Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Relatório final para o programa IC – Edital 01/2019

Algoritmos em Grafos

Aluno:
Lucas da Silva de Lima

Supervisor:
Carla Negri Lintzmayer

Setembro/2020

Resumo

Grafos são estruturas centrais em Ciência da Computação que usam um mesmo formalismo para representar diversas situações em diversas áreas. Identificar e formular problemas utilizando grafos é uma habilidade importante para qualquer bom projetista de algoritmos. O objetivo deste projeto foi introduzir o aluno à pesquisa científica por meio do estudo de importantes resultados em Teoria dos Grafos, que em geral, não são apresentados em disciplinas da graduação, e de implementações de alguns algoritmos importantes. Este relatório apresenta os resultados finais desta iniciação científica. Em particular, descrevemos o Algoritmo de Borůvka, que é utilizado para encontrar árvores geradoras mínimas, e o Algoritmo de LEC, utilizado para testes de planaridade em grafos biconexos. Implementamos ambos utilizando a linguagem Julia, visando contribuir com o desenvolvimento da linguagem e da biblioteca LightGraphs.

Sumário

1 Introdução

A Teoria dos Grafos se trata de um ramo da matemática e da computação que estuda relações entre objetos e para tal são definidas estruturas denominadas *grafos*.

A importância deste ramo e de seus algoritmos pode ser vista não apenas de forma teórica, mas também devido à grande quantidade de situações que podem ser modeladas por meio de grafos. Temos, por exemplo, o algoritmo *PageRank* utilizado pelo *Google Search* para ranquear as páginas resultantes de uma pesquisa e então mostrá-las em uma ordem adequada para o usuário.

Dessa forma, nos propusemos a estudar e a implementar alguns algoritmos em grafos, como forma de introdução à pesquisa em Ciência da Computação. Para atingir esses objetivos, nossa ideia foi contribuir com o crescimento do pacote LightGraphs [4], da linguagem Julia.

Esse relatório mostra os resultados finais de nossa pesquisa. Na Seção 2 descrevemos como se deu o estudo de alguns conceitos fundamentais de Estruturas de Dados e Análise de Algoritmos, visando obter maior compreensão em Teoria dos Grafos e seus algoritmos, para então implementá-los.

Na Seção 3 introduzimos conceitos fundamentais em Teoria dos Grafos para a compreensão do relatório. Em seguida, na Seção 4 discutimos sobre o Algoritmo de Borůvka [8], que resolve o problema de encontrar uma árvore geradora mínima. Tal algoritmo foi implementado e serviu de contribuição à biblioteca LightGraphs.

Na Seção 5 apresentamos algumas definições e propriedades importantes de grafos planares e introduzimos conceitos necessários para o entendimento do algoritmo para teste de planaridade. Primeiramente lidamos com um algoritmo que se utiliza de descrições combinatórias, explicado na Seção 6. Tal algoritmo foi implementado e também foi disponibilizado para ser agregado à biblioteca LightGraphs.

Em seguida, estudamos o Teorema de Kuratowski [1], por ser a grande base dos algoritmos correspondentes, como por exemplo o Algoritmo de Lempel, Even e Cederbaum (LEC). Dado que o Algoritmo de LEC se utiliza de uma lec-ordenação e de XY-caminhos, explicamos eles nas Seções 7 e 8 e apenas posteriormente explicamos o Algoritmo de LEC, na Seção 9.

Dado o objetivo de contribuir com a biblioteca LightGraphs, na Seção 10 falamos um pouco sobre a linguagem e sobre como se deu a contribuição para a mesma.

Por fim, temos nossas considerações finais na Seção 11.

2 Metodologia

Para os estudos e implementações de algoritmos relacionados a Teoria dos Grafos foi necessário, em primeiro momento, o estudo sobre Estruturas de Dados e Análise de Algoritmos. O estudo se deu através de disciplinas da própria universidade e, além disso, houve um aprofundamento sobre os tópicos com o livro *"Algorithms in C"* [8]. Com os conhecimentos adquiridos sobre ambos os assuntos, foi possível aplicá-los na decisão sobre quais estruturas seriam mais adequadas para cada implementação, dado que podemos reduzir a complexidade de um algoritmo e torná-lo mais eficiente, tanto em questão memória quanto de tempo, otimizando-o.

Também foi necessário o estudo sobre os conceitos em Teoria dos Grafos, dado que ainda não havia conhecimento prévio. Os conhecimentos adquiridos se deram, em primeiro momento, de forma introdutória, com a disciplina Análise de Algoritmos e, posteriormente, houve o aprofundamento com a leitura de partes do livro *"Graph Theory"* [1]. Além disso, para aumentar a familiaridade com implementações de algoritmos em grafos, e também para nos acostumarmos com a linguagem Julia, houve também a implementação de alguns algoritmos mais conhecidos, como *Busca em Profundidade*, *Busca em Largura*, e *Dijkstra*.

Em busca de contribuir com o crescimento da biblioteca LightGraphs [4] foi necessário o estudo da linguagem Julia, que se deu principalmente através da documentação presente no site oficial da linguagem [5]. Além da sintaxe da própria linguagem também foi necessário conhecer os recursos presentes nas próprias bibliotecas. Tais recursos se dão através de funções e estruturas de dados comumente utilizados em grafos, de modo a utilizar os recursos já presentes a fim de não recriar implementações, otimizando o tempo de trabalho e também dando uniformidade conforme os padrões formados nas implementações já contidas na biblioteca. Durante o estudo e implementações também se fez necessário recorrer a uma outra biblioteca parcialmente importada em LightGraphs, chamada DataStructures [3], a qual foi de suma importância para a implementação do Algoritmo de Borůvka e do Algoritmo de LEC que serão apresentados posteriormente neste relatório.

Como primeira contribuição para a biblioteca LightGraphs, houve o estudo e implementação do Algoritmo de Borůvka, que é utilizado para encontrar uma árvore geradora mínima em um dado um grafo com arestas de pesos distintos. Os conceitos envolvendo o algoritmo, incluindo conceitos fundamentais de Teoria dos Grafos necessários para o entendimento do mesmo, serão explicados na Seção 3.

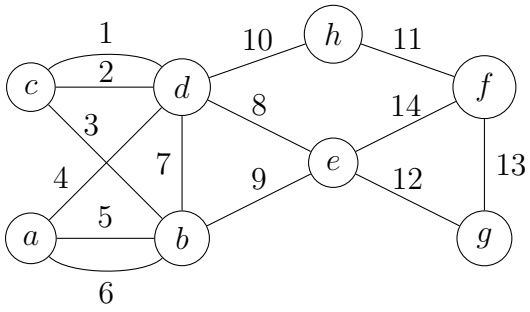


Figura 1: Representação gráfica de um grafo G para o qual $V(G) = \{a, b, c, d, e, f, g\}$, $E(G) = \{1, 2, \ldots, 14\}$ e ψ_G pode ser vista na figura (por exemplo, $\psi_G(3) = \{b, c\}$).

Além disso, os estudos sobre planaridade se deram, principalmente, através da leitura da dissertação de mestrado "*Análise experimental de algoritmos de planaridade*" [7], sendo de grande ajuda para a implementação do teste de planaridade por Descrições Combinatórias, implementação que foi submetida para integração com a biblioteca LightGraphs, e também para nossa implementação do algoritmo de Lempel, Even e Cederbaum, as quais serão discutidas nas Seções 6 e 9.

3 Conceitos Fundamentais em Teoria dos Grafos

Um grafo G é definido por um par-ordenado (V, E), onde V é um conjunto de elementos chamados *vértices* e E é um conjunto de elementos chamados de *arestas*, e uma função de incidência ψ_G que associa um par não-ordenado de vértices a uma aresta de G. Caso tenhamos uma função ψ_G que associa pares ordenados de vértices a uma aresta de G, temos então um *grafo direcionado*, também chamado de *digrafo*. Nesse caso, arestas costumam ser chamadas de *arcos*. Se $a \in E(G)$ e $\psi_G(a) = \{u, v\}$ (ou $\psi_G(a) = (u, v)$, no caso de digrafos), com $u, v \in V(G)$, dizemos que u e v são *extremos* da aresta a.

Grafos podem ser representados graficamente, utilizando círculos para representar os vértices e linhas para as arestas, como visto na Figura 1.

Em geral, dado qualquer grafo $J = (A, B)$, sempre denotamos por $V(J)$ o conjunto de vértices (no caso, $V(J) = A$) e por $E(J)$ o conjunto de arestas (no caso, $V(J) = B$). Dado um grafo G, note que $V(G)$ e $E(G)$ podem ser vazios e uma aresta pode ligar um vértice a ele mesmo.

As seções a seguir descrevem outros conceitos importantes e necessários neste relatório.

3.1 Grafos Simples

Uma aresta cujos extremos são idênticos se chama *laço*. Um grafo é considerado *simples* quando dois vértices são associados por, no máximo, uma aresta, ou seja, $\forall a, b \in E(G)((\psi_G(a) = \psi_G(b)) \iff (a = b))$. Nesse caso, note que a função ψ_G se torna dispensável, pois qualquer aresta pode ser unicamente identificada pelos seus extremos. Assim, se $a \in E(G)$ e $\psi_G(a) = \{u, v\}$, escreveremos apenas $a = uv$ quando o grafo for simples.

3.2 Incidência

Dizemos que um vértice é *incidente* a uma aresta caso o vértice seja um dos extremos da aresta. Dois vértices u e v são *adjacentes* entre si caso estejam conectados por uma aresta, ou seja, $\exists e \in E(G)(\psi_G(e) = \{u, v\})$. O *grau* de um vértice v é o número de arestas incidentes a v e é denotado por $deg(v)$. Note que um laço é contado duas vezes no grau de um vértice.

3.3 Passeios, Trilhas e Caminhos

Um *passeio* em um grafo G é uma sequência que alterna vértices e arestas, sendo w um passeio, então seria da forma $w = (v_1, e_1 \ldots, e_k, v_{k+1})$ onde todo e_i é uma aresta com extremos v_i e v_{i+1} para todo $1 \leq i < k$. Uma *trilha* é um passeio em que não se repetem arestas, isto é, $\forall\ i, j\ (e_i = e_j \iff i = j)$. Um *caminho* é um passeio em que não há repetição de vértices e, consequentemente, não há repetição de arestas, ou seja, temos que $\forall\ i, j\ ((e_i = e_j \vee v_i = v_j) \iff i = j)$.

Um passeio é *fechado* caso tenha comprimento (número de vértices contidos no passeio) positivo e o vértice de início da sequência seja idêntico ao último, ou seja, se $v_1, v_2, ..., v_k$ temos que $v_1 = v_k$. Um caminho fechado recebe um nome especial: *ciclo*.

3.4 Subgrafos

Dizemos que um grafo H é subgrafo de um grafo G se temos $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, com $E(H)$ contendo pares de vértices em $V(H)$. Um exemplo de um grafo e um subgrafo do mesmo é apresentado na Figura 2. Além disso, temos que um subgrafo de G é *gerador* se ele contém todos os vértices de G.

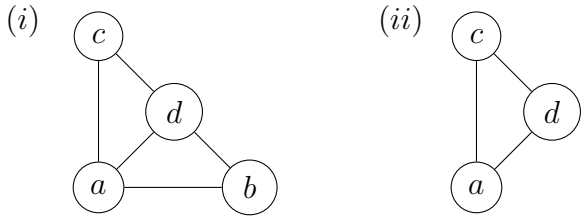


Figura 2: Exemplo de um grafo G em (i) e um subgrafo de G em (ii).

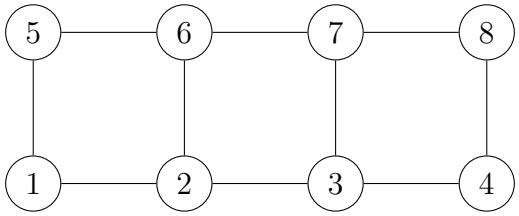


Figura 3: Exemplo de grafo biconexo.

3.5 Conexidade, Componentes e Blocos

Um grafo G é *conexo* se existe caminho entre quaisquer dois vértices distintos, caso contrário, ele é *desconexo*. Consideramos um grafo desconexo como um conjunto de *componentes* conexas, as quais são subgrafos conexos *maximais*. Podemos denotar o número de componentes de G por $c(G)$. Temos que uma aresta e é uma *aresta de corte* em G se sua remoção aumenta o número de componentes, o mesmo vale para vértices de corte, os quais são vértices tais que dada sua remoção temos um aumento no número de componentes.

Denominamos um grafo G como *k-vértice-conexo* ou *k-conexo* se ele tem mais de k vértices e permanece conexo sempre que são removidos $k-1$ vértices. Dessa forma, temos que um grafo é 2-conexo ou *biconexo*, por exemplo, se ele continua conexo ao removermos apenas um vértice. Tal caso é representado na Figura 3.

