Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Desenho retangular em grade para DCELs

André Victor dos Santos Nakazawa

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
2021

Agradecimentos

Ao meu orientador Carlos Eduardo Ferreira, carinhosamente conhecido como Carlinhos, por todo o seu apoio sem o qual o desenvolvimento deste trabalho não teria acontecido.

Aos meus pais, pelo enorme carinho e compreensão que foram essenciais em toda a minha vida.

Ao meu melhor amigo, Rodrigo Aparecido Enju, pela sua amizade e nossas conversas que me ajudaram nos momentos mais difíceis.

A todos que, mesmo que não mencionados diretamente neste trabalho, contribuíram de alguma forma para a minha formação.

Resumo

André Victor dos Santos Nakazawa. **Desenho retangular em grade para DCELs**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Um desenho retangular de um grafo G é uma imersão plana de G em que todas as arestas são desenhadas como segmentos de reta verticais ou horizontais e todas as faces são retangulares. O algoritmo de desenho retangular para um grafo plano com cantos fixos é utilizado na construção de outras técnicas de desenhos de grafos, como desenhos caixa-retangulares e desenhos ortogonais, e é base para o algoritmo de desenhos retangulares generalizado para grafos planares. Neste trabalho, apresentamos uma implementação do algoritmo de desenho retangular para grafos planos com cantos fixos representados por listas de arestas duplamente ligadas. O algoritmo que desenvolvemos consome tempo linear e, como resultado do algoritmo, um desenho retangular com coordenadas em grade é construído.

Palavras-chave: Desenho de grafos. Grafo plano. Desenho retangular. Desenho em grade. Listas de arestas duplamente ligadas.

Abstract

André Victor dos Santos Nakazawa. **Rectangular grid drawing of DCELs**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

A rectangular drawing of a graph G is a planar embedding of G such that each edge is drawn as a vertical or horizontal line segment and each face is a rectangle. The rectangular drawing algorithm for plane graphs with fixed corners is used on other graph drawing techniques, such as box-rectangular drawing and orthogonal drawing, and it is fundamental in the development of the more general rectangular drawing algorithm for planar graphs. In this study, we present an implementation of the rectangular drawing algorithm for plane graphs with fixed corners represented as doubly connected edge lists. The algorithm implemented has linear time complexity and, as a result of this algorithm, a rectangular drawing on a grid is obtained.

Keywords: Graph drawing. Plane graph. Rectangular drawing. Grid drawing. Doubly connected edge list.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Grafos são extensamente utilizados em diversos âmbitos da sociedade, tanto pela possibilidade de representar visualmente as relações entre coisas, quanto pelo seu apelo estético quando desenhado. Não à toa, existem teoremas dentro da teoria dos grafos bastante conhecidos como o Teorema das Quatro Cores e o Teorema de Kuratowski, e que estão relacionados às propriedades de desenhos de grafos.

Na área de desenho de grafos, diversas teorias e técnicas foram sendo desenvolvidas ao longo dos anos e relacionadas aos mais diversos tipos de desenhos: planar, tridimensional, retangular, ortogonal, direcionados por força, etc. (o livro-texto de Tamassia é uma ótima referência para explorar esse universo (Tamassia, 2016)). Mas, neste trabalho estudamos apenas uma pequena parte desse mundo, os desenhos retangulares de grafos.

Num desenho retangular, o grafo é desenhado de tal forma que vemos um retângulo formado por partes menores também retangulares. Tais desenhos retangulares de grafos são interessantes não só pela teoria, mas os métodos desenvolvidos podem ser aplicados em projeto de circuitos VLSI, por exemplo, assim como outros estudos relacionados a grafos (Ellis-Monaghan e Gutwin, 2003).

Sobretudo, será possível encontrar neste trabalho um detalhamento de alguns resultados desenvolvidos em (M. Rahman, Nakano *et al.*, 1998) e uma explicação geral sobre a implementação do algoritmo para o desenho retangular de grafos planos com cantos fixos.

O ponto de partida para esse estudo foi a dissertação de Assunção (Assunção, 2012), que trabalhou em cima dos problemas de representação retangular e de desenho retangular. Então, tomamos os artigos de Rahman, Nakano, Nishizeki e Ghosh como fonte principal para estudo e desenvolvimento dos códigos (M. Rahman, Nakano *et al.*, 1998), (M. Rahman, Nakano *et al.*, 2002), (M. Rahman, Nishizeki *et al.*, 2004).

1.1 Noções preliminares

Um **grafo** G é um par ordenado $G = (V, E)$ onde V é o conjunto de **vértices** e E é o conjunto de **arestas**. Um vértice v é um elemento de V, enquanto uma aresta e é um elemento de E. Se $v \in V$ também dizemos que v está contido em G e, igualmente, se

$e \in E$ dizemos que e está contido em G. Cada aresta é um conjunto $e = \{u, v\}$ de vértices distintos u e v, ou seja, $E \subseteq \binom{V}{2}$ onde $\binom{V}{2}$ representa a família dos subconjuntos de V com cardinalidade 2. Também podemos utilizar a notação uv para denotar a aresta $\{u, v\}$. Ademais, é comum utilizarmos $V(G)$ para denotar o conjunto de vértices de G e $E(G)$ para denotar o conjunto de arestas de G.

Dados dois vértices u e v, dizemos que u é **adjacente** a v se $e = uv \in E$. Nesse caso, os vértices u e v são as **pontas** de e, e dizemos que u é **incidente** a e. Igualmente, dizemos que v é incidente a e.

Dado um vértice v, o número de arestas incidentes a v é o **grau** de v, denotado por $\text{grau}(v)$. Também definimos o **grau máximo** e **grau mínimo** de um grafo G como $\Delta(G) = \max_{v \in V} \text{grau}(v)$ e $\delta(G) = \min_{v \in V} \text{grau}(v)$, respectivamente.

Um **grafo completo** de n vértices, denotado por K_n, é um grafo em que $|V| = n$ e cada vértice é adjacente a todos os outros.

Um grafo H é dito **subgrafo** de um grafo G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Dois grafos G_1 e G_2 são **isomorfos** se existe uma função $\varphi : V(G_1) \longrightarrow V(G_2)$ tal que, dados dois vértices u e v, $uv \in E(G_1)$ se e somente se $\varphi(u)\varphi(v) \in E(G_2)$. Nesse caso, dizemos que φ é um isomorfismo entre G_1 e G_2 e denotamos que G_1 e G_2 são isomorfos por $G_1 \simeq G_2$.

Dados dois grafos G_1 e G_2, a **união** entre G_1 e G_2 denotada por $G = G_1 \cup G_2$ é o grafo tal que $V(G) = V(G_1) \cup V(G_2)$ e $E(G) = E(G_1) \cup E(G_2)$. Similarmente, a **intersecção** entre G_1 e G_2 denotada por $G' = G_1 \cap G_2$ é o grafo tal que $V(G') = V(G_1) \cap V(G_2)$ e $E(G') = E(G_1) \cap E(G_2)$.

Dado um conjunto de vértices $V' \subseteq V$, o grafo $G' = G - V'$ obtido de G por uma **remoção de vértices** é tal que $V(G') = V(G) - V'$ e $E(G') = E(G) \cap \binom{V(G)-V'}{2}$.

Dado um grafo G e uma aresta $e = uv$, a **contração** de e é a operação que transforma G em um grafo G' onde $V(G') = V(G) \cup \{w\} - \{u, v\}$, sendo que $w \notin V(G)$, e $E(G') = \{e \in E(G) : e \cap \{u, v\} = \emptyset\} \cup E'$ com $E' = \{xw : xu \in E(G)\} \cup \{wx : vx \in E(G)\}$. Ou seja, w é um novo vértice em G' que representa a junção das pontas da aresta contraída. Assim, w é adjacente a todos os vértices adjacentes a u ou v no grafo original G.

Um **passeio** P é uma sequência finita de vértices $P = (v_1, v_2, \dots, v_k)$. Podemos nos referir a v_1 e v_k como pontas de P e também dizer que P é um passeio entre v_1 e v_k. Dizemos que P é **aberto** se $v_1 \neq v_k$ e **fechado** caso contrário. O **comprimento** de P é o comprimento da sequência, ou seja, k.

Um **caminho** $P = (v_1, v_2, \dots, v_k)$ é um passeio em que nenhum dos vértices são iguais, ou seja, $v_i \neq v_j$ para $1 \leq i < j \leq k$. Um **subcaminho** P' de um caminho P é um caminho $P' = (v_{k_1}, \dots, v_{k_2})$ com $1 \leq k_1 < k_2 \leq k$. Dizemos que P' é um **prefixo** se $k_1 = 1$. Similarmente, P' é um **sufixo** se $k_2 = k$. Em alguns casos, podemos identificar um caminho P como um grafo e nos referir a seus vértices e arestas. Dados dois caminhos $P = (v_1, v_2, \dots, v_k)$ e $Q = (u_1, u_2, \dots, u_l)$, se $v_i \neq u_j$ para $1 \leq i < k$ e $1 < j \leq l$ e $v_k = u_1$, definimos a **concatenação** $P \cdot Q$ como o caminho $P \cdot Q = (v_1, v_2, \dots, v_k = u_1, u_2, \dots, u_l)$. Um **ciclo** C é um caminho em que as pontas são adjacentes.

Um grafo G é **conexo** se dados dois vértices distintos u e v contidos em G, existe

um caminho entre u e v. Dado um grafo G qualquer, dizemos que um subgrafo H é uma **componente conexa** de G se H é um subgrafo conexo maximal de G, ou seja, H é conexo e qualquer subgrafo H' de G tal que H é subgrafo de H' e $H \neq H'$ não é conexo.

Dado um grafo conexo G, dizemos que um conjunto de vértices S **separa** G se $G - S$ não é conexo. Também podemos dizer que S separa subgrafos, arestas ou vértices que estão em componentes conexas distintas de $G - S$ nesse caso. Definimos que G é k**-conexo**, sendo $k \geq 1$ inteiro, se $|V(G)| \geq k + 1$, e $G \simeq K_{k+1}$ ou não existe um conjunto de vértices S com $|S| < k$ tal que S separa G.

Uma **árvore** T é um grafo acíclico conexo, isto é, T é conexo e não há ciclos em T. Dado um grafo G, se T é uma árvore e subgrafo de G, dizemos que T é **árvore geradora** de G se $V(T) = V(G)$.

Um **grafo plano** é uma imersão plana de um grafo G, isto é, um desenho de G em que os vértices são representados como pontos e as arestas são representadas como curvas entre suas pontas. Numa imersão plana, duas arestas distintas nunca se intersectam, exceto nas pontas quando há uma ponta em comum entre elas. Nem todo grafo G admite uma imersão plana. Se G admite uma imersão plana dizemos que G é **planar**.

A imersão plana de um grafo G divide o plano em regiões conexas chamadas **faces**. Sendo que apenas uma dessas faces é ilimitada, tal face é o que chamamos de **face externa**.

Dada uma face f de um grafo plano G, definimos o **contorno** de f denotado por $C(f)$ como o passeio fechado formado pelas arestas que estão na fronteira de f. O **contorno externo** de G, denotado por $C_o(G)$ é o contorno da face externa. Dizemos que uma aresta e é **incidente** à face f se e está no contorno de f.

Dado um grafo plano G, dizemos que e é uma **aresta externa** de G se e está em $C_o(G)$. Caso contrário, dizemos que e é uma **aresta interna**. Se um passeio $P = (v_1, v_2, \ldots, v_k)$ é tal que todas as arestas $v_i v_{i+1}$ com $1 \leq i < k$ são internas, dizemos que P é um **passeio interno**.

Dado um ciclo C e um grafo plano G, definimos $G(C)$ como o **grafo plano contido em** C, incluindo o próprio C. Note que $C_o(G(C)) = C$.

1.2 Definição do problema

Um **desenho retangular** de um grafo G é um grafo plano D isomorfo a G tal que:

(i) cada aresta de D é desenhada como um segmento vertical ou horizontal, e

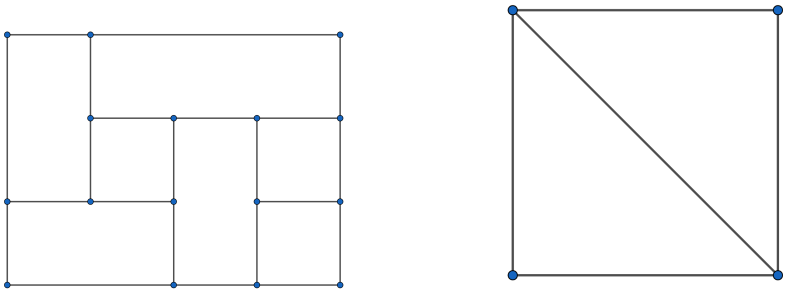
(ii) cada face de D é um retângulo, incluindo a face externa C_o

Os desenhos retangulares que estudaremos neste trabalho pressupõe $\Delta(G) \leq 3$. Ao eliminarmos a possibilidade de um vértice de grau 4 de um desenho retangular, conseguimos garantir que se duas faces compartilham um vértice, também compartilham uma aresta. Um fato importante é que pela **fórmula de Euler** para grafos planos temos

$$|V| - |E| + |F| = 2$$

Logo, sendo $n = |V|$, se $\Delta(G) \leq 3$ então $|E| = O(n)$ e $|F| = O(n)$.

Nem todo grafo admite um desenho retangular. Por outro lado, há grafos que admitem diversos desenhos retangulares. Confira os exemplos da figura 1.1.



(a) *Desenho retangular de um grafo.*

(b) *Desenho de um grafo que não tem desenho retangular.*

Figura 1.1: *Exemplo de dois grafos, um que admite desenho retangular e outro que não.*

Devemos nos atentar ao fato de que na definição de desenho retangular exigimos que a face externa de D também seja um retângulo. Logo, o grafo plano da figura 1.2 pode parecer um desenho retangular, mas não é.

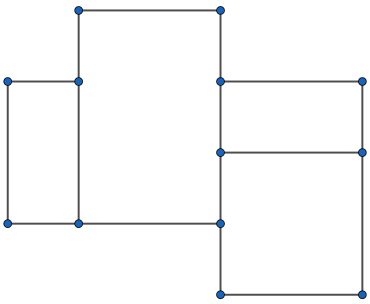


Figura 1.2: *Desenho que não é retangular.*

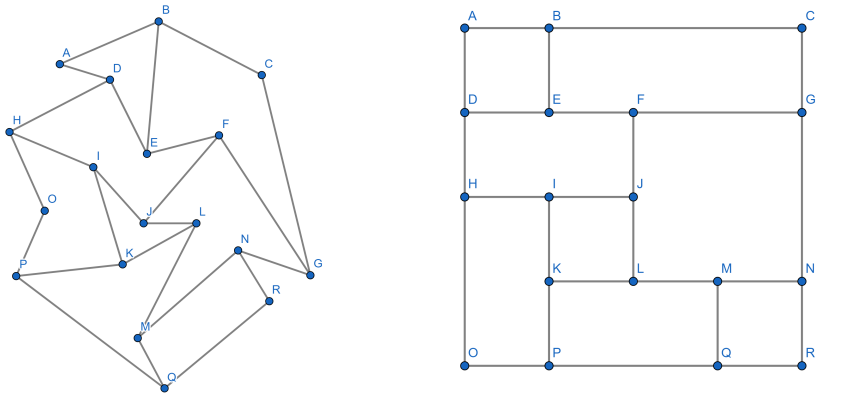
O **problema do desenho retangular** pode ser dividido em dois tipos de problemas: o problema da decisão e o problema da construção.

PROBLEMAS (I)

Problema da decisão para grafos: *Dado um grafo G, decidir se existe um desenho retangular D de G.*

Problema da construção para grafos: *Dado um grafo G que tem desenho retangular, construir o desenho retangular D certificando que G tem desenho retangular.*

A entrada para ambos os problemas é um grafo G, mas a saída para o primeiro problema é uma resposta binária SIM/NÃO, enquanto para o segundo é o próprio desenho (como no caso da figura 1.3).



(a) *Um exemplo de entrada: grafo G.* **(b)** *Um exemplo de saída: grafo plano D*

Figura 1.3: *Exemplo não tão intuitivo em que um grafo admite desenho retangular.*

Em um contexto geral de computação, estamos interessados em ambos os problemas. Idealmente, queremos que um problema com entrada E tenha solução eficiente (S, C), sendo a saída $S \in \{\text{SIM}, \text{NÃO}\}$, e C um objeto que possamos verificar, também eficientemente, para certificar a corretude de S. Porém, nem sempre conseguimos obter uma solução como essa.

Outra distinção que pode ser feita entre os problemas é em relação ao problema restrito a grafos planos.

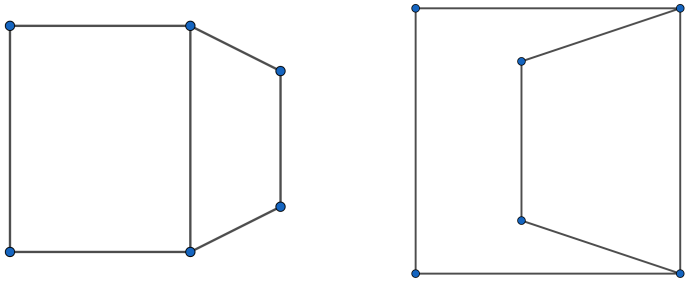
PROBLEMAS (II)

Problema da decisão para grafos planos: *Dado um grafo plano G, decidir se existe um desenho retangular D mantendo C_o como face externa e a relação de vizinhança entre as faces de G.*

Problema da construção para grafos planos: *Dado um grafo plano G que tem desenho retangular mantendo C_o como face externa e também mantendo a relação de vizinhança entre as faces, construir o desenho retangular D certificando que G tem desenho retangular com tais propriedades.*

A distinção entre os problemas anteriores e esses fica mais clara com o seguinte exemplo da figura 1.4. Nessa figura, temos desenhos distintos do mesmo grafo.

Ao explorar a relação de vizinhança, entramos no conceito de dualidade. O dual de um grafo G, de forma simples, é um outro grafo H em que os vértices de H correspondem às



(a) *Um exemplo de grafo plano G que tem saída SIM em (I) e também em (II).*

(b) *Um exemplo de grafo plano G que tem saída SIM em (I) e NÃO em (II).*

Figura 1.4: *Exemplos para ilustrar a distinção entre os problemas.*

faces de G, e cada aresta de H representa que as faces correspondentes às suas pontas são vizinhas em G.

O problema de representação planar, estudado por Assunção em sua dissertação de mestrado (Assunção, 2012), é um problema dual ao problema do desenho retangular. Mais precisamente, a entrada para o problema da representação é o dual da entrada do problema para o desenho. Como pode ser visto na figura 1.5, o grafo de entrada representa a vizinhança entre as faces (internas).

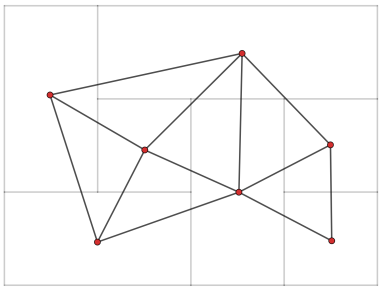


Figura 1.5: *Um grafo G (entrada) para o problema da representação retangular, sobreposto à representação retangular (saída).*

Podemos construir a entrada para o problema de representação retangular a partir da entrada para o desenho retangular e vice-versa em tempo $O(n)$, $n = |V|$, de modo que a saída seja invariante. Logo, os problemas de desenho retangular e representação retangular podem ser considerados equivalentes.

Rahman, Nakano e Nishizeki desenvolveram primeiramente os resultados para problemas de desenho retangular de grafos planos. A solução para o desenho retangular de grafos em geral foi desenvolvida com Ghosh e pressupõe tais resultados anteriores. Portanto, nosso foco neste trabalho será nos problemas envolvendo grafos planos.

1.3 Estrutura deste Trabalho

No capítulo 2, trazemos os resultados encontrados nos trabalhos (M. Rahman, Nakano *et al.*, 1998) e (Assunção, 2012) que utilizamos de referência. Nele, será possível encontrar os fundamentos teóricos necessários para o entendimento do algoritmo de desenho retangular para grafos planos com cantos fixos e do algoritmo de desenho retangular em grade. Ao final, indicamos referências para trabalhos relacionados a este que desenvolvemos.

Uma explicação geral da implementação feita pode ser encontrada no capítulo 3. Nesse capítulo, introduzimos a estrutura de dados utilizada e explicamos as funções principais do código desenvolvido. Também mostramos como a eficiência do algoritmo pode ser garantida por meio da memoização que implementamos.

Capítulo 2

Desenvolvimento Teórico

2.1 Desenho retangular de grafo plano com cantos fixos

Nesta seção iremos apresentar o trabalho desenvolvido por Rahman, Nakano e Nishizeki para solução do problema do desenho retangular para grafos planos (M. Rahman, Nakano *et al.*, 1998). A solução proposta neste artigo pressupõe que o grafo plano de entrada G tenha 4 vértices predeterminados como cantos. Ou seja, o algoritmo encontrará um desenho retangular de G somente se existir um desenho com tais 4 cantos fixados. Caso existam desenhos retangulares de G, mas nenhum deles tenha os mesmos 4 cantos que os predeterminados, a resposta do algoritmo será negativa. Outra hipótese que não deve ser negligenciada nesse trabalho é que G é 2-conexo e tem todos os vértices com grau 3 exceto os 4 cantos, que devem ter grau 2.

Os 4 vértices que chamamos de cantos são v_{NO}, v_{NE}, v_{SE} e v_{SO}, vértices situados em $C_o(G)$, no canto superior esquerdo ("noroeste"), canto superior direito ("nordeste"), canto inferior direito ("sudoeste") e canto inferior esquerdo ("sudoeste"), respectivamente. Os cantos separam $C_o(G)$ em 4 caminhos, caminho norte P_N, caminho leste P_L, caminho sul P_S e caminho oeste P_O, conforme a figura 2.1. Note que, ao definirmos os cantos, fixamos o $C_o(G)$ em forma de um retângulo.

Sendo H um subgrafo de G, dizemos que H é uma C_o-componente de G se:

(a) $H \simeq K_2$, seus vértices estão em C_o mas sua única aresta não está, ou

(b) H é formado por uma componente conexa K de $G - V(C_o)$ e as arestas que existem entre K e C_o, incluindo as pontas que estão em C_o

A figura 2.2 ilustra as C_o-**componentes** de um grafo plano, diferenciando cada uma delas por estilo e cor.

Dado C um ciclo de G, uma **perna** de C é qualquer aresta incidente em um vértice de C que esteja fora de C, ou seja, não contido em $G(C)$.

Lema 2.1: *G não tem desenho retangular se uma de suas C_o-componentes tiver um ciclo com menos que 4 pernas.*

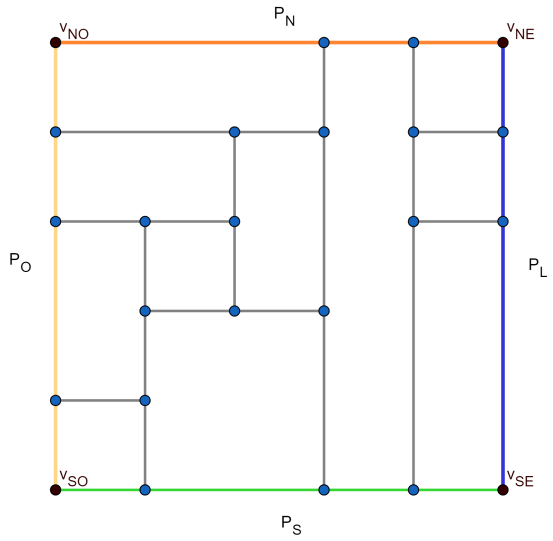


Figura 2.1: *Grafo plano com seus cantos e caminhos externos destacados.*

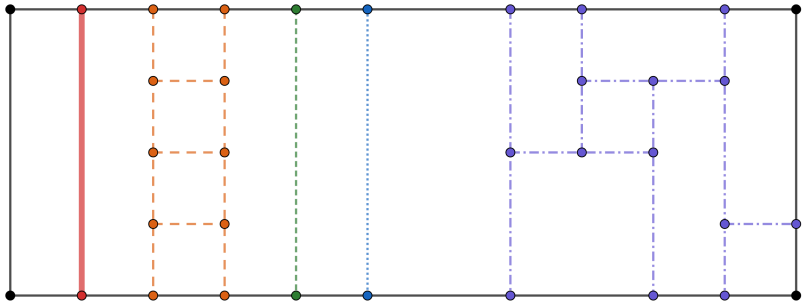


Figura 2.2: *Grafo plano e suas C_o-componentes.*

Demonstração: Seja C um ciclo com menos que 4 pernas em uma C_o-componente de G. Se C tem apenas 3 vértices, não conseguimos desenhar as arestas de C em vertical ou horizontal. Se C tem pelo menos 4 vértices, ao desenhar C, uma das faces externas incidentes a C terá um ângulo interno obtuso. Logo, tal face não seria retangular. □

Verifique a figura 2.3 para uma ilustração do lema 2.1.

