MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE DEFESA

GERALDO AVELINO DE OLIVEIRA NETO

PROBLEMAS EM GRAFOS DINÂMICOS: OUTERPLANARIDADE E APLICAÇÕES

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA

GERALDO AVELINO DE OLIVEIRA NETO

PROBLEMAS EM GRAFOS DINÂMICOS: OUTERPLANARIDADE E APLICAÇÕES

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Defesa do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências e Engenharia de Defesa.

Orientadora: Prof^{a}. Claudia Marcela Justel - D.Sc.

Rio de Janeiro
2017

c2017

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

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

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

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

004.69 Neto, Geraldo Avelino de Oliveira
S586e Problemas em Grafos Dinâmicos: Outerplanaridade e Aplicações / Geraldo Avelino de Oliveira Neto, orientado por Claudia Marcela Justel - Rio de Janeiro: Instituto Militar de Engenharia, 2017.

141p.: il.

Dissertação (mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2017.

1. Curso de Engenharia de Defesa - teses e dissertações. 1. Palavra 01. 2. Palavra 02. 3. Palavra 03. I. Justel, Claudia Marcela. II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

GERALDO AVELINO DE OLIVEIRA NETO

PROBLEMAS EM GRAFOS DINÂMICOS: OUTERPLANARIDADE E APLICAÇÕES

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Defesa do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Mestre em Ciências e Engenharia de Defesa.

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

Aprovada em 10 de Agosto de 2017 pela seguinte Banca Examinadora:

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

Prof. Paulo Fernando Ferreira Rosa - Ph.D. do IME

Prof^a. Lilian Markenzon - D.Sc. do NCE/UFRJ

Rio de Janeiro
2017

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

AGRADECIMENTOS

Primeiramente, agradeço a Deus, por nortear o meu caminho, me ajudar nos momentos complicados e me dar força nos momentos de dificuldade.

Aos meus familiares e a minha namorada pela paciência e pelo incentivo.

À minha orientadora Claudia Justel pela disponibilidade e ajuda na realização do trabalho.

Ao pesquisador Renato Paes Leme pela importante colaboração neste trabalho. A comunicação mantida durante o ano 2015 permitiu conhecer vários assuntos na área de algoritmos e auxiliou no entendimento dos trabalhos mais recentes sobre conectividade em grafos dinâmicos.

SUMÁRIO

LISTA DE ILUSTRAÇÕES 8

LISTA DE TABELAS 10

1 INTRODUÇÃO 13

1.1 Objetivo 19

1.2 Justificativa 20

1.3 Contribuição 21

1.4 Organização dos Capítulos 21

2 CONCEITOS BÁSICOS 22

3 ALGORITMOS EM GRAFOS DINÂMICOS PARA MANUTENÇÃO DE CONECTIVIDADE E PLANARIDADE 29

3.1 Algoritmos de conectividade 29

3.1.1 Algoritmo exato para manutenção dinâmica de conectividade- Holm, Lichtenberg e Thorup 29

3.1.2 Algoritmo incremental probabilístico para manutenção dinâmica de conectividade- Kapron, King e Mountjoy 31

3.1.2.1 A Estrutura de Dados Cutset 31

3.1.2.2 Algoritmo Geral 32

3.2 Algoritmos de planaridade 34

3.2.1 Algoritmo para manutenção dinâmica de planaridade - Italiano, La Poutré e Rauch 34

3.2.2 Algoritmo para imersão planar de grafos dinâmicos - Holm e Rotenberg 35

4 MANUTENÇÃO DE OUTERPLANARIDADE EM GRAFOS DINÂMICOS 37

4.1 Conceitos e propriedades dos grafos outerplanares 37

4.2 Inserção de aresta 40

4.2.1 Adição de ponte 42

4.2.2 Adição de corda 43

4.2.3 Adição de aresta hamiltoniana 44

4.3 Remoção de aresta 46

4.3.1 Remoção de ponte ... 47
4.3.2 Remoção de corda ... 47
4.3.3 Remoção de aresta hamiltoniana ... 48

5 IMPLEMENTAÇÃO DO ALGORITMO ... 51
5.1 Estruturas de Dados Utilizadas ... 51
5.2 Inicialização das Estruturas de Dados ... 57
5.3 Inserção de Aresta ... 58
5.3.1 Adição de ponte ... 58
5.3.2 Adição de corda ... 62
5.3.3 Adição de aresta hamiltoniana ... 65
5.3.4 Pseudocódigo para inserção de aresta ... 74
5.4 Remoção de Aresta ... 75
5.4.1 Remoção de Ponte ... 76
5.4.2 Remoção de Corda ... 79
5.4.3 Remoção de Aresta Hamiltoniana ... 81
5.4.4 Pseudocódigo para remoção de aresta ... 90

6 ANÁLISE DA COMPLEXIDADE ... 91
6.1 Inserção de aresta ... 92
6.1.1 Adição de ponte ... 92
6.1.2 Adição de corda ... 93
6.1.3 Adição de aresta hamiltoniana ... 94
6.2 Remoção de aresta ... 95
6.2.1 Remoção de ponte ... 95
6.2.2 Remoção de corda ... 96
6.2.3 Remoção de aresta hamiltoniana ... 97

7 CONSIDERAÇÕES FINAIS ... 98
7.1 Conclusão ... 98
7.2 Trabalhos futuros ... 99

8 REFERÊNCIAS BIBLIOGRÁFICAS ... 100

9 ANEXOS ... 104
9.1 ANEXO 1: Implementação do Algoritmo ... 105

LISTA DE ILUSTRAÇÕES

FIG.1.1 Classes de grafos. 16

FIG.4.1 Grafo *outerplanar*. 38
FIG.4.2 Grafo *outerplanar* maximal. 38
FIG.4.3 Grafo bloco conexão $BC(G)$ associado ao grafo G. 39
FIG.4.4 Inserção de ponte no grafo G. 40
FIG.4.5 Inserção de corda no grafo G. 41
FIG.4.6 Inserção de aresta hamiltoniana no grafo G. 41
FIG.4.7 Inserção de corda no grafo G que invalida a *outerplanaridade*. 41
FIG.4.8 Inserção de aresta hamiltoniana em G que invalida a *outerplanaridade*. 41
FIG.4.9 Inserção de ponte entre vértices ligados a conexões. 42
FIG.4.10 Inserção de ponte entre vértices ligados a blocos. 42
FIG.4.11 Inserção de ponte entre um vértice ligado a conexão e outro ligado a bloco. 43
FIG.4.12 Inserção de corda em G. 44
FIG.4.13 Inserção de aresta hamiltoniana entre vértices da mesma conexão. 45
FIG.4.14 Aresta de p_{uv} fronteira entre faces internas. 45
FIG.4.15 Arestas consecutivas de p_{uv} pertencentes ao mesmo bloco. 46
FIG.4.16 Inserção de aresta hamiltoniana. 46
FIG.4.17 Remoção de ponte em G. 47
FIG.4.18 Remoção de corda em G. 48
FIG.4.19 Remoção de aresta hamiltoniana com sequência de arestas hamiltonianas em p_{uv}. 49
FIG.4.20 Remoção de aresta hamiltoniana com corda em p_{uv}. 49
FIG.4.21 Remoção de aresta hamiltoniana com cordas consecutivas em p_{uv}. 50

FIG.5.1 Adição de ponte no caso em que dois blocos são conectados. 60
FIG.5.2 Adição de ponte no caso em que um bloco e uma conexão são conectados. 61
FIG.5.3 Adição de ponte no caso em que duas conexões são conectadas. 62
FIG.5.4 Adição inválida de uma corda. 64
FIG.5.5 Adição válida de uma corda. 65
FIG.5.6 Adição de aresta hamiltoniana em uma conexão. 69

FIG.5.7 Adição de aresta hamiltoniana inválida em que há aresta fronteira entre faces internas em p_{uv}. 70
FIG.5.8 Adição de aresta hamiltoniana inválida em que há arestas consecutivas em p_{uv} pertencentes ao mesmo bloco. 71
FIG.5.9 Adição de aresta hamiltoniana com formação de articulação. 73
FIG.5.10 Adição de aresta hamiltoniana ligando vértices distintos em $BC(G)$. . . . 74
FIG.5.11 Remoção de ponte em G. 78
FIG.5.12 Remoção de ponte em G. 79
FIG.5.13 Remoção de corda em G. 81
FIG.5.14 Remoção de aresta hamiltoniana em G. 84
FIG.5.15 Remoção de aresta hamiltoniana em G. 86
FIG.5.16 Remoção de aresta hamiltoniana em G. 87
FIG.5.17 Remoção de aresta hamiltoniana em G. 89

LISTA DE TABELAS

TAB.1.1 Problemas restritos a classe de grafos *outerplanares* e complexidades 17
TAB.1.2 Limite para área no desenho de diferentes classes de grafos 18
TAB.1.3 Menor número de páginas para imersão em *book* em algumas classes de grafos 19

TAB.6.1 Estruturas de dados utilizadas e respectivos tamanhos. 92

RESUMO

Algoritmos em grafos desempenham um importante papel na área de computação. Muitos problemas aplicados ao mundo real podem ser modelados como um grafo e resolvidos utilizando-se algoritmos que exploram propriedades pertencentes aos mesmos. Nesse trabalho, o interesse principal é enfatizar os algoritmos aplicados a grafos dinâmicos, ou seja, grafos que se modificam conforme o passar do tempo (nesses tipos de algoritmos, a solução para o problema evita com que seja necessário efetuar um novo cálculo de todas as informações acerca do grafo após determinada operação de atualização no mesmo). Mais especificamente, estamos interessados em algoritmos para realizar a manutenção da *outerplanaridade* (um grafo é *outerplanar* se é planar e possui uma imersão no plano, sem cruzamento de arestas, de maneira tal que seus vértices pertencem à face exterior) em um grafo dinâmico.

Nesta dissertação implementaremos um algoritmo completamente dinâmico para resolver o problema da manutenção da *outerplanaridade* em grafos dinâmicos.

ABSTRACT

Graph algorithms play an important role in computer science area. Many problems applied to the real world can be modeled as a graph and solved using algorithms that explore the graph properties. In this work, the main interest is to emphasize the algorithms applied to dynamic graphs, that is, graphs that change over time (in these types of algorithms, a solution to the problem avoids the need to perform a new calculation of all information about the graph after an update on it). More specifically, we are interested in algorithms to perform the maintenance of the *outerplanarity* (a graph is *outerplanar* if it is planar and has an embedding, without crossing of edges, in such a way that its vertices belong to the external face) in a dynamic graph.

In this dissertation, the implementation of an algorithm for maintaining outerplanarity in dynamic graphs is presented.

1 INTRODUÇÃO

Grafos são uma poderosa ferramenta para realização da modelagem computacional de problemas que possuem diversas aplicações reais. Os algoritmos em grafos tentam resolver eficientemente problemas explorando as propriedades dos mesmos. Em algumas ocasiões, é comum que o grafo seja modificado ao longo do tempo, situação na qual temos um grafo dinâmico. Nesse trabalho, estamos interessados mais especificamente em algoritmos que resolvam o problema de manutenção da *outerplanaridade* (informalmente, um grafo é dito *outerplanar* se admite um desenho no plano sem cruzamento de arestas e tal que todo vértice está na fronteira da face externa) em um grafo dinâmico. Mais precisamente, estruturas de dados que processem as seguintes operações sem a necessidade de recalcular todas as informações necessárias para responder as questões de *outerplanaridade* sempre que o grafo mudar: (1) inserir uma aresta e no grafo G, (2) remover uma aresta e de G, e (3) testar se após uma inserção ou remoção o grafo G se mantém *outerplanar*. Vale ressaltar que qualquer operação pode ser realizada em qualquer instante e não tem-se nenhuma informação sobre operações futuras. Para o caso em que as operações de atualização do grafo sejam somente de inserção de arestas, denominamos o problema como incremental. No caso em que as atualizações são apenas de remoção das arestas, o problema é denominado decremental. No caso geral, tem-se um problema completamente dinâmico.