Além disso, temos que um *bloco* se trata de um subgrafo biconexo maximal, o que inclui ciclos e arestas, por exemplo.

3.6 Grafos Completos

Denominamos um grafo como *completo* quando para todo par u e v de vértices temos uma aresta, e somente uma, que os conecta. Grafos completos com n vértices são denotados por K_n. Temos na Figura 4 o grafo K_5.

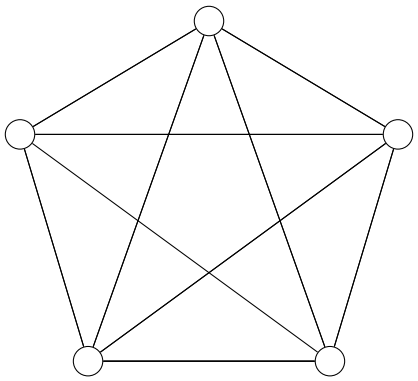


Figura 4: Grafo completo com 5 vértices, K_5.

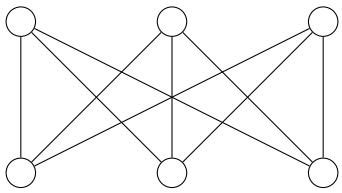


Figura 5: Grafo bipartido completo com 3 vértices em cada parte da bipartição, $K_{3,3}$.

3.7 Grafos Bipartidos

Grafos bipartidos são grafos G em que podemos dividir $V(G)$ em dois conjuntos A e B tais que $A \neq \emptyset$, $B \neq \emptyset$, $A \cup B = V(G)$, $A \cap B = \emptyset$, $\forall uv \in E(G)((u \in A) \wedge (v \in B))$. Chamamos (A, B) de bipartição de G.

3.8 Grafo Bipartido Completo

Um grafo é *bipartido completo* se é bipartido e para todo $u \in A$ e $v \in B$, u está conectado a v por uma única aresta. Grafos bipartidos completos com partições A e B, sendo $|A| = m$ e $|B| = n$ são denotados por $K_{m,n}$. Temos como exemplo grafo $K_{3,3}$ na Figura 5.

3.9 Árvores e Florestas

Uma *árvore* é um grafo conexo e *acíclico* (não contém ciclos). Se G é uma árvore, então G tem $|V(G)| - 1$ arestas. Um exemplo de árvore é dado na Figura 6. Uma *floresta* é um grafo acíclico. Assim, cada componente de uma floresta é uma árvore.

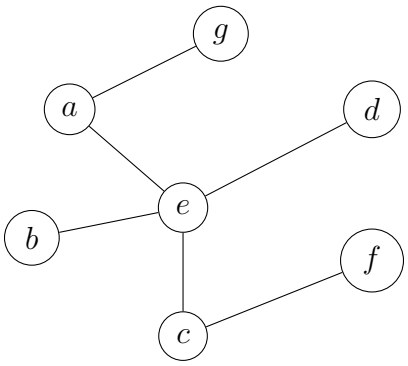


Figura 6: Exemplo de uma árvore.

3.10 Árvore Geradora Mínima

Uma *árvore geradora* de um grafo G se trata de um subgrafo que é uma árvore e que possui todos os vértices de G. Dado um grafo G e uma função $w : E(G) \to \mathbb{R}$ de pesos nas arestas, dizemos que uma árvore geradora T de G tem custo $w(T) = \sum_{e \in E(T)} w(e)$. Dessa forma, ao usarmos o termo *mínima* dizemos que dentre todas as árvores geradoras existentes, queremos a de menor custo.

3.11 Busca em Profundidade

A *Busca em Profundidade*, também conhecida por DFS, se trata de um algoritmo em que tendo um vértice u como inicial, visitamos todos os vértices v que tenham um uv-caminho em um dado grafo G, ou seja, todos os vértices alcançáveis à partir de u. Sendo $n = |V(G)|$ e $m = |E(G)|$, a complexidade do algoritmo é $O(n + m)$. Uma versão recursiva da Busca em Profundidade é apresentada no Algoritmo 1, onde temos a estrutura $visitado[1 \dots n]$ que, dado $v \in V(G)$, em $visitado[v]$ indica se o vértice v foi visitado.

Algoritmo 1: Busca em Profundidade(G, s)

Entrada: Grafo G e um vértice s de início

```
1 início
2     visitado[s] = 1
3     para cada v vizinho de s faça
4         se visitado[v] == 0 então
5             visitado[v] = 1
6             Busca em Profundidade(G, v)
```

4 Algoritmo de Borůvka

O algoritmo de Borůvka trata o problema de encontrar uma árvore geradora mínima de um grafo G quando todas as arestas possuem pesos distintos. O funcionamento do algoritmo de Borůvka é formalizado no Algoritmo 2, que é detalhado na Seção 4.1.

A ideia geral do algoritmo é sempre manter uma floresta geradora. Na primeira iteração, cada vértice é uma componente dessa floresta. A cada iteração, encontramos as arestas de menor custo que possuem extremos em componentes distintas e então adicionamos todas elas, de uma só vez, à floresta atual. Toma-se um cuidado para que as arestas adicionadas não gerem ciclos. Repetimos essas iterações até não haver mais arestas que conectam duas componentes, obtendo uma árvore geradora mínima para cada componente conexa do grafo original.

Em nosso pseudocódigo e em sua implementação, utilizamos a estrutura de dados *Union-find*. Tal estrutura, em suma, oferece a possibilidade de particionar ou classificar nossos dados (no caso, o conjunto de vértices) e também oferece as funções UNION, que une dois componentes ao receber dois vértices pertencentes a tais componentes como parâmetro, e a função FIND, que identifica a qual componente pertence um vértice recebido como parâmetro.

Vale observar que a forma clássica do algoritmo não funciona para grafos desconexos. Por isso, o algoritmo descrito na próxima seção e sua implementação foram modificados para lidar com essa situação.

4.1 Detalhes do Pseudocódigo

Começamos na linha 2 por iniciar uma floresta F utilizando uma estrutura de dados denominada *Disjoint Sets*, a qual começa particionando o conjunto em $|V(G)|$ árvores, uma para cada vértice de G, com cada vértice sendo o representante da sua própria árvore ou conjunto. Além disso, na linha seguinte inicializamos o conjunto solução MST como vazio, que durante a execução do algoritmo é preenchido com as arestas escolhidas.

Dentro do laço iniciado na linha 4, nós primeiramente inicializamos um vetor S que será responsável por armazenar em $S[i]$ a aresta de menor custo que conecta a componente que contém o vértice i a qualquer outra componente. Para manter isso, o laço da linha 6 percorre todas as arestas $e \in E(G)$. Se a aresta e que está sendo analisada possui peso menor que a aresta candidata armazenada em $S[i]$ ou se $S[i]$ for

Algoritmo 2: Algoritmo de Borůvka

Entrada: Grafo G com função w de peso em que todas arestas possuem pesos distintos

Saída: Conjunto de arestas que formam uma árvore geradora mínima em cada componente conexa de G

```
1  início
2      Inicializar uma floresta F composta de |V(G)| árvores de um único vértice
3      MST := ∅
4      enquanto houver arestas entre componentes de F faça
5          Crie um vetor S[1..|V(G)|] com elementos nulos
6          para cada e := uv ∈ E(G) faça
7              x := FIND(u)
8              y := FIND(v)
9              se x ≠ y então
10                 e1 := S[x]
11                 e2 := S[y]
12                 se e1 = nulo ou w(e) < w(e1) então
13                     S[x] := e
14                 se e2 = nulo ou w(e) < w(e2) então
15                     S[y] := e
16         para cada v ∈ V(G) faça
17             se S[v] ≠ nulo então
18                 xy := S[v]
19                 se FIND(x) ≠ FIND(y) então
20                     UNION(x, y)
21                     MST := MST ∪ {xy}
22     retorna MST
```

nulo, então armazenamos e em $S[i]$.

No laço da linha 16, fazemos de fato a escolha por quais arestas inserir. Na linha 19, nos utilizamos do valor-verdade de $\text{FIND}(x) \neq \text{FIND}(y)$ para verificar se a aresta xy armazenada em $S[v]$ de fato une duas árvores diferentes. Caso seja verdade, então unimos essas componentes e adicionamos a aresta à nossa solução.

Dessa forma, chegará um momento em que não há mais arestas entre duas ou mais componentes distintas de F, caso em que temos uma árvore geradora para cada componente conexa de G, e então retornamos nossa solução.

Com melhorias no algoritmo é possível utilizá-lo em grafos com arestas de peso repetido. Para isso, precisaríamos adicionar uma função de *tie-breaking*, ou seja, um método consistente de decisão para quando há duas arestas de mesmo peso que conectam duas componentes durante a execução.

4.2 Complexidade do Algoritmo

A complexidade do algoritmo depende, principalmente, da forma como o Union-Find funciona. Consideraremos que temos um vetor P com $|V(G)|$ posições, de tal forma que em $P[i]$ temos o vértice representante do grupo que contém o vértice i, obtendo assim o representante em tempo $\Theta(1)$ com a função FIND. Para as uniões, vamos considerar que ao unir duas componentes A e B, atualizaremos os valores em P nos índices correspondentes aos vértices contidos na componente de menor tamanho. Assim, no pior caso temos que praticamente metade dos vértices de um grafo G têm seus representantes atualizados e UNION é $O(n)$. No melhor caso, temos de atualizar apenas o representante de um único vértice, sendo UNION $\Omega(1)$. Com isso, uma análise mais profunda da execução do UNION pode nos ajudar a encontrar uma complexidade mais precisa.

Analisando a complexidade do nosso algoritmo por partes podemos notar que, para um grafo G qualquer, considere $m = |E(G)|$ e $n = |V(G)|$. Primeiro note que a cada iteração do laço da linha 4, a quantidade de componentes da floresta é reduzida por um fator de, no mínimo, 2. Por isso, esse laço executa $O(\log n)$ vezes. Assim, o laço da linha 6 nos dá uma complexidade de $O(m \log n)$, pois temos m iterações sempre que o laço da linha 6 é executado.

O laço da linha 16, analisando superficialmente, nos daria o custo das n iterações multiplicadas pelo custo da função UNION, que é $O(n)$, e com isso poderíamos dizer que temos uma complexidade de $O(n^2)$ neste trecho. Com uma análise mais cuidadosa, uma

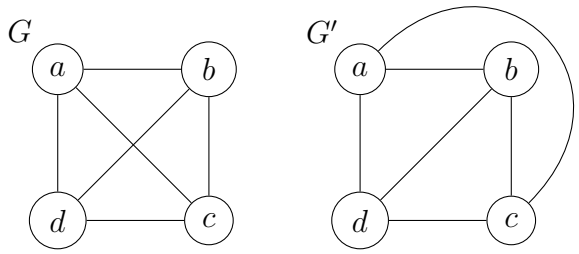


Figura 7: Grafo planar G e sua imersão plana G' (grafo plano).

execução da função UNION leva tempo $\Theta(t)$ sendo t o tamanho da menor componente dentre as componentes contendo os vértices x e y (linha 18), pois precisaríamos atualizar os representantes dos vértices da menor componente. Dado que na operação UNION apenas atualizamos os representantes da menor componente, então toda vez que isso acontece, os elementos da menor componente terminam em uma nova componente com, no mínimo, o dobro do tamanho da anterior. Dessa forma, todo vértice u terá seu representante atualizado por, no máximo, $\log n$ vezes. Assim, temos que o laço da linha 16 leva tempo $O(n \log n)$.

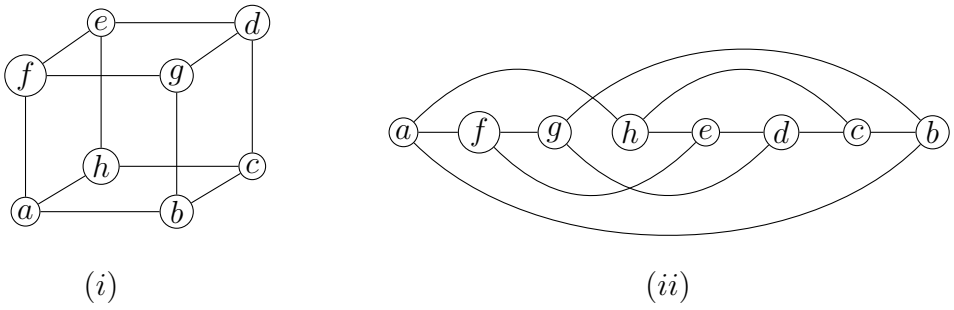
Na totalidade do algoritmo, obtemos tempo $O(m \log n) + O(n \log n)$ e, como geralmente, nos problemas de árvore geradora mínima, $m > n$, temos que o tempo é $O(m \log n)$.