Dizemos que um ciclo C está **anexado** a um caminho P se:

(i) P não contém nenhum vértice do interior de $G(C)$, e

(ii) $C \cap P$ é um único subcaminho de P

Seja v_{ini} o vértice inicial de tal subcaminho e v_{fim} o vértice final. Chamamos v_{ini} de **vértice anterior** e v_{fim} de **vértice posterior**. Sendo G um grafo plano, cada subcaminho pelo contorno de uma face admite uma orientação. Seja $Q_H(C)$ o caminho de v_{ini} a v_{fim} passando por C em sentido horário, e seja $Q_{AH}(C)$ o caminho de v_{ini} a v_{fim} passando por

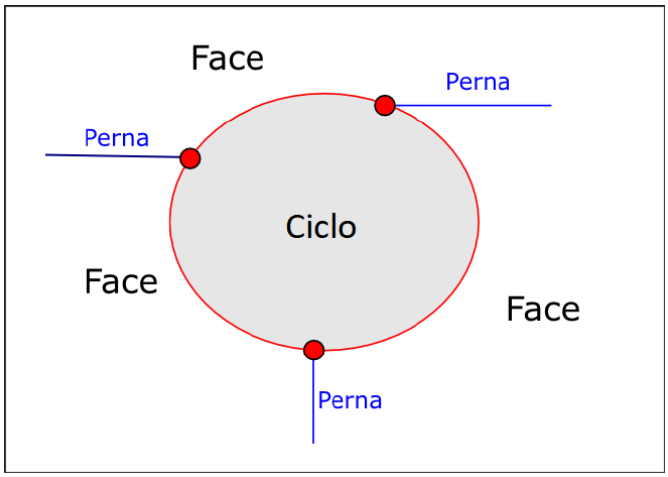


Figura 2.3: *Ilustração do lema 2.1. Retirado de (Assunção, 2012).*

C em sentido anti-horário. Uma perna de C é uma **perna horária** em relação a P se é incidente em um vértice de $V(Q_H(C)) - \{v_{ini}, v_{fim}\}$. Denotamos por $n_H(C)$ o número de pernas horárias de C em relação a P. Similarmente, uma perna de C é uma **perna anti-horária** em relação a P se é incidente em um vértice de $V(Q_{AH}(C)) - \{v_{ini}, v_{fim}\}$. E denotamos por $n_{AH}(C)$ o número de pernas horárias de C em relação a P. Não incluímos referência a P nessa notação, mas ao utilizarmos $Q_H(C)$ ou $n_H(C)$, por exemplo, estará claro em relação a qual P estamos nos referindo.

Um ciclo C anexado a P é dito **ciclo horário** se $Q_{AH}(C)$ é subcaminho de P. Se $Q_H(C)$ é subcaminho de P, dizemos que C é um **ciclo anti-horário**. Um ciclo C é **crítico** se C é ciclo horário e $n_H(C) \leq 1$ ou C é ciclo anti-horário e $n_{AH}(C) \leq 1$.

Na figura 2.4 podemos ver as definições dadas anteriormente.

Lema 2.2: *G não tem desenho retangular se tem um ciclo crítico C anexado a um, e apenas um, dos caminhos P_N, P_L P_S ou P_O.*

Demonstração: Sem perda de generalidade, suponha que C está anexado a P_N. Se C tem apenas 3 vértices, não conseguimos desenhar todas as arestas de C em vertical ou horizontal. Se C tem pelo menos 4 vértices, ao desenhar C, uma das faces externas a C e incidentes a $Q_{AH}(C)$ terá um ângulo interno obtuso. Logo, tal face não seria retangular. □

A figura 2.5 ilustra o lema 2.2.

Lema 2.3: *G não tem desenho retangular se tem um ciclo com $n_{AH}(C) = 0$ anexado a um, e apenas um, dos caminhos $P_N \cdot P_L$, $P_L \cdot P_S$ $P_S \cdot P_O$ ou $P_O \cdot P_N$.*

Demonstração: Sem perda de generalidade, suponha que C está anexado a $P_N \cdot P_L$. Logo, só existe uma face interna de G incidente em C. Qualquer desenho de C impossibilitaria tal face de ser retangular, pois seria necessário ocupar o canto v_{NE}. □

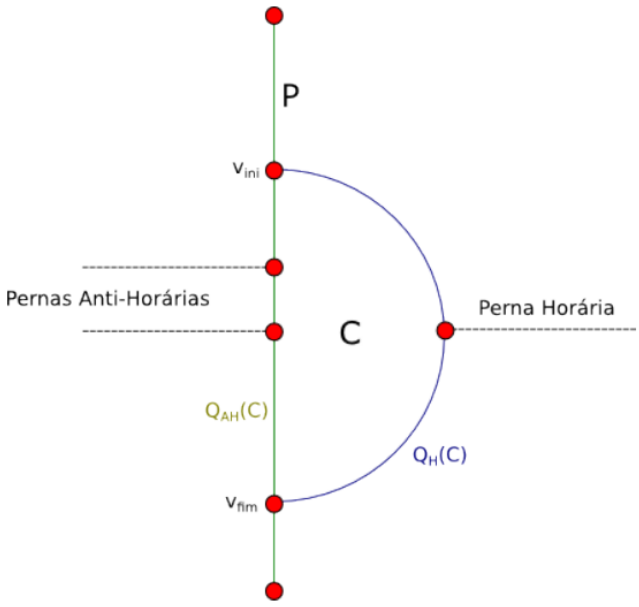
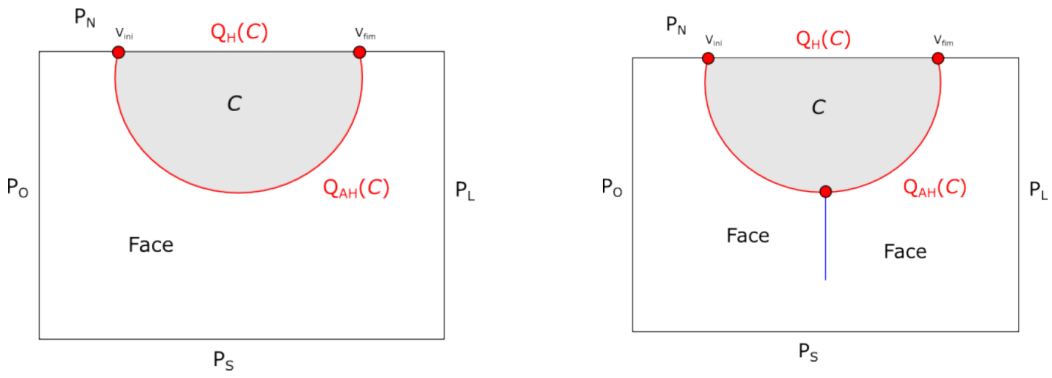


Figura 2.4: *Ciclo horário C anexado ao caminho P. C é crítico, pois $n_H(C) = 1$. Retirado de (Assunção, 2012).*



(a) *Caso em que $n_{AH}(C) = 0$.*

(b) *Caso em que $n_{AH}(C) = 1$.*

Figura 2.5: *Ilustração do lema 2.2. Retirado de (Assunção, 2012).*

Conforme a figura 2.6, o ciclo do lema 2.3 ocupa um canto de G.

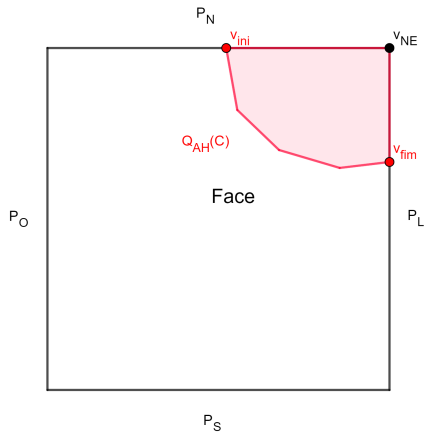


Figura 2.6: *Ilustração do lema 2.3.*

Uma C_o-componente é dita **componente ruim** se satisfaz algum dos lemas 2.1, 2.2 ou 2.3. Para facilitar a identificação, podemos chamar tal C_o-componente de:

1. **componente interna ruim** se satisfaz o lema 2.1
2. **borda ruim** se satisfaz o lema 2.2
3. **canto ruim** se satisfaz o lema 2.3

Tendo esta definição, finalmente chegamos a um dos principais resultados de (M. Rahman, Nakano *et al.*, 1998).

Teorema 2.4: *Seja G um grafo plano tal que todo vértice tem grau 3 exceto os 4 cantos de grau 2 que dividem o ciclo externo C_o em 4 caminhos P_N, P_L, P_S e P_O. Então G tem um desenho retangular se e somente se G não tem componente ruim.*

O teorema 2.15 é uma caracterização de grafos com desenhos retangulares. De acordo com Rahman, Nakano e Nishizeki, este teorema é essencialmente equivalente à **Caracterização de Thomassen**, um teorema clássico sobre desenhos retangulares (Thomassen, 1984).

Teorema 2.5 (Caracterização de Thomassen): *Seja G um grafo plano 2-conexo de modo que 4 vértices do ciclo da face externa $C_o(G)$ possuem grau 2 e todos os outros vértices possuem grau 3.*

Vamos particionar $C_o(G)$ em quatro caminhos: P_O, P_N, P_L e P_S, onde somente os extremos de cada caminho possuem grau 2.

Seja D um grafo retangular isomorfo ao grafo G. O ciclo da face externa $C_o(D)$ (que é um retângulo) é formado pelos caminhos P_i, onde $i \in \{O, N, L, S\}$ ($C_o(D) = P_O \cup P_N \cup P_L \cup P_S$) de modo que todas as arestas de P_N e P_S são horizontais e todas as arestas de P_O e P_L são verticais.

Existe um grafo D com $C_o(D)$ fixo que é desenho retangular de G se e somente se:

(i) Para qualquer vértice $v \in V(G) - V(C_o(G))$ e qualquer conjunto X de no máximo 3 arestas, $G - X$ possui um caminho de v a $C_o(G)$ a menos que X seja o conjunto de arestas incidentes a v

(ii) Cada C_o-componente de G intercepta ao menos 2 caminhos P_i, $i \in \{O, N, L, S\}$

(iii) Para qualquer aresta $e \in E(C_o(G))$, cada C_o-componente de $G - e$ intercepta pelo menos pelo menos 2 caminhos P_i, $i \in \{O, N, L, S\}$

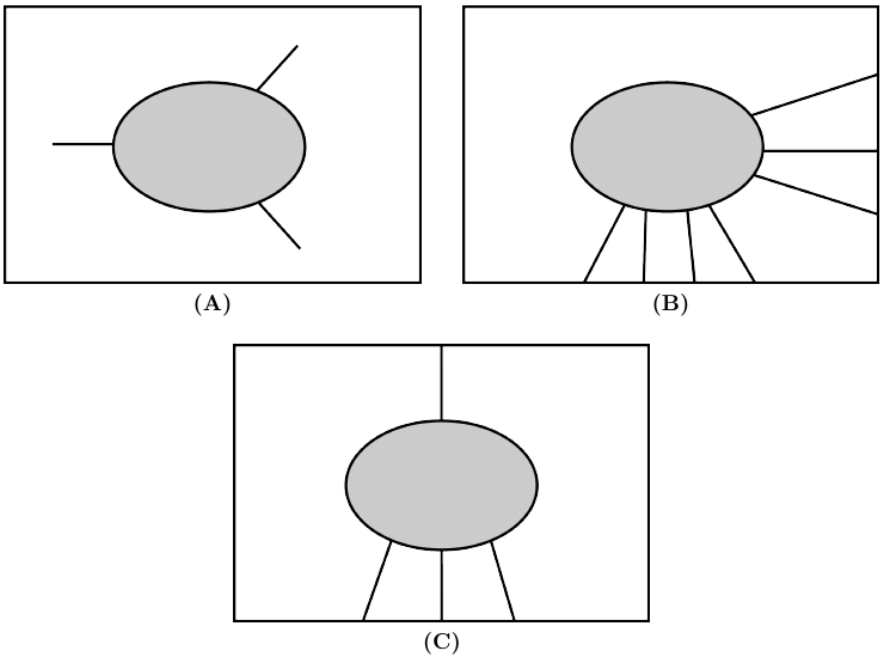


Figura 2.7: *As configurações proibidas, sendo (A) referente a (i), (B) referente a (ii) e (C) referente a (iii). Retirado de (Assunção, 2012)*

Este enunciado do teorema 2.5 foi retirado do capítulo 9 da dissertação de Assunção (Assunção, 2012), onde é possível encontrar a demonstração também.

No início de (M. Rahman, Nakano *et al.*, 1998) consta que uma implementação direta do método baseado na prova deste teorema levaria a um algoritmo de tempo $O(n^3)$, sendo $n = |V|$. E, em paralelo a (Thomassen, 1984), Kozminski e Kinnen desenvolveram um algoritmo $O(n^2)$ para a representação retangular (Kozminski e Kinnen, 1984), a partir do qual alguns trabalhos se sucederam para simplificar e melhorar o tempo da solução para $O(n)$. Na dissertação de Assunção, é possível encontrar os detalhes da implementação de um desses trabalhos, o algoritmo de Bhaskar e Sahni (Bhasker e Sahni, 1988).

Voltando ao teorema 2.5, a prova da necessidade segue imediatamente dos lemas 2.1, 2.2 e 2.3 a partir da definição de componente ruim. Portanto, no resto dessa seção, iremos nos ocupar em provar a suficiência do teorema. Essa prova construtiva resultará num algoritmo $O(n)$ para o problema do desenho retangular.

Antes de seguirmos com a prova da suficiência, vamos mostrar o seguinte resultado.

Com ele, poderemos assumir que há apenas uma C_o-componente em G.

Lema 2.6: *Sejam H_1, H_2, ..., H_p as C_o-componentes de um grafo plano G, e seja $G_i = C_o(G) \cup H_i$, para $1 \leq i \leq p$. Então, G tem um desenho retangular com imersão retangular fixa de $C_o(G)$ se e somente se cada G_i tem um desenho retangular com imersão retangular fixa de $C_o(G_i) = C_o(G)$, $1 \leq i \leq p$.*

O lema 2.6 é bastante intuitivo quando visualizamos as C_o-componentes de um grafo plano como na figura 2.2. No entanto, não vamos prová-lo imediatamente. Primeiramente, definiremos o que é uma **face separadora**. Dado um grafo plano G, uma face f é dita face separadora se f é uma face de G tendo o contorno $C(f)$ tal que:

(i) $C(f) = P'_N \cdot P \cdot P'_S \cdot Q$, ou

(ii) $C(f) = P'_N \cdot P_O \cdot P'_S \cdot P$, ou

(iii) $C(f) = P'_N \cdot P \cdot P'_S \cdot P_L$, ou

(iv) $C(f) = P'_L \cdot P \cdot P'_O \cdot Q$, ou

(v) $C(f) = P'_L \cdot P_N \cdot P'_O \cdot P$, ou

(vi) $C(f) = P'_L \cdot P \cdot P'_O \cdot P_S$

sendo P'_X subcaminho de P_X, com $X \in \{N, L, S, O\}$, e P, Q caminhos internos de G.

Se f satisfaz (a), (b) ou (c), dizemos que é uma **face separadora vertical**. Se f satisfaz (d), (e) ou (f), dizemos que é uma **face separadora horizontal**. Na figura 2.8 podemos ver destacadas as faces separadoras verticais do grafo plano.

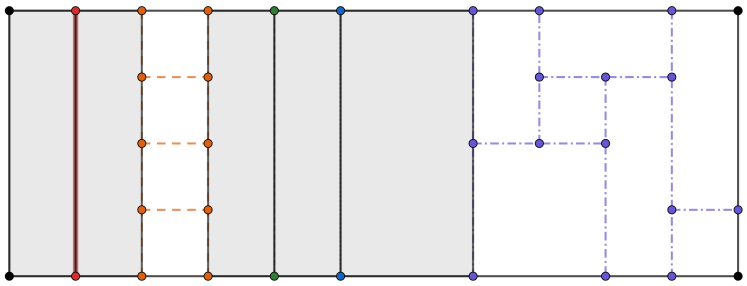


Figura 2.8: *Faces separadoras de um grafo plano.*

Como se pode notar, dadas 2 C_o-componentes distintas de um desenho retangular G, existe uma face separadora entre elas. Os próximos lemas serão para confirmar essa intuição:

Lema 2.7: *Sejam H_1, H_2, ..., H_p as C_o-componentes de um grafo plano G. Se G tem um desenho retangular com imersão retangular fixa de $C_o(G)$, então:*

(i) Se existe uma aresta $e = uv \in E(P_X)$, com $X \in \{N, L, S, O\}$, e $u \in V(H_i)$, $v \in V(H_j)$, $i \neq j$, então e está no contorno de uma face separadora.

(ii) Algum caminho P_X, com $X \in \{N, L, S, O\}$, tem pelo menos um vértice de cada C_o-componente H_i, $1 \leq i \leq p$, ou seja, toda C_o-componente intercepta P_X. Ademais, os vértices em P_X estão ordenados por C_o-componente, isto é, sendo $P_X = (v_1, \ldots, v_k)$, se $v_{k_1} \in V(H_i)$, $v_{k_2} \in V(H_j)$, $i \neq j$, e $k_1 < k_2$, não existe $v_{k_3} \in V(H_i)$ tal que $k_2 < k_3$

Demonstração:

(i) Suponha que exista $e = uv \in E(P_N)$ com $u \in V(H_i)$, $v \in V(H_j)$, $i \neq j$, e que v esteja à esquerda de u, sem perda de generalidade.

Seja f a face interna de G tal que $e \in E(C(f))$. Temos que $C(f) = e \cdot R$, onde R é um caminho de v a u. R não pode ser um caminho interno, pois u e v estão em C_o-componentes distintas e $\Delta(G) \leq 3$. Logo, existem vértices $u', v' \in V(C_o)$, $u' \neq v'$, tais que $R = P \cdot R' \cdot Q$, com P sendo um caminho interno de v a v' passando pelas arestas de H_j e Q um caminho interno de u a u' passando pelas arestas de H_i.

Não podemos ter $u' \in V(P_N)$ com u' mais à direita que u, pois P seria o contorno de uma borda ruim em G. Também não podemos ter $u' \in V(P_L)$, pois P seria o contorno de um canto ruim em G. Similarmente, como G tem desenho retangular, não podemos ter $v' \in V(P_N)$ com v' mais à esquerda de v, nem $v' \in V(P_O)$. Logo, como as C_o-componentes não se interceptam, $u', v' \in V(P_S)$.

Consequentemente, R' é subcaminho de P_S. Se R' não fosse subcaminho de P_S, R' necessariamente passaria pelas arestas de uma C_o-componente H_k distinta de H_i e H_j. Como G tem desenho retangular, pelo item (ii) da Caracterização de Thomassen, H_k precisa interceptar algum P_X, $X \in \{N, L, S, O\}$, além de P_S. O que não é possível, visto que $Q' \cdot e \cdot P'$ separaria H_k de qualquer P_X com $X \in \{O, L\}$ e e é uma aresta.

Portanto, tomando $P'_N = e$ e $P'_S = R'$, concluímos que e está no contorno de uma face separadora (vertical, nesse caso).

(ii) Vamos primeiramente provar que existe um caminho $P_X = (v_1, \ldots, v_k)$, $X \in \{N, L, S, O\}$, com pelo menos um vértice de cada C_o-componente.

Se $p = 1$, o resultado é imediato.

Suponha que $p = 2$. Pelo item (ii) da Caracterização de Thomassen, cada uma das C_o-componentes H_1, H_2 precisam interceptar 2 caminhos de C_o. Vamos supor, pelo contrário, que não existe nenhum P_X contendo vértices das 2 C_o-componentes. Como H_1 e H_2 não podem se interceptar, a única configuração possível é, sem perda de generalidade (considerando as simetrias), H_1 interceptando P_O, P_N e H_2 interceptando P_L, P_S. Numa configuração dessas, temos que os cantos em diagonal v_{NE}, v_{SO} estão no contorno de uma mesma face f. Mas, tal f não admitiria desenho retangular, pois o único desenho retangular possível de f precisaria conter todos os cantos. Portanto, se G tem desenho retangular, algum caminho P_X deve conter vértices de H_1 e H_2.

Se $p \geq 3$, considerando o item (ii) da Caracterização de Thomassen, é necessário que exista pelo menos algum P_X, $X \in \{N, L, S, O\}$, com $u, v \in V(P_X)$ e $u \in V(H_i)$, $v \in V(H_j)$, $i \neq j$. Sem perda de generalidade, considere $X = N$.

Por (i), temos que existe uma face separadora entre H_i e H_j e, como $X = N$, tal face separadora é vertical. Afirmamos que P_N tem pelo menos um vértice de cada C_o-componente. Suponha que alguma C_o-componente H_k não intercepte P_N. Pelo item (ii) da Caracterização de Thomassen, existem P_Y, P_Z, $Y, Z \in \{N, L, S, O\}$, $Y \neq Z$ tal que H_k intercepta P_Y, P_Z. Sabemos que existe uma face separadora entre as C_o-componentes que interceptam P_N. Logo, toda C_o-componente que intercepta P_N também intercepta P_S. Se $Y = S$ ou $Z = S$, aplicando (i) para P_S, teríamos que H_k intercepta P_N, contradição. Então, necessariamente temos que H_k intercepta P_O e P_L, o que é impossível dado que G tem uma face separadora vertical.

Portanto, toda C_o-componente intercepta P_N.

Como entre cada uma das C_o-componentes existe uma face separadora vertical, os vértices em P_N precisam aparecer ordenados da direita para a esquerda por C_o-componente.

□

Lema 2.8: *Sejam H_1, H_2, ..., H_p as C_o-componentes de um grafo plano G, e seja $G_i = C_o(G) \cup H_i$, para $1 \leq i \leq p$. Se cada G_i tem um desenho retangular com imersão retangular fixa de $C_o(G_i) = C_o(G)$, $1 \leq i \leq p$:*

(i) Se existe uma aresta $e = uv \in E(P_X)$, com $X \in \{N, L, S, O\}$, e $u \in V(H_i)$, $v \in V(H_j)$, $i \neq j$, então e está no contorno de uma face separadora.

(ii) Algum caminho P_X, com $X \in \{N, L, S, O\}$, tem pelo menos um vértice de cada C_o-componente H_i, $1 \leq i \leq p$. Ademais, os vértices em P_X estão ordenados por C_o-componente, isto é, sendo $P_X = (v_1, \ldots, v_k)$, se $v_{k_1} \in V(H_i)$, $v_{k_2} \in V(H_j)$, $i \neq j$, e $k_1 < k_2$, não existe $v_{k_3} \in V(H_i)$ tal que $k_2 < k_3$

Demonstração: Análogo ao lema 2.7. □

A demonstração do lema 2.8 é bastante similar ao do lema anterior. Note que, a existência de um desenho retangular de G na prova do lema 2.7 foi utilizada para argumentar sobre a inexistência de bordas ruins e cantos ruins e para aplicar o item (ii) da Caracterização de Thomassen. Como todas as C_o-componentes compartilham do mesmo contorno externo $C_o(G)$, a demonstração pode ser facilmente adaptada para o caso do lema 2.8, utilizando o respectivo G_i em cada caso.

Os resultados anteriores confirmam nossa intuição de que todo desenho retangular tem as C_o-componentes ordenadas da direita para a esquerda (com faces separadoras verticais), ou de cima para baixo (com faces separadoras horizontais). Utilizando tais fatos, voltamos finalmente ao lema 2.6.

Demonstração: (Lema 2.6)

($\Longrightarrow$) Considere um i qualquer, com $1 \leq i \leq p$. Tome a direção de todas as arestas de H_i como está em G para construir um desenho de G_i. Todas as faces de G em que só aparecem arestas de H_i e C_o no contorno também existem em G_i. Logo, podemos desenhar tais faces como retângulos em G_i iguais ao desenho retangular de G. Caso exista uma face

f que H_i compartilha seu contorno $C(f)$ com outra C_o-componente, essa face é separadora vertical. Então, as arestas de H_i nesse contorno estarão na vertical e, portanto, H_i está compreendido entre 2 segmentos de reta verticais de P_N a P_S em G_i. Portanto, as faces entre H_i e C_o que diferem entre G e G_i também são retangulares. Logo, o desenho resultante de G_i é retangular.

