

**MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**

**BRUNA ALVES RAMALHO
FABIO VIANA LOPES ANDRADE DA SILVA**

**IMPLEMENTAÇÃO DE ESTRUTURAS DE DADOS AVANÇADAS PARA
PROBLEMAS EM GRAFOS DINÂMICOS**

**Rio de Janeiro
2019**

INSTITUTO MILITAR DE ENGENHARIA

**BRUNA ALVES RAMALHO
FABIO VIANA LOPES ANDRADE DA SILVA**

**IMPLEMENTAÇÃO DE ESTRUTURAS DE DADOS
AVANÇADAS PARA PROBLEMAS EM GRAFOS
DINÂMICOS**

Projeto de Fim de Curso, do Curso de Graduação em Engenharia
de Computação do Instituto Militar de Engenharia.

Orientadora: Prof^ª. Cláudia Marcela Justel - D.Sc.
Co-Orientadora: Prof^ª. Lilian Markenzon - Ph.D.

Rio de Janeiro
2019

c2019

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

Ramalho, Bruna Alves

Implementação de estruturas de dados avançadas para problemas em grafos dinâmicos / Bruna Alves Ramalho, Fabio Viana Lopes Andrade Da Silva, orientado por Cláudia Marcela Justel e Lilian Markenzon - Rio de Janeiro: Instituto Militar de Engenharia, 2019.

52p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2019.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. I. Justel, Cl.udia Marcela. II. Markenzon, Lilian . III. Título. IV. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

BRUNA ALVES RAMALHO
FABIO VIANA LOPES ANDRADE DA SILVA

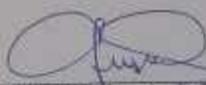
IMPLEMENTAÇÃO DE ESTRUTURAS DE DADOS
AVANÇADAS PARA PROBLEMAS EM GRAFOS
DINÂMICOS

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientadora: Prof^a. Cláudia Marcela Justel - D.Sc.

Co-Orientadora: Prof^a. Lillian Markezon - Ph.D.

Aprovado em 10 de Outubro de 2019 pela seguinte Banca Examinadora:



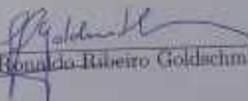
Prof^a. Cláudia Marcela Justel - D.Sc. do IME - Presidente



Prof^a. Lillian Markezon - Ph.D. da UFRJ



Maj Humberto Henriques de Arrada - M.Sc. do IME



Prof. Ronaldo Ribeiro Goldschmidt - D.Sc. do IME

Rio de Janeiro
2019

Ao Instituto Militar de Engenharia, alicerce da minha formação e aperfeiçoamento.

AGRADECIMENTOS

Agradeço a todas as pessoas que me incentivaram, apoiaram e possibilitaram esta oportunidade de ampliar meus horizontes.

Meus familiares, cônjuge e mestres.

Em especial a minha Professora Orientadora Claudia Justel e à Professora Co-orientadora Lilian Markenzon, por suas disponibilidades e atenções.

“Só dará errado se você tentar.”

AUTOR ANÔNIMO

SUMÁRIO

LISTA DE ILUSTRAÇÕES	7
1 INTRODUÇÃO	10
1.1 Objetivo	10
1.2 Motivação	10
1.3 Metodologia	11
1.4 Organização da monografia	11
2 CONCEITOS BÁSICOS	12
2.1 Teoria de Grafos	12
2.1.1 Conceitos Gerais	12
2.1.2 Representação de Grafos	14
2.1.3 Classes de Grafos	14
2.1.4 Grafos Cordais	15
2.2 Ferramentas Estruturais para Grafos Cordais	15
2.3 Algoritmos Dinâmicos	21
2.3.1 Definição e operações	21
2.3.2 Classificação	21
3 SISTEMA PROPOSTO	23
3.1 Geração aleatória de grafos bloco	23
3.1.1 Geração estática	23
3.1.2 Geração dinâmica	25
3.2 Projeto da Interface	32
3.2.1 Identificação dos requisitos	32
3.2.2 Escolha das ferramentas computacionais	35
4 IMPLEMENTAÇÃO	37
4.1 Algoritmos de reconhecimento	37
4.1.1 Reconhecimento de grafo cordal	37
4.1.1.1 Representação de um grafo	37
4.1.1.2 Percorso em largura lexicográfica	38
4.1.1.3 Verificação de um eep	42
4.1.2 Reconhecimento de grafo bloco	42

4.1.2.1	Determinação do conjunto dos smv e suas multiplicidades.....	42
4.2	Algoritmos de geração de grafos bloco.....	45
4.2.1	Geração estática.....	45
4.2.2	Geração dinâmica.....	45
4.3	A interface desenvolvida.....	45
4.3.1	Página inicial.....	45
4.3.2	Reconhecer um grafo a uma classe.....	46
4.3.3	Gerar um grafo bloco.....	47
5	CONCLUSÃO	51
6	REFERÊNCIAS BIBLIOGRÁFICAS	52

LISTA DE ILUSTRAÇÕES

FIG.3.1	Grafo inicial	27
FIG.3.2	Grafo do caso 1.a	27
FIG.3.3	Grafo do caso 1.b	28
FIG.3.4	Grafo do caso 2.a	29
FIG.3.5	Fluxo do programa	33
FIG.4.1	Grafo cordal não bloco	37
FIG.4.2	Representação de um grafo	38
FIG.4.3	Representação do grafo da FIG 4.1	38
FIG.4.4	Estruturas de dados usadas no percurso em largura lexicográfica	40
FIG.4.5	Estruturas do percurso em largura lexicográfica para o grafo da FIG 4.1	41
FIG.4.6	Estruturas de dados usadas na determinação de \mathbb{S} e multiplicidades	43
FIG.4.7	Estruturas do algoritmo de determinação de \mathbb{S} e suas multiplicida- des para o grafo da FIG 4.1	44
FIG.4.8	Página Inicial	46
FIG.4.9	Grafo não cordal	46
FIG.4.10	Página de reconhecimento de um grafo não cordal representado como lista de adjacência	47
FIG.4.11	Grafo cordal não bloco	47
FIG.4.12	Página de reconhecimento de um grafo cordal representado como matriz de adjacência	48
FIG.4.13	Página de reconhecimento de um grafo não bloco representado como matriz de adjacência	48
FIG.4.14	Página de geração de um grafo bloco usando o algoritmo estático	49
FIG.4.15	Grafo bloco gerado estaticamente	49
FIG.4.16	Página de geração de um grafo bloco usando o algoritmo dinâmico	50
FIG.4.17	Grafo bloco gerado dinamicamente	50

RESUMO

Algoritmos dinâmicos para resolução de problemas em grafos representam um ramo da Computação que vem recebendo cada vez mais atenção nos últimos anos. O intuito deste trabalho é disponibilizar uma interface de fácil utilização para usuários de diferentes domínios, que não necessite de conhecimento de programação, abordando alguns problemas em grafos, como reconhecimento de classes e geração estática e dinâmica de grafos aleatórios pertencentes a determinadas classes. Para o desenvolvimento destes algoritmos, são apresentadas as caracterizações dessas classes de grafos, em especial para os grafos cordais e suas subclasses. Finalmente, foram implementadas estruturas de dados avançadas, que tornam estes algoritmos eficientes.

ABSTRACT

Algorithms for problem solving in dynamic graphs represent a branch of computing that has received increasing attention in recent years. The purpose of this work is to provide an easy-to-use tool for users from different domains, who do not need programming knowledge, addressing some problems in graphs, such as graph class recognition and both static and dynamic generation of random graphs for certain classes. For the development of these algorithms, we will discuss the characterizations of some classes of graphs, especially for the chordal graphs and their subclasses, besides certain problems for this family, and finally implement advanced data structures, that make these algorithms efficient.

1 INTRODUÇÃO

1.1 OBJETIVO

Os objetivos principais deste trabalho são: aprofundamento no estudo de teoria de grafos, em particular grafos cordais; desenvolvimento e análise de algoritmos estáticos e dinâmicos para geração aleatória de grafos; implementação dos algoritmos estudados; implementação da ferramenta para manipular subclasses de grafos cordais: reconhecimento e geração.

Como objetivo secundário do trabalho podemos mencionar o desenvolvimento de um sistema que permita manipular algoritmos em grafos cordais e subclasses. Esse sistema permitirá tanto reconhecer grafos pertencentes a essas subclasses, quanto fazer geração aleatória dos mesmos.

1.2 MOTIVAÇÃO

A Teoria de Grafos é uma área de estudo que vem recebendo cada vez mais atenção de pesquisadores de diversos campos, como matemática, computação, estatística, entre outros. Em especial, algoritmos dinâmicos estão relacionados a diversos problemas, com aplicações em redes sociais e eletrônica, por exemplo.

Esse crescimento de pesquisas na área torna ainda mais importante a criação de ferramentas para integrar a manipulação de algoritmos em grafos, sem que seja necessário algum conhecimento de programação prévio. O presente trabalho será desenvolvido com essa motivação.

No âmbito da SE/9, este trabalho vem dar continuidade ao estudo de grafos cordais. Grafos cordais são grafos para os quais qualquer ciclo de tamanho maior ou igual a quatro possui uma aresta conectando vértices não consecutivos no ciclo (aresta chamada corda). Os grafos cordais são uma subclasse de grafos perfeitos e possuem a particularidade de serem reconhecidos em tempo linear. Além disso, alguns problemas que são NP-completo para grafos em geral, admitem solução polinomial quando restritos à classe de grafos cordais. Por um lado, foi realizada uma dissertação de mestrado sobre algoritmos dinâmicos para grafos outerplanares (NETO, 2017). Os relatórios (RAMALHO, 2017) e (RAMALHO, 2018) apresentaram um estudo sobre classes de grafos cordais, resultando na implementação dos algoritmos de reconhecimento de grafos cordais, estritamente cordais e estritamente de intervalo e no desenvolvimento de um algoritmo alternativo para

reconhecimento de grafos estritamente de intervalo dentro do projeto de Iniciação Científica. Pretende-se facilitar a utilização de implementações já desenvolvidas assim como novas implementações através da ferramenta proposta neste trabalho.

1.3 METODOLOGIA

O presente trabalho foi desenvolvido seguindo as seguintes etapas.

Inicialmente foram estudados alguns conceitos e definições mais gerais sobre a Teoria de Grafos e de Algoritmos Dinâmicos, para então aprofundar na classe de grafos cordais, em particular nos grafos bloco.

Posteriormente foram levantadas as funcionalidades e os requisitos para a criação da interface, e verificadas as ferramentas necessárias para o desenvolvimento dos algoritmos e da interface, assim como suas implementações. Após essas etapas, foi realizada a implementação do produto final.

1.4 ORGANIZAÇÃO DA MONOGRAFIA

O restante deste trabalho está organizado da seguinte maneira.

O Capítulo 2 apresenta tópicos da Teoria de Grafos, que formam a base teórica para o desenvolvimento do trabalho, e são abordadas ferramentas estruturais, que podem estabelecer propriedades de classes de grafos. Primeiramente são introduzidos alguns conceitos mais simples, como *cliques*, *vértices simpliciais*, entre outros, assim como diferentes formas de representação computacional de grafos, classes de grafos, e em especial a classe dos grafos cordais. Após essa introdução, é possível definir conceitos que são fundamentais para caracterizar os grafos cordais, como *sequência de vértices* e *esquema de eliminação*, entre outros tópicos necessários para o desenvolvimento do trabalho.

O Capítulo 3 trata do sistema proposto, com detalhes da geração estática e dinâmica de grafos bloco, que será a principal classe que será utilizada pelo sistema. Também trata dos aspectos técnicos do trabalho, como a identificação das funcionalidades, o levantamento de requisitos, a descrição dos casos de uso, a linguagem de programação escolhida e o modelo do sistema.

O Capítulo 4 descreve o funcionamento da interface que será desenvolvida nesse trabalho. São apresentados detalhes do uso da interface, além dos algoritmos gerados e como foram implementados.

2 CONCEITOS BÁSICOS

As próximas seções têm por finalidade apresentar os conceitos sobre Teoria de Grafos necessária para o desenvolvimento do projeto. Os conceitos aqui apresentados foram retirados de (MARKENZON; WAGA, 2016) e (RAMALHO, 2017)

2.1 TEORIA DE GRAFOS

Nessa seção serão estudados alguns conceitos mais básicos sobre grafos. Serão introduzidos os grafos cordais e algumas de suas subclasses.

