UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

LUCAS RODRIGUES KEILER

ESTUDO ALGORÍTMICO DE PROBLEMAS DE INFECÇÃO EM GRAFOS

FORTALEZA

2019

LUCAS RODRIGUES KEILER

ESTUDO ALGORÍTMICO DE PROBLEMAS DE INFECÇÃO EM GRAFOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Rudini Menezes Sampaio

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

K36e Keiler, Lucas Rodrigues.
ESTUDO ALGORÍTMICO DE PROBLEMAS DE INFECÇÃO EM GRAFOS / Lucas Rodrigues Keiler. – 2019.
66 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia de Computação, Fortaleza, 2019.
Orientação: Prof. Dr. Rudini Menezes Sampaio.

1. Infecção em Grafos. 2. Algorítmos. 3. Percolação. I. Título.

CDD 621.39

LUCAS RODRIGUES KEILER

ESTUDO ALGORÍTMICO DE PROBLEMAS DE INFECÇÃO EM GRAFOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Rudini Menezes Sampaio (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Pablo Mayckon Silva Farias
Universidade Federal do Ceará (UFC)

Prof. Dr. Carlos Estêvão Rolim Fernandes
Universidade Federal do Ceará (UFC)

Prof. Dr. Jorge Herbert Soares de Lira
Universidade Federal do Ceará (UFC)

A família e amigos, por tornarem tudo possível.

AGRADECIMENTOS

A meus pais, Adriana e Friedrich, por estarem sempre ao meu lado.

Ao meu irmão Felipe, pela amizade e por compartilhar o gosto pela computação.

Ao professor Rudini Menezes Sampaio, pela orientação, paciência e conselhos durante a elaboração deste trabalho, além de ótimas aulas durante o curso.

A todos os amigos, dentro e fora da Universidade, por facilitarem esta jornada.

A todos os integrantes do Laboratório de Sistemas e Banco de Dados (LSBD), pela cordialidade e amizade. Agradeço especialmente ao professor Javam Machado e a professora Rosélia Machado, pela excelente oportunidade de estágio e pelo apoio e compreensão no desenvolvimento desta tese.

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

Por fim, a todos os professores e funcionários da Universidade Federal do Ceará. Agradeço especialmente ao professor Pablo Mayckon por inúmeras conversas sobre os mais diferentes tópicos e ao professor Jorge Herbert pela orientação durante meu período como bolsista de iniciação científica e por solidificar minha paixão pela matemática.

“Viva como se fosse morrer amanhã. Aprenda como se fosse viver para sempre.”

(Mahatma Gandhi)

RESUMO

Este trabalho tem como foco o estudo de alguns problemas de infecção em grafos de um ponto de vista prático. Os problemas analisados possuem algoritmos polinomiais ou pseudo-polinomiais propostos, mas não possuem implementações destes algoritmos. Assim, desenvolve-se implementações em linguagem `C++` para todos os problemas estudados. Estratégias de teste para as implementações também foram estruturadas, de modo que garantam a validade dos códigos resultantes. A partir das verificações realizadas, apresenta-se implementações que resolvem os problemas com a mesma complexidade proposta.

Palavras-chave: Infecção em Grafos. Algoritmos. Percolação

ABSTRACT

This work focuses on the study of specific infection related problems on graphs from a practical standpoint. Every analysed problem has polynomial or pseudo-polynomial algorithms, however, no implementation of this algorithms are available. Therefore the aim of this study is to develop C++ implementations of every analysed problem. To guarantee the correctness of every implementation, a test process was also developed. The achieved results are implementations that not only solve the problems correctly, but also maintain the expected computational complexity.

Keywords: Infection on Graphs. Algorithms. Percolation

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

$d(v)$	Grau de um vértice
$\Delta(G)$	Maior grau de um vértice no grafo G
$N(v)$	Vizinhança do vértice v
$N_2(v)$	Vizinhança a distância 2 de v
$N_{\geq n}(v)$	Vizinhança a distância maior ou igual a n (vértices a distância $\geq n$ de v)

SUMÁRIO

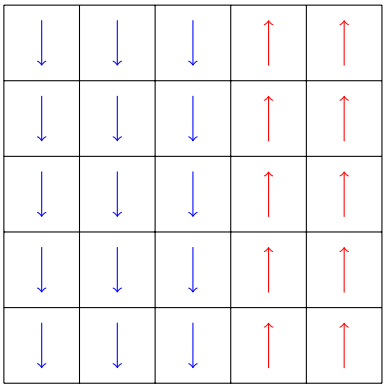
1 INTRODUÇÃO

O presente trabalho tem como foco problemas de infecção em grafos. Esta classe de problemas é bastante abrangente e não só restrita à Teoria dos Grafos. Na verdade, possui aplicações em quase todas as áreas da ciência. A seguir, apresenta-se um breve histórico do problema, bem como os objetivos deste trabalho.

1.1 Histórico

Uma primeira aparição de infecção em grafos ocorreu em mecânica estatística no início do século XX, no hoje denominado *Modelo de Ising*, que descreve o ferromagnetismo na matéria. Este modelo foi inicialmente proposto pelo físico alemão Wilhelm Lenz em 1920 e depois passado para seu orientando Ernst Ising, que desenvolveu a formulação matemática para o modelo em uma dimensão em 1924. De maneira breve, este modelo esquematiza a estrutura da matéria em *grids* $d-$dimensionais. Cada átomo é representado por um vértice no *grid* e possui dois valores possíveis de spin: -1 e 1. Cada átomo é influenciado por seus vizinhos mais próximos e a mudança de estrutura do *grid* leva em conta a diferença de energia associada a cada estado (NISS, 2005).

Figura 1 – Estrutura do Modelo de Ising para 2 dimensões



Fonte: o autor.

Embora o *Modelo de Ising* não seja formalmente definido como grafo, certamente sua estrutura é familiar ao atual estudo de problemas de infecção: dois estados associados a vértices de um *grid* com alguma regra de mudança entre esses estados.

Saindo do domínio da Física Teórica, encontra-se mais uma vez estudos relacionados a problemas de infecção no trabalho de John von Neumann em autômatos celulares. Em 1948, John von Neumann trabalhava com a ideia de autorreplicação de máquinas. Seguindo

uma sugestão do então colega no Laboratório de Los Alamos Stanislaw Ulam, von Neumann estruturou seu trabalho com o arcabouço de autômatos celulares (SCHIFF, 2008).

Autômatos celulares são comumente estruturados como *grids* d-dimensionais de elementos discretos, denominados células. Cada célula possui $k \geq 2$ estados possíveis. Mudanças de estado ocorrem em momentos discretos no tempo seguindo uma função local de transição de estados. Esta função leva em consideração o estado atual da célula e de sua vizinhança (SCHIFF, 2008). Assim, vê-se a estrutura característica de infecção em grafos mais uma vez.

De maneira menos detalhada, trabalhos com infecção em grafos continuam a aparecer em estudos de várias áreas da ciência durante o século XX, como em difusão de fluídos (BROADBENT; HAMMERSLEY, 1957), infectologia (GRASSBERGER, 1983) e física de estado sólido (CHALUPA *et al.*, 1979).

1.2 Objetivos

Motivado pelo número de aplicações que problemas de infecção possuem em diferentes áreas, este trabalho tem como objetivo estudar problemas específicos de infecção em grafos sobre o ponto de vista algorítmico. Assim, propõe-se implementações em linguagem `C++` dos problemas analisados.

A apresentação de implementações, por sua vez, criam a necessidade de verificações da validade dos códigos desenvolvidos. Assim, são objetivos secundários os testes das implementações, de modo a garantir a correta resolução dos problemas propostos.

2 FUNDAMENTAÇÃO TEÓRICA

A base de conhecimento na qual o presente trabalho se apoia é constituída de conhecimentos básicos em Teoria dos Grafos e, mais especificamente, sobre problemas de infecção em grafos. Apresenta-se a seguir um panorama geral sobre infecção em grafos e alguns resultados sobre os quais os algoritmos implementados neste trabalho são baseados.

2.1 Infecção em Grafos

Uma definição mínima de problema de infecção em grafos encontra-se a seguir.

Definição 2.1.1. *Seja $G = (V, E)$ um grafo qualquer. Define-se dois estados possíveis para cada vértice $v \in V(G)$: infectado ou sadio. Mudando de problema a problema, define-se uma regra de infecção, ou seja, as condições pelas quais um vértice deve passar para mudar entre os estados. Este processo de mudança de estado ocorre em turnos iterados. Um problema de infecção, por fim, estuda características do processo de infecção em um grafo, concentrando-se, normalmente, em alguma métrica específica.*

A definição acima é propositalmente genérica, pois os problemas de infecção, embora agrupados pelo nome e pelas características mencionadas, são muito diversificados. A regra de infecção, por exemplo, pode ser determinística (um vértice se torna infectado caso um número específico ou a maioria dos vizinhos estiverem infectados) ou não (cada vértice tem probabilidade P de ser infectado). A depender do problema, um vértice pode ou não ir do estado *infectado* para o *sadio*.

Essa multiplicidade de abordagens persiste nos objetivos de cada problema de infecção. Estes podem restringir o tipo de grafo a ser estudado e tentar maximizar (ou minimizar) alguma métrica. Por exemplo, pode-se estudar a probabilidade de infecção geral no grafo (BERGER *et al.*, 2005), o tempo máximo de infecção em determinados grafos (MARCILON; SAMPAIO, 2018a), o conjunto mínimo de vértices para infectar um grafo (CHEN, 2009) ou mesmo a influência que a estrutura do grafo tem no processo de infecção (Ganesh *et al.*, 2005).

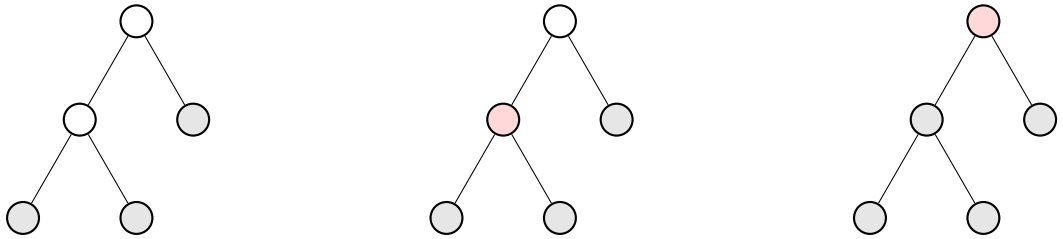
No presente trabalho, entretanto, o foco dos estudos encontra-se em apenas dois tipos de problema de infecção. A seguir uma breve definição de ambos:

- **Tempo máximo de infecção em *2-bootstrap percolation***: Neste problema, a regra de infecção é determinística e a definição segue: um vértice sadio é infectado no tempo atual

se dois vértices vizinhos foram infectados em tempos anteriores. Um vértice infectado permanece infectado para sempre. O objetivo do problema é, dado um grafo G, estudar o tempo máximo de infecção $t(G)$, supondo a regra de infecção mencionada. Um exemplo encontra-se na figura 2, em que $t(G) = 2$.

- ***Target Set Selection***: Aqui, a regra de infecção é parecida, mas um pouco mais relaxada. Cada vértice $v \in V(G)$ possui um valor limite $t(v)$, tal que, se $t(v)$ vizinhos de v estiverem infectados em determinado momento, v será infectado no tempo seguinte. Mais uma vez, um vértice infectado fica infectado para sempre. O objetivo aqui é, dado um grafo G e valores $t(v)$ para cada $v \in V(G)$, descobrir a cardinalidade mínima do conjunto inicial de vértices infectados para que o grafo todo seja infectado.

Figura 2 – Exemplo de infecção



(a) Tempo 0 (b) Tempo 1 (c) Tempo 2

Os algoritmos estudados neste trabalho dependem ainda de lemas e teoremas mais específicos. A seguir, encontram-se seções com a definição de cada problema estudado, bem como os resultados base que sustentam a corretude de cada algoritmo.

2.1.1 Problema de decisão $t(G) \geq k$ para grafos com $\Delta \leq 3$

Neste primeiro problema, limita-se os grafos estudados àqueles de grau máximo menor ou igual a três. Assim, para um grafo G com $\Delta(G) \leq 3$ e um inteiro $k > 0$, deseja-se saber se $t(G) \geq k$, ou seja, se o maior tempo de infecção possível para G é maior ou igual ao inteiro k. No exemplo da figura 2, o problema de decisão iria ter solução positiva para $k \geq 2$ e negativa caso contrário.

Um lema imprescindível para a construção do algoritmo que resolve este problema encontra-se a seguir. Este lema foi proposto e provado em (MARCILON; SAMPAIO, 2018a).

Lema 2.1.1. *Seja G um grafo conexo com $\Delta(G) \leq 3$ e k um inteiro não negativo. Então, $t(G) \geq k$ se ,e somente se, G possui um caminho induzido P tal que, ou P tem $2k-1$ vértices, todos com*

grau três; ou P tem k vértices, todos com grau três, à exceção de um extremo, que possui grau dois.

A busca por caminhos induzidos é de fácil implementação e de complexidade mais baixa do que a busca completa por conjuntos infectados inicialmente que levam ao tempo máximo de infecção. Assim, o algoritmo desenvolvido em (MARCILON; SAMPAIO, 2018a) e implementado neste trabalho utiliza-se desta ideia para resolver o problema proposto.

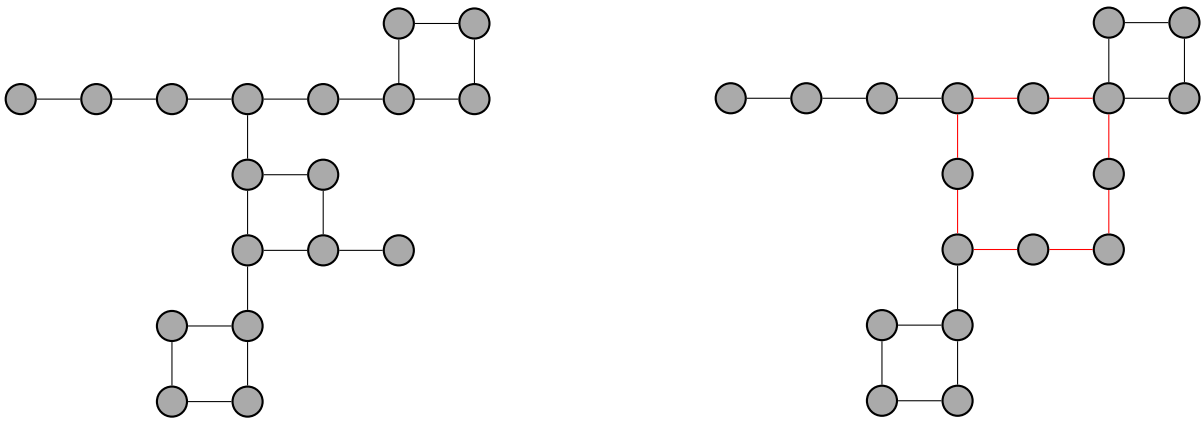
2.1.2 Problema de cálculo de $t(G)$ para Grids Sólidos com $\Delta = 3$

A definição do problema é encontrar o tempo máximo de infecção $t(G)$ em grafos G pertencentes a classe de *Grids* Sólidos que possuam grau máximo igual a três.

Primeiramente, é necessário definir-se a classe de grafos *Grid* Sólidos.

Esta classe de grafos consiste em grafos *grid* (ou seja, grafos cuja representação gráfica consiste em uma malha 2D) cujas áreas limitadas possuem área um (considerando cada aresta com comprimento igual a uma unidade de medida). Na figura 3, encontra-se um exemplo de *grid* sólido e de um *grid* que não pertence a esta classe.

Figura 3 – Exemplo de Grafos que pertencem ou não a classe de *Grids* Sólidos



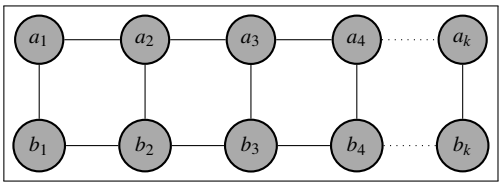
(a) *Grid* Sólido (b) *Grid* não sólido (Área ≥ 1 destacada)

Fonte: o autor.

Define-se ainda o conceito de *escadas*. Escadas em um *grid* sólido são subgrafos maximais grid de dimensão $2 \times k$. A estrutura geral de uma escada, encontra-se na figura 4.

O algoritmo proposto em (MARCILON; SAMPAIO, 2018a) para a obtenção de $t(G)$ em *grids* sólidos com $\Delta = 3$ transforma as escadas em K_4 ponderados e utiliza-se do

Figura 4 – Escada em um *Grid* Sólido



Fonte: o autor.

lema 2.1.1 para encontrar caminhos induzidos de peso máximo. Uma visão mais detalhada do funcionamento do algoritmo encontra-se na seção 4.2.1.