($\Longleftarrow$) Análogo ao caso ($\Longrightarrow$), mas precisamos olhar para as faces separadoras entre G_{i-1} e G_i, $2 \leq i \leq p$, para concluir que todas as faces até H_i formam um desenho retangular. □

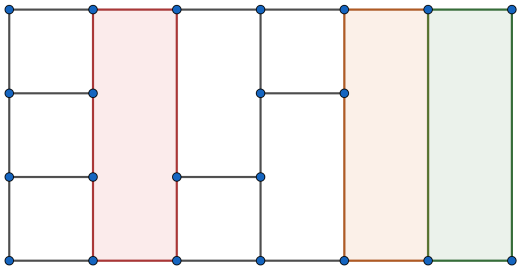
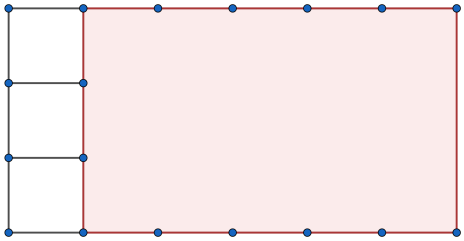
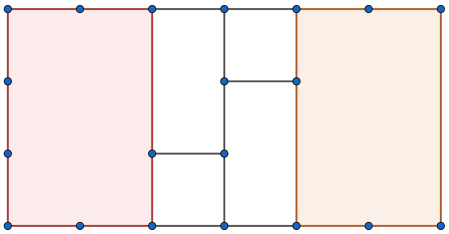


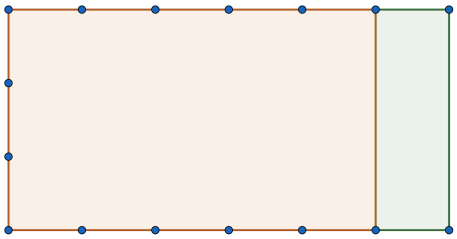
Figura 2.9: *Ilustração de um grafo plano G com 3 C_o-componentes.*



(a) *Grafo plano $G_1 = C_o \cup H_1$.*



(b) *Grafo plano $G_2 = C_o \cup H_2$.*



(c) *Grafo plano $G_3 = C_o \cup H_3$.*

Figura 2.10: *Ilustração de G_1, G_2, G_3 para G do lema 2.9.*

Nos ocuparemos em provar a suficiência do teorema 2.15 daqui em diante. A prova que teremos ao final é construtiva e servirá de base para o algoritmo. Então, *a partir desse ponto vamos assumir que G não tem componente ruim.* Pelo lema 2.6, também podemos assumir que há apenas uma C_o-componente em G.

Seja $P_N = (v_0, \ldots, v_p)$ e $P_S = (u_0, \ldots, u_q)$, considerando C_o em sentido horário. Dizemos que um caminho $P = (p_0, \ldots, p_k)$ é um **NS-caminho** se o vértice inicial p_0 está em P_N,

o vértice final p_k está em P_S e nenhum outro vértice de P está em C_o. Ou seja, P é um caminho interno com $p_0 = v_i$ e $p_k = u_j$, para algum i, j. Um NS-caminho divide G em dois subgrafos G_O^P e G_L^P, em que G_O^P é a parte oeste de G, em relação a P, incluindo P e G_L^P é a parte leste de G incluindo P.

Ao desenharmos P como um segmento de reta, $C_o(G_O^P)$ será um retângulo com:

- caminho norte $P'_N = (v_0, \dots, v_i)$,
- caminho leste $P'_L = P$,
- caminho sul $P'_S = (u_j, \dots, u_q)$ e
- caminho oeste $P'_O = P_O$.

Similarmente, $C_o(G_L^P)$ será um retângulo com:

- caminho norte $P''_N = (v_i, \dots, v_p)$,
- caminho leste $P''_L = P_L$,
- caminho sul $P''_S = (u_0, \dots, u_j)$ e
- caminho oeste $P''_O = P$.

Dizemos que P é um **NS-caminho particionador** se nem G_O^P nem G_L^P tem uma componente ruim.

De modo similar, definimos **SN-caminho**, **OL-caminho** e **LO-caminho**. Também utilizaremos os termos **SN-caminho particionador**, **OL-caminho particionador** e **LO-caminho particionador**.

Se G tem um caminho particionador P, podemos obter o desenho retangular de G recursivamente. Por exemplo, se P é um NS-caminho particionador, podemos particionar G em G_O^P e G_L^P, encontrar o desenho retangular de G_O^P e G_L^P, e então combinar as duas partes para chegar a um desenho retangular de G.

Uma face interna de G é dita **face de borda** se pelo menos uma aresta de seu contorno está em C_o. Um **caminho de borda** é um caminho maximal contido em uma face de borda e que conecta dois vértices de C_o sem passar por nenhuma aresta de C_o. Um caminho de borda é direcionado ao considerarmos o contorno da face de borda em sentido horário. Para $X, Y \in \{N, L, S, O\}$, definimos XY**-caminho de borda** como o caminho de borda começando em um vértice de P_X e terminando em um vértice de P_Y.

Lema 2.9: *Qualquer NS-, SN-, LO- ou OL-caminho de borda P de G é um caminho particionador.*

Demonstração: Sem perda de generalidade, vamos supor que P é um NS-caminho. Logo, G é particionado em G_O^P e G_L^P. Como P é caminho de borda, G_O^P é um ciclo. Logo, G_O^P não tem componente ruim. Sejam P'_X, $X \in \{N, L, S, O\}$, os caminhos de norte, leste, sul e oeste considerando os respectivos valores de X.

Se G_L^P tiver uma componente interna ruim, G também terá. Se G_L^P tiver um canto ruim em contato com P'_L, G também terá. E se G_L^P tiver uma borda ruim em contato com P'_N, P'_L

ou P'_S, G também terá. Em qualquer caso desses temos uma contradição, dado que G tem desenho retangular.

Por outro lado, se G_L^P tiver um canto ruim em contato com $P'_O = P$, como G_O^P é um ciclo, G terá uma borda ruim. E se G_L^P tiver uma borda ruim em contato com $P'_O = P$, G terá uma componente interna ruim. Também temos uma contradição.

Portanto, G_L^P não tem componente ruim. Ou seja, P é um caminho particionador. □

Vamos assumir que G não tem nenhum NS-, SN-, LO- ou OL-caminho de borda. Ou seja, não existe nenhuma face ocupando inteiramente um dos lados de C_o. Nesse caso, para conseguir particionar G mantendo suas partes livre de componentes ruins, vamos construir dois caminhos: P_H e P_{AH}. P_H e P_{AH} são NS-caminhos e podem coincidir inteiramente em alguns casos, assim como podem coincidir em algumas partes. No entanto, P_H e P_{AH} sempre têm um subcaminho inicial e final coincidentes.

Quando P_H e P_{AH} não forem coincidentes, a diferença simétrica $E(P_H) \triangle E(P_{AH}) = E(P_H) \cup E(P_{AH}) - E(P_H) \cap E(P_{AH})$ resulta em ciclos disjuntos. Considere $C_1, \ldots, C_k$, com $k \geq 1$, como sendo tais ciclos. Nesse caso, existem $k + 1$ subcaminhos maximais em $E(P_H) \cap E(P_{AH})$. Ademais, P_H passará por $Q_H(C_i)$ e P_{AH} passará por $Q_{AH}(C_i)$, para $1 \leq i \leq k$. A figura 2.11 ilustra as propriedades de P_H e P_{AH}.

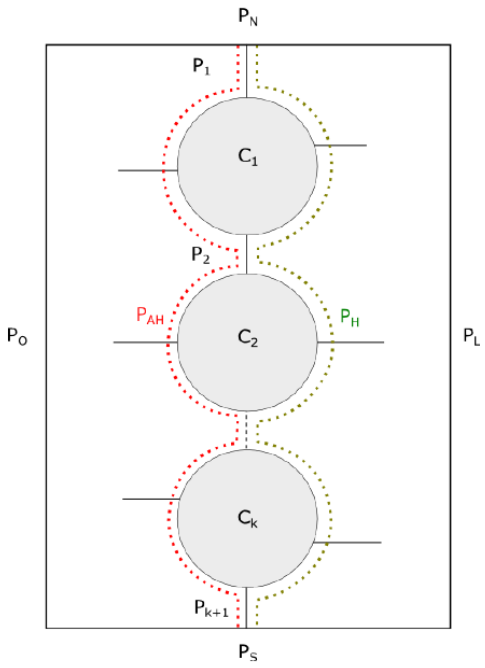


Figura 2.11: *Ilustração de P_H e P_{AH}. Retirado de (Assunção, 2012).*

A construção que faremos de P_H e P_{AH} garantirá que cada C_i tenha exatamente 4 pernas. Em sentido horário, teremos: uma perna em P_i, uma em Q_H, outra em P_{i+1} e finalmente uma em Q_{AH}.

A partição resultante de P_H e P_{AH} será: $G_O^{P_AH}, G_L^{P_H}, G(C_1), \ldots, G(C_k)$.

Lema 2.10: *Suponha que um ciclo C na C_o-componente de G tenha 4 pernas, divi-*

dindo C em 4 caminhos: P'_N, P'_L, P'_S, P'_O. Então, o subgrafo $G(C)$ não tem componente ruim qualquer que seja a imersão retangular de C fixada pelos caminhos P'_N, P'_L, P'_S e P'_O.

Demonstração: Se $G(C)$ tiver uma componente ruim, independente de qual seja, teremos um ciclo com menos de 3 pernas em G. Ou seja, G também terá uma componente ruim, contradição. □

No lema 2.10, não pressupomos como seria a imersão retangular de C_i, $1 \leq i \leq k$. Na verdade, há 2 possibilidades para cada C_i, como indicado na figura 2.12. Ou seja, há um total de 2^k possibilidades para a partição por P_H e P_{AH}. O resultado do lema 2.10 nos garante que podemos escolher arbitrariamente qualquer uma dessas possibilidades e $G(C_i)$ não terá componente ruim.

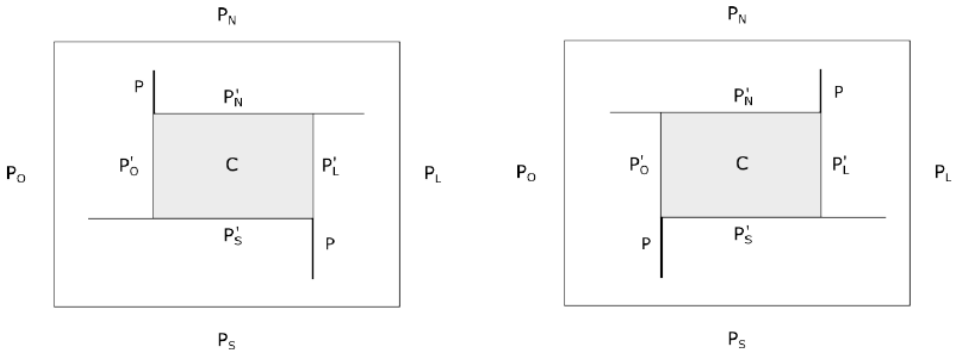


Figura 2.12: *Possíveis imersões de C_i. Retirado de (Assunção, 2012).*

Seja G_1 o grafo obtido de $G_O^{P_{AH}}$ ao contrair todas as arestas de P_{AH} que são horizontais e estão na imersão retangular de cada C_i, $1 \leq i \leq k$. Vamos usar P'_{AH} para nos referir ao caminho obtido após tal operação de contração. P'_{AH} é, então, um segmento de reta vertical e consequentemente $C_o(G_1)$ é retangular. Caso exista um desenho retangular de G_1 podemos transformar tal desenho retangular em um desenho de $G_O^{P_{AH}}$. Tudo isso é possível porque, pela construção de P_{AH}, as únicas arestas que incidem no contorno de cada C_i, $1 \leq i \leq k$, são suas 4 pernas. Ou seja, nenhuma aresta de $G_O^{P_{AH}}$ incide nas partes de P_{AH} que estão na horizontal. E, portanto, $G_O^{P_{AH}}$ tem todas as outras arestas em comum com G_1.

Similarmente, seja G_2 o grafo obtido de $G_L^{P_H}$ ao contrair todas arestas de P_H que são horizontais e estão na imersão retangular de cada C_i, $1 \leq i \leq k$. Tomando P'_H para nos referir ao caminho obtido após tal operação de contração, também temos que G_2 é retangular e seu desenho retangular pode ser utilizado para desenhar $G_L^{P_H}$.

Pelos fatos anteriores, temos então que G_1 tem componente ruim se e somente se $G_O^{P_{AH}}$ também tem. Igualmente, G_2 tem componente ruim se e somente se $G_L^{P_H}$ também tem. Logo, podemos particionar G em $G_O^{P_{AH}}$, $G_L^{P_{AH}}$, $G(C_1), \ldots, G(C_k)$ e transformar $G_O^{P_{AH}}$, $G_L^{P_{AH}}$ em G_1, G_2, respectivamente, e se não houver componente ruim em nenhum G_1, G_2, $G(C_1), \ldots, G(C_k)$, conseguimos um desenho retangular de G.

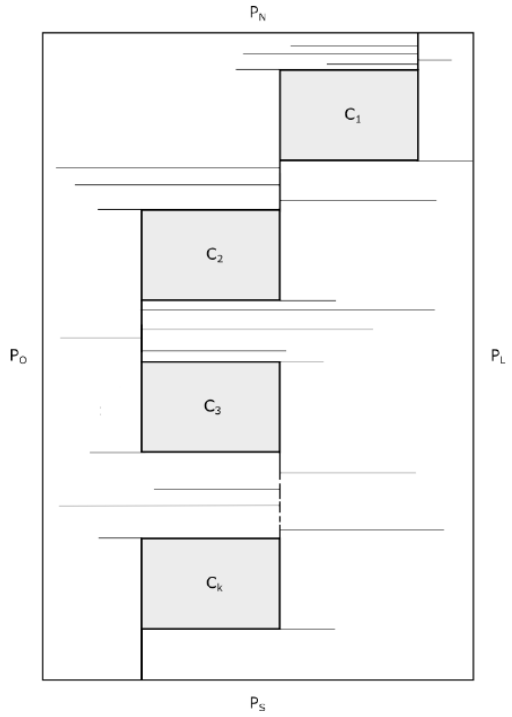


Figura 2.13: *Uma possível partição por P_H e P_{AH}. Retirado de (Assunção, 2012).*

Chamamos P_H e P_{AH} de **par particionador** se nem $G_O^{P_{AH}}$ nem $G_L^{P_H}$ tem componente ruim. Vale lembrar que também pode ocorrer $P_H = P_{AH}$.

A figura 2.14 ilustra desde o particionamento por um par particionador até a construção de G_1 e G_2.

Então, temos o problema de encontrar um par particionador eficientemente. A ideia encontrada em Rahman, Nakano e Nishizeki para tal é construir P_H e P_{AH} a partir do NS-caminho mais ao oeste (mais à esquerda) de G. Um **NS-caminho mais à esquerda** P é um caminho que satisfaz:

(i) P começa em v_1 (o segundo vértice de P_N),

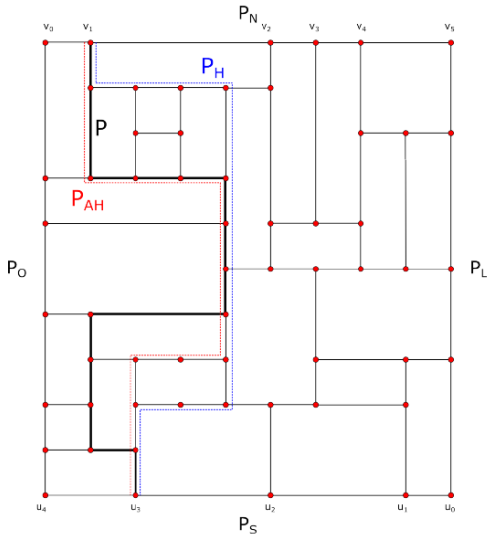
(ii) P termina em u_{q-1} (o penúltimo vértice de P_S), e

(iii) o número de arestas em G_O^P é mínimo.

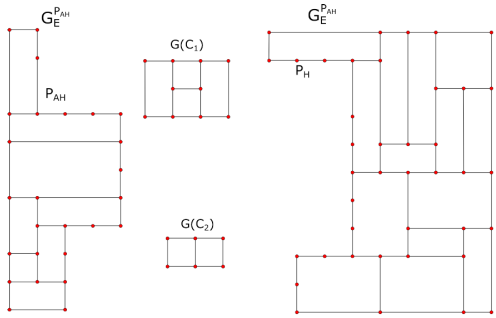
Ou seja, P é o NS-caminho com menos arestas à esquerda.

Um NS-caminho mais à esquerda pode ser construído ao fazer uma busca em profundidade anti-horária a partir de v_1. Uma **busca em profundidade anti-horária** é uma busca em profundidade em que a próxima aresta a ser visitada é escolhida em sentido anti-horário. Uma maneira ilustrativa de ver essa escolha é olhar para as arestas incidentes ao vértice atual v como ponteiros de um relógio analógico centrado em v.

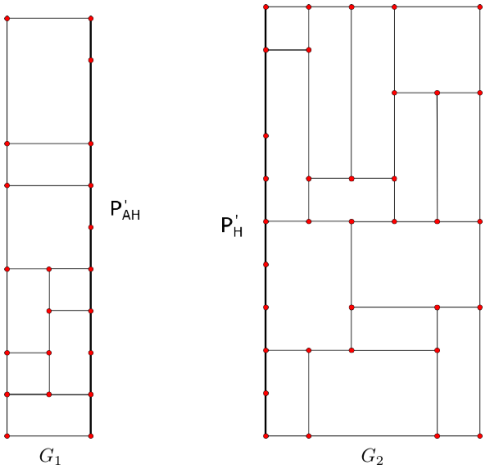
A C_o-componente de G pode ter ciclos anexados ao NS-caminho mais à esquerda P. Pela escolha de P, sabemos que todos os ciclos estarão anexados em sentido horário a P, isto é, o subcaminho de P estará no sentido anti-horário.



(a) *Par particionador para um grafo plano G.*



(b) *Particionamento de* G *em* $G_O^{P_{AH}}$, $G_L^{P_{AH}}$, $G(C_1), G(C_2)$.



(c) *Construção de* G_1 *e* G_2.

Figura 2.14: *Ilustração do particionamento por* P_H, P_{AH} *e construção de* G_1, G_2. *Retirado de (*Assunção, 2012*).*

Lema 2.11: *Sejam C_1, C_2 ciclos críticos anexados a P, sendo que $G(C_1) \not\subseteq G(C_2)$ nem $G(C_2) \not\subseteq G(C_1)$. Então, se $G(C_1) \cap G(C_2) \neq \emptyset$, existe C ciclo crítico anexado a P tal que $G(C_1), G(C_2) \subseteq G(C)$.*

Demonstração: Se C_1 e C_2 são ciclos com $G(C_1) \cap G(C_2) \neq \emptyset$ e nem $G(C_1) \not\subseteq G(C_2)$ nem $G(C_2) \not\subseteq G(C_1)$, pelo menos uma perna de C_1 está em C_2 e, igualmente, pelo menos uma perna de C_2 está em C_1. Como C_1 e C_2 são críticos, essas são as únicas pernas. Tome C como sendo o contorno externo de $G(C_1) \cup G(C_2)$. Considerando o primeiro entre os vérticeis anteriores de C_1 e C_2 e o último entre os vértices posteriores de C_1 e C_2, vemos que C está anexado a P também. Ademais, todas as arestas de C_1 e C_2 estão contidas em $G(C)$, pois $G(C_1), G(C_2) \subseteq G(C)$ por construção. Logo, C não tem nenhuma perna e, portanto, é crítico. □

Um ciclo é **maximal** se não está contido dentro de nenhum outro ciclo. Para a construção do par particionador, iremos nos atentar aos ciclos críticos maximais. Segue o passo a passo da construção de P_H e P_{AH}:

Seja $P = (v_0, \dots, v_k)$ o NS-caminho mais à esquerda em G. Vamos mostrar como construir o par particionador P_H e P_{AH} a partir de P.

Primeiramente, queremos encontrar subcaminhos iniciais e finais compartilhados pelo par particionador. Seja P_{ini} tal subcaminho inicial e P_{fim} o subcaminho final.

Seja i o maior índice tal que $e_a = v_{i-1}v_i \in E(P)$ e e_a pertença ao contorno de um NN-caminho de borda ou LN-caminho de borda. Ou seja, e_a está no contorno de uma face à direita de P que seja incidente em P_N. Considere $Q = (v_i, w_1, w_2, \dots, w_l)$, onde $w_1 = v_{i-1}$, $w_l \in V(P_N)$, e $w_1, w_2, \dots, w_l$ é um subcaminho interno em sentido horário no contorno da face de borda. Definimos $P_{ini} = (w_l, \dots, w_1, v_i)$.

De maneira similar, seja j o menor índice tal que $e_b = v_j v_{j+1} \in E(P)$ e e_b pertença ao contorno de um SS-caminho de borda ou SL-caminho de borda. Ou seja, e_b está no contorno de uma face à direita de P que seja incidente em P_S. Considere $Q' = (x_1, x_2, \dots, x_r, v_j)$, onde $x_r = v_{j+1}$, $x_1 \in V(P_S)$, e $x_1, x_2, \dots, x_r$ é um subcaminho interno em sentido horário no contorno da face de borda. Definimos $P_{fim} = (v_j, x_r, x_{r-1}, \dots, x_1)$.

Note que $i \leq j$, por $v_{i-1}v_i$ e $v_j v_{j+1}$ estarem no contorno das faces de borda incidentes a P_N e P_S, respectivamente.

Até agora temos que $P_{ini}, P_{fim} \subseteq E(P_H) \cap E(P_{AH})$. Considere o subcaminho restante $P' = (v_{i-1}, v_i, \dots, v_j, v_{j+1})$. Caso uma aresta e de P' não esteja contida em nenhum ciclo crítico maximal anexado a P, também $e \subseteq E(P_H) \cap E(P_{AH})$.

Para concluir a construção, precisamos definir o que fazer com as partes em que P' tem um ciclo crítico maximal. Vamos escolher P_H e P_{AH} a depender de cada ciclo crítico maximal C. Lembrando que $n_H(C) \leq 1$, temos os seguintes casos a considerar:

Caso 1: $n_H(C) = 0$

Então, $n_{AH}(C) \geq 2$. P_H e P_{AH} devem passar por $Q_H(C)$.

Caso 2: $n_H(C) = 1$ e $n_{AH}(C) \leq 1$

Nesse caso, $n_{AH}(C) = 1$. P_H deve passar por $Q_H(C)$ enquanto P_{AH} deve passar por $Q_{AH}(C)$. Logo, C é um dos ciclos de $E(P_H) \triangle E(P_{AH})$

Caso 3: $n_H(C) = 1$ e $n_{AH}(C) \geq 1$

Precisamos analisar os seguintes subcasos:

Caso 3.1: $G(C)$ não tem nenhum ciclo crítico anti-horário anexado a $Q_H(C)$

P_H e P_{AH} devem passar por $Q_H(C)$

Caso 3.2: $G(C)$ tem um ciclo crítico anti-horário anexado a $Q_H(C)$

Nesse caso, $G(C)$ não pode ter 2 ciclos críticos anti-horários e disjuntos anexados a $Q_H(C)$. Caso existissem 2, um dos ciclos teria $n_H = 1$, ou seja, apenas 3 pernas em G, sendo assim uma componente interna ruim. Seja C' o único ciclo crítico anti-horário. Temos que necessariamente $n_{AH}(C') = 1$, logo C' tem exatamente 4 pernas. P_H e P_{AH} devem passar por $E(Q_H(C)) - E(Q_H(C'))$. P_H também deve passar por $Q_H(C')$ e P_{AH} por $Q_{AH}(C')$. Assim, C' é um dos ciclos de $E(P_H) \triangle E(P_{AH})$.

Ao escolher dessa forma, vemos que cada ciclo de $E(P_H) \triangle E(P_{AH})$ tem 4 pernas, 2 das quais são compartilhadas entre P_H e P_{AH}.

Observe a figura 2.15 de exemplo da construção de um par particionador. Nela, C_{m3}, C_{m4}, C_{m5} e C_{m6} são os ciclos críticos maximais de P'. C_{m3} representa o Caso 1, C_{m4} o Caso 2, C_{m5} o Caso 3.1 e C_{m6} o Caso 3.2.

Lema 2.12: *Se G não tem nenhum NS-, SN-, LO- ou OL-caminho de borda, G tem um par particionador.*

Demonstração: Considere P_H e P_{AH} construídos a partir do NS-caminho mais à esquerda P, como descrevemos anteriormente. Vamos mostrar que P_H, P_{AH} é um par particionador, ou seja, nem $G_O^{P_{AH}}$ nem $G_L^{P_H}$ tem componente ruim.

Se $G_O^{P_{AH}}$ tiver uma componente interna ruim, é fácil ver que G teria uma componente interna ruim também. Se $G_O^{P_{AH}}$ tiver uma borda ruim incidente em P_N, P_O ou P_S, também é fácil ver que G teria uma borda ruim. Se $G_O^{P_{AH}}$ tiver um canto ruim incidente em P_O, similarmente.

Suponha que $G_O^{P_{AH}}$ tem uma borda ruim incidente em P_{AH}, e seja B o ciclo que satisfaz o lema 2.2. Como P é o NS-caminho mais à esquerda de G, B está entre P e P_{ini}, P e P_{fim} ou está contido em algum ciclo crítico maximal C anexado a P'. Se B está entre P e P_{ini}, como P_{ini} está no contorno de uma face, B implicaria na existência de uma componente interna ruim em G. Analogamente, o mesmo aconteceria ao considerar P_{fim}. Então B está contido em algum ciclo crítico maximal C. Temos 2 casos que poderiam ocorrer:

1. C satisfaz o Caso 1

   Como $n_H(C) = 0$, B implicaria em uma componente interna ruim de G, contradição.