2.1.1 CONCEITOS GERAIS

Um grafo não orientado G é um par (V, E) , ou (V_G, E_G) , em que V (ou V_G) é o conjunto de vértices e E (ou E_G) é o conjunto de arestas. Uma *aresta* é um par não ordenado de vértices, que por sua vez são denominados *extremidades*. Chamamos de *laço* uma aresta com extremidades iguais.

Um grafo é *finito* quando possui um número finito de vértices e não possui laços. O número de vértices é chamado de ordem e pode ser representado por n ($n = |V|$). A quantidade de arestas é chamada de tamanho e pode ser representada por m ($m = |E|$).

Quando dois vértices são extremidades de uma aresta, são chamados de *adjacentes* ou *vizinhos*. Da mesma forma, duas arestas distintas que incidem sobre o mesmo vértice são arestas *adjacentes*. Chamamos de *vizinhança aberta* de um vértice $v \in V$ o conjunto de vértices que são vizinhos de v e é denotado por $N(v)$. Formalmente, $N(v) = \{w \in V \mid v, w \in E\}$. Uma *vizinhança fechada* é o conjunto $N[v] = N(v) \cup \{v\}$. Podemos ainda estender esse conceito para subconjuntos. Tomando $Y \subset V$, temos $N(Y) = \bigcup_{v \in Y} N(v)$ e $N[Y] = \bigcup_{v \in Y} N[v]$.

O grau de um vértice, $d(v)$, é a quantidade de vértices que estão na vizinhança aberta de v , ou seja, a cardinalidade de $N(v)$. O *grau máximo* de um grafo G é $\Delta(G) = \max\{d(v)\}, v \in V$. Dizemos que um vértice é *universal* quando $N[v] = V$ (v é vizinho de todos os outros vértices de V) e dizemos que ele é *pendente* quando $d(v) = 1$. Um grafo é dito *k-regular* quando todos os seus vértices possuem grau k .

Dois vértices são *gêmeos verdadeiros* se possuem a mesma vizinhança fechada (se $N[u] = N[v]$) e são *gêmeos falsos* se possuem a mesma vizinhança aberta, porém não são

vizinhos (se $N(u) = N(v)$, mas $\{u, v\} \notin E$).

Dado um grafo $G = (V, E)$, Temos que $G' = (V', E')$ é um *subgrafo* de G quando $V' \subseteq V$ e $E' \subseteq E$. Para um subconjunto $Y \subseteq V$, denominamos $G[Y] = (Y, \{\{x, y\} \in E : x, y \in Y\})$ o *subgrafo de G induzido por Y* , ou seja, é o subgrafo que possui Y como conjunto de vértices e como conjunto de arestas todas aquelas do grafo e que possuem extremidades em Y .

Uma *clique* é um conjunto $C \subseteq V$ em que $G[C]$ é um grafo *completo*, ou seja, é um subgrafo de G que possui todas as arestas possíveis. Uma *k -clique*, com $k \geq 1$ é uma clique de cardinalidade k e a *cardinalidade de uma clique máxima* ($\omega(G)$) é a maior cardinalidade entre todas as cliques contidas no grafo G . Um vértice $v \in V$ é dito *simplicial* em G quando $N(v)$ é uma clique.

Um *caminho* em $G = (V, E)$ é uma sequência $\langle v_1, \dots, v_n \rangle$ de $k \geq 1$ vértices distintos tal que para $k > 1$ tem-se $\{v_i, v_{i+1}\} \in E$, para $i = 1, \dots, k-1$. O *comprimento* do caminho é igual a $k - 1$. Definimos *ciclo* como um caminho de comprimento maior ou igual a 3 em que apenas o primeiro e o último vértices coincidem. Um grafo é *acíclico* quando não possui ciclos. Uma *corda* é uma aresta que é incidente a dois vértices de um ciclo, em que os vértices não são consecutivos.

Um grafo é *conexo* se, para todo par de vértices distintos, existe um caminho entre eles. Quando essa condição não é observada, o grafo é *desconexo*. A *distância* entre dois vértices u e v , denotada por $dist(u, v)$ ou por $dist_g(u, v)$ é o menor caminho entre esses vértices. A maior distância de um grafo conexo é chamada de *diâmetro* ($diam(G)$).

Um caminho é dito *euleriano* quando passa por todas as arestas apenas uma vez, mas admite repetição de vértices. Um caminho é *hamiltoniano* quando passa por todos os vértices apenas uma vez.

Um grafo G' é *componente conexa* de G se G' for *subgrafo conexo maximal* de G , ou seja, quando não existir um grafo $G'' = (V'', E'')$ com $V' \subset V'' \subset V$ e $E' \subset E'' \subset E$ e $G' \neq G''$. Um grafo é *k -conexo* quando for necessário remover pelo menos k vértices para desconectá-lo. Um grafo *2-conexo* também é denotado *biconexo*.

Uma *árvore* T é um grafo que é conexo e acíclico. Chamamos de *folhas* os vértices pendentes de uma árvore. Uma *árvore geradora* de G é uma subárvore de G que possui todos os vértices de G .

Dado um subconjunto $Y \subset V$ de vértices de um grafo $G = (V, E)$, dizemos que Y é um *conjunto dominante* de G quando $N[Y] = V$ e dizemos que Y é um *conjunto independente* de G se quaisquer 2 vértices de Y não são adjacentes. A cardinalidade do maior conjunto independente maximal é chamada de *número de independência* de G , e é

denotado por $\beta_0(G)$.

Dois grafos $G = (V, E)$ e $G' = (V', E')$ são chamados de *isomorfos* quando existe uma função bijetora $\lambda : V \rightarrow V'$ que preserva a adjacência, ou seja, para quaisquer $u, v \in V$, $\{u, v\} \in E \Leftrightarrow \{\lambda(u), \lambda(v)\} \in E'$.

2.1.2 REPRESENTAÇÃO DE GRAFOS

Computacionalmente falando, não é conveniente utilizar para os grafos a representação geométrica por meio de pontos que representam os vértices e os segmentos de reta que os ligam, representando as arestas. Duas estruturas que são bastante utilizadas são as *matrizes de adjacências* e os *conjuntos de adjacências*.

A matriz de adjacência é uma matriz $A_{n \times n}$, em que n é a ordem do grafo. Cada linha e coluna representa um vértice, e dizemos que $A(i, j) = 1$, se $\{i, j\} \in E$, e $A(i, j) = 0$, caso contrário.

Um conjunto de adjacência é a representação de todos os pares $(v, N(v))$, em que $v \in V$. Essa representação é comumente mais adotada, e a complexidade dos conjuntos de adjacência é da ordem de $O(n^2)$.

2.1.3 CLASSES DE GRAFOS

Dizemos que o conjunto de todos os grafos que possuem uma dada propriedade constitui uma *classe de grafos*. É interessante de se definir novas classes porque, embora a solução de grafos em geral para um problema pode ser difícil de se resolver, o mesmo problema, dentro de uma classe, pode possuir uma solução mais simples. Algumas classes são bastante conhecidas. A seguir serão apresentadas algumas definições:

- a) Um grafo é *completo*, K_n , se é $(n-1)$ -regular, ou seja, se todos os vértices são adjacentes a cada um dos outros $n - 1$ vértices.
- b) Um grafo é um *ciclo*, denotado por C_n quando for m grafo conexo, onde todos os vértices possuem grau 2, ou seja, $\delta(v) = 2, \forall v \in V$.
- c) O grafo *roda* é o grafo em que todos os vértices de C_{n-1} são adjacentes a um vértice $v \notin C_{n-1}$.
- d) Um grafo *bloco* é um grafo em que todas as componentes biconexas são cliques.
- e) Um grafo *caminho* é um grafo que é uma árvore com exatamente 2 folhas.

- f) Um grafo *bipartido* é um grafo em que podemos particionar V em $V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, de modo que nenhuma aresta tenha ambas as extremidades no mesmo subconjunto, e o denotamos por $G = (V_1 \cup V_2, E)$.

Dizemos que a *definição* de uma classe é a propriedade que é satisfeita pelos seus integrantes. Uma *caracterização* é uma propriedade alternativa, mas que é equivalente à propriedade apresentada pela definição. Normalmente as caracterizações das classes ocorrem quando proíbe-se a existência de algum tipo de subgrafo. Dessa forma, se um grafo G é *H-livre* ou *livre de H*, onde H é um conjunto de grafos, quando nenhum subgrafo de G pertence a H . Assim, os grafos de H são *subgrafos proibidos* para a classe dos grafos *H-livres*.

Quando se define uma nova classe de grafos, existe o *problema de reconhecimento da pertinência*, que é, em termos computacionais, a obtenção de um algoritmo que verifica se um dado grafo pertence a essa classe. Se a classe é a interseção de outras classes com algoritmos de reconhecimento conhecidos, podemos utilizar duas abordagens: verificar se o grafo pertence a essas classes já conhecidas, ou desenvolver um algoritmo específico para essa classe.

2.1.4 GRAFOS CORDAIS

Um grafo cordal $G = (V, E)$ é um grafo em que todo ciclo de tamanho maior do que 3 possui uma corda. Também podem ser encontrados na literatura como *grafos triangulados* ou *grafos de circuito rígido*.

2.2 FERRAMENTAS ESTRUTURAIS PARA GRAFOS CORDAIS

Esta seção será dedicada às ferramentas estruturais que poderão estabelecer algumas das principais propriedades das classes de grafos estudadas, em especial os grafos cordais.

Uma *sequência* é uma lista de objetos considerados em uma ordem especial. Cada elemento de uma sequência recebe um número de 1 a n , que define a posição do elemento na sequência.

Um *esquema de eliminação* é uma sequência $\langle v_1, \dots, v_n \rangle$ de vértices de um grafo G que satisfaz a seguinte propriedade P : para todo $i \in \{1, \dots, n\}$, o vértice v_i satisfaz a propriedade P no grafo restante $G_i = G[\{v_i, \dots, v_n\}]$. A seguir será apresentado um exemplo de esquema de eliminação perfeita.

Seja $G = (V, E)$ um grafo e $\sigma = \langle v_1, \dots, v_n \rangle$ uma sequência de vértices. σ será um *esquema de eliminação perfeita (eep)* se cada v_i for um vértice simplicial no subgrafo

induzido $G[\{v_i, \dots, v_n\}]$. Utilizaremos a notação $\sigma(i)$ para determinar o elemento v_i e $\sigma^{-1}(v)$ para a posição do vértice v na sequência.

O esquema de eliminação perfeita é importante para caracterizar os grafos cordais.

Teorema 2.1 *Um grafo G é cordal se e somente se G admite um esquema de eliminação perfeita.*

Podemos utilizar percursos especiais para determinar um *eep* de um grafo cordal, entre eles o *percurso de cardinalidade máxima* e o *percurso em largura lexicográfica*. Dizemos que um *percurso* sobre um grafo é um processo sistemático de varredura que explora cada elemento da estrutura exatamente uma vez.

No *percurso de largura lexicográfica*, o critério de escolha é baseado em um rótulo atribuído aos vértices, que armazena os vizinhos dos vértices v já explorados até o momento. O critério de escolha do vértice a ser visitado é o vértice com maior rótulo, baseados na ordenação lexicográfica, ou seja, a ordem do dicionário. Para auxiliar a construção do rótulo, para cada vértice v é utilizada uma sequência de números inteiros, denominada $label[v]$. O *eep* é construído do fim para o início, o que significa que, quando o vértice é escolhido, ele é acrescentado ao início da sequência que determina o *eep*. No final da execução, a variável $index(v)$ indica a posição do vértice na sequência.

O Algoritmo 2.1 mostra o pseudocódigo que realiza o percurso em largura lexicográfica.

O Algoritmo 2.1 é mais eficiente quando, em sua implementação, empregamos algumas estruturas de dados específicas. O conjunto V' deve ser mantido em uma lista "vertical" duplamente encadeada, cujos elementos representam valores distintos de $label$, ordenados em ordem decrescente. Cada nó dessa lista "vertical" contém uma outra lista "horizontal", que também é duplamente encadeada, de vértices que possuem o mesmo valor de $label$. Utilizando essa estrutura de armazenamento, o algoritmo possui complexidade de tempo $O(n + m)$.

De acordo com o Teorema 2.1, essa sequência obtida é um *eep* se e somente se o grafo G for cordal. Logo, é possível reconhecer se um grafo é cordal verificando os seguintes passos:

Passo 1: Aplicar o algoritmo percurso em largura lexicográfica em um grafo G como entrada para determinar uma sequência de vértices.

Passo 2: Verificar se essa sequência é um *eep*.

