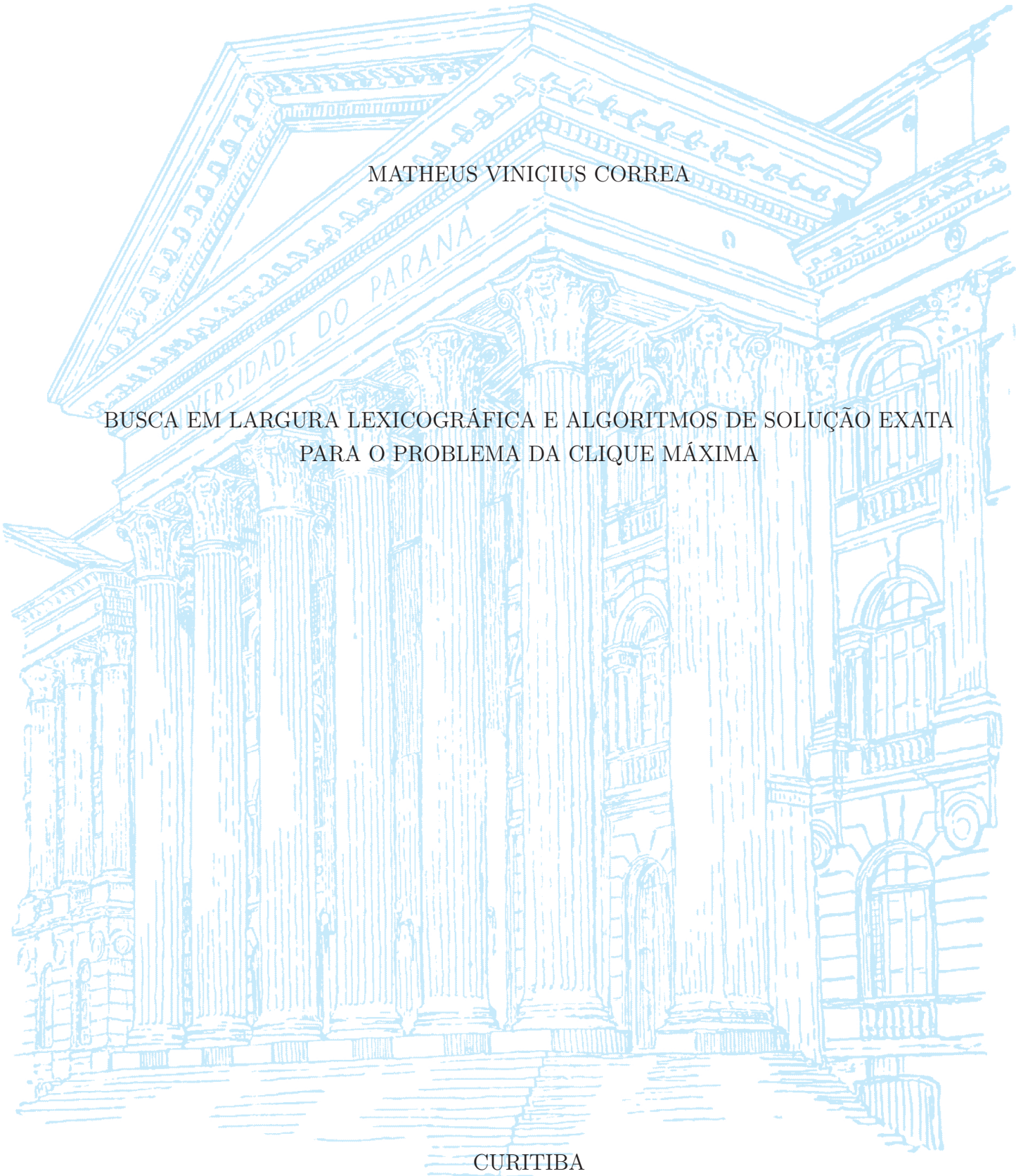
UNIVERSIDADE FEDERAL DO PARANÁ



MATHEUS VINICIUS CORREA

BUSCA EM LARGURA LEXICOGRÁFICA E ALGORITMOS DE SOLUÇÃO EXATA PARA O PROBLEMA DA CLIQUE MÁXIMA

CURITIBA
2020

MATHEUS VINICIUS CORREA

BUSCA EM LARGURA LEXICOGRÁFICA E ALGORITMOS DE SOLUÇÃO EXATA PARA O PROBLEMA DA CLIQUE MÁXIMA

Dissertação apresentada como requisito parcial para a obtenção do título de Mestre em Informática no Programa de Pós-Graduação Informática, Departamento de Informática, Setor de Ciências Exatas da Universidade Federal do Paraná.

Orientador: Prof. Dr. Renato Carmo
Coorientador: Prof. Dr. Alexandre Prusch Züge

CURITIBA
2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

C824b Correa, Matheus Vinicius
Busca em largura lexicográfica e algoritmos de solução exata para o problema da clique máxima [recurso eletrônico] / Matheus Vinicius Correa. – Curitiba, 2020.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Renato Carmo.
Coorientador: Alexandre Prusch Züge.

1. Algoritmos. 2. Algorítmos computacionais. I. Universidade Federal do Paraná. II. Carmo, Renato. III. Züge, Alexandre Prusch. IV. Título.

CDD: 518.1

Bibliotecária: Vanusa Maciel CRB- 9/1928

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **MATHEUS VINICIUS CORREA** intitulada: **Busca em Largura Lexicográfica e Algoritmos de Solução Exata Para o Problema da Clique Máxima**, sob orientação do Prof. Dr. RENATO JOSÉ DA SILVA CARMO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 20 de Agosto de 2020.

Assinatura Eletrônica
21/08/2020 10:35:22.0
RENATO JOSÉ DA SILVA CARMO
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
25/08/2020 14:14:51.0
JAIME COHEN
Avaliador Externo (UNIVERSIDADE ESTADUAL DE PONTA GROSSA)

Assinatura Eletrônica
27/08/2020 11:47:51.0
MURILO VICENTE GONÇALVES DA SILVA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

AGRADECIMENTOS

A Deus, pela força e fé alcançadas. Ao meu orientador Prof. Dr. Renato Carmo, pela paciência e empatia durante todo esse tempo. Ao meu coorientador Prof. Dr. Alexandre Prusch Züge, pelos concelhos valiosos. Aos professores da banca de qualificação e defesa pelas sugestões. Aos meus pais Odinei e Raquel, pelo apoio incondicional. Aos meus irmãos Muryllo e Mayara, que muito me incentivaram. A todos ótimos amigos que fiz e que me ajudaram em todos momentos. A todos aqueles que passaram em meu caminho e que de alguma forma me ajudaram até aqui.

RESUMO

O problema da Clique Máxima (CM) é o problema de encontrar uma clique de tamanho máximo em um grafo dado. Existem algoritmos de solução exata que fazem uso da técnica de *branch and bound* para o CM que utilizam a coloração de vértices com menor número possível de cores como limitante superior. Assim como o CM, o problema de coloração de vértices é um problema difícil do ponto de vista computacional, contudo pode ser resolvido em complexidade de tempo linear quando restrito a certas classes de grafos. O algoritmo conhecido como Busca em Largura Lexicográfica (LexBFS) é capaz de produzir ordenações dos vértices de algumas classes de grafos que levam à solução de problemas difíceis em tempo polinomial. O objetivo deste trabalho é estudar algoritmos *branch and bound* que utilizam coloração de vértices na solução do CM, porém modificados com o Algoritmo LexBFS. Para avaliar tais modificações, foram empregados métodos da Análise de Experimental de Algoritmos. Com isso, foi possível fazer inferências sobre os resultados obtidos utilizando o método estatístico de teste de hipótese. A conclusão que se chega após a modificação com o Algoritmo LexBFS é que o espaço de busca necessário é menor, mas o tempo de execução é maior.

Palavras-chaves: Busca em Largura Lexicográfica. Algoritmos para o Problema da Clique Máxima. Análise Experimental de Algoritmos.

ABSTRACT

The Maximum Clique (MC) problem is the problem of finding a maximum size clique on a given graph. There are exact solution algorithms that use the branch and bound technique for MC and a coloring of graph vertices with as few colors as possible as the upper bound. As MC, the vertex coloring problem is a computationally difficult problem, however it can be solved in linear time complexity when restricted to certain graph classes. The algorithm known as Lexicographic Breadth-first Search (LexBFS) is capable of producing vertex ordering of some classes of graphs with properties that lead to the solution of difficult problems in polynomial time. The aim of this work is to study branch and bound algorithms that use vertex coloring to solve MC, but modified with the LexBFS algorithm. To evaluate such changes, methods of Experimental Analysis of Algorithms were used. Thus, it was possible to make inferences about the results obtained using the statistical method of hypothesis testing. The conclusion that is reached after the modification with the LexBFS algorithm is that, the search space is smaller, but the execution time was longer.

Key-words: Lexicographic Breadth-first Search. Maximum Clique Problem Algorithms. Experimental Analysis of Algorithms.

LISTA DE ILUSTRAÇÕES

FIGURA 1 – Cografo e sua coárvore onde R é a raiz, as folhas correspondem a um vértice no grafo e seus nós internos representam as operação de união (0) e junção (1). 20

FIGURA 2 – Caracterização de uma ordenação LexBFS 21

FIGURA 3 – Primeiros passos do algoritmo LexBFS(G) no grafo G da Figura 1. Inicialmente a lista de partes contém apenas uma parte, o conjunto de vértices. O resultado da ordenação é $\pi = (x, y, w, z, u, v, a, d, c, b, e)$. 24

FIGURA 4 – Estrutura de dados aproveitada pelo Algoritmo LexBFS(G) utilizando a técnica de particionamento. Partes são retângulos com pontas arredondados. Quadrados são elementos da lista duplamente ligada que representam vértices do grafo. 24

FIGURA 5 – Alguns passos de LexBFS$^+(G, \pi)$ no grafo da Figura 1 e a ordenação $\pi = (x, y, w, z, u, v, a, d, c, b, e)$ como entrada, previamente computada por LexBFS(G). O primeiro passo apenas inverte a ordenação de π. Em seguida, o Algoritmo LexBFS é chamado com os vértices inicialmente ordenados como π, dada como entrada. 28

FIGURA 6 – O grafo Bull 28

FIGURA 7 – A coárvore do cografo G da Figura 1. As áreas escuras representam subárvores T^x_{0i} e T^x_{1i}. 29

FIGURA 8 – Um estado (Q, K) tem dois filhos: um que aumenta a clique Q com um vértice $x \in K$ e atualiza o conjunto de candidatos K; outro que apenas atualiza o conjunto de candidatos K removendo um vértice x. 37

FIGURA 9 – Execução do algoritmo NoBound com o grafo (9a) como entrada. (9b) apresenta a árvore de estados percorrida pelo algoritmo. 39

FIGURA 10 – Número de estados gerados para 150 grafos aleatórios $\mathcal{G}_{n,p}$ com $n = |V|$ vértices cada e $p = 1/2$. Existe uma amostra de tamanho 10 para cada número de $|V|$ vértices. 93

FIGURA 11 – Número de estados gerados por cada algoritmo $\mathcal{A}$ e $\mathcal{A}_L$ utilizando como entrada grafos cordais aleatórios. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ grafos cordais com $|V|$ vértices cada. Com exceção de um único grafo G, $|\mathcal{T}_{\mathcal{A}}(G)| = |\mathcal{T}_{\mathcal{A}_L}(G)|$ para todos os outros. 96

FIGURA 12 – Número de estados gerados por cada algoritmo $\mathcal{A}$ e $\mathcal{A}_L$ utilizando como entrada cografos aleatórios. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ cografos com $|V|$ vértices cada. Para todo grafo $|\mathcal{T}_{\mathcal{A}}(G)| = |\mathcal{T}_{\mathcal{A}_L}(G)|$. 97

LISTA DE TABELAS

TABELA 1 – Número de cores computada por Greedy por duas ordenações diferentes dos vértices de instâncias DIMACS como entrada: uma ordenação π dos vértices e uma ordenação π^L dos vértices produzida por LexBFS. 61

TABELA 2 – Comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ_L com instâncias DIMACS como entrada. 62

TABELA 3 – Comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada. 67

TABELA 4 – Comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada. 72

TABELA 5 – Comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo MCR e pelo Algoritmo MCR_L com instâncias DIMACS como entrada. 77

TABELA 6 – Comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada. 82

TABELA 7 – Teste com $\mathsf{Greedy}(G)$ onde $G \in \mathcal{G}_{n,1/2}$. Quantidade de instâncias onde $c(G,\pi_L) < c(G,\pi)$, bem como, $c(G,\pi_L) = c(G,\pi)$. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho 10. 87

TABELA 8 – Teste de hipótese entre os algoritmos MCLIQ e MCLIQ_L em relação a média de estados gerados e tempo de execução. 90

TABELA 9 – Teste de hipótese entre os algoritmos MCQ e MCQ_L em relação a média de estados gerados e tempo de execução. 91

TABELA 10 – Teste de hipótese entre os algoritmos DYN e DYN_L em relação a média de estados gerados e tempo de execução. 91

TABELA 11 – Teste de hipótese entre os algoritmos MCR e MCR_L em relação a média de estados gerados e tempo de execução. 92

TABELA 12 – Teste de hipótese entre os algoritmos MCS e MCS_L em relação a média de estados gerados e tempo de execução. 92

TABELA 13 – Teste com $\mathsf{Greedy}(G)$ onde G é cordal. Como esperado (Observação 4), em nenhuma teste o número $c(G,\pi_L) > c(G,\pi)$ para toda amostra. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ grafos cordais com $|V|$ vértices cada. 94

TABELA 14 – Teste com $\mathsf{Greedy}(G)$ onde G é cografo. Como esperado (Teorema 12), em nenhuma teste com Greedy o número $c(G,\pi_L) > c(G,\pi)$ para toda amostra. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ cografos com $|V|$ vértices cada. 95

TABELA 15 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS. 117
TABELA 16 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS. 122
TABELA 17 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS. 127

TABELA 18 – Distribuição t de Student. Corpo da tabela contém os valores calculados para t_γ. Cada coluna representa um nível de confiança γ dado em proporção. Uma linha é o número ν de graus de liberdade. . . . 134

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS 14
1.2 ORGANIZAÇÃO DO TRABALHO 14
1.3 DEFINIÇÕES E NOTAÇÃO 14
1.4 GRAFOS CORDAIS, GRAFOS DE INTERVALO E COGRAFOS 16

2 BUSCA EM LARGURA LEXICOGRÁFICA 21
2.1 RECONHECIMENTO DE GRAFOS CORDAIS 25
2.2 RECONHECIMENTO DE GRAFOS DE INTERVALO UNITÁRIO . . . 26
2.3 RECONHECIMENTO DE COGRAFOS 29
2.3.1 Estrutura dos Cografos 29
2.3.2 Algoritmo de Reconhecimento 31

3 CLIQUE MÁXIMA E COLORAÇÃO DE VÉRTICES 34
3.1 O PROBLEMA DA CLIQUE MÁXIMA 34
3.2 ALGORITMOS MCBB 35
3.2.1 Algoritmo NoBound 37
3.2.2 Algoritmos Basic, CP e DF 40
3.3 COLORAÇÃO DE VÉRTICES 40
3.3.1 Algoritmos para Clique Máxima que Utilizam Coloração de Vértices . . . 44
3.3.1.1 Algoritmos CHI e CHI + DF 44
3.3.1.2 Algoritmo MCLIQ 45
3.3.1.3 Algoritmo MCQ 46
3.3.1.4 Algoritmo DYN 46
3.3.1.5 Algoritmo MCR 47
3.3.1.6 Algoritmo MCS 47
3.4 MODIFICAÇÃO COM O ALGORITMO LEXBFS 48
3.4.1 Algoritmos CHI_L e CHI_L + DF 48
3.4.2 Algoritmo MCLIQ_L 48
3.4.3 Algoritmo MCQ_L 49
3.4.4 Algoritmo DYN_L 49
3.4.5 Algoritmo MCR_L 49
3.4.6 Algoritmo MCS_L 49

4 ANÁLISE EXPERIMENTAL 51
4.1 INFERÊNCIA ESTATÍSTICA 52

4.2 INSTÂNCIAS 56
4.3 RESULTADOS EXPERIMENTAIS 58
4.3.1 Resultados para Instâncias DIMACS 58
4.3.2 Grafos Aleatórios 87
4.3.2.1 Teste de Hipótese 87
4.3.2.2 Resultados para Grafos Cordais e Cografos 93

5 CONCLUSÃO 98

REFERÊNCIAS 99

APÊNDICES 107

APÊNDICE A IMPLEMENTAÇÃO DO ALGORITMO LEXBFS . 108

APÊNDICE B AMBIENTE COMPUTACIONAL 114

APÊNDICE C RESULTADOS EXPERIMENTAIS 117
C.1 GRAFOS ALEATÓRIOS 117
C.2 GRAFOS CORDAIS 122
C.3 COGRAFOS 127

ANEXOS 133

ANEXO A DISTRIBUIÇÃO T DE STUDENT 134

1 INTRODUÇÃO

O problema da Clique Máxima (CM) é o problema de encontrar uma clique de tamanho máximo em um grafo dado. Uma abordagem recorrente na literatura (FAHLE, 2002; TOMITA; SEKI, 2003; KONC; JANEZIC, 2007; TOMITA; KAMEDA, 2007; TOMITA; SUTANI et al., 2010) é abordar o CM com soluções que fazem uso da estratégia *branch and bound* (BB).

Existem algoritmos BB para o CM que utilizam a coloração de vértices como limitante superior. O problema da coloração de vértices o problema de atribuir o menor número de cores possível os vértices de um grafo de maneira que dois vértices vizinhos não possuam a mesma cor. Assim como o CM esse é um problema $\mathcal{NP}$-difícil (GAREY; JOHNSON, 2002). Desta forma, em algoritmos BB para o CM presentes na literatura, essa coloração é feita através de algoritmos gulosos.

Apesar de tanto o CM quanto o problema da coloração de vértices serem $\mathcal{NP}$-difíceis, existem classes de grafos onde esses problemas são resolvidos por algoritmos de complexidade de tempo polinomial. Este é caso das classes dos grafos cordais (BRANDSTADT; SPINRAD et al., 1999) e dos cografos (BRANDSTADT; SPINRAD et al., 1999). Em classes como essas, o algoritmo conhecido como *Busca em Largura Lexicográfica* (LexBFS) pode ser empregado no reconhecimento de tais classes de grafos e como passo para resolver o CM e coloração de vértices em tempo polinomial (GAVRIL, 1972).

Neste trabalho foram estudadas algoritmos para o CM que utilizam a estratégia BB para resolver o problema. Mas, especificamente foram escolhidos algoritmos que fazem uso da coloração de vértices como limitante superior. Soma-se a esses o estudo de algoritmos gulosos para coloração de vértices. Ademais, foi estudado o Algoritmo LexBFS e classes de grafos onde esse algoritmo é capaz de resolver tanto o CM quanto a coloração de vértices em complexidade de tempo linear.

Com isso, foi possível fazer uma alteração em algoritmos BB que resolvem o CM de maneira que seja computada uma ordenação com LexBFS. Essa ordenação por sua vez é utilizada como entrada para algoritmos gulosos que colorem os vértices seguindo essa ordem pré-computada. Feito isso, cada algoritmo modificado foi avaliado em comparação com suas versões originais, considerando diferentes instâncias de grafos.

Uma maneira de avaliar tais modificações é utilizar conceitos da Análise Experimental de Algoritmos. Por essa abordagem é possível tratar algoritmos como objetos de laboratório, focando no controle de parâmetros, isolamento de componentes chave, construção de um modelo e análise estatística. Nesse trabalho, além de fazer análises apenas quantitativas, foi utilizado método estatístico de teste hipótese. Por essa metodologia foi possível fazer inferências sobre parâmetros de desempenho de algoritmos. A escolha dos métodos é baseada no tipo de instâncias de grafos que foi possível ter à mão.

1.1 OBJETIVOS

Os objetivos deste trabalho estão listados em seguida.

- Estudar o Algoritmo LexBFS.
- Estudar algoritmos BB para o CM que utilizam coloração de vértices, porém modificados com LexBFS.
- Aplicar conceitos de Análise Experimental de Algoritmos, produzindo resultados experimentais.
- Fazer inferências estatística utilizando o método estatístico de teste de hipótese.

1.2 ORGANIZAÇÃO DO TRABALHO

O restante do texto está organizado como segue. As próximas seções apresentam definições e notação utilizadas no decorrer do texto bem como algumas classes de grafos. O Capítulo 2 aborda os algoritmos de Busca em Largura Lexicográfica. O Capítulo 3 discorre sobre o CM e coloração de vértices, apresentando no final o algoritmo para o CM modificados com LexBFS. O Capítulo 4 apresenta o conceito de Análise Experimental de Algoritmos e resultados experimentais. Por fim, o Capítulo 5 é a conclusão do trabalho.

1.3 DEFINIÇÕES E NOTAÇÃO

Dados um conjunto S e um inteiro k, denotamos por $\binom{S}{k}$ o conjunto de subconjuntos de S de tamanho k. O *tamanho* ou *cardinalidade* de um conjunto S, denotado por $|S|$, é o número de elementos presentes em S.

Um *grafo* G é um par ordenado de conjuntos $(V(G), E(G))$ tal que $V(G)$ é finito e $E(G) \subseteq \binom{V(G)}{2}$. Os elementos do conjunto $V(G)$ são chamados *vértices* de G, e os elementos do conjunto $E(G)$ são chamados de *arestas*. Quando estiver claro no contexto, se G é um grafo, será adotado V e E ao invés de $V(G)$ e $E(G)$, omitindo G.

O complemento de um grafo G, denotado por $\overline{G}$, chamado de *grafo complementar*, é o grafo $\overline{G} \coloneqq (V(G), \binom{V(G)}{2} \setminus E(G))$.

Um grafo $G = (V, E)$ tem seu conjunto de arestas E composto por pares não ordenados $\{x, y\}$ onde $x, y \in V$. Convencionamos a notação xy para nos referir à aresta $\{x, y\} \in E$.

Dizemos que um vértice x é *vizinho* de um vértice y em G se $xy \in E$. A *vizinhança* de um vértice x em G, denotada $N(x)$, é conjunto de seus vizinhos em G, isto é, $N(x) \coloneqq \{y \in V | xy \in E\}$. A *não vizinhança* de um vértice x, denotada $\overline{N}(x)$, é o conjunto $\overline{N}(x) \coloneqq \{y \in V \mid xy \notin E\}$. A *vizinhança fechada*, denotada por $N[x]$, é formada por $N(x) \cup x$.

O *grau* de um vértice x em G, denotado por $d(x)$, é o tamanho da sua vizinhança, isto é, $d(x) \coloneqq |N(x)|$. O maior grau de um vértice em G é denotado $\Delta(G)$.

Um *passeio* é uma sequência de vértices e arestas $(x_0, x_1, \ldots, x_l)$ do grafo G onde x_{i-1} e x_i são vizinhos para todo $1 \leq i \leq l$. Um *caminho* é um passeio onde seus vértices não repetem. O *tamanho* de um caminho é um inteiro l. Denotamos por P_l um caminho de tamanho l. Uma *trilha* é um passeio que não repete arestas. Um *ciclo* é uma trilha onde $l \geq 3$, $x_0 = x_l$ e demais vértices não reptem. Denotamos C_l um ciclo de tamanho l. Um C_3 é chamado de *triangulo*.

Um grafo G é *conexo* se existe um caminho entre todo par de vértices.

Dizemos que o grafo H é *subgrafo* de G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Dado um conjunto $S \subseteq V(G)$, o *subgrafo induzido* por S em G é denotado por $G[S]$, dado por $G[S] \coloneqq (S, E(G) \cap \binom{S}{2})$.

Um grafo $G = (V, E)$ é *completo* se $E = \binom{V}{2}$. Denotamos por K_n um grafo completo com n vértices. Uma *clique* Q em G é um subconjunto de V tal que $G[Q]$ é completo. Uma clique em G é *maximal* se não é subconjunto próprio de qualquer outra clique em G. Uma clique em G é *máxima* se possui o maior número de vértices dentre todas cliques de G. Uma clique de tamanho k é chamada k-clique. O maior k tal que existe k-clique em G é chamado *número de clique*, denotado $\omega(G)$. Um subconjunto $S \subseteq V(G)$ é independente se $E(G[S]) = \emptyset$. O *conjunto independente máximo* é o conjunto independente com maior número de vértices dentre todos conjuntos independentes em G, denotado por $\alpha(G)$. Nota-se que uma clique no grafo G é um conjunto independente no grafo complementar $\overline{G}$.

O problema computacional da clique máxima é enunciado em seguida.

Clique Máxima(CM)
Instância : Um Grafo G.
Resposta : Uma clique de tamanho máximo de G.

Dado um inteiro k, uma *k-coloração* dos vértices do grafo $G = (V, E)$ é uma função sobrejetora $c : V \to \{1, 2, \ldots, k\}$ tal que $c(x) \neq c(y)$ se $xy \in E$ para todo x, y. Dizemos que $c(x)$ é a *cor* do vértice x. Os conjuntos $c^{-1}(j)$ são chamados *classe de cores* da coloração, onde $1 \leq j \leq k$. Uma *coloração dos vértices* é uma k-coloração de G para algum k. O *número cromático* de G, dentado $\chi(G)$, é definido como sendo o menor k tal que G possui uma k-coloração.

O problema computacional referente à coloração de vértices é descrito como se segue.

Coloração de Vértices
Instância : Um Grafo G.
Resposta : Uma $\chi(G)$-coloração de G.

Um grafo G é *perfeito* se para todos subgrafos induzidos H de G temos que $\omega(H) = \chi(H)$.

Dado um grafo $G = (V, E)$, um vértice x é *simplicial* se $G[N(x)]$ é uma clique. Uma ordenação $(x_1, x_2, \ldots, x_n)$ dos vértices em V é uma *ordem perfeita de eliminação* se para todo $i \in \{1, 2, \ldots, n\}$ o vértice x_i é simplicial no grafo $G_i[\{x_1, x_2, \ldots, x_i\}]$.

Dois grafos são disjuntos se não têm vértices em comum. A *união* de dois grafos disjuntos G e H é o grafo $G \cup H$ com os vértices $V(G) \cup V(H)$ e arestas $E(G) \cup E(H)$. A operação de *composição em paralelo* ou *união disjunta* de k grafos $G_1, G_2, \ldots, G_k$ define o grafo $G_1 \cup G_2 \cup \cdots \cup G_k$. A operação *composição em série* ou *junção* dos grafos $G_1, G_2, \ldots, G_k$ é o grafo $G_1 \cup G_2 \cup \cdots \cup G_k$ adicionadas as arestas xy onde todo par de vértices x e y pertencente a diferentes grafos G_i e G_j.

Uma *partição* de V, denotada $P = \{X_1, X_2, \ldots, X_k\}$, é um conjunto de subconjuntos disjuntos de V tal que $\bigcup_{X \in P} X = V$. Os elementos de P são chamados *partes*. Uma partição *ordenada* de V, denotada $P = [X_1, X_2, \ldots, X_k]$, temos que $x_i <_P x_j$ se e somente se $x_i \in X_i, x_j \in X_j$ e $i < j$. Além disso, temos que $X_i < X_j$ em P se e somente se $i < j$ ou $X_i \geq X_j$ em P se e somente se $i \geq j$.

Um grafo $G = (V, E)$ é *bipartido* se seu conjunto de vértices pode ser particionado em duas partes X e Y. Se todo vértice em X é vizinho de todo vértice em Y, então G é um grafo *bipartido completo*. Um grafo bipartido completo onde $m = |X|$ e $n = |Y|$ será denotado $K_{m,n}$.

Uma *árvore* é um grafo conexo sem ciclos. Um subgrafo conexo de uma árvore é chamado de *subárvore*. Uma *árvore enraizada* é uma árvore onde um vértice é chamado *raiz*. Considerando o único caminho em uma árvore enraizada da raiz até um vértice x, se a última aresta desse caminho é a aresta yx então y é *pai* de x, e x é *filho* de y. Um vértice sem filhos é uma *folha*.

Seja $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ uma família de subconjuntos de um conjunto. Um grafo $G = (V, E)$ é chamado de *grafo de intersecção de* $\mathcal{F}$ se existe uma bijeção entre o conjunto de vértices $V = \{x_1, x_2 \ldots, x_n\}$ do grafo G e os conjuntos em $\mathcal{F}$ tal que x_i e x_j são adjacentes se e somente se $F_i \cap F_j \neq \emptyset$, para $1 \leq i, j \leq n$. No caso em que existe uma árvore $\mathcal{T}$ tal que todo conjunto em $\mathcal{F}$ corresponde ao conjunto de vértices de uma subárvore de $\mathcal{T}$, então G é chamado de *grafo de intersecção de subárvores de uma árvore*.

1.4 GRAFOS CORDAIS, GRAFOS DE INTERVALO E COGRAFOS

Nessa seção, serão apresentadas as classes de grafos utilizadas no restante do texto.

O conjunto de grafos que possuem determinada propriedade formam uma *classe* ou *família* de grafos. Uma forma de descrever algoritmos de tempo polinomial para problemas $\mathcal{NP}$-difíceis em grafos é utilizando propriedades que generalizem determinadas propriedades de árvores (BRANDSTADT; SPINRAD et al., 1999).

Como exemplo, a classe dos grafos cordais é uma classe que pode ser caracterizada por ambas formas ditas acima. Outro exemplo é a família de grafos que não contêm um P_4 como subgrafo induzido, a classe dos cografos. Estas duas classes, especialmente, serão discutidas mais adiante no texto. Além disso, deve-se destacar que, tanto o CM quanto o problema da coloração de vértices podem ser respondidos por algoritmos de tempo polinomial em outras famílias como a dos grafos bipartidos, planares, grafos de intervalo, grafos de comparabilidade, grafos perfeitos, entre outras. (BRANDSTADT; SPINRAD et al., 1999)

De fato, se para duas classes de grafos A e B tal que $A \subset B$, se um problema L pode ser resolvido em tempo polinomial na classe B, então L pode ser resolvido em tempo polinomial na classe A. O mesmo vale para o caso em que L é $\mathcal{NP}$-difícil em A, o que quer dizer que L é $\mathcal{NP}$-difícil em B. Uma classe exemplar é a classe dos grafos perfeitos que inclui os grafos cordais, cografos e grafos de intervalo. Nessa classe, o problema da clique máxima, conjunto independente máximo e coloração de vértices podem ser resolvidos em tempo polinomial (GRÖTSCHEL; LOVÁSZ; SCHRIJVER, 2012).

Apesar de existirem muitas outras classes discutidas em Brandstadt, Spinrad et al. (1999), essa seção se atém à definição das classe dos grafos cordais e classe dos cografos. Tais classes serão uteis, especialmente, na discussão do algoritmo do algoritmo LexBFS, o qual viabiliza tanto o reconhecimento de classes de maneira simplificada em tempo polinomial, quanto serve como um passo na resolução dos problemas de interesse nesse trabalho.

A classe dos grafos cordais é definida na Definição 1.

Definição 1. *Um grafo é cordal se todo ciclo induzido é um triangulo em G.*

Um grafo completo e as árvores são exemplos simples de grafos cordais.

Dado um grafo cordal G, um fato relevante é que um subgrafo induzido H de G também é um grafo cordal. Assim, o subgrafo obtido de um grafo cordal ao remover um de seus vértices continua sendo cordal.

O resultado do Teorema 1 é celebre na Teoria dos Grafos, pois abre a possibilidade de reconhecer a classe dos grafos cordais em tempo polinomial.

Teorema 1 (Rose, Tarjan e Lueker (1976))**.** *Um grafo G é cordal se e somente se existe uma ordem perfeita de eliminação dos seus vértices.*

No trabalho de Rose, Tarjan e Lueker (1976) é demonstrado que uma ordem perfeita de eliminação de um grafo cordal pode ser encontrada a partir de um algoritmo LexBFS. Mais especificamente, uma ordem perfeita de eliminação pode ser obtida revertendo a ordem produzida por um algoritmo de LexBFS. Computar uma ordenação com LexBFS e testar se essa ordenação é uma ordem perfeita de eliminação pode ser feito em tempo linear, portanto reconhecer a classe dos cordais pode ser feito tempo linear (HABIB; MCCONNELL et al., 2000).

Para demonstrar que todo grafo cordal tem uma ordem perfeita de eliminação é suficiente demonstrar que em qualquer grafo cordal exista pelo menos um vértice simplicial. O resultado apresentado pelo Teorema 2 garante que todo grafo cordal tem algum vértice simplicial.

Teorema 2 (Golumbic (2004)). *Se um grafo G é cordal, então ou G é completo ou G tem ao menos dois vértices simpliciais não vizinhos.*

Uma outra caracterização que será útil da classe dos grafos cordais está descrita no Teorema 3.

Teorema 3 (Gavril (1974)). *Um grafo G é cordal se e somente se G é o grafo de intersecção de subárvores de uma árvore.*

É possível resolver o CM na classe de grafos cordais em tempo polinomial (BERRY; POGORELCNIK, 2011). Novamente, da Definição 1, os grafos cordais são os grafos onde todo ciclo induzido é um triângulo. Do Teorema 1 os cordais são os grafos que possuem uma ordem perfeita de eliminação. Dada uma ordem perfeita de eliminação $\pi = (x_1, x_2, \ldots, x_n)$, é possível enumerar todas cliques maximais de um grafo em tempo polinomial da seguinte forma: considerando um grafo cordal G, após encontrar uma ordem perfeita de eliminação π, existe uma clique formada por cada vértice x em π junto com os vizinhos que tem índice maior do que de x na ordenação π. Assim, basta procurar por uma clique que seja a maior possível em G na ordem em que aparecem na ordem perfeita de eliminação.

A seguir, a Definição 2 caracteriza os grafos de intervalo. A classe dos grafos de intervalo é utilizada para apresentar uma variação do algoritmo LexBFS no Capítulo 2.

Definição 2. *Um grafo de intervalo G é um grafo que pode ser modelado atribuindo, para cada vértice, um intervalo correspondente na reta. Dois vértices são adjacentes no grafo se e somente se seus intervalos na reta se intersectam.*

Os grafos de intervalo são subconjunto próprio dos cordais e portanto são grafos perfeitos (LEKKEIKERKER; BOLAND, 1962). Se todos intervalos na reta tem o mesmo tamanho, então o grafo G é chamado de *grafo de intervalo unitário*, também conhecido como *grafo de intervalo próprio*, onde nenhum intervalo pode propriamente conter outro intervalo.

O reconhecimento dos grafos de intervalo pode ser feito em tempo linear utilizando LexBFS. Os trabalhos de Habib, McConnell et al. (2000) e Corneil (2004b) apresentam algoritmos para o reconhecimento de grafos de intervalos que utilizam mais de um procedimento LexBFS. De maneira similar, nos trabalhos de Corneil (2004b) e Corneil (2004a) é apresentado um algoritmo que reconhece os grafos de intervalo unitários.

Por fim, a classe dos cografos é a classe formada a partir do K_1 fechada em relação às operações de união disjunta e junção nos vértices do grafo. Cografos foram descobertos

por diferentes pesquisadores desde a década de 1970. Nomes como $D^* - graphs$ e grafos HD (*Hereditary Dacey*) já foram utilizados para se referir aos cografos. Na Definição 3 é possível utilizar a operação de junção ao invés da operação de complemento.

Definição 3. *Considerando dois grafos G e H, então*

1. *K_1 é um cografo.*
2. *Se G é um cografo, então seu complemento também é um cografo.*
3. *Se G e H são cografos, então a união disjunta $G \cup H$ é um cografo.*

Existem diferentes caracterizações dos cografos, dentre elas a caracterização que estabelece que um cografo não tem um P_4 induzido.

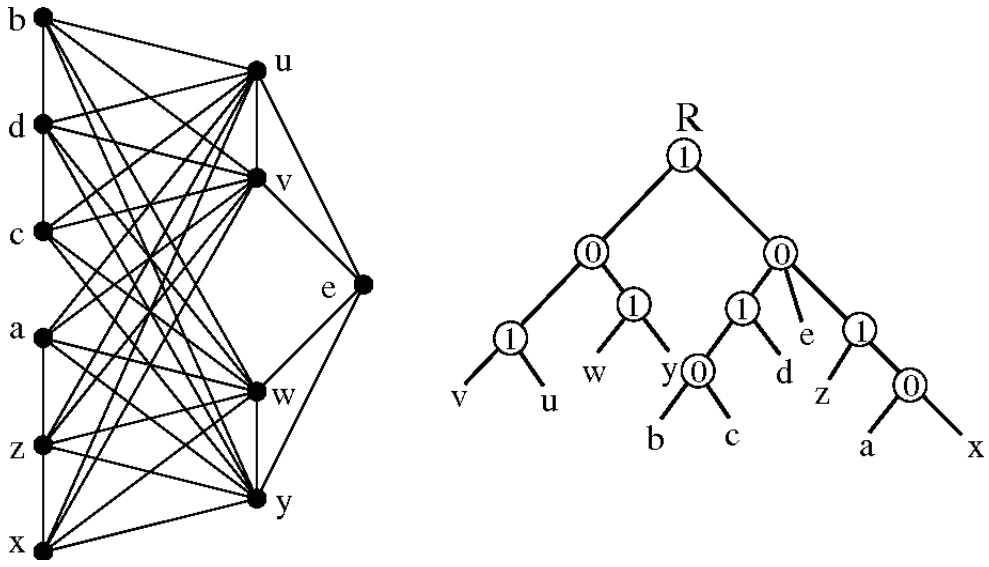
Teorema 4 (Corneil, Lerchs e Burlingham (1981))**.** *Um grafo $G = (V, E)$ é um cografo se e somente se não contém um P_4 como um subgrafo induzido.*

Da Definição 3 é possível notar que todo cografo pode ser formado a partir de um único vértice aplicando uma série de operações de junção e união disjunta. Essa sequência de operações define uma árvore, conhecida como *coárvore*, que representa o cografo.