5 Conceitos Fundamentais em Planaridade

Um grafo é *planar* quando é possível desenhá-lo no plano de tal forma que as arestas se encontrem apenas nos pontos correspondentes aos seus extremos em comum. O desenho em questão é chamado *imersão planar*. Um exemplo de grafo planar é dado na Figura 7. Iremos nos referir a uma imersão planar de um grafo planar de *grafo plano*.

5.1 Árvore de Busca em Profundidade e Bp-Ordenação

Uma *árvore de busca em profundidade*, também chamada de *bp-árvore*, é obtida ao realizarmos uma busca em profundidade em um grafo G de tal forma que $\forall uv \in E(G)$, com u e v visitados, temos que u e v possuem uma relação de sucessor e predecessor. Uma ordenação $v_1, v_2, v_n, \ldots, v_k$ de $V(G)$ é dita uma *bp-ordenação* de $V(G)$ se, $\forall v_i, v_j \in V(G)$, se $i > j$ então v_i é predecessor de v_j em uma bp-árvore T de G.



(i) (ii)

Figura 8: (i) Um grafo G. (ii) Uma lec-ordenação de G.

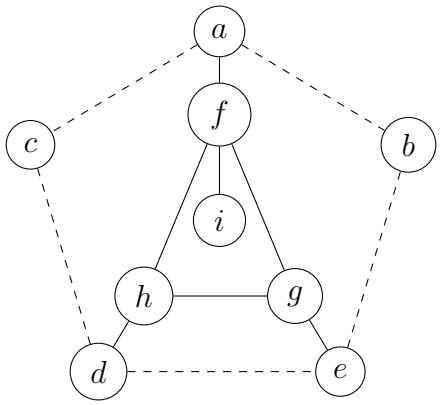


Figura 9: Um grafo G onde as linhas tracejadas representam uma moldura M de G.

5.2 Lec-Ordenação

Dado um grafo G, uma *lec-ordenação* se trata de uma ordenação dos vértices de G tal que para $\forall v_i \in v_1, v_2, \ldots, v_n$, os subgrafos $G[\{v_1, \ldots, v_i\}]$ e $G[\{v_{i+1}, \ldots, v_n\}]$ são conexos. Apenas podemos garantir que há uma lec-ordenação de um dado grafo G se G for biconexo. Temos um exemplo de uma lec-ordenação na Figura 8.

5.3 Molduras

Seja M um subgrafo de um grafo H. Dizemos que M é uma *moldura* de H se existe um desenho plano de H em que M é o grafo induzido pelas arestas adjacentes à face externa. Dessa forma, é evidente que um grafo possui uma moldura se e somente se é planar. Se H é um subgrafo induzido de um grafo G e M é moldura de H, temos então que M é uma *moldura de H em G* se $\forall v \in V(H)$ vizinho de $u \in V(G) \setminus V(H)$, v é um vértice de M. Com isso, podemos inferir que molduras são formadas apenas por ciclos e arestas de corte. Temos um exemplo de uma moldura na Figura 9.

5.4 Caminhos Perimetrais

Sendo M uma moldura de um dado grafo H, um caminho em M é denominado *perimetral*. Sejam $C_1, C_2, \ldots, C_k$ todos os blocos de M que contêm ao menos uma aresta em comum com um caminho perimetral P. Definimos, para $i = 1, 2, \ldots, k$, consideramos $P_i = P \cap C_i$ e $\overline{P_i} = P_i$ se C_i tem apenas uma aresta, caso contrário, $\overline{P_i} = C_i \setminus P_i$. Por fim, temos que $\overline{P} = \bigcup_{i=1}^{k} \overline{P_i}$ é denominado *caminho complementar* de P. Caso $E(P) = \emptyset$ então $\overline{P} = P$. Além disso, temos que a *base* de P é o subgrafo $\bigcup_{i=1}^{k} C_i$ de M. Se $E(P) = \emptyset$, então a base é P.

5.5 XY-Caminhos

Sejam A um subconjunto de vértices e B um subconjunto de arestas de um grafo G. Dizemos que v *enxerga* A *através de* B se existe um caminho em G com um extremo em v e outro extremo em $u \in A$ tal que internamente haja apenas arestas em B. Seja M uma moldura de um grafo conexo e sejam X e Y subconjuntos de $V(M)$. Um caminho perimetral P em M com base S é um XY*-caminho* se:

1. as pontas de P estão em X;
2. todo vértice em S que enxerga X através de $E(M) \setminus E(S)$ está em P;
3. todo vértice em S que enxerga Y através de $E(M) \setminus E(S)$ está em $\overline{P}$;
4. nenhuma componente de $M \setminus V(S)$ contém vértices em X e em Y.

Temos um exemplo de um XY-caminho na Figura 10.

5.6 XY-Obstruções

Dados uma moldura M de um grafo conexo e subconjuntos X e Y de $V(M)$, nem sempre conseguimos um XY-caminho em M. Existem três tipos de XY*-obstruções*:

1. uma quíntupla (C, v_1, v_2, v_3, v_4) onde C é um bloco de M e $v_1, \ldots, v_4$ são vértices distintos em C e que aparecem nesta ordem, tais que v_1 e v_3 enxergam X através de $E(M) \setminus E(C)$ e v_2 e v_4 enxergam Y através de $E(M) \setminus E(C)$;
2. uma quádrupla (C, v_1, v_2, v_3) onde C é um bloco de M e v_1, v_2 e v_3 são vértices distintos em C que enxergam X e Y através de $E(M) \setminus E(C)$;

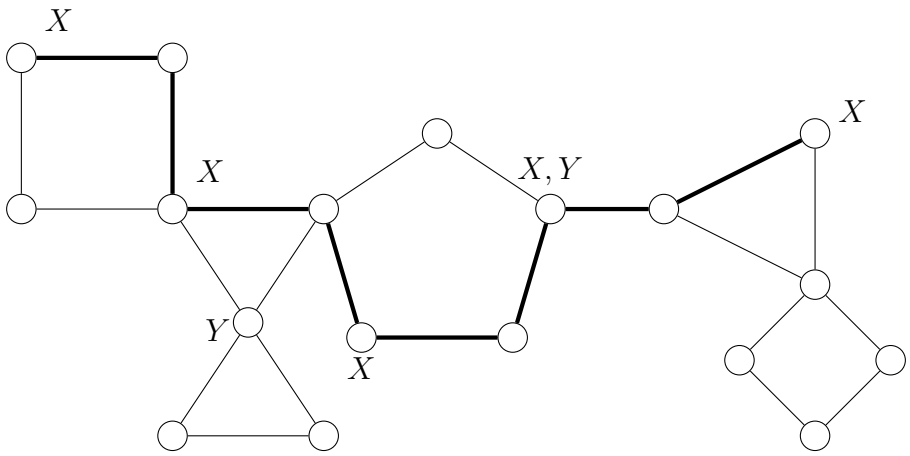


Figura 10: Um exemplo de um XY-caminho.

3. uma quádrupla (v, K_1, K_2, K_3) tal que $v \in V(M)$, e K_1, K_2 e K_3 são três componentes distintos de $M - v$, tais que $V(K_i) \cap X \neq \emptyset$ e $V(K_i) \cap Y \neq \emptyset$.

5.7 Árvore de Blocos

Dado um grafo G, para montar sua *árvore de blocos* T começamos por encontrar cada um dos blocos de G e então criar um vértice novo para cada um dos blocos existentes, ou seja, $V(T) = V(G) \cup C$ sendo C o conjunto dos novos vértices. Após isso, no processo de finalizar nossa árvore, sendo $C(u)$ o conjunto dos vértices pertencentes ao bloco que u representa, fazemos com que $E(T) = \{uv : (u \in C) \wedge (v \in C(u))\}$. Temos um exemplo de uma árvore de blocos na Figura 11.

Vértices em T que pertencem ao conjunto $V(G)$ são denominados *p-nós* enquanto que os vértices pertencentes a C são denominados *c-nós*.

5.8 Subdivisão

A *subdivisão* de uma aresta uv em um grafo G é uma operação que transforma uma aresta em um caminho através da adição de um vértice de grau 2, como exemplificado na Figura 12.

Um grafo H é subdivisão de um grafo G quando podemos obter o grafo H através de subdivisões das arestas de G.

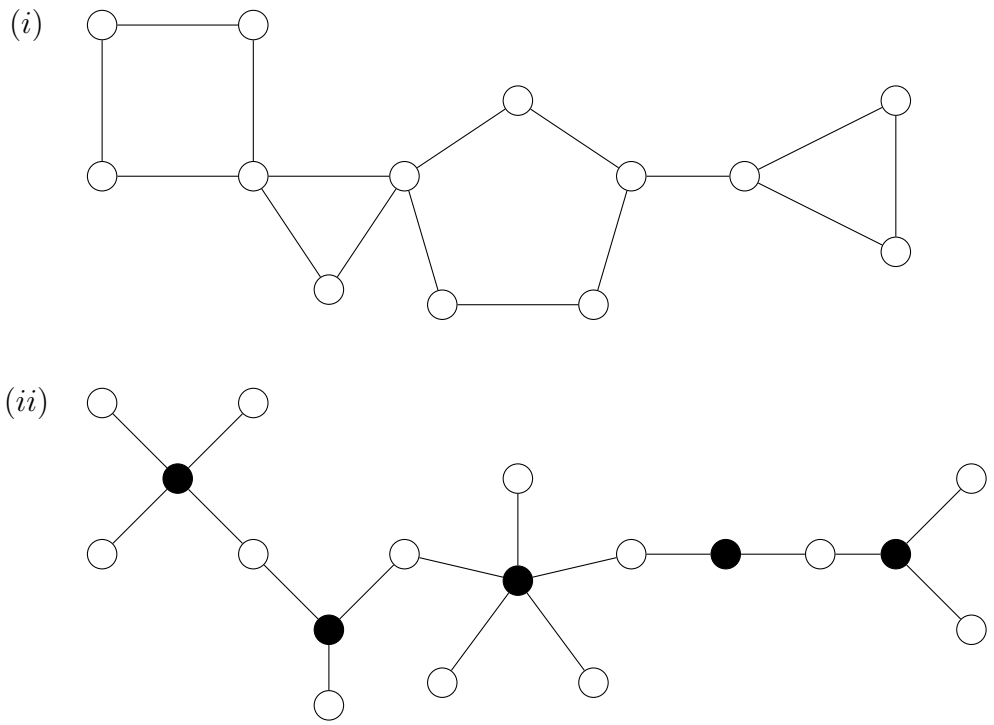


Figura 11: (i) Um moldura M de um grafo G qualquer. (ii) Árvore de blocos de M, onde os vértices pintados são c-nós.

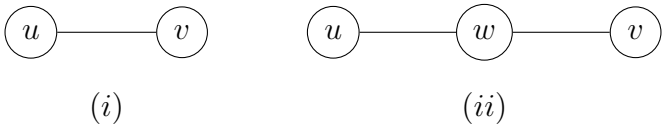
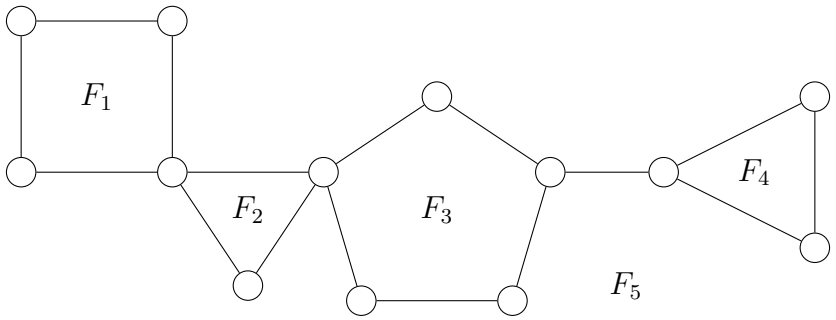


Figura 12: Em (i) uma aresta qualquer e uma subdivisão da mesma em (ii).



$$deg(F_1) = 4 \qquad deg(F_3) = 5 \qquad deg(F_5) = 17$$
$$deg(F_2) = 3 \qquad deg(F_4) = 3$$

Figura 13: Faces de um grafo G e seus graus.

5.9 Faces

Um grafo plano G divide o resto do plano em conjuntos abertos conectados por arestas. Chamamos estes conjuntos de *faces*, o conjunto contendo as faces de um grafo G é denotado por $F(G)$. O número de faces é denotado por $\phi(G)$. Em um grafo planar, o grau de uma face f, denotado por $deg(f)$, é o número de arestas incidentes a f, com arestas de corte sendo contadas duas vezes.

5.10 Resultados importantes

Nessa subseção enunciaremos alguns teoremas e resultados que nos dizem um pouco mais sobre as propriedades de nossos grafos no que tange a questão de planaridade. Dado um grafo G, consideramos $n = |V(G)|$ e $m = |E(G)|$.