2.1.3 Problema de decisão $t(G) \geq n-k$ para grafos quaisquer

Este problema é o mais simples. Dado um grafo G qualquer e um inteiro $0 \leq k \leq n$, pretende-se resolver o problema de decisão $t(G) \geq n-k$.

É interessante apresentar-se uma observação, presente em (MARCILON; SAMPAIO, 2018a), que sustenta o algoritmo desenvolvido.

Afirmação 2.1.1. *Suponha que um grafo G possui $t(G) \geq n-k$. Em um processo de infecção com tempo $n-k$, há pelo menos um vértice infectado em tempo* 1*, um vértice infectado em tempo* 2 *e, assim por diante, um vértice infectado em tempo $n-k$. Deste modo, existem, no máximo, k vértices infectados no tempo* 0*.*

Desta afirmação, nasce a ideia do algoritmo: testar todos os conjuntos inicialmente infectados de até k vértices e verificar se o tempo de infecção de algum deles é $n-k$ ou maior.

2.1.4 Problema de decisão $t(G) \geq 3$ para grafos bipartidos

O problema é definido como, dado um grafo G bipartido, decidir se $t(G) \geq 3$.

Um importante resultado mostrado e provado em (MARCILON; SAMPAIO, 2018b) segue:

Lema 2.1.2. *Seja G um grafo bipartido e T_0 o conjunto de vértices v de G com $d(v) = 1$. Assim, $t(G) \geq 3$ se, e somente se, há três vértices u, $v \in N(u)$ e $s \in N_2(u)$ tal que $T_0 \cup N_{\geq 3}(u) \cup \{v,s\}$ infecta u em tempo 3.*

O lema 4.4 valida a estratégia de verificar-se iteradamente todos as combinações de u e $T_0 \cup N_{\geq 3}(u) \cup \{v,s\}$, com $v \in N(u)$ e $s \in N_2(u)$, para a verificação de $t(G) \geq 3$.

2.1.5 Problema de decisão $t(G) \geq 3$ para grafos quaisquer

Embora este problema seja uma extensão do problema da seção anterior para grafos quaisquer, é necessário apresentar algumas definições e lemas que dão suporte à corretude do algoritmo implementado neste trabalho. O algoritmo e os lemas são desenvolvidos e provados em (MARCILON; SAMPAIO, 2018b).

Definição 2.1.2. *Seja $\mathscr{T}_0^u$ uma família de subconjuntos de $V(G)$ tal que um conjunto $T_0 \in \mathscr{T}_0^u$ se, e somente se, para cada articulação v e cada componente conexo $H_{v,i}$ de $G-v$ tal que $u \notin V(H_{v,i})$ e $V(H_{v,i}) \subseteq N(v)$, T_0 contém exatamente um vértice de $H_{v,i}$ e todo vértice de T_0 possui essa propriedade.*

Desta definição, parte o seguinte lema:

Lema 2.1.3. *Seja G um grafo conexo. $t(G) \geq 3$ se, e somente se, existe um vértice u, um conjunto $T_0 \in \mathscr{T}_0^u$ e um conjunto F com, no máximo, 4 vértices tal que $T_0 \cup N_{\geq 3}(u) \cup F$ infecta u no tempo* 3.

O lema 2.1.3 tem papel semelhante ao do lema 4.4 na seção anterior para a motivação do algoritmo desenvolvido. Ele delimita um conjunto de busca para a verificação da propriedade $t(G) \geq 3$. O espaço de busca, entretanto, seria consideravelmente maior se não fosse pelo último lema:

Lema 2.1.4. *Se existe um vértice u, um conjunto $T_0 \in \mathscr{T}_0^u$ e um conjunto de vértices F, com $|F| \leq 4$, tal que $T_0 \cup N_{\geq 3}(u) \cup F$ infecta u no tempo* 3*, então para qualquer conjunto $T_0' \in \mathscr{T}_0^u$ existe um conjunto de vértices F', com $|F'| \leq 4$, tal que $T_0 \cup N_{\geq 3}(u) \cup F$ infecta u no tempo* 3.

O lema 2.1.4 restringe o espaço de busca a qualquer conjunto $T_0 \in \mathscr{T}_0^u$, evitando o teste iterado na família de conjuntos $\mathscr{T}_0^u$.

Todos os lemas foram desenvolvidos e provados em (MARCILON; SAMPAIO, 2018b).

3 METODOLOGIA

O foco do presente trabalho é a obtenção de implementações corretas para os algoritmos propostos. Deste modo, apresentam-se as metodologias de implementação e teste utilizadas.

3.1 Implementações

As implementações dos algoritmos foram realizadas em linguagem `C++`, utilizando-se da versão `C++11`. Três funcionalidades desta versão da linguagem foram a razão para tal escolha:

1. **Inferência de Tipo:** Nesta versão, utilizando-se da *keyword* `auto`, é possível declarar variáveis sem especificar explicitamente seu tipo.
2. ***Range Based Loops***: Permitem o percurso iterado de coleções de elementos, o que é especialmente útil para a codificação de algoritmos com listas de adjacência.
3. ***Keyword*** `nullptr`: Substitui a macro `NULL` de uma forma menos suscetível a erro.

O ponto de partida de qualquer implementação foi um pseudocódigo do algoritmo proposto. Em alguns algoritmos, estes não estavam explicitamente disponíveis e foi necessário estruturar a descrição destes em pseudocódigos padronizados.

3.2 Testes

Naturalmente, após a implementação do algoritmo, é necessário verificar-se se o código escrito é, de fato, o do algoritmo proposto. Essa avaliação foi realizada considerando-se dois pontos: a complexidade da implementação e a correta solução de conjuntos de casos de teste.

3.2.1 Testes de Complexidade

Por testes de complexidade, pretende-se denominar verificações que comprovem que a execução do código segue a complexidade do algoritmo proposto. Isto foi realizado medindo-se o tempo de execução para entradas redimensionáveis nas variáveis que impactam o número de operações do algoritmo. Os resultados de tempo de execução foram, então, representados graficamente. Estes gráficos são o resultado final do teste, na medida que sugerem a complexidade

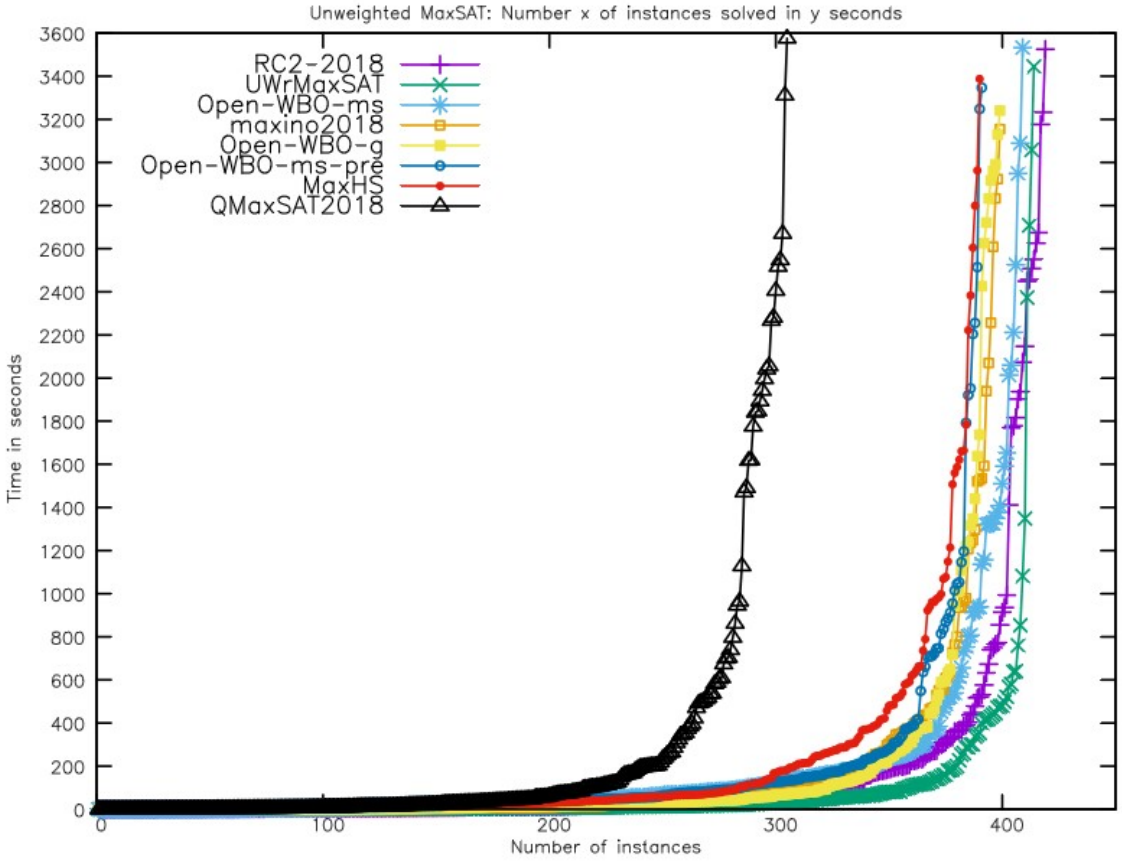
de execução da implementação. De modo procedural, o método de validação de complexidade foi:

- Gerar grafos de acordo com as limitações de entrada do problema. A dimensão do grafo (seu número de vértices e arestas) é variada de acordo com a função de complexidade esperada. Ressalta-se que a escolha da classe de grafo é feita de modo a garantir o pior (ou quase pior) caso de execução.
- Executar o algoritmo com as entradas previamente construídas.
- Para cada instância de entrada, associar o tempo de execução com as variáveis da função de complexidade esperada.
- Plotar o tempo de execução como função das variáveis de complexidade do algoritmo.

Como o objetivo é obter-se indícios de que a complexidade de execução é da mesma ordem de magnitude da função de complexidade, a análise aqui realizada será inteiramente gráfica. Modelos mais complexos de análise empírica de complexidade já foram propostos (GOLDSMITH *et al.*, 2007), mas, como este não é o objetivo principal do estudo, atém-se ao método mais simples aqui proposto.

Gráficos de tempo de execução também são utilizados para avaliar a complexidade de códigos em eventos de otimização. Na figura 5, um exemplo de tal gráfico no evento *MAXSAT Evaluation 2019*. Nestes eventos, diferentes implementações de resolvedores do problema MAXSAT são analisados, de modo a encontrar o mais eficiente.

Figura 5 – Gráfico de tempo de execução para resolvedores MAXSAT



Fonte: Página do evento *MAXSAT Evaluation 2019*[1]

A diferença entre a figura 5 e os gráficos aqui gerados se dá pelo fato de que aqui compara-se o tempo de execução do algoritmo implementado com gráficos das funções de complexidade esperadas. Para o caso de funções mais simples, como $O(n^2)$, realizou-se tomadas de tempo de algoritmos com este tempo esperado (dois laços aninhados, por exemplo). Já para funções com maior número de variáveis, apenas uma comparação do gráfico de tempo de execução com o da função matemática de sua complexidade esperada foi feita.

3.3 Testes da Implementação

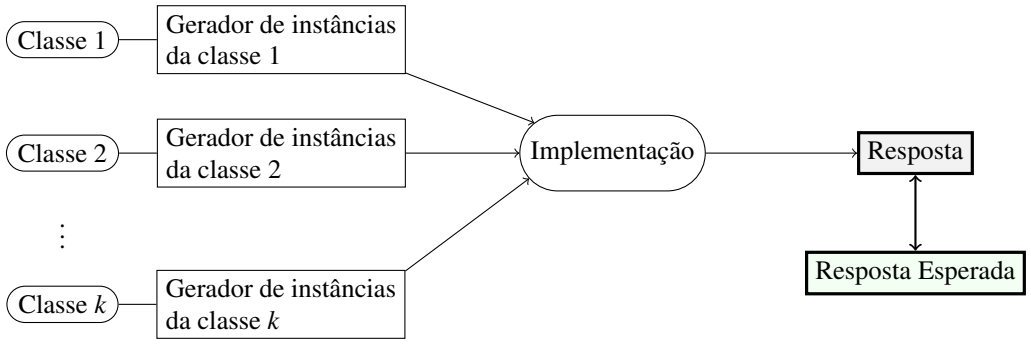
O segundo foco dos testes é garantir a correta implementação do algoritmo. Para tanto, utilizou-se de duas estratégias:

1. **Casos de Teste:** Nesta modalidade, escolheu-se classes de grafos de entrada cujas respostas sejam possíveis de equacionar em função do número de vértices, arestas ou alguma outra métrica do grafo em questão. Posteriormente, gera-se várias instâncias de grafos

[1] Disponível em <https://maxsat-evaluations.github.io/2019/results/complete/unweighted/summary.html> - Acessado em 20/10/2019

das classes previamente escolhidas e compara-se o resultado da implementação com a equacionada. O processo de teste para esta abordagem encontra-se esquematizado na figura 6.

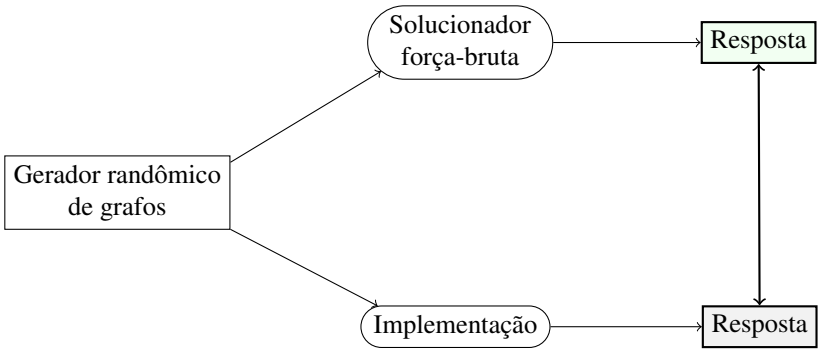
Figura 6 – Diagrama para o processo de validação com Casos de Teste



Fonte: o autor

2. **Geradores Randômicos com Solucionadores Força Bruta:** Para algoritmos cujas entradas fossem menos restritivas, criou-se um gerador de grafos randômicos e um solucionador do problema no estilo "força-bruta". Tal verificador testa iteradamente todas as possíveis respostas para o problema proposto e, ao encontrar a solução,a compara com aquela gerada pelo algoritmo a ser validado. Um esquema para esta validação encontra-se na figura 7.

Figura 7 – Diagrama para o processo de validação com Entradas Randômicas



Fonte: o autor

4 RESULTADOS

A seguir, encontram-se os algoritmos estudados e seus resultados.

Nota sobre Reprodutibilidade

As implementações de todos os algoritmos aqui estudados, bem como as rotinas de teste e os *scripts* para geração dos gráficos encontram-se disponíveis para consulta em repositório público [1].

4.1 Algoritmo de Decisão $t(G) \geq k$ em grafos com $\Delta \leq 3$

Este primeiro algoritmo trata do problema de decisão $t(G) \geq k$ para grafos conexos quaisquer com grau máximo limitado a três.

4.1.1 O Algoritmo

Valendo-se do lema 2.1.1, o algoritmo desenvolvido em (MARCILON; SAMPAIO, 2018a) para a resolução do problema de decisão tem como ideia central encontrar caminhos induzidos com as profundidades relacionadas ao valor de k: $2k-1$, caso todos os vértices do caminho tenham grau três ou k, caso algum extremo possuir grau 2, com todos os outros vértices de grau três. A existência de tais caminhos garantem uma resposta positiva ao problema de decisão $t(G) \geq k$. Naturalmente, sua inexistência implica uma resposta negativa.

Assim, utiliza-se de uma busca em profundidade (*Depth First Search* - **DFS**) modificada para encontrar tais caminhos. As modificações de uma DFS padrão serão as seguintes:

1. Inicia-se uma busca apenas em vértices de grau dois ou três.
2. Ao "caminhar-se" para um vértice vizinho não visitado, é necessário garantir a indução do caminho atual. Esta verificação pode ser feita com o auxílio de um dado já utilizado na DFS: a informação de um vértice ser visitado ou não. Assim, a busca prossegue apenas se não existirem arestas do vértice analisado com um vértice marcado (que não seja seu predecessor na busca). Ao final da execução em um nó, faz-se necessário desmarcá-lo, pois a busca deve verificar todos os possíveis caminhos.
3. Por fim, se a busca estiver nas profundidades limites, retorna-se a resposta positiva do

[1] Disponível em <https://github.com/lucaskeiler/AlgoritmosTCC>

problema de decisão (só é necessário um caminho).

O pseudocódigo para este problema de decisão encontra-se no Algoritmo 1. A função `decide` é a ser executada quando deseja-se saber se $t(G) \geq k$. As funções `dfsInduzido` e `induzido` fazem, respectivamente, a busca por caminhos de profundidade $k-1$ ou $2k-2$ e a verificação da indução do caminho atual, caso adicione-se o vértice s.