Antes de prosseguir, será definida uma nomenclatura para coárvores diferente das árvores já definidas anteriormente. Convencionamos que os vértices de uma coárvore serão chamados de *nós*. Além disso, um nó que não é folha é um *nó interno*. Portanto, a coárvore é uma árvore enraizada cujo as folhas representam o conjunto de vértices do grafo e cada nó interno é marcado de acordo com uma das operações de união (representado por 0) ou junção (representado por 1) nos seus filhos. Na coárvore, os nós internos são marcados de maneira tal que os nós do tipo 0 e do tipo 1 estão alternados em todo caminho da raiz até qualquer folha. Os nós do tipo 0 e do tipo 1 também são chamados, respectivamente, de nós do tipo *paralelo* e nós do tipo *séries*. Dois vértices x e y do cografo são vizinhos se e somente o menor ancestral comum de x e y na coárvore é um nó do tipo junção. Os grafos de Moon-Moser (MOON; MOSER, 1965) são exemplos de cografos. A Figura 1 é um exemplo de um cografo e sua coárvore.

Para resolver o CM nos cografos, basta tomar um cografo G e sua coárvore. Inicialmente, atribuir pesos inicializados com $a = 1$ para suas folhas e restante dos nós, pesos $a = 0$. A ideia é semelhante a realizar uma travesseia em pós-ordem na coárvore. O algoritmo usa os nós internos da árvore para calcular um valor associado ao subgrafo enraizado em cada nó interno. Seja k o número de filhos de um nó interno. A partir das folhas da coárvore em direção a sua raiz, a cada nó do tipo junção encontrado atualizar o valor de a com $\sum_{i=1}^{k} a_i$ onde $a_1 \ldots, a_k$ são valores associados aos filhos. Do mesmo modo, partindo das folhas em direção à raiz, para cada nó do tipo junção atualizar o valor de a com $\max\limits_{1 \leq i \leq k} a_i$. Ao final, o valor de a na raiz será $\omega(G) = \chi(G)$, já que um cografo é um grafo perfeito.

FIGURA 1 – Cografo e sua coárvore onde R é a raiz, as folhas correspondem a um vértice no grafo e seus nós internos representam as operação de união (0) e junção (1).



FONTE: Adaptado de Bretscher et al. (2003)

2 BUSCA EM LARGURA LEXICOGRÁFICA

Nesse capítulo é apresentado o Algoritmo LexBFS e duas variações.

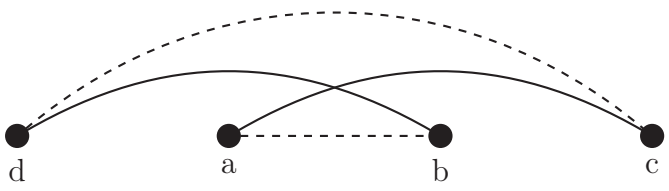
Dado um grafo $G = (V, E)$ e um vértice inicial x, uma Busca em Largura (BFS, do inglês *Breadth-First Search*) é o algoritmo que descobre todos vértices à distância k antes de descobrir os vértices à distância $k+1$ em relação a x. A sequência $(x_1, x_2, \ldots, x_n)$ em que os vértices são visitados pelo Algoritmo BFS estabelece que um vértice x_i está à *esquerda* de um vértice x_j se $i < j$ para $1 \leq i \leq n$ e $1 \leq j \leq n$. Diante disso, uma outra forma de proceder é visitar um vértice x ainda não visitado e que tem um vizinho já visitado o mais esquerda possível durante a busca. Porém, existem casos de vértices *empatados*, isto é, um conjunto de vértices com um mesmo vizinho à esquerda na sequência de vértices visitados.

O algoritmo proposto por Rose, Tarjan e Lueker (1976) para o reconhecimento da classe de grafos cordais, conhecido como Busca em Largura Lexicográfica (LexBFS, do inglês *Lexicographic Breadth-First Search*), pode ser visto como uma especialização do Algoritmo BFS. A diferença é que o Algoritmo LexBFS fornece uma regra para a escolha do próximo vértice a ser visitado, desempatando vértices com um mesmo vizinho à esquerda. Uma das formas de descrever esse algoritmo é fazendo uso da técnica de *rotulação* ou da técnica de *particionamento* de vértices. A saída desse procedimento é uma ordenação do conjunto de vértices de um grafo dado. Essa ordenação é frequentemente chamada de *ordenação LexBFS*

O resultado do Teorema 5 caracteriza uma ordenação π como uma ordenação LexBFS. A Figura 2 ilustra a propriedade da ordenação.

Teorema 5 (Corneil (2004b))**.** *Uma ordenação $\prec$ dos vértices de um grafo $G = (V, E)$ é uma ordenação LexBFS se e somente se para todo a, b, c tais que $ac \in E$ e $ab \notin E$, se $a \prec b \prec c$ então existe um vértice d, adjacente ao vértice b mas não adjacente ao vértice c, tal que $d \prec a$.*

FIGURA 2 – Caracterização de uma ordenação LexBFS



Em uma ordenação π dos vértices de um grafo, se o vértice $x <_\pi y$, dizemos que o vértice x aparece à *esquerda* do vértice y em π. Intuitivamente o vértice x *está à esquerda*

de y. Procedemos de maneira análoga no caso em que $x >_\pi y$, dizendo que x *está à direita* de y em π.

Como mencionado anteriormente, é possível implementar uma Busca em Largura Lexicográfica adotando um sistema de *rótulos* para desempatar entre vértices utilizando uma ordem lexicográfica sobre rótulos já adicionados. O termo *lexicográfica*, presente no nome da busca, vem do uso dessa estrategia. O Algoritmo LexBFS-R(G, s) faz uso do paradigma de rotulação. Na sua entrada são esperados um grafo G e um vértice s, por onde a busca se inicia. A saída desse algoritmo pode ser vista como uma ordenação que satisfaz as propriedades descritas no Teorema 5.

LexBFS-R(G, s)

Entrada : Um grafo $G = (V, E)$
Um vértice inicial s
Saída : Uma ordenação π dos vértices de G

$\pi \leftarrow$ um vetor de tamanho $|V|$ indexado por vértices
rótulo $\leftarrow$ um vetor de tamanho $|V|$, indexado por vértices, contendo uma listas para cada vértice
Para cada $y \in V$
  rótulo$[y] \leftarrow$ uma lista vazia
Acrescente o número $|V|$ ao final da lista em rótulo[r]
Para cada $i \leftarrow |V|$ decrescendo até 1
  Tome um vértice y não definido em π com maior rótulo em ordem lexicográfica
  $\pi[y] \leftarrow |V| + 1 - i$
  Para cada vértice $z \in N(y)$ e $\pi[z] = \emptyset$
    Acrescente i ao final da lista em rótulo[z]
Devolva π

Em seguida será descrita um outro algoritmo para Busca em Largura Lexicográfica. O Algoritmo LexBFS(G) é algoritmo que faz uso da técnica de particionamento. O resultado desse procedimento é uma ordenação $\pi = (x_1, x_2, \ldots . x_n)$ do conjunto V que satisfaz as condições do Teorema 5. Será utilizado $\pi(i) = x$ para indicar que um vértice x está na i-ésima posição em π. Assim, a ordenação π é a ordem em que os vértices de G são visitados pelo algoritmo. Inicialmente, nenhum vértice tem uma posição na ordenação.

Para continuação da descrição de LexBFS, considere uma lista duplamente ligada contendo ponteiros para o *primeiro* elemento da lista, também, um ponteiro para o *antecessor* e para o *sucessor* de um elemento na lista. Assuma uma partição ordenada P como uma lista duplamente ligada. Assumindo também que as partes $X \in P$ são listas duplamente ligadas, o *pivô* será o primeiro vértice de uma parte X o qual é removido de P e recebe posição na ordenação π. Utilizando os vizinhos do vértice pivô, o algoritmo LexBFS(G) procede particionando o conjunto de vértices. A Figura 3 mostra os passos do algoritmo.

LexBFS(G)

Entrada : Um grafo $G = (V, E)$
Saída : Uma ordenação π de V

```
1  Tome P como uma lista duplamente ligada inicializada com uma partição
     ordenada vazia
2  π ← um vetor de tamanho |V| inicialmente vazio
3  P ← [V], uma parte unitária contendo todos vértices de V
4  i ← 1
5  Enquanto P ≠ ∅
6  |  Tome X_a como a primeira parte em P
7  |  Remova o primeiro vértice x de X_a ▷ pivô
8  |  Se X_a = ∅
9  |  |  remova X_a de P
10 |  π[i] ← x
11 |  i ← i + 1
12 |  Para cada parte X_b ≥ X_a em P
13 |  |  Y ← X_b ∩ N(x)
14 |  |  Se Y ≠ ∅ e Y ≠ X_b
15 |  |  |  X_b ← X_b \ Y
16 |  |  |  Insira Y como antecessor de X_b em P
17 Devolva π
```

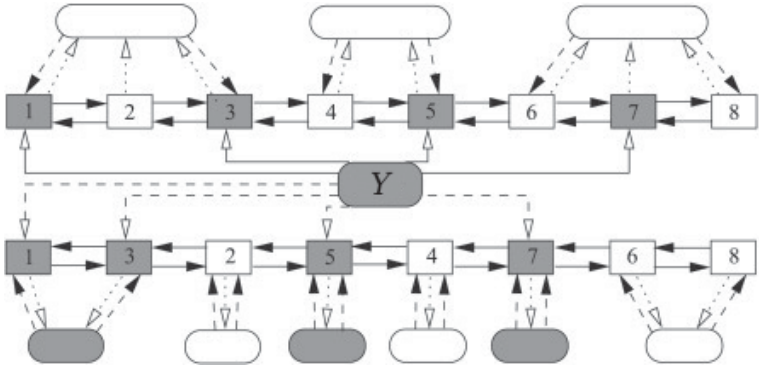
Para alcançar complexidade de tempo linear, porém, o Algoritmo LexBFS faz uso de uma estrutura de dados que utiliza ponteiros. A Figura 4 mostra um diagrama das estruturas de dados assumidas. A parte superior representa um momento na execução antes do laço da Linha 12. A parte inferior representa o momento seguinte ao término do laço. Todos elementos de V são guardados numa lista duplamente ligada P. Cada elemento (quadrados) de V tem um ponteiro para seu antecessor e seu sucessor (linhas cheias com pontas preenchidas). Cada elemento mantêm um ponteiro (linhas pontilhadas com pontas não preenchidas) em direção a parte que pertence. Cada parte (retângulos com pontas arredondadas, superior) mantém dois ponteiros (linhas tracejadas com pontas preenchidas) em direção à lista duplamente ligada: um para o seu primeiro elemento e outro para o seu último elemento. Assim, os elementos de uma parte X permanecem consecutivos na lista, formando um intervalo. O conjunto Y de vizinhos do pivô x, na Linha 13 do Algoritmo LexBFS(G), é chamado *conjunto pivô*. O conjunto pivô é constituído por ponteiros em direção aos seus elementos (em cinza) na lista duplamente ligada. As operações que se iniciam no laço da Linha 12, formam uma nova parte (inferior, em cinza) movendo os elementos na lista P.

FIGURA 3 – Primeiros passos do algoritmo LexBFS(G) no grafo G da Figura 1. Inicialmente a lista de partes contém apenas uma parte, o conjunto de vértices. O resultado da ordenação é $\pi = (x, y, w, z, u, v, a, d, c, b, e)$.

i	pivô	P
		[x d y u e v w c a z b]
1	x	[y u v w z] [d e c a b]
2	y	[w z] [u v] [d e c a b]
3	w	[z] [u v] [d e c a b]
4	z	[u v] [a] [d e c b]
5	u	[v] [a] [d e c b]
6	v	[a] [d e c b]
7	a	[d e c b]
8	d	[c b] [e]
9	c	[b] [e]
10	b	[e]
11	e	

FONTE: Adaptado de Bretscher et al. (2003)

FIGURA 4 – Estrutura de dados aproveitada pelo Algoritmo LexBFS(G) utilizando a técnica de particionamento. Partes são retângulos com pontas arredondados. Quadrados são elementos da lista duplamente ligada que representam vértices do grafo.



FONTE: Adaptado de Habib e Paul (2010)

Para remover os elementos do conjunto pivô Y que estão em X_b e inseri-los em uma nova parte antecessora de X_b, basta tomar cada $y \in Y$ e trocar o elemento para o qual y aponta com o primeiro elemento da parte X_b. Em outras palavras, um a um os elementos de Y são levados para o início de uma parte e formam uma nova parte. Note que y aponta para um elemento em P que guarda a parte à qual y pertence. É necessário atualizar o elemento para o qual y aponta com a nova parte e atualizar o ponteiro para o primeiro elemento da parte X_b. Essa operação toma tempo $O(|Y|)$ no número de trocas. Assim, como cada vértice é escolhido como pivô uma única vez o Algoritmo LexBFS(G) tem complexidade de tempo $O(|V| + |E|)$. O Apêndice A traz uma implementação do Algoritmo LexBFS(G) em linguagem de programação C++.

Considerando o Algoritmo LexBFS(G), em sua Linha 7, note que pode existir um conjunto de vértices, ao invés de um único vértice, candidatos a pivô. Esse conjunto de vértices é chamado de *fatia*. A Definição 1 é utilizada por algoritmos de reconhecimento de diferentes classes de grafos.

Observação 1. *Os vértices na Linha 7 do Algoritmo LexBFS(G) estão empatados, i.e, compartilham os mesmos vizinhos já definidos em π. Esse conjunto de vértices forma uma fatia, denotada por S.*

Um exemplo de fatia é a parte [y u v w z] na Figura 3, formada pelos vértices vizinhos do vértice x. Observe que o conjunto de todos os vértices também é uma fatia. Utilizando colchetes para denotar uma fatia ao executar uma LexBFS(G) iniciada pelo vértice x no grafo da Figura 1, temos um resultado das fatias aninhadas como

$$\pi = [\mathbf{x}[\mathbf{y}[\mathbf{w}[\mathbf{z}]][\mathbf{u}[\mathbf{v}]]][\mathbf{a}][\mathbf{d}[\mathbf{c}[\mathbf{b}]][\mathbf{e}]]]$$

As seções seguintes discutem aplicações do Algoritmo LexBFS(G) e suas variações. A Seção 2.1 discute uma aplicação através do reconhecimento da classe dos grafos cordais. As seções 2.2 e 2.3 discutem duas variações conhecidas como LexBFS$^+$ e LexBFS$^-$.

2.1 RECONHECIMENTO DE GRAFOS CORDAIS

O resultado do Teorema 6 estabelece a base para o algoritmo de reconhecimento de grafos cordais.

Teorema 6 (Rose, Tarjan e Lueker (1976) e Habib, McConnell et al. (2000))**.** *Um dado grafo G é cordal se e somente se o reverso de uma ordenação computada pelo Algoritmo LexBFS(G) fornece uma ordem perfeita de eliminação.*

O Algoritmo OPE(G, π) verifica se uma ordenação π obtida do Algoritmo LexBFS(G) é uma ordem perfeita de eliminação. Para verificar se um dado gafo G tem uma ordem perfeita de eliminação, o Algoritmo OPE(G, π) recebe uma ordenação π como entrada. Além disso, o OPE(G, π) faz uso de uma lista $D(x)$ definida como sendo os vizinhos de um vértice x que são sucessores dele na ordem inversa produzida por LexBFS(G). De resto, a função $pai(x)$ é a função que devolve o vértice mais à esquerda da lista devolvida por $D(x)$, de acordo com a ordenação π.

Novamente, fornecendo como entrada ao algoritmo OPE(G, π) um grafo G e o reverso de uma ordenação obtida através do procedimento LexBFS(G), então temos que $\{x\} \cup D(x)$ é uma clique, onde o resultado devolvido por $\pi(x)$ é o menor dentre todos vértices dessa clique. O teste na Linha 7 falha ao tomar um vértice x no qual $\{x\} \cup D(x)$ **não** é uma clique, quer dizer, pela escolha de x, existe algum vizinho de $pai(x)$ que não é adjacente a x em G e está mais à direita em π.

OPE(G, π)

Entrada : Um grafo $G = (V, E)$
Uma ordenação π obtida ao reverter o resultado de LexBFS(G)
Saída : *Sim* se π dos vértices do grafo G; *Não* caso contrário

```
1 D ← um vetor de tamanho |V| indexado por vértices
2 pai ← um vetor de tamanho |V| indexado por vértices
3 Para cada vértice x ∈ V
4 |   D[x] ← {y | y ∈ N(x) e π[y] > π[x]}
5 |   pai[x] ← y onde min{π[y] | y ∈ D[x]}
6 Para cada vértice x ∈ V
7 |   Se D[x] \ pai[x] ⊈ D[pai[x]]
8 |   |   Devolva Não
9 Devolva Sim
```

Em termos de complexidade de tempo, construir D para todo vértice x em um grafo leva tempo $O(|V| + |E|)$. Para obter o resultado da linha 7 em tempo linear pode-se manter as listas de D na ordem que aparecem em π de maneira que a verificação seja reduzida a encontrar um subconjunto comum entre duas listas ordenadas. Assim, a complexidade de tempo do algoritmo OPE(G, π) está na ordem de $O(|V| + |E|)$.

Com base no Teorema 6 é descrito o Algoritmo Cordal(G) que reconhece se um dado grafo é um cordal.

Cordal(G)

Entrada : Um grafo $G = (V, E)$
Saída : *Sim* se G é Cordal; *Não* caso contrário

```
1 π ← LexBFS(G)
2 Inverta a ordem dos elementos de π
3 Se π é uma ordem perfeita de eliminação
4 |   Devolva Sim
5 Senão
6 |   Devolva Não
```

2.2 RECONHECIMENTO DE GRAFOS DE INTERVALO UNITÁRIO

Para reconhecer um grafo de intervalo unitário será apresentado o Algoritmo LexBFS$^+$.

Um grafo de intervalo unitário é um grafo em que seus vértices podem ser colocados em correspondência de um para um com intervalos unitários na reta real. Dois vértices no grafo são adjacentes se e somente se seus correspondentes intervalos tem intersecção. A caracterização da classe dos grafos de intervalo unitário utilizada nesse texto está presente no Teorema 7. O reconhecimento dessa classe de grafos é baseado na propriedade 3 do teorema. A seguinte definição é empregada: dada um ordenação π, os vértices na vizinhança

fechada $N[x]$ de um dado vértice x são *consecutivos* se existe $1 \leq i \leq |V|$ e $1 \leq j \leq |V|$ tal que $N[x] = \{y \mid y = \pi(k), i \leq k \leq j\}$.

Teorema 7 (Corneil (2004a)). *Seja $G = (V, E)$ um grafo. As seguintes propriedades são equivalentes:*

1. *G é um grafo de intervalo unitário*
2. *G é um grafo de intervalo que não contém um $K_{1,3}$ induzido*
3. *Existe uma ordenação π do conjunto de vértices V tal que para todo $x \in V$, os vértices de $N[x]$ são consecutivos em π.*

O Algoritmo LexBFS$^+(G, \pi)$ é a uma variação do Algoritmo LexBFS(G) apresentado anteriormente, que desempata vértices (Observação 1) na escolha do pivô na Linha 7 de LexBFS(G). Para isso, ele utiliza uma ordenação computada anteriormente dada como entrada. Esse algoritmo recebe um grafo $G = (V, E)$ e uma ordenação π do conjunto de vértices V. O resultado de LexBFS$^+(G, \pi)$ é uma ordenação dos vértices de G, denotada por π^+.

Uma descrição de LexBFS$^+(G, \pi)$ pode ser feita como segue. Execute LexBFS(G), no passo da Linha 6, tome uma fatia S como a primeira parte de P. Agora, escolha x como sendo o vértice pivô de S com maior posição em π. A saída desse procedimento é uma ordenação π^+ dos vértices de G.

LexBFS$^+(G, \pi)$

Entrada : Um grafo $G = (V, E)$ e uma ordenação π de V
Saída : Uma ordenação π^+ de V
1 Execute uma LexBFS(G)
2 $S \leftarrow X_a$, a primeira fatia em P
3 Tome x como o vértice pivô de S com maior posição em π ▷ `modificação na Linha 7 de LexBFS`
4 **Devolva** π^+

Contudo, se a ordenação π, dada como entrada, for o resultado invertido de LexBFS, então o primeiro vértice da fatia S equivale ao vértice pivô x à ser escolhido pelo Algoritmo LexBFS$^+$. Portanto, esta é uma outra forma de conceber LexBFS$^+$: executar uma LexBFS, mas em sua Linha 3, iniciar P com o inverso de uma ordenação previamente computada pelo próprio LexBFS. A Figura 5 demonstra alguns passos desses algoritmo.

Com uma descrição do Algoritmo LexBFS$^+(G, \pi)$ é possível prosseguir para um algoritmo de reconhecimento da classe dos grafos de intervalo unitário. O algoritmo proposto em Corneil (2004a) utiliza a propriedade 3 do Teorema 7, chamada *condição da vizinhança*, para reconhecer um grafo de intervalo unitário. O Algoritmo UIG(G) é o algoritmo que utiliza tanto uma LexBFS quanto uma LexBFS$^+$ para isso.

Alguns dos passos da execução de UIG podem ser apresentados tomando o grafo conhecido como *Bull*. A seguir as ordenações π, π^+ e π^{++} obtidas dentro Algoritmo UIG(G)

FIGURA 5 – Alguns passos de $\mathsf{LexBFS}^+(G, \pi)$ no grafo da Figura 1 e a ordenação $\pi = (x, y, w, z, u, v, a, d, c, b, e)$ como entrada, previamente computada por $\mathsf{LexBFS}(G)$. O primeiro passo apenas inverte a ordenação de π. Em seguida, o Algoritmo LexBFS é chamado com os vértices inicialmente ordenados como π, dada como entrada.

i	pivô	P
		[e b c d a v u z w y x]
1	e	[v u w y] [b c d a z x]
2	v	[u] [w y] [b c d a x]
3	u	[w y] [b c d a x]

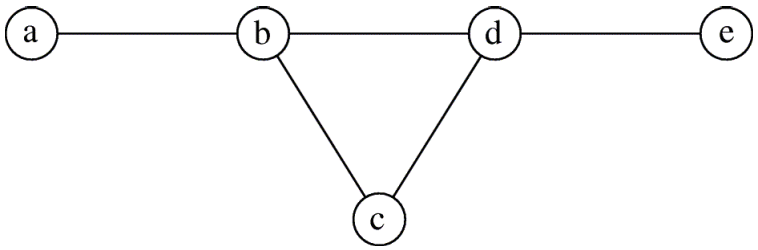
FONTE: Adaptado de Bretscher et al. (2003)

UIG(G)

Entrada : Um grafo $G = (V, E)$
Saída : *Sim* se G é um *uig*; *Não* caso contrário

```
1 π ← LexBFS(G)
2 π⁺ ← LexBFS⁺(G, π)
3 π⁺⁺ ← LexBFS⁺(G, π⁺)
4 Se π⁺⁺ satisfaz a condição da vizinhança
5 |  Devolva Sim
6 Senão
7 |  Devolva Não
```

FIGURA 6 – O grafo Bull



tomando o grafo *Bull* da Figura 6 como entrada. Entre colchetes estão os vértices em uma fatia.

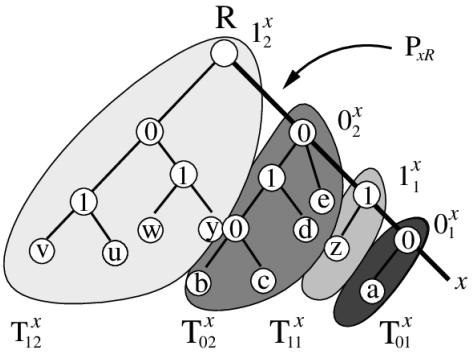
π = c [b d] a e,

π^+ = e d [b c] a,

π^{++} = a b [c d] e.

Na execução de $\mathsf{LexBFS}^+(G, \pi)$ dentro de UIG, utilizando π previamente computada como entrada, LexBFS^+ escolhe o vértice e como o primeiro elemento da ordenação π^+, pois esse é o vértice com maior posição em π. Na sequência da execução do LexBFS^+, quando a fatia $S = \{b, c\}$ é encontrada, o vértice b é escolhido antes que o vértice c — já que b também tem maior posição em π. Nesse exemplo, não é difícil notar que a ordenação π^{++} satisfaz a *condição da vizinhança* e portanto o grafo *Bull* é um grafo de intervalo unitário.

FIGURA 7 – A coárvore do cografo G da Figura 1. As áreas escuras representam subárvores T^x_{0i} e T^x_{1i}.



FONTE: Adaptado de Bretscher et al. (2003)

2.3 RECONHECIMENTO DE COGRAFOS

Através de um algoritmo para o reconhecimento de cografos será apresentado o Algoritmo LexBFS$^-$.

2.3.1 Estrutura dos Cografos

As definições seguintes serão úteis. Tome T como uma coárvore do cografo $G = (V, E)$ cuja a raiz é R. Seja x uma folha da coárvore T. A notação P_{xR} denota o único caminho em T de x até R. Note que os nós internos no caminho P_{xR} alternam entre nós marcados com 0 ou 1. Seja $(0^x_1, 0^x_2, \ldots, 0^x_k)$ a sequência de nós 0 no caminho P_{xR}. De maneira similar $(1^x_1, 1^x_2, \ldots, 1^x_k)$ denota os nós internos marcados com 1 no caminho P_{xR}. Considere uma subárvore enraizada em 0^x_i, o i-ésimo nó marcado com 0 no caminho P_{xR}. Denotamos por T^x_{0i} a subárvore enraíza por 0^x_i menos a subárvore enraizada pelo filho que está sobre o caminho P_{xR}. De maneira análoga denotamos 1^x_i e T^x_{1i}. A Figura 7 ilustra essas definições.

Daqui em diante, serão apresentadas novas definições e notação. Novamente, utilizando colchetes para representar uma fatia, ao executar LexBFS(G) iniciando pelo vértice x considerando o cografo da Figura 1, temos que

$$\pi = \mathbf{x}[\mathbf{y}[\mathbf{w}[\mathbf{z}]][\mathbf{u}[\mathbf{v}]]][\mathbf{a}][\mathbf{d}[\mathbf{c}[\mathbf{b}]][\mathbf{e}]]$$

Em π, omitimos os colchetes do primeiro nível, a fatia que contém todos os vértices. É possível notar que as fatias estão aninhadas em π. Tais fatias aninhadas são chamadas de *subfatias*. A seguinte notação considera o primeiro nível de subfatias.

Considere somente o primeiro nível de subfatias aninhadas em S. Seja $S^A(x) :=$ [a subfatia de S com vértices adjacentes a x]. Cada subfatia na sequência contém os vértices não adjacentes a x. Assim, $\langle S^N(x)\rangle := S^N_1(x), S^N_2(x), \ldots, S^N_k(x)$ denota as subfatias formada por vértices não adjacentes a x em S. Com isso temos que $S := [x, S^A(x), \langle S^N(x)\rangle]$,

uma fatia construída durante uma LexBFS(G) onde x é o primeiro vértice de S. Dessas definições emerge a Observação 2.

Observação 2 (Bretscher et al. (2003))**.** *Considere a execução do Algoritmo LexBFS(G). Seja S a fatia inicial com todos os vértices. Após a execução, S é particionado em $[x, S^A(x), \langle S^N(x)\rangle]$ e cada fatia S_i^N contém exatamente as folhas da subárvore T_{0i}^x de T.*

Para exemplificar a Observação 2, se S é igual a π, então $S^A(x) = [ywzuv]$, $S_1^N(x) = [a]$ e $S_2^N(x) = [dcbe]$. Os vértices na fatia $S_1^N(x)$ correspondem às folhas da subárvore T_{01}^x e, os vértices em $S_2^N(x)$ correspondem às folhas da subárvore T_{02}^x (veja a Figura 7). Note que isso está em conformidade com a Observação 2. Logo, temos que

$$\pi = \mathbf{x}[\mathbf{ywzuv}][\mathbf{a}][\mathbf{dcbe}]$$

Para as subárvores T_{1i}^x existe uma relação semelhante. Uma coárvore $T(\overline{G})$ do grafo $\overline{G}$ é uma coárvore T com as marcações trocados: nós antes marcados com 0 agora marcados com 1 e nós marcados com 1 agora marcados com 0. Portanto, uma LexBFS no complemento de G, iniciando no vértice x, deve particionar as folhas das subárvore T_{1i}^x dentro de fatias. Uma LexBFS($\overline{G}$) produz $\overline{\pi}$ e as fatias $\overline{S} \coloneqq [x, \overline{S^A}(x), \langle \overline{S^N}(x)\rangle]$ onde a fatia $\overline{S^A}(x) \coloneqq$ [os vértices adjacentes a x em $\overline{G}$] e $\langle \overline{S^N}(x)\rangle \coloneqq \overline{S_1^N}(x), \overline{S_2^N}(x), \ldots, \overline{S_k^N}(x)$ denota as subfatias com vértices não adjacentes a x em $\overline{G}$.

Considerando o cografo G da Figura 7, ao executar LexBFS($\overline{G}$) no complemento de G, iniciada a partir do vértice x, obtemos a seguinte ordenação

$$\overline{\pi} = \mathbf{x}[\mathbf{dabce}][\mathbf{z}][\mathbf{uwyv}]$$

É possível notar que $\overline{S_1^N}(x) = [z]$ é folha na subárvore T_{11}^x e $\overline{S_2^N}(x) = [uwyv]$ está contido na subárvore T_{12}^x na Figura 7. Na verdade, essa é uma propriedade que se estende para qualquer cografo como descrito em Bretscher et al. (2003).

A fim de apresentar a *propriedade de subconjunto de vizinhança* estabelecemos a Definição 4 em seguida. Considerando a execução de LexBFS, um vértice é *numerado* se já tem uma posição i na ordenação π que está sendo construída.

Definição 4. *Dada uma fatia $S_i^N(x)$, definimos a vizinhança numerada de $S_i^N(x)$ como sendo*

$$N_i(x) \coloneqq \{y \mid y \text{ numerado em } \pi \text{ e } y \in N(z), \forall z \in S_i^N(x)\}$$

Em sequência, a Definição 5 diz respeito a Propriedade de Subconjunto de Vizinhança.

Definição 5. *Uma LexBFS satisfaz a propriedade de subconjunto de vizinhança se somente se*

$$N_{i+1}(x) \subset N_i(x), \forall x \in V, \forall i \geq 1$$

2.3.2 Algoritmo de Reconhecimento

O Algoritmo LexBFS$^-$ é uma variação do Algoritmo LexBFS(G) que acrescenta uma regra para escolha do pivô. Para conceber esse algoritmo, a ideia é partir de uma ordenação inicial dos vértices de G e utilizar essa primeira ordenação para desempatar vértices no passo 7 do Algoritmo LexBFS(G).

Uma descrição de LexBFS(G, π) pode ser feita como segue. Faça uma LexBFS$(\overline{G})$, no passo da Linha 6, tome uma fatia S como a primeira parte de P. Agora, escolha x como sendo o vértice pivô de S com o menor posição em π. A saída desse procedimento é uma ordenação dos vértices de $\overline{G}$, denotada $\overline{\pi}^-$.

LexBFS$^-(G, \pi)$

Entrada : Um grafo $G = (V, E)$ e uma ordenação π de V
Saída : Uma ordenação $\overline{\pi}^-$ de V
1 Execute uma LexBFS$(\overline{G})$
2 $S \leftarrow X_a$, a primeira fatia em P
3 Tome x como o vértice pivô de S com menor posição em π ▷ `modificação na Linha 7 de LexBFS`
4 **Devolva** $\overline{\pi}^-$

Para conceber essa variação, contudo, não é preciso computar o grafo complementar $\overline{G}$ explicitamente. Primeiro, note que o mesmo resultado obtido pelo Algoritmo LexBFS(G) é alcançado quando ao invés de formar o conjunto X_b removendo vértices vizinhos ao pivô x, forem removidos os vértices que **não** são vizinhos a x. Porém, é preciso alterar a ordem em que o conjunto Y é inserido, isto é, inseri-lo como sucessor de X_b na lista P. Assim, a decisão de onde posicionar os vizinhos do vértice pivô dá uma alternativa que leva à Obervação 3.

Observação 3. *Inserir os vizinhos de x como sucessor, ao invés de inseri-los como antecessor de X_b, permite computar π no* ***complemento*** *do grafo G.*

Da Observação 3 é possível computar uma LexBFS$^-(G)$ fazendo uma alteração na Linha 16 de LexBFS(G). Assim, para implementar uma LexBFS$^-$, exceto pela Linha 16, todo o resto permanece idêntico à LexBFS(G). A parte inicial em P serão os vértices de G na ordenados como dado em π. Isso significa que esse novo algoritmo tem uma regra de desempate que espelha a regra de desempate de uma LexBFS$^+$.

Na seção anterior foi exemplificada uma ordenação $\overline{\pi}$. A ordenação $\overline{\pi}$ é semelhante a ordenação $\overline{\pi}^-$ produzida por uma LexBFS$^-(\overline{G}, \pi)$. Por exemplo, dada a ordenação $\pi = (x, y, w, z, u, v, a, d, c, b, e)$ do cografo G da Figura 1, temos que uma LexBFS$^-(\overline{G}, \pi)$ produz

$$\overline{\pi}^- = \mathbf{x}[\mathbf{adecb}][\mathbf{z}][\mathbf{yuvw}]$$

O Teorema 8 estabelece que é possível reconhecer a classe dos cografos utilizando uma ordenação produzida por LexBFS$^-$.

Teorema 8 (Bretscher et al. (2008))**.** *Dado um grafo $G = (V, E)$, uma ordenação $\overline{\pi}^-$ produzido por LexBFS$^-(\overline{G}, \pi)$ e uma ordenação π^- resultante da execução de LexBFS$^-(G, \overline{\pi}^-)$. Um grafo G é um cografo se e somente se $\overline{\pi}^-$ e π^- satisfazem a Propriedade de Subconjunto de Vizinhança.*

O Algoritmo Cografo(G) é o algoritmo para o reconhecimento de cografos. Note que checar se uma ordenação satisfaz a essa propriedade leva tempo linear. O algoritmo Cografo(G), descrito a seguir, faz três execuções de LexBFS(G) sobre os vértices de G para reconhecer se G pertence a classe dos cografos. Se um dado grafo G é um cografo, o procedimento Coárvore(x), descrito em detalhes em Bretscher et al. (2008), constrói a coárvore de G ao percorrer recursivamente as sequências de fatias $\langle S^N(x)\rangle$ e $\langle \overline{S^N}(x)\rangle$ em complexidade de tempo $O(|V| + |E|)$. Esse procedimento é baseado no Teorema 9. Do Teorema 10 é possível construir a coárvore em complexidade de tempo $O(|V| + |E|)$. O procedimento $P_4(\overline{\pi}^-, \pi^-)$, descrito em detalhes em Bretscher et al. (2003), retorna um certificado, um caminho com 4 vértices, em complexidade de tempo $O(|N(x)|)$ no caso em que a verificação da Linha 4 falha, como estabelece o Teorema 11. Nenhum dos passos desse algoritmo leva tempo maior que linear no tamanho do grafo. Logo, a complexidade de tempo de Cografo(G) é proporcional a $O(|V| + |E|)$.

Teorema 9 (Bretscher et al. (2008))**.** *Dadas as ordenações $\overline{\pi}^-$ e π^- de um cografo G, a coárvore T pode ser construída recursivamente das fatias de $\overline{\pi}^-$ e π^-.*

Teorema 10 (Bretscher et al. (2003))**.** *Seja x o primeiro vértice das ordenações $\overline{\pi}^-$ e π^- de um cografo G. É possível construir uma coárvore de G em complexidade de tempo $O(|V| + |E|)$ partindo x.*

Teorema 11 (Bretscher et al. (2003, 2008))**.** *Dadas as fatias $S_i^N(x)$ e $S_{i+1}^N(x)$ que não satisfazem a propriedade de subconjunto de vizinhança, então um P_4 pode ser construído em complexidade de tempo $O(|N(x)|)$, percorrendo a vizinhança numerada dessas fatias.*

Cografo(G)

Entrada : Um grafo $G = (V, E)$
Saída : Uma coárvore se G é cografo
Um P_4 caso contrário

1 $\pi \leftarrow \mathsf{LexBFS}(G)$
2 $\overline{\pi}^- \leftarrow \mathsf{LexBFS}^-(\overline{G}, \pi)$
3 $\pi^- \leftarrow \mathsf{LexBFS}^-(G, \overline{\pi}^-)$
4 **Se** $\overline{\pi}^-, \pi^-$ satisfaz a *propriedade de subconjunto de vizinhança*
5 | **Devolva** Coárvore(x)
6 **Senão**
7 | **Devolva** $P_4(\overline{\pi}^-, \pi^-)$

3 CLIQUE MÁXIMA E COLORAÇÃO DE VÉRTICES

No presente capítulo, serão apresentados o CM e o problema da coloração de grafos, dois problemas $\mathcal{NP}$-difíceis relacionados entre si. Na Seção 3.1 são apresentados algoritmos exatos presentes na literatura e resultados acerca de sua complexidade de tempo. De maneira similar a Seção 3.3 discorre acerca do problema da coloração em grafos.

3.1 O PROBLEMA DA CLIQUE MÁXIMA

O problema da Clique Máxima é um problema fundamental de otimização combinatória. Sua versão de decisão se encontra na lista de 21 problemas de decisão $\mathcal{NP}$-completos em Karp (1972). Além disso, é um problema relacionado a outros problemas de otimização como o problema coloração de vértices (WU; HAO, 2012) e *clustering* de grafos (SCHAEFFER, 2007). Do ponto de vista das aplicações práticas existem diversas aplicações em variados campos do conhecimento, como por exemplo: bioinformática (MALOD-DOGNIN; ANDONOV; YANEV, 2010), economia (BOGINSKI; BUTENKO; PARDALOS, 2006), redes sociais (PATTILLO; YOUSSEF; BUTENKO, 2012), telecomunicações (BALASUNDARAM; BUTENKO, 2006), entre outros. Nos trabalhos de Pardalos e Xue (1994) e Bomze et al. (1999) o CM é coberto de maneira extensiva.

