorre por meio do relaxamento da sua própria definição: o algoritmo não necessariamente precisa parar de rodar ou até mesmo apresentar um certificado caso a pré-condição não seja atendida.

Tais resultados foram enunciados e demonstrados por McConnell *et al*. (MCCONNELL, R. M. et al., 2011).

3.7 **Verificadores**

Um verificador para um terno *W(x,y,w)* composto por entrada, saída e certificado, respectivamente, é um algoritmo que recebe tal terno como entrada e retorna se *W(x,y,w)* é válido ou não. O verificador nada mais é que um algoritmo usado para avaliar se w realmente certifica que *y* é a saída correta para *x*. Algumas propriedades são desejáveis aos verificadores:

a) Corretude: eles devem ser corretos acima de tudo. Essa deve ser a característica obrigatória de um verificador.

b) Tempo de execução: é desejável que ele seja linear no tamanho da entrada, a tripla *(x,y,w).*

c) Complexidade lógica: um verificador deve ser, e na maioria das vezes é, um programa simples de se implementar e entender.

É bom frisar no fato da corretude, afinal o verificador deve garantir a confiabilidade da saída. É desejável que seja feita uma prova formal que comprove que o verificador está correto, mas na maioria dos casos tal algoritmo é tão simples que tal demonstração se torna dispensável.

Outro fato importante a se citar é sobre a simplicidade do certificado. Em alguns casos, o certificado é tão simples que a implementação de um verificador se torna opcional, pois o usuário poderá facilmente comprovar a validade de *W(x,y,w).*

3.8 **Exemplo de algoritmo certificador: Caminhos mínimos**

Iremos apresentar agora um dos exemplos clássicos e que será de grande importância para o nosso trabalho: certificação para o problema dos caminhos mínimos.

Este algoritmo e a demonstração mostrada a seguir feita por McConnell *et al.* (MCCONNELL, R. M. et al. – 2011) garantirão que as distâncias dadas pelas árvores de largura são corretas e que estas podem ser usadas para certificar caminhos mínimos.

Seja $G = (V, E)$ um grafo direcionado, e seja s um vértice especial, o vértice de partida ou fonte ou ainda origem, e seja $c : E \rightarrow \mathbb{R}_+^*$ uma função de custo positivo sobre as arestas.

Seja p um caminho. O custo do caminho $c(p)$ é a soma dos custos das arestas em p. O caminho mínimo do vértice s até outro vértice v é aquele de custo mínimo entre todos os caminhos de s a v.

Sejam a função d a distância correta do vértice de origem aos outros vértices e D a função que representa o valor calculado pelo algoritmo de distâncias mínimas. Para demonstrar que a função D retorna o valor correto, ou seja, certificá-la, precisamos da definição básica de caminhos mínimos. O que queremos mostrar é que a saída denotada pela função $D : V \rightarrow \mathbb{R}_+$ é o valor correto de d, isto é $D(v) = d(v)$.

Para comprovar tal fato, teremos que comprovar que D atende às seguintes propriedades:

a) $D(s) = 0$ → com relação ao vértice de origem, temos que o custo de chegar até ele deve ser 0 (valor inicial e base para início do algoritmo);

b) Para toda aresta (u, v), temos que $D(v) \leq D(u) + c(u,v)$ → baseado na desigualdade triangular e no fato de os custos das arestas sempre serem positivos, temos que o custo para alcançar o vértice v a partir de u é igual a $D(u)$ mais o custo da aresta de u para v, podendo ser u o vértice anterior no caminho mínimo, neste caso $D(v) = D(u) + c(u,v)$, ou não, neste caso $D(v) < D(u) + c(u, v)$;

c) Para todo vértice $v \neq s$, existe uma aresta de u para v com $D(v) = D(u) + c(u, v)$.

Primeiramente, repare que a inexistência de custos negativos, garante de forma direta que $D(s) = d(s) = 0$.

Para provarmos que $D(v) = d(v)$ para outros vértices, fornecemos uma prova por indução, usando o caso anterior como base. Considere qualquer vértice $v \neq s$ e seja $v_0 = s, v_1, ..., v_k = v$ o caminho mínimo de s até v, ou seja, o caminho de custo $d(v)$. Temos que $d(v_{i+1}) = d(v_i) + c(v_i, v_{i+1})$ para todo i ≥ 0 e $D(v_0) = 0 \leq 0 = d(v_0)$, conforme demonstrado anteriormente. Como nossa hipótese de indução, suponha que $D(v_i) \leq d(v_i)$ para algum $i \geq 0$. Então:

$D(v_{i+1}) \leq D(v_i) + c(v_i, v_{i+1})$, pela desigualdade triangular
$\leq d(v_i) + c(v_i, v_{i+1})$, pela hipótese
$= d(v_{i+1})$

E assim mostramos indutivamente que $D(v) \leq d(v)$. Agora, precisamos garantir que $D(v) = d(v)$. Para tal demonstração iremos supor que $D(v) < d(v)$ para pelo menos algum vértice v e isso nos levará a uma contradição. Dentre aqueles vértices que $D(v) < d(v)$, tome o vértice com menor valor de $D(v)$. Partindo do fato que $D(s) = d(s) = 0$, temos que $v \neq s$, pois assumimos que $D(v)$ é estritamente menor que $d(v)$.

Sabemos que existe um vértice u tal que $D(v) = D(u) + c(u, v)$, o vértice anterior a v no caminho. Pelo fato de só haverem custos positivos, $D(u) < D(v)$. Devido a nossa escolha de v, $D(u) = d(u)$, do contrário u seria o vértice de menor valor para a função D dentre aqueles que o valor desta função é menor que o da função d.

Sendo assim, há um caminho de s a u de custo $D(u) = d(u)$ e consequentemente há um caminho de custo $D(v) = D(u) + c(u,v) = d(u) + c(u,v) \geq d(v)$ de s para v, ou seja, $d(v) \leq D(v)$, o que é absurdo de acordo com nossa hipótese, portanto. $D(v) = d(v)$.

Assim, podemos afirmar que $d(v) = D(v)$, e mais: que o algoritmo certificador para caminhos mínimos pode apresentar como certificado a aresta (u, v) que faz com que $D(v) = D(u) + c(u, v) = d(u) + c(u,v) = d(v)$.

4 **ALGORITMOS CERTIFICADORES PARA CONVEXIDADE GEODÉSICA**

A convexidade geodésica é a convexidade definida pelos caminhos mínimos. Neste capítulo, apresentaremos a convexidade geodésica e suas definições. Além disso, apresentaremos os algoritmos certificadores criados em torno deste tema, além das demonstrações envolvidas no processo de criação.

4.1 **Noções de Convexidade Geodésica**

Uma geodésica entre dois vértices *u* e *v* é um caminho de *u* a *v* em *G* com *d(u,v)* arestas, ou seja, um caminho mínimo entre *u* e *v*. O intervalo fechado, *I[S]* de um conjunto $S \subseteq V(G)$ é o conjunto de todos os vértices que se encontram em algum geodésica entre todos pares de vértices de *S*, incluindo os vértices em *S*.

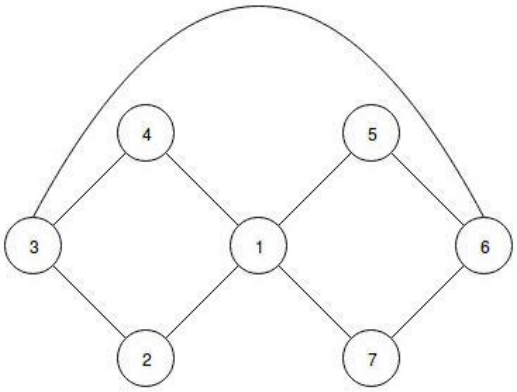

Figura 4 - Grafo simples

No grafo da Figura 4, por exemplo:

a) Para $S = \{1,3\}$, temos que $I[S] = \{1,2,3,4\}$.
b) Para $S = \{3,6,7\}$, temos que $I[S] = \{3,6,7\}$
c) Para $S = \{1,4\}$, temos que $I[S] = \{1,4\}$.