Os problemas relacionados a grafos dinâmicos vêm sendo ativamente estudados desde o início da década de 1980. Muitos resultados importantes foram obtidos com relação a algoritmos completamente dinâmicos, sejam eles determinísticos ou simplesmente randomizados. Vale ressaltar também a grande quantidade de resultados obtidos para classes de grafo restritas e para algoritmos limitados unicamente para a inserção ou para a remoção de arestas. Para o problema de conectividade dinâmica, em 1983, Frederickson (FREDERICKSON (1985)) introduziu o conceito de uma estrutura de dados conhecida como árvore topológica, para a solução do problema completamente dinâmico de menor floresta geradora com um custo do pior caso de $O(\sqrt{m})$ por atualização e resposta para a consulta sobre conectividade em tempo $O(\log n/\log(\sqrt{m}/\log n)) = O(1)$. Em 1992, Eppstein et al. (EPPSTEIN (1997)) melhorou o tempo de atualização para $O(\sqrt{n})$ utilizando uma ferramenta conhecida como técnica de esparsificação. Em 1997, Henzinger e King (HENZINGER (1997a)) propuseram um algoritmo com tempo de atualização

amortizado $O(\sqrt[3]{n}\log n))$ e custo $O(1)$ para consulta para o problema de conectividade.

Em 1995, Henzinger e King (HENZINGER (1999)) utilizaram técnicas de randomização para desenvolver o primeiro algoritmo que realiza operações de atualização em tempo esperado polilogarítmico, $O(\log^3 n)$ para o problema de conectividade dinâmica. Para as operações de consulta, o algoritmo executa em tempo $O(\log n/\log(\log n))$. Em 1996, Henzinger e Thorup (HENZINGER (1997b)) melhoraram o custo de atualização no algoritmo para $O(\log^2 n)$. Em trabalhos separados, Henzinger e Fredman (FREDMAN (1998)) e Miltersen et al. (MILTERSEN (1994)) demonstraram o limite inferior $\Omega(\log n/\log(\log n))$ no tempo amortizado por operação para o problema de conectividade dinâmica utilizando algoritmos aleatórios. Em 1998, Holm, de Lichtenberg e Thorup (HOLM (2001)) propuseram uma estrutura de dados mais simples e determinística baseando-se na decomposição do grafo atual, com operações de atualização em $O(\log^2 n)$ e operações de consulta em $O(\log n/\log(\log n))$. Em 2000, Thorup (THORUP (2000)) sugeriu uma estrutura de dados randomizada com tempo de atualização amortizado $O(\log n(\log(\log n))^3)$ e com consultas de conectividade em tempo $O(\log n/\log(\log n))$. Note que a estrutura proposta por Thorup (THORUP (2000)) possui complexidade computacional próxima aos limites provados por Henzinger, Fredman (FREDMAN (1998)) e Miltersen (MILTERSEN (1994)). Vale ressaltar que todos esses algoritmos que operam em tempo amortizado polilogarítmico possuem instâncias em que a atualização pode custar $\Theta(n)$.

Kapron, King e Mountjoy (KAPRON (2013)) elaboraram um algoritmo randomizado que, apesar de possuir maior complexidade computacional, consegue evitar o custo de pior caso $\Theta(n)$. O método proposto possui tempo de execução no pior caso $O(\log^4 n)$, para inserção de aresta no grafo, $O(\log^5 n)$ para remoção de aresta e $O(\log n/\log(\log n))$ para uma consulta de conectividade.

Wullf-Nilsen (WULFF-NILSEN (2013)) obteve um resultado determinístico no problema de conectividade dinâmica cujo custo de atualização é $O(\log^2 n/\log(\log n))$ e as consultas de conectividade executam com complexidade $O(\log n/\log(\log n))$.

Considerando o problema de conectividade dinâmica em classes restritas de grafos, alguns resultados também são interessantes: para árvores, é possível obter um custo $O(\log n)$ tanto para as atualizações quanto para consultas, utilizando estruturas de dados como árvores Euler-Tour (HENZINGER (1999)) ou árvores Link-Cut (SLEATOR (1983), TARJAN (1984)). Para grafos planares, Eppstein et al. (EPPSTEIN (1997)) provou que é possível obter $O(\log^2 n)$ para remoção de arestas e $O(\log n)$ para a inserção de arestas e também para consulta com relação à conectividade entre vértices. Apesar de conseguir resolver o problema com um custo computacional relativamente baixo, o algoritmo

proposto por Eppstein (EPPSTEIN (1997)) possui um fator constante alto, o que acaba elevando o tempo de execução do algoritmo em si.

Avanços também foram conseguidos com relação ao problema de conectividade dinâmica sujeito a restrições: para o caso em que apenas adiciona-se arestas ao grafo (incremental) e quando apenas retiram-se arestas (decremental). Para o caso incremental, o melhor resultado já obtido utiliza a estrutura de dados union-find e a complexidade de solução do problema tanto para atualizações como para consultas é $\Theta(\alpha(n, m))$ (TARJAN (1975)), onde $\alpha(n, m)$ é a função inversa de Ackermann. Para o caso decremental, Thorup (THORUP (1999)) demonstrou um limite $O(\log n)$ começando com $\Omega(n \log^6 n)$ arestas e um limite $O(1)$ começando com $\Omega(n^2)$ arestas.

Patrascu e Demaine (PATRASCU (2005)) provaram dois limites inferiores para o problema geral de conectividade dinâmica: para $x > 1$, caso um algoritmo possua tempo de atualização $O(x \log n)$, o tempo para consulta será ao menos $\Omega(\log n / \log x)$. Do mesmo modo, caso um algoritmo possua tempo de consulta $O(x \log n)$, o tempo para execução de cada atualização será ao menos $\Omega(\log n \log x)$.

Alguns estudos relevantes também foram realizados na área de planaridade. Hopcroft e Tarjan (HOPCROFT (1974)) foram os primeiros a desenvolver um algoritmo para o teste de planaridade em grafos estáticos em tempo linear. Para algoritmos de planaridade dinâmica, Galil et al. (ITALIANO (1999)) desenvolveram uma estrutura de dados para o teste completamente dinâmico de manutenção da planaridade cujo custo de atualização do grafo no pior caso é $O(n^{2/3})$. Para a manutenção de imersões de grafos planares, Italiano, La Poutré e Rauch (ITALIANO (1993)) propuseram uma solução completamente dinâmica e determinística, cuja estrutura de dados para solução do problema suporta atualizações e consultas na imersão de um grafo planar em $O(\log^2 n)$ no pior caso. Em 2015, Holm e Rotenberg (HOLM (2015)) propuseram uma solução que suporta atualizações e consultas no grafo em tempo $O(\log^2 n)$ no pior caso. Além disso, o algoritmo possui uma operação de girar componentes na imersão (também com complexidade $O(\log^2 n)$ no pior caso), o que pode permitir que a inserção de determinada aresta seja possível, dada a imersão planar.

Para o caso incremental, o teste de planaridade foi resolvido por La Poutré (LA POUTRE (1994)) melhorou os trabalhos de Di Battista e Tamassia (DI BATTISTA (1989), DI BATTISTA (1990)) e Westbrook (WESTBROOK (1992)), para obter um tempo total de execução $O(\alpha(q, n))$, onde q é o número de operações.

Em trabalhos separados, Henzinger e Fredman (FREDMAN (1998)) e Miltersen et al. (MILTERSEN (1994)) demonstraram o limite inferior $\Omega(\log n / \log(\log n))$ para o teste

de planaridade dinâmica.

Diversos trabalhos importantes também foram divulgados para os grafos *outerplanares*, porém a *outerplanaridade* dinâmica não é uma área com bastante exploração. Essa classe de grafos, também denominada "periplanar", foi caracterizada por subgrafos proibidos por Chartrand e Harary (CHARTRAND (1967)) em 1969. De maneira independente, Harary et al. (FLEISCHNER (1974)) e Syslo (SYSLO (1976)) mostraram que um grafo é *outerplanar* se e somente se seu grafo dual geométrico, eliminado o vértice correspondente à face externa, é uma árvore. Em 1979, Syslo (SYSLO (1979)) mostrou que um grafo *outerplanar* maximal contém um único ciclo hamiltoniano. Também em 1979, Mitchell (MITCHELL (1979)) apresentou algoritmos lineares para reconhecimento de grafos *outerplanares* e *outerplanares* maximais.

Os grafos *outerplanares* estão relacionados com diversas classes de grafos (DERIDDER (2017)). Algumas dessas classes e suas relações de inclusão estrita são ilustradas na Figura 1.1

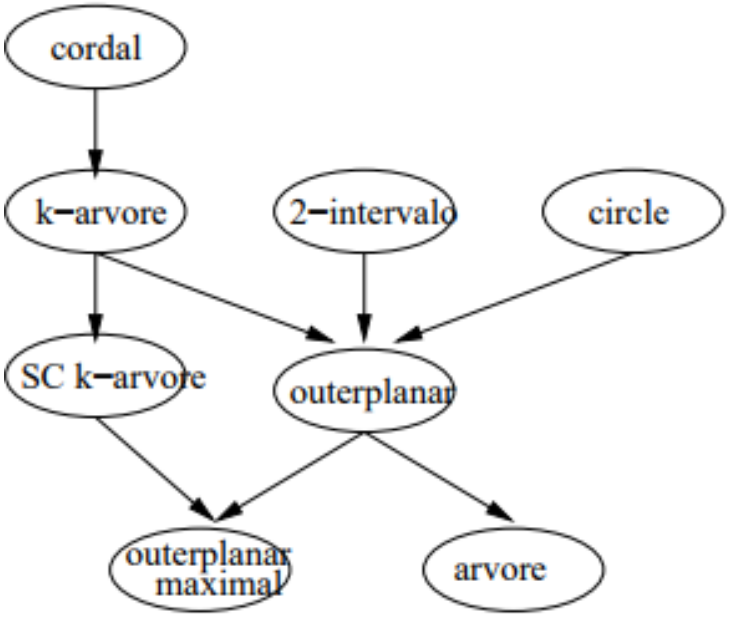


FIG. 1.1: Classes de grafos.

Existem vários problemas NP-completos para grafos em geral que quando restritos a grafos *outerplanares* admitem algoritmo polinomial (ver Tabela 1.1, com informações extraídas de (DERIDDER (2017))).

Uma linha de pesquisa importante relacionada com aplicações em diversas áreas é a de desenho de grafos no plano com limitação de área. Em Valiant (VALIANT (1981)) é mostrado que um grafo qualquer $G = (V, E)$ com $|V| = n$ e $\Delta(G) = 3, 4$, pode ser

Problem	Complexity
booleanwidth decomposition	Polynomial
cliquewidth decomposition	Polynomial
treewidth decomposition	Linear
3-Colourability	Linear
Clique	Linear
Clique cover	Polynomial
Colourability	Linear
Domination	Linear
Feedback vertex set	Polynomial
Graph isomorphism	Linear
Hamiltonian cycle	Linear
Hamiltonian path	Polynomial
Independent dominating set	Polynomial
Independent set	Linear
Maximum cut	Linear
Recognition	Linear
Weighted clique	Linear
Weighted feedback vertex set	Polynomial
Weighted independent dominating set	Polynomial
Weighted independent set	Linear
Weighted maximum cut	Polynomial

TAB. 1.1: Problemas restritos a classe de grafos *outerplanares* e complexidades

desenhado num plano (num *grid*) com área limitada por uma função $O(n^2)$. Já para árvores também com $|V| = n$ e $\Delta(T) = r$, a área é limitada por $O(n)$. No caso dos grafos planares com $|V| = n$ e $\Delta(G) = 4$, o limite da área é dado por uma função $O(n(log_2(n))^2)$. Os grafos *outerplanares* também foram considerados nesta linha de pesquisa. Em 2007, Garg e Rusu (GARG (2007)) mostraram que é possível desenhar um grafo *outerplanar* com limite de área $O(\Delta n^{1,48})$. Em 2009, di Battista e Fratti (DI BATTISTA (2009)) diminuíram a limitação da área para $O(n^{1,48})$. A Tabela 1.2 traz uma lista das limitações de área para os diferentes tipos de grafos estudados.