Existe, portanto, uma motivação para a procura de soluções exata para o problema. Dentre os muitos algoritmos desenvolvidos podemos citar o algoritmo proposto por Carraghan e Pardalos (1990) que aborda o problema através da estratégia BB produzindo um algoritmo de fácil adaptação. Esse algoritmo será chamado de Algoritmo CP. Por meio de sucessivas modificações nesse algoritmo, posteriormente, são propostos novos algoritmos. Existem trabalhos que propõem modificar CP aplicando uma coloração nos vértices do grafo. Os algoritmos propostos por Tomita e Seki (2003) e posteriormente Tomita e Kameda (2007) se baseiam no fato de que se um grafo pode ser colorido com k cores, então a clique máxima é menor ou igual a k.

É importante dizer que os grafos de Moon-Moser possuem $\Theta(3^{n/3})$ cliques maximais sendo estes os grafos com o maior número possível de cliques maximais para um dado tamanho do conjunto de vértices (MOON; MOSER, 1965). É possível encontrar a maior clique de um grafo listando todas cliques maximais em tempo $O(3^{n/3})$ e retornando a maior delas (TOMITA; TANAKA; TAKAHASHI, 2006). Porém, existem limitantes superiores melhores para o problema. O melhor resultado conhecido até o momento da escrita deste trabalho é o de Robson (ROBSON, 2001), equivalente a $O(2^{0,25n})$. Além disso, a menos que $\mathcal{P} = \mathcal{NP}$, para qualquer número real $\varepsilon > 0$, não pode existir uma aproximação de tempo polinomial para o CM dentro de um fator melhor que $O(n^{1-\varepsilon})$ (ZUCKERMAN, 2006).

A revisão de Wu e Hao (2015) cita, entre outros, os seguintes algoritmos num viés analítico: CP conforme descrito em Carraghan e Pardalos (1990), MCQ conforme Tomita e Seki (2003), DYN conforme Konc e Janezic (2007), MCR conforme Tomita e Kameda (2007), MCS conforme Tomita, Sutani et al. (2010). Além dos algoritmos citados à cima, são implementados um total de 13 algoritmos em Züge e Carmo (2018), discutidos do ponto de vista analítico e experimental com o objetivo de compará-los. O trabalho de Prosser (2012) demonstra, experimentalmente, como pequenas mudanças afetam o desempenho de algoritmos BB exatos para o problema. Mais recentemente, San Segundo, Coniglio et al. (2019) e San Segundo, Furini e Artieda (2019) propõem novos algoritmos BB para uma variação do CM que leva em conta pesos associados, respectivamente, aos vértices e arestas do grafo.

Uma vez que, muitos desses algoritmos são adaptações uns dos outros, existe um *framework* subjacente que pode ser especializado para o algoritmo escolhido. Os trabalhos de Carmo e Züge (2012), Wu e Hao (2015), Anjos, Züge e Carmo (2016) e Züge e Carmo (2018) exploram tal conceito. Essa ideia será discutida com mais detalhes em sequência, na Seção 3.2.

3.2 ALGORITMOS MCBB

Como já citado anteriormente na Seção 3.1 é possível implementar algoritmos BB para o CM sob um algoritmo genérico. A partir de modificações no algoritmo proposto por Bron e Kerbosch (1973), que chamamos de BK, a ideia de um *framework* aparece. Porém, BK não é um algoritmo BB. A primeira variação desse algoritmo que pode ser considerada como um algoritmo BB é o algoritmo CP descrito em Carraghan e Pardalos (1990). Em seguida, os algoritmos CHI + DF (FAHLE, 2002) e MCQ, modificam CP com base na coloração de vértices.

Na dissertação de mestrado de Züge (2012), partindo de BK, é demonstrado em mais detalhes uma forma de implementar diferentes algoritmos BB para o CM modificando um único algoritmo, chamado MCBB. Como resultado, foram criadas funções que podem ser especializadas para conceber diferentes algoritmos para o CM. Essa ideia será útil pois todos algoritmos discutidos em sequência serão descritos, implementados e comparados num contexto unificado. Além disso, é uma opção que pode facilitar a reprodução dos resultados obtidos.

Os seguintes algoritmos serão descritos sob MCBB.

- CP conforme Carraghan e Pardalos (1990)
- DF, CHIe CHI + DF conforme descritos em Fahle (2002)
- MCQ conforme Tomita e Seki (2003)
- DYN conforme Konc e Janezic (2007)

- MCR conforme Tomita e Kameda (2007)
- MCS conforme Tomita, Sutani et al. (2010)
- Basic conforme Carmo e Züge (2012)

MCBB(G)

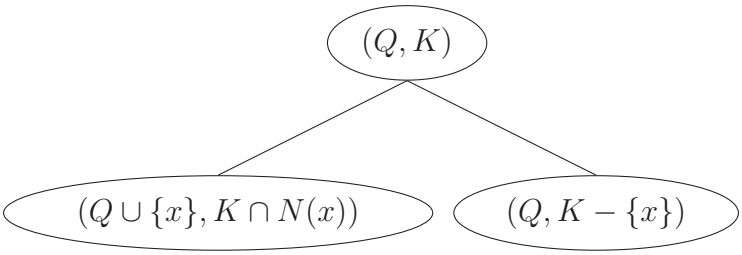
Entrada : Um grafo G
Saída : Uma clique de G de tamanho máximo

```
1  Q ← um conjunto que guarda os vértices de uma clique em G
2  C ← um conjunto que guarda os vértices da clique máxima em G
3  K ← uma lista que guarda os vértices candidatos a fazer parte de uma clique
     máxima
4  S ← uma pilha de pares (Q, K)
5  (S, C) ← pre-process(G)
6  Enquanto S ≠ ∅
7  |  Desempilhe um estado (Q, K) de S  ▷ raiz
8  |  pre-process-state(G, Q, K, C) ▷ chamada 1
9  |  Enquanto K ≠ ∅ e |C| < |Q| + bound(G, Q, K)
10 |  |  x ← remove(K)  ▷ vértice pivô
11 |  |  Empilhe (Q, K) em S
12 |  |  (Q, K) ← (Q ∪ {x}, K ∩ N(x))
13 |  |  pre-process-state(G, Q, K, C) ▷ chamada 2
14 |  |  Se |C| < |Q|
15 |  |  |  C ← Q
16 Devolva post-process(G, C)
```

No Algoritmo MCBB(G) é utilizada uma pilha $\mathcal{S}$ de pares (Q, K), chamados de *estados*, onde Q é uma *clique* em G e K é conjunto dos vértices *candidatos* a fazer parte de uma clique máxima, isto é, um subconjunto de $V(G) \setminus Q$ onde todos seus vértices são adjacentes a todo vértice em Q em G. Também é mantido o conjunto C que guarda a maior clique do grafo G até o aquele momento da execução. As demais funções são definidas de acordo com o algoritmo escolhido. O algoritmo procede avaliando um estado (Q, K) a cada passo. Uma forma de interpretar isso é a Definição 6. A Figura 8 ilustra essa ideia.

Definição 6. *A árvore de estados $\mathcal{T}(G)$ do Algoritmo MCBB(G) é uma árvore binária enraizada no primeiro estado criado na Linha 7. À medida que o algoritmo é executado, cada um de seus nós é um estado (Q, K) da pilha $\mathcal{S}$. O filho direito de um estado (Q, K) é criado na Linha 11 e o seu filho esquerdo é criado na Linha 12.*

FIGURA 8 – Um estado (Q, K) tem dois filhos: um que aumenta a clique Q com um vértice $x \in K$ e atualiza o conjunto de candidatos K; outro que apenas atualiza o conjunto de candidatos K removendo um vértice x.



A execução do algoritmo se dá, inicialmente, criando um estado (Q, K). Em seguida, o passo realizado na Linha 9 é chamado de *passo bounding*, realizado através da função $\mathsf{bound}(G, Q, K)$. Nesse passo é calculado um limitante para o tamanho da clique máxima. Dentro do laço iniciado na Linha 9, as linhas de 10 até 12 formam o *passo branching*. Nesse momento o problema é subdivido em subproblemas menores. Aqui, são criados dois novos estados com base num vértice x, removido do conjunto K na Linha 10, chamado de pivô. Cada novo estado criado é pré processado nas duas chamadas da função $\mathsf{pre\text{-}process\text{-}state}(G, Q, K, C)$, sendo a primeira na Linha 8 e a segunda na Linha 13. O algoritmo progride recursivamente até esgotar a pilha de estados $\mathcal{S}$. O passo da Linha 15 atualiza o conjunto C que devolve uma clique máxima do dado grafo.

Para se obter um algoritmo mais rápido é preciso definir estratégias para diminuir o número de estados na árvore de estados. Essa estratégia passa por, por exemplo, utilizar coloração de vértices como limitante para o tamanho da clique máxima. Os algoritmos estudos nesse trabalho são algoritmos que fazem uso dessa estratégia e são discutidos na Seção 3.3.1. Por certo, além dessa estratégia, existem outras disponíveis na literatura. Na seção seguinte será apresentado um algoritmo que, inicialmente, não computa um limitante e que será modificado para calcular uma coloração de vértices.

3.2.1 Algoritmo NoBound

O Algoritmo NoBound é o algoritmo mais simples da lista de algoritmos que podem ser descritos sob MCBB. Isso porque, tal algoritmo não aplica nenhum tipo de estratégia com o objetivo de diminuir o espaço de busca, isto é, ele realiza todos passos *branching* possíveis. Logo, esse algoritmo enumera todas cliques de um grafo dado. Particularmente, todas cliques estarão em uma folha da árvore de estados.

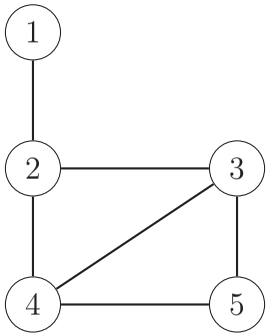
Para conceber NoBound, utilizando o Algoritmo $\mathsf{MCBB}(G)$, suas funções podem ser especializadas da seguinte forma: Nas Linhas 8 e 13, as chamadas para a função $\mathsf{pre\text{-}process\text{-}state}(G, Q, K, C)$ não fazem nada; as demais funções são definidas como segue. Antes, note que a desigualdade da Linha 9 será sempre satisfeita se a função $\mathsf{bound}(G, Q, K)$ retornar um valor como $|V(G)| + 1$.

- $\mathsf{pre\text{-}process}(G)$: retorna o estado $(\emptyset, V(G))$ e o conjunto $C = \emptyset$.

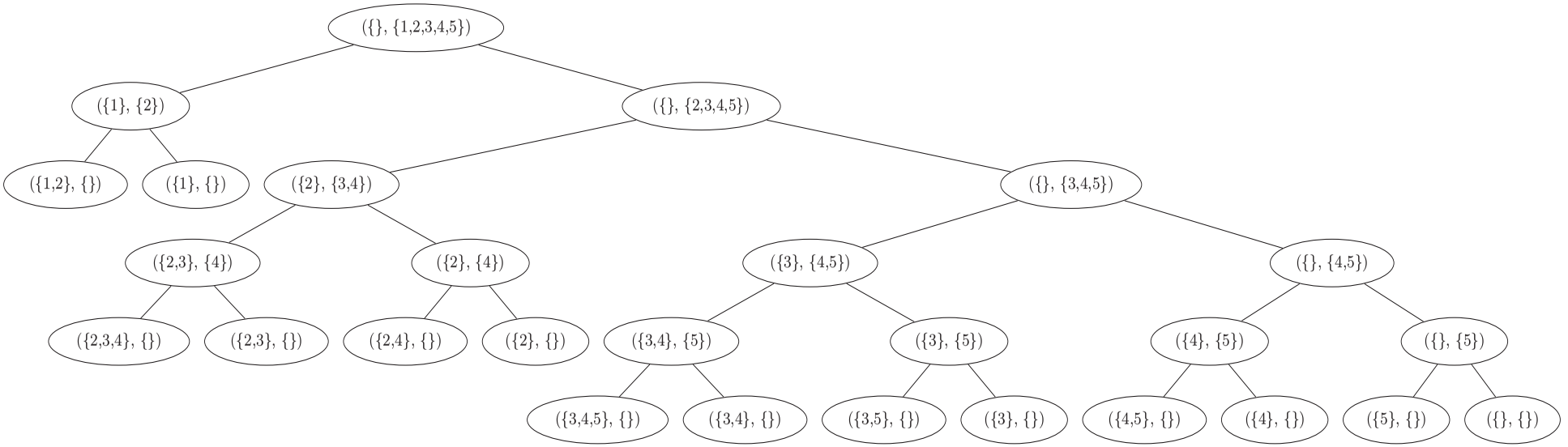
- bound(G, Q, K): devolve $|V(G)| + 1$.
- remove(K): remove e devolve o último vértice de uma sequência do conjunto K.
- post-process(G, C): devolve C.

Um exemplo da execução do Algoritmo NoBound está na Figura 9.

FIGURA 9 – Execução do algoritmo NoBound com o grafo (9a) como entrada. (9b) apresenta a árvore de estados percorrida pelo algoritmo.



(a) Grafo



(b) Árvore de estados

3.2.2 Algoritmos Basic, CP e DF

A partir do Algoritmo NoBound é possível implementar outros algoritmos como Basic (CARMO; ZÜGE, 2012) e CP. O Algoritmo Basic é concebido tendo em vista a observação de que $|Q| + |K|$ define um limitante superior para o tamanho da maior clique no grafo. É possível computar esse limitante com uma alteração simples na função bound(G, Q, K), do Algoritmo NoBound, que passa a devolver $|K|$.

O Algoritmo CP aproveita a mesma estratégia empregada no Algoritmo Basic no passo *bounding*. A diferença entre esses dois algoritmos é a estratégia de *branching*, sendo que, CP escolhe vértices com o menor grau como pivô. Do trabalho original (CARRAGHAN; PARDALOS, 1990), a função pre-process(G) realiza uma ordenação inicial $K = (x_1, x_2, \ldots, x_{|V|})$ do conjunto vértices V do grafo G de maneira que para todo $i \in \{1, 2, \ldots, |V|\}$ o vértice x_i é o vértice de grau mínimo no grafo induzido $G[\{x_1, x_2 \ldots, x_i\}]$. Por fim, um vértice pivô é escolhido ao fazer remove(K) retornar o vértice de menor grau em K seguindo tal ordenação inicial.

O Algoritmo DF, apresentado em Fahle (2002), apresenta uma estratégia chamada de *filtro de domínio* no pré-processamento, antes dos passos de BB. Isso se dá pela remoção dos vértices com graus menores que $|C| - |Q|$ do conjunto K no grafo induzido $G[K]$ em um estado (Q, K), já que esses vértices não podem fazer parte de uma clique maior que C em G. No grafo $G[K]$ resultante dessa operação, os vértices com grau igual a $|K| - 1$ são movidos para Q. Essa operação deve acontecer em todo estado da árvore de estados na execução de pre-process-state(G, Q, K, C), que implementa a operação descrita acima. Enfim, o Algoritmo DF fica completo com as demais funções de MCBB implementadas iguais à Basic.

Nesse trabalho, os algoritmos de interesse são aqueles que fazem uso de coloração de vértices para alcançar limitantes mais justos com relação ao tamanho da maior clique de um grafo. Para tanto os algoritmos descritos até agora serão modificados. A seguir são apresentados conceitos e resultados acerca da coloração de vértices.

3.3 COLORAÇÃO DE VÉRTICES

A coloração de vértices de um grafo está relacionada ao problema de particionar um conjunto de objetos em classes distintas onde os membros de cada classe possuem determinada característica. Esse é um problema de grande interesse que conta com uma extensa literatura, sendo revisado em trabalhos recentes como Malaguti e Toth (2010) e Galinier et al. (2013). Exemplos de aplicações surgem no escalonamento de tarefas (BONDY; MURTY, 2008), química (BONDY; MURTY, 2008) e, em jogos como Sudoku (LEWIS, 2015).

De modo geral todo grafo G satisfaz a inequação

$$\chi(G) \geq \frac{|V(G)|}{\alpha(G)} \tag{3.1}$$

uma vez que cada conjunto de cor é um conjunto independente contendo no máximo $\alpha(G)$ vértices. A Equação 3.1 fornece uma noção do porque o número cromático de um grafo pode ser arbitrariamente grande. Além disso, se existe uma cópia de um grafo completo K_r, então $\chi(G) \geq r$. Assim, para todo grafo G,

$$\chi(G) \geq \omega(G).$$

Como já citado, encontrar o número cromático de um grafo é um problema $\mathcal{NP}$-difícil cuja versão de decisão se encontra na lista de 21 problemas $\mathcal{NP}$-completos de Karp (1972). Além disso, encontrar uma k-coloração tal que $k \geq 3$ é difícil do ponto de vista computacional. Na aproximação do problema da coloração de vertices o resultado disponível na literatura se iguala ao resultado obtido na aproximação do CM, ou seja, uma aproximação da forma $n^{1-\varepsilon}$ é $\mathcal{NP}$-difícil para todo $\varepsilon > 0$ (ZUCKERMAN, 2006). Assim, a menos que $\mathcal{P} = \mathcal{NP}$, não existe um algoritmo de tempo polinomial que aproxime o problema da coloração por um fator melhor que $O(n^{1-\varepsilon})$ (LUND; YANNAKAKIS, 1994; GAREY; JOHNSON, 2002).

Ainda que computar o número cromático de um grafo seja $\mathcal{NP}$-difícil, é possível computar *uma* coloração de seus vértices em tempo polinomial. Para tanto, basta tomar o conjunto de vértices em uma sequência predefinida e colorir cada vértice com a menor cor disponível sem que dois vértices vizinhos recebam a mesma cor. Essa abordagem pode ser feita por um *algoritmo guloso* como o descrito por $\mathsf{Greedy}(G, \pi)$, o qual, colore o conjunto de vértices do grafo G um a um na ordem em que aparecem numa ordenação π dos vértices. Note que colorir um grafo com menor número de cores possível com o algoritmo $\mathsf{Greedy}(G, \pi)$ depende da ordenação π, fato discutido adiante.

Numa coloração gulosa a ordenação dos vértices pode ter grande impacto no número de cores usadas. Por um lado, é possível provar que, dependendo da ordenação fornecida a um algoritmo guloso, é possível colorir os vértices de um grafo de maneira que número de cores usadas seja igual a $\chi(G)$. Por outro lado, também é possível provar que uma determinada ordenação pode levar o algoritmo guloso a usar um número de cores distante do ótimo. Um exemplo desse fato é considerar um grafo bipartido completo $K_{n/2,n/2}$ com partes $X \coloneqq \{x_1, x_2, \ldots, x_{n/2}\}$ e $Y \coloneqq \{y_1, y_2, \ldots, y_{n/2}\}$. Ao remover as arestas $\{x_i y_i : 1 \leq i \leq n/2\}$ desse grafo o número de cores usadas pelo algoritmo $\mathsf{Greedy}(G, \pi)$ será $n/2$, considerando uma ordenação dos vértices $\pi = x_1, y_1, x_2, y_2, \ldots, x_{n/2}, y_{n/2}$. Porém, considerando a ordenação $\pi = x_1, x_2, \ldots, x_{n/2}, y_1, y_2, \ldots, y_{n/2}$ o algoritmo guloso utilizará apenas duas cores.

Outro exemplo do impacto da ordenação utilizada numa coloração gulosa é o limitante em Welsh e Powell (1967), conhecido como *limitante de Welsh e Powell*: dados

Greedy(G, π)

Entrada : Um grafo $G = (V, E)$
Uma ordenação $\pi = (x_1, x_2, \ldots, x_n)$ do conjunto V
Saída : Uma coloração dos vértices de G

```
1  c ← um vetor indexado por inteiros contendo uma lista para cada inteiro
2  cor ← 1
3  Para i ← 1 até |V|
4  |  x ← π(i)
5  |  k ← 1
6  |  Enquanto k ≤ cor e N(x) ∩ c[k] ≠ ∅
7  |  |  k ← k + 1
8  |  Se k > cor
9  |  |  cor ← k
10 |  Acrescente x ao final de c[k]
11 Devolva c
```

um grafo G e uma sequência π do conjunto de vértices em ordem não crescente de graus o algoritmo Greedy(G, π) utiliza no máximo $\Delta(G) + 1$ cores.

Ainda assim, independentemente da ordem, o número de cores usadas pelo algoritmo Greedy(G, π) não passa de $\Delta(G) + 1$ (BONDY; MURTY, 2008, p. 360). Quando um vértice v está para ser colorido, o número de vizinhos já coloridos é no máximo seu grau $d(v)$, que não é maior que $\Delta(G)$. Assim, umas das cores $\{1, 2, \ldots, \Delta(G) + 1\}$ certamente será atribuída a v. Com isso, para qualquer grafo G,

$$\chi(G) \leq \Delta(G) + 1. \tag{3.2}$$

Ainda sobre a ordenação dos vértices utilizada por um algoritmo guloso, em Maffray (2003) é feita uma revisão aprofundada sobre os fatos que vem a seguir. É conhecido que todo grafo tem uma ordenação de seus vértices que leva a uma coloração ótima por um algoritmo guloso. Já que existem $n!$ diferentes permutações dos vértices de um grafo, encontrar uma ordenação perfeita é difícil computacionalmente. Apesar disso, uma ordenação computado por LexBFS tem as propriedades do Teorema 12 em várias classes de grafos. Seja π uma ordenação do conjunto de vértices de um grafo, $\pi^{-1}(x) = i$ significa a posição i do vértices x na ordenação π.

Teorema 12 (Berge e Chvátal (1984))**.** *Uma ordenação π dos vértices do grafo G é perfeita se e somente se não existe um P_4 com vértices a, b, c, d tal que $\pi^{-1}(a) < \pi^{-1}(b)$ e $\pi^{-1}(d) < \pi^{-1}(c)$.*

No caso dos cografos, qualquer ordenação dos vértices é uma ordenação perfeita, já que os cografos não possuem P_4 como subgrafo induzido. Já para os grafos cordais, a ordem $(x_1, x_2, \ldots, x_i)$ onde o vértice x_i é simplicial no grafo $G[\{x_1, x_2, \ldots, x_i\}]$ é uma ordenação perfeita. O Teorema 5 garante que uma ordem perfeita de eliminação produzida

pelo Algoritmo LexBFS satisfaz a condição de ordenação perfeita. Por fim, pelo Teorema 12, qualquer ordem perfeita de eliminação é uma ordenação perfeita. Assim, existe algoritmo de tempo polinomial para encontrar uma coloração ótima nessas classes.

Observação 4 (Hoàng e Mahadev (1989))**.** *Dado um grafo G que admite uma ordenação $\pi = (x_1, x_2, \ldots, x_i)$ onde x_i é simplicial no grafo $G[\{x_1, x_2, \ldots, x_i\}]$ para todo i, o Algoritmo Greedy(G, π) colore os vértices de G com uma coloração ótima.*

Outro algoritmo de complexidade polinomial resolve o problema da coloração de vértices através de uma heurística gulosa é o Algoritmo DSATUR, proposto em Brélaz (1979). Os vértices são coloridos um a um com a cor de menor número possível, começando pelo vértice de maior grau. Para determinar qual o próximo vértice a ser colorido é escolhido o vértice com maior *grau de saturação*, quer dizer, o vértice que possuir o maior número vizinhos já coloridos. Naturalmente, podem surgir empates que são resolvidos escolhendo o vértice de maior grau. Se ainda existirem empates, o algoritmo escolhe arbitrariamente um vértice.

DSATUR(G, π)

Entrada : Um grafo $G = (V, E)$
Uma ordenação $\pi = (x_1, x_2, \ldots, x_n)$ do conjunto V
Saída : Uma coloração dos vértices de G

```
1  c ← um vetor indexado por inteiros contendo uma lista para cada inteiro
2  cor ← 1
3  x ← um vértice de grau máximo em π
4  Acrescente x ao final de c[cor]
5  Enquanto existirem vértices não coloridos
6  |  y ← um vértice com o maior grau de saturação. Desempate escolhendo um
   |    vértice de maior grau. Se ainda existirem empates, escolha um vértice
   |    arbitrariamente.
7  |  k ← 1
8  |  Enquanto k ≤ cor e N(y) ∩ c[k] ≠ ∅
9  |  |  k ← k + 1
10 |  Se k > cor
11 |  |  cor ← k
12 |  Acrescente y ao final de c[k]
13 |  Marque y como colorido
14 Devolva c
```

A seguir, são apresentados algoritmos para o CM que utilizam coloração de vértices. Tais algoritmos utilizam Greedy ou DSATUR com alguma modificação para encontrar limitantes para o CM. Além desses, alguns ainda propõem outras heurísticas baseadas em coloração de vértices.

3.3.1 Algoritmos para Clique Máxima que Utilizam Coloração de Vértices

Nessa seção, serão apresentados algoritmos BB para o CM que utilizam coloração de vértices como limitante superior para maior clique em um dado grafo. São implementadas variações da coloração gulosa, ordenando de diferentes formas o conjunto de vértices a fim de alcançar uma coloração com número de cores que seja próximo ao número cromático χ. Tais heurísticas são aplicadas no passo *branch* e passo *bound*. Assim, é possível continuar utilizando o algoritmo MCBB como framework de implementação.

Nesse sentido, o trabalho de San Segundo, Lopez et al. (2016) aborda novas estratégias de ordenação inicial dos vértices – correspondente a função pre-process(G) no algoritmo MCBB – apresentando resultados da coloração aproximada dos algoritmos MCS e BBMCX.

Em Lavnikevich (2013) é apresentado um limitante inferior para a complexidade de tempo, proporcional a $\Omega(2^{n/5})$ para algoritmos que utilizam coloração de vértices como limitante superior. Esse resultado foi aproveitado por Züge (2017) como estabelecido no Teorema 13. Assim, o Algoritmo MCBB(G) não afasta a complexidade de tempo de execução exponencial se o algoritmo apenas colore vértices para computar um limitante superior para a clique máxima.

Teorema 13 (Züge (2017))**.** *Qualquer algoritmo da família MCBB que utiliza apenas coloração de vértices como limitante superior tem tempo de execução* $\Omega(2^{n/5})$.

Apesar do resultado do Teorema 13, uma análise detalhada mostra que em certas instâncias de grafos, quando previamente identificadas, a complexidade de tempo desses algoritmos é polinomial. Assim, a família MCBB foi expandida para que seja possível tratar instâncias formadas pela operação de junção entre subgrafos. Essa mesma operação ocorre na classe dos cografos.

Em seguida, serão apresentados algoritmos que utilizam coloração de vértices na função bound(G, Q, K).

3.3.1.1 Algoritmos CHI e CHI + DF

Os Algoritmos CHI e CHI + DF, ambos apresentados no trabalho de Fahle (2002), são modificações diretas dos algoritmos vistos anteriormente (Seção 3.2.2) CP e DF sob o framework MCBB respectivamente.

Em CHI o passo de *bounding* é implementado utilizando dois algoritmos de coloração gulosa. A primeira consiste em construir um conjunto independente para cada cor. Quando o conjunto atual não puder mais ser estendido um novo é criado. O número de conjuntos necessários para que todo vértice pertença a um conjunto independente será o número de cores. A outra coloração utilizada é a mesma que o Algoritmo DSATUR. Dentro da função bound(G, Q, K), é oferecida como entrada para as duas heurísticas, os vértices de K em ordem não decrescente de graus, e essa mesma ordem invertida. Com

isso, são realizadas quatro colorações no passo *bounding* do Algoritmo MCBB, das quais, o menor número de cores computado será escolhido.

Para completar o Algoritmo CHI sob MCBB, o restante de suas funções devem ser implementadas como o Algoritmo Basic, anteriormente discutido na Seção 3.2.2 desse capítulo.

O Algoritmo CHI + DF é a união dos algoritmos CHI e algoritmo DF, também na Seção 3.2.2. Isso porque o limitante superior alcançado por uma coloração pode ser melhor que o alcançado pelo filtro de domínio (FAHLE, 2002).

3.3.1.2 Algoritmo MCLIQ

O Algoritmo MCLIQ foi proposto no trabalho de Tomita, Kohata e Takahashi (1988), sendo antecessor de algoritmos como MCQ, na Seção 3.3.1.3. O Algoritmo MCLIQ foi aproveitado nesse trabalho pois utiliza coloração dos vértices nos passos *branching* e *bounding*.

Dado um grafo $G = (V, E)$, inicialmente é feita uma ordenação do conjunto de vértices K em ordem não crescente de graus. Assim, a ordenação do conjunto de candidatos K será utilizada como entrada para o Algoritmo Greedy nos passos seguintes. O detalhe aqui é que a ordenação inicial dos vértices deve ser mantida conforme os estados da árvore de estados vão sendo avaliados, isto é, o filho de um estado mantém a ordenação estável em relação ao seu pai.

Tal coloração gulosa tem por característica que ao final do sua execução cada vértice tem vizinhos em cores menores que a sua. Uma vez que uma clique tem todos vértices em cores diferentes umas das outras, faz sentido escolher um vértice pivô em MCBB com maior cor. Concluindo, o passo *branching* se da pela escolha do vértice com maior cor na coloração computada por Greedy.

Com essa forma de escolher o vértice pivô x não é preciso colorir os vértices de K nos filhos direitos (Linha 11), pois é mantida a mesma coloração de K relativa ao estado pai, exceto a cor do vértice pivô removido.

Por fim, o passo *bounding* apenas aproveita o número de cores utilizada pelo Algoritmo Greedy para colorir o grafo $G[K]$.

É possível modificar o Algoritmo Basic (Seção 3.2.2) para descrever o Algoritmo MCLIQ da seguinte forma:

- pre-process(G): devolve o estado $(\emptyset, K)$ onde K é composto pelos elementos de V em ordem não crescente de graus.
- pre-process-state(G, Q, K, C) (Linha 8): devolve o estado (Q, K) igual ao estado pai.
- pre-process-state(G, Q, K, C) (Linha 13): colore os vértices de $G[K]$ com Greedy; faz uma ordenação de K em ordem não decrescente de cores estável em relação a ordenação de K no estado pai.

- bound(G, Q, K): devolve o número de cores computado em pre-process-state(G, Q, K, C).
- remove(K): remove e devolve o último vértice do conjunto K, i.e, o vértice com maior cor em $G[K]$.
- post-process(G, C): devolve C.

3.3.1.3 Algoritmo MCQ

O Algoritmo MCQ (TOMITA; SEKI, 2003), originalmente é uma modificação de CP. Porém, sob MCBB, também é facilmente percebido como uma modificação do Algoritmo MCLIQ. De maneira análoga ao MCLIQ, em MCQ é feita ordenação inicial dos vértices em ordem não crescente de graus. Porém, já que nenhuma clique é maior que $\Delta(G) + 1$, esse limitante pode ser aproveitado já em pre-process(G). Para alcançar esse resultado MCQ faz uso de um conjunto auxiliar, chamado KN, que guarda uma *numeração* que pode ser implementada como descrito em seguida.

pre-process(G)

```
1 K ← V em ordem não crescente de graus
2 Para i ← 1 até Δ(G)
3 |   KN[K[i]] ← i
4 Para i ← Δ(G) + 1 até |K|
5 |   KN[K[i]] ← Δ(G) + 1
```

Assim, para terminar a descrição do Algoritmo MCQ, as demais funções de MCBB são iguais em MCLIQ, anteriormente apresentadas.

3.3.1.4 Algoritmo DYN

No trabalho de Konc e Janezic (2007), o Algoritmo DYN propõe alterações em MCQ com relação à coloração gulosa no passo de *branching*.

A função pre-process-state(G, Q, K, C) reordena o conjunto K em ordem não crescente de graus. Uma vez que realizar essa ordenação em todo estado pode piorar o tempo de execução do algoritmo, a estratégia é reordenar os vértices de K em alguns estados. Para isso, considerando uma execução, o algoritmo conta o número de estados (Q', K') na árvore de estados já percorrida que satisfazem $|Q'| \leq |Q|$ onde Q é a clique no momento da execução de pre-process-state(G, Q, K, C). Enquanto esse número for menor que 2.5% do total de estados percorridos, tal reordenação de K é realizada.

Ainda na mesma função, em seguida, a coloração dos vértices em K no estado (Q, K) é feita de maneira que todo vértice $k \in K$ com cor $c(k) \leq |C| - |Q|$ mantenham a mesma ordem relativa a que estavam no estado pai. Isso corresponde a uma alteração na função Greedy.

Sob o Algoritmo MCBB, a implementação de DYN tem suas funções definidas como em MCQ, alterando apenas uma delas:

- pre-process-state(G, Q, K, C) (Linha 8): devolve o estado (Q, K) igual ao estado pai.
- pre-process-state(G, Q, K, C) (Linha 13): reordena K enquanto poucos estados (Q',K') satisfazem $|Q'| \leq |Q|$ no estado atual; colore $G[K]$ com Greedy e ordena K em ordem não decrescente de cores mantendo os vértices com cores menores à $|C| - |Q|$ em uma ordenação estável em relação ao estado pai.

3.3.1.5 Algoritmo MCR

O Algoritmo MCR (TOMITA; KAMEDA, 2007) é uma modificação de MCQ. O que diferencia esses dois algoritmos é o pré-processamento realizado no grafo. Em MCR, a função pre-process(G) faz uma ordenação do conjunto de vértices em ordem não decrescente de graus como em CP e aplica uma numeração similar a feita por MCQ para calcular os limitantes superiores para clique máxima. Considerando uma implementação sob o MCBB, apenas a função pre-process(G) precisa ser modificada. As demais funções permanecem iguais a implementação do Algoritmo MCQ.

- pre-process(G): devolve $(\emptyset, K)$ onde conjunto $K = V$ é ordenado tal que o vértice k_i para todo $i \in \{1, 2, \ldots, |K|\}$ é o vértice de menor grau em $G[\{k_1, k_2, \ldots, k_i\}]$; em caso de empates, o vértice com menor soma dos graus de seus vizinhos em $G[\{k_1, k_2, \ldots, k_{i-1}\}]$ é escolhido.

3.3.1.6 Algoritmo MCS

O Algoritmo MCS (TOMITA; SUTANI et al., 2010) pode ser visto como uma evolução Algoritmo MCR, pois aproveita suas ideias de pré-processamento, mas modifica a ordenação inicial. Além disso, o algoritmo pressupõe que a representação do grafo G é feita utilizando uma matriz de adjacências.

- pre-process(G): inicializa o conjunto K com vértices de G ordenados como em MCR; modifica a matriz de adjacências de maneira que a ordem dos vizinhos de um vértice é equivalente a ordenação inicial de K.

Outra mudança presente em MCS, é no Algoritmo Greedy(G, π) utilizado em pre-process-state(G, Q, K, C). Essa mudança aproveita o fato de que vértices com cores menores que $|C| - |Q|$ não podem levar a uma clique Q maior que a clique máxima C encontrada até esse ponto da execução. Se um vértice x tem seus vizinhos todos coloridos com cores menores que $|C| - |Q|$, então uma cor que contém um único vértice y adjacente é procurada e o algoritmo tenta recolorir x e y de forma que ambos obtenham cores menores que $|C| - |Q|$.

Para completar o Algoritmo MCR, na função post-process(G, C) são desfeitas as alterações na matriz de adjacências.

3.4 MODIFICAÇÃO COM O ALGORITMO LEXBFS

Nessa seção, os algoritmos para o CM baseados em coloração de vértices, descritos na Seção 3.3.1, serão descritos utilizando o Algoritmo LexBFS, descrito no início do Capítulo 2.

Como já é conhecido, as instâncias de grafos cordais e cografos fazem parte do conjunto de classes que satisfazem as condições para uma ordenação perfeita. Para essas classes, dado um grafo G, uma ordenação do conjunto de vértices produzida por LexBFS(G) fornecida como entrada para Algoritmo Greedy(G, π) produz uma coloração ótima (Observação 4).

Faz sentido uma alteração nos algoritmos para o CM de maneira que uma ordenação LexBFS seja previamente computada antes de colorir os candidatos à clique máxima com um algoritmo guloso. Isso equivale modificar o passo *bounding* dos algoritmos descritos sob MCBB. Com isso, o número de cores computado em cada algoritmo modificado com LexBFS pode ser menor. A fim de explorar essa possível vantagem, os algoritmos CHI, CHI + DF, MCLIQ, MCQ, MCR, MCS e DYN podem ser modificados para colorir os vértices do grafo $G[K]$ na ordem que aparecem em uma ordenação produzida pelo Algoritmo LexBFS.

No Capítulo 4, serão apresentados resultados dos algoritmos modificados comparados contra os resultados de suas versões originais. As próximas seções apresentam as modificações feitas.

3.4.1 Algoritmos CHI$_L$ e CHI$_L$ + DF

No Algoritmo CHI, utilizando duas heurísticas na função bound(G, Q, K), são realizadas quatro colorações tomando como entrada os vértices do grafo induzido pelo conjunto de candidatos $G[K]$. Nessas duas heurísticas é feita uma ordenação inicial dos vértices de $G[K]$. A partir disso, basta substituir essa ordenação inicial pela ordenação de vértices gerada pelo Algoritmo LexBFS. Dessas modificações, surge o algoritmo que será chamado de CHI$_L$. Como o Algoritmo CHI + DF é apenas uma união de CHI com DF, a junção dos algoritmos CHI$_L$ e DF é o algoritmo que será chamado de CHI$_L$ + DF.

3.4.2 Algoritmo MCLIQ$_L$

Com uma coloração gulosa Greedy, o algoritmo MCLIQ faz uma coloração dos vértices candidatos a fazer parte de uma clique máxima. A seguinte alteração diz respeito ao algoritmo MCLIQ implementado sob MCBB. Considerando sua descrição na Seção 3.3.1.2, foi alterada a função pre-process-state(G, Q, K, C) para o seguinte:

- pre-process-state(G, Q, K, C) (Linha 8): devolve o estado (Q, K) igual ao estado pai.

- pre-process-state(G, Q, K, C) (Linha 13): ordena os vértices de K com LexBFS; faz uma coloração gulosa dos vértices do subgrafo induzido $G[K]$, considerando a ordem em que aparecem em K; faz uma ordenação estável do conjunto K em ordem não decrescente de cores.

O Algoritmo MCLIQ com essa modificação em pre-process-state(G, Q, K, C) será chamado de MCLIQ$_L$.