2. C satisfaz o Caso 3.2

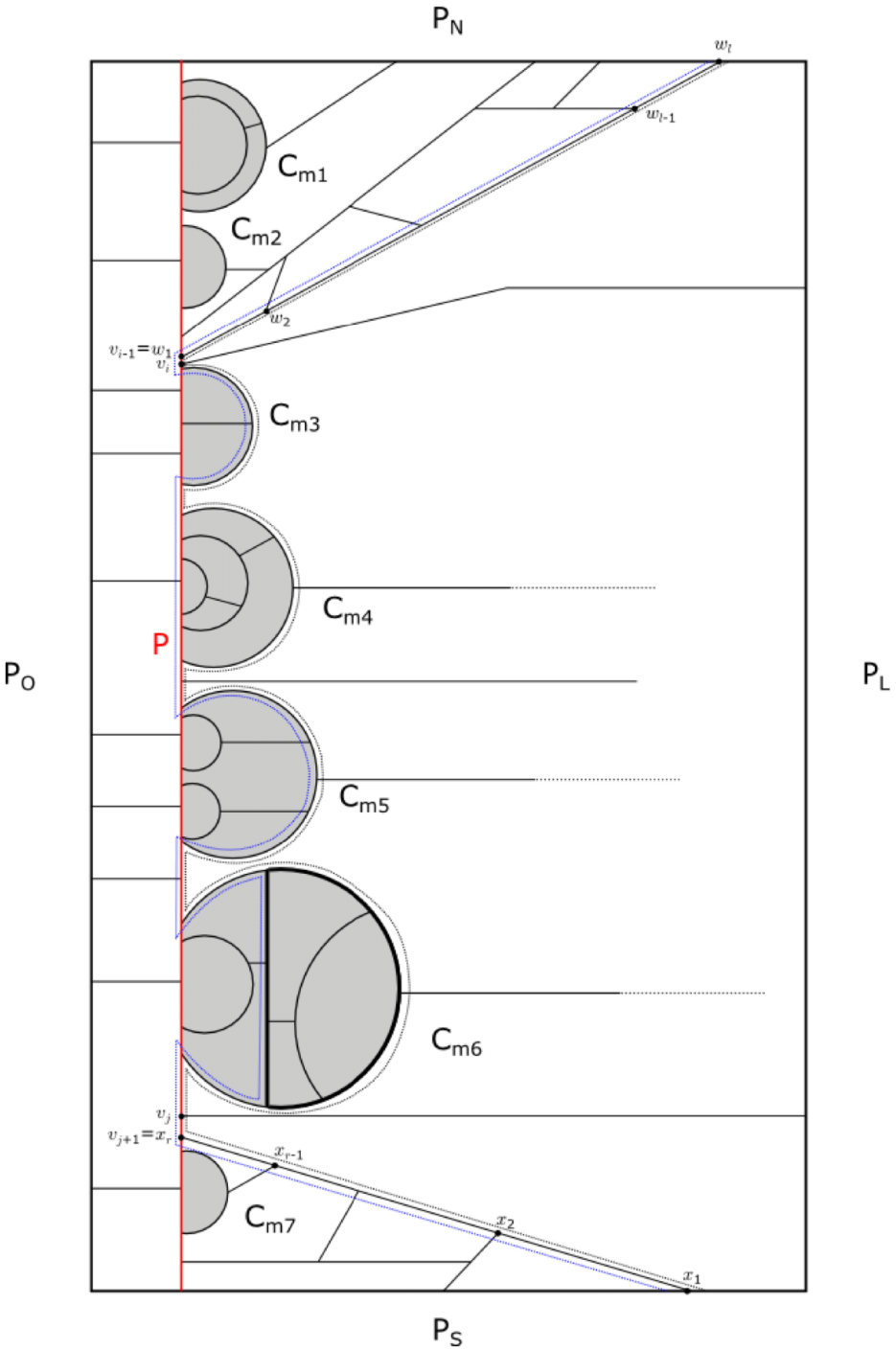


Figura 2.15: *Ilustração da construção do par particionador. Retirado de (Assunção, 2012).*

Considere C' como no Caso 3.2. C' não pode estar contido propriamente em B, pois B seria um ciclo crítico e C' é maximal. Também, B não pode ser disjunto de C', pois teríamos uma componente interna ruim em G. Então B passa pela única perna anti-horária de C'. No entanto, nesse caso existiria um ciclo crítico contendo B e C', contrariando a escolha de C', contradição.

Portanto, $G_O^{P_{AH}}$ não tem borda ruim.

Suponha então que $G_O^{P_{AH}}$ tem um canto ruim entre P_N e P_{AH}. Seja R_O um ciclo que satisfaz o lema 2.3. Como P é o NS-caminho mais à esquerda, R_O está contido na região entre P_N, P e P_{ini}. Como P_{ini} é contorno de uma face, a existência de R_O implicaria em uma borda ruim em G, contradição.

Pela simetricidade na escolha de P_{ini} e P_{fim}, o argumento para a não-existência de um canto ruim entre P_S e P_{AH} é análogo. Logo, $G_O^{P_{AH}}$ também não tem canto ruim.

Portanto, $G_O^{P_{AH}}$ não tem componente ruim.

Analogamente a $G_O^{P_{AH}}$, para $G_L^{P_H}$ precisamos nos atentar somente aos casos de bordas ruins e cantos ruins que poderiam ser incidentes em P_H.

Suponha que $G_L^{P_H}$ tem uma borda ruim incidente em P_H, e seja R o ciclo que satisfaz o lema 2.2. Sejam v_{ini}, v_{fim} os vértices anteriores e posteriores de R. Temos $v_{ini}, v_{fim} \notin V(P_{ini})$ e $v_{ini}, v_{fim} \notin V(P_{fim})$, pela escolha de P_{ini} e P_{fim} pois estão no contorno de uma face de borda incidente em P_N e P_S, respectivamente. Vamos analisar os outros 3 casos:

1. $v_{ini}, v_{fim} \in V(P')$

   Nesse caso, R seria um ciclo crítico maximal anexado a P', o que não pode ocorrer pela construção de P_H, contradição.

2. $v_{ini}, v_{fim} \notin V(P')$

   Então v_{ini} e v_{fim} estão no Q_H de algum ciclo crítico maximal anexado a P'. Necessariamente v_{ini} e v_{fim} estão no contorno de 2 ciclos distintos, digamos C_1 e C_2, já que são críticos. Consequentemente, R passa pela única perna de C_1 e C_2. Logo, existe um ciclo crítico contendo C_1, C_2 e R anexado a P', contrariando a escolha de P_H, contradição.

3. $|\{v_{ini}, v_{fim}\} \cap V(P')| = 1$

   Nesse caso, v_{ini} ou v_{fim} está no contorno de algum ciclo crítico C maximal anexado a P'. Analogamente ao caso anterior, como C é crítico, R passa pela única perna de C. Consequentemente, existe um ciclo crítico contendo C e R anexado a P', contradição.

Portanto, $G_L^{P_H}$ não tem borda ruim.

Suponha então que $G_L^{P_H}$ tem um canto ruim entre P_N e P_{AH}, e seja R_L um ciclo que satisfaz o lema 2.3. Note que R_L contém a face que tem P_{ini} como contorno. Tome v_R como sendo o vértice anterior de R_L. Primeiramente, claramente v_R não pode estar em P_{ini} nem P_{fim}. Ademais, se $v_R \in V(P')$, como R_L, a aresta em P incidente a v_R e que está fora de R_L contraria a escolha de P_{ini}. Logo, v_R precisaria estar no contorno de algum ciclo crítico maximal C anexado a P'. Mas, nesse caso também, a perna de C incidente ao vértice

posterior de C estaria no contorno de uma face de borda incidente em P_N, contrariando a escolha de P_{ini}.

Novamente, pela simetricidade da escolha de P_{ini} e P_{fim}, o argumento para a não-existência de um canto ruim entre P_S e P_H é análogo. Logo, $G_L^{P_H}$ também não tem canto ruim.

Portanto, nem $G_O^{P_{AH}}$ nem $G_L^{P_H}$ tem componente ruim. Finalmente concluímos que P_H, P_{AH} é um par particionador. □

As figuras 2.16, 2.17 e 2.18 podem auxiliar no entendimento da prova do lema 2.12.

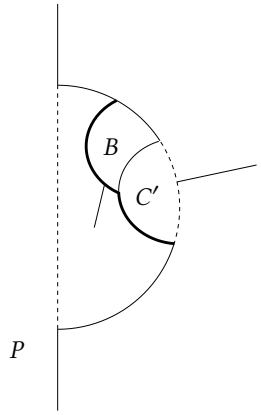
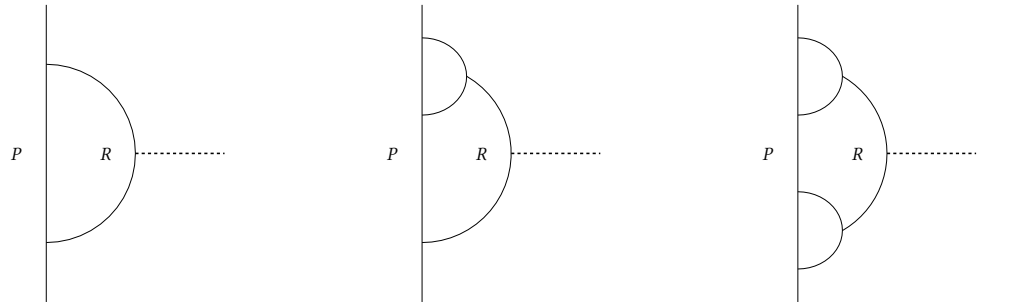


Figura 2.16: *Ilustração do caso 2 de borda ruim em $G_O^{P_{AH}}$ do lema 2.12.*



(a) $v_{ini}, v_{fim} \in V(P')$ **(b)** $v_{ini}, v_{fim} \notin V(P')$ **(c)** $|\{v_{ini}, v_{fim}\} \cap V(P')| = 1$

Figura 2.17: *Ilustração dos possíveis casos de borda ruim em $G_L^{P_H}$ do lema 2.12.*

Finalmente, temos todos os lemas necessários para o teorema 2.15:

Demonstração: (Lema 2.15)

($\Longrightarrow$) Por indução no número de faces, aplicando os lemas 2.9, 2.10 e 2.12, sabemos que G tem desenho retangular se G não tem componente ruim.

($\Longleftarrow$) Pelos lemas 2.1, 2.2 e 2.3, sabemos que se G tem componente ruim, G não tem desenho retangular. Pela contrapositiva temos que G tem desenho retangular somente se G não tem componente ruim. □

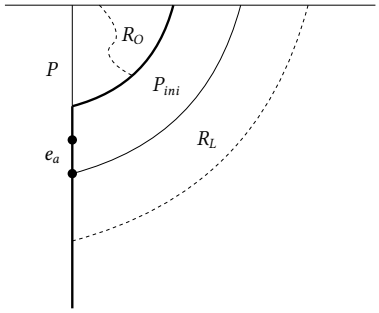


Figura 2.18: *Ilustração dos possíveis casos de canto ruim do lema 2.12.*

Usando os resultados dos lemas 2.9, 2.10 e 2.12 e as construções descritas anteriormente conseguimos recursivamente construir um desenho retangular de G.

O algoritmo descrito em (M. Rahman, Nakano *et al.*, 1998) encontra um desenho retangular de um grafo plano G caso exista. A saída do algoritmo é a direção de cada aresta (horizontal ou vertical). Na seção 2.2 será explicado como podemos obter coordenadas para os vértices uma vez que temos a direção de cada aresta no desenho retangular.

Programa 2.1 Algoritmo do desenho retangular.

```
1   FUNCAO Desenha-Grafo(G)   ▷ Função principal
2       defina a direção das arestas de P_N e P_S como horizontal e de P_O e P_L como vertical
3       encontre todas as C_o−componentes H_1, H_2, ..., H_p
4       para cada C_o−componente faça
5         G_i ⟵ C_o ∪ H_i
6         Desenha(H_i, G_i)
7       fim
8   fim

1   FUNCAO Desenha(H, G)
2       se G tem um NS−caminho de borda então
3         defina a direção das arestas de P como vertical
4         se |E(P)| ≥ 2 então
5             sejam F_1, F_2, ..., F_q as C_o−componentes de G_E^P para a imersão retangular fixa de
                  C_o(G_E^P)
6             para cada F_i faça Desenha-Retangular-Recur(F_i, C_o(G_E^P) ∪ F_i)
7         fim
8       senão se G tem um SN−caminho de borda então
9         defina a direção das arestas de P como vertical
10        se |E(P)| ≥ 2 então
11          sejam F_1, F_2, ..., F_q as C_o−componentes de G_O^P para a imersão retangular fixa de
                C_o(G_O^P)
12          para cada F_i faça Desenha-Retangular-Recur(F_i, C_o(G_O^P) ∪ F_i)
13        fim
14      senão se G tem um LO−caminho de borda então
15        defina a direção das arestas de P como horizontal
16        se |E(P)| ≥ 2 então
```

cont ⟶

$\longrightarrow$ *cont*

sejam $F_1, F_2, \ldots, F_q$ *as* C_o*−componentes* de G_S^P para *a imersão retangular fixa* de $C_o(G_S^P)$
para cada F_i **faça** `Desenha-Retangular-Recur`$(F_i, C_o(G_S^P) \cup F_i)$
fim
senão se G *tem um OL−caminho* de *borda* **então**
defina a direção das arestas de P *como horizontal*
se $|E(P)| \geq 2$ **então**
sejam $F_1, F_2, \ldots, F_q$ *as* C_o*−componentes* de G_N^P para *a imersão retangular fixa* de $C_o(G_N^P)$
para cada F_i **faça** `Desenha-Retangular-Recur`$(F_i, C_o(G_N^P) \cup F_i)$
fim
senão
seja P *o NS−caminho mais à esquerda*
construa P_H e P_{AH} *a partir* de P
se $P_H = P_{AH}$ **então**
defina a direção de *todas as arestas* de P_H *como vertical*
sejam $G_1 = G_O^{P_{AH}}$ e $G_2 = G_L^{P_H}$ *subgrafos* de *modo que tenham* $C_o(G_O^{P_{AH}})$ e $C_o(G_L^{P_H})$ *como imersão retangular fixa*
senão
defina a direção das arestas de P_H e P_{AH} *como uma sequência alternativa* de *vertical* e *horizontal*
seja G_1 *o grafo obtido ao contrair todas as arestas* de P_{AH} *que estão na horizontal das imersões* de $C_1, C_2, \ldots, C_k$
seja G_2 *o grafo obtido ao contrair todas as arestas* de P_H *que estão na horizontal das imersões* de $C_1, C_2, \ldots, C_k$
sejam $G_3 = G(C_1), \ldots, G_{k+2} = G(C_k)$ *os subgrafos com imersão retangular fixa* de $C_1, \ldots, C_k$ *respectivamente*
fim
para cada G_i **faça**
sejam $F_1, F_2, \ldots, F_q$ *as* C_o*−componentes* de G_i
para cada F_j **faça** `Desenha-Retangular-Recur`$(F_j, C_o(G_i) \cup F_j)$
fim
fim
fim

Rahman, Nakano e Nishizeki mostram que o algoritmo de desenho retangular tem complexidade linear utilizando o fato de que cada face é percorrida um número constante de vezes.

Primeiramente, todas as C_o-componentes de G são encontradas e para cada C_o-componente é verificado se existe NS-, SN-, LO- ou OL-caminho de borda. Isso é feito ao visitar todas as faces de borda de G utilizando uma busca em profundidade anti-horária e marcando, para cada caminho de borda de G, rótulos como NN, NE, ..., OO em função da localização (P_N, P_S, P_L ou P_O) dos vértices anteriores e posteriores. Assim, podemos verificar se um caminho de borda é um NS-, SN-, LO- ou OL-caminho de borda em tempo $O(1)$.

Caso não encontremos tal caminho de borda, é necessário que o NS-caminho mais à esquerda P seja construído. Isso é feito utilizando busca em profundidade anti-horária novamente, ao percorrer os caminhos de borda incidentes em P_O. A última aresta de P_{ini} e a primeira aresta de P_{fim} também podem ser encontradas durante a construção de P, visto que já temos os rótulos dos caminhos de borda.

Seja P' o subcaminho de P entre e_a e e_b, como definimos anteriormente. Para encontrar os ciclos críticos maximais vamos percorrer o contorno das faces horárias anexadas a P'. Ao percorrer as faces, as arestas que são percorridas 2 vezes e que não incidem em P são detectadas como pernas. Ademais, como os vértices anteriores e posteriores dos ciclos críticos maximais que queremos encontrar obedecem a regra dos parênteses, utilizamos tal fato para avaliar quais ciclos críticos encontrados são maximais. Encontrados os ciclos críticos maximais basta aplicar os casos que foram descritos anteriormente para construir P_H e P_{AH}. As seguintes arestas são percorridas um número constante de vezes para isso:

(i) arestas de P_H e P_{AH}

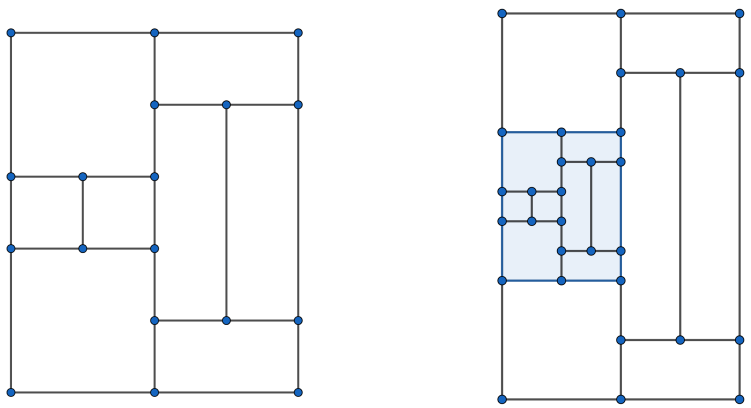
(ii) arestas dos contornos das faces anexadas em sentido horário em P'

(iii) arestas dos caminhos de borda criados ao dividir o grafo P_H e P_{AH}

Após encontrar o par particionador precisamos atualizar os rótulos dos novos caminhos de borda. Isso pode ser feito simplesmente percorrendo os novos caminhos de borda uma vez.

Existe um problema, no entanto, se em um subgrafo H de G escolhermos um subcaminho R de P como NS-caminho mais à esquerda. Ao repetirmos o processo de percorrer os contornos das faces anexadas recursivamente, o tempo resultante pode não ser linear.

A figura 2.19 mostra um exemplo de família de grafos planos que, ao servirem de entrada para o algoritmo, resultaria em complexidade de tempo não-linear.



(a) *Grafo plano G_1.*

(b) *Grafo plano G_2 formado a partir de G_1.*

Figura 2.19: *Família de grafos planos autossimilares que resultam em um processamento ineficiente.*

Para contornar este problema as seguintes informações precisam ser mantidas e repassadas para os subgrafos:

(i) uma lista de todas as arestas que pertencem a um NN- ou LN-caminho de borda

(ii) uma lista de todas as arestas que pertencem a um SS- ou SL-caminho de borda

(iii) um vetor de comprimento $|V|$ que indique para cada vértice v, se v é um vértice anterior ou posterior de um ciclo crítico C anexado a P, e também se $n_{AH}(C) = 1$ ou $n_{AH}(C) > 1$

As listas de (i) e (ii) devem ser utilizadas para encontrar P_{ini} e P_{fim} em algum estágio recursivo. O vetor de (iii) deve ser utilizado para avaliar a existência de caminhos críticos maximais em R.

Assim, durante a execução do algoritmo, cada face torna-se uma face de borda e é percorrida um número constante de vezes nesse processo. Mas, uma vez face de borda, tal face não deixa de ser face de borda para voltar a ser novamente. Logo, o algoritmo é linear (M. Rahman, Nakano *et al.*, 1998).

2.2 Desenho retangular em grade

Na seção 2.1, foi possível desenvolver um algoritmo para o desenho retangular que definia a direção, vertical ou horizontal, de cada aresta de um grafo plano G. Nesta seção, estaremos interessados em definir as coordenadas de cada vértice uma vez que já temos a direção de cada aresta. Dizemos que o desenho é em grade porque iremos atribuir coordenadas inteiras para os vértices. Assim, existe um espaçamento uniforme no desenho, e os vértices ficam localizados em pontos de uma grade. A teoria que vamos apresentar aqui é baseada no mesmo artigo (M. Rahman, Nakano *et al.*, 1998) que mencionamos em 2.1 e trabalha com as mesmas hipóteses.

O método que iremos descrever determina as y-coordenadas de G. O mesmo método pode ser utilizado para determinar as x-coordenadas com poucas modificações, pensando no desenho rotacionado por $\frac{\pi}{2}$ rad. A ideia geral apresentada por Rahman, Nakano e Nishizeki é a seguinte:

Para cada vértice v de $V(G)$, mantemos um inteiro $temp(v)$ como valor temporário da y-coordenada de v. Para cada segmento horizontal maximal L do desenho retangular de G, atribuímos a $y(L)$ uma y-coordenada que deve ser a y-coordenada de todos os vértices em L. Há 2 casos:

(a) existe um vértice u adjacente a v que esteja abaixo de v

Nesse caso, consideramos $temp(v) = y(L') + 1$, onde L' é o segmento horizontal maximal contendo u.

(b) não existe um vértice u adjacente a v que esteja abaixo de v

Então, basta considerarmos $temp(v) = 0$.

Feito isso, basta estabelecer que $y(L) = \max_{v \in V(L)} temp(v)$. Note que $y(P_S) = 0$, pois não há vértices abaixo de P_S.

Diremos que uma aresta uv é ascendente se o vértice v está acima de u em G. Ao utilizar esta expressão, estamos assumindo uma orientação para as arestas de G. Similarmente, também utilizaremos a expressão aresta descendente. Seja T_y o vértice obtido ao remover de G todos as arestas verticais das 3 configurações indicados na figura 2.20. Note que cada aresta indicada como proibida é a aresta mais baixa da borda leste de cada face retangular.

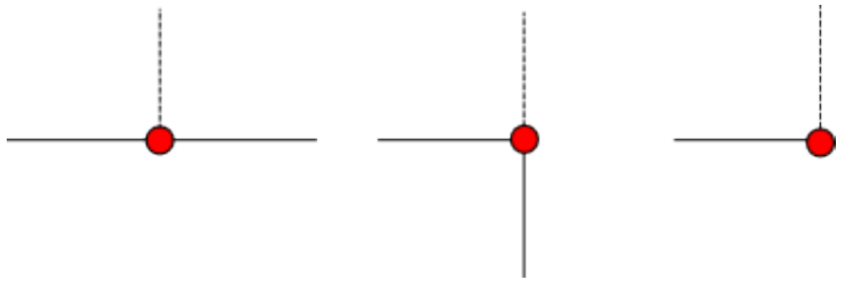


Figura 2.20: *Configurações de arestas ascendentes proibidas. Retirado de (Assunção, 2012).*

Lema 2.13: *T_y é uma árvore geradora de G.*

Demonstração: Primeiramente, vamos mostrar que para todo vértice $v \in V(G)$ existe um caminho entre v e v_{NO} passando somente pelas arestas de T_y.

Seja L_1 o caminho horizontal maximal contendo v_0. Considere v_1' o vértice no extremo esquerdo de L_1. Chamaremos o caminho de v_0 a v_1' de P_1. Como v_1' é extremo esquerdo, não existe nenhuma aresta à esquerda de v_1'. Se não existir aresta ascendente $v_1'w$ em T_y, sabemos que também não há em G, pois todo vértice que é a ponta inferior de uma aresta ascendente proibida tem aresta à esquerda em T_y. Logo, nesse caso já temos que $v_1' = v_{NO}$. Suponha, então, que exista tal aresta ascendente. Seja L_1' o caminho vertical maximal em T_y e considere v_1 o seu extremo superior. Chamaremos o caminho de v_1' a v_1 de P_1'. Se não existir aresta à esquerda de v_1, temos que v_1 não tem aresta ascendente em G também. Nesse caso, já temos que $v_1 = v_{NO}$. Poderíamos supor novamente que $v_1 \neq v_{NO}$ e repetir este processo para obter $v_2', v_2, \ldots$.

Repetindo o processo anterior temos uma sequência de vértices $S = (v_0, v_1', v_1, v_2', \ldots)$ onde cada termo s_{k+1} é estritamente mais "ao noroeste"que o termo anterior s_k, $k \geq 0$. Como G é finito, a sequência é necessariamente finita e o último termo é v_{NO}. Considerando P o caminho resultante das concatenações de P_k, P_k' associado a S, temos um caminho de v_0 a v_{NO}. Se v_0 já é extremo esquerdo de L_1, bastaria considerar $u_1 = v_0$ e desconsiderar P_1 no caminho resultante.

Portanto, como existe um caminho entre cada vértice v e v_{NO} passando por T_y, existe um caminho entre cada par de vértices de G em T_y. Então, para mostrar que T_y é árvore geradora, basta mostrarmos que T_y é acíclico.

Suponha, por contradição, que existe um ciclo C em T_y. Considerando o desenho retangular de G, claramente C não pode ser formado apenas de arestas horizontais. Logo, existe pelo menos uma aresta vertical em C. Considere uma aresta vertical e mais ao sudeste em C. Pela escolha de e, a próxima aresta em C no sentido anti-horário a partir da

ponta inferior de e é uma aresta à esquerda. Então e é uma aresta proibida, contradição.

Logo, T_y é acíclico e, portanto, T_y é uma árvore geradora de G. □

Na figura 2.21 podemos ver um G e sua árvore T_y. As arestas removidas estão tracejadas.