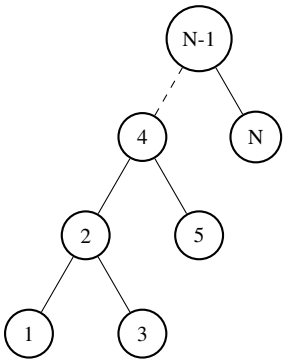
Algoritmo 1: Algoritmo para decisão de $t(G) \geq k$, com $\Delta \leq 3$

```
Função dfsInduzido(v, d):
    marcar: v
    se d = máximo então
        resposta ← verdadeiro
        retorna
    fim
    para cada vértice s vizinho de v faça
        se s não marcado e induzido(v,s) então
            dfsInduzido(s,d+1)
        fim
    fim
    desmarcar: v

Função induzido(s, predecessor):
    para cada vértice x vizinho de v faça
        se x marcado e x ≠ predecessor então
            retorna falso
        fim
    fim
    retorna verdadeiro

Função decidir(G, k):
    resposta ← falso
    para cada vértice v em G faça
        se grau(v) = 2 então
            máximo ← k−1
            dfsInduzido(v,0)
        fim
        senão se grau(v) = 3 então
            máximo ← 2k−2
            dfsInduzido(v,0)
        fim
    fim
    retorna resposta
```

4.1.2 Testes da Implementação

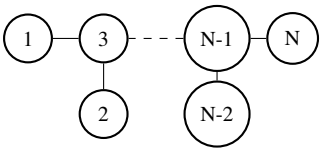
Para os testes da implementação do presente algoritmo, desenvolveu-se geradores automáticos de grafos determinísticos. As classes de grafos escolhidas para os casos de teste são mostradas na figura 8. Dentre elas, encontram-se algumas classes conhecidas, como a árvore binária e o grafo *caterpillar*.

Figura 8 – Classes de Grafos para Teste da Implementação

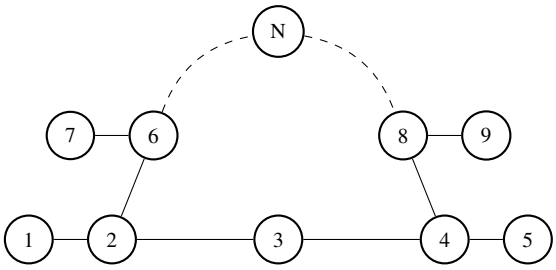


(a) Classe 1 [G1]

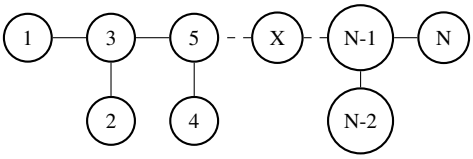
(b) Classe 2 [G2]



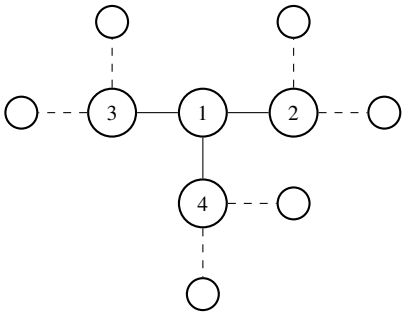
(c) Classe 3 [G3]



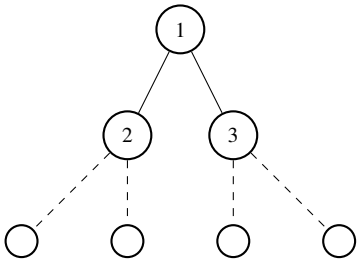
(d) Classe 4 [G4]



(e) Classe 5 [G5]



(f) Classe 6 [G6]



(g) Classe 7 [G7]

Fonte: o autor.

Os testes foram gerados utilizando o seguinte padrão:

1. Para cada classe, executar o gerador para um determinado número de vértices N, sendo $N \in [1,V]$.
2. Em se tratando de um problema de decisão, variou-se k no intervalo $[1,N]$ e executou-se o algoritmo para cada k.
3. As classes de grafo escolhidas permitem respostas equacionadas para o tempo de infecção (veja a tabela 1. Deste modo, compara-se, por fim, o resultado do algoritmo com o esperado.

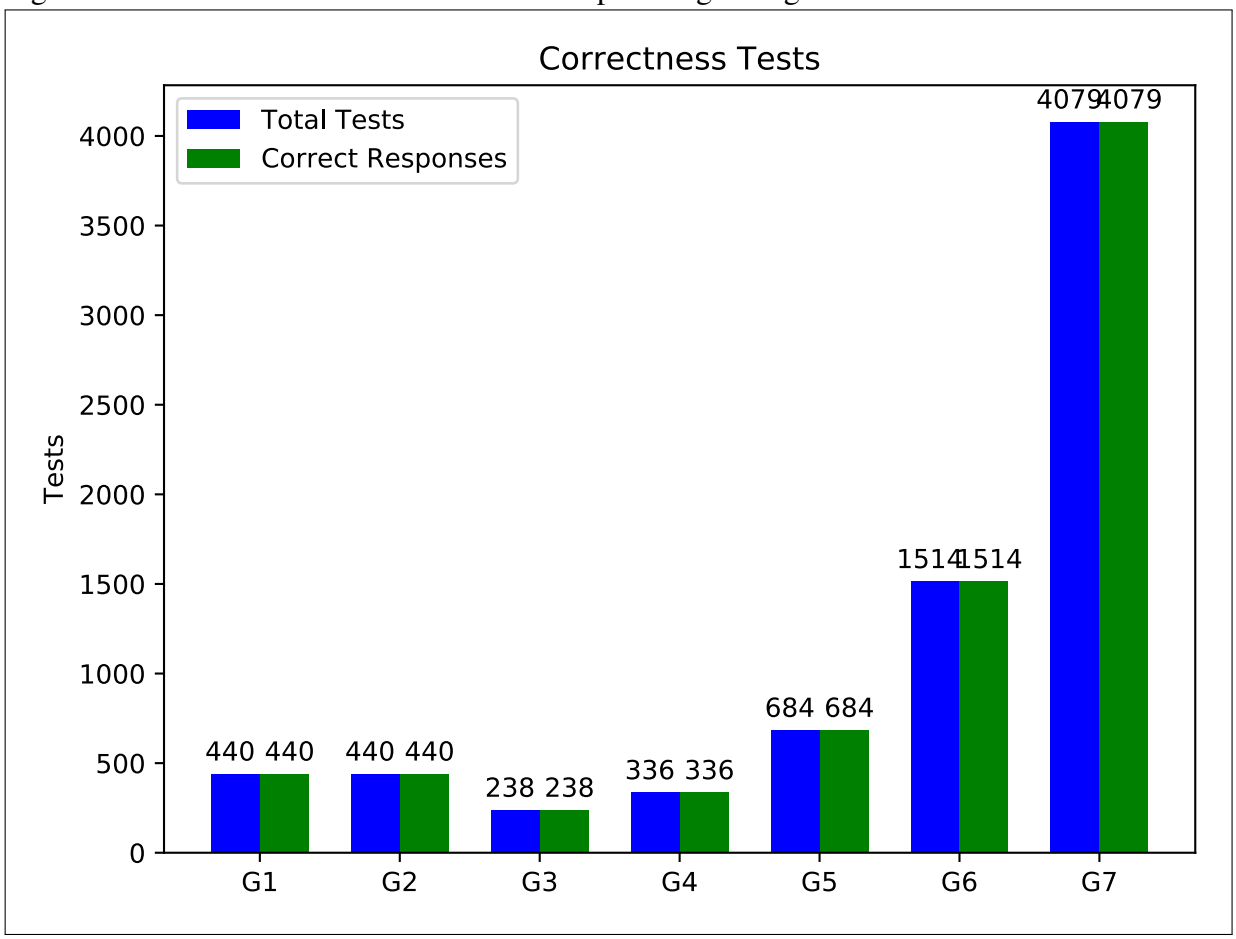
Tabela 1 – Tempo de infecção para as classes de teste

Classe	Número de Vértices	$t(G)$
G1	$2K+1, K \geq 1$	K
G2	$K, K \geq 3$	1
G3	$2K, K \geq 3$	K
G4	$2K, K \geq 3$	$\lceil K/2 \rceil$
G5	$2*(a+b+1)+1$	$(max(a,b)-1)/2$
G6	$1+3(2^k-1)$	k
G7	N	$log(N)$

Fonte: o autor.

Os resultados foram todos positivos, demonstrando robustez na implementação realizada. O volume de testes para cada classe pode ser verificado na figura 9.

Figura 9 – Resultados dos testes de corretude para os grafos gerados



Fonte: o autor.

4.1.3 Testes de Complexidade

Uma análise de complexidade já pode ser realizada a partir do pseudocódigo do Algoritmo 1. A seguir, a complexidade esperada de cada sub-rotina do algoritmo:

1. `induzido`: Trata-se de um laço de tamanho $d(v)$, para um vértice qualquer v. Como os grafos são limitados a $d(v) \leq 3$, esta sub-rotina tem complexidade $O(1)$.
2. `dfsInduzido`: As modificações que desviariam a complexidade da esperada para uma busca em profundidade convencional seriam a invocação do método `induzido` e o fato de cada vértice ser desmarcado ao final de seu processamento. A seguir, suas análises:
   - **Método `induzido`:** Como constatado acima, o método `induzido` tem complexidade constante. Assim, o laço da busca tem como maior complexidade a recursão da **DFS** e pode-se ignorar a complexidade deste método.
   - **Vértice desmarcado ao final da DFS**: Isto adiciona complexidade, na medida que um vértice (e uma aresta) pode ser visitado (ou utilizado) mais do que uma vez.

Entretanto, um caminho só é analisado uma vez. Portanto, analisa-se o número de operações pelo número de caminhos possíveis a partir do vértice inicial. Note que um limite superior é considerar que cada vértice possui grau 3 e, desse modo, o número de caminhos de tamanho l é limitado a:

$$3 \cdot 2 \cdot 2 \ldots 2 = 3 \cdot 2^{l-1}$$

Considerando que todo vértice v possui $d(v) = 3$ e, portanto, a profundidade de busca é $2k-2$, tem-se:

$$3 \cdot 2^{l-1} = 3 \cdot 2^{2k-3} = \frac{3}{8} \cdot 2^{2k}$$

Por fim, considerando $k \leq c \cdot \log_2 n$:

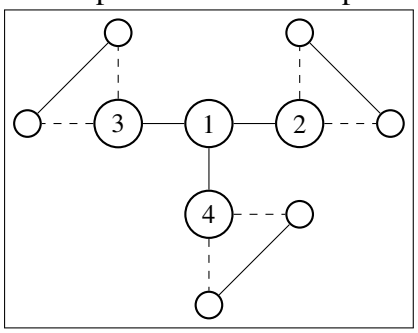
$$\frac{3}{8} \cdot 2^{2k} \leq \frac{3}{8} \cdot n^{2c}$$

3. `decide`: Este é o método inicial do algoritmo. Trata-se de um laço que invoca `dfsInduzido`, a depender do grau de um vértice. No pior dos casos, todos os vértices teriam grau três. A partir da complexidade do da sub-rotina `dfsInduzido`, tem-se que a complexidade deste método é $O(n^{2c+1})$.

Os testes de complexidade foram realizados utilizando-se da classe de grafo da figura 10. Esta escolha foi realizada para simular um pior caso do algoritmo. Nota-se que a obtenção de um grafo com execução de pior caso para este algoritmo não é trivial, na medida que grafos muito densos tendem a possuir caminhos de profundidade máxima procurada iniciando-se em todos (ou quase todos) os vértices (a execução do algoritmo encerra no momento em que um caminho de profundidade maior ou igual a procurada é encontrado). Assim, o grafo escolhido é bastante interessante, pois não possui caminhos de profundidade máxima determinante para uma resposta positiva ao problema de decisão.

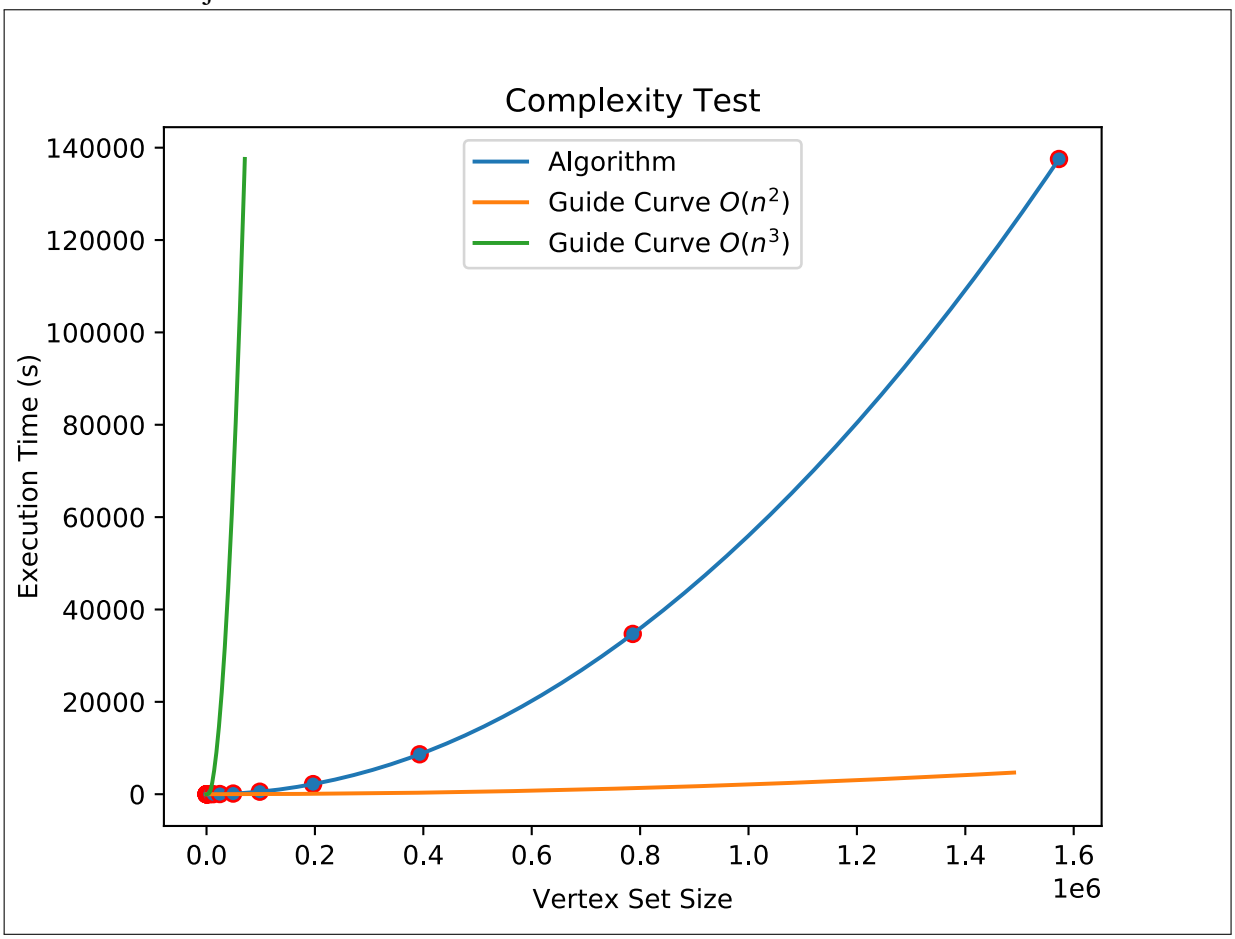
De fato, a execução do algoritmo para grafos da topologia da figura 10, com número de vértices redimensionáveis, indica complexidade $O(n^3)$ (embora frouxa), o que indica concordância com o limite calculado anteriormente. O resultado do teste de complexidade encontra-se no gráfico da figura.

Figura 10 – Classe de grafo utilizada para o teste de complexidade



Fonte: o autor.

Figura 11 – Tempo de execução do algoritmo para a topologia da figura 10 e cardinalidade do conjunto de vértices variada.



Fonte: o autor.

4.2 Infecção em Grids Sólidos com $\Delta = 3$

Limitando-se o grau de um grafo G *grid* sólido em três, é possível calcular $t(G)$ em $O(n^2)$. A seguir a apresentação deste algoritmo e os testes de sua implementação.

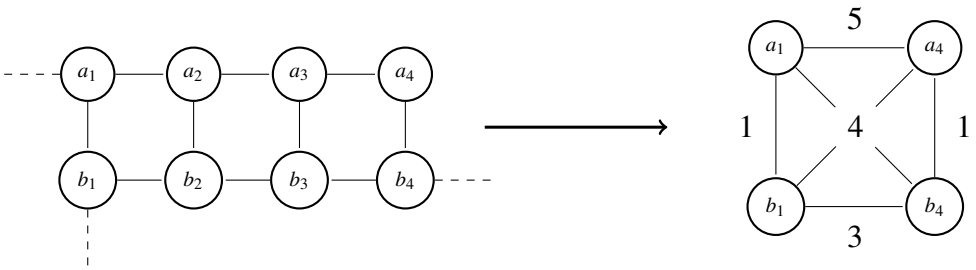
4.2.1 O Algoritmo

O presente algoritmo foi proposto e teve sua corretude provada em (MARCILON; SAMPAIO, 2018a).