3.4.3 Algoritmo MCQ$_L$

Em relação ao Algoritmo MCQ implementado sob MCBB, descrito na Seção 3.3.1.3, ao aproveitar exatamente a mesma função pre-process-state(G, Q, K, C) (Linha 13) modificada no Algoritmo MCLIQ$_L$ e mantendo as demais funções como a proposta original de MCQ, emerge um algoritmo que será chamado de MCQ$_L$.

3.4.4 Algoritmo DYN$_L$

A modificação do Algoritmo DYN sob MCBB é simples. Esse algoritmo já realiza uma ordenação que será substituída por uma ordenação LexBFS do conjunto de vértices K candidatos à maior clique.

- pre-process-state(G, Q, K, C) (Linha 8): devolve o estado (Q, K) igual ao estado pai.
- pre-process-state(G, Q, K, C) (Linha 13): reordena K com uma LexBFS enquanto poucos estados (Q',K') satisfazem $|Q'| \leq |Q|$ no estado atual; colore $G[K]$ com Greedy e ordena K em ordem não decrescente de cores mantendo os vértices com cores menores a $|C| - |Q|$ em uma ordenação estável em relação ao seu estado pai.

As demais funções são iguais a MCQ, na 3.3.1.4. Esse algoritmo será chamado de DYN$_L$.

3.4.5 Algoritmo MCR$_L$

Considerando um implementação sob MCBB, uma das modificações com LexBFS mais triviais é a do Algoritmo MCR. É suficiente utilizar a mesma função pre-process-state(G, Q, K, C) de MCQ$_L$. O restante das funções são como em MCR, na Seção 3.3.1.5.

3.4.6 Algoritmo MCS$_L$

A função pre-process-state(G, Q, K, C) de MCS é modificada de forma que ordene o conjunto de vértices K segundo com LexBFS, antes de K ser colorido com uma modificação de Greedy. O restante das funções permanecem como descritas em 3.3.1.4.

- pre-process-state(G, Q, K, C)(Linha 8): devolve o estado (Q, K) igual ao estado pai.

- pre-process-state(G, Q, K, C)(Linha 13): ordena os vértices de K com LexBFS; colore K com Greedy tal que se um vértice x tem seus vizinhos todos coloridos em cores menores que $|C| - |Q|$, então uma cor que contém um único vizinho y é procurada; tenta recolorir x e y de forma que ambos obtenham cores menores $|C| - |Q|$.

  Essa modificação de MCS será chamada de MCS_L.

4 ANÁLISE EXPERIMENTAL

A Análise Experimental de Algoritmos visa proporcionar discernimento em relação a corretude, desempenho e outras informações úteis que um algoritmo fornece. Também pode ser empregada quando o objetivo é escrever algoritmos melhores ou mais rápidos. Para alcançar tais objetivos é possível tratar algoritmos como objetos de laboratório, focando no controle de parâmetros, isolamento de componentes chave, construção de um modelo e análise estatística (MCGEOCH, 2012). Dito isso, a abordagem da análise experimental não deve substituir uma análise teórica ou uma análise puramente empírica para entender a desempenho de algoritmos, mas, ao invés disso, somar-se com essas abordagens para produzir um resultado verossímil.

De maneira geral, serão utilizadas definições e processos propostos em McGeoch (2012), onde é discutida de forma extensiva os principais pontos da condução de uma análise experimental de algoritmos, contanto ainda com exemplos de análise de experimentos planejados para problemas $\mathcal{NP}$-difíceis.

Inciando com a Análise Experimental de Algoritmos, existe um ciclo formado pelas principais etapas e conceitos, são eles:

- **Planejar Experimento**: o planejamento experimental é a fase onde são feitas as formulações iniciais de aspectos como
  - **Formular Pergunta**: a formulação de uma pergunta a ser respondida com um experimento.
  - **Ambiente computacional**: a definição do ambiente computacional, ferramentas de mensuração, programa de teste. Envolve determinar instâncias de entrada — caso necessite, decidir métodos para geração de instâncias de entrada.
  - **Experimento**: decidir quais propriedades serão mensuradas (tais como tempo de execução e consumo de memória). Também identificar e decidir sobre parâmetros que afetem a mensuração dos resultados, sendo tais parâmetros relacionados à instância de entrada, ao ambiente computacional ou mesmo ao algoritmo. Nessa etapa também são escolhidas instâncias e número de testes que serão realizados.
- **Executar experimento**: ao executar experimentos é preciso se preocupar com questões relacionadas a coleta e análise de dados.
  - **Executar testes**: testar a implementação de um algoritmo analisado sobre uma entrada escolhida e coletar dados.

– **Analisar de dados**: análise de maneira que seja possível minimizar o esforço do leitor em ter que fazer conclusões, realizando ajustes no número de testes e dados amostrais, caso seja necessário.

– **Reportar resultados**: idealmente nesse momento são consideradas novas questões, o que reinicia todo o processo, começando pelo planejamento.

Uma das referências para os experimentos conduzidos está em Anjos (2015), onde os conceitos de análise experimental de algoritmos citados anteriormente são abordados de maneira mais aprofundada. São aplicados tais conceitos para a análise de algoritmos BB para o CM presentes na Seção 3.2.1, Seção 3.2.2 e Seção 3.3.1, todos implementados sob o Algoritmo MCBB.

Seguindo as etapas de uma análise experimental, a definição de ambiente computacional, detalhes da implementação e endereço para o repositório de acesso público onde se encontram junto aos resultados obtidos estão no Apêndice B. As instâncias de entrada são indicadas na Seção 4.2. Para mensurar o desempenho dos algoritmos, será aproveitada a ideia de árvore de estados (Definição 6) e tempo de execução em CPU. Logo, dado um grafo G, o interesse é mensurar $|\mathcal{T}(G)|$, o número de estados criados em uma árvore de estados por um algoritmo tendo G como entrada. Dessa forma, um dos parâmetros de medida de desempenho será $|\mathcal{T}(G)|$. Uma observação nesse sentido é que uma árvore de estados menor não necessariamente implica que um algoritmo toma um tempo de execução menor.

Dada a natureza das instâncias de entrada tratadas aqui, a análise e exposição dos dados é feita através da método estatístico conhecido como testes de hipótese. Para tanto, conceitos de estatística e probabilidade se fazem necessários. Esse procedimento é abordado com mais detalhes nas seções 4.1 e 4.3.2.1.

4.1 INFERÊNCIA ESTATÍSTICA

O objetivo dessa seção é revisar conceitos de probabilidade e estatística utilizados na Seção 4.2 que trata de instâncias de grafos aleatórios e Seção 4.3.2.1 onde são feitas comparações entre os diferentes algoritmos apresentados anteriormente. Os material a seguir foi feito a partir das referências de Prévôt e Röckner (2007), Bussab e Morettin (2009), Stroock (2010) e Devore (2015). Em seguida, de maneira breve, são apresentadas definições fundamentais de probabilidade e estatística.

Inicialmente, *população* é o conjunto de todos elementos sob investigação. O conceito de *amostra* é definido como qualquer subconjunto de uma população. Um *parâmetro* é uma mensuração ou medida de todos elementos da população. Uma *estatística* é uma mensuração ou medida da amostra.

Um *espaço de probabilidade* é uma tripla $(\Omega, \mathcal{F}, P)$: Ω é um conjunto chamado de *espaço amostral*; $\mathcal{F}$ é σ-álgebra de subconjuntos de Ω, isto é, $\mathcal{F}$ é fechado sob as operações

de união e complementação de conjuntos; P é uma função que associa uma *probabilidade* entre 0 e 1 a cada elemento de $\mathcal{F}$, e $P(\Omega) = 1$. Uma *variável aleatória* (v.a.) X é uma função no espaço de probabilidade, $X : \Omega \to \mathbb{R}$. Assim, $X(w) = x$ significa que x é o valor associado ao elemento $w \in \Omega$ pela v.a. X. Se a imagem da v.a. X é um conjunto contável, então X é chamada de v.a. *discreta*. A *distribuição de probabilidade* de uma v.a. discreta X, com valores associados $(x_1, x_2, \ldots, x_n)$, é definida por

$$p(x_i) = P(X = x_i) = P(\forall w \in \Omega : X(w) = x_i), i \in \{1, 2, \ldots, n\}.$$

A *função densidade* ou *função massa de probabilidade* de uma v.a. discreta X com valores $(x_1, x_2, \ldots, x_n)$ com probabilidade $p(x_i)$ é definida, para todo x, por

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i) = \sum_{x_i \leq x} p(x_i).$$

Para qualquer valor de x, $F(x)$ é a probabilidade dos valores observados de X serem no máximo x.

O *valor médio* ou *valor esperado* de uma v.a. discreta X que assume valores $(x_1, x_2, \ldots, x_n)$ é $E(X) = \mu = \sum_{i=1}^{n} x_i p(x_i)$. A *variância* da v.a. X é $\sigma^2 = \sum_{i=1}^{n} (x_i - \mu)^2 p(x_i)$. O *desvio padrão* da v.a. X, denotado por σ, é $\sigma = \sqrt{\sigma^2}$.

Uma v.a. X é dita ter uma *distribuição normal* com parâmetros μ, $-\infty < \mu < \infty$ e σ^2, $-\infty < \sigma^2 < \infty$, se sua função densidade é

$$f(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}, -\infty < x < \infty.$$

Uma distribuição normal é comumente denotada $N(\mu, \sigma^2)$. Uma v.a. X com distribuição normal com parâmetros μ e σ^2 é denotada $X \sim N(\mu, \sigma^2)$.

Uma distribuição de probabilidade de uma v.a. discreta importante é a distribuição de Bernoulli, presente em modelos de grafos aleatórios e base para outras distribuições de probabilidade. Uma v.a. discreta X cujos seus valores possíveis são apenas o valor 1 com probabilidade p ou valor 0 com probabilidade $q = 1 - p$ é chamada de *variável aleatória de Bernoulli*. Os experimentos de uma v.a. de Bernoulli são chamados de *ensaios de Bernoulli*. Um exemplo frequente na literatura é o lançamento de uma moeda onde cada experimento tem dois resultados possíveis para se obter cara: *sucesso* (1), pois o resultado foi cara; *fracasso* (0), caso contrário. Numa sequências de n ensaios de Bernoulli, por conta da independência dos ensaios, a probabilidade de $k = 0, 1, \ldots, n$ sucessos e $n - k$ fracassos, com $P(X = 1) = p$ e $P(X = 0) = q$, é

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k} = \binom{n}{k} p^k q^{n-k}.$$

Bem como foram definidos parâmetros, serão definidas estatísticas úteis. Antes, observe que uma amostra de uma v.a. X obtida por um processo aleatório tem valores $x_1, x_2, \ldots, x_n$ que na verdade também são variáveis aleatórias, já que a cada vez que uma

amostra é obtida, cada x_i assume um valor aleatório. Portanto, uma *amostra aleatória* de tamanho n é uma n-upla ordenada $(X_1, X_2, \ldots, X_n)$, onde a v.a. X_i denota a i-ésima observação, tal que X_i para todo $i \in \{1, 2, \ldots, n\}$ tem a mesma distribuição de probabilidade e são independentes. Considerando uma amostra $(X_1, X_2, \ldots, X_n)$, a *média da amostra*, denotada $\overline{X}$, é dada por $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. A *variância da amostra*, denotada S^2, é dada por $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2$. O *desvio padrão*, denotado S, é dado por $s = \sqrt{S^2}$.

Nesse ponto, definimos que o valor calculado ou observado de uma estatística será denotado por uma letra minúscula. Assim, o valor calculado da média de uma amostra $\overline{X}$ será $\overline{x}$. Por exemplo, se a média da amostra $\overline{X}$ for calculada de maneira que o resultado obtido seja 5, então $\overline{x} = 5$. De forma análoga, os valores calculados da variância de uma amostra S^2 e desvio padrão S serão s^2 e s respectivamente.

Quando $\overline{X}$ é a média de uma amostra aleatória $(X_1, X_2, \ldots, X_n)$ de tamanho n de uma distribuição normal com média μ, a v.a.

$$T = \frac{\overline{X} - \mu}{S/\sqrt{n}}$$

tem uma distribuição de probabilidade chamada t de Student com $n-1$ graus de liberdade. Portanto, o valor observado de T será

$$t = \frac{\overline{x} - \mu}{s/\sqrt{n}}. \tag{4.1}$$

As estatísticas apresentadas serão aplicadas dentro do contexto de teste de hipótese, apresentado a seguir. Os passos do teste de hipótese que vem a seguir é conhecido como *teste de t de Student*.

Um *teste de hipótese* é um método que utiliza estatísticas para decidir entre duas afirmações contraditórias qual é correta em relação a um parâmetro θ. A primeira afirmação, chamada *hipótese nula*, denotada por H_0, é uma afirmação que se assume como verdadeira inicialmente. A segunda, *hipótese alternativa*, denotada por H_1, é uma afirmação que contradiz H_0. A hipótese nula será rejeitada em favor da hipótese alternativa, de acordo com o procedimento que vem em seguida. Dito isso, existem duas possíveis conclusões para um teste de hipótese: *rejeitar* H_0 ou *não rejeitar* H_0.

O *valor-p* é a probabilidade de se obter uma estatística igual ou maior que a observada em um experimento, considerando que a hipótese nula H_0 é verdadeira. Quer dizer, essa é a probabilidade da estatística observada de uma amostra ser mais provável ou não, supondo H_0 como verdadeira. Por exemplo, dado o valor-$p = 0{,}01$, estamos diante da chance pequena (1 em 100) de obter uma amostra que contradiz a hipótese nula, assumida como verdadeira. Se H_0 é de fato a afirmação correta em relação a um parâmetro, então uma amostra deveria ter uma estatística com maior probabilidade de ocorrência. Dito isso, a interpretação do valor-p é a seguinte: como este é um valor entre 0 e 1, quanto mais próximo de 0 mais evidência se tem para rejeitar a hipótese nula H_0 em favor da hipótese alternativa H_1.

É possível utilizar o valor-p para tomar decisões em relação à rejeição de H_0. Antes disso é definido um valor α, chamado *nível de significância*, que seja próximo de 0. Em seguida a regra para o tomar uma decisão adotada será

$$\text{rejeitar } H_0 \text{ se o valor-}p \leq \alpha,$$
$$\text{não rejeitar } H_0 \text{ se o valor-}p > \alpha.$$

No caso em que H_0 é rejeitada, então é assumida a hipótese alternativa H_1 como correta. A regra para a definição do nível de significância α pode ser arbitrária, porém os valores comumente utilizados são α iguais a $0{,}05; 0{,}01$ ou $0{,}001$.

Por fim, o que se faz é calcular o valor-p. Seja $F(t)$ a função de massa de probabilidade da v.a. T que assume valor t, temos que

$$\text{valor-}p = P(T \geq t) = P(T > t) = 1 - F(t). \tag{4.2}$$

Os passos para a construção de um de teste de hipótese podem ser seguidos como listado em seguida.

1. Definir a hipótese nula H_0 e a hipótese alternativa H_1.
2. Decidir qual estatística será utilizada
3. Fixar o nível de significância α do teste
4. Utilizar a observação da amostra para calcular o valor-p
5. Utilizando o valor calculado para o valor-p no Passo 4, rejeitar ou não a hipótese nula H_0.

Considere o problema de comparar dois parâmetros μ_1 e μ_2: duas amostras foram retiradas e com base nessas amostras deseja-se fazer inferências sobre $\mu_1 - \mu_2$, a diferença entre as duas médias. Um exemplo frequente desse problema é na avaliação entre dois tratamentos médicos onde são feitas a medidas de alguma característica de um individuo antes e depois da aplicação do tratamento. Esse problema pode ser entendido como o problema de comparar dois métodos aplicados em conjuntos ou indivíduos.

Assumindo duas amostras $(X_1, X_2, \ldots, X_n)$ e $(Y_1, Y_2, \ldots, Y_n)$ pareadas, isto é, os pares $(X_1, Y_2), (X_2, Y_2), \ldots, (X_n, Y_n)$. É possível definir a v.a. $D = X - Y$ onde $D_1 = X_1 - Y_1, D_2 = X_2 - Y_2, \ldots, D_n = X_n - Y_n$ é a diferença entre os pares. Observe que isso é uma redução do problema a uma população e não duas. Já que X e Y são normais é possível assumir que D tem distribuição normal $N(\mu_D, \sigma_D^2)$ com média μ_D e variança σ_D^2. Com isso,

$$\overline{D} = \frac{1}{n}\sum_{i=1}^{n} D_i = \overline{X} - \overline{Y}$$

e como

$$\mu_D = E(X) - E(Y) = \mu_1 - \mu_2$$

qualquer hipótese sobre $\mu_1 - \mu_2$ corresponde a uma afirmação sobre μ_D. Agora, pode-se utilizar o teste de hipótese de forma que hipótese nula seja $H_0 : \mu_D = 0$, a hipótese alternativa $H_1 : \mu_D \neq 0$ (ou $\mu_D \geq 0$ ou $\mu_D \leq 0$) e o valor da estatística t, que vem de 4.1, dado por

$$t = \frac{\overline{d} - \mu_D}{s_D/\sqrt{n}}. \tag{4.3}$$

Com isso, após definir α e calcular o valor-p pode ser calculado com base em 4.3 e pode-se tomar uma decisão entre rejeitar ou não a hipótese nula. Esse, portanto, é o método utilizado na Seção 4.3.2.1 para fazer inferência entre o desempenho de dois algoritmos. Por esse método, ao rejeitar a hipótese nula, existe diferença significativa entre os algoritmos comparados, e o valor $\overline{D}$ qual das média das amostras é maior.

A partir de uma amostra aleatória é possível estimar um intervalo que contenha o parâmetro μ de uma população sempre que esse intervalo for estimado um determinado número de vezes. Para isso, o que se faz inicialmente é definir o *nível de confiança*, denotado por γ, como o numero $\gamma := 1 - \alpha$. Esse intervalo é chamado de *intervalo de confiança* (IC) calculado como a seguir. Dados uma amostra aleatória $(X_1, X_2, \ldots, X_n)$ de tamanho n e o número γ, o intervalo

$$\left(\overline{x} - t_\gamma \frac{s}{\sqrt{n}}, \overline{x} + t_\gamma \frac{s}{\sqrt{n}}\right)$$

é o $100(1-\alpha)\%$ intervalo de confiança. No Anexo A, a Tabela 18 fornece valores para t_γ dado o nível de confiança γ. Na Tabela 18, uma coluna representa γ e uma linha presenta ν graus de liberdade. A ideia é que ao estimar uma certa quantidade de intervalos de valores plausíveis para um parâmetro θ não conhecido, uma proporção γ desses intervalos contenham o verdadeiro valor do parâmetro θ.

É possível estabelecer uma relação entre o teste de hipótese e o IC. Seja (θ_L, θ_U) um intervalo de confiança para um parâmetro θ com nível de confiança $100(1-\alpha)\%$. Então o teste $H_0 : \theta = \theta_0$ contra $H_0 : \theta \neq \theta_0$ com nível de significância α, rejeita H_0 se o valor nulo θ_0 não pertence ao IC, caso contrário aceitar H_0 se o valor pertence ao IC.

Por fim, para tomar uma decisão em um teste hipótese é conveniente determinar e reportar o valor-p junto ao teste com IC. Esse é o formato escolhido para reportar as tabelas listadas na Seção 4.3.2.1, tendo em uma coluna o valor-p e em outra o intervalo de confiança.

4.2 INSTÂNCIAS

Segue uma descrição das instâncias escolhidas como entrada para realização de testes computacionais.

Foram utilizadas instâncias de grafos proveniente do DIMACS Second Implementation Challenge[1] (JOHNSON; TRICK, 1996), que serão chamadas de *instâncias* DIMACS. Utilizar instâncias disponíveis publicamente é vantajoso pois são entradas frequentemente empregadas na análise experimental de algoritmos. Tais instâncias são utilizadas por diversos autores para testar e comparar algoritmos para o CM, dentre eles Fahle (2002), Tomita e Seki (2003), Konc e Janezic (2007), Tomita e Kameda (2007), Tomita, Sutani et al. (2010), Carmo e Züge (2012), Anjos, Züge e Carmo (2016) e Züge (2017). Uma característica das instâncias DIMACS é que são geradas para determinar questões de desempenho com viés extremamente pessimista e com modelos probabilísticos projetados para serem de pior caso (JOHNSON; TRICK, 1996).

Novamente, de acordo com McGeoch (2012), instâncias de entrada aleatórias são úteis, sobre certas circunstâncias, para mensurar o desempenho de algoritmos no caso médio, bem como, observar o alcance de todos possíveis resultados. Foram empregadas instâncias de grafos aleatórios como entrada para proceder com a análise experimental. O modelo $\mathcal{G}_{n,p}$ de grafos aleatórios (BOLLOBÁS, 2001) é o espaço de probabilidade em que seus pontos são grafos com conjunto de vértices $V = \{1, 2, \ldots, n\}$ onde as arestas são escolhidas independentemente com probabilidade p. Nesse modelo, a probabilidade de se obter um grafo H que tenha m arestas é

$$P(G = H) = p^m(1-p)^{N-m}$$

onde $N = \binom{n}{2}$.

Fixar $p = 1/2$, implica que quaisquer dois grafos tem a mesma probabilidade de ser escolhido, formando um espaço de probabilidade uniforme. A biblioteca do SageMath (THE SAGE DEVELOPERS, 2016) é capaz de gerar grafos aleatórios do modelo $\mathcal{G}_{n,p}$ como o citado acima, em especial é possível ajustar o parâmetro p para obter grafos de $\mathcal{G}_{n,1/2}$. Por isso, para geração das amostras de grafos aleatórios foi utilizada essa biblioteca.

Para o processo de amostragem ainda é preciso definir o número de vértices n que um grafo $\mathcal{G}_{n,1/2}$ deve ter. A escolha do número de vértices de um grafo aleatório pode ser feita de forma arbitrária, porém o número de vértices é um parâmetro que tem impacto no tempo de execução de um algoritmo para o CM. Para determinar o tamanho do conjunto de vértices V de um grafo aleatório $\mathcal{G}_{n,1/2}$, foram realizados testes preliminares: para $|V| = n \in \{100, 150, \ldots, 500\}$ foram gerados grafos aleatórios e submetidos ao algoritmo MCS_L. Foram escolhidos, para o menor tamanho n do conjunto de vértices, um valor entre $100, 150, \ldots, 500$ cujo tempo de execução em CPU fosse aproximadamente 1 segundo (que possui acurácia de aproximadamente 90% por ser maior que tempo gasto pelo processo escalonador de tarefas (BRYANT; DAVID RICHARD; DAVID RICHARD, 2003 apud ANJOS, 2015)), nesse caso foi observado que $n = 300$ atende esse critério. Com base nessas observações e tempo hábil para execução dos testes, foi definido o maior valor de $n = 1000$

[1] disponível em http://dimacs.Gerute.edu/Challenges

vértices. Ao final, a amostragem gerada e utilizada na Seção 4.3.2 tem um total de 150 grafos $\mathcal{G}_{n,1/2}$ com $n \in \{300, 350, \ldots, 1000\}$ vértices cada.

Também foram criados grafos aleatórios para a classe de grafos cordais e para a classe dos cografos. Para gerar um grafo cordal aleatório com n vértices foi utilizado o algoritmo proposto por Şeker et al. (2018) no qual é criado um grafo da intersecção de n subárvores aleatórias de uma árvore com n nós (algoritmo baseado no Teorema 3). A implementação desse algoritmo está disponível na biblioteca de geradores de grafos do *software* livre SageMath (Apêndice B) . Para criar um cografo aleatório com n vértices, foi implementada uma versão modificada de um algoritmo presente em Utkina (2016) que percorre a coárvore em pós-ordem adicionando aleatoriamente vértices em suas folhas. Uma observação aqui é que esses métodos de geração para grafos cordais aleatórios ou cografos aleatórios não tem distribuição de probabilidade conhecida.

4.3 RESULTADOS EXPERIMENTAIS

Os seguintes algoritmos para o CM foram implementados em linguagem de programação C++:

- MCLIQ (TOMITA; KOHATA; TAKAHASHI, 1988) e MCLIQ_L (Seção 3.4.2)
- MCQ (TOMITA; SEKI, 2003) e MCQ_L (Seção 3.4.3)
- DYN (KONC; JANEZIC, 2007) e DYN_L (Seção 3.4.4)
- MCR (TOMITA; KAMEDA, 2007) e MCR_L (Seção 3.4.5)
- MCS (TOMITA; SUTANI et al., 2010) e MCS_L (Seção 3.4.6)

As variáveis observadas são o número de estados $|\mathcal{T}(G)|$ e o tempo de execução em CPU dado em segundos, denotado por Γ.

Seja $\mathcal{A}$ um algoritmo dentre $\{\mathsf{MCLIQ}, \mathsf{MCQ}, \mathsf{DYN}, \mathsf{MCR}, \mathsf{MCS}\}$. Anteriormente, um algoritmo de $\mathcal{A}$ foi modificado com o Algoritmo $\mathsf{LexBFS}(G)$. Seja $\mathcal{A}_L$, um algoritmo dentre $\{\mathsf{MCLIQ}_L, \mathsf{MCQ}_L, \mathsf{DYN}_L, \mathsf{MCR}_L, \mathsf{MCS}_L\}$. Dado um grafo G, o número de estados gerados durante uma execução de $\mathcal{A}$ é denotado por $|\mathcal{T}_{\mathcal{A}}(G)|$. De maneira análoga, $|\mathcal{T}_{\mathcal{A}_L}(G)|$ denota o número de estados gerados durante uma execução de $\mathcal{A}_L$. A notação $\Gamma_{\mathcal{A}}$ indica o tempo de execução de um algoritmo $\mathcal{A}$ em segundos. A notação $\Gamma_{\mathcal{A}_L}$ indica o tempo de execução de um algoritmo $\mathcal{A}_L$ em segundos.

4.3.1 Resultados para Instâncias DIMACS

Inicialmente são apresentados os resultados da comparação entre o algoritmo de coloração de vértices $\mathsf{Greedy}(G, \pi)$ contra o mesmo algoritmo, porém, dessa vez utilizando como entrada uma ordenação π_L do conjunto de vértices produzida pelo Algoritmo

$\mathsf{LexBFS}(G)$. Na a Tabela 1, são utilizadas 80 instâncias de grafos DIMACS como entrada. O Algoritmo $\mathsf{Greedy}(G, \pi)$ devolve um vetor de cores c. Denotamos o número de cores computado por $\mathsf{Greedy}(G, \pi)$ por $c(G, \pi)$. De forma análoga, $c(G, \pi_L)$ denota o número de cores computado por $\mathsf{Greedy}(G, \pi_L)$. É possível observar que em 44 instâncias o número de cores computado foi menor quando utilizada π_L como entrada de Greedy. Por outro lado, em 18 instâncias o número de cores foi maior considerando π_L como entrada de Greedy. Também, em outras 18 instâncias, o número de cores se igualou utilizando π ou π_L como entrada para Greedy.

Além dos resultados do algoritmo de coloração de vértices Greedy, serão apresentados os resultados para os algoritmos para o CM. Foram submetidas 80 instâncias DIMACS aos algoritmos de $\mathcal{A}$ e $\mathcal{A}_L$. O limite de tempo de execução para cada instância foi de 3600 segundos. Durante a execução dos testes foi mantida uma contagem de quantos estados da árvore de estados, dentro da função $\mathsf{pre\text{-}process\text{-}state}(G, Q, K, C)$ de um algoritmo $\mathcal{A}_L$, o número de cores por $\mathsf{Greedy}(G, \pi_L)$ foi menor que o número de cores computado por $\mathsf{Greedy}(G, \pi)$. Esse número é denotado por τ_L nas tabelas em seguida.

A Tabela 2 mostra uma comparação em relação ao número de estados gerados durante a execução dos algoritmos MCLIQ e MCLIQ_L. Das 80 instâncias DIMACS, 50 tiveram conclusão dentro do tempo limite de 3600 segundos na execução de MCLIQ_L. Para MCLIQ, um total de 55 instâncias terminaram dentro de 3600 segundos. Das 50 instâncias, em 28 o número de estados $|\mathcal{T}_{\mathsf{MCLIQ}_L}(G)|$ foi menor que $|\mathcal{T}_{\mathsf{MCLIQ}}(G)|$. Em 8 instâncias $|\mathcal{T}_{\mathsf{MCLIQ}}(G)|$ e $|\mathcal{T}_{\mathsf{MCLIQ}_L}(G)|$ se igualaram onde $\tau_L = 0$. Por fim, em 14 instâncias $|\mathcal{T}_{\mathsf{MCLIQ}_L}(G)|$ foi maior que $|\mathcal{T}_{\mathsf{MCLIQ}}(G)|$. A coluna da razão $\frac{|\mathcal{T}_{\mathsf{MCLIQ}_L}(G)|}{|\mathcal{T}_{\mathsf{MCLIQ}}(G)|}$ mostra a proporção entre número de estados geradas por cada algoritmo. Em relação ao tempo de execução, em nenhuma instância DIMACS o tempo de execução $\Gamma_{\mathsf{MCLIQ}_L}$ foi menor que Γ_{MCLIQ}.

A Tabela 3 mostra uma comparação em relação ao número de estados geradas durante a execução dos algoritmos MCQ e MCQ_L. Das 80 instâncias DIMACS, 53 tiveram conclusão dentro do tempo limite de 3600 segundos na execução de MCQ_L. Para MCQ, um total de 58 instâncias terminaram dentro de 3600 segundos. Das 53 instâncias, em 28 o número de estados $|\mathcal{T}_{\mathsf{MCQ}_L}(G)|$ foi menor que $|\mathcal{T}_{\mathsf{MCQ}}(G)|$. Em 10 instâncias $|\mathcal{T}_{\mathsf{MCQ}}(G)|$ e $\mathcal{T}_{\mathsf{MCQ}_L}(G)$. Por fim, em 15 instâncias $|\mathcal{T}_{\mathsf{MCQ}_L}(G)|$ foi maior que $|\mathcal{T}_{\mathsf{MCQ}}(G)|$, na comparação. A coluna da razão $\frac{|\mathcal{T}_{\mathsf{MCQ}_L}(G)|}{|\mathcal{T}_{\mathsf{MCQ}}(G)|}$ mostra a proporção entre número de estados geradas por cada algoritmo. Em relação ao tempo de execução, em nenhuma instância DIMACS o tempo de execução Γ_{MCQ_L} foi menor que Γ_{MCQ}.

A Tabela 4 mostra uma comparação em relação ao número de estados geradas durante a execução dos algoritmos DYN e DYN_L. Das 80 instâncias DIMACS, 53 tiveram conclusão dentro do tempo limite de 3600 segundos na execução de DYN_L. Para DYN, um total de 58 instâncias terminaram dentro de 3600 segundos. Das 53 instâncias, em 8 o número de estados $|\mathcal{T}_{\mathsf{DYN}_L}(G)|$ foi menor que $|\mathcal{T}_{\mathsf{DYN}}(G)|$. Em 34 instâncias $|\mathcal{T}_{\mathsf{DYN}}(G)|$ e $|\mathcal{T}_{\mathsf{DYN}_L}(G)|$ se igualaram. Por fim, em 35 instâncias $|\mathcal{T}_{\mathsf{DYN}_L}(G)|$ foi maior que $|\mathcal{T}_{\mathsf{DYN}}(G)|$. A

coluna da razão $\frac{|\mathcal{T}_{\mathsf{DYN}_L}(G)|}{|\mathcal{T}_{\mathsf{DYN}}(G)|}$ mostra a proporção entre número de estados geradas por cada algoritmo. Em relação ao tempo de execução, apenas na istância `hamming-8-4` a variável Γ_{DYN_L} foi menor que Γ_{DYN}. Em todas outras o tempo de execução Γ_{DYN_L} foi maior que Γ_{DYN}.

A Tabela 5 mostra uma comparação em relação ao número de estados geradas durante a execução dos algoritmos MCR e MCR_L. Das 80 instâncias DIMACS, 51 tiveram conclusão dentro do tempo limite de 3600 segundos na execução de MCR_L. Para MCR, um total de 57 instâncias terminaram dentro de 3600 segundos. Das 51 instâncias, em 28 o número de estados $|\mathcal{T}_{\mathsf{MCR}_L}(G)|$ foi menor que $|\mathcal{T}_{\mathsf{MCR}}(G)|$. Em 5 instâncias $|\mathcal{T}_{\mathsf{MCR}}(G)|$ e $|\mathcal{T}_{\mathsf{MCR}_L}(G)|$ se igualaram. Por fim, em 16 instâncias $|\mathcal{T}_{\mathsf{MCR}_L}(G)|$ foi maior que $|\mathcal{T}_{\mathsf{MCR}}(G)|$. A coluna da razão $\frac{|\mathcal{T}_{\mathsf{MCR}_L}(G)|}{|\mathcal{T}_{\mathsf{MCR}}(G)|}$ mostra a proporção entre número de estados geradas por cada algoritmo. Em relação ao tempo de execução, em nenhuma instância DIMACS o tempo de execução Γ_{MCR_L} foi menor que Γ_{MCR}.

A Tabela 6 mostra uma comparação em relação ao número de estados geradas durante a execução dos algoritmos MCS e MCS_L. Das 80 instâncias DIMACS, 54 tiveram conclusão dentro do tempo limite de 3600 segundos na execução de MCS_L. Para MCS, um total de 61 instâncias terminaram dentro de 3600 segundos. Das 54 instâncias, em 9 o número de estados $|\mathcal{T}_{\mathsf{MCS}_L}(G)|$ foi menor que $|\mathcal{T}_{\mathsf{MCS}}(G)|$. Em 7 instâncias $|\mathcal{T}_{\mathsf{MCS}}(G)|$ e $|\mathcal{T}_{\mathsf{MCS}_L}(G)|$ se igualaram. Por fim, em 37 instâncias $|\mathcal{T}_{\mathsf{MCS}_L}(G)|$ foi maior que $|\mathcal{T}_{\mathsf{MCS}}(G)|$. A coluna da razão $\frac{|\mathcal{T}_{\mathsf{MCS}_L}(G)|}{|\mathcal{T}_{\mathsf{MCS}}(G)|}$ mostra a proporção entre número de estados geradas por cada algoritmo. Em relação ao tempo de execução, em nenhuma instância DIMACS o tempo de execução Γ_{MCS_L} foi menor que Γ_{MCS}.

TABELA 1 – Número de cores computada por Greedy por duas ordenações diferentes dos vértices de instâncias DIMACS como entrada: uma ordenação π dos vértices e uma ordenação π^L dos vértices produzida por LexBFS.

