Análise experimental de algoritmos de planaridade

Alexandre Noma

Dissertação apresentada ao
Instituto de Matemática e
Estatística da
Universidade de São Paulo
como parte dos requisitos
para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profa. Dra. Cristina Gomes Fernandes.

— São Paulo, Maio de 2003 —

– Durante o desenvolvimento deste trabalho, o aluno recebeu apoio financeiro da FAPESP–

Análise experimental de algoritmos
para teste de planaridade

Este exemplar corresponde à redação final
da dissertação devidamente corrigida e
defendida por Alexandre Noma
e aprovada pela comissão julgadora.

São Paulo, 9 de maio de 2003

Banca examinadora:

- Profa. Dra. Cristina Gomes Fernandes - IME-USP
- Prof. Dr. Paulo Feofiloff - IME-USP
- Prof. Dr. Jorge Stolfi - IC-UNICAMP

Aos meus pais Massagi e Eliana.
Aos meus irmãos Eduardo e Beatriz.
A todos meus amigos ...
... e à Ayumi, com muito amor.

Agradecimentos

Agradeço a Deus por me conceder uma família tão amorosa e pela oportunidade de estudar numa faculdade como o IME.

Meus sinceros agradecimentos à professora *Cristina Gomes Fernandes* e ao professor *José Coelho de Pina*, que não foi citado na capa apesar de ter sido a peça fundamental para esta dissertação. Agradeço pela amizade, forma dedicada e paciente com que me orientaram durante todo o período do mestrado.

Agradeço aos meus pais, *Massagi* e *Eliana*, por todo amor, carinho e dedicação durante todos os dias de minha vida, e aos meus irmãos, *Eduardo* e *Beatriz*, pelas brincadeiras, risadas e companhia.

Aos meus tios *Newton* e *Toshiko* e aos meus avós *Tatsuma* e *Yolanda* por todo amor e apoio.

Ao *Renato Carmo* pelo apoio durante a fase inicial deste trabalho e ao *Shigueo* e *Wagner Dias* pela amizade e ajuda durante os períodos de graduação e de pós.

Aos amigos *Cebola*, *William*, *Henrique* e *Masaishi* pelas aulas de tênis e pelo prazer de suas companhias.

Aos colegas *Rodrigo*, *Eduardo*, *Maurício*, *Romão*, *Ruth*, *Hamada*, *Vaz*, *Herbert*, *Leirê*, *Agnês*, *Tadao*, *Said*, *Fábio*, *Autoexec*, *Roy*, *Leston*, *Claus*, *Gruber* e a todos aqueles que não foram citados, mas que estão presentes em meu coração.

Um obrigado especial aos professores *Zé Augusto*, *Nami*, *Feofiloff*, *Carlinhos*, *Antônio Elias*, *Yoshiko*, *Siang* e *Mandel* pela atenção e por todos os ensinamentos que adquiri.

À FAPESP pelo apoio financeiro e a todos os funcionários do IME.

Finalmente, um obrigado muitíssimo especial à minha namorada *Ayumi* pela paciência, amor e por todo carinho.

Resumo

O *problema da planaridade* consiste em: dado um grafo G, decidir se G é ou não é planar. A resposta deve vir acompanhada de uma justificativa. Se G é planar, então uma justificativa é uma representação de um desenho de G no plano sem cruzamento de arestas. Se G não é planar, então uma justificativa é uma subdivisão do $K_{3,3}$ ou do K_5 em G.

Existem vários algoritmos lineares para testar planaridade. Dois deles são bem conhecidos: o algoritmo proposto por Auslander e Parter [1], posteriormente corrigido por Goldstein [9] (APG), e o algoritmo proposto por Lempel, Even e Cederbaum [12] (LEC).

Hopcroft e Tarjan [10] apresentaram em 1974 a primeira implementação linear do algoritmo de APG. Pouco depois, Booth e Lueker apresentaram uma implementação linear do algoritmo de LEC, introduzindo uma nova estrutura de dados chamada PQ-árvore. Recentemente, Shih e Hsu [15] e Boyer e Myrvold [3] publicaram duas implementações lineares do algoritmo de LEC que evitam o uso da PQ-árvore.

Este trabalho apresenta uma descrição do algoritmo de LEC e uma descrição da implementação de Shih e Hsu, bem como um estudo comparativo desta implementação com as implementações de Hopcroft e Tarjan, Booth e Lueker e de Boyer e Myrvold.

Abstract

The *planarity testing problem* consists of: given a graph G, decide whether G is planar or not. The answer must be followed by a justification. If G is planar, then a justification is a representation of a drawing of G in the plane with no crossing edges. If G is not planar, then a justification is a subdivision of $K_{3,3}$ or K_5 in G.

There are several linear-time algorithms to test planarity. Two of them are well known: the algorithm proposed by Auslander and Parter [1], later corrected by Goldstein [9] (APG), and the algorithm proposed by Lempel, Even e Cederbaum [12] (LEC).

In 1974, Hopcroft and Tarjan [10] presented the first linear-time implementation of APG's algorithm. Not long after that, Booth and Lueker presented a linear-time implementation of LEC's algorithm introducing a new data structure named PQ-tree. Recently, Shih and Hsu [15] and Boyer and Myrvold [3] proposed two linear-time implementations of LEC's algorithm that avoids using PQ-tree.

This work presents a description of LEC's algorithm and a description of Shih and Hsu's implementation, as well as a comparative study involving this implementation, Hopcroft and Tarjan's, Booth and Lueker's and Boyer and Myrvold's implementation.

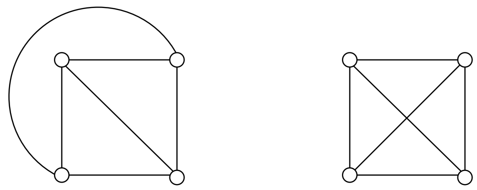
Índice

Lista de Figuras

Introdução

Um desenho de um grafo no plano tal que suas arestas se intersectam somente nos seus pontos extremos, caso exista, é chamado de desenho plano do grafo e um grafo para o qual existe um desenho plano é dito planar (veja figura 1).

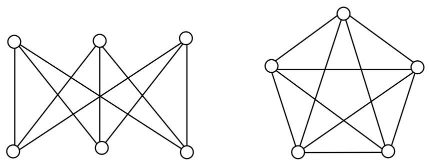
Existem situações práticas onde é necessário determinar se um dado grafo possui um desenho plano. Por exemplo, em projetos de circuitos VLSI, existe o interesse em obter-se um desenho desse tipo que representa um circuito elétrico.



(a) (b)

Figura 1: (a) Um desenho plano de um grafo planar. (b) Um desenho que não é plano do mesmo grafo.

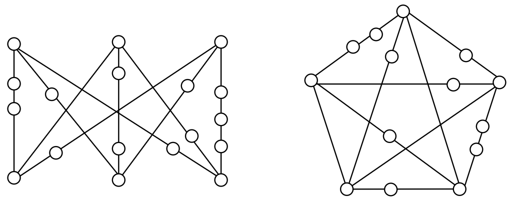
Os grafos da figura 2 não são planares.



(a) (b)

Figura 2: Exemplos de grafos não-planares. (a) $K_{3,3}$. (b) K_5.

Uma caracterização muito simples de grafos planares foi dada por Kuratowski [11]. Ele demonstrou que os grafos $K_{3,3}$ e K_5 são, basicamente, os únicos grafos não-planares: os grafos da figura 3 são chamados de grafos de Kuratowski e um grafo é planar se e somente se ele não contém um subgrafo de Kuratowski.



(a) (b)

Figura 3: Grafos de Kuratowski. (a) Cada aresta do $K_{3,3}$ é substituída por um caminho de comprimento pelo menos um. (b) O mesmo para o K_5.

Apesar de elegante, a caracterização de Kuratowski não fornece um algoritmo prático para o teste de planaridade. Ademais, não está claro como obter um desenho plano de um grafo planar através dessa caracterização.

Uma possível estratégia para decidir se um grafo G é planar é construir incrementalmente um desenho plano de G. Se for possível completar a construção de um desenho plano de G, então G é planar, caso contrário, idealmente, um subgrafo de Kuratowski é encontrado em G, justificando a não planaridade de G. O primeiro algoritmo incremental para teste de planaridade foi proposto por Auslander e Parter [1]. Dado um grafo G, primeiramente um circuito de G é encontrado. O grafo resultante da remoção desse circuito de G é composto de 'partes' conexas. O algoritmo é chamado recursivamente para obter um desenho plano de cada uma dessas partes junto com o circuito. Depois, esses desenhos planos são combinados, se possível, para obter um desenho plano de G. Se não for possível combinar esses desenhos planos, um subgrafo de Kuratowski é encontrado em G. O algoritmo de Auslander e Parter continha um erro que foi corrigido por Goldstein [9], resultando no algoritmo de Auslander, Parter e Goldstein (APG). A complexidade de tempo desse algoritmo é cúbica.

Lempel, Even e Cederbaum [12] (LEC) apresentaram uma maneira alternativa de se construir um desenho plano incrementalmente. Eles começam com o desenho de um único vértice e adicionam ao desenho todas as arestas incidentes a este vértice. Depois adicionam um novo vértice incidente a uma das arestas já inseridas e todas as arestas incidentes a este novo vértice. O processo continua dessa maneira. Cada iteração do algoritmo começa com um desenho plano de parte do grafo e consiste em adicionar um novo vértice e todas as arestas incidentes a este vértice ao desenho plano corrente. O processo continua até que todo o grafo tenha sido desenhado no plano ou tenha-se detectado que o grafo não é planar. Este tipo de algoritmo incremental é chamado de *vertex addition algorithm*. Tarjan [17] apresentou uma implementação do algoritmo de LEC que consome tempo quadrático.

Em 1974, Hopcroft e Tarjan [10] apresentaram o primeiro algoritmo linear para decidir se um dado grafo é planar. O algoritmo de Hopcroft e Tarjan é uma implementação envolvente do algoritmo de APG.

Hopcroft e Tarjan não deram muitos detalhes de como o seu algoritmo de teste de planaridade pode ser estendido para construir de fato um desenho plano do grafo dado. Uma descrição completa da fase de construção de um desenho plano por esse algoritmo foi feita por Mehlhorn e Mutzel [13] e a respectiva implementação por Mutzel [14]. Até hoje não se conhece uma implementação do algoritmo de Hopcroft e Tarjan que, em tempo linear, devolve um certificado de não planaridade, ou seja, um subgrafo de Kuratowski de um grafo não-planar.

Em 1976, Booth e Lueker [2] apresentaram uma implementação do algoritmo de LEC que consome tempo linear. Para isto, Booth e Lueker empregaram uma estrutura de dados chamada de PQ-árvore. Recentemente, Shih e Hsu [15] e Boyer e Myrvold [3] descreveram implementações similares para o algoritmo de LEC. Ambas gastam tempo linear e evitam o uso de PQ-árvores.

Os dois algoritmos lineares para teste de planaridade mais discutidos na literatura são os algoritmos de APG e o de LEC. Canfield e Williamson [5] fazem uma comparação entre esses dois algoritmos argumentando que os dois, geralmente vistos como métodos bem diferentes, têm na realidade muitas semelhanças.

Esta dissertação apresenta uma descrição do algoritmo de LEC e uma descrição da implementação proposta por Shih e Hsu [15]. Neste trabalho, a implementação de Shih e Hsu foi desenvolvida sobre a plataforma LEDA [7]. Além disso, é feito um estudo comparativo entre esta implementação, as duas implementações presentes no LEDA (uma implementação de Hopcroft e Tarjan [10] e uma implementação de Booth e Lueker [2]) e a implementação de Boyer e Myrvold [3].

A figura 4 mostra um resumo do consumo de tempo de algumas implementações dos principais algoritmos de planaridade existentes até hoje.

Organização da dissertação

O capítulo 1 contém a maior parte da notação, conceitos e definições usados ao longo desta dissertação. Em seguida, no capítulo 2, são apresentados alguns conceitos e resultados sobre planaridade que são fundamentais para a descrição do algoritmo de LEC [12], presente no capítulo 3, e para a descrição da implementação desse algoritmo proposta por Shih e Hsu [15], nos capítulos seguintes.

Algoritmo	Implementação	Complexidade
Auslander, Parter e Goldstein (APG)	original [1, 9]	cúbico
	Hopcroft e Tarjan [10]	linear
Lempel, Even e Cederbaum (LEC)	original [12]	quadrático
	Booth e Lueker [2]	linear
	Boyer e Myrvold [3]	linear
	Shih e Hsu [15]	linear

Figura 4: Tabela dos principais algoritmos de planaridade e algumas de suas implementações.

No capítulo 7 é apresentada uma análise experimental entre as implementações de Booth e Lueker (BL) [2], Hopcroft e Tarjan (HT) [10], Boyer e Myrvold (BM) [3] e de Shih e Hsu (SH) [15]. Finalmente, no capítulo 8, encontra-se um resumo da nossa experiência durante o desenvolvimento deste trabalho.

Código fonte da implementação de Shih e Hsu

A implementação de Shih e Hsu foi desenvolvida pelos autores dessa dissertação sobre a plataforma LEDA, versão `4.4`, e testada num computador pessoal com sistema GNU/Linux.

Para compilar o código fonte da implementação, é necessário possuir o LEDA instalado no seu sistema, além de um compilador `C++`.

A versão mais recente do LEDA para avaliação pode ser obtida pela internet, no endereço `http://www.algorithmic-solutions.com/` e o pacote compactado do código fonte das implementações de SH e de BM podem ser obtidos pela internet, no endereço `http://www.ime.usp.br/~noma/sh/planarity.tar.gz`.

Para descompactar o arquivo `planarity.tar.gz` no diretório `meudir/`, utilize o comando

```
meu_prompt> tar -zxvpf planarity.tar.gz -C meudir/
```

Após a descompactação, o diretório `meudir/planarity/` é criado. O código fonte da implementação de BM foi gentilmente cedido por John M. Boyer, cuja página pessoal se encontra no endereço `http://pacificcoast.net/~lightning/jboyer.htm`.

Para compilar os programas *planarity_ time* e *sh_planar*, basta executar

```
meu_prompt> make
```

planarity_time. Este programa imprime, na saída padrão, uma tabela de comparação de tempos, em formato LaTeX, entre as implementações de BL, HT, BM e de SH.

Código fonte da dissertação

Para compilar a dissertação, é necessário possuir o LEDA, o LaTeX e o `noweb` devidamente instalados no seu sistema. (O `noweb` é uma ferramenta para programação literária (*literate programming*).)

Uma versão do `noweb` pode ser obtida pela internet, no endereço `http://www.eecs.harvard.edu/~nr/noweb`.

O código fonte da dissertação pode ser obtido pela internet, no endereço `http://www.ime.usp.br/~noma/sh/dissertacao.tar.gz`.

Para descompactar o arquivo `dissertação.tar.gz` no diretório `meudir/`, utilize o comando

```
meu_prompt> tar -zxvpf dissertacao.tar.gz -C meudir/
```

Após a descompactação, o diretório `meudir/dissertacao/` é criado.

Para compilar a dissertação, execute

```
meu_prompt> ./mkdissertacao
```

O arquivo `dissertacao.ps.gz` é criado.

Capítulo 1

Preliminares

Neste capítulo é apresentada a maior parte da notação e definições usadas ao longo desta dissertação.

1.1 Teoria dos grafos

Esta seção introduz os conceitos de grafos, conexidade, planaridade e outros elementos básicos da teoria dos grafos.

Grafos orientados, subgrafos orientados e grafos

Um **grafo orientado** é um objeto da forma (V, A), onde V é um conjunto finito de elementos, chamados **vértices**, e A é um conjunto de pares ordenados de elementos distintos de V, chamados **arcos**. Se $V = \emptyset$ então o grafo é **vazio**.

Se G é um grafo orientado, o conjunto dos vértices de G é denotado por V_G e o conjunto dos arcos de G por A_G. V_G A_G

Um arco (u, v) é denotado por uv. Diz-se que vu é o **reverso** de uv. Se uv é um arco, então u e v são as pontas **inicial** e **final** de uv, respectivamente. Diz-se também que u e v são **vizinhos** e que uv **sai** de u e **entra** em v. Se vu é um arco, então diz-se que uv **tem reverso**.

Diz-se que H é um **subgrafo orientado** de G se H é um grafo orientado tal que $V_H \subseteq V_G$ e $A_H \subseteq A_G$. Se H contém todos os arcos de G com ambas as pontas em V_H então H é um **subgrafo induzido** de G. Neste caso, denota-se H por $G[V_H]$ e diz-se que H é o **subgrafo de G induzido por V_H**. $G[V_H]$

Se $V \subseteq V_G$ então $G - V$ é o subgrafo orientado de G resultante da remoção de todos os vértices de V e de todos os arcos com pelo menos uma ponta em V. Ou seja, $G - V$ é o grafo $G[V_G \setminus V]$. $G - V$

Seja H um subgrafo induzido de G e w um vértice de G. Denota-se por $H + w$ o grafo $G[V_H \cup \{w\}]$. Similarmente, denota-se por $H - w$ o grafo $G[V_H \setminus \{w\}]$.

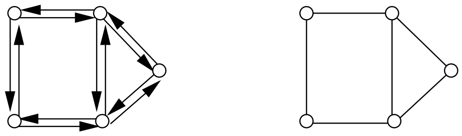
Um **grafo** (simétrico) é um grafo orientado em que todo arco tem reverso. Num grafo, um conjunto $\{uv, vu\}$ é chamado de **aresta**. Abusando da notação, uma aresta $\{uv, vu\}$ é denotada simplesmente por uv ou vu. Dada uma aresta uv, diz-se que uv é **incidente** a u e a v e que u e v são as **pontas** de uv. Além disso, u e v são ditos **adjacentes**.

Se u é um vértice, então o **grau** de u é o número total de arestas incidentes a u.

E_G Se G é um grafo, E_G é o conjunto das arestas de G e os números $|V_G|$ e $|E_G|$ são denotados, respectivamente, por n e m. O **tamanho de** G é o número $n + 2m$.

n m

Um grafo pode ser naturalmente representado através de um diagrama como o da figura 1.1(a), onde os vértices são pequenos círculos, os arcos são setas ligando os dois círculos que correspondem as suas pontas. Geralmente esse diagrama é simplificado para aquele representado na figura 1.1(b), onde cada arco, junto com seu reverso, é substituído por uma linha simples.



(a) (b)

Figura 1.1: (a) Diagrama de um grafo. (b) Versão simplificada do diagrama.

Passeios, ciclos, caminhos e circuitos

Um **passeio** num grafo orientado é uma seqüência

$$\langle v_0, a_1, v_1, \ldots, v_{k-1}, a_k, v_k \rangle$$

onde $v_0, v_1, \ldots, v_k$ são vértices e, para cada i, a_i é o arco $v_{i-1}v_i$. Diz-se que o passeio é entre v_0 e v_k, ou que começa em v_0 e termina em v_k e que v_0 e v_k são as **pontas** do passeio. Tal passeio é muitas vezes denotado por $\langle v_0, v_1, \ldots, v_{k-1}, v_k \rangle$. O comprimento de um passeio P é o número $|P| = k$ e os vértices $v_1, \ldots, v_{k-1}$ são os vértices **internos** do passeio. Um **ciclo** num grafo G é um subgrafo orientado de G formado pelos vértices e arcos de um passeio que começa e termina num mesmo vértice. Um ciclo determinado pelo passeio $\langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$ é denotado por $\langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$. Um **caminho** é um passeio sem repetição de vértices. Um **circuito** é um passeio $\langle v_0, v_1, \ldots, v_{k-1}, v_k \rangle$ de comprimento pelo menos três em que $v_0, v_1, \ldots, v_{k-1}$ são vértices distintos e $v_k = v_0$.

As vezes confundimos um grafo ou um passeio com seu conjunto de arestas. Seja P um caminho tal que E_P é vazio. Então P é dito **trivial**.

Conexidade, blocos e grafos bipartidos

Um grafo é **conexo** se, para todo par de vértices u e v do grafo, existe um caminho entre u e v. Um **componente** de um grafo é um subgrafo conexo maximal. O grafo vazio é conexo.

Um **vértice de corte** de um grafo G é um vértice v tal que $G - v$ tem mais componentes que G. Um grafo é **biconexo** se é conexo e não possui vértices de corte. Um subgrafo biconexo maximal é chamado de **bloco**.

Um grafo G é **bipartido** se V_G pode ser particionado em dois conjuntos R e S tais que, para cada aresta uv de G, se u é um vértice de R então v está em S, e se u é um vértice de S então v está em R.

Árvores, árvores enraizadas e busca em profundidade

Uma **árvore** é um grafo conexo sem circuitos. Numa árvore, existe um único caminho entre de qualquer vértice a qualquer outro.

Uma árvore com um vértice especial r é dita **enraizada**. O vértice r é chamado de **raiz**.

Seja v um vértice de uma árvore T com raiz r. Qualquer vértice u no caminho de r a v em T é chamado de **ancestral** de v em T. Um ancestral u de v é **próprio** se $u \neq v$. Se u é um ancestral de v, então v é um **descendente** de u. Um descendente v de u é **próprio** se $v \neq u$. Se $v \neq r$ e uv é a última aresta do caminho de r a v, então u é o **pai** de v e v é um **filho** de u em T. Dois vértices com o mesmo pai são chamados de **irmãos**.

Uma **árvore geradora** de um grafo conexo G é um subgrafo de G que é uma árvore e contém todos os vértices de G.

A subárvore de T com raiz u cujos vértices são todos os descendentes de u em T é denotada por T_u. T_u

Uma **floresta** é um grafo tal que cada um de seus componentes é uma árvore.

Uma **bp-árvore** (ou uma **árvore de busca em profundidade**) de um grafo conexo G é uma árvore geradora enraizada de G tal que, para cada aresta uv em G, ou u é descendente de v na árvore ou v é descendente de u (figura 1.2(b)).

Seja T uma bp-árvore de G. Uma ordenação $v_1, v_2, \ldots, v_n$ dos vértices de G tal que

$$\text{se } v_i \text{ é ancestral de } v_j \text{ em } T \text{ então } i > j$$

é dita uma **bp-ordenação** de G (em relação a T). Veja a figura 1.2. O **bp-número** de um

bp(u) vértice u (em relação a uma bp-ordenação), denotado por bp(u), é o índice i tal que $u = v_i$. Note que se r é a raiz de T então bp(r) $= n$.

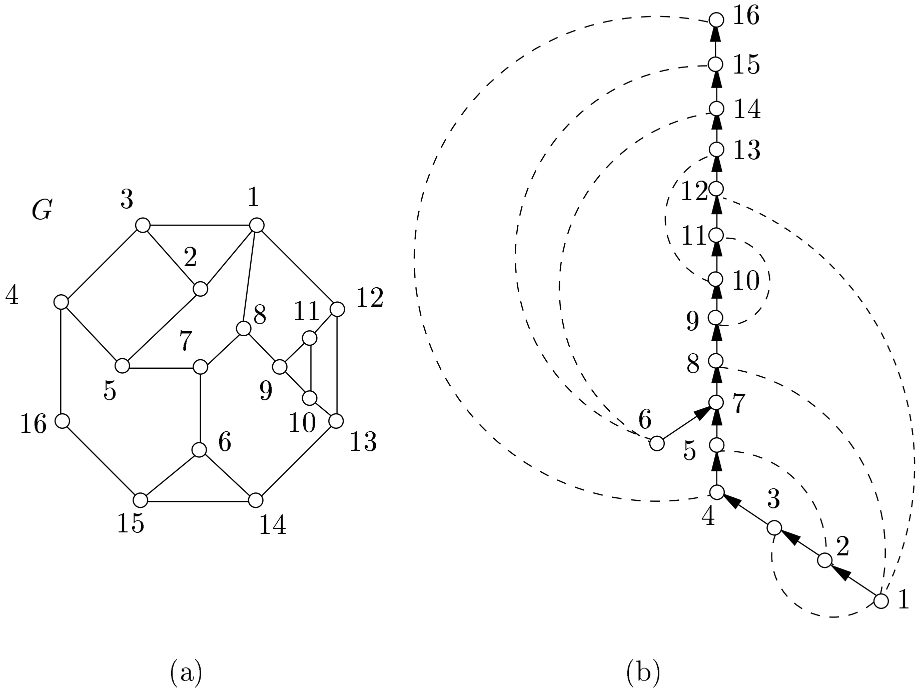


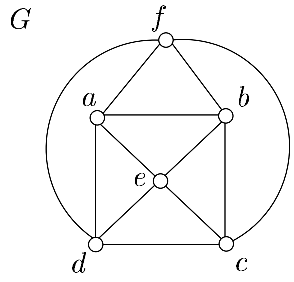
Figura 1.2: (a) Um grafo G com vértices numerados de acordo com uma bp-ordenação. (b) Uma bp-árvore T de G, indicada por linhas sólidas, enraizada no vértice 16. As setas apontam para o pai de cada nó em T.

Representação de grafos em um computador

Um grafo pode ser representado em um computador pelas suas **listas de adjacências**. A

Adj(u) lista de adjacência de um vértice u, denotada por $Adj(u)$, consiste de uma lista que contém todos os arcos com ponta inicial u (figura 1.3(a)). O espaço consumido por esta representação é $O(n+m)$.

Note que as listas de adjacências de um grafo podem ser representadas por uma lista única de arcos (figura 1.3(b)) e pode-se converter de uma representação para a outra em tempo $O(n+m)$.



Listas de adjacências:

$\text{Adj}(a) = \langle af, ab, ae, ad\rangle$
$\text{Adj}(b) = \langle bc, be, ba, bf\rangle$
$\text{Adj}(c) = \langle cd, ce, cb, cf\rangle$
$\text{Adj}(d) = \langle df, da, de, dc\rangle$
$\text{Adj}(e) = \langle ea, eb, ec, ed\rangle$
$\text{Adj}(f) = \langle fc, fb, fa, fd\rangle$

(a)

Representação por uma lista:

$\langle af, bc, be, ab, ba, fc, bf, cd, ce, cb, cf, ae, df, da, de, dc, ea, eb, ec, ad, ed, fb, fa, fd\rangle$

(b)

Figura 1.3: (a) Um grafo e uma representação deste por listas de adjacências. (b) Representação das listas de adjacências por uma única lista de pares de vértices.

1.2 LEDA

LEDA (*Library of Efficient Data Types and Algorithms* [7]) é uma plataforma para algoritmos combinatórios desenvolvida, inicialmente, por Näher e Mehlhorn. Ela foi utilizada para a implementação de Shih e Hsu descrita neste trabalho.

Abaixo encontra-se uma breve apresentação de como representar grafos orientados, vértices e arcos no LEDA e os comandos usados para criar e percorrer vértices e os arcos de um dado grafo. São apresentadas também algumas estruturas de dados do LEDA que são utilizadas neste trabalho.

Grafos orientados, vértices e arcos

Um grafo orientado é representado no LEDA pelo tipo **graph**, um vértice pelo tipo **node** e um arco pelo tipo **edge** (figura 1.4).

As funções *source(uv)* e *target(uv)* devolvem, respectivamente, as pontas inicial e final de um arco uv. Ambas gastam tempo constante.

O número de vértices e de arcos de um grafo são obtidos, respectivamente, pelas funções *number_of_nodes* e *number_of_edges*. Ambas consomem tempo constante.

12a ⟨declarações 12a⟩≡ (14)

graph G;
node u, v;
edge uv;

Figura 1.4: A variável G representa um grafo orientado vazio. As variáveis u, v são apontadores para vértices e uv é um apontador para um arco. Inicialmente, u, v e uv estão com valores indefinidos.

Manipulação de grafos

Para adicionar um vértice e um arco a um grafo G, usa-se, respectivamente, as funções *new_node* e *new_edge* (figura 1.5). Para remover um vértice e um arco de G, usa-se, respectivamente, as funções *del_node* e *del_edge*. Estas quatro funções consomem tempo constante.

12b ⟨construção de um grafo 12b⟩≡ (14)

$u = G.new\_node()$;
$v = G.new\_node()$;
$uv = G.new\_edge(u, v)$;
$G.new\_edge(G.new\_node(), u)$;

Figura 1.5: Construção de um grafo no LEDA.

Para percorrer todos os vértices e arcos de G, usa-se, respectivamente, as macros *forall_nodes* e *forall_edges* (figura 1.6). Elas gastam tempo proporcional a $|V_G|$ e a $|E_G|$, respectivamente.

Listas e vetores

Para representar uma lista de elementos de um mesmo tipo, usa-se o tipo **list** e para representar um vetor de elementos, usa-se o tipo **array**. Por exemplo, **list**<**edges**> é usado para declarar uma lista de arestas.

Para inserir um elemento no início de uma lista, usa-se a função *push* e para inserir um elemento no fim de uma lista, usa-se a função *append*. Um apontador para um elemento de uma lista é do tipo **item_list**. Dado um apontador para um elemento de uma lista, usa-se a função *del* para remover o elemento indicado pelo apontador.

Para percorrer todos os elementos de uma lista, usa-se a macro *forall*.

bmlist

O tipo **bmlist**, proposto por Boyer e Myrvold [3], consiste de uma lista que pode ser invertida em tempo constante. Nela, cada célula possui dois apontadores (campos *ptr1* e *ptr2*) e, para passar para o próximo elemento na lista, olha-se os dois apontadores e escolhe-se o apontador que não leva ao elemento anteriormente percorrido. Assim, para inverter essa lista, basta trocar as células que são os extremos da lista.

Para percorrer todos os elementos de uma **bmlist**, foi criado a macro *myforall*.

Associando informações a vértices e arcos de um grafo

Para atribuir informações a todos os vértices e arcos de um grafo G, usa-se vetores indexados por vértices e arcos, respectivamente (figura 1.6). Os vetores indexados por vértices são os tipos **node_array** e **node_map**. A diferença entre estes é que o primeiro é usado para grafos estáticos e o segundo para grafos dinâmicos. Analogamente, os vetores indexados por arcos são os tipos **edge_array** e **edge_map**.

Os tipos **map**<**node**,. . . > e **map**<**edge**,. . . > podem ser usados nos lugares de **node_map** e **edge_map**, respectivamente. O tipo **map** é útil quando deseja-se associar informações a um pequeno número de vértices ou arcos do grafo.

O tempo gasto para acessar cada elemento de um vetor **node_array** ou **edge_array** é constante, enquanto que o tempo gasto esperado para os tipos **map**, **node_map** e **edge_map** é constante.

13 ⟨enumeração dos vértices e arcos 13⟩≡ (14)

```
node_array<int> nodenumber(G);
edge_array<int> edgenumber(G);
int i = 0;
node w;
forall_nodes(w, G) nodenumber[w] = ++i;
i = 0;
edge e;
forall_edges(e, G) edgenumber[e] = ++i;
```

Figura 1.6: Enumeração dos vértices e arcos do grafo construído na figura 1.5.

Estrutura de um programa

Um exemplo de um programa em LEDA é ilustrado na figura 1.7.

Dados dois elementos a e b, a função *leda_swap(a,b)* troca o conteúdo de a pelo conteúdo de b e vice-versa.

Dados dois arcos a e b, a função *set_reversal(a,b)* é usada para guardar a informação de que a é o reverso de b e vice-versa. Dado um arco, o seu reverso (caso exista tal informação) é devolvido pelas funções *reverse* e *reversal*.

14 ⟨Programa 14⟩≡

```
#include <LEDA/graph.h>
#include <LEDA/graph_misc.h>
#include <stdio.h>
#if defined(LEDA_NAMESPACE)
using namespace leda;
#endif
int main() {
  ⟨declarações 12a⟩
  ⟨construção de um grafo 12b⟩
  ⟨enumeração dos vértices e arcos 13⟩
  cout≪"u ="≪nodenumber[u]≪",v ="≪nodenumber[v]≪",uv ="≪edgenumber[uv]≪"\n";
  return 0;
}
```

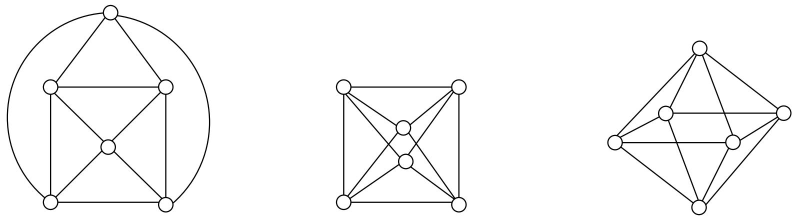
Figura 1.7: Exemplo de um programa em LEDA. Como saída, o programa imprime $u = 1, v = 2, uv = 1$.

Planaridade

Neste capítulo, são apresentados alguns ingredientes básicos que envolvem planaridade. Conceitos topológicos de planaridade tais como curvas e desenhos são tratados de maneira superficial. A intenção é trocar, o mais rapidamente possível, esses elementos topológicos por objetos combinatórios que, do ponto de vista algorítmico deste trabalho, são de fundamental importância. Assim, este capítulo pode ser encarado como uma (rápida) transição da intuição topológica para a combinatória de planaridade e estruturas de dados.

2.1 Desenhos, faces, grafos planares e grafos planos

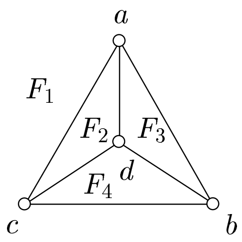
Um **desenho** de um grafo G é uma função I que associa a cada vértice v de G um ponto distinto $I(v)$ no plano e associa a cada aresta uv uma curva $I(uv)$, sem auto-intersecção, de $I(u)$ a $I(v)$ no plano, i.e., $I(uv)$ é uma função injetora contínua do intervalo $[0,1]$ no plano tal que $I(uv)(0) = I(u)$ e $I(uv)(1) = I(v)$. Na figura 2.1 são apresentados três desenhos diferentes de um mesmo grafo.



(a) (b) (c)

Figura 2.1: Três desenhos diferentes de um mesmo grafo.

Um desenho de um grafo decompõe o plano em regiões formadas por pontos que podem ser ligados por curvas que não intersectam a imagem do desenho. Essas regiões são as **faces** do desenho. A região ilimitada é chamada de **face externa** (figura 2.2).



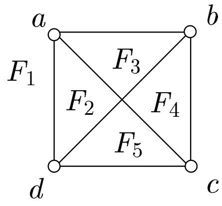


Figura 2.2: Dois desenhos diferentes de um mesmo grafo e suas respectivas faces. Em (a) e (b), F_1 é a face externa.

Um desenho I de um grafo G é **plano** se

(a) a imagem das arestas não contêm imagens dos vértices no seu interior, ou seja, $I(uv)(x) \neq I(w)$ para toda aresta uv, vértice w e número real x no intervalo $(0, 1)$;

(b) as imagens de arestas distintas não se intersectam no seu interior, ou seja, $I(uv)(x) \neq I(st)(y)$ para quaisquer arestas distintas uv e st e números reais x e y no intervalo $(0, 1)$.

Na figura 2.1 somente (a) é um desenho plano. Um grafo G é dito **planar** se ele admite um desenho plano. O grafo vazio é planar. Um grafo planar junto com um de seus desenhos planos é chamado de **grafo plano**. Neste trabalho, quando dizemos que G é um grafo plano, estamos implicitamente considerando um desenho plano de G. Assim, por exemplo, faz sentido falar-se no número de faces de um grafo plano.

2.2 Fórmula de Euler

Uma fórmula simples relacionando o número de vértices, de arestas e de faces de um grafo plano conexo foi dada por Euler.

Teorema 2.1 (fórmula de Euler): *Se G é um grafo plano conexo com n vértices, m arestas e f faces, então*

$$n - m + f = 2.$$

Demonstração: Fixamos n e aplicamos indução em m. Se G é uma árvore, então $m = n - 1$, $f = 1$ e a fórmula é válida. Agora suponha que G não é uma árvore. Então G possui uma aresta e

tal que $G' := G - e$ é um grafo conexo e, evidentemente, plano. Sejam n', m' e f' o número de vértices, arestas e faces de G'. É claro que $n' = n$, $m' = m - 1$ e $f' = f - 1$. Finalmente, por indução, temos que

$$2 = n' - m' + f' = n - (m - 1) + (f - 1) = n - m + f.$$

∎

Do teorema acima seguem alguns corolários. Cada um deles fornece uma delimitação superior para o número de arestas de um grafo planar.

Corolário 2.2: *Se G é um grafo planar com m arestas, n vértices, $n \geq 3$, então*

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

Demonstração: Podemos supor que se u e v são vértices de G tais que uv não é aresta de G, então $G + uv$ é não-planar, já que esta operação faz crescer o lado esquerdo da desigualdade acima e mantém constante o lado direito. É claro que G é conexo. Considere um desenho plano de G e seja f o número de faces desse desenho. Como $n \geq 3$, então cada face do desenho possui exatamente três arestas em sua fronteira. Logo, $3f = 2m$. Pela fórmula de Euler tem-se que

$$2 = n - m + f = n - m + (2/3)m = n - (1/3)m,$$

donde

$$m = 3n - 6.$$

∎

O menor comprimento de um circuito num grafo é a sua **cintura**. Se o grafo não possui circuito algum, então diz-se que a sua cintura é "infinita". O corolário abaixo pode ser visto como uma certa generalização do corolário 2.2.

Corolário 2.3: *Se G é um grafo com m arestas, n vértices, e cintura (finita) k, $k \geq 3$, então*

$$m \leq (k/(k-2))(n-2).$$

Demonstração: Podemos supor que se u e v são vértices de G tais que uv não é aresta de G, então $G + uv$ é não-planar ou tem cintura menor que k. É claro que G é conexo. Considere um desenho plano de G e seja f o número de faces desse desenho. Como G tem cintura k, então cada face do desenho possui pelo menos k arestas em sua fronteira. Logo, $kf \leq 2m$. Pela fórmula de Euler tem-se que

$$2 = n - m + f \leq n - m + (2/k)m = n - ((k-2)/k)m,$$

donde

$$m \leq (k/(k-2))(n-2).$$

∎

2.3 Grafos de Kuratowski

Os corolários no final da seção anterior podem ser usados para verificar que os grafos da figura 2.3 são não-planares. Os grafos das figuras 2.3(a) e (b) são denotadas por K_5 e $K_{3,3}$, respectivamente.

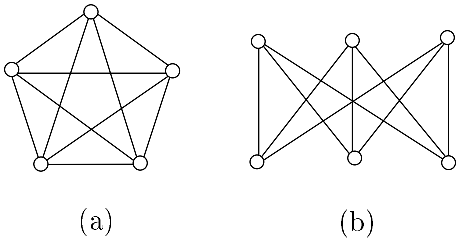


Figura 2.3: (a) K_5. (b) $K_{3,3}$.

Corolário 2.4: *K_5 é não-planar.*

Demonstração: Suponha por contradição que K_5 é planar. Então, pelo corolário 2.2, tem-se que $m = 10 \leq 3 \times 5 - 6 = 9$, uma contradição. Portanto K_5 não pode ser planar. ∎

Corolário 2.5: *$K_{3,3}$ é não-planar.*

Demonstração: Suponha por contradição que $K_{3,3}$ é planar. Então, pelo corolário 2.3, como $K_{3,3}$ tem cintura 4, tem-se que $m = 9 \leq 2 \times 6 - 4 = 8$, uma contradição. Portanto $K_{3,3}$ não pode ser planar. ∎

Subdividir uma aresta uv de um grafo G significa substituir uv por um caminho de comprimento dois com pontas u e v (figura 2.4). Uma **subdivisão** de um grafo G é um grafo que pode ser obtido a partir de G por uma seqüência finita de subdivisões de arestas. Um **grafo de Kuratowski** é qualquer subdivisão do grafo K_5 ou do grafo $K_{3,3}$.

O lema a seguir será utilizado, no capítulo 3, para obter uma caracterização de planaridade, conhecida por Teorema de Kuratowski. A essência desse teorema diz que a recíproca do lema abaixo é verdadeira.

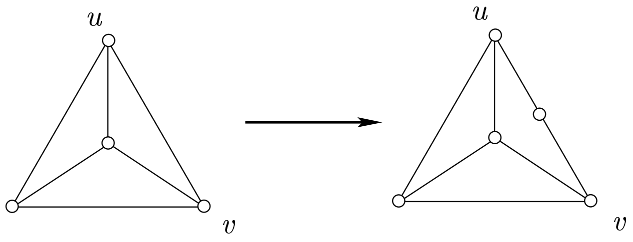


Figura 2.4: Subdivisão da aresta uv.

Lema 2.6: *Se G é um grafo que possui um subgrafo de Kuratowski, então G é não-planar.*

Demonstração: É claro que se G possui um subgrafo que não é planar, então G não é planar. Ademais, toda subdivisão de um grafo que não é planar também não é planar. Portanto o lema segue dos corolários 2.4 e 2.5. ∎

2.4 Descrições combinatórias e ciclos faciais

Convém lembrar que neste trabalho um grafo (simétrico) é um grafo orientado tal que uv é um arco do grafo se e somente se o seu reverso vu é um arco do grafo. Em um grafo uma aresta uv é um conjunto de arcos $\{uv, vu\}$. Assim, dependendo do contexto, algumas vezes é conveniente nos referirmos a aresta uv e em outras oportunidades ao arco uv.

Um desenho de um grafo induz[1] uma ordem cíclica nos arcos com ponta inicial em cada vértice (figura 2.5). Essa ordem pode ser horária ou anti-horária. Usaremos sempre a ordem horária. Uma lista formada por todas essas ordens cíclicas de arcos é chamada de uma **descrição combinatória** do desenho. Mais precisamente, uma lista D é uma descrição combinatória de um desenho se, para cada vértice v, $D(v)$ é a ordem cíclica de todos os arcos com ponta inicial em v, induzida pelo desenho. Cada ordem cíclica é representada por uma lista de arcos.

É evidente que uma descrição combinatória pode corresponder a uma infinidade de desenhos. Note que as listas de adjacências de um grafo formam uma descrição combinatória de um desenho do grafo. Compare as figuras 2.6(a) e 1.3 da página 11.

Seja D uma descrição combinatória de um desenho de um grafo. Um **ciclo facial** de D é um ciclo $\langle v_0, v_1, \ldots, v_{k-1}, v_0\rangle$ onde, para cada i, v_iv_{i+1} é o sucessor cíclico de v_iv_{i-1} em $D(v_i)$, considerando-se os índices módulo k (figura 2.7). Os ciclos faciais C_1, C_2, C_3 e C_4 da figura 2.7(a) correspondem as faces F_1, F_2, F_3 e F_4 da figura 2.2, respectivamente.