Teorema 1. *Para qualquer grafo G,*

$$\sum_{v \in V(G)} deg(v) = 2m.$$

Teorema 2. *Sendo G um grafo plano, denotamos o conjunto de faces em G por $F(G)$. Segue que:*

$$\sum_{f \in F(G)} deg(f) = 2m.$$

Teorema 3 (Fórmula de Euler)**.** *Sendo G um grafo plano e conexo, a seguinte relação*

é verdadeira:

$$n - m + \phi(G) = 2.$$

Corolário 4. *Sendo G um grafo simples e planar com $n \geq 3$, vale que*

$$m \leq 3n - 6.$$

Corolário 5. *K_5 é não-planar.*

Demonstração. Sendo $m(K_5)$ o número de arestas de K_5, temos que

$$m(K_5) = \binom{5}{2} = 10.$$

Pelo Corolário 3, sendo $n(K_5)$ o número de vértices de K_5, temos

$$m(K_5) = 10 > 3n(K_5) - 6 = 9,$$

portanto, K_5 é não-planar. □

Corolário 6. *$K_{3,3}$ é não-planar.*

Demonstração. Suponha que $K_{3,3}$ é planar e denotemos G ser a representação planar do $K_{3,3}$. Como $K_{3,3}$ não possui ciclo menor que quatro, cada face de G possui, no mínimo, grau quatro. Então, pelo Teorema 2, temos que

$$4\phi \leq \sum_{f \in F(G)} deg(f) = 2m = 18,$$

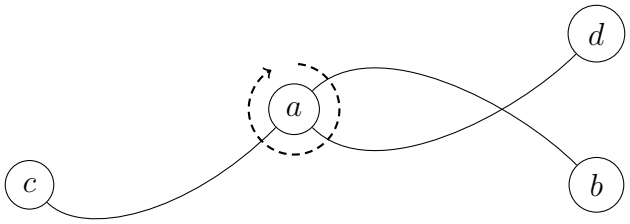
implicando que $\phi(G) \leq 4$. Com a fórmula de Euler, no Teorema 3, segue que

$$2 = n - m + \phi(G) \leq 6 - 9 + 4 = 1,$$

chegando assim, a um absurdo, portanto, $K_{3,3}$ é não-planar. □

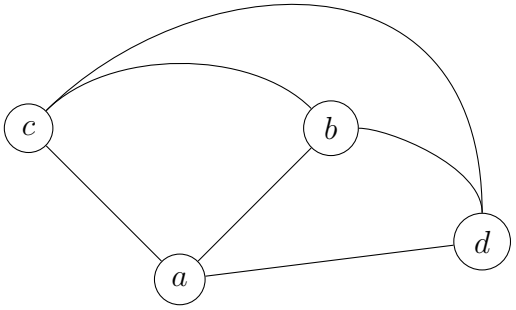
6 Descrições Combinatórias e Ciclos Faciais

Nesta seção, os grafos simples serão vistos como digrafos, de forma que a aresta uv indica a presença dos arcos uv e vu.



$$D(a) = \{ac, ab, ad\}$$

Figura 14: Ordem cíclica dos arcos com ponta inicial em a.



$$D(a) = \{ac, ab, ad\}$$
$$D(b) = \{bd, ba, bc\}$$
$$D(c) = \{cd, cb, ca\}$$
$$D(d) = \{ac, ab, ad\}$$

Figura 15: Descrição combinatória D.

Um desenho de um grafo acaba por induzir uma ordem cíclica (horário ou anti-horária) dos arcos com ponta inicial em cada vértice. Ao construirmos uma lista formada por todas as ordens cíclicas da representação de um dado grafo G, obtemos a *descrição combinatória* do desenho. Temos que uma lista D é uma descrição combinatória de um desenho se $\forall v \in V(G)$, $D(v)$ é a ordem cíclica de todos os arcos com ponta inicial em v no desenho, dessa forma, para cada ordem $D(v)$ teremos uma lista $D(v) = \{vu_1, vu_2, \dots vu_x\}$, onde x é o grau de v, como na Figura 14.

Vale ressaltar que uma descrição combinatória pode corresponder a uma infinidade de grafos, inclusive as próprias listas de adjacências de um grafo formam uma descrição combinatória do mesmo. Podemos ver um exemplo de descrição na Figura 15.

Um *ciclo facial* de uma descrição D é um ciclo da forma $\{v_0, v_1, v_2, \dots, v_0\}$ tal que $\forall k, v_k v_{k+1}$ é o sucessor de $v_k v_{k-1}$ em $D(v_k)$.

Consideramos que uma descrição combinatória de um desenho plano é uma descrição *plana*. Dessa forma, por definição, todo grafo planar possui uma descrição combinatória mas nem toda descrição combinatória de um grafo planar é plana. Note que se D é uma descrição combinatória plana de um grafo com pelo menos uma aresta, temos uma

correspondência bijetora entre os ciclos faciais de D e as faces do desenho.

6.1 Critério Combinatório

O critério para determinar se uma dada descrição combinatória é plana se utiliza fortemente da correspondência bijetora entre os ciclos faciais de uma descrição combinatória e as faces existentes no desenho, se utilizando assim, de um critério combinatório de planaridade que é naturalmente traduzido para código. Esse procedimento encontra-se no Algoritmo 3.

Teorema 7. *Seja G um grafo conexo planar com pelo menos uma aresta, com $n = |V(G)|$ e $m = |E(G)|$. Uma descrição combinatória D de um desenho de G é plana se e somente se*

$$n - m + f_D = 2\,,$$

onde f_D é o número de ciclos faciais na descrição D.

Demonstração. Seja D uma descrição de um desenho plano de G. Dada a correspondência bijetora que há entre os ciclos faciais de D e as faces do desenho, temos que dado G com pelo menos uma aresta, o Teorema 3, de Euler, vale. Resta mostrar que se uma dada descrição combinatória D satisfaz o Teorema 3, então D é plana. Faremos isso por indução em f_D.

Se G tem pelo menos uma aresta, então $f_D \geq 1$. Caso $f_D = 1$, temos que $n-m = 1$, ou seja, G é uma árvore e, portanto, D é plana, dado que toda descrição combinatória de uma árvore é plana.

Considere então $f_D > 1$. Suponha que todo par uv e vu de arcos pertencem a um mesmo ciclo facial e considere um vértice v de G, com $D(v) = \{vv_1, vv_2, \ldots, vv_k\}$. Dado que vv_i e v_iv pertencem a um mesmo ciclo facial, temos que todos os arcos com uma ponta em v também pertencem ao ciclo. Como G é conexo, então todos os arcos de G estão no mesmo ciclo facial, o que faria $f_D = 1$, um absurdo. Podemos afirmar então que há duas arestas uv e vu em G que não pertencem a um mesmo mesmo ciclo facial.

Sejam uv e vu arestas de G que não pertencem a um mesmo ciclo facial de D. Sejam $C_{uv} := \{v = w_1, w_2, \ldots, w_{k-1} = u, v = w_1\}$ e $C_{vu} := \{u = x_1, x_2, \ldots, x_{k-1} = v, u = x_1\}$ ciclos faciais distintos de D contendo vu e uv. Pela construção de C_{vu} e C_{uv} temos que, ao retirar as arestas uv e vu, obtemos um grafo G' conexo, pois ambos os ciclos faciais possuem vértices em comum e cada um desses vértices faz parte de dois ciclos faciais.

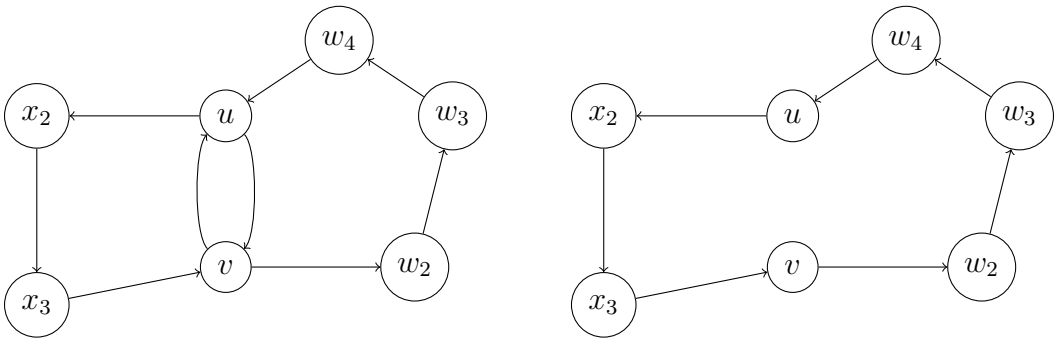


Figura 16: Removendo arestas uv e vu de seus ciclos faciais.

Seja D' a descrição combinatória de G' induzida pela descrição D de G. Temos que os ciclos faciais tratados anteriormente serão, de certa forma, agrupados, e darão origem a um novo ciclo facial que denotaremos pelo ciclo facial C', de tal forma que $C' = \{u = x_1, x_2, \ldots, x_{k-1} = v = w_1, w_2, \ldots, w_{k-1} = u\}$, como pode ser visto na Figura 16. Portanto, o número de faces da descrição D apenas foi diminuído em 1, sendo $f_{D'} = f_D - 1$ e então, por hipótese de indução, D' é uma descrição combinatória plana de um desenho plano de G'.

Como foi dito anteriormente, há uma bijeção entre o número de ciclos faciais de D' e o número de faces. Deste modo, temos que C' corresponde a uma face F' no desenho plano. Com isso, é possível estendermos o desenho de G' para um desenho plano de G fazendo com que as arestas uv e vu estejam contidas em F', fazendo assim, com que D seja plana. □

6.2 Complexidade do Algoritmo do Critério Combinatório

Analisaremos a complexidade nos utilizando do pseudocódigo descrito no Algoritmo 3. Na linha 2 inicializamos um vetor indexado por arestas de G, com *falso* em todos os campos, e então temos uma complexidade de tempo $\Theta(m)$. Tal vetor indexado por arestas nos informa em $U[e_j]$ se a aresta e_j já foi utilizada na construção de um ciclo facial de D.

Em seguida, na linha 4, temos um laço em que basicamente escolhemos uma aresta $e_i \in E(G)$ e tentamos montar nosso ciclo facial iterando através de sucessores cíclicos de e_i, enquanto eles não pertencem a nenhum outro ciclo facial. Dessa forma, no laço todo, temos uma complexidade de $O(m)$.

Finalmente, apenas aplicamos o Teorema 7, onde temos operações aritméticas que

Algoritmo 3: Critério combinatório

Entrada: Grafo G conexo com pelo menos uma aresta e uma descrição combinatória implícita na listas de adjacência de G

Saída: Retorna SIM caso a descrição seja plana e NÃO caso contrário

```
1  início
2      Crie um vetor U[1 ... |E(G)|] com elementos de valor booleano falso
3      Inicialize f_D := 0
4      para cada e ∈ E(G) faça
5          se ¬U[e] então
6              e_i := e
7              i := 1
8              enquanto U[e_i] ≠ e ou i = 1 faça
9                  U[e_i] := verdadeiro
10                 e_i := sucessor cíclico de e_i
11                 i := i + 1
12                 f_D := f_D + 1
13     Sendo n = |V(G)| e m = |E(G)|
14     se f_D = m − n + 2 então
15         retorna SIM
16     senão
17         retorna NÃO
```

podem ser resumidas a uma complexidade $O(1)$.

Dessa forma, a complexidade total do algoritmo é $\Theta(m) + O(m) + O(1) = \Theta(m)$.

7 Algoritmo da Lec-ordenação

O algoritmo que será descrito na Seção 9 tenta sempre estender uma moldura M acrescentando um novo vértice à ela. Os novos vértices são escolhidos pela ordem de uma lec-ordenação do grafo, de forma que nesta seção iremos discutir o algoritmo para encontrar tal sequência.

De forma simples, o algoritmo utilizado para se obter uma lec-ordenação, definida na Subseção 5.2, se baseia em um processo recursivo que utiliza contração de arestas. Dado um grafo G biconexo com $n = |V(G)|$ e $m = |E(G)|$, temos que para $n \leq 2$ qualquer ordenação dos vértices já é uma lec-ordenação. Dessa forma, utilizaremos esse como nosso caso base. Agora supondo que temos $n > 2$ precisaremos, antes da chamada recursiva, determinar uma aresta uv que seja ponta de um caminho maximal no grafo, sendo essa uma operação dependente da realização da bp-ordenação, apresentada na Subseção 5.1. Em seguida, ao determinarmos essa aresta, encontramos uma lec-ordenação de G' resultante da contração de uv em G.

Por fim, dado que obtivemos uma lec-ordenação de G', uma lec-ordenação de G pode ser obtida ao trocarmos o vértice w resultante da contração de uv pela sequência v, u ou u, v. A sequência a ser escolhida será a que mantém a propriedade da lec-ordenação. O procedimento descrito é apresentado no Algoritmo 4.