Dado um conjunto $S \subseteq V(G)$, S é *convexo* se $I[S] = S$. No grafo da Figura 4, temos que:

a) Para $S = \{1, 2, 3, 4\}$, temos que $I[S] = \{1,2,3,4\}$, portanto *S* é convexo.
b) Para $S = \{1,3\}$, temos que $I[S] = \{1,2,3,4\}$, portanto *S* não é convexo.

Dado um conjunto de vértices $S \subseteq V(G)$, temos que S é *geodésico* se $I[S] = V(G)$. No grafo da Figura 4, temos que:

a) Para $S = \{1,3,6\}$, temos que $I[S] = \{1,2,3,4,5,6,7\} = V(G)$, logo S é geodésico.

b) Para $S = \{5, 7\}$, temos que $I[S] = \{1,5,6,7\} \neq V(G)$, logo S não é geodésico

Seja $S \subseteq V(G)$. A *envoltória convexa*, denotada por $I_h[S]$, de S é o menor conjunto convexo de G que contém todos os vértices de S. Usando aplicações sucessivas do intervalo fechado é possível obter outra definição equivalente. Considere que $I^k[S] = I[I^{k-1}[S]]$, ou seja, considerando $I[S] = I^1[S]$, temos que $I^2[S] = I[I[S]]$, por exemplo20j. Seja k o menor inteiro tal que $I^k[S] = I^{k+1}[S]$, então, $I_h[S] = I^k[S]$.

Um conjunto S é denominado *envoltório* se $I_h[S] = V(G)$.

No grafo da Figura 4, temos que:

a) Para $S = \{1,2,6\}$, temos que $I^1[S] = \{1,2,3,5,6,7\}$ e $I^2[S] = \{1,2,3,4,5,6,7\} = V(G)$, logo S é envoltório.

b) Para $S = \{1,3\}$, não existe $k < n+1$ tal que $I^k[S] = V(G)$, portanto S não é envoltório.

4.2 **Algoritmos certificadores para convexidade geodésica**

O problema de decidir se um conjunto $S \subseteq V(G)$ é convexo possui um algoritmo simples como solução. Sua versão simplificada é apresentada no Algoritmo 3.

Tal algoritmo é executado em tempo $O(n^3)$ e não apresenta nenhum certificado. Apresentaremos agora os detalhes e implementação do algoritmo certificador para conjuntos convexos, geodésicos e envoltórios.

Algoritmo 3: Algoritmo para decidir se um conjunto é convexo (FARBER, Martin; JAMISON, Robert E. – 1987)

Entrada: Grafo simples *G = (V, E),* conjunto *S* que se deseja saber se é convexo.

1. Executar $|V(G)| = n$ buscas em largura para calcular as distâncias entre todos os vértices.

2. Para todo par de vértices $u, v \in S$, se existe $w \notin S$ tal que $d(u,v) = d(u,w) + d(w,v)$.

3. Se este *w* existe, *S* é não-convexo. Senão, *S* é convexo.

4.2.1 Ancestrais

Para nossos algoritmos certificadores para convexidade geodésica, criamos uma definição simples que será usada de base para todos eles: Tome uma árvore enraizada em $r \in S$, sendo *S* um conjunto de vértices do grafo, o qual se quer saber se este é convexo ou não. Tal árvore é obtida pela busca em largura iniciada em *r*.

Além das arestas-pai próprias da árvore, consideraremos as arestas-tio, sendo esses tipos de arestas importantes para nossa análise de distâncias. Afinal, arestas-irmão e arestas-primo indicam caminhos maiores, enquanto arestas-tio apresentam caminhos alternativos da raiz até um determinado vértice, sem "aumentar" a distância da raiz até tal nó.

O conjunto de *ancestrais* de um vértice *v* em uma árvore enraizada em *r* é obtido recursivamente como:

$$Ancestrais(v) = \begin{cases} \emptyset, & se\ v = r \\ Pai(v) \cup Tios(v) \cup Ancestrais\big(Pai(v)\big) \cup \bigcup_{w \in Tios(v)} Ancestrais(w), & se\ v \neq r \end{cases}$$

Nesta definição, *Pai(v)* é o conjunto unitário contendo o vértice do nível acima da árvore alcançável diretamente por uma aresta-pai. *Tios(v)* é o conjunto de vértices do nível acima de *v* na árvore alcançáveis por uma aresta-tio.

A ideia do algoritmo certificador é usar as árvores da Busca em largura enraizadas nos vértices de *S* e ancestrais como modo de certificar se um conjunto *S* de vértices é convexo ou não.

Como a árvore apresenta todos os caminhos mínimos entre a raiz e qualquer outro vértice do grafo, tentaremos por meio dos ancestrais traçar todos os caminhos mínimos entre a raiz $r \in S$ e qualquer outro vértice $s \in S$. Resumindo, dado um vértice $s \in S$ em uma árvore enraizada em $r \in S$ iremos do nível dele até a raiz *r* de

forma ascendente por meio dos seus ancestrais, e verificando se estes são ou não pertencentes a *S*. Isto é, por meio dos ancestrais desses vértices, estaremos verificando qualquer caminho mínimo de *r* até *s* e analisando se haverá ou não um vértice $z \notin S$, comprovando a não-convexidade. Caso ele não exista, os caminhos mínimos entre todos os vértices de *S* já configuram um certificado de que tal conjunto é convexo.

Precisamos, portanto, mostrar que os ancestrais são realmente confiáveis para certificar que um dado conjunto é convexo.

Considerando o caso, para certificar que *S* não é convexo, teríamos um ancestral $z \notin S$ de algum vértice *y* pertencente a *S* na árvore enraizada em algum $x \in S$. Para garantir a corretude de nosso algoritmo, provamos a Proposição 1:

Proposição 1: Seja *T* a árvore de largura enraizada em *x*. Se existem vértices *y* e *z* tais que $y \in S$, $z \notin S$ e $z \in I[x, y]$, então existe um vértice *v* em uma geodésica entre *z* e *y* tal que $v \in S$ e existe *v'* em $Pai(v) \cup Tio(v)$ tal que $v' \notin S$.

Demonstração: Como $z \notin S$, então $z \neq x$ e $z \neq y$, pois $x, y \in S$.

Seja *v* um vértice mais próximo de *z* em alguma geodésica *P* de *z* a *y* tal que $v \in S$. Considere *v'* o vértice anterior a *v* no caminho de *z* a *v* em *P*.

Se $v' \in S$ então temos um absurdo, pois $d(v', z) < d(v, z)$ e *v'* seria o vértice mais próximo de *z* que pertence a *S*. Logo $v' \notin S$ e $v' \in Pai(v) \cup Tio(v)$. ■

Assim, se algum *z* não pertencente a *S* estiver numa geodésica entre a raiz $x \in S$ e um vértice da árvore $y \in S$ então ele será ancestral de *y,* o que garante a confiabilidade e corretude de nosso algoritmo. Como exemplo, tome o grafo da Figura 4 e considere o conjunto $S = \{1,3\}$, cujo intervalo fechado é $I[S] = \{1, 2, 3, 4\}$. De acordo com a Proposição 1, os vértices 2 e 4 são ancestrais de algum vértice pertencente a *S* ou de algum outro ancestral de um vértice de *S* em alguma árvore enraizada em também um algum $r \in S$, isto é, se 2 e 4 participam de uma geodésica entre 1 e 3, então eles são ancestrais de pelo menos um destes ou de algum outro vértice que seja ancestral destes. A Figura 5 mostra 2 e 4 como ancestrais de 3.

4.2.2 Montagem da árvore usada pelo algoritmo

O primeiro algoritmo criado em nosso trabalho é o procedimento que cria as árvores da busca em largura com as arestas-pai e arestas-tio, de forma que os

algoritmos certificadores analisem os ancestrais de cada vértice do conjunto *S* para o que queremos decidir se é convexo ou não. Este algoritmo é apresentado como Algoritmo 4. Nos algoritmos usaremos a seguinte notação: *dist[v][u]* representa a distância de *v* até *u*; *pai[v][u]* representa o pai de *u* na árvore enraizada em *v* (caso o valor seja -1, ele não possui pai); e *tio[v][u]* representa o conjunto de tios de *u* na árvore enraizada em *v*.