[1] Esta ordem é induzida numa "vizinhança suficientemente pequena" de um vértice.

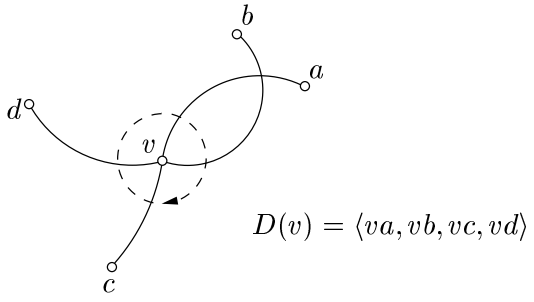
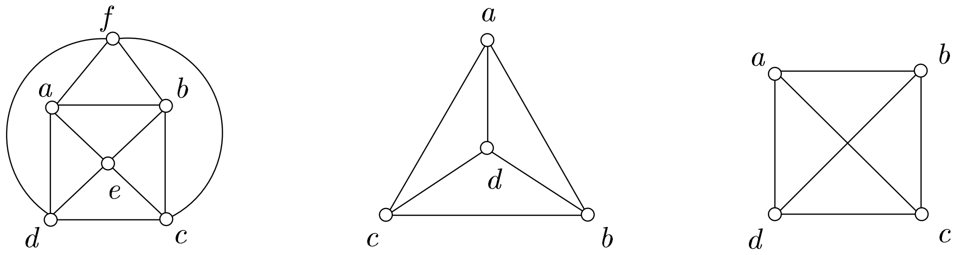


Figura 2.5: Ordem cíclica dos arcos com ponta inicial v induzida por um desenho. A ordem cíclica $D(v)$ é representada pela lista va, vb, vc, vd.



Descrição combinatória D_1:
$D_1(a) = \langle af, ab, ae, ad \rangle$
$D_1(b) = \langle bc, be, ba, bf \rangle$
$D_1(c) = \langle cd, ce, cb, cf \rangle$
$D_1(d) = \langle df, da, de, dc \rangle$
$D_1(e) = \langle ea, eb, ec, ed \rangle$
$D_1(f) = \langle fc, fb, fa, fd \rangle$

(a)

Descrição combinatória D_2:
$D_2(a) = \langle ab, ad, ac \rangle$
$D_2(b) = \langle bc, bd, ba \rangle$
$D_2(c) = \langle ca, cd, cb \rangle$
$D_2(d) = \langle da, db, dc \rangle$

(b)

Descrição combinatória D_3:
$D_3(a) = \langle ab, ac, ad \rangle$
$D_3(b) = \langle bc, bd, ba \rangle$
$D_3(c) = \langle cd, ca, cb \rangle$
$D_3(d) = \langle da, db, dc \rangle$

(c)

Figura 2.6: (a) Um desenho plano e uma descrição combinatória induzida pelo desenho. Em (b) e (c), são ilustrados dois desenhos de um mesmo grafo: um plano e um não-plano, respectivamente.

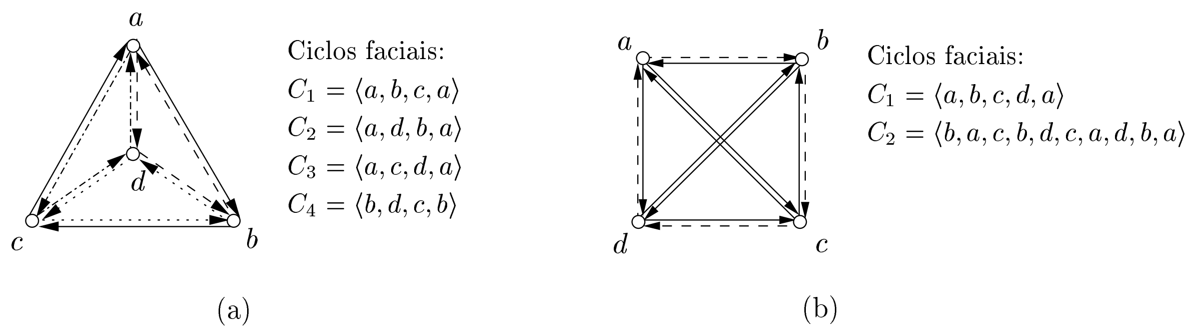


Figura 2.7: (a) Ciclos faciais determinados pela descrição combinatória da figura 2.6(b). (c) Ciclos faciais determinados pela descrição combinatória da figura 2.6(c).

Uma descrição combinatória de um desenho plano é dita **plana**. É claro que todo grafo planar possui uma descrição combinatória plana. Nas figuras 2.6(a) e (b) encontram-se dois desenhos planos e suas respectivas descrições combinatória planas.

No contexto deste trabalho, descrições combinatórias planas têm um papel fundamental. É fácil ver que que toda descrição combinatória de uma árvore é plana. Note ainda que se D é uma descrição combinatória plana de um grafo com pelo menos uma aresta, então existe uma correspondência bijetora entre ciclos faciais de D e faces do desenho: cada ciclo facial "corresponde" a fronteira de uma face[2]. Estas observações são usadas na próxima seção.

2.5 Critério combinatório de planaridade

Nesta seção é apresentado um critério combinatório para testar se uma dada descrição combinatória é ou não plana. Este critério é facilmente transformado num trecho de código usado para verificar a correção das respostas fornecida pelas implementações feitas neste trabalho.

Teorema 2.7: *Seja G um grafo conexo com pelo menos uma aresta. Uma descrição combinatória D de um desenho de G é plana se e somente se*

$$n - m + f_D = 2 \tag{2.1}$$

onde f_D é o número de ciclos faciais de D.

Demonstração: Suponha que D é uma descrição combinatória de um desenho plano de G. Não é difícil ver que existe uma correspondência bijetora entre os ciclos faciais de D e os ciclos na fronteira das faces do desenho. Logo, como G tem pelo menos uma aresta e f_D é igual ao número de faces do desenho, da fórmula de Euler, segue que (2.1) vale.

[2]É óbvio que o nome "ciclo facial" se deva a esta correspondência.

Reciprocamente, é demonstrado a seguir, por indução em f_D, que se D é uma descrição combinatória que satisfaz (2.1), então ela D é plana.

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

Suponha agora que $f_D \geq 2$. Afirmamos que existem arcos uv e vu de G que não pertencem a um mesmo ciclo facial. De fato, suponha que cada par uv e vu de arcos de G pertencem a um mesmo ciclo facial e considere um vértice v de G. Seja $D(v) = \langle vv_0, vv_1, \ldots, vv_k \rangle$ a ordem cíclica dos arcos com ponta inicial em v (figura 2.8). Como vv_{i+1} é o sucessor cíclico de vv_i em $D(v)$ e v_iv e vv_i pertencem a um mesmo ciclo facial ($i = 0, 1, \ldots, k$), então todos os arcos com uma ponta em v pertencem a um mesmo ciclo facial . Como G é conexo, então todos os arcos de G pertencem a um mesmo ciclo facial. Portanto, $f_D = 1$, uma contradição.

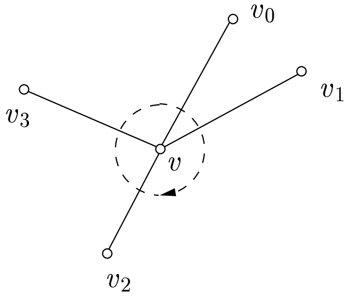


Figura 2.8: Todos os arcos com uma ponta em v estão num mesmo ciclo facial.

Sejam uv e vu arcos de G que não pertencem a um mesmo ciclo facial D. Sejam

$$C_{uv} := \langle v = w_0, w_1, \ldots, w_{t-1} = u, v = w_0 \rangle \quad \text{e} \quad C_{vu} := \langle u = z_0, z_1, \ldots, z_{l-1} = v, u = z_0 \rangle$$

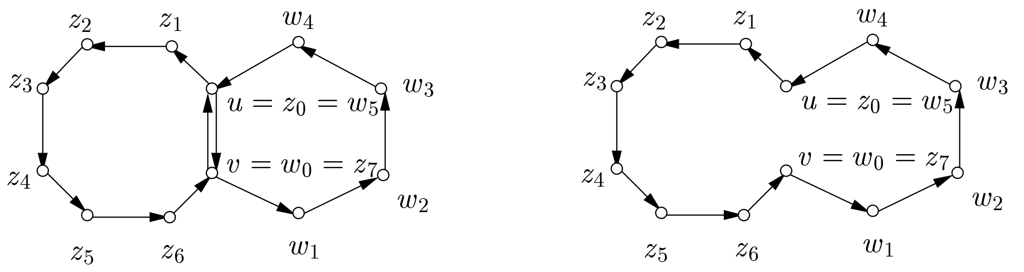
os ciclos faciais de D contendo uv e vu, respectivamente (figura 2.9).

Devido a escolha de uv e vu, tem-se que $G' := G - \{uv, vu\}$ é conexo e tem pelo menos uma aresta. Seja D' a descrição combinatória de G' induzida por D. Em D', os ciclos faciais C_{uv} e C_{vu} deram lugar ao ciclo facial

$$C' = \langle u = z_0 = w_{t-1}, z_1, \ldots, z_{l-1} = v = w_0, w_1, \ldots, w_{t-1} = z_0 = u \rangle$$

e os demais ciclos faciais permanecem os mesmos. Logo, $f_{D'} = f_D - 1$ e, por indução, D' é uma descrição combinatória de um desenho plano I' de G'.

Como existe uma correspondência bijetora entre ciclos faciais de D' e faces de I', então C' corresponde a uma face F' de I'. Pode-se estender o desenho plano I' de G' a um desenho plano I de G fazendo com que a imagem da aresta uv por I esteja contida em F'. Tem-se que D é descrição combinatória de I e, portanto, é plana. ∎



$C_{uv} = \langle v = w_0, w_1, w_2, w_3, w_4, w_5 = u \rangle$
$C_{vu} = \langle u = z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7 = v \rangle$

(a)

$C = \langle u = z_0 = w_5, z_1, z_2, z_3, z_4, z_5, z_6, v =$
$= z_7 = w_0, w_1, w_2, w_3, w_4, u = w_5 = z_0 \rangle$

(b)

Figura 2.9: Dois ciclos faciais em (a) que se fundem em (b) após a remoção da aresta uv.

A função na figura 2.10 traduz o critério de planaridade para a linguagem da plataforma LEDA. O código supõe que o grafo dado é conexo e tem pelo menos uma aresta. Essa função devolve verdadeiro se a descrição combinatória é plana e falso caso contrário. (Uma descrição combinatória é dada pela representação do grafo no LEDA. O sucessor cíclico de um arco e, no ciclo facial que contém e, num grafo G é acessado pela função *G.face_cycle_succ(e)*.)

2.6 Formulação dos problemas

Há dois tipos de problemas relacionados a planaridade tratados neste trabalho. O mais simples é o de teste de planaridade.

Problema do teste de planaridade: Dado um grafo G, decidir se G é planar.

O problema acima pede por uma simples resposta SIM ou NÃO:

- SIM, o grafo dado possui uma descrição combinatória plana;
- NÃO, o grafo dado não possui uma descrição combinatória plana.

Este problema, por ser mais simples, é o primeiro a ser tratado. Um algoritmo de teste de planaridade é apresentado no capítulo 3 e uma implementação é discutida no capítulo 4.

Os capítulos 3 e 4 servirão de passos intermediários no caminho para entender e implementar um algoritmo para o problema da planaridade.

24 ⟨Critério combinatório 24⟩≡

```
bool sh_planar_impl::check_embedding(graph& G)
{ edge_array<bool> considered(G, false);
  int f_D = 0;
  edge e;
  forall_edges(e,G) {
     if (¬considered[e]) {
        edge e1 = e;
        do { considered[e1] = true; e1 = G.face_cycle_succ(e1); } while (e1 ≠ e);
        f_D++;
      }
   } // forall
  int m = G.number_of_edges() ÷ 2,  // número de arestas de G;
    n = G.number_of_nodes();      // número de vértices de G;
  if (f_D ≠ m - n + 2) return false;
  return true;
  }
```

Figura 2.10: Critério combinatório para verificação de uma descrição combinatória plana. Recebe um grafo conexo G com pelo menos uma aresta e uma descrição combinatória D implícita nas listas de adjacência na representação de G. Devolve *true* se D é plana e *false* caso contrário.

Problema da planaridade: Dado um grafo G, determinar uma descrição combinatória plana de G ou um certificado de não-planaridade de G, ou seja, um subgrafo de Kuratowski em G.

A palavra certificado é usada de maneira vaga na descrição do problema acima, apesar de possuir um significado técnico bem preciso. Para os propósitos deste trabalho, basta saber que certificados típicos são subgrafos de Kuratowski, como mostra o capítulo 3.

Nos capítulos 5 e 6, o algoritmo de teste de planaridade (capítulo 3) e sua implementação (capítulo 4) são estendidos para resolver o problema da planaridade em tempo linear.

Algoritmo de Lempel, Even e Cederbaum

Neste capítulo é descrito o algoritmo de Lempel, Even e Cederbaum [12] (LEC) que decide se um dado grafo é planar. O algoritmo de LEC examina cada vértice do grafo dado, um após o outro, em uma certa ordem chamada de st-ordenação (*st-numbering*). Em cada iteração, o algoritmo procura acrescentar o vértice sendo examinado ao desenho plano do subgrafo induzido pelos vértices já examinados.

A descrição original do algoritmo de LEC é feita através do conceito da chamada forma de bush (*bush form*). A forma de bush pode ser entendida como uma estrutura de dados que mantém toda a informação necessária para estender o desenho plano corrente a um desenho plano do grafo dado, se possível. Tarjan [17] apresentou uma implementação do algoritmo de LEC que consome tempo quadrático.

Em 1976, Booth e Lueker [2] empregaram a chamada PQ-árvore (PQ-*tree*) para representar uma forma de bush. A implementação resultante consome tempo linear.

Mais recentemente, Shih e Hsu [15] e Boyer e Myrvold [3] apresentaram novas implementações do algoritmo de LEC que consomem tempo linear. Shih e Hsu utilizaram uma estrutura de dados que, devido a sua semelhança com as PQ-árvores, foi batizada de PC-árvore. Boyer e Myrvold utilizaram uma estrutura de dados diferente das PQ-árvores.

A descrição do algoritmo de LEC neste capítulo não é feita em termos de formas de bush. Em vez disso, a descrição utiliza o conceito de molduras (*frames*), que foi proposto por Thomas [18]. Além disso, este capítulo não trata de descrições combinatórias planas nem de imersões em geral. A versão do algoritmo de LEC descrita é de teste de planaridade. A intenção com isto é separar (o máximo possível) o algoritmo abstrato das (muito importantes) tecnicalidades da implementação, que serão tratadas nos próximos capítulos. Ao final deste capítulo é apresentada uma implementação não elaborada do algoritmo de LEC que consome tempo quadrático.

3.1 Molduras, XY-caminhos e XY-obstruções

Thomas [18] descreve a implementação de Shih e Hsu [15] do algoritmo de LEC por meio do conceito de molduras, XY-caminhos e XY-obstruções. Uma descrição de cada um desses objetos e suas inter-relações vem a seguir.

Molduras

Seja M um subgrafo de um grafo H. Diz-se que M é uma **moldura** de H se

(m1) existe um desenho plano de H em que M é o subgrafo induzido pelas arestas adjacentes à face externa.

É evidente que um grafo possui uma moldura se e somente se é planar.

Se H é um subgrafo induzido de um grafo G e M é uma moldura de H, então M é uma **moldura de H em G** se

(m2) todo vértice de H vizinho de um vértice em $V_G \setminus V_H$ é um vértice de M.

A figura 3.1 mostra um subgrafo induzido e sua moldura.

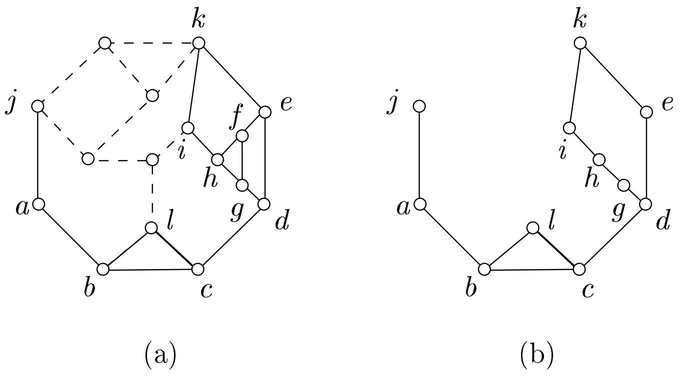
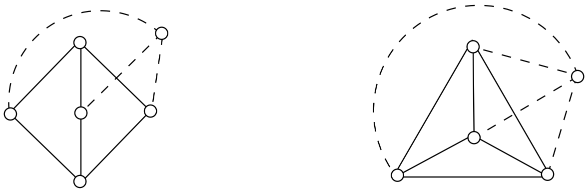


Figura 3.1: (a) Um grafo G onde as linhas sólidas mostram um subgrafo induzido H. (b) Uma moldura de H em G.

Nem todo subgrafo induzido planar de um grafo G possui uma moldura em G (figura 3.2). Na verdade, nem todo subgrafo induzido de um grafo planar G possui uma moldura em G (figura 3.3).

A relação entre planaridade e molduras, que será fundamental para a descrição do algoritmo de LEC, está no seguinte lema.



(a) (b)

Figura 3.2: As linhas sólidas ilustram dois subgrafos induzidos de (a) $K_{3,3}$ e (b) K_5 para os quais não existe moldura (em $K_{3,3}$ e K_5, respectivamente).

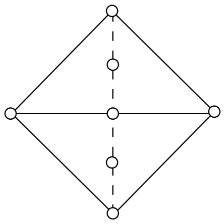


Figura 3.3: As linhas sólidas mostram um subgrafo induzido de um grafo G para o qual não existe moldura em G.

Lema 3.1 (das molduras): *Se H é um subgrafo induzido de um grafo planar G tal que $G - V_H$ é conexo, então H possui uma moldura em G.*

Demonstração: Considere um desenho plano de G. Este desenho induz um desenho plano de H. Seja M o subgrafo de H induzido pelas suas arestas adjacentes a face externa do desenho. Como $G - V_H$ é conexo, então todo vértice de H vizinho de um vértice de $V_G \setminus V_H$ é vértice de M. Logo, M é uma moldura de H em G. ∎

Caminhos perimetrais e XY-caminhos

Seja M uma moldura de um grafo H. Um caminho em M é chamado de **perimetral**. Sejam $C_1, C_2, \ldots, C_k$ todos os blocos de M que contém pelo menos uma aresta de um caminho perimetral P (para cada i, C_i é uma aresta ou um circuito). Para $i = 1, 2, \ldots, k$, seja $P_i := P \cap C_i$. Seja $\overline{P}_i := C_i \setminus P_i$ se C_i tiver mais do que uma aresta, e $\overline{P}_i := P_i$, caso contrário. O caminho $\overline{P} := \overline{P}_1 \cup \overline{P}_2 \cup \ldots \cup \overline{P}_k$ é o **caminho complementar** de P (figura 3.4). Se $E_P = \emptyset$ então $\overline{P} := P$. A **base** (*span*) de P (em M) é o subgrafo $C_1 \cup C_2 \cup \ldots \cup C_k$ de M. Se $E_P = \emptyset$ então a base de P é o próprio P.

Sejam W e Z subconjuntos de vértices de um grafo G e v um vértice de G. Dizemos que v

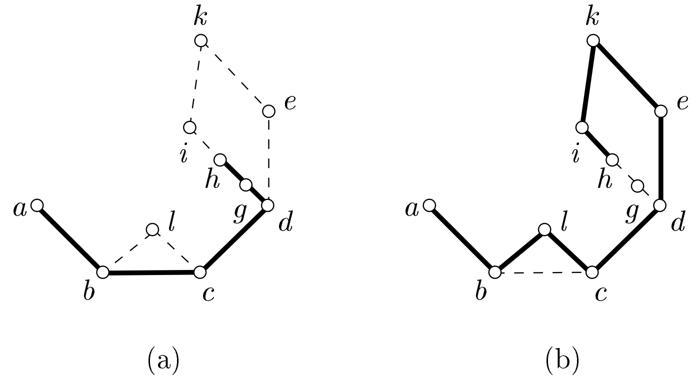


Figura 3.4: (a) Um caminho perimetral P da moldura da figura 3.1. (b) O caminho complementar de P.

enxerga W **através de** Z se existe um caminho em G de v a algum vértice em W com todos os seus vértices internos em Z. Se Z é um subconjunto de arestas, então dizemos que v **enxerga** W **através de** Z se existe um caminho em G de v a algum vértice em W com todas as arestas em Z.

Seja M uma moldura de um grafo conexo e sejam X e Y subconjuntos de V_M. Um caminho perimetral P em M com base S é um XY**-caminho** se:

(c1) as pontas de P estão em X;

(c2) todo vértice de S que enxerga X através de $E_M \setminus E_S$ está em P;

(c3) todo vértice de S que enxerga Y através de $E_M \setminus E_S$ está em $\overline{P}$;

(c4) nenhum componente de $M - V_S$ contém vértices em X e em Y.

Veja a figura 3.5.

XY-obstruções

Dados uma moldura M de um grafo conexo e subconjuntos X e Y de V_M, nem sempre é possível obter-se um XY-caminho em M. Existem três tipos de objetos que tornam impossível a existência de um XY-caminho, as chamadas XY**-obstruções** em M:

(o1) uma quíntupla (C, v_1, v_2, v_3, v_4) onde C é um bloco de M e v_1, v_2, v_3 e v_4 são vértices distintos em C e que aparecem nesta ordem em C, tais que v_1 e v_3 enxergam X através de $E_M \setminus E_C$ e v_2 e v_4 enxergam Y através de $E_M \setminus E_C$;

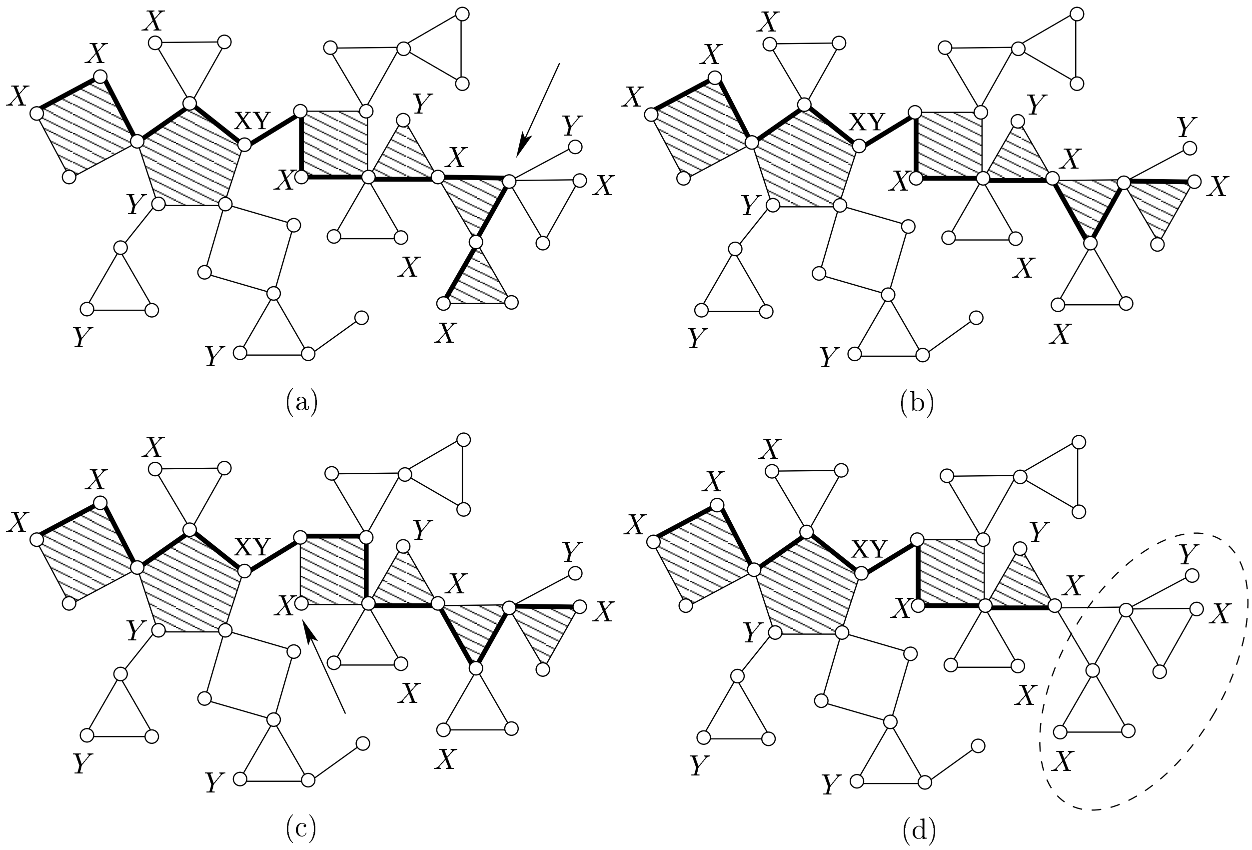


Figura 3.5: Em (a), (b), (c) e (d), denota-se por P o caminho ilustrado em linhas grossas e os blocos da base do respectivo caminho estão hachurados. (a) P não é um XY-caminho pois viola (c3). (b) P é um XY-caminho. (c) P não é um XY-caminho pois viola (c2). (d) P não é um XY-caminho pois viola (c4).

(o2) uma quádrupla (C, v_1, v_2, v_3) onde C é um bloco de M e v_1, v_2 e v_3 são vértices distintos em C que enxergam X e Y através de $E_M \setminus E_C$;

(o3) uma quádrupla (v, K_1, K_2, K_3) tal que $v \in V_M$, e K_1, K_2 e K_3 são três componentes distintos de $M - v$, tais que K_i contém vértices em X e em Y.

XY-caminhos versus XY-obstruções

Como é mostrado a seguir, XY-caminhos e XY-obstruções são objetos que não podem coexistir.

Lema 3.2 (da XY-obstrução): *Seja M uma moldura de um grafo e sejam X e Y subconjuntos de V_M. Se existe uma XY-obstrução em M, então não existe um XY-caminho em M.*

Demonstração: Seja (C, v_1, v_2, v_3, v_4) uma XY-obstrução do tipo (o1). Das condições (c2) e (c3), tem-se que C não pode pertencer à base de um XY-caminho. Já a condição (c4) implica que C deve pertencer a base de todo XY-caminho. Logo, uma obstrução do tipo (o1) impede a existência de um XY-caminho em M. A demonstração de que uma obstrução do tipo (o2) impede a existência de um XY-caminho é análoga.

Suponha agora que (v, K_1, K_2, K_3) é uma obstrução do tipo (o3). A condição (c4) implica que a base de qualquer XY-caminho em M deve conter vértices em cada um dos componentes K_1, K_2 e K_3, um absurdo. ∎

Lema 3.3 (do XY-caminho): *Seja M uma moldura de um grafo conexo H e sejam X e Y subconjuntos de V_M. Se não existe uma XY-obstrução em M, então existe um XY-caminho em M.*

Demonstração: A demonstração é por indução no número de blocos de M. Suponha que M tem apenas um bloco C. Se $X = \emptyset$ então defina $P := \emptyset$. Se $X \neq \emptyset$ então existe um caminho P com pontas em X que contém todos os vértices de X e que não contém vértices internos em Y, pois, caso contrário, C ou teria vértices distintos v_1, v_2, v_3 e v_4 tais que (C, v_1, v_2, v_3, v_4) é uma XY-obstrução do tipo (o1), ou teria vértices distintos v_1, v_2 e v_3 tais que (C, v_1, v_2, v_3) é uma XY-obstrução do tipo (o2). Assim, P é um XY-caminho em M.

Suponha então que M possui pelo menos dois blocos. O restante da demonstração apóia-se na demonstração de alguns fatos.

Fato 3.4: *Se v é um vértice de corte de M, então podemos supor que cada componente de $M - v$ possui um elemento de $X \cup Y$.*

Demonstração: Seja K um componente de $M - v$ tal que $V_K \cap (X \cup Y) = \emptyset$. Seja $M' := M - V_K$. M' é moldura de um subgrafo induzido conexo de H. Ademais, M' não possui uma XY-obstrução; toda XY-obstrução em M' é uma XY-obstrução em M. Logo, por indução, existe um XY-caminho P em M'. Pode-se verificar que P é um XY-caminho em M. ∎

Fato 3.5: *Se v é um vértice de corte de M, então podemos supor que cada componente de $M - v$ possui um elemento de X.*

Demonstração: Seja K um componente de $M - v$ tal que $V_K \cap X = \emptyset$. Pelo fato 3.4, podemos supor que $V_K \cap Y \neq \emptyset$. Sejam $M' := M - V_K$ e $Y' := Y \cup \{v\}$. M' é moldura de um subgrafo induzido conexo de H. Ademais, M' não possui uma XY'-obstrução, já que toda XY'-obstrução em M' induz uma XY-obstrução em M. Assim, por indução, existe um XY'-caminho P em M'. Não é difícil ver que P é um XY-caminho em M. ∎

Fato 3.6: *Se v é um vértice de corte de M, então podemos supor que cada componente de $M - v$ possui um elemento de Y.*

Demonstração: Seja K um componente de $M - v$ tal que $V_K \cap Y = \emptyset$. Sejam $M' := M - V_K$ e $M'' := K + v$. M' e M'' são molduras de subgrafos induzidos conexos de H que têm apenas o vértice v em comum. Pelo fato 3.5, podemos supor que ambos, V_K e $V_M \setminus (V_K \cup \{v\})$ têm um elemento de X. Sejam $X' := (X \cap V_{M'}) \cup \{v\}$, $X'' := (X \cap V_K) \cup \{v\}$. Seja $Y'' := \{v\}$ se $Y \cap V_{M'} \neq \emptyset$, e $Y'' := \emptyset$, caso contrário. Toda $X'Y$-obstrução em M' induz uma XY-obstrução em M. Logo, por indução, existe um $X'Y$-caminho P' em M'. Podemos supor que v é ponta de P' e que v não é um elemento de X, pois, caso contrário, pode-se verificar que P' é um XY-caminho em M. Analogamente, existe um $X''Y''$-caminho P'' em M'' que tem v como uma de suas pontas. Finalmente, não é difícil verificar que, nessas condições, o caminho $P := P' \cup P''$ é um XY-caminho em M. ∎

Fato 3.7: *Se v é um vértice de corte de M, então podemos supor que $M - v$ tem exatamente dois componentes.*

Demonstração: Suponha que $M - v$ tem componentes distintos K_1, K_2 e K_3. Pelos fatos 3.5 e 3.6 podemos supor que K_i possui um elemento de X e um elemento de Y, para $i = 1, 2, 3$. Portanto, (v, K_1, K_2, K_3) é uma XY-obstrução do tipo (o3), uma contradição. ∎

Fato 3.8: *Se C é um bloco de M, então podemos supor que $M - V_C$ tem no máximo dois componentes.*

Demonstração: Suponha que $M - V_C$ tem componentes distintos K_1, K_2 e K_3. Pelos fatos 3.5 e 3.6 tem-se que K_i possui um vértice de X e um vértice de Y, $i = 1, 2, 3$. Seja v_i o vértice de C

que tem vizinhos em K_i, $i = 1, 2, 3$. Pelo fato 3.7 podemos supor que v_1, v_2 e v_3 são vértices distintos. Pelos fatos 3.5 e 3.6 tem-se que cada v_i enxerga X e Y através de $E_M \setminus E_C$. Portanto, (C, v_1, v_2, v_3) é uma XY-obstrução do tipo (o2), uma contradição. ∎

Agora, estamos preparados para concluir a demonstração do lema 3.3. Pelos fatos 3.7 e 3.8, podemos supor que os blocos $C_1, \ldots, C_k$ de M são tais que

$$\text{se } C_i \text{ e } C_j \text{ têm um vértice em comum então } |i - j| \leq 1. \tag{3.1}$$

Sejam v um vértice de corte de M e K um componente de $M - v$. Seja M' o grafo $M - V_K$ e M'' o subgrafo de M induzido por $V_K \cup \{v\}$. M' e M'' são molduras de subgrafos induzidos conexos de H. Pelos fatos 3.5 e 3.6 tem-se que M' e M'' têm elementos de X e Y. Sejam $X' := (X \cap V_{M'}) \cup \{v\}$, $Y' := (Y \cap V_{M'}) \cup \{v\}$, $X'' := (X \cap V_{M''}) \cup \{v\}$ e $Y'' := (Y \cap V_{M''}) \cup \{v\}$. Toda $X'Y'$-obstrução em M' induz uma XY-obstrução em M. Logo, por indução, existe um $X'Y'$-caminho P' em M'. Como v está em $X' \cap Y'$ e devido a (3.1) tem-se que v é uma ponta de P'. De maneira similar, existe um $X''Y''$-caminho P'' com ponta v em M''. Pode-se verificar que $P' \cup P''$ é um XY-caminho em M. ∎

O teorema abaixo resume o conteúdo dos lemas 3.2 e 3.3 da maneira conveniente para a descrição do algoritmo de LEC.

Teorema 3.9: *Seja M uma moldura de um grafo conexo e sejam X e Y subconjuntos de V_M. Vale uma e apenas uma das seguintes afirmações:*

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

∎

3.2 Lec-ordenação

Um dos ingredientes importantes do algoritmo de LEC é uma certa ordenação $v_1, v_2, \ldots, v_n$ dos vértices de um grafo G tal que, para $i = 1, \ldots, n$, os subgrafos

$$G[\{v_1, \ldots, v_i\}] \quad \text{e} \quad G[\{v_{i+1}, \ldots, v_n\}] \text{ são ambos conexos.} \tag{3.2}$$

Equivalentemente, G é conexo e, para $i = 2, \ldots, n-1$, tem-se que

$$v_i \text{ é adjacente a vértices } v_j \text{ e } v_k \text{ com } 1 \leq j < i < k \leq n.$$

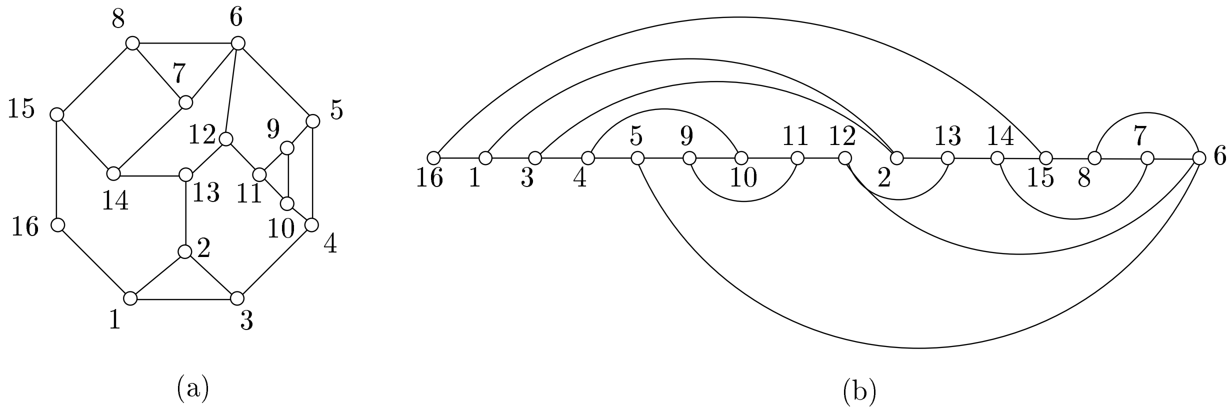


Figura 3.6: (a) Um grafo G. (b) Uma lec-ordenação de G.

Uma tal ordenação é dita uma **lec-ordenação** de G (figura 3.6). Uma lec-ordenação onde $v_1 v_n$ é uma aresta do grafo é conhecida como ***st*-ordenação** [12]. A numeração dos vértices do grafo da figura 3.6 é uma st-ordenação.

Nem todo grafo possui uma lec-ordenação, como mostra a figura 3.7. Entretanto, é possível demonstrar que todo grafo biconexo possui uma lec-ordenação.

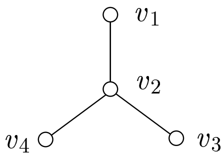


Figura 3.7: Um grafo que não possui uma lec-ordenação.

Lema 3.10 (da lec-ordenação): *Todo grafo biconexo possui uma lec-ordenação.*

Demonstração: Por indução no número de vértices do grafo. O lema vale para grafos com no máximo dois vértices. Assim, seja $G = (V, E)$ um grafo com n vértices, $n > 2$. Seja uv a aresta em uma das pontas de um caminho maximal P em G. Seja x um elemento que não está em V. Sejam

$$V' := (V \setminus \{u, v\}) \cup \{x\} \quad \text{e} \quad E' := (E \setminus \{uv\}) \cup \{xw : uw \in E \text{ ou } vw \in E\}.$$

Como G é biconexo, então da maximalidade de P tem-se que $G - \{u, v\}$ é conexo. Portanto $G' = (V', E')$ é biconexo. É dito que G' corresponde ao grafo obtido de G pela **contração da aresta** uv (e pela remoção de eventuais "arestas paralelas"). Por indução, G' possui uma lec-ordenação $v'_1, v'_2, \ldots, v'_{n-1}$ com $v'_i = x$ para algum i em $\{1, 2, \ldots, n-1\}$. Trocando x na lec-ordenação de G' por u, v ou v, u obtém-se uma lec-ordenação de G. ∎

3.3 Descrição do algoritmo

Dado um grafo biconexo G, o algoritmo de LEC examina os vértices de G um após o outro de acordo com uma lec-ordenação. Em cada iteração, o algoritmo procura estender uma moldura do subgrafo induzido pelos vértices já examinados. Caso isso não seja possível, o algoritmo declara que o grafo não é planar e pára.

Algoritmo de LEC(G). Recebe um grafo biconexo G e devolve SIM se G é planar, e NÃO, caso contrário.

Cada iteração começa com um subgrafo induzido H de G, uma moldura M de H em G, e uma lista L dos vértices do grafo $G - V_H$.

No início da primeira iteração tem-se $H = \emptyset$, $M = \emptyset$ e L é uma lec-ordenação de G. Cada iteração consiste no seguinte:

Caso 1: $L = \emptyset$.

Devolva SIM e pare.

Caso 2: $L \neq \emptyset$.

Seja w o primeiro vértice na lista L.

Seja L' a lista resultante após a remoção de w de L.

Seja $X := \{u \in V_M : uw \in E_G\}$.

Seja $Y := \{u \in V_M : \text{ existe } v \in V_G \setminus (V_H \cup \{w\}), uv \in E_G\}$.

Caso 2A: Existe uma XY-obstrução em M.

Devolva NÃO e pare.

Caso 2B: Existe um XY-caminho P em M.

Seja $\overline{P} := \langle w_0, w_1, \ldots, w_k \rangle$ o caminho complementar de P.

Seja R o conjunto dos vértices em $V_M \setminus V_{\overline{P}}$ que não enxergam $Y \setminus V_{\overline{P}}$ através de $V_M \setminus V_{\overline{P}}$ (figura 3.8(b)).

Seja M' o grafo resultante após acrescentar-se o vértice w e as arestas ww_0 e ww_k ao grafo $M - R$ (figura 3.8(d)).

Seja $H' := H + w$ (figura 3.8(c)).

Comece nova iteração com H', M' e L' nos papéis de H, M e L respectivamente.

▮

3.4 Invariantes

A correção do algoritmo de LEC baseia-se nas demonstrações da validade de uma série de invariantes. Estes invariantes são afirmações envolvendo o grafo dado e os objetos H, M e L. As afirmações são válidas no início de cada iteração do algoritmo e dizem como estes objetos se relacionam entre si e com os dados do problema. As seguintes afirmações são invariantes do algoritmo de LEC:

(lec1) H e $G - V_H$ são grafos conexos;

(lec2) M é uma moldura de H em G.

Para demonstrar-se que as afirmações acima são legítimos invariantes deve-se verificar que:

(a) as afirmações valem no início da primeira iteração e

(b) se as afirmações valem no início de uma iteração em que ocorre o caso 2B, então as afirmações também valem no final da iteração com H', M' e L' nos papéis de H, M e L, respectivamente.

De (a) e (b) conclui-se que (lec1) e (lec2) também valem no início da última iteração, quando ocorre o caso 1 ou o caso 2A.

É evidente que as afirmações valem no início da primeira iteração e não é difícil verificar (b) para (lec1). A seguir está a demonstração de (b) para (lec2).

Demonstração de (lec2): Considere uma iteração em que ocorre o caso 2B. No início de cada iteração tem-se que H e $G - V_H$ são grafos conexos (invariante (lec1)) e que M é uma moldura de H em G (invariante (lec2)). Note que, de (m2) na definição de moldura, vale que

$$X = \{u \in V_H : uw \in E_G\} \quad \text{e} \quad Y = \{u \in V_H : \text{ existe } v \in V_G \setminus (V_H \cup \{w\}), uv \in E_G\}.$$

Considere um desenho plano D de H correspondente à moldura M (figura 3.8(a)). Seja S a base do XY-caminho P em M determinado na iteração. Podemos supor que os caminhos P e $\overline{P}$ não se "cruzam" (figura 3.8(b)).

Pela definição de R (vértices marcados com quadrados na figura 3.8(b)), tem-se que não existe aresta com uma ponta em R e outra em $V_G \setminus (R \cup V_{\overline{P}} \cup \{w\})$. Além disso, da condição (c2) na definição de XY-caminho, vale que todo vizinho de w em V_H está em $V_P \cup R$. Logo, um desenho plano D' de H' pode ser obtido acrescentando-se o vértice w e as arestas de E_G com uma ponta em X e outra em w. Para isto, basta acrescentar w ao desenho D de tal forma que o circuito formado pelas arestas ww_0 e ww_k e pelo caminho P tem em seu interior somente vértices de R (figura 3.8(c)).