7.1 Complexidade da Lec-ordenação

Como podemos ver no Algoritmo 4, inicializamos uma lista L, a qual armazenará o resultado de nossa lec-ordenação, e também começamos com uma lista I, representando uma bp-ordenação do grafo de entrada G. Para realizar a bp-ordenação e construir a árvore T temos um custo inicial de $O(n + m)$. Analisemos agora as condições dentro do laço da linha 4.

Caso $L = \emptyset$, apenas fazemos uma operação em tempo $O(1)$, que seria colocar v em L. Caso $L \neq \emptyset$, teremos de encontrar u predecessor de v em T (custo $O(1)$ para cada v) e, dada as condições das linhas 12 e 14, também precisamos acessar descendentes de v em T ($O(n)$ para cada v, pois T tem $|E(T)| = O(n)$)) e também predecessores e sucessores de u em L (custo $O(n)$ ao todo para cada v).

Algoritmo 4: Lec-ordenação(G)

Entrada: Grafo G biconexo
Saída: Uma lista contendo uma lec-ordenação de G

```
1  início
2      L := lista vazia
3      I := ordenação dos vértices de acordo com um percurso pré-ordem em uma
         bp-árvore T de G
4      enquanto I ≠ ∅ faça
5          v := primeiro vértice de I
6          Remova v de I
7          se L = ∅ então
8              L := (v)
9          senão
10             Seja u o predecessor de v em T
11             Seja L = (w_1, ..., w_i, u, w_{i+1}, ..., w_k)
12             se existe aresta de um descendente de v em T a um predecessor de u
                 em L então
13                 L := (w_1, ..., w_i, v, u, w_{i+1}, ..., w_k)
14             se existe aresta de um descendente de v em T a um sucessor de u
                 em L então
15                 L := (w_1, ..., w_i, u, v, w_{i+1}, ..., w_k)
16     retorna L
```

Após isso, para verificar essa existência de arestas das linhas 12 e 14 teremos de iterar por cada x descendente de v em T (chamaremos tal conjunto de A) e também iterar por cada y predecessor ou sucessor de u em L (chamaremos o conjunto de B), verificando se $xy \in E(G)$. Dessa forma, para cada v, teremos que o passo de iterar por cada $x \in A$ terá custo $O(n)$ e que o passo de iterar por cada $y \in B$ terá custo $O(n^2)$, pois cada x possui $O(n)$ predecessores ou sucessores, e executamos isso para cada $x \in A$, resultando em $O(n^2)$. A fase em que verificamos se $xy \in E(G)$ executa $\sum_{x \in A} \sum_{y \in B} deg(x)$ vezes e, então, pelo Teorema 1, sabemos que temos um custo de $O(nm)$. Em seguida, apenas inserimos v em uma posição específica de L com custo $O(n)$ para cada v.

Finalmente, dado que dentro do laço da linha 4 temos complexidade $O(n)+O(n^2)+O(nm)$ e que executamos o laço $\Theta(n)$ vezes, então o custo do laço seria:

$$\Theta(n)\left(O(n) + O(n^2) + O(nm)\right) = O(n^2) + O(n^3) + O(n^2m) = O(n^3 + n^2m).$$

Somando ao pré-processamento temos, no total do algoritmo, uma complexidade de $O(n^3) + O(n^2m) + O(n + m) = O(n^3 + n^2m)$.

8 Algoritmo XY-caminho

O algoritmo que será descrito na Seção 9 tenta sempre estender uma moldura M e se utiliza de conjuntos X e Y de vértices de tal forma que se encontramos uma XY-obstrução o grafo em questão é não planar e se encontramos um XY-caminho utilizamo-lo para continuar estendendo nossa moldura. Dessa forma, nesta seção iremos discutir como encontrar os XY-caminhos e as XY-obstruções.

O método para se encontrar uma XY-obstrução ou um XY-caminho consiste de reduções proporcionadas pela remoção de toda folha f da árvore de blocos T que não está em $X \cap Y$ em uma dada iteração. Após essas reduções verificamos se temos uma XY-obstrução ou um XY-caminho. No pseudocódigo descrito no Algoritmo 5, os conjuntos X_T e Y_T armazenam os vértices que estão atualmente em T que têm relação com os vértices originalmente em X e Y. O conjunto W (resp. Z) armazena p-nós que fazem parte de um caminho entre um vértice em X_T (resp. Y_T) e um vértice em X (resp. Y).

Ao final das reduções, caso T não seja um caminho, então já temos garantia de uma

Algoritmo 5: XY-caminho(M, X, Y)

Entrada: Moldura conexa M e subconjuntos X e Y de $V(M)$
Saída: NULO se encontrar uma XY-obstrução ou devolve um XY-caminho

```
1  início
2      T := árvore de blocos de M
3      X_T := X
4      Y_T := Y
5      W := X
6      Z := Y
7      enquanto existe folha f de T que não está em X_T ∩ Y_T faça
8          u := vizinho de f em T
9          T := T − f
10         se f ∈ X_T então
11             X_T := X_T ∪ {u}
12             se u é um p-nó então
13                 W := W ∪ {u}
14         se f ∈ Y_T então
15             Y_T := Y_T ∪ {u}
16             se u é um p-nó então
17                 Z := Z ∪ {u}
18     se T é um caminho então
19         R := conjunto de todos os p-nós em T
20         para cada c-nó C_i em T faça
21             X_{C_i} := V(C_i) ∩ (W ∪ R)
22             Y_{C_i} := V(C_i) ∩ (Z ∪ R)
23         se Cada c-nó C_i em T possui um caminho P_{C_i} contendo X_{C_i} e
            internamente disjunto de Y_{C_i} então
24             P_T := caminho em M gerado pela concatenação dos caminhos P_{C_i}
25             P := caminho em M contendo P_T e com extremos em X de tal
                forma que para cada bloco C_i na base de P, P contenha os vértices
                em V(C_i) ∩ W
26             retorna P
27         senão
28             retorna NULO
29     senão
30         retorna NULO
```

XY-obstrução do tipo (C, v_1, v_2, v_3) ou do tipo (v, K_1, K_2, K_3), como visto nos itens 2 e 3 na Subseção 5.6, e por isso devolvemos NULO. Se T for um caminho, então tantamos construi-lo com base em cada c-nó que restou em T. No caso em que a condição da linha 23 é satisfeita, precisamos ainda estender tal caminho (pois, por definição, um XY-caminho precisa ter extremos em X), e finalmente o algoritmo irá devolver tal XY-caminho. Veja que essa extensão é simples, pois basta seguir por vértices que estão em W até encontrar um vértice originalmente em X. Caso contrário, se a condição não for satisfeita, então temos uma XY-obstrução do tipo (C, v_1, v_2, v_3, v_4), como no item 1 da Subseção 5.6, e então devolvemos NULO também.

8.1 Complexidade do Algoritmo XY-caminho

No início do Algoritmo 5 geramos a árvore de blocos T de M. Tal processo pode ser concluído em tempo linear proporcional a $O(n)$ [2].

Iniciaremos analisando a verificação existente no laço da linha 7. É possível verificar tal condição se utilizando de uma lista F que contenha todas as folhas que não estão em $X \cap Y$ e então poderíamos determinar o valor-verdade da condição apenas checando se $F = \emptyset$, sendo então algo de complexidade $O(1)$. Vale lembrar que é necessário inserir e remover vértices da lista F na parte de redução da árvore T do algoritmo para obter sucesso com essa implementação, sendo essas também operações $O(1)$ e, portanto, o laço apenas possui operações que levam tempo $O(1)$. Note que a construção inicial de F, no entanto, leva tempo $O(n)$.

Após o laço, precisamos verificar, na linha 18 se T é um caminho. Para isso, basta que todo vértice de T tenha grau 1 ou 2, o que pode ser determinado em tempo $O(n)$ utilizando uma estratégia similar à do parágrafo anterior.

Caso T seja um caminho, temos que verificar a condição da linha 23. Podemos determinar ela em tempo $O(nm)$, pensando que visitaríamos cada c-nó ($O(n)$) e tentaríamos montar um caminho dentro dele ($O(m)$). Além disso, caso a condição seja satisfeita poderíamos executar o resto em tempo $O(n+m)$, pois apenas percorreríamos o grafo partindo das pontas do XY-caminho. Apesar deste método citado, utilizando a estrutura PC-árvore e seus métodos descritos na dissertação de Noma [7], seria possível verificar tal condição e executar o que está dentro dela em tempo $O(n)$.

Por fim, todos os outros casos apenas retornariam NULO em tempo $O(1)$. Dessa forma, no total, temos um laço na linha 7 que é executado $O(n)$ vezes e apenas possui operações $O(1)$, resultando em complexidade $O(n)$ e, posteriormente, caso as condições

das linhas 18 e 23 sejam satisfeitas, teremos complexidade $O(n)$ no total. Somando então o laço da linha 7 com o custo máximo do que há após ele temos então complexidade $O(n)$ para o algoritmo como um todo.

9 Algoritmo de Lempel, Even e Cederbaum

O algoritmo de Lempel, Even e Cederbaum (LEC) [6] trata o problema de verificar se um dado grafo G é planar e utiliza como base os conceitos presentes no Teorema de Kuratowski.

Teorema 8 (Teorema de Kuratowski)**.** *Um grafo G é planar se e somente se G não contém um subgrafo que seja subdivisão do K_5 ou do $K_{3,3}$.*

Definição 1. Grafos que são subdivisões do K_5 ou do $K_{3,3}$ são denominados *Grafos de Kuratowski*.

Infelizmente, o teorema não nos oferece, em sua demonstração, um algoritmo intuitivo para sua implementação e então nos focamos nos estudos do Algoritmo de LEC. Note que este algoritmo apenas nos diz se G é planar ou não, e não nos oferece um desenho planar se o grafo for planar, ou um subgrafo de Kuratowski se for não-planar, apesar que existem algoritmos que fazem isso.

A descrição original do algoritmo se utiliza de um estrutura de dados denominada *forma de bush*, responsável por manter toda a informação necessária para a extensão do desenho plano de um grafo dado, quando possível. Apesar disso, descreveremos o algoritmo utilizando o conceito de molduras, proposto por Robin Thomas [10], apresentadas na Subseção 5.3.

O funcionamento do algoritmo LEC é formalizado no Algoritmo 6. A ideia geral do algoritmo de LEC se dá por examinar cada um dos vértices na ordem dada por uma lec-ordenação, buscando, a cada iteração, estender uma moldura. A ideia é que teremos um subgrafo H do grafo original G que contém os vértices já examinados (H é portanto construído a partir da lec-ordenação), e ao estender uma moldura de H estaremos estendendo um desenho plano de H. Isso será feito por meio de XY-caminhos e caminhos complementares.

Antes de demonstrar que o Algoritmo de LEC funciona iremos apresentar alguns resultados estudados que serão utilizados para o entendimento da demonstração. Todos esses resultados e suas demonstrações completas encontram-se na dissertação de Noma [7].

Lema 9. *Todo grafo biconexo possui uma lec-ordenação.*

Teorema 10. *Seja M uma moldura de um grafo conexo G e sejam X e Y subconjuntos de $V(M)$, segue que:*

1. *existe uma XY-obstrução em M;*
2. *existe um XY-caminho em M.*

Lema 11. *Seja M a moldura no início de uma iteração do Algoritmo de LEC em que $L \neq \emptyset$. Se existe uma XY-obstrução em M então o grafo G de entrada possui um subgrafo de Kuratowski.*

Teorema 12. *O Algoritmo de LEC decide se um dado grafo G biconexo é planar.*

Demonstração. Dado que o grafo G recebido como entrada é biconexo, então sabemos que G possui uma lec-ordenação [7, p. 33]. Quando $L = \emptyset$, significa que foi possível sempre estender a moldura. Pela definição de moldura, temos que G é planar e então a resposta do algoritmo está correta.

Suponha que em uma dada iteração temos que $L \neq \emptyset$. Seja M a moldura na iteração atual e X e Y os conjuntos de vértices da mesma iteração. Temos então que, pelo Teorema 10, M possui apenas uma XY-obstrução ou um XY-caminho, de forma mutuamente exclusiva.

Dado que em toda iteração fazemos com que H e $G - V(H)$ sejam conexos e que M é sempre mantida como uma moldura de H em G, então caso tenhamos um XY-caminho, essas invariantes se manterão na próxima iteração. Caso tenhamos uma XY-obstrução então já iremos retornar NÃO, declarando G não-planar, o que é reafirmado pelo Lema 11. Nesse caso, há um subgrafo de Kuratowski em G e, dado o Teorema 8, temos que G não é planar. □

Apesar do Algoritmo de LEC funcionar apenas para grafos biconexos, é possível, com algumas alterações, decidir se um grafo G não-biconexo é planar ou não. Note que para G não biconexo basta aplicar o Algoritmo de LEC às componentes biconexas maximais, se todas as componentes forem planares então G é planar, caso contrário, G é não-planar.