Repare que o procedimento nada mais é que uma busca em largura que faz a análise de arestas-pai e arestas-tio ao longo da execução e armazena os tios e pais de cada vértice nos vetores de mesmo nome.

Algoritmo 4: Montar árvores.

Entrada: Grafo G.

Saída: Árvores de largura enraizadas em todos os vértices contendo apenas arestas-pai e arestas-tio, representadas pelos vetores pai e tio.

```
1. para cada v ∈ V(G) faça
2.     para cada w ∈ V(G) faça
3.         dist[v][w] ← ∞.
4.         pai[v][w] ← -1
5.         tio[v][w] ← ∅
6.     dist[v][v] ← 0
7.     insira v na uma fila Q
8.     enquanto Q não estiver vazia faça
9.         u←início de Q
10.        remover u de Q
11.        para cada w ∈ N(u) faça
12.            se dist[v][w] = ∞ então
13.                insira w em Q
14.                dist[v][w] ← dist[v][u]+1
15.                pai[v][w] ← u
16.            senão se dist[v][w] ≠ ∞ e pai[v][w] ≠ u então
17.                se |dist[v][w] – dist[v][u] | = 1 então
18.                    se dist[v][w] < dist[v][u] então
19.                        adicione w a tio[v][u]
20. retorna vetores pai e tio
```

4.2.3 Algoritmo certificador para conjuntos convexos

O algoritmo certificador para conjuntos convexos faz uso das árvores construídas no Algoritmo 4. Utilizando a Proposição 1 e no fato que as árvores da

Busca em Largura apresentam as distâncias mínimas corretas, construímos o Algoritmo 5.

O Algoritmo 5 passa por cada árvore de largura dos vértices do conjunto *S* verificando os pais e tios destes. O algoritmo conta com uma estrutura de pilha, que implementa propriamente a recursividade da definição dos ancestrais e o acesso controlado à estrutura da árvore, e um vetor (marcado) para fazer o controle de qual vértice deve ter seus ancestrais avaliados. Caso um vértice $v \in S$ seja encontrado com *marcado[v]* igual a 0, ele deve entrar na pilha para ter seus ancestrais avaliados naquela mesma árvore.

Algoritmo 5 Certificador para conjuntos convexos.
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e conjunto S de vértices.
Saída: Certificados para o SIM ou para o NÃO.

1. para cada v ∈ S **faça**
2. defina marcado como um vetor de tamanho |V(G)|
3. **INICIALIZAÇÃO(marcado)**
4. **para cada** w ∈ S e w ≠ v **faça**
5. empilhe w em uma pilha P
6. **enquanto** P não estiver vazia **faça**
7. u ← topo de P
8. remova u de P
9. marcado[u]← 1
10. **TESTE_PAI(v, u, w, marcado, pai, P)**
11. **TESTE_TIO(v, u, w, marcado, tio, P)**
12. exiba lista de pais e tios para certificar o SIM

O procedimento de inicialização simplesmente atribui para cada vértice o valor 0 no vetor marcado, ou seja, a cada árvore analisada, o vetor é reinicializado, pois assim podemos analisar todos os ancestrais corretamente em cada uma das árvores.

Procedimento INICIALIZAÇÃO(vetor marcado):

1. para cada w ∈ |V(G)| **faça**
2. marcado[w] ← 0

Após a inicialização, começamos a analisar os ancestrais dos vértices de S dentro da árvore enraizada em *v*. A análise de ancestrais é dividida em dois procedimentos: análise do pai e análise dos tios.

No procedimento de análise do pai *(TESTE_PAI)* verificamos se o pai do vértice *u* é um vértice de *S*. Caso seja e este vértice não esteja com o valor de marcado como 1, ele será posto na pilha para análise posterior de seus ancestrais. Caso não pertença a *S*, já temos um certificado para o NÃO e podemos encerrar a execução do algoritmo. Neste último caso, o certificado pode ser o caminho do vértice *v* até o vértice *w* passando pelo pai de *u*.

Procedimento TESTE_PAI(vértice v, vértice u, vértice w, vetor marcado, vetor pai, pilha P):

1.se pai[v][u] ≠ -1 **então**
2. **se** pai[v][u] ∉ S **então**
3. exiba caminho v → pai[v][u] → w como certificado para o NÃO
4. **pare**
5. **senão**
6. **se** marcado[pai[v][u]] = 0 **então**
7. empilhe pai[v][u] em P

No procedimento de análise dos tios (*TESTE_TIO*) verificamos se os tios do vértice *u* são vértices de *S.* Caso sejam, todos estes vértices que não estejam com o valor de marcado como 1 serão postos na pilha para análise posterior de seus ancestrais. Caso algum deles não pertença a *S*, já temos um certificado para o NÃO e podemos encerrar a execução do algoritmo. Neste último caso, o certificado pode ser o caminho do vértice *v* até o vértice *w* passando pelo tio de *u*.

Procedimento TESTE_TIO(vértice v, vértice u, vértice w, vetor marcado, vetor pai, pilha P):

1.se tio[v][u] ≠∅ **então**
2. **se** algum x ∈ tio[v][u] não pertence a S **então**
3. exiba caminho v → x → w como certificado para o NÃO
4. **pare**
5. **senão**
6. **para cada** t ∈ tio[v][u] **faça.**
7. **se** marcado[t] = 0 **então**
8. empilhe t em P

Repare que, se o processo iterativo de busca entre os ancestrais dos vértices de *S* finaliza, isso implica que *S* é convexo. Podemos apresentar o conjunto dos pais e dos tios de *S* como certificado para o SIM, atestando que nenhum deles está fora de *S*. Note também que é fácil implementar um verificador para ambos os certificados: no caso do SIM, ele apenas percorreria cada lista verificando se os elementos pertencem a algum *S*, e no caso do NÃO apenas verificaria a existência do caminho mínimo com elemento não pertencente a *S*.

Outro detalhe importante é com relação à eficiência de nosso algoritmo que se equipara à versão não-certificadora. A versão completa do algoritmo pode ser encontrada no Apêndice A.

Como exemplo, considere o grafo da Figura 4 e o conjunto $S = \{1, 3\}$.

O Algoritmo 4 criaria duas árvores de busca em largura: uma enraizada no vértice 1 e outra enraizada no vértice 3. Essas árvores seriam passadas para o Algoritmo 5. Iniciando pela árvore do vértice 1, apresentada na Figura 5, ele faria a análise de pais e tios de cada vértice $v \in S$, no caso o vértice 3.

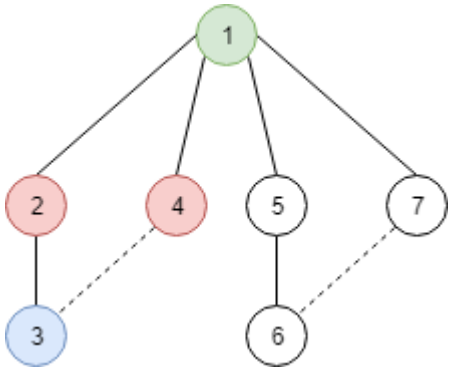

Figura 5 - Árvore da Busca em Largura enraizada em 1. Arestas-tio em tracejado. Os vértices 2 e 4 não pertencem a *S* e são ancestrais de 3.

Na análise de pais do vértice 3, logo o algoritmo encontraria o vértice 2 que não pertence a *S*. Assim, o caminho 1 → 2 → 3, seria um certificado para o NÃO. Caso o algoritmo ainda fizesse a análise de tios, também encontraria o vértice 4 que não pertence a *S* e o caminho 1 → 4 → 3 configuraria outro certificado para o NÃO.

Ainda considerando o grafo da Figura 4, tome agora o conjunto $S = \{1, 2, 3, 4\}$.