O algoritmo que testa se uma sequência dada σ é uma *eep* está descrito no Algoritmo 2.2:

Algoritmo 2.1 Percurso Largura Lexicográfica

Entrada: grafo cordal $G = (V, E)$;

Saída: ep σ ;

Início

$V' \leftarrow V; i \leftarrow n + 1; \sigma \leftarrow \langle \rangle$

Para $v \in V$ **faça**

$label[v] \leftarrow \langle \rangle$;

$index(v) \leftarrow 0$;

Enquanto $V' \neq \emptyset$ **faça**

escolher $v \in V'$ tal que $label[v]$ seja máximo;

$V' \leftarrow V' \setminus \{v\}$;

$\sigma \leftarrow \langle v \rangle \parallel \sigma$;

$i \leftarrow i - 1$;

$index(v) \leftarrow i$;

Para $w \in N(v)$ tal que $index(w) = 0$ **faça**

$label[w] \leftarrow label[w] \parallel i$;

Fim.

A complexidade de tempo para reconhecer um grafo cordal também é de $O(n + m)$.

Seja $G = (V, E)$ um grafo cordal e o conjunto $Q \subseteq V$ uma clique do grafo G . Q é uma clique *maximal* quando não existir uma clique Q' em G tal que $Q \subseteq Q'$ e é *máxima* em G quando, para toda clique Q' de G , $|Q'| \leq |Q|$. O conjunto de todas as cliques maximais de um grafo G é denotada por \mathcal{Q} ,

O Algoritmo 2.3 determina as cliques maximais de um grafo cordal G :

Uma caracterização entre vértices simpliciais e cliques maximais pode ser descrita pelo Teorema 2.2 (RAMALHO, 2017)

Teorema 2.2 *Um vértice é simplicial em um grafo cordal G se e somente se pertence a exatamente uma clique maximal.*

Esse teorema será necessário para o algoritmo de geração dinâmica.

O conjunto S é um *separador de vértices* para vértices u e v se a remoção de S do grafo G separar u e v em componentes conexas distintas (também é chamado de uv – *separador*). S será um *uv -separador minimal* se nenhum subconjunto próprio de S

Algoritmo 2.2 Algoritmo Testar-*eep*

Entrada: grafo cordal $G = (V, E)$ e sequência $\sigma = \langle v_1, \dots, v_n \rangle$;

Saída: σ é *eep* ou não;

Início

$eep \leftarrow true$;

Para $v \in V$ **faça** $A(v) \leftarrow 0$;

Para $i = 1, \dots, n$ **faça**

$v \leftarrow \sigma(i)$;

$X \leftarrow \{X \in N(v) : \sigma^{-1}(v) < \sigma^{-1}(x)\}$;

Se $X \neq \emptyset$ **então**

$u \leftarrow \sigma(\min\{\sigma^{-1}(x) : x \in X\})$;

$A(u) \leftarrow A(u) \cup (X \setminus \{u\})$;

Se $A(v) \setminus N(v) \neq \emptyset$ **então**

$eep \leftarrow false$;

Fim.

é um uv -separador, e sendo S um uv -separador minimal para algum par u e v de vértices, então ele é dito *separador minimal de vértices (smv)*.

O Algoritmo 2.4 determina os separadores minimais de vértices de um grafo cordal G :

Uma subclasse dos grafos cordais, os grafos *bloco*, definidos na página 10, podem ser caracterizados pelo seguinte teorema:

Teorema 2.3

1. G é um grafo bloco.
2. G é um grafo conexo onde cada componente biconexa é um grafo completo.
3. G é cordal e qualquer separador minimal de vértices é um conjunto unitário.

A terceira afirmação do Teorema 2.3 permite obter uma boa caracterização para implementar um algoritmo de reconhecimento de grafos bloco, assim como para sua geração. No próximo capítulo serão apresentados algoritmos para grafos cordais que serão usados para reconhecer e gerar grafos bloco a partir deste teorema.

Algoritmo 2.3 Algoritmo Determinar Cliques Maximais de um grafo cordal

Entrada: grafo cordal $G = (V, E)$ e *eep* $\sigma = \langle v_1, \dots, v_n \rangle$;

Saída: \mathbb{Q} ;

Início

$q \leftarrow 1; p \leftarrow 0; \mathbb{Q} \leftarrow \emptyset$;

$Q_1 \leftarrow \{v_n\}; \eta(v_n) \leftarrow 1$

Para $i = n - 1, \dots, 1$ **faça**

$X_\sigma(v_i) \leftarrow \{u \in N(v_i) : \sigma^{-1}(u) > \sigma^{-1}(v_i)\}$;

$w \leftarrow z \in X_\sigma(v_i)$ tal que $\sigma^{-1}(z)$ é mínimo;

$k \leftarrow \eta(w)$;

Se $|X_\sigma(v_i)| < |Q_k|$ **então**

$q \leftarrow q + 1$

$Q_q \leftarrow \{v_i\} \cup X_\sigma(v_i)$;

$\mathbb{Q} \leftarrow \mathbb{Q} \cup Q_q$;

$\eta(v_i) \leftarrow q$;

senão

$Q_k \leftarrow \{v_i\} \cup Q_k$;

$\eta(v_i) \leftarrow k$;

Fim.

Algoritmo 2.4 Algoritmo Determinar Separadores Minimais de um grafo cordal

Entrada: grafo cordal $G = (V, E)$ e eep σ fornecida pelo algoritmo 2.1;

Saída: \mathbb{S} e multiplicidade dos separadores minimais de vértices;

Início

$q \leftarrow 1; j \leftarrow 0;$

$Q_1 \leftarrow \{v_n\}; \eta(v_n) \leftarrow 1;$

Para $i = 1, \dots, n$ **faça** $last[i] \leftarrow 0$

$\mathbb{S} \leftarrow \emptyset; S_0 \leftarrow \emptyset;$

Para $i = n - 1, \dots, 1$ **faça**

$X_\sigma(v_i) \leftarrow \{u \in N(v_i) : \sigma^{-1}(u) > \sigma^{-1}(v)\};$

$w \leftarrow z \in X_\sigma(v_i)$ tal que $\sigma^{-1}(z)$ é mínimo;

$k \leftarrow \eta(w);$

Se $|X_\sigma(v_i)| < |Q_k|$ **então**

$q \leftarrow q + 1;$

$Q_q \leftarrow \{v_i\} \cup X_\sigma(v_i);$

$\eta(v_i) \leftarrow q;$

$tam \leftarrow |X_\sigma(v_i)|; c \leftarrow last[tam];$

Se $X_\sigma(v_i) \neq S_c$ **então**

$j \leftarrow j + 1; S_j \leftarrow X_\sigma(v_i);$

$last[tam] \leftarrow j; \mu(j) \leftarrow 1;$

$\mathbb{S} \leftarrow \mathbb{S} \cup S_j;$

senão

$\mu(c) \leftarrow \mu(c) + 1;$

senão

$Q_k \leftarrow \{v_i\} \cup Q_k$

$\eta(v_i) \leftarrow k$

Fim.

2.3 ALGORITMOS DINÂMICOS

Esta seção visa fornecer base teórica para o desenvolvimento do algoritmo dinâmico de geração de grafos bloco, classe escolhida para iniciar a implementação para este trabalho.

2.3.1 DEFINIÇÃO E OPERAÇÕES

Um grafo é dinâmico quando pelo menos uma de suas entidades (vértices, arestas ou qualquer atributo associados a seus vértices ou arestas, como coloração e peso) sofrem alteração com o tempo (NANNICINI; LIBERTI, 2008). Normalmente, procura-se responder perguntas sobre o grafo à medida em que ele é alterado, como por exemplo sobre sua conectividade ou pertencimento a uma classe de grafos.

Na maioria dos casos, soluções para problemas de otimização combinatória em grafos dinâmicos se afastam significativamente do caso original em grafos estáticos, na medida em que os métodos empregados se tornam diferentes e mais otimizados. Tipicamente, problemas de grafos dinâmicos envolvem responder perguntas, a cada atualização do grafo, como conectividade ou o menor caminho entre dois vértices fixos. O diferencial dessa abordagem é manter uma forma mais eficiente de solucionar tais perguntas, em vez de recalcular do zero a resposta a cada iteração, como se o estado anterior do grafo não fosse conhecido. Por isso, algoritmos e estruturas de dados dinâmicos tendem a ser mais complexos do que sua versão estática (MEHTA; SAHNI, 2004).

Em termos de aplicação, vemos algoritmos dinâmicos sendo usados para calcular desde os caminhos mais curtos em estradas, por exemplo, cujas arestas são ponderadas com um tempo de viagem que depende das condições de tráfego e conseqüentemente mutável ao longo do tempo (ZAROLIAGIS, 2002), como também seu uso em redes de comunicação, computação gráfica e VLSI design. Nas últimas décadas, por conta da vasta gama de aplicações, esse ramo da computação vem recebendo mais atenção, produzindo recentemente cada vez mais resultados revelantes para a área.

2.3.2 CLASSIFICAÇÃO

Segundo Mehta e Sahni (2004), classificamos um algoritmo envolvendo grafos dinâmicos com base nos tipos de atualizações que são permitidas.

1 - Completamente dinâmico: Grafos que recebem essa classificação podem ter operações de inserção e remoção de arestas e/ou vértices.

2 - Parcialmente dinâmico: Grafos que recebem essa classificação podem ter apenas um tipo de operação, isto é inserção ou remoção.

3 - Incremental: Grafos que recebem essa classificação podem sofrer apenas inserção.

4 - Decremental: Grafos que recebem essa classificação podem sofrer apenas remoção.

3 SISTEMA PROPOSTO

3.1 GERAÇÃO ALEATÓRIA DE GRAFOS BLOCO

Nesta seção, são abordados algoritmos para geração aleatória especificamente para grafos bloco. Cada algoritmo garante que todos os grafos pertencentes à classe em questão são capazes de serem gerados, porém não garantimos que todos os grafos gerados têm a mesma probabilidade de geração.

Primeiramente, a seção aborda a geração estática, isto é, a partir do número de vértices desejados, é construído um grafo bloco que atenda a essa condição.

Em seguida, é discutida a geração parcialmente dinâmica para a mesma classe, cuja construção é feita de forma incremental, acrescentando-se vértices. Esta abordagem permite que, ao final da construção do grafo com o número de vértices previamente estabelecido, sejam acrescentados mais vértices, dependendo da necessidade do usuário.

3.1.1 GERAÇÃO ESTÁTICA

O Algoritmo 3.1 propõe, em pseudocódigo, a geração estática aleatória de grafos bloco.

Foram utilizadas as seguintes notações de funções:

- $random(i)$: gerador de um inteiro aleatório de 1 a i
- $complete(i)$: gerador de um grafo completo com i vértices
- $G.add(C)$: união do grafo C ao grafo G , mantendo ambos desconexos (a rotulação dos vértices originais de G permanece inalterada)
- $G.createEdge(i, j)$: criação de aresta entre os vértices de rótulos i e j no grafo G .

Na primeira etapa em (*), particiona-se o conjunto de n vértices aleatoriamente, construindo os conjuntos iniciais de cliques, as quais podem não ser as cliques maximais no final do algoritmo.

Em (**), é considerado que todo separador minimal de vértices de um grafo bloco é necessariamente um conjunto unitário pelo Teorema 2.3. Por isso, escolhe-se aleatoriamente um vértice v no conjunto de cliques obtidas no passo anterior, que será o novo

Algoritmo 3.1 Geração Estática Aleatória

Entrada: número de vértices (n);

Saída: grafo bloco aleatório $G = (V, E)$, tal que $|V| = n$;

Início

$c \leftarrow \text{random}(n)$

$G \leftarrow \text{complete}(c)$

$\text{remaining} \leftarrow n - c$

enquanto $\text{remaining} > 0$

$c \leftarrow \text{random}(\text{remaining})$ (*)

$C \leftarrow \text{complete}(c)$

$G \leftarrow G.\text{add}(C)$

$v \leftarrow \text{random}(n - \text{remaining})$

para $n - \text{remaining} + 1 \leq i \leq n - \text{remaining} + c$

$G \leftarrow G.\text{createEdge}(v, i)$ (**)

$\text{remaining} \leftarrow \text{remaining} - c$

Fim.

separador. Logo, é necessário que esse vértice (separador) seja adjacente a todos os vértices na clique sendo criada no passo atual. O vértice v pode ser um separador ou um vértice simplicial no passo anterior.

- Suponha que v seja conexo a apenas 1 vértice. Poderíamos então gerar o mesmo grafo usando desta vez uma clique a mais durante o processo de partição, no caso uma clique com apenas 1 vértice.
- Suponha que v seja conexo a mais de 1 vértice, mas não a todos os vértices da clique. Isso invalidaria a caracterização de grafos bloco, porque teríamos um novo conjunto separador de vértices não unitário (formado por estes vértices ligados ao vértice separador de G).