Algoritmo 6: Algoritmo de LEC

Entrada: Grafo G biconexo
Saída: Retorna SIM se G é planar e *não*, caso contrário

início
  $H := \emptyset$
  Seja M um grafo vazio
  $L :=$ Lec-ordenação(G)
  enquanto $L \neq \emptyset$ **faça**
    $w :=$ primeiro vértice de L
    Remova w de L
    $X := \{u \in V(M) : uw \in E(G)\}$
    $Y := \{u \in V(M) : v \in V(G) \setminus (H \cup \{w\}), uv \in E(G)\}$
    $P := XY$-caminho(M, X, Y)
    se $P =$ NULO **então**
      retorna NÃO
    senão
      Sendo $\overline{P} := (z_1, z_2, \ldots, z_k)$ o caminho complementar de P
      $A := V(M) \setminus V(\overline{P})$
      $B := Y \setminus V(\overline{P})$
      $R := \{v \in A : v$ não enxerga B através de $A\}$
      $M := M - R - E(P)$
      $M := M + w + wz_1 + wz_k$
      $H := H \cup \{w\}$
  retorna SIM

9.1 Complexidade do Algoritmo de LEC

No início do Algoritmo 6 temos um pré-processamento na construção da lista L pois utilizamos o Algoritmo 4. Dessa forma, neste início temos um custo de $O(n^3 + n^2m)$.

A seguir, temos um laço baseado em uma comparação que pode ser feita em $O(1)$ e dentro dele temos operações $O(1)$ como atribuição e remoção e também montamos os conjuntos X e Y, o que pode ser uma operação $O(nm)$ dado que temos de consultar todos os vértices da moldura e verificar as condições necessárias para eles fazerem parte dos conjuntos.

Após isso, verificamos se temos uma XY-obstrução ou um XY-caminho em M. Tal operação tem custo $O(n)$, como visto na Subseção 8.1. Caso encontremos uma XY-obstrução o algoritmo finaliza devolvendo NÃO em tempo $O(1)$. Caso seja encontrado um XY-caminho, temos algumas operações constantes ou lineares e uma operação de tempo $O(n + m)$ para construir o conjunto de vértices R, o qual precisamos percorrer todas as componentes que são separadas pelo caminho complementar e, em caso de não haver um vértice pertencente a Y na componente, colocamos os vértices da mesma em R.

Com isso, como estamos removendo elementos de L a cada iteração, temos $O(n)$ iterações que levam tempo $O(nm)$, tendo então o algoritmo todo complexidade $O(n^2m)$. Somando ao pré-processamento, teremos no total um custo de $O(n^2m)+O(n^3+n^2m) = O(n^3 + n^2m)$ para o algoritmo de LEC.

10 Implementações

Todos os algoritmos mencionados nesse documento foram implementados em Julia. A linguagem Julia se trata de uma linguagem de programação dinâmica de alto nível e *open source* com enfoque em computação científica e numérica, lançada oficialmente em 2012. Projetada desde o início para obter alto desempenho, Julia possui uma sintaxe simples, com enfoque para computação científica e numérica mas também eficaz para programação de propósito geral [5].

Todo o código fonte de Julia está publicamente disponível na plataforma GitHub [9]. Atualmente, a comunidade registra que existem mais de 2400 pacotes disponíveis para uso. Dentre estes, o pacote LightGraphs [4], que fornece várias implementações para manipulação de grafos, se destaca como um dos mais utilizados e com mais contribuidores dentre os pacotes relacionados a grafos.

Figura 17: *Pull resquest* para a implementação do Algoritmo de Borůvka.

10.1 Recursos da Linguagem

O estudo da linguagem foi realizado, principalmente, através da documentação presente no site oficial da linguagem [5]. Devido ao grande número de recursos presentes, foi necessário dar um enfoque maior para os recursos que seriam provavelmente utilizados para a implementação dos algoritmos. Dessa forma, primeiramente, foi necessário entender declarações de variáveis, condicionais, *loops*, escopos de funções e variáveis, estruturas de dados básicas, vetores e matrizes, entre outros recursos comumente aprendidos em cursos de introdução a linguagens de programação.

Após conhecer os recursos básicos foi necessário também estudar funções já existentes na biblioteca LightGraphs, buscando não repetir códigos, acelerando o progresso, diminuindo o risco de erros, e dando uniformidade aos programas.

Além disso, foi necessário também estudar algumas estruturas de dados existentes no pacote DataStructures para verificarmos sua usabilidade em nossas implementações. Por exemplo, utilizamos *Disjoint Sets* no Algoritmo de Borůvka.

10.2 Contribuições

Contribuições na linguagem Julia se dão através de *pull requests*. Como podemos ver nas Figuras 17 e 18, fizemos duas requisições para adicionar nossos programas e nossos testes ao pacote LightGraphs.

Sabendo que seria necessária a aprovação de ao menos um dentre os contribuidores habilitados a fazer mudanças no pacote, aceitamos as sugestões dos desenvolvedores que entraram em contato conosco, obtendo assim um suporte maior para adequar nossos códigos aos padrões, tanto da linguagem Julia, quanto da biblioteca LightGraphs. Visamos sempre pensar na usabilidade do pacote e também na clareza do código para

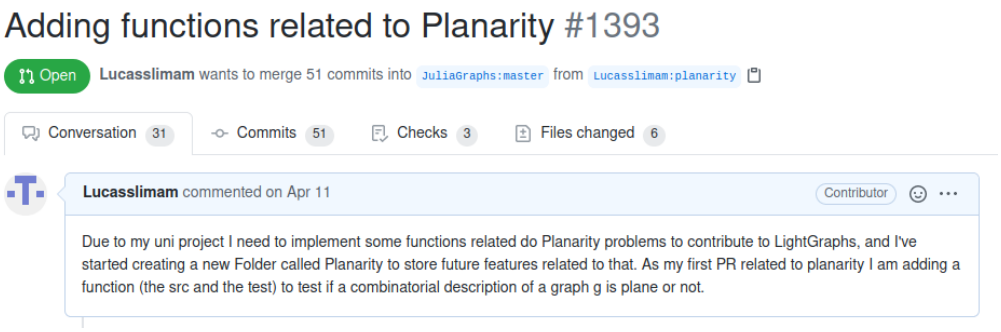


Figura 18: *Pull Request* para integrar o algoritmo do critério combinatório para verificação de desenho planar.

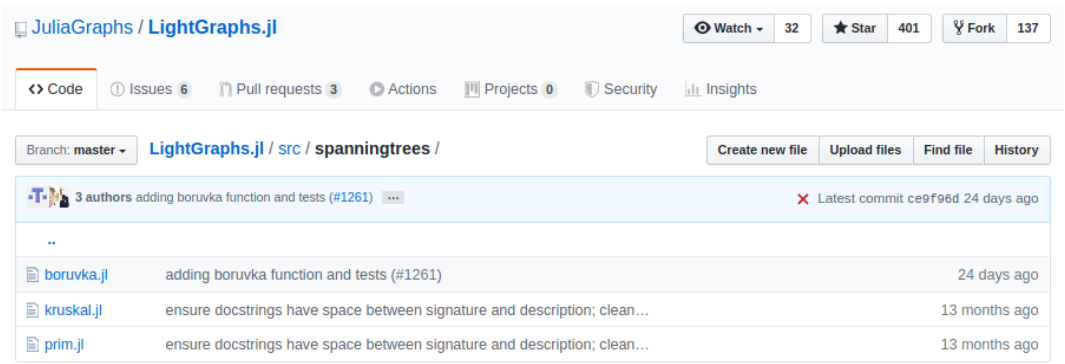


Figura 19: Diretório da biblioteca contendo códigos para encontrar árvores geradoras mínimas.

outros desenvolvedores pois, em caso de aperfeiçoamento ou reparo, o código estaria visualmente limpo e bem explicado através dos detalhes sobre a utilização e também pelos comentários colocados no mesmo.

Além disso, nossos testes possuem um diferencial em relação ao modelo de testes utilizado anteriormente para os outros algoritmos de árvores geradoras mínimas, Kruskal e Prim, que já estavam implementados. Diferentemente dos outros dois casos mencionados, que validavam o resultado apenas pelo conjunto de arestas obtidas, nós verificamos se o resultado obtido estava correto pela própria definição do que seria uma árvore geradora mínima: nós verificamos o custo, o número de arestas e também o número de vértices. Dessa forma, caso comparássemos dois vetores de solução com arestas em ordem diferente ou arestas com as extremidades armazenadas de forma diferente, como (u, v) ou (v, u), ainda avaliaríamos um vetor nessas condições como solução correta.

Tais práticas fizeram tanto o programa principal quanto os testes para o Borůvka serem aceitos. Podemos verificá-los já adicionados na Figura 19 e na Figura 20. O

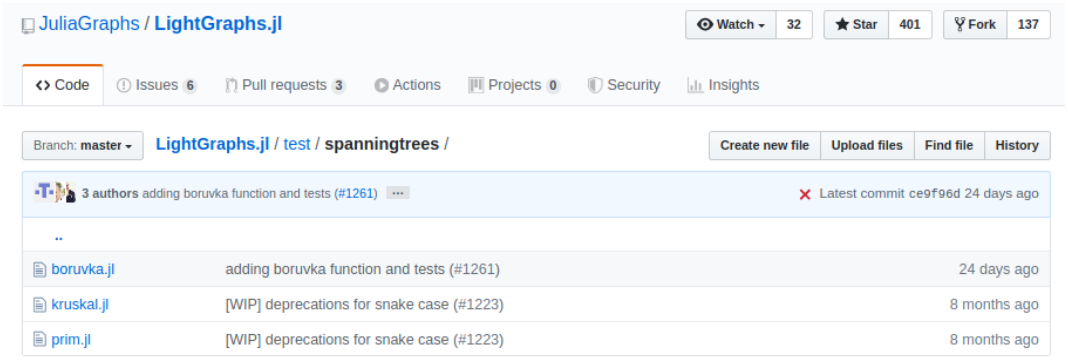


Figura 20: Diretório da biblioteca contendo testes para os códigos de árvores geradoras mínimas.

Código 1, em anexo, é o mesmo que está incluído na LightGraphs.

11 Considerações Finais

Nesta iniciação científica, começamos por estudar e implementar o Algoritmo de Borůvka, responsável por encontrar uma árvore geradora mínima, utilizando como base o livro de Sedgewick [8]. Tal implementação, após passar por melhorias sugeridas pelos administradores, serviu de contribuição para a biblioteca LightGraphs. Decidimos por fazer essa contribuição inicial simples de um problema já bem conhecido para aprendermos a linguagem e entendermos melhor o funcionamento das ferramentas utilizadas para gerenciamento da linguagem Julia.

Posteriormente, houve o estudo e implementação de algoritmos relacionados ao teste de planaridade em grafos. Primeiramente, lidamos com o algoritmo do Critério Combinatório, o qual também foi disponibilizado para ser integrado à biblioteca LightGraphs mas, pelo processo ser demorado, principalmente em uma situação de pandemia, ainda não houve a integração da mesma. O Código 2, em anexo, é o mesmo que foi submetido à biblioteca. Esse algoritmo não é utilizado diretamente no algoritmo da verificação de planaridade, mas decidimos começar por ele devido à sua simplicidade, o que nos ajudou a compreender melhor várias definições importantes em planaridade.

Por fim, trabalhamos com o Algoritmo de LEC, cuja implementação e testes foram terminados recentemente e, por isso, não houve tempo hábil para fazer a *pull request*. Apesar do nosso código ter funcionado para todos os 23 casos de teste, acreditamos que mais testes ainda são necessários. Os Códigos 3, 4, 5, 6 e 7, em anexo, contêm nossas implementações.

Seria interessante, como trabalho futuro, fazer uma nova implementação do Algoritmo de LEC utilizando a estrutura PC-árvore [7], o que garantiria uma maior eficiência do algoritmo.

Referências

[1] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[2] S. Even. Graph algorithms, 1979.

[3] JuliaCollections. DataStructures – Julia implementation of a variety of data structures. https://github.com/JuliaCollections/DataStructures.jl. Acesso em 27 de setembro de 2020.

[4] JuliaGraphs. LightGraphs – An optimized graphs package for the Julia programming language. https://github.com/JuliaGraphs/LightGraphs.jl. Acesso em 27 de setembro de 2020.

[5] JuliaLang. The Julia Programming Language. https://julialang.org/. Acesso em 27 de setembro de 2020.

[6] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Proceddings International Symposium on Theory of Graphs*, New York, July 1967. Gordon and Breach.

[7] A. Noma. Análise experimental de algoritmos de planaridade. Master's thesis, Universidade de São Paulo, 2003.

[8] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Massachusetts, 1998.

[9] TheJuliaLanguage. A fresh approach to technical computing. https://github.com/JuliaLang/julia. Acesso em 27 de setembro de 2020.

[10] R. Thomas. Graph planarity and related topics. Master's thesis, School of Mathematics, Georgia Institute of Technology, 1997.