O Algoritmo 4 geraria as quatro árvores necessárias para o Algoritmo 5, apresentadas na Figura 6. Cada uma dessas árvores está enraizada em um vértice de *S*.

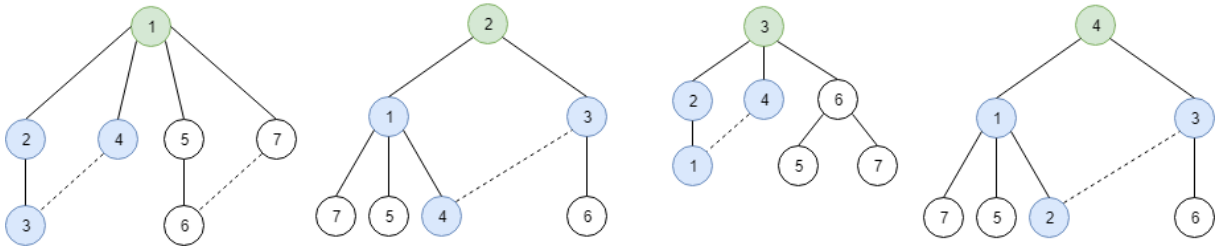

Figura 6 - Árvores de busca em largura enraizadas nos vértices de *S = {1,2,3,4}.* Arestas-tio em tracejado.

Note que, em cada árvore, os vértices ancestrais de 1, 2, 3 e 4 são apenas vértices de *S*. Por exemplo, na árvore enraizada em 4, com relação ao vértice 2: seu pai é o vértice 1 $\in S$ e seu tio é o vértice 3 também pertencente a *S.* Portanto, tanto na análise de tios, quanto na de pais em todas as árvores, para todos os vértices de *S* não encontraríamos nenhum vértice *u* tal que $u \notin S$.

4.2.4 Algoritmo certificador para conjuntos geodésicos

O algoritmo certificador para conjuntos geodésicos acrescenta um novo conceito: o de momento. Agora, o importante a se saber é se todos os vértices do grafo foram incluídos em *I[S]*, sendo *S* o conjunto a verificar se é geodésico. O momento, representado pelo vetor de mesmo nome, nada mais é do que um par ordenado indicando em que árvore ele foi inserido em *I[S*] e de qual vértice ele é ancestral. Assim, sabemos em qual momento da análise dos ancestrais ele foi inserido e isto configura um certificado simples para o SIM (caso todos os vértices sejam incluídos em *I[S]*), pois aponta detalhadamente para o usuário em qual árvore e a partir de qual vértice ele pode encontrar a inclusão deste como ancestral, que equivale a ser incluído em *I[S]*. O Algoritmo Certificador para conjuntos geodésicos é apresentado como Algoritmo 6.

Algoritmo 6 Certificador para conjuntos geodésicos
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e conjunto S de vértices
Saída: Certificados para o SIM ou para o NÃO

1. PREPARAÇÃO(marcado, geodesico)
2. para cada v ∈ S **faça**
3. defina o vetor marcado de tamanho |V(G)|
4. **INICIALIZAÇÃO(marcado)**
5. **para cada** w ∈ S e w ≠ v **faça**
6. empilhe w em uma pilha P
7. **enquanto** P não estiver vazia **faça**
8. u ← topo da pilha P
9. remova u de P
10. marcado[u] ← 1
11. **TESTE_PAI(geodesico, P)**
12. **TESTE_TIO(geodesico, P)**
13. APRESENTAÇÃO_CERTIFICADO (geodésico, momento, pai, tio)

Para o certificado NÃO, optamos como alternativa a exibição das árvores que visualmente demonstram que determinado vértice que não entrou para o intervalo fechado não participa de nenhum caminho mínimo. Tal certificado pode ser facilmente avaliado por um verificador que buscaria pelo tal vértice não incluso em *I[S]* e faria uma análise descendente nas árvores, por meio das arestas-pai e arestas-tio, verificando que abaixo dele não há nenhum vértice de *S* que o faça ser adicionado em *I[S].*

O procedimento de preparação apenas inicializa os vetores que guardarão para cada vértice seus “momentos” (caso entrem para *I[S]*) e se eles estão em *I[S]* (vetor *geodésico*).

Procedimento PREPARAÇÃO (vetor marcado, vetor geodésico):

1. para cada v ∈ V(G) **faça**
2. geodésico[v] ← 0
3. momento[v] ← λ

O procedimento de inicialização tem a mesma tarefa que tem no Algoritmo 5: inicializar para cada vértice o valor de *marcado[v]* como 0 a cada árvore avaliada.

O procedimento de teste para os pais avalia quem são os pais dos vértices pertencentes ao conjunto *S* e de seus ancestrais.

Procedimento TESTE_PAI(vértice v, vértice u, vetor pai, vetor geodésico, vetor momento, pilha P):

```
1. se pai[v][u] ≠ -1 então
2.       z ← pai[v][u]
3.             se marcado[z] = 0 então
4.                   empilhe z em P
5.                   se geodésico[z] = 0 então
6.                         momento[z] ← (v, u)
7.                         geodésico[z] ← 1
```

Repare que o vértice é colocado na pilha para análise posterior de seus ancestrais caso não esteja marcado. Além disso, se este não estiver marcado, assinala-se o seu valor de momento como o par ordenado contendo a árvore onde foi incluído em *I[S]* e o vértice que o fez ser adicionado ao conjunto (vértice *u*). Além disso, é marcado no vetor geodésico que este foi incluído (valor 1 atribuído a posição correspondente no vetor geodésico).

A análise de tios é semelhante à análise de pais e é feita pelo procedimento *TESTE_TIO*.

Procedimento TESTE_TIO(vértice v, vértice u, vetor tio, vetor geodésico, vetor momento, pilha P):

```
1.se tio[v][u] ≠ ∅ então
2.       para cada z ∈ tio[v][u] faça
3.             se marcado[z] = 0 então
4.                   empilhe z em P
5.             se geodésico[z] = 0 então
6.                   momento[z] ← (v, u)
7.                   geodésico[z] ← 1
```

Após as análises terem sido feitas, o procedimento *APRESENTAÇÃO_CERTIFICADO* mostra se o conjunto é ou não geodésico, além do certificado correspondente.

Não é descartada a possibilidade de haver uma característica entre tais vértices que nunca são inclusos, tanto no caso geodésico, quanto no caso envoltório, que será vista na seção 4.2.5.

Procedimento APRESENTAÇÃO_CERTIFICADO (vetor geodésico, vetor momento, vetor pai, vetor tio):

1.se geodésico[v] = 1 para todo v ∈ V(G) **então**
2. exiba momento[v] para cada vértice como certificado para SIM
3.senão
4. apresente todas as árvores por meio dos vetores pai e tio – certificado para NÃO

Foram detectados casos particulares, como por exemplo, em uma clique, se um dos vértices não está incluso em *S*, jamais entrará em *I[S]* ou $I^k[S]$, pois há vértices diretos entre todos os outros (caminhos mínimos de comprimento 1). A versão completa do algoritmo pode ser encontrada no Apêndice B.

4.2.5 Algoritmo certificador para conjuntos envoltórios

Para conjuntos envoltórios, precisamos expandir a ideia de momento utilizada no Algoritmo 6, pois no caso destes, várias iterações de cálculos de intervalos fechados são feitas. Então, para tornarmos mais preciso o instante de entrada de um vértice no conjunto final, indicamos em que iteração *k*, ou seja, em qual $I^k[S]$ ele foi incluído primeiramente, além do vértice que o fez ser incluído e a árvore na qual ocorreu a inclusão. O algoritmo certificador para conjuntos envoltórios é o Algoritmo 7.

Outro cuidado tomado no Algoritmo Certificador para conjuntos envoltórios foi com relação ao retrabalho de verificar diversas vezes uma árvore de largura de algum vértice. O vetor "*árvoreVerificada*" faz o controle de quais vértices já tiveram suas árvores analisadas pelo processo de inclusão de ancestrais, pois avaliar novamente os ancestrais de um vértice numa mesma árvore não trará nada de novo ao algoritmo.

