

Universidade Federal Rural do Rio de Janeiro
Instituto Multidisciplinar
Ciência da Computação
Grafos e Algoritmos

LABORATÓRIO DE GRAFOS E ALGORITMOS

Gabriel Marinho
João Pedro Ribeiro de Moura
Li Victor Lucas
Nickolas da Rocha Machado
Pedro Conrado

Nova Iguaçu
28 de Novembro
2019

Conteúdo

1	Introdução	2
2	Definições	2
3	Implementação	3
3.1	Força Bruta	3
3.2	Funções do Algoritmo Força Bruta	4
3.3	Algoritmo Otimizado (Busca em Largura Lexicográfica) . . .	6
3.4	Funções do Algoritmo Otimizado	7
4	Parte 2	8
4.1	Como foi feito?	8
5	Testes e métricas	9
5.1	Ambiente de teste	9
5.2	Testes	9
6	Instruções de uso	11
6.1	Editor de Grafos	11
6.2	Linux	11
6.3	Windows	12
7	Referências	12
8	Agradecimentos	12

Resumo

Este artigo tem como objetivo mostrar e discutir sobre os métodos por nós implementados de Reconhecimento de Grafos Cordais e suas aplicações. Ademais, é apresentada definições e detalhes sobre os métodos utilizados, tais como as funções que foram utilizadas e também, informações sobre a implementação, na qual foi utilizada a linguagem Python na sua versão 3.7.

1 Introdução

Primeiramente, é apresentado o algoritmo de Força Bruta, no qual o algoritmo faz literalmente um “Força Bruta”, ou seja, verifica todas as possibilidades, uma por uma, verificando se a propriedade do grafo “ser cordal” é verdadeira ou não em todos os ciclos do grafo. Posteriormente, o algoritmo nos dá a resposta se o grafo que foi estudado, é cordal ou não. Ademais, foi implementada também um algoritmo otimizado que utiliza da Busca Lexicográfica em Largura para o reconhecimento de Grafos Cordais, na qual gera uma ordenação de vértices de acordo com a quantidade de vizinhos que cada vértice tem.

2 Definições

- **Grafo Cordal:** Um grafo é denominado cordal (ou triangularizado) quando todo ciclo de comprimento maior do que 3 possui uma corda (Szwarcfiter, J. Teoria Computacional de Grafos. Cap 4).

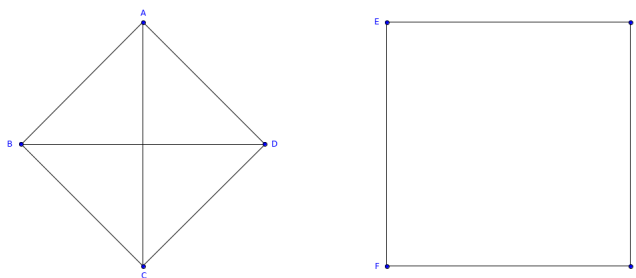


Figura 1: **Grafo Cordal à esquerda e o não cordal à direita.**

- **Passeio:** uma sequência v_1, v_2, \dots, v_k de vértices de V com $V_i V_j \in E$.
- **Ciclo:** Ciclo é um passeio fechado $v_1, v_2, \dots, v_k = v_1$, onde v_1, v_2, \dots, v_{k-1} é um **caminho**.
- **Busca em Largura Lexicográfica:** Uma busca em largura para um grafo não direcionado $G(V, E)$ é lexicográfica quando para todo

$v \in V$ e $w, w_1 \in A^*(v) = \{w \in N(v) \mid w \text{ não foi marcado}\}$, se w é mais remoto do que w_1 , ou seja, existe um vértice z adjacente a w e não a w_1 , que foi alcançado na busca antes de qualquer vértice z' adjacente a w_1 e não a w ; então w é explorado antes de w_1 na busca. Esta prioridade diminuirá a liberdade de escolha dos vértices, mas nem sempre a eliminará, ou seja, a prioridade não conduzirá a escolha necessariamente única de um vértice.

- **Esquema de Eliminação Perfeita:** Diz-se que uma sequência de vértices é um esquema de eliminação perfeita se cada vértice v_i for um vértice simplicial do subgrafo induzido por $\{v_i, v_{i+1}, \dots, v_n\}$, ou seja, para cada v_i , o subgrafo induzido por $A(v_i) = \{v_1, v_2, \dots, v_{i-1}\}$ é completo.
- **vert_inicial:** Vértice inicial.
- **v:** Vértice atual.
- **ciclo:** Ciclo atual.
- **ciclos:** Lista de ciclos.
- **n:** Número de vértices.
- **G9:** Grafo com 9 vértices.