Ainda na área de desenho de grafos, Frati (FRATI (2012)) desenvolveu um algoritmo para desenho com linhas retas de grafos *outerplanares* com n vértices e com grau máximo Δ com área $O(\Delta n \log n)$.

G, $\|V\| = n$	$\Delta(G)$	Área
geral	$3, 4$	$O(n^2)$
árvore	r	$O(n)$
planar	4	$O(n(log_2(n))^2)$
outerplanar	Δ	$O(\Delta n^{1,48})$
outerplanar	Δ	$O(n^{1,48})$

TAB. 1.2: Limite para área no desenho de diferentes classes de grafos

Além do desenho de grafo com área limitada que apresenta resultados interessantes para grafos *outerplanares*, um outro problema da Teoria de Grafos relacionado com os grafos *outerplanares* é o seguinte: dado um grafo não direcionado $G = (V, E)$, fazer uma imersão do grafo num *book*, com a espinha contendo todos os vértices e as arestas formando as páginas de maneira tal que as arestas não se cruzem dentro da página (CHUNG (1987)). Uma versão desse problema considera procurar pelo menor número de páginas na imersão, de modo que a largura de cada página seja mínima (largura = número máximo de arestas dentro da página que intersectam uma linha perpendicular à espinha). E uma terceira formulação consiste em realizar uma imersão no plano de maneira tal que os vértices do grafo pertençam a um ciclo e que as cordas (arestas fora do ciclo) tenham níveis atribuídos tais que cordas no mesmo nível não tem cruzamento. Da última versão do problema foi provado que um grafo pode ser imerso num *book* com no máximo uma página se e somente se for *outerplanar*.

Dados os problemas a seguir:

Problema 1: "minimizar o número de páginas necessárias para fazer a imersão de um grafo num *book*, quando a ordenação dos vértices na espinha é conhecida".

é equivalente ao problema:
Problema 2: "determinar o menor número de cores necessários para colorir um grafo circular (grafo circular = interseção de cordas num ciclo)".
Como o Problema 2 é NP-Completo, o Problema 1 também é NP-Completo.

Segundo Bernhar e Kainen (BERNHAR (1979)), o menor número de páginas do *book* no qual pode ser imerso um grafo, $bt(G)$, é menor ou igual a 0, se G é um caminho, menor ou igual a 1, se G é *outerplanar* ou menor ou igual a 2, se G é planar e hamiltoniano.

Eles afirmam que como existem grafos planares maximais não hamiltonianos, então existem grafos planares cuja imersão num *book* precisa de pelo menos 3 páginas. Neste caso, a ordenação dos vértices na espinha não é conhecida. A Tabela 1.3 resume os resultados sobre $bt(G)$.

G	$bt(G)$
caminho	≤ 0
outerplanar	≤ 1
planar e hamiltoniano	≤ 2
planar maximal não hamiltoniano	≥ 3

TAB. 1.3: Menor número de páginas para imersão em *book* em algumas classes de grafos

Finalmente, Hong et al. (HONG (2015)) desenvolveram um algoritmo para uma classe que inclui os grafos *outerplanares*. Um grafo é dito *outer-1-planar* se possui uma imersão na qual cada vértice está na face externa e cada aresta possui no máximo um cruzamento. O algoritmo em questão testa em tempo linear se um dado grafo é *outer-1-planar*. O algoritmo pode ser utilizado para produzir uma imersão *outer-1-planar* também em tempo linear, se existir.

1.1 OBJETIVO

O objetivo principal deste trabalho é implementar um algoritmo completamente dinâmico para manutenção da *outerplanaridade* em grafos dinâmicos. O algoritmo implementado é baseado no trabalho desenvolvido por Araújo em 2004 (ARAUJO (2004)), no qual é realizada uma descrição das características dos grafos *outerplanares* e das operações de inserção e remoção de arestas.

1.2 JUSTIFICATIVA

Existem diversos trabalhos na área de segurança que utilizam grafos para modelar situações específicas. A seguir são referenciados três exemplos de aplicações que utilizam grafos para resolver problemas na área de segurança.

Na dissertação recente do Programa de Engenharia de Defesa desenvolvida por Dias (DIAS (2016)), são apresentados estudos comparativos para detecção de comunidades em redes complexas criadas a partir de dados do website mantido pela prefeitura da cidade de Nova Iorque, nos Estados Unidos (OFNEW YORK (2017)), que contém informações sobre crimes ocorridos na cidade. A utilização de diferentes abordagens em redes complexas permitiu a detecção de comunidades baseado na localização dos crimes ocorridos no ano de 2015.

O trabalho desenvolvido por Campbell et al. (CAMPBELL (2013)) utiliza um grafo gerado a partir de informações obtidas do Institute for the Study of Violent Groups. Esse grupo mantém uma base de dados com documentos sobre atividades terroristas e criminais. Os dados são abertos e podem ser achados artigos, documentos judiciais e relatórios policiais. A partir da base foi criado um grafo e foram analisadas comunidades detectadas por algoritmos que utilizam o grafo como entrada. Os diferentes algoritmos para detecção de comunidades apresentaram alta precisão ao encontrar comunidades que correspondem a grupos realmente violentos (como por exemplo o Al Qaeda).

Palucka et al. (PALUCKA (2016)) descreve um estudo que permite identificar como mudam os padrões de comportamento dentro de uma comunidade. Para tal fim, foram realizados experimentos com o objetivo de propor normas "anti-conflito"em grupos de estudantes de escolas de segundo grau nos Estados Unidos. A partir desse estudo foi possível identificar condições sob as quais o comportamento de referentes sociais pode influenciar outros estudantes e aumentar o nível de conflito dentro da escola.

Nos três trabalhos mencionados, a partir de dados de origens diversas são modelados grafos, e com esses grafos como entrada são aplicados algoritmos para resolver problemas concretos. Com o uso crescente da Internet, os dados gerados pelo uso de dispositivos interconectados e os volumes de informação armazenada são cada vez maiores, e para poder analisar esses dados não estruturados, uma das abordagens possíveis é analisar a topologia de grafos convenientemente modelados.

Nesta dissertação abordamos o problema de grafos dinâmicos, ou seja, grafos que mudam através do tempo. O caso particular estudado considera que novas arestas podem ser inseridas ou removidas. Para situações reais, é bastante comum que determinadas

características sejam modificadas com o passar do tempo, o que possibilita a modelagem computacional utilizando grafos dinâmicos.

1.3 CONTRIBUIÇÃO

Este trabalho apresenta uma implementação para o algoritmo de manutenção da *outerplanaridade*, cujos detalhes teóricos podem ser vistos em (ARAUJO (2004)).

Dentre as contribuições desta dissertação podemos mencionar acerca das estruturas de dados utilizadas durante a execução do algoritmo e das operações realizadas para cada situação de inserção ou remoção de arestas do grafo.

Outra contribuição é a disponibilização do código do algoritmo, que pode ser utilizado em aplicações específicas. Além disso, o código implementado também pode ser alterado, de forma a resolver o problema dinâmico incremental da manutenção da *outerplaanridade* utilizando a estrutura Union-Find. Vale ressaltar que mesmo utilizando a estrutura de dados em questão, a complexidade geral do algoritmo se mantém a mesma (linear no pior caso).

1.4 ORGANIZAÇÃO DOS CAPÍTULOS

Este texto está organizado da seguinte maneira: no Capítulo 2 são apresentados conceitos básicos que serão utilizados ao longo dos demais capítulos. O Capítulo 3 aborda alguns algoritmos presentes na literatura que tratam sobre conectividade e planaridade em grafos dinâmicos. O Capítulo 4 apresenta uma visão geral teórica sobre o algoritmo de manutenção da *outerplanaridade* em grafos dinâmicos proposto em (ARAUJO (2004)). Todos os detalhes de implementação e estruturas de dados utilizadas no algoritmo de manutenção da *outerplanaridade* em grafos dinâmicos são descritos no Capítulo 5. A ánalise de complexidade, avaliando cada um dos casos descritos no algoritmo é feita no Capítulo 6. O Capítulo 7 apresenta as considerações finais, abordando a conclusão e possíveis trabalhos futuros.

2 CONCEITOS BÁSICOS

Serão descritos nesta seção alguns conceitos básicos para a compreensão do estudo. Algumas das definições podem ser encontradas em ARAUJO (2004), BONDY (2010), CORMEN (2001), CPLUSPLUS.COM (2017b), CPLUSPLUS.COM (2017a) e SZWARCFITER (1986).

Definição 2.1 *Um grafo $G = (V, E)$ consiste em um par de conjuntos tais que V é finito e não vazio e $E \subseteq \{(x, y)|x, y \in V\}$. Os elementos de V e E são chamados vértices e arestas, respectivamente. A cardinalidade do conjunto V é representada por n, enquanto a do conjunto E por m.*

Definição 2.2 *Dois vértices x e y são adjacentes em G quando $(x, y) \in E$. O conjunto de adjacência de um vértice v, $Adj(v)$, é o conjunto formado por todos os vértices do grafo que são adjacentes a v.*

Definição 2.3 *O grafo $G' = (V', E')$ é um subgrafo de $G = (V, E)$ quando $V' \subseteq V$ e $E' \subseteq E$.*

Definição 2.4 *Uma sequência de vértices $p_{v_1,v_k} = v_1, v_2, ..., v_k$, tal que $(v_i, v_{i+1}) \in E, 1 \leq i \leq k-1$ e os vértices $v_i, 1 \leq i \leq k$ são diferentes, é denominada um caminho de v_1 a v_k.*

Definição 2.5 *A distância $d(u, v)$ entre dois vértices u e v de um grafo G é o comprimento (número de arestas) do caminho mais curto entre eles, se existir; caso contrário, $d(u, v) = \infty$.*

Definição 2.6 *Um ciclo é definido por uma sequência de vértices $C = v_1, v_2, ..., v_k, v_1$ onde $v_1, v_2, ..., v_k$ é um caminho e $k \geq 3$.*

Definição 2.7 *Um ciclo hamiltoniano é um ciclo que passa por todos os vértices do grafo G.*

Definição 2.8 *Seja S um conjunto e $S' \subseteq S$. Diz-se que S' é maximal em relação a uma certa propriedade P quando S' satisfaz a propriedade P e não existe subconjunto $S'' \supseteq S'$ que também satisfaz P.*

Definição 2.9 *Um grafo G é conexo se para quaisquer par de vértices u e v em V existe um caminho. Caso contrário, o grafo é desconexo.*

Definição 2.10 *Os subgrafos maximais conexos são denominados componentes conexos de G.*

Definição 2.11 *Um grafo é biconexo se não existe nenhum vértice cuja remoção torna o restante do grafo desconexo.*

Definição 2.12 *Os blocos de um grafo são os seus subgrafos maximais biconexos.*

Definição 2.13 *Um vértice de articulação de G é qualquer vértice que, se removido do grafo, aumenta o número de componentes conexos de G.*

Definição 2.14 *Uma aresta (u, v) de G é denominada ponte se a sua remoção aumenta o número de componentes conexos de G.*

Definição 2.15 *Uma corda é uma aresta que une dois vértices não consecutivos de um ciclo.*

A busca em largura (do inglês, *Breadth-First Search*-BFS, (CORMEN (2001))) é um procedimento utilizado para determinar quais os vértices que podem ser alcançados por um caminho desde um vértice dado. O mesmo inicia-se por um vértice, denominado raiz ou fonte, e explora todos os vértices vizinhos. O nome do procedimento é devido ao critério de expansão da fronteira entre os vértices alcançados e não alcançados durante o procedimento.

A busca em profundidade (do inglês, *Depth-First Search*-DFS (CORMEN (2001))) é também um procedimento usado para determinar quais os vértices que podem ser alcançados por um caminho desde um vértice dado. O mesmo inicia-se por um vértice raiz ou fonte, e explora vértices com um critério diferente da BFS. Neste caso o objetivo é explorar vértices em profundidade, enquanto seja possível. O critério usado neste caso é a aresta a ser explorada é incidente ao vértice mais recentemente alcançado pelo algoritmo. O procedimento continua até que não existam mais vértices que sejam alcançados por um caminho desde a raiz.