De acordo com o lema 2.1.1, para encontrar o tempo de infecção de um grafo com $\Delta \leq 3$, basta encontrar-se o maior caminho induzido com todos os vértices de grau três, à exceção de, no máximo, uma extremidade de grau dois. Isto pode ser feito em $O(n^2)$ para *grids* sólidos em dois passos: transformando as escadas do grafo em K_4 e percorrendo-o em busca do maior caminho induzido.

As transformações de escadas em K_4 se fazem necessárias para diminuir o espaço de busca de caminhos induzidos. Note que, entre um par de extremidades de uma escada, existem $O(2^{k-1})$ caminhos (a cada vértice interno à escada, há duas escolhas de bifurcação). De todos estes caminhos, para encontrar-se $t(G)$, estamos interessados apenas nos maiores que são induzidos. Assim, é possível simplificar qualquer escada em um K_4 ponderado. As arestas teriam como peso o comprimento do maior caminho induzido com vértices internos de grau três (à exceção de, no máximo, uma extremidade). Caso tal caminho não exista, assume-se peso negativo infinito. Um exemplo de tal transformação encontra-se na figura 12.

Figura 12 – Exemplo de transformação de uma escada em K_4



Fonte: o autor.

De maneira formal, sendo u, v extremidades de uma escada maximal L_k (tamanho k), o peso $w(u,v)$ da aresta correspondente no K_4 será (MARCILON; SAMPAIO, 2018a):

$$w(u,v) = \begin{cases} 1, & \text{se } u,v \text{ vizinhos} \\ d_b(k), & \text{se ambas as outras extremidades têm grau 3} \\ d_b(k-1)+1, & \text{se apenas uma extremidade possui grau 1} \\ d_b(k-2)+2, & \text{se ambas as outras extremidades têm grau 2 e } k > 2 \\ -\infty, & \text{caso contrário.} \end{cases}$$

Onde:

$$d_b(k) = \begin{cases} (k-1)+2\cdot\lfloor (k+1)/4 \rfloor, & \text{se } b = 0 \\ k+2\cdot\lfloor (k-1)/4 \rfloor, & \text{se} b = 1 \end{cases}$$

Finalmente, após as transformações das escadas maximais em G (resultando em um grafo modificado G'), executa-se uma busca em profundidade modificada que encontra o caminho induzido de maior peso com vértices internos de grau três e extremidades de grau pelo menos dois. O peso resultante de tal busca é decrescido em uma unidade, caso ambas as extremidades possuam grau dois. Isso é necessário, pois os caminhos induzidos maximais com as propriedades do lema 2.1.1 em G, terminam em vértices internos de escadas, que foram suprimidas em G'. Assim, tais caminhos terminam em uma extremidade de uma escada em G' e deve-se encurtá-lo em uma unidade.

O pseudocódigo do presente algoritmo encontra-se nos algoritmos 2 e 3.

Algoritmo 2: Algoritmo para Encontrar $t(G)$ em Grids Sólidos (Parte 1)

```
Função tempoMáximoInfecçãoSolidGrid(G):
    G' ← Transformar(G)
    tempoMáximo ← 0
    para cada vértice v em G' com d(v) ≥ 2 faça
        se grau(v) = 2 então
            atual ← dfsInduzidoPesoMáximo(G',v) + 1;
        fim
        senão
            atual ← ⌊(dfsInduzidoPesoMáximo(G',v) + 2)/2⌋
        fim
        tempoMáximo ← max(tempoMáximo, atual)
    fim
    retorna tempoMáximo
```

Em especial, ressalta-se a dificuldade de implementação da função `Transformar`

no algoritmo 3. Sua codificação requer a atenção a muitos casos na detecção de escadas e no cálculo de arestas do posterior grafo K_4.

Algoritmo 3: Algoritmo para Encontrar $t(G)$ em Grids Sólidos (Parte 2)

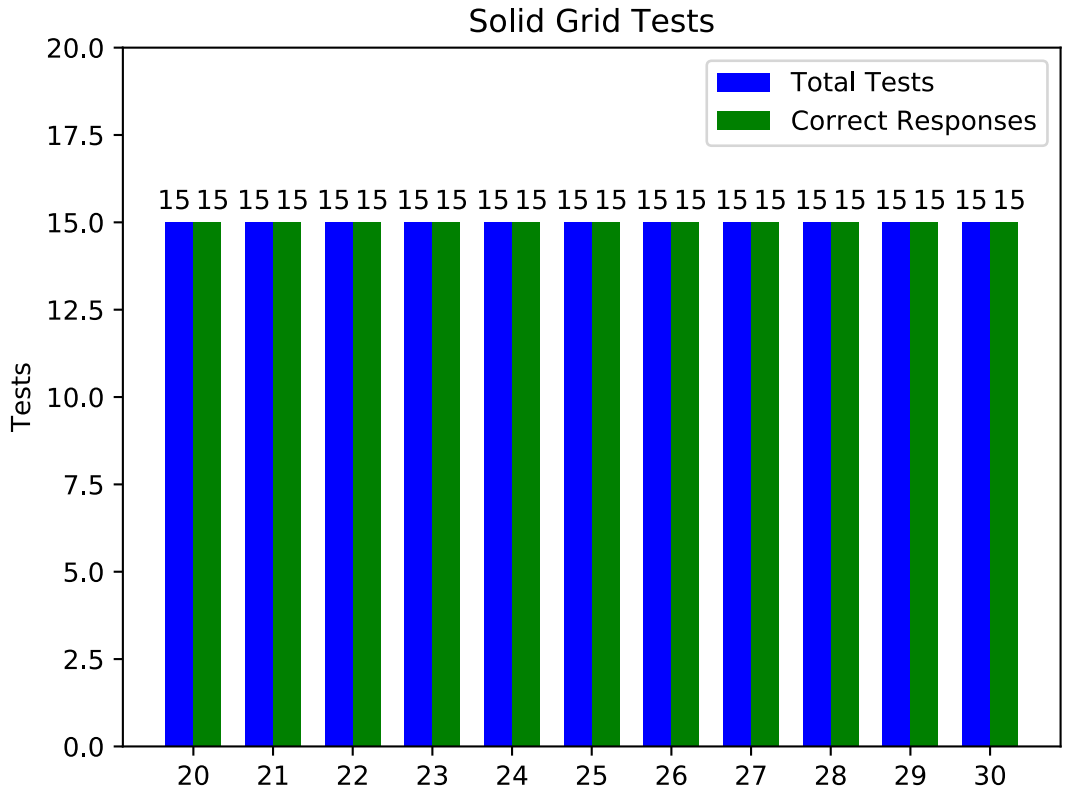
```
Função Transformar(G):
    escadas ← detectarEscadas(G)
    para cada escada e em escadas faça
        para cada vértice v de e faça
            se v é extremidade de e então
                inserir v em extremidades[e]
            fim
        fim
    fim
    para cada escada e em escadas faça
        para cada par de vértices (u,v) em extremidades[e] faça
            inserir aresta (u,v,w) em G
        fim
        para cada vértice v em e faça
            se v não está em extremidades[e] então
                remover v de G
            fim
        fim
    fim
    retorna G

Função dfsInduzidoPesoMáximo(G,v,inicio,pesoAtual):
    marcar v
    se v ≠ inicio e grau[v] = 2 então
        se grau[inicio] = 2 então
            retorna pesoAtual - 1
        fim
        senão
            retorna pesoAtual
        fim
    fim
    pesoMáximo ← pesoAtual
    para cada vértice u não marcado de G vizinho de v faça
        pesoMáximo = max(pesoMáximo,
        dfsInduzidoPesoMáximo (G,u,inicio,pesoAtual + w(v,u)))
    fim
    retorna pesoMáximo
```

4.2.2 Testes da Implementação

Em geral, *grids* sólidos de grau limitado a três são de difícil geração para casos de teste determinísticos. Deste modo, utiliza-se a estratégia de casos de teste randômicos. Isto é, codificou-se um gerador de *grids* sólidos semi-determinísticos: a partir de um vértice inicial de grau sempre igual a três, constrói-se três caminhos, em que, para cada vértice, existe uma probabilidade de 50% de ele pertencer a uma escada. O grafo resultante é submetido ao algoritmo aqui estudado e um testador "força-bruta", que testa todos os possíveis grupos de vértices infectados no tempo 0. Caso o testador consiga encontrar um conjunto inicial de vértices que leva à infecção do grafo original em um tempo $t' > t$ (t seria a resposta do algoritmo), considera-se uma falha de implementação.

A execução desta estratégia de teste foi realizada para grafos semi-determinísticos com cardinalidades do conjunto de vértices entre 20 e 30, com 15 repetições para cada tamanho. Assim, totalizou-se 165 casos de teste, todos bem-sucedidos. O resultado dos testes encontra-se na figura 13.

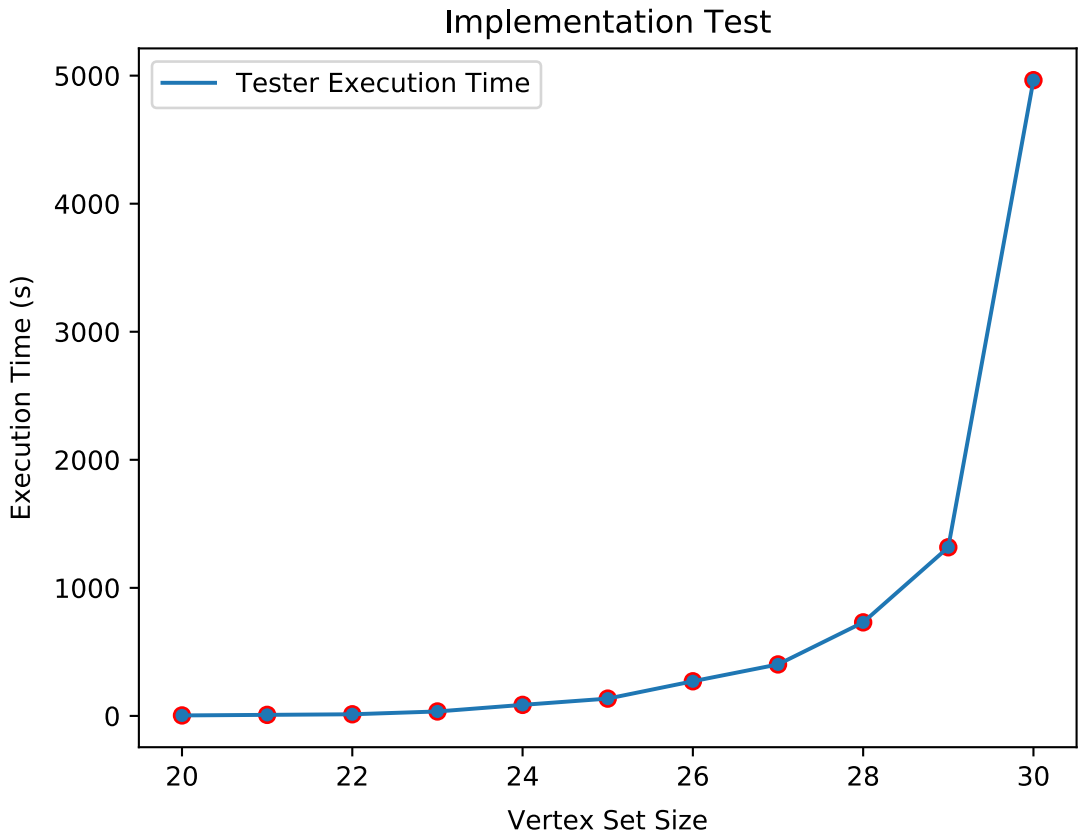
Figura 13 – Resultados dos testes de implementação do algoritmo de infecção em *Grids* Sólidos



Fonte: o autor.

Como indicativo da limitação deste método de teste no tocante ao tamanho do grafo, mostra-se na figura 14 o tempo necessário para cada cardinalidade de conjunto de vértices.

Figura 14 – Tempo de execução para cada teste

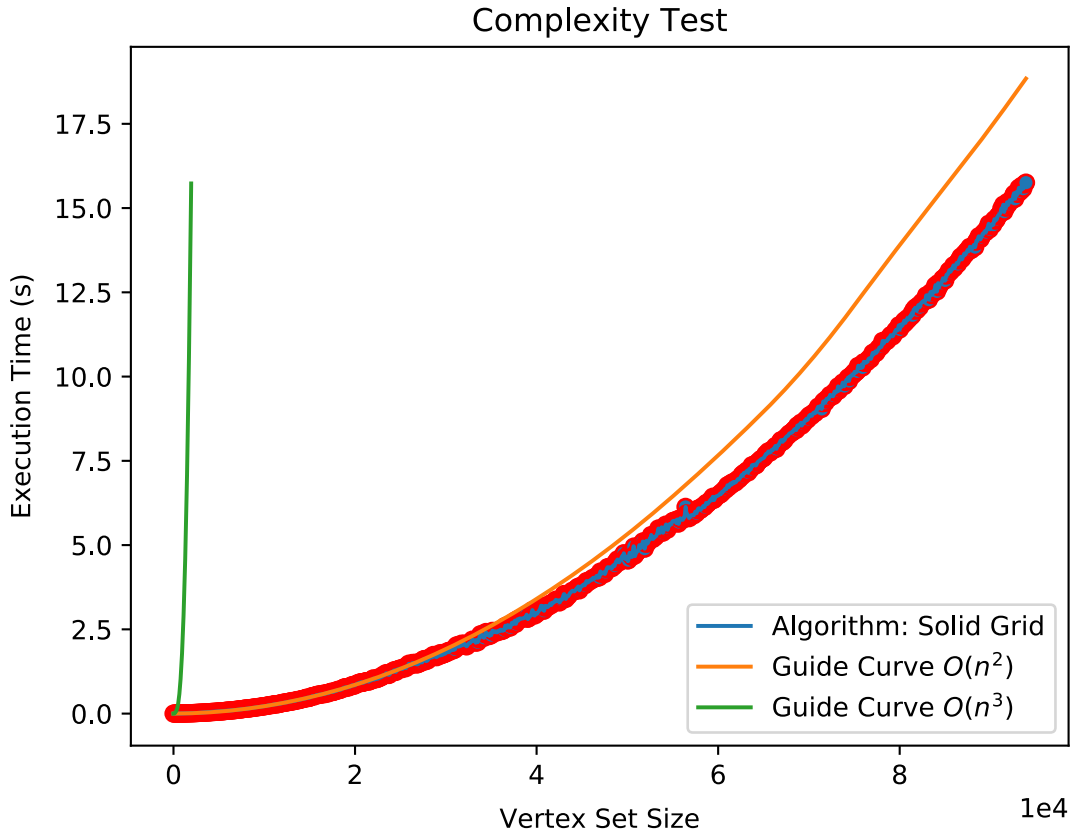


Fonte: o autor.

4.2.3 Testes de Complexidade

Para os testes de complexidade, utilizou-se do mesmo gerador de *grids* sólidos semi-determinísticos da seção anterior. Agora, entretanto, a cardinalidade do conjunto de vértices pode ser testada em um intervalo bem mais amplo. Plotando-se o tempo de execução gasto para cada grafo e duas linhas guia para complexidades $O(n^2)$ e $O(n^3)$, é possível concluir que a complexidade encontra-se dentro da esperada: $O(n^2)$. O resultado encontra-se na figura 15.

Figura 15 – Resultados dos testes de complexidade para o algoritmo de infecção em *Grids* Sólidos



Fonte: o autor.

4.3 Algoritmo para o problema $t(G) \geq n - k$ para grafos quaisquer

Este algoritmo é, provavelmente, o mais simples de todos os aqui estudados. Ele pretende resolver o seguinte problema de decisão:

$$t(G) \geq n - k$$

Sendo G um grafo qualquer e $0 \leq k \leq n$ uma constante qualquer.

O algoritmo proposto em (MARCILON; SAMPAIO, 2018a) para resolver este problema tem como ideia testar todos os conjuntos de vértices inicialmente infectados com cardinalidade k. O pseudo-código encontra-se no algoritmo 4.