Teorema 3.1 *Todo grafo gerado pelo Algoritmo 1 é um grafo bloco.*

Prova

É imediato que todos os grafos gerados pelo Algoritmo 3.1 são cordais, porque todos os ciclos são na verdade subgrafos completos.

Segundo (**) o separador de vértices v é adjacente a todos os vértices do subgrafo completo a ser adicionado. Por isso, essa adição não cria um novo separador de vértices.

Por caracterização, sabemos que um grafo é bloco se e somente se o grafo é cordal e qualquer separador minimal de vértices é um conjunto unitário.

Logo, o grafo gerado pelo Algoritmo 3.1 satisfaz as duas condições. ■

Teorema 3.2 *Todo grafo bloco pode ser gerado pelo Algoritmo 3.1.*

Prova

Suponha que exista um grafo bloco G que não possa ser gerado pelo Algoritmo 3.1.

Pelo Teorema 2.3, sabe-se que um grafo é bloco se e somente se o grafo é conexo e cada componente biconexa é um grafo completo.

Existe alguma componente biconexa de G que possui apenas um vértice não simplicial. Caso contrário, o grafo seria completo, o que contradiz nossa hipótese, pois todo grafo completo pode ser gerado pelo Algoritmo 3.1 (o valor sorteado para o tamanho da primeira clique maximal deve ser igual ao número de vértices dado como entrada, $c = n$). Repare que a remoção dos vértices simpliciais ainda preserva a conectividade do grafo.

Seja G' o grafo obtido pela retirada de tais vértices. Veja que essa remoção ainda preserva a propriedade do grafo ser bloco. Com isso, como G não pode ser gerado pelo Algoritmo 3.1, então G' também não pode ser, visto que a operação inversa de adição da componente biconexa, que é um subgrafo completo, está contemplada no Algoritmo 3.1.

Basta repetir o processo de retirada de uma componente biconexa quantas vezes forem necessárias até se chegar a um grafo completo, o que contradiz nossa hipótese.

Logo, não existe nenhum bloco que não possa ser gerado pelo Algoritmo 3.1. ■

3.1.2 GERAÇÃO DINÂMICA

Nesta seção, é apresentada a idéia de um algoritmo dinâmico incremental para geração de grafo bloco por adição de vértices.

A seguir apresenta-se a construção da idéia baseada em uma análise de casos para gerar aleatoriamente um grafo bloco

Seja $G = (V, E)$ um grafo bloco, $\mathbb{Q} = \{Q_1, Q_2, \dots, Q_l\}$ seu conjunto de cliques maximais e $\mathbb{S} = \{S_1, S_2, \dots, S_g\}$ o seu conjunto de separadores minimais de vértices. Para a

inserção de um vértice novo v , será sorteado um vértice já existente $v_0 \in V$, se existir. Dessa forma, v_0 ou será simplicial, ou separador de vértices. Para cada um desses casos, temos:

1. Se v_0 é simplicial:

- a. Acrescentar apenas a aresta (v_0, v) , criando uma nova clique $Q_{l+1} = \{v, v_0\}$ e criando também um novo separador minimal de vértices $S_{q+1} = \{v_0\}$.
- b. Acrescentar as arestas $(v, w), \forall w \in Q_j$ tal que $v_0 \in Q_j$, ou seja, aumentar o tamanho de uma clique já existente, adicionando um vértice adjacente a todos os vértices da clique Q_j . Dessa forma, $Q_j = Q_j \cup \{v\}$. Observar que neste caso os separadores são os mesmos que existiam antes da inserção do novo vértice.

2. Se v_0 é um separador de vértices:

- a. Acrescentar apenas a aresta (v_0, v) , criando uma nova clique $Q_{l+1} = \{v, v_0\}$.

Para a geração dinâmica, precisamos manter as seguintes informações: as cliques maximais do grafo e a classificação dos vértices em dois tipos, simplicial e pertencente a um smv . Considere a seguinte notação para classificar os vértices. Seja $clique_i^0 = \{v \in V : v \in Q_i \text{ e } v \text{ é simplicial}\}$ e $clique_i^1 = \{v \in V : v \in Q_i \text{ e } v \text{ é separador}\}$. Por outro lado, para cada índice $1 \leq i \leq li$, $clique_i = clique_i^0 \cup clique_i^1$. Utilizando essa notação, pode-se representar os vértices dos conjuntos $clique_i^0$ e $clique_i^1$ por pares ordenados (a, b) , onde a é o vértice v e $b = j$, com $v \in clique_i^j, j \in \{0, 1\}$.

Essa estrutura de dados permite acessar uma clique e obter os vértices que pertencem a essa clique. Para poder fazer a inserção de um novo vértice mantendo o grafo bloco de maneira eficiente, é necessária uma outra estrutura que faça o oposto: dado um vértice, a quais cliques este pertence. Para isso, pode-se utilizar uma lista $Cliques$ que, para cada vértice v , $Cliques(v) = \{i : 1 \leq i \leq l, v \in Q_i\}$. De acordo com o Teorema 2.2, se $|Cliques(v)| = 1$, então v é simplicial. Por consequência, se $|Cliques(v)| > 1$, então v é separador.

Para ilustrar esses casos, observe o grafo bloco da Figura 3.1. Seja $l = 4$ o número de cliques maximais onde $Q_1 = \{a, b, c, d\}$, $Q_2 = \{c, e, f\}$, $Q_3 = \{c, g, h, i, j\}$ e $Q_4 = \{i, k, l, m\}$ e $q = 2$ o número de separadores minimais sendo $S_1 = \{c\}$ e $S_2 = \{j\}$ do grafo G .

Para o grafo inicial, as configurações das estruturas para $clique_i$ e $Cliques(v)$ estão nas Tabelas 3.1 e 3.2, respectivamente:

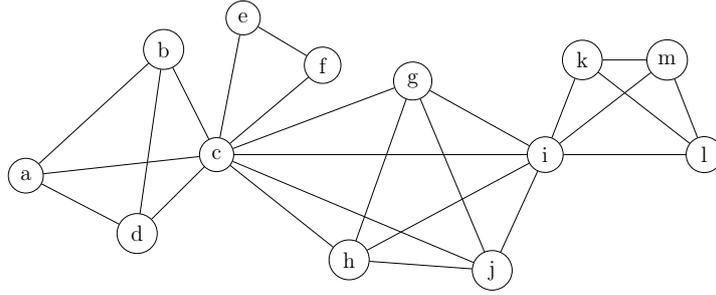


FIG. 3.1: Grafo inicial

i	clique_i
0	(a,0), (b,0), (c,1), (d,0)
1	(c,1), (e,0), (f,0)
2	(c,1), (g,0), (h,0), (i,1), (j,0)
3	(j,1), (k,0), (l,0), (m,0)

TAB. 3.1: *cliques_vertices* do grafo inicial

v	a	b	c	d	e	f	h	i	j	k	l	m	n
cliques(v)	{0}	{0}	{0,1,2}	{0}	{1}	{1}	{2}	{2,3}	{2}	{2}	{3}	{3}	{3}

TAB. 3.2: *Cliques* do grafo inicial

Para o caso de sortear um vértice simplicial, vamos supor que o vértice sorteado seja o vértice d (ou seja, $v_0 = d$) e, além disso, estamos acrescentando o vértice n (ou seja, $v = n$). Dessa forma, estaremos acrescentando apenas a aresta (d, n) , aumentando em 1 o número de cliques ($Q_{4+1} = \{d, n\}$) e adicionando um novo separador minimal de vértices ($S_{2+1} = \{d\}$). A inserção desse novo vértice e a nova aresta está ilustrada na figura 3.2.

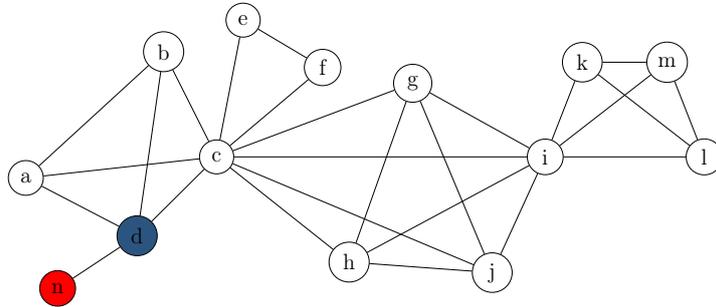


FIG. 3.2: Grafo do caso 1.a

Para o Caso 1.a, as configurações das estruturas para $clique_i$ e $Cliques(v)$ estão nas Tabelas 3.3 e 3.4, respectivamente:

i	clique_i
0	(a,0), (b,0), (c,1), (d,1)
1	(c,1), (e,0), (f,0)
2	(c,1), (g,0), (h,0), (i,1), (j,0)
3	(j,1), (k,0), (l,0), (m,0)
4	(d,1), (n,0)

TAB. 3.3: *cliques_vertices* do caso 1.a

v	a	b	c	d	e	f	h	i	j	k	l	m	n
cliques(v)	{0}	{0}	{0,1,2}	{0,4}	{1}	{1}	{2}	{2,3}	{2}	{2}	{3}	{3}	{4}

TAB. 3.4: *Cliques* do caso 1.a

Ainda sorteando um vértice simplicial, pode-se aumentar o tamanho de uma clique. Tomando $v_0 = d$, deve-se acrescentar as arestas (n, a) , (n, b) , (n, c) e (n, d) . Dessa forma tem-se uma nova clique $Q_1 = Q_1 \cup \{n\}$. Esse caso está ilustrado na figura 3.3:

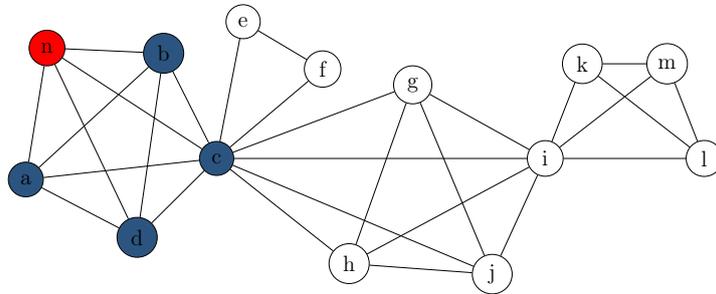


FIG. 3.3: Grafo do caso 1.b

Para o Caso 1.b, as configurações das estruturas para $clique_i$ e $Cliques(v)$ estão nas Tabelas 3.5 e 3.6, respectivamente:

i	clique_i
0	(a,0), (b,0), (c,1), (d,0), (n,0)
1	(c,1), (e,0), (f,0)
2	(c,1), (g,0), (h,0), (i,1), (j,0)
3	(j,1), (k,0), (l,0), (m,0)

TAB. 3.5: *cliques_vertices* do caso 1.b

v	a	b	c	d	e	f	h	i	j	k	l	m	n
cliques(v)	{0}	{0}	{0,1,2}	{0}	{1}	{1}	{2}	{2,3}	{2}	{2}	{3}	{3}	{0}

TAB. 3.6: *Cliques* do caso 1.b

Por fim, para o caso de sortear um vértice que seja separador minimal de vértices, suponha que o sorteado seja o vértice i ($v_0 = i$) e adiciona-se um novo vértice n e apenas a aresta (i, n) . Dessa forma, é criada uma nova clique $Q_{l+1} = \{i, n\}$, como pode-se ver na figura 3.4.