Definição 2.16 *Uma imersão no plano é uma representação (desenho) de um grafo G em uma superfície plana.*

Definição 2.17 *Duas arestas de G se cruzam em uma imersão no plano, se existe um ponto, diferente de suas extremidades, em que elas se interceptam.*

Definição 2.18 *Um grafo G é planar se existe uma imersão no plano do mesmo sem cruzamento de arestas.*

Definição 2.19 *Dada uma imersão no plano de um grafo planar sem cruzamento de arestas, uma face interna é uma região maximal limitada por vértices e arestas. A face externa é a única face ilimitada por vértices e arestas.*

Teorema 2.1 (Teorema de Euler) *Em um grafo planar com n vértices, m arestas, f faces e c componentes conexas verifica-se: $n - m + f = c + 1$.*

Definição 2.20 *Um grafo G é outerplanar se admite uma imersão no plano sem cruzamento de arestas, tal que todos os vértices pertencem à face externa na imersão.*

Definição 2.21 *Um grafo $G = (V, E)$ é outerplanar maximal, se G é outerplanar a adição de uma aresta entre qualquer par de vértices resulta em um novo grafo G' que não é outerplanar.*

Teorema 2.2 *Em um grafo outerplanar maximal com n vértices, o número de arestas é $2n - 3$.*

A seguir, apresentamos alguns conceitos relacionados a grafos dinâmicos e algoritmos estáticos e dinâmicos.

Definição 2.22 *Um grafo dinâmico designa um grafo que está sujeito a uma série de atualizações (adição/remoção de vértices e/ou arestas) com o passar do tempo.*

Definição 2.23 *Um algoritmo estático fornece um resultado de acordo com os dados apresentados de uma só vez no início da sua execução, não sendo permitido qualquer tipo de modificação dos dados.*

Definição 2.24 *Um algoritmo dinâmico possui a capacidade de encontrar e adaptar uma solução de um problema a partir de soluções anteriores, com base nos dados introduzidos ou modificados, sem ser necessário recalcular toda a solução.*

Os problemas de grafos dinâmicos podem ser classificados de acordo com os tipos de atualizações permitidas. A seguir apresentamos os diferentes tipos conhecidos na literatura.

Definição 2.25 *Um problema de grafo dinâmico é denominado completamente dinâmico se as operações de atualização incluem inserção e remoção de arestas ou vértices. Por outro lado, o problema de grafo dinâmico é denominado parcialmente dinâmico se somente um tipo de atualização é permitido. O problema de grafo dinâmico é incremental se somente inserções são permitidas, enquanto é denominado decremental se somente remoções são permitidas.*

A seguir, definem-se os conceitos associados à complexidade do pior caso. Suponha um problema P, que seja resolvido por um algoritmo A e é dada uma entrada de tamanho de entrada n, e seja $T(n)$ o número de operações elementares necessárias para P ser solucionado pelo algoritmo A. Diz-se que A executa em pior caso, se $T(n)$ é o tempo máximo em que qualquer instância de tamanho n irá ser processada. Analogamente, A executa em melhor caso, se $T(n)$ for o tempo mínimo em que qualquer instância de tamanho n é processada. Para comparar tempos de execução de algoritmos, utilizam-se as notações **Theta(Θ)**, **O-grande(O)** e **Ômega(Ω)**, definidas a seguir, baseando-se em CORMEN (2001).

Definição 2.26 *Dadas duas funções $f, g : R \rightarrow R$, tais que existam c e k naturais que satisfaçam a $f(n) \leq c.g(n), \forall n \geq k$, define-se f como $O(g)$ (f é O-grande de g). Ou seja, a função f cresce assintoticamente mais lentamente que g.*

Exemplo: Para $g(n) = n^2$ e $f(n) = n$, tem-se que f é $O(g)$, pois $f(n) \leq g(n), \forall n \geq 1$.

Definição 2.27 *Dadas duas funções $f, g : R \rightarrow R$, tais que existam c e k naturais que satisfaçam a $f(n) \geq c.g(n), \forall n \geq k$, define-se f como $\Omega(g)$ (f é Ômega de g). Ou seja, a função f cresce assintoticamente mais rapidamente que g. Logo, pode-se dizer que se f é $O(g)$, g é $\Omega(f)$.*

Exemplo: Para $g(n) = n^2$ e $f(n) = n$, diz-se que g é $\Omega(f)$.

Definição 2.28 *Dadas duas funções $f, g : R \rightarrow R$, tais que existam c, d e k naturais que satisfaçam a $c.g(n) \leq f(n) \leq d.g(n), \forall n \geq k$, define-se f como $\Theta(g)$ (f é Theta de g). Com a notação O-grande define-se um limite superior, com a notação Ômega define-se um limite inferior e com a notação Theta define-se ambos.*

Exemplo: Para $g(n) = n^2$ e $f(n) = n^2 + n$, diz-se que f é $\Theta(g)$.

A seguir serão apresentadas as estruturas de dados utilizadas no algoritmo de manutenção da *outerplanaridade*.

Definição 2.29 *Um array ou vetor é uma estrutura de dados que armazena uma coleção finita de elementos de tal forma que cada um dos elementos possa ser identificado por um índice ou chave. Os elementos de um array podem ser acessados ou atualizados rapidamente por meio de seus índices. Um array possui as seguintes operações:*

- Inserir(i, v): dado um valor v e a posição (índice) do vetor i na qual se deseja inserir v, a inserção no array pode ser feita pela associação $array[i] = v$, o que garante uma complexidade constante, $O(1)$;

- Buscar(i): dada uma posição i, a operação de busca retorna o valor associado ao índice i, i.e., $array[i]$. A complexidade desse tipo de busca é $O(1)$;

- Buscar(v): dado um valor v, a operação de busca retorna a posição i tal que $array[i] = v$. Como é possível que essa busca percorra todo o vetor, a complexidade dessa operação é linear no tamanho do vetor, $O(n)$.

Definição 2.30 *Uma lista duplamente encadeada é uma estrutura de dados de acesso sequencial na qual cada elemento mantém a informação de como localizar o elemento anterior e o elemento posterior. Sua característica permite operações de inserção e remoção em tempo constante, mas não é possível realizar um acesso a um elemento específico (para acessar um elemento específico, deve-se percorrer a lista). As listas duplamente encadeadas possuem as seguintes operações:*

- Inserir(p, v): dado um valor v e uma posição p da lista duplamente encadeada, a operação de inserção adiciona um elemento com valor v à lista após a posição p. Como a lista permite apenas o acesso sequencial, a complexidade para inserção de um elemento é $O(n)$;

- Buscar(v): dado um valor v, a operação de busca em uma lista duplamente encadeada consiste na procura do primeiro elemento com valor v. Como a busca deve percorrer os elementos presentes na lista, a busca é executada com complexidade $O(n)$;

- Remover(p): dada uma posição p, a operação de remoção em uma lista encadeada apaga o elemento presente na posição p da lista. Como é necessário um acesso sequencial até a posição desejada, tem-se que a complexidade também é linear no pior caso.

Definição 2.31 *Uma pilha é uma estrutura de dados que admite a remoção e a inserção de elementos segundo a seguinte regra: o último elemento a ser inserido na pilha é o primeiro elemento a ser removido da mesma. As operações a serem realizadas em uma pilha são as seguintes:*

- Inserir(v): dado um valor v, a operação de inserção adiciona v ao topo da pilha. A complexidade da operação é constante, $O(1)$;

- Remover: a operação de remoção retira o elemento do topo da pilha. Para a realização dessa tarefa, a complexidade é $O(1)$.

Definição 2.32 *Um hash, ou tabela de dispersão (CORMEN (2001)), é uma estrutura de dados que associa chaves de pesquisa a valores. Seu objetivo é indexar o vetor usando chaves, para fazer uma busca rápida e obter o valor desejado, generalizando a ideia de vetor. Quando o número de chaves usadas é relativamente menor ao número de valores armazenados, esta estrutura é uma alternativa efetiva para fazer busca de valores. E nesse caso, mais de uma chave corresponde ao mesmo valor produzindo o conceito de colisão. Existem várias abordagens para tratamento de colisões, sendo uma delas o armazenamento de valores em listas encadeadas. As operações a serem realizadas em um hash são as seguintes:*

- Inserir(v): dado um valor v, a operação de inserção adiciona o elemento v na tabela de dispersão. A utilização de uma função de hash adequada garante a inserção com complexidade $O(1)$ no caso médio. Para o caso em que ocorrem diversas colisões, a inserção ocorre com complexidade $O(n)$ no pior caso;

- Buscar(v): dado um valor v, a operação de busca realiza a verificação da existência de um elemento com valor v na tabela de dispersão. A busca de um determinado valor ocorre com complexidade constante no caso médio e complexidade linear no pior caso;

- Remover(v): dado um valor v, a operação de remoção retira o elemento com valor v do hash. De forma semelhante às outras operações, a complexidade é $O(1)$ no caso médio e $O(n)$ no pior caso.

Definição 2.33 *Uma estrutura de dados Union-Find é uma estrutura que armazena os conjuntos de elementos particionados em subconjuntos disjuntos. A estrutura Union-Find suporta as seguintes operações:*

- CriarConjunto(v): a operação cria um novo conjunto que será associado ao elemento de valor v. Tal operação executa com complexidade constante, $O(1)$;

- Buscar(v): dado um elemento v, a operação de busca encontra o conjunto ao qual v está atualmente associado. A complexidade da operação é $\Theta(\alpha(n))$, onde $\alpha(n)$ é a função inversa de Ackermann;

- Unir(u, v): dados dois elementos u e v, a operação de união é a fusão dos dois conjuntos aos quais os elementos u e v estão associados (caso sejam distintos). Assim como na operação de busca, a complexidade da operação é $\Theta(\alpha(n))$

Definição 2.34 *Um conjunto não ordenado (CPLUSPLUS.COM (2017b)) é uma estrutura de dados que armazena elementos únicos sem nenhuma ordem específica e que permite*

uma rápida busca de elementos baseado em seu valor. Internamente, os elementos de um conjunto não ordenado não estão armazenados em nenhuma ordem específica, mas estão organizados em posições que dependem dos seus valores de hash para permitir um rápido acesso aos elementos. As operações e suas respectivas complexidades em um conjunto não ordenado são semelhantes às de um hash.

Definição 2.35 *Um mapeamento não ordenado (CPLUSPLUS.COM (2017a)) é uma estrutura de dados associativa que armazena elementos formados pela combinação de uma chave e um valor, e que permite uma rápida busca pelos elementos baseada em suas chaves. Internamente, os elementos de um mapeamento não ordenado não estão armazenados em nenhuma ordem específica, mas estão organizados em posições que dependem dos valores de hash das suas chaves para permitir um rápido acesso aos elementos. As operações e suas respectivas complexidades em um mapeamento não ordenado são semelhantes às de um hash.*

3 ALGORITMOS EM GRAFOS DINÂMICOS PARA MANUTENÇÃO DE CONECTIVIDADE E PLANARIDADE

Existem na literatura diversos resultados importantes para os diferentes tipos de problemas de grafos dinâmicos. Este capítulo apresenta brevemente algumas soluções que exploram propriedades como a conectividade e a planaridade. A primeira delas é uma solução determinística para o problema de manutenção da conectividade, proposta por Holm, Lichtenberg e Thorup (HOLM (2001)), cujo custo computacional amortizado para atualização é $O(\log^2 n)$ e o custo para consulta, $O(\log n/\log(\log n))$. A segunda consiste num método probabilístico, proposto por Kapron, King e Mountjoy (KAPRON (2013)), com custo computacional de pior caso $O(\log^4 n)$ para inserção de aresta no grafo, $O(\log^5 n)$ para remoção de aresta e $O(\log n/\log(\log n))$ para consultas sobre conectividade. Já a terceira solução apresenta uma abordagem completamente dinâmica e determinística para testar a planaridade em imersões planares de grafos, proposta por Italiano, La Poutré e Rauch (ITALIANO (1993)), cuja estrutura de dados para solução do problema suporta atualizações e consultas na imersão de um grafo planar em $O(\log^2 n)$ no pior caso. Finalmente, a quarta solução apresenta um algoritmo determinístico para suportar a imersão planar de um grafo dinâmico, proposta por Holm e Rotenberg (HOLM (2015)). A solução suporta atualizações e consultas no grafo em tempo $O(\log^2 n)$ no pior caso. Além disso, o algoritmo possui uma operação de girar componentes na imersão (também com complexidade $O(\log^2 n)$ no pior caso), o que pode permitir que a inserção de determinada aresta seja possível, dada a imersão planar.