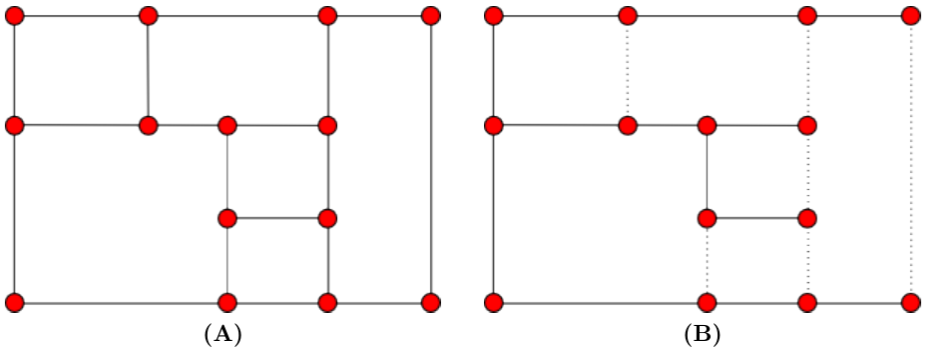


Figura 2.21: *(A) é um desenho retangular G, (B) é a árvore geradora T_y. Retirado de (Assunção, 2012).*

Para computar $y(L)$, poderíamos fazer uma busca em profundidade anti-horária em T_y a partir da aresta descendente do canto v_{NO}, como sugerido por Rahman, Nakano e Nishizeki. Na verdade, T_y também pode ser obtido fazendo uma busca em profundidade anti-horária em G. Para provar tal fato, considere que $T(w)$ é a árvore da busca em profundidade anti-horária em G a partir da da aresta descendente do canto v_{NO} até o vértice w tal que, durante a busca ignoramos todas as arestas ascendentes uv. Em $T(w)$ consideraremos as arestas orientadas pela ordem em que os vértices foram encontrados na busca. Assim, uma aresta ascendente em $T(w)$ indica que percorremos alguma aresta de G de baixo para cima.

Lema 2.14: *Se $T = T(w)$ para algum vértice w de G, então T não tem nenhuma aresta $e = uv$ orientada para a esquerda, isto é, em que v está à esquerda de u.*

Demonstração: Suponha, por contradição, que existe tal aresta $e = uv$. Sem perda de generalidade, suponha também que uv seja a primeira aresta orientada na construção de T em que v está à esquerda de u.

Se e está em P_N, a aresta anterior a e deve ser ser ascendente pois e é a primeira aresta para a esquerda. Mas não há arestas ascendentes em T, contradição.

Então, e é uma aresta da borda inferior de alguma face f de G. Seja v' o primeiro vértice do contorno de f que pertenceu a T. v' só pode estar na borda esquerda de f, pois caso contrário a aresta $u'v'$ de T contraria a escolha de e ou o fato de não existirem arestas ascendentes em T. Também não existe nenhuma aresta e' horizontal incidente a um vértice v'' abaixo de v' na borda esquerda de f, pois e' seria percorrida para a esquerda antes de e, contrariando a escolha de e.

Logo, a partir de v', a busca em profundidade anti-horária percorre a borda esquerda

até o canto inferior esquerdo. Utilizando o fato de que todas as arestas até e em T foram descendentes ou para a direita, a existência de e na borda inferior de f é uma contradição. □

A figura 2.22 ilustra os casos de v' na prova do lema 2.14.

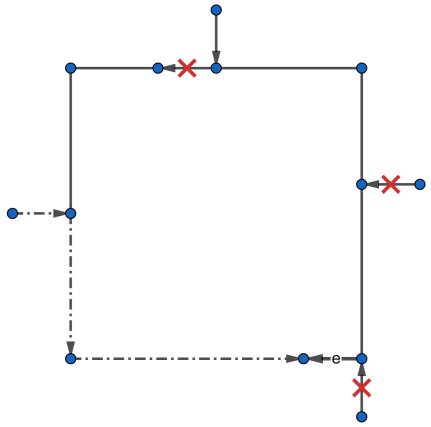


Figura 2.22: *Ilustração dos casos de v' do lema 2.14.*

Pelo lema 2.14, se não percorrermos as arestas ascendentes de G numa busca em profundidade anti-horária iniciada de v_{NO} até o vértice w, garantimos que a última aresta percorrida foi descendente ou para a direita e a próxima aresta não será para a esquerda. Então, sempre que estivermos percorrendo uma aresta uv de G na busca em profundidade anti-horária, é possível identificar se alguma das próximas aresta com origem em v é proibida. Observe pela 2.20 que toda aresta proibida vw é a última aresta vertical na ordem anti-horária a partir de uma aresta orientada para a direita uv.

O algoritmo abaixo, que criamos em cima das ideias apresentadas anteriormente, é baseado em uma busca em profundidade anti-horária no próprio G. Ele pode ser modificado para gerar a árvore T_y também.

Para o funcionamento do algoritmo, podemos manter dois vetores y e $visitado$, ambos de comprimento $|V|$. Em $visitado$, mantemos uma variável *booleana* para cada vértice v indicando se v teria sido visitado caso arestas pudessem ser percorridas de baixo para cima. Assim, sabemos posteriormente que uma aresta vertical de v não deverá ser percorrida pois não está em T_y. A complexidade de tempo desse algoritmo é a complexidade de uma busca em profundidade, ou seja, $O(n)$.

Programa 2.2 Algoritmo do desenho retangular em grade.

```
1  FUNCAO Busca-y(G)  ▷ Função principal
2      para cada vértice v faça
3          y[v] ← 0
4          visitado[v] ← Falso
5      fim
```

cont ⟶

```
    ⟶ cont
6       seja uv a aresta vertical com origem v_NO
7       seja uv' a aresta horizontal com origem v_NO
8       Busca(G, u, v, y, visitado)
9       Busca(G, u, v', y, visitado)
10  fim
11
12  FUNCAO Busca(G, u, v, y, visitado)
13      se a direção de uv for horizontal então
14        y[v] ← max{y[u], y[v]}
15      fim
16      sejam w_1, ..., w_k, com k ≤ 2, os vértices adjacentes a v, distintos de u e em ordem anti-
            horária a partir de uv
17      para cada w_i faça
18        se a direção de uw_i for horizontal então
19          Busca(G, u, w_i, y, visitado)
20        senão se i = 2 ou v = v_SE então   ▷ vw_i é aresta proibida
21          visitado[w_i] ← Verdadeiro
22          y[w_i] ← y[v] + 1
23        senão se não visitado[v] então
24          Busca(G, u, w_i, y, visitado)
25        fim
26      fim
27      se a direção de uw_i for horizontal então
28        y[u] ← y[v]
29      senão
30        y[u] ← max{y[u]+1, y[v]}
31      fim
32  fim
```

Considere as coordenadas de v_{SO} como sendo $(0, 0)$ e as coordenadas de v_{NE} sendo (W, H). Então, a largura do desenho resultante é W e a altura é H. As coordenadas geradas pelo método anterior formam um desenho **compacto**. Um desenho é compacto se para cada i, $0 \leq i \leq W$, existe um segmento vertical de x-coordenada i e, para cada j, $0 \leq j \leq H$, existe um segmento horizontal de y-coordenada j.

Teorema 2.15: *Seja D um desenho retangular em grade de um grafo plano com n vértices. Se D é compacto com largura W e altura H, então $2W + 2H \leq n$ e $WH \leq (\frac{n}{4})^2$.*

Demonstração: Seja l_h o número de segmentos horizontais maximais e seja l_v o número de segmentos verticais maximais. Considere também $l = l_h + l_v$. Cada segmento maximal tem 2 pontas e cada vértice de grau 3 de D é ponta de algum segmento maximal L, sendo L nenhum dos P_N, P_E, P_S ou P_O. Portanto, desconsiderando os 4 cantos e os 4 segmentos maximais de C_o, obtemos a seguinte equação:

$$n - 4 = 2(l - 4) \tag{2.1}$$

Logo,

$$n = 2(l - 2) \tag{2.2}$$

Como D é compacto, temos que

$$H \leq l_h - 1 \tag{2.3}$$

e

$$W \leq l_v - 1 \tag{2.4}$$

Utilizando as equações anteriores, obtemos

$$2W + 2H \leq 2(l_h - 1) + 2(l_v - 1) = 2(l - 2) = n \tag{2.5}$$

Portanto, o perímetro do desenho é limitado por n.

Isolando W na equação acima, temos

$$W \leq \frac{n}{2} - H \tag{2.6}$$

Logo,

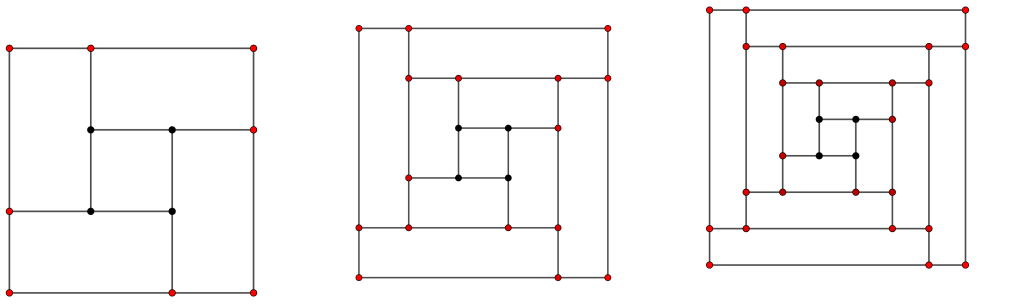
$$WH \leq H(\frac{n}{2} - H) = -H^2 + \frac{n}{2}H \tag{2.7}$$

Como a função quadrática $-H^2 + \frac{n}{2}H$ atinge seu mínimo em $H = \frac{-(n/2)}{2(-1)} = \frac{n}{4}$, obtemos

$$WH \leq -(\frac{n}{4})^2 + \frac{n}{2}\frac{n}{4} = \frac{n^2}{16} = (\frac{n}{4})^2 \tag{2.8}$$

Portanto, a área do desenho é limitada por $(\frac{n}{4})^2$. □

Um desenho compacto é ótimo no sentido de que o perímetro e a área do desenho resultante tem limitante superior justo. Existem infinitos exemplos em que o limitante superior é atingido, como os exemplos da figura 2.23.



(a) *Exemplo 1.*

(b) *Exemplo 2, contendo o Exemplo 1.*

(c) *Exemplo 3, que aparece na capa de (Nishizeki e Md Saidur Rahman, 2004).*

Figura 2.23: *Família de desenhos retangulares em que o limitante é justo.*

2.3 Considerações adicionais

Para concluir este capítulo, vamos esclarecer alguns pontos envolvendo as hipóteses adicionais que utilizamos e trazer algumas referências relacionadas a este trabalho.

Como mencionamos no começo do capítulo, pressupomos que o grafo plano G de entrada é 2-conexo e todos os vértices têm grau 3 exceto os 4 cantos predeterminados. Como todo desenho retangular é 2-conexo, pela contrapositiva, se G não é 2-conexo já sabemos que G não admite desenho retangular. Utilizando o algoritmo clássico de Hopcroft e Tarjan que pode ser encontrado em (Hopcroft e Tarjan, 1973) podemos testar em tempo linear se G é 2-conexo. Logo, esta hipótese não limita nossa solução.

Por outro lado, queremos encontrar soluções para qualquer G com $\Delta(G) \leq 3$, então precisamos lidar com os vértices de grau 2 que não são cantos. Em (M. Rahman, Nakano *et al.*, 2000), Rahman, Nakano e Nishizeki descrevem uma operação chamada **remoção de um vértice de grau 2** que podemos utilizar para resolver este problema. Dado um vértice v em um grafo conexo G, esta operação remove as arestas u_1v e vu_2 e substitui por uma única aresta u_1u_2. Também define-se que um grafo G' é **grafo minimal homomorfo** a G se G' é obtido de G ao repetir a remoção de vértices de graus 2 até que não existam vértices de grau 2 ou o grafo resultante seja isomorfo a K_2.

De modo similar à construção de um grafo minimal homomorfo, dada uma entrada G, podemos criar uma cópia G' na qual iremos repetir a remoção de vértices de grau 2 até que tenhamos apenas os 4 cantos com grau 2. Existe uma função $\varphi : E(G) \longrightarrow E(G')$ mapeando cada aresta e de G para uma aresta e' de G' em que e' substitui e nas remoções de vértices de grau 2 ou $e' = e$ no caso em que e se mantém em G'. Uma vez encontradas as direções de cada aresta de G' pelo algoritmo de desenho retangular, podemos definir a direção de cada aresta e do grafo original como a direção de $\varphi(e)$.

Também teríamos que adaptar o algoritmo de desenho retangular em grade que descrevemos 2.2 para poder definir as coordenadas dos vértices. Há algumas adaptações possíveis para esse algoritmo, a que implementamos é baseada na ideia de considerar o comprimento do caminho entre vértices que são de grau 3 ou cantos. Tais vértices existem tanto em G quanto G'. Podemos atribuir coordenadas para os vértices de G' mas, em vez de considerar que a distância entre dois vértices adjacentes u' e v' é 1, tomamos a distância de u' e v' como sendo $k = |\varphi^{-1}(u'v')|$. Pela construção de G', $\varphi^{-1}(u'v')$ forma um caminho $(u', u_1, u_2, \dots, u_{k-1}, v')$ em G. Uma vez tendo as coordenadas de u' e v', é fácil definir as coordenadas dos vértices $u_1, u_2, \dots, u_{k-1}$ de G. Considerando as y-coordenadas como fizemos na seção 2.2, basta tomar $y(u_j) = (1 - \frac{j}{k})y(u') + \frac{j}{k}y(v')$. Também é possível atualizar as coordenadas de todos os vértices de G em uma única busca em profundidade utilizando esta ideia, como fizemos na implementação.

No resto desta seção, vamos falar sobre outras referências que podem servir de base para trabalhos futuros. Primeiramente, o algoritmo de desenho retangular que estudamos é utilizado em alguns trabalhos de Rahman, Nakano e Nishizeki, por exemplo (M. Rahman, Nakano *et al.*, 2000) e (Md. Saidur Rahman *et al.*, 1999).

Em (M. Rahman, Nakano *et al.*, 2000), a ideia é encontrar um desenho retangular onde cada vértice do grafo plano de entrada G pode ser representado por uma caixa ou um

ponto. Tal desenho é chamado **desenho caixa-retangular** de G. Uma caixa no desenho retangular resultante D é uma face retangular, ou seja, D não é isomorfo a G, mas existe um mapeamento de vértices de G para vértices e faces de D. As arestas dos vértices de G que são representados por caixas em D são as pernas das caixas. Na figura 2.24, podemos ver um desenho caixa retangular em que as caixas estão destacadas em azul. Observe que não há limitação no grau dos vértices de um desenho caixa-retangular.

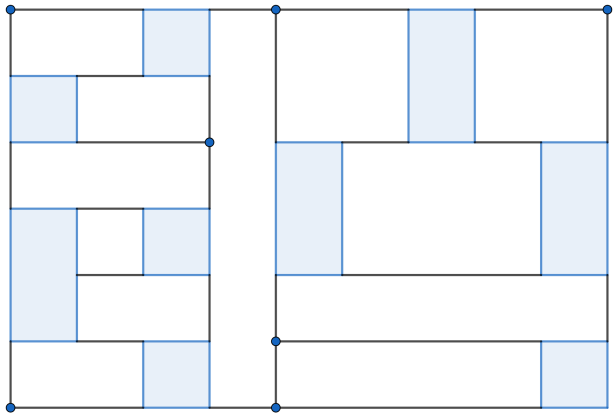
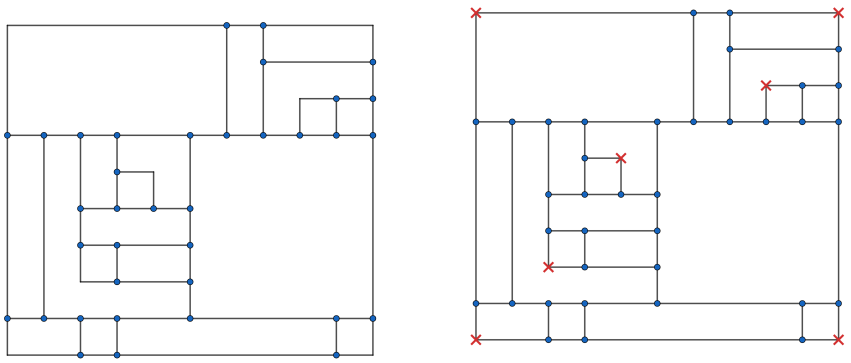


Figura 2.24: *Um desenho caixa-retangular.*

Já em (Md. Saidur Rahman *et al.*, 1999), queremos encontrar um **desenho ortogonal** de G. Num desenho ortogonal, cada aresta é representada por uma sequência alternada de segmentos de reta verticais e horizontais. Chamamos de dobra essa alternância de direção entre os segmentos que pode existir no desenho de uma aresta. Rahman, Nakano e Nishizeki desenvolvem um algoritmo que dado um grafo plano 3-conexo G em que todos os vértices têm grau 3, é encontrado um desenho ortogonal D com o mínimo de dobras. A figura 2.25 mostra o desenho ortogonal de um grafo plano G com o mínimo de dobras. As dobras aparecem destacadas com x na segunda ilustração.



(a) *Um desenho ortogonal.*

(b) *O mesmo desenho ortogonal com as dobras destacadas com x.*

Figura 2.25: *Exemplo de um desenho ortogonal com o mínimo de dobras.*

Finalmente, não poderíamos terminar este capítulo sem falar de (M. Rahman, Nakano *et al.*, 2002) e (M. Rahman, Nishizeki *et al.*, 2004) que mencionamos no capítulo introdutório

1. Ambos os trabalhos servem de continuação ao que apresentamos aqui e generalizam a solução que desenvolvemos para o desenho retangular.

Em (M. Rahman, Nakano *et al.*, 2002), Rahman, Nakano e Nishizeki desenvolvem um algoritmo linear para a escolha dos 4 cantos em um grafo plano G com contorno externo $C_o(G)$ fixo. Se G tem desenho retangular, utilizando tal algoritmo podemos encontrar os 4 cantos corretamente para então desenhar com o algoritmo que apresentamos. Ou seja, temos um algoritmo linear que encontra o desenho retangular de grafos planos em geral. Já em (M. Rahman, Nishizeki *et al.*, 2004), Rahman, Nishizeki e Ghosh desenvolvem um algoritmo linear que encontra um desenho retangular para grafos planares. Tal algoritmo utiliza o algoritmo de desenho retangular para grafos planos que mencionamos.

A título de curiosidade, enunciaremos o teorema principal de (M. Rahman, Nakano *et al.*, 2002) a seguir.

Teorema 2.16: *Seja G um grafo plano com $\delta(G) = 2$, $\Delta(G) \leq 3$ e suponha que todos os vértices de grau 2 estão em $C_o(G)$. Então, G tem um desenho retangular se e somente se satisfaz as condições a seguir:*

(i) *G não tem um ciclo com exatamente 1 perna*

(ii) *Todo ciclo de G com exatamente 2 pernas contém pelo menos 2 vértices de grau 2*

(iii) *Todo ciclo de G com exatamente 3 pernas contém pelo menos 1 vértice de grau 2*

(iv) *se S é um conjunto de ciclos disjuntos em G, tendo c_2 ciclos com exatamente 2 pernas e c_3 ciclos com exatamente 3 pernas, então $2c_2 + c_3 \leq 4$*

A prova da necessidade do teorema 2.16 é mais fácil que a suficiência, assim como no teorema 2.5, e a prova da suficiência do teorema 2.16 irá resultar num algoritmo para a escolha dos 4 cantos. O trabalho desenvolvido em (M. Rahman, Nakano *et al.*, 2002) também segue uma estrutura bem parecida com o de (M. Rahman, Nakano *et al.*, 1998) e envolve conceitos próximos ao que estudamos neste capítulo. Por esta razão, (M. Rahman, Nakano *et al.*, 2002) pode ser uma referência ideal para a continuação dos estudos de desenhos retangulares.

Capítulo 3

Implementação

Todos os códigos desenvolvidos para este trabalho podem ser encontrados em https://linux.ime.usp.br/~andrevsn/mac0499/implementacao.tar. Os módulos contidos foram desenvolvidos em `Python 3.9.2`, sendo que dentre eles os de interesse para este capítulo são:

- `PlaneGraph.py`

  Contém a implementação de DCEL (*doubly connected edge list*, ou listas de arestas duplamente ligadas), estrutura de dados que utilizamos para representar grafos planos.

- `RectangularDrawing.py`

  Contém a implementação do algoritmo de desenho retangular, sendo `DrawGraph` a função principal.

- `RectangularGridDrawing.py`

  Contém a implementação dos algoritmos para definição das coordenadas do desenho retangular, sendo `HorizontalSpanningTree` e `VerticalSpanningTree` as funções principais.

Para cada função principal referida anteriormente também criamos uma função que gera uma animação em `.gif` mostrando o funcionamento do algoritmo. Tais funções estão contidas no mesmo módulo que suas versões que não geram animação e são, respectivamente: `AnimatedDrawGraph`, `AnimatedHorizontalSpanningTree` e `AnimatedVerticalSpanningTree`.

Além dos módulos anteriores, foram desenvolvidos os seguintes módulos:

- `Plotters.py`

  Contém funções que foram utilizadas para poder visualizar o grafo plano e as informações contidas nos seus vértices, arestas e faces. Também contém funções que foram utilizadas na geração da animação. As bibliotecas utilizadas foram `matplotlib 3.5.1` e `imageio 2.14.1`.

- `Generators.py`

Contém funções para geração de alguns grafos planos. Foram utilizados para teste do módulo `PlaneGraph.py` inicialmente.

- `Positioners.py`

  Contém algumas heurísticas que foram testadas inicialmente na tentativa de atribuir coordenadas para grafos planos.

- `Utilities.py`

  Contém funções genéricas utilizadas por outros módulos, como `Positioners.py` e `Generators.py`. Também contém as funções `intoargs` e `matrixtoargs` que utilizamos para facilitar a criação de entradas para teste.

No mesmo diretório dos módulos `.py` podem ser encontrados notebooks (arquivos `.ipynb`) contendo exemplos de uso e teste das funções desenvolvidas. Durante o desenvolvimento, a versão do servidor `Jupyter` utilizado foi `6.2.0` com `IPython 7.20.0`.

Há bastantes detalhes a serem considerados na implementação do algoritmo de desenho retangular em relação aos pseudocódigos que apresentamos no capítulo 2. Podemos elencar os seguintes pontos como mais importantes para entender a implementação:

1. Como o grafo plano foi representado computacionalmente, ou seja, como funciona a estrutura de dados utilizada?
2. Como as operações de união e contração de arestas foram adaptadas para tal estrutura de dados?
3. O que é feito quando não há desenho retangular?
4. Como foi lidado com vértices de grau 2?
5. Como a memoização foi implementada para garantir linearidade de tempo?

A primeira questão será esclarecida na seção 3.1. Já as questões 2, 3 e 4 serão abordadas na seção 3.2. Finalmente, na seção 3.3 iremos explicar a memoização.

3.1 Estrutura de dados

Como mencionado anteriormente, utilizamos uma DCEL como estrutura de dados para representar grafos planos e sua implementação se encontra em `PlaneGraph.py`. A referência principal para o desenvolvimento desse módulo foi a seção 2.2 do livro de Berg, Cheong, Kreveld e Overmars (Berg *et al.*, 2000). A explicação que daremos aqui é de uma versão simplificada considerando que os grafos planos que temos são conexos.

Um objeto `PlaneGraph` `G` representando um grafo plano mantém 3 listas:

- `G.vertices`: lista de vértices
- `G.edges`: lista de "arestas"
- `G.faces`: lista de faces

Cada vértice é representado por um objeto `Vertex` e cada face é representada por um objeto `Face`. Como cada aresta de um grafo plano é incidente a 2 faces, numa DCEL convenientemente representamos cada aresta por duas semi-arestas cada uma incidente a apenas uma face. Assim, o que está de fato contido na lista de arestas são duplas de objetos `HalfEdge`.

Se existe uma aresta $e = uv$ no grafo plano que queremos representar, teremos 2 objetos `HalfEdge` cada um representando uma das semi-arestas $\vec{e} = (u, v)$ e $\overleftarrow{e} = (v, u)$. Dizemos que as semi-arestas $\vec{e}$ e $\overleftarrow{e}$ são **semi-arestas gêmeas**. E que $\vec{e}$ é **gêmea de** $\overleftarrow{e}$ e vice-versa. Como as semi-arestas assumem uma orientação, também podemos dizer que u é **origem** de $\vec{e}$ e v é **destino** de $\vec{e}$.

Por padrão, consideramos que as semi-arestas estão em sentido anti-horário no contorno das faces internas. Logo, se $e \in C(f)$ onde f é uma face interna e, ao percorrer f em sentido horário, passamos pelos vértices u e v nessa ordem, então $\vec{e}$ é incidente à face f. Consequentemente, se e tem faces distintas em cada lado, temos que $\overleftarrow{e}$ não é incidente a f.