4.3.1 Testes da Implementação

Em se tratando de um algoritmo de busca completa, os testes realizados não foram de volume muito extenso. Utiliza-se as quatro classes de grafo da figura 16, que possuem tempo

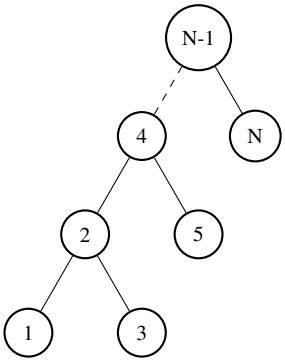
Algoritmo 4: Algoritmo para problema de decisão $t(G) \geq n - k$

```
Função VerificarTempoN-K(G,k):
    para cada permutação P de vértices em G faça
        tempo ← 0 enquanto houver infecção em G faça
            para cada vértice v não infectado faça
                se mais de um vizinho de v está infectado então
                    infectar(v)
                fim
            fim
            tempo ← tempo + 1
        fim
        se tempo - 1 ≥ n - k então
            retorna verdadeiro
        fim
    fim
    retorna falso
```

de infecção equacionado. Desta maneira, redimensiona-se os grafos de entrada e compara-se o resultado com o previamente calculado.

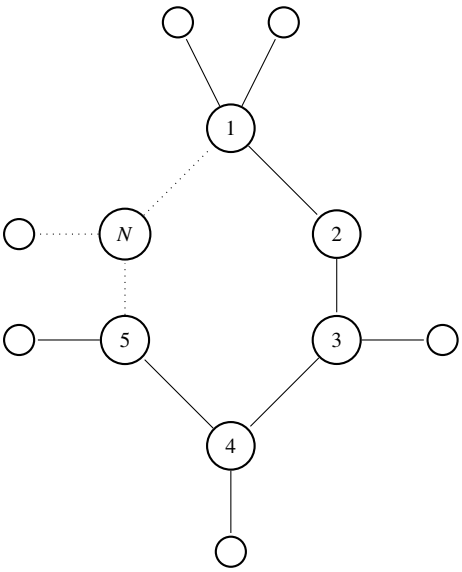
Todos os testes foram bem-sucedidos. Detalhes dos resultados e do volume de testes para cada classe encontram-se na figura 17.

Figura 16 – Classes de Grafos para Teste da Implementação do Algoritmo 4.3

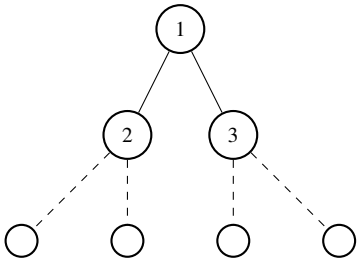


(a) Classe 1 [G1]

(b) Classe 2 [G2]



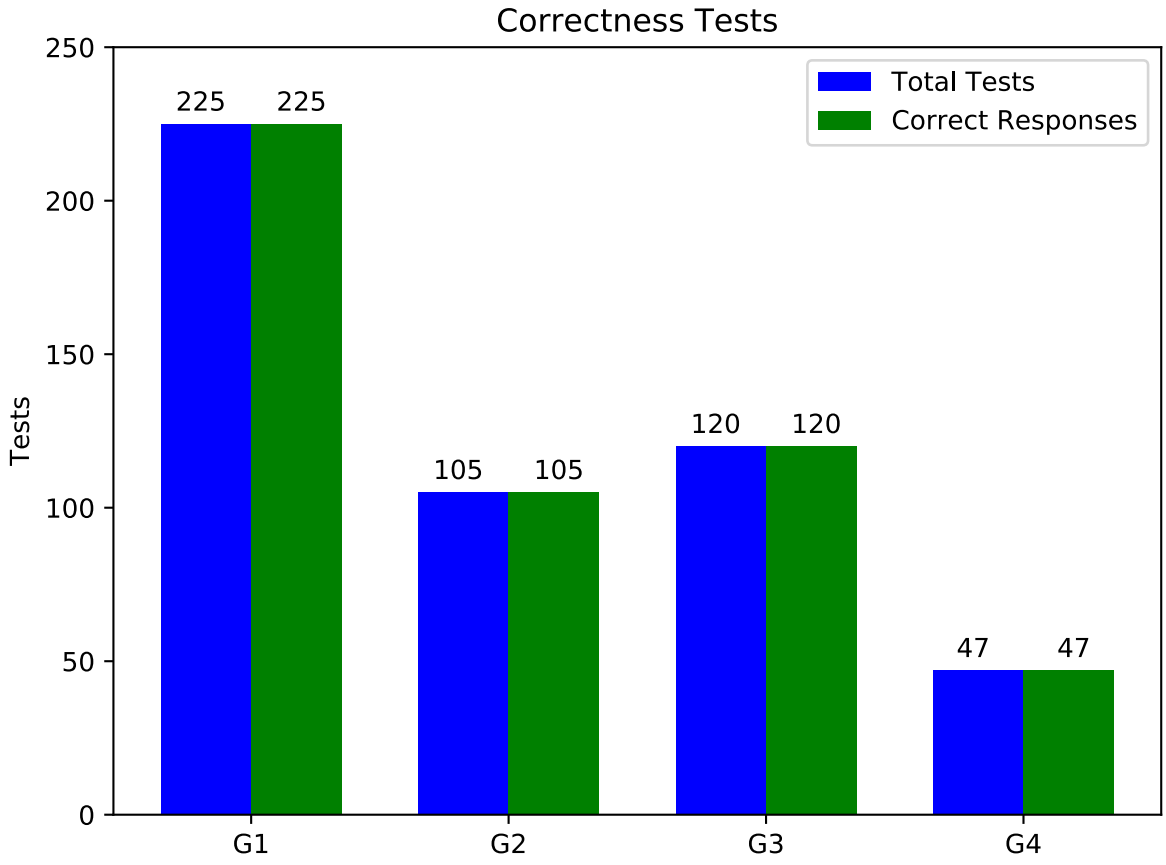
(c) Classe 3 [G3]



(d) Classe 4 [G4]

Fonte: o autor.

Figura 17 – Resultados dos testes de implementação do algoritmo de decisão $t(G) \geq n-k$



Fonte: o autor.

4.3.2 Testes de Complexidade

Supondo que exista um processo de infecção com tempo $n-k$, há pelo menos um vértice infectado no tempo 1, pelo menos um infectado no tempo 2 e, seguindo tal lógica, um infectado no tempo $n-k$. Deste modo, existem no máximo k vértices inicialmente infectados. Como existem $O(n^k)$ subconjuntos de vértices de tamanho k, e cada simulação de infecção leva um tempo $O(nm)$, a decisão $t(G) \geq n-k$ é feita em $O(m \cdot n^{k+1})$. Este cálculo de tempo de execução encontra-se em (MARCILON; SAMPAIO, 2018a).

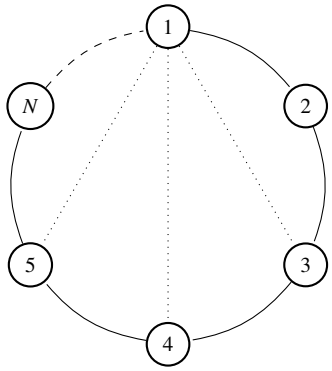
Note que, na complexidade $O(m \cdot n^{k+1})$, existem três variáveis que condicionam o tempo de execução do algoritmo. Deste modo, é impossível plotar-se um só gráfico que permita a correta visualização das relações entre as variáveis de entrada e o tempo de execução, como foi feito nas seções 4.2 e 4.1.

A estratégia aqui será, portanto, ligeiramente diferente. Irá-se testar dois pares de relação de complexidade: o par (n,k) e o par (n,m). Para cada par, será gerado um gráfico em

três dimensões (dois eixos para as variáveis e outro para o tempo de execução), utilizando-se de classes de grafos redimensionáveis. As classes para cada par de variáveis são:

1. **Par (n,k):** Aqui, faz-se necessário retirar a influência do número de arestas no tempo de execução. Isto é feito, utilizando-se de árvores binárias. Esta escolha foi feita pela fácil geração de tais grafos e pelo fato de que, para qualquer árvore, $m = n - 1$, fazendo com que a complexidade do presente algoritmo seja $O(n^{k+2})$.
2. **Par (n,m):** Neste caso, utiliza-se de um ciclo com número de arestas variáveis. Esta escolha foi feita pela fácil implementação de geradores para esta classe de grafos, podendo-se modificar o número de arestas para um mesmo número de vértices. A classe de grafos utilizada encontra-se na figura 18. As arestas internas são adicionadas ou retiradas, a depender da relação (n,m) a ser testada.

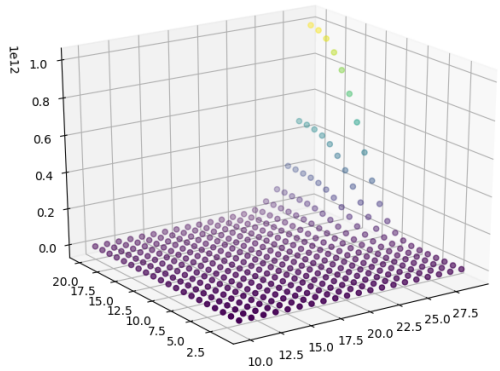
Figura 18 – Classe de grafo utilizada para o teste de complexidade no par (n,m)



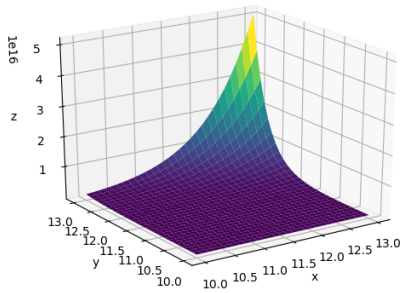
Fonte: o autor.

Os resultados gráficos encontram-se na figura 19, para o par n,k, e na figura 20, para o par n,m. Plotou-se em cada figura a superficie esperada para cada complexidade e a obtida por sucessivas execuções com as classes de grafos já mencionadas. Tem-se como resultado curvas semelhantes, sugerindo a correta complexidade de implementação.

Figura 19 – Testes para o par (n,k)



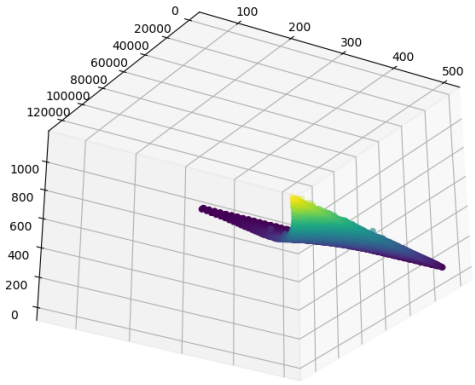
(a) Vista 1 do resultado experimental



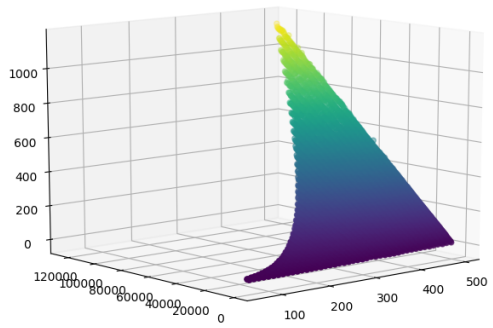
(b) Superfície de $f(x,y) = x^{(y+2)}$

Fonte: o autor.

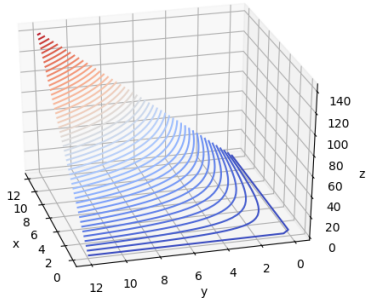
Figura 20 – Testes para o par (n,m)



(a) Vista 1 do resultado experimental



(b) Vista 2 do resultado experimental



(c) Superfície de $f(x,y) = x \cdot y$

Fonte: o autor.

4.4 Algoritmo para o problema de decisão $t(G) \geq 3$ em grafos bipartidos

Baseando-se no lema , o algoritmo proposto em (MARCILON; SAMPAIO, 2018b) verifica todas as possíveis escolhas de trios de vértices u, v, e s, simulando o processo de infecção resultante e verificando o tempo de infecção de u.

Algoritmo 5: Algoritmo para o problema de decisão $t(G) \geq 3$ em grafos bipartidos

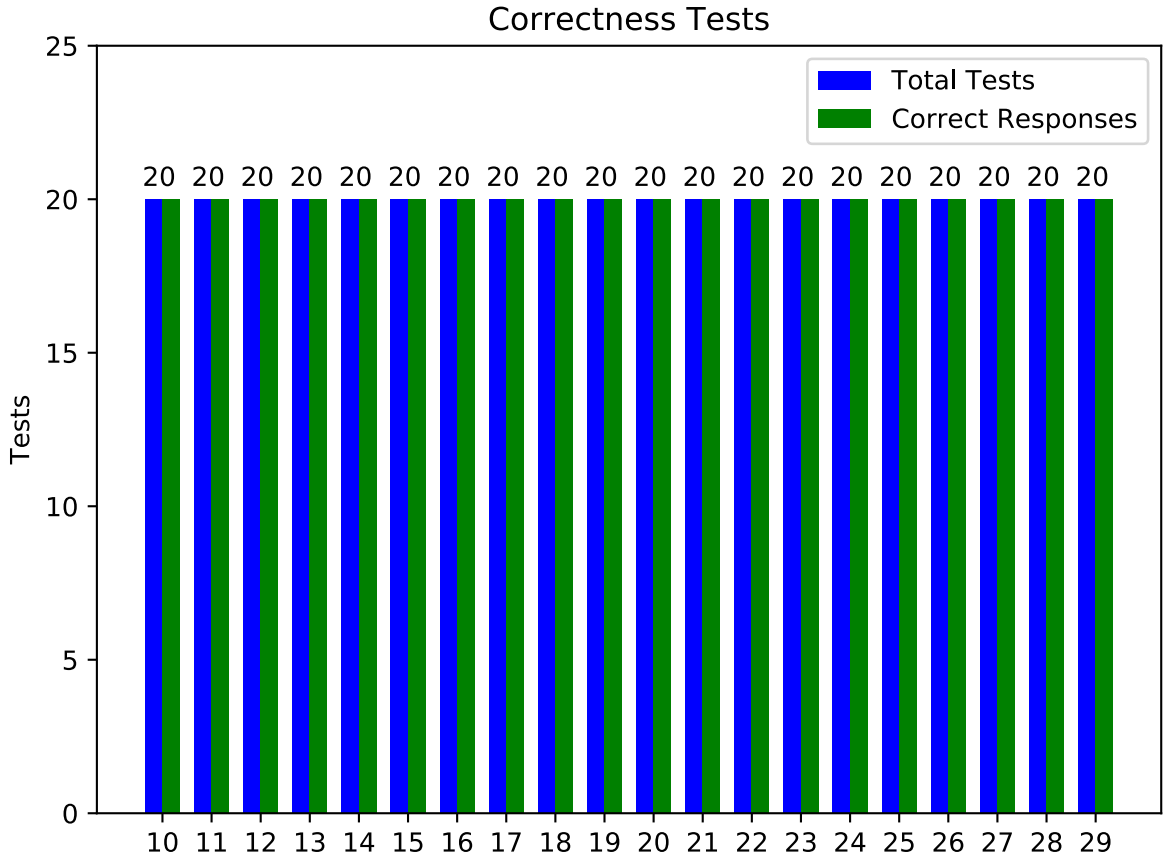
```
Função Verificar-t(G)(G):
    para cada u ∈ V(G) faça
        para cada v ∈ N(u) faça
            para cada s ∈ N_2(u) faça
                se T_0 ∪ N_{≥3}(u) ∪ {v,s} infecta u no tempo 3 então
                    retorna verdadeiro
                fim
            fim
        fim
    fim

    retorna falso
```

4.4.1 Testes da Implementação

De maneira análoga a seção 4.2, gera-se grafos randômicos como teste para o presente algoritmo. As respostas são validadas utilizando-se de uma estratégia "força bruta", ou seja, testa-se todos os conjuntos de vértices inicialmente infectados V_0, tal que $|V_0| \leq n-3$.

As cardinalidades dos grafos gerados variaram de 10 a 29, com 20 repetições por tamanho. Todos os resultados foram bem sucedidos, como demonstra o gráfico da figura 21.

Figura 21 – Resultados dos testes de implementação do algoritmo de decisão $t(G) \geq 3$ em grafos bipartidos



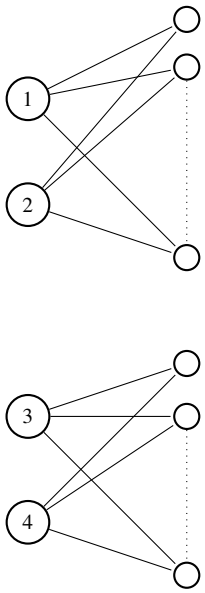
Fonte: o autor.

4.4.2 Testes de Complexidade

O presente algoritmo tem complexidade de $O(m \cdot n^3)$ (MARCILON; SAMPAIO, 2018b). De modo a verificar a implementação realizada, deveria-se, idealmente, gerar classes de grafos facilmente redimensionáveis tanto no número de vértices, como no número de arestas. Entretanto, isto não é facilmente realizável para o presente algoritmo, já que grafos mais densos, possuem, em geral, $t(G) \geq 3$, o que faz a execução do algoritmo ser finalizada logo que uma escolha de trios u, v, s com $t(u) = 3$ é encontrada.