3.1 ALGORITMOS DE CONECTIVIDADE

3.1.1 ALGORITMO EXATO PARA MANUTENÇÃO DINÂMICA DE CONECTIVIDADE-HOLM, LICHTENBERG E THORUP

O algoritmo utiliza como ideia inicial o armazenamento de uma floresta geradora para o grafo de entrada G com n vértices utilizando árvores Euler-Tour como estrutura auxiliar. Somente com essa ideia, é fácil ver que para responder a pergunta sobre conectividade, basta que ambos os nós pertençam à mesma árvore, o que pode ser feito utilizando uma operação *FindRoot* para cada nó e verificando se as raízes são as mesmas. Uma operação *Insert(u, v)* pode ser feita verificando se u e v pertencem à mesma árvore e, caso não

pertençam, adicionando a aresta (u, v) à floresta geradora. O problema com esta abordagem é a operação *Delete(u, v)*. Para lidar com remoções, deve-se modificar a ideia inicial dividindo o grafo G em $\log n$ subgrafos e armazenar uma floresta geradora para cada subgrafo. Para implementar esta ideia, deve-se atribuir um nível para cada aresta. O nível de uma aresta é um inteiro entre 0 e $\log n$, que só pode diminuir ao longo das operações. Define-se G_i como o subgrafo do grafo G composto por arestas de nível i ou menor. Note que $G_{\log n} = G$. Seja F_i a floresta geradora de G_i.

Durante a execução do algoritmo deve-se manter as duas invariantes a seguir:

- Cada componente conexa de G_i deve ter no máximo 2^i vértices;

- $F_0 \subseteq F_1 \subseteq F_2 \subseteq ... \subseteq F_{\log n}$. Em outras palavras, $F_0 = F_{\log n} \cap G_i$, e $F_{\log n}$ é a floresta geradora de $G_{\log n}$.

O algoritmo realiza as seguinte operações:

- *Connected(u, v)*: Realiza a verificação se os vértices u e v estão conectados. Para fazer isso, primeiro consulta-se $F_{\log n}$ para saber se u e v estão na mesma árvore. Isto pode ser feito através da verificação em $F_{\log n}$ se *FindRoot(u)* = *FindRoot(v)*. O custo da operação é $O(\log n/\log(\log n))$ usando árvores Euler-Tour baseadas em alguma árvore $(\log n)$-ária.

- *Insert(e=(u, v))*: Insere a aresta $e = (u, v)$ no grafo. Para fazer isso, define-se o nível de aresta como $\log n$ e atualiza-se as listas de adjacência de u e v. Caso u e v pertençam a árvores distintas em $F_{\log n}$, adiciona-se e em $F_{\log n}$. O custo da operação é $O(logn)$.

- *Delete(e=(u, v))*: Remove a aresta $e = (u, v)$ do grafo. Para fazer isso, primeiramente atualiza-se as listas de adjacência de u e v. Caso e não pertença a $F_{\log n}$, o problema está resolvido. Caso contrário:

1. Remove-se e de F_i para todo $i \geq nível(e)$. A seguir, deve-se procurar uma aresta de substituição para reconectar u e v.

• Observe que a aresta de substituição não pode estar em um nível inferior ao $nível(e)$ pela segunda invariante (pois cada F_i é uma floresta geradora).

• Deve-se então procurar uma aresta de substituição no $nível(e)$ para manter as invariantes.

A aresta de substituição deve ser procurada da seguinte maneira:

2. Para $i = nível(e)$ até $\log n$:

(A) Seja T_u a árvore que contém u, e seja T_v a árvore contendo v. Sem perda de generalidade podemos assumir que $\mid T_v \mid \leq \mid T_u \mid$.

(B) pela primeira invariante, sabe-se que $\mid T_u \mid + \mid T_v \mid \leq 2^i$, de modo que $\mid T_v \mid \leq 2^{i-1}$. Isto significa que pode-se transferir todas as arestas de T_v para o nível $i - 1$.

(C) Para cada aresta (x, y) no nível i com x em T_v:

i. Se y está em T_u, adiciona-se (x, y) em $F_i, F_{i+1}, ..., F_{\log n}$, e a operação termina.

ii. Caso contrário atribui-se $nível(x, y) = i - 1$.

Para obter mais eficiência nas operações é necessário aumentar informações na árvore Euler Tour, por exemplo, armazenando o tamanho da árvore e as adjacências de cada nível para cada nó. O custo amortizado da operação de remoção é $O(\log^2 n)$.

3.1.2 ALGORITMO INCREMENTAL PROBABILÍSTICO PARA MANUTENÇÃO DINÂMICA DE CONECTIVIDADE- KAPRON, KING E MOUNTJOY

O algoritmo, assim como no caso da solução determinística, utiliza a ideia do armazenamento de uma floresta geradora do grafo G, podendo-se também utilizar árvores Euler-Tour como uma estrutura auxiliar para representação da floresta. O fator probabilístico do algoritmo corresponde à utilização de outra estrutura de dados auxiliar, a estrutura *cutset*. A utilização de tal estrutura fornece um algoritmo do tipo *Monte Carlo*, no qual o tempo de execução de determinada operação é fixo, porém existe uma determinada probabilidade de que o algoritmo gere uma resposta incorreta. Para o caso em questão, é garantido que se existir um caminho entre dois nós u e v no grafo G, o algoritmo responderá sempre corretamente. No caso em que não existe caminho entre os dois nós, existe uma alta probabilidade de que a resposta seja correta, porém essa resposta não é sempre correta.

3.1.2.1 A ESTRUTURA DE DADOS CUTSET

A principal ideia da estrutura de dados *cutset* é conseguir determinar facilmente uma aresta de corte. Em outras palavras, para o caso do algoritmo em que armazena-se a floresta geradora F do grafo G, deseja-se obter para uma dada árvore não maximal $T \in F$, uma aresta no conjunto de corte $(T, V \backslash T)$. A estrutura de dados responde de maneira determinística no caso em que apenas uma aresta separa os dois conjuntos. A ideia para o caso determinístico consiste em atribuir para cada aresta (x, y), $x \in V$, $y \in V$, um valor $x_b.y_b$, onde x_b e y_b são as representações binárias dos nós x e y respectivamente e $x_b.y_b$ representa a concatenação representações x_b e y_b. Para cada nó associa-se o valor igual ao XOR acumulado dos valores de todas as arestas incidentes ao dado nó. Para cada árvore $T \in F$, a árvore Euler-Tour associada a T mantém o XOR acumulado de todos os nós da árvore. A estrutura de dados *cutset* suporta as seguinte operações:

- *Insert(x, y)*: Insere a aresta no grafo G adicionando o valor $x_b.y_b$ no valor dos nós

x e y. Caso a aresta una duas árvores da floresta F, realiza-se a junção das árvores Euler-Tour associadas.

- *Delete(x, y)*: Remove a aresta de G, subtraindo o valor $x_b.y_b$ do valor dos nós x e y (basta aplicar o XOR novamente em cada um dos nós). Caso a aresta pertença a F, deve-se removê-la da árvore Euler-Tour associada.

- *Find(T)*: Encontra a aresta no corte $(T, V \backslash T)$, onde T é uma árvore em F. Pela suposição de que no máximo uma aresta corta os conjuntos, é possível afirmar que para a árvore T com valor acumulado $z = z_1.z_2$, $z \neq 0$, a aresta que corta os conjuntos é (z_1, z_2).

Para o caso em que mais de uma aresta corta os dois conjuntos, a técnica anterior não funciona, pois o valor acumulado encontrado pode não estar mais representando dois vértices. Nesse caso, realiza-se uma extensão da estrutura de dados para que a mesma consiga determinar com alta probabilidade uma aresta de corte. Na extensão, utilizam-se níveis i, $1 \leq i \leq 2log(n)$, para cada uma das $c \log n$ versões j, onde c é uma constante qualquer. Define-se $s_{i,j}(x)$ do nível i na versão j para o nó x. Quando uma aresta (x, y) é inserida em G, para cada $i = 0, 1, ..., 2 \log n - 1$ e cada $j = 1, 2, ..., c \log n$ adiciona-se com probabilidade $1/2^i$ o valor $x_b.y_b$ a $s_{i,j}(x)$ e $s_{i,j}(y)$. Com essa extensão, para qualquer j, quando o tamanho do conjunto de corte é aproximadamente 2^i, existe, com probabilidade constante, exatamente uma aresta do conjunto de corte com a propriedade de que para algum x em T, o valor da aresta de corte é adicionado a $s_{i,j}(x)$. Então repetindo essa ideia para $c \log n$ versões resulta em uma alta probabilidade de que essa condição seja verdadeira para pelo menos uma versão. As operações para esse caso são semelhantes ao caso determinístico, com as devidas modificações em virtude da extensão da estrutura de dados.

3.1.2.2 ALGORITMO GERAL

O algoritmo utiliza uma abstração da árvore de Boruvka utilizada no algoritmo homônimo para calcular uma árvore geradora mínima (EISNER (1997)). O algoritmo de Boruvka constrói uma sequência de camadas, onde a camada 0 possui apenas os nós do grafo e a camada l, possui as árvores formadas quando para cada árvore de camada $l - 1$, uma aresta une outra árvore de camada $l - 1$. Cada árvore na camada l é representada como um nó e é pai das árvores (nós) da camada $l - 1$ que o formam, constituindo a árvore (ou floresta) de Boruvka.

O algoritmo para conectividade dinâmica funciona com uma estrutura semelhante. Seja $top = \log n$. Para cada camada $l = 0, 1, ..., top$, mantém-se uma estrutura *cutset*

expandida para G com a floresta F_l. Para cada camada l, os bits aleatórios são gerados independentemente. Representa-se cada árvore T em cada F_l como um nó na camada l na árvore de Boruvka, e T é dita uma **componente de camada l da árvore de Boruvka**. Dada uma aresta e, seja l a menor camada tal que $e \in F_l$. Essa é uma **aresta de camada l**. Caso um nó na árvore de Boruvka não possui irmão e não foi marcado como maximal, isto é, uma árvore que gera uma componente maximal de G, diz-se que o nó **não tem correspondência**. Para garantir a corretude do algoritmo, são mantidas as invariantes a seguir na árvore de Boruvka:

- Os nós da camada 0 da árvore (ou floresta) de Boruvka são os vértices de G;

- Em cada camada l, $F_l \subseteq F_{l+1}$;

- Todo nó interno da árvore de Boruvka que não corresponde a uma árvore geradora de uma componente conexa maximal possui no mínimo um irmão;

- A estrutura a floresta na camada l é independente dos bits aleatórios em camadas superiores;

- Os nós da camada superior da árvore de Boruvka abrangem uma floresta geradora de G.

O algoritmo é inicializado criando-se para cada camada $l = 0, 1, ..., top$, uma estrutura de dados *cutset*, $S^l = s^l_{i,j}(x)$, $x \in V$, $0 \leq i \leq 2 \log n$, $1 \leq j \leq c \log n$, onde inicialmente F_l contém n árvores unitárias, uma para cada vértice de G. Caso o grafo inicial possua arestas, deve-se realizar a operação de inserção, que será descrita a seguir, para cada aresta. O algoritmo suporta as seguinte operações:

- *Insert(x, y)*: Insere a aresta (x, y) no grafo G. A inserção da aresta é feita adicionando-a na estrutura de dados *cutset* em cada camada. Se a aresta a ser adicionada conecta duas árvores desconectadas em F_{top}, adiciona-se a aresta em F_l, para todo $l > 0$.