Um objeto `Vertex` `v` armazena por padrão os seguintes atributos:

- `v.index`: índice de `v` em `G.vertices`
- `v.name`: nome (opcional), utilizado para identificar o vértice em visualizações
- `v.incident_edge`: referência a um `HalfEdge` representando uma das semi-arestas que tem `v` como origem
- `v.degree`: o grau de `v`
- `v.coords`: uma dupla de números representando as coordenadas de `v` no plano, também opcional

Adicionalmente, as coordenadas podem ser acessadas como `v.x` e `v.y`. Apesar das coordenadas serem opcionais, para a visualização utilizando as funções de `Plotter.py` é necessário que todos os vértices de `G` tenham coordenadas.

Já um objeto `HalfEdge` `e` armazena por padrão os seguintes atributos:

- `e.index`: dupla de índices para identificar `e` em `G.edges`
- `e.name`: nome (opcional), utilizado para identificar a semi-aresta em visualizações
- `e.origin`: referência a um `Vertex` representando o vértice de origem de `e`
- `e.twin`: referência a um `HalfEdge` representando a semi-arestas gêmea de `e`
- `e.incident_face`: referência a um `Face` representando a face a qual `e` é incidente
- `e.next_edge`: referência a um `HalfEdge` representando a próxima semi-aresta no contorno da face incidente
- `e.prev_edge`: referência a um `HalfEdge` representando a semi-aresta anterior no contorno da face incidente

O destino de `e` pode ser acessado como `e.destination`. E é importante ressaltar que a definição de próxima semi-aresta e aresta anterior que utilizamos considera o contorno das faces internas em sentido anti-horário. É uma consequência desse fato que o contorno da face externa estará em sentido horário.

Em alguns casos, queremos manter informações sincronizadas entre as semi-arestas gêmeas. Para `e.direction` e `e.thickness` utilizados nos algoritmos que desenvolvemos implementamos essa sincronização. Ou seja, uma vez que `e.direction` é atualizado, `e.twin.direction` também é atualizado automaticamente.

Finalmente, um objeto `Face f` armazena por padrão os seguintes atributos:

- `f.index`: índice de `f` em `G.faces`
- `f.name`: nome (opcional), utilizado para identificar a face em visualizações
- `f.inner_component`: referência a um `HalfEdge` representando uma semi-aresta no contorno de `f`, quando `f` é a face externa
- `f.outer_component`: referência a um `HalfEdge` representando uma semi-aresta no contorno de `f`, quando `f` é uma face interna

Na implementação que fizemos, a face externa necessariamente deverá ter índice 0. No entanto, também é possível verificar se uma face é externa sem verificar o índice, utilizando `f.isOuterFace`.

É necessário passar algumas listas de tuplas para o construtor de `PlaneGraph` para que se possa representar um grafo plano G. Usando a implementação que temos, precisaríamos informar pelo menos as pontas de todas as arestas e o contorno de todas as faces. Como precisávamos gerar alguns grafos específicos para testar os algoritmos, as seguintes possibilidades foram desenvolvidas para facilitar a criação de entradas:

A. Leitura de arquivos `.in`
B. Leitura de arquivos `.mtx`
C. Métodos `subdivide` e `join`

Todas essas possibilidades podem ser entendidas pelas documentações que temos nos códigos, assim como o uso do próprio construtor de `PlaneGraph`. Uma especificação mais detalhada dos arquivos `.in` e `.mtx`, por exemplo, pode ser encontrada na documentação das funções `intoargs` e `matrixtoargs`, respectivamente. Nosso objetivo no restante desta seção será apresentar brevemente como podemos utilizar cada uma delas.

A seguir, mostramos o conteúdo do `exemploA.in` criado em `exemplo_implementacao.ipynb`. O arquivo tem 21 linhas, sendo que as linhas 1, 9 e 18 indicam o número de vértices, arestas e faces, respectivamente. As informações da cada vértice, aresta e face ocupam 1 linha.

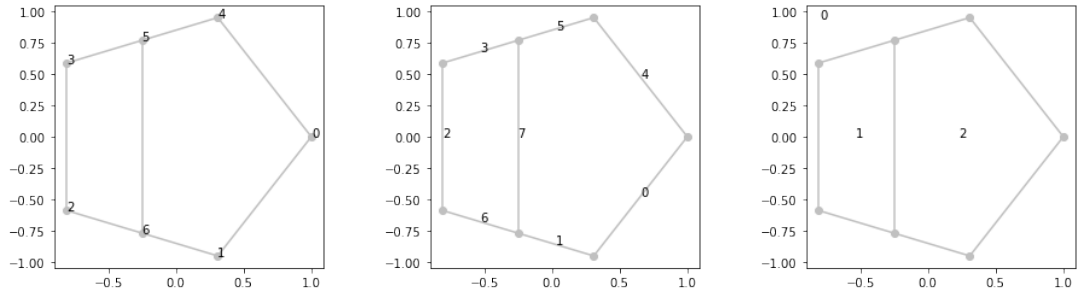
Nas linhas 2-8 temos as coordenadas dos vértices de índice 0 a 6. Os nomes `A` e `B` estão atribuídos aos vértices de índice 4 e 5, respectivamente.

Nas linhas 10-17 podemos ver as arestas de índice 0 a 7. Para cada aresta, são descritos os índice dos vértices que formam suas pontas. A aresta de índice 5 recebe o nome `e`.

Finalmente, nas linhas 19-21 estão descritas as faces de índice 0 a 2 pelos seus contornos. O caractere + na linha 19 esclarece que para percorrer o contorno da face no sentido anti-horário, a aresta de índice 0 deve ser percorrida no sentido dado, ou seja, da ponta de índice 0 a 1. Caso o caractere fosse −, estaria indicando que a aresta deve ser percorrida no sentido contrário ao dado na linha 10. A face de índice 2, apresentada na linha 21, é nomeada como `X`.

```
 1   7
 2   1.0000 0.0000
 3   0.3090 -0.9511
 4   -0.8090 -0.5878
 5   -0.8090 0.5878
 6   A 0.3090 0.9511
 7   B -0.2500 0.7694
 8   -0.2500 -0.7694
 9   8
10   0 1
11   1 6
12   2 3
13   3 5
14   4 0
15   e 5 4
16   6 2
17   6 5
18   3
19   + 0 1 6 2 3 5 4
20   + 7 3 2 6
21   X - 7 1 0 4 5
```

A figura 3.1 mostra o grafo plano representado pelo `exemploA.in` desenhado com as funções de `Plotters.py`. Na figura 3.1 é possível ver os elementos com os nomes atribuídos.

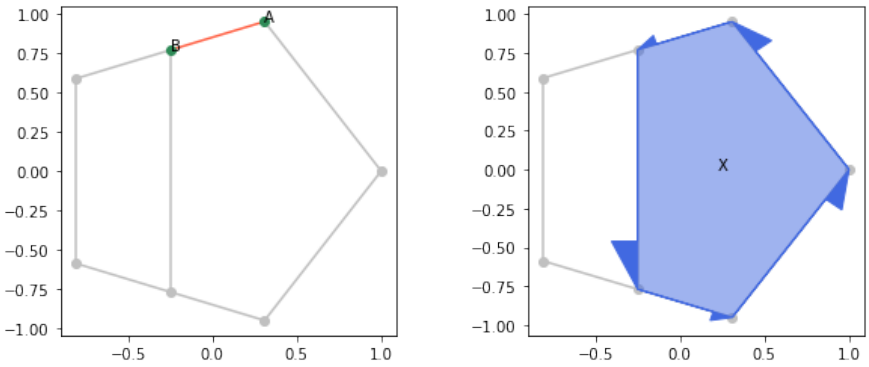


(a) *Desenho com o índice dos vértices.* **(b)** *Desenho com o índice das arestas.* **(c)** *Desenho com o índice de cada face.*

Figura 3.1: *Desenho do grafo plano representado no arquivo* `exemploA.in`.

Para exemplificar o uso de arquivos `.mtx`, vamos utilizar o `exemploB.mtx`. Na primeira linha do arquivo mostrado a seguir, temos o número de linhas e colunas do desenho em `ASCII` representando um desenho retangular. E, nas próximas linhas, temos o desenho em si onde os vértices são representados por `+`, e as arestas por `-` e `|`.

A figura 3.3 mostra o grafo plano de `exemploB.mtx`. Apesar da facilidade de visualiza-



(a) *Desenho destacando os vértices e arestas com nomes.* **(b)** *Desenho destacando a única face com nome.*

Figura 3.2: *Desenho do grafo plano de* `exemploA.in` *destacando o nome dos elementos.*

ção dos arquivos em `.mtx`, devido à necessidade de leitura de todos os 5×7 caracteres e à restrição de que devemos conhecer o desenho retangular de antemão, o uso de `.in` pode ser mais vantajoso.

```
1   5 7
2   +-+-+-+
3   | |   |
4   +-+-+-+
5   |   | |
6   +---+-+
```

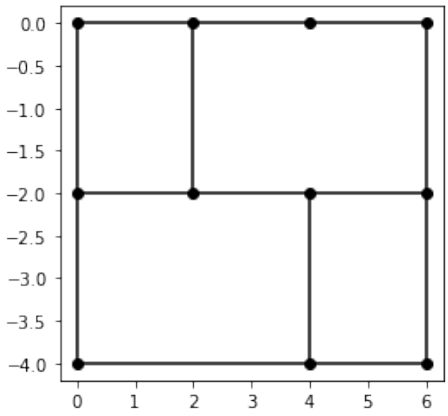


Figura 3.3: *Desenho do grafo representado pelo* `exemploB.mtx`.

A última possibilidade que mencionamos é a utilização dos métodos `subdivide` e `join` da classe `PlaneGraph`. Nesse caso, é necessário partir de um um `PlaneGraph G` já existente.

Dado um `HalfEdge e`, `G.subdivide(e)` cria um novo vértice que subdivide a aresta de `e` em duas. E, dados `Vertex v1` e `v2` não adjacentes e um `Face f` contendo `v1` e `v2` em seu contorno, `G.join(v1, v2, f)` cria uma aresta entre `v1` e `v2` passando por `f` e forma uma nova face que contém a semi-aresta (v_1, v_2).

O `exemploA.in` foi criado a partir de um ciclo de 5 vértices utilizando tais operações. De fato, qualquer grafo 2-conexo poderia ser criado a partir de um ciclo utilizando repetidamente as operações de `subdivide` e `join` descritas anteriormente. O único cuidado necessário seria em relação à escolha de `v1` e `v2`, pois o custo de um `join` como implementamos é linear do número de vértices do contorno da nova face formada.

3.2 Algoritmos

Tanto o `HorizontalSpanningTree` quanto o `VerticalSpanningTree` são adaptações do pseudocódigo 2.2 para lidar com vértices de grau 2. As adaptações feitas são simples, e a ideia geral é percorrer caminhos em que todos os vértices tenham grau 2 exceto as pontas, que podem ser vértices de grau 3 ou cantos. Veja o resultado da atribuição de y-coordenadas pelo `HorizontalSpanningTree` na figura 3.4.

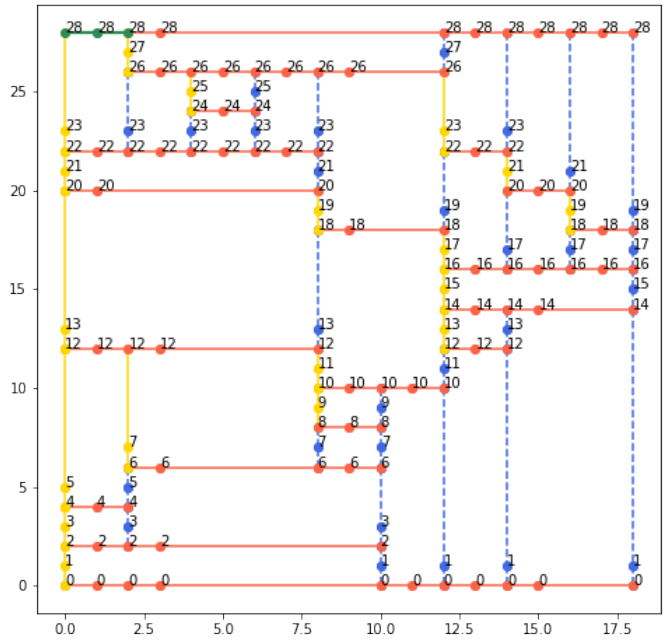


Figura 3.4: *Resultado de `VerticalSpanningTree` para um grafo plano contendo vértices de grau 2. As linhas tracejadas representam as arestas proibidas.*

Vale ressaltar que o teorema 2.15 não se aplica a grafos planos contendo vértices internos de grau 2. No entanto, como se pode observar pela figura 3.4, as y-coordenadas são escolhidas de modo a tentar minimizar o tamanho do desenho na nossa implementação também.

Nesta seção, iremos nos ocupar majoritariamente com o algoritmo do desenho retangular. Diferentemente do que mencionamos anteriormente sobre os algoritmos de `RectangularGridDrawing`, a implementação contida em `RectangularDrawing` difere

em alguns pontos do pseudocódigo 2.1. Observe a nossa implementação de `Desenha-Grafo` (`DrawGraph`) e `Desenha` (`Draw`) no programa 3.1.

Programa 3.1 Funções `DrawGraph` e `Draw`.

```
1   def DrawGraph(G, corners):
2      for e in G.edges:
3         e[0].direction = None
4         e[0].label, e[1].label = None, None
5         e[0].is_on_p, e[1].is_on_p = False, False
6      for f in G.faces: f.label = None
7
8      prepareOuterFace(G, corners)
9      success, certificate = Draw(G, corners)
10     if not success: return success, certificate
11
12     return True, [e[0].direction for e in G.edges]
```

```
1   def Draw(G, corners, update_side=None, memo=(None, None, None)):
2      updateOuterEdges(G, corners, update_side)
3      if isSquare(G, corners, update_side): return True, None
4
5      success, certificate, separators = BoundariesUpdate(G, corners,
           update_side)
6      if not success: return success, certificate
7
8      if len(separators) > 0:
9         success, certificate, H_attrs = separateComponents(G, corners,
              separators, memo)
10        if not success: return success, certificate
11        for corners, update_side, memo in H_attrs:
12           success, certificate = Draw(G, corners, update_side, memo)
13           if not success: return success, certificate
14        return True, None
15     else:
16        success, certificate, H_attrs = Partition(G, corners, memo)
17        if not success: return success, certificate
18        for corners, update_side, memo in H_attrs:
19           success, certificate = Draw(G, corners, update_side, memo)
20           if not success: return success, certificate
21        return True, None
```

Para não sacrificar a legibilidade do texto, a partir de agora iremos dizer que `v` é um vértice se `v` é um objeto `Vertex`, `e` é uma semi-aresta se `e` é um objeto `HalfEdge`, etc. O mesmo será feito em relação aos objetos nativos do `Python`, por exemplo, diremos que `l` é uma lista se `l` é um objeto `list`. Já nos casos em que nos referirmos a uma aresta, estamos nos referindo a um par de semi-arestas gêmeas.

A função principal a ser chamada para o desenho do algoritmo retangular é `DrawGraph`, que pode ser visto no programa 3.1. `DrawGraph` recebe dois argumentos: um grafo plano 2-conexo `G` e uma 4-tupla de vértices `corners` contendo os 4 cantos predeterminados em ordem NO, NL, SL e SO.

Na função `DrawGraph`, alguns atributos associados a cada semi-aresta de `G` são iniciali-

zados. São eles:

- `e.direction`: direção da aresta associada à semi-aresta `e`. Admite os valores `"horizontal"`, `"vertical"` e `None`.
- `e.label`: rótulo indicando à qual borda P_N, P_L, P_O ou P_S a semi-aresta `e` pertence. Admite os valores `"N"`, `"E"`, `"S"`, `"W"` e `None`.
- `e.is_on_p`: rótulo indicando se a semi-aresta `e` pertence ao NS-caminho mais à esquerda P. Na implementação que temos, P é representado por uma lista de semi-arestas apontando para S. Admite os valores *booleanos* `True` e `False`.

Quando `e.direction` é atualizado para um valor diferente de `None`, `e.label` também é e a semi-aresta gêmea `e.twin` também tem direção e rótulo atualizados. Uma vez que `e` tem uma direção associada, consideramos que `e` é uma (semi-)aresta externa. Caso contrário, isto é, se `e.direction` é `None`, `e` é considerada interna.

Também em `DrawGraph`, cada face `f` tem seu rótulo `f.label` inicializado. Esse atributo é utilizado para armazenar o rótulo dos caminhos de borda. Se um caminho de borda tem rótulo NN, por exemplo, e `f` é a face de borda associada a tal caminho, temos que o valor de `f.label` é `"NN"`. Os possíveis valores são `None` e `"XY"` com `X`, `Y` podendo ser `N`, `E`, `S` ou `W`.

A função `prepareOuterFace` irá preparar `G` para servir de argumento à função `Draw`. A função `Draw` requer que sejam passados os mesmos `G` e `corners`, mas `G` deve satisfazer a seguinte premissa: existe um ciclo C de semi-arestas externas, orientadas em sentido anti-horário, e os vértices de `corners` separam tal C em 4 subcaminhos, P_N, P_L, P_O e P_S. Para isso, em `prepareOuterFace` percorremos a face externa de `G` e atribuímos rótulo e direção a cada aresta.

A premissa que acabamos de enunciar é essencial para o funcionamento de `Draw`. Nas chamadas recursivas de `Draw`, `G` se mantém o mesmo, mas `corners` muda e existirá um ciclo C como mencionamos para o novo `corners`.

Uma vez que `G` está preparado para ser utilizado, chamamos a função recursiva `Draw`. `Draw` devolve `success` e `certificate`:

- `success`: indica se foi possível obter um desenho retangular de `G` com sucesso. Admite os valores *booleanos* `True` e `False`.
- `certificate`: vale `None` se `success` é `True`. Senão, não existe desenho retangular de `G`. Logo, existe um ciclo de uma componente interna ruim, uma borda ruim ou um canto ruim. `certificate` será uma lista contendo as semi-arestas de tal ciclo.

Toda aparição de variáveis com nome `success` e `certificate` seguirá essa definição. Uma vez que existe desenho retangular de `G`, ou seja, `success` é `True`, `DrawGraph` devolve `success` e um certificado de que há desenho retangular: uma lista contendo a direção atribuída a cada aresta. Se não há desenho retangular, `DrawGraph` devolve `success` e o certificado de que não há desenho retangular, `certificate`.

A função `Draw` do programa 3.1 tem um papel similar à função `Desenha` do programa 2.1. No entanto, os argumentos diferem de `Desenha`. Isso se deve à premissa que mencionamos

anteriormente. Existe uma propriedade a mais que não mencionamos: para cada vez que `Draw` é chamado, existe um único C que satisfaz as premissas que mencionamos anteriormente e seja tal que todas as arestas de $G(C)$ que não estão em $C = C_o(G(C))$ são internas – de acordo com a definição de aresta externa baseada em `e.direction`. A função `Draw` encontra um desenho retangular de $H = G(C)$.

Para conseguir desenhar `Draw` dessa forma, não realizamos a operação de união entre grafos planos que aparece no programa 2.1. Se implementássemos exatamente como é feito no pseudocódigo, teríamos um $C = C_o$ compartilhado entre diferentes subgrafos e não teríamos a separação total entre cada $H = G(C)$, que são as partes dos grafos que tentamos desenhar em cada chamada de `Draw`.

As operações de separação em C_o-componentes e partição utilizando NS-, SN-, LO- ou OL-caminho de borda também foram adaptadas e são feitas de uma só vez utilizando a ideia de faces separadoras.

A figura 3.5 exemplifica o processo feito por `Draw` à medida que a profundidade da recursão aumenta. É importante notar a vantagem que o uso de DCEL traz para tornar esse processo mais intuitivo: uma vez que separamos os subgrafos de `G`, podemos atualizar os atributos da semi-aresta que ficou de um lado sem nos preocuparmos com o outro lado.

A partir de agora, iremos nos referir ao subgrafo que `Draw` está desenhando como `H`. Além de `G` e `corners`, `Draw` pode receber 2 argumentos: `update_side` e `memo`. Vamos explicar a função de `update_side` primeiramente.

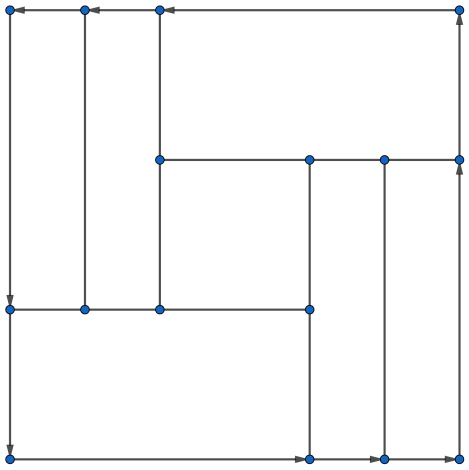
`update_side` admite uma variedade de valores `"NS"`, `"S"`, `"NSEW"`, `"EW"`, `"N"`, etc. O valor de `update_side` indica quais bordas dentre P_N, P_L, P_S e P_O precisa ser atualizada. Quando `update_side` é `None`, a interpretação que temos é a mesma de `"NSEW"`, ou seja, toda borda precisa ser atualizada. A atualização da borda envolve:

1. A atualização de `e.outer_prev` e `e.outer_next` para cada semi-aresta `e` contida na borda
2. A atualização do rótulo das faces de borda
3. A detecção de cantos ruins ou bordas ruins em `H`

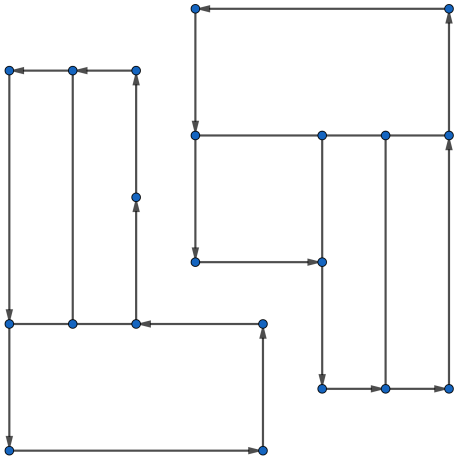
Se `H` é subgrafo próprio de `G`, percorrer o ciclo externo ou as bordas de `H` não é tão simples. Diferentemente de `G` que tem uma face externa da qual podemos facilmente percorrer o contorno, o contorno de `H` precisaria ser construído. Para resolver esse problema, nós mantemos em cada semi-aresta externa `e` uma referência à semi-aresta anterior `e.outer_prev` e posterior `e.outer_prev` no contorno externo de `H`. Assim como na face externa de `G`, a ordem é considerada em sentido horário.

Essa atualização mencionada no item 1 é feita por `updateOuterEdges`. Assim como nos itens 2 e 3, basta percorrer as faces das bordas indicadas por `update_side` um número constante de vezes para fazer essa atualização.

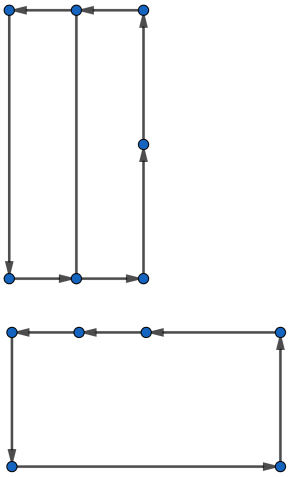
Vale ressaltar que é realmente necessário utilizar `update_side` para que apenas as faces de borda necessárias sejam percorridas a cada chamada de `Draw`. A figura 3.6 mostra



(a) *Primeiramente* `Draw` *recebe todo o* `G`.



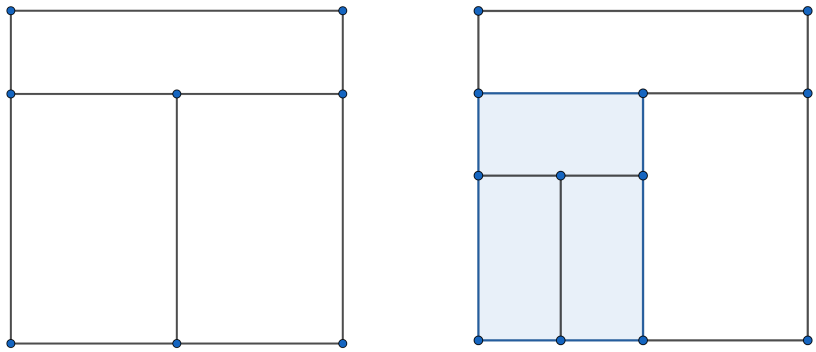
(b) *Como não há face separadora,* `Draw` *particiona* `G` *utilizando* P_H *e* P_{AH}. *O subgrafo mais à esquerda* `H` *aparece destacado.*



(c) `H` *tem um OL-caminho de borda, que é identificado como uma face separadora horizontal. A face separadora horizontal aparece destacada do restante de* `H`.

Figura 3.5: *Desenho do processo de divisão que é feito recursivamente por* `Draw`.

uma família de grafos em que, se percorrermos todas as faces de borda a cada chamada de `Draw` percorreríamos uma das faces $\Omega(n)$ vezes.



(a) *Grafo plano* G_1. **(b)** *Grafo plano* G_2 *formado a partir de* G_1.