O conjunto "*ConjuntoEnvóltorio*" representa o $I^k[S]$ para a iteração *k* atual (controlado pela variável *k*). A cada iteração onde pelo menos um vértice é adicionado ao "*conjuntoEnvoltório*", haverá uma outra iteração, assim como sugere a definição. O que queremos dizer é que enquanto *conjuntoEnvoltório* ≠ conjuntoAnterior isto é, $I[I^{k-1}[S]] \neq I^{k-1}[S]$, o processo continua, mas agora avaliando as árvores dos vértices incluídos.

Algoritmo 7 Certificador para conjuntos envoltórios
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e conjunto S de vértices
Saída: Certificados para o SIM ou para o NÃO

1. defina os vetores envoltório, momento e árvoreVerificada de tamanho |V(G)| e uma variável inteira k.
2. PREPARACAO (k, envoltório, momento, árvoreVerificada, conjuntoEnvoltório, conjuntoAnterior)
2. enquanto conjuntoEnvoltorio ≠ conjuntoAnterior **faça**
3. conjuntoAnterior ← conjuntoEnvoltorio
4. **para cada** v ∈ conjuntoEnvoltorio **faça**
5. **se** arvoreVerificada[v] = 0 **então**
6. arvoreVerificada[v] ← 1
7. defina o vetor marcado de tamanho |V(G)|
8. **INICIALIZAÇÃO(marcado)**
9. **para cada** w ∈ conjuntoEnvoltorio e w ≠ v **faça**
10. empilhe w numa pilha P
11. **enquanto** P não estiver vazia **faça**
12. u ← topo de P
13. remova u de P
14. marcado[u] ← 1
15. **TESTE_PAI(k, v, u, pai, envoltório, momento, P, conjuntoEnvoltório)**
16. **TESTE_TIO(k, v, u, tio, envoltório, momento, P, conjuntoEnvoltório)**
17. k ← k+1
18. APRESENTAÇÃO CERTIFICADO (conjuntoEnvoltório, momento, pai, tio)

A primeira parte do algoritmo é a de preparação, onde inicializamos vetores e conjuntos. O "*conjuntoEnvoltório*" inicialmente já conta com todos os elementos de *S*. Pelo menos as árvores dos elementos de *S* passarão pela análise de ancestrais.

Procedimento PREPARAÇÃO (inteiro k, vetor envoltório, vetor momento, vetor árvoreVerificada, conjunto conjuntoEnvoltório, conjunto conjuntoAnterior):

1. para cada v ∈ V(G) **faça**
2. **se** v ∈ S **então**
3. envoltório[v] ← 1
4. momento[v] ← (0, v, v)
5. **senão**
6. envoltorio[v] ← 0
7. árvoreVerificada[v] ← 0
8. k ← 1
9. conjuntoEnvoltório ← S
10. conjuntoAnterior ← ∅

A cada iteração nova, o "*conjuntoEnvoltório*" passa a ser o "*conjuntoAnterior*", como maneira de controlar se algum novo vértice foi ou não incluído e se $I[I^{k-1}[S]] \neq I^{k-1}[S]$.

O procedimento de inicialização é idêntico ao dos Algoritmos 5 e 6.

Para as análises de ancestrais (*TESTE_PAI* e *TESTE_TIO*), a ideia de momento agora envolve o fator *k*, a iteração onde tal vértice foi incluído. Como nos Algoritmos 5 e 6, vértices com valor de marcado como 0 são postos na pilha para análise posterior. Os procedimentos *TESTE_PAI* e *TESTE_TIO* também marcam se o vértice está no conjunto envoltório.

```
Procedimento TESTE_PAI(inteiro k, vértice v, vértice u, vetor pai, vetor envoltório,
vetor momento, pilha P, conjunto conjuntoEnvoltório):

1.se pai[v][u] ≠ -1 então
2.      z ← pai[v][u]
3.      se envoltorio[z] = 0 então
4.              momento[z] ← (k, v, u)
5.              envoltório[z] ← 1
6.              adicionar z a conjuntoEnvoltório
7.              se marcado[z] = 1 então
8.                      empilhe z em P
```

```
Procedimento TESTE_TIO(inteiro k, vértice v, vértice u, vetor tio, vetor envoltório,
vetor momento, pilha P, conjunto conjuntoEnvoltório):

1.se tio[v][u] ≠ ∅  então
2.      para cada z ∈ tio[v][u] faça
3.              se envoltorio[z] = 0 então
4.                      momento[z] ← (k, v, u)
5.                      envoltorio[z] ← 1
6.                      adicionar z a conjuntoEnvoltório
7.                      se marcado[z] = 0 então
8.                              empilhe z em P
```

Como certificado para o SIM, mantivemos o momento, porém agora como uma tripla *(k, v, u)* onde *k* é a iteração onde ele foi incluído, *v* é o vértice em qual árvore ele foi incluído no "*conjuntoEnvoltório*" e *u* é o vértice do qual ele é ancestral. Os três valores indicam exatamente quando e como o vértice foi incluído.

O certificado para o NÃO são as próprias árvores em si, onde um verificador pode fazer a análise descendente a partir dos vértices não inclusos em

conjuntoEnvoltório, verificando que não há vértice do *conjuntoEnvoltório* final do qual ele seja ancestral.

Ao final do algoritmo, há a apresentação desses certificados por meio do teste que verifica se todos os vértices de *G* entraram no conjunto envoltório.

Procedimento APRESENTAÇÃO_CERTIFICADO (conjunto conjuntoEnvoltório, vetor momento, vetor pai, vetor tio):

1.se conjuntoEnvoltorio = S então
2. exiba momento[z] para todo z ∈ V(G)
3.senão
4. exiba todas as árvores por meio dos vetores pai e tio

A versão completa do algoritmo pode ser encontrada no Apêndice C.

5 **ALGORITMOS CERTIFICADORES PARA CONVEXIDADE MONOFÔNICA**

5.1 **Convexidade monofônica**

Dado um caminho entre dois vértices *u* e *v*, dizemos que ele é *induzido* se não há arestas entre vértices não consecutivos no caminho. Um caminho é induzido se ele é um subgrafo induzido de *G*. Na Figura 7, o caminho (1, 3, 5, 6) corresponde a um caminho induzido no grafo, enquanto o caminho (1, 2, 3, 5, 6) não corresponde a um caminho induzido, pois há uma aresta entre 1 e 3 que são vértices não consecutivos no caminho.

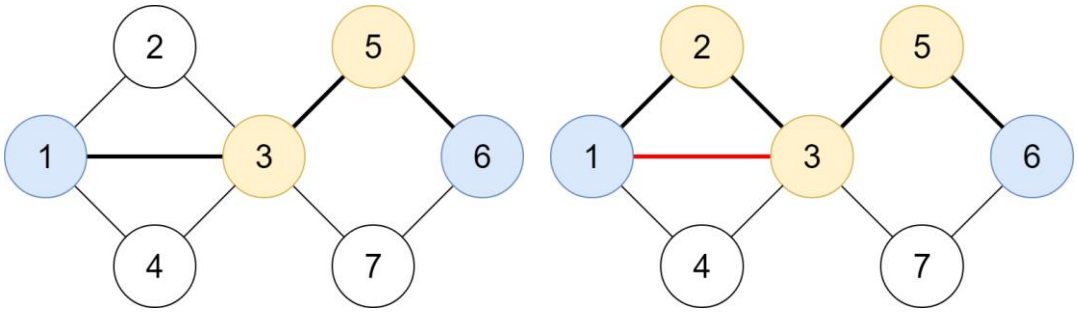

Figura 7 - Um exemplo de caminho induzido (à esquerda) e um não induzido (à direita)

A convexidade definida por meio de caminhos induzidos de um grafo é conhecida como *convexidade monofônica*. Cada conceito definido para convexidade geodésica tem um semelhante para convexidade monofônica.