- *Connected(x, y)*: Realiza a verificação se os vértices x e y estão conectados. Para fazer isso, primeiro consulta-se F_{top} para saber se x e y estão na mesma árvore. Se $T(x) = T(y)$, o algoritmo retorna "sim", caso contrário, retorna-se "não".

- *Search(T, S^l)*: Realiza uma busca na estrutura de dados *cutset* tentando encontrar uma aresta no corte $(T, V \backslash T)$ na camada l.

- *Delete(x, y)*: Realiza a remoção da aresta (x, y) do grafo G. O objetivo ao realizar as remoções é encontrar arestas de substituição (se existirem) preservando as invariantes do algoritmo. Primeiramente a aresta (x, y) é removida das estruturas de dados *cutset* que a contiverem. Isso pode causar a violação de uma das invariantes, pois a componente que contém x e/ou a componente que contém y pode, após a remoção, não ter correspondência.

Para preservar a invariante, deve-se encontrar o ancestral mais baixo A de x na árvore de Boruvka que não tem correspondência em sua camada e executar a operação *Reconnect(A, k)* (descrita a seguir) para unir A a um irmão em sua camada k. A operação se repete até que não haja mais ancestrais de x que não tem correspondência. O mesmo procedimento é executado para o nó y.

- *Reconnect(A, k)*: Realiza a conexão da componente A, que não tem correspondência, com um irmão, na camada k. Primeiramente deve-se tentar encontrar uma aresta de corte na camada k induzida pela componente A. Isso pode ser feito utilizando-se a função *Search(A, S^k)*. Se nenhuma aresta é encontrada, marca-se A como componente maximal. Caso contrário, seja $e = (u, v)$ a aresta encontrada. Deve-se então determinar se já existe um caminho entre v e w em alguma floresta $F_{k'}, k' > k+1$. Em caso afirmativo, remove-se uma das arestas do caminho de todas as florestas que a contiverem. A seguir insere-se e em $F_{k'}, k' > k$ e agora A possui um irmão, preservando a invariante.

3.2 ALGORITMOS DE PLANARIDADE

3.2.1 ALGORITMO PARA MANUTENÇÃO DINÂMICA DE PLANARIDADE - ITALIANO, LA POUTRÉ E RAUCH

O algoritmo em questão mantém uma imersão de um grafo planar sujeito a remoções e inserções (se possível) de arestas arbitrárias. A ideia geral do algoritmo consiste na utilização de árvores topológicas (FREDERICKSON (1985)) aumentadas com algumas informações adicionais para manter eficientemente as informações acerca da imersão do grafo planar. Além disso, todas as operações são realizadas no grafo transformado $\hat{G} = (V', E')$, onde V' é um conjunto cujos vértices possuem grau máximo igual a 3. O algoritmo proposto realiza as seguintes operações:

- *Query(u, v)*: Retorna "Sim"caso a inserção da aresta entre os vértices u e v mantenha a imersão planar do grafo e "Não"caso contrário;

- *Insert(u, v)*: Realiza a inserção da aresta entre os vértices u e v, caso a inserção preserve a planaridade da imersão;

- *Delete(u, v)*: Remove a aresta (u, v) do grafo.

Para a obtenção de $\hat{G}$ a partir de $G = (V, E)$, com n vértices, usa-se a seguinte transformação: seja $v \in V$ com grau $d > 3$ e cuja lista de adjacências consista dos vértices $u_1, u_2, ..., u_d$. Em $\hat{G}$, v é substituído por uma cadeia de $d - 1$ arestas onde o vértice v é eliminado e são acrescentados d vértices $v_1, v_2, ..., v_d$, além de arestas $(v_i, u_i), 1 \leq i \leq d$, e arestas $(v_i, v_{i+1}), 1 \leq i \leq d-1$. Com o grafo $\hat{G}$ gerado, a floresta geradora de $\hat{G}$ é mantida

e as arestas de árvore são armazenadas em uma árvore topológica. Uma estrutura auxiliar de agrupamento de arestas armazena as arestas que não pertencem à árvore geradora.

As operações de atualização do grafo G (inserção e remoção de arestas), além da consulta sobre a possibilidade de inserção de uma aresta podem ser realizadas com complexidade $O(\log^2 n)$ no pior caso. As estruturas podem ser construídas em tempo e espaço lineares, $O(n)$.

3.2.2 ALGORITMO PARA IMERSÃO PLANAR DE GRAFOS DINÂMICOS - HOLM E ROTENBERG

A ideia principal do algoritmo em questão é manter dinamicamente uma árvore geradora T para o grafo $G = (V, E)$, planar e conexo e também uma árvore geradora T^*, denotada co-árvore, para o grafo dual de G, $G^* = (F, E^*)$, onde F é o conjunto de faces do grafo G e E^* é o conjunto de arestas que existem sempre que duas faces possuam uma aresta de G em comum. O algoritmo baseia-se na observação de que se $E_T \subseteq E$ induz uma árvore geradora T em G, então $(E \backslash E_T)^*$ induz a co-árvore T^* em G^*. Uma outra observação importante utilizada no algoritmo é o fato de que se u e v são vértices, T é uma árvore geradora em G e e é qualquer aresta no caminho entre u e v em T, então qualquer face contendo ambos u e v está no ciclo induzido por e^* na co-árvore T^*. Logo, a ideia principal do algoritmo é procurar caminhos na co-árvore.

Para a manutenção eficiente da árvore geradora T e da co-árvore T^* são utilizadas *top trees* (ALSTRUP (2005)). O algoritmo também utiliza o conceito de **esquinas** em G, que são quádruplas (f, v, e_1, e_2), onde f é uma face, v é um vértice, e e_1, e_2 são arestas (não necessariamente distintas) que são consecutivas na lista de arestas de v e f. A ideia da utilização das esquinas é o posicionamento exato de novas arestas na imersão planar.

Em sua execução, o algoritmo realiza as seguintes operações:

- *Linkable(u, v)*: Determina se uma aresta entre u e v pode ser inserida em G sem violar a planaridade e sem modificar a imersão. Retorna o par de esquinas para u e v, (c_u, c_v), entre os quais a aresta pode ser adicionada, onde c_u representa a esquina na extremidade do nó u e c_v a posição da outra extremidade no vértice v;

- *Insert(c_u, c_v)*: Insere uma nova aresta no grafo G, entre as arestas da esquina c_u em uma extremidade e entre as arestas de c_v na outra. Retorna a nova aresta inserida;

- *remove(e)*: Remove a aresta e do grafo G. Retorna as duas esquinas que poderiam ser utilizadas para inserir a aresta novamente;

- *join(c_u, c_v)*: Combina os vértices u e v em um único vértice w e retorna as duas

novas esquinas c_w e c'_w que podem ser usadas para dividir o vértice novamente utilizando *cut*(c_w, c'_w);

- *cut*(c_w, c'_w): Divide o vértice em dois novos vértices u e v e retorna as esquinas c_u e c_v que podem ser utilizadas para juntar os vértices novamente utilizando *join*(c_u, c_v);

- *flip(C)*: Dada uma componente conexa C do grafo G, inverte a ordem da lista de arestas para cada vértice/face da componente;

- *one-flip-linkable(u, v)*: retorna a sugestão de um *flip* que torna a ligação entre os vértices u e v possível, se existir.

Para todas as operações de atualização do grafo G, o algoritmo apresenta complexidade $O(\log^2 n)$. No caso da operação *linkable(u, v)*, o algoritmo pode ser modificado de modo a retornar a lista com todas as esquinas que possiblitam a inserção da aresta (u, v), caso esse em que a complexidade do pior caso da operação passa a ser $O(\log^2 n + k)$, onde k é a quantidade de elementos da lista retornada.

4 MANUTENÇÃO DE OUTERPLANARIDADE EM GRAFOS DINÂMICOS

Este capítulo visa apresentar o algoritmo completamente dinâmico para manutenção da *outerplanaridade* proposto em (ARAUJO (2004)). O algoritmo considera um grafo simples $G = (V, E)$ (sem loops e multiarestas), sem pesos nas arestas, não direcionado e definido da seguinte forma:
Seja $G = (V, E)$ um grafo *outerplanar*, onde V é um conjunto fixo de n vértices e uma sequência *online* de atualizações (inserções ou remoções de arestas, e). São feitas consultas da seguinte forma: O grafo $G + e$ é *outerplanar*? O grafo $G - e$ é *outerplanar*?

A apresentação neste capítulo do algoritmo completamente dinâmico para manutenção de *outerplanaridade* permitirá descrever a implementação proposta no Capítulo 5. Para facilitar o desenvolvimento do algoritmo, são analisadas as operações, tanto para o caso da inserção quanto para a remoção de arestas, considerando o tipo da aresta que será inserida ou removida. Para o caso do algoritmo proposto, os tipos de arestas possíveis são: pontes (arestas cuja remoção aumentam a quantidade de componentes conexas no grafo), arestas hamiltonianas (arestas que estão presentes em um subgrafo que induz um ciclo hamiltoniano) e cordas (arestas entre dois vértices não consecutivos no ciclo hamiltoniano). Ao realizar uma operação de inserção ou remoção de aresta, o algoritmo identifica em qual dos três casos a aresta a ser considerada corresponde (apenas um deles) e realiza todas as modificações necessárias nas estruturas de dados para a manutenção da *outerplanaridade*.

4.1 CONCEITOS E PROPRIEDADES DOS GRAFOS OUTERPLANARES

Os grafos *outerplanares* e *outerplanares* maximais foram definidos no Capítulo 2. As Figuras 4.1 e 4.2 exibem exemplos de cada um desses grafos.

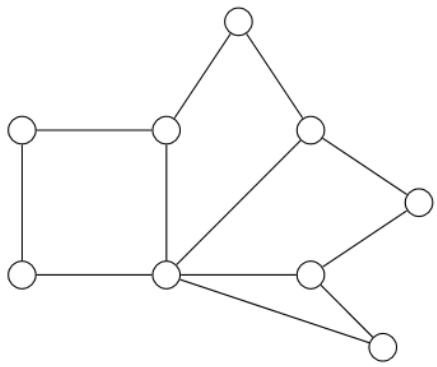


FIG. 4.1: Grafo *outerplanar*.

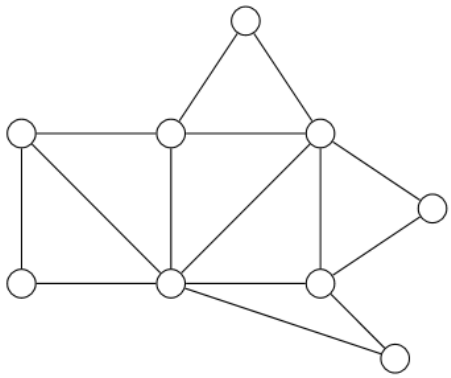


FIG. 4.2: Grafo *outerplanar* maximal.

Lema 4.1 *Seja G um grafo outerplanar sujeito a uma operação de remoção de aresta. O grafo resultante da remoção, G', também é outerplanar.*

Teorema 4.1 *Em um grafo outerplanar sem múltiplas arestas e loops, dois vértices possuem zero, uma ou duas faces internas em comum.*

A seguir define-se a estrutura do grafo bloco conexão, que é importante para manutenção de informações acerca dos grafos *outerplanares* no algoritmo.

Definição 4.1 (ARAUJO (2004)) *Seja $B = (V_b, E_b)$ o subgrafo induzido pelos vértices que compõem os blocos de tamanho dois de um grafo G outerplanar. Uma conexão C é um subgrafo conexo maximal de B ou um vértice de articulação de G não pertencente a V_b.*

Definição 4.2 (ARAUJO (2004)) *O grafo bloco conexão associado ao grafo G, denotado por $BC(G) = (V_{BC}, E_{BC})$, é o grafo cujos vértices são blocos com $n \geq 3$ e conexões*

de G, tal que dois vértices $\bar{x}$ e $\bar{y}$ de $BC(G)$ são adjacentes quando os blocos ou conexões correspondentes em G têm um vértice em comum e os vértices $\bar{x}$ e $\bar{y}$ não são simultaneamente blocos ou conexões. Além disso, atribui-se à aresta $(\bar{x}, \bar{y})$ um atributo, que é o vértice em comum a $\bar{x}$ e $\bar{y}$.