3 Implementação

A implementação do algoritmo de Reconhecimento de Grafos Cordais foi desenvolvida e baseada na linguagem Python na sua versão 3.7. O método de ordenação utilizado no algoritmo de Força Bruta foi a **ordenação topológica**, e no algoritmo otimizado, foi utilizado o método da **Busca em Largura Lexicográfica** em conjunto com o **Esquema de Eliminação Perfeita**.

3.1 Força Bruta

O algoritmo Força Bruta do Reconhecimento de Grafos Cordais, verifica todos os ciclos existentes no grafo em questão, e verifica se estes ciclos são ou não cordais.

Em seguida, é apresentada as funções usadas na implementação do Força Bruta, com o seu protótipo (pseudo-código) e uma breve explicação da função.

3.2 Funções do Algoritmo Força Bruta

Algoritmo 1: achaCiclo (vert_inicial, v, lst_aberta, inicio, ciclo, ciclos)

```
if inicio == False and v == vert_inicial then
    if ta_dentro(ciclo,copy()) == False then
        | adiciona o ciclo na lista
    end if
    retorna
end if
for aresta em lista_aberta do
    if v em aresta then
        cria uma cópia da lista_aberta
        remove aresta atual da cópia da lista
        atribui os vértices da aresta a 2 variaveis a e b
        if v é igual ao primeiro elemento da aresta e o segundo não
            está no ciclo ou é o vert inicial then
            | adiciona o segundo elemento (b) em ciclo
            | chama achaCiclo() com o segundo elemento como vértice
            | inicial
            | remove o segundo elemento do ciclo
        end if
    else
        if v é o segundo elem da aresta e o primeiro nao está no
            ciclo ou é o vert inicial then
            | adiciona o primeiro elemento no ciclo
            | chama achaCiclo() com o primeiro elemento como
            | vértice inicial
            | remove o primeiro elemento do ciclo
        end if
    end if
end if
end for
```

Algoritmo 2: trataCiclo(ciclos)

```
for ciclo em ciclos do
    if se tamanho do ciclo é maior ou igual a 4 then
        | adiciona o ciclo achado
    end if
end for
retorna lista
```

Algoritmo 3: identificaCiclo(edge_list,n)

adiciona 0 na fila

while *fila* $\neq \emptyset$ **do**

 coloca 0 no vértice atual e tira o 0 da fila

for *arestas na lista de arestas* **do**

if *se aresta não está em lista_aberta* **then**

if *se vértice está em aresta* **then**

 adiciona a aresta em *lista_aberta*

if *se vértice é o primeiro elemento da aresta* **then**

 chama achaCiclo() com o segundo vértice da
 aresta como vértice inicial e atual

end if

else

 adiciona o segundo elemento na fila

 visitados[segundo elemento] = 1

end if

end if

else

if *vértice é o segundo elemento da aresta* **then**

if *primeiro elemento da aresta está na fila* **then**

 chama achaCiclo() com o primeiro vértice
 como vértice inicial e atual

end if

else

 adiciona o primeiro elemento na fila

 visitados[primeiro elemento] = 1

end if

end if

end if

end for

if *fila* == \emptyset **then**

for *i = 1 até i = n* **do**

if *visitados[i] == 0* **then**

 adiciona i na fila

 visitados[i] = 1

end if

end for

end if

end while

retorna os ciclos tratados

Algoritmo 4: `is_chordal_brute(n)`

```
chama função que identifica os ciclos
gera lista de adjacência
for ciclo em ciclos do
    for for vértice em ciclo do
        grau = 0
        for vizinho em N(v) do
            if vizinho em ciclo then
                grau = grau + 1
            end if
        end for
        if grau maior ou igual a 3 then
            quebra
        end if
    end for
else
    retorna False, ciclo
end if
end for
retorna True
```

3.3 Algoritmo Otimizado (Busca em Largura Lexicográfica)

O algoritmo otimizado do Reconhecimento de Grafos Cordais, baseia-se no uso do método da Busca em Largura Lexicográfica, onde é retornado um conjunto de vértices ordenados pela quantidade de vizinhos a ele associados.

Esse algoritmo também é baseado no Teorema dos Grafos Cordais, no qual mostra que:

Teorema: Um grafo $G(V, E)$ é cordal se e somente se G possuir um esquema de eliminação perfeita.

Lema: Seja $G(V, E)$ um grafo cordal aplicado ao algoritmo de busca em largura lexicográfica. Então a sequência S de vértices v ordenados decrescentemente segundo $\text{largura}(v)$ é um esquema de eliminação perfeita.

Em seguida, é apresentado as funções usadas na implementação do Algoritmo Otimizado, com o seu protótipo (pseudo-código) e uma breve explicação da função.