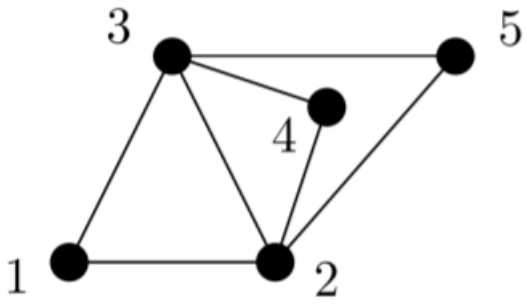

Figura 8 - Grafo simples. Reproduzida de (ESPERET, Louis; LEMOINE, Laetitia; MAFFRAY, Frédéric. – 2017).

O *intervalo monofônico J[u, v]* para dois vértices *u, v* $\in V(G)$ é o conjunto de todos os vértices que participam de pelo menos um caminho induzido entre u e v. Para um conjunto $X \subseteq V(G)$, o *fecho monofônico J[X]* é a união dos intervalos

fechados $J[u, v]$ para todo par de vértices $u, v \in X$. Na Figura 8, por exemplo, $J[1,5] = \{1, 2, 3, 5\}$ e com $X = \{1,2,4\}$, temos que $J[X] = \{1, 2, 3, 4\}$.

Caso $J[X] = X$, dizemos que o conjunto X é *monofonicamente convexo* ou *m-convexo*. É fácil notar que $X = \{v\}$ e $X = V(G)$ são monofonicamente convexos.

O menor conjunto convexo contendo X é denotado por $J_h[X]$. Tal conjunto é chamado de conjunto m-envoltório convexo de X. É fato que $X \subseteq J[X] \subseteq J_h[X] \subseteq V(G)$. Um subconjunto de vértices $X \subseteq V(G)$ é dito *monofônico* se $J[X] = V(G)$ e *m-envoltório* se $J_h[X] = V(G)$. Na Figura 8, $X = \{1, 4, 5\}$ é monofônico e m-envoltório, pois $J[X] = J_h[X] = V(G)$.

Um *vértice* é denominado *simplicial* se sua a vizinhança induz uma clique. É fato que todo conjunto monofônico ou m-envoltório de G deve conter todos os vértices simpliciais de G. (DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L., 2010)

Existem alguns teoremas importantes em torno da convexidade monofônica que precisamos citar. O primeiro deles traz uma maneira de verificar se um conjunto é m-convexo sem precisar calcular todos os caminhos induzidos entre todos os pares de vértices.

Teorema 1 (DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L., 2010): Seja $G = (V, E)$ um grafo. Um subconjunto $X \subseteq V(G)$ é m-convexo se e se somente se para todo par de vértices não adjacentes u e $v \in X$ e toda componente conexa C de $G-X$, $V(C) \cap N(v) = \emptyset$ ou $V(C) \cap N(u) = \emptyset$.

Demonstração: Assuma em primeiro caso que X é m-convexo. A existência de um par de vértices não adjacentes $u, v \in X$ e uma componente conexa C de $G-X$ contendo pelo menos um par u', v' e que $u' \in V(C) \cap N(u)$ e $v' \in V(C) \cap N(v)$ implica na existência de uma sequência de *vértices* $w_0 = u, w_1 = u', w_2, ..., w_{k-1}, w_k = v', w_{k+1} = v$ com $k \geq 1$ tal que:

(i) $w_i \notin X$ para algum $i \in \{1, ..., k\};$ ou

(ii) ou $u' = v'$ ou $(w_i, w_{i+1}) \in E(G-X), 1 \leq i \leq k$

Assim, existe um caminho induzido unindo u a v e contendo pelo menos um vértice fora de X, uma contradição.

Assuma agora que X não é m-convexo. Seja $w_0 = u$, $w_1 = u'$, w_2, ..., w_{k-1}, $w_k = v'$, $w_{k+1} = v$ um caminho induzido unindo um par de vértices $u, v \in X$ e que $k \geq 1$ e $w_i \notin X$ para algum $i \in \{1, \ldots, k\}$. Seja j um índice tal que $w_{j-1} \in X$ e $w_j, w_{j+1}, \ldots, w_i \in V(G) \setminus X$ (Tal índice existe, desde que $u \in X$). Analogamente, seja l um índice tal que $w_i, w_{i+1}, \ldots, w_l \in V(G) \setminus X$ e $w_{l+1} \in X$. Isto implica que w_{j-1}, w_{l+1} é um par de vértices não adjacentes em X e há uma componente conexa C de G-X tal que $V(C) \cap N(w_{j-1}) \neq \emptyset$ e $V(C) \cap N(w_{l+1}) \neq \emptyset$. ■

Tal teorema se mostra uma maneira poderosa de se verificar se um dado conjunto é m-convexo. Esse teorema se torna ainda mais importante pelo exposto no Teorema 2.

Teorema 2 (HAAS, Robert; HOFFMANN, Michael., 2006): Sejam u, v e w três vértices distintos de um grafo G. Decidir se $w \in J[u, v]$ é NP-Completo.

A demonstração deste teorema pode ser encontrada em (HAAS, Robert; HOFFMANN, Michael, 2006). O que esse teorema mostra é que existe um desafio ainda maior em torno dos conjuntos m-convexos, pois não se pode simplesmente verificar para todo vértice se esse pertence a $J[X]$. Assim, o Teorema 1 se mostra como uma excelente base para um algoritmo que decide se um conjunto X de vértices é m-convexo.

O Algoritmo 8 apresenta uma versão simplificada do algoritmo convencional para verificar se um dado conjunto é m-convexo.

Algoritmo 8 Algoritmo para conjuntos m-convexos.
Entrada: Grafo G e conjunto X de vértices para o qual se quer testar a m-convexidade.
Saída: SIM (é m-convexo) ou NÃO (não é m-convexo).

1. Computar todas as componentes conexas de G - X: $C_1, \ldots, C_n$.
2. Defina conjuntos C'_i inicialmente vazios para cada uma das componentes encontradas no passo 1.
3. para cada aresta (u,v) $\in$ E(G), tal que u $\in$ X e w $\notin$ X **faça**
4. Inclua u em C'_i se w $\in C_i$
5. **se** C'_i possui dois vértices não-adjacentes **então**
6. X não é m-convexo. **Responda NÃO**
7. **pare**
8. se o processo finaliza e $\forall C'_i$, G[V(C'_i)] é um grafo completo **então**
9. X é m-convexo. **Responda SIM**

5.2 **Algoritmo certificador para conjuntos m-convexos**

Nossos algoritmos basearam-se fortemente no Teorema 1 para a certificação de conjuntos m-convexos. Apresentaremos agora o algoritmo certificador para conjuntos m-convexos o Algoritmo 9.

O grafo induzido *H* (linha 15) é utilizado para traçarmos um caminho induzido entre os vértices "problemáticos" encontrados na componente. Sabe-se que todo caminho mínimo é um caminho induzido e dado o fato que $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, isto é, $H \subseteq G$, então se existir um caminho induzido em *H*, também existe este mesmo caminho em *G* (Linhas 15 a 17). Tal fato é usado para certificar o NÃO, onde é necessário apresentar um ou mais caminhos induzidos entre dois vértices de *X* que passam por outros não pertencentes a *X* como certificado.

Algoritmo 9 Algoritmo certificador para conjuntos m-convexos.
Entrada: Grafo G e conjunto X de vértices para o qual se quer testar a m-convexidade.
Saída: SIM (é m-convexo) ou NÃO (não é m-convexo) e um certificado.

1. Gere G' = G – X
2. Gere todas as componentes conexas de G', ditas C_1, …, C_k
3. para cada componente C_i **faça**
4. Gere $C_i' \leftarrow \emptyset$
5. para cada aresta (u,v) $\in$ E(G) **faça**
6. **se** u $\in$ X e v $\notin$ X **então**
7. **para cada** C_i **faça**
8. **se** v $\in$ C_i **então**
9. inclua u em C_i'
10. para cada C_i' **faça**
11. gere $G[V(C_i')]$
12. **para cada** u $\in$ $V(C_i')$ **faça**
13. **para cada** v $\in$ $V(C_i')$ e u $\neq$ v **faça**
14. **se** u e v não são adjacentes **então**
15. Gere H = $G[V(C_i) \cup \{u,v\}]$
16. P $\leftarrow$ qualquer caminho mínimo entre u e v em H
17. **Apresente P como certificado para o NÃO**
18. **pare**
19. para cada C_i' **faça**
20. gere $G[V(C_i')]$
21. **Apresente os grafos completos $G[V(C'_i)]$ como certificado para o SIM.**

Como certificado para o SIM, atestamos que $G[V(C_i')]$ é um grafo completo para todo C_i' gerado. Esta certificação é inteiramente baseada no Teorema 1, que

garante que tal fato comprova a m-convexidade. Repare que o Teorema 1 se torna essencial para que o algoritmo certifique o SIM.