A Figura 4.3 mostra um grafo *outerplanar*, G, e o grafo bloco conexão associado, $BC(G)$.

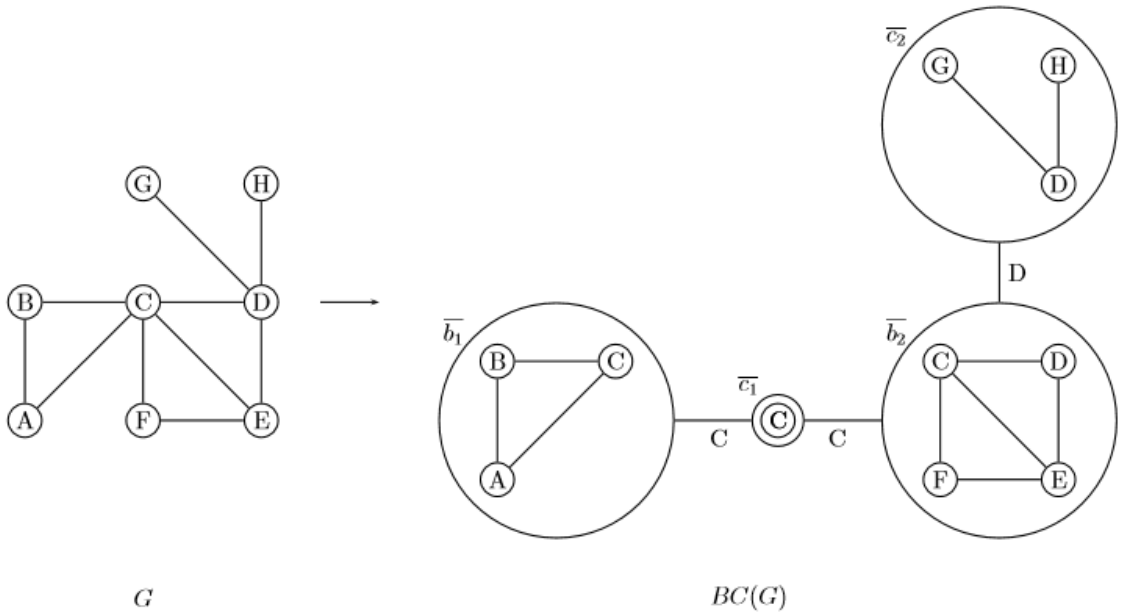


FIG. 4.3: Grafo bloco conexão $BC(G)$ associado ao grafo G.

Lema 4.2 (ARAUJO (2004)) *O subgrafo S de G associado ao vértice do tipo conexão é uma árvore.*

Lema 4.3 (ARAUJO (2004)) *O grafo bloco conexão $BC(G)$ associado a um grafo outerplanar G é uma floresta.*

A partir do Lema 4.3 pode-se concluir que entre qualquer par de vértices $\bar{u}$ e $\bar{v}$ na mesma árvore do grafo bloco conexão, sempre existe caminho de comprimento maior ou igual a zero (zero quando $\bar{u} = \bar{v}$).

Definição 4.3 (ARAUJO (2004)) *Seja $V_u \subseteq V_{BC(G)}$ o subconjunto de vértices de $BC(G)$ tal que $u \in V$ é interior a todos os vértices de V_u. Se $|V_u| > 1$, então existe um único vértice $\bar{w} \in V_{BC(G)}$, tal que $\bar{w}$ é conexão. Nesse caso, diz-se que u é ligado a $\bar{w}$. Caso $|V_u| = 1$, então u é ligado ao único elemento de V_u.*

4.2 INSERÇÃO DE ARESTA

O Lema 4.1 da Seção 4.1 garante que a remoção de uma aresta não invalida a *outerplanaridade* de um grafo $G = (V, E)$. Por outro lado, a inserção de uma aresta pode invalidar a *outerplanaridade* e essa possibilidade deve ser analisada. O algoritmo de manutenção da *outerplanaridade* apresentado em (ARAUJO (2004)) divide a adição das arestas em casos distintos para que esses possam ser analisados individualmente e com maior facilidade. A inserção de uma aresta $e = (u, v), u, v, \in V$ num grafo outerplanar $G = (V, E)$ pode ser classificada em um dos seguintes casos:

- e é uma ponte e conecta componentes conexas distintas em G;

- e é uma corda, cujas extremidades são vértices não consecutivos em um ciclo hamiltoniano, dentro de um bloco de G;

- e é uma aresta hamiltoniana, isto é, uma aresta que determina um novo ciclo em G, e por conseguinte um novo bloco.

Para o grafo da Figura 4.3, por exemplo, temos a seguinte classificação para as arestas:

- Pontes: (D, G) e (D, H);

- Cordas: (C, E);

- Arestas hamiltonianas: $(A, B), (A, C), (B, C), (C, D), (C, F), (D, E)$ e (E, F).

A seguir são apresentados os detalhes da operação de inserção de aresta para cada um dos três tipos. É interessante observar que nos casos em que se adiciona uma corda ou uma aresta hamiltoniana ao grafo, a *outerplanaridade* pode ser invalidada, enquanto a adição de uma ponte mantém o grafo *outerplanar*.

As Figuras 4.4, 4.5 e 4.6 ilustram os três tipos de aresta que podem ser inseridas utilizando o algoritmo de manutenção da *outerplanaridade*. As Figuras 4.7 e 4.8 exemplificam casos em que adição de uma aresta pode invalidar a *outerplanaridade*.

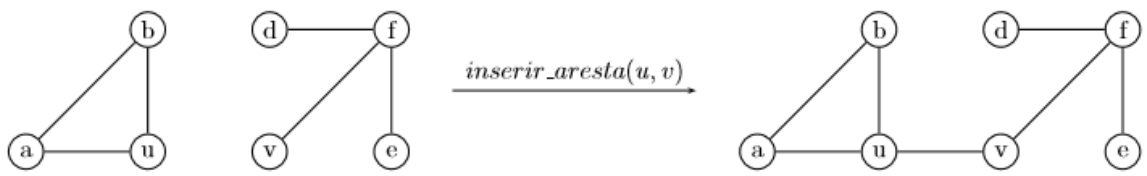


FIG. 4.4: Inserção de ponte no grafo G.

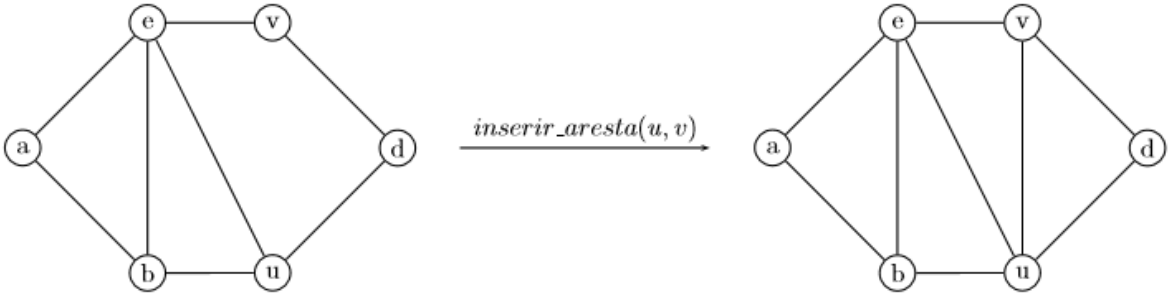


FIG. 4.5: Inserção de corda no grafo G.

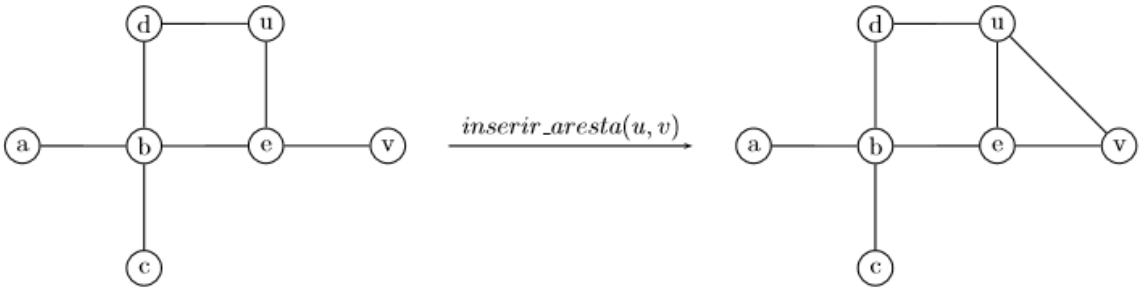


FIG. 4.6: Inserção de aresta hamiltoniana no grafo G.

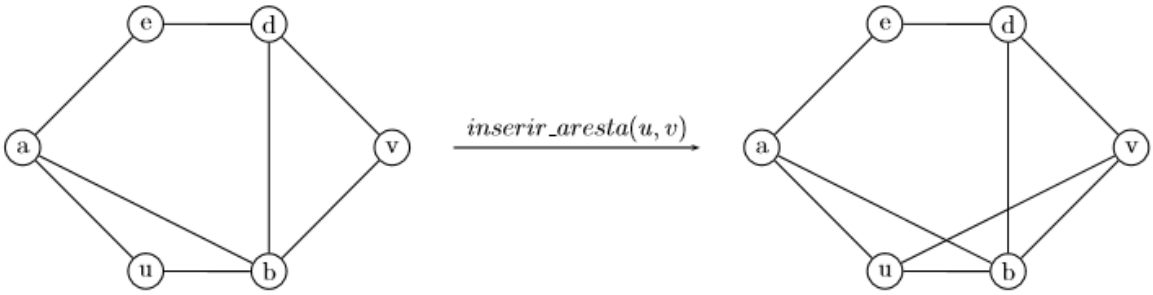


FIG. 4.7: Inserção de corda no grafo G que invalida a *outerplanaridade*.

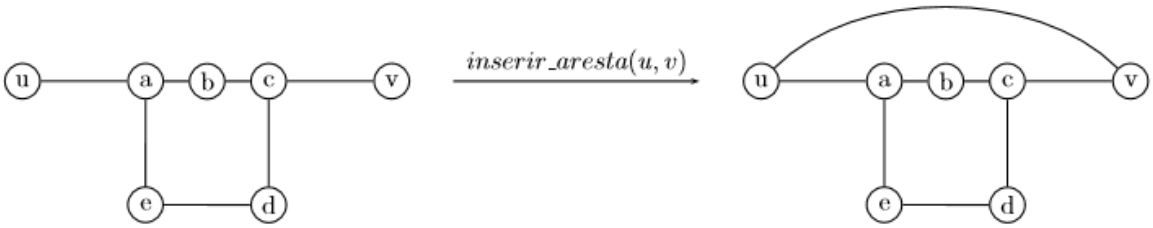


FIG. 4.8: Inserção de aresta hamiltoniana em G que invalida a *outerplanaridade*.

4.2.1 ADIÇÃO DE PONTE

Para o caso em que uma ponte (u, v) está sendo adicionada ao grafo *outerplanar*, ou seja, G é desconexo e u e v pertencem a duas componentes conexas distintas, deve-se analisar os vértices aos quais u e v estão ligados em $BC(G)$. Sejam $\bar{u} \in V_{BC(G)}$ o vértice ao qual u é ligado e $\bar{v} \in V_{BC(G)}$ o vértice ao qual v é ligado. Três subcasos devem ser analisados:

- $\bar{u}$ e $\bar{v}$ são conexões. Nesse caso a adição da ponte (u, v) implica na fusão das conexões $\bar{u}$ e $\bar{v}$ em uma única conexão, denotada por $\bar{c}$. Assim, todas as arestas em $BC(G)$ que tinham inicialmente como extremidade $\bar{u}$ ou $\bar{v}$, passam a ter a nova extremidade, $\bar{c}$. A Figura 4.9 ilustra o caso descrito.