G	$c(G,\pi)$	$c(G,\pi_L)$
C125.9	57	57
C250.9	98	98
C500.9	184	**176**
C1000.9	319	319
C2000.5	226	**220**
C2000.9	592	**582**
C4000.5	**402**	405
DSJC500_5	72	74
DSJC1000_5	127	127
MANN_a9	18	19
MANN_a27	**135**	140
MANN_a45	372	373
MANN_a81	**1134**	1153
brock200_1	59	**55**
brock200_2	36	36
brock200_3	45	44
brock200_4	49	48
brock400_1	**102**	105
brock400_2	**100**	103
brock400_3	103	**100**
brock400_4	**100**	103
brock800_1	144	144
brock800_2	144	144
brock800_3	143	144
brock800_4	148	**142**
c-fat200-1	**12**	15
c-fat200-2	24	24
c-fat200-5	**68**	85
c-fat500-1	14	14
c-fat500-2	26	26
c-fat500-5	64	64
c-fat500-10	126	126
gen200_p0.9_44	76	76
gen200_p0.9_55	80	**77**
gen400_p0.9_55	127	**117**
gen400_p0.9_65	136	**120**
gen400_p0.9_75	143	**121**
hamming6-2	32	32
hamming6-4	8	10
hamming8-2	128	128
hamming8-4	32	**27**
hamming10-2	512	512
hamming10-4	**128**	140

G	$c(G,\pi)$	$c(G,\pi_L)$
johnson8-2-4	**6**	9
johnson8-4-4	20	**14**
johnson16-2-4	**14**	21
johnson32-2-4	**30**	45
keller4	37	**34**
keller5	175	**143**
keller6	781	**261**
p_hat300-1	29	28
p_hat300-2	56	**52**
p_hat300-3	85	**81**
p_hat500-1	45	**41**
p_hat500-2	87	**80**
p_hat500-3	131	**123**
p_hat700-1	53	53
p_hat700-2	114	**106**
p_hat700-3	171	**163**
p_hat1000-1	69	**64**
p_hat1000-2	148	**137**
p_hat1000-3	230	**216**
p_hat1500-1	95	93
p_hat1500-2	213	**194**
p_hat1500-3	326	**307**
san200_0.7_1	49	**30**
san200_0.7_2	35	**18**
san200_0.9_1	92	**76**
san200_0.9_2	86	**70**
san200_0.9_3	73	**44**
san400_0.5_1	29	**13**
san400_0.7_1	81	**67**
san400_0.7_2	67	**30**
san400_0.7_3	59	**22**
san400_0.9_1	163	**149**
san1000	47	**15**
sanr200_0.7	52	53
sanr200_0.9	82	80
sanr400_0.5	62	62
sanr400_0.7	91	90

TABELA 2 – Comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}$ (%)	Γ_{MCLIQ}	$\Gamma_{\mathsf{MCLIQ}_L}$	$\frac{\Gamma_{\mathsf{MCLIQ}_L}}{\Gamma_{\mathsf{MCLIQ}}}$ (%)
C125.9	186 715	**178 155**	95,42	36 687	20,59	0,94	8,29	883,29
C250.9	≥ 966 406 138	≥ 125 166 261	–	27 867 930	–	–	–	–
C500.9	≥1 194 200 353	≥ 112 902 455	–	25 144 422	–	–	–	–
C1000.9	≥1 290 452 992	≥ 129 641 716	–	29 415 113	–	–	–	–
C2000.5	≥1 684 611 754	≥ 419 867 611	–	72 940 529	–	–	–	–
C2000.9	≥1 076 648 260	≥ 105 874 901	–	23 744 334	–	–	–	–
C4000.5	≥1 617 494 708	≥ 382 457 403	–	66 430 314	–	–	–	–
DSJC500_5	2 697 735	**2 358 173**	87,41	403 608	17,12	9,93	25,68	258,64
DSJC1000_5	180 139 971	**165 164 317**	91,69	28 679 276	17,36	750,26	2673,61	356,36
MANN_a9	143	**89**	62,24	7	7,87	0,00	0,00	496,07
MANN_a27	**76 041**	101 877	133,98	5246	5,15	2,62	125,09	4773,08
MANN_a45	5 703 151	≥ 657 145	–	29 368	–	1260,17	–	–
MANN_a81	≥ 3 834 933	≥ 45 875	–	3836	–	–	–	–
brock200_1	1 736 439	**989 867**	57,01	192 883	19,49	5,77	15,62	270,75
brock200_2	**8669**	16 259	187,55	2604	16,02	0,03	0,16	626,65
brock200_3	**35 639**	51 399	144,22	8849	17,22	0,11	0,81	712,19
brock200_4	161 667	**98 159**	60,72	18 206	18,55	0,57	1,41	247,60
brock400_1	684 947 921	≥ 282 207 146	–	56 678 533	–	3025,80	–	–

continua

TABELA 2 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	Γ_{MCLIQ}	$\Gamma_{\mathsf{MCLIQ}_L}$	$\frac{\Gamma_{\mathsf{MCLIQ}_L}}{\Gamma_{\mathsf{MCLIQ}}}$ (%)
brock400_2	449 678 155	≥ 150 325 462	–	30 465 668	–	1893,81	–	–
brock400_3	388 806 129	≥ 214 420 445	–	43 020 875	–	1330,24	–	–
brock400_4	164 112 187	**112 710 093**	68,68	22 770 986	20,20	973,31	3226,89	331,54
brock800_1	≥1 824 637 201	≥ 275 472 040	–	51 385 314	–	–	–	–
brock800_2	≥1 725 389 495	≥ 298 702 006	–	56 036 351	–	–	–	–
brock800_3	≥1 903 638 047	≥ 253 746 788	–	47 788 232	–	–	–	–
brock800_4	≥1 257 858 110	≥ 311 582 267	–	58 142 016	–	–	–	–
c-fat200-1	51	51	100,00	0	0,00	0,00	0,00	231,54
c-fat200-2	49	49	100,00	0	0,00	0,00	0,00	365,28
c-fat200-5	281	281	100,00	0	0,00	0,00	0,00	289,56
c-fat500-1	253	253	100,00	0	0,00	0,00	0,02	746,37
c-fat500-2	53	53	100,00	0	0,00	0,00	0,00	816,20
c-fat500-5	129	129	100,00	0	0,00	0,00	0,00	550,80
c-fat500-10	29	29	100,00	0	0,00	0,00	0,00	349,19
gen200_p0.9_44	**7 877 427**	12 436 061	157,87	2 812 300	22,61	36,85	557,25	1512,11
gen200_p0.9_55	**900 421**	2 253 643	250,29	507 843	22,53	6,21	108,77	1750,90
gen400_p0.9_55	≥1 356 171 828	≥ 93 168 554	–	21 278 879	22,84	–	–	–
gen400_p0.9_65	≥1 398 184 546	≥ 101 610 259	–	23 057 114	22,69	–	–	–

continua

TABELA 2 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	Γ_{MCLIQ}	$\Gamma_{\mathsf{MCLIQ}_L}$	$\frac{\Gamma_{\mathsf{MCLIQ}_L}}{\Gamma_{\mathsf{MCLIQ}}}$ (%)
gen400_p0.9_75	≥1 051 310 956	≥ 106 269 206	–	27 644 509	26,01	–	–	–
hamming6-2	65	65	100,00	0	0,00	0,00	0,00	624,06
hamming6-4	**165**	203	123,03	4	1,97	0,00	0,00	129,79
hamming8-2	**257**	1087	422,96	10	0,92	0,00	0,12	5583,82
hamming8-4	82 985	**8953**	10,79	1899	21,21	0,39	0,38	96,93
hamming10-2	**1025**	≥ 1 073 598	–	254 119	–	0,11	–	–
hamming10-4	≥1 273 235 882	≥ 136 061 097	–	32 260 388	–	–	–	–
johnson8-2-4	**73**	97	132,88	2	2,06	0,00	0,00	240,43
johnson8-4-4	289	**77**	26,64	8	10,39	0,00	0,00	74,46
johnson16-2-4	**646 073**	2 068 961	320,24	36 108	1,75	0,79	5,99	760,69
johnson32-2-4	≥5 935 957 663	≥2 933 419 774	–	59 127 926	–	–	–	–
keller4	**26 233**	29 685	113,16	3402	11,46	0,08	0,39	481,48
keller5	≥ 776 286 449	≥ 94 785 295	–	16 429 835	–	–	–	–
keller6	≥ 582 146 170	≥ 46 862 192	–	9 323 841	–	–	–	–
p_hat300-1	3461	**2885**	83,36	495	17,16	0,01	0,03	248,50
p_hat300-2	**27 639**	35 723	129,25	9631	26,96	0,16	1,49	934,01
p_hat300-3	7 658 035	**6 639 479**	86,70	1 751 912	26,39	53,04	441,91	833,14
p_hat500-1	25 821	**23 735**	91,92	4320	18,20	0,10	0,28	288,88

continua

TABELA 2 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	Γ_{MCLIQ}	$\Gamma_{\mathsf{MCLIQ}_L}$	$\frac{\Gamma_{\mathsf{MCLIQ}_L}}{\Gamma_{\mathsf{MCLIQ}}}$ (%)
p_hat500-2	2 044 409	**1 219 651**	59,66	380 175	31,17	15,53	76,84	494,77
p_hat500-3	≥1 124 671 614	≥ 82 083 562	–	24 102 761	–	–	–	–
p_hat700-1	73 861	**68 913**	93,30	12 820	18,60	0,29	0,78	265,13
p_hat700-2	37 936 343	**16 780 375**	44,23	5 504 648	32,80	338,92	1492,35	440,33
p_hat700-3	≥1 033 245 141	≥ 94 511 407	–	27 900 773	–	–	–	–
p_hat1000-1	474 881	**437 195**	92,06	80 793	18,48	1,86	3,91	210,17
p_hat1000-2	≥ 847 965 835	≥ 68 605 728	–	21 402 144	–	–	–	–
p_hat1000-3	≥ 864 075 048	≥ 61 271 748	–	17 367 790	–	–	–	–
p_hat1500-1	3 285 973	**2 589 321**	78,80	505 031	19,50	12,13	52,74	434,64
p_hat1500-2	≥ 723 334 403	≥ 69 287 186	–	21 952 948	–	–	–	–
p_hat1500-3	≥ 679 901 628	≥ 64 802 527	–	19 376 335	–	–	–	–
san200_0.7_1	24 723	**3753**	15,18	849	22,62	0,10	0,11	109,43
san200_0.7_2	1551	**705**	45,45	109	15,46	0,01	0,01	172,03
san200_0.9_1	1965	**787**	40,05	27	3,43	0,03	0,02	89,41
san200_0.9_2	2 299 165	**5571**	0,24	1045	18,76	16,10	0,84	5,23
san200_0.9_3	16 520 715	**2 761 451**	16,72	598 063	21,66	111,99	170,58	152,32
san400_0.5_1	7929	**4665**	58,83	194	4,16	0,05	0,39	841,54
san400_0.7_1	**110 039**	263 603	239,55	70 653	26,80	0,97	22,27	2292,35
san400_0.7_2	1 212 335	**963 455**	79,47	171 858	17,84	9,95	47,23	474,58

continua

TABELA 2 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCLIQ e pelo Algoritmo MCLIQ$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCLIQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCLIQ}}(G)\rvert}$ (%)	Γ_{MCLIQ}	$\Gamma_{\mathsf{MCLIQ}_L}$	$\frac{\Gamma_{\mathsf{MCLIQ}_L}}{\Gamma_{\mathsf{MCLIQ}}}$ (%)
san400_0.7_3	**1 165 313**	2 490 389	213,71	311 633	12,51	7,67	110,37	1438,54
san400_0.9_1	≥1 255 019 324	≥ 63 296 788	–	20 151 062	–	–	–	–
san1000	605 801	**299 009**	49,36	13 037	4,36	13,41	165,51	1234,56
sanr200_0.7	412 535	**346 223**	83,93	64 685	18,68	1,63	5,02	308,49
sanr200_0.9	**88 944 573**	135 128 061	151,92	30 045 738	22,24	635,84	4961,85	780,37
sanr400_0.5	760 309	**581 667**	76,50	98 631	16,96	1,80	5,75	319,28
sanr400_0.7	202 427 063	**170 112 883**	84,04	32 848 211	19,31	658,91	2754,05	417,97

TABELA 3 – Comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}$ (%)	Γ_{MCQ}	Γ_{MCQ_L}	$\frac{\Gamma_{\mathsf{MCQ}_L}}{\Gamma_{\mathsf{MCQ}}}$ (%)
C125.9	**97 723**	135 785	138,95	27 375	20,16	0,50	5,96	1192,01
C250.9	≥1 460 981 609	≥ 141 357 757	–	30 934 933	–	–	–	–
C500.9	≥1 323 553 221	≥ 144 289 576	–	31 452 288	–	–	–	–
C1000.9	≥1 084 406 355	≥ 108 151 290	–	24 463 250	–	–	–	–
C2000.5	≥1 706 108 574	≥ 314 552 586	–	54 582 530	–	–	–	–
C2000.9	≥1 418 012 564	≥ 108 785 800	–	24 721 245	–	–	–	–
C4000.5	≥1 404 528 963	≥ 352 764 499	–	61 265 937	–	–	–	–
DSJC500_5	2 621 585	**2 338 145**	89,19	397 263	16,99	9,21	33,11	359,61
DSJC1000_5	181 423 599	**162 266 449**	89,44	28 134 171	17,34	803,97	3140,12	390,58
MANN_a9	191	**161**	84,29	6	3,73	0,00	0,00	413,10
MANN_a27	76 509	**76 485**	99,97	3910	5,11	2,30	94,92	4133,22
MANN_a45	5 704 471	≥ 589 117	–	21 973	–	768,85	–	–
MANN_a81	≥ 3 313 784	≥ 47 902	–	2981	–	–	–	–
brock200_1	900 785	**823 079**	91,37	160 410	19,49	3,34	12,44	372,36
brock200_2	8639	**8255**	95,56	1308	15,84	0,03	0,10	367,59
brock200_3	33 047	**32 789**	99,22	5902	18,00	0,13	0,57	454,06
brock200_4	126 155	**121 767**	96,52	22 044	18,10	0,40	1,43	357,05
brock400_1	643 210 693	≥ 245 150 086	–	49 052 451	–	2284,01	–	–
brock400_2	232 447 887	**200 021 533**	86,05	40 349 462	20,17	1222,73	5516,18	451,14

continua

TABELA 3 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	Γ_{MCQ}	Γ_{MCQ_L}	$\frac{\Gamma_{\mathsf{MCQ}_L}}{\Gamma_{\mathsf{MCQ}}}$ (%)
brock400_3	596 387 657	≥ 279 632 606	–	55 646 268	–	2325,00	–	–
brock400_4	237 710 751	**200 301 914**	84,26	40 161 603	20,05	1205,51	7198,22	597,11
brock800_1	≥1 708 491 715	≥ 320 539 587	–	60 277 806	–	–	–	–
brock800_2	≥1 842 871 118	≥ 309 643 045	–	58 081 889	–	–	–	–
brock800_3	≥1 589 764 594	≥ 295 685 245	–	55 414 423	–	–	–	–
brock800_4	≥1 401 049 408	≥ 309 101 350	–	58 128 603	–	–	–	–
c-fat200-1	437	437	100,00	0	0,00	0,00	0,00	109,49
c-fat200-2	487	487	100,00	0	0,00	0,00	0,00	239,83
c-fat200-5	621	621	100,00	0	0,00	0,00	0,01	393,29
c-fat500-1	1045	1045	100,00	0	0,00	0,04	0,09	247,48
c-fat500-2	1093	1093	100,00	0	0,00	0,00	0,01	315,33
c-fat500-5	1245	1245	100,00	0	0,00	0,01	0,02	400,00
c-fat500-10	1493	1493	100,00	0	0,00	0,00	0,01	161,22
gen200_p0.9_44	**1 054 699**	1 304 711	123,70	299 752	22,97	5,69	130,69	2296,89
gen200_p0.9_55	1 924 375	**1 615 993**	83,97	355 851	22,02	8,30	74,09	892,89
gen400_p0.9_55	≥1 308 197 521	≥ 126 009 814	–	28 821 204	–	–	–	–
gen400_p0.9_65	≥1 330 244 664	≥ 104 138 528	–	25 643 148	–	–	–	–
gen400_p0.9_75	≥1 085 732 572	≥ 123 469 953	–	28 999 011	–	–	–	–

continua

TABELA 3 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	Γ_{MCQ}	Γ_{MCQ_L}	$\frac{\Gamma_{\mathsf{MCQ}_L}}{\Gamma_{\mathsf{MCQ}}}$ (%)
hamming6-2	127	127	100,00	0	0,00	0,00	0,00	549,10
hamming6-4	**221**	349	157,92	0	0,00	0,00	0,00	218,36
hamming8-2	511	511	100,00	0	0,00	0,01	0,16	1566,15
hamming8-4	83 209	**20 573**	24,72	3679	17,88	0,40	0,65	163,69
hamming10-2	2047	2047	100,00	0	0,00	0,21	11,98	5596,42
hamming10-4	≥1 332 204 660	≥ 116 277 182	–	28 101 032	–	–	–	–
johnson8-2-4	**95**	161	169,47	0	0,00	0,00	0,00	231,71
johnson8-4-4	513	**303**	59,06	44	14,52	0,00	0,00	197,39
johnson16-2-4	**860 263**	2 420 461	281,36	49 722	2,05	1,08	6,87	637,37
johnson32-2-4	≥8 127 769 333	≥2 031 486 709	–	40 936 654	–	–	–	–
keller4	**25 503**	30 237	118,56	3428	11,34	0,08	0,40	509,84
keller5	≥ 941 429 762	≥ 104 951 183	–	18 564 567	–	–	–	–
keller6	≥ 516 358 055	≥ 48 823 564	–	10 704 429	–	–	–	–
p_hat300-1	4431	**4053**	91,47	562	13,87	0,01	0,03	223,06
p_hat300-2	**20 069**	21 093	105,10	4638	21,99	0,10	0,76	721,95
p_hat300-3	**4 945 471**	4 975 437	100,61	1 214 051	24,40	33,11	305,16	921,58
p_hat500-1	22 569	**21 793**	96,56	3526	16,18	0,08	0,23	294,53
p_hat500-2	**1 087 049**	1 255 825	115,53	328 544	26,16	7,86	80,96	1030,39

continua

TABELA 3 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	Γ_{MCQ}	Γ_{MCQ_L}	$\frac{\Gamma_{\mathsf{MCQ}_L}}{\Gamma_{\mathsf{MCQ}}}$ (%)
p_hat500-3	$\geq$ 474 153 771	$\geq$ 63 424 496	–	18 557 479	–	3881,08	–	–
p_hat700-1	**68 843**	69 845	101,46	11 417	16,35	0,28	0,89	315,23
p_hat700-2	**8 838 757**	9 855 323	111,50	2 801 576	28,43	79,59	661,28	830,90
p_hat700-3	$\geq$ 795 298 096	$\geq$ 71 121 599	–	22 222 213	–	–	–	–
p_hat1000-1	405 361	**394 315**	97,28	66 365	16,83	1,36	4,82	353,62
p_hat1000-2	443 593 479	$\geq$ 58 144 296	–	17 843 894	30,69	3521,41	–	–
p_hat1000-3	$\geq$ 876 629 541	$\geq$ 58 983 228	–	16 934 877	–	–	–	–
p_hat1500-1	2 566 891	**2 503 719**	97,54	437 251	17,46	11,48	42,12	366,92
p_hat1500-2	$\geq$ 586 921 209	$\geq$ 52 015 322	–	16 542 569	–	–	–	–
p_hat1500-3	$\geq$ 822 628 224	$\geq$ 42 485 168	–	12 465 388	–	–	–	–
san200_0.7_1	**2487**	2589	104,10	611	23,60	0,02	0,14	745,52
san200_0.7_2	3161	**3105**	98,23	377	12,14	0,01	0,09	621,26
san200_0.9_1	453 769	**112 161**	24,72	25 620	22,84	2,93	6,57	224,07
san200_0.9_2	715 291	**565 611**	79,07	130 011	22,99	4,35	47,24	1084,91
san200_0.9_3	2 364 515	**1 444 191**	61,08	319 006	22,09	17,96	148,99	829,35
san400_0.5_1	**6819**	7495	109,91	291	3,88	0,04	0,59	1532,70
san400_0.7_1	**195 463**	206 207	105,50	52 870	25,64	1,67	13,82	829,35
san400_0.7_2	132 947	**59 587**	44,82	12 147	20,39	1,54	9,93	643,87
san400_0.7_3	1 516 645	**900 167**	59,35	112 170	12,46	9,62	57,27	595,25

continua

TABELA 3 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCQ e pelo Algoritmo MCQ_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCQ}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCQ}}(G)\rvert}$ (%)	Γ_{MCQ}	Γ_{MCQ_L}	$\frac{\Gamma_{\mathsf{MCQ}_L}}{\Gamma_{\mathsf{MCQ}}}$ (%)
san400_0.9_1	**1 416 747**	9 287 959	655,58	2 825 345	30,42	49,23	1929,18	3918,76
san1000	502 281	**322 965**	64,30	14 240	4,41	8,54	79,62	931,91
sanr200_0.7	367 057	**298 311**	81,27	55 444	18,59	1,42	5,74	404,92
sanr200_0.9	85 730 303	**61 517 011**	71,76	13 718 068	22,30	584,43	2405,43	411,59
sanr400_0.5	600 347	**544 745**	90,74	92 073	16,90	2,13	5,75	270,57
sanr400_0.7	180 182 651	**152 983 087**	84,90	29 494 435	19,28	781,41	2353,74	301,22

TABELA 4 – Comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}$ (%)	Γ_{DYN}	Γ_{DYN_L}	$\frac{\Gamma_{\mathsf{DYN}_L}}{\Gamma_{\mathsf{DYN}}}$ (%)
C125.9	**69 663**	158 417	227,40	5017	3,17	0,41	3,27	789,95
C250.9	≥1 130 560 210	≥ 196 822 375	–	8 776 936	–	–	–	–
C500.9	≥1 065 067 344	≥ 159 200 611	–	9 010 174	–	–	–	–
C1000.9	≥1 001 275 836	≥ 234 648 212	–	14 123 458	–	–	–	–
C2000.5	≥1 569 105 528	≥ 524 420 672	–	11 690 285	–	–	–	–
C2000.9	≥1 105 068 531	≥ 183 283 128	–	12 461 751	–	–	–	–
C4000.5	≥1 383 598 467	≥ 357 430 301	–	7 164 090	–	–	–	–
DSJC500_5	**2 418 453**	2 482 261	102,64	49 921	2,01	8,32	15,68	188,39
DSJC1000_5	**165 035 703**	170 382 587	103,24	3 711 180	2,18	557,67	2177,46	390,46
MANN_a9	**191**	293	153,40	9	3,07	0,00	0,00	232,68
MANN_a27	**76 509**	2 593 343	3389,59	64 828	2,50	3,44	957,70	27 831,41
MANN_a45	5 704 471	≥ 1 435 646	–	36 225	–	1343,14	–	–
MANN_a81	≥ 3 858 268	≥ 101 026	–	799	–	–	–	–
brock200_1	**735 279**	916 885	124,70	22 789	2,49	3,61	7,46	206,86
brock200_2	**8233**	8791	106,78	105	1,19	0,03	0,05	170,86
brock200_3	**31 191**	35 483	113,76	132	0,37	0,12	0,24	193,65
brock200_4	**110 917**	127 431	114,89	1500	1,18	0,46	1,02	221,89
brock400_1	465 377 749	≥ 400 206 699	–	12 828 175	–	2180,99	–	–
brock400_2	**183 302 549**	219 089 005	119,52	6 359 491	2,90	1065,31	5760,55	540,74

continua

TABELA 4 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	Γ_{DYN}	Γ_{DYN_L}	$\frac{\Gamma_{\mathsf{DYN}_L}}{\Gamma_{\mathsf{DYN}}}$ (%)
brock400_3	426 790 829	≥ 393 685 596	–	13 151 281	–	2187,64	–	–
brock400_4	**177 854 633**	233 757 047	131,43	6 894 859	2,95	712,75	5217,01	731,95
brock800_1	≥1 633 252 410	≥ 489 587 104	–	13 699 559	–	–	–	–
brock800_2	≥1 619 916 099	≥ 439 109 992	–	13 233 167	–	–	–	–
brock800_3	≥1 564 342 850	≥ 452 519 571	–	13 363 565	–	–	–	–
brock800_4	≥1 358 567 946	≥ 445 032 251	–	12 367 652	–	–	–	–
c-fat200-1	437	437	100,00	0	0,00	0,00	0,01	518,88
c-fat200-2	487	487	100,00	0	0,00	0,00	0,00	184,85
c-fat200-5	621	621	100,00	0	0,00	0,00	0,06	1487,83
c-fat500-1	1045	1045	100,00	0	0,00	0,02	0,17	718,50
c-fat500-2	1093	1093	100,00	0	0,00	0,00	0,01	333,35
c-fat500-5	1245	1245	100,00	0	0,00	0,01	0,06	986,67
c-fat500-10	1493	1493	100,00	0	0,00	0,00	0,01	172,15
gen200_p0.9_44	**643 579**	1 265 181	196,59	47 076	3,72	3,88	42,02	1084,43
gen200_p0.9_55	**644 633**	1 787 813	277,34	56 489	3,16	3,52	60,21	1710,93
gen400_p0.9_55	≥1 336 133 469	≥ 205 971 890	–	12 455 930	–	–	–	–
gen400_p0.9_65	≥1 232 591 624	≥ 180 172 943	–	7 567 064	–	–	–	–
gen400_p0.9_75	≥ 955 358 871	≥ 221 794 689	–	11 060 253	–	–	–	–

continua

TABELA 4 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}(\mathrm{G})$ gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	Γ_{DYN}	Γ_{DYN_L}	$\frac{\Gamma_{\mathsf{DYN}_L}}{\Gamma_{\mathsf{DYN}}}$ (%)
hamming6-2	127	127	100,00	0	0,00	0,00	0,00	463,22
hamming6-4	**221**	343	155,20	0	0,00	0,00	0,00	191,62
hamming8-2	511	511	100,00	0	0,00	0,01	0,16	1487,79
hamming8-4	70 201	**18 107**	25,79	286	1,58	0,37	0,25	68,32
hamming10-2	2047	2047	100,00	0	0,00	0,38	13,02	3427,74
hamming10-4	$\geq$ 904 334 502	$\geq$ 195 567 668	–	10 567 742	–	–	–	–
johnson8-2-4	**95**	155	163,16	0	0,00	0,00	0,00	190,91
johnson8-4-4	489	**213**	43,56	21	9,86	0,00	0,00	176,50
johnson16-2-4	**637 507**	2 556 649	401,04	38 426	1,50	0,93	5,14	554,35
johnson32-2-4	$\geq$8 278 594 227	$\geq$2 674 724 294	–	64 074 232	–	–	–	–
keller4	**28 213**	30 563	108,33	331	1,08	0,09	0,17	180,26
keller5	$\geq$1 123 584 039	$\geq$ 196 255 269	–	2 679 752	–	–	–	–
keller6	$\geq$ 629 845 493	$\geq$ 45 139 509	–	661 612	–	–	–	–
p_hat300-1	4325	**4223**	97,64	46	1,09	0,01	0,02	126,17
p_hat300-2	**15 357**	20 723	134,94	92	0,44	0,09	0,21	226,17
p_hat300-3	**1 990 675**	4 343 337	218,18	111 289	2,56	15,16	143,56	946,85
p_hat500-1	**21 689**	23 093	106,47	72	0,31	0,08	0,13	166,57
p_hat500-2	**493 671**	1 356 975	274,87	23 398	1,72	4,07	47,35	1162,24

continua

TABELA 4 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	Γ_{DYN}	Γ_{DYN_L}	$\frac{\Gamma_{\mathsf{DYN}_L}}{\Gamma_{\mathsf{DYN}}}$ (%)
p_hat500-3	**71 555 155**	≥ 101 785 416	–	1 313 032	–	684,54	–	–
p_hat700-1	**64 691**	74 985	115,91	145	0,19	0,18	0,43	240,50
p_hat700-2	**2 629 241**	9 084 433	345,52	144 525	1,59	29,50	559,26	1896,04
p_hat700-3	≥ 582 850 521	≥ 108 969 144	–	1 205 190	–	–	–	–
p_hat1000-1	**381 411**	429 601	112,63	2137	0,50	1,51	2,48	164,08
p_hat1000-2	92 685 275	≥ 82 865 793	–	768 526	0,93	840,25	–	–
p_hat1000-3	≥ 589 581 949	≥ 95 406 331	–	1 502 737	–	–	–	–
p_hat1500-1	**2 414 293**	2 723 023	112,79	15 736	0,58	12,27	22,30	181,72
p_hat1500-2	≥ 448 596 170	≥ 78 393 176	–	841 329	–	–	–	–
p_hat1500-3	≥ 435 067 176	≥ 81 086 939	–	1 165 970	–	–	–	–
san200_0.7_1	3821	**2315**	60,59	132	5,70	0,02	0,07	368,00
san200_0.7_2	**2851**	3303	115,85	22	0,67	0,01	0,04	307,13
san200_0.9_1	258 913	**186 331**	71,97	10 916	5,86	1,14	4,54	397,55
san200_0.9_2	**278 843**	441 525	158,34	18 164	4,11	2,02	15,73	778,71
san200_0.9_3	**1 868 271**	2 424 073	129,75	130 388	5,38	16,77	105,07	626,50
san400_0.5_1	**9283**	12 283	132,32	44	0,36	0,06	0,22	400,45
san400_0.7_1	373 171	**314 155**	84,19	6048	1,93	2,93	14,68	500,79
san400_0.7_2	220 021	**69 803**	31,73	1096	1,57	2,46	2,85	116,15
san400_0.7_3	1 821 129	**1 044 581**	57,36	17 788	1,70	10,77	36,88	342,33

continua

TABELA 4 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo DYN e pelo Algoritmo DYN_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{DYN}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{DYN}}(G)\rvert}$ (%)	Γ_{DYN}	Γ_{DYN_L}	$\frac{\Gamma_{\mathsf{DYN}_L}}{\Gamma_{\mathsf{DYN}}}$ (%)
san400_0.9_1	**1 504 365**	2 591 391	172,26	123 973	4,78	54,51	519,67	953,41
san1000	**632 841**	660 215	104,33	2515	0,38	7,39	136,96	1853,69
sanr200_0.7	**318 635**	320 565	100,61	5734	1,79	1,32	3,07	232,78
sanr200_0.9	**38 468 211**	63 005 859	163,79	2 325 109	3,69	279,77	1946,23	695,65
sanr400_0.5	**556 523**	579 241	104,08	8687	1,50	2,36	4,85	205,21
sanr400_0.7	**139 547 519**	170 121 153	121,91	5 108 055	3,00	702,82	1963,13	279,32

TABELA 5 – Comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCR e pelo Algoritmo MCR$_L$ com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}$ (%)	Γ_{MCR}	Γ_{MCR_L}	$\frac{\Gamma_{\mathsf{MCR}_L}}{\Gamma_{\mathsf{MCR}}}$ (%)
C125.9	**83 461**	127 285	152,51	26 810	21,06	0,44	5,72	1294,71
C250.9	≥1 382 649 815	≥ 153 518 188	–	34 562 154	–	–	–	–
C500.9	≥1 263 238 106	≥ 119 985 734	–	26 821 924	–	–	–	–
C1000.9	≥1 258 624 480	≥ 120 420 634	–	26 955 001	–	–	–	–
C2000.5	≥1 481 450 497	≥ 386 904 172	–	67 190 111	–	–	–	–
C2000.9	≥1 452 434 681	≥ 136 601 460	–	31 023 654	–	–	–	–
C4000.5	≥1 061 902 317	≥ 286 907 145	–	49 815 991	–	–	–	–
DSJC500_5	2 628 355	**2 357 755**	89,70	400 646	16,99	10,37	34,63	334,06
DSJC1000_5	181 529 705	**162 842 409**	89,71	28 232 727	17,34	831,54	2548,46	306,47
MANN_a9	155	**109**	70,32	6	5,50	0,00	0,00	155,43
MANN_a27	97 191	**67 899**	69,86	1038	1,53	2,59	85,99	3325,96
MANN_a45	5 925 885	≥ 659 084	–	747	–	1477,70	–	–
MANN_a81	≥ 2 491 309	≥ 39 798	–	175	–	–	–	–
brock200_1	914 613	**793 751**	86,79	155 966	19,65	3,00	16,66	555,21
brock200_2	7961	**7831**	98,37	1246	15,91	0,05	0,11	246,63
brock200_3	32 555	**31 681**	97,32	5588	17,64	0,12	0,40	336,99
brock200_4	144 253	**130 811**	90,68	23 793	18,19	0,48	1,51	315,88
brock400_1	656 354 231	≥ 232 415 384	–	46 483 008	–	3461,71	–	–
brock400_2	241 984 915	**194 095 846**	80,21	39 058 139	20,12	1022,11	6192,19	605,82

continua

TABELA 5 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCR e pelo Algoritmo MCR_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	Γ_{MCR}	Γ_{MCR_L}	$\frac{\Gamma_{\mathsf{MCR}_L}}{\Gamma_{\mathsf{MCR}}}$ (%)
brock400_3	498 571 359	≥ 272 855 020	–	54 329 741	–	2217,79	–	–
brock400_4	218 638 195	**185 502 237**	84,84	37 416 554	20,17	756,31	7170,62	948,11
brock800_1	≥1 586 740 005	≥ 266 263 625	–	50 190 526	–	–	–	–
brock800_2	≥1 914 659 497	≥ 260 509 013	–	48 910 251	–	–	–	–
brock800_3	≥1 577 478 783	≥ 312 728 238	–	58 655 042	–	–	–	–
brock800_4	≥1 297 229 208	≥ 336 535 981	–	63 217 578	–	–	–	–
c-fat200-1	377	377	100,00	0	0,00	0,00	0,00	167,01
c-fat200-2	353	353	100,00	0	0,00	0,00	0,01	176,97
c-fat200-5	285	285	100,00	0	0,00	0,01	0,02	254,05
c-fat500-1	973	973	100,00	0	0,00	0,11	0,26	241,05
c-fat500-2	949	949	100,00	0	0,00	0,02	0,02	111,44
c-fat500-5	873	873	100,00	0	0,00	0,05	0,11	237,46
c-fat500-10	749	749	100,00	0	0,00	0,01	0,01	108,02
gen200_p0.9_44	1 252 035	**1 240 763**	99,10	290 780	23,44	7,10	109,89	1548,70
gen200_p0.9_55	4 930 799	**2 598 407**	52,70	598 387	23,03	30,68	179,66	585,65
gen400_p0.9_55	≥1 272 223 010	≥ 117 266 598	–	26 723 958	–	–	–	–
gen400_p0.9_65	≥1 505 311 345	≥ 118 257 446	–	28 536 961	–	–	–	–
gen400_p0.9_75	≥1 555 074 901	≥ 101 435 306	–	25 807 596	–	–	–	–

continua

TABELA 5 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCR e pelo Algoritmo MCR_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	Γ_{MCR}	Γ_{MCR_L}	$\frac{\Gamma_{\mathsf{MCR}_L}}{\Gamma_{\mathsf{MCR}}}$ (%)
hamming6-2	**65**	127	195,38	2	1,57	0,00	0,00	474,07
hamming6-4	**159**	223	140,25	1	0,45	0,00	0,00	118,59
hamming8-2	**257**	20 337	7913,23	4077	20,05	0,01	6,86	16 210,13
hamming8-4	26 871	**21 211**	78,94	3967	18,70	0,10	0,78	752,63
hamming10-2	**1025**	$\geq$ 4 496 658	–	691 517	–	0,09	–	–
hamming10-4	$\geq$1 524 270 532	$\geq$ 146 974 596	–	33 110 431	–	–	–	–
johnson8-2-4	**49**	57	116,33	0	0,00	0,00	0,00	159,15
johnson8-4-4	**233**	239	102,58	35	14,64	0,00	0,00	370,48
johnson16-2-4	**533 629**	1 939 721	363,50	36 778	1,90	0,51	3,73	733,01
johnson32-2-4	$\geq$8 724 803 139	$\geq$2 826 042 072	–	57 152 138	–	–	–	–
keller4	**25 779**	27 059	104,97	3448	12,74	0,07	0,34	474,39
keller5	$\geq$1 153 064 539	$\geq$ 113 014 565	–	20 364 387	–	–	–	–
keller6	$\geq$ 508 505 808	$\geq$ 33 592 327	–	6 870 117	–	–	–	–
p_hat300-1	4041	**3779**	93,52	521	13,79	0,04	0,06	133,88
p_hat300-2	**9735**	13 247	136,08	3106	23,45	0,14	0,62	440,08
p_hat300-3	**3 879 157**	5 814 145	149,88	1 440 282	24,77	27,59	328,43	1190,57
p_hat500-1	**21 599**	21 901	101,40	3515	16,05	0,24	0,37	158,73
p_hat500-2	**677 945**	890 697	131,38	235 301	26,42	5,78	59,47	1028,92

continua

TABELA 5 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCR e pelo Algoritmo MCR_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert$		$\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	Γ_{MCR}	Γ_{MCR_L}	$\frac{\Gamma_{\mathsf{MCR}_L}}{\Gamma_{\mathsf{MCR}}}$ (%)
p_hat500-3	311 000 275	≥	70 248 049	–	21 001 698	–	2612,16	–	–
p_hat700-1	**68 605**		70 591	102,89	11 772	16,68	0,51	1,33	260,81
p_hat700-2	**5 640 897**		7 948 935	140,92	2 186 568	27,51	43,85	565,64	1290,03
p_hat700-3	≥ 588 005 141	≥	50 840 046	–	16 865 865	–	–	–	–
p_hat1000-1	405 625		**400 223**	98,67	67 330	16,82	3,09	6,87	222,34
p_hat1000-2	379 697 715	≥	44 329 050	–	13 989 246	–	2753,03	–	–
p_hat1000-3	≥ 478 274 824	≥	33 785 213	–	10 954 692	–	–	–	–
p_hat1500-1	2 514 819		**2 478 089**	98,54	433 551	17,50	17,79	49,09	275,90
p_hat1500-2	≥ 442 140 373	≥	19 535 630	–	7 221 634	–	–	–	–
p_hat1500-3	≥ 240 308 239	≥	12 472 351	–	4 658 639	–	–	–	–
san200_0.7_1	6563		**2911**	44,35	571	19,62	0,06	0,12	188,05
san200_0.7_2	3107		**2913**	93,76	377	12,94	0,03	0,10	280,47
san200_0.9_1	400 545		**37 815**	9,44	8848	23,40	2,68	3,24	120,82
san200_0.9_2	760 115		**301 243**	39,63	68 084	22,60	5,61	28,56	508,70
san200_0.9_3	**64 007**		875 273	1367,46	176 000	20,11	0,52	53,09	10 280,85
san400_0.5_1	**4325**		5681	131,35	248	4,37	0,16	0,60	376,80
san400_0.7_1	218 339		**151 175**	69,24	37 852	25,04	2,82	27,40	973,16
san400_0.7_2	55 197		**50 205**	90,96	8691	17,31	0,76	6,91	908,26
san400_0.7_3	797 757		**618 561**	77,54	77 040	12,45	5,46	39,81	729,52

continua

TABELA 5 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCR e pelo Algoritmo MCR_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCR}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCR}}(G)\rvert}$ (%)	Γ_{MCR}	Γ_{MCR_L}	$\frac{\Gamma_{\mathsf{MCR}_L}}{\Gamma_{\mathsf{MCR}}}$ (%)
san400_0.9_1	≥1 816 617 595	≥ 66 514 845	–	21 721 149	–	–	–	–
san1000	443 593	**290 073**	65,39	13 037	4,49	10,46	144,04	1377,12
sanr200_0.7	362 711	**306 015**	84,37	57 115	18,66	1,16	5,90	508,78
sanr200_0.9	76 512 469	**64 648 321**	84,49	14 456 525	22,36	425,49	2669,89	627,49
sanr400_0.5	600 811	**553 081**	92,06	93 612	16,93	2,57	7,76	301,66
sanr400_0.7	176 185 167	**148 219 673**	84,13	28 638 636	19,32	831,08	2368,29	284,97