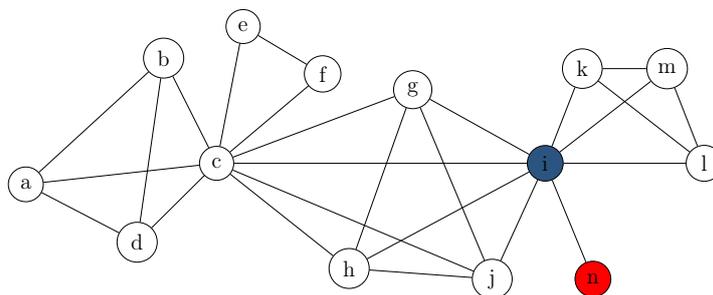


FIG. 3.4: Grafo do caso 2.a

Para o Caso 2.a, as configurações das estruturas para $clique_i$ e $Cliques(v)$ estão nas Tabelas 3.7 e 3.8, respectivamente:

i	clique_i
0	(a,0), (b,0), (c,1), (d,0)
1	(c,1), (e,0), (f,0)
2	(c,1), (g,0), (h,0), (i,1), (j,0)
3	(j,1), (k,0), (l,0), (m,0)
4	(i,1), (n,0)

TAB. 3.7: *cliques_vertices* do caso 2.a

v	a	b	c	d	e	f	h	i	j	k	l	m	n
cliques(v)	{0}	{0}	{0,1,2}	{0}	{1}	{1}	{2}	{2,3,4}	{2}	{2}	{3}	{3}	{4}

TAB. 3.8: *Cliques* do caso 2.a

O Algoritmo 3.2 representa o pseudocódigo do algoritmo proposto para a geração aleatória dinâmica, que foi desenvolvido segundo a seção anterior, onde serão utilizadas as seguintes notações:

- $random(i)$: gerador de um inteiro aleatório de 0 a i
- $list.add(i)$: adiciona o elemento i à lista $list$
- $list.change(i, j)$: na lista $list$, altera o elemento i para o elemento j
- $Cliques$: lista referente à estrutura de dados que armazena $Cliques(v)$
- $cliques_vertices$: lista referente à estrutura de dados que armazena $clique_i$

Algoritmo 3.2 Manutenção Dinâmica Incremental

Entrada: $cliques_vertices$ e $Cliques$

Saída: $cliques_vertices$ e $Cliques$ com a adição de 1 vértice.

Início

Se $|Cliques| = 0$ **então**

$Cliques.add(\{0\}); cliques_vertices.add(\{(0, 0)\})$

senão

$v \leftarrow random(|Cliques| - 1)$

Se $|Cliques[v]| > 0$ **então**

$cliques_vertices.add(\{(v, 1), (|Cliques|, 0)\});$

$Cliques[v].add(|cliques_vertices| - 1)$

$Cliques.add(\{|cliques_vertices| - 1\})$

senão

$operacao \leftarrow random(1)$

Se $operacao = 1$ **então**

$cliques_vertices.add(\{(v, 1), (|Cliques|, 0)\});$

$Cliques[v].add(|cliques_vertices| - 1)$

$Cliques.add(\{|cliques_vertices| - 1\})$

$cliques_vertices[Cliques[v][0]].change((v, 0), (v, 1));$

senão

$cliques_vertices[Cliques[v][0]].add((|Cliques|, 0));$

$Cliques.add(\{Cliques[v][0]\})$

Fim.

Uma forma alternativa realizar uma geração estática aleatória, partindo do grafo vazio e utilizando o Algoritmo Manutenção Dinâmica Incremental é:

Algoritmo 3.3 Geração Estática Aleatória

Entrada: número de vértices n ;

Saída: $cliques_vertices$ e $Cliques$

Início

$cliques_vertices \leftarrow \langle \rangle$; $Cliques \leftarrow \langle \rangle$;

Enquanto $n > 0$ **faça**

 Executa Algoritmo Manutenção Dinâmica Incremental

$n \leftarrow n - 1$

Fim.

3.2 PROJETO DA INTERFACE

Foram escolhidos os algoritmos de reconhecimento de grafos cordais e bloco (apresentados no Capítulo 2) e os de geração aleatória estática e dinâmica de grafos (apresentados no Capítulo 3) para implementar no sistema.

3.2.1 IDENTIFICAÇÃO DOS REQUISITOS

Para desenvolver o trabalho proposto será necessário implementar algoritmos de reconhecimento e geração de grafos cordais, bem como o reconhecimento e a geração de grafos pertencentes a uma subclasse, os grafos bloco. Também desenvolver um sistema para manipular os grafos e os correspondentes algoritmos.

O sistema seguirá o processo descrito a seguir. O usuário terá a sua disposição constantemente a escolha de qual tipo de algoritmo será aplicado, são eles: reconhecimento e geração (estática ou dinâmica). O fluxo seguinte dependerá desta escolha.

- Caso tenha optado por reconhecimento, o usuário deverá escolher a qual classe será verificado seu pertencimento e por fim o grafo em si. As representações aceitas para grafo serão lista de adjacência e matriz de adjacência. Com isso, identificaremos se o grafo pertence a tal classe selecionada e retornaremos ao usuário.
- Caso tenha optado por geração, o usuário deverá escolher o número de vértices e o tipo de algoritmo a ser aplicado, ou seja, estático ou dinâmico. Com isso, exibiremos um grafo gerado nas condições selecionadas, representado tanto como lista de adjacência e matriz de adjacência.

O sistema foi pensado de forma que o usuário não precise sair da mesma tela, disponibilizando todas as opções no mesmo ambiente. Assim, o usuário pode ir navegando de modo iterativo. Por exemplo, pode solicitar a geração de um grafo de determinada classe e em seguida verificar se o grafo gerado pertence a outra classe. O mesmo campo será usado tanto para exibição quanto para a entrada da representação do grafo.

A Figura 3.5 mostra o processo descrito anteriormente, para o qual os seguintes requisitos são necessários:

Requisito 1. Permitir ao usuário alterar a representação do grafo para o reconhecimento e exibir a geração nos dois formatos (lista e matriz).

Requisito 2. Permitir que o usuário escolha a classe de grafo, número de vértices e tipo de algoritmo a ser aplicado, quando for necessário para a operação solicitada.

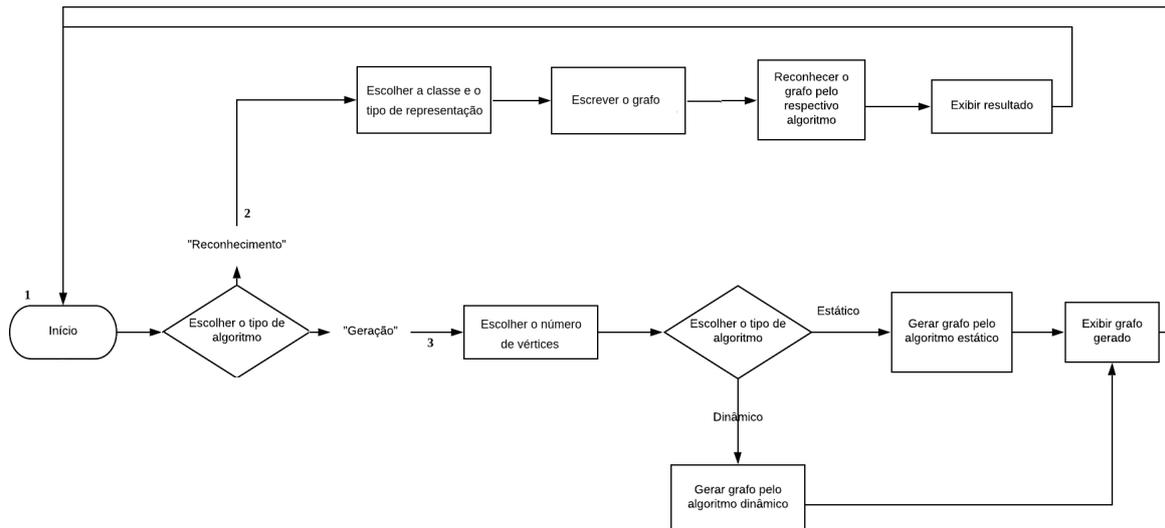


FIG. 3.5: Fluxo do programa

Requisito 3. Permitir que o usuário visualize uma mensagem de resultado ou o grafo gerado, se houver.

Requisito 4. Permitir que o usuário solicite outra operação, iterando em cima do mesmo grafo quantas vezes desejar, de forma que ele possa editar o grafo exibido como resultado de uma operação, por exemplo, para servir de input para outra operação.

Requisito 5. Permitir que o usuário reconheça se um grafo pertence a uma classe.

Requisito 6. Permitir que o usuário gere um grafo utilizando algoritmo estático.

Requisito 7. Permitir que o usuário gere um grafo utilizando algoritmo dinâmico.

Além dos requisitos 7 apresentados, o sistema deverá obedecer à seguinte regra:

Regra I. Só serão aceitas representações de grafo como lista de adjacência ou matriz de adjacência, e não será feita nenhuma validação se o formato é compatível com a representação escolhida, no caso de input do usuário.

Os requisitos acima foram divididos em quatro grandes grupos para a implementação das funcionalidades:

- Grupo I: Criar arcabouço do sistema, contendo os requisitos de 1 a 4;
- Grupo II: Reconhecimento de um grafo em uma classe, contendo o requisito 5;
- Grupo III: Geração de um grafo de forma estática, contendo o requisito 6; e
- Grupo IV: Geração de um grafo de forma dinâmica, contendo o requisito 7.

De acordo com esses requisitos, consegue-se identificar 4 casos de uso principais para

o sistema:

Caso de Uso 01: Reconhecimento de Grafo Cordal

Objetivo: Reconhecer se um grafo de entrada é Cordal ou não.

Requisitos: -

Ator: Usuário

Fluxo Principal:

1. O sistema fornece as opções de Reconhecimento/Geração.
2. O usuário seleciona Reconhecimento.
3. O sistema fornece as opções de Cordal/Bloco.
4. O usuário seleciona Cordal.
5. O sistema fornece as opções de Lista/Matriz para representação do grafo.
6. O usuário seleciona a opção que desejar.
7. O usuário insere o grafo no formato selecionado.
8. O usuário clica no botão "Go".
9. O sistema informa uma mensagem com "É Cordal"/"Não é Cordal".

Caso de Uso 02: Reconhecimento de Grafo Bloco

Objetivo: Reconhecer se um grafo de entrada é Bloco ou não.

Requisitos: -

Ator: Usuário

Fluxo Principal:

1. O sistema fornece as opções de Reconhecimento/Geração.
2. O usuário seleciona Reconhecimento.
3. O sistema fornece as opções de Cordal/Bloco.
4. O usuário seleciona Bloco.
5. O sistema fornece as opções de Lista/Matriz para representação do grafo.
6. O usuário seleciona a opção que desejar.
7. O usuário insere o grafo no formato selecionado.
8. O usuário clica no botão "Go".
9. O sistema informa uma mensagem com "É Bloco"/"Não é Bloco".

Caso de Uso 03: Geração Estática de grafo bloco

Objetivo: Realizar a geração estática de grafo bloco

Requisitos: -

Ator: Usuário

Fluxo Principal:

1. O sistema fornece as opções de Reconhecimento/Geração.
2. O usuário seleciona Geração.
3. O usuário insere o número de vértices a serem gerados.
4. O sistema fornece as opções de Estático/Dinâmico para o tipo do algoritmo.
5. O usuário seleciona Estático.
6. O usuário clica no botão "Go".
7. O sistema mostra o grafo gerado em lista e matriz de adjacência.

Caso de Uso 04: Geração Dinâmica de grafo bloco

Objetivo: Realizar a geração dinâmica de grafo bloco

Requisitos: -

Ator: Usuário

Fluxo Principal:

1. O sistema fornece as opções de Reconhecimento/Geração.
2. O usuário seleciona Geração.
3. O usuário insere o número de vértices a serem gerados.
4. O sistema fornece as opções de Estático/Dinâmico para o tipo do algoritmo.
5. O usuário seleciona Dinâmico.
6. O usuário clica no botão "Go".
7. O sistema mostra o grafo gerado em lista e matriz de adjacência.

3.2.2 ESCOLHA DAS FERRAMENTAS COMPUTACIONAIS

Neste capítulo são apresentadas algumas das ferramentas computacionais utilizadas, bem como algumas considerações observadas nas etapas de identificação das funcionalidades e levantamento de requisitos.

Primeiramente, o sistema foi implementado como uma página web, de forma a facilitar a entrada de dados e a exibição dos resultados. Foi descartado o uso de um aplicativo Desktop por conta da complexidade de tratar diferentes sistemas operacionais, além da maior dificuldade para instalação por parte do usuário. De modo geral, o sistema pretende evitar qualquer barreira para o usuário, na hora de resolver os problemas de reconhecimento e geração de grafos pertencentes a uma classe específica.

Para implementar os algoritmos em grafos, bem como suas estruturas de dados avançadas, foram utilizadas as linguagens C/C++. A vantagem principal no uso destas linguagens é a facilidade de implementar tais estruturas de dados de forma mais eficiente que em outras linguagens.

De modo geral, visando utilizar os conhecimentos adquiridos durante a graduação, a codificação do serviço web foi feita em AngularJS, utilizando conhecimentos de CSS e HTML, além de um servidor em Node.js, que chama os arquivos com a implementação dos algoritmos em grafos.

4 IMPLEMENTAÇÃO

Após a escolha dos requisitos e as ferramentas a serem utilizadas para implementá-los, deu-se início ao desenvolvimento da interface propriamente dita. O presente capítulo visa apresentar o funcionamento por trás da interface desenvolvida.