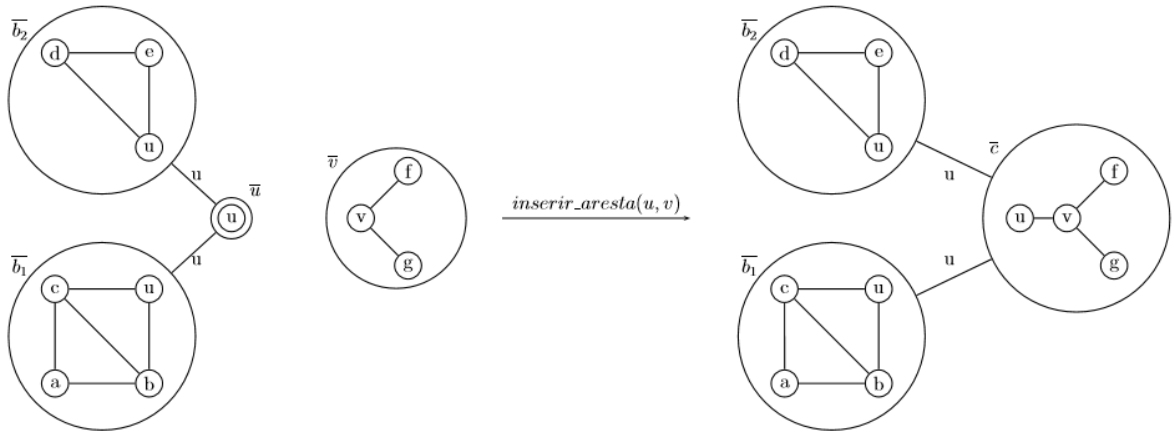


FIG. 4.9: Inserção de ponte entre vértices ligados a conexões.

- $\bar{u}$ e $\bar{v}$ são blocos. Nesse caso u e v pertencerão a uma nova conexão, que se ligará por arestas com atributos u e v aos blocos $\bar{u}$ e $\bar{v}$, respectivamente. A Figura 4.10 ilustra um exemplo de adição de uma ponte entre vértices ligados a blocos em $BC(G)$.

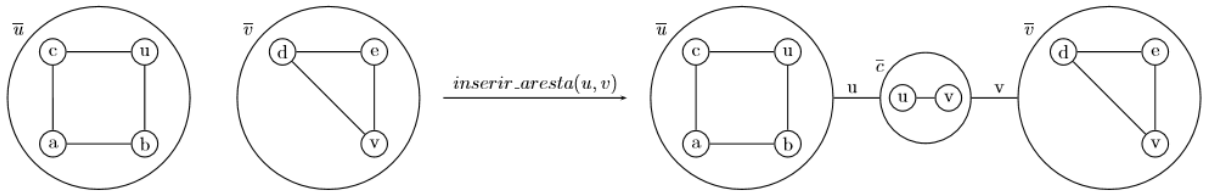


FIG. 4.10: Inserção de ponte entre vértices ligados a blocos.

- $\bar{u}$ é conexão e $\bar{v}$ é bloco (o caso em que $\bar{u}$ é bloco e $\bar{v}$ é conexão é análogo). Nesse caso v deixará de estar ligado ao bloco $\bar{v}$ e passará a fazer parte da conexão $\bar{u}$. A Figura 4.11 ilustra um exemplo deste caso.

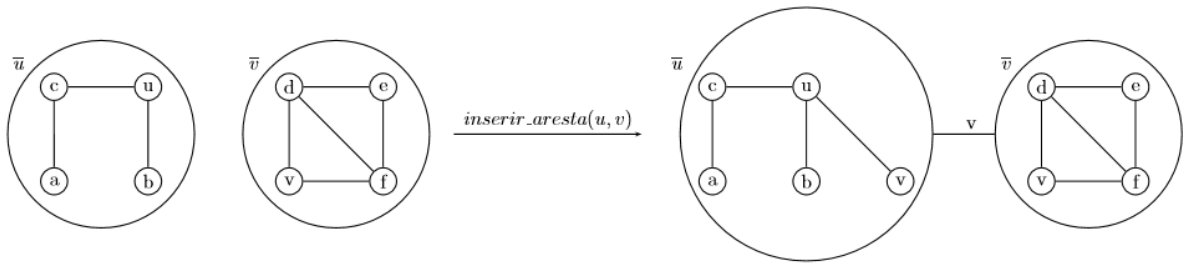


FIG. 4.11: Inserção de ponte entre um vértice ligado a conexão e outro ligado a bloco.

Após verificar em que caso a adição da ponte a ser inserida se encaixa, sendo sempre uma inserção que não invalida a *outerplanaridade*, deve-se atualizar a informação sobre as componentes conexas, por meio do uso de conjuntos que associam cada vértice do grafo G a uma componente conexa do grafo.

4.2.2 ADIÇÃO DE CORDA

A operação de adição de corda também requer atualização de algumas estruturas de dados, porém ela não modifica o grafo bloco conexão, visto que a inserção da aresta em questão é feita internamente a um bloco existente. Em compensação, é possível que a adição da aresta deste tipo invalide a *outerplanaridade*. Esse caso acontece quando a inserção da corda produz o cruzamento de arestas no grafo, ou seja, quando os vértices u e v não possuem uma face em comum, exceto pela face externa. Seja $F(u)$ o conjunto de faces associadas ao vértice u (faces às quais u faz parte) e $F(v)$ o conjunto de faces associadas ao vértice v. A adição da corda $(u, v), u, v, \in V$ é possível se $F(u) \bigcap F(v) \neq \varnothing$. Caso a adição da corda seja possível, uma nova face é criada no grafo resultante, e o conjunto de faces associadas a cada vértice deve ser atualizado.

A Figura 4.12 ilustra o caso de inserção de uma corda no grafo *outerplanar*.

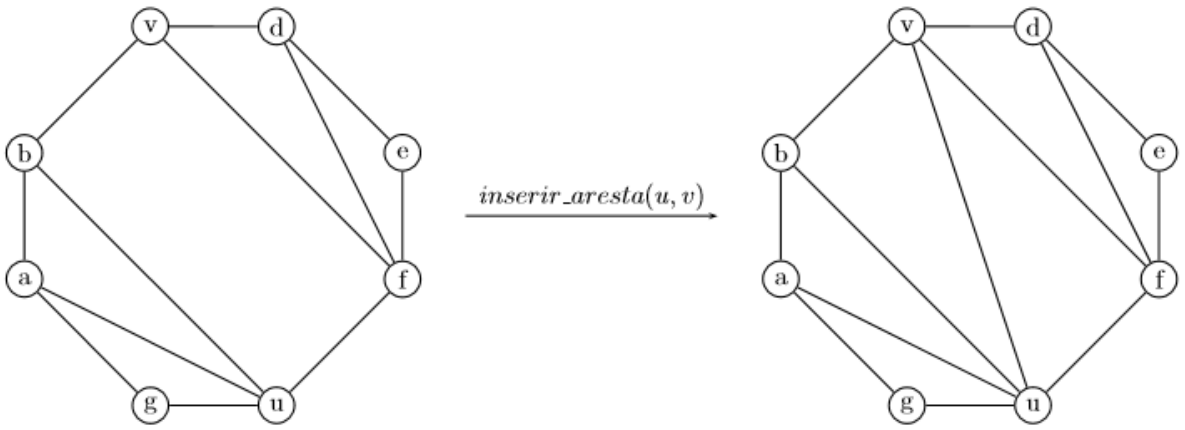


FIG. 4.12: Inserção de corda em G.

4.2.3 ADIÇÃO DE ARESTA HAMILTONIANA

Ao adicionar uma aresta hamiltoniana $(u, v), u, v \in V$ ao grafo $G = (V, E)$ (caso isso seja possível), forma-se um novo bloco, criando-se uma nova componente biconexa. A formação do novo bloco pode ser dividida em dois subcasos distintos:

- u e v estão ligados à mesma conexão, $\bar{c}$, no grafo bloco conexão $BC(G)$ associado. Nesse caso, a adição da aresta (u, v) gera um novo ciclo em G. Esse ciclo implica na formação de um novo bloco $\bar{b}$ que fará parte de $BC(G)$. Portanto, deve-se realizar as devidas atualizações no grafo bloco conexão, para preservar suas propriedades. Eventualmente, a adição da aresta (u, v) não envolve todos os vértices do subgrafo S associado à conexão $\bar{c}$. Sendo assim, os vértices que não estão envolvidos na criação do novo bloco devem ser associados a novas conexões, que se ligarão a $\bar{b}$. Os vértices que farão parte do novo bloco podem ser encontrados calculando-se o caminho mínimo entre u e v, p_{uv}. Os vértices que farão parte de novas conexões podem ser encontrados fazendo uma busca a partir de cada vértice do caminho p_{uv}, que explora apenas os vértices que eram ligados à conexão $\bar{c}$ e despreza outros vértices de p_{uv} (exceto aquele do qual se inicia a busca). Deve-se observar que, as arestas do caminho p_{uv} deixam de ser pontes e se tornam hamiltonianas, assim como a aresta (u, v).

A Figura 4.13 exemplifica um caso em que forma-se um novo bloco e os vértices estão ligados à mesma conexão.

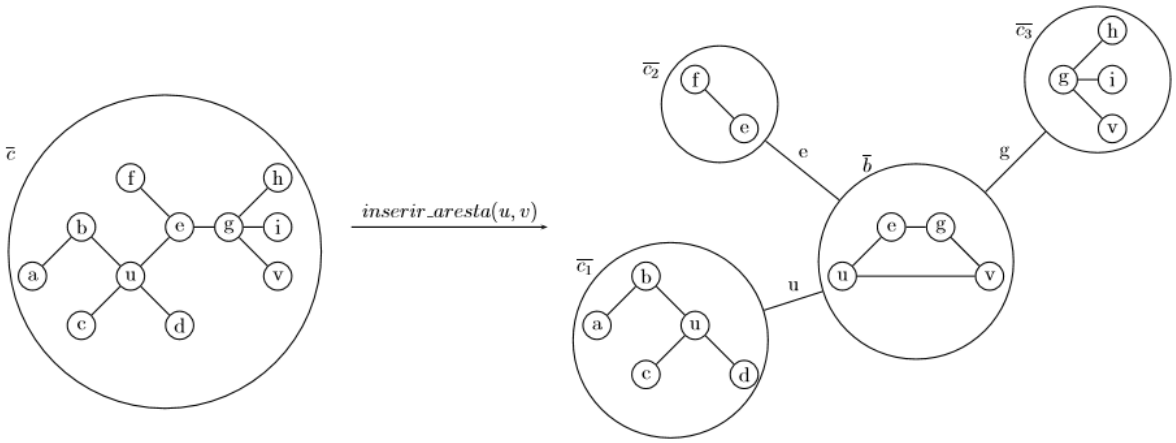


FIG. 4.13: Inserção de aresta hamiltoniana entre vértices da mesma conexão.

- u e v estão ligados a vértices distintos em $BC(G)$. Nesse caso é possível que a adição da aresta (u, v) invalide a *outerplanaridade*. Sendo assim deve-se realizar uma análise prévia se é possível inserir a aresta em G. O Teorema 4.2 apresenta as condições necessárias e suficientes para a realização da verificação:

Teorema 4.2 (ARAUJO (2004)) *Seja $G = (V, E)$ um grafo outerplanar e sejam u e v vértices de G não ligados ao mesmo vértice em $BC(G)$. O grafo $G' = (V, E \bigcup (u, v))$, é outerplanar se, e somente se as arestas pertencentes ao caminho mínimo p_{uv} satisfazem às seguintes condições: nenhuma aresta pertencente a p_{uv} é fronteira entre faces internas de G e duas arestas consecutivas de p_{uv} não pertencem ao mesmo bloco de G.*

As Figuras 4.14 e 4.15 ilustram os dois casos do Teorema 4.2 em que a adição de uma aresta invalida a *outerplanaridade*. Nas duas figuras são apresentados os grafos G e seus correspondentes grafos bloco conexão $BC