TABELA 6 – Comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}$ (%)	Γ_{MCS}	Γ_{MCS_L}	$\frac{\Gamma_{\mathsf{MCS}_L}}{\Gamma_{\mathsf{MCS}}}$ (%)
C125.9	**12 601**	30 739	243,94	2845	9,26	0,11	1,66	1575,78
C250.9	492 218 109	≥ 111 490 943	–	15 318 495	–	4038,12	–	–
C500.9	≥ 717 444 173	≥ 85 700 761	–	15 761 645	–	–	–	–
C1000.9	≥ 948 199 346	≥ 89 831 245	–	21 862 694	–	–	–	–
C2000.5	≥1 276 827 240	≥ 235 391 425	–	36 447 071	–	–	–	–
C2000.9	≥ 858 069 555	≥ 72 753 071	–	16 361 172	–	–	–	–
C4000.5	≥ 999 407 328	≥ 198 477 373	–	31 513 789	–	–	–	–
DSJC500_5	**1 388 293**	1 457 085	104,96	210 576	14,45	7,61	27,17	357,25
DSJC1000_5	**98 502 673**	102 140 347	103,69	15 428 255	15,10	548,51	2252,54	410,67
MANN_a9	93	**83**	89,25	5	6,02	0,00	0,00	209,91
MANN_a27	**18 773**	20 923	111,45	2445	11,69	0,93	19,96	2156,61
MANN_a45	**422 727**	528 284	124,97	100 542	19,03	69,23	4434,84	6406,40
MANN_a81	≥ 3 335 248	≥ 52 616	–	2899	–	–	–	–
brock200_1	**285 599**	364 375	127,58	50 793	13,94	1,66	10,72	645,83
brock200_2	**4821**	5007	103,86	631	12,60	0,03	0,10	290,32
brock200_3	**16 017**	17 751	110,83	2531	14,26	0,11	0,32	299,52
brock200_4	**59 143**	68 327	115,53	9427	13,80	0,31	1,47	472,80
brock400_1	**173 390 583**	≥ 216 185 856	–	32 507 098	–	1081,17	–	–
brock400_2	**64 214 365**	95 521 231	148,75	14 193 603	14,86	357,15	4479,29	1254,19

continua

TABELA 6 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	Γ_{MCS}	Γ_{MCS_L}	$\frac{\Gamma_{\mathsf{MCS}_L}}{\Gamma_{\mathsf{MCS}}}$ (%)
brock400_3	**124 234 093**	181 376 109	146,00	27 751 507	15,30	644,97	5954,83	923,28
brock400_4	**57 829 317**	84 902 079	146,81	12 474 851	14,69	411,56	3929,40	954,76
brock800_1	≥1 293 442 040	≥ 222 086 033	–	34 236 765	–	–	–	–
brock800_2	≥1 201 683 061	≥ 255 388 298	–	39 851 402	–	–	–	–
brock800_3	1 160 511 882	≥ 269 774 813	–	42 338 456	–	7080,56	–	–
brock800_4	720 680 877	≥ 236 004 365	–	36 892 364	–	4981,44	–	–
c-fat200-1	377	377	100,00	0	0,00	0,00	0,00	156,64
c-fat200-2	353	353	100,00	0	0,00	0,00	0,01	143,89
c-fat200-5	285	285	100,00	0	0,00	0,01	0,01	146,78
c-fat500-1	973	973	100,00	0	0,00	0,12	0,16	140,13
c-fat500-2	949	949	100,00	0	0,00	0,02	0,03	132,21
c-fat500-5	873	873	100,00	0	0,00	0,08	0,10	126,36
c-fat500-10	749	749	100,00	0	0,00	0,02	0,02	103,66
gen200_p0.9_44	**55 383**	257 067	464,16	19 742	7,68	0,71	20,38	2877,60
gen200_p0.9_55	**247 895**	647 377	261,15	101 916	15,74	2,03	57,01	2805,14
gen400_p0.9_55	≥ 676 442 562	≥ 82 445 315	–	3 518 145	–	–	–	–
gen400_p0.9_65	≥ 736 692 220	≥ 82 440 516	–	10 979 864	–	–	–	–
gen400_p0.9_75	≥ 833 773 639	≥ 88 488 386	–	19 386 634	–	–	–	–

continua

TABELA 6 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	Γ_{MCS}	Γ_{MCS_L}	$\frac{\Gamma_{\mathsf{MCS}_L}}{\Gamma_{\mathsf{MCS}}}$ (%)
hamming6-2	**65**	115	176,92	0	0,00	0,00	0,00	241,94
hamming6-4	**155**	171	110,32	0	0,00	0,00	0,00	130,96
hamming8-2	**257**	4317	1679,77	205	4,75	0,01	0,77	9539,29
hamming8-4	18 317	**10 513**	57,39	2034	19,35	0,12	0,31	267,10
hamming10-2	**1025**	≥ 3 826 814	–	1 737 991	–	0,13	–	–
hamming10-4	≥1 357 180 710	≥ 92 951 806	–	26 879 756	–	–	–	–
johnson8-2-4	**49**	57	116,33	0	0,00	0,00	0,00	204,42
johnson8-4-4	173	**149**	86,13	16	10,74	0,00	0,00	256,48
johnson16-2-4	**474 641**	1 407 555	296,55	120 201	8,54	0,41	3,44	834,69
johnson32-2-4	≥8 758 602 811	≥2 049 771 990	–	420 536 909	–	–	–	–
keller4	**14 871**	16 937	113,89	2184	12,89	0,08	0,28	355,97
keller5	≥ 821 108 878	≥ 71 436 861	–	17 675 684	–	–	–	–
keller6	≥ 487 704 086	≥ 26 710 989	–	11 208 043	–	–	–	–
p_hat300-1	2971	**2783**	93,67	275	9,88	0,04	0,06	137,57
p_hat300-2	**3981**	5649	141,90	398	7,05	0,11	0,30	272,47
p_hat300-3	**473 405**	1 964 775	415,03	132 107	6,72	4,58	142,55	3109,59
p_hat500-1	15 383	**15 331**	99,66	1603	10,46	0,22	0,35	157,12
p_hat500-2	**114 997**	314 653	273,62	18 054	5,74	1,57	25,53	1626,46

continua

TABELA 6 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert$		$\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	Γ_{MCS}	Γ_{MCS_L}	$\frac{\Gamma_{\mathsf{MCS}_L}}{\Gamma_{\mathsf{MCS}}}$ (%)
p_hat500-3	**15 132 785**	≥	65 126 832	–	2 828 749	–	240,87	–	–
p_hat700-1	**43 791**		47 007	107,34	4807	10,23	0,72	1,19	165,78
p_hat700-2	**577 767**		2 444 701	423,13	119 517	4,89	10,17	322,91	3175,84
p_hat700-3	186 502 409	≥	39 313 357	–	1 011 480	–	3717,79	–	–
p_hat1000-1	**233 123**		262 479	112,59	30 120	11,48	3,19	5,45	170,88
p_hat1000-2	**22 886 363**	≥	40 286 036	–	1 423 309	–	374,26	–	–
p_hat1000-3	≥ 413 639 177	≥	24 628 236	–	594 029	–	–	–	–
p_hat1500-1	**1 545 743**		1 710 267	110,64	190 447	11,14	15,18	41,98	276,49
p_hat1500-2	≥ 425 221 285	≥	19 856 800	–	215 249	–	–	–	–
p_hat1500-3	≥ 201 768 285	≥	9 687 673	–	97 170	–	–	–	–
san200_0.7_1	1325		**1257**	94,87	259	20,60	0,04	0,11	256,28
san200_0.7_2	**1465**		1687	115,15	140	8,30	0,03	0,08	232,76
san200_0.9_1	46 047		**10 165**	22,08	994	9,78	0,49	1,02	208,54
san200_0.9_2	59 251		**48 959**	82,63	4047	8,27	0,58	5,62	966,58
san200_0.9_3	**11 749**		150 091	1277,48	11 436	7,62	0,15	10,90	7214,08
san400_0.5_1	**3281**		3593	109,51	403	11,22	0,16	0,45	272,84
san400_0.7_1	**48 069**		53 735	111,79	15 228	28,34	1,08	10,61	984,19
san400_0.7_2	28 455		**20 157**	70,84	4542	22,53	0,55	3,09	560,58
san400_0.7_3	**254 345**		416 317	163,68	62 062	14,91	2,50	27,39	1096,04

continua

TABELA 6 – *Continuação*: comparação entre a árvore de estados $\mathcal{T}$(G) gerada pelo Algoritmo MCS e pelo Algoritmo MCS_L com instâncias DIMACS como entrada.

Instância	$\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert$	$\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert$	$\frac{\lvert\mathcal{T}_{\mathsf{MCS}_L}(G)\rvert}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	τ_L	$\frac{\tau_L}{\lvert\mathcal{T}_{\mathsf{MCS}}(G)\rvert}$ (%)	Γ_{MCS}	Γ_{MCS_L}	$\frac{\Gamma_{\mathsf{MCS}_L}}{\Gamma_{\mathsf{MCS}}}$ (%)
san400_0.9_1	$\geq$ 713 387 377	**4 783 765**	–	992 981	–	–	1436,66	–
san1000	**168 973**	189 941	112,41	21 541	11,34	7,17	82,31	1147,55
sanr200_0.7	**134 305**	153 403	114,22	21 551	14,05	0,72	3,58	500,18
sanr200_0.9	**6 676 229**	16 012 911	239,85	2 176 160	13,59	76,58	1357,68	1772,89
sanr400_0.5	**327 945**	345 063	105,22	48 026	13,92	1,97	6,23	316,45
sanr400_0.7	**56 304 553**	72 742 955	129,20	10 737 965	14,76	272,72	1868,26	685,04

4.3.2 Grafos Aleatórios

Nessa seção serão apresentados resultados para grafos aleatórios. Também é utilizado métodos de inferência estatística. Antes de prosseguir com essa abordagem, foi calculado o número de cores que Greedy coloriu os vértices das instâncias de grafos $\mathcal{G}_{n,1/2}$, tomando como entrada as ordenações π e π_L. O número de cores computado por $\mathsf{Greedy}(G, \pi)$ será denotado por $c(G, \pi)$. De forma análoga, $c(G, \pi_L)$ denota o número de cores computado por $\mathsf{Greedy}(G, \pi_L)$. A Tabela 7 sumariza quantas vezes $c(G, \pi_L) < c(G, \pi)$ e quantas vezes $c(G, \pi_L) = c(G, \pi)$. Foram testadas 150 instâncias grafos aleatórios $\mathcal{G}_{n,1/2}$. Dessas, em 28 número de cores $c(G, \pi_L)$ foi menor que $c(G, \pi)$. Por outro lado, em 34 o número de cores $c(G, \pi_L)$ foi maior que $c(G, \pi)$. Por fim, em 18 instâncias as duas abordagens se igualaram no número de cores computado. Os resultados para todas instâncias estão na Tabela 15, no Apêndice C.

TABELA 7 – Teste com $\mathsf{Greedy}(G)$ onde $G \in \mathcal{G}_{n,1/2}$. Quantidade de instâncias onde $c(G, \pi_L) < c(G, \pi)$, bem como, $c(G, \pi_L) = c(G, \pi)$. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho 10.

$\lvert V\rvert$	$\mu_{\lvert E\rvert}$	$c(G, \pi_L) < c(G, \pi)$	$c(G, \pi_L) = c(G, \pi)$	$c(G, \pi_L) > c(G, \pi)$
300	224 861	1	3	6
350	304 683	4	1	5
400	399 112	6	0	4
450	505 276	3	1	6
500	624 073	6	2	2
550	754 778	2	4	4
600	898 827	3	3	4
650	1 054 309	3	1	6
700	1 223 442	4	2	4
750	1 404 643	5	0	5
800	1 598 804	3	3	4
850	1 804 326	3	2	5
900	2 023 463	5	3	2
950	2 254 718	4	4	2
1000	2 498 386	3	4	3

Por meio dos métodos de teste de hipótese, a próxima seção apresenta resultados relacionados as diferenças nas medidas de desempenho de algoritmos de $\mathcal{A}$ e $\mathcal{A}_L$.

4.3.2.1 Teste de Hipótese

A base estatística necessária para essa seção pode ser consultada na Seção 4.1. Aqui, o objetivo é fazer inferências em relação ao desempenho dos algoritmos modificados com a ordenação LexBFS utilizando como entrada instâncias de grafos aleatórios do modelo $\mathcal{G}_{n,p}$, onde $p = \frac{1}{2}$. Com o objetivo de realizar testes de hipótese, são considerados os seguintes algoritmos: MCLIQ contra MCLIQ_L (Tabela 8), MCQ contra MCQ_L (Tabela 9),

DYN contra DYN_L (Tabela 10), MCR contra MCR_L (Tabela 11) e MCS contra MCS_L (Tabela 12). O desempenho desses algoritmos será avaliado tanto pelo número de estados quanto pelo tempo de execução.

Como visto anteriormente, um dos primeiros passos para se prosseguir com o teste de hipótese é construir a v.a. D formada pelas diferenças $D_1 = X_1 - Y_1, D_2 = X_2 - Y_2, \ldots, D_n = X_n - Y_n$ de duas amostras pareadas. No nosso caso, pela geração dos grafos aleatórios, as variáveis $|\mathcal{T}(G)|$ e Γ são variáveis aleatórias. Inicialmente, esse primeiro passo para o teste hipótese será conduzido considerando a v.a. $|\mathcal{T}(G)|$. O intuito é comparar as amostras $(|\mathcal{T}_{1,\mathcal{A}}(G)|, |\mathcal{T}_{2,\mathcal{A}}(G)|, \ldots, |\mathcal{T}_{n,\mathcal{A}}(G)|)$ e $(|\mathcal{T}_{1,\mathcal{A}_L}(G)|, |\mathcal{T}_{2,\mathcal{A}_L}(G)|, \ldots, |\mathcal{T}_{n,\mathcal{A}_L}(G)|)$ e construir a v.a. $D_\mathcal{T}$ como $D_1 = |\mathcal{T}_{1,\mathcal{A}}(G)| - |\mathcal{T}_{1,\mathcal{A}_L}(G)|, D_2 = |\mathcal{T}_{2,\mathcal{A}}(G)| - |\mathcal{T}_{2,\mathcal{A}_L}(G)| \ldots D_n = |\mathcal{T}_{n,\mathcal{A}}(G)| - |\mathcal{T}_{n,\mathcal{A}_L}(G)|$. A mesma ideia vale para a v.a. Γ.

Será realizado o teste de hipótese seguindo o teste t de Student. Antes de dar prosseguimento é preciso dizer que foram realizados testes de normalidade para o conjunto de dados. Foram utilizados os testes de Anderson-Darling (ANDERSON; DARLING, 1952), Cramér-von Mises (CRAMÉR, 1928), Kolmogorov-Smirnov (SHIRYAYEV, 1992) e Shapiro-Wilk (SHAPIRO; WILK, 1965). Em ao menos um tipo de teste o resultado foi que a maioria dos conjuntos de dados pode ser modelado por uma distribuição normal.

Para os resultados das tabelas de 8 a 12 a diferença entre a médias do número de estados gerados por um algoritmo $\mathcal{A}$ e sua correspondente alteração $\mathcal{A}_L$, será denotada por $\overline{D}_\mathcal{T}$, e o valor-p é calculado com a estatística t da Equação 4.1, calculada como

$$T = \frac{\overline{D}_\mathcal{T} - \mu_{D_\mathcal{T}}}{S_{D_\mathcal{T}}\sqrt{n}}$$

onde $D_\mathcal{T}$ é formado pelas diferenças $D_i = (|\mathcal{T}_{i,\mathcal{A}}(G)| - |\mathcal{T}_{i,\mathcal{A}_L}(G)|)$, isto é, a diferença na i-ésima observação entre o número de estados $|\mathcal{T}(G)|$ gerados por um algoritmo $\mathcal{A}$ e seu correspondente $\mathcal{A}_L$, e

$$\overline{D}_\mathcal{T} = \frac{1}{n}\sum_{i=1}^{n}(|\mathcal{T}_{i,\mathcal{A}}(G)| - |\mathcal{T}_{i,\mathcal{A}_L}(G)|) = \mu_{\mathcal{T}_\mathcal{A}(G)} - \mu_{\mathcal{T}_{\mathcal{A}_L}(G)},$$

$$S^2_{D_\mathcal{T}} = \frac{1}{n-1}\sum_{i=1}^{n}(D_i - \overline{D}_\mathcal{T})^2.$$

Para o teste de hipótese, o objetivo é determinar se, ao gerar a árvore de estados $\mathcal{T}$(G), um algoritmo $\mathcal{A}$ é mais eficiente em podar a árvore de estados em relação a sua correspondente modificação em $\mathcal{A}_L$ quando o valor calculado $\overline{d}_\mathcal{T}$ de $\overline{D}_\mathcal{T}$ for negativo. O nível de significância definido é $\alpha = 0{,}05$. Os demais passos para o teste de hipótese estão descritos abaixo. Seja $\mu_{D_\mathcal{T}} = \mu_{\mathcal{T}_\mathcal{A}(G)} - \mu_{\mathcal{T}_{\mathcal{A}_L}(G)}$.

1. $H_0 : \mu_{D_\mathcal{T}} = 0$ e $H_1 : \mu_{D_\mathcal{T}} \neq 0$

2. Sob a hipótese H_0, a diferença $\mu_{D_\mathcal{T}}$ tem distribuição normal $N(0, \sigma^2_{D_\mathcal{T}})$. Por isso, com base em na Equação 4.3, a estatística a ser utilizada será

$$t = \frac{\overline{d} - 0}{s_{D_\mathcal{T}}/\sqrt{n}} = \frac{\overline{d}}{s_{D_\mathcal{T}}/\sqrt{n}}$$

3. $\alpha = 0{,}05$

4. Calcular o valor-p como em 4.2

O intervalo de confiança $100(1-\alpha)\%$ para a média $\mu_{D_\mathcal{T}}$ é baseado no fato de que $T = \frac{\overline{D}_\mathcal{T} - \mu_{D_\mathcal{T}}}{S_{D_\mathcal{T}}\sqrt{n}}$ tem distribuição t de Student com $n-1$ graus de liberdade. O IC para $\mu_{D_\mathcal{T}}$ é

$$\overline{d}_\mathcal{T} - t_\gamma \frac{s_{D_\mathcal{T}}}{\sqrt{n}} < \mu_{D_\mathcal{T}} < \overline{d}_\mathcal{T} + t_\gamma \frac{s_{D_\mathcal{T}}}{\sqrt{n}}.$$

Assim, definido o nível de confiança para $\gamma = 0{,}95$, o IC reportado nas tabelas de 8 até 12 é o IC com 95% de confiança. Lembrando que os valores de t_γ estão na Tabela 18, no Anexo A. Além disso, para facilitar a comparação na Figura 10 estão apresentados gráficos em escala logarítmica que mostram, para cada instância de grafo aleatório $\mathcal{G}_{n,1/2}$ com $n \in \{300, 350, \ldots, 1000\}$ vértices, o número de estados da árvore de estados gerada por um algoritmo de $\mathcal{A}$ e de $\mathcal{A}_L$.

Note que o mesmo raciocínio aplicado até aqui para a variável $|\mathcal{T}(G)|$ vale para a variável Γ. É preciso definir o nível de significância que, assim como antes, será $\alpha = 0{,}05$. O teste de hipótese sob a variável Γ é equivalente a determinar se um algoritmo $\mathcal{A}$ tem tempo de execução $\Gamma_\mathcal{A}$ menor que o tempo de execução $\Gamma_{\mathcal{A}_L}$, de um algoritmo $\mathcal{A}_L$, quando o valor calculado $\overline{d}_\Gamma$ de $\overline{D}_\Gamma$ for negativo. Os cálculos são feitos substituindo $|\mathcal{T}(G)|$ por Γ. Com isso, D_Γ é análoga a $D_\mathcal{T}$, o que leva ao IC para média μ_{D_Γ} ser calculado com 95% de nível de confiança.

Após, todos valores computados e reportados, pode-se verificar que a modificação com o algoritmo LexBFS fez diferença significativa, pois valor-$p \leq \alpha$, tanto para o número de estados quanto para o tempo de execução. Na Tabela 8 o algoritmo MCLIQ_L é mais eficiente na poda da árvore de estados em quase toda instância testada com o teste hipótese. Olhando para o valor-p na coluna $|\mathcal{T}|$, a exceção são as instâncias onde $|V| = 300$, $|V| = 850$ e $|V| = 900$ vértices. Na mesma tabela, sob a v.a. Γ é possível verificar que valor-$p \leq \alpha$ para todas instâncias. Mas, os valores de $\overline{d}_\Gamma = \mu_{\Gamma_{\mathsf{MCLIQ}}(G)} - \mu_{\Gamma_{\mathsf{MCLIQ}_L}(G)}$ são negativos para todos testes. Logo, o Algoritmo MCLIQ toma significativamente menos tempo de execução em CPU do que MCLIQ_L.

Na Tabela 9 o algoritmo MCQ_L é mais eficiente na poda da árvore de estados em toda instância. Olhando para a coluna $|\mathcal{T}|$, o valor-$p \leq \alpha$ em todas instâncias . Na coluna Γ é possível verificar que valor-$p \leq \alpha$ para todas instâncias. Mas, os valores de $\overline{d}_\Gamma = \mu_{\Gamma_{\mathsf{MCQ}}(G)} - \mu_{\Gamma_{\mathsf{MCQ}_L}(G)}$ são negativos para todos testes. Logo, o Algoritmo MCQ toma significativamente menos tempo de execução em CPU do que MCQ_L.

Na Tabela 10 o algoritmo DYN_L *não* é mais eficiente na poda da árvore de estados em toda instância. Olhando para a coluna $|\mathcal{T}|$, o valor-$p \leq \alpha$ em todas instâncias. Porém, os valores de $\overline{d}_{\mathcal{T}} = \mu_{\mathcal{T}_{\mathsf{DYN}}(G)} - \mu_{\mathcal{T}_{\mathsf{DYN}_L}(G)}$ são negativos para todos testes. Na coluna Γ é possível verificar que valor-$p \leq \alpha$ para toda instância. Mas, os valores de $\overline{d}_{\Gamma} = \mu_{\Gamma_{\mathsf{DYN}}(G)} - \mu_{\Gamma_{\mathsf{DYN}_L}(G)}$ são negativos para todos testes. Logo, o Algoritmo DYN toma significativamente menos tempo de execução em CPU do que DYN_L.

Na Tabela 11 o algoritmo MCR_L é mais eficiente na poda da árvore de estados em toda instância. Olhando para a coluna $|\mathcal{T}|$, o valor-$p \leq \alpha$ em todas instâncias. Na coluna Γ é possível verificar que valor-$p \leq \alpha$ para toda instância. Mas, os valores de $\overline{d}_{\Gamma} = \mu_{\Gamma_{\mathsf{MCR}}(G)} - \mu_{\Gamma_{\mathsf{MCR}_L}(G)}$ são negativos para todos testes. Logo, o Algoritmo MCR toma significativamente menos tempo de execução em CPU do que MCR_L.

Na Tabela 12 o algoritmo MCS_L *não* é mais eficiente na poda da árvore de estados em toda instância. Olhando para a coluna $|\mathcal{T}|$, o valor-$p \leq \alpha$ em todas instâncias. Porém, os valores de $\overline{d}_{\mathcal{T}} = \mu_{\mathcal{T}_{\mathsf{MCS}}(G)} - \mu_{\mathcal{T}_{\mathsf{MCS}_L}(G)}$ são negativos para todos testes. Na coluna Γ é possível verificar que valor-$p \leq \alpha$ para toda instância. Mas, os valores de $\overline{d}_{\Gamma} = \mu_{\Gamma_{\mathsf{MCS}}(G)} - \mu_{\Gamma_{\mathsf{MCS}_L}(G)}$ são negativos para todos testes. Logo, o Algoritmo MCS toma significativamente menos tempo de execução em CPU do que MCS_L.

TABELA 8 – Teste de hipótese entre os algoritmos MCLIQ e MCLIQ_L em relação a média de estados gerados e tempo de execução.

	$\lvert\mathcal{T}\rvert$			Γ		
$\lvert V\rvert$	valor-p	$\overline{d}_{\mathcal{T}}$	IC($\mu_{D_{\mathcal{T}}}$, 95%)	valor-p	$\overline{d}_{\Gamma}$	IC($\mu_{D_{\Gamma}}$, 95%)
300	$\mathbf{2{,}93\times10^{-1}}$	5315,4	($-5{,}5\times10^{3}$, $1{,}6\times10^{4}$)	$1{,}44\times10^{-6}$	$-1{,}09$	($-1{,}31$, $-0{,}87$)
350	$8{,}38\times10^{-5}$	26 433,4	($1{,}8\times10^{4}$, $3{,}5\times10^{4}$)	$5{,}98\times10^{-7}$	$-2{,}86$	($-3{,}38$, $-2{,}33$)
400	$4{,}85\times10^{-2}$	50 861,6	($4{,}1\times10^{2}$, $1{,}0\times10^{5}$)	$4{,}87\times10^{-8}$	$-5{,}69$	($-6{,}47$, $-4{,}91$)
450	$7{,}74\times10^{-3}$	165 832,0	($5{,}6\times10^{4}$, $2{,}8\times10^{5}$)	$4{,}44\times10^{-9}$	$-13{,}13$	($-14{,}50$, $-11{,}76$)
500	$3{,}53\times10^{-5}$	268 637,2	($1{,}9\times10^{5}$, $3{,}5\times10^{5}$)	$2{,}50\times10^{-6}$	$-25{,}13$	($-30{,}58$, $-19{.}69$)
550	$3{,}48\times10^{-2}$	546 819,0	($4{,}9\times10^{4}$, $1{,}0\times10^{6}$)	$2{,}64\times10^{-6}$	$-43{,}98$	($-53{,}57$, $-34{,}39$)
600	$6{,}34\times10^{-4}$	833 043,2	($4{,}6\times10^{5}$, $1{,}2\times10^{6}$)	$1{,}12\times10^{-7}$	$-80{,}25$	($-92{,}35$, $-68{,}15$)
650	$3{,}06\times10^{-3}$	1 579 995,6	($6{,}9\times10^{5}$, $2{,}5\times10^{6}$)	$5{,}67\times10^{-7}$	$-105{,}80$	($-125{,}04$, $-86{,}56$)
700	$1{,}51\times10^{-2}$	2 040 281,8	($5{,}0\times10^{5}$, $3{,}6\times10^{6}$)	$8{,}65\times10^{-8}$	$-205{,}18$	($-235{,}19$, $-175{,}17$)
750	$4{,}90\times10^{-8}$	3 018 134,4	($2{,}6\times10^{6}$, $3{,}4\times10^{6}$)	$6{,}27\times10^{-7}$	$-380{,}08$	($-450{,}03$, $-310{,}14$)
800	$3{,}86\times10^{-5}$	4 628 911,8	($3{,}2\times10^{6}$, $6{,}0\times10^{6}$)	$2{,}30\times10^{-7}$	$-530{,}61$	($-617{,}50$, $-443{,}71$)
850	$\mathbf{1{,}26\times10^{-1}}$	5 113 446,6	($-1{,}7\times10^{6}$, $1{,}2\times10^{7}$)	$2{,}29\times10^{-8}$	$-874{,}44$	($-984{,}30$, $-764{,}58$)
900	$\mathbf{5{,}08\times10^{-2}}$	7 935 305,8	($-3{,}3\times10^{4}$, $1{,}6\times10^{7}$)	$6{,}24\times10^{-8}$	$-1094{,}76$	($-1249{,}01$, $-940{,}52$)
950	$4{,}32\times10^{-3}$	12 301 750,4	($4{,}9\times10^{6}$, $2{,}0\times10^{7}$)	$1{,}95\times10^{-10}$	$-1468{,}26$	($-1576{,}01$, $-1360{,}50$)
1000	$4{,}09\times10^{-4}$	21 423 725,0	($1{,}3\times10^{7}$, $3{,}0\times10^{7}$)	$2{,}93\times10^{-11}$	$-2359{,}02$	($-2499{,}05$, $-2218{,}99$)

Em negrito testes onde valor-$p > 0{,}05$

TABELA 9 – Teste de hipótese entre os algoritmos MCQ e MCQ_L em relação a média de estados gerados e tempo de execução.

		$\lvert\mathcal{T}\rvert$			Γ		
$\lvert V\rvert$	valor-p	$\overline{d}_{\mathcal{T}}$	IC($\mu_{D_{\mathcal{T}}}$, 95%)		valor-p	$\overline{d}_{\Gamma}$	IC($\mu_{D_{\Gamma}}$, 95%)	
300	$1{,}98\times10^{-6}$	11 137,6	($8{,}8\times10^{3}$,	$1{,}3\times10^{4}$)	$8{,}54\times10^{-7}$	−1,11	(−1,33,	−0,90)
350	$2{,}47\times10^{-7}$	25 094,2	($2{,}1\times10^{4}$,	$2{,}9\times10^{4}$)	$9{,}39\times10^{-7}$	−2,38	(−2,84,	−1,92)
400	$3{,}45\times10^{-5}$	51 570,0	($3{,}6\times10^{4}$,	$6{,}7\times10^{4}$)	$6{,}66\times10^{-8}$	−5,36	(−6,12,	−4,60)
450	$2{,}66\times10^{-6}$	116 133,2	($9{,}1\times10^{4}$,	$1{,}4\times10^{5}$)	$2{,}39\times10^{-8}$	−12,32	(−13,87,	−10,76)
500	$3{,}22\times10^{-7}$	231 421,0	($1{,}9\times10^{5}$,	$2{,}7\times10^{5}$)	$7{,}89\times10^{-9}$	−26,24	(−29,16,	−23,32)
550	$1{,}90\times10^{-6}$	384 434,6	($3{,}0\times10^{5}$,	$4{,}7\times10^{5}$)	$5{,}45\times10^{-8}$	−41,03	(−46,72,	−35,34)
600	$4{,}23\times10^{-6}$	690 943,4	($5{,}3\times10^{5}$,	$8{,}5\times10^{5}$)	$2{,}62\times10^{-7}$	−83,53	(−97,42,	−69,64)
650	$7{,}94\times10^{-9}$	1 004 369,8	($8{,}9\times10^{5}$,	$1{,}1\times10^{6}$)	$2{,}51\times10^{-7}$	−117,74	(−137,22,	−98,27)
700	$3{,}04\times10^{-7}$	1 614 023,2	($1{,}3\times10^{6}$,	$1{,}9\times10^{6}$)	$5{,}14\times10^{-7}$	−200,05	(−236,02,	−164,08)
750	$1{,}20\times10^{-11}$	3 195 507,2	($3{,}0\times10^{6}$,	$3{,}4\times10^{6}$)	$5{,}60\times10^{-8}$	−344,03	(−391,91,	−296,15)
800	$9{,}78\times10^{-9}$	5 310 768,4	($4{,}7\times10^{6}$,	$5{,}9\times10^{6}$)	$3{,}71\times10^{-7}$	−544,93	(−639,26,	−450,59)
850	$1{,}01\times10^{-7}$	7 493 970,8	($6{,}4\times10^{6}$,	$8{,}6\times10^{6}$)	$2{,}11\times10^{-8}$	−796,76	(−895,91,	−697,61)
900	$5{,}84\times10^{-7}$	9 363 241,6	($7{,}7\times10^{6}$,	$1{,}1\times10^{7}$)	$7{,}60\times10^{-9}$	−1095,41	(−1216,83,	−973,98)
950	$9{,}12\times10^{-10}$	13 064 092,0	($1{,}2\times10^{7}$,	$1{,}4\times10^{7}$)	$4{,}14\times10^{-12}$	−1409,25	(−1476,48,	−1342,02)
1000	$1{,}08\times10^{-10}$	19 517 230,4	($1{,}8\times10^{7}$,	$2{,}1\times10^{7}$)	$3{,}87\times10^{-10}$	−2292,65	(−2474,34,	−2110,96)

TABELA 10 – Teste de hipótese entre os algoritmos DYN e DYN_L em relação a média de estados gerados e tempo de execução.

		$\lvert\mathcal{T}\rvert$			Γ		
$\lvert V\rvert$	valor-p	$\overline{d}_{\mathcal{T}}$	IC($\mu_{D_{\mathcal{T}}}$, 95%)		valor-p	$\overline{d}_{\Gamma}$	IC($\mu_{D_{\Gamma}}$, 95%)	
300	$4{,}70\times10^{-6}$	−6399,6	($-7{,}9\times10^{3}$,	$-4{,}9\times10^{3}$)	$1{,}03\times10^{-5}$	−0,45	(−0,57,	−0,34)
350	$8{,}96\times10^{-8}$	−16 148,4	($-1{,}9\times10^{4}$,	$-1{,}4\times10^{4}$)	$9{,}90\times10^{-7}$	−1,21	(−1,45,	−0,98)
400	$9{,}13\times10^{-8}$	−30 202,6	($-3{,}5\times10^{4}$,	$-2{,}6\times10^{4}$)	$6{,}38\times10^{-6}$	−2,95	(−3,66,	−2,23)
450	$4{,}15\times10^{-6}$	−46 486,4	($-5{,}7\times10^{4}$,	$-3{,}6\times10^{4}$)	$1{,}51\times10^{-5}$	−6,80	(−8,64,	−4,97)
500	$3{,}86\times10^{-5}$	−74 233,4	($-9{,}7\times10^{4}$,	$-5{,}2\times10^{4}$)	$3{,}85\times10^{-6}$	−14,08	(−17,29,	−10,86)
550	$1{,}66\times10^{-5}$	−110 213,4	($-1{,}4\times10^{5}$,	$-8{,}0\times10^{4}$)	$4{,}49\times10^{-6}$	−26,24	(−32,34,	−20,14)
600	$9{,}10\times10^{-4}$	−158 692,4	($-2{,}3\times10^{5}$,	$-8{,}5\times10^{4}$)	$4{,}01\times10^{-6}$	−47,18	(−58,00,	−36,36)
650	$6{,}14\times10^{-7}$	−256 317,0	($-3{,}0\times10^{5}$,	$-2{,}1\times10^{5}$)	$3{,}26\times10^{-8}$	−76,15	(−86,11,	−66,19)
700	$8{,}15\times10^{-7}$	−482 946,8	($-5{,}7\times10^{5}$,	$-3{,}9\times10^{5}$)	$4{,}32\times10^{-7}$	−112,91	(−132,80,	−93,02)
750	$4{,}13\times10^{-6}$	−540 923,8	($-6{,}7\times10^{5}$,	$-4{,}2\times10^{5}$)	$8{,}04\times10^{-7}$	−212,86	(−253,19,	−172,53)
800	$6{,}74\times10^{-8}$	−921 146,6	($-1{,}1\times10^{6}$,	$-7{,}9\times10^{5}$)	$2{,}98\times10^{-6}$	−329,49	(−402,40,	−256,58)
850	$7{,}34\times10^{-8}$	−1 524 006,8	($-1{,}7\times10^{6}$,	$-1{,}3\times10^{6}$)	$1{,}36\times10^{-6}$	−522,81	(−628,16,	−417,45)
900	$4{,}76\times10^{-9}$	−2 435 116,4	($-2{,}7\times10^{6}$,	$-2{,}2\times10^{6}$)	$4{,}46\times10^{-8}$	−710,40	(−806,72,	−614,08)
950	$1{,}59\times10^{-8}$	−3 546 173,0	($-4{,}0\times10^{6}$,	$-3{,}1\times10^{6}$)	$7{,}11\times10^{-10}$	−923,00	(−1001,32,	−844,69)
1000	$1{,}98\times10^{-8}$	−4 902 904,0	($-5{,}5\times10^{6}$,	$-4{,}3\times10^{6}$)	$2{,}78\times10^{-10}$	−1608,93	(−1731,79,	−1486,06)

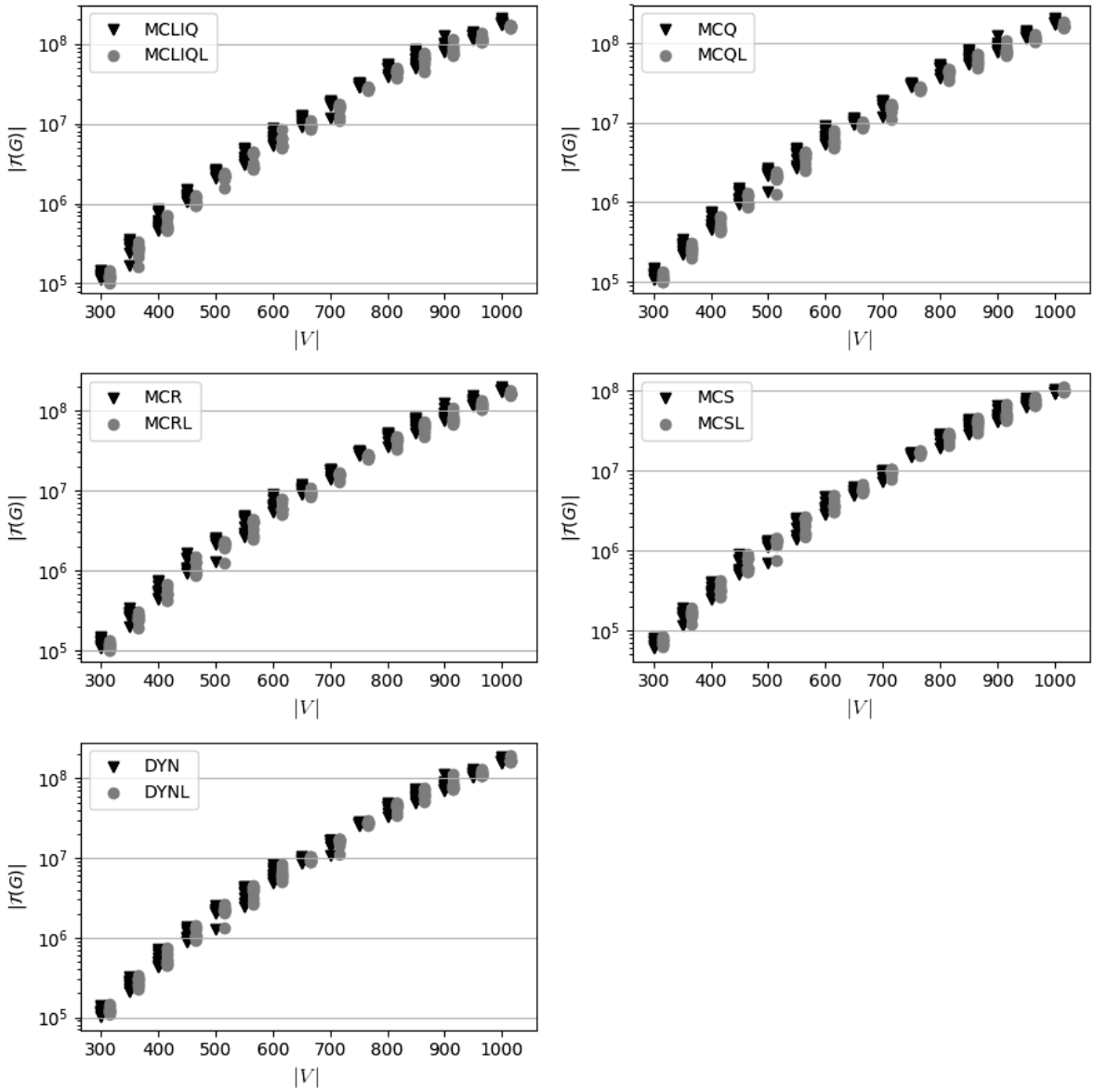
TABELA 11 – Teste de hipótese entre os algoritmos MCR e MCR_L em relação a média de estados gerados e tempo de execução.