Figura 3.6: *Família de grafos planos autossimilares que tem a face do canto inferior direito percorrida* $\Omega(n)$ *vezes se* `update_side` *não é utilizado.*

O último argumento de `Draw` é uma tripla `memo`. Na nossa implementação, sempre utilizamos os nomes de variáveis `a_list`, `b_list` e `p` para cada entrada de `memo`, isto é, `memo = (a_list, b_list, p)`. A função de `memo` é passar para os níveis recursivos mais profundos essas 3 variáveis:

(i) `a_list`: uma lista de todas as semi-arestas de `p` que pertencem a um NN- ou LN-caminho de borda de H_0, onde H_0 é o menor supergrafo de `H` em que ocorreu uma partição por P_H e P_{AH}. As semi-arestas aparecerão na mesma ordem que em `p`.

(ii) `b_list`: uma lista de todas as semi-arestas de `p` que pertencem a um SS- ou SL-caminho de borda de H_0, onde H_0 é o menor supergrafo de `H` em que ocorreu uma partição por P_H e P_{AH}. As semi-arestas aparecerão em ordem contrária a de `p`.

(iii) `p`: o NS-caminho mais à esquerda P de algum supergrafo de `H`. É uma lista contendo as semi-arestas de P em sentido sul.

Ou seja, são as variáveis de memoização que precisamos para garantir a linearidade do algoritmo. Note que `p` não é atualizado desde a sua construção, enquanto `a_list` e `b_list` são à medida que partições por P_H e P_{AH} são feitas recursivamente. Os detalhes a respeito da atualização dessas variáveis podem ser vistos na seção 3.3.

Voltando para o corpo de `Draw`, temos a função `isSquare`. `isSquare` simplesmente verifica se `H` é uma face. Isso é feito percorrendo uma face de borda e vendo se todas as semi-arestas do contorno são externas. Nesse caso, não há nada a ser feito em `Draw`.

Então, em `BoundariesUpdate` são feitas as atualizações de bordas mencionadas nos itens 2 e 3.

O programa 3.2 mostra o corpo de `BoundariesUpdate`. Na função `BoundaryUpdate` olhamos cada face de borda como uma concatenação de caminhos internos maximais. Se `H` não tem desenho retangular ou tem faces separadoras, podem existir faces com vários caminhos internos maximais, ou seja, não existiria um único caminho de borda. É possível

Programa 3.2 Função `BoundariesUpdate`.

```
1   def BoundariesUpdate(G, corners, update_side=None):
2      if update_side is None or update_side == "NESW": update_side = [None]
3      for side in update_side[::-1]:
4         success, certificate, separators = BoundaryUpdate(G, corners, side)
5         if not success: return success, certificate, None
6      if update_side[0] == "W" or update_side[0] == "S":
7         separators = list(reversed(separators))
8
9      if update_side[0] == None: update_side = "NESW"
10     for side in update_side[::-1]:
11        success, certificate = checkBoundaryCycles(G, corners, side)
12        if not success: return success, certificate, None
13
14     return True, None, separators
```

tanto verificar se existem cantos ruins quanto verificar se uma face é separadora olhando para os rótulos que daríamos a cada caminho interno desses, pensando cada um como caminho de borda.

Uma vez analisados e atualizados os rótulos das faces de borda, com a função `checkBoundaryCycles` podemos verificar se há ciclos críticos anexados à cada borda de `H`. Um ciclo crítico forma uma borda ruim em `H`, logo teríamos que `H` não admite desenho retangular. A ideia que utilizamos para verificar os ciclos críticos nesse caso é parecida com a que é utilizada para encontrar os ciclos críticos do NS-caminho mais à esquerda P. Como, no entanto, precisamos encontrar apenas um ciclo crítico caso exista, fazemos o seguinte:

- Atribuímos uma ordem à cada face anexada à borda de interesse, considerando a ordem que elas aparecem ao percorrermos tal borda;
- Ao percorrermos todas as faces, se encontramos uma aresta que é compartilhada por duas faces e tais faces não são consecutivas na ordenação, tal aresta indica a existência de uma borda ruim com $n_{AH}(C) = 1$;
- Ao percorrermos a borda, se as arestas da borda não aparecem ordenadas, existe uma borda ruim com $n_{AH}(C) = 0$.

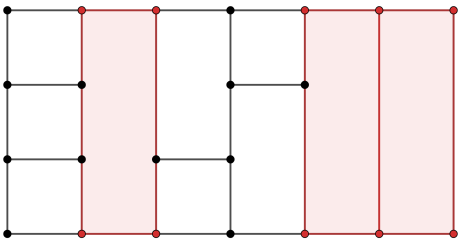
Voltando à função `Draw` do programa 3.2, temos 2 casos: `H` tem face separadora ou não. Se `H` tem face separadora, `separators` será uma lista contendo as faces separadoras ordenadas de cima para baixo ou da esquerda à direita. Então, a função `separateComponents` é utilizada para preparar `H` para que `Draw` seja chamado para seus subgrafos.

A figura 3.7 ilustra o processo feito por `separateComponents` quando há faces separadoras verticais:

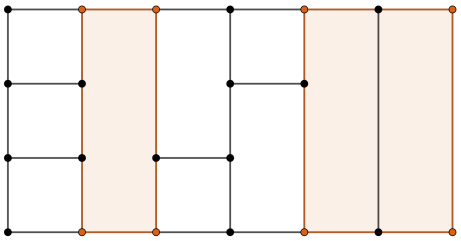
1. Cada face separadora é percorrida e todas as arestas internas de seu contorno têm direção e rótulos atualizados. Durante esse processo, são obtidos os cantos de cada face separadora.
2. A lista dos cantos, ordenada da esquerda à direita, é contraída de modo a obter os

cantos das regiões em que há só faces separadoras.

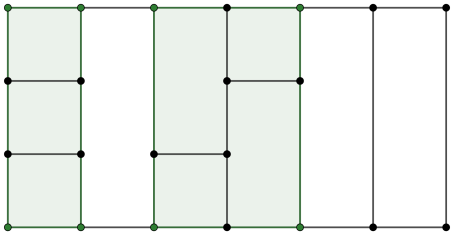
3. Utilizando `corners` e a lista anterior, são construídos os novos `corners` para cada subgrafo de `H` a ser processado recursivamente.



(a) *Em vermelho, estão destacados os cantos de cada face separadora.*



(b) *Os cantos coincidentes das faces separadoras vizinhas são desconsiderados. Os cantos que restam estão em amarelo.*



(c) *Analisando os cantos que restaram em relação aos cantos de `H`, é possível obter os cantos destacados em verde. São os cantos dos subgrafos para os quais `Draw` será chamado.*

Figura 3.7: *Desenho do processo de computação dos novos `corners` para os subgrafos de `H`.*

Existe um ponto que merece atenção no processo que descrevemos anteriormente. Pode existir um ciclo de 2 pernas entre 2 faces separadoras vizinhas. Nesse caso, essa componente ruim precisa ser detectada enquanto percorremos as faces separadoras.

Sempre que há uma variável nomeada como `H_attrs` em `Draw`, ela é uma tripla `(corners, update_side, memo)` que devolvemos com os argumentos para desenhar os subgrafos `H`. Então, outra questão importante que precisamos mencionar sobre `separateComponents` é como `update_side` e `memo` são computados. Quando há faces separadoras

verticais, `memo` de `H` é passado apenas ao subgrafo mais à esquerda e que compartilhe dos mesmos cantos noroeste e sudoeste que `H`. Se, por outro lado, há faces separadoras horizontais, `p` é repassado para todos os subgrafos de `H`. Já `a_list` é passado apenas ao subgrafo mais acima e que tenha os mesmos cantos noroeste e nordeste que `H`. `b_list`, similarmente, é passado apenas ao subgrafo mais abaixo e que tenha os mesmos cantos sudeste e sudoeste que `H`.

A escolha de `update_side` é simples. No caso de separação por faces separadoras verticais os valores possíveis são `"N"`, `"NS"` e `"S"`. Já no caso de faces separadoras horizontais são `"E"`, `"EW"` e `"W"`. Os casos em que só há uma borda para atualização acontece quando o subgrafo compartilha uma borda com `H`.

Se não há face separadora em `H`, `Partition` é chamado para que seja feita uma partição por P_H e P_{AH}. No programa 3.3 é possível ver o corpo da função `Partition`.

Programa 3.3 Função `Partition`.

```
1   def Partition(G, corners, memo=(None, None, None)):
2      a_list, b_list, p = memo
3
4      if not p:
5         p = WestmostPath(G, corners)
6         prepareClockwiseFaces(G, corners, p)
7         countEdgeTraversals(G, corners, p)
8         findCriticalCycles(G, corners, p, 0, len(p))
9         countCCLegs(G, corners, p, 0, len(p))
10
11     a_list, b_list = updateEdgesMemo(G, corners, (a_list, b_list, p))
12
13     leg = checkStartingPathLegs(G, corners, a_list[-1])
14     if leg: correctCriticalCycles(G, corners, p, leg)
15     leg = checkEndingPathLegs(G, corners, b_list[-1])
16     if leg: correctCriticalCycles(G, corners, p, leg)
17
18     return PartitionWithMemo(G, corners, (a_list, b_list, p))
```

Na função `Partition`, construímos ou atualizamos as variáveis para memoização `a_list`, `b_list` e `p` e chamamos `PartitionWithMemo`. Em `PartitionWithMemo`, `H` é particionado por P_H e P_{AH} utilizando `a_list`, `b_list` e `p`.

Se o NS-caminho mais à esquerda P de `H` não foi construído, `p` é `None`. Nesse caso, construímos `p` chamando `WestmostPath`. Tal construção é feita utilizando uma busca em profundidade anti-horária como descrito em (M. Rahman, Nakano *et al.*, 1998).

Uma vez tendo `p`, utilizamos `prepareClockwiseFaces` e `countEdgeTraversals` para percorrer as faces horárias anexadas a `p` de norte a sul e contar quantas vezes cada aresta foi percorrida. Uma vez tendo esta contagem atribuída a cada semi-aresta, `findCriticalCycles` encontra os ciclos críticos maximais anexados a `p`. Durante `prepareClockwiseFaces`, também atualizamos `e.is_on_p` de cada semi-aresta `e` contida em `p` e atribuímos o índice `e.p_index` de `e` em `p`. Então, em `countCCLegs` fazemos a contagem das pernas anti-horárias de cada ciclo crítico maximal C. Tal informação é mantida nos vértices anteriores e posteriores de C.

Ao final da execução das linhas 4-9, para cada vértice anterior (ou posterior) `v` de um ciclo crítico maximal anexado a `p` temos os seguintes atributos:

- `v.end_type`: `string` valendo `"H"` se `v` é anterior e `"T"` se `v` é posterior.
- `v.other_end`: semi-aresta `e` contida em `p` que é perna de C e é incidente ao vértice `u` posterior, se `v` é anterior, ou anterior, se `v` é posterior.
- `v.leg`: semi-aresta `e` contida em `p` que é perna horária de C e tem `v` no contorno da sua face incidente. Vale `None` se $n_H(C) = 0$.
- `v.ncc`: o número de pernas anti-horárias de C, ou seja, $n_{AH}(C)$.

Também temos que `v.end_type` é `None` para vértices em `p` que não são anteriores nem posteriores de algum ciclo crítico maximal. Pela definição anterior, se C é um ciclo crítico maximal anexado a `p`, `u` é vértice anterior de C e `v` é vértice posterior de C, então:

1. `u` é `v.other_end.destination`
2. `v` é `u.other_end.origin`
3. Se $n_H(C) = 1$, `u.leg` e `v.leg` são semi-arestas gêmeas

A figura 3.8 mostra os vértices anteriores e posteriores dos ciclos críticos maximais anexados ao NS-caminho mais à esquerda de um grafo plano.

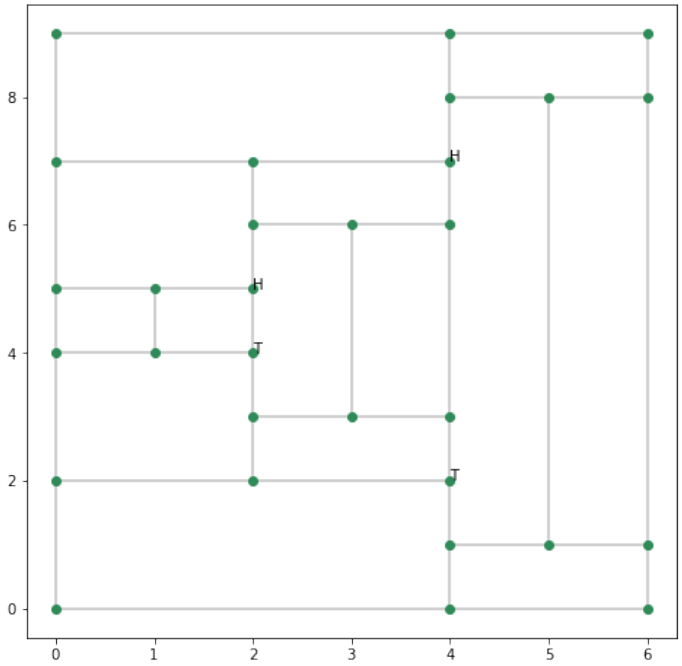


Figura 3.8: *Desenho de um grafo plano com os vértices antiores e posteriores dos ciclos críticos maximais rotulados com H e T.*

Para o restante do código, considere que temos `p` construído, assim como as informações relativas aos ciclos críticos maximais anexados a `p`. Logo, o NS-caminho mais à esquerda P' do subgrafo `H` que estamos desenhando é subcaminho de `p`.

Em `updateEdgesMemo` atualizamos e/ou construímos `a_list` e `b_list` para que se adéque ao P' atual. Ao particionar `H` por P'_H e P'_{AH} pode existir um ciclo crítico maximal C que tenha apenas uma parte de $Q_H(C)$ transformado em borda. Essa situação só ocorre quando P'_{ini} ou P'_{fim} passa pela perna horária de um ciclo crítico maximal C pois, por definição, P'_{AH} passa pelo Q_H de todos os ciclos críticos maximais entre e_a e e_b. E, quando isso ocorre, pode ser necessário recomputar os ciclos críticos maximais de uma parte de `p`. Em `checkStartingPathLegs` e `checkEndingPathLegs` verificamos se há algum ciclo crítico maximal com perna em P'_{ini} ou em P'_{fim}, respectivamente. Caso exista alguma correção a ser feita, chamamos `correctCriticalCycles`. A seção 3.3 pode ser consultada para um melhor entendimento dessas atualizações e correções envolvendo a memoização.

Programa 3.4 Função `PartitionWithMemo`.

```
1   def PartitionWithMemo(G, corners, memo):
2      a_list, b_list, p = memo
3
4      north_corner = StartingPartition(G, corners, a_list)
5
6      H_attrs = []
7      k = a_list[-1].p_index
8      while k <= b_list[-1].p_index:
9         setDirection(p[k], "vertical", "W", "E")
10        if p[k].destination.end_type == "H" and p[k].destination.other_end.
              p_index <= b_list[-1].p_index:
11           head_edge = p[k]
12           success, certificate, new_corners = partitionByCase(G, corners, memo,
                  head_edge)
13           if not success: return success, certificate, None
14           if new_corners: H_attrs += [(new_corners, None, (None, None, None))]
15           k = head_edge.destination.other_end.p_index
16        else:
17           k += 1
18
19     south_corner = EndingPartition(G, corners, b_list)
20
21     west_H_attr = [((corners[NW], north_corner, south_corner, corners[SW]), "E
           ", memo)]
22     east_H_attr = [((north_corner, corners[NE], corners[SE], south_corner), "W
           ", (None, None, None))]
23
24     return True, None, west_H_attr + H_attrs + east_H_attr
```

Em `PartitionWithMemo`, que pode ser visto no programa 3.4, finalmente particionamos `H`. Assim como em `separateComponents`, precisamos definir a direção e o rótulo para as arestas que farão parte das novas bordas e construir os parâmetros para às novas chamadas de `Draw`, ou seja, a lista de `H_attrs` para `Draw`.

Utilizando `StartingPartition` definimos a direção e o rótulo para as arestas de P'_{ini} e utilizando `EndingPartition` definimos a direção e o rótulo para as arestas de P'_{fim}. `north_corner` é o primeiro vértice de P'_{ini} enquanto `south_corner` é o último vértice de P'_{fim}. `west_H_attr` são os parâmetros para a chamada de `Draw` para $H_O^{P'}$, enquanto

`east_H_attr` é para $H_L^{P'}$. Note que `memo` é passado apenas para $H_O^{P'}$, pois P_H está à direita de P.

Nas linhas 6-17, percorremos o subcaminho de P' entre e_a, que é o último elemento de `a_list`, e e_b, que é o último elemento de `b_list`. Para cada ciclo crítico maximal que encontramos nesse subcaminho, chamamos `partitionByCase`. O conteúdo de `partitionByCase` pode ser visto no programa 3.5.

Programa 3.5 Função `partitionByCase`.

```
1   def partitionByCase(G, corners, memo, head_edge):
2      leg = head_edge.destination.leg
3      ncc = head_edge.destination.ncc
4      new_corners = None
5
6      if not leg:
7         partitionClockwisePath(G, corners, memo, head_edge)
8      elif ncc < 1:
9         return False, BadBigCycle(G, leg, head_edge), None
10     elif ncc == 1:
11        new_corners = partitionBigCycle(G, corners, memo, head_edge)
12     else:
13        success, certificate, found, cycle_legs = findMaximalCriticalCycle(G,
              corners, memo, head_edge)
14        if not success: return success, certificate, None
15        if found:
16           new_corners = partitionSmallCycle(G, corners, memo, head_edge,
                 cycle_legs)
17        else:
18           partitionClockwisePath(G, corners, memo, head_edge)
19
20     return True, None, new_corners
```

Em `partitionByCase`, `head_edge` é a perna de um ciclo crítico maximal C anexado a P' que é incidente ao vértice anterior de C. Nessa função, verificamos se C satisfaz o caso 1, 2, 3.1 ou 3.2 da construção de P_H e P_{AH} que descrevemos para o lema 2.12.

Para os casos 1 e 3.1, a partição é a mesma e é feita em `partitionClockwisePath`. Já para o caso 2, temos um ciclo entre P_H e P_{AH}. A partição é feita em `partitionBigCycle` e obtemos os cantos do ciclo como `new_corners`. Como C é crítico, se $Q_{AH}(C) < 1$, sabemos que C forma uma componente interna ruim. Nesse caso chamamos `BadBigCycle` para obter o certificado, que é o próprio C.

A função `findMaximalCriticalCycle` verifica se C satisfaz o caso 3.2, ou seja, se existe um ciclo crítico anti-horário anexado a $Q_H(C)$. Caso o ciclo crítico maximal C' que encontramos anexado a $Q_H(C)$ não tenha 4 pernas, temos uma componente interna ruim e não é possível continuar a partição. Portanto, `findMaximalCriticalCycle` devolve `success` e `certificate`, que analisamos antes de chamar `partitionSmallCycle`. A variável `found` é *booleana* e representa se C satisfaz o caso 3.2, enquanto `cycle_legs` é uma tripla contendo as pernas de C' contidas em $G(C)$. Finalmente, em `partitionSmallCycle` particionamos de acordo com o caso 3.2 e devolvemos os cantos do ciclo entre P_H e P_{AH} como `new_corners`.

Na nossa implementação, não fazemos contração de arestas para transformar $H_O^{p'}$ e $H_L^{p'}$ em retângulos de fato, pois não utilizamos a direção das semi-arestas durante a computação. Em vez disso, utilizamos sempre os rótulos e, em situações como o caso 2 e 3.2 em que há ciclos entre P_H e P_{AH}, rotulamos as semi-arestas como na figura 3.9. Assim, uma semi-aresta com rótulo `"N"` ou `"S"` é sempre horizontal, mas uma semi-aresta com rótulo `"E"` ou `"W"` nem sempre é vertical.

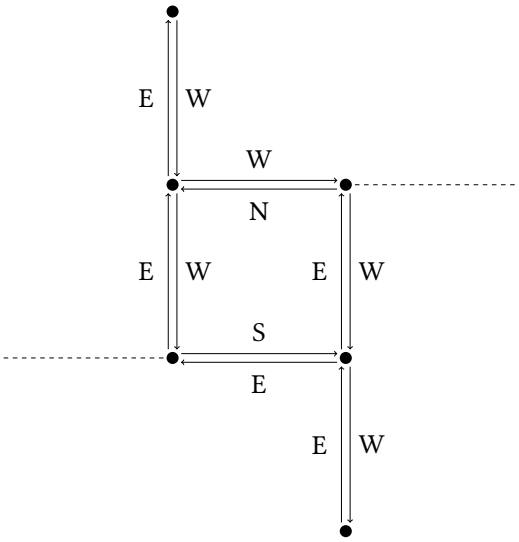


Figura 3.9: *Ilustração dos rótulos das semi-arestas em uma partição por P_H e P_{AH}.*

Portanto, não realizamos as operações de união e concatenação que aparecem no capítulo 2.

Pelo processo de divisão que fazemos recursivamente, as componentes internas ruins tornam-se bordas ou cantos ruins. No entanto, há alguns casos em que precisamos verificar se há componentes ruins durante a divisão. Resumidamente, nós lidamos com a inexistência de desenho retangular verificando:

A. Existência de bordas ruins ou cantos ruins

B. Existência de ciclo de 2 pernas entre faces separadoras

C. Existência de ciclo com no máximo 3 pernas nos casos 2 e 3.2

Para finalizar esta seção, nós lidamos com vértices de grau 2 na construção de `p` e detecção de ciclos críticos maximais. Um cuidado adicional é necessário na detecção dos ciclos críticos maximais pois se, por exemplo, C é um ciclo crítico maximal e a ponta da perna horária e de C é um vértice v de grau 2, a outra aresta e' incidente a v também poderá ser detectada como perna. Isso aconteceria se levássemos em consideração apenas a contagem feita em `countEdgeTraversals`, pois e' também é percorrida 2 vezes. Para lidar com essas situações em `findCriticalCycles`, desconsideramos e' ao verificar que a semi-aresta gêmea de e' tem a mesma face incidente que a semi-aresta gêmea de e.

Se G não tem vértices de grau 2 fora de $C_o(G)$, todo subgrafo $H = G(C)$, em que C é um ciclo de G, não terá vértices de grau 2 fora de $C_o(H) = C$. Portanto, utilizando o conceito

de grafo homomórfico minimal que mencionamos na seção 2.3 é possível simplificar a construção de p e detecção de ciclos críticos maximais.

3.3 Memoização

Finalmente, nesta seção vamos considerar a complexidade de tempo do algoritmo. Para isso, faremos referência às funções que foram apresentadas na seção 3.2.

Primeiramente, na função `DrawGraph` fazemos um pré-processamento de `G` que é $O(n)$ antes de chamarmos `Draw`. Nas funções `updateOuterEdges`, `isSquare`, `BoundariesUpdate` e `separateComponents`, as faces de borda das bordas indicadas por `update_side` são percorridas um número constante de vezes. Similarmente, em `PartitionWithMemo` as novas faces de borda geradas pela partição por P_H e P_{AH} são percorridas um número constante de vezes. Considerando apenas essas operações, pelo mesmo argumento dado ao final da seção 2.1 vemos que o algoritmo é linear. Logo, o nosso desafio é entender o custo de `Partition`.

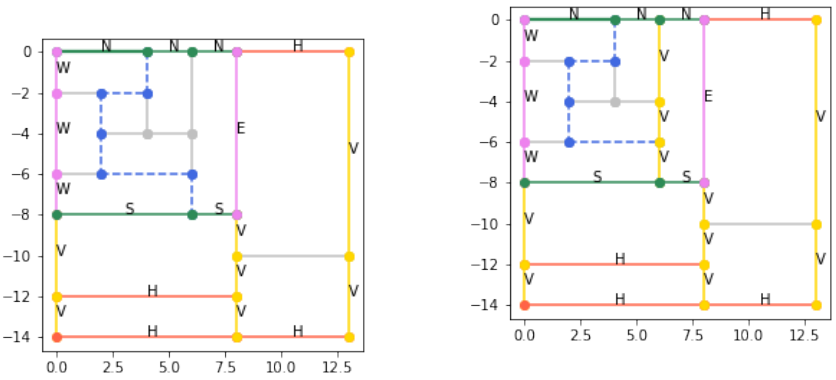
Como mencionado na seção 2.1, não podemos reconstruir `p` novamente em todas as chamadas recursivas. Logo, nós passamos `p` para os níveis mais profundas da recursão por meio de `memo` e acabamos gerando a necessidade de atualizar `memo`. O custo que precisamos considerar é, portanto:

1. Custo de correção dos ciclos críticos maximais detectados, ou seja, correção dos atributos dos vértices de `p`
2. Custo de construção e atualização de `a_list` e `b_list`

Primeiramente, vejamos o exemplo da figura 3.10 que mostra um passo do desenho retangular de um grafo plano G. As bordas do subgrafo H que está sendo desenhado estão destacadas, com os rótulos aparentes, e o NS-caminho mais à esquerda P' também está destacado por linhas tracejadas. Nesse exemplo, um ciclo crítico maximal anexado a P' deixa de ser ciclo crítico maximal. Isto ocorre após uma divisão feita por uma face separadora vertical. Como resultado, um ciclo crítico C' que não tinha sido considerado pela regra dos parênteses passa a ser um ciclo crítico maximal do próximo subgrafo a ser desenhado.