Esta seção é de extrema importância para entender o funcionamento da parte do projeto de interface, especialmente para auxiliar futuras continuações e aperfeiçoamentos que este projeto possa ter.

4.1 ALGORITMOS DE RECONHECIMENTO

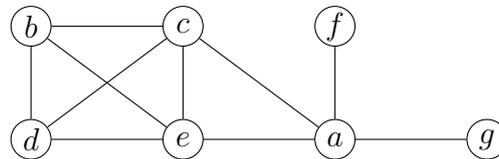


FIG. 4.1: Grafo cordal não bloco

Para acompanhar a descrição das implementações apresentadas nas próximas subseções, o grafo da FIG 4.1 será utilizado como entrada em cada algoritmo.

4.1.1 RECONHECIMENTO DE GRAFO CORDAL

O reconhecimento de um grafo cordal se dá pela geração de uma sequência de vértices pelo percurso em largura lexicográfica, seguida da verificação se esta sequência é um eep. O grafo é cordal se e somente se esta sequência for um eep.

Por isso, esta seção irá tratar desses dois algoritmos necessários para o reconhecimento de um cordal.

4.1.1.1 REPRESENTAÇÃO DE UM GRAFO

Para armazenar o grafo de entrada $G = (V, E)$ e todos os grafos auxiliares necessários, será utilizada uma lista de adjacências.

Usamos uma lista duplamente encadeada para representar a lista de adjacência de cada vértice (FIG 4.2). Dessa forma, cada grafo fica representado apenas com n (quantidade de vértices) e um ponteiro para a estrutura de lista de adjacência criada, tendo assim complexidade de espaço de $O(n + m)$, sendo $|V| = n$ e $|E| = m$.

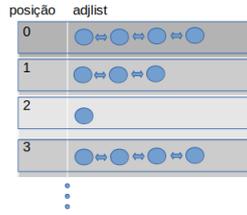


FIG. 4.2: Representação de um grafo

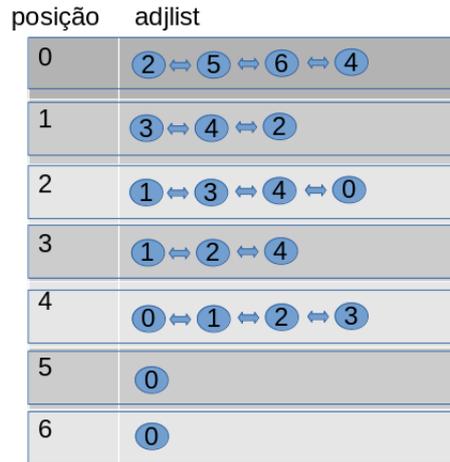


FIG. 4.3: Representação do grafo da FIG 4.1

A FIG 4.3 mostra a lista de adjacência para o grafo da FIG 4.1, onde cada índice segue a ordem alfabética de nomeação dos vértices.

4.1.1.2 PERCURSO EM LARGURA LEXICOGRÁFICA

O Algoritmo 2.1 permite, dado um grafo, obter uma sequência de vértices σ .

Para mantermos a linearidade deste algoritmo, utilizamos duas estruturas de dados a mais.

O principal gargalo na implementação para manter sua linearidade se encontra na escolha do vértice com maior *label* lexicograficamente, além da atualização dos rótulos de sua vizinhança. Manteremos então, além da própria lista de adjacência do grafo, duas estruturas adicionais a fim de contornar esse problema, chamadas de *VertexSet* e *SetPointer*.

A estrutura *VertexSet* é formada por listas duplamente encadeadas ordenadas.

Chamaremos cada lista dessas como *vertexSet*, que reúne vértices com o mesmo *label*, além de um campo adicional denominado *flag*.

A *VertexSet* mantém sempre suas listas lexicograficamente ordenadas de modo de-

crescente, inclusive após cada iteração.

Dessa forma, não precisamos armazenar o *label* do vértice. Apenas a posição na *VertexSet* do *vertexSet* ao qual pertence nos interessa, pois, na atualização, necessariamente criamos uma nova lista duplamente encadeada de vértices com o novo *label*. Mais ainda, esta lista ocupará uma posição imediatamente acima do *label* original, pois o novo rótulo difere em uma posição à direita do rótulo anterior. Veja também que, antes de uma modificação, o *label* do *vertexSet* imediatamente acima do *vertexSet* que sofre alteração necessariamente será maior que o novo *label* criado, pois eles continuam diferindo em um algarismo mais à esquerda.

VertexSet é inicializado como uma lista com um único elemento, contendo todos os vértices do grafo e sua *flag* zero, visto que todos os vértices iniciam com *label* vazio.

A segunda estrutura adicional, *SetPointer*, correlaciona cada vértice do grafo com seu *vertexSet* dentro da *VertexSet*, além de seu próprio nó dentro da *vertexSet*. Logo, mantemos uma estrutura com complexidade de espaço $O(n)$. *SetPointer* é implementado como uma lista de estruturas lineares cujos elementos contém os seguintes campos: *index*, *setPointer* e *node*. O *index* corresponde ao índice obtido pelo algoritmo, *setPointer* é um ponteiro para o *vertexSet* onde se encontra o vértice em questão e, por último, *node* é o ponteiro para o nó dentro deste *vertexSet*.

SetPointer é inicializado como uma lista com n elementos, com *index* igual a zero, *setPointer* apontando para o único *vertexSet* criado e *node* apontando para o seu próprio nó.

Quando um vértice v é escolhido pelo Algoritmo 2.1 para ser numerado, removemos o mesmo do *vertexSet* ao qual pertence. Por isso, armazenamos para cada vértice, um ponteiro para o *vertexSet* ao qual pertence (campo *setPointer*), além do ponteiro para o próprio nó em seu *vertexSet* (campo *node*), a fim de manter essa operação em $O(1)$.

Ainda na estrutura auxiliar *SetPointer*, guardamos no campo *index* a posição do vértice na sequência σ , à medida que ele vai sendo escolhido, ordenação essa que corresponde à numeração do vértice dada como saída do Algoritmo 2.1.

No final de cada iteração, deve ser atualizado o rótulo de cada vizinho w de v . Lembre-se que um nó ser atualizado significa que ele irá para um novo *vertexSet* imediatamente superior. Além disso, todos os vértices a serem atualizados que estiverem na mesma estrutura de *vertexSet* original devem ser movimentados para um mesmo *vertexSet*. Por isso, utilizamos a *flag* para identificar quando já foi inserido um novo *vertexSet* em determinada iteração.

Por exemplo, vamos supor que o *vertexSet* posicionado em quarto lugar (última

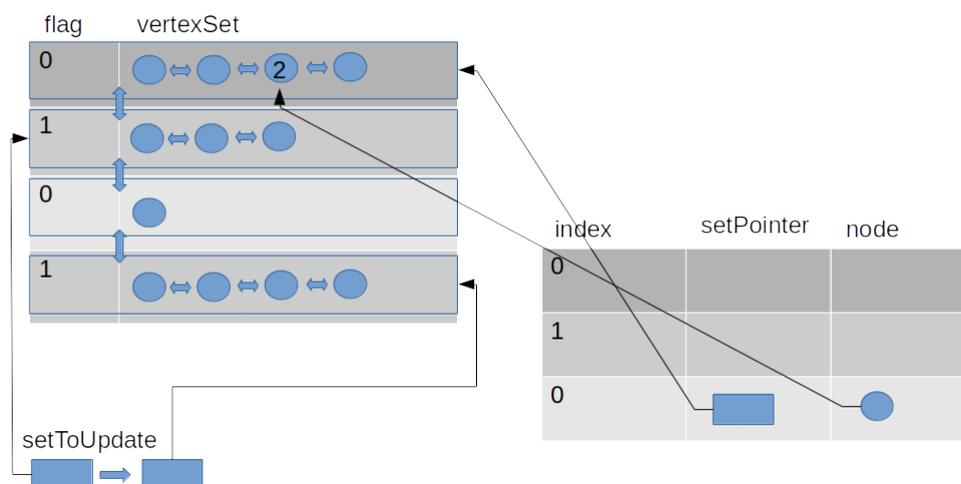


FIG. 4.4: Estruturas de dados usadas no percurso em largura lexicográfica

lista encadeada da estrutura *VertexSet* à esquerda da FIG 4.4) contenha dois nós cujos rótulos serão atualizados. Na implementação, o primeiro desses nós exigirá a criação de uma nova lista *vertexSet* imediatamente acima, entre a terceira e quarta posição da estrutura *VertexSet*. Este *vertexSet* criado constará apenas do primeiro nó com rótulo atualizado.

A seguir, na atualização do segundo nó, observe que ele terá o mesmo *label* do primeiro nó. Logo, deverá entrar no *vertexSet* recentemente criado. Nesse caso, utilizamos o campo *flag* da nova lista para guardar a informação de que já existe a lista encadeada criada nesta iteração.

Mantemos essa estrutura de *flag* e *vertexSet* como uma lista duplamente encadeada, a fim de podermos inserir e excluir elementos em $O(1)$.

Por último, ao final de cada iteração, precisamos igualar todos os campos *flag* em *VertexSet* para zero, para que sejam devidamente usados na nova iteração para indicar listas já criadas. Por isso, precisamos manter uma lista encadeada para manter essa operação em $O(1)$ (na FIG 4.5, ilustramos essa estrutura como *setToUpdate*).

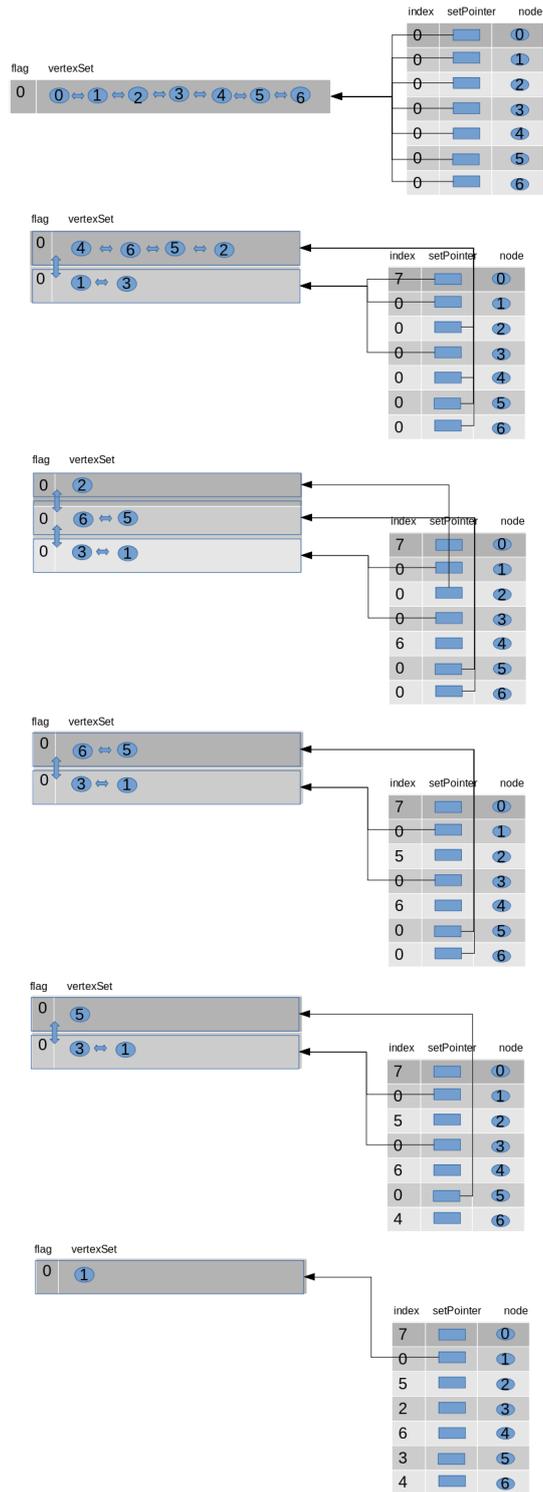


FIG. 4.5: Estruturas do percurso em largura lexicográfica para o grafo da FIG 4.1

A Figura 4.5 exibe cada passo do Algoritmo 2.1 usando o grafo da FIG 4.1 como entrada, mostrando as estruturas de dados *SetPointer* e *VertexSet* usadas na implemen-

tação.