O grafo M' definido pelo algoritmo é uma moldura de H' em G correspondente ao desenho D' (figura 3.8(d)). De fato, M' é a moldura de H' correspondente ao desenho D'. Ademais, de (c3) na definição de XY-caminho, tem-se que todo vértice de H' que é vizinho de um vértice de $V_G \setminus V_{H'}$ é um vértice de M'. Logo, M' é uma moldura de H' que satisfaz (m2). Portanto, M' é uma moldura de H' em G. ∎

3.5 Correção

A correção do algoritmo de LEC é facilmente demonstrada através dos invariantes apresentados e do lema a seguir.

Lema 3.11: *Seja M a moldura no início de uma iteração do algoritmo de* LEC *em que ocorre o caso 2. Se existe uma XY-obstrução em M então G possui um subgrafo de Kuratowski.*

Demonstração: Pelo invariante (lec1), tem-se que $G - V_H$ é conexo. Seja T uma árvore geradora de $G - V_H$. Inicialmente L é uma lec-ordenação dos vértices de G, logo, devido a (3.2), podemos supor que w é uma folha de T. Logo, $G - (V_H \cup \{w\})$ é um grafo conexo.

Existem três possibilidades a serem consideradas dependendo do tipo da XY-obstrução encontrada no caso 2A. Verificaremos que cada uma dessas possibilidades implica na existência de um subgrafo de Kuratowski em G.

(1) Existe uma XY-obstrução (C, v_1, v_2, v_3, v_4) do tipo (o1) em M.
Pela definição de XY-obstrução do tipo (o1), existem caminhos internamente disjuntos P_1, P_2, P_3 e P_4 com todos os vértices internos em $V_M \setminus V_C$, tais que:

(1a) P_1 é um caminho entre v_1 e w. P_3 é um caminho entre v_3 e w. Todas as arestas de P_1 e P_3 estão em E_M, exceto as com ponta em w;

(1b) P_2 é um caminho entre v_2 e um vértice u_2 em $V_G \setminus (V_H \cup \{w\})$. P_4 é um caminho entre v_4 e um vértice u_4 em $V_G \setminus (V_H \cup \{w\})$. Todas as arestas de P_2 e P_4 estão em E_M, exceto as com ponta em u_2 e u_4.

Seja T' a subárvore de T com folhas em $\{u_2, u_4, w\}$. Pode-se verificar que o subgrafo de G formado por $C \cup P_1 \cup P_2 \cup P_3 \cup P_4 \cup T'$ é uma subdivisão de $K_{3,3}$ (figura 3.9).

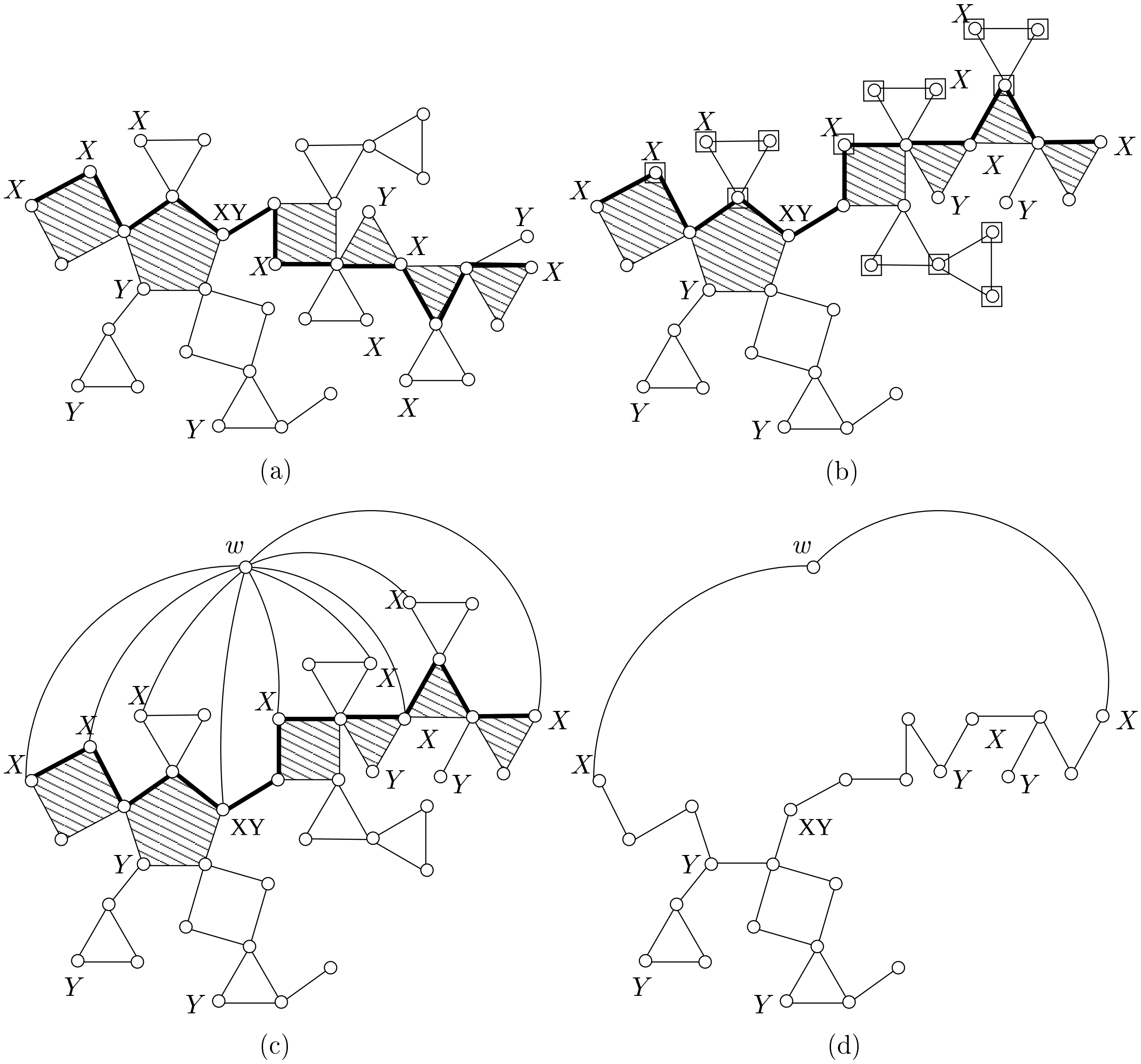


Figura 3.8: (a) Uma moldura M para H e um XY-caminho P (ilustrado em linhas grossas). (b) M após deixar os elementos de X de um lado e os de Y de outro em relação a P. Os vértices em $V_M \setminus V_{\overline{P}}$ que não enxergam $Y \setminus V_{\overline{P}}$ através de $V_M \setminus V_{\overline{P}}$ estão marcados com quadrados, onde $\overline{P}$ é o caminho complementar de P. (c) M juntamente com as arestas com uma ponta em M e outra em w. (d) Uma moldura para $H + w$.

(2) Existe uma XY-obstrução (C, v_1, v_2, v_3) do tipo (o2) em M.
Existem caminhos $P_1, P_2, P_3, Q_1, Q_2, Q_3$ com todos os vértices internos em $V_M \setminus V_C$, tais que:

(2a) Para $i = 1, 2, 3$, Q_i é um caminho entre v_i e w e todas as arestas de Q_i estão em E_M, exceto a com ponta em w;

(2b) Para $i = 1, 2, 3$, P_i é um caminho entre v_i e um vértice u_i em $V_G \setminus (V_H \cup \{w\})$ e todas as arestas de P_i estão em E_M, exceto a com ponta em u_i;

(2c) Para $i = 1, 2, 3$, $P_i \cup Q_i$ é uma árvore cujas folhas estão em $\{u_i, v_i, w\}$;

(2d) Para $i \neq j$, os vértices em comum de $P_i \cup Q_i$ e $P_j \cup Q_j$ estão em $\{w, u_i\}$.

Seja T' a subárvore de T com folhas em $\{u_1, u_2, u_3, w\}$. Seja O o subgrafo de G definido por $C \cup P_1 \cup P_2 \cup P_3 \cup Q_1 \cup Q_2 \cup Q_3 \cup T'$. Se, para $i = 1, 2, 3$, P_i e Q_i são internamente disjuntos e existe um vértice em T' de grau 4 em O, então O é uma subdivisão de K_5 (figura 3.10(a)), caso contrário, O é uma subdivisão de $K_{3,3}$ (figuras 3.10(b) e 3.10(c)).

(3) Existe uma XY-obstrução (v, K_1, K_2, K_3) do tipo (o3) em M.
Pela definição de XY-obstrução do tipo (o3), existem caminhos $P_1, P_2, P_3, Q_1, Q_2, Q_3$ com todos os vértices internos em V_M, tais que

(3a) Para $i = 1, 2, 3$, Q_i é um caminho entre v e w e todas as arestas de Q_i estão em E_{K_i}, exceto as com ponta em w ou v;

(3b) Para $i = 1, 2, 3$, P_i é um caminho entre v e um vértice u_i em $V_G \setminus (V_H \cup \{w\})$ e todas as arestas de P_i estão em E_{K_i}, exceto as com ponta em w ou v;

(3c) Para $i = 1, 2, 3$, $P_i \cup Q_i$ é uma árvore com folhas u_i, v e w. (Em particular $P_i \cap Q_i$ é um caminho com pelo menos uma aresta.)

(3d) Para $i \neq j$, os vértices em comum de $P_i \cup Q_i$ e $P_j \cup Q_j$ estão em $\{u_i, v, w\}$.

Seja T' a subárvore de T com folhas em $\{u_1, u_2, u_3, w\}$. O subgrafo de G definido por $P_1 \cup P_2 \cup P_3 \cup Q_1 \cup Q_2 \cup Q_3 \cup T'$ é uma subdivisão de $K_{3,3}$ (figura 3.11).

Portanto, cada tipo de XY-obstrução em M implica na existência de um subgrafo de Kuratowski em G. Isto conclui a demostração do lema 3.11. ∎

Teorema 3.12 (da correção): *O algoritmo de* LEC *decide se um dado grafo é planar.*

Demonstração: Como o grafo G dado é biconexo, então, pelo lema 3.10, G possui uma lec-ordenação (calculada antes da primeira iteração do algoritmo).

Se o algoritmo pára após executar o caso 1 e devolve SIM então, da definição de moldura, o grafo dado é planar e a resposta está correta.

Suponha que o algoritmo está numa iteração em que ocorre o caso 2. Seja M a moldura no início da iteração e X e Y os conjuntos definidos no início do caso 2. Pelo teorema 3.9, ocorre

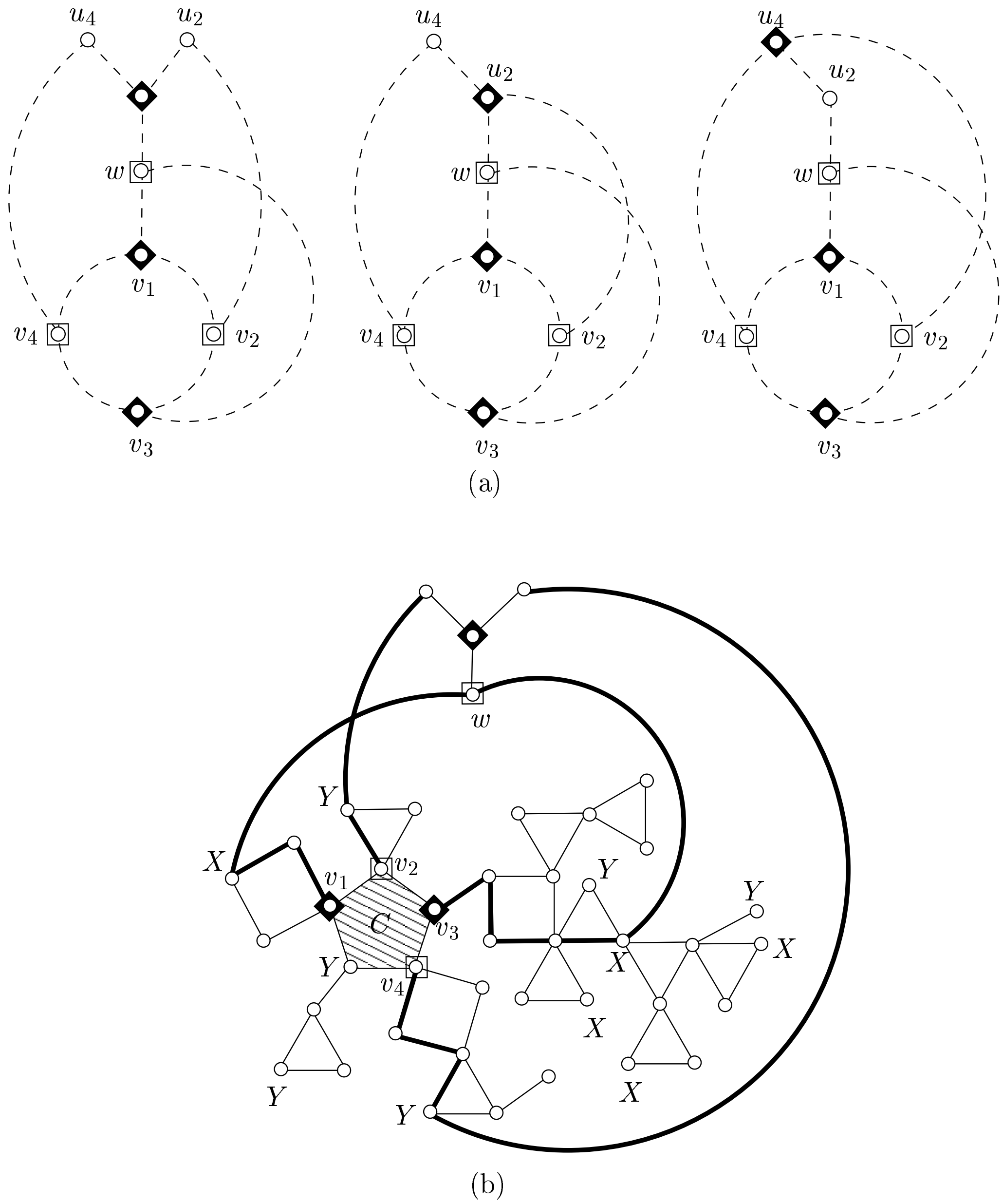


Figura 3.9: (a) Os quadrados e losangos ilustram os vértices da bipartição de uma subdivisão de $K_{3,3}$. (b) Um exemplo concreto onde pode ser vista uma subdivisão de $K_{3,3}$, como na primeira figura em (a).

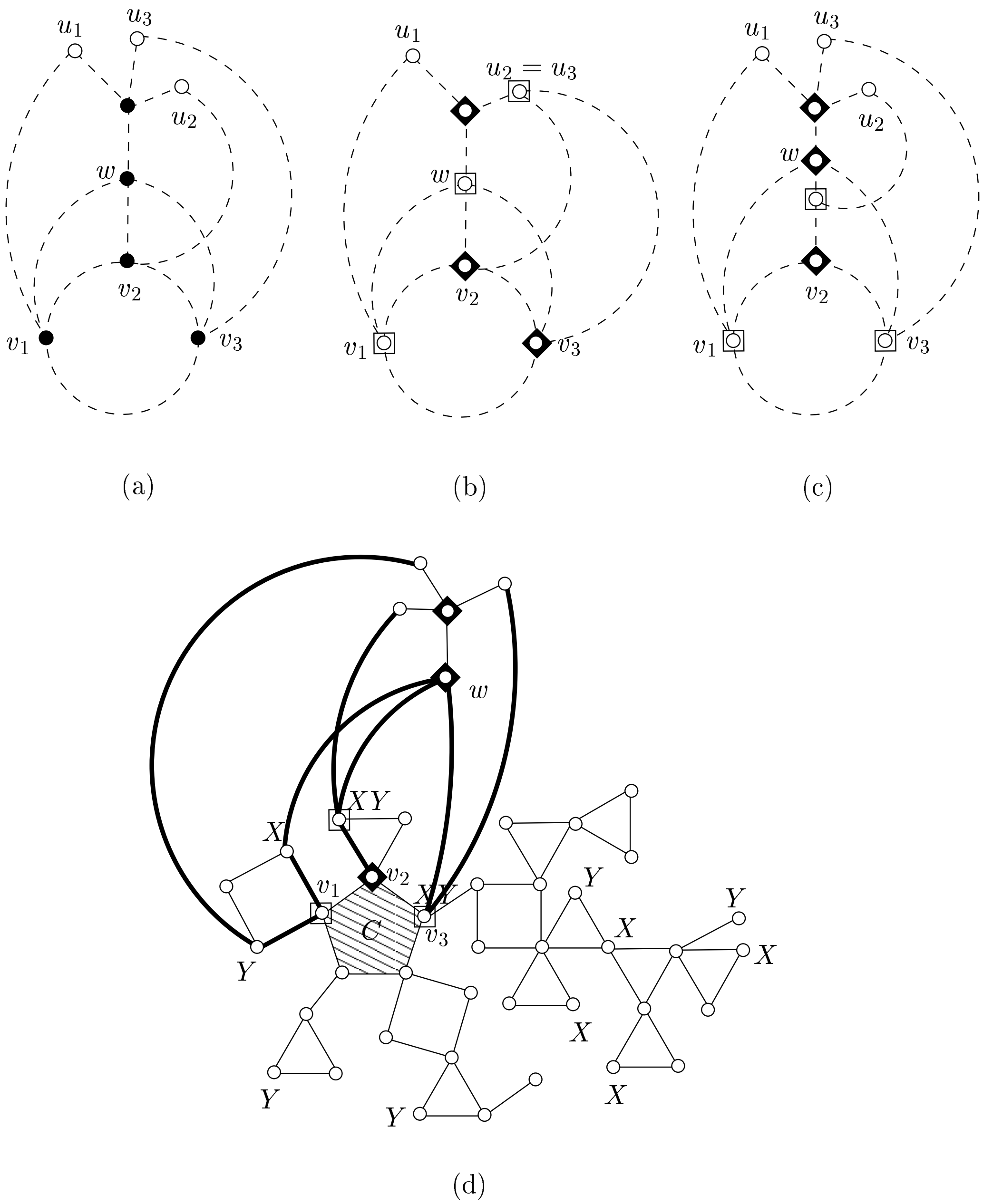


Figura 3.10: (a) Uma subdivisão de K_5. Em (b) e (c), os quadrados e losangos ilustram os vértices da bipartição de uma subdivisão de $K_{3,3}$. (d) Um exemplo concreto onde pode ser vista uma subdivisão de $K_{3,3}$, como em (c).

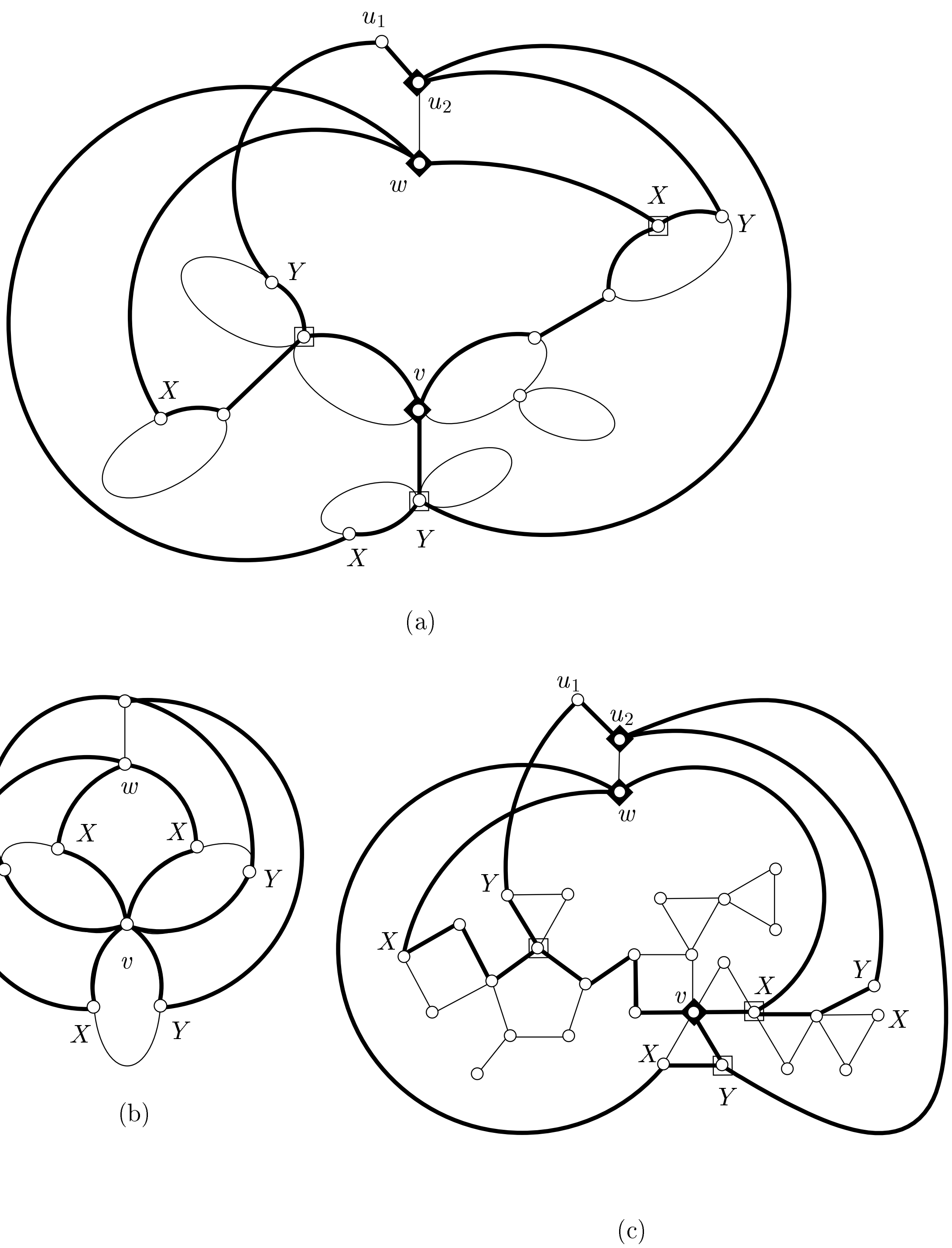


Figura 3.11: (a) Obstrução do tipo (o3). (b) Não há um subgrafo de Kuratowski que contém os caminhos em linhas grossas. (c) Exemplo concreto onde pode ser vista uma subdivisão de $K_{3,3}$, como em (a).

um e apenas um, entre os casos 2A e 2B. Se ocorre o caso 2B, então, pelos resultados da seção anterior, os invariantes (lec1) e (lec2) valem no início da próxima iteração. Se ocorre o caso 2A, então o algoritmo pára e corretamente declara o grafo como não-planar. De fato, pelo lema 3.11, G possui um subgrafo de Kuratowski e portanto, pelo lema 2.6, G não é planar.

Finalmente, é evidente que o algoritmo pára depois de no máximo n iterações, onde n é o número de vértices de G. ∎

3.6 Teorema de Kuratowski

Uma conseqüência importante da correção do algoritmo de LEC é o bem-conhecido teorema de Kuratowski.

Corolário 3.13 (Teorema de Kuratowski): *Um grafo é planar se e somente se não possui um subgrafo de Kuratowski.*

Demonstração: Seja G um grafo. É claro que G é planar se e somente se cada um dos seus componentes biconexos é planar. Logo, podemos supor que G é biconexo. Pelo lema 2.6, tem-se que se G possui um subgrafo de Kuratowski, então G não é planar. Reciprocamente, suponha que G não é planar e considere uma aplicação do algoritmo de LEC tendo G como entrada. Como G não é planar, então, pelo teorema 3.12, o algoritmo pára em alguma iteração após a ocorrência do caso 2A, depois de encontrar uma XY-obstrução na moldura corrente. Pelo lema 3.11, G possui um subgrafo de Kuratowski. ∎

3.7 Eficiência

A eficiência do algoritmo de LEC depende do tempo consumido para resolver cada um dos seguintes subproblemas:

Problema da lec-ordenação: Dado um grafo biconexo G, encontrar uma lec-ordenação dos vértices de G.

Problema do XY-caminho: Dados uma moldura conexa M e subconjuntos X e Y de V_M, encontrar um XY-caminho ou uma XY-obstrução em M.

Problema da nova moldura: Dados uma moldura M de um grafo conexo H, subconjuntos X e Y de V_M, um XY-caminho P em M, e um elemento w que não está em V_H, construir uma moldura M' para o grafo $(V_H \cup \{w\}, E_H \cup \{wu : u \in X\})$.

A seguir, cada subproblema é tratado independentemente. Ao final desta seção, é apresentado o consumo de tempo resultante para o algoritmo de LEC.

Determinação de uma lec-ordenação

Seja G um grafo biconexo com n vértices e m arestas. O lema 3.10 pode ser facilmente transformado em um algoritmo recursivo para obter uma lec-ordenação dos vértices de G. Se $n \leq 2$, na base da recursão, qualquer ordenação dos vértices de G é uma lec-ordenação; este caso consome tempo constante. Suponha que $n > 2$. Neste caso, o algoritmo, antes da chamada recursiva, precisa determinar uma aresta uv que seja ponta de um caminho maximal em G. Depois da determinação de uv, o algoritmo obtém, recursivamente, uma lec-ordenação para o grafo G' resultante de G após a contração da aresta uv. Qualquer aresta incidente a uma folha de uma árvore de busca em profundidade (bp-árvore) de G é ponta de um caminho maximal em G. A determinação de uv e a construção de G' consomem tempo $O(n+m)$.

Finalmente, dada uma lec-ordenação de G', uma lec-ordenação de G é obtida trocando-se o vértice x resultante da contração de uv por u, v ou v, u. A determinação de qual entre as ordens u, v e v, u produz uma lec-ordenação pode ser feita em tempo $O(n+m)$. O consumo de tempo do algoritmo resultante é $O(n^2+m^2)$. A seguir encontra-se uma versão iterativa do algoritmo descrito. No algoritmo, se $L = \langle v_0, v_1, \ldots, v_i, \ldots, v_k \rangle$ é uma lista, então dizemos que $v_0, \ldots, v_{i-1}$ são os **predecessores** de v_i em L e $v_{i+1}, ldots, v_n$ são os **sucessores** de v_i.

> **Algoritmo lec-ordenação**(G). Recebe um grafo biconexo G e devolve uma lec-ordenação de V_G.

Cada iteração começa com duas listas L e I de vértices de G. No início da primeira iteração, $L = \emptyset$ e I representa uma ordenação de V_G obtida ao percorrer uma bp-árvore T de G em pré-ordem. (A lista L representa uma lec-ordenação dos vértices de G após a contração das arestas de T que não têm ambas as pontas em I.) Cada iteração consiste em

Caso 1: $I = \emptyset$.

Devolva L e pare.

Caso 2: $L = \emptyset$.

Seja v o primeiro vértice na lista I.

Seja $I' := I \setminus \{v\}$.

Seja $L' := \langle v \rangle$.

Comece nova iteração com L' e I' nos papéis de L e I.

Caso 3: $I \neq \emptyset$ e $L \neq \emptyset$.

Seja v o primeiro vértice na lista I.

Seja $I' := I \setminus \{v\}$.

Seja u o pai de v em T.

Caso 3A: Existe uma aresta de um descendente de v em T a um predecessor de u em L.

Seja L' a lista obtida após inserir v imediatamente antes de u em L.

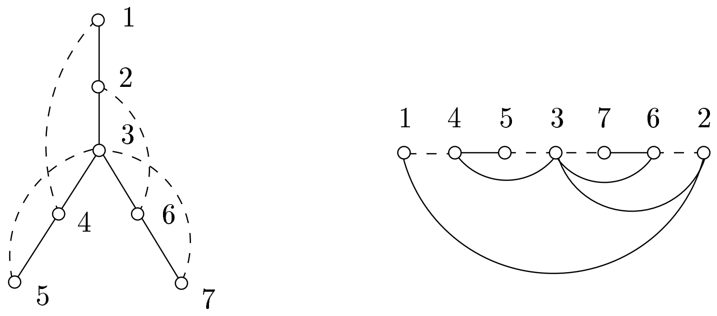
Comece nova iteração com L' e I' nos papéis de L e I.

Caso 3B: Existe uma aresta de um descendente de v em T a um sucessor de u em L.

Seja L' a lista obtida após inserir v imediatamente depois de u em L.

Comece nova iteração com L' e I' nos papéis de L e I.

∎



v	L
–	$\langle\rangle$
1	$\langle 1 \rangle$
2	$\langle 1, 2 \rangle$
3	$\langle 1, 3, 2 \rangle$
4	$\langle 1, 4, 3, 2 \rangle$
5	$\langle 1, 4, 5, 3, 2 \rangle$
6	$\langle 1, 4, 5, 3, 6, 2 \rangle$
7	$\langle 1, 4, 5, 3, 7, 6, 2 \rangle$

(a) (b)

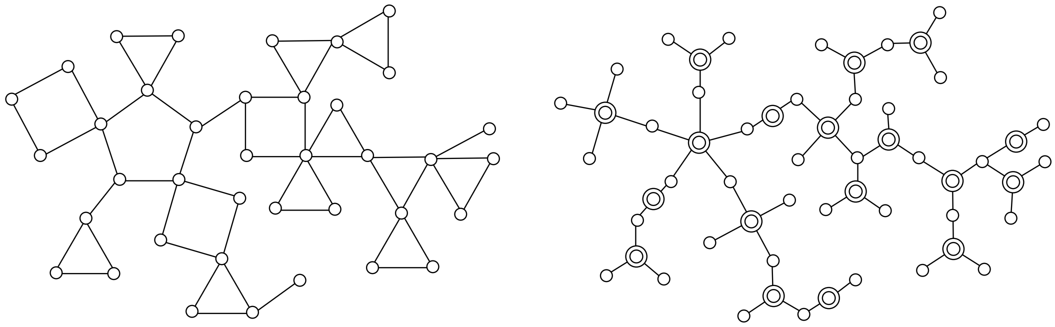
Figura 3.12: (a) As linhas sólidas ilustram uma bp-árvore de um grafo G cujos vértices estão numerados de acordo com a visita pré-ordem na árvore. Inicialmente, $I = \langle 1, 2, 3, 4, 5, 6, 7 \rangle$. (b) Uma lec-ordenação obtida pelo algoritmo lec-ordenação(G) e a lista L para cada vértice v examinado pelo algoritmo.

Busca por um XY-caminho

Seja M uma moldura conexa e sejam X e Y subconjuntos de V_M. Para resolver o problema do XY-caminho, seguimos os passos da demonstração do lema 3.3.

Encontrar um XY-caminho ou uma XY-obstrução numa moldura que é uma árvore não é uma tarefa difícil. O algoritmo a seguir resolve o problema do XY-caminho manipulando uma árvore que representa a moldura dada.

Seja H um grafo conexo e $\mathcal{B}$ o conjunto dos blocos de H. Pode-se definir de maneira natural um grafo bipartido que tem $V_H \cup \mathcal{B}$ como conjunto de vértices e arestas da forma Bv onde B é um bloco de H e v é um vértice de B. É evidente que um tal grafo é uma árvore que denominamos de a **árvore de blocos**[1] de H (figura 3.13).



(a) (b)

Figura 3.13: (a) Uma moldura e (b) sua respectiva árvore de blocos.

Seja T a árvore de blocos de um grafo conexo H. Cada nó de T que é um bloco de H é dito um **C-nó** e um nó de T que é um vértice de H é chamado de **P-nó**.

Passamos agora a descrever o algoritmo que resolve o problema do XY-caminho. O algoritmo utiliza a representação da moldura dada através de sua árvore de blocos[2].

> **Algoritmo XY-caminho(M, X, Y).** Recebe uma moldura conexa M e subconjuntos X e Y de V_M e devolve um XY-caminho ou uma XY-obstrução em M.

Seja T_0 a árvore de blocos de M. O algoritmo é iterativo e cada iteração começa com uma subárvore T de T_0, subconjuntos X_T e Y_T de V_T e subconjuntos W e Z de V_M. Os conjuntos X_T e Y_T contém os nós da árvore T que enxergam X e Y através de $E_{T_0} \setminus E_T$, respectivamente. Os conjuntos W e Z contém os P-nós da árvore T_0 que enxergam X e Y através de $E_{T_0} \setminus E_T$, respectivamente. No início da primeira iteração, $T = T_0$, $X_T = X$, $Y_T = Y$, $W = X$ e $Z = Y$. Cada iteração consiste em:

Caso 1: Toda folha de T está em $X_T \cap Y_T$ e T é um caminho.

[1]Por conveniência, esta definição é levemente diferente da usual [6].

[2]A estrutura da árvore de blocos de uma moldura assemelha-se à PC-árvore usada no próximo capítulo para representar e manipular molduras (seção 4.2).

Seja R o conjunto dos P-nós em T.

Para cada C-nó C em T, seja $X_C := V_C \cap (W \cup R)$ e $Y_C := V_C \cap (Z \cup R)$.

Caso 1A: Cada C-nó C em T possui um caminho P_C contendo X_C e internamente disjunto de Y_C.

Seja P_T o caminho em M obtido através da concatenação dos caminhos em $\{P_C : C$ é um C-nó em T$\}$.

Seja P um caminho em M contendo P_T e com extremos em X tal que para cada bloco C na base de P, tem-se que P contém os vértices em $V_C \cap W$.

Devolva P e pare.

Caso 1B: Existe um C-nó C em T que não possui um caminho contendo X_C e internamente disjunto de Y_C.

Sejam v_1, v_2, v_3 e v_4 vértices distintos de C e que aparecem nesta ordem em C tais que v_1 e v_3 estão em X_C e v_2 e v_4 estão em Y_C.

Devolva (C, v_1, v_2, v_3, v_4) e pare.

Caso 2: Toda folha de T está em $X_T \cap Y_T$ e existe um nó v em T com grau maior que 2.

Caso 2A: v é um C-nó.

Seja C o bloco de M correspondente a v.

Sejam v_1, v_2 e v_3 P-nós distintos adjacentes a v em T.

Devolva (C, v_1, v_2, v_3) e pare.

Caso 2B: v é um P-nó.

Sejam C_1, C_2 e C_3 C-nós distintos adjacentes a v em T.

Sejam K_1, K_2 e K_3 componentes de $M - v$ tais que, para $i = 1, 2, 3$, C_i é bloco de $K_i + v$.

Devolva (v, K_1, K_2, K_3) e pare.

Caso 3: Existe uma folha f de T que não está em $X_T \cap Y_T$.

Seja u o nó de T adjacente a f.

Seja $T' := T - f$.

Seja $X_{T'} := (X_T \setminus \{f\}) \cup \{u\}$ se f está em X_T; caso contrário $X_{T'} := X_T$.

Seja $Y_{T'} := (Y_T \setminus \{f\}) \cup \{u\}$ se f está em Y_T; caso contrário $Y_{T'} := Y_T$.

Seja $W' := W \cup \{u\}$ se f está em X_T e u é um P-nó; caso contrário $W' := W$.

Seja $Z' := Z \cup \{u\}$ se f está em Y_T e u é um P-nó; caso contrário $Z' := Z$.

Comece nova iteração com $T', X_{T'}, Y_{T'}, W'$ e Z' nos papéis de T, X_T, Y_T, W e Z, respectivamente.

∎

A execução do algoritmo consiste de uma seqüência de "reduções" feitas no caso 3 seguida de uma ocorrência do caso 1 ou do caso 2. O caso 3 do algoritmo realiza as reduções descritas nos fatos 3.4, 3.5 e 3.6 na demonstração do lema 3.3.

No caso 1A, o algoritmo pára após devolver um XY-caminho P em M. Já nos casos 1B, 2A e 2B, o algoritmo pára após devolver uma XY-obstrução em M: o caso 1B devolve uma obstrução (C, v_1, v_2, v_3, v_4) do tipo (o1); caso 2A devolve uma obstrução (C, v_1, v_2, v_3) do tipo (o2); e o caso 2B devolve uma obstrução (v, K_1, K_2, K_3) do tipo (o3).

Todos os blocos de um dado grafo podem ser determinados em tempo linear [8]. Dado um grafo e seus blocos, a árvore de blocos do grafo dado pode ser construída em tempo linear. Logo, a quantidade de tempo gasta pelo algoritmo com pré-processamento é proporcional a $|V_M|$.

Após uma ocorrência do caso 1 ou do caso 2, o algoritmo pára. O tempo gasto com esta ocorrência é proporcional a $|V_M| + |E_M|$. Como $|E_M| = O(|V_M|)$, então a quantidade total de tempo gasto com os casos 1 e 2 é $O(|V_M|)$.

O número de ocorrências do caso 3 é não superior a $|V_M| + |E_M|$. O tempo gasto com cada ocorrência do caso 3 é constante. Logo, o consumo total de tempo do algoritmo gasto com ocorrências do caso 3 é $O(|V_M|)$. Portanto, a quantidade total de tempo gasta pelo algoritmo é $O(|V_M|)$.

No início da última iteração, após a seqüência de reduções, as folhas de T são chamadas de **nós terminais**. O conceito de nó terminal é usado de maneira fundamental pela implementação descrita no próximo capítulo.

Construção da nova moldura

Sejam M uma moldura de um grafo conexo H, X e Y subconjuntos de V_M, P um XY-caminho em M e um elemento w que não está em V_H. O problema da construção da nova moldura é o mais simples dos três problemas. Na verdade, a sua solução está descrita em três linhas no caso 2B do algoritmo de LEC:

> ...
>
> Seja $\overline{P} := \langle w_0, w_1, \ldots, w_k \rangle$ o caminho complementar de P.
>
> Seja R o conjunto dos vértices em $V_M \backslash V_{\overline{P}}$ que não enxergam $Y \backslash V_{\overline{P}}$ através de $V_M \backslash V_{\overline{P}}$ (figura 3.8(b)).
>
> Seja M' o grafo resultante após acrescentar-se o vértice w e as arestas ww_0 e ww_k ao grafo $M - R$ (figura 3.8(d)).
>
> ...

Devido ao invariante (lec2) tem-se que o grafo M', na terceira linha acima, é uma moldura para o grafo $(V_H \cup \{w\}, E_H \cup \{wu : u \in X\})$.

O caminho $\overline{P}$ pode ser construído em tempo $O(|V_M|)$. De fato, suponha que cada aresta de M em P foi marcada. Uma lista $\mathcal{L}$ com os blocos de M pode ser determinada em tempo linear [8]. Os blocos na base S de P podem ser identificados percorrendo-se a lista $\mathcal{L}$ e examinando-se as arestas em cada bloco. A determinação de S consome, portanto, tempo $O(|E_S|) = O(|V_M|)$. Finalmente, com S e P em mãos o caminho $\overline{P}$ pode ser obtido em tempo $O(|E_S|)$.

O conjunto $V_M \setminus R$ de vértices (e portanto R) pode ser construído através de uma busca em largura no grafo M a partir dos vértices em Y. Na busca, os vértices em $\overline{P}$ não devem ser visitados. Esta busca em largura consome tempo $O(|V_M|)$.

Dados R, w, w_0 e w_k, o consumo de tempo para construir-se M' é proporcional a $|V_M|$.

Portanto, como o consumo de tempo de cada uma das três linhas acima do caso 2B é $O(|V_M|)$, então o problema da construção da nova moldura pode ser resolvido em tempo linear.

Consumo de tempo do algoritmo de LEC

O teorema a seguir mostra o consumo de tempo do algoritmo de LEC quando resolvemos os problemas da lec-ordenação, do XY-caminho e da construção da nova moldura como descrito anteriormente.

Teorema 3.14 (da eficiência): *O algoritmo de* LEC *gasta tempo* $O(n^2 + m^2)$.

Demonstração: Seja G um grafo biconexo com n vértices e m arestas. O tempo gasto pelo algoritmo de LEC com o pré-processamento do grafo G é dominado pelo consumo de tempo para obter-se uma lec-ordenação dos vértices de G. Isto pode ser feito em tempo $O(n^2 + m^2)$.

Considere uma iteração em que ocorre o caso 2 do algoritmo de LEC. No início deste caso, o tempo gasto para determinar w, L', X e Y é linear. Decidir qual entre os casos 2A e 2B ocorre consiste em resolver o problema do XY-caminho, o que pode ser feito em tempo proporcional a $|V_M|$. Uma ocorrência do caso 2A gasta tempo constante. Já uma ocorrência do caso 2B consiste, essencialmente, em construir uma nova moldura, o que pode ser feito em tempo linear. Como cada ocorrência do caso 2 gasta tempo $O(n + m)$ e este caso ocorre no máximo n vezes, então o tempo total gasto com o caso 2 é $O(n(n + m))$.

Portanto, como o caso 1 ocorre no máximo uma vez e gasta tempo constante, então o consumo de tempo total do algoritmo é $O(n^2 + m^2)$. ∎

No próximo capítulo é apresentada uma descrição da implementação do algoritmo de LEC proposta por Shih e Hsu [15].

Implementação de Shih e Hsu

Neste capítulo é apresentada a implementação de Shih e Hsu [15] ("algoritmo de SH") para o algoritmo de LEC. Esta implementação resolve o problema da planaridade: dado um grafo G, ela devolve uma descrição combinatória plana de G ou um subgrafo de Kuratowski. Neste capítulo é descrita apenas a parte da implementação referente ao teste de planaridade. Nos capítulos seguintes são tratados os aspectos da implementação referentes à determinação de um subgrafo de Kuratowski e à construção de uma descrição combinatória plana.

4.1 Ordem de processamento dos vértices

Na implementação de Shih e Hsu, os vértices são examinados de acordo com uma bp-ordenação $v_1, v_2, \ldots, v_n$ do grafo dado (capítulo 1). A cada iteração, a partir de uma moldura de $G[\{v_1, \ldots, v_{i-1}\}]$, o algoritmo, se possível, obtém uma moldura de $G[\{v_1, \ldots, v_{i-1}, v_i\}]$.

Uma bp-ordenação pode ser vista como uma "relaxação" da lec-ordenação. Seja G um grafo conexo e $v_1, v_2, \ldots, v_n$ uma bp-ordenação de G (em relação a uma bp-árvore arbitrária). Então o grafo

$$G[\{v_i, \ldots, v_n\}] \text{ é conexo para } i = 1, \ldots, n. \tag{4.1}$$

Como uma bp-ordenação não garante que $H := G[\{v_1, \ldots, v_{i-1}\}]$ seja conexo, se existe uma moldura M de H e H não é conexo, então M também não é conexo. Com isso, para calcular uma moldura M' de $H' := H + v_i$, pode ser necessário calcular mais de um XY-caminho em M.