		$\lvert\mathcal{T}\rvert$			Γ		
$\lvert V\rvert$	valor-p	$\overline{d}_{\mathcal{T}}$	IC($\mu_{D_{\mathcal{T}}}$, 95%)		valor-p	$\overline{d}_{\Gamma}$	IC($\mu_{D_{\Gamma}}$, 95%)	
300	$2{,}49\times10^{-6}$	8908,8	($7{,}0\times10^{3}$,	$1{,}1\times10^{4}$)	$1{,}23\times10^{-7}$	$-1{,}02$	($-1{,}18$,	$-0{,}86$)
350	$1{,}28\times10^{-6}$	22 236,8	($1{,}8\times10^{4}$,	$2{,}7\times10^{4}$)	$6{,}16\times10^{-9}$	$-2{,}67$	($-2{,}96$,	$-2{,}38$)
400	$3{,}00\times10^{-5}$	42 933,6	($3{,}0\times10^{4}$,	$5{,}6\times10^{4}$)	$8{,}28\times10^{-7}$	$-5{,}08$	($-6{,}05$,	$-4{,}11$)
450	$1{,}04\times10^{-4}$	103 350,6	($6{,}8\times10^{4}$,	$1{,}4\times10^{5}$)	$6{,}93\times10^{-7}$	$-12{,}28$	($-14{,}57$,	$-9{,}99$)
500	$3{,}10\times10^{-7}$	192 294,8	($1{,}6\times10^{5}$,	$2{,}2\times10^{5}$)	$3{,}55\times10^{-6}$	$-23{,}00$	($-28{,}19$,	$-17{,}80$)
550	$1{,}27\times10^{-5}$	358 712,0	($2{,}6\times10^{5}$,	$4{,}5\times10^{5}$)	$1{,}61\times10^{-6}$	$-40{,}60$	($-48{,}95$,	$-32{,}26$)
600	$5{,}27\times10^{-6}$	627 034,2	($4{,}8\times10^{5}$,	$7{,}8\times10^{5}$)	$9{,}63\times10^{-7}$	$-78{,}37$	($-93{,}54$,	$-63{,}20$)
650	$1{,}66\times10^{-8}$	915 155,0	($8{,}0\times10^{5}$,	$1{,}0\times10^{6}$)	$5{,}74\times10^{-7}$	$-108{,}77$	($-128{,}58$,	$-88{,}96$)
700	$1{,}59\times10^{-8}$	1 534 253,2	($1{,}3\times10^{6}$,	$1{,}7\times10^{6}$)	$1{,}89\times10^{-8}$	$-212{,}47$	($-238{,}59$,	$-186{,}35$)
750	$3{,}40\times10^{-12}$	2 952 316,2	($2{,}8\times10^{6}$,	$3{,}1\times10^{6}$)	$5{,}73\times10^{-7}$	$-334{,}34$	($-395{,}23$,	$-273{,}45$)
800	$2{,}09\times10^{-8}$	4 994 458,4	($4{,}4\times10^{6}$,	$5{,}6\times10^{6}$)	$4{,}24\times10^{-7}$	$-514{,}60$	($-605{,}07$,	$-424{,}13$)
850	$1{,}15\times10^{-7}$	7 440 597,6	($6{,}3\times10^{6}$,	$8{,}6\times10^{6}$)	$9{,}75\times10^{-7}$	$-859{,}87$	($-1026{,}52$,	$-693{,}22$)
900	$7{,}30\times10^{-7}$	9 277 507,2	($7{,}5\times10^{6}$,	$1{,}1\times10^{7}$)	$7{,}64\times10^{-9}$	$-1112{,}10$	($-1235{,}45$,	$-988{,}75$)
950	$7{,}30\times10^{-9}$	13 065 361,4	($1{,}2\times10^{7}$,	$1{,}5\times10^{7}$)	$1{,}34\times10^{-11}$	$-1374{,}42$	($-1449{,}20$,	$-1299{,}64$)
1000	$1{,}31\times10^{-11}$	18 230 854,6	($1{,}7\times10^{7}$,	$1{,}9\times10^{7}$)	$1{,}56\times10^{-9}$	$-2273{,}07$	($-2483{,}77$,	$-2062{,}38$)

TABELA 12 – Teste de hipótese entre os algoritmos MCS e MCS_L em relação a média de estados gerados e tempo de execução.

		$\lvert\mathcal{T}\rvert$		Γ			
$\lvert V\rvert$	valor-p	$\overline{d}_{\mathcal{T}}$	IC($\mu_{D_{\mathcal{T}}}$, 95%)	valor-p	$\overline{d}_{\Gamma}$	IC($\mu_{D_{\Gamma}}$, 95%)	
300	$1{,}45\times10^{-8}$	$-3070{,}0$	($-3{,}4\times10^{3}$, $-2{,}7\times10^{3}$)	$6{,}53\times10^{-7}$	$-0{,}83$	($-0{,}98$,	$-0{,}67$)
350	$3{,}47\times10^{-7}$	$-6070{,}8$	($-7{,}1\times10^{3}$, $-5{,}0\times10^{3}$)	$9{,}35\times10^{-9}$	$-2{,}07$	($-2{,}30$,	$-1{,}83$)
400	$3{,}28\times10^{-8}$	$-15\,964{,}8$	($-1{,}8\times10^{4}$, $-1{,}4\times10^{4}$)	$1{,}25\times10^{-8}$	$-4{,}85$	($-5{,}42$,	$-4{,}28$)
450	$2{,}88\times10^{-7}$	$-33\,878{,}6$	($-4{,}0\times10^{4}$, $-2{,}8\times10^{4}$)	$1{,}77\times10^{-6}$	$-9{,}16$	($-11{,}06$,	$-7{,}25$)
500	$4{,}92\times10^{-8}$	$-67\,911{,}8$	($-7{,}7\times10^{4}$, $-5{,}9\times10^{4}$)	$3{,}97\times10^{-8}$	$-20{,}31$	($-23{,}02$,	$-17{,}59$)
550	$1{,}64\times10^{-8}$	$-112\,153{,}8$	($-1{,}3\times10^{5}$, $-9{,}9\times10^{4}$)	$6{,}35\times10^{-7}$	$-32{,}03$	($-37{,}93$,	$-26{,}12$)
600	$2{,}94\times10^{-8}$	$-198\,271{,}2$	($-2{,}2\times10^{5}$, $-1{,}7\times10^{5}$)	$5{,}55\times10^{-8}$	$-64{,}00$	($-72{,}90$,	$-55{,}10$)
650	$1{,}15\times10^{-9}$	$-313\,474{,}4$	($-3{,}4\times10^{5}$, $-2{,}9\times10^{5}$)	$1{,}04\times10^{-7}$	$-95{,}41$	($-109{,}67$,	$-81{,}16$)
700	$2{,}57\times10^{-10}$	$-505\,330{,}4$	($-5{,}4\times10^{5}$, $-4{,}7\times10^{5}$)	$1{,}95\times10^{-9}$	$-169{,}67$	($-185{,}80$,	$-153{,}54$)
750	$3{,}67\times10^{-10}$	$-680\,605{,}8$	($-7{,}3\times10^{5}$, $-6{,}3\times10^{5}$)	$8{,}04\times10^{-10}$	$-296{,}82$	($-322{,}35$,	$-271{,}28$)
800	$1{,}74\times10^{-11}$	$-908\,165{,}2$	($-9{,}6\times10^{5}$, $-8{,}6\times10^{5}$)	$5{,}36\times10^{-8}$	$-499{,}15$	($-568{,}27$,	$-430{,}04$)
850	$1{,}03\times10^{-9}$	$-1\,416\,939{,}0$	($-1{,}5\times10^{6}$, $-1{,}3\times10^{6}$)	$1{,}22\times10^{-7}$	$-660{,}03$	($-760{,}49$,	$-559{,}56$)
900	$6{,}08\times10^{-9}$	$-1\,955\,873{,}0$	($-2{,}2\times10^{6}$, $-1{,}7\times10^{6}$)	$3{,}22\times10^{-8}$	$-895{,}88$	($-1012{,}89$,	$-778{,}87$)
950	$1{,}38\times10^{-9}$	$-3\,101\,365{,}2$	($-3{,}4\times10^{6}$, $-2{,}8\times10^{6}$)	$5{,}43\times10^{-12}$	$-1213{,}29$	($-1272{,}96$,	$-1153{,}62$)
1000	$2{,}05\times10^{-10}$	$-4\,287\,166{,}0$	($-4{,}6\times10^{6}$, $-4{,}0\times10^{6}$)	$1{,}52\times10^{-9}$	$-1857{,}87$	($-2029{,}58$,	$-1686{,}15$)

FIGURA 10 – Número de estados gerados para 150 grafos aleatórios $\mathcal{G}_{n,p}$ com $n = |V|$ vértices cada e $p = 1/2$. Existe uma amostra de tamanho 10 para cada número de $|V|$ vértices.



4.3.2.2 Resultados para Grafos Cordais e Cografos

Foram geradas 150 instâncias de grafos cordais aleatórios com número de vértices $|V| \in \{300, 350, \ldots, 1000\}$. Também foram geradas outras 150 instâncias de cografos aleatórios com $|V| \in \{300, 350, \ldots, 1000\}$ vértices cada. Não é de conhecimento desse autor algoritmos que geram instâncias de grafos cordais e cografos aleatórios uniformemente, como é foi caso dos grafos $\mathcal{G}_{n,1/2}$. Como se tratam de instâncias geradas com distribuição de probabilidade desconhecida não é possível utilizar o método estatístico de teste de hipótese para comparar o desempenho dos algoritmos. Também não foi feito nenhum teste de normalidade para grafos cordais ou cografos aleatórios.

Como anteriormente, o Algoritmo Greedy foi executado no conjunto de instâncias de grafos cordais e cografos. O número de cores $c(G, \pi)$ computado por Greedy utilizando como entrada uma ordenação π do conjunto de vértices estão presentes na Tabela 13. Também estão presentes o número de cores $c(G, \pi_L)$ computado por Greedy com π_L como entrada. A Tabela 13 mostra uma contagem de quantas vezes $c(G, \pi_L) < c(G, \pi)$ e quantas vezes $c(G, \pi_L) = c(G, \pi)$. É importante dizer que os resultados de coloração ótima obtidos tomando π_L como entrada estão de acordo a Observação 4 presente na Seção 3.3. Resumidamente, em 48 instâncias de grafos cordais utilizar a ordenação π_L como entrada de Greedy resultou em um número de cores menor que utilizar uma ordenação π. Novamente, como esperado, em nenhuma dessas instâncias de grafos cordais utilizar π_L resultou em um número de cores maior que utilizar π. Em 102 instâncias o número de cores foi o mesmo para as duas ordenações. Os resultados para todas instâncias podem ser consultados na Tabela 16, Apêndice C.

TABELA 13 – Teste com Greedy(G) onde G é cordal. Como esperado (Observação 4), em nenhuma teste o número $c(G, \pi_L) > c(G, \pi)$ para toda amostra. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ grafos cordais com $|V|$ vértices cada.

$\|V\|$	$\mu_{\|E\|}$	$c(G, \pi_L) < c(G, \pi)$	$c(G, \pi_L) = c(G, \pi)$	$c(G, \pi_L) > c(G, \pi)$
300	20 297	2	8	0
350	25 174	4	6	0
400	30 758	3	7	0
450	35 196	5	5	0
500	42 093	2	8	0
550	47 087	5	5	0
600	53 072	2	8	0
650	60 275	3	7	0
700	67 065	3	7	0
750	74 477	4	6	0
800	80 611	3	7	0
850	92 220	2	8	0
900	97 171	3	7	0
950	100 861	4	6	0
1000	108 677	3	7	0

Como estabelece o Teorema 12, no caso das instâncias de cografos $c(G, \pi_L) = c(G, \pi)$ em todos testes feitos. A Tabela 14 apresenta, resumidamente, os resultados obtidos. Os resultados para todas instâncias podem ser consultados na Tabela 17, presente no Apêndice C.

Em seguida, de maneira sumarizada, as figuras 11 e 12 apresentam a dispersão do número de estados gerados para cada algoritmo de $\mathcal{A}$ em comparação com $\mathcal{A}_L$. Considerando grafos cordais aleatórios como entrada, na comparação entre os algoritmo de $\mathcal{A}$ e $\mathcal{A}_L$, existe uma coincidência no número de estados criados. Com exceção de uma instância de

TABELA 14 – Teste com $\mathsf{Greedy}(G)$ onde G é cografo. Como esperado (Teorema 12), em nenhuma teste com Greedy o número $c(G,\pi_L) > c(G,\pi)$ para toda amostra. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ cografos com $|V|$ vértices cada.

$\lvert V\rvert$	$\mu_{\lvert E\rvert}$	$c(G,\pi_L) < c(G,\pi)$	$c(G,\pi_L) = c(G,\pi)$	$c(G,\pi_L) > c(G,\pi)$
300	315 291	0	10	0
350	374 766	0	10	0
400	683 733	0	10	0
450	596 269	0	10	0
500	840 196	0	10	0
550	1 031 854	0	10	0
600	742 751	0	10	0
650	1 405 181	0	10	0
700	1 851 159	0	10	0
750	2 029 593	0	10	0
800	1 614 894	0	10	0
850	2 642 600	0	10	0
900	2 339 840	0	10	0
950	3 309 198	0	10	0
1000	4 254 682	0	10	0

grafo cordal com número de vértices $|V| = 950$ e número de arestas $|E| = 10968$ onde $|\mathcal{T}_{\mathsf{MCLIQ}}(G)| - |\mathcal{T}_{\mathsf{MCLIQ}_L}(G)| = 50$, o resultado de $|\mathcal{T}_{i,\mathcal{A}}(G)| - |\mathcal{T}_{i,\mathcal{A}_L}(G)|$ é igual a 0 para toda observação $i \in \{1, 2, \ldots, 150\}$ dos grafos cordais.

Considere o tempo de execução em CPU medido durante a execução dos algoritmos de $\mathcal{A}$ e $\mathcal{A}_L$ com instâncias de grafos cordais aleatórios como entrada. O Algoritmo MCLIQ_L teve um tempo menor para todas 150 instâncias quando comparado ao Algoritmo MCLIQ. O Algoritmo MCQ_L teve um tempo menor para 22 instâncias quando comparado ao Algoritmo MCQ. O Algoritmo DYN_L teve um tempo menor para 30 instâncias quando comparado ao Algoritmo DYN. O Algoritmo MCR_L teve um tempo menor para 37 instâncias quando comparado ao Algoritmo MCR. O Algoritmo MCS_L teve um tempo menor para 62 instâncias quando comparado ao Algoritmo MCS.

No caso das instâncias de cografos a subtração $|\mathcal{T}_{i,\mathcal{A}}(G)| - |\mathcal{T}_{i,\mathcal{A}_L}(G)|$ resultou em 0 para todas $i \in \{1, 2, \ldots, 150\}$ observações. Ainda na Figura 12, com relação ao desempenho do Algoritmo MCR executado com instâncias de cografos aleatórios, esse algoritmo foi capaz de gerar uma árvore com um único estado, consequentemente os algoritmos MCR_L, MCS e MCS_L também tiverem o mesmo comportamento.

Ao considerar a o tempo de execução em CPU medido durante a execução dos algoritmos de $\mathcal{A}$ e $\mathcal{A}_L$ com instâncias de cografos aleatórios como entrada em 90 instâncias MCR_L foi tomou tempo de execução menor quando comparado com o Algoritmo MCR. O Algoritmo MCS_L também teve tempo de execução menor que MCS em 67 instâncias.

FIGURA 11 – Número de estados gerados por cada algoritmo $\mathcal{A}$ e $\mathcal{A}_L$ utilizando como entrada grafos cordais aleatórios. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ grafos cordais com $|V|$ vértices cada. Com exceção de um único grafo G, $|\mathcal{T}_{\mathcal{A}}(G)| = |\mathcal{T}_{\mathcal{A}_L}(G)|$ para todos os outros.

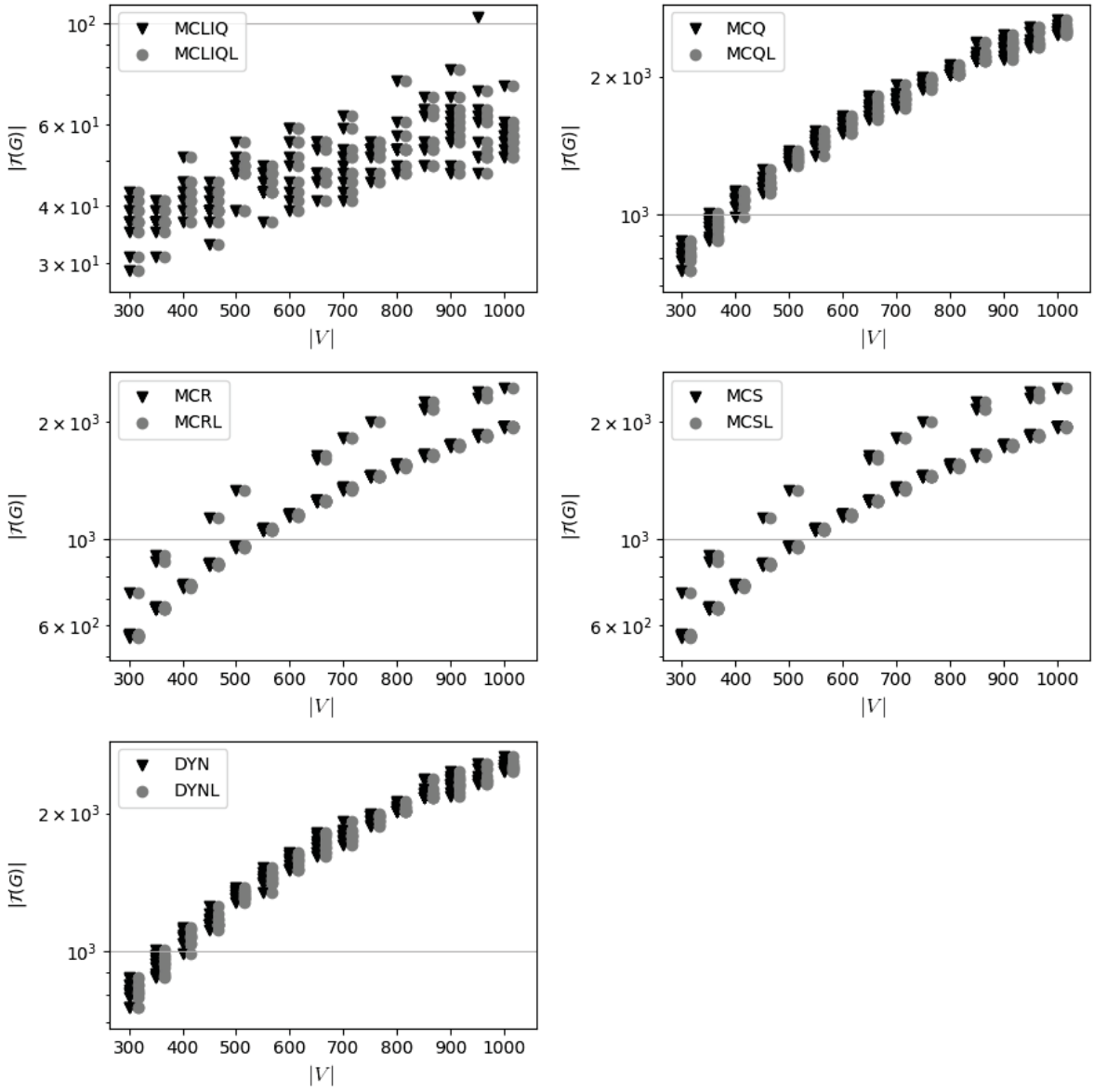
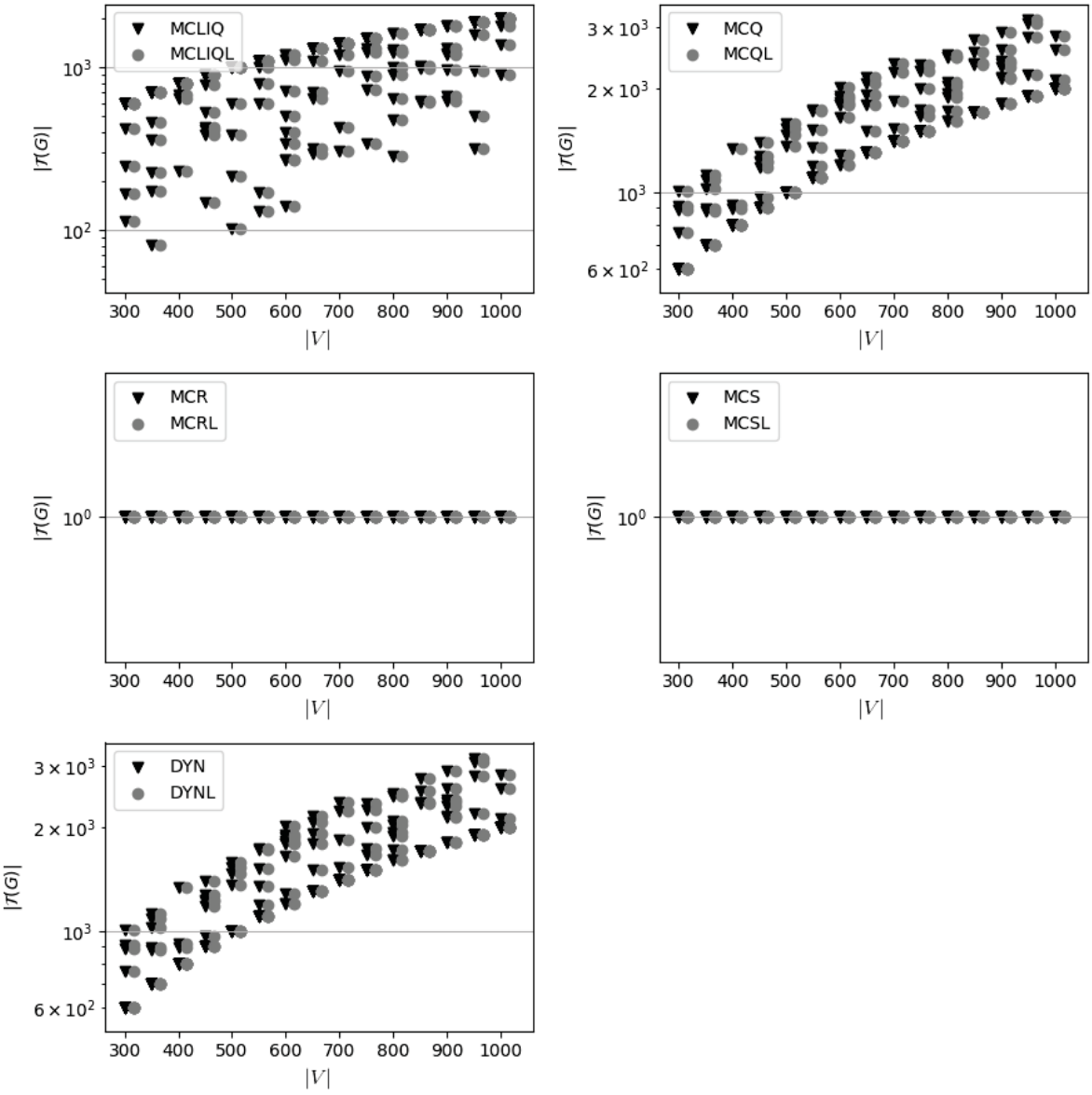


FIGURA 12 – Número de estados gerados por cada algoritmo $\mathcal{A}$ e $\mathcal{A}_L$ utilizando como entrada cografos aleatórios. Para cada $|V| \in \{300, 350, \ldots, 1000\}$, existem amostras de tamanho $n = 10$ cografos com $|V|$ vértices cada. Para todo grafo $|\mathcal{T}_{\mathcal{A}}(G)| = |\mathcal{T}_{\mathcal{A}_L}(G)|$.



5 CONCLUSÃO

O caráter fundamental do CM motivou o estudo de algoritmos que resolvem o problema bem como entender sua relação com outros problemas igualmente fundamentais. Estratégias de Projeto de Algoritmos como a técnica de *branch and bound* ganharam destaque na solução exata para o CM. Nesse processo, algoritmos para o problema de coloração de vértices se mostram uma ferramenta importante na construção de algoritmos cada vez mais eficientes para o CM.

Em classes de grafos como a classe dos grafos cordais e cografos, o Algoritmo LexBFS pode ser empregado tanto como um passo no reconhecimento dessas classes como no passo para resolver o CM e coloração de vértices em tempo polinomial. O objetivo desse trabalho foi estudar uma modificação com o Algoritmo LexBFS dentro de algoritmos BB que utilizam coloração de vértices na solução exata do CM, criando uma versão modificada para cada algoritmo estudado, emergindo um conjunto $\mathcal{A}_L$ de algoritmos. As versões originais são agrupadas no conjunto $\mathcal{A}$ de algoritmos. Essa proposta foi implementada e testada com o objetivo de comparar as diferentes versões. Visando uma comparação o mais justa possível, foram empregados conceitos da Análise Experimental de Algoritmos. Nesse sentido, a utilização do Algoritmo MCBB como um *framework* possibilitou descrever e implementar todos algoritmos sob um contexto unificado.

A condução da análise experimental dos algoritmos propostos foi feita utilizando diferentes instâncias de grafos. Foram feitos testes considerando instâncias de grafos DIMACS, instâncias de grafos aleatórios do modelo $\mathcal{G}_{n,p}$ e instâncias das classes de grafos cordais e cografos. Quando os resultados são analisados com relação ao número de estados gerados por $\mathcal{A}_L$, é possível observar um desempenho melhor em comparação com $\mathcal{A}$. Por outro, lado o tempo de execução em CPU de $\mathcal{A}_L$ nem sempre foi menor na comparação. Ao realizar um teste de hipótese utilizando grafos aleatórios $\mathcal{G}_{n,1/2}$, constatou-se que para um algoritmo $\mathcal{A}_L$ existe diferença significativa em comparação com $\mathcal{A}$, tanto em relação ao número de estados da árvore de estados quanto ao tempo de execução em CPU. Por esse método estatístico, os algoritmos modificados DYN_L e MCS_L geraram árvores de estados significativamente maiores que as versões originais DYN e MCS. Nos outros 3 algoritmos modificados e analisados as árvores de estados foram menores significativamente. Além disso, em todas testes de hipótese o tempo de execução foi significativamente maior para $\mathcal{A}_L$.

Para concluir, as implementações dos algoritmos do Capítulo 4, resultados brutos e resultados tratados apresentados no decorrer desse texto, estão disponíveis em repositório de acesso público da internet.

REFERÊNCIAS

ANDERSON, Theodore W; DARLING, Donald A. Asymptotic theory of certain"goodness of fit"criteria based on stochastic processes. **The annals of mathematical statistics**, JSTOR, p. 193–212, 1952. Citado 1 vez na página 88.

ANJOS, Cleverson Sebastião dos. **Análise experimental de algoritmos**. 2015. Diss. (Mestrado) – Universidade Federal do Paraná. Citado 2 vezes nas páginas 52, 57.

ANJOS, Cleverson Sebastião dos; ZÜGE, Alexandre Prusch; CARMO, Renato. An experimental analysis of exact algorithms for the maximum clique problem. **Matemática Contemporânea**, v. 44, p. 1–20, 2016. Citado 2 vezes nas páginas 35, 57.

BALASUNDARAM, Balabhaskar; BUTENKO, Sergiy. Graph domination, coloring and cliques in telecommunications. In: HANDBOOK of Optimization in Telecommunications. [S.l.]: Springer, 2006. p. 865–890. Citado 1 vez na página 34.

BEHNEL, S.; BRADSHAW, R.; CITRO, C.; DALCIN, L.; SELJEBOTN, D.S.; SMITH, K. Cython: The Best of Both Worlds. **Computing in Science Engineering**, v. 13, n. 2, p. 31–39, mar. 2011. ISSN 1521-9615. DOI: 10.1109/MCSE.2010.118. Citado 1 vez na página 114.

BERGE, Claude; CHVÁTAL, Vašek. **Topics on perfect graphs.** [S.l.]: Elsevier Science, 1984. (North-Holland Mathematics Studies). ISBN 9780080871998. : https://books.google.com.br/books?id=m34PYru8s7MC. Citado 1 vez na página 42.

BERRY, Anne; POGORELCNIK, Romain. A simple algorithm to generate the minimal separators and the maximal cliques of a chordal graph. **Information Processing Letters**, Elsevier, v. 111, n. 11, p. 508–511, 2011. Citado 1 vez na página 18.

BOGINSKI, Vladimir; BUTENKO, Sergiy; PARDALOS, Panos M. Mining market data: a network approach. **Computers & Operations Research**, Elsevier, v. 33, n. 11, p. 3171–3184, 2006. Citado 1 vez na página 34.

BOLLOBÁS, Béla. **Random graphs**. [S.l.]: Cambridge university press, 2001. Citado 1 vez na página 57.

BOMZE, Immanuel M; BUDINICH, Marco; PARDALOS, Panos M; PELILLO, Marcello. The maximum clique problem. In: HANDBOOK of combinatorial optimization. [S.l.]: Springer, 1999. p. 1–74. Citado 1 vez na página 34.

BONDY, JA; MURTY, USR. Graph theory (2008). **Grad. Texts in Math**, 2008. Citado 3 vezes nas páginas 40, 42.

BRANDSTADT, Andreas; SPINRAD, Jeremy P et al. **Graph classes: a survey**. [S.l.]: Siam, 1999. v. 3. Citado 5 vezes nas páginas 13, 16, 17.

BRÉLAZ, Daniel. New methods to color the vertices of a graph. **Communications of the ACM**, ACM, v. 22, n. 4, p. 251–256, 1979. Citado 1 vez na página 43.

BRETSCHER, Anna; CORNEIL, Derek; HABIB, Michel; PAUL, Christophe. A simple linear time LexBFS cograph recognition algorithm. In: SPRINGER. INTERNATIONAL Workshop on Graph-Theoretic Concepts in Computer Science. [S.l.: s.n.], 2003. p. 119–130. Citado 6 vezes nas páginas 20, 24, 28–30, 32, 108.

BRETSCHER, Anna; CORNEIL, Derek; HABIB, Michel; PAUL, Christophe. A simple linear time LexBFS cograph recognition algorithm. **SIAM Journal on Discrete Mathematics**, SIAM, v. 22, n. 4, p. 1277–1296, 2008. Citado 4 vez na página 32.

BRON, Coen; KERBOSCH, Joep. Algorithm 457: finding all cliques of an undirected graph. **Communications of the ACM**, ACM New York, NY, USA, v. 16, n. 9, p. 575–577, 1973. Citado 1 vez na página 35.

BRYANT, Randal E; DAVID RICHARD, O'Hallaron; DAVID RICHARD, O'Hallaron. **Computer systems: a programmer's perspective**. [S.l.]: Prentice Hall Upper Saddle River, 2003. v. 2. Citado 1 vez na página 57.

BUSSAB, Wilton de Oliveira; MORETTIN, Pedro Alberto. Estatística básica, 2009. Citado 1 vez na página 52.

CARMO, Renato; ZÜGE, Alexandre Prusch. Branch and bound algorithms for the maximum clique problem under a unified framework. **Journal of the Brazilian Computer Society**, Springer, v. 18, n. 2, p. 137–151, 2012. Citado 4 vezes nas páginas 35, 36, 40, 57.

CARRAGHAN, Randy; PARDALOS, Panos M. An exact algorithm for the maximum clique problem. **Operations Research Letters**, Elsevier, v. 9, n. 6, p. 375–382, 1990. Citado 5 vezes nas páginas 34, 35, 40.

CORNEIL, Derek G. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. **Discrete Applied Mathematics**, Elsevier, v. 138, n. 3, p. 371–379, 2004. Citado 3 vezes nas páginas 18, 27.

CORNEIL, Derek G. Lexicographic breadth first search–a survey. In: SPRINGER. INTERNATIONAL Workshop on Graph-Theoretic Concepts in Computer Science. [S.l.: s.n.], 2004. p. 1–19. Citado 4 vezes nas páginas 18, 21, 108.

CORNEIL, Derek G; LERCHS, Helmut; BURLINGHAM, L Stewart. Complement reducible graphs. **Discrete Applied Mathematics**, Elsevier, v. 3, n. 3, p. 163–174, 1981. Citado 1 vez na página 19.

CRAMÉR, Harald. On the composition of elementary errors: First paper: Mathematical deductions. **Scandinavian Actuarial Journal**, Taylor & Francis, v. 1928, n. 1, p. 13–74, 1928. Citado 1 vez na página 88.

DEVORE, Jay L. **Probability and Statistics for Engineering and the Sciences**. [S.l.]: Cengage Learning, 2015. Citado 1 vez na página 52.

FAHLE, Torsten. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In: SPRINGER. EUROPEAN Symposium on Algorithms. [S.l.: s.n.], 2002. p. 485–498. Citado 7 vezes nas páginas 13, 35, 40, 44, 45, 57.

GALINIER, Philippe; HAMIEZ, Jean-Philippe; HAO, Jin-Kao; PORUMBEL, Daniel. Recent advances in graph vertex coloring. In: HANDBOOK of optimization. [S.l.]: Springer, 2013. p. 505–528. Citado 1 vez na página 40.

GAREY, Michael R; JOHNSON, David S. **Computers and intractability**. [S.l.]: wh freeman New York, 2002. v. 29. Citado 2 vezes nas páginas 13, 41.

GAVRIL, Fănică. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. **SIAM Journal on Computing**, SIAM, v. 1, n. 2, p. 180–187, 1972. Citado 1 vez na página 13.

GAVRIL, Fănică. The intersection graphs of subtrees in trees are exactly the chordal graphs. **Journal of Combinatorial Theory, Series B**, Elsevier, v. 16, n. 1, p. 47–56, 1974. Citado 1 vez na página 18.

GOLUMBIC, Martin Charles. **Algorithmic graph theory and perfect graphs**. [S.l.]: Elsevier, 2004. Citado 1 vez na página 18.

GRÖTSCHEL, Martin; LOVÁSZ, László; SCHRIJVER, Alexander. **Geometric algorithms and combinatorial optimization**. [S.l.]: Springer Science & Business Media, 2012. v. 2. Citado 1 vez na página 17.

HABIB, Michel; MCCONNELL, Ross; PAUL, Christophe; VIENNOT, Laurent. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. **Theoretical Computer Science**, Elsevier, v. 234, n. 1-2, p. 59–84, 2000. Citado 4 vezes nas páginas 17, 18, 25, 108.

HABIB, Michel; PAUL, Christophe. A survey of the algorithmic aspects of modular decomposition. **Computer Science Review**, Elsevier, v. 4, n. 1, p. 41–59, 2010. Citado 0 vez na página 24.

HOÀNG, Chính T; MAHADEV, Nadimpalli VR. A note on perfect orders. **Discrete mathematics**, Elsevier, v. 74, n. 1-2, p. 77–84, 1989. Citado 1 vez na página 43.

HUNTER, J. D. Matplotlib: A 2D Graphics Environment. **Computing in Science Engineering**, v. 9, n. 3, p. 90–95, 2007. Citado 1 vez na página 116.

ISO, ISO. IEC 14882: 2011 Information technology—Programming languages—C++. **International Organization for Standardization, Geneva, Switzerland**, v. 27, p. 59, 2012. Citado 1 vez na página 114.

JOHNSON, David S; TRICK, Michael A. **Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993**. [S.l.]: American Mathematical Soc., 1996. v. 26. Citado 2 vez na página 57.

KARP, Richard M. Reducibility among combinatorial problems. In: COMPLEXITY of computer computations. [S.l.]: Springer, 1972. p. 85–103. Citado 2 vezes nas páginas 34, 41.

KONC, Janez; JANEZIC, Dušanka. An improved branch and bound algorithm for the maximum clique problem. **proteins**, Citeseer, v. 4, n. 5, 2007. Citado 6 vezes nas páginas 13, 35, 46, 57, 58.

LAVNIKEVICH, Nikolay. On the complexity of maximum clique algorithms: usage of coloring heuristics leads to the $\Omega(2^{n/5})$ algorithm running time lower bound. **URL http://arxiv. org/abs/1303.2546**, 2013. Citado 1 vez na página 44.

LEKKEIKERKER, C; BOLAND, J. Representation of a finite graph by a set of intervals on the real line. **Fundamenta Mathematicae**, v. 51, n. 1, p. 45–64, 1962. Citado 1 vez na página 18.

LEWIS, R.M.R. **A Guide to Graph Colouring: Algorithms and Applications**. [S.l.]: Springer International Publishing, 2015. ISBN 9783319257280. : https://books.google.com.br/books?id=fgcjjgEACAAJ. Citado 1 vez na página 40.

LOVE, Robert. **Linux kernel development**. [S.l.]: Pearson Education, 2010. Citado 1 vez na página 114.

LUND, Carsten; YANNAKAKIS, Mihalis. On the hardness of approximating minimization problems. **Journal of the ACM (JACM)**, ACM, v. 41, n. 5, p. 960–981, 1994. Citado 1 vez na página 41.

MAFFRAY, Frédéric. On the coloration of perfect graphs. In: RECENT Advances in Algorithms and Combinatorics. [S.l.]: Springer, 2003. p. 65–84. Citado 1 vez na página 42.

MALAGUTI, Enrico; TOTH, Paolo. A survey on vertex coloring problems. **International transactions in operational research**, Wiley Online Library, v. 17, n. 1, p. 1–34, 2010. Citado 1 vez na página 40.

MALOD-DOGNIN, Noël; ANDONOV, Rumen; YANEV, Nicola. Maximum cliques in protein structure comparison. In: SPRINGER. INTERNATIONAL Symposium on Experimental Algorithms. [S.l.: s.n.], 2010. p. 106–117. Citado 1 vez na página 34.