A Códigos dos algoritmos implementados em Julia

Código 1: Implementação do Algoritmo de Borůvka.

```
function boruvka_mst end

@traitfn function boruvka_mst(g::AG::(!IsDirected),
      distmx::AbstractMatrix{T} = weights(g);
      minimize = true) where {T<:Real, U, AG<:AbstractGraph{U}}

   djset = IntDisjointSets(nv(g))
   # maximizing Z is the same as minimizing -Z
   # mode will indicate the need for the -1 multiplication
   mode = minimize ? 1 : -1
   mst = Vector{edgetype(g)}()
   sizehint!(mst, nv(g) - 1)
   weight = zero(T)

   while true
      cheapest = Vector{Union{edgetype(g), Nothing}}(nothing, nv(g))
      # find cheapest edge that connects two components
      found_edge = false
      for edge in edges(g)
         set1 = find_root(djset, src(edge))
         set2 = find_root(djset, dst(edge))
         if set1 != set2
            found_edge = true
            e1 = cheapest[set1]
            if e1 === nothing || distmx[src(e1), dst(e1)] * mode > distmx[src(edge), dst(edge)] * mode
               cheapest[set1] = edge
            end
            e2 = cheapest[set2]
            if e2 === nothing || distmx[src(e2), dst(e2)] * mode > distmx[src(edge), dst(edge)] * mode
               cheapest[set2] = edge
            end
         end
      end
      # no more edges between two components
      !found_edge && break
      # add cheapest edges to the tree
      for v in vertices(g)
         if cheapest[v] !== nothing
            edge = cheapest[v]
            if !in_same_set(djset, src(edge), dst(edge))
               weight += distmx[src(edge), dst(edge)]
               union!(djset, src(edge), dst(edge))
               push!(mst, edge)
            end
         end
      end
   end
   return (mst = mst, weight = weight)
```

```
end
```

Código 2: Implementação do Algoritmo do Critério Combinatório.

```
function is_plane_description end

function is_plane_description(g::SG,
                    description::Vector{Vector{T}}) where
                    {T <: Integer, SG <: SimpleGraph{T}}

    nvg = nv(g)
    neg = ne(g)
    neg_diff = zero(Int)

    for e in edges(g)
        if src(e) != dst(e)
            neg_diff +=1
        end
    end

    loops = neg - neg_diff
    nvg >= 3 && neg_diff > 3*nvg - 6 && return false

    # the direction is important due to the clockwise description, so the second copy is reversed
    # in(e, used) indicates if edge e has been used in a facial cycle
    used = Set{edgetype(g)}()
    faces = zero(T)

    # counting facial cycles
    for e in edges(g)
        e1 = nothing
        start = nothing

        if !in(e, used)
            e1 = e
            start = e
        elseif !in(reverse(e), used)
            e1 = reverse(e)
            start = reverse(e)
        end

        if e1 !== nothing
            while true
                push!(used, e1)
                # we find the successor of src(e1) in the list of dst(e1)
                u, v = src(e1), dst(e1)
                v_list = description[v]
                u_index = findfirst(x -> x == u, v_list)

                if u_index == length(v_list)
                    e1 = Edge(v, first(v_list))
```

```
                else
                    e1 = Edge(v, v_list[u_index + 1])
                end
                e1 == start && break
            end
            faces += 1
        end
    end
    # using Euler's formula
    return faces + loops == neg - nvg + 2
end
```

Código 3: Implementação para obter uma bp-ordenação.

```
function _dfs_ord(g::AG, s::Int) where {AG <: AbstractGraph}
    dfs_ord = Vector{Int}()
    next_neigh = ones(Int, nv(g))
    parents = zeros(Int, nv(g))
    visited = falses(nv(g))
    stack = Stack{Int}()
    parents[s] = s

    push!(stack, s)
    push!(dfs_ord, s)
    visited[s] = true

    while !isempty(stack)
        v = first(stack)
        found = false
        outneigh = collect(outneighbors(g, v))
        for i = next_neigh[v]:length(outneigh)
            u = outneigh[i]
            if !visited[u]
                found = true
                next_neigh[v] = i
                visited[u] = true
                parents[u] = v
                push!(stack, u)
                push!(dfs_ord, u)
                break
            end
        end

        if !found
            pop!(stack)
        end
    end

    return dfs_ord, parents
end
```

Código 4: Implementação da Lec-Ordenação.

```
function lec_ord(g::AbstractGraph)
    s = first(vertices(g))
    lec_ord = Vector{Int}()
    visit_ord, parents = _dfs_ord(g, s)
    visiteds = visit_ord

    while true
        if isempty(visit_ord)
            return lec_ord
        end

        if isempty(lec_ord)
            v = popfirst!(visit_ord)
            push!(lec_ord, v)
            continue

        elseif length(lec_ord) == 1
            v = popfirst!(visit_ord)
            push!(lec_ord, v)
            continue
        end

        if !isempty(lec_ord) && !isempty(parents)
            v = popfirst!(visit_ord)
            u = parents[v]
            index = findfirst(isequal(u), lec_ord)
            desc_v = Vector{Int}()
            check = false

            push!(desc_v, v)
            for x in visiteds
                if parents[x] in desc_v
                    push!(desc_v, x)
                end
            end

            for i = 1:(index - 1)
                for x in desc_v
                    check = has_edge(g, x, lec_ord[i])
                    check && break
                end
                check && break
            end

            if check
                insert!(lec_ord, index, v)
                continue
            end

```

```
            for i = index + 1:length(lec_ord)
                for x in desc_v
                    check = has_edge(g, x, lec_ord[i])
                    check && break
                end
                check && break
            end

            if check
                insert!(lec_ord, index + 1, v)
                continue
            end
        end
    end
end
```

Código 5: Implementação do algoritmo que gera uma árvore de blocos a partir de uma moldura.

```
function _block_tree(g::AG, s::Int) where {AG <: AbstractGraph}

    nvg = nv(g)
    block_tree = SimpleGraph(nvg)
    nv_block_tree = nvg
    parents = zeros(Int, nvg)
    next_neigh = ones(Int, nvg)

    visited = zeros(nvg)
    stack = Stack{Int}()
    parents[s] = s
    push!(stack, s)
    visited[s] = 1
    index_visited = 2

    cnodes_groups = Array{Array{Int}, 1}()
    cnodes = 0
    while !isempty(stack)
        v = first(stack)
        found = false
        outneigh = collect(outneighbors(g, v))
        for i = next_neigh[v]:length(outneigh)
            u = outneigh[i]
            if visited[u] == 0
                found = true
                next_neigh[v] = i
                visited[u] = index_visited
                index_visited += 1
                parents[u] = v
                push!(stack, u)
                break

```

```
            elseif parents[v] != u && visited[v] > visited[u]
                vertex = v
                add_vertex!(block_tree)
                nv_block_tree += 1
                cnodes += 1
                add_edge!(block_tree, nv_block_tree, u)
                push!(cnodes_groups, [v])
                while vertex != u
                    add_edge!(block_tree, nv_block_tree, vertex)
                    vertex = parents[vertex]
                    push!(cnodes_groups[cnodes], vertex)
                end
            end
        end

        if !found
            pop!(stack)
        end
    end

    cut_edges = bridges(g)
    for e in cut_edges
        add_vertex!(block_tree)
        nv_block_tree += 1
        push!(cnodes_groups, [src(e), dst(e)])
        add_edge!(block_tree, nv_block_tree, src(e))
        add_edge!(block_tree, nv_block_tree, dst(e))
    end

    return (block_tree, cnodes_groups)
end
```