5.3 **Algoritmo certificador para conjuntos monofônicos**

A teoria de convexidade monofônica apresenta certos detalhes bastante curiosos. Decidir se um conjunto é m-convexo é um problema que se pode resolver em tempo polinomial, a saber $O(nm)$, onde $n = |V(G)|$ e $m = |E(G)|$. Também é possível saber em tempo polinomial se um conjunto é m-envoltório, a saber, em $O(n^2m)$, onde $n = |V(G)|$ e $m = |E(G)|$. Ambos resultados sobre as complexidades citadas podem ser encontrados em (DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L, 2010).

Como apresentado no Teorema 2, dados 3 vértices $u, v, w,$ decidir se $w \in J[u,v]$ é NP-Completo (HAAS, Robert; HOFFMANN, Michael., 2006). Por conseguinte, o problema de descobrir se um conjunto X de vértices é monofônico também é NP-Completo. Tal resultado foi provado em (DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L, 2010) por uma redução do problema de decidir se w pertence a $J[u,v]$, dados os vértices u, v e w.

5.4 **Algoritmo certificador para conjuntos m-envoltórios**

Para decidirmos se um conjunto é m-envoltório, passamos pela mesma dificuldade apresentada no Teorema 2, porém esta é “vencida” neste caso.

Repare que, para um conjunto $X \subseteq V(G)$ ser m-envoltório, é necessário que seja possível achar caminhos induzidos entre pares de vértices deste ou dos posteriormente adicionados ao fecho até que todo conjunto de vértices seja incluído no fecho monofônico, isto é, algum $J^k[X] = V(G)$ para algum k.

Para que ele não seja m-envoltório, é necessário que X ou algum $J^k[X]$ seja convexo, isto é, nenhum vértice de fora de X ou $J^k[X]$ para algum k participa de algum caminho induzido entre pares de vértices destes.

Dada a dificuldade em saber se dado vértice w está contido em $J^k[X]$, conforme o que o Teorema 2 indica, será necessário fazer uso da ideia de que todo caminho mínimo é induzido.

Como feito no Algoritmo 9, graças ao resultado do Teorema 1, conseguimos identificar os vértices “problemáticos”: aqueles que não são adjacentes nos grafos induzidos pelas componentes criadas pelo próprio algoritmo. Além disso, podemos afirmar que há um caminho induzido entre eles que usa vértices de fora de X e que pelo menos o caminho mínimo entre esses vértices “problemáticos” é um caminho induzido entre eles.

A tática aplicada pelo Algoritmo 10 é criar um conjunto inicialmente com os elementos de $X \subseteq V(G)$, no caso o conjunto “*EnvoltorioAtual”*. A partir daí, verificar se este é convexo. Caso seja, não haveria como um vértice $v \notin X$ ser incluso ao fecho, portanto X não é m-envoltório. Caso não seja, tomamos os vértices “problemáticos”, digamos u, v pertencentes a algum C_i', e traçamos os caminhos mínimos entre eles no grafo induzido $H = G[V(C_i) \cup \{u,v\}]$. Tais caminhos, que no Algoritmo 9 eram certificados, agora servirão como base para incluir novos vértices ao conjunto “*EnvoltorioAtual”,* que continha inicialmente apenas os vértices de X. Todos os vértices que participam desses caminhos mínimos e não pertencem a *“EnvoltorioAtual”* serão inclusos neste conjunto. Toma-se esse novo conjunto “*EnvoltorioAtual”* como base para a próxima iteração e repete-se o processo descrito: verifica-se se este é m-convexo e assim por diante.

Repare que o algoritmo que segue esta tática deve parar em duas situações: o conjunto *“EnvoltorioAtual”* se torna convexo ou este já possui todos os vértices de G. Em ambos os casos, de uma iteração pra outra, *“EnvoltorioAtual”* não muda.

A tática descrita funciona pelos seguintes motivos:

a) não precisamos calcular $J^k[X]$ de forma direta;
b) se o conjunto X é m-envoltório, existirá pelo menos um caminho induzido entre pares vértices de X ou do conjunto “*EnvoltorioAtual”*, e este é o caminho mínimo que incluirá vértices que ainda não estão no conjunto;
c) se o conjunto não é m-envoltório, o conjunto “*EnvoltorioAtual”* gerado pelo algoritmo convergirá a $J_h[X]$, o menor conjunto convexo contendo X, pois, como a inclusão dos vértices é feita pelos caminhos mínimos, estamos incluindo o menor número de vértices possível nele.

O certificado para o SIM é construído com pelo menos um caminho induzido entre vértices do conjunto “*EnvoltorioAtual”* para cada $v \notin X$. Estes caminhos justificam a entrada desses vértices em algum $J^k[X]$.

O certificado para o NÃO é constituído pelo próprio conjunto "*EnvoltorioAtual",* que nesse caso é $J_h[X]$. Portanto, nenhum outro vértice poderia ser incluído a "*EnvoltorioAtual"*. O usuário de posse de tal conjunto pode inclusive executar o Algoritmo 9 usando o conjunto final gerado pelo algoritmo como entrada para certificar que este é realmente m-convexo. Portanto, podemos até mesmo afirmar que o Algoritmo 9 é um verificador para o Algoritmo 10.

Algoritmo 10 Algoritmo para conjuntos m-envoltórios.
Entrada: Grafo G e conjunto X de vértices para o qual se quer testar se é m-envoltório.
Saída: SIM (é m-envoltório) ou NÃO (não é m-envoltório) e um certificado.

1. EnvoltorioAnterior ← ∅
2. EnvoltorioAtual ← X
2. caminhos ← ∅
3. enquanto EnvoltorioAnterior ≠ EnvoltorioAtual **faça**
4. EnvoltorioAnterior ← EnvoltorioAtual
5. Gere G' = G – EnvoltorioAtual
6. Gere todas as componentes conexas de G', ditas C_1, …, C_k
7. **para cada** componente C_i **faça**
8. Gere C_i' ← ∅
9. **para cada** aresta (u,v) ∈ E(G) **faça**
10. **se** u ∈ EnvoltorioAtual e v ∉ EnvoltorioAtual **então**
11. **para cada** C_i **faça**
12. **se** v ∈ C_i **então**
13. inclua u em C_i'
14. **para cada** C_i' **faça**
15. gere G[V(C_i')]
16. **para cada** u ∈ V(C_i') **faça**
17. **para cada** v ∈ V(C_i') e u ≠ v **faça**
18. **se** u e v não são adjacentes **então**
19. Gere H = G[V(C_i) ∪ {u,v}]
20. P ← qualquer caminho mínimo entre u e v em H
21. EnvoltorioAtual ← EnvoltorioAtual ∪ V(P)
22. adicione P em caminhos
23.
24. **se** EnvoltorioAtual = V(G) **então**
25. **apresente cada P de caminhos como certificado para o SIM**
26. **senão**
27. **apresentar EnvoltorioAtual como conjunto m-convexo como certificado para o NÃO**

6 **CONCLUSÃO**

Gerar e manter *softwares* corretos é uma tarefa difícil, mas que deve ser muito bem executada para garantir saídas confiáveis para usuários. O trabalho em torno de teste de *software* e todo o estudo sobre Algoritmo Certificadores faz parte de um esforço em gerar cada vez mais programas corretos sem que se gaste grande quantidade de tempo corrigindo códigos ou os impactos causados por programas incorretos.