Na última iteração, o vértice de índice 1, equivalente ao vértice b no grafo original, será retirado e o algoritmo termina com sequência de saída $\sigma = \langle b, d, f, g, c, e, a \rangle$. A sequência σ obtida pelo Algoritmo 2.1 depende da ordenação dos vértices na lista de adjacência.

4.1.1.3 VERIFICAÇÃO DE UM EEP

Este algoritmo não exige uma estrutura complexa, apenas um novo vetor de listas duplamente encadeadas para cada vértice (equivalente à representação de um grafo pelas suas listas de adjacência).

Primeiramente, determinamos o conjunto de adjacência monótona $X_\sigma(v) = \{x \in N(v) | \sigma^{-1}(v) < \sigma^{-1}(x)\}$ percorrendo a lista de adjacência do próprio vértice do grafo e comparando seu *index*.

Para achar o vértice da adjacência monótona com menor índice em σ , basta percorrer a própria adjacência monótona, o que preserva a linearidade.

Ao final, para checar se a adjacência monótona possui algum elemento que não se encontra na adjacência, criamos um vetor de adjacência, de forma que essa última consulta seja verificada com complexidade constante.

4.1.2 RECONHECIMENTO DE GRAFO BLOCO

O reconhecimento de um grafo bloco se dá pelo reconhecimento de grafo bloco, seguido da determinação dos conjuntos de separadores minimais de vértices. Por fim, um grafo será bloco se e somente se todos os smv forem conjuntos unitários.

O algoritmo de verificação se estes conjuntos são unitários não exige nenhuma estrutura de dados extra, apenas percorre todos os smvs gerados e verifica seu tamanho.

Por isso, esta seção irá tratar apenas desse outro algoritmo adicional necessário para o reconhecimento de um bloco, isto é, a determinação dos smvs.

4.1.2.1 DETERMINAÇÃO DO CONJUNTO DOS SMV E SUAS MULTIPLICIDADES

Para mantermos a linearidade do algoritmo, utilizamos as seguintes estruturas de dados:

Um vetor de inteiros, *position*, de tamanho n , que armazena qual foi a primeira clique maximal onde cada vértice entrou.

Um vetor de inteiros, *last*, de tamanho n , que armazena o índice do último smv de cardinalidade correspondente à posição neste vetor. Por exemplo, se criarmos um novo

smv S_2 de cardinalidade 4, vamos colocar na posição $last[3]$ o valor 2 correspondente ao seu índice no conjunto de separadores minimais de vértices.

Um vetor de listas de adjacência, $qSet$, também de tamanho n , que armazena todas as cliques maximais. Perceba que o maior número de cliques maximais possíveis é n , por isso usamos como tamanho do vetor.

Por fim, criamos uma estrutura auxiliar smv , composta do próprio conjunto de vértices do separador, sob a forma de uma lista de adjacência, além da multiplicidade do separador. Utilizamos um vetor $smvSet$ cujos elementos são da estrutura smv , descrita anteriormente, novamente de tamanho n , por conta do número máximo de separadores minimais de vértice em um grafo.

Na FIG 4.6, essas estruturas mencionadas são apresentadas visualmente.

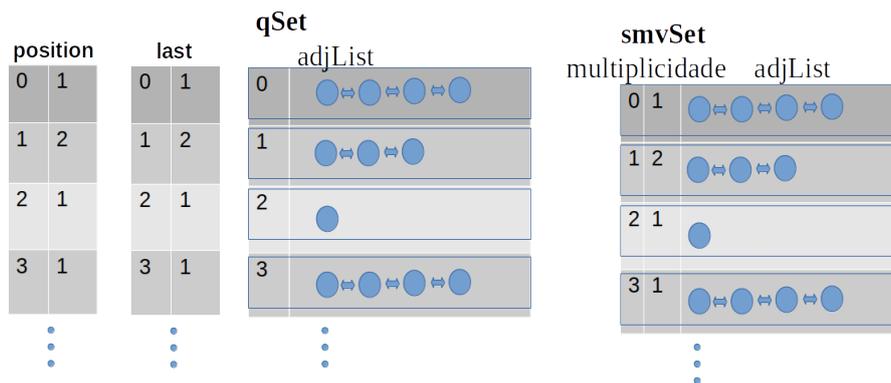


FIG. 4.6: Estruturas de dados usadas na determinação de \mathbb{S} e multiplicidades

Na FIG 4.7, mostramos o passo a passo de como as estruturas de dados se comportam durante a implementação do algoritmo, usando o grafo G_1 como exemplo.

Ao final, obtemos $\mathbb{S} = \{\{0\}, \{4, 2\}\}$, com multiplicidades 2 e 1, respectivamente.



FIG. 4.7: Estruturas do algoritmo de determinação de \mathcal{S} e suas multiplicidades para o grafo da FIG 4.1

4.2 ALGORITMOS DE GERAÇÃO DE GRAFOS BLOCO

Esta seção irá tratar da geração aleatória de um grafo bloco, a partir da quantidade de vértices desejada.

4.2.1 GERAÇÃO ESTÁTICA

Este algoritmo não exige uma estrutura complexa, usando apenas a estrutura de representação de um grafo já exibida.

Primeiramente, criamos um grafo com a quantidade de vértices desejada, sem nenhuma aresta entre os vértices. Conforme o Algoritmo 3.1, iterativamente é sorteada uma quantidade de vértices, dentre os que ainda não tiverem aresta, criando-se uma nova clique, além do sorteio de qual vértice será feita a ligação entre a clique e os demais vértices da iteração anterior.

4.2.2 GERAÇÃO DINÂMICA

Conforme a Seção 3.1.2, serão utilizadas as duas estruturas: uma que mapeia os vértices nas cliques e a outra que diz, para cada clique, quais vértices a ela pertencem. Será fornecido o número de vértices e sobre esse valor será aplicado o Algoritmo 3.2.

4.3 A INTERFACE DESENVOLVIDA

Durante a implementação, foram feitas adaptações nos requisitos, para facilitar o funcionamento ou que melhorem consideravelmente a interação com o usuário.

A seguir, alguns exemplos serão utilizados para demonstrar um caso de uso da interface, ressaltando detalhes da implementação julgados importantes.

4.3.1 PÁGINA INICIAL

A página inicial apresenta a opção principal de escolha entre "Reconhecimento" ou "Geração", como mostra a FIG. 4.8. No fluxograma da FIG 3.5, esta página representa o bloco 1, denominado Início. Por padrão, no primeiro acesso à página, a opção "Reconhecimento" estará escolhida.

Repare que adicionamos um campo abaixo da área de texto em que o usuário irá inserir tanto a Lista de Adjacência quanto a Matriz de Adjacência, a fim de mostrar um exemplo do formato esperado de entrada.

Em todos os casos, o botão "Go" irá acionar o algoritmo solicitado.

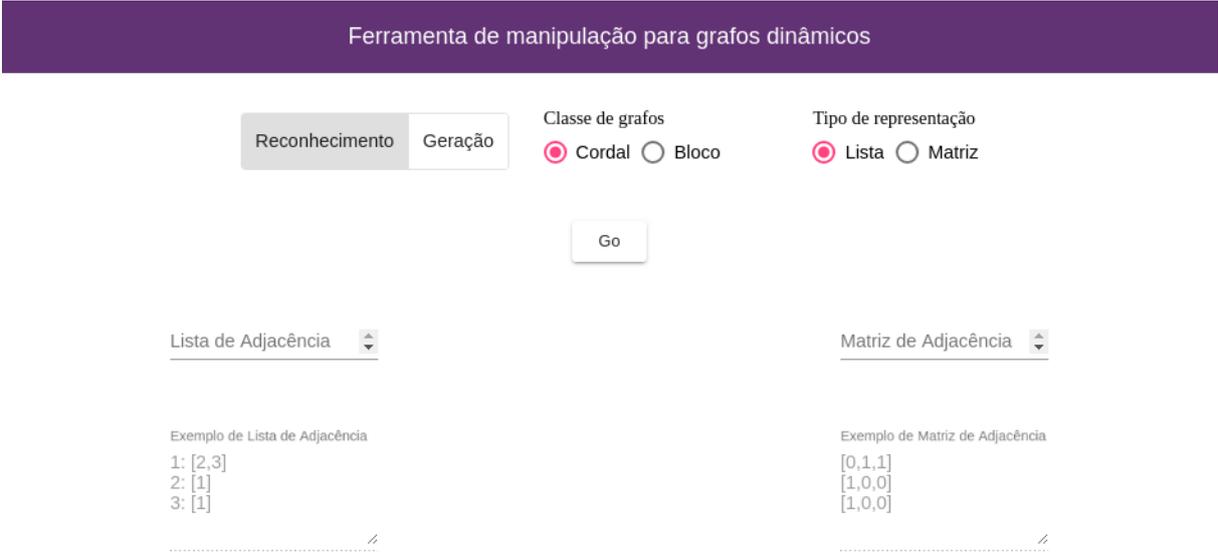


FIG. 4.8: Página Inicial

4.3.2 RECONHECER UM GRAFO A UMA CLASSE

No fluxograma da FIG 3.5, esta página representa o fluxo fruto da escolha em 2.

Neste caso, o usuário deverá clicar no botão "Reconhecimento", que já está como padrão na abertura da página, selecionar a Classe de grafos e o Tipo de representação. Por fim, dependendo do Tipo de representação escolhido, o usuário irá escrever o grafo a ser reconhecido na área respectivamente destinada.

A resposta do algoritmo, ou seja, se o grafo pertence ou não à classe, será exibido na tela como a forma de uma caixa de diálogo, como mostra a FIG. 4.10.

Abaixo, vamos simular alguns exemplos de comportamento.

Primeiramente, vamos testar se o grafo da FIG. 4.9 representado como lista de adjacência é um grafo cordal. O resultado está exibido na FIG. 4.10.

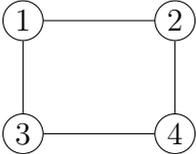


FIG. 4.9: Grafo não cordal

A FIG. 4.12 mostra o resultado do reconhecimento do grafo da FIG. 4.11 representado como matriz de adjacência. Repare que o grafo é cordal, mas a FIG. 4.13 mostra o mesmo grafo aplicando o reconhecimento para bloco. De fato, o grafo é cordal, mas não é bloco.

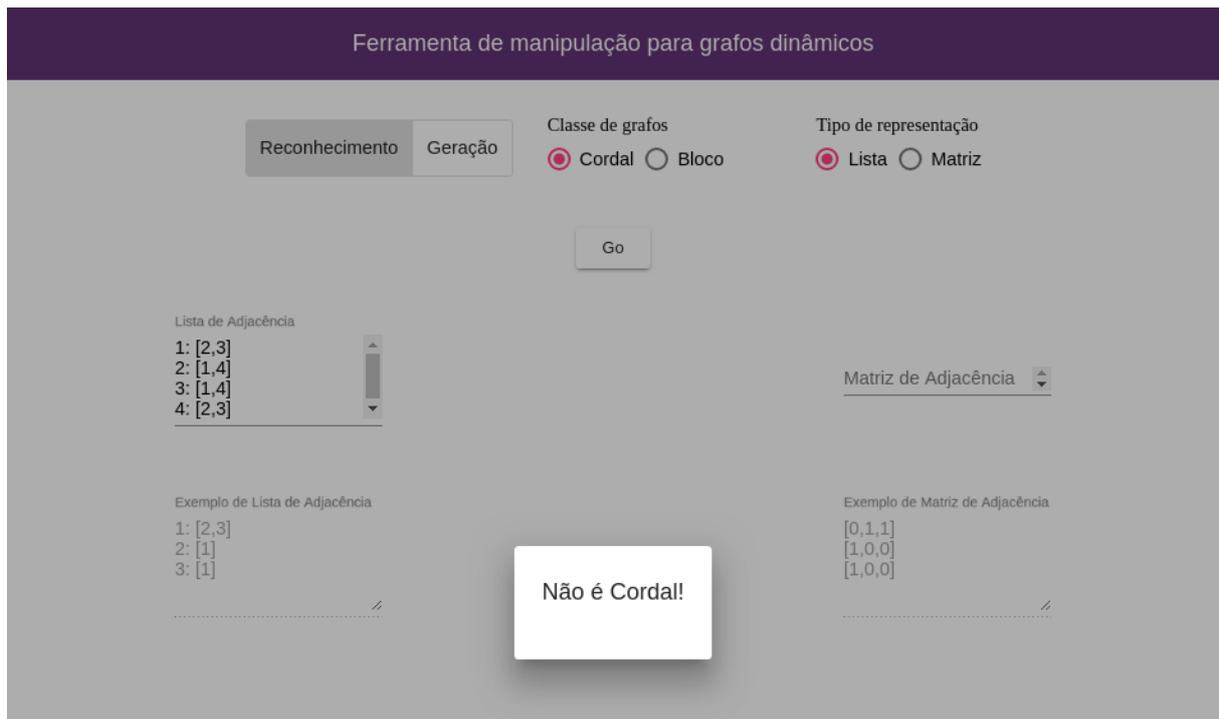


FIG. 4.10: Página de reconhecimento de um grafo não cordal representado como lista de adjacência

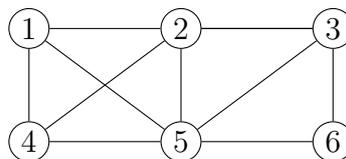


FIG. 4.11: Grafo cordal não bloco

4.3.3 GERAR UM GRAFO BLOCO

No fluxograma da FIG 3.5, esta página representa o fluxo fruto da escolha em 3.