Código 6: Implementação do algoritmo que constrói um XY-caminho ou detecta uma XY-obstrução.

```
function _xy_path(frame::SimpleGraph, X::Set{Int}, Y::Set{Int})

    # find the block tree of the frame
    bt, cnodes_groups = _block_tree(frame, 1)
    cnodes_num = length(cnodes_groups)
    original_deg = degree(bt)
    # degrees[i] will be -1 if the vertex is not in the current block tree
    degrees = degree(bt)
    nv_bt = nv(bt)
    nv_frame = nv(frame)

    # auxiliar structures
    x_curr = copy(X)
    y_curr = copy(Y)
    w = copy(X)
    z = copy(Y)
```

```
17
18      # xc (resp. yc) contains nodes from the group which are in W (resp. Z) or in the current tree but
        ↪  are not c-nodes
19      xc = Set{Int}()
20      yc = Set{Int}()
21
22      leaves_not = Array{Int, 1}()
23
24      # sets X and Y will be updated as we remove vertices from the block tree
25      # X will keep original X set
26      # x will keep current X set
27      # w will keep vertices from the frame neighbors of a vertex in x_curr when such vertex is removed
28
29      # verifying which vertices of the current tree are leaves but are not in both X and Y
30      for i = 1:nv_bt
31          if degrees[i] == 1 && !(i in x_curr && i in y_curr)
32              push!(leaves_not, i)
33          end
34      end
35
36      while true
37
38          # if the leaves are all in both X and Y, we reached a base case
39          # we must verify whether an XY-path can be formed or if there is an obstruction
40          if isempty(leaves_not)
41
42              # if the remaining tree is not a path (thus there is a vertex with degree >= 3), then there
                ↪  is an obstruction
43              for i = 1:nv_bt
44                  if degrees[i] != -1 && degrees[i] >= 3
45                      return nothing, nothing
46                  end
47              end
48
49              # otherwise, we must try to build a special kind of path for each remaining c-node, called
                ↪  c-path
50              c_paths = Array{edgetype(frame), 1}()
51              for c = 1:cnodes_num
52                  if degrees[c + nv_frame] != -1
53                      # group will have the nodes from the frame which are in the block of the c-node
                        ↪  being analyzed
54                      group = cnodes_groups[c]
55
56                      for v in group
57                          if v in w || (v <= nv_frame && degrees[v] != -1)
58                              push!(xc, v)
59                          end
60
61                          if v in z || (v <= nv_frame && degrees[v] != -1)
62                              push!(yc, v)
63                          end
64                      end
```

```
65
66                      # auxiliar
67                      len_xc = length(xc)
68                      len_yc = length(yc)
69                      len_group = length(group)
70
71                      # the special path must contain all vertices in xc and no vertex of yc can appear
                        ↪ internally
72                      # every p-node currently in the tree which is a neighbor of the c-node must be an
                        ↪ extreme of the path
73                      extremes = Array{Int, 1}()
74                      for v in neighbors(bt, (c + nv_frame))
75                          if degrees[v] != -1
76                              push!(extremes, v)
77                          end
78                      end
79
80                      # if there is no p-node neighbor of the c-node, then the remaining tree contains
                        ↪ only one c-node
81                      # therefore the c-path must only cover all vertices in xc without having vertices
                        ↪ from yc internally
82                      if isempty(extremes)
83
84                          # entry and exit will tell the positions, in group, where the c-path starts and
                            ↪ ends
85                          entry = exit = zero(Int)
86                          # if there is less than 2 vertices in yc, then the c-path can trivially be
                            ↪ built
87                          if len_yc < 2
88                              index_y = findfirst(v->v in yc, group)
89                              # if there is no vertex in yc, the path must only cover all vertices in xc
90                              if index_y === nothing
91                                  for i = 1:len_group
92                                      if (group[i] in xc && entry == 0) entry = i end
93                                      if group[i] in xc exit = i end
94                                  end
95                                  for i = entry:exit - 1
96                                      push!(c_paths, Edge(group[i], group[i + 1]))
97                                  end
98                              # otherwise the path must start at such vertex, because otherwise it would
                                ↪ be internal to the path
99                              else
100                                 entry = index_y
101                                 for i = union(index_y + 1:len_group, 1:index_y - 1)
102                                     if group[i] in xc exit = i end
103                                 end
104                                 if entry <= exit
105                                     for i = entry:exit-1
106                                         push!(c_paths, Edge(group[i], group[i + 1]))
107                                     end
108                                 else
109                                     push!(c_paths, Edge(group[len_group], group[1]))
```

```
                        for i = union(entry:len_group - 1, 1:exit - 1)
                            push!(c_paths, Edge(group[i], group[i + 1]))
                        end
                    end
                end
            # if there is 2 or more vertices in yc, then we have an obstruction if group is
            ↪  of one of the following forms:
            # (... xc ... yc ... xc ... yc ...)
            # (... yc ... xc ... yc ... xc ...)
            else
                first_y = toggled = false
                entry = entry2 = exit = zero(Int)
                for i = 1:len_group
                    if group[i] in yc
                        if entry == 0 first_y = true end
                        if toggled && entry2 != 0 return (nothing, nothing) end
                        if entry != 0 toggled = true end
                    end
                    if group[i] in xc
                        if entry == 0 entry = i end
                        if !toggled  exit = i end
                        if toggled && entry2 == 0 entry2 = i end
                        if entry2 != 0 && first_y return (nothing, nothing) end
                    end
                end

                # if there is no obstruction then group is of one of the following forms:
                # (... xc ... yc ... xc ...)
                # (... yc ... xc ... yc ...)
                # (... xc ... yc ...)
                # (... yc ... xc ...)
                if entry2 == 0
                    for i = entry:exit - 1
                        push!(c_paths, Edge(group[i], group[i + 1]))
                    end
                else
                    push!(c_paths, Edge(group[len_group], group[1]))
                    for i = union(entry2:len_group - 1, 1:exit - 1)
                        push!(c_paths, Edge(group[i], group[i + 1]))
                    end
                end
            end

        # if there is some p-node remaining, then the extremes of the c-path is predefined
        else

            entry = findfirst(isequal(first(extremes)), group)
            exit = findfirst(isequal(last(extremes)), group)
            entry, exit = min(entry, exit), max(entry, exit)
            last_xc = 0

            # if the c-node is a leaf of the tree, then only one of the extremes of the
            ↪  c-path is predefined
```

```
161                         if entry == exit
162                             # we must verify if it is possible to start at such p-node and cover all
                                ↪ vertices in xc
163                             # without passing through vertices if yc (except if it is an extreme of the
                                ↪ path)
164
165                             # we must walk through the group in two directions (group is a cycle)
166                             count_xc = len_xc
167                             for i = union(entry:len_group, 1:entry - 1)
168                                 count_xc = group[i] in xc ? count_xc - 1 : count_xc
169                                 # if a vertex of yc not in xc is reached, there are still vertices in
                                    ↪ xc to be covered and some vertices of xc were already covered, then
                                    ↪ there is an obstruction
170                                 last_xc = group[i] in xc ? i : last_xc
171                                 group[i] in yc && i != entry && break
172                             end
173
174                             # if all vertices of xc were covered in this direction, we found our c-path
175                             if count_xc == 0
176                                 if last_xc > entry
177                                     for i = entry:last_xc - 1
178                                         push!(c_paths, Edge(group[i], group[i+1]))
179                                     end
180                                 else
181                                     push!(c_paths, Edge(group[len_group], group[1]))
182                                     for i = union(entry:len_group - 1, 1:last_xc - 1)
183                                         push!(c_paths, Edge(group[i], group[i+1]))
184                                     end
185                                 end
186                             # otherwise we must try in the other direction
187                             else
188                                 count_xc = len_xc
189                                 for i = union(reverse(1:entry), reverse(entry+1:len_group))
190                                     count_xc = group[i] in xc ? count_xc - 1 : count_xc
191                                     last_xc = group[i] in xc ? i : last_xc
192                                     group[i] in yc && i != entry && break
193                                 end
194
195                                 # if not all vertices were covered in this direction, then there is an
                                    ↪ obstruction
196                                 count_xc != 0 && return nothing, nothing
197
198                                 if last_xc > entry
199                                     push!(c_paths, Edge(group[len_group], group[1]))
200                                     for i = union(last_xc:len_group - 1, 1:entry)
201                                         push!(c_paths, Edge(group[i], group[i+1]))
202                                     end
203                                 else
204                                     for i = last_xc:entry - 1
205                                         push!(c_paths, Edge(group[i], group[i+1]))
206                                     end
207                                 end
```

```
                        end

                    # at last, the c-node is not a leaf of the tree, so both extremes are defined
                    else

                        # there is only two directions to follow to try and cover all vertices in
                        ↪ xc
                        count_xc = len_xc
                        for i = entry:exit
                            count_xc = group[i] in xc ? count_xc - 1 : count_xc
                            group[i] in yc && i != entry && i != exit && break
                        end

                        if count_xc == 0
                            for i = entry:exit - 1
                                push!(c_paths, Edge(group[i], group[i+1]))
                            end
                        else
                            count_xc = len_xc
                            for i = union(exit:len_group, 1:entry)
                                count_xc = group[i] in xc ? count_xc - 1 : count_xc
                                group[i] in yc && i != entry && i != exit && break
                            end

                            count_xc != 0 && return nothing, nothing

                            push!(c_paths, Edge(group[len_group], group[1]))
                            for i = union(exit:len_group - 1, 1:entry - 1)
                                push!(c_paths, Edge(group[i], group[i+1]))
                            end
                        end
                    end
                end
                # clearing xc and yc so they can be used for the next c-node
                empty!(xc)
                empty!(yc)
            end
        end

        # now that we built all c-paths for all c-nodes, we must reunite them into one path
        sg, vmap = induced_subgraph(frame, c_paths)
        nv_sg = nv(sg)
        degrees_sg = degree(sg)
        extremes = zeros(Int, 2)
        comp_path = Array{edgetype(frame), 1}()
        path_set = Set{edgetype(frame)}()

        for e in c_paths
            min, max = minmax(src(e), dst(e))
            push!(path_set, Edge(min, max))
        end
        c_paths = nothing
```

```
259
260             for i = 1:nv_sg
261                 if degrees_sg[i] == 1
262                     if extremes[1] == 0
263                         extremes[1] = vmap[i]
264                     else
265                         extremes[2] = vmap[i]
266                         break
267                     end
268                 end
269             end
270             visited = falses(nv_frame)
271             next_neigh = ones(Int, nv_frame)
272
273             for i = 1:nv_frame
274                 visited[i] = degrees[i] != -1
275             end
276
277             # we then must extend the path so that its extremes are in the original X set
278             for s in extremes
279                 s == 0 && break
280                 curr = s
281                 while !(curr in X)
282                     for v in neighbors(frame, curr)
283                         e = sorted_edge(curr, v)
284                         if v in w && !in(e, path_set)
285                             curr = v
286                             push!(path_set, e)
287                             break
288                         end
289                     end
290                 end
291             end
292
293             # we now collect the blocks through which the path passes
294             related_blocks = Array{Int, 1}()
295             for b = 1:length(cnodes_groups)
296                 block = cnodes_groups[b]
297                 e = sorted_edge(first(block), last(block))
298                 if in(e, path_set)
299                     push!(related_blocks, b)
300                     continue
301                 end
302                 for i = 1:length(block) - 1
303                     e = sorted_edge(block[i], block[i + 1])
304                     if in(e, path_set)
305                         push!(related_blocks, b)
306                         break
307                     end
308                 end
309             end
310
```

```
311             # and build the complement path, which passes on the same blocks
312             for num_block in related_blocks
313                 block = cnodes_groups[num_block]
314                 if length(block) == 2
315                     push!(comp_path, Edge(first(block), last(block)))
316                 else
317                     e = sorted_edge(first(block), last(block))
318                     if !in(e, path_set)
319                         push!(comp_path, Edge(first(block), last(block)))
320                     end
321                     for i = 1:length(block) - 1
322                         e = sorted_edge(block[i], block[i + 1])
323                         if !in(e, path_set)
324                             push!(comp_path, e)
325                         end
326                     end
327                 end
328             end
329
330             # we return the path and the complement path
331             return induced_subgraph(frame, collect(path_set)), induced_subgraph(frame, comp_path)
332
333         # if there are leaves of the tree which are not in the intersection of X and Y, we will remove
            ↪  one of them
334         else
335             leaf = pop!(leaves_not)
336             # find the vertex in the tree which is neighbor of this leaf
337             u = 0
338             for v in neighbors(bt, leaf)
339                 if degrees[v] != -1
340                     u = v
341                     break
342                 end
343             end
344
345             # remove the leaf from the tree
346             degrees[leaf] = -1
347             # decrement the degree of its parent
348             if u != 0 degrees[u] -= 1 end
349
350             # if u is a p-node and leaf was in X or Y, we must update sets W or Z
351             # in any case, sets X and Y should be updated
352             is_pnode = u <= nv_frame
353             if leaf in x_curr
354                 delete!(x_curr, leaf)
355                 if u != 0
356                     push!(x_curr, u)
357                     if is_pnode push!(w, u) end
358                 end
359             end
360
361             if leaf in y_curr
```

```
                delete!(y_curr, leaf)
                if u != 0
                    push!(y_curr, u)
                    if is_pnode push!(z, u) end
                end
            end

            # if u became a leaf, then we must check if it is in the intersection of X and Y
            if u != 0 && degrees[u] == 1 && !(u in x_curr && u in y_curr)
                push!(leaves_not, u)
            end
        end
    end #while
end #function

function sorted_edge(u::Int, v::Int)
    min, max = minmax(u, v)
    return Edge(min, max)
end
```

Código 7: Implementação do algoritmo de LEC.

```
function lec(g::SimpleGraph)

    # find a lec-ordering of g
    lec_ord_list = lec_ord(g)
    # in_subgraph will maintain the set of visited vertices
    in_subgraph = falses(nv(g))
    # frame will maintain a frame of already visited vertices
    frame = SimpleGraph()
    frame_vmap = Array{Int, 1}()
    # R will help extend the frame whenever a new vertex is visited
    R = Set{Int}()

    add_vertex!(frame)
    vertex = popfirst!(lec_ord_list)
    push!(frame_vmap, vertex)
    in_subgraph[vertex] = true

    while !isempty(lec_ord_list)

        # w is removed from the lec-ordering and visited
        w = popfirst!(lec_ord_list)
        in_subgraph[w] = true

        # X must contain vertices of the frame that are neighbors of w
        X = Set{Int}()
        # Y must contain vertices of the frame that are neighbors of vertices that are not in the
        ↪ subgraph
        Y = Set{Int}()
        for u in vertices(frame)
```

```
            if has_edge(g, frame_vmap[u], w)
                push!(X, u)
            end
            for v in neighbors(g, frame_vmap[u])
                if !in_subgraph[v]
                    push!(Y, u)
                    break
                end
            end
        end

        # If X has only one vertex, then the XY-path is defined to be that vertex
        if length(X) == 1
            paths = ((SimpleGraph(1), collect(X)), (SimpleGraph(1), collect(X)))
        else
            paths = _xy_path(frame, X, Y)
        end

        # if there is an XY-obstruction, then the graph is definitely not planar
        if paths === (nothing, nothing)
            return false
        end

        # otherwise, _xy_path returned the path (xy_path) and its complement (comp_path)
        (xy_path, xy_vmap), (comp_path, comp_vmap) = paths

        # R must contain vertices that need to pass through vertices of the complement path
        # in order to reach a vertex in Y
        visited = falses(nv(frame))
        for v in comp_vmap visited[v] = true end
        empty(R)
        for z in vertices(frame)
            has_y = false
            if !visited[z]
                visited[z] = true
                has_y = z in Y
                comp = Array{Int, 1}()
                if !has_y push!(comp, z) end
                queue = Queue{Int}()
                enqueue!(queue, z)

                while !isempty(queue)
                    u = dequeue!(queue)
                    for v in neighbors(frame, u)
                        if !visited[v]
                            visited[v] = true
                            enqueue!(queue, v)
                            if v in Y has_y = true end
                            if !has_y push!(comp, v) end
                        end
                    end
                end
```

```
            if !has_y union!(R, comp) end
        end
    end

    # now we can update the frame
    old_vmap = frame_vmap
    frame_vertices = Array{Int, 1}()
    frame_orig_vmap = Array{Int, 1}()

    # first we need to remove the edges of the xy_path from the frame
    bridges_set = Set{edgetype(frame)}(bridges(frame))
    for e in edges(xy_path)
        ef = Edge(xy_vmap[src(e)], xy_vmap[dst(e)])
        if !(ef in bridges_set || reverse(ef) in bridges_set)
            rem_edge!(frame, ef)
            rem_edge!(frame, reverse(ef))
            # if by any change the vertex became isolated, we will remove it too
            if degree(frame, src(e)) == 0 push!(R, src(e)) end
            if degree(frame, dst(e)) == 0 push!(R, dst(e)) end
        end
    end

    # and remove the vertices from R from the frame
    for v in vertices(frame)
        if !(v in R)
            push!(frame_vertices, v)
            push!(frame_orig_vmap, frame_vmap[v])
        end
    end

    # the frame now must include vertex w and the edges that connect w to the extremes of the
    ↪  complement path
    frame, frame_vmap = induced_subgraph(frame, frame_vertices)
    frame_vmap = frame_orig_vmap
    add_vertex!(frame)
    push!(frame_vmap, w)
    n = nv(frame)

    # the exception is that if the complement path has only one vertex, then only one edge will be
    ↪  added to the frame
    if nv(comp_path) == 1
        for i = 1:length(frame_vmap)
            if frame_vmap[i] == old_vmap[comp_vmap[1]]
                add_edge!(frame, i, n)
                break
            end
        end
    end
    for v in vertices(comp_path)
        if degree(comp_path, v) == 1
            for i = 1:length(frame_vmap)
                if frame_vmap[i] == old_vmap[comp_vmap[v]]
```

```
                        add_edge!(frame, i, n)
                        break
                    end
                end
            end
        end
    end

    # if an obstruction was never found, then the graph must be planar
    return true
end
```