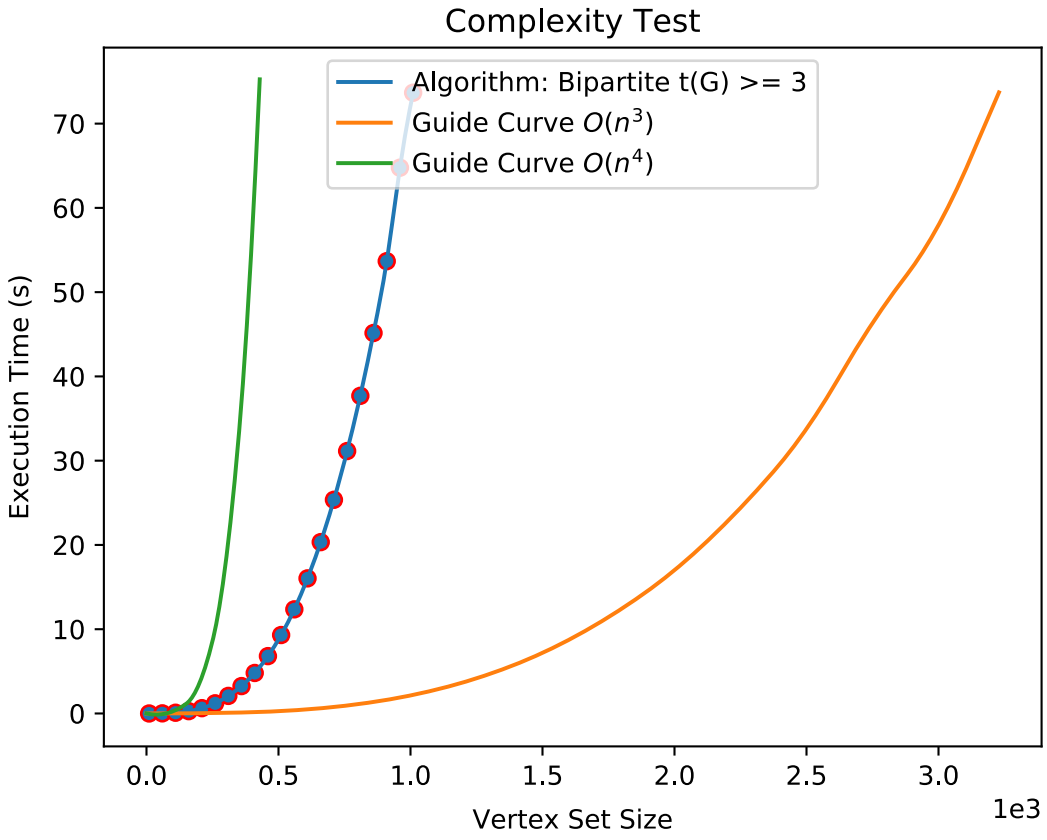
Assim, de modo a verificar-se indiretamente a correta complexidade da implementação, utilizou-se da classe de grafos da figura 22. Esta classe possui $t(G) < 3$, de modo que todos os laços do algoritmo 5 são percorridos completamente. Espera-se, portanto, uma complexidade de execução $O(n^4)$, já que $m = O(n)$. De fato, é o que ocorre, como verifica-se nos resultados da simulação, presente na figura 23.

Figura 22 – Classe de teste para geração de curva de complexidade



Fonte: o autor.

Figura 23 – Resultados dos testes de complexidade do algoritmo de decisão $t(G) \geq 3$ em grafos bipartidos



Fonte: o autor.

4.5 Algoritmo para o problema de decisão $t(G) \geq 3$ em grafos quaisquer

A partir dos lemas 2.1.3 e 2.1.4, desenvolve-se o algoritmo 6. Este algoritmo executa uma busca completa nos trios u, T_0 e F de modo a encontrar um que infecta u no tempo 3 (aplicação direta dos lemas mencionados). A seguir um pseudo-código para o algoritmo.

Algoritmo 6: Algoritmo para o problema de decisão $t(G) \geq 3$ em grafos quaisquer

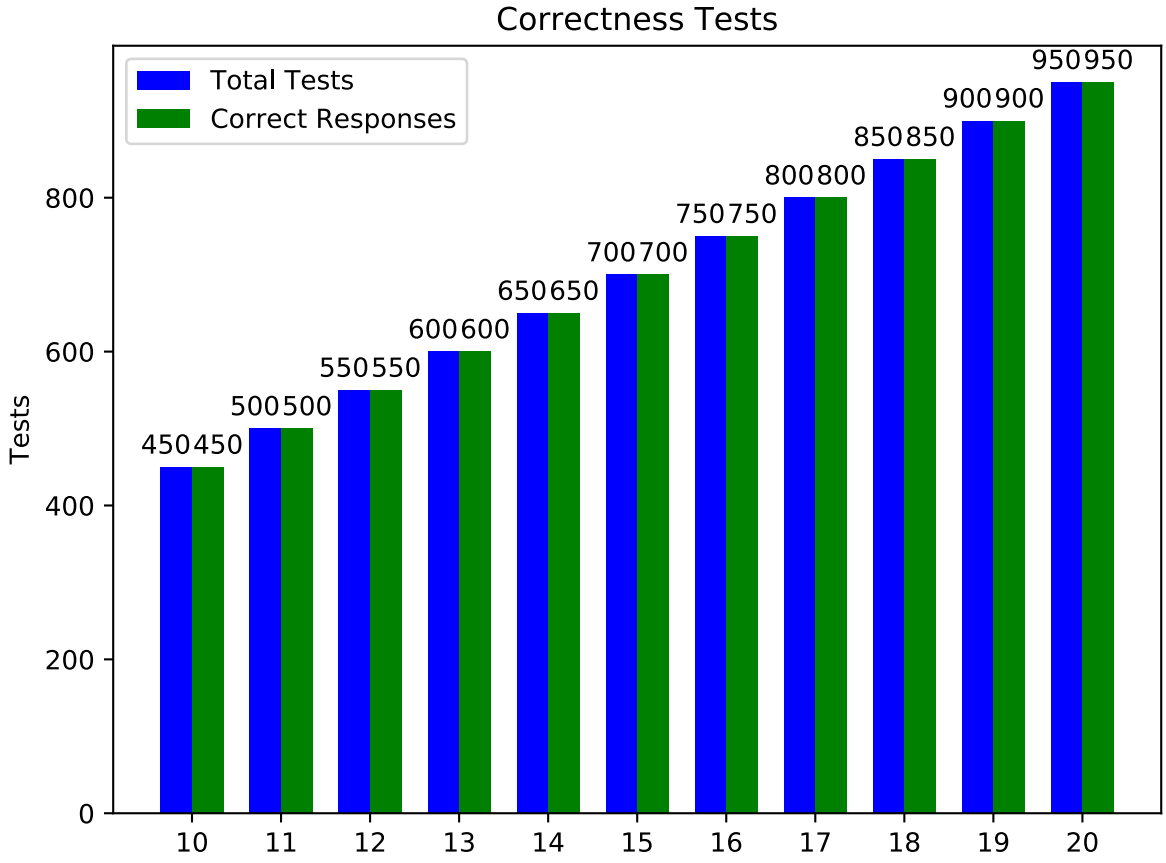
```
Função Verificar-t(G)(G):
    para cada u ∈ V(G) faça
        Encontrar um conjunto T_0 ∈ 𝒯_0^u
        para cada F ⊆ V(G), onde |F| ≤ 4 faça
            se F ∪ N_{≥3}(u) ∪ T_0 infecta u no tempo 3 então
                retorna verdadeiro
            fim
        fim
    fim

    retorna falso
```

4.5.1 Testes da Implementação

Em se tratando de grafos quaisquer, gera-se grafos randômicos variando-se o número de vértices e o de arestas. Para cada grafo, verifica-se o resultado do algoritmo desta seção com um verificador "força bruta".

Os casos de teste contiveram grafos de cardinalidades 10 a 20, com número de arestas variando no intervalo $[|V|, 2 * |V|]$. Em se tratando de um gerador randômico, repete-se cada par (n,m) 50 vezes. Todos os testes foram bem-sucedidos, como é possível verificar na figura 24.

Figura 24 – Resultados dos testes da implementação do algoritmo de decisão $t(G) \geq 3$ em grafos quaisquer



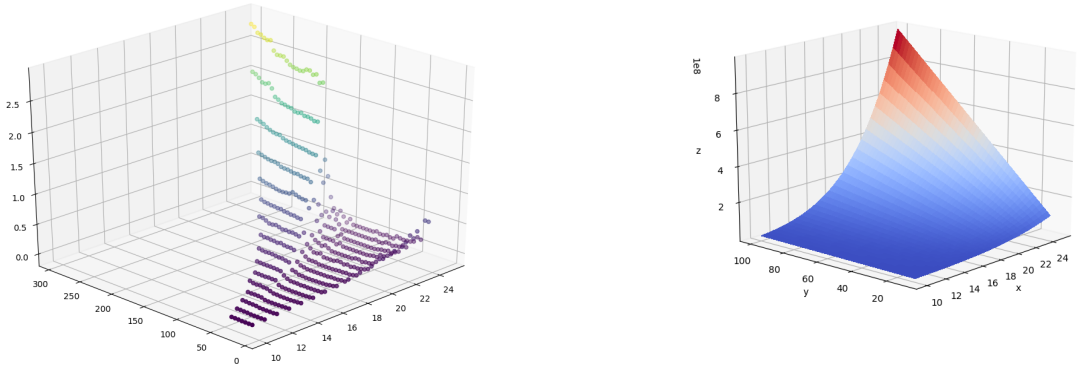
Fonte: o autor.

4.5.2 Testes de Complexidade

Em se tratando de um algoritmo para grafos quaisquer, adotou-se a estratégia a seguir. Para cada $n \in [10,25]$ e $m \in [n, \binom{n}{2}]$m, gera-se 100 grafos com n vértices e m arestas randomicamente. Para cada par (n,m), escolhe-se o pior tempo de execução dos 100 grafos gerados. Os resultados de complexidade possuem uma representação gráfica semelhante a esperada, como demonstrado na figura 25.

Ressalta-se a existência de uma leve depressão na figura 25a, inexistente na superfície de controle da figura 25b. Esta depressão não impacta o resultado do teste de complexidade (trata-se de uma depressão a superfície esperada) e é provavelmente causada pela topologia dos grafos randômicos gerados. Como não existe controle sobre os casos de testes gerados, estes podem possuir soluções rapidamente encontradas na execução do algoritmo, gerando os resultados discrepantes neste teste.

Figura 25 – Testes de tempo de execução para o algoritmo de decisão $t(G) \geq 3$ para grafos quaisquer



(a) Resultado experimental

(b) Superfície de $f(n,m) = m \cdot n^5$

Fonte: o autor.

4.6 Target Set Selection em Árvores

Este presente algoritmo resolve um problema distinto dos demais: aqui pretende-se encontrar o menor conjunto inicial de vértices que infectados resultam em uma infecção total do grafo. Neste problema, o limite $t(v)$ de vizinhos para a transição $não\ infectado \longrightarrow infectado$ é distinto para cada vértice v e, naturalmente, pertence ao intervalo $[1, d(v)]$.

Tem-se ainda que a classe de grafos é restrita à árvores (grafos acíclicos) enraizadas.

4.6.1 O Algoritmo

O ponto de partida do algoritmo é a seguinte observação: para vértices folhas f, ou seja $d(f) = 1$, tem-se que $t(f) = d(f)$. Assim, ou apenas f ou apenas seu pai estará no menor conjunto inicial infectado (*Target Set*). Isto permite que o *Target Set Selection* seja iniciado pelas folhas, não inserindo-as no conjunto de vértices inicialmente infectados. Isto não invalida o resultado, pois, se um conjunto inicial mínimo contém uma folha, o conjunto com a troca da folha pelo seu pai é de mesmo tamanho e continua infectando o grafo completamente.

A ideia do algoritmo é analisar vértices cujos filhos já estejam determinados. Aqui, um vértice é "*determinado*" caso já tenha sido descartado do conjunto inicial infectado ou tenha sido escolhido. A seleção ou descarte dos vértices é, por sua vez, feito através do valor de um limite dinâmico $t'(v)$ no momento de sua análise. Este limite dinâmico inicia-se com o valor do limite $t(v)$ e é decrementado caso algum filho seja escolhido para o *Target Set* ou algum filho possua $t'(v) \leq 0$. Tais decrementos tem como ideia quantificar o número de vizinhos "restantes"

a serem infectados para que o vértice em questão seja infectado.

Deste modo, um vértice cujos filhos já foram analisados deverá ser infectado se $t'(v) \geq 2$ ou $t'(v) \geq 1$ e v é raíz. Estas condições podem ser traduzidas como "o vértice v ainda necessita de $t'(v)$ vizinhos para ser infectado, mas só resta o pai para ser analisado". Todas as condições são traduzidas no pseudocódigo do algoritmo 7.

Ressalta-se que a ideia do algoritmo, seu pseudocódigo e a prova de sua validade encontra-se no trabalho de (CHEN, 2009).

Algoritmo 7: Algoritmo para o problema de Target Set Selection em Árvores

```
Função TargetSetSelection(T,t):
    t'(v) ← t(v), para cada v ∈ V(T)
    tss(v) ← 0, para cada v folha de T
    Enquanto existe vértice não determinado faça
        para cada v cujos filhos foram determinados faça
            se t'(v) ≥ 2 ou (v é raiz e t(v) ≥ 1) então
                tss(v) ← 1
                t'(p) ← t'(p) − 1, sendo p pai de v
            fim

            senão
                tss(v) ← 0
                se t'(v) ≤ 0 então
                    t'(p) ← t'(p) − 1, sendo p pai de v
                fim

            fim

        fim

    fim

    retorna tss
```

Correção do Algoritmo Original

A algoritmo 7 possui uma pequena correção em relação ao algoritmo original proposto em (CHEN, 2009). Na figura 26, encontra-se o algoritmo original publicado. Vê-se que existe uma condição a mais na primeira condicional do algoritmo 7 em comparação com a figura 26. Esta condição se faz necessária para casos em que a raíz da árvore deve ser marcada, mas

possui limite $t'()$ inferior a 2.

Figura 26 – Pseudocódigo original de (CHEN, 2009) para Target Set Selection em árvores

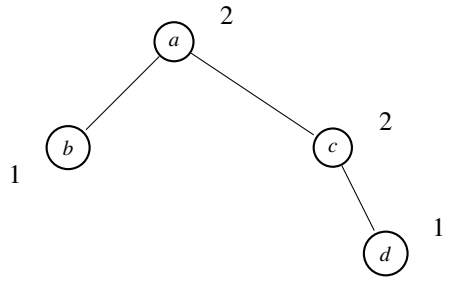
ALG-TREE

1. Let $t'(v) = t(v)$, for $v \in V$
2. Let $x(v) = 0$, for each leaf $v \in V$
3. While there is $x(v)$ not defined yet
4. for any vertex u where all $x(\cdot)$'s of its children have been defined
5. let w be u's parent
6. if $t'(u) \geq 2$
7. let $x(u) = 1$
8. let $t'(w) \leftarrow t'(w) - 1$
9. else
10. let $x(u) = 0$
11. if $t'(u) \leq 0$
12. let $t'(w) \leftarrow t'(w) - 1$
13. Output the target set $\{v \in V \mid x(v) = 1\}$

Fonte: (CHEN, 2009)

De modo a demonstrar a inconsistência no algoritmo original, considere a figura 27. Nela, encontra-se um caso simples que pontua o problema.

Figura 27 – Contra-exemplo para o algoritmo original de Target Set Selection



Fonte: o autor.

Seguindo o algoritmo original, teríamos a sequência de operações abaixo:

1. **Descarte das Folhas para o Target Set:** Inicialmente, descarta-se os vértices b e d.

2. **Primeiro vértice pronto:** A seguir, o vértice c possui todos os filhos analisados. Como $t'(c) = 2$, o vértice c é adicionado ao Target Set. Mais ainda, $t'(a)$ é decrementado, tendo valor $t'(a) = 1$.
3. **Segundo vértice pronto:** Por fim, a raíz a é analisada (filhos b e c já processados). Como $t'(a) = 1$, a é descartado do Target Set
4. **Resultado:** Target Set = $\{c\}$.

Claramente, o conjunto resultante acima não percola o grafo completamente. A inconsistência ocorre no processamento da raíz. Mais uma vez, retorna-se à ideia de inserção de um vértice v ao Target Set: verificados todos os filhos, se ainda assim $t'(v) \geq 2$, ele deve ser inserido, pois só existe o pai dele para ser verificado. Porém, no caso da raíz, não há pai a ser verificado. Portanto, a condição para a inserção da raíz será de uma unidade a menos, ou seja, $t'(v) \geq 1$. Justamente esta condição é adicionada ao algoritmo 7.