4.2 Estrutura de dados para a moldura

O algoritmo de SH mantém uma moldura em G do grafo $G[v_1, \ldots, v_{i-1}]$. A estrutura de dados usada, proposta por Shih e Hsu [15], é conhecida como PC-árvore e é definida a seguir.

PC-árvore

Uma **PC-árvore** é uma árvore enraizada com dois tipos de nós — os P-nós e os C-nós — e com as seguintes propriedades:

(pc1) a raiz e as folhas são P-nós;

(pc2) os filhos de cada C-nó têm uma ordem.

Exceto pela raiz, cada nó tem um apontador para indicar quem é o seu pai na PC-árvore.

Considere o início da iteração do algoritmo de SH em que o vértice v_i é examinado. Tal iteração começa com uma moldura M do grafo $G[\{v_1, \ldots, v_{i-1}\}]$ dada por uma PC-árvore T da seguinte maneira.

(T1) cada P-nó de T é um vértice de M;

B_c (T2) cada C-nó c de T corresponde a um bloco B_c de M com pelo menos três arestas e vice-versa;

(T3) o pai de um C-nó c é o vértice em B_c de maior bp-número e os filhos de c são os demais vértices de B_c;

(T4) cada C-nó c vem acompanhado de uma lista, denotada por RBC(c) (*Representative Boundary Cycle*)[1] , com seu pai no início e seus filhos na ordem em que aparecem em B_c. RBC(c)

Veja as figuras 4.1(a) a (c). Diz-se que T está associada à moldura M.

Note a semelhança entre tal PC-árvore e a árvore de blocos de M (seção 3.7). Basicamente, além da orientação, a única diferença é que, na árvore de blocos, há nós representando blocos que consistem em apenas uma aresta, enquanto que na PC-árvore não.

É fácil manter uma PC-árvore se não há preocupação com o consumo de tempo do algoritmo. Para uma implementação quadrática, por exemplo, é trivial manter-se uma PC-árvore. Abaixo, segue uma descrição das modificações feitas na PC-árvore nesta implementação para garantir que o algoritmo gaste tempo linear no número de vértices do grafo dado. Uma dessas mudanças é em relação à atualização dos pais de alguns filhos dos C-nós que serão "englobados" por um novo C-nó. Essa mudança provoca a formação de caminhos em T formados por C-nós consecutivos. Com uma idéia similar à heurística da compressão de caminhos da estrutura de dados conhecida como *union-find* [16], obtém-se algo melhor do que quadrático, porém possivelmente não-linear.

[1] Conforme é descrito mais adiante, a implementação usa uma PC-árvore alterada, onde é mantido uma versão "compacta" dessa lista.

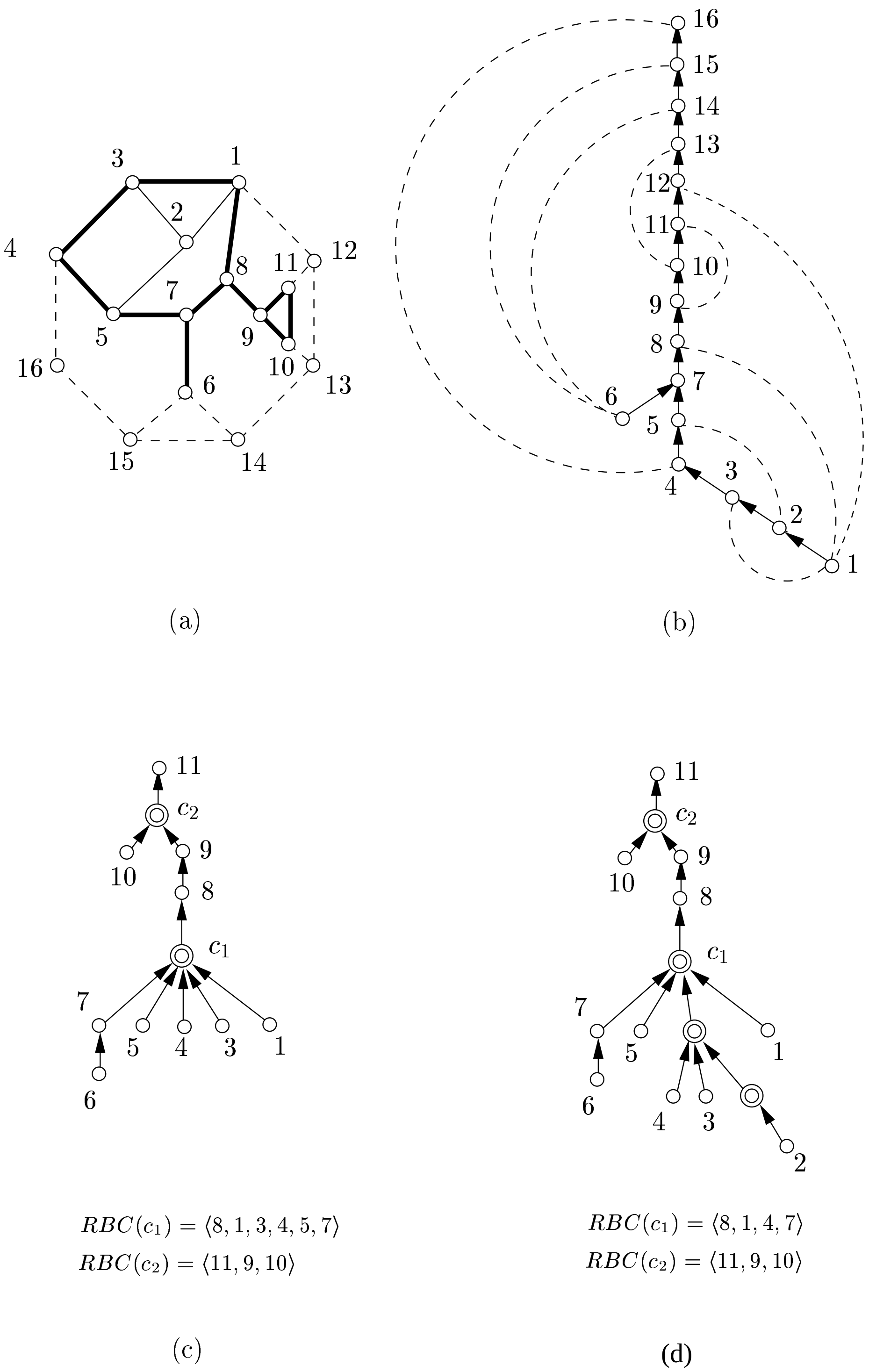


Figura 4.1: Os C-nós são ilustrados por dois círculos concêntricos e as setas indicam o pai de cada nó na PC-árvore. (a) Um grafo G e uma moldura M (em linhas grossas) de $G[\{1, \ldots, 11\}]$. (b) Uma bp-árvore de G. (c) Uma PC-árvore de M e as listas $RBC(c_1)$ e $RBC(c_2)$. (d) Uma PC-árvore alterada, correspondente a (c).

PC-árvore alterada

Na implementação aqui apresentada, para garantir que o tempo consumido pelo algoritmo seja linear, há duas diferenças fundamentais na maneira como a PC-árvore é mantida frente à descrição acima. A primeira diz respeito aos apontadores que indicam o pai dos filhos de um C-nó. A segunda, ao RBC de um C-nó. Essa maneira de implementar a PC-árvore do algoritmo de SH é chamada de **PC-árvore alterada**.

A primeira diferença, relativa aos filhos de um C-nó, grosseiramente falando, consiste na postergação da atualização de alguns apontadores para o pai. Como conseqüência de tal medida, permanecem na PC-árvore C-nós de blocos de iterações anteriores que não estão mais na moldura corrente. Tais C-nós são ditos **inativos** e são caracterizados por terem seu ponteiro pai apontando para um outro C-nó. Os C-nós que de fato correpondem a um bloco da moldura corrente são chamados de **ativos** e apenas C-nós ativos têm um RBC. Na verdade, além de C-nós inativos, ficam presentes na árvore, sem prejuízo para o algoritmo, vértices de G que não fazem mais parte da moldura corrente. Veja na figura 4.1(d) um possível exemplo de PC-árvore alterada correspondente à PC-árvore exibida na figura 4.1(c). Note a presença do P-nó 2 na PC-árvore da figura 4.1(d).

A segunda diferença consiste em manter-se uma versão "compacta" do RBC. Seja $L :=$
L $\{v_i, \dots, v_n\}$. Para cada C-nó ativo d, vértices de B_d que não enxergam L através de $V_G \setminus V_{B_d}$ são omitidos do RBC(d) no momento em que são visitados pelo algoritmo. Por exemplo, os vértices 3 e 5 são omitidos do RBC do C-nó c_1 na figura 4.1(d).

Na PC-árvore alterada T, diz-se que um C-nó c é o **pai real** de um P-nó p (e que p é o **filho real** de c) se o pai de p é um C-nó e c é o primeiro C-nó ativo encontrado no caminho de p à raiz de T.

Abaixo são mostradas versões adaptadas de (T2), (T3) e (T4) à PC-árvore alterada:

(T2') cada C-nó ativo c de T corresponde a um bloco B_c de M com pelo menos três arestas e vice-versa;

(T3') cada C-nó inativo de T é filho de um C-nó em T; o pai de um C-nó ativo c é o vértice em B_c de maior bp-número e os filhos reais de c incluem os vértices de B_c que enxergam L através de $V_G \setminus V_{B_c}$;

(T4') cada C-nó ativo c vem acompanhado de uma lista, denotada por RBC(c), com seu pai no início e os filhos reais de c em T que enxergam L através de $V_G \setminus V_{B_c}$, na ordem em que aparecem em B_c.

Para que o percurso de uma PC-árvore alterada possa ser feito eficientemente, informações extras são necessárias. Cada P-nó que é filho de um C-nó tem um apontador para a célula do RBC correpondente a este P-nó, caso exista. Para cada C-nó ativo d, as células dos extremos do RBC(d) têm um apontador para d. Na seção 4.5, encontra-se uma descrição de como uma PC-árvore alterada é percorrida e como essas informações são utilizadas.

Declaração de uma PC-árvore alterada

A declaração de uma PC-árvore alterada T na implementação do algoritmo de SH é ilustrada na figura 4.2. A variável *parent[u]* guarda um apontador para o pai de u em T. Para cada C-nó ativo c de T, *RBC[c]* é o RBC(c) armazenado numa lista do tipo **bmlist**.

Para cada P-nó u cujo pai real é um C-nó ativo c, se u está em RBC(c) então *ptritem[u]* aponta a célula correspondente a u em RBC(c).

Para cada C-nó ativo c, sejam i_1 e i_2 apontadores para a primeira e a última célula de RBC(c), respectivamente. As variáveis *activecnode[i_1]* e *activecnode[i_2]* apontam para c.

Veja na figura 4.3 como estão as variáveis no caso da PC-árvore alterada da figura 4.1(d).

53 ⟨declaração de uma PC-árvore 53⟩≡

```
node_map<node> parent;
map<node, bmlist<node> > RBC;
map<bmlist_item, node> activecnode;
node_map<bmlist_item> ptritem;
```

Figura 4.2: Declaração de uma PC-árvore alterada na implementação do algoritmo de SH.

4.3 Algoritmo de SH

Seja G o grafo conexo dado. O algoritmo de SH constrói uma bp-árvore T_0 de G e passa T_0
então à parte iterativa, onde os vértices de G são examinados um após o outro, de acordo com uma bp-ordenação em relação a T_0.

Cada iteração começa com o conjunto L dos vértices não-examinados e uma árvore enraizada T tal que $T[L] = T_0[L]$ e $T[V_G \setminus L]$ é uma PC-árvore alterada associada a uma moldura M em G do grafo $G[V_G \setminus L]$. (No algoritmo de SH, $G[V_G \setminus L]$ e M podem não ser conexos. Algumas vezes, diz-se, para simplificar, que T é a PC-árvore.) Inicialmente, $L = V_G$ e $T = T_0$. Seja w o vértice em L com o menor bp-número; w é o vértice examinado nesta iteração.

Sejam $X := \{u \in V_M : uw \in E_G\}$ e $Y := \{u \in V_M :$ existe $v \in V_G \setminus (V_H \cup \{w\}), uv \in E_G\}$. X, Y
Em cada iteração do algoritmo de SH, busca-se uma XY-obstrução. A primeira etapa nessa busca é a determinação dos chamados terminais de T (definidos na seção 4.6), que indicarão os

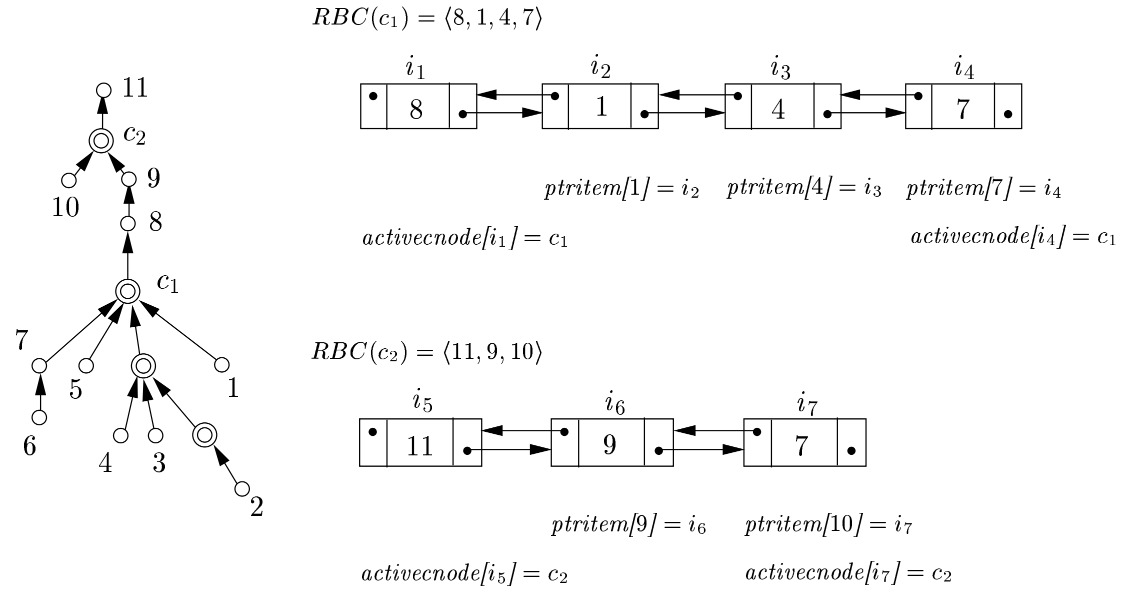


Figura 4.3: Estrutura de dados de uma PC-árvore alterada.

candidatos a ponta dos XY-caminhos em M. Se, na determinação de algum dos XY-caminhos, uma XY-obstrução é detectada, o algoritmo pára. Caso contrário, a PC-árvore T é atualizada de forma a incluir os novos blocos e vai-se para a próxima iteração. Isto é feito pela função *is_planar* (figura 4.4), que é descrita a seguir.

As inicializações são realizadas pela função *pre_processing*, que constrói uma bp-árvore T_0 de G e, para cada vértice de G, atribui um bp-número (variável *dfspos_num[ ]*). Além disso, essa função calcula outras informações importantes para o algoritmo de SH, que são mencionadas mais adiante, conforme se tornam necessárias.

Na próxima seção, encontra-se uma descrição das informações associadas a cada vértice de G (valores b e h) que são utilizadas para calcular os terminais e XY-caminhos ou XY-obstruções.

4.4 Valores de b para cada nó

Durante a execução do algoritmo de SH, cada nó u na PC-árvore corrente T mantém o bp-número do vértice mais "alto" em T_0 que é vizinho, no grafo G, de um descendente de u em T.
$h(u)$ Mais precisamente, seja $h(u)$ o maior bp-número (na bp-árvore T_0) de um vizinho de u em G.
$b(u)$ Durante a execução, o algoritmo mantém, para cada nó u de T, um número $b(u)$, que é o maior valor de $h(v)$ tal que v é um descendente de u na PC-árvore corrente T (figura 4.5).

Na implementação descrita aqui, os valores de h e de b são representados pelas variáveis

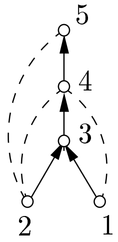
55 ⟨função principal testar planaridade 55⟩≡

```
bool sh_planar_impl::is_planar(graph& G)
{ int n = G.number_of_nodes();
  pre_processing(G);
  node_array<list<node> > terminal_nodes(G);
  bool planar = true;
  for (int i = 1; i < n ∧ planar; i++) {
      node w = node_with_dfspos[i];
      list<node> list_of_components;
      find_terminals(G, w, list_of_components, terminal_nodes);
      node comp;
      forall(comp, list_of_components)
        if (terminal_nodes[comp].size() > 0) {
            node new_cnode = G.new_node();
            dfspos_num[new_cnode] = -(++total_cnodes); // marks as c-node;
            if (find_obstruction(G, w, terminal_nodes[comp])) { planar = false; break; }
            (1) % marca de onde entrará uma parte da construção do desenho
            set_info_for_new_cnode(G, w, new_cnode, terminal_nodes[comp]);
          }
    }
  (2) % marca de onde entrará uma parte da construção do desenho
  node u;
  forall_nodes(u, G) if (is_cnode(u)) G.del_node(u); // restores the graph;
  return planar;
}
```

Figura 4.4: Código da função *is_planar*.

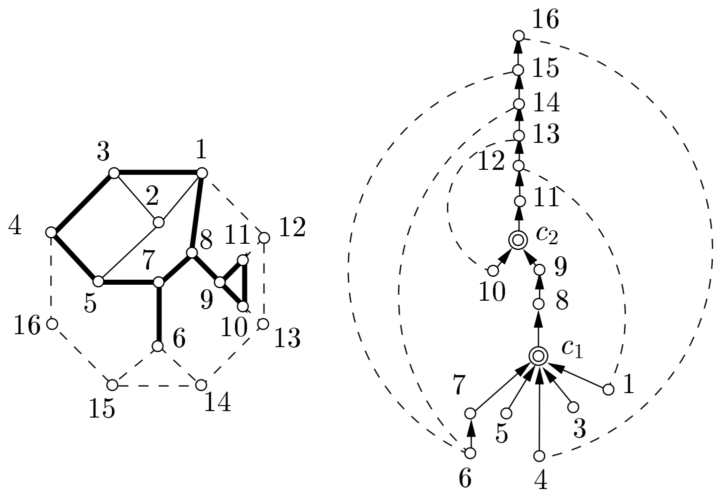
largest_neighbor[ ] e *label_b[ ]*, respectivamente. O cálculo de h é feito somente uma vez, durante o pré-processamento do algoritmo, pela função *pre_processing*, onde também é calculado o valor inicial de b de cada vértice.

Os valores de b e de h são extremamente utilizados durante o cálculo dos terminais, na seção 4.6, e na determinação dos XY-caminhos ou detecção de uma XY-obstrução, na seção 4.7. Por exemplo, seja c um C-nó ativo em T e seja u um vértice em $\mathrm{RBC}(c)$. Para determinar se u enxerga $L \setminus \{w\}$ através de $V_G \setminus V_{B_c}$, basta verificar se $b(u) > \mathrm{bp}(w)$, sendo w o primeiro elemento em L.



$h(1) = 4$	$b(1) = 4$
$h(2) = 5$	$b(2) = 5$
$h(3) = 4$	$b(3) = 5$
$h(4) = 5$	$b(4) = 5$
$h(5) = 4$	$b(5) = 5$

(a)



$h(1) = 12$	$b(1) = 12$
$h(3) = 4$	$b(3) = 4$
$h(4) = 16$	$b(4) = 16$
$h(5) = 7$	$b(5) = 7$
$h(6) = 15$	$b(6) = 15$
$h(7) = 8$	$b(7) = 15$
	$b(c_1) = 16$
$h(8) = 9$	$b(8) = 16$
$h(9) = 11$	$b(9) = 16$
$h(10) = 13$	$b(10) = 13$
	$b(c_2) = 16$
$h(11) = 12$	$b(11) = 16$
$h(12) = 13$	$b(12) = 16$
$h(13) = 14$	$b(13) = 16$
$h(14) = 15$	$b(14) = 16$
$h(15) = 16$	$b(15) = 16$
$h(16) = 15$	$b(16) = 16$

(b)

Figura 4.5: (a) Uma bp-árvore de um grafo juntamente com as demais arestas e os valores de h e b para cada nó. (b) Um grafo G e uma moldura M de $G[\{1, \ldots, 11\}]$ (em linhas grossas), uma PC-árvore correspondente a M e os valores de h e de b para essa PC-árvore.

4.5 Percurso numa PC-árvore alterada

Antes de passar à determinação dos terminais, vale a pena explicar como é feito o percurso numa PC-árvore alterada.

Para garantir a linearidade do algoritmo, a PC-árvore alterada é percorrida da seguinte maneira. A partir de um P-nó, sobe-se normalmente na PC-árvore por meio dos apontadores pai, exceto quando chega-se a um P-nó u que é filho de um C-nó. Neste ponto, a função *find_ active_ cnode* é acionada (figura 4.6), com o P-nó u como parâmetro. Essa função devolve um apontador para o C-nó ativo que é o pai real de u. Seja c o pai real de u. Antes da primeira execução dessa função, cada vértice em RBC(c) tem a marca NOT_VISITED na variável *state[ ]*. Após uma execução dessa função, cada vértice percorrido neste processo está marcado como VISITED_IN_RBC e para cada um desses vértices, exceto pelo pai de c, seu ponteiro pai passa a apontar para c.

57 ⟨procura c-nó ativo 57⟩≡

```
node sh_planar_impl::find_active_cnode(node u, node w, list<node>& nl)
{ list<node> traversed_nodes_in_RBC;
  if (state[u] ≠ NOT_VISITED) return parent[u];
  bmlist_item it1 = ptritem[u];
  state[u] = VISITED_IN_RBC; traversed_nodes_in_RBC.append(u);
  bmlist_item it = search_RBC(1, it1, w, traversed_nodes_in_RBC);
  if (it ≡ nil) it = search_RBC(0, it1, w, traversed_nodes_in_RBC);
  node v = *(const node*)(&it→e), cnode;
  if (it→ptr1 ≠ nil ∧ it→ptr2 ≠ nil) cnode = parent[v];
  else cnode = activecnode[it];
  node first = RBC[cnode].contents(RBC[cnode].first_item());
  forall(v, traversed_nodes_in_RBC) {
     if (v ≠ first) {
        if (v ≠ u)  nl.append(v);
        parent[v] = cnode;
      }
     else state[v] = NOT_VISITED;
   }
  return cnode;
}
```

Figura 4.6: Código da função *find_ active_ cnode*.

A busca do pai real do P-nó u é feita da seguinte maneira. Parte-se da célula de u indicada pelo apontador *ptritem[u]* (variável *it1* na função *find_ active_ cnode*), e percorre-se o RBC em questão numa direção arbitrária, até atingir:

(c1) ou uma das extremidades do RBC;

(c2) ou uma célula que corresponde a um vértice marcado como VISITED_IN_RBC;

(c3) ou uma célula que corresponde a um vértice que enxerga L através de $V_G \setminus V_{B_c}$.

Seja q a célula atingida. Se q satisfaz (c1) ou (c2), então o C-nó ativo é indicado ou pelo apontador *activecnode[q]*, ou pelo seu apontador pai, respectivamente. Caso contrário, a rotina percorre o RBC a partir de u, agora na outra direção, como descrito anteriormente, exceto ignorando a condição (c3). Observe que neste segundo percurso, se ocorrer a condição (c3) então uma obstrução do tipo (o1) é detectada. Este fato é ignorado neste ponto e é tratado somente durante a busca de um XY-caminho (seção 4.7). Isto é feito desta maneira devido a parte do algoritmo que encontra um subgrafo de Kuratowski no grafo, descrita no capítulo 6.

O percurso do RBC numa dada direção é feito pela função *search_RBC* (figura 4.7) e uma ilustração do processo de determinação de um C-nó ativo se encontra na figura 4.8.

58 ⟨função auxiliar para procurar c-nó ativo 58⟩≡

```
bmlist_item sh_planar_impl::search_RBC(int dir, bmlist_item it, node w, list<node>& nl)
{ if (it ≠ nil ∧ (it→ptr1 ≡ nil ∨ it→ptr2 ≡ nil)) return it;
  bmlist_item prev = it, aux1 = it→ptr1, aux2 = it→ptr2;
  if (dir ≡ 1) leda_swap(aux1, aux2);
  it = aux1;
  node u = *(const node*)(&it→e);
  int b = label_b[u];
  while (it ≠ nil ∧ (b ≤ dfspos_num[w] ∨ dir ≠ 1) ∧ state[u] ≡ NOT_VISITED) {
     aux1 = it→ptr1;
     if (aux1 ≡ prev) aux1 = it→ptr2;
     prev = it; it = aux1;
     u = *(const node*)(&prev→e);
     state[u] = VISITED_IN_RBC; nl.append(u);
     if (it ≠ nil) { u = *(const node*)(&it→e); b = label_b[u]; }
   }
  if (it ≡ nil) return prev;
  else if (state[u] ≠ NOT_VISITED ∨ (it→ptr1 ≡ nil ∨ it→ptr2 ≡ nil)) return it;
  else return nil;
}
```

Figura 4.7: Código da função *search_RBC*.

Para simplificar a exposição e o entendimento dos próximos procedimentos, deste ponto até a análise do algoritmo, que é feita na seção 4.9, a PC-árvore alterada é muitas vezes confundida

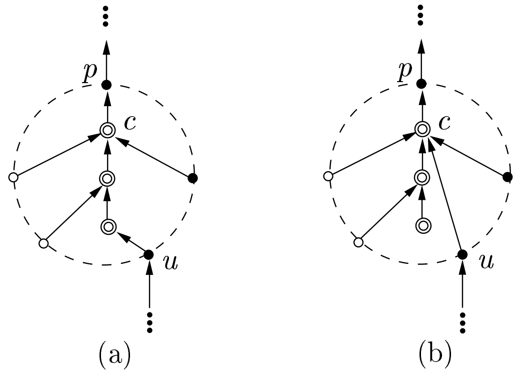


Figura 4.8: Cálculo do C-nó ativo c a partir de um P-nó u. O RBC(c) é ilustrado por linhas tracejadas e p é o pai de c. (a) Os vértices do RBC(c) que foram percorridos durante o processo estão ilustrados por círculos pretos. (b) Cada vértice percorrido, exceto p, tem seu pai atualizado, apontando para c.

com a PC-árvore original e fica implícito que a implementação está lidando, na verdade, com uma PC-árvore alterada.

4.6 Determinação dos terminais

As arestas de G que não estão em T_0 são chamadas de **arestas de retorno**. (Na figura 4.5, as arestas de retorno são ilustradas por linhas tracejadas.) Seja R o grafo obtido de T pela inclusão das arestas de retorno com uma ponta em L e outra em $V_G \setminus L$ (PC-árvore da figura 4.5(b)), e seja $L' := L \setminus \{w\}$. Um nó t de $T - L$ é um **terminal** se, em R, L'

(nt1) t enxerga $\{w\}$ e L' através de V_{T_t} (onde T_t é a subárvore de T enraizada em t);

(nt2) nenhum descendente próprio t' de t em T enxerga $\{w\}$ e L' através de $V_{T_{t'}}$.

Na figura 4.5(b), para $L = \{12, \ldots, 16\}$ e $w = 12$, o C-nó c_1 é terminal.

Os valores de b, definidos para cada nó em T, podem ser usados na determinação dos terminais, como mostra o seguinte lema.

Lema 4.1: *Um terminal de $T - L$ é um nó t que satisfaz as seguintes propriedades:*

(t1) $b(t) > \mathrm{bp}(w)$;

(t2) t é vizinho de w em G ou t tem um descendente próprio em T que é vizinho de w em G;

(t3) nenhum descendente próprio de t satisfaz as propriedades (t1) e (t2) simultaneamente.

Demonstração: A seguir, é demonstrado que vale (nt1) se e somente se valem (t1) e (t2).

Seja t um nó de $T - L$ que satisfaz (t1). Então t tem um descendente u em T tal que $h(u) > \text{bp}(w)$, ou seja, u é vizinho de um vértice em L' e portanto t enxerga L' através de V_{T_t}. Se t satisfaz (t2), então t tem um descendente u em T tal que $h(u) = \text{bp}(w)$, ou seja, u é vizinho de w e portanto t enxerga $\{w\}$ através de V_{T_t}. Portanto, dadas as condições (t1) e (t2), a condição (nt1) da definição de nó terminal é satisfeita.

Agora suponha que t satisfaz a condição (nt1). Se t enxerga L' através de V_{T_t}, então t tem um descendente u em T tal que $h(u) > \text{bp}(w)$, ou seja, $b(t) > \text{bp}(w)$. Se t enxerga $\{w\}$ através de V_{T_t}, então t é vizinho de w em G ou t tem um descendente em T que é vizinho de w em G. Portanto, dada a condição (nt1), as condições (t1) e (t2) são satisfeitas.

Claramente as condições (nt2) e (t3) são equivalentes. ∎

Na figura 4.5(b), em que c_1 é um terminal para $L = \{12, \ldots, 16\}$ e $w = 12$, veja que $b(c_1) = 16 > 12 = \text{bp}(w)$ (condição (t1) do lema 4.1), 1 é vizinho de w e é filho de c_1 (condição (t2)), e c_1 satisfaz (t3).

Observe que, se $L = \{w\}$ num dado momento da execução do algoritmo, então w é a raiz da PC-árvore corrente T e tem-se que, para cada nó t de T, vale que $b(t) \leq \text{bp}(w)$. Neste caso, não existe terminal em $T - L$ e o grafo em questão é planar. Observe também que se, num dado momento do algoritmo, se não há terminais em $T - L$, então não existe XY-obstrução na moldura corrente e o algoritmo passa para a próxima iteração.

O lema 4.1 sugere o seguinte processo para determinar os terminais de $T - L$. Dado w tal que w não é a raiz de T, para cada vizinho u de w que é descendente de w em T (ou seja, tal que $\text{bp}(u) < \text{bp}(w)$), percorra T a partir de u, subindo em direção a w em T, até o primeiro nó v tal que $b(v) > \text{bp}(w)$, caso exista, marcando v como terminal.

Infelizmente, um nó v marcado por esse processo pode não ser um terminal pois esse processo pode ter marcado também um descendente próprio de v. Na figura 4.5(a), considere $w = 4$. Partindo de 1, o processo marca 3 como um terminal. Partindo de 2, o processo marca 2 como terminal. Por causa de 2, o nó 3 não é um terminal. Além disso, pelo processo acima, um vértice pode ser desnecessariamente examinado várias vezes, conforme ilustrado na figura 4.9. Para $w = v_1$, $i = 2, \ldots, j$, v_i é examinado k vezes no processo de encontrar os terminais, onde v_2 é o único terminal.

Para consertar o processo e torná-lo mais rápido, marca-se todos os nós de $T - L$ inicialmente como não-visitados, marca-se w como visitado e percorre-se T a partir de u em direção a w,

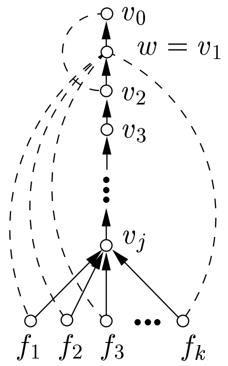


Figura 4.9: As setas indicam o pai de cada nó da PC-árvore ilustrada.

parando quando atingir-se um nó v tal que

(1) v já foi visitado ou

(2) $b(v) > \mathrm{bp}(w)$.

No caso (1), não há nada a fazer além de marcar os nós percorridos como visitados. No caso (2), inclui-se v numa lista de candidatos a terminais, marca-se em v a posição de v nessa lista e prossegue-se subindo até um nó v' já visitado. Ao chegar-se em v', remove-se v' da lista de candidatos a terminais se ele estiver lá. Além disso, marca-se todos os nós percorridos como visitados. Ao final desse processo, a lista de candidatos a terminais é exatamente a lista de terminais dessa iteração.

O cálculo dos terminais é feito pela função *find_terminals* (figura 4.10), que é descrita a seguir.

A função *find_terminals* recebe como entrada o grafo G e o vértice w, que é o vértice correntemente examinado pelo algoritmo de SH, além da PC-árvore T da iteração corrente (variáveis globais *parent[ ]*, *RBC[ ]*, *ptritem* e *activecnode*).

O parâmetro de saída *list_of_components* da função *find_terminals* é a lista das raízes das árvores de $T - L$ que contém algum terminal. O parâmetro de saída *terminal_nodes[ ]* guarda a lista dos terminais (e candidatos a terminais durante o processo) de cada componente de $T - L$.

A variável global *state[ ]* contém a marca VISITED, TERMINAL, NOT_VISITED ou VISITED_IN_RBC para cada nó. A marca VISITED_IN_RBC é usada na determinação dos C-nós ativos, conforme descrita na seção 4.5. Na fase inicial do algoritmo de SH, *state[u]* é inicializado com NOT_VISITED para cada nó u. No final da função *find_terminals*, para cada nó u que teve seu estado alterado, *state[u]* é restaurado para NOT_VISITED. Durante o processo, a marca TERMINAL indica os candidatos a terminais.

62 ⟨cálculo de terminais 62⟩≡

```
void sh_planar_impl::find_terminals(graph& G, node w, list<node>& list_of_components,
                                     node_array<list<node> >& terminal_nodes)
{ map<node, list_item> terminal_nodes_item;
  map<node, node> component_of;
  list<node> traversed_nodes;
  component_of[w] = nil; state[w] = VISITED; traversed_nodes.append(w);
  edge e;
  forall_out_edges(e, w) {
     node u = target(e);
     if (dfspos_num[u] < dfspos_num[w] ∧ is_back_edge(e)) {
        list<node> S;
        node terminal_node = nil;
(1)     while (state[u] ≡ NOT_VISITED) {
           if (is_cnode(parent[u])) {
              node cnode = find_active_cnode(u, w, traversed_nodes); counter[cnode]++;
           }
           if (terminal_node ≡ nil ∧ label_b[u] > dfspos_num[w]) {
              state[u] = TERMINAL; terminal_node = u;
           }
           else state[u] = VISITED;
           traversed_nodes.append(u); S.push(u); u = parent[u];
        } // end while;
        if (u ≡ w) {
           node c = S.pop(); component_of[c] = c; list_of_components.append(c);
        }
(2)     while (¬S.empty()) {
           node u = S.pop(); component_of[u] = component_of[parent[u]];
        }
        if (terminal_node ≠ nil) {
           node c = component_of[terminal_node];
           terminal_nodes_item[terminal_node] = terminal_nodes[c].append(terminal_node);
           if (state[u] ≡ TERMINAL) {
              state[u] = VISITED; terminal_nodes[c].del(terminal_nodes_item[u]);
           }
        }
     } // end if (dfspos_num[u] < dfspos_num[w] && is_back_edge(e));
  } // end forall_out_edges;
  node u;
  forall (u, traversed_nodes) state[u] = NOT_VISITED;
}
```

Figura 4.10: Código da função *find_terminals*.

Para cada candidato v a terminal, a variável *terminal_ nodes_ item[v]* guarda a posição de v na lista *terminal_ nodes[ ]* e, para cada C-nó ativo d, *counter[d]* contém o número de filhos reais de d que são visitados por esse processo (é o número de filhos reais de d que têm um descendente em T que é vizinho de w em G). Para cada vértice visitado u, guarda-se na variável auxiliar *component_ of[u]* a raiz da árvore de $T - L$ que contém u. A variável S é uma variável auxiliar que guarda os vértices visitados até determinar-se o componente em que estão.

Após a construção da lista de terminais para cada componente de $T - L$ pela função acima, uma iteração do algoritmo de LEC é aplicada para cada componente com pelo menos um terminal.

4.7 Busca por uma XY-obstrução

Seja M uma moldura em G de $H := G[V_G \setminus L]$ dada por uma PC-árvore T. Seja w o vértice de menor bp-número em L. Sejam $X := \{u \in V_M : uw \in E_G\}$ e $Y := \{u \in V_M :$ existe $v \in V_G \setminus (V_H \cup \{w\}), uv \in E_G\}$. O algoritmo de SH verifica a presença de uma XY-obstrução em cada componente de $T[V_G \setminus L]$ com terminais. Isto é feito pela função *find_ obstruction* e seu código é ilustrado nas figuras 4.11 e 4.12.

```
63   ⟨procura uma obstrução 63⟩≡                                                64 ▷
    bool sh_planar_impl::find_obstruction(graph& G, node w, list<node>& terminal_nodes)
    { cnode_of_possible_k33_obstruction = nil; // reset global variable;
     list<node> list_terminal = terminal_nodes;
     switch(terminal_nodes.size()) {
         case 1:
           { node t1 = list_terminal.pop();
             if (test_obstruction_from_terminal(G, w, t1, w)) return true;
             if (cnode_of_possible_k33_obstruction ≠ nil)
               { node v = find_node_with_label_b_greater_than_dfs_w(
                           false, G, cnode_of_possible_k33_obstruction, w);
                 if (v ≠ nil) {
                     (2) % marca de onde entrará a construção da lista de arcos da obstrução
                     return true;
                   }
               }
           }
           break;
```

Figura 4.11: Código da função *find_ obstruction*.

Existem três casos a serem considerados: o caso em que há um único terminal, o caso em que

64 ⟨procura uma obstrução 63⟩+≡ ◁63

```
    case 2:
     { node t1 = list_terminal.pop(), t2 = list_terminal.pop(),
         m = lca_between_term_nodes(t1, t2);
      list_current_terminal_nodes.append(t1); list_current_terminal_nodes.append(t2);
      node t12 = last_pnode(t1, m), t22 = last_pnode(t2, m);
      if (test_obstruction_from_terminal(G, w, t1, t12)) return true;
      if (cnode_of_possible_k33_obstruction ≠ nil) {
         (2) % marca de onde entrará a construção da lista de arcos da obstrução
         return true;
      }
      if (test_obstruction_from_terminal(G, w, t2, t22)) return true;
      if (cnode_of_possible_k33_obstruction ≠ nil) {
         (2) % marca de onde entrará a construção da lista de arcos da obstrução
         return true;
      }
      if (is_cnode(m) ∧ test_cnode_counter(G, m, w, t12, t22)) {
         (1) % marca de onde entrará a construção da lista de arcos da obstrução
         return true;
       }
      node v = find_node_with_label_b_greater_than_dfs_w(true, G, m, w);
      if (v ≠ nil){
         (3) % marca de onde entrará a construção da lista de arcos da obstrução
         return true;
       }
     }
     break;
    default:
     {
      (4) % marca de onde entrará a construção da lista de arcos da obstrução
      return true;
     }
   } // end switch;
  return false;
 }
```

Figura 4.12: (Continuação) Código da função *find_obstruction*.

há dois terminais e o caso em que há três ou mais terminais no componente.

O caso em que há um terminal, digamos t, é tratado da seguinte forma. Seja Q o caminho Q
de t a w em T. A função *test_obstruction_from_terminal* (figura 4.13) examina cada C-nó ativo em Q, subindo em T e pára quando ocorrer uma das situações ilustradas nas figuras 4.14(a) e (b). Caso seja detectada a situação da figura 4.14(a), uma obstrução do tipo (o1) foi encontrada. Caso contrário, se existe um C-nó ativo em Q nas condições da figura 4.14(b), a variável global *cnode_of_possible_k33_obstruction* passa a apontar para o primeiro C-nó detectado nessas condições. Seja d tal C-nó e seja Q' o subcaminho de Q do pai de d até o filho de w em Q. Q'
Neste caso, é preciso fazer o seguinte teste do caminho Q'.

65 ⟨teste partindo de um terminal 65⟩≡

```
bool sh_planar_impl::test_obstruction_from_terminal(graph& G, node w, node t1, node u)
{ node v = t1, pred_v = nil;
  while (v ≠ u) {
    if (is_cnode(v)) { // v is an active c-node;
      if (test_cnode_counter(G, v, w, pred_v, nil)) {
        (1) % marca de onde entrará a construção da lista de arcos da obstrução
        return true;
      }
      if (cnode_of_possible_k33_obstruction ≠ nil) break;
      v = parent[v];
      if (v ≡ u) break;
    }
    pred_v = v; // never c-node;
    v = parent[v];
  }
  return false;
}
```

Figura 4.13: Código da função *test_obstruction_from_terminal*.

Seja v um nó em Q' e seja $T_v^* := T_v - T_{v'}$, onde v' é o filho de v em Q. É preciso verificar se existe um nó v em Q' tal que T_v^* contém um vizinho de um vértice em $L\setminus\{w\}$ (figura 4.14(c)). Isto é feito pela função *find_node_with_label_b_greater_than_dfs_w* (figura 4.17) cuja descrição encontra-se mais adiante. Se existir tal v, essa função devolve um apontador para v, se v é um P-nó, ou para um filho real de v em T_v^*, caso contrário. (O vértice v, caso exista, é usado na construção da lista de arcos da obstrução, descrita no capítulo 6.)

Para o caso de dois terminais, digamos t_1 e t_2, seja m o ancestral mais baixo de t_1 e t_2 em T e sejam t_1' e t_2' definidos como a seguir. Para $i = 1, 2$, $t_i' = m$ se m é um P-nó. Caso contrário, t_i' é o filho real de m que é ancestral de t_i. (Os cálculos de t_i' são feitos pela função *last_pnode*.)