3.4 Funções do Algoritmo Otimizado

Algoritmo 5: get_lexBfs()

```
for  $v \in V$  do
  |  $R(v) = \emptyset$ 
end for
for  $j = n$  até  $j = 1$  do
  | escolher  $v \in V$ :  $s^{-1}(v)$  e lexicograficamente maximo  $s(j) = v$ 
  | for  $w \in Adj(v)$ :  $s^{-1}(w)$  não definido fazer do
  | | incluir  $j$  a direita em  $R(w)$ 
  | end for
end for
```

Algoritmo 6: elimPerfeita(grafo, correção = **None**)

```
lex = lista de vértices ordenados pela função lexBfs()
for  $i = 1$  até  $i = n$  do
  | adiciona um conjunto vazio na lista L
end for
for  $v \in lex$  do
  | if rotulo de  $v$  não esteja vazio then
  | | seleciona o vértice  $u$  com menor valor de largura
  | | adiciona na lista do vértice  $u$ , o conjunto de vértices a serem
  | | verificados
  | end if
  | else
  | | quebra
  | end if
end for
for  $i = 1$  até  $i = n$  do
  | if lista do vértice, menos seus adjacentes não for vazia then
  | | adicione o vértices problemáticos
  | | if numero dos vértices estiver errado then
  | | | return False, e o conjunto de vértices que formam o ciclo
  | | | proibido
  | | end if
  | | else
  | | | retorna False , e o conjunto de vértices que formam o
  | | | ciclo proibido
  | | end if
  | end if
end for
retorna True
```

Algoritmo 7: *is_chordal*

```
for componente  $\in$  componentes do
  verifica se há uma eliminação perfeita
  if se não houver eliminação perfeita then
    | retorna problema
  end if
end for
retorna True
```

O algoritmo otimizado para verificar se um grafo é cordal parte da premissa que o grafo é conexo, por conta da busca *lexBfs*.

Contudo, na implementação aqui realizada, há um pré processamento para que as componentes conexas sejam verificadas separadamente, e assim pode-se verificar se um grafo não conexo é cordal da mesma maneira.

4 Parte 2

O nosso objetivo, foi inferir a interferência da otimização do processo de reconhecimento do grafo cordal no reconhecimento de outra classe de grafo, o grafo de intervalo.

O Grafo de Intervalo tem várias aplicações tais quais análise de cadeias de DNA e seriação arqueológica.

Existem muitos estudos sobre grafos e ordens de intervalo, tanto pelo interesse puramente teórico, quanto pelo papel central que desempenham em certas aplicações.

Eles surgem em muitas aplicações práticas que requerem a construção de uma linha do tempo onde, a cada evento relacionado ao problema, corresponde um intervalo representando a sua duração.

Conforme, dentre tais aplicações, estão aquelas relacionadas a planejamento, alocação de tarefas, arqueologia, lógica temporal, diagnóstico médico e desenho de circuitos integrados.

Há também aplicações não relacionadas com eventos numa linha de tempo. Nesta categoria, estão aplicações na área de genética, problema do mapeamento físico de DNA e psicologia comportamental.

Eles surgem em muitas aplicações práticas que requerem a construção de uma linha do tempo onde, a cada evento relacionado ao problema, corresponde um intervalo representando a sua duração.

4.1 Como foi feito?

Por definição um grafo é um grafo de intervalos se e somente se, esse grafo é cordal e AT-Free, e para um grafo ser AT-Free, ele não possui tripla

asteroidal. Uma tripla asteroidal é definida da seguinte forma: Três vértices de um grafo formam uma tripla asteroidal, se para todos dois vertices eles são conectados por um caminho evitando a vizinhança do terceiro.

Logo foi feita uma função adicional para verificar se o grafo é AT-Free. Assim podemos checar se um grafo é um grafo de intervalos observando os retornos das funções "is_chordal" e "is_at_free" se as duas forem validas, o retorno da função "is_interval_graph", que é a função que checa o que foi discutido acima, é verdadeiro.

5 Testes e métricas

Abaixo, é exibido os resultados obtidos nos testes, o ambiente de teste usados no desenvolvimento do projeto e as métricas obtidas.

5.1 Ambiente de teste

- **Processador:** Intel Core i7-7700HQ
- **Memória:** 16GB DDR4
- **Sistema Operacional:** Ubuntu 19.04

5.2 Testes

Os testes foram feitos usando os algoritmos de Reconhecimento de Grafos desenvolvidos nesse projeto no ambiente de teste acima definido. A seguir serão apresentados os resultados obtidos nos testes em cada grafo utilizado.