4.6.2 Testes da Implementação

Em se tratando de árvores, escolheu-se como processo de validação a criação de casos de teste randômicos, verificados por uma estratégia de força bruta.

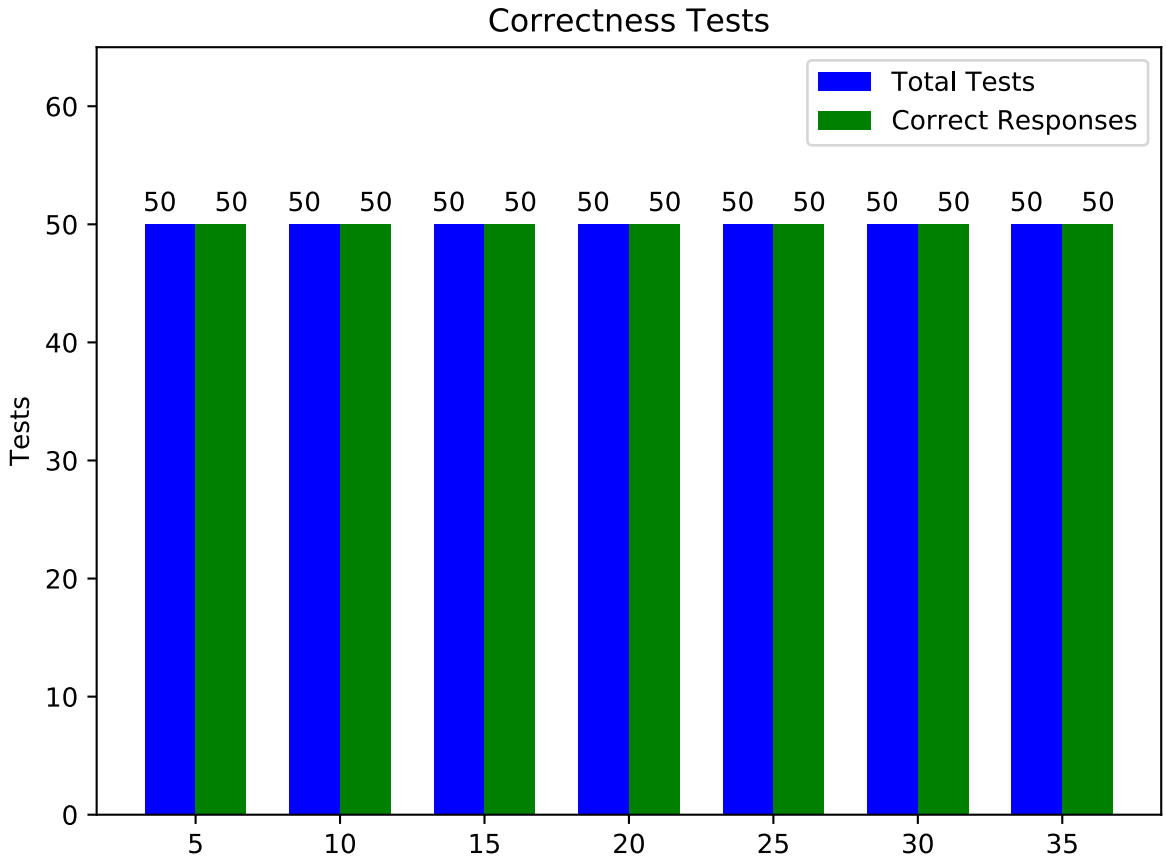
A criação de árvores randômicas é relativamente simples. Seu passo a passo pode ser descrito como:

1. Escolhe-se o número de vértices n.
2. Como para árvores $m = n - 1$, repete-se enquanto $m \neq n - 1$ vezes a:
    a) Criação de uma aresta randômica.
    b) Verificação de ciclo. Caso positiva, retirada da aresta. Caso negativa, adição do número de arestas.

A validação por força bruta, por sua vez, é a busca completa por conjuntos iniciais de tamanho iteradamente superior que infectam o grafo completamente.

Utilizando-se dos procedimentos descritos, variou-se n no intervalo $[5, 35]$, com saltos de 5 vértices. Como trata-se de casos de teste randômicos, para cada n executou-se 50 repetições do teste, totalizando 350 verificações. Todos os testes foram bem sucedidos. O gráfico de validação encontra-se na figura 28.

Figura 28 – Resultados dos testes de implementação do algoritmo de TSS em árvores



Fonte: o autor.

4.6.3 Testes de Complexidade

Este algoritmo não possui análise de complexidade em (CHEN, 2009), porém esta verificação pode ser feita sem muito esforço com o pseudo-código do algoritmo 7. Abaixo, uma análise das principais rotinas do algoritmo:

1. **Inicialização de t'**(): Isto deve ser feito para cada vértice uma vez. Logo, complexidade $O(n)$.
2. **Descarte das folhas para o Target Set:** Basta percorrer todas as folhas. Mais uma vez, complexidade $O(n)$.
3. **Laço de escolha dos vértices restantes:** Note que cada vértice só está em posição de ter todos os filhos definidos uma única vez, ou seja, um vértice só será escolhido ou descartado uma única vez. Assim, esta última rotina executa O(n) vezes (mais precisamente, a cardinalidade do conjunto de vértices não folha da árvore).

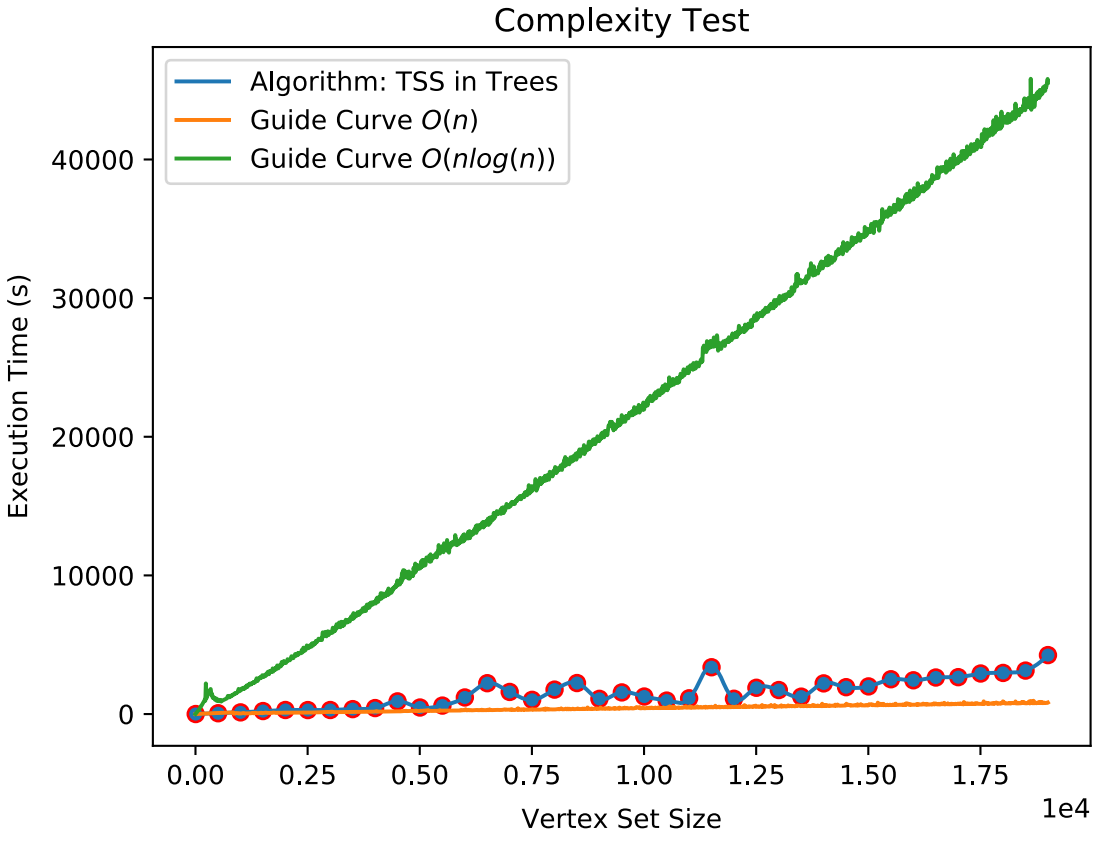
Deste modo, conclui-se que trata-se de um algoritmo linear.

A estratégia de validação da complexidade da implementação foi desenhada de modo a reutilizar o gerador de árvores randômicas mencionado na verificação da implementação. Assim, executou-se repetidamente a implementação para árvores randômicas de tamanhos variados. Mais especificamente, variou-se a cardinalidade do conjunto de vértices no intervalo $[5, 19005]$, com aumento de 500 vértices por iteração. Para cada uma destas cardinalidades, repetiu-se a geração de árvores e a tomada de tempo de execução 100 vezes e escolheu-se o pior desempenho nestas tentativas, como um forma de descartar a execução de casos de teste triviais.

Posteriormente, interpolou-se as medidas de tempo de execução e comparou-se com curvas de tempo de execução para as complexidades $O(n)$ e $O(n.log(n))$. Esta comparação tomou como base o gráfico da figura 29.

Da comparação das curvas de tempo de execução, conclui-se que a implementação do algoritmo parece, de fato, possuir complexidade linear.

Figura 29 – Tempos de execução dos testes de complexidade para o algoritmo de TSS em árvores



Fonte: o autor.

4.7 Tempo Máximo de Infecção em Árvores

A apresentação deste último algoritmo será ligeiramente mais extensa, pelo fato de ser um algoritmo não publicado. Trata-se da determinação do tempo máximo de infecção em árvores (grafos acíclicos). Como trata-se de um algoritmo novo, escolhe-se por apresentá-lo completamente na seção de resultados, incluindo a demonstração de lemas de sustentação de sua corretude.

4.7.1 O Algoritmo

A ideia deste algoritmo está diretamente relacionada ao lema que segue.

Observação

No lema a seguir, denomina-se $H(S)$ o *Convex Hull* do conjunto de vértices S, isto é, o conjunto final de vértices infectados considerando-se S o conjunto de nós infectados em tempo zero (denomina-se este conjunto de vértices inicialmente infectados por *Hull Set*).

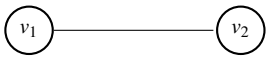
Lema 4.7.1. *Sendo G uma árvore, $\forall v \in V(G), \exists S : H(S) = S$ e $H(S \cup \{v\}) = V(G)$*

Demonstração. Por indução forte no número de vértices n de G.

- **Caso Base:** $n = 2$.

  Seja a figura abaixo a representação de G:

Figura 30 – Representação de G no caso base



Fonte: o autor.

  Sem perda de generalidade, faça $v = v_1$.

  Fazendo $S = \emptyset$, tem-se:

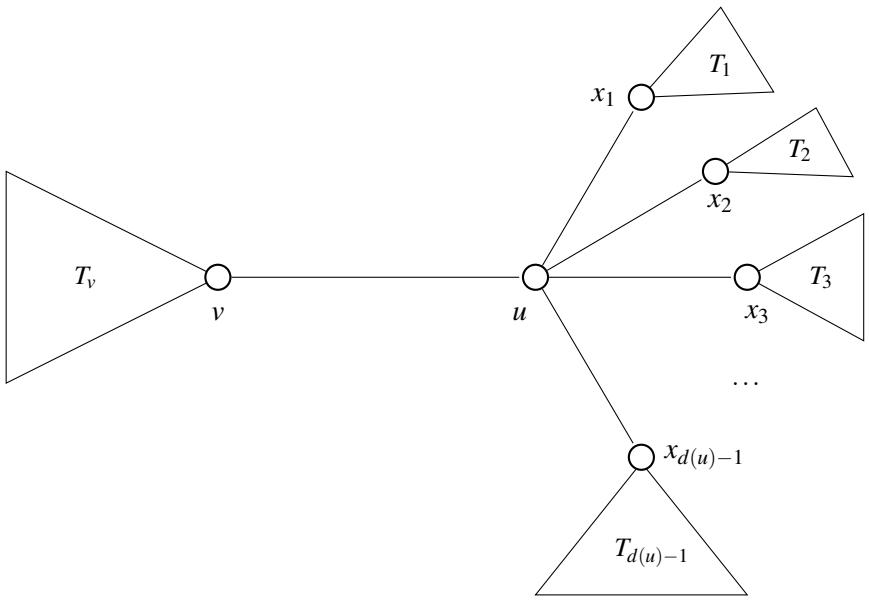
  - $H(S) = S$, por vacuidade.
  - $H(S \cup \{v\}) = V(G)$, pois $l(v_2) \in [1, d(v_2)] \Rightarrow l(v_2) = 1$.

- **Hipótese de indução:** O lema vale para $n \leq k - 1$.

- **Passo de indução:** $n = k$.

  Seja $v \in V(G)$ um vértice qualquer e u um vizinho de v. Sejam ainda $x_1, x_2, \ldots, x_{d(u)-1}$ os vértices em $N(u) - v$. Mais ainda, seja T_i a árvore enraizada pelo vértice $x_i \in N(u) - v$, além de T_v a árvore enraizada por v (veja figura 31).

Figura 31 – Representação de G no caso base



Fonte: o autor.

Como $|V(T_i)| \leq k-1$, existe, pela hipótese de indução, um conjunto S_i tal que $H(S_i) = S_i$ e $H(S_i \cup \{x_i\}) = V(T_i)$ (analogamente para T_v e S_v).

Considere ainda a subárvore de u e todos os filhos $x_{l(u)}, x_{l(u)+1}, \ldots, x_{d(u)-1}$ e chama-se de $S_{\geq l(u)}$ o conjunto que unido a u percola esta subárvore (a existência deste conjunto também deriva da hipótese de indução).

Suponha finalmente o seguinte conjunto:

$$S = T_1 \cup T_2 \cup T_3 \cdots \cup T_{l(u)-1} \cup S_{\geq l(u)} \cup S_v$$

Note que:

- $H(S) = S$: Isso ocorre, pois, a partir de S, u não é infectado (apenas $l(u) - 1$ vizinhos infectados), v não é infectado (S_v não infecta v pela hipótese de indução) e em cada subárvore restante, ou ela já está infectada (caso de $T_1, T_2, \ldots T_{l(u)-1}$) ou não existe infecção ($H(S_v) = S_v$ e $H(S_{\geq l(u)}) = S_{\geq l(u)}$).
- $H(S \cup \{v\}) = V(G)$: Com v infectado, a subárvore T_v é infectada , pois $H(S_v \cup \{v\}) =$

$V(T_v)$. O vértice u possui agora $l(u)$ vizinhos infectados, então é infectado também e, por fim, $S_{\geq l(u)} \cup \{u\}$ irá infectar a subárvore restante.

Assim, o conjunto S é tal que $H(S) = S$ e $H(S \cup \{v\}) = V(G)$.

□

O lema 4.7.1 permite uma estratégia recursiva de cálculo do maior tempo de infecção de cada vértice em uma árvore G qualquer. A ideia segue abaixo.

Suponha $s(u,v)$ o maior tempo de infecção de um vértice u supondo a subárvore $G - \{v\}$, sendo $v \in N(u)$. Mais ainda, é possível determinar o maior tempo de infecção de um vértice u em G através da seguinte regra:

$$t(u) = \begin{cases} 0, & \text{se } d(u) < l(u) \\ 1 + \text{o maior valor } s(x,u) : x \in N(u), & \text{se } d(u) \geq l(u) \end{cases}$$

O primeiro caso da equação acima é trivial, já que só é possível infectar um vértice cujo limite de infecção é superior ao seu grau se ele mesmo estiver no conjunto inicialmente infectado.

O segundo caso decorre do lema 4.7.1. Segundo este lema, é garantida a existência de um *Hull Set* S em $G - \{v\}$, tal que $H(S) = S$ e $H(S \cup \{u\}) = V(G - \{v\})$, para todo vértice $u \in G$ e $v \in N(u)$. Assim, em ideia parecida ao passo de indução da prova do lema, pode-se infectar $l(u) - 1$ vizinhos de u (e suas subárvores) em $t = 0$ e u será infectado no tempo $s(v,u) + 1$. Ressalta-se que $s(v,u)$ é o maior tempo de infecção de v na subárvore $G - \{u\}$. Assim, naturalmente, o maior $t(u)$ irá ocorrer no cenário com maior $s(v,u)$, exatamente o que foi descrito na definição de $t(u)$ apresentada.

Por fim, a definição de $s(u,v)$ decorre dos mesmos argumentos e definem a seguinte regra:

$$s(u,v) = \begin{cases} 0, & \text{se } d(u) \leq l(u) \\ 1 + \text{o maior valor } s(x,u) : x \in N(u) \setminus \{v\}, & \text{se } d(u) > l(u) \end{cases}$$

As pequenas diferenças na definição de $s(u,v)$ em comparação à regra de $t(u)$ decorrem do fato de que $s(u,v)$ não poder considerar o vizinho v, por definição.