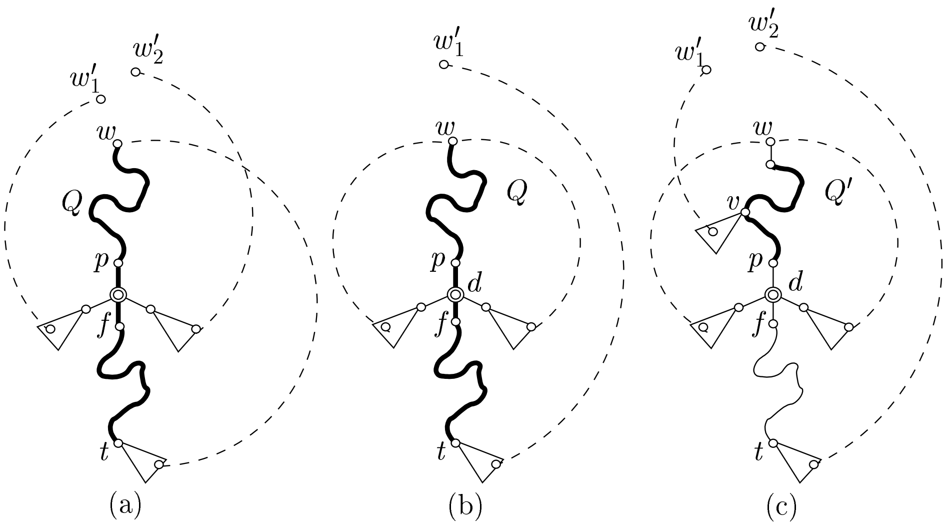


Figura 4.14: Teste de obstrução para o caso de um terminal t, onde p é o pai do C-nó que faz parte de uma obstrução do tipo (o1), w é o vértice correntemente sendo examinado pelo algoritmo de SH e w'_1 e w'_2 são ancestrais de w na PC-árvore. Os caminhos Q e Q' são ilustrados por curvas grossas. (a) Uma obstrução do tipo (o1). (b) d é o primeiro C-nó ativo, subindo em Q, com dois filhos, em cada um dos lados determinados por f e p, com um descendente que é vizinho de w. (c) Uma obstrução do tipo (o1).

Seja Q_i, para $i = 1, 2$, o caminho de t_i a t'_i em T. Analogamente ao caso de um terminal, submete-se Q_i à função *test_ obstruction_ from_ terminal*. Após o teste de cada Q_i, se a variável global *cnode_ of_ possible_ k33_ obstruction* não é *nil*, então há uma obstrução do tipo (o1) devida à presença do terminal que não está em Q_i. Além disso, se m é um C-nó, m é submetido à função *test_ cnode_ counter* para verificar se m não faz parte de uma obstrução do tipo (o1).

Finalmente, deve-se testar o caminho Q' (figura 4.15), do pai p de m a um filho de w, da mesma maneira como descrita para o caso de um terminal.

Para o caso em que há três ou mais terminais num mesmo componente de $T - L$, tem-se uma obstrução do tipo (o3).

Teste de um C-nó

Testar se um C-nó ativo d faz parte de uma obstrução do tipo (o1) significa, informalmente, determinar se é possível ou não escolher um dos lados do bloco B_d na moldura corrente para fazer parte do XY-caminho. Conforme descrito na próxima seção, para cada C-nó ativo d, a lista RBC(d) contém apenas os vértices de B_d cujo valor de b é não inferior a bp(w).

O teste para detectar o caso mais simples de obstrução (figura 4.14(a)) é feito pela função

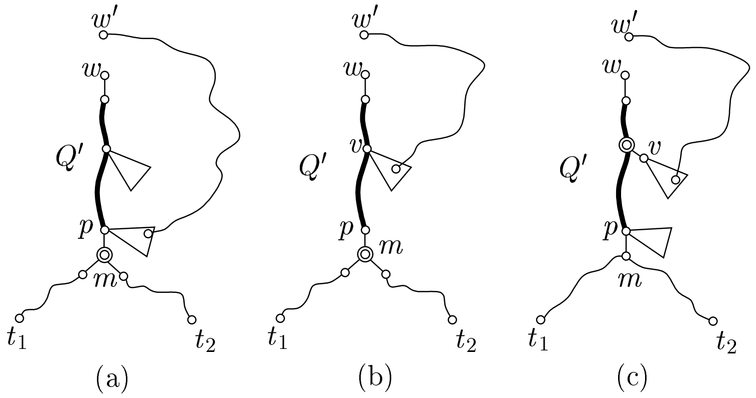


Figura 4.15: Sejam t_1 e t_2 dois terminais, m o ancestral comum de t_1 e t_2 em T com menor bp-número, p o pai de m e w' um ancestral de w. O caminho Q' em T é ilustrado por curvas grossas. Em (a) e (b), m é um C-nó que faz parte de uma obstrução do tipo (o2). Em (c) é ilustrada uma situação em que há uma obstrução do tipo (o3).

test_ cnode_ counter, cujo código se encontra na figura 4.16. Esta função calcula o número de filhos reais de um C-nó ativo d, consecutivos e adjacentes ao pai de d no RBC(d), que possuem um descendente que é um vizinho de w. Mais precisamente, dado um C-nó ativo d, esta função percorre o RBC(d), partindo do pai de d (primeiro elemento dessa lista), para os dois lados, até parar num vértice de cada lado, cujo valor de b é superior ao bp(w). Sejam l e r tais vértices, respectivamente. Durante este processo, calcula-se o número de vértices cujo valor de b é igual ao bp(w). Se l e r são ancestrais de terminais, então eles também são contabilizados na soma. Ao final da função, a soma obtida é comparada com o valor de *counter[d]*. Caso a soma seja diferente de *counter[d]*, uma obstrução do tipo (o1) foi detectada.

Somente com a função *test_ cnode_ counter*, o algoritmo não consegue detectar o caso ilustrado na figura 4.14(c). Para lidar com este caso, faz-se a variável global *cnode_ of_ possible_ k33_ obstruction* apontar para o primeiro C-nó em Q, subindo em T, onde ocorre a situação ilustrada na figura 4.14(b), que é detectada pela função *test_ cnode_ counter* quando os dois lados do RBC são percorridos durante a contagem dos filhos. Neste caso, é preciso fazer uma verificação em Q', conforme é descrito a seguir.

Teste do caminho Q'

O teste do caminho Q' para o caso de um ou dois terminais (figuras 4.14(b) e 4.15, respectivamente) é feito pela função *find_ node_ with_ label_ b_ greater_ than_ dfs_ w*, cujo código se encontra na figura 4.17.

Para cada nó u em Q', seja u' o filho de u no caminho Q. Esta função faz com que u' deixe

68 ⟨teste de um c-nó 68⟩≡

```
bool sh_planar_impl::test_cnode_counter(graph& G, node cnode, node w, node w1, node w2)
{ if (RBC[cnode].size() ≤ 2) return false;
  bmlist_item it1 = RBC[cnode].first_item(), it1l = RBC[cnode].cyclic_pred(it1, nil),
    it1r = RBC[cnode].cyclic_succ(it1, nil);
  node l = RBC[cnode].contents(it1l), r = RBC[cnode].contents(it1r), ul = nil, ur = nil;
  int count = 1;    // starts counting parent[cnode];
  bmlist_item a, s = it1;
  while (label_b[l] ≡ dfspos_num[w]) // goes to the left;  (2)
    { ul = l; count++; a = it1l;
      it1l = RBC[cnode].cyclic_pred(it1l, s); s = a;
      l = RBC[cnode].contents(it1l);
    }
  bmlist_item p = it1;
  while (label_b[r] ≡ dfspos_num[w]) // goes to the right; (3)
    { ur = r; count++; a = it1r;
      it1r = RBC[cnode].cyclic_succ(it1r, p); p = a;
      r = RBC[cnode].contents(it1r);
    }
  if (l ≡ w1 ∨ l ≡ w2) count++;
  if (r ≠ l ∧ (r ≡ w1 ∨ r ≡ w2)) count++;
  if (count < counter[cnode]) return true;                (1)
  node f = nil;
  if (ul ≠ nil ∧ ur ≠ nil)
    f = l; // or f = r;
  else if (ul ≠ nil ∧ w1 ≠ nil ∧ w1 ≠ l)
     { f = l; ur = w1; }
  else if (ur ≠ nil ∧ w1 ≠ nil ∧ w1 ≠ r)
    { f = r; ul = w1; }
  if (cnode_of_possible_k33_obstruction ≡ nil ∧ f ≠ nil)
    cnode_of_possible_k33_obstruction = cnode;
  return false;
}
```

Figura 4.16: Código da função *test_cnode_counter*.

69 ⟨testa caminho Q' 69⟩≡

```
node sh_planar_impl::find_node_with_label_b_greater_than_dfs_w(bool save_last_node_traversed,
                                                    graph& G, node u, node w)
{ node result = nil, v_pred = u, v = parent[u];
  list<node> nl, nl2; // to restore parent, label_b and node_label_b;
  map<node, node> p;
  map<node, int> b;
  p[u] = parent[u]; // saves info;
  nl.append(u);
  parent[u] = nil;
  while (v ≠ w) {
     if (is_cnode(v)) {
        node v1;
        myforall(v1, RBC[v])
          if (label_b[v1] > dfspos_num[w] ∧ v1 ≠ u ∧ v1 ≠ parent[v]) { result = v1; break; }
       }
     else {
        nl2.append(v); b[v] = label_b[v]; // saves infos;
        update_label_b(v);
        if (label_b[v] > dfspos_num[w]) { result = v; break; }
       }
     v_pred = v; v = parent[v]; p[v_pred] = parent[v_pred];
     nl.append(v_pred); parent[v_pred] = nil;
    } // while
  forall(v, nl) parent[v] = p[v];  // restores parent;
  if (result ≠ nil) return result;
  forall(v, nl2) label_b[v] = b[v];  // restores info if result is nil;
  // to update label_b of w correctly in future iteration
  // (v_pred is the child of w in T that belongs to midpath);
  if (save_last_node_traversed ∧ v_pred ≠ u)
     last_node_in_midpath = v_pred; // see set_info_for_new_cnode;
  else last_node_in_midpath = nil;
  return nil;
}
```

Figura 4.17: Código da função *find_node_with_label_b_greater_than_dfs_w*.

de ser, temporariamente, um filho de u. Se u é um P-nó, então o valor de $b(u)$ é recalculado. Se o novo valor $b(u)$ é superior ao $\mathrm{bp}(w)$, então a função devolve este P-nó. Se u é um C-nó, então a função devolve, caso exista, um P-nó v que é filho de u tal que $b(v) > \mathrm{bp}(w)$ e v não está em Q. Ao final da função, os pais e valores de b são restaurados.

A variável global *last_node_in_midpath* guarda o último nó de Q' quando *save_last_node_traversed* é verdadeiro. Isto é importante para que o recálculo dos valores de b, feitos pela rotina *update_label_b*, cujo código é ilustrado na figura 4.18, funcione corretamente. A variável *last_node_in_midpath* é usada durante o cálculo do RBC para um novo C-nó, explicado na próxima seção.

Cálculo dos valores de b

À medida em que a PC-árvore é atualizada durante a execução do algoritmo de SH, é necessário atualizar-se os valores de b para os vértices que farão parte do RBC do novo C-nó. Isto é feito pela rotina *update_label_b* (figura 4.18).

70 ⟨atualiza rótulos b 70⟩≡

```
void sh_planar_impl::update_label_b(node u)
{ label_b[u] = largest_neighbor[u];
  node v;
  if (¬children_in_T0[u].empty())  v = children_in_T0[u].contents(children_in_T0[u].first());
  else return;
  while (v ≠ nil) {
      if (parent[v] ≠ nil ∧ is_cnode(parent[v]) ∧ parent[parent[v]] ≡ u)
        { v = parent[v]; break; }
      if (parent[v] ≡ u) break;
      // v became sibling of u;
      children_in_T0[u].del(children_in_T0[u].first());
      if (¬children_in_T0[u].empty()) v = children_in_T0[u].contents(children_in_T0[u].first());
      else v = nil;
    }
  if (v ≠ nil ∧ label_b[u] < label_b[v]) label_b[u] = label_b[v]; //restores info;
}
```

Figura 4.18: Código da função *update_label_b*.

Para que esse processo seja eficiente, uma lista dos filhos de cada vértice na árvore T_0, em ordem não-decrescente do valor de b, é construída durante o pré-processamento. Esta lista é dada pela variável global *children_in_T0[ ]*. A idéia é que, para cada vértice u, os vértices de *children_in_T0[u]* que se tornaram irmãos de u na PC-árvore corrente T são removidos de *children_ in_ T0[u]*. Seja v o primeiro elemento dessa lista após a remoção dos vértices que se tornaram irmãos de u, caso exista. Na atualização, se $b(v) > h(u)$, então $b(u)$ passa a ser $b(v)$. Se a lista *children_ in_ T0[u]* está vazia ou $b(v) \leq h(u)$, então $b(u)$ passa a ser $h(u)$.

4.8 Construção de um novo C-nó

Neste ponto do algoritmo, um componente K de $T - L$ com um ou dois terminais está em foco e sabe-se que existe um XY-caminho P em K. Essa fase consiste na criação de um C-nó c para representar o bloco B_c de $H + w$ que contém vértices de K. Ela consiste em duas tarefas: a atualização da árvore T e do vetor b e a construção do RBC(c). Tanto a atualização de T e de b quanto a construção de RBC(c) são feitas pela rotina *set_ info_for_ new_ cnode*, cujo código é ilustrado na figura 4.19.

71 ⟨constrói novo c-nó 71⟩≡

```
void sh_planar_impl::set_info_for_new_cnode(graph& G, node w, node new_cnode,
                                            list<node>& terminal_nodes)
{ label_b[new_cnode] = dfspos_num[w];
  state[new_cnode] = NOT_VISITED;
  counter[new_cnode] = 0;
  parent[new_cnode] = w;
  calculate_new_RBC(G, new_cnode, w, terminal_nodes);
  if (last_node_in_midpath ≠ nil) {
     parent[last_node_in_midpath] = new_cnode;
     last_node_in_midpath = nil; // resets global variable;
   }
  RBC[new_cnode].push(w);
  bmlist_item first = RBC[new_cnode].first_item(), last = RBC[new_cnode].last_item();
  activecnode[first] = new_cnode;
  activecnode[last] = new_cnode;
}
```

Figura 4.19: Código da função *set_ info_for_ new_ cnode*.

Essa função inicializa os valores de b, *state[ ]* e *counter[ ]* do novo C-nó e faz w ser o pai desse C-nó. Em seguida, calcula o RBC para o novo C-nó através da função *calculate_ new_ RBC* (figura 4.20), descrita mais adiante. As atualizações de T e de b são feitas durante o cálculo do novo RBC. Feito isso, se o componente K tem exatamente dois terminais, então a variável *last_ node_ in_ midpath* aponta para o último vértice de Q', caso exista. Se existir tal vértice, seu pai passa a apontar para o novo C-nó. Isto é necessário para que a rotina *update_ label_ b* funcione corretamente.

Finalmente, w é inserido no início do novo RBC e, para cada uma das pontas dessa lista, a variável global *activecnode[ ]* é atualizada ou inicializada, passando a apontar para o novo C-nó.

Cálculo do novo RBC e atualização de T e de b

O cálculo do novo RBC e a atualização de T e de b são feitos pela função *calculate_ new_ RBC* (figura 4.20). Na implementação aqui apresentada, tem-se o seguinte invariante: para cada C-nó ativo c, a lista RBC(c) contém apenas os vértices de B_c cujo valor de b é maior que bp(w).

Para o caso de um terminal t em K, seja Q o caminho de t a w em T. O cálculo do novo RBC é feito através da função *calc_ new_ RBC_from_ terminal_ node* (figura 4.21). Para cada nó v em Q, subindo em T, se v é um P-nó, então seu pai passa a apontar para o novo C-nó e $b(v)$ é recalculado pela rotina *update_ label_ b*. Se o novo valor de $b(v)$ é maior do que bp(w), então v é inserido no fim do novo RBC e sua posição nessa lista é guardada em *ptritem[v]*.

Se v é um C-nó, então a rotina *add_ old_ cnode_ RBC_ to_ new_ RBC* (figuras 4.22 e 4.23) é acionada. Ela "compacta" o RBC(v) e depois verifica se é preciso inverter essa lista antes de concatená-la ao final do novo RBC. Observe que, para cada vértice u do RBC(v) compactado, não é preciso recalcular *ptritem[u]*.

Para a compactação do RBC(v), um processo similar à função *test_ cnode_ counter* é realizado: o RBC(v) é percorrido, partindo-se do pai de v (primeiro elemento dessa lista) para os dois lados, até encontrar um vértice de cada lado cujo b é maior que bp(w). (Se u é um ancestral de um terminal em RBC(v), então seu b já está atualizado para o novo C-nó, por isso foi necessário incluir as condições (1) e (2) nas figuras 4.22 e 4.23.) O pai de v e os vértices percorridos nesse processo cujo $b \leq$ bp(w) são removidos do RBC(v). Se u é um ancestral de um terminal em RBC(v), então u também é removido do RBC(v), pois ele foi inserido no RBC do novo C-nó se o novo valor de $b(u)$ é maior que bp(w).

Para verificar se é preciso inverter o RBC(v) antes da concatenação, se v é um terminal, então não é preciso fazer nada. (Na verdade, na implementação aqui apresentada, se v é um terminal e algum vértice à "esquerda" do pai de v (figura 4.24(a)) foi removido de RBC(v), então essa lista é invertida para manter a consistência com o que é feito no capítulo 5 para a construção da descrição combinatória plana.) Caso contrário, seja f o vértice em RBC(v) que está em Q. Se f é o último elemento de RBC(v) então essa lista é invertida (figura 4.24(c)). No final da função, o RBC(v) é concatenado no fim do novo RBC.

Para o caso de dois terminais (usando a mesma notação usada anteriormente), de forma análoga ao caso de um terminal, calcula-se os dois "pedaços" do novo RBC (atualizando T e b), devidos aos caminhos Q_1 e Q_2, através da função *calc_ new_ RBC_from_ terminal_ node*. Em seguida, m é adicionado ao novo RBC de maneira apropriada e o pai de m passa a apontar para

73 ⟨constrói RBC do novo c-nó 73⟩≡

```
void sh_planar_impl::calculate_new_RBC(graph& G, node new_cnode, node w,
                                       list<node>& terminal_nodes)
{ switch(terminal_nodes.size()) {
    case 1:
     { node t1 = terminal_nodes.pop();
       calc_new_RBC_from_terminal_node(new_cnode, w, t1, w, RBC[new_cnode]);
     }
     break;
    case 2:
     { node m = nil, t1 = terminal_nodes.pop(), t2 = terminal_nodes.pop(),
          aux1 = t1, aux2 = t2;
       // makes t1 with lower dfspos_num than t2;
       if (is_cnode(aux1)) aux1 = parent[aux1];
       if (is_cnode(aux2)) aux2 = parent[aux2];
       if (dfspos_num[aux1] > dfspos_num[aux2]) leda_swap(t1, t2);
       m = lca_between_term_nodes(t1, t2);
       node t12 = last_pnode(t1, m), t22 = last_pnode(t2, m);
       bmlist<node> auxnodelist;
       calc_new_RBC_from_terminal_node(new_cnode, w, t1, t12, auxnodelist);
       calc_new_RBC_from_terminal_node(new_cnode, w, t2, t22, RBC[new_cnode]);
       // adds m to RBC[new_cnode];
       if (is_cnode(m)) {
          m = active_cnode_of(false, m);
          parent[t12] = parent[t22] = new_cnode;
          update_label_b(t12); update_label_b(t22);
          if (label_b[t12] > dfspos_num[w]) ptritem[t12] = auxnodelist.append(t12);
          if (label_b[t22] > dfspos_num[w]) ptritem[t22] = RBC[new_cnode].append(t22);
          add_old_cnode_RBC_to_new_RBC(m, new_cnode, w, t22, t12, RBC[new_cnode]);
          parent[m] = new_cnode;
        }
       else {
          parent[m] = new_cnode; update_label_b(m);
          if (label_b[m] > dfspos_num[w]) ptritem[m] = RBC[new_cnode].append(m);
        }
       if (label_b[m] > label_b[new_cnode]) label_b[new_cnode] = label_b[m];
       auxnodelist.reverse(); RBC[new_cnode].conc(auxnodelist);
     }
     break;
  }
}
```

Figura 4.20: Código da função *calculate_new_RBC*.

74a ⟨constrói RBC partindo de um terminal 74a⟩≡

```
void sh_planar_impl::calc_new_RBC_from_terminal_node(node new_cnode, node w, node t1, node t2,
                                                      bmlist<node>& nodelist)
{ node  t = t1, pred_t = nil, parent_t = nil;
  while (t ≠ t2) {
    parent_t = parent[t];
    if (is_cnode(t)) {
      add_old_cnode_RBC_to_new_RBC(t, new_cnode, w, pred_t, nil, nodelist);
      parent_t = parent[t]; parent[t] = new_cnode;
      if (label_b[t] > label_b[new_cnode]) label_b[new_cnode] = label_b[t];
     }
    else {
      parent[t] = new_cnode; update_label_b(t);
      if (label_b[t] > dfspos_num[w]) ptritem[t] = nodelist.append(t);
      if (label_b[t] > label_b[new_cnode]) label_b[new_cnode] = label_b[t];
     }
    if (¬is_cnode(t)) pred_t = t; // pred_t is never a c-node;
    t = parent_t;
  }
}
```

Figura 4.21: Código da função *calc_new_RBC_from_terminal_node*.

74b ⟨emenda um RBC antigo ao novo RBC 74b⟩≡ 75 ▹

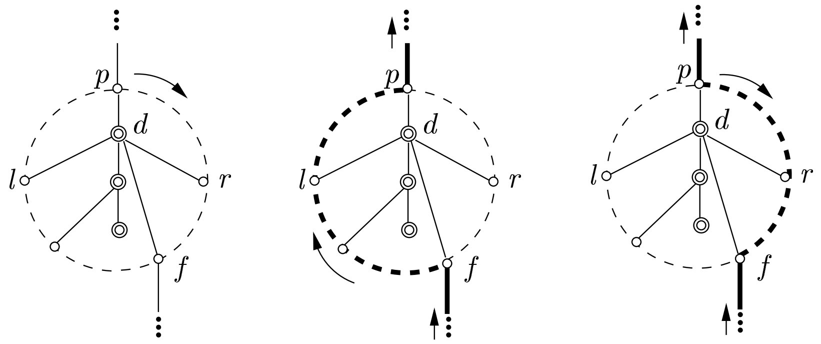
```
void sh_planar_impl::add_old_cnode_RBC_to_new_RBC(node oldcnode, node newcnode, node w,
                                                  node w1, node w2, bmlist<node>& nodelist)
{ // compress RBC[oldcnode];
  bmlist_item it1 = RBC[oldcnode].first_item(), it1l = RBC[oldcnode].cyclic_pred(it1, nil),
    it1r = RBC[oldcnode].cyclic_succ(it1, nil);
  node jl = RBC[oldcnode].contents(it1l), jr = RBC[oldcnode].contents(it1r),
    ul = nil, ur = nil;
  while (label_b[jl] ≡ dfspos_num[w]  // goes to the left;
       ∧ jl ≠ w1 ∧ jl ≠ w2)       (1)
    { if (ul ≡ nil) ul = jl;
      bmlist_item aux1 = it1l;
      it1l = RBC[oldcnode].cyclic_pred(it1l, it1);
      jl = RBC[oldcnode].contents(it1l);
      RBC[oldcnode].del(aux1);
    }
```

Figura 4.22: Código da função *add_old_cnode_RBC_to_new_RBC*.

75 ⟨emenda um RBC antigo ao novo RBC 74b⟩+≡ ◃74b

```
while (label_b[jr] ≡ dfspos_num[w] // goes to the right;
     ∧ jr ≠ w1 ∧ jr ≠ w2)     (2)
  { if (ur ≡ nil) ur = jr;
    bmlist_item aux1 = it1r;
    it1r = RBC[oldcnode].cyclic_succ(it1r, it1);
    jr = RBC[oldcnode].contents(it1r);
    RBC[oldcnode].del(aux1);
  }
RBC[oldcnode].del(RBC[oldcnode].first_item());
node first = w1;
if (w1 ≡ nil)
  if (ul ≠ nil) first = jl;
  else first = jr;
// checks if RBC[oldcnode] flipped;
if (RBC[oldcnode].contents(RBC[oldcnode].last_item()) ≡ first) RBC[oldcnode].reverse();
// removes w1 and w2 from RBC[oldcnode];
if (w1 ≠ nil) RBC[oldcnode].del(RBC[oldcnode].first_item());
if (w2 ≠ nil) RBC[oldcnode].del(RBC[oldcnode].last_item());
nodelist.conc(RBC[oldcnode]);
}
```

Figura 4.23: (Continuação) Código da função *add_old_cnode_RBC_to_new_RBC*.



(a) (b) (c)

Figura 4.24: Os vértices l e r em RBC(d) estão a esquerda e a direita de p, respectivamente, onde p é o pai de d em T. (a) Os elementos de RBC(d) estão no sentido horário. (b) O lado direito de RBC(d) é removido durante a compactação de RBC(d). (c) O lado esquerdo de RBC(d) é removido na compactação e é necessário inverter RBC(d).

o novo C-nó. Se m é um C-nó, então os valores de $b(t'_1)$ e $b(t'_2)$ são recalculados. Se m é um P-nó, então $b(m)$ é recalculado e *ptritem[m]* guarda a posição de m no novo RBC. Observe que um dos pedaços, devido a Q_1 ou a Q_2, do novo RBC deve ser invertido. (Na implementação aqui apresentada, t_1 é o terminal de menor bp-número e o pedaço do novo RBC devido a Q_1 foi invertido. Tudo isso foi necessário para manter a consistência com o que é feito no capítulo 5, para a construção da descrição combinatória plana.)

Uma simulação da PC-árvore alterada para o grafo da figura 4.1(a) é ilustrada nas figuras 4.25 e 4.26, onde, para cada C-nó ativo c, somente os filhos reais de c que estão no RBC(c) estão ilustrados. Os filhos reais de c que foram omitidos do RBC(c) também são omitidos nas figuras.

4.9 Correção

A correção baseia-se na validade dos seguintes invariantes envolvendo o grafo G, o conjunto L e a árvore T mantida pelo algoritmo:

(ish1) o subgrafo $G[L]$ é conexo;

(ish2) T é uma PC-árvore alterada associada a uma moldura de $G[V_G \setminus L]$ em G.

O primeiro invariante decorre facilmente do fato de que os vértices são processados de acordo com uma bp-ordenação.

Para mostrar a validade do segundo invariante, deve-se verificar que, ao final de uma iteração do algoritmo de SH, ou uma XY-obstrução é detectada ou T é uma PC-árvore alterada associada a uma moldura de $G[V_G \setminus L']$ em G.

O algoritmo modifica a árvore T, por meio da função *find_active_cnode*, enquanto busca um XY-caminho ou uma XY-obstrução. Nessa função, são feitas apenas modificações no valor de *parent[u]*, que passa a apontar para o pai real de um P-nó u. Claramente tais modificações não afetam a validade do invariante (ish2).

A função que verifica a existência de uma XY-obstrução é a função *find_obstruction*. Os lemas a seguir mostram que essa função devolve **true** se e somente se existe uma XY-obstrução no componente em questão da moldura M.

Lema 4.2: *Se o algoritmo atinge um dos pontos marcados com (1) nas funções find_obstruction e test_obstruction_from_terminal (figuras 4.11 e 4.13, nas páginas 63 e 65, respectivamente), há uma obstrução do tipo (o1) em M.*

Demonstração: Considere o caso da função *test_obstruction_from_terminal*, onde uma obs-

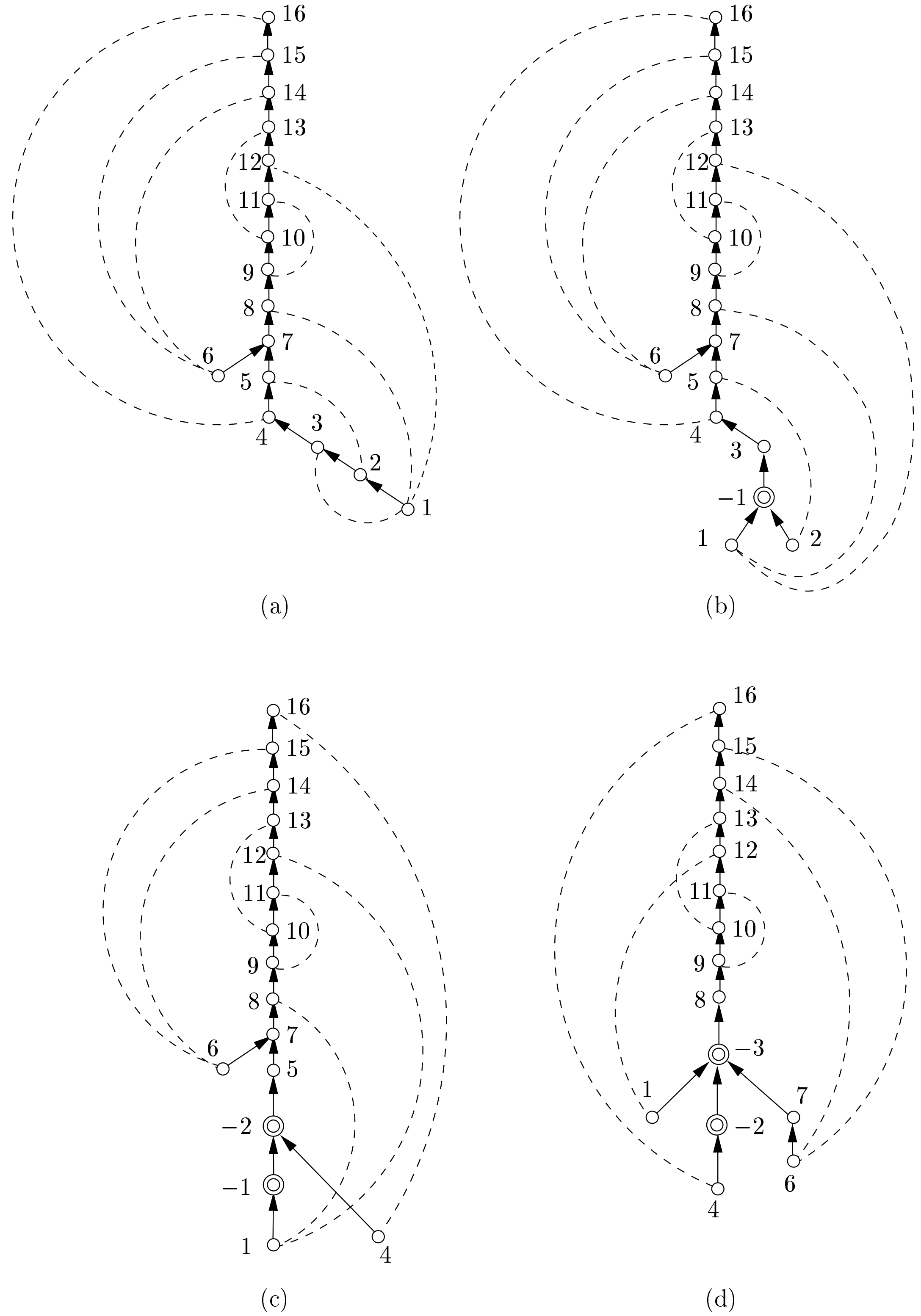


Figura 4.25: Execução do algoritmo sobre o grafo da figura 4.1(a). (a) Configuração inicial da PC-árvore alterada T. (b) T após o algoritmo examinar o vértice 3. (c) T após examinar 5. (d) T após examinar 8.

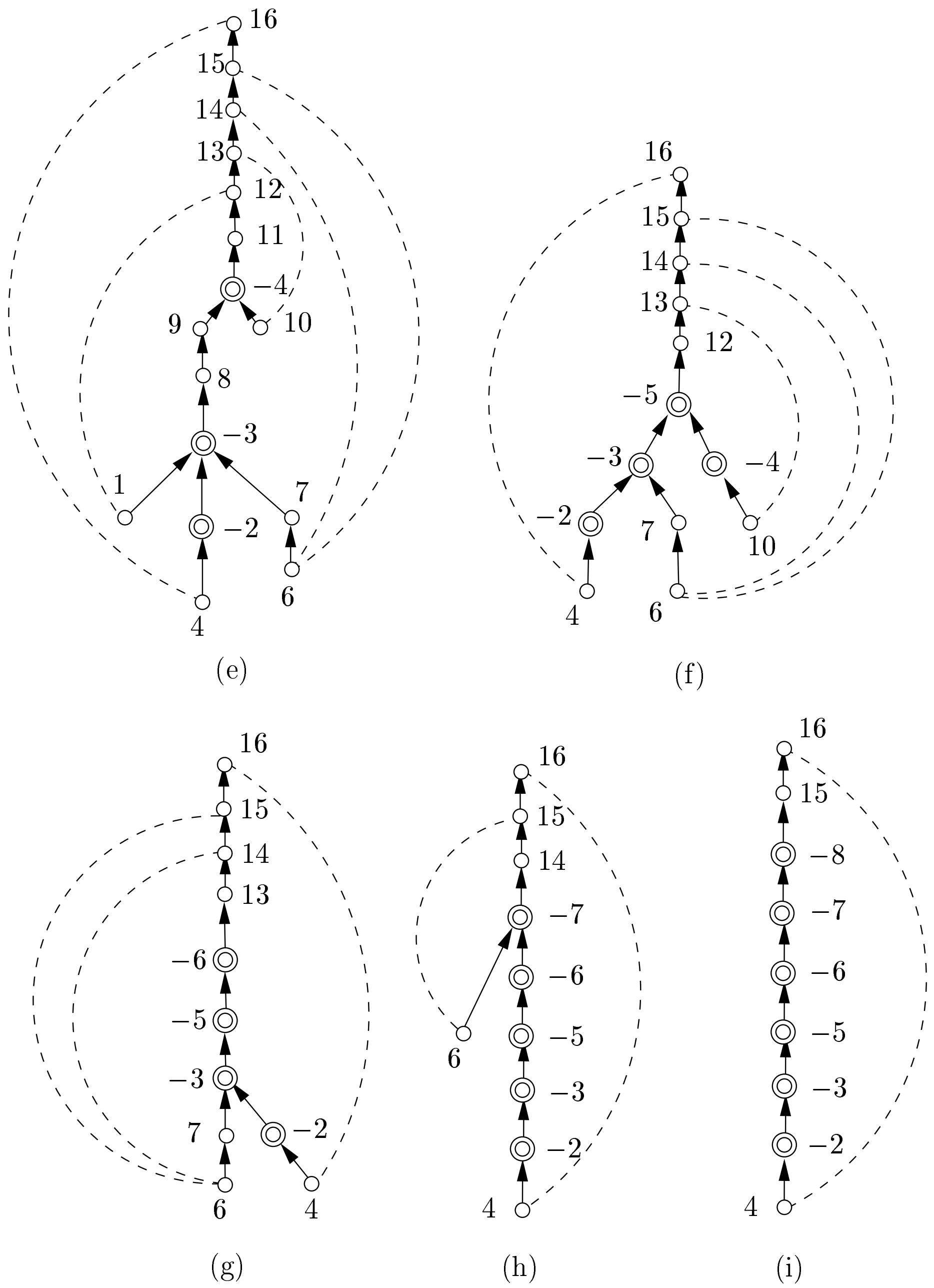


Figura 4.26: (Continuação) Execução do algoritmo sobre o grafo da figura 4.1(a). (e) T após examinar 11. (f) T após examinar 12. (g) T após examinar 13. (h) T após examinar 14. (i) T após examinar 15.

trução é detectada quando, na função *test_cnode_counter* (figura 4.16), a condição (1) é verdadeira. Seja c um C-nó ativo no caminho Q. Lembre-se que a variável *counter[c]* contém o número de filhos reais de c que enxergam X através de $(V_G \setminus L) \setminus V_{B_c}$ e note que, claramente, *count* $\leq$ *counter[c]*, onde *count* é o contador da função *test_cnode_counter*. Caso estes dois contadores não coincidam, há uma obstrução do tipo (o1), conforme mostrado a seguir. Seja p o pai de c em T. Neste caso, existe um filho real f de c que não foi contado em *count* e f está entre dois vértices no $RBC(c)$, digamos l e r, que são distintos entre si e distintos de p, e que enxergam Y através de $(V_G \setminus L) \setminus V_{B_c}$. Além disso, f está no lado oposto a p no RBC(c) e a XY-obstrução encontrada é (C, p, l, f, r), onde C é o bloco B_c (B_c é o bloco de M correspondente a c, que é o ciclo facial externo de c, definido no próximo capítulo) (figura 4.14, página 66). ∎

O próximo lema diz respeito aos casos marcados com (2) na função *find_obstruction*, que é parcialmente detectado pela função *test_cnode_counter*, que modifica o conteúdo da variável global *cnode_of_possible_k33_obstruction*.

Lema 4.3: *Se o algoritmo atinge um dos pontos marcados com (2) na função find_obstruction (figuras 4.11 e 4.12, páginas 63 e 64), então há uma obstrução do tipo (o1) em M.*

Demonstração: Para ocorrer este caso, a variável *cnode_of_possible_k33_obstruction* foi modificada pela função *test_cnode_counter* e aponta para um C-nó ativo d como ilustrado na figura 4.14(b). Seja p o pai de d em T. Neste caso, d tem três filhos reais l, r e f, distintos entre si, tais que f enxerga Y através de $(V_G \setminus L) \setminus V_{B_d}$ e l e r enxergam X através de $(V_G \setminus L) \setminus V_{B_d}$.

Claramente, tanto para o caso de um terminal como para o caso de dois terminais, as marcas (2) são atingidas somente se p enxerga Y através de $(V_G \setminus L) \setminus V_{B_d}$. Logo, a XY-obstrução encontrada neste caso é (C, p, l, f, r), onde C é o bloco B_d. ∎

A situação em que uma obstrução é detectada no ponto marcado com (3) na função *find_obstruction* é analisada no lema abaixo.

Lema 4.4: *Se o algoritmo atinge o ponto marcado com (3) na função find_obstruction (figura 4.12, página 64), então há uma obstrução do tipo (o2) ou do tipo (o3) em M.*

Demonstração: Este caso se manifesta num componente de $T - L$, correspondente a um componente K em M, com dois terminais, digamos t_1 e t_2. Seja m o ancestral comum entre t_1 e t_2 em T com o menor bp-número, seja Q o caminho em T de um dos terminais a w e seja Q' o subcaminho de Q, do pai de m ao filho de w.

Para um nó v em Q', seja $T_v^* := T_v - T_{v'}$, onde v' é o filho de v em Q e seja V^* o conjunto dos P-nós em T_v^* a menos de v. Se o algoritmo atinge o ponto marcado com (3), então existe v tal que v é ou um P-nó em Q' ou um filho real de um C-nó em Q' tal que v enxerga L através de V^*.

Se m é um C-nó (ativo) então, para $i = 1, 2$, seja t'_i o filho real de m que é ancestral de t_i. Sejam p o pai de m e $C = B_m$. A obstrução (C, t'_1, t'_2, p) é do tipo (o2).

Se m é um P-nó então sejam K_1, K_2 e K_3 os componentes resultantes de $K - m$ que contém, respectivamente, t_1, t_2 e v. Neste caso, (m, K_1, K_2, K_3) é uma obstrução do tipo (o3). ∎

Finalmente, a última situação em que uma obstrução é detectada pelo algoritmo é descrita no lema abaixo.

Lema 4.5: *Se o algoritmo atinge o ponto marcado com (4) na função find_ obstruction (figura 4.12, página 64), então há uma obstrução do tipo (o2) ou do tipo (o3) em M.*

Demonstração: Este caso ocorre quando existem três ou mais terminais num componente de $T - L$. Sejam t_1, t_2 e t_3 três terminais distintos nesse componente e seja m o ancestral comum de t_1, t_2 e t_3 com o menor bp-número. Se m é um C-nó (ativo) então sejam t'_1, t'_2 e t'_3 os filhos reais de m e ancestrais de t_1, t_2 e t_3, respectivamente. Neste caso existe uma obstrução (C, t'_1, t'_2, t'_3) do tipo (o2), onde C é o bloco B_m.

Se m é um P-nó, então seja v o ancestral comum em T de pelo menos dois entre t_1, t_2 e t_3 com o menor bp-número. Se v é um C-nó então, analogamente ao que foi descrito acima, uma obstrução do tipo (o2) é encontrada. Caso contrário, sejam K_1, K_2 e K_3 os componentes resultantes de $M - v$ que contém, respectivamente, t_1, t_2 e t_3. Assim, (v, K_1, K_2, K_3) é uma obstrução do tipo (o3). ∎

Se nenhum dos casos tratados pelos lemas 4.2, 4.3, 4.4 e 4.5 ocorre, existe um XY-caminho, conforme atesta o lema abaixo.

Lema 4.6: *Se a função find_ obstruction devolve* **false**, *então existe um XY-caminho no componente em questão.*

Demonstração: No caso de um terminal, seja Q o caminho em T do terminal a w. O XY-caminho consiste numa extensão do caminho P' em M, do terminal ao filho de w em Q, obtido da seguinte maneira. Cada P-nó em Q está em P' e, para cada C-nó (ativo) c em Q, seja f definido a seguir. Se c é um terminal, então seja f um filho real de c tal que $b(f) = \mathrm{bp}(w)$. Caso contrário, seja f o filho real de c que está em Q. O lado de B_c, determinado por f e pelo pai de c, associado ao lado do RBC(c) que foi removido durante a compactação de RBC(c) pela função *add_ old_ cnode_ RBC_ to_ new_ RBC* (figuras 4.22 e 4.23) está em P'.

O caso de dois terminais é análogo. ∎

Os cinco lemas acima mostram que a função *find_obstruction* funciona corretamente. Com isso, se o algoritmo atinge a linha marcada com (1) na função *is_planar* (figura 4.4, página 55), existe um XY-caminho P no componente em questão. A função *set_info_for_new_cnode* (seção 4.8) cria um novo C-nó c para representar o componente biconexo correspondente e modifica a árvore T da seguinte maneira.

Seja $\overline{P}$ o caminho complementar de P. Durante a construção do RBC(c), faz-se w ser o pai de c em T e, para cada vértice inserido em RBC(c), seu pai real passa a ser c. Os vértices inseridos em RBC(c) são vértices em $\overline{P}$ cujo novo valor de b, após a alteração de seu pai, é superior ao bp(w). Ao término da construção do RBC(c), cada vértice de $\overline{P}$ que enxerga L' através de $V_G \setminus V_{B_c}$ tem como pai real c e a propriedade (T3') é válida no final de uma iteração do algoritmo. Claramente a propriedade (T4') é satisfeira.