	Mémoria	Tempo de Execução
K5	0mb	0,000601s
K10	0mb	3m 48s
K20	0mb	mais de 20m
K100	0.82mb	indeterminado
K200	1mb	indeterminado
G9	0mb	0,0002s
G10	0mb	0,006321s
G18	0mb	0,00325s

Figura 2: Tabela com os resultados obtidos no Força Bruta.

	Mémoria	Tempo de Execução
K5	0mb	0,0005s
K10	0mb	0,00012
K20	0mb	0,0002s
K100	0mb	0,005214s
K200	0,2mb	0,0421s
G9	0mb	0,00005s
G10	0mb	0,00005s
G18	0mb	0,0002s

Figura 3: Tabela com os resultados obtidos no Algoritmo Otimizado.

Em alguns testes, o uso de memória foi bem baixo, por exemplo, no teste do K10 onde foi obtido um uso de memória na ordem de $2 \cdot 10^{-6}$, sendo assim, aproximando assim para 0.

6 Instruções de uso

Abaixo, será apresentado o passo a passo de como executar os algoritmos nos sistemas operacionais Linux e Windows. Foi implementado também um editor de grafos, para gerar o arquivo de texto que será usado no algoritmo de Reconhecimento de Grafos Cordais. Para rodar o editor é preciso fazer os seguintes passos.

Para facilitar o uso do algoritmo de Reconhecimento de Grafos Cordais, foi implementado também um Editor de Grafos, no qual o usuário desenha graficamente o grafo, no qual se quer usar como entrada do algoritmo de Reconhecimento de Grafos Cordais. Logo abaixo, é mostrado instruções para o uso desse Editor.

6.1 Editor de Grafos

Ao iniciar o Editor, temos ao fundo uma tela branca vazia, e os seguintes comandos podem ser feitos:

- Ao apertar a tecla V é selecionada a opção de adicionar vértice, e ao clicar com o mouse em qualquer lugar da tela, é adicionado um vértice no local.
- Ao apertar a tecla E é selecionada a opção de adicionar aresta, e para adicionar uma aresta, é necessário clicar em dois vértices consecutivamente para assim formar uma aresta entre eles.

Após desenhado o Grafo, ao lado direito terá dois botões, nos quais determinam se o grafo é cordal ou de intervalo. Ao apertar em algum desses botões, é verificado se o grafo desenhado é cordal, se for escolhido o primeiro botão, ou se o grafo é de intervalo, se for escolhido o segundo botão.

- **Grafo Cordal:** verifica se o Grafo é cordal ou não. Se a propriedade for verdadeira, o vértices do Grafo ficaram todos verdes, e se não for verdadeiro, mostra alguns vértices que ficam em preto do ciclo proibido.
- **Grafo de Intervalo:** Igual a opção de Grafo Cordal, se for verdadeira a propriedade, colore todos os vértices com verde, se não, colore todos os vértice com preto.

6.2 Linux

1. Caso não tenha o Java instalado, instale o openjdk-11.

2. Caso não tenha Python instalado, abra o terminal e use o comando **sudo apt-get install python3**
3. Abra o terminal com o comando CTRL + ALT + T e instale a biblioteca networkx com os seguinte comando: **pip3 install networkx**
4. Instale o Openjfx pelo comando: **sudo apt-get install openjfx**
5. Entre o na pasta do programa pelo terminal e rode **java -module-path linux/ -add-modules=javafx.controls -jar chordalgraph-0.0.1.jar**
6. Caso já tenha o java instalado em uma versão inferior, execute o comando **sudo update-alternatives --config java** e selecione a opção que marca o openjdk-11

6.3 Windows

1. Caso não tenha Python instalado, instale o Python 3.7.
2. Caso não tenha o Java instalado ou for uma versão inferior, instale o openjdk-11.
3. Copiar o conteúdo da pasta bin do **OpenJFX** e colar na bin do **OpenJDK11**
4. Rode o programa pelo comando **java -module-path windows -add-modules=javafx.controls -jar chordalgraph-0.0.1.jar**
5. Caso a versão do Java não estiver sendo alterada (por conta da versão antiga instalada), usar o caminho completo do **OpenJDK11** pra rodar o programa - **"C:\Program Files\Java\jdk11\bin\java -module-path windows -add-modules=javafx.controls -jar"**

7 Referências

- Teoria Computacional de Grafos Jayme Luiz Szwarcfiter.
- https://en.wikipedia.org/wiki/Lexicographic_breadth-first_search
- <https://www.cos.ufrj.br/uploadfile/1368208720.pdf>

8 Agradecimentos

Agradecemos a Prof(a). Fernanda Couto, por nos auxiliar no desenvolvimento do projeto e por explicar todas as dúvidas que tivemos sobre o mesmo.