Neste caso, o usuário deverá clicar no botão "Geração", inserir o número de vértices e o tipo de algoritmo desejado.

O resultado da operação será exibido tanto na forma de lista de adjacência, quanto matriz de adjacência.

A FIG 4.14 mostra a geração de um grafo bloco com 7 vértices, usando o algoritmo estático. No caso, o grafo gerado é o grafo da FIG 4.15, que de fato é bloco.

Por fim, a FIG 4.16 mostra a geração de um grafo bloco também com 7 vértices, usando o algoritmo dinâmico. No caso, o grafo gerado é o grafo da FIG 4.17, que de fato é bloco.

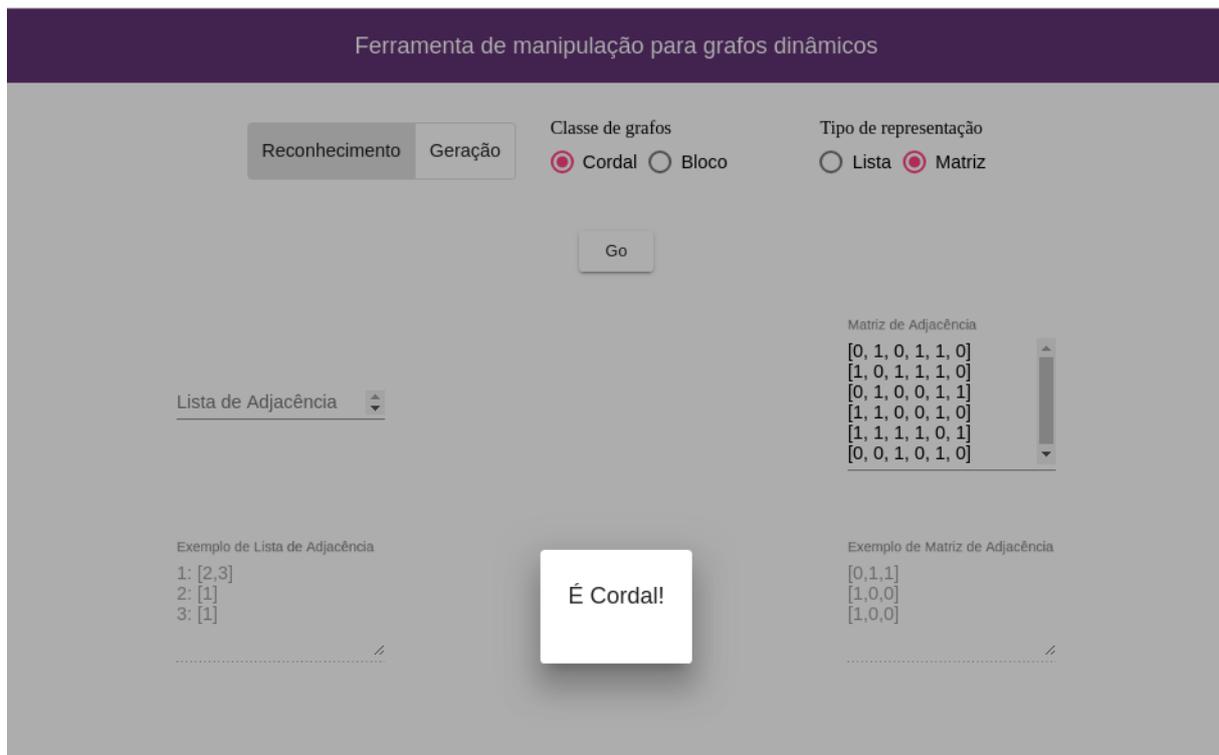


FIG. 4.12: Página de reconhecimento de um grafo cordal representado como matriz de adjacência

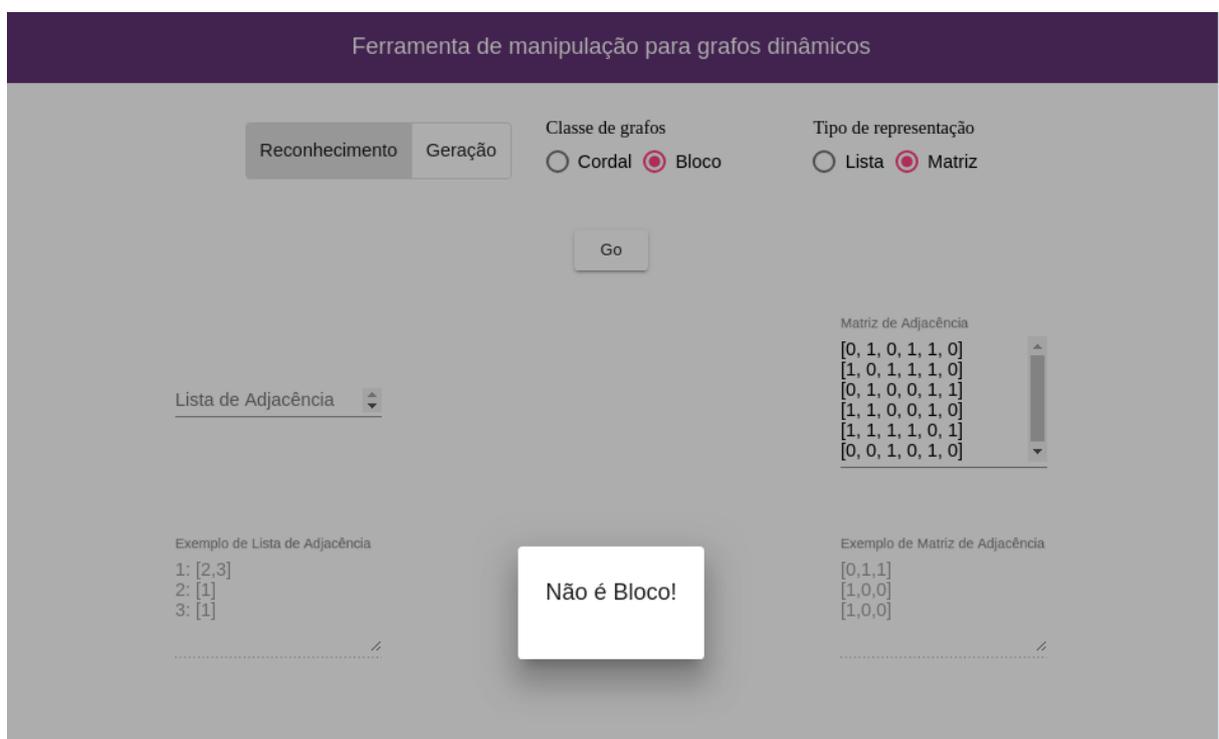


FIG. 4.13: Página de reconhecimento de um grafo não bloco representado como matriz de adjacência

Ferramenta de manipulação para grafos dinâmicos

Reconhecimento
Geração

Número de vértices
 7

Tipo de algoritmo
 Estático Dinâmico

Go

Lista de Adjacência

```

1: [6, 5, 4, 2]
2: [3, 1]
3: [2]
4: [1, 6, 5]
5: [7, 1, 6, 4]
6: [1, 5, 4]
7: [5]

```

Exemplo de Lista de Adjacência

```

1: [2,3]
2: [1]
3: [1]

```

Matriz de Adjacência

```

[0, 1, 0, 1, 1, 1, 0]
[1, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 1, 1, 0]
[1, 0, 0, 1, 0, 1, 1]
[1, 0, 0, 1, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0]

```

Exemplo de Matriz de Adjacência

```

[0,1,1]
[1,0,0]
[1,0,0]

```

FIG. 4.14: Página de geração de um grafo bloco usando o algoritmo estático

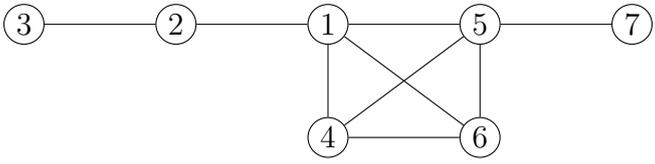


FIG. 4.15: Grafo bloco gerado estaticamente

Ferramenta de manipulação para grafos dinâmicos

Reconhecimento

Geração

Número de vértices

Tipo de algoritmo

 Estático Dinâmico

Lista de Adjacência

```

0: [1, 2, 3, 4, 5]
1: [0, 2, 3, 5]
2: [0, 1, 3, 5]
3: [0, 1, 2, 5]
4: [0, 6]
5: [0, 1, 2, 3]
6: [4]
        
```

Matriz de Adjacência

```

[0, 1, 1, 1, 1, 1, 0]
[1, 0, 1, 1, 0, 1, 0]
[1, 1, 0, 1, 0, 1, 0]
[1, 1, 1, 0, 0, 1, 0]
[1, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0]
        
```

Exemplo de Lista de Adjacência

```

1: [2,3]
2: [1]
3: [1]
        
```

Exemplo de Matriz de Adjacência

```

[0,1,1]
[1,0,0]
[1,0,0]
        
```

FIG. 4.16: Página de geração de um grafo bloco usando o algoritmo dinâmico

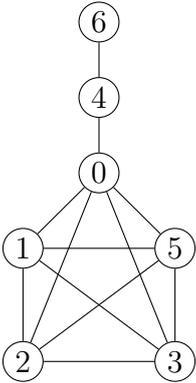


FIG. 4.17: Grafo bloco gerado dinamicamente

5 CONCLUSÃO

A contribuição deste trabalho foi o desenvolvimento de um sistema que permite manipular algoritmos em grafos cordais e blocos, disponibilizando uma interface de fácil utilização para usuários de diferentes domínios, que não necessita de conhecimento de programação.

O sistema foi projetado a partir de uma interface que permite o reconhecimento e a geração aleatória de grafos correspondentes a duas classes: cordal e bloco. Para implementar o sistema foi necessário aprofundar os conhecimentos teóricos sobre grafos cordais e grafos blocos, suas caracterizações e algoritmos eficientes para reconhecimento e geração aleatória. A interface foi desenvolvida utilizando AngularJs e NodeJs. Todos os algoritmos utilizados foram implementados em C e C++.

Como trabalho futuro podemos mencionar a extensão da interface para reconhecer e gerar outras subclasses de grafos cordais, assim como para incluir outros algoritmos relacionados com problemas computacionais restritos à classe de grafos cordais e suas subclasses.

6 REFERÊNCIAS BIBLIOGRÁFICAS

- MARKENZON, L.; WAGA, C. F. E. M. **Ferramentas Estruturais em Grafos Cor-dais**. São Carlos, SP: SBMAC, 2016.
- MEHTA, D. P.; SAHNI, S., editores. **Handbook of Data Structures and Appli-cations**. [S.l.]: Chapman and Hall/CRC, 2004. ISBN 978-1-58488-435-4.
- NANNICINI, G.; LIBERTI, L. Shortest paths on dynamic graphs. **International Tran-sactions in Operational Research**, v. 15, n. 5, p. 551–563, 2008.
- NETO, G. Problemas em grafos dinâmicos: outerplanaridade e aplicações. **Dissertação (Mestrado em Engenharia de Defesa) - Instituto Militar de Engenharia**, v. 1, n. 1, p. 1–52, 2017.
- RAMALHO, B. Relatório final de iniciação científica. **PIBITI-CNPq, Engenharia de Computação, Instituto Militar de Engenharia**, v. 1, n. 1, p. 1–52, 2017.
- RAMALHO, B. Relatório final de iniciação científica. **PIBITI-CNPq, Engenharia de Computação, Instituto Militar de Engenharia**, v. 1, n. 1, p. 1–24, 2018.
- ZAROLIAGIS, C. D. Implementations and experimental studies of dynamic graph algo-rithms. **Lecture Notes Comput Sci**, v. 2547, p. 229–278, 2002.