O nosso trabalho constitui mais um novo passo em direção a um futuro com menos programas errados e, por consequência, um futuro com menos usuários insatisfeitos. Além disso, queremos com esse trabalho incentivar um aprofundamento maior em torno deste assunto que ainda não é tão bem explorado dentro da Computação.

Criar mais algoritmos certificadores para os mais diversos domínios de estudo é uma tarefa difícil, mas que cada vez mais se torna mais necessária em meio a um futuro que se apresenta cada vez menos tolerante a falhas de *software*.

Neste trabalho, criamos vários algoritmos certificadores para problemas que antes não possuíam versão certificadora de seus algoritmos: para conjuntos convexos, geodésicos e envoltórios no caso da convexidade geodésica e para conjuntos m-convexos e m-envoltórios no caso da convexidade monofônica.

Como trabalhos futuros podemos criar versões certificadoras de outros algoritmos que resolvem problemas clássicos em grafos, além dos de convexidade ou até mesmo criar certificadores para outros tipos de convexidade. Uma outra possibilidade é simplificarmos ainda mais os certificados criados neste trabalho, de maneira a tornarmos mais facilmente entendíveis pelo usuário ou mais facilmente verificáveis por este.

REFERÊNCIAS

BONDY, J. A.; MURTY, U. S. R. **Graph theory** (2008). Grad. Texts in Math, 2008.

CORMEN, T. H., LEISERSON C. E.; RIVEST, R. L.; STEIN, C. . Algoritmos: teoria e prática. **Editora Campus**, 2002.

DOURADO, Mitre C.; PROTTI, Fábio; SZWARCFITER, Jayme L. Complexity results related to monophonic convexity. **Discrete Applied Mathematics**, v. 158, n. 12, p. 1268-1274, 2010.

HAAS, Robert; HOFFMANN, Michael. Chordless paths through three vertices. **Theoretical Computer Science**, v. 351, n. 3, p. 360-371, 2006.

KAPLAN, Haim; NUSSBAUM, Yahav. Certifying algorithms for recognizing proper circular-arc graphs and unit circular-arc graphs. **Discrete Applied Mathematics**, v. 157, n. 15, p. 3216-3230, 2009.

KRATSCH, D.; MCCONNELL, R. M.; MEHLHORN, K.; SPINRAD, J. P. .**Certifying algorithms for recognizing interval graphs and permutation graphs**. SIAM Journal on Computing, v. 36, n. 2, p. 326-353, 2006.

MCCONNELL, Ross M.; MEHLHORN, K.; NÄHER, S.; SCHWEITZER, P. Certifying algorithms. **Computer Science Review**, v. 5, n. 2, p. 119 161, 2011.

PAPADIMITRIOU C. H.; DASGUPTA, C. H.; VAZIRANI, Umesh V. **Algorithms**. 2015.

PELAYO, Ignacio M. **Geodesic convexity in graphs**. New York: Springer, 2013.

APÊNDICE A – ALGORITMO CERTIFICADOR PARA CONJUNTOS CONVEXOS

Algoritmo Certificador para convexidade
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e conjunto S de vértices
Saída: Certificados para o SIM ou para o NÃO

```
1. para cada v ∈ S faça
2.     para todo w ∈ V(G) faça
3.         marcado[w] ← 0
4.     para cada w ∈ S e w ≠ v faça
5.         empilhe w em uma pilha P
6.         enquanto P não estiver vazia faça
7.             u ← topo de P
8.             remova u de P
9.             marcado[u]← 1
10.            se pai[v][u] ≠ -1 então
11.                se pai[v][u] ∉ S então
12.                    exiba caminho v → pai[v][u] → w como certificado para o NÃO
13.                    pare
14.                senão
15.                    se marcado[pai[v][u]] = 0 então
16.                        empilhe pai[v][u] em P
17.            se tio[v][u] ≠∅ então
18.                se algum x ∈ tio[v][u] não pertence a S então
19.                    exiba caminho v → x → w como certificado para o NÃO
20.                    pare
21.                senão
22.                    para cada t ∈ tio[v][u] faça.
23.                        se marcado[t] = 0 então
24.                            empilhe t em P
25. exiba lista de pais e tios para certificar o SIM
```

APÊNDICE B – ALGORITMO CERTIFICADOR PARA CONJUNTOS GEODÉSICOS

Algoritmo Certificador para conjuntos geodésicos
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e conjunto S de vértices
Saída: Certificados para o SIM ou para o NÃO

```
1. para cada v ∈ V(G) faça
2.      geodésico[v] ← 0
3.      momento[v] ← λ
4. para cada v ∈ S faça
5.      para cada w ∈ V(G) faça
6.              marcado[w] ← 0
7.      para cada w ∈ S e w ≠ v faça
8.              empilhe w em uma pilha P
9.              enquanto P não estiver vazia faça
10.                     u ← topo da pilha P
11.                     remova u de P
12.                     marcado[u] ← 1
13.                     se pai[v][u] ≠ -1 então
14.                             z ← pai[v][u]
15.                             se marcado[z] = 0 então
16.                                     empilhe z em P
17.                             se geodesico[z] = 0 então
18.                                     momento[z] ← (v, u)
19.                                     geodesico[z] ← 1
20.                     se tio[v][u] ≠ ∅ então
21.                             para cada z ∈ tio[v][u] faça
22.                                     se marcado[z] = 0 então
23.                                             empilhe z em P
24.                                     se geodesico[z] = 0 então
25.                                             momento[z] ← (v, u)
26.                                             geodesico[z] ← 1
27.     se geodesico[v] = 1 para todo v ∈ V(G) então
28.             exiba momento[v] para cada vértice como certificado para SIM
29.     senão
30.             apresente todas as árvores – certificado para NÃO
```

APÊNDICE C – ALGORITMO CERTIFICADOR PARA CONJUNTOS ENVOLTÓRIOS

Algoritmo Certificador para conjuntos envoltórios
Entrada: Grafo G, Vetores pai e tio construídos pelo Algoritmo 4 e e conjunto S de vértices
Saída: Certificados para o SIM ou para o NÃO

```
1. para cada v ∈ V(G) faça
2.      se v ∈ S então
3.              envoltorio[v] ← 1
4.              momento[v] ← (0, v, v)
5.      senão
6.              envoltorio[v] ← 0
7.              arvoreVerificada[v] ← 0
8. k ← 1
9. conjuntoEnvoltorio ← S
10. conjuntoAnterios ← ∅
11. enquanto conjuntoEnvoltorio ≠ conjuntoAnterior faça
12.     conjuntoAnterior ← conjuntoEnvoltorio
13.     para cada v ∈ conjuntoEnvoltorio faça
14.             se arvoreVerificada[v] = 0 então
15.                     arvoreVerificada[v] ← 1
16.                     para cada w ∈ V(G) faça
17.                             marcado[w] ← 0
18.                     para cada w ∈ conjuntoEnvoltorio e w ≠ v faça
19.                             empilhe w numa pilha P
20.                             enquanto P não estiver vazia faça
21.                                     u ← topo de P
22.                                     remova u de P
23.                                     marcado[u] ← 1
24.                                     se pai[v][u] ≠ -1 então
25.                                             z ← pai[v][u]
26.                                             se envoltorio[z] = 0 então
27.                                                     momento[z] ← (k, v, u)
28.                                                     envoltorio[z] ← 1
29.                                                     adicionar z a conjuntoEnvoltorio
30.                                                     se marcado[z] = 1 então
31.                                                             empilhe z em P
32.                                             se tio[v][u] ≠ 0 então
33.                                                     para cada z ∈ tio[v][u] faça
34.                                                             se envoltorio[z] = 0 então
35.                                                                     momento[z] ← (k, v, u)
36.                                                                     envoltorio[z] ← 1
37.                                                                     adicionar z a
conjuntoEnvoltorio
38.                                                             se marcado[z] = 0 então
39.                                                                     empilhe z em P
40.     k ← k+1
41.se conjuntoEnvoltorio = S então
42.     exiba momento[z] para todo z ∈ V(G)
43.senão
44.     exiba todas as árvores
```