Essas correções podem ocorrer devido à partição utilizando P_{ini}, P_{fim} ou pela separação da face separadora vertical f mais à esquerda. No caso de P_{ini}, podemos ter um ciclo crítico C' com o mesmo vértice anterior que um ciclo crítico maximal C. Similarmente, no caso de P_{fim}, podemos ter um ciclo crítico C' com o mesmo vértice posterior que um ciclo crítico maximal C. Enquanto ao separar por f, podemos ter ambos os casos. Vamos analisar o custo de correção no caso em que C' compartilha o vértice anterior com C, esquematizado na figura 3.11. A análise é similar para o outro caso.

O que o esquema mostra resumidamente é a atualização do vértice anterior de C e dos vértices posteriores de C e C'. Após isso, também é necessário revermos todos os ciclos críticos anexados a P que estão contidos em $G(C')$. Para que tudo isso seja possível, precisamos conseguir detectar C' eficientemente. Isso é feito reutilizando a



(a) *Passo k: existe um ciclo crítico maximal C anexado a P′.*

(b) *Passo k + 1: C deixa de ser ciclo crítico maximal e C′, ciclo crítico contido em G(C), passa a ser ciclo crítico maximal tendo o mesmo vértice anterior que C.*

Figura 3.10: *Exemplo de passo do desenho retangular de um grafo plano G em que a correção dos ciclos críticos maximais é necessária.*

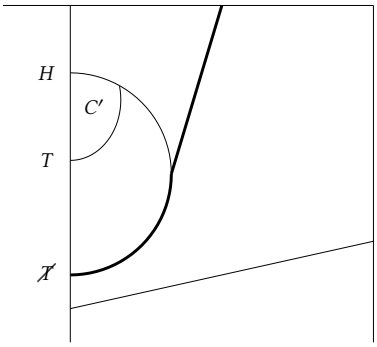


Figura 3.11: *Esquema de correção de um ciclo crítico maximal C anexado ao NS-caminho mais à esquerda P para o caso em que o vértice anterior é compartilhado.*

contagem que fizemos em `countEdgeTraversals`. O programa 3.6 mostra o corpo da função `correctCriticalCycles`.

Na função `correctCriticalCycles`, `p` é o NS-caminho mais à esquerda e `leg` é semi-aresta associada à perna horária de C' que já foi encontrada. `leg`, nesse caso, é a semi-aresta incidente à mesma face que contém o vértice anterior de C'. `head_edge` obtido nas linhas 2-4 é a semi-aresta de `p` incidente ao vértice anterior de C'. Similarmente, `tail_edge` obtido nas linhas 6-8 é a semi-aresta de `p` incidente ao vértice posterior de C'. Então, nas linhas 10-15 atualizamos os atributos dos vértices anteriores e posteriores de C e C'.

Tendo os índices de `head_edge` e `tail_edge` em `p`, sabemos a parte de `p` que precisa de correção. A função `reprepareClockwiseFaces` é chamada para preparar as faces horárias anexadas nessa parte e repetirmos `findCriticalCycles` e `countCCLegs`. `reprepareClockwiseFaces` irá reinicializar todos os `v.end_type` para `None` dos vértices `v` de `p` dentro de $G(C')$ e, além disso, irá trocar a contagem das arestas que estão com -1 para 2. Essa contagem representa quantas vezes a aresta foi percorrida quando percorremos as faces horárias anexadas a `p`, e é a que utilizamos para detectar pernas de ciclos

Programa 3.6 Função `correctCriticalCycles`.

```
1   def correctCriticalCycles(G, corners, p, leg):
2      e = leg
3      while not e.is_on_p: e = e.prev_edge
4      head_edge = e
5
6      e = leg.twin
7      while not e.is_on_p: e = e.next_edge
8      tail_edge = e
9
10     head_vertex, tail_vertex = head_edge.destination, tail_edge.origin
11     if head_vertex.end_type: head_vertex.other_end.origin.end_type = None
12     if tail_vertex.end_type: tail_vertex.other_end.destination.end_type = None
13     head_vertex.end_type, tail_vertex.end_type = "H", "T"
14     head_vertex.other_end, tail_vertex.other_end = tail_edge, head_edge
15     head_vertex.leg, tail_vertex.leg = leg, leg.twin
16
17     start, stop = head_edge.p_index+1, tail_edge.p_index
18     reprepareClockwiseFaces(G, corners, p, start, stop)
19
20     e = leg
21     while not e.is_on_p:
22        if e.counter == 2: setCounter(G, e, -1)
23        e = e.prev_edge
24     e = leg.twin
25     while not e.is_on_p:
26        if e.counter == 2: setCounter(G, e, -1)
27        e = e.next_edge
28
29     findCriticalCycles(G, corners, p, start, stop)
30     countCCLegs(G, corners, p, start-1, stop+1)
```

críticos.

Um detalhe que não foi mencionado quando falamos de `findCriticalCycles` e que é importante para entender a relação entre os ciclos críticos maximais é a existência de ciclos críticos que compartilham vértices anteriores ou posteriores, como é o caso de C' e C. Quando construímos `p` inicialmente, somos obrigados a ignorar C', ou seja, não podemos detectar a perna de C' como a perna de um ciclo crítico maximal, pois isso violaria a regra dos parênteses. Isso é feito atualizando a contagem da perna de C' para o valor -1 uma vez que encontramos C. Esse processo é idêntico ao que fazemos nas linhas 20-27 utilizando `setCounter`, onde atualizamos a contagem das arestas de C' de modo que toda perna que estiver em C' recebe o valor -1.

A figura 3.12 mostra a contagem associada às arestas após a execução do algoritmo de desenho retangular para um grafo plano. As arestas com contagem -1 foram percorridas duas vezes. Note que o grafo plano é o mesmo da figura 3.10 e as arestas com contagem -1 são as pernas de C e C' mencionadas anteriormente.

Então, para corrigir um ciclo crítico maximal C contendo um C' como mencionamos anteriormente, nós percorremos as arestas de C e as faces contidas em $G(C')$ um número

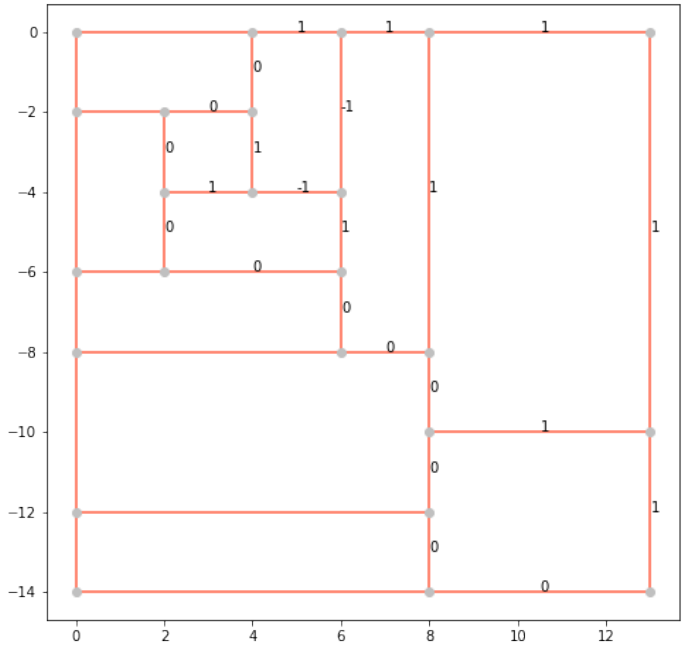


Figura 3.12: *Desenho de um grafo plano mostrando a contagem associada à cada aresta.*

constante de vezes. Se tivéssemos C, C', C'', ... aninhados e fosse necessário fazer correções recursivamente numa parte de `p` que já foi corrigida, o tempo resultante poderia não ser linear. Isso não ocorre, pois os ciclos críticos maximais que precisamos corrigir são disjuntos.

A existência de ciclos críticos maximais como C implica na existência de um ciclo crítico C_N anexado a $P_N^D \cdot P$, sendo P_N^D o subcaminho da borda norte P_N à direita de P. Na figura 3.11, $Q_H(C_N)$ está destacado com linhas mais grossas.

Pela divisão que fazemos, todo subgrafo contendo um subcaminho P' de `p` terá como P_N^D um subcaminho da borda norte do grafo plano em que `p` foi originalmente construído ou um subcaminho de `p`. Se existir um ciclo crítico maximal C'' em $G(C)$ contendo um C''' que necessite de correção, P_N^D será um subcaminho de `p`. Logo, existiria um C_N'' crítico contendo C'' e a consideração de C'' como maximal violaria a regra dos parênteses que utilizamos.

Portanto, isso mostra que os ciclos críticos maximais que necessitam de correção são disjuntos e o custo das correções não eleva a complexidade do algoritmo. A análise é similar para o caso dos vértices posteriores, considerando P_S no lugar de P_N.

Para entender o custo de construção e atualização de `a_list` e `b_list`, vamos utilizar o programa 3.7 como apoio. As linhas 4-8 mostram a atualização de `a_list` e construção de `b_list` para um grafo plano H que já recebeu `a_list` construído. Já as linhas 9-13 mostram a construção de `a_list` e atualização de `b_list` para um H que já recebeu `b_list` construído. Por fim, nas linhas 14-19 temos a construção de `a_list` e `b_list` para H sem ambos.

O primeiro fato que precisamos notar é que `Partition`, e consequentemente `updateEdgesMemo`, nunca será chamado para um subgrafo H que tenha ambos `a_list` e

Programa 3.7 Função `updateEdgesMemo`.

```
1   def updateEdgesMemo(G, corners, memo):
2      a_list, b_list, p = memo
3
4      if a_list:
5         while isOuterEdge(G, a_list[-1]): a_list.pop()
6         first_index = a_list[-1].p_index
7         last_index = LastPIndex(G, corners)
8         b_list = EndingPathCandidates(G, corners, (a_list, b_list, p), (
              first_index, last_index))
9      elif b_list:
10        while isOuterEdge(G, b_list[-1]): b_list.pop()
11        first_index = FirstPIndex(G, corners)
12        last_index = b_list[-1].p_index
13        a_list = StartingPathCandidates(G, corners, (a_list, b_list, p), (
              first_index, last_index))
14     else:
15        first_index = FirstPIndex(G, corners)
16        last_index = LastPIndex(G, corners)
17        a_list = StartingPathCandidates(G, corners, (a_list, b_list, p), (
              first_index, last_index))
18        first_index = a_list[-1].p_index
19        b_list = EndingPathCandidates(G, corners, (a_list, b_list, p), (
              first_index, last_index))
20
21     return a_list, b_list
```

`b_list` construídos. Isso se deve ao fato que e_a e e_b passam a fazer parte da borda uma vez que a partição por P_H e P_{AH} ocorre. Assim, P, quando diferente de P_H, é quebrado em subcaminhos e na próxima chamada de `Partition` as listas `a_list` e `b_list` que foram construídas anteriormente estarão em subgrafos planos disjuntos. A reconstrução dos subcaminhos de P como NS-caminhos mais à esquerda de subgrafos planos disjuntos é feita pelo `separateComponents` com faces separadoras horizontais.

A função `isOuterEdge` verifica se uma semi-aresta `e` é externa. A atualização de `a_list` ocorre na linha 5, enquanto a de `b_list` ocorre na linha 10 utilizando tal função. Em ambos os casos, retiramos as últimas semi-arestas da lista que se tornaram semi-arestas externas. Então, uma vez construídas tais listas, elas somente diminuem. Vamos analisar porque `a_list` pode ser atualizado dessa forma. A análise é similar para `b_list`.

Suponha que um subgrafo plano H recebe `a_list` e irá executar `Partition`. Seja H_0 o subgrafo plano contendo H onde `Partition` foi executado pela última vez antes de H. Durante a partição de H_0 por P_{ini}, as últimas semi-arestas de `a_list` passaram a ser externas. Logo, a remoção das últimas semi-arestas de `a_list` que são externas manteria `a_list` corretamente. Após isso, algumas execuções de `separateComponents` podem ter ocorrido até obtermos H, gerando os subgrafos intermediários $H_1, H_2, \cdots, H_k$ para os quais `Draw` foi chamado. Todos os H_i estão contidos em H_0 e contêm H, e podemos considerar que $H_{i+1} \subseteq H_i$ para todo i, $1 \leq i \leq k$ sem perda de generalidade. Ademais, todos os H_i mantém `a_list` intactos na nossa implementação. No entanto, considere que `a_list` é atualizado entre os H_i retirando-se as semi-arestas que já são externas. Isto poderá facilitar

o entendimento da explicação que se segue.

Pela maneira que repassamos `a_list`, a borda norte de qualquer H_i é prefixo da borda norte de H_0 e o NS-caminho mais à esquerda de H_i é prefixo do NS-caminho mais à esquerda de H_0. Suponha que fazemos uma separação por faces separadoras verticias e f é a face separadora mais à esquerda em H_i. Se f não intercepta o NS-caminho mais à esquerda P_i de H_i, o `a_list` se mantém igual. Então, suponha que f intercepta P_i. Nesse caso, as semi-arestas em comum entre o contorno de f e P_i são parte de `a_list`. Ademais, se `e` é uma semi-aresta de $C(f) \cup P_i$, toda semi-aresta `e'` em `a_list` após `e` também deverá fazer parte de $C(f) \cup P_i$, pois a existência de f impede que exista um caminho de borda contendo `e'` e interceptando a borda norte, contrariando que `e'` esteja em `a_list`. Então, podemos atualizar `a_list` corretamente removendo as últimas semi-arestas que são externas.

Se, por outro lado, temos uma separação por faces separadoras horizontais em H_i, nenhuma semi-aresta de `a_list` poderá estar abaixo de uma face separadora horizontal. Isso se deve ao fato de que as semi-arestas mantidas em `a_list` necessariamente estão em um caminho de borda que intercepta a borda norte. Logo, numa separação por faces separadoras horizontais, a única face separadora que poderá conter semi-arestas de `a_list` é a face mais acima f. Como as semi-arestas de `a_list` fazem parte de `p`, a intersecção entre `a_list` e o contorno de f é subcaminho de `p`. Então, as semi-arestas de `a_list` que passarão a ser externas são todas as últimas semi-arestas de `a_list` contidas nesse subcaminho. Portanto, podemos atualizar, `a_list` corretamente removendo as últimas semi-arestas que são externas.

Voltando à função `updateEdgesMemo`, as variáveis `first_index` e `last_index` que utilizamos indicam os índices do subcaminho de `p` que iremos percorrer para construir `a_list` ou `b_list`. A função `FirstPIndex` nos dá o `p_index` da primeira semi-aresta do NS-caminho mais à esquerda de P' do subgrafo plano H que estamos desenhando. Similarmente, a função `LastPIndex` nos dá o `p_index` da última semi-aresta de P'. Para calcular `FirstPIndex` nós percorremos a parte da borda norte à esquerda de P' e, similarmente, para `LastPIndex` percorremos a parte esquerda da borda sul. Uma vez que `a_list` é construído, como o prefixo da borda norte sempre se mantém, é fácil ver que o custo de `FirstPIndex` não afeta a complexidade do tempo resultante do algoritmo. O mesmo argumento pode ser feito considerando `b_list` e `LastPIndex`.

Quando `a_list` já está construído, atualizamos `a_list` e utilizamos a última semi-aresta como `first_index` pois e_a sempre vem antes de e_b na ordem de `p`. Assim, todas as semi-arestas antes de e_a não são percorridas. Similarmente, quando `b_list` já está construído, utilizamos a última semi-aresta como `last_index`.

Resta entender o custo de `StartingPathCandidates` e `EndingPathCandidates`. Tanto em `StartingPathCandidates` quanto `EndingPathCandidates` percorremos o subcaminho de `p` entre `first_index` e `last_index` verificando o rótulo das faces incidentes às semi-arestas. Também vale que, em ambos, não passamos pelas semi-arestas que estão contidas em algum ciclo crítico anexado a P', pois tais semi-arestas com certeza não estarão em `a_list` nem `b_list`. Para que isso possa ser feito eficientemente, utilizamos os atributos `other_end` dos vértices anteriores e posteriores dos ciclos críticos maximais.

Então, quando construímos `a_list` e `b_list` pela primeira vez percorremos um número constante de vezes as seguintes semi-arestas de `p`:

(i) semi-arestas que fazem parte de `a_list` ou `b_list`

(ii) semi-arestas que estão em $P'_H \cup P'_{AH}$

(iii) semi-arestas que estão no subcaminho inicial de P' terminando em e_a mas não fazem parte de `a_list` nem estão contidas em algum ciclo crítico de P'

(iv) semi-arestas que estão no subcaminho final de P' iniciado em e_b mas não fazem parte de `b_list` nem estão contidas em algum ciclo crítico de P'

As semi-arestas em (ii) se tornarão externas e não serão percorridas novamente. Já as semi-arestas em (i) também não serão percorridas novamente, pelo fato de que `a_list` e `b_list` somente diminuem e pela definição que fazemos de `first_index` e `last_index` quando `a_list` ou `b_list` já está construído.

Poderíamos ter um problema com a complexidade do algoritmo se as semi-arestas em (iii) e (iv) fossem percorridas repetidas vezes em `updateEdgesMemo` nos próximos níveis recursivos. Isso não ocorre, pois na próxima chamada de `updateEdgesMemo` as semi-aresta de (iii) que forem percorridas se tornarão (i) ou (ii). O caso que analisaremos aqui é o (iii), o argumento é similar para (iv) considerando a escolha de `b_list`.

Supondo que construímos `a_list` em H_0, onde H_0 é novamente o menor supergrafo de H em que `Partition` foi executado. As semi-arestas que estavam no caso (iii) durante a partição de H_0 e que não se tornaram externas nas próximas separações aparecem no NS-caminho mais à esquerda P' de H. Em H, executamos as linhas 4-8 e construímos `b_list` percorrendo o subcaminho P'_a de P' entre e_a e a última semi-aresta de P'.

Suponha que uma semi-aresta e que satisfazia (iii) na partição de H_0 está em P'_a. Como percorremos e na partição por H_0, e não está em um ciclo crítico de P'. Logo, se e não se tornou (i) nem (ii) no `updateEdgesMemo` chamado para H, como as semi-arestas anteriores a e_a não foram percorridas, e é uma semi-aresta do caso (iv) para H.

Considere o P'_{fim} construído a partir do `b_list` de H. Se e está entre duas semi-arestas de P'_{fim} em P', e está em um ciclo crítico de P', contradição. Então, como e satisfaz o caso (iv), e precisa estar no ciclo C formado entre P'_{fim}, P'^D_S e P', sendo P'^D_S o subcaminho da borda sul à direita de P'. Nesse caso, P'^D_S é subcaminho do NS-caminho mais à esquerda de H_0. Logo, C era ciclo crítico em H_0 e e não pode ter sido percorrido em H_0, contradição.

Portanto, isso mostra que percorremos as semi-arestas de `p` um número constante de vezes para construir e atualizar as variáveis `a_list` e `b_list`.

Finalmente, podemos concluir que as operações de `Partition` são eficientes e o algoritmo de desenho retangular que implementamos é linear.

Capítulo 4

Conclusão

Neste trabalho, conseguimos desenvolver e detalhar algumas partes da teoria de desenhos retangulares para cantos fixos que não estavam totalmente detalhadas nas referências que utilizamos. Também foi possível implementar, com algumas adaptações, os algoritmos fundamentais de desenhos retangulares que são utilizados em outros métodos de desenhos como desenhos caixa-retangulares e desenhos ortogonais.

O material que foi construído ao longo do desenvolvimento deste tema poderá servir de suporte para entender as partes mais complicadas do algoritmo memoizado. Ademais, as ideias que desenvolvemos para explicar a corretude e eficiência do algoritmo poderão ser úteis, do ponto de vista teórico, para outros algoritmos de desenhos de grafos.

Sem dúvida, o trabalho que desenvolvemos deixa aberta a possibilidade para diversos trabalhos futuros. Como mencionamos na seção 2.3, o estudo e implementação dos algoritmos de algumas referências como (M. Rahman, Nakano *et al.*, 2002) e (M. Rahman, Nishizeki *et al.*, 2004) serviriam de continuação natural ao que desenvolvemos aqui. Mas, além disso, temas relacionados aos diferentes desenhos retangulares que um mesmo grafo plano pode ter ou o desenvolvimento de algoritmos para novos desenhos baseados em desenhos retangulares podem ser temas interessantes para pesquisa.

Referências

[Assunção 2012] Guilherme Puglia Assunção. "Representações retangulares de grafos planares". Diss. de mestr. São Paulo, Brasil: Instituto de Matemática e Estatística, Universidade de São Paulo, abr. de 2012 (citado nas pgs. 1, 6, 7, 11, 12, 14, 20–23, 26, 33, 34).

[Berg *et al.* 2000] Mark de Berg, Marc van Kreveld, Mark Overmars e Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Third. Springer-Verlag, 2000, pgs. 29–33. URL: http://www.cs.uu.nl/geobook/ (citado na pg. 42).

[Bhasker e Sahni 1988] Jayaram Bhasker e Sartaj Sahni. "A linear algorithm to find a rectangular dual of a planar triangulated graph". Em: *Algorithmica* 3 (nov. de 1988), pgs. 247–278. DOI: 10.1007/BF01762117 (citado na pg. 14).

[Ellis-Monaghan e Gutwin 2003] Joanna Ellis-Monaghan e Paul Gutwin. "Graph theoretical problems in next-generation chip design". Em: *Congressus Numerantium* (jan. de 2003) (citado na pg. 1).

[Hopcroft e Tarjan 1973] John Hopcroft e Robert Tarjan. "Algorithm 447: efficient algorithms for graph manipulation". Em: *Commun. ACM* 16.6 (jun. de 1973), pgs. 372–378. ISSN: 0001-0782. DOI: 10.1145/362248.362272. URL: https://doi.org/10.1145/362248.362272 (citado na pg. 38).

[Kozminski e Kinnen 1984] Krzysztof Kozminski e Edwin Kinnen. "An algorithm for finding a rectangular dual of a planar graph for use in area planning for VLSI integrated circuits." Em: *Proceedings of the 21st Design Automation Conference*. DAC '84. Albuquerque, New Mexico, USA: IEEE Press, 1984, pgs. 655–656. ISBN: 0818605421 (citado na pg. 14).

[Nishizeki e Md Saidur Rahman 2004] Takao Nishizeki e Md Saidur Rahman. *Planar Graph Drawing*. WORLD SCIENTIFIC, 2004. DOI: 10.1142/5648. eprint: https://www.worldscientific.com/doi/pdf/10.1142/5648. URL: https://www.worldscientific.com/doi/abs/10.1142/5648 (citado na pg. 37).

[Md. Saidur Rahman *et al.* 1999] Md. Saidur Rahman, Shin-Ichi Nakano e Takao Nishizeki. "A linear algorithm for bend-optimal orthogonal drawings of triconnected cubic plane graphs". Em: *J. Graph Algorithms Appl.* 3.4 (1999), pgs. 31–62. doi: 10.7155/jgaa.00017. url: https://doi.org/10.7155/jgaa.00017 (citado nas pgs. 38, 39).

[M. Rahman, Nakano *et al.* 1998] Md.Saidur Rahman, Shin-ichi Nakano e Takao Nishizeki. "Rectangular grid drawings of plane graphs". Em: *Computational Geometry* 10.3 (1998), pgs. 203–220. issn: 0925-7721. doi: https://doi.org/10.1016/S0925-7721(98)00003-0. url: https://www.sciencedirect.com/science/article/pii/S0925772198000030 (citado nas pgs. 1, 7, 9, 13, 14, 29, 32, 40, 55).

[M. Rahman, Nakano *et al.* 2000] Md.Saidur Rahman, Shin-ichi Nakano e Takao Nishizeki. "Box-rectangular drawings of plane graphs". Em: *Journal of Algorithms* 37.2 (2000), pgs. 363–398. issn: 0196-6774. doi: https://doi.org/10.1006/jagm.2000.1105. url: https://www.sciencedirect.com/science/article/pii/S0196677400911052 (citado na pg. 38).

[M. Rahman, Nakano *et al.* 2002] Md.Saidur Rahman, Shin-ichi Nakano e Takao Nishizeki. "Rectangular drawings of plane graphs without designated corners". Em: *Computational Geometry* 21.3 (2002), pgs. 121–138. issn: 0925-7721. doi: https://doi.org/10.1016/S0925-7721(01)00061-X. url: https://www.sciencedirect.com/science/article/pii/S092577210100061X (citado nas pgs. 1, 39, 40, 67).

[M. Rahman, Nishizeki *et al.* 2004] Md.Saidur Rahman, Takao Nishizeki e Shubhashis Ghosh. "Rectangular drawings of planar graphs". Em: *Journal of Algorithms* 50.1 (2004), pgs. 62–78. issn: 0196-6774. doi: https://doi.org/10.1016/S0196-6774(03)00126-3. url: https://www.sciencedirect.com/science/article/pii/S0196677403001263 (citado nas pgs. 1, 39, 40, 67).

[Tamassia 2016] Roberto Tamassia. *Handbook of Graph Drawing and Visualization*. 1st. Chapman & Hall/CRC, 2016. isbn: 113803424X (citado na pg. 1).

[Thomassen 1984] Carsten Thomassen. "Plane representations of graphs". Em: *Progress in Graph Theory, 1984* (1984), pgs. 43–69 (citado nas pgs. 13, 14).