∎

4.10 Eficiência

A análise do teste de planaridade consiste na análise do tempo consumido pela função *is_planar* (página 55). Para analisar essa função, primeiramente analisa-se as rotinas *pre_processing*, *find_terminals*, *find_obstruction* e *set_info_for_new_cnode*.

Função *pre_processing*

Esta claramente consome tempo linear, já que a bp-árvore e os valores de h e b podem ser calculados em tempo $O(n+m)$, por meio de uma busca em profundidade no grafo. É nessa função também que é inicializado o vetor *state[ ]* e algumas outras informações associadas a cada vértice, o que consome tempo $O(n)$.

Função *find_terminals*

A menos dos **while**'s (1) e (2) e do **forall** na última linha (figura 4.10, página 62), o tempo gasto pela função é $O(d_w)$ devido ao **forall_out_edges**, onde d_w é o grau do vértice w em G.

O tempo gasto pelo **while** (2) é certamente limitado pelo tempo consumido pelo **while** (1), já que S é declarada imediatamente antes do **while** (1) e nós são empilhados em S apenas no **while** (1), um por iteração. Para determinar o tempo consumido pelo **forall**, note que a variável *traversed_nodes*, declarada no início da função *find_terminals*, contém, imediatamente antes do

forall, todos os nós percorridos no **while** (1). Ou seja, o tempo do **forall** também é dominado pelo tempo do **while** (1).

Denote por k_w o número de nós armazenados na variável *traversed_nodes* depois do **forall_out_edges**. O tempo total consumido pelo **while** (1), considerando todas as iterações do **forall**, é $\theta(k_w + d_w)$ pois, em cada iteração do **forall_out_edges**, o **while** (1) visita no máximo dois vértices já percorridos — um antes de sair do **while** (1) e outro possivelmente dentro da função *find_active_cnode*.

Função *find_obstruction*

A função *find_obstruction* (figuras 4.11 e 4.12, páginas 63 e 64) basicamente percorre um subconjunto dos nós percorridos pela função *find_terminals*. Os únicos nós percorridos por ela que não foram percorridos por *find_terminals* são alguns dos nós visitados pela função *test_cnode_counter* (figura 4.16, página 68). Tais nós podem ser incluídos no contador k_w que é analisado globalmente. Resumindo, o tempo consumido pela função *find_obstruction* é $O(k_w)$, onde k_w é o número de nós percorridos ou pela função *find_terminals* ou pela função *find_obstruction* durante a iteração em que w é processado.

Função *set_info_for_new_cnode*

O tempo consumido por essa rotina é $O(k_w)$ pois ela consome tempo linear no número de nós que foram visitados anteriormente por *find_terminals* ou por *find_obstruction*.

Assim, para concluir a análise da função *is_planar*, resta totalizar o tempo consumido nas iterações do **for** da função *is_planar*. Isto é, resta calcular uma delimitação superior para

$$\sum_{w=1}^{n} (k_w + d_w) = \sum_{w=1}^{n} k_w + 2m.$$

Isso corresponde a dar uma delimitação superior para $\sum_{w=1}^{n} k_w$, ou seja, para o número total de nós percorridos pelo algoritmo.

Número de nós percorridos

Observe que apenas C-nós ativos e P-nós são percorridos na PC-árvore alterada, conforme descrito na seção 4.5. Um C-nó ativo é percorrido apenas em uma iteração: ao ser percorrido, ou encontra-se uma obstrução ou ele se torna inativo.

Similarmente um P-nó de bp-número menor que o de w e que é filho de um outro P-nó em T é percorrido em apenas uma iteração nessas condições: depois de ser percorrido, se nenhuma obstrução foi detectada, então ele passa a ser filho de um C-nó até o algoritmo terminar de executar.

Resta calcular quantas vezes um P-nó que é filho de um C-nó pode ser visitado durante o algoritmo. Observe que, para um C-nó ativo c da PC-árvore alterada T, no início de cada iteração cada vértice do RBC(c) tem o valor de b maior ou igual a bp(w).

No percurso na PC-árvore alterada T, para cada C-nó ativo c que se torna inativo devido a contrução de um novo C-nó, os filhos reais de c, cujo valor de b é igual ao bp(w), são visitados um número constante de vezes e depois são omitidos do RBC do novo C-nó e no máximo dois filhos reais de c cujo valor de b é superior ao bp(w) são visitados (e são visitados um número constante de vezes).

Assim, totalizando o número de nós visitados pelo algoritmo durante todo o processamento é $O(n + n_c)$, onde n_c é o número de C-nós criados. Como $n_c = O(n)$, pois no máximo $3n - 6$ C-nós são criados, então $\sum_{w=1}^{n} k_w = O(n)$. Juntando isso aos cálculos anteriores, conclui-se que o tempo total gasto pela função *is_planar* é $O(n)$.

∎

Capítulo 5

Construção de um desenho plano

Neste capítulo, são descritas as adaptações o teste de planaridade descrito no capítulo 4, necessárias para a obtenção de uma descrição combinatória plana do grafo, caso este seja planar e biconexo.

Como observado na seção 2.4, uma descrição combinatória plana é formada por listas de adjacências e pode ser representada por uma lista única de arcos. Esta representação é utilizada nesta implementação.

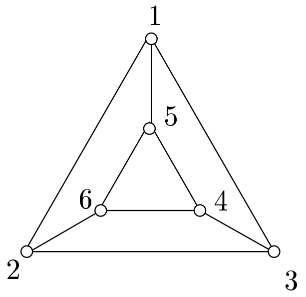
5.1 Alterações na PC-árvore e no corpo do algoritmo

Seja H um grafo planar e v um vértice de H. Uma descrição combinatória plana D de H é **alinhada por** v se um de seus ciclos faciais que contém v, digamos $C := \langle u_0, u_1, \ldots, u_{k-1}, u_0 \rangle$, é tal que, para cada i, o arco $u_i u_{i+1}$ é o primeiro arco da lista $D(u_i)$, considerando-se os subíndices módulo k. Neste caso, C é o ciclo facial **determinado por** D **e** v (figura 5.1).

Para obter uma descrição combinatória plana do grafo dado, o algoritmo de SH mantém, para cada C-nó ativo c da PC-árvore, uma descrição combinatória plana particular do bloco em $H + w$ associado ao bloco B_c da moldura, denotada por DCP(c). A DCP(c) é alinhada pelo pai de c na PC-árvore T. Além disso, o ciclo facial determinado pela DCP(c) e pelo pai de c em T, chamado de **ciclo facial externo de** c, contém todos os vértices do RBC(c). (Na verdade, o ciclo facial externo de um C-nó ativo c corresponde ao bloco B_c da moldura corrente.) DCP(c)

Com essa informação extra, é possível obter a moldura em cada iteração do algoritmo: basta juntar as arestas de $T - L$ cujas pontas são vértices de G e os ciclos faciais externos de cada C-nó ativo de T.

Na implementação, para cada C-nó ativo c de T, a DCP(c) é armazenada na variável global *embed_list[c]*, que é uma lista duplamente ligada do tipo **bmlist** descrito na declaração de uma



Descrição combinatória plana D alinhada por 1 e por 4:

$D = \langle(6,4),(1,3),(1,5),(2,1),(2,6),(2,3),(3,2),(3,4),(3,5),(4,5),$
$(4,3),(4,6),(5,6),(5,1),(5,4),(6,2),(6,5),(1,2)\rangle$

$D(1) = \langle(1,3),(1,5),(1,2)\rangle$
$D(2) = \langle(2,1),(2,6),(2,3)\rangle$
$D(3) = \langle(3,2),(3,4),(3,5)\rangle$
$D(4) = \langle(4,5),(4,3),(4,6)\rangle$
$D(5) = \langle(5,6),(5,1),(5,4)\rangle$
$D(6) = \langle(6,4),(6,2),(6,5)\rangle$

Ciclo facial determinado por D e 1: $\langle 1,3,2,1\rangle$
Ciclo facial determinado por D e 4: $\langle 4,5,6,4\rangle$

Figura 5.1: Uma descrição combinatória plana alinhada por 1 e por 4.

PC-árvore alterada na página 53. Na última iteração, o algoritmo constrói uma lista (chamada de *final_emb_list* na implementação) que representa a descrição combinatória construída para G, caso G seja planar. Finalmente, através da função *sort_edges* do LEDA, as listas de adjacência de G são reordenadas de acordo com a ordem em *final_emb_list*.

Em cada iteração do algoritmo, durante a construção de um novo C-nó c, a função *calculate_partial_embedding* (figuras 5.2 e 5.3) é acionada pela função *is_planar*, no ponto marcado com (1) na figura 4.4 (página 55) para construir a DCP(c).

86 ⟨cálculo da DCP para o novo c-nó 86⟩≡ 87 ▷

```
void sh_planar_impl::calculate_partial_embedding(graph& G, node w, node new_cnode,
                                        list<edge>& list_back_edges, list<node>& terminal_nodes)
{ list<node> traversed_nodes, list_representants;
  switch (terminal_nodes.size()) {
    case 1: {
        map<node, node> backedge_representant;
        node term = terminal_nodes.contents(terminal_nodes.first());
        mark_path_in_T(term, w, backedge_representant, traversed_nodes);
        map<node, list<edge> > b_edges_repres;
        b_edges_repres = group_back_edges_by_repr(G, list_back_edges,
                                        backedge_representant, traversed_nodes, list_representants);
        list<node> L_l;
        L_l = embed_upward_T(term, w, G, w, b_edges_repres, traversed_nodes, embed_list[new_cnode]);
```

Figura 5.2: Código da função *calculate_partial_embedding*.

87 ⟨cálculo da DCP para o novo c-nó 86⟩+≡ ◁86

```
        // embeds all nodes in L_l;
        node t;
        forall(t, L_l) {
          bmlist<edge> el;
          embed_back_edges(false, G, t, traversed_nodes, b_edges_repres[t], el);
          el.reverse();
          el.conc(embed_list[new_cnode]);
          el.swap(embed_list[new_cnode]);
          embed_list[new_cnode].conc(list_back_edges_out_w);
        }
      } // case 1;
    break;
    case 2:
      . . .
  }
```

Figura 5.3: (Continuação) Código da função *calculate_partial_embedding*.

5.2 Construção da DCP de um novo C-nó

Considere uma iteração do algoritmo. Lembre-se de que w denota o vértice examinado nesta iteração e K é o componente em $T - L$ que está sendo processado neste instante.

A construção da DCP consiste em decidir a ordem cíclica em torno de w das arestas de retorno com uma ponta em w e outra em K. (Esta ordem induz a ordem das demais arestas em K.) A construção da DCP(c) para o caso em que há apenas um terminal é descrita a seguir. O caso de dois terminais é similar, mas exige mais detalhes técnicos, por isso será omitido.

Grosseiramente, a construção da DCP(c) pode ser descrita da seguinte maneira. Seja P' o subcaminho do XY-caminho P do terminal t ao filho de w (em T) em questão. Cada aresta de retorno em questão tem uma ponta em alguma subárvore enraizada num nó de P', conforme a figura 5.4(a). O primeiro passo da construção da DCP(c) consiste na separação dessas arestas de retorno em grupos dados pela raiz em P' da subárvore em questão. O segundo passo é a construção da DCP(c) para cada um desses grupos.

Na verdade, devido à assimetria introduzida pela orientação da PC-árvore T, algumas arestas de retorno são associadas a vértices que não estão em P'. Mais precisamente, algumas arestas são associadas a P-nós (ou a filhos reais dos C-nós) que estão no caminho de p a w em T da figura 5.4(b). Tal situação ocorre quando o XY-caminho P não termina por uma aresta que está em T. As arestas de retorno associadas a vértices fora de P' devem aparecer antes das

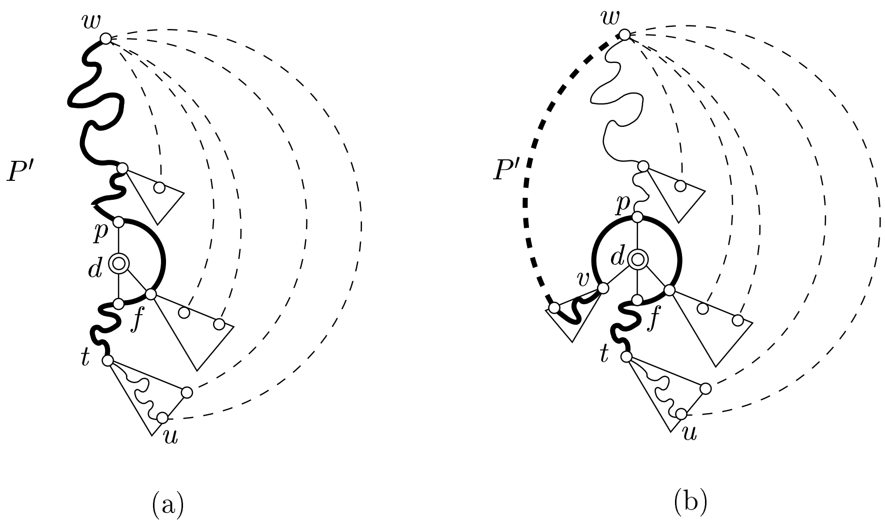


Figura 5.4: Em (a) e (b), o subcaminho P' de um XY-caminho com um único terminal t é indicado por linhas grossas e as arestas de retorno por linhas tracejadas.

arestas de retorno associadas a vértices que estão na parte final de P', após o pai do último C-nó atravessado por P' — o vértice p da figura 5.4(b). Neste caso, a função *embed_ upward_ T* (descrita mais adiante) devolve uma lista de vértices correspondente a parte final de P' (e os vértices à "esquerda" no RBC dos C-nós ativos no caminho de p a w em T da figura 5.4(b)) e tais vértices são processados após a chamada da função. (Esse processamento é feito fora da função *embed_ upward_ T* para que esta possa ser usada também no caso de dois terminais.)

Abaixo encontra-se uma descrição da função *calculate_ partial_ embedding* (figuras 5.2 e 5.3), bem como das funções invocadas por ela.

O primeiro passo da construção da DCP(c) consiste na separação das arestas de retorno em questão em grupos. A idéia é que a ordem relativa entre arestas em grupos distintos é determinada pela ordem em que as raízes das subárvores correspondentes aparecem. Esta ordenação, juntamente com a ordenação das arestas num mesmo grupo, induz uma ordem cíclica das arestas de retorno em torno de w e das demais arestas em K.

Seja Q o caminho de t a w em T. Uma aresta de retorno $f = wz$ é representada pelo vértice z. Assim, na verdade, associa-se vértices a vértices. Inicialmente o vértice z, representando f, é associado ao nó v de Q se existe caminho internamente disjunto de Q de z a v. Se v é um C-nó, então z é associado ao filho real de v que está no caminho de z a v em T. Isso é feito da seguinte maneira. Primeiramente marca-se apenas os nós do caminho Q como visitados e cada um desses nós é associado a si próprio, através da variável *backedge_ representant*. Isso é feito pela rotina *mark_ path_ in_ T* (figura 5.5).

89 ⟨marca caminho em T como VISITADO 89⟩≡

```
void sh_planar_impl::mark_path_in_T(node t, node w,
                  map<node, node>& backedge_representant, list<node>& traversed_nodes)
{ state[w] = VISITED; backedge_representant[w] = w; traversed_nodes.append(w);
  node u = t;
  while (state[u] ≡ NOT_VISITED)
    { state[u] = VISITED; traversed_nodes.append(u); backedge_representant[u] = u; u = parent[u]; }
}
```

Figura 5.5: Código da função *mark_path_in_T*.

Para cada aresta $f = wz$, tal que $\mathrm{bp}(z) < \mathrm{bp}(w)$, sobe-se em T a partir de z até chegar-se a um nó visitado v. Se v está associado a um C-nó (neste caso, note que v está associado a si próprio), então z é associado ao filho real de v que está no caminho de z a v em T. Caso contrário, seja u o vértice ao qual v está associado. Neste caso, z é associado a u. Durante este processo, marca-se todos os nós de z até v como visitados, associando-os ao mesmo vértice ao qual z foi associado. Terminado este processo, cada aresta de retorno em questão (na verdade, a sua ponta mais baixa em T) está associada a um P-nó de Q ou a um filho real de um C-nó de Q. O código correspondente a este processo é dado pela função *group_back_edges_by_repr* (figuras 5.6 e 5.7).

O segundo passo é a construção da $\mathrm{DCP}(c)$ para cada um dos grupos criados no passo anterior. Este passo é efetuado pela função *embed_upward_T* (figura 5.8), que é descrita a seguir.

Inicialmente, a lista $\mathrm{DCP}(c)$ é vazia. Para cada nó v de Q, subindo em T a partir de t, se v é um C-nó ativo, então a função *add_old_cnode_to_embedding* (figura 5.13) é chamada para incluir a $\mathrm{DCP}(v)$, de maneira conveniente, no início da $\mathrm{DCP}(c)$. Se v é um P-nó e tem um filho em Q, então seja v' esse filho. Se v' é um P-nó, os pares vv' e $v'v$ são adicionados no início da $\mathrm{DCP}(c)$. Em seguida, a função *embed_back_edges* (figuras 5.9 e 5.10) é chamada para incluir à $\mathrm{DCP}(c)$ os pares referentes à subárvore de T enraizada em v, denotada por T_v^*, definida a seguir. A árvore T_v^* coincide com T_v se $v = t$ e, caso contrário, $T_v^* := T_v - T_{v'}$, onde v' é o filho de v em Q.

Para cada C-nó ativo c em Q, os vértices à "esquerda" no $\mathrm{RBC}(c)$ são inseridos na lista *to_embed_later*, pela função *add_old_cnode_to_embedding*, explicada mais adiante. Essa lista contém tais vértices na ordem em que as arestas de retorno associadas aparecem em torno de w, e é devolvida ao final da função *embed_back_edges*.

90a ⟨group back-edges 90a⟩≡ 90b ▷

```
map<node, list<edge> > sh_planar_impl::group_back_edges_by_repr(graph& G,
                                    list<edge>& list_back_edges, map<node,node>& backedge_representant,
                                    list<node>& traversed_nodes, list<node>& list_representants)
{ list<node> nl; edge e;
  forall(e, list_back_edges) {
    list<node> S;
    node t = source(e), pnode = t; // pnode is never a c-node;
    while (state[t] ≡ NOT_VISITED) {
      if (¬is_cnode(t)) pnode = t;
      if (state[t] ≡ NOT_VISITED) { state[t] = VISITED; nl.append(t); S.push(t); }
      t = parent[t];
     }
    node repr;
    if (¬is_cnode(backedge_representant[t])) repr = backedge_representant[t];
    else { repr = pnode; traversed_nodes.append(repr); }
    forall(t, S) backedge_representant[t] = repr;
    if (¬has_back_edge[repr]) { has_back_edge[repr] = true; list_representants.append(repr); }
  } // end forall(e, list_back_edges);
```

Figura 5.6: Código da função *group_back_edges_by_repr*.

90b ⟨group back-edges 90a⟩+≡ ◁ 90a

```
  map<node, list<edge> > list_edges;
  forall(e, list_back_edges) { node u = source(e); list_edges[backedge_representant[u]].append(e); }
  node t;
  forall(t, nl) state[t] = NOT_VISITED;
  forall(t, list_representants) state[t] = VISITED;
  return list_edges;
}
```

Figura 5.7: (Continuação) Código da função *group_back_edges_by_repr*.

91 ⟨cálculo da DCP subindo em T 91⟩≡

```
list<node> sh_planar_impl::embed_upward_T(node t, node u, graph& G, node w,
                                          map<node, list<edge> >& b_edges_repres,
                                          list<node>& traversed_nodes, bmlist<edge>& emb_list)
{ list<node> to_embed_later;
  node v = t, pred_v = nil;
  while (pred_v ≠ u) {
     if (is_cnode(v)) {
        node f = pred_v, oldcnode = v;
        add_old_cnode_to_embedding(G, w, oldcnode, f, b_edges_repres, traversed_nodes,
                                   to_embed_later, emb_list);
        v = parent[oldcnode];
        if (v ≡ u) return to_embed_later;
      }
     else if (pred_v ≠ nil) {
            emb_list.push(G.reverse(T0_edge_in[pred_v]));
            if (v ≠ w) emb_list.push(T0_edge_in[pred_v]);
            else emb_list.append(T0_edge_in[pred_v]);
          }
     if (has_back_edge[v] ∧ v ≠ u)
       embed_back_edges(G, v, traversed_nodes, b_edges_repres[v], emb_list);
     pred_v = v; v = parent[v];
   } // while;
  return to_embed_later;
}
```

Figura 5.8: Código da função *embed_upward_T*.

função *embed_ back_ edges*

A função *embed_ back_ edges* (figuras 5.9 e 5.10) constrói, para um dado vértice v, duas listas de pares de vértices, $L_T(v)$ e $L_w(v)$, que são concatenadas no início e no fim da DCP(c) corrente, respectivamente.

92 ⟨desenha arestas de retorno 92⟩≡ 93 ▷

```
void sh_planar_impl::embed_back_edges(graph& G, node v, list<node>& traversed_nodes,
                                      list<edge>& list_back_edges, bmlist<edge>& emb_list1)
{ if (list_back_edges.empty()) return;
  edge e;
  bmlist<edge> L_T, L_w;
  node w = target(list_back_edges.contents(list_back_edges.first()));
  array<edge> back_edge;
  int n = sort_back_edges_by_dfs(w, v, list_back_edges, back_edge);
  for(int i = 1; i ≤ n; i++) {
    e = back_edge[i];
    if (e ≠ nil) L_w.append(G.reverse(e));
  }
```

Figura 5.9: Código da função *embed_ back_ edges*.

Inicialmente, as arestas de retorno associadas a v são ordenadas por uma bp-ordenação de T_v^*, respeitando a ordem fixada para cada C-nó ativo em T_v^*. Esse processo é realizado pela função *sort_ back_ edges_ by_ dfs* (figura 5.11). Essa função começa calculando uma subárvore J de T_v^* onde, para cada aresta de retorno wz associada a v, J é a união dos caminhos de z a v em T.

A lista de adjacências de cada C-nó ativo d da árvore J é reorganizada de maneira a preservar a ordem dos filhos reais de d em T (lembre-se que a atualização de T ainda não foi executada nesta iteração). A função termina fazendo uma busca em profundidade na árvore J, obtendo uma lista R_v das arestas de retorno wz associadas a v, ordenada pelo bp-número de z. (Essa lista é representada pela variável *back_ edge[ ]* nas figuras 5.9 e 5.10.)

Após a ordenação das arestas associadas a um vértice v, prossegue-se à construção da DCP(c) pela função *embed_ back_ edges*. Nessa função, marca-se todos os vértices de J como não-visitados, exceto v que é marcado como visitado. As listas $L_T(v)$ e $L_w(v)$ começam vazias. Para cada aresta wz na lista R_v, inclui-se no fim de $L_w(v)$ o arco wz.

Para cada aresta wz na lista R_v, em ordem reversa, inicia-se uma lista auxiliar *el* com o arco zw e percorre-se a árvore J, a partir de z, subindo até o primeiro nó visitado. Cada aresta xy percorrida nesse percurso acarreta a inclusão, no início da lista *el*, dos arcos xy e yx. Ao final deste percurso, a lista *el* é concatenada ao fim da lista $L_T(v)$ corrente e passa-se à próxima aresta

```
93 ⟨desenha arestas de retorno 92⟩+≡                                  ◁92
    for(int i = n; i ≥ 1; i--) {
      e = back_edge[i];
      if (e ≠ nil) {
        bmlist<edge> el; el.push(e);
        node pred_t = source(e), t = parent[pred_t];
        while (state[pred_t] ≡ NOT_VISITED) {
          state[pred_t] = VISITED; traversed_nodes.append(pred_t);
          if (is_c_node(t)) { embed_list[t].conc(el); embed_list[t].swap(el); t = parent[t]; }
          else { el.push( T0_edge_in[pred_t]); el.push(G.reverse( T0_edge_in[pred_t])); }
          pred_t = t; t = parent[t];
        } // end while
        L_T.conc(el);
      } // end if (e != nil)
    } // end for
    L_T.conc(emb_list1); L_T.swap(emb_list1); emb_list1.conc(L_w);
  }
```

Figura 5.10: (Continuação) Código da função *embed_back_edges*.

de retorno de R_v. Isso completa a descrição da função *embed_back_edges*. Uma ilustração do que é feito por esta função se encontra na figura 5.12.

função *add_old_cnode_to_embedding*

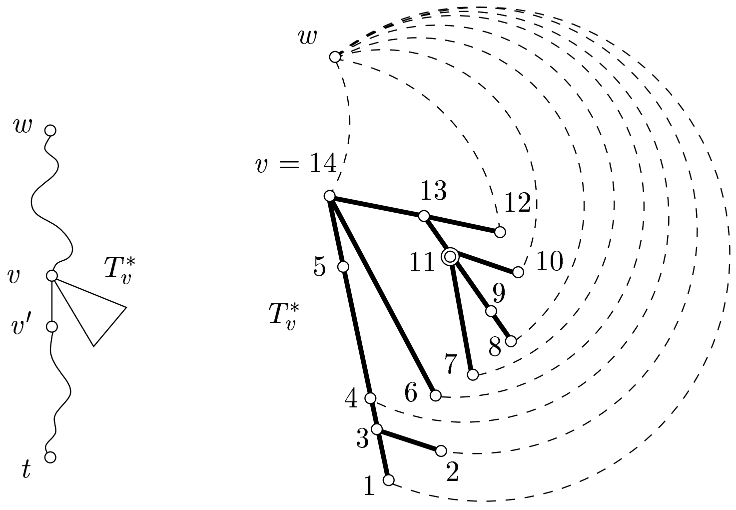
Seja v um C-nó ativo e sejam f o filho real de v que está no XY-caminho P e p o pai de v. Seja L_l o conjunto dos vértices em RBC(v) entre f e p, à esquerda (*cyclic_pred*) de p, e seja L_r o conjunto dos vértices em RBC(v) entre f e p, à direita (*cyclic_succ*) de p. Se algum vértice de L_r enxerga Y através de $(V_G \setminus L) \setminus V_{B_v}$, então a DCP($v$) é invertida e as listas L_l e L_r são trocadas entre si (deste modo, nenhum vértice em L_r enxerga Y através de $(V_G \setminus L) \setminus V_{B_v}$). Para cada vértice u em L_r que enxerga X através de $(V_G \setminus L) \setminus V_{B_v}$, na ordem dada pelo RBC(v), a função *embed_back_edges* é chamada para incluir os arcos devido a subárvore T_u à DCP(v). Em seguida, concatena-se DCP(v) no início da lista DCP(c) e concatena-se L_l no início da lista *emblater*, que contém os vértices a serem incluídos à DCP(c) no final da execução da função *calculate_partial_embedding* (figura 5.3).

Para encerrar a construção de uma descrição combinatória plana de um grafo planar, resta apenas inserir os arcos à descrição que têm uma ponta na raiz r de T. Isto é feito pela função *embed_root*, chamada no ponto marcado com (2) na função *is_planar* (figura 4.4, página 55). Neste caso, cada aresta de retorno rz é associada a r. Seja J a união dos caminhos de z a r

94 ⟨ordena arestas de retorno 94⟩≡

```
int sh_planar_impl::sort_back_edges_by_dfs(node w, node repr, list<edge>& list_back_edges,
                                             array<edge>& back_edge)
{ graph D;
  list<node> list_nodes, list_cnodes;
  map<node, node> node_in_D, node_in_G;
  edge e;
  node_in_D[repr] = D.new_node();
  node_in_G[node_in_D[repr]] = repr;
  forall(e, list_back_edges) {
    node t = source(e), pred_t = nil;
    while(state[t] ≡ NOT_VISITED) {
      if (is_cnode(t)) list_cnodes.append(t);
      state[t] = VISITED; list_nodes.append(t);
      node_in_D[t] = D.new_node();
      node_in_G[node_in_D[t]] = t;
      if (pred_t ≠ nil) D.new_edge(node_in_D[t], node_in_D[pred_t]);
      pred_t = t; t = parent[t];
     } // while
    if (pred_t ≠ nil) D.new_edge(node_in_D[t], node_in_D[pred_t]);
   } // forall
  node u;
  forall(u, list_nodes) state[u] = NOT_VISITED;
  map<node, bool> is_in_D;
  forall(u, list_cnodes) {
    node t;
    myforall(t, RBC1[u]) is_in_D[t] = false;
    list<edge> el;
    forall_out_edges(e, node_in_D[u]) { is_in_D[node_in_G[target(e)]] = true; el.append(e); }
    D.del_edges(el);
    myforall_rev(t, RBC1[u])
      if(is_in_D[t]) D.new_edge(node_in_D[u], node_in_D[t]);
   }
  node_array<int> dfspos(D, 0);
  DFS_POS(D, dfspos);
  int tot = D.number_of_nodes();
  back_edge.resize(1, tot);
  for (int i = 1; i ≤ tot; i++) back_edge[i] = nil;
  forall(e, list_back_edges) { node u = source(e); back_edge[dfspos[node_in_D[u]]] = e; }
  return tot;
}
```

Figura 5.11: Código da função *sort_back_edges_by_dfs*.



(a) (b)

DCP$(c)'$:
$\langle(14,w),(14,13),(13,14),(13,12),(12,w),[\mathrm{DCP}(11)],(10,w),(9,8),(8,9),(8,w),$
$(7,w),(14,6),(6,14),(6,w),(14,5),(5,14),(5,4),(4,5),(4,w),(4,3),(3,4)(3,2),(2,3),(2,w)$
$(3,1),(1,3),(1,w),(14,v'),(v',14),\mathrm{DCP}(c),$
$(w,1),(w,2),(w,4),(w,6),(w,7),(w,8),(w,10),(w,12),(w,14)\rangle$

(c)

Figura 5.12: (a) Dado um vértice v, para construir a DCP de num novo C-nó, é necessário decidir a ordem cíclica das arestas incidentes em v, bem como as arestas contidas em cada subárvore T_v^* (ilustradas por triângulos). (b) Construção da DCP(c) em T_v^* (função *embed_ back_ edges*). As arestas de retorno estão em linhas tracejadas. (c) DCP(c) após processar v.

96 ⟨inclui DCP de um c-nó antigo a DCP de um novo c-nó 96⟩≡

```
void sh_planar_impl::add_old_cnode_to_embedding(graph& G, node w, node oldcnode, node u,
                          map<node, list<edge> >& b_edges_repres, list<node>& traversed_nodes,
                          list<node>& emblater, bmlist<edge>& emb_list1)
{ bmlist_item it = RBC1[oldcnode].first_item(),
   itl = RBC1[oldcnode].cyclic_pred(it, nil), itr = RBC1[oldcnode].cyclic_succ(it, nil);
  node jl = RBC1[oldcnode].contents(itl), jr = RBC1[oldcnode].contents(itr);
  list<node> L_l, L_r;
  bmlist_item aux, s = it;
  while (label_b[jl] ≤ dfspos_num[w]) {
     if (label_b[jl] ≡ dfspos_num[w]) L_l.append(jl);
     aux = itl; itl = RBC1[oldcnode].cyclic_pred(itl, s); s = aux; jl = RBC1[oldcnode].contents(itl);
   }
  bmlist_item p = it;
  while (label_b[jr] ≤ dfspos_num[w]) {
     if (label_b[jr] ≡ dfspos_num[w]) L_r.append(jr);
     aux = itr; itr = RBC1[oldcnode].cyclic_succ(itr, p); p = aux; jr = RBC1[oldcnode].contents(itr);
   }
  // u == nil when oldcnode is a terminal node;
  bool flipped = ((L_l.size() > 0 ∧ (jl ≡ u ∨ u ≡ nil)) ∨ (jr ≠ u ∧ u ≠ nil));
  if (flipped) L_l.swap(L_r);
  L_r.reverse_items();
  node t;
  forall(t, L_r) embed_back_edges(G, t, traversed_nodes, b_edges_repres[t], emb_list1);
  if (flipped) embed_list[oldcnode].reverse();
  embed_list[oldcnode].conc(emb_list1); embed_list[oldcnode].swap(emb_list1);
  emblater.conc(L_l, LEDA::before);
}
```

Figura 5.13: Código da função *add_old_cnode_to_embedding*.

em T. A função *embed_root* inclui os arcos restantes devido a r à descrição combinatória plana do grafo de acordo com uma bp-árvore em J, respeitando a ordem dos filhos reais para cada C-nó ativo em J (o código da função *embed_root* é omitido). Na próxima seção, encontra-se a correção e a análise do tempo gasto para esta parte do algoritmo de SH.

As DCP's correspondentes a cada C-nó criado durante a simulação da PC-árvore alterada nas figuras 4.25 e 4.26 (páginas 77 e 78) são ilustradas na figura 5.14. A descrição combinatória plana calculada pela implementação de SH para o grafo na figura 4.1(a) se encontra na figura 5.15.

(b) DCP(-1) :
$\langle(2,3),(2,1),(1,2),(1,3),(3,1),(3,2)\rangle$

(c) DCP(-2) :
$\langle(4,5),(4,3),(3,4),(3,2),(3,1),(1,3),$
$(1,2),(2,1),(2,3),(2,5),(5,2),(5,4)\rangle$

(d) DCP(-3) :
$\langle(7,8),(7,5),(5,7),(4,5),(4,3),(3,4),(3,2),$
$(3,1),(1,3),(1,2),(2,1),(2,3),(2,5),(5,2),$
$(5,4),(1,8),(8,1),(8,7)\rangle$

(e) DCP(-4) :
$\langle(10,11),(10,9),(9,10),(9,11),(11,9),(11,10)\rangle$

(f) DCP(-5) :
$\langle(11,12),(10,11),(10,9),(9,10),(9,11),(11,9),$
$(11,10),(9,8),(8,9),(7,8),(7,5),(5,7),$
$(4,5),(4,3),(3,4),(3,2),(3,1),(1,3),(1,2),$
$(2,1),(2,3),(2,5),(5,2),(5,4),(1,8),(8,1),$
$(8,7),(1,12),(12,1),(12,11)\rangle$

(g) DCP(-6) :
$\langle(12,13),(12,11),(12,1),(1,12),(8,7),(8,1),$
$(1,8),(5,4),(5,2),(2,5),(2,3),(2,1),(1,2),(1,3),$
$(3,1),(3,2),(3,4),(4,3),(4,5),(5,7),(7,5),$
$(7,8),(8,9),(9,8),(11,10),(11,9),(9,11),(9,10),$
$(10,9),(10,11),(11,12),(10,13),(13,10),(13,12)\rangle$

(h) DCP(-7) :
$\langle(13,14),(12,13),(12,11),(12,1),(1,12),(8,7),$
$(8,1),(1,8),(5,4),(5,2),(2,5),(2,3),(2,1),(1,2),$
$(1,3),(3,1),(3,2),(3,4),(4,3),(4,5),(5,7),$
$(7,5),(7,8),(8,9),(9,8),(11,10),(11,9),(9,11),$
$(9,10),(10,9),(10,11),(11,12),(10,13),(13,10),$
$(13,12),(7,6),(6,7),(6,14),(14,6),(14,13)\rangle$

(i) DCP(-8) :
$\langle(14,15),(13,14),(12,13),(12,11),(12,1),(1,12),$
$(8,7),(8,1),(1,8),(5,4),(5,2),(2,5),(2,3),(2,1),$
$(1,2),(1,3),(3,1),(3,2),(3,4),(4,3),(4,5),$
$(5,7),(7,5),(7,8),(8,9),(9,8),(11,10),(11,9),$
$(9,11),(9,10),(10,9),(10,11),(11,12),(10,13),$
$(13,10),(13,12),(7,6),(6,7),(6,14),(14,6),$
$(14,13),(6,15),(15,6),(15,14)\rangle$

Figura 5.14: DCP's correspondentes aos itens (b) a (i) da simulação da PC-árvore alterada nas figuras 4.25 e 4.26 (páginas 77 e 78).

1 :	12, 8, 2, 3
2 :	5, 3, 1
3 :	1, 2, 4
4 :	3, 5, 16
5 :	4, 2, 7
6 :	7, 14, 15
7 :	5, 8, 6
8 :	7, 1, 9
9 :	8, 11, 10
10 :	9, 11, 13
11 :	10, 9, 12
12 :	13, 11, 1
13 :	14, 10, 12
14 :	15, 6, 13
15 :	6, 14, 16
16 :	15, 4

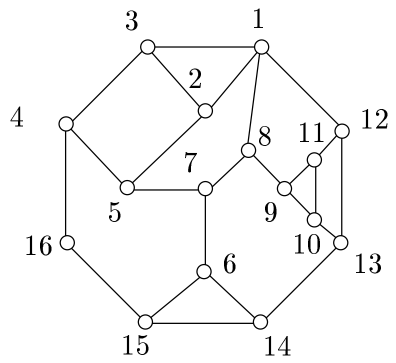


Figura 5.15: Descrição combinatória plana calculada pela implementação para o grafo na figura 4.1(a).

5.3 Correção

Seja c o C-nó criado para representar o bloco B em $H+w$ associado ao componente em $T-L$ correntemente examinado pelo algoritmo. Para mostrar que a construção de uma descrição combinatória plana descrita neste capítulo é correta, basta mostrar que:

(1) DCP(c) é uma descrição combinatória plana de B;

(2) todos os vértices de B que enxergam Y através de $(V_G \setminus L) \setminus V_B$ estão no ciclo facial externo de c.

A seguir, mostra-se que os invariantes acima valem após a construção da DCP de um novo C-nó c para o caso de um terminal no componente corrente de $T-L$. (O caso de dois terminais é análogo e foi omitido.)

Seja Q o caminho em T do terminal a w. Para cada nó v em Q, o algoritmo acrescenta a DCP(c) os arcos devidos a v da seguinte maneira.

Se v é um P-nó que não é terminal, então sejam v' o filho de v em Q e p o pai de v em T. A construção de DCP(c) é feita de tal forma que o arco vp, os arcos devidos a $T_v - T_{v'}$ e o arco vv' aparecem nesta ordem em DCP(c) (figura 5.16). Assim, os únicos vértices desta parte que possivelmente enxergam Y através de $(V_G \setminus L) \setminus V_B$ são p, v e v', que estão no ciclo facial externo de c. Lembrando que os arcos devidos a $T_v - T_{v'}$ são inseridos de acordo com uma bp-árvore, respeitando a ordem dos filhos reais dos C-nós ativos em questão, e que cada arco com ponta inicial em w é inserido em DCP(c) refletindo a ordem de inserção do seu reverso, podemos concluir que a lista DCP(c) é uma descrição combinatória plana.

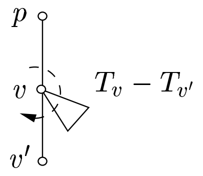


Figura 5.16: Ordem dos arcos em torno de um P-nó v que não é terminal.

Se v é um P-nó e um terminal, então os arcos vp e aqueles devido a T_v são inseridos nesta ordem em DCP(c). Analogamente ao caso anterior, os vértices v e p estão no ciclo facial externo de c (figura 5.17) e DCP(c) é uma descrição combinatória plana.

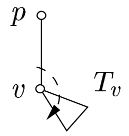


Figura 5.17: Ordem dos arcos em torno de um P-nó v que é terminal.

Se v é um C-nó ativo que não é um terminal, então sejam p o pai de v e v' o filho real de v em Q. Seja Y_v o conjunto dos vértices de B_v que enxergam Y através de $(V_G \setminus L) \setminus V_{B_v}$. Durante a construção da nova descrição, a DCP(v) é invertida, se necessário, de forma que os vértices em Y_v estejam no lado esquerdo, determinado por p e v', no ciclo facial externo de v (figura 5.18). Assim, após concatenar DCP(v) no início de DCP(c), os vértices em Y_v estão no ciclo facial externo de c. Observe que o arco pp', os arcos em DCP(v) e os arco $v'u'$ são inseridos nesta ordem à DCP(c) e que, para cada vértice u no lado direito, determinado por p e v', no ciclo facial externo de v, basta concatenar a lista dos arcos devidos a T_u à DCP(c).

Se v é um C-nó ativo que é terminal, então v' é o filho real de v que enxerga somente X através de $(V_G \setminus L) \setminus V_{B_v}$ e é adjacente a um vértice de Y_v no RBC(v). Este caso é análogo ao caso anterior e isto conclui a correção para o caso de um terminal.

∎

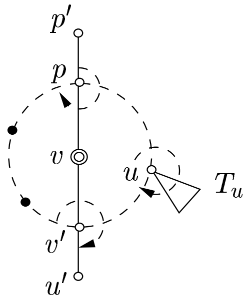


Figura 5.18: Arcos devidos a um C-nó ativo v. Os vértices em $\mathrm{RBC}(v)$ que enxergam Y através de $(V_G \setminus L) \setminus V_{B_v}$ estão à esquerda de v e estão ilustrados por círculos pretos.

5.4 Eficiência

Seja wz uma aresta de retorno e seja J a união dos caminhos de z a w em T. O tempo consumido pela função *calculate_partial_embedding* é proporcional ao tempo gasto para percorrer J mais o tempo gasto para inserir a DCP de cada C-nó ativo no componente à nova DCP pela função *add_old_cnode_to_embedding*.

A função *add_old_cnode_to_embedding* é similar à função *test_cnode_counter* analisada no capítulo anterior, e o tempo gasto para percorrer J é limitado superiormente pelo tempo gasto durante a fase de busca por uma XY-obstrução. A análise para o número de nós percorridos também é análoga ao que foi feito no capítulo anterior e, portanto, o algoritmo de SH constrói uma descrição plana de um grafo planar em tempo $O(n)$.

∎

Obstruções

Neste capítulo, são descritas as adaptações do teste de planaridade descrito no capítulo 4 necessárias para a obtenção de uma lista de arcos de uma obstrução, caso o grafo não seja planar. Abaixo, encontra-se uma descrição de cada uma das funções usadas para este cálculo.