Assim, a partir dos valores de $s(u,v)$ obtém-se, então, $t(u)$. O maior tempo de infecção de G é, finalmente, definido por $t(G) = max_{u \in V(G)} t(u)$.

A ideia do algoritmo está, portanto, apresentada. Basta calcular-se o valor de $s(u,v)$, para todo $u \in G$ e todo $v \in N(u)$. O pseudo-código de tal algoritmo encontra-se na seção do algoritmo 8.

Algoritmo 8: Algoritmo para o Problema de Tempo Máximo de Infecção em Árvores

```
Função TempoInfecção(G,l):
    l' ← l
    Preprocessamento(G, l')
    para cada u em G faça
        se t(u) já foi determinado então
            s(u,v) ← t(u), para todo v ∈ N(u)
        fim
        senão
            se d(u) ≤ l(u) então
                s(u,v) ← 0, para todo v ∈ N(u)
            fim
            senão
                s(u,v) ← 1 + max( s(x,u)), para todo x ∈ N(u) \ v
            fim

        fim

    fim

    Para todo u ∈ G com t(u) não determinado, t(u) ← max( s(x,u)), para todo x ∈ N(u)
    t(G) ← max( t(v)), para todo v em G
    retorna t(G)
```

O algoritmo 8 possui uma sub-rotina de pré-processamento. Este pré-processamento é feito de modo a relaxar os requisitos do problema. Note que o lema 4.7.1 considera que $l(v) \leq d(v)$ para todo vértice do grafo. Esta assunção é correta para a estrutura do problema de *Target Set Selection*, em que explicitamente garante-se $l(v) \in [1, d(v)]$ (esta propriedade é inclusive utilizada na resolução do problema). Para o caso de convexidades *Pn* (todo vértice é infectado se $n-1$ ou mais vizinhos estiverem infectados) a limitação de $l(v)$ não é verdadeira (todos os vértices tem o mesmo limite, mesmo se $l(v) > d(v)$).

Entretanto, a existência de limites superiores ao grau implicam uma infecção forçosa destes respectivos vértices (caso contrário, é impossível percolar todo o grafo). Assim, o pré-processamento é bastante simples. Ele adiciona os vértices com $l > d$ no conjunto inicialmente infectado e propaga a infecção destes vértices, determinando o valor de $t(v)$ já neste passo.

Assim, ao final do pré-processamento, tem-se uma árvore G em que todos os vértices v com $t(v)$ não determinado: $l(v) \leq d(v)$.

A seguir, no algoritmo 9, tem-se o detalhamento da rotina de pré-processamento.

Algoritmo 9: Pré-processamento - Algoritmo de Tempo Máximo em Árvores

```
Função Preprocessamento(G,l'):
    R ← fila vazia
    para cada v em G com l(v) > d(v) faça
        t(v) ← 0
        para cada u vizinho de v faça
            l'(u) ← l'(u) − 1
            se l'(u) = 0 então
                insere u em R
                t(u) ← max( t(x)) + 1, onde x ∈ N(u)
            fim
        fim

    fim

    Enquanto R não vazia faça
        x ← próximo de R
        para cada u vizinho de x faça
            l'(u) ← l'(u) − 1
            se l'(u) = 0 então
                insere u em R
                t(u) ← max( t(y)) + 1, onde y ∈ N(u)
            fim
        fim

    fim
```

Análise de Complexidade

Analisa-se os algoritmos 8 e 9.

Começa-se pelo pré-processamento, ou seja, o algoritmo 9. Supondo que todo vértice possui $l(v) > d(v)$, tem-se que o primeiro laço irá executar $n = O(n)$ vezes e o laço interno irá executar no máximo $\sum_{v \in G} d(v) = 2 \cdot m = O(n)$ vezes. O laço "enquanto", analogamente, irá executar no máximo $n = O(n)$ vezes e o laço interno no máximo $\sum_{v \in G} d(v) = 2 \cdot m = O(n)$ vezes. Conclui-se, assim, que o pré-processamento tem complexidade $O(n)$.

Prossegue-se, então, com o algoritmo 8. Após o pré-processamento, existe um laço que executa em todo vértice do grafo G. Internamente, existe atribuições que estão "mascaradas", porém são laços também (todos as atribuições "para todo"). De qualquer maneira, todas as atribuições serão executadas no máximo $\sum_{v \in G} d(v)$ vezes, ou seja, $O(n)$. A atribuição de $s(u,v)$ no último caso também possui a mesma complexidade. Assim, também trata-se de um algoritmo $O(n)$.

No todo, portanto, o algoritmo de cálculo de tempo máximo de infecção em árvores possui complexidade $O(n)$, ou seja, é linear.

4.7.2 Testes de Implementação

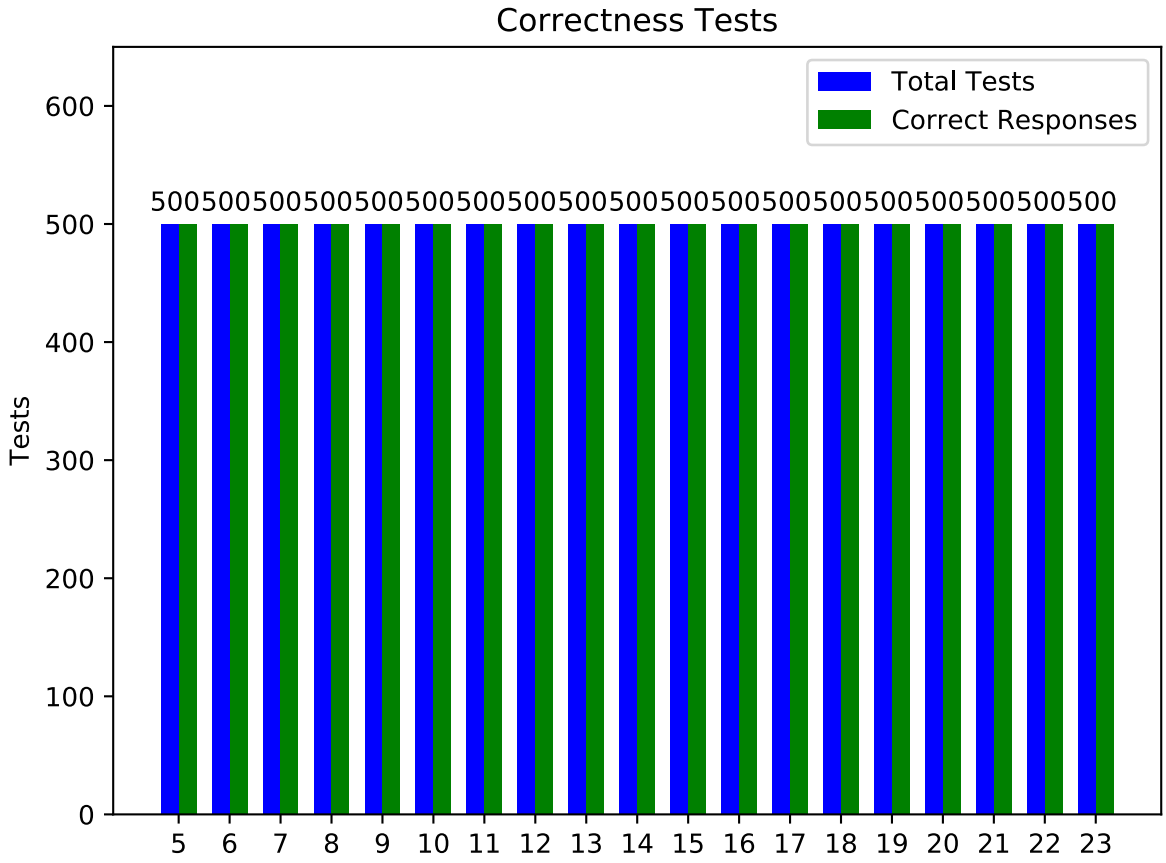
A estratégia de teste de implementação para o presente algoritmo foi similar ao do algoritmo anterior. Utilizou-se do gerador de árvores randômicas já implementado como gerador de casos de teste. As saídas dos casos de teste foram validados por um verificador "força bruta" que testa todos os subconjuntos de vértices do grafo como conjunto inicial infectado e seleciona aquele cujo tempo de infecção total seja o maior.

Especificamente, foram geradas árvores com número de vértices variando de no intervalo [5,23]. Para cada cardinalidade do conjunto de vértices foram geradas 50 árvores randômicas que, por sua vez, foram validados com 10 conjuntos randômicos de limite de infecção de cada vértice, totalizando, portanto, 500 testes por cardinalidade de árvore ou 9500 testes no total. Este passo extra de randomização se faz necessário, pois é possível que a implementação tenha problemas com limites de infecção específicos.

Mais uma vez, ressalta-se que a cardinalidade do conjunto de vértices deve ser relativamente baixo para viabilizar a validação "força-bruta" que é extremamente lenta.

Os resultados dos testes foram todos bem sucedidos, como pode-se ver no gráfico da figura 32.

Figura 32 – Testes da Implementação para o Algoritmo de Tempo Máximo em Árvores



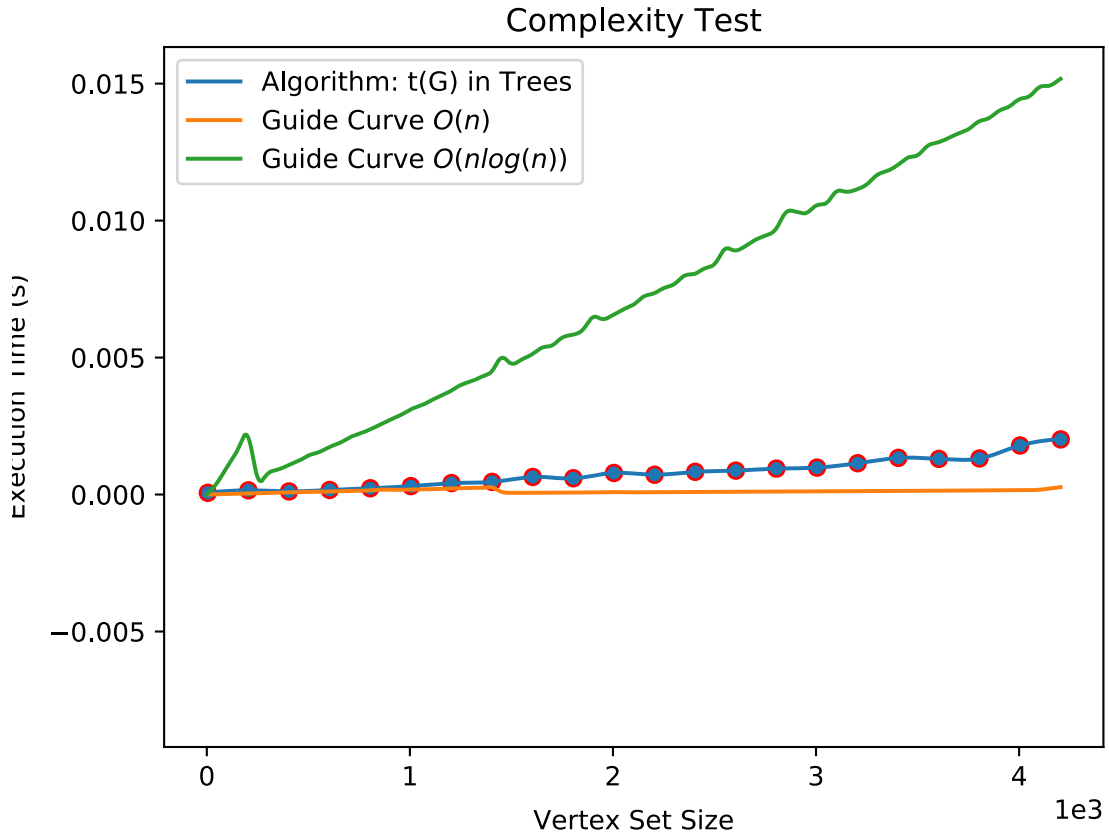
Fonte: o autor.

4.7.3 Testes de Complexidade

Por fim, os testes de complexidade. Estes foram conduzidos da mesma maneira que no algoritmo anterior: gera-se árvores randômicas para determinados números de vértices e compara-se com linhas guia de complexidade.

Este algoritmo tem complexidade teórica $O(n)$ e, de fato, atingiu um resultado de tempo de execução esperado. Esta validação foi feita com base no gráfico da figura 33. A linha de tempo de execução está na mesma ordem de magnitude da linear.

Figura 33 – Tempos de execução dos testes de complexidade para o algoritmo de t(G) em árvores



Fonte: o autor.

5 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi apresentado um estudo algorítmico breve sobre alguns problemas de infecção em grafos. Em especial, foram desenvolvidas implementações de problemas até o momento unicamente apresentados de maneira teórica.

Em análise dos objetivos, conclui-se que todos foram totalmente atingidos. Estratégias de verificação foram desenvolvidas e as implementações foram bem-sucedidas em todos os testes aqui propostos. O trabalho foi, na verdade, levemente além do esperado, na medida que corrigiu um algoritmo já publicado e apresenta um novo algoritmo para o cálculo do maior tempo de infecção para árvores.

Naturalmente, em razão de limitações computacionais, deve-se ressaltar que os testes de verificação das implementações não possuem uma abrangência muito grande. Assim, testes mais profundos no que diz respeito à correta resposta das implementações seriam interessantes.

Quanto às análises de tempo de execução, ressalta-se a importância dos gráficos de complexidade empíricos, na medida que implementações de algoritmos propostos teoricamente são corriqueiras e uma comparação entre a complexidade teórica e prática é de grande utilidade didática.

Assim, muitos pontos podem ser foco de estudos futuros, como o desenvolvimento ou a aplicação de estratégias de teste mais sofisticadas aos algoritmos aqui desenvolvidos. A expansão de implementações de problemas de infecção com algoritmos já propostos também pode ser de grande utilidade, seja ela meramente didática em estudos algorítmicos do problema ou real em que algum cenário que necessite da execução de tais algoritmos.

REFERÊNCIAS

BERGER, N.; BORGS, C.; CHAYES, J. T.; SABERI, A. On the spread of viruses on the internet. In: **Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms**. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005. (SODA '05), p. 301–310. ISBN 0-89871-585-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1070432.1070475>.

BROADBENT, S. R.; HAMMERSLEY, J. M. Percolation processes: I. crystals and mazes. **Mathematical Proceedings of the Cambridge Philosophical Society**, Cambridge University Press, v. 53, n. 3, p. 629–641, 1957.

CHALUPA, J.; LEATH, P. L.; REICH, G. R. Bootstrap percolation on a bethe lattice. **Journal of Physics C: Solid State Physics**, IOP Publishing, v. 12, n. 1, p. L31–L35, jan 1979. Disponível em: <https://doi.org/10.1088%2F0022-3719%2F12%2F1%2F008>.

CHEN, N. On the approximability of influence in social networks. **SIAM Journal on Discrete Mathematics**, v. 23, n. 3, p. 1400–1415, 2009. Disponível em: <https://doi.org/10.1137/08073617X>.

Ganesh, A.; Massoulie, L.; Towsley, D. The effect of network topology on the spread of epidemics. In: **Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.** [S.l.: s.n.], 2005. v. 2, p. 1455–1466 vol. 2.

GOLDSMITH, S. F.; AIKEN, A. S.; WILKERSON, D. S. Measuring empirical computational complexity. In: **Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 395–404. ISBN 978-1-59593-811-4. Disponível em: <http://doi.acm.org/10.1145/1287624.1287681>.

GRASSBERGER, P. On the critical behavior of the general epidemic process and dynamical percolation. **Mathematical Biosciences**, v. 63, n. 2, p. 157 – 172, 1983. ISSN 0025-5564. Disponível em: <http://www.sciencedirect.com/science/article/pii/0025556482900360>.

MARCILON, T.; SAMPAIO, R. The maximum infection time of the p3 convexity in graphs with bounded maximum degree. **Discrete Applied Mathematics**, v. 251, p. 245 – 257, 2018. ISSN 0166-218X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0166218X18303147>.

MARCILON, T.; SAMPAIO, R. The maximum time of 2-neighbor bootstrap percolation: Complexity results. **Theoretical Computer Science**, v. 708, p. 1 – 17, 2018. ISSN 0304-3975. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0304397517307429>.

NISS, M. History of the lenz-ising model 1920–1950: From ferromagnetic to cooperative phenomena. **Archive for History of Exact Sciences**, v. 59, n. 3, p. 267–318, Mar 2005. ISSN 1432-0657. Disponível em: <https://doi.org/10.1007/s00407-004-0088-3>.

SCHIFF, J. **Cellular automata : a discrete view of the world**. Hoboken, N.J: Wiley-Interscience, 2008. ISBN 978-1-118-03063-9.