MCGEOCH, Catherine C. **A guide to experimental algorithmics**. [S.l.]: Cambridge University Press, 2012. Citado 3 vezes nas páginas 51, 57.

MINT, Linux. Linux mint. **URL http://www.linuxmint.com**, 2010. Citado 1 vez na página 114.

MOON, John W; MOSER, Leo. On cliques in graphs. **Israel journal of Mathematics**, Springer, v. 3, n. 1, p. 23–28, 1965. Citado 2 vezes nas páginas 19, 34.

OLIPHANT, Travis E. **A guide to NumPy**. [S.l.]: Trelgol Publishing USA, 2006. v. 1. Citado 1 vez na página 116.

PARDALOS, Panos M; XUE, Jue. The maximum clique problem. **Journal of global Optimization**, Springer, v. 4, n. 3, p. 301–328, 1994. Citado 1 vez na página 34.

PATTILLO, Jeffrey; YOUSSEF, Nataly; BUTENKO, Sergiy. Clique relaxation models in social network analysis. In: HANDBOOK of Optimization in Complex Networks. [S.l.]: Springer, 2012. p. 143–162. Citado 1 vez na página 34.

PRÉVÔT, Claudia; RÖCKNER, Michael. **A concise course on stochastic partial differential equations**. [S.l.]: Springer, 2007. v. 1905. Citado 1 vez na página 52.

PROSSER, Patrick. Exact algorithms for maximum clique: A computational study. **Algorithms**, Multidisciplinary Digital Publishing Institute, v. 5, n. 4, p. 545–587, 2012. Citado 1 vez na página 35.

ROBSON, John M. **Finding a maximum independent set in time O (2n/4)**. [S.l.], 2001. Citado 1 vez na página 34.

ROSE, Donald J; TARJAN, R Endre; LUEKER, George S. Algorithmic aspects of vertex elimination on graphs. **SIAM Journal on computing**, SIAM, v. 5, n. 2, p. 266–283, 1976. Citado 4 vezes nas páginas 17, 21, 25.

THE SAGE DEVELOPERS. **SageMath, the Sage Mathematics Software System (Version 7.1.0)**. [S.l.], 2016. https://www.sagemath.org. Citado 2 vezes nas páginas 57, 114.

SAN SEGUNDO, Pablo; CONIGLIO, Stefano; FURINI, Fabio; LJUBIĆ, Ivana. A new branch-and-bound algorithm for the maximum edge-weighted clique problem. **European Journal of Operational Research**, Elsevier, 2019. Citado 1 vez na página 35.

SAN SEGUNDO, Pablo; FURINI, Fabio; ARTIEDA, Jorge. A new branch-and-bound algorithm for the Maximum Weighted Clique Problem. **Computers & Operations Research**, Elsevier, v. 110, p. 18–33, 2019. Citado 1 vez na página 35.

SAN SEGUNDO, Pablo; LOPEZ, Alvaro; BATSYN, Mikhail; NIKOLAEV, Alexey; PARDALOS, Panos M. Improved initial vertex ordering for exact maximum clique search. **Applied Intelligence**, Springer, v. 45, n. 3, p. 868–880, 2016. Citado 1 vez na página 44.

SCHAEFFER, Satu Elisa. Graph clustering. **Computer science review**, Elsevier, v. 1, n. 1, p. 27–64, 2007. Citado 1 vez na página 34.

SEABOLD, Skipper; PERKTOLD, Josef. statsmodels: Econometric and statistical modeling with python. In: 9TH Python in Science Conference. [S.l.: s.n.], 2010. Citado 1 vez na página 116.

ŞEKER, Oylum; HEGGERNES, Pinar; EKIM, Tınaz; TAŞKIN, Z Caner. Generation of random chordal graphs using subtrees of a tree. **arXiv preprint arXiv:1810.13326**, 2018. Citado 1 vez na página 58.

SHAPIRO, Samuel Sanford; WILK, Martin B. An analysis of variance test for normality (complete samples). **Biometrika**, JSTOR, v. 52, n. 3/4, p. 591–611, 1965. Citado 1 vez na página 88.

SHIRYAYEV, AN. 15. On The Empirical Determination of A Distribution Law. In: SELECTED Works of AN Kolmogorov. [S.l.]: Springer, 1992. p. 139–146. Citado 1 vez na página 88.

STALLMAN, R. **GCC Developer Community. Using the GNU Compiler Collection for GCC version 7.4.0**. [S.l.: s.n.], 2017. : https://gcc.gnu.org/onlinedocs/gcc-7.4.0/gcc.pdf. Citado 1 vez na página 114.

STROOCK, Daniel W. **Probability theory: an analytic view**. [S.l.]: Cambridge university press, 2010. Citado 1 vez na página 52.

THE GNOME PROJECT. **The Gnumeric Spreadsheet: Free, Fast, Accurate — pick any three**. Versão 1.12.35. 2017. : http://www.gnumeric.org. Acesso em: 19 mai. 2019. Citado 1 vez na página 115.

THE PANDAS DEVELOPMENT TEAM. **pandas-dev/pandas: Pandas**. [S.l.]: Zenodo, fev. 2020. DOI: 10.5281/zenodo.3509134. : https://doi.org/10.5281/zenodo.3509134. Citado 1 vez na página 116.

TOMITA, Etsuji; KAMEDA, Toshikatsu. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. **Journal of Global optimization**, Springer, v. 37, n. 1, p. 95–111, 2007. Citado 7 vezes nas páginas 13, 34–36, 47, 57, 58.

TOMITA, Etsuji; KOHATA, Yasuhiro; TAKAHASHI, Haruhisa. **A Simple Algorithm for Finding a Maximum Clique**. [S.l.: s.n.], 1988. Citado 2 vezes nas páginas 45, 58.

TOMITA, Etsuji; SEKI, Tomokazu. An efficient branch-and-bound algorithm for finding a maximum clique. In: SPRINGER. INTERNATIONAL Conference on Discrete Mathematics and Theoretical Computer Science. [S.l.: s.n.], 2003. p. 278–289. Citado 7 vezes nas páginas 13, 34, 35, 46, 57, 58.

TOMITA, Etsuji; SUTANI, Yoichi; HIGASHI, Takanori; TAKAHASHI, Shinya; WAKATSUKI, Mitsuo. A simple and faster branch-and-bound algorithm for finding a maximum clique. In: SPRINGER. INTERNATIONAL Workshop on Algorithms and Computation. [S.l.: s.n.], 2010. p. 191–203. Citado 6 vezes nas páginas 13, 35, 36, 47, 57, 58.

TOMITA, Etsuji; TANAKA, Akira; TAKAHASHI, Haruhisa. The worst-case time complexity for generating all maximal cliques and computational experiments. **Theoretical computer science**, Elsevier, v. 363, n. 1, p. 28–42, 2006. Citado 1 vez na página 34.

UTKINA, Irina. Using modular decomposition technique to solve the maximum clique problem. In: SPRINGER. INTERNATIONAL Conference on Network Analysis. [S.l.: s.n.], 2016. p. 121–131. Citado 1 vez na página 58.

VAN ROSSUM, Guido et al. Python Programming Language. In: USENIX annual technical conference. [S.l.: s.n.], 2007. v. 41, p. 36. Citado 1 vez na página 114.

VIRTANEN, Pauli; GOMMERS, Ralf; OLIPHANT, Travis E.; HABERLAND, Matt; REDDY, Tyler; COURNAPEAU, David; BUROVSKI, Evgeni; PETERSON, Pearu; WECKESSER, Warren; BRIGHT, Jonathan; VAN DER WALT, Stéfan J.; BRETT, Matthew; WILSON, Joshua; JARROD MILLMAN, K.; MAYOROV, Nikolay; NELSON, Andrew R. J.; JONES, Eric; KERN, Robert; LARSON, Eric; CAREY, CJ; POLAT, İlhan; FENG, Yu; MOORE, Eric W.; VAND ERPLAS, Jake; LAXALDE, Denis; PERKTOLD, Josef; CIMRMAN, Robert; HENRIKSEN, Ian; QUINTERO, E. A.; HARRIS, Charles R; ARCHIBALD, Anne M.; RIBEIRO, Antônio H.; PEDREGOSA, Fabian; VAN MULBREGT, Paul; CONTRIBUTORS, SciPy 1. 0. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. **Nature Methods**, v. 17, p. 261–272, 2020. DOI: https://doi.org/10.1038/s41592-019-0686-2. Citado 1 vez na página 116.

WELSH, Dominic JA; POWELL, Martin B. An upper bound for the chromatic number of a graph and its application to timetabling problems. **The Computer Journal**, Oxford University Press, v. 10, n. 1, p. 85–86, 1967. Citado 1 vez na página 41.

WU, Qinghua; HAO, Jin-Kao. A review on algorithms for maximum clique problems. **European Journal of Operational Research**, Elsevier, v. 242, n. 3, p. 693–709, 2015. Citado 2 vez na página 35.

WU, Qinghua; HAO, Jin-Kao. Coloring large graphs based on independent set extraction. **Computers & Operations Research**, Elsevier, v. 39, n. 2, p. 283–290, 2012. Citado 1 vez na página 34.

ZUCKERMAN, David. Linear degree extractors and the inapproximability of max clique and chromatic number. In: ACM. PROCEEDINGS of the thirty-eighth annual ACM symposium on Theory of computing. [S.l.: s.n.], 2006. p. 681–690. Citado 2 vezes nas páginas 34, 41.

ZÜGE, Alexandre Prusch. **Algoritmos para o problema da clique máxima: análise e comparação experimental**. 2017. Tese (Doutorado) – Universidade Federal do Paraná. Citado 4 vezes nas páginas 44, 57, 114.

ZÜGE, Alexandre Prusch. **Solução Exata do Problema da Clique Máxima**. 2012. Diss. (Mestrado) – Universidade Federal do Paraná. Citado 1 vez na página 35.

ZÜGE, Alexandre Prusch; CARMO, Renato. On comparing algorithms for the maximum clique problem. **Discrete Applied Mathematics**, Elsevier, v. 247, p. 1–13, 2018. Citado 2 vez na página 35.

APÊNDICES

APÊNDICE A – IMPLEMENTAÇÃO DO ALGORITMO LEXBFS

Nesse apêndice é apresentado o Algoritmo LexBFS(G) descrito em Habib, McConnell et al. (2000), Bretscher et al. (2003) e Corneil (2004b) implementado em linguagem de programação C++ utilizando a *Standard Template Library* (STL). A versão C++11 foi escolhida. Essa implementação utiliza a técnica de particionamento, como descrita no Capítulo 2.

Listing A.1 – Algoritmo LexBFS

```
1  #ifndef ITEM_H
2  #define ITEM_H
3
4  class Part; // Forward declaration
5
6  class Item {
7  public:
8      int v;
9      Item(int _v);
10     Part* get_part() const;
11     void set_part(Part *_part);
12 private:
13     Part *part;
14 };
15
16 Item::Item(int _v) : v(_v) {}
17 Part* Item::get_part() const {return part;}
18 void Item::set_part(Part *_part) {part = _part;}
19
20 #endif
21
22 #include <list>
23
24 #ifndef PART_H
25 #define PART_H
26
27 using namespace std;
28
29 class Item;
30
```

```
31 class Part {
32 public:
33     Part(list<Item> *_set);
34     list<Item>::iterator begin() const;
35     list<Item>::iterator end() const;
36     size_t size() const;
37     bool empty() const;
38     list<Item>::iterator pop_front();
39     list<Item>::iterator pop_back();
40     void resize(size_t n);
41     void push_back(const list<Item>::iterator it);
42     void set_elements(const list<Item>::iterator _first,
43                       const list<Item>::iterator _last);
44 private:
45     list<Item> *set;
46     list<Item>::iterator first;
47     list<Item>::iterator last;
48     size_t number_of_elements;
49 };
50
51 Part::Part(list<Item> *_set) {
52     set = _set;
53     number_of_elements = 0;
54 };
55 // Returns iterator to begin of the part.
56 list<Item>::iterator Part::begin() const {return first;}
57
58 // Returns iterator to end of the part.
59 list<Item>::iterator Part::end() const {return last;}
60
61 // Returns the current number of elements of the part.
62 size_t Part::size() const {return number_of_elements;}
63
64 // Returns true if the current number of elements is 0, false, otherwise.
65 bool Part::empty() const {
66     if (size() <= 0)
67         return true;
68     return false;
69 }
```

```
70
71  // Removes the first element in the part, effectively reducing its size by
72  // one.
73  list<Item>::iterator Part::pop_front() {
74      if (empty())
75          throw out_of_range("trying to pop_front() but part is empty");
76
77      list<Item>::iterator it = first;
78      ++first;
79      it->set_part(NULL);
80      number_of_elements -= 1;
81
82      return it;
83  }
84
85  // Removes the last element in the part, effectively reducing its size by
86  // one.
87  list<Item>::iterator Part::pop_back() {
88      if (empty())
89          throw out_of_range("trying to pop_back() but part is empty");
90
91      list<Item>::iterator it = last;
92      --last;
93      it->set_part(NULL);
94      number_of_elements -= 1;
95
96      return it;
97  }
98
99  // Effectively update the number of elements of the part.
100 void Part::resize(size_t n) {number_of_elements = n;}
101
102 // Add a new element from the set at the end of the part, and resize the
103 // part. Notice that a part form the interval [first, last] in the set
104 void Part::push_back(const list<Item>::iterator it) {
105     if (set->empty())
106         throw out_of_range("trying to push_back() but the set is \
107 empty");
108
```

```
109     last = it;
110     last->set_part(this);
111 
112     if (last == set->end())
113         throw out_of_range("trying to push_back() the set::end, which \
114 returns an iterator referring to the past-the-end element in the list \
115 container");
116 
117     if (empty())
118         first = last;
119 
120     number_of_elements += 1;
121 }
122 
123 // Setting elements in the rage [first, last] from the set, and resize the
124 // part.
125 void Part::set_elements(const list<Item>::iterator _first,
126                         const list<Item>::iterator _last) {
127     if (set->empty())
128         throw out_of_range("trying to set_elements() but the set is \
129 empty");
130 
131     first = _first;
132     last = _last;
133 
134     int d = distance(first,last);
135 
136     if (d == 0 && set->size() >= 1)
137         d = 1;
138 
139     resize(d);
140 
141     if (last == set->end())
142         --last;
143 
144     list<Item>::iterator it = first;
145     while (d--) {
146         it->set_part(this);
147         ++it;
```

```
148     }
149 }
150
151 #endif
152
153 vector<int> LexBFS(Graph &g) {
154     list<Part> P;
155     list<Item> L;
156     vector<int> vertices = g.vertices;
157     vector<int> pi(g.order() + 1);
158     Part first(&L);
159
160     for (size_t i = 0; i < vertices.size(); ++i)
161         L.push_back(Item(vertices[i]));
162
163     first.set_elements(L.begin(), L.end());
164     P.push_back(first);
165     int i = 1;
166
167     while (!P.empty()) {
168         Part *a = &P.front();
169         int x = a->begin()->v;
170         L.erase(a->pop_front());
171
172         if (a->empty())
173             P.pop_front();
174
175         pi[x] = i;
176         i++;
177
178         for (list<Part>::iterator p = P.begin(); p != P.end(); ++p) {
179             list< list<Item>::iterator > S;
180             list<Item>::iterator it = p->begin();
181
182             for (size_t k = 0; k < p->size(); ++k) {
183                 if (g.isNeighbor(it->v, x))
184                     S.push_back(it);
185                 ++it;
186             }
```

```
187
188             if (S.empty() || S.size() >= p->size())
189                 continue;
190
191             Part r(&L);
192             list<Item>::iterator newly;
193
194             for (auto s: S) {
195                 Item z = *s;
196
197                 if (s == p->end()) {
198                     p->pop_back();
199                     newly = L.insert(p->begin(), z);
200                     r.push_back(newly);
201                     L.erase(s);
202                 } else if (s == p->begin()) {
203                     newly = p->pop_front();
204                     r.push_back(newly);
205                 } else {
206                     newly = L.insert(p->begin(), z);
207                     r.push_back(newly);
208                     L.erase(s);
209                     p->resize(p->size()-1);
210                 }
211             }
212             P.insert(p, r);
213         }
214     }
215
216     return pi;
217 }
```

APÊNDICE B – AMBIENTE COMPUTACIONAL

O *software* livre SageMath (THE SAGE DEVELOPERS, 2016) é um conjunto de programas matemáticos implementados em linguagem de programação Python (VAN ROSSUM et al., 2007) que oferece recursos das áreas de álgebra, cálculo, estatística, teoria dos grafos, entre outras. O SageMath versão 8.8 foi utilizado para realizar diferentes partes do projeto, incluindo um protótipo do que vem a seguir.

A implementação dos algoritmos LexBFS e MCBB, descritos anteriormente nos capítulos 2 e 3, foi feita em linguagem de programação C++ utilizando a *Standard Library*, padronizadas pela ISO (2012) — conhecida como *C++11* — compilada com **GCC** (STALLMAN, 2017) versão 7.4.0. Essa linguagem de programação foi escolhida por ser uma linguagem compilada e orientada a objetos. Tais características facilitaram as implementações uma vez que sob MCBB algoritmos são descritos aproveitando um outro descrito anteriormente. Essa implementação juto as dados brutos estão disponíveis[1] em um repositório de acesso público na internet. Para representação de grafos, foi utilizada uma matriz de adjacências que facilitou a implementação de alguns algoritmos que tem como pré-requisito esse tipo de representação de grafo.

Existe, ainda, uma implementação dos algoritmos descritos nas seções 3.2.1 e 3.3.1, fornecido no trabalho de Züge (2017), disponível[2] em um repositório público na internet. Diferente da implementação feita em C++, seus algoritmos foram implementados utilizando a linguagem programação **Cython** (BEHNEL et al., 2011), contanto ainda com o auxílio das bibliotecas fornecidas pelo *software* livre SageMath. Com isso, foi possível utilizar essa implementação como contra-prova dos algoritmos implementados em C++. Durante esse processo foi identificado um erro em uma das colorações utilizada nos algoritmos CHI e CHI + DF, que foi corrigido.

O sistema computacional utilizado foi provido pelas servidoras do Centro de Computação Cientifica e Software Livre[3] (C3SL) da Universidade Federal do Paraná. Os experimentos foram realizados na servidora chamada *cpu2* que conta sistema operacional **Linux Mint 19.1** (MINT, 2010) arquitetura X86-64, **Linux Kernel 4.19.16+** (LOVE, 2010), e um total de 196.79 GiB de memória RAM. A seguinte arquitetura de processador foi empregada.

```
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 30
```

[1] disponível em https://gitlab.c3sl.ufpr.br/apzuge/maxcliquebb
[2] a saber, https://gitlab.c3sl.ufpr.br/apzuge/maxcliquebb
[3] https://www.c3sl.ufpr.br/

```
On-line CPU(s) list: 0-29
Thread(s) per core: 1
Core(s) per socket: 1
Socket(s): 30
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 62
Model name: Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
Stepping: 4
CPU MHz: 2992.788
BogoMIPS: 5985.57
Virtualization: VT-x
Hypervisor vendor: Microsoft
Virtualization type: full
L1d cache: 32K
L1i cache: 32K
L2 cache: 4096K
L3 cache: 16384K
NUMA node0 CPU(s): 0-29
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
   pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm
   constant_tsc arch_perfmon rep_good nopl xtopology cpuid tsc_known_freq
    pni pclmulqdq vmx ssse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt
   tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm
   topoext cpuid_fault pti tpr_shadow vnmi flexpriority ept vpid fsgsbase
    tsc_adjust smep erms xsaveopt arat umip
```

Antes de ter uma versão implementada em linguagem C++, foram realizados testes preliminares com os algoritmos MCQ e MCQ_L. Para esse fim, foi escolhida a linguagem de programação Python e o *software* livre SageMath para implementar esses dois algoritmos. Foram utilizadas tanto a representação de grafos quanto uma implementação da LexBFS oferecidas pelo SageMath. Assim, o esforço para implementar uma primeira versão do algoritmo foi atenuado. Para essa fase do trabalho foram escolhidas instâncias provenientes do DIMACS (Seção 4.2) que tomam pouco tempo de execução de acordo com outros trabalhos similares, disponíveis na literatura, e o ambiente computacional à disposição.

Ademais, para o tratamento dos resultados provenientes dos experimentos descritos na Seção 4.3 foram escolhidos *softwares* como **Gnumeric** (THE GNOME PROJECT, 2017), para planilhas eletrônicas, e a biblioteca implementada em linguagem Python,

conhecida como **pandas** (THE PANDAS DEVELOPMENT TEAM, 2020). Também foram empregadas as bibliotecas **SciPy** (VIRTANEN et al., 2020), **NumPy** (OLIPHANT, 2006) e **statsmodels** (SEABOLD; PERKTOLD, 2010) para cálculos estatísticos como valor-p apresentado na Seção 4.1. Para plotar gráficos em 2D, foi utilizada a biblioteca **Matplotlib** (HUNTER, 2007).

APÊNDICE C – RESULTADOS EXPERIMENTAIS

Em seguida são apresentados os resultados experimentais obtidos da execução do o Algoritmo Greedy sob instâncias de grafos aleatórios $\mathcal{G}_{n,p}$, grafos cordais aleatórios e cografos aleatórios. A Seção C.1 apresenta os resultados para grafos aleatórios $\mathcal{G}_{n,p}$ onde $p = 1/2$. A Seção C.2 apresenta os resultados para a grafos cordais aleatórios. A Seção C.3 contém os resultados para cografos aleatórios.

C.1 GRAFOS ALEATÓRIOS

Os resultados da coloração de vértices computados por Greedy em grafos aleatórios $\mathcal{G}_{n,p}$, tomando como entrada as ordenação π dos vértices, estão presentes na Tabela 15. Também estão presentes os resultados da coloração de vértices com Greedy com π_L como entrada.

TABELA 15 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G,\pi)$	$c(G,\pi_L)$
300	22 319	47	48
300	22 320	49	48
300	22 367	49	51
300	22 405	48	49
300	22 462	49	50
300	22 510	50	50
300	22 528	48	50
300	22 585	49	49
300	22 682	49	49
300	22 683	48	51
350	30 252	54	56
350	30 340	54	56
350	30 373	53	56
350	30 415	53	54
350	30 427	57	**53**
350	30 468	54	53
350	30 505	55	55

continua

TABELA 15 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
350	30 535	53	55
350	30 643	58	**54**
350	30 725	56	54
400	39 702	61	60
400	39 818	61	60
400	39 822	63	61
400	39 895	59	60
400	39 903	60	62
400	39 906	61	62
400	39 964	61	60
400	39 974	62	60
400	40 000	63	**60**
400	40 128	60	62
450	50 278	66	70
450	50 355	67	**64**
450	50 370	65	67
450	50 411	66	67
450	50 480	67	67
450	50 516	68	67
450	50 533	66	67
450	50 601	66	67
450	50 803	69	67
450	50 929	68	69
500	62 082	75	**72**
500	62 277	73	72
500	62 351	73	74
500	62 354	73	73
500	62 401	75	**72**
500	62 458	73	72
500	62 484	70	73
500	62 535	74	72
500	62 552	72	72

continua

TABELA 15 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
500	62 579	73	71
550	74 938	80	**77**
550	75 334	78	78
550	75 456	79	77
550	75 458	76	79
550	75 521	79	79
550	75 543	78	79
550	75 557	78	80
550	75 585	80	80
550	75 649	77	79
550	75 737	79	79
600	89 490	82	82
600	89 664	82	84
600	89 820	83	84
600	89 854	84	84
600	89 892	83	85
600	89 960	86	**82**
600	89 971	84	84
600	90 019	86	85
600	90 056	87	85
600	90 101	82	85
650	105 070	88	89
650	105 136	89	87
650	105 284	91	89
650	105 437	90	91
650	105 498	90	94
650	105 518	88	90
650	105 528	88	89
650	105 568	87	92
650	105 627	90	90
650	105 643	92	**89**
700	122 022	97	95

continua

TABELA 15 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
700	122 081	97	95
700	122 093	94	94
700	122 268	95	94
700	122 346	98	**94**
700	122 435	95	96
700	122 444	95	95
700	122 478	95	96
700	122 627	95	96
700	122 648	94	95
750	140 128	101	99
750	140 197	101	99
750	140 354	101	99
750	140 462	100	101
750	140 480	100	101
750	140 508	101	103
750	140 546	102	101
750	140 588	101	100
750	140 628	101	104
750	140 752	98	100
800	159 338	106	109
800	159 626	107	**103**
800	159 659	105	106
800	159 665	106	105
800	159 854	106	105
800	159 972	106	107
800	159 991	107	107
800	160 152	105	105
800	160 231	106	106
800	160 316	105	108
850	179 741	112	**108**
850	180 017	110	109
850	180 322	109	110

continua

TABELA 15 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G,\pi)$	$c(G,\pi_L)$
850	180 380	112	112
850	180 579	110	111
850	180 584	112	112
850	180 625	111	112
850	180 668	111	113
850	180 673	113	**110**
850	180 737	109	111
900	202 073	118	116
900	202 151	117	115
900	202 158	116	116
900	202 193	116	118
900	202 197	116	117
900	202 276	117	117
900	202 294	116	116
900	202 570	118	117
900	202 716	117	115
900	202 835	118	117
950	225 009	120	122
950	225 030	121	122
950	225 159	122	122
950	225 403	122	122
950	225 549	122	122
950	225 625	124	**121**
950	225 650	123	121
950	225 701	123	121
950	225 706	121	119
950	225 886	122	122
1000	249 395	127	128
1000	249 460	127	127
1000	249 633	125	127
1000	249 674	126	126
1000	249 889	126	126

continua

TABELA 15 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de $\mathcal{G}_{n,1/2}$ como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
1000	249 927	128	126
1000	249 984	125	127
1000	250 009	126	125
1000	250 194	124	123
1000	250 221	129	129

C.2 GRAFOS CORDAIS

Os resultados da coloração de vértices computados por Greedy em grafos cordais, utilizando como entrada uma ordenação π dos vértices, estão presentes na Tabela 16. Também estão presentes os resultados da coloração de vértices com Greedy com π_L como entrada.

TABELA 16 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
300	1723	14	14
300	1867	15	15
300	1935	19	18
300	1969	19	18
300	1996	19	19
300	2026	21	21
300	2034	19	19
300	2139	17	17
300	2209	18	18
300	2399	20	20
350	2248	19	19
350	2353	16	15
350	2358	18	18

continua

TABELA 16 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
350	2539	17	17
350	2539	20	19
350	2562	19	18
350	2584	20	20
350	2638	20	20
350	2639	19	18
350	2714	18	18
400	2807	18	18
400	2824	19	19
400	2846	20	20
400	3100	22	21
400	3102	19	19
400	3143	20	20
400	3183	21	21
400	3200	24	22
400	3260	23	22
400	3293	25	25
450	3306	18	18
450	3307	16	16
450	3346	21	19
450	3413	21	21
450	3511	21	21
450	3512	20	19
450	3611	20	20
450	3685	20	19
450	3724	20	19
450	3781	24	22
500	3749	23	23
500	3858	19	19
500	4081	25	25
500	4137	23	23
500	4226	24	24

continua

TABELA 16 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
500	4256	24	24
500	4402	27	27
500	4418	20	19
500	4425	24	23
500	4541	24	24
550	4231	20	18
550	4551	21	21
550	4620	25	23
550	4661	22	21
550	4764	24	22
550	4769	23	23
550	4824	21	21
550	4878	23	23
550	4889	21	21
550	4900	25	24
600	4767	20	20
600	4921	19	19
600	5098	21	21
600	5141	23	22
600	5369	27	27
600	5468	25	25
600	5492	29	29
600	5534	27	27
600	5630	25	24
600	5652	29	29
650	5585	23	23
650	5592	20	20
650	5816	27	27
650	5975	23	23
650	5995	28	27
650	6169	22	22
650	6221	26	26

continua

TABELA 16 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
650	6222	24	22
650	6314	24	23
650	6386	27	27
700	6076	23	23
700	6137	21	20
700	6620	22	22
700	6641	26	26
700	6732	32	31
700	6779	21	21
700	6846	25	25
700	6873	29	29
700	7007	26	**23**
700	7354	25	25
750	7060	24	22
750	7084	26	25
750	7182	23	23
750	7347	27	27
750	7356	26	26
750	7535	27	27
750	7604	26	26
750	7618	27	26
750	7830	27	26
750	7861	25	25
800	7761	24	23
800	7791	26	26
800	7832	23	23
800	7961	37	37
800	7969	27	26
800	7991	25	24
800	8143	26	26
800	8248	28	28
800	8373	26	26

continua

TABELA 16 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
800	8542	30	30
850	8431	24	24
850	8714	26	26
850	8792	32	32
850	9040	25	24
850	9142	27	27
850	9391	34	34
850	9400	27	27
850	9445	27	26
850	9843	32	32
850	10 022	31	31
900	8931	27	27
900	9184	23	23
900	9442	26	24
900	9501	30	30
900	9642	32	32
900	9814	29	29
900	9869	31	31
900	9948	30	28
900	10 332	35	34
900	10 508	39	39
950	9426	31	31
950	9614	25	25
950	9721	27	27
950	9738	25	23
950	9739	26	25
950	10 066	30	30
950	10 193	36	35
950	10 464	25	25
950	10 932	33	32
950	10 968	26	26
1000	10 158	25	25

continua

TABELA 16 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de grafos cordais como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
1000	10 262	27	27
1000	10 322	28	28
1000	10 442	28	27
1000	10 870	29	29
1000	11 143	27	26
1000	11 302	26	25
1000	11 314	28	28
1000	11 335	30	30
1000	11 529	36	36

C.3 COGRAFOS

Os resultados da coloração de vértices computados por Greedy em cografos, utilizando como entrada uma ordenação π dos vértices, estão presentes na Tabela 17. Também estão presentes os resultados da coloração de vértices com Greedy com π_L como entrada.

TABELA 17 – Comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
300	3410	56	56
300	8825	83	83
300	11 093	123	123
300	22 863	210	210
300	44 850	300	300
300	44 850	300	300
300	44 850	300	300
300	44 850	300	300
300	44 850	300	300

continua

TABELA 17 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
300	44 850	300	300
350	1827	40	40
350	6952	86	86
350	10 828	112	112
350	22 015	178	178
350	27 769	229	229
350	61 075	350	350
350	61 075	350	350
350	61 075	350	350
350	61 075	350	350
350	61 075	350	350
400	15 858	114	114
400	51 692	321	321
400	57 583	336	336
400	79 800	400	400
400	79 800	400	400
400	79 800	400	400
400	79 800	400	400
400	79 800	400	400
400	79 800	400	400
400	79 800	400	400
450	6123	73	73
450	21 156	192	192
450	25 170	207	207
450	25 185	213	213
450	38 503	263	263
450	76 032	390	390
450	101 025	450	450
450	101 025	450	450
450	101 025	450	450
450	101 025	450	450
500	5954	50	50

continua

TABELA 17 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
500	12 727	107	107
500	22 587	191	191
500	50 428	299	299
500	124 750	500	500
500	124 750	500	500
500	124 750	500	500
500	124 750	500	500
500	124 750	500	500
500	124 750	500	500
550	7141	65	65
550	10 945	84	84
550	51 298	299	299
550	82 219	397	397
550	125 376	500	500
550	150 975	550	550
550	150 975	550	550
550	150 975	550	550
550	150 975	550	550
550	150 975	550	550
600	7723	70	70
600	20 577	135	135
600	21 813	135	135
600	29 905	197	197
600	30 186	170	170
600	45 608	251	251
600	73 820	358	358
600	153 719	554	554
600	179 700	600	600
600	179 700	600	600
650	25 756	146	146
650	27 604	157	157
650	71 235	349	349

continua

TABELA 17 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
650	75 358	318	318
650	150 603	545	545
650	210 925	650	650
650	210 925	650	650
650	210 925	650	650
650	210 925	650	650
650	210 925	650	650
700	37 339	153	153
700	41 930	213	213
700	125 317	472	472
700	178 673	596	596
700	244 650	700	700
700	244 650	700	700
700	244 650	700	700
700	244 650	700	700
700	244 650	700	700
700	244 650	700	700
750	27 468	169	169
750	82 981	361	361
750	110 029	438	438
750	191 597	616	616
750	213 143	649	649
750	280 875	750	750
750	280 875	750	750
750	280 875	750	750
750	280 875	750	750
750	280 875	750	750
800	20 663	142	142
800	43 703	239	239
800	78 129	318	318
800	106 415	450	450
800	129 756	498	498

continua

TABELA 17 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
800	193 684	622	622
800	196 218	621	621
800	207 126	637	637
800	319 600	800	800
800	319 600	800	800
850	70 301	311	311
850	113 401	302	302
850	139 520	494	494
850	154 428	512	512
850	360 825	850	850
850	360 825	850	850
850	360 825	850	850
850	360 825	850	850
850	360 825	850	850
850	360 825	850	850
900	79 458	332	332
900	104 772	312	312
900	128 731	484	484
900	184 586	592	592
900	200 540	604	604
900	213 769	650	650
900	214 334	647	647
900	404 550	900	900
900	404 550	900	900
900	404 550	900	900
950	37 789	158	158
950	70 091	251	251
950	177 519	470	470
950	319 149	788	788
950	450 775	950	950
950	450 775	950	950
950	450 775	950	950

continua

TABELA 17 – *Continuação:* comparação do número de cores computada pelo Algoritmo Greedy por duas ordenações diferentes de instâncias de cografos como entrada: uma ordenação π do conjunto de vértices e uma ordenação π_L do conjunto de vértices produzida por LexBFS.

$\|V\|$	$\|E\|$	$c(G, \pi)$	$c(G, \pi_L)$
950	450 775	950	950
950	450 775	950	950
950	450 775	950	950
1000	110 772	446	446
1000	249 284	683	683
1000	398 126	892	892
1000	499 500	1000	1000
1000	499 500	1000	1000
1000	499 500	1000	1000
1000	499 500	1000	1000
1000	499 500	1000	1000
1000	499 500	1000	1000
1000	499 500	1000	1000

ANEXOS

ANEXO A – DISTRIBUIÇÃO T DE STUDENT

TABELA 18 – Distribuição t de Student. Corpo da tabela contém os valores calculados para t_γ. Cada coluna representa um nível de confiança γ dado em proporção. Uma linha é o número ν de graus de liberdade.

ν	60,0%	75,0%	80,0%	87,5%	90,0%	95,0%	97,5%	99,0%	99,5%	99,9%
1	0,325	1,000	1,376	2,414	3,078	6,314	12,706	31,821	63,657	318,310
2	0,289	0,816	1,061	1,604	1,886	2,920	4,303	6,965	9,925	22,327
3	0,277	0,765	0,978	1,423	1,638	2,353	3,182	4,541	5,841	10,215
4	0,271	0,741	0,941	1,344	1,533	2,132	2,776	3,747	4,604	7,173
5	0,267	0,727	0,920	1,301	1,476	2,015	2,571	3,365	4,032	5,893
6	0,265	0,718	0,906	1,273	1,440	1,943	2,447	3,143	3,707	5,208
7	0,263	0,711	0,896	1,254	1,415	1,895	2,365	2,998	3,499	4,785
8	0,262	0,706	0,889	1,240	1,397	1,860	2,306	2,896	3,355	4,501
9	0,261	0,703	0,883	1,230	1,383	1,833	2,262	2,821	3,250	4,297
10	0,260	0,700	0,879	1,221	1,372	1,812	2,228	2,764	3,169	4,144
11	0,260	0,697	0,876	1,214	1,363	1,796	2,201	2,718	3,106	4,025
12	0,259	0,695	0,873	1,209	1,356	1,782	2,179	2,681	3,055	3,930
13	0,259	0,694	0,870	1,204	1,350	1,771	2,160	2,650	3,012	3,852
14	0,258	0,692	0,868	1,200	1,345	1,761	2,145	2,624	2,977	3,787
15	0,258	0,691	0,866	1,197	1,341	1,753	2,131	2,602	2,947	3,733
16	0,258	0,690	0,865	1,194	1,337	1,746	2,120	2,583	2,921	3,686
17	0,257	0,689	0,863	1,191	1,333	1,740	2,110	2,567	2,898	3,646
18	0,257	0,688	0,862	1,189	1,330	1,734	2,101	2,552	2,878	3,610
19	0,257	0,688	0,861	1,187	1,328	1,729	2,093	2,539	2,861	3,579
20	0,257	0,687	0,860	1,185	1,325	1,725	2,086	2,528	2,845	3,552
21	0,257	0,686	0,859	1,183	1,323	1,721	2,080	2,518	2,831	3,527
22	0,256	0,686	0,858	1,182	1,321	1,717	2,074	2,508	2,819	3,505
23	0,256	0,685	0,858	1,180	1,319	1,714	2,069	2,500	2,807	3,485
24	0,256	0,685	0,857	1,179	1,318	1,711	2,064	2,492	2,797	3,467
25	0,256	0,684	0,856	1,178	1,316	1,708	2,060	2,485	2,787	3,450
26	0,256	0,684	0,856	1,177	1,315	1,706	2,056	2,479	2,779	3,435
27	0,256	0,684	0,855	1,176	1,314	1,703	2,052	2,473	2,771	3,421
28	0,256	0,683	0,855	1,175	1,313	1,701	2,048	2,467	2,763	3,408
29	0,256	0,683	0,854	1,174	1,311	1,699	2,045	2,462	2,756	3,396
30	0,256	0,683	0,854	1,173	1,310	1,697	2,042	2,457	2,750	3,385
35	0,255	0,682	0,852	1,170	1,306	1,690	2,030	2,438	2,724	3,340
40	0,255	0,681	0,851	1,167	1,303	1,684	2,021	2,423	2,704	3,307
45	0,255	0,680	0,850	1,165	1,301	1,679	2,014	2,412	2,690	3,281
50	0,255	0,679	0,849	1,164	1,299	1,676	2,009	2,403	2,678	3,261
55	0,255	0,679	0,848	1,163	1,297	1,673	2,004	2,396	2,668	3,245
60	0,254	0,679	0,848	1,162	1,296	1,671	2,000	2,390	2,660	3,232
∞	0,253	0,674	0,842	1,150	1,282	1,645	1,960	2,326	2,576	3,090