6.1 Informações adicionais para o cálculo da lista de arcos

Para o cálculo da lista de arcos de uma obstrução, são necessárias algumas informações adicionais. Durante o algoritmo de SH, para cada vértice v de G, a variável global *node_ label_ b[v]* é um apontador para um vértice que é descendente de v em T cujo valor de h é igual a $b(v)$, e para cada terminal t, a variável global *neighbor_ w_ terminal[t]* é um apontador para um vizinho de w que é descendente de t em T. Durante o pré-processamento, guarda-se T_0 através da variável global *p0[ ]*, que é um apontador para o pai em T_0 definido para cada vértice de G. Se r é a raiz de T_0 então *p0[r]* é *nil*. Além disso, para cada nó v em T_0, a variável global *T0_ edge_ in[v]* guarda um apontador para o arco com ponta inicial no pai de v em T_0 e ponta final em v.

6.2 Obstrução para o teste do contador de um C-nó

A função *obstruction_ edges_ cnode_ counter* (figura 6.1) é acionada quando a função *test_ cnode_ counter*, descrita na seção 4.7, detecta uma obstrução do tipo (o1) durante o teste de um C-nó ativo c. Ela é chamada pela função *test_ obstruction_ from_ terminal_ node* (figura 4.13, página 65) e pela função *find_ obstruction* (figuras 4.11 e 4.12, páginas 63 e 64) nos pontos marcados com (1).

Seja p o pai de c e sejam f, l e r vértices do RBC(c) tais que f tem um descendente em T que é vizinho de w, e l e r são vértices distintos de p e f tais que $b(l)$ e $b(r)$ são maiores que bp(w) e l e r estão em lados opostos do RBC(c) determinados por p e f (figura 6.2).

102 ⟨obstrução pelo contador do c-nó 102⟩≡ 106 ▷

```
void sh_planar_impl::obstruction_edges_cnode_counter(graph& G, node cnode, node w,
                                                     node jl, node jr, node t1, node t2)
{ bool flag = false;
  node f = nil, t12 = nil, t22 = nil;
  if (t1 ≠ nil ∧ t1 ≠ cnode) t12 = last_pnode(t1, cnode);
  if (t2 ≠ nil ∧ t2 ≠ cnode) t22 = last_pnode(t2, cnode);
  // seachs for a node f in RBC[cnode] between jl and jr s.t.
  // f has a descendant that is a neighbor of w in G;
  myforall(f, RBC[cnode])
    if (f ≡ jl ∨ f ≡ jr) flag = true;
    else if (flag ∧ (f ≡ t12 ∨ f ≡ t22 ∨ label_b[f] ≡ dfspos_num[w])) break;
  if (f ≡ t12) node_label_b[f] = neighbor_w_terminal[t1];
  else if (f ≡ t22) node_label_b[f] = neighbor_w_terminal[t2];
  edge e = find_edge(G, node_label_b[f], w);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[jl], node_with_dfspos[label_b[jl]]);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[jr], node_with_dfspos[label_b[jr]]);
  obstruction_edges.append(e);
  node w1 = node_with_dfspos[label_b[jl]], w2 = node_with_dfspos[label_b[jr]];
  if (dfspos_num[w2] < dfspos_num[w1]) leda_swap(w1, w2);
  list_edges_upward_T0(parent[cnode], w2);
  list_edges_upward_T0(node_label_b[jl], jl);
  list_edges_upward_T0(node_label_b[jr], jr);
  list_edges_upward_T0(node_label_b[f], f);
  extract_boundary_cycle(cnode, obstruction_edges);
}
```

Figura 6.1: Código da função *obstruction_edges_cnode_counter*.

Para determinar f, a função *obstruction_ edges_ cnode_ counter* procura, no lado do RBC(c) determinado por l e r e oposto a p, por um vértice cujo valor de b é igual ao bp(w) ou por um vértice que é ancestral de um terminal. (A função *last_ pnode* determina o último P-nó no caminho de t a c em T e consome tempo proporcional ao comprimento do caminho.)

Sejam w_1 e w_2 tais que $\text{bp}(w_i) = b(l)$ ou $\text{bp}(w_i) = b(r)$, para $i = 1, 2$, e $\text{bp}(w_1) \leq \text{bp}(w_2)$. Sejam l' e r' os vértices apontados por *node_ label_ b[l]* e por *node_ label_ b[r]*, respectivamente, e seja u um descendente de f que é vizinho de w, obtido da seguinte maneira: se $b(f) = \text{bp}(w)$, então u é o vértice apontado por *node_ label_ b[f]* (figura 6.2(a)). Caso contrário, f é um ancestral de um terminal t (figura 6.2(b)), e u é o vértice apontado por *neighbor_ w_ terminal[t]*.

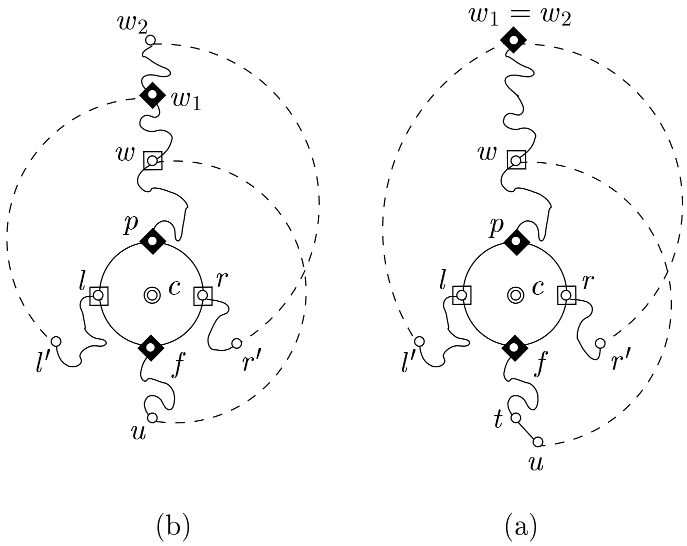


Figura 6.2: Obstrução detectada pela função *test_ cnode_ counter*, onde p é o pai do C-nó c, w_1 e w_2 são ancestrais de w em T e l e r são vértices em RBC(c) cujos valores de b são maiores que bp(w). (a) f é um vértice em RBC(c) cujo valor de b é igual ao bp(w). (b) f é um vértice em RBC(c) que é ancestral de um terminal t.

A lista dos arcos de uma obstrução é representada pela variável global *obstruction_ edges*. Neste caso, ela é construída da seguinte maneira.

Primeiramente, os arcos uw, $l'w_i$ e $r'w_j$ (i e j em $\{1, 2\}$) são adicionados a essa lista. Para encontrar um arco xy em G, usa-se a função *find_ edge* (figura 6.3), que percorre a lista de adjacência de x e devolve um apontador para o arco xy, caso exista, ou *nil* caso contrário.

Feito isso, adiciona-se a essa lista, os arcos dos seguintes caminhos em T_0: de p a w_2, de l' a l, de r' a r e de u a f. A adição dos arcos de um caminho em T_0 de x a y à lista *obstruction_ edges* é feita pela rotina *list_ edges_ upward_ T0* (figura 6.4). Para cada vértice v distinto de y nesse caminho, essa função adiciona o arco $v'v$ a *obstruction_ edges*, onde v' é o pai de v em T_0.

104a ⟨procura arco 104a⟩≡

```
edge sh_planar_impl::find_edge(graph& G, node x, node y)
{ edge e;
  forall_out_edges(e, x) {
     node v = target(e);
     if (v ≡ y) return e;
   }
  return nil;
}
```

Figura 6.3: Código da função *find_edge*.

104b ⟨adiciona arcos subindo em T0 104b⟩≡

```
void sh_planar_impl::list_edges_upward_T0(node x, node y)
{ if (x ≡ y) return;
  node v = x;
  while (v ≠ y) {
     edge e = T0_edge_in[v];
     obstruction_edges.append(e);
     v = p0[v];
   }
}
```

Figura 6.4: Código da função *list_edges_upward_T0*.

Finalmente, os arcos do ciclo facial externo de c são adicionados à lista *obstruction_edges* pela rotina *extract_boundary_cycle* (figura 6.5), que primeiramente calcula as listas de adjacências do bloco em $H+w$ representado por c através da lista DCP(c), representada pela variável global *embed_list[c]*, e depois obtém o ciclo facial determinado por DCP(c) e p.

105 ⟨lista de arcos na fronteira da face externa 105⟩≡

```
void sh_planar_impl::extract_boundary_cycle(node cnode, list<edge>& list_edges)
{ map<node, list<edge> > el;
  edge e;
  node u;
  myforall(e, embed_list[cnode]) { u = source(e); el[u].append(e); }
  u = parent[cnode];
  e = el[u].contents(el[u].first()); list_edges.append(e);
  u = target(e);
  while (u ≠ parent[cnode]) {
      e = el[u].contents(el[u].first());
      list_edges.append(e);
      u = target(e);
   }
}
```

Figura 6.5: Código da função *extract_boundary_cycle*.

6.3 Obstrução para *cnode_of_possible_k33_obstruction*

Seja d o C-nó (ativo) apontado pela variável global *cnode_of_possible_k33_obstruction*. Conforme descrito na seção 4.7, d é o primeiro C-nó detectado nas condições da figura 4.14(b) (página 66) pela função *test_cnode_counter*.

Seja Q o caminho de um terminal t a w em T e seja Q' o caminho do pai de d a w em T. Existe uma obstrução do tipo (o1) se, para um nó u em Q', u é vizinho de um vértice em $L \setminus \{w\}$ ou u tem um filho (P-nó) v em T que não está em Q e v tem um descendente em T que é vizinho de um vértice em $L \setminus \{w\}$ (figura 4.14(c)). (Observe que, no caso de dois terminais, v pode ser simplesmente o terminal t' distinto de t, se t' é um P-nó, ou o pai de t' em T, caso contrário.)

A função *obstruction_edges_possible_obstr_confirmed* (figura 6.6) é acionada pela função *find_obstruction* (pontos marcados com (2) nas figuras 4.11 e 4.12, páginas 63 e 64) para calcular uma lista de arcos de uma obstrução do tipo (o1), caso exista.

Sejam p o pai de d e l, r e f como mostrados na figura 6.7. Sejam w_1 e w_2 tais que $\text{bp}(w_i) = b(p)$ ou $\text{bp}(w_i) = b(f)$, para $i = 1, 2$, e $\text{bp}(w_1) \leq \text{bp}(w_2)$. Seja l' o vértice apon-

106 ⟨obstrução pelo contador do c-nó 102⟩+≡ ◁102

```
void sh_planar_impl::obstruction_edges_possible_obstr_confirmed(graph& G,
                                                                node w, node t, node v)
{ node cnode = cnode_of_possible_k33_obstruction,
    f = obstruction_nodes.pop(), jl = obstruction_nodes.pop(), jr = obstruction_nodes.pop();
  if (label_b[jl] > dfspos_num[w]) node_label_b[jl] = neighbor_w_terminal[t];
  if (label_b[jr] > dfspos_num[w]) node_label_b[jr] = neighbor_w_terminal[t];
  edge e = find_edge(G, node_label_b[jl], w);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[jr], w);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[f], node_with_dfspos[label_b[f]]);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[v], node_with_dfspos[label_b[v]]);
  obstruction_edges.append(e);
  node w1 = node_with_dfspos[label_b[v]], w2 = node_with_dfspos[label_b[f]];
  if (dfspos_num[w2] < dfspos_num[w1]) leda_swap(w1, w2);
  list_edges_upward_T0(w, w2);
  list_edges_upward_T0(node_label_b[jl], jl);
  list_edges_upward_T0(node_label_b[jr], jr);
  list_edges_upward_T0(node_label_b[f], f);
  node m = lca_between(parent[cnode], v, p0);
  list_edges_upward_T0(parent[cnode], m);
  list_edges_upward_T0(node_label_b[v], m);
  extract_boundary_cycle(cnode, obstruction_edges);
}
```

Figura 6.6: Código da função *obstruction_ edges_ possible_ obstr_ confirmed*.

tado por *node_ label_ b[l]*, se $b(l) = \mathrm{bp}(w)$, ou o vértice apontado por *neighbor_ w_ terminal[t]*, onde t é um terminal que é descendente de l, caso contrário. Analogamente, seja r' o vértice apontado por *node_ label_ b[r]*, se $b(r) = \mathrm{bp}(w)$, ou o vértice apontado por *neighbor_ w_ terminal[t′]*, onde t' é um terminal que é descendente de r, caso contrário. Seja u o vértice apontado por *node_ label_ b[f]*.

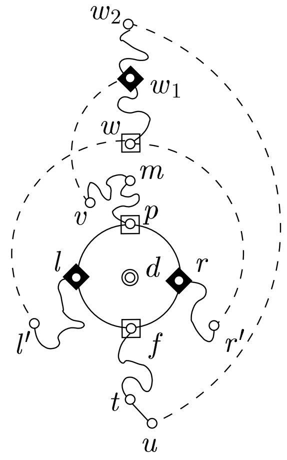


Figura 6.7: Obstrução detectada pelo teste de Q' da figura 4.14(c) (página 66), para o caso de um terminal, ou pela presença de um C-nó d como na figura 4.14(b), para o caso de dois terminais.

Seja v o P-nó devolvido pela função *find_ node_ with_ label_ b_ greater_ than_ dfs_ w* (figura 4.17, página 69) para o caso de um terminal. Para o caso de dois terminais, seja t' o terminal distinto de t. Neste caso, $v = t'$, se t' é um P-nó (figura 6.8(a)). Caso t' seja um C-nó, v é o pai (P-nó) de t' e copia-se as informações das variáveis *node_ label_ b[ ]* e *label_ b[ ]* de t' para v, pois t' pode ser um C-nó terminal tal que $b(t') < b(t)$ e v é o ancestral mais baixo entre os dois terminais em T (figura 6.8(b)). Finalmente, seja m o ancestral mais baixo entre p e v em T_0.

A lista dos arcos para esta obstrução é construída da seguinte maneira. Primeiramente, os arcos $l'w$, $r'w$, uw_i e vw_j, onde $\mathrm{bp}(w_i) = b(u)$ e $\mathrm{bp}(w_j) = b(v)$ (i e j em $\{1, 2\}$), são adicionados a essa lista. Em seguida, adiciona-se a esta lista, os arcos dos seguintes caminhos em T_0: de w a w_2, de l' a l, de r' a r, de u a f, de p a m e de v a m. Por fim, os arcos do ciclo facial externo de c são adicionados à lista *obstruction_ edges*.

6.4 Cálculo de uma subdivisão de K_5

A função *obstruction_ edges_ K5* (figura 6.9) é acionada pela função *find_ obstruction* nos pontos marcados com (3) e (4) nas figuras 4.11 e 4.12 (páginas 63 e 64) para o caso de calcular uma subdivisão de K_5 em G. Seja c o C-nó para o qual foi detectada uma obstrução (C, v_1, v_2, v_3) do

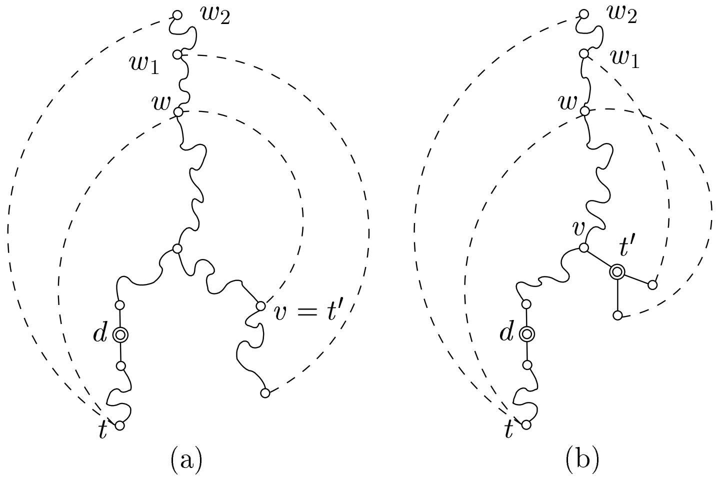


Figura 6.8: O C-nó d é o C-nó apontado por *cnode_ of_ possible_ k33_ obstruction*, e t e t' são dois terminais. (a) v coincide com o P-nó terminal t'. (b) v é o pai do C-nó terminal t'.

tipo (o2) e seja w_1 o vértice de menor bp-número entre os vértices apontados por *node_ label_ b[v_i]*, para $i = 1, 2, 3$. A função *obstruction_ edges_ K5* (figura 6.9) é acionada se

($K_5$1) ou v_i é um terminal ou $v_i = p$, onde p é o pai de c e existe um descendente v de p em T que não é descendente de c e é vizinho de um vértice em $L \setminus \{w\}$;

($K_5$2) pelo menos dois vértices de $\{v_1, v_2, v_3\}$ têm o apontador *node_ label_ b[ ]* para w_1 (figuras 6.10(a) e (b)).

Observe que, se algum v_i é o pai de um C-nó terminal, também é possível construir uma subdivisão de K_5 em G (figura 6.10(c)). Mas neste caso, podemos obter uma subdivisão de $K_{3,3}$, conforme é descrito mais adiante.

Seja w_2 o vértice de maior bp-número entre os vértices apontados por *node_ label_ b[v_i]*, para $i = 1, 2, 3$. A lista dos arcos para esta obstrução é construída da seguinte maneira. Primeiramente, adiciona-se a esta lista, os arcos do caminho de w a w_2 em T_0. Em seguida, os arcos do ciclo facial externo de c são adicionados à lista *obstruction_ edges*.

Finalmente, se $v_i = p$ para algum i em $\{1, 2, 3\}$, então seja v o vértice apontado por *node_ label_ b[v_i]*. A função *obstr_ edges_ pnode* (figura 6.11) adiciona à lista *obstruction_ edges* o arco vw_j (j em $\{1, 2\}$) e os arcos no caminho de v a w em T_0.

Se v_i é um P-nó terminal, então sejam u_1 e u_2 os vértices apontados por *neighbor_ w_ terminal[v_i]* e por *node_ label_ b[v_i]*, respectivamente. A função *obstr_ edges_ terminal* (figura 6.12) adiciona os arcos u_1w e u_2w_j à lista *obstruction_ edges*, além de adicionar os caminhos de u_1 a v_i e de u_2 a v_i em T_0 nessa lista.

109a ⟨obstrução K5 109a⟩≡

```
void sh_planar_impl::obstruction_edges_K5(graph& G, node w, node cnode,
                                          node v1, node v2, node v3)
{ if (v3 ≡ nil) v3 = parent[cnode];
  int max = label_b[v1];
  if (max < label_b[v2]) max = label_b[v2];
  if (max < label_b[v3]) max = label_b[v3];
  list_edges_upward_T0(w, node_with_dfspos[max], obstruction_edges);
  extract_boundary_cycle(cnode, obstruction_edges);
  obstr_edges_terminal(G, w, v1, v1);
  obstr_edges_terminal(G, w, v2, v2);
  if (v3 ≠ parent[cnode]) obstr_edges_terminal(G, w, v3, v3);
  else obstr_edges_pnode(G, parent[cnode], w);
}
```

Figura 6.9: Código da função *obstruction_edges_K5*.

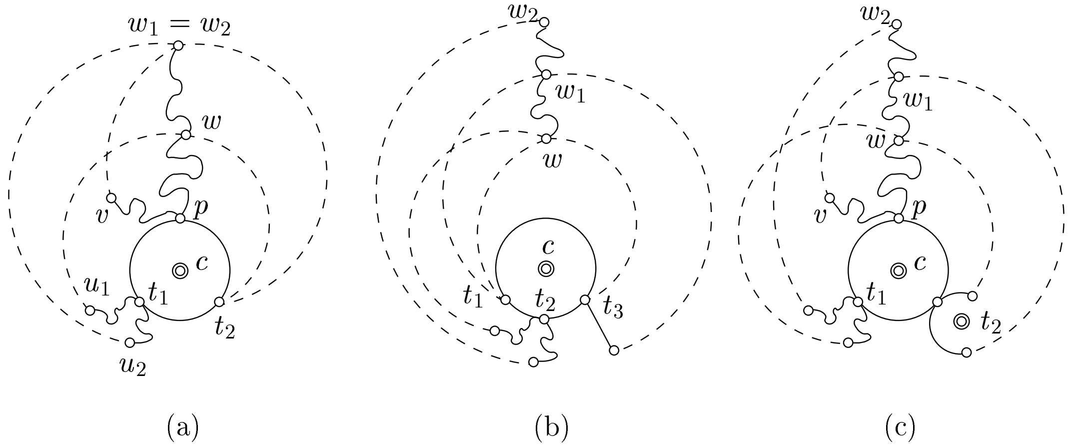


Figura 6.10: Cálculo de uma subdivisão de K_5. (a) Caso de dois terminais t_1 e t_2. (b) Caso em que ocorre três terminais t_1, t_2 e t_3. (c) t_2 é um C-nó terminal.

109b ⟨arcos do pai de um c-nó 109b⟩≡

```
void sh_planar_impl::obstr_edges_pnode(graph& G, node p, node u)
{ edge e = find_edge(G, node_label_b[p], node_with_dfspos[label_b[p]]);
  obstruction_edges.append(e);
  list_edges_upward_T0(node_label_b[p], u);
}
```

Figura 6.11: Código da função *obstr_edges_pnode*.

110 ⟨arcos em T0 do terminal e arcos de retorno 110⟩≡

```
void sh_planar_impl::obstr_edges_terminal(graph& G, node w, node t, node u)
{ edge e = find_edge(G, neighbor_w_terminal[t], w);
  obstruction_edges.append(e);
  e = find_edge(G, node_label_b[t], node_with_dfspos[label_b[t]]);
  obstruction_edges.append(e);
  node mm = lca_between(node_label_b[t], neighbor_w_terminal[t], p0);
  list_edges_upward_T0(neighbor_w_terminal[t], mm, obstruction_edges);
  list_edges_upward_T0(node_label_b[t], mm, obstruction_edges);
  list_edges_upward_T0(mm, u, obstruction_edges);
}
```

Figura 6.12: Código da função *obstr_edges_terminal*.

Na seção 6.5 a seguir, é descrita a construção de uma subdivisão de $K_{3,3}$ no caso de apenas a condição ($K_5 1$) ser satisfeita. Já nas seções seguintes (6.6 e 6.7), uma subdivisão de $K_{3,3}$ é construída no caso da condição ($K_5 1$) não ser satisfeita.

6.5 Subdivisão de $K_{3,3}$ devido ao grau de $w1$

A função *obstruction_edges_countmin_1* (figura 6.13) é acionada pela função *find_obstruction* nos pontos marcados com (3) e (4) nas figuras 4.11 e 4.12 (páginas 63 e 64) para calcular uma subdivisão de $K_{3,3}$ em G para uma obstrução (C, v_1, v_2, v_3) do tipo (o2). Seja c o C-nó para o qual foi detectada a obstrução e seja w_1 o vértice de menor bp-número entre os vértices apontados por *node_label_b[v_i]*, para $i = 1, 2, 3$. Neste caso daria para construir uma subdivisão de K_5 em G se não fosse pela presença de um único vértice em $\{v_1, v_2, v_3\}$ tal que variável *node_label_b[v_i]* aponta para w_1 (figura 6.14). A seguir encontra-se uma descrição da função *obstruction_edges_countmin_1*.

Primeiramente, os vértices v_i são ordenados em ordem não decrescente de b pela função *sort_by_label_b* (figura 6.15). Nela, a função *leda_swap(x,y)* é usada para trocar os valores de x com o de y. Para $i = 1, 2, 3$, seja w_i o vértice de bp-número igual a $b(v_i)$. São adicionados à lista *obstruction_edges* os arcos do caminho de w a w_3 em T_0, pela função *list_edges_upward_T0*, e os arcos no lado do ciclo facial externo de c, determinado por v_2 e v_3, que contém v_1, pela função *add_part_of_bc* (figura 6.16).

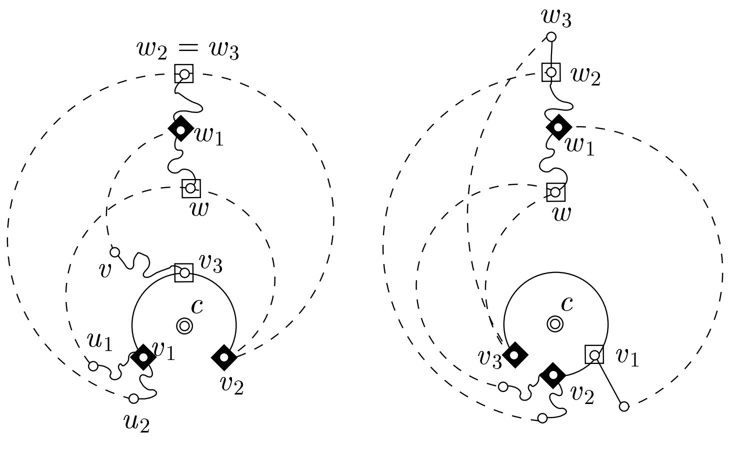
Em seguida, seja v o vértice apontado por *node_label_b[v_1]*. Adiciona-se à *obstruction_edges* o arco vw_1 e os arcos no caminho de v a v_1 em T_0.

Finalmente, para $i = 2, 3$, v_i é ou o pai de c ou é um P-nó terminal e adiciona-se os arcos

111 ⟨obstrução de três no mesmo c-nó 111⟩≡

```
void sh_planar_impl::obstruction_edges_countmin_1(graph& G, node w, node cnode,
                                                  node v1, node v2, node v3)
{ if (v3 ≡ nil) v3 = parent[cnode];
  sort_by_label_b(v1, v2, v3);
  list_edges_upward_T0(w, node_with_dfspos[label_b[v3]]);
  add_part_of_bc(cnode, v1, v2, v3);
  edge e = find_edge(G, node_label_b[v1], node_with_dfspos[label_b[v1]]);
  obstruction_edges.append(e);
  list_edges_upward_T0(node_label_b[v1], v1);
  if (v2 ≠ parent[cnode]) obstr_edges_terminal(G, w, v2, v2);
  else obstr_edges_pnode(G, parent[cnode], w);
  if (w3 ≠ parent[cnode]) obstr_edges_terminal(G, w, v3, v3);
  else obstr_edges_pnode(G, parent[cnode], w);
}
```

Figura 6.13: Código da função *obstruction_edges_countmin_1*.



(a) (b)

Figura 6.14: Cálculo de uma subdivisão de $K_{3,3}$. Neste caso, não é possível obter uma subdivisão de K_5 pois w_1 não tem grau 4. (a) Caso de dois terminais v_1 e v_2 (v_3 é o pai de c em T). (b) Caso de três terminais v_1, v_2 e v_3.

112a ⟨ordena nós pelo valor de b 112a⟩≡

```
void sh_planar_impl::sort_by_label_b(node &t1, node &t2, node &t3)
{ if (label_b[t2] < label_b[t1]) leda_swap(t1, t2);
  if (label_b[t3] < label_b[t1]) leda_swap(t1, t3);
  if (label_b[t3] < label_b[t2]) leda_swap(t2, t3);
}
```

Figura 6.15: Código da função *sort_by_label_b*.

112b ⟨alguns arcos na fronteira da face externa 112b⟩≡

```
void sh_planar_impl::add_part_of_bc(node cnode, node w1, node w2, node w3)
{ list<edge> bc, el1, el2;
  extract_boundary_cycle(cnode, bc);
  edge e;
  bool flag = false;
  int side = 0;
  forall(e, bc) {
      if (source(e) ≡ w1)
        if (flag) side = 1; else side = 2;
      if (source(e) ≡ w2 ∨ source(e) ≡ w3) flag = ¬flag;
      if (flag) el1.append(e); else el2.append(e);
    } // forall
  if (side ≡ 1) obstruction_edges.conc(el1, LEDA::after);
  else obstruction_edges.conc(el2, LEDA::after);
}
```

Figura 6.16: Código da função *add_part_of_bc*.

devidos a v_i da mesma maneira que foi descrita na seção anterior, através das funções *obstr_edges_pnode* e *obstr_edges_terminal*, respectivamente.

6.6 Subdivisão de $K_{3,3}$ com parte do ciclo externo de um C-nó

Seja (C, v_1, v_2, v_3) uma obstrução do tipo (o2) detectada para um C-nó c e seja m o ancestral comum em T_0 de pelo menos dois entre v_1, v_2 e v_3, com o menor bp-número. A função *obstruction_edges_countmin_2_3* (figura 6.17) é acionada pela função *find_obstruction* nos pontos marcados com (3) e (4) nas figuras 4.11 e 4.12 (páginas 63 e 64) para calcular uma subdivisão de $K_{3,3}$ em G se a condição $(K_5 1)$ não é satisfeita e se m coincide com um terminal (figura 6.18).

113 ⟨obstrução de dois no mesmo c-nó 113⟩≡

```
void sh_planar_impl::obstruction_edges_countmin_2_3(graph& G, node w, node cnode,
                                                    node v1, node v2, node v3, node q, node v)
{ node k1 = v1, k2 = v2, k3 = v3;
  if (v3 ≡ nil) k3 = v;
  sort_by_label_b(k1, k2, k3);
  list_edges_upward_T0(node_with_dfspos[label_b[k1]], node_with_dfspos[label_b[k3]]);
  k2 = last_pnode(v2, cnode);
  if (k2 ≠ q) add_part_of_bc(cnode, q, v1, k2);
  else add_part_of_bc(cnode, q, v1, parent[cnode]);
  obstr_edges_terminal(G, w, v1, v1);
  obstr_edges_terminal(G, w, v2, k2);
  if (v3 ≠ nil) {
      node u = lca_between(node_label_b[v3], q, p0);
      // "u" can be an ancestor of q when v3 is not descendant of cnode;
      obstr_edges_terminal(G, w, v3, u);
      list_edges_upward_T0(q, u);
    }
  else {
      obstr_edges_pnode(G, v, w);
      node mm = lca_between(v, parent[cnode], p0);
      list_edges_upward_T0(parent[cnode], mm);
    }
}
```

Figura 6.17: Código da função *obstruction_edges_countmin_2_3*.

Seja q um vértice tal que, se $v_i = p$ para algum i em $\{1, 2, 3\}$, então q coincide com p (figura 6.18(a)). Caso contrário, q coincide com v_j, j em $\{1, 2, 3\}$, tal que v_j não é terminal (figura 6.18(b)). A lista de arcos da obstrução é calculada da seguinte maneira.

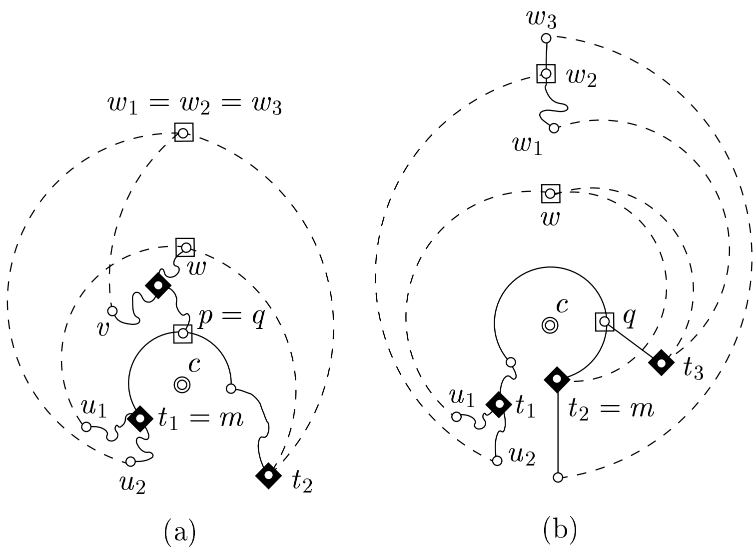


Figura 6.18: Cálculo de uma subdivisão de $K_{3,3}$ com um "pedaço" do ciclo facial externo de um C-nó c. (a) Caso de dois terminais t_1 e t_2. (b) Caso de três terminais t_1, t_2 e t_3.

Primeiramente, os vértices v_1, v_2 e v_3 são ordenados em ordem não-decrescente de b e, para $i = 1, 2, 3$, seja w_i o vértice de bp-número igual a $b(v_i)$. Adiciona-se à lista *obstruction_edges* os arcos do caminho de w_1 a w_3 em T_0 e os arcos no lado do ciclo facial externo de c que contém q, determinado por v_j e v_k, onde j e k estão em $\{1, 2, 3\}$ e $j \neq k$, $v_j \neq q$ e $v_k \neq q$. (A função *last_pnode(x,y)* devolve o último P-nó no caminho de x a y em T, caso exista.)

Para finalizar, adiciona-se os arcos devidos a v_i da seguinte maneira. Se v_i coincide com p, então adiciona-se os arcos devidos a v_i conforme descrito na seção 6.4, através da função *obstr_edges_pnode*. Caso contrário, seja t_i o terminal que é descendente de v_i em T e sejam u_1 e u_2 os vértices apontados por *neighbor_w_terminal[t_i]* e por *node_label_b[t_i]*, respectivamente. A função *obstr_edges_terminal* adiciona os arcos u_1w e u_2w_i à lista *obstruction_edges*, além de adicionar os caminhos em T_0 de u_1 a t_i, de u_2 a t_i e de t_i a v_i nessa lista.

6.7 Subdivisão de $K_{3,3}$, subindo em T_0

A função *obstruction_edges_T0* (figura 6.19) é acionada pela função *find_obstruction* nos pontos marcados com (3) e (4) nas figuras 4.11 e 4.12 (páginas 63 e 64) para calcular uma subdivisão de $K_{3,3}$ em G ou devido a uma obstrução do tipo (o3) (m, K_1, K_2, K_3) (figuras 6.20(a) e (b)), ou devido a uma obstrução do tipo (o2) (C, v_1, v_2, v_3) tal que a condição $(K_5 1)$ não é satisfeita e o ancestral comum m em T_0 de pelo menos dois entre v_1, v_2 e v_3 com o menor bp-número não é um terminal (figura 6.20(c)).

115 ⟨obstrução por T0 115⟩≡

```
void sh_planar_impl::obstruction_edges_T0(graph& G, node w, node parent_cnode,
                                           node t1, node t2, node t3, node v)
{ if (t3 ≡ nil) t3 = v;
  node q1 = t1, q2 = t2, q3 = t3;
  sort_by_label_b(q1, q2, q3);
  list_edges_upward_T0(node_with_dfspos[label_b[q1]], node_with_dfspos[label_b[q3]]);
  q1 = t1; q2 = t2; q3 = t3;
  if (is_cnode(q1)) q1 = parent[q1];
  if (is_cnode(q2)) q2 = parent[q2];
  if (is_cnode(q3)) q3 = parent[q3];
  node m1 = lca_between(q1, q2, p0), m2 = lca_between(q1, q3, p0), m3 = lca_between(q2, q3, p0);
  int min = dfspos_num[m1], max = dfspos_num[m1];
  if (dfspos_num[m2] < min) { leda_swap(t2, t3); min = dfspos_num[m2]; }
  if (dfspos_num[m3] < min) {
      node k1 = t2, k2 = t3, k3 = t1;
      t1 = k1; t2 = k2; t3 = k3;
      min = dfspos_num[m3];
    }
  if (dfspos_num[m2] > max) max = dfspos_num[m2];
  if (dfspos_num[m3] > max) max = dfspos_num[m3];
  obstr_edges_terminal(G, w, t1, node_with_dfspos[max]);
  obstr_edges_terminal(G, w, t2, node_with_dfspos[min]);
  if (t3 ≠ v) obstr_edges_terminal(G, w, t3, node_with_dfspos[max]);
  else {
      node u = w;
      if (lca_between(v, node_with_dfspos[min], p0) ≡ node_with_dfspos[min])
        { // v is in the same c-node of node_with_dfspos[min];
          node cnode = active_cnode_of(true, v);
          add_part_of_bc(cnode, parent[cnode], v, node_with_dfspos[min]);
          u = v;
          list_edges_upward_T0(parent[cnode], w);
        }
      obstr_edges_pnode(G, v, u);
    }
}
```

Figura 6.19: Código da função *obstruction_edges_T0*.

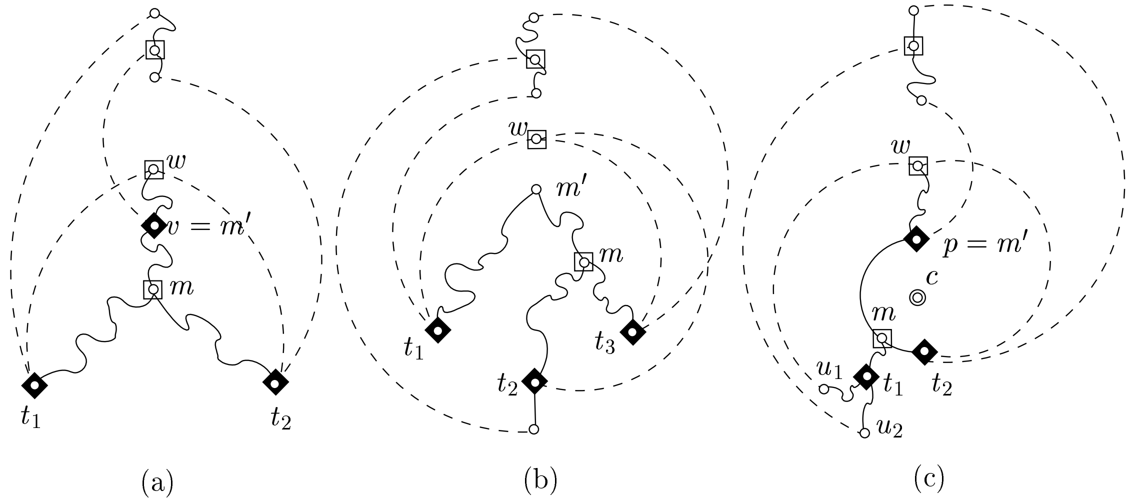


Figura 6.20: Cálculo de uma subdivisão de $K_{3,3}$ subindo em T_0. (a) Caso de três terminais t_1, t_2 e t_3. (b) Caso de dois terminais t_1 e t_2.

Sejam q_1, q_2 e q_3 vértices distintos tais que, para $i = 1, 2, 3$, q_i coincide com um terminal ou com o vértice v devolvido pela função *find_node_with_label_b_greater_than_dfs_w*. A lista de arcos da obstrução é construída da seguinte maneira. Primeiramente, os vértices q_1, q_2 e q_3 são ordenados em ordem não-decrescente de b e, para $i = 1, 2, 3$, seja w_i o vértice de bp-número igual a $b(q_i)$. Os arcos no caminho de w_1 a w_3 em T_0 são adicionados à lista *obstruction_edges*.

Sejam m e m' os ancestrais comuns em T_0 de pelo menos dois entre q_1, q_2 e q_3 com o menor e com o maior bp-número, respectivamente. Seja q_j um vértice em $\{q_1, q_2, q_3\}$ que é descendente de m e sejam q_k e q_l vértices distintos em $\{q_1, q_2, q_3\}$ tais que $k \neq j$ e $l \neq j$. Falta adicionar a *obstruction_edges* os arcos devidos aos vértices q_1, q_2 e q_3. Isto é feito da seguinte maneira.

Para $i = 1, 2, 3$, se q_i é um terminal, então sejam u_1 e u_2 os vértices apontados por *neighbor_w_terminal[q_i]* e por *node_label_b[q_i]*, respectivamente. A função *obstr_edges_terminal* (figura 6.12) adiciona os arcos u_1w e u_2w_i à lista *obstruction_edges*, além de adicionar os caminhos em T_0 de u_1 a q_i e de u_2 a q_i nessa lista. Além disso, se $i = k$ ou $i = l$ então adiciona-se o caminho em T_0 de q_i a m' à lista de arcos. Caso contrário, $i = j$ e adiciona-se o caminho em T_0 de q_j a m à lista de arcos.

Se q_i não é um terminal, então $q_i = v$. Seja u o vértice apontado por *node_label_b[v]* e seja s um vértice tal que s coincide com v se v é um descendente de m em T_0, e s coincide com w caso contrário.

Se v é um descendente de m em T_0, então v e m são irmãos em T e estão em RBC(d), para algum C-nó ativo d (figura 6.21), e este é o único caso especial em que a função *obstruction_edges_T0* precisa escolher um dos lados do ciclo facial externo de um C-nó: adiciona-se à lista *obstruction_ edges* os arcos que estão no lado do ciclo facial externo de d determinado por v e por m que contém o pai de d. Além disso, adiciona-se à lista de arcos da obstrução os arcos no caminho do pai de d a w em T_0.

Finalmente, se $q_i = v$, a função *obstr_ edges_ pnode* (figura 6.11) adiciona à lista *obstruction_ edges* o arco uw_i e o caminho de u a s em T_0.

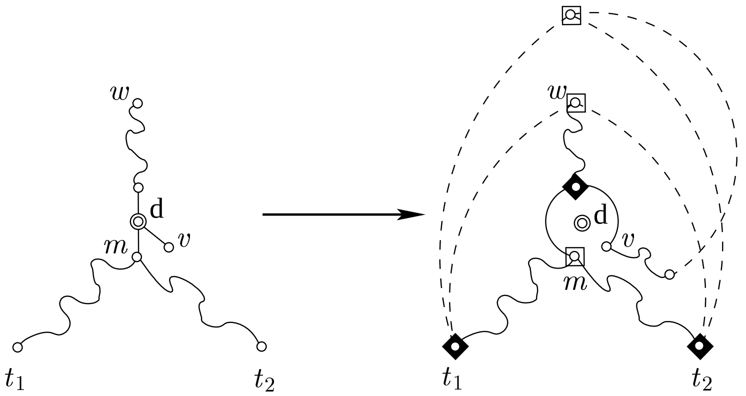


Figura 6.21: Na figura 6.20(a), se v é descendente de m em T_0 então v e m estão em RBC(d), para um C-nó ativo d, e adiciona-se à lista *obstruction_ edges* os arcos que estão no lado do ciclo facial externo de d determinado por v e por m que contém o pai de d.

117 ⟨encontra c-nó ativo 117⟩≡

```
node sh_planar_impl::active_cnode_of(node u)
{ node cnode = u;
  if (¬is_cnode(u)) cnode = parent[u];
  if (¬is_cnode(cnode)) return nil;
  while (is_cnode(parent[cnode])) cnode = parent[cnode];
  return cnode;
}
```

Figura 6.22: Código da função *active_ cnode_ of*.

6.8 Correção

Observe que os caminhos em T_0 determinados por cada função que calcula uma lista de arcos de uma obstrução são internamente disjuntos entre si. Além disso, para cada função que inclui o ciclo facial externo C (ou parte dele) de um C-nó na lista *obstruction_ edges*, cada caminho em T_0 calculado pela função é internamente disjunto de C. Disso decorre facilmente a correção desta parte do algoritmo.

∎

6.9 Eficiência

Cada função apresentada neste capítulo executa, basicamente, três tarefas para o cálculo da lista *obstruction_edges*: cálculo de um número constante de caminhos em T_0; cálculo de no máximo um ciclo facial externo de um C-nó; e procura por um número constante de arcos (no máximo seis arcos que representam uma aresta de retorno em T, com uma ponta em w ou em ancestrais de w).

A adição de arcos em caminhos em T_0 à lista *obstruction_edges* é feita pela função *list_edges_upward_T0* (figura 6.4) e o tempo gasto por essa função é proporcional ao comprimento do caminho. Como cada caminho em T_0 tem comprimento máximo $n-1$, onde n é o número total de vértices no grafo, então esta tarefa consome tempo $O(n)$.

Para o cálculo do ciclo facial externo de um C-nó c, feito pela rotina *extract_boundary_cycle* (figura 6.5)), percorre-se no máximo duas vezes o número total de arcos no bloco em $H+w$ representado por c. Se a função *add_part_of_bc* (figura 6.16) é executada, então percorre-se no máximo mais uma vez o número total de arcos no bloco em $H+w$ representado por c. Como B_c é planar, então o tempo consumido por esta tarefa é $O(n)$.

Finalmente, para encontrar um arco no grafo, usa-se a função *find_edge* (figura 6.3), que percorre a lista de adjacência da ponta inicial do arco, consumindo assim tempo $O(n)$. Logo, totalizando o tempo gasto pelas três tarefas, tem-se que o tempo consumido por cada uma das funções que calcula uma lista de arcos de um obstrução é $O(n)$.

∎

Capítulo 7

Resultados experimentais

Neste capítulo são mostrados alguns resultados experimentais obtidos da comparação de tempos de execução entre algumas implementações para o problema da planaridade: as implementações de Boyer e Myrvold (BM), de Shih e Hsu (SH) e as duas implementações presentes no LEDA: uma implementação de Booth e Lueker (BL) para o algoritmo de LEC e uma implementação de Hopcroft e Tarjan (HT) para o algoritmo de Auslander, Parter e Goldstein.

Lembre-se de que o problema da planaridade consiste em decidir se um dado grafo é ou não é planar e justificar a resposta com um certificado, que neste caso é ou uma descrição combinatória plana ou um subgrafo de Kuratowski do grafo. Dentre as implementações citadas acima, apenas HT não justifica a resposta para grafos não-planares.

Os dados presentes neste capítulo foram obtidos por uma versão modificada do programa *planarity_time*, que é um programa de demonstração que integra o pacote do LEDA. Basicamente, foram adicionadas as implementações de BM e de SH ao programa original.

Durante os testes realizados, a implementação de BM foi a que apresentou o melhor desempenho, mas apenas esta implementação, dentre as quatro, não foi desenvolvida sobre a plataforma LEDA.

Além disso, o programa *planarity_time* modificado converte o formato de entrada do LEDA para o formato de entrada da implementação de BM, mas a saída construída por BM não foi convertida para o formato LEDA, pois seria necessário mais tempo para estudar e mudar o seu código (os rótulos dos vértices no grafo de entrada são modificados durante a execução de BM). Por esse mesmo motivo, como o código original de BM foi escrito especificamente para resolver o problema da planaridade, não foi medido o tempo gasto pela implementação de BM para testar planaridade sem a construção de um certificado para a resposta.

7.1 Ambiente experimental

Os testes foram rodados num PC com sistema GNU/Linux (RedHat 7.1), com um processador Celeron de 700MHz e 256MB de memória RAM. O programa de testes foi compilado com o `gcc` versão `2.96`, com as opções de compilação `-DLEDA_CHECKING_OFF -O` e com o LEDA versão `4.4`.

As instâncias usadas nos testes foram os mesmos "tipos" de grafos (descritos mais adiante) usados no programa original *planarity_time*. Em todos os testes, para cada valor de n, onde n é o número de vértices do grafo, foi obtida a média entre os valores de tempo relativos a dez grafos. Cada valor de tempo foi medido pela função *used_time* do LEDA, cujo código é ilustrado na figura 7.1.

120 ⟨tempo de uso de CPU 120⟩≡

```
float LEDA::used_time() {
  return float(clock())÷CLOCKS_PER_SEC;
}
float LEDA::used_time(float& T) {
  float t = T;
  T = LEDA::used_time();
  return T-t;
}
```

Figura 7.1: Código da função *used_time* do LEDA versão `4.2`.

Os tipos de grafos usados nos testes foram os seguintes:

(G1) grafos planares aleatórios;

(G2) grafos com um $K_{3,3}$. Cada grafo do tipo (G2) é obtido da seguinte maneira. Seis vértices de um grafo planar aleatório G são escolhidos para a construção de um $K_{3,3}$ em G, adicionando-se arestas a G;

(G3) grafos com um K_5. Cada grafo do tipo (G3) é obtido da seguinte maneira. Cinco vértices de um grafo planar aleatório G são escolhidos para a construção de um K_5 em G, adicionando-se arestas a G;

(G4) grafos planares maximais;

(G5) grafos planares maximais a menos de uma aresta. Cada grafo do tipo (G5) é obtido da seguinte maneira. Seja G um grafo planar maximal e sejam u e v dois vértices de G tais que uv não está em E_G. O grafo $G + uv$ é um grafo do tipo (G5).

As funções usadas para gerar grafos planares maximais e grafos aleatórios planares usam as funções *maximal_planar_map* (figura 7.2) e *random_planar_map* (figura 7.3), respectivamente. A primeira gera um grafo planar maximal com n vértices e a segunda gera um grafo planar com n vértices e $m \leq 3n - 6$ arestas. (O reverso de um arco e num grafo G é acessado pela função *G.reversal(e)* e o sucessor cíclico de e, no ciclo facial que contém e, é acessado pela função *G.face_cycle_succ(e)*.)

121 ⟨gera 2 121⟩≡

```
void maximal_planar_map(graph& G, int n)
{ G.clear();
  if (n ≤ 0 ) return;
  node a = G.new_node();
  n--;
  if (n ≡ 0) return;
  node b = G.new_node();
  n--;
  edge* E = new edge[6*n];
  E[0] = G.new_edge(a,b);
  E[1] = G.new_edge(b,a);
  G.set_reversal(E[0],E[1]);
  int m = 2;
  while (n--) {
    edge e = E[rand_int(0,m-1)];
    node v = G.new_node();
    while (target(e) ≠ v) {
      edge x = G.new_edge(v,source(e));
      edge y = G.new_edge(e,v,LEDA::after);
      E[m++] = x; E[m++] = y;
      G.set_reversal(x,y);
      e = G.face_cycle_succ(e);
    }
  }
  delete[] E;
}
```

Figura 7.2: Código da função *maximal_planar_map* do LEDA versão `4.2`.

A tabela com os tempos de execução de cada implementação é ilustrada na figura 7.5. Nela, não foram medidos os tempos de teste de planaridade sem o cálculo do certificado para a implementação de BM, onde a coluna T foi substituída pela coluna TJ.

A seguir, para cada tipo de grafo usado nos testes, encontram-se uma descrição de como os grafos foram gerados, uma tabela do consumo de espaço para cada uma das implementações e dois

122 ⟨gera 1 122⟩≡

```
void random_planar_map(graph& G, int n, int m)
{ maximal_planar_map(G,n);
  list<edge> E;
  edge e;
  edge_array<bool> marked(G,false);
  forall_edges(e,G) {
     if (¬marked[e]) E.append(e);
     marked[G.reversal(e)] = true;
   }
  E.permute();
  while (E.length() > m) {
     edge e = E.pop();
     G.del_edge(G.reversal(e));
     G.del_edge(e);
   }
}
```

Figura 7.3: Código da função *random_planar_map* do LEDA versão `4.2`.

gráficos obtidos com os dados da figura 7.5, um para o teste sem cálculo do certificado e outro para o teste com construção do certificado e verificação deste. (Na implementação de SH a verificação de uma descrição combinatória plana é feita pela função *check_embedding* na seção 2.5.)

O espaço consumido foi medido pela função *print_statistics* do LEDA, que mostra um resumo da memória usada como mostrado na figura 7.4.

STD_MEMORY_MGR (memory status)

size	used	free	blocks	bytes
8	0	1022	1	8176
12	0	20430	30	245160
44	0	8325	45	366300
52	0	2028	13	105456
64	0	630	5	40320
time: 0.20 sec			space: 751.63 kb	

Figura 7.4: Saída da função *print_statistics* do LEDA, que mostra um resumo da memória usada.

		BL		HT		BM		SH	
Tipo	n	T	TJC	T	TJC	TJ	TJC	T	TJC
(G1)	1000	0.03	0.06	0.09	0.13	0.02	0.02	0.12	0.77
	2000	0.06	0.13	0.19	0.30	0.05	0.06	0.28	1.65
	4000	0.13	0.29	0.43	0.69	0.12	0.13	0.60	3.38
	8000	0.27	0.60	0.89	1.49	0.24	0.27	1.26	6.88
	16000	0.57	1.23	1.87	3.19	0.50	0.56	2.63	13.99
	32000	1.20	2.59	3.95	6.93	1.08	1.19	5.45	28.70
	64000	2.55	5.52	8.40	15.13	2.22	2.40	11.58	58.07
(G2)	1000	0.02	0.22	0.11	–	0.02	0.03	0.11	0.21
	2000	0.05	0.46	0.23	–	0.06	0.06	0.24	0.47
	4000	0.10	0.94	0.48	–	0.13	0.13	0.43	0.75
	8000	0.20	1.86	0.99	–	0.27	0.29	1.04	2.09
	16000	0.42	3.93	2.03	–	0.57	0.61	1.82	3.40
	32000	0.83	7.81	4.24	–	1.04	1.10	3.94	7.45
	64000	1.79	16.86	8.62	–	2.24	2.39	8.01	15.30
(G3)	1000	0.03	0.24	0.12	–	0.02	0.02	0.10	0.19
	2000	0.05	0.52	0.25	–	0.06	0.06	0.25	0.48
	4000	0.11	1.02	0.52	–	0.13	0.14	0.47	0.89
	8000	0.22	2.03	1.05	–	0.27	0.29	1.05	2.04
	16000	0.45	4.23	2.11	–	0.50	0.55	1.84	3.21
	32000	0.91	8.71	4.38	–	1.06	1.12	3.89	7.15
	64000	1.95	18.19	8.92	–	2.16	2.31	8.29	15.48
(G4)	1000	0.05	0.11	0.18	0.28	0.03	0.03	0.25	1.52
	2000	0.10	0.22	0.37	0.59	0.07	0.08	0.51	3.08
	4000	0.21	0.44	0.73	1.21	0.16	0.17	1.02	5.94
	8000	0.42	0.88	1.46	2.48	0.30	0.34	2.03	12.21
	16000	0.85	1.76	2.95	5.16	0.65	0.68	4.12	23.05
	32000	1.72	3.59	6.19	11.00	1.28	1.41	8.54	48.58
	64000	3.61	7.62	12.48	23.15	2.64	2.99	18.24	94.94
(G5)	1000	0.04	0.33	0.08	–	0.03	0.03	0.21	0.43
	2000	0.07	0.62	0.16	–	0.07	0.08	0.40	0.86
	4000	0.14	1.18	0.33	–	0.17	0.18	0.85	1.88
	8000	0.34	2.78	0.66	–	0.34	0.36	1.77	4.28
	16000	0.57	4.79	1.39	–	0.67	0.72	3.10	6.74
	32000	1.24	10.26	2.78	–	1.35	1.48	6.78	15.49
	64000	2.71	21.69	5.83	–	2.76	2.94	14.59	33.40

Figura 7.5: Na tabela acima, encontram-se os tempos medidos em segundos para cada uma das implementações: BL, HT, BM e SH. Para cada valor de n, onde n é o número de vértices, cada valor é a média entre dez medidas de tempo e, para os tipos (G1), (G2) e (G3), usou-se $m = 2n$ arestas. A coluna T representa o tempo gasto para testar planaridade sem justificar o resultado. Já as colunas TJ e TJC representam os tempos gastos para testar e obter um certificado. A diferença entre estes dois é que TJC inclui o tempo gasto para verificar o certificado.

7.2 Grafos planares aleatórios (G1)

Para gerar grafos aleatórios planares, usou-se a função *random_planar_graph* do LEDA (figura 7.6), que recebe como parâmetros uma variável G do tipo **graph** e inteiros n e m, que são, respectivamente, o número de vértices e o número de arestas do grafo G a ser gerado. Para os testes, usou-se $m = 2n$.

124 ⟨gera grafos aleatórios planares 124⟩≡

```
void random_planar_graph(graph& G, int n, int m)
{
  random_planar_map(G,n,m);
  list<edge> E;
  edge_array<bool> marked(G,false);
  edge e;
  forall_edges(e,G) {
    if (¬marked[e]) E.append(e);
    marked[e] = true;
    marked[G.reversal(e)] = true;
  }
  forall(e,E) G.del_edge(e);
}
```

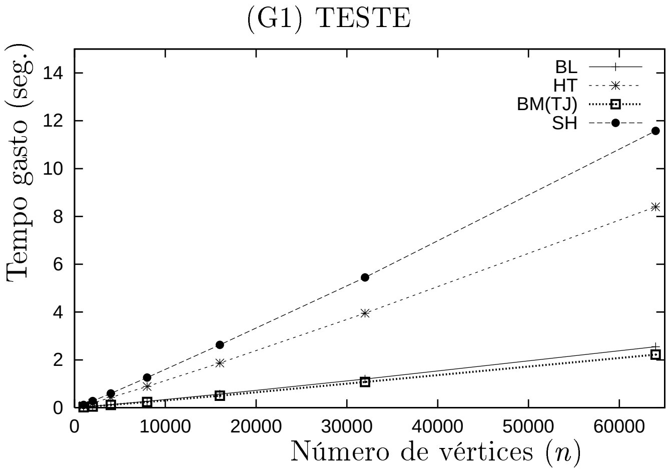
Figura 7.6: Código da função *random_planar_graph* do LEDA.

Na figura 7.7 encontra-se uma tabela com as medidas do espaço consumido por cada implementação para grafos do tipo (G1). Neste caso, observou-se que HT obteve o pior desempenho.

n	BL	HT	BM	SH
1000	751.63	1195.43	671.67	815.60
2000	1487.27	2382.85	1327.35	1615.21
4000	2886.59	4725.71	2646.71	3206.43
8000	5725.20	9443.41	5285.42	6388.88
16000	11362.45	18854.83	10562.84	12721.79
32000	22652.93	37701.65	21117.68	25443.57
64000	45042.00	75339.33	42227.37	50831.17

Figura 7.7: Espaço consumido para grafos planares aleatórios (G1) com n vértices. Para cada valor de n foi medido o espaço gasto por cada algoritmo, em Kbytes.

Os gráficos com os dados da tabela 7.5 para grafos aleatórios planares são ilustrados na figura 7.8. Nos dois gráficos, observa-se que as implementações de BM e de BL obtiveram os melhores desempenhos.



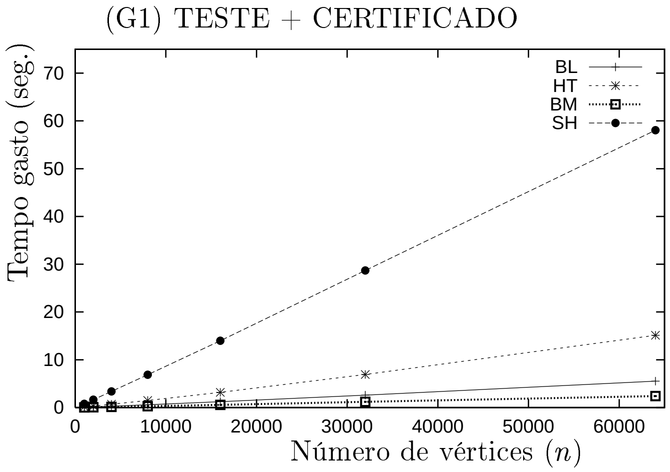


Figura 7.8: Tempos gastos pelas implementações com grafos planares aleatórios (G1). O primeiro gráfico ilustra o tempo para apenas testar planaridade. Neste, apenas BM executa o cálculo do certificado. O segundo ilustra o tempo gasto para testar, justificar e verificar o certificado construído.

7.3 Grafos não-planares do tipo (G2)

Os grafos não-planares do tipo (G2) foram gerados conforme mostrado na figura 7.9. Primeiramente um grafo planar G com n vértices e $m = 2n$ arestas é gerado aleatoriamente. Em seguida, seis vértices de G são escolhidos para fazer parte de um $K_{3,3}$ em G, construído adicionando-se arestas a G.

126 ⟨gera grafos aleatórios planares + K33 126⟩≡

```
m = 2*n;
random_planar_graph(G, n, m);
array<node> V(6);
for (int j = 0; j < 6; j++) V[j] = G.choose_node();
for (int j = 0; j < 3; j++)
  for (int k = 3; k < 6; k++) {
    bool inG = false;
    edge e;
    forall_edges(e, G)
    if ((source(e) ≡ V[j] ∧ target(e) ≡ V[k]) ∨ (target(e) ≡ V[j] ∧ source(e) ≡ V[k]))
    inG = true;
    if (¬inG) G.new_edge(V[j],V[k]);
  }
```

Figura 7.9: Trecho de código (do LEDA) para gerar grafos não-planares do tipo (G2).

Na figura 7.10 encontra-se uma tabela com as medidas do espaço consumido por cada implementação para grafos do tipo (G2). Neste caso, observou-se que HT e BL consumiram mais espaço.

n	BL	HT	BM	SH
1000	1143.44	1171.44	671.67	687.66
2000	2047.00	2310.88	1327.35	1359.34
4000	4078.01	4597.77	2654.70	2678.69
8000	8715.74	9179.54	5293.41	5357.38
16000	16168.10	18295.10	10578.83	10642.80
32000	32200.27	36574.20	21117.68	21589.45
64000	62993.23	73300.33	42227.37	42595.19

Figura 7.10: Espaço consumido para grafos do tipo (G2) com n vértices. Para cada valor de n foi medido o espaço gasto por cada algoritmo, em Kbytes.

Os gráficos com os dados da tabela 7.5 para grafos do tipo (G2) são ilustrados na figura 7.11. No primeiro, apesar de BL ter apresentado o melhor desempenho, juntamente com BM, BL apresentou o pior desempenho no segundo, juntamente com SH. Como a implementação de HT não calcula o certificado para grafos não-planares, esta foi omitida do segundo gráfico.

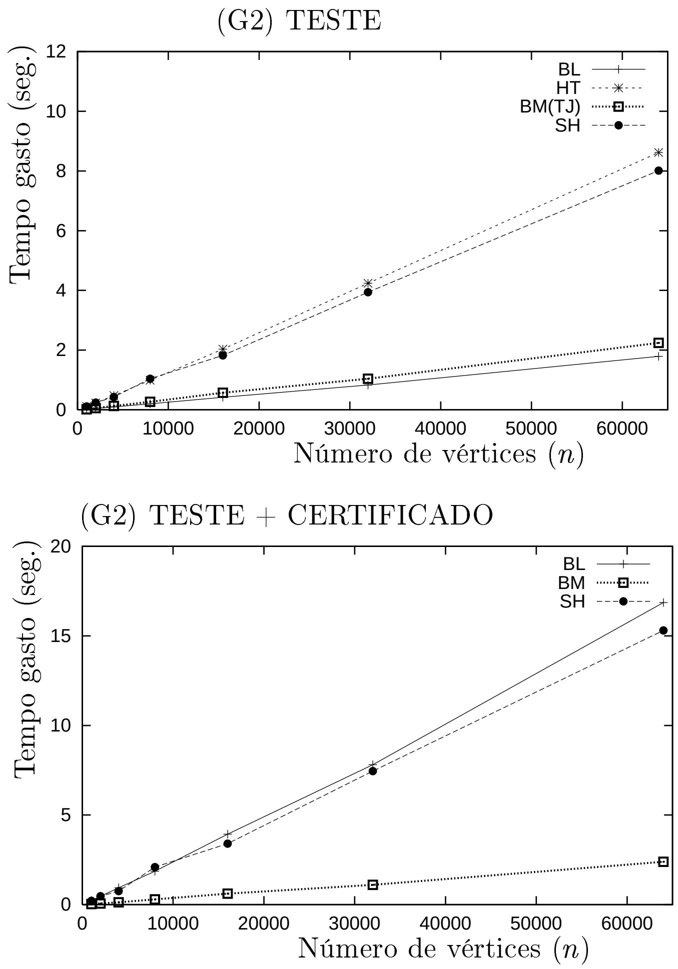


Figura 7.11: Tempos gastos pelas implementações com grafos não-planares do tipo (G2). O primeiro gráfico ilustra o tempo gasto para apenas testar planaridade. Neste, apenas BM executa o cálculo do certificado. O segundo ilustra o tempo gasto para testar, justificar e verificar o certificado construído.

7.4 Grafos não-planares do tipo (G3)

Os grafos não-planares do tipo (G3) foram gerados conforme mostrado na figura 7.12. Primeiramente um grafo planar G com n vértices e $m = 2n$ arestas é gerado aleatoriamente. Em seguida, cinco vértices de G são escolhidos para fazer parte de um K_5 em G, construído adicionando-se arestas a G.

128 ⟨gera grafos aleatórios planares + K5 128⟩≡

```
m = 2*n;
random_planar_graph(G, n, m);
array<node> V(5);
for (int j = 0; j < 5; j++)
  for (int k = j + 1; k < 5; k++) {
    bool inG = false;
    edge e;
    forall_edges(e, G)
      if ((source(e) ≡ V[j] ∧ target(e) ≡ V[k])∨(target(e) ≡ V[j] ∧ source(e) ≡ V[k]))
        inG = true;
      if (¬inG) G.new_edge(V[j],V[k]);
}
```

Figura 7.12: Trecho de código (do LEDA) para gerar grafos não-planares do tipo (G3).

Na figura 7.13 encontra-se uma tabela com as medidas do espaço consumido para grafos do tipo (G3). Neste caso, observou-se novamente que HT e BL consumiram mais espaço.

n	BL	HT	BM	SH
1000	1103.46	1171.44	671.67	687.66
2000	2023.01	2310.88	1327.35	1359.34
4000	3910.09	4597.77	2654.70	2702.68
8000	7972.11	9171.54	5293.41	5397.36
16000	16583.90	18295.10	10578.83	10762.74
32000	33983.40	36574.20	21117.68	21245.62
64000	62961.24	73300.33	42227.37	42715.13

Figura 7.13: Espaço consumido para grafos do tipo (G3) com n vértices. Para cada valor de n foi medido o espaço gasto por cada algoritmo, em Kbytes.

Os gráficos com os dados da tabela 7.5 para grafos do tipo (G3) são ilustrados na figura 7.14. Como observado na seção anterior, BL e BM apresentaram um bom desempenho no primeiro gráfico. Já no segundo, BL e SH apresentaram os piores desempenhos.

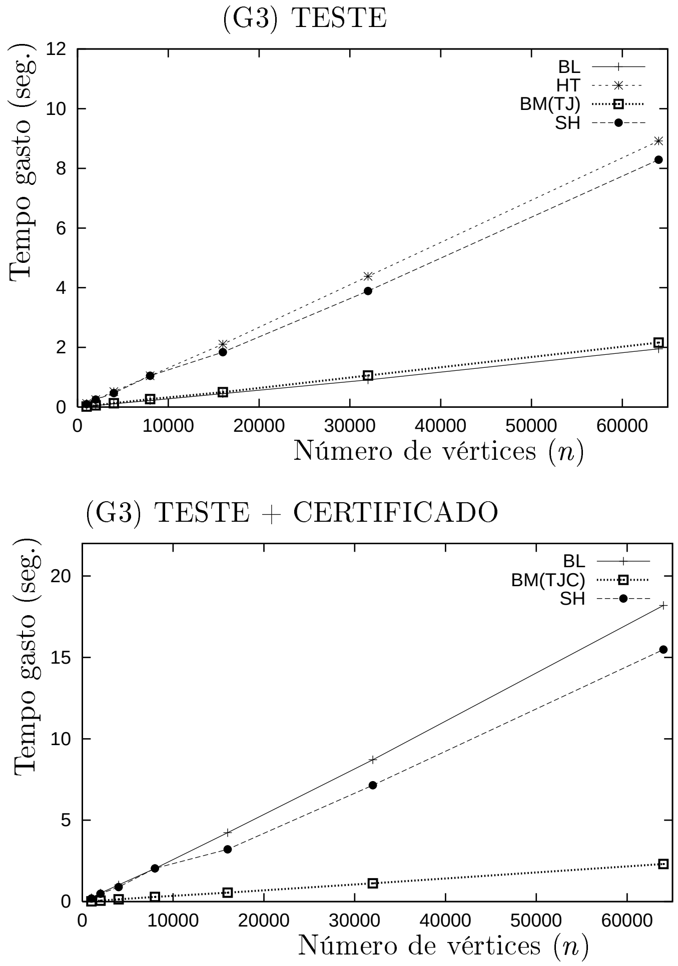


Figura 7.14: Tempos gastos pelas implementações com grafos não-planares do tipo (G3). O primeiro gráfico ilustra o tempo gasto para apenas testar planaridade. Neste, apenas BM executa o cálculo do certificado. O segundo ilustra o tempo gasto para testar, justificar e verificar o certificado construído.

7.5 Grafos planares maximais (G4)

Para gerar grafos maximais planares, usou-se a função *maximal_planar_graph* do LEDA (figura 7.15), que recebe como parâmetros uma variável G do tipo **graph** e um inteiro n representando o número de vértices do grafo a ser gerado.

130 ⟨gera grafos maximais planares 130⟩≡

```
void maximal_planar_graph(graph& G, int n)
{
  maximal_planar_map(G,n);
  list<edge> E;
  edge_array<bool> marked(G,false);
  edge e;
  forall_edges(e,G) {
    if (¬marked[e]) E.append(e);
    marked[e] = true;
    marked[G.reversal(e)] = true;
   }
  forall(e,E) G.del_edge(e);
}
```

Figura 7.15: Código da rotina *maximal_planar_graph* do LEDA.

A tabela da figura 7.16 mostra as medidas do espaço consumido para grafos planares maximais. Neste caso, observou-se que HT foi a implementação que consumiu mais espaço.

n	BL	HT	BM	SH
1000	1023.50	1651.20	943.54	1111.46
2000	1991.03	3286.41	1887.08	2198.93
4000	3950.07	6532.82	3766.16	4365.87
8000	7796.19	13049.65	7516.33	8691.75
16000	15552.40	26067.30	15024.66	17383.51
32000	30768.97	52102.62	30033.33	34743.03
64000	61617.90	104189.23	60058.66	69414.09

Figura 7.16: Espaço consumido para grafos planares maximais (G4) com n vértices.

Os gráficos com os dados da tabela 7.5 para grafos planares maximais são ilustrados na figura 7.17. Nos dois gráficos, BL e BM apresentaram os melhores desempenhos e SH foi o pior.

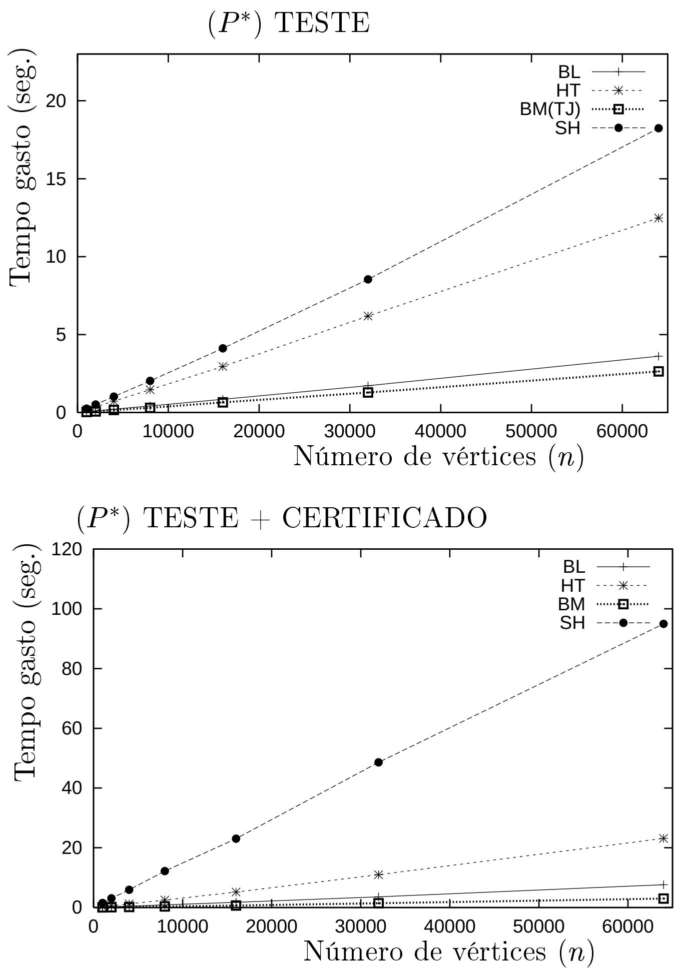


Figura 7.17: Tempos gastos pelas implementações com grafos planares maximais (G4). O primeiro gráfico ilustra o tempo gasto para apenas testar planaridade. Neste, apenas BM executa o cálculo do certificado. O segundo ilustra o tempo gasto para testar, justificar e verificar o certificado construído.

7.6 Grafos não-planares do tipo (G5)

Os grafos não-planares do tipo (G5) usados nos testes foram gerados conforme mostra a figura 7.18. Primeiramente foi gerado um grafo planar maximal G com n vértices e em seguida adicionou-se uma nova aresta a G entre dois vértices não-adjacentes.

```
132 ⟨gera grafos maximais planares mais uma aresta 132⟩≡
    maximal_planar_graph(G,n);
    node v = G.choose_node();
    node w; bool _ok; edge e;
    do {
      _ok = true;
      w = G.choose_node();
      forall_adj_edges(e,v)
        if ( G.target(e) ≡ w ) _ok = false;
    } while ( _ok ≡ false );
    G.new_edge(v,w);
```

Figura 7.18: Trecho de código para gerar grafos não-planares do tipo (G5).

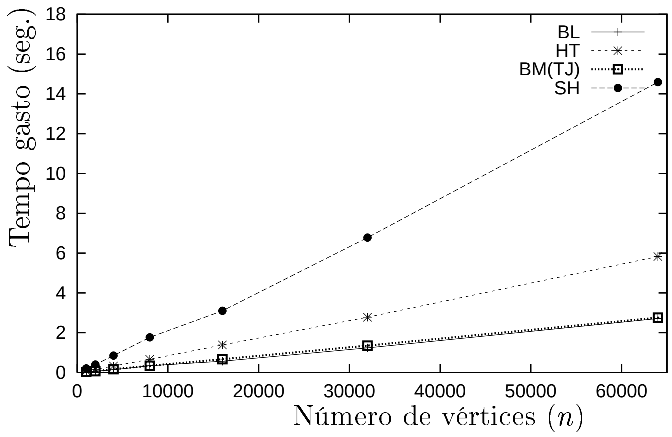
A tabela da figura 7.19 mostra as medidas do espaço consumido para grafos do tipo (G5). Neste caso, observou-se que BL foi a implementação que consumiu mais espaço.

n	BL	HT	BM	SH
1000	1799.12	1267.39	943.54	975.52
2000	2862.60	2518.78	1887.08	1935.05
4000	6109.02	5021.56	3766.16	3846.12
8000	11354.45	10035.12	7516.33	7732.22
16000	22948.79	20054.24	15024.66	15328.51
32000	47856.62	40100.48	30033.33	30832.94
64000	85574.20	80184.96	60058.66	60866.27

Figura 7.19: Espaço consumido para grafos planares maximais (G5).

Os gráficos com os dados da tabela 7.5 para grafos planares maximais são ilustrados na figura 7.17. Nos dois gráficos, SH obteve o pior desempenho. Como foi observado para grafos não-planares dos tipos (G2) e (G3), BL apresentou o melhor desempenho no primeiro, junto com BM. Já no segundo, BL teve um desempenho ruim.

(G5) TESTE



(G5) TESTE + CERTIFICADO

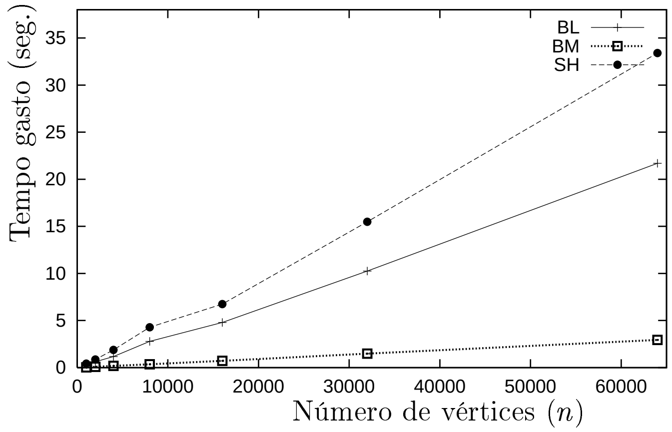


Figura 7.20: Tempos gastos pelas implementações com grafos não-planares do tipo (G5). O primeiro gráfico ilustra o tempo gasto para apenas testar planaridade. Neste, apenas BM executa o cálculo do certificado. O segundo ilustra o tempo gasto para testar, justificar e verificar o certificado construído.

Comentários finais

Nesta dissertação, apresentamos uma descrição do algoritmo de LEC e uma implementação deste, proposta por Shih e Hsu. A implementação de SH descrita neste trabalho foi realizada sobre a plataforma LEDA.

Nosso estudo começou pelo artigo publicado por Shih e Hsu [15]. Nele encontra-se uma descrição para resolver o problema do teste de planaridade, mas com alguns erros, conforme mostra Boyer [4]. Além disso, muitos detalhes são omitidos neste artigo. Como exemplos, podemos citar o uso do contador para cada C-nó para testar a presença de uma XY-obstrução do tipo (o1), a existência de C-nós inativos na PC-árvore alterada T e a determinação dos C-nós ativos em T.

O desenvolvimento da implementação de SH foi iniciada após o estudo do artigo publicado por Thomas [18]. Através deste artigo conseguimos enxergar o algoritmo de SH “de um nível mais alto” e nos ajudou a desvendar muitos dos detalhes omitidos no primeiro artigo.

Nossa primeira versão da implementação que resolvia o problema do teste de planaridade consumia tempo quadrático no número de vértices do grafo e consistia basicamente de dois passos, além do pré-processamento: um passo para calcular as componentes de $T - L$ e seus respectivos terminais e um outro passo para estender a PC-árvore alterada T para a próxima iteração, caso não existisse uma XY-obstrução. Observe que nessa versão, a busca por uma XY-obstrução e a atualização de T eram feitas em um passo apenas. Apesar de usarmos duas listas duplamente ligadas para representar cada RBC (cada uma num sentido para evitar o custo para inverter a lista), esta versão era quadrática pois C-nós inativos eram percorridos durante os dois passos básicos executados nesta versão.

Numa segunda versão da implementação, aplicamos a heurística *union-find* [16] para compressão de caminhos formados por C-nós inativos consecutivos. A idéia era comprimir todos os caminhos percorridos em T durante a fase do cálculo dos terminais, garantindo assim que, na fase seguinte, cada caminho percorrido em T tivesse no máximo dois C-nós consecutivos. Aparentemente, essa versão ficou bem mais rápida que a anterior e partimos para o desenvolvimento de uma nova versão que resolvesse o problema da planaridade. O primeiro passo nesse sentido foi deduzir como construir uma descrição combinatória plana de um grafo planar, já que tanto o artigo de Shih e Hsu quanto o artigo de Thomas omitiam essa parte. (Essa foi a parte mais complicada no desenvolvimento da implementação de SH.)

Após conseguirmos uma versão que resolvesse o problema da planaridade, partimos para os experimentos, nos baseando no programa de demonstração presente no LEDA, chamado *planarity_time*. Adaptamos este programa de forma a incorporar a nossa implementação de SH e, nas versões `4.2` e `4.3` do LEDA, notamos que as implementações de BL e de HT estavam quadráticas. (Aparentemente, na versão `4.4` do LEDA esses problemas foram solucionados.) Nos primeiros testes realizados com *planarity_time*, notamos que a nossa implementação de SH também estava quadrática, mas com o uso do programa `gprof`, conseguimos ajustar nosso programa de modo que consumisse tempo linear.

Durante essa fase de testes, tivemos a oportunidade de olhar rapidamente o artigo de Boyer e Myrvold [3] e entramos em contato com o John M. Boyer, que gentilmente nos cedeu seu código fonte em *C*, que foi adiciondo ao programa *planarity_time*, além de nos fornecer informações valiosas para corrigir e melhorar nossa implementação de SH. Com estas informações, eliminamos as duplicações das listas RBC e DCP associadas a C-nós ativos, através do uso do tipo *bmlist* (proposto por Boyer e Myrvold) e corrigimos a maneira de como era feito o percurso na PC-árvore alterada T (conforme descrito na seção 4.5) de modo que C-nós inativos só sejam percorridos no caso do cálculo da lista de arcos de uma XY-obstrução.

A versão final da implementação de SH, que resolve o problema da planaridade, ficou com quatro passos básicos: cálculo dos terminais, busca por uma XY-obstrução, cálculo da descrição combinatória plana para cada novo C-nó e atualização da PC-árvore alterada T para a próxima iteração. Note que um dos passos da primeira versão foi dividido em: um passo para buscar uma XY-obstrução e outro para atualizar T para a próxima iteração, caso nenhuma XY-obstrução seja detectada no passo anterior. Isso foi feito pois tanto o cálculo da lista de arcos de uma XY-obstrução quanto o cálculo de uma descrição combinatória de um novo C-nó utilizam a árvore T corrente, além dos valores de b correntes.

Apesar da versão atual consumir tempo linear no número de vértices do grafo, conforme demonstrado neste trabalho, os resultados obtidos pelos testes realizados no capítulo 7 mostram que nossa implementação está muito lenta (principalmente quando submetemos grafos planares ao algoritmo), quando comparada às implementações presentes no LEDA. Na realidade, esta versão é um protótipo onde muito pode ser melhorado, principalmente na parte em que ocorre o cálculo de uma descrição combinatória plana para um novo C-nó e uma possível proposta de trabalho seria reescrever este programa de forma que fique “mais limpo” e “comparável” às implementações já existentes.

Capítulo 9

Índice

9.1 Blocos do código

Referências Bibliográficas

[1] L. Auslander e S.V. Parter, On imbeddings graphs in the plane, *J. Math. and Mech.* **10** (1961), no. 3, 517–523. Citado na(s) página(s) [ii, 2, 4]

[2] K.S. Booth e G.S. Lueker, Testing the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *Journal of Computer and System Science* **13** (1976), 335–379. Citado na(s) página(s) [3, 4, 25]

[3] J. Boyer e W. Myrvold, Stop minding your P's and Q's: A simplified $O(n)$ planar embedding algorithm, *Procedding of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland), ACM Special Interest Group on Algorithms and Computation Theory and SIAM Activity Group on Discrete Mathematics, ACM Press, January 1999, pp. 140–146. Citado na(s) página(s) [ii, 3, 4, 13, 25, 136]

[4] John M. Boyer, *Corrections to the PC-tree planarity algorithm*, `http://pacificcoast.net/~lightning/jboyer.htm`, January 2003. Citado na(s) página(s) [135]

[5] E.R. Canfield e S.G. Williamson, The two basic linear time planarity algorithms: Are they the same?, *Linear and Multilinear Algebra* **26** (1990), 243–265. Citado na(s) página(s) [3]

[6] Reinhard Diestel, *Graph theory*, Graduate Texts in Mathematics, Springer, 1991. Citado na(s) página(s) [45]

[7] K. Mehlhorn e St. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999. Citado na(s) página(s) [3, 11]

[8] Shimon Even, *Graph algorithms*, Computer Science Press, 1979. Citado na(s) página(s) [47, 48]

[9] A.J. Goldstein, An efficient and constructive algorithm for testing whether a graph can be embedded in a plane, *Graph and Combinatorics Conference*, Dept. Math., Princeton University, 1963, pp. 16–18. Citado na(s) página(s) [ii, 2, 4]

[10] J. Hopcroft e R. Tarjan, Efficient planarity testing, *Journal of the Association for Computing Machinery* **21** (1974), no. 4, 549–568. Citado na(s) página(s) [ii, 3, 4]

[11] C. Kuratowski, Sur le probleme des corbes gauches en topologie, *Fundamenta Mathematicae* **15** (1930), 271–283. Citado na(s) página(s) [2]

[12] A. Lempel, S. Even e I. Cederbaum, An algorithm for planarity testing of graphs, *Proceddings International Symposium on Theory of Graphs* (New York) (P. Rosenstiehl, ed.), Gordon and Breach, July 1967, pp. 215–232. Citado na(s) página(s) [ii, 2, 3, 4, 25, 33]

[13] K. Mehlhorn e P. Mutzel, On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm, *Algorithmica* **16** (1996), no. 2, 233–242. Citado na(s) página(s) [3]

[14] P. Mutzel, *A fast $O(n)$ embedding algorithm based on the Hopcroft-Tarjan planarity test*, Tech. Report 92.107, Universität zu Köln, 1992, `http://www.mpi-sb.mpg.de/~mutzel/mpireports/publications.html`. Citado na(s) página(s) [3]

[15] W.-K. Shih e W.-L. Hsu, A new planarity test, *Theoretical Computer Science* **223** (1999), 179–191. Citado na(s) página(s) [ii, 3, 4, 25, 26, 48, 49, 135]

[16] Cormen & Leiserson & Rivest & Stein, *Introduction to algorithms*, The MIT Press, 2001. Citado na(s) página(s) [50, 135]

[17] R.E. Tarjan, *Implementation of an efficient algorithm for planarity testing of graphs*, Dec. 1969. Citado na(s) página(s) [3, 25]

[18] R. Thomas, *Planarity in linear time*, `http://www.math.gatech.edu/~thomas/`, June 1997. Citado na(s) página(s) [25, 26, 135]

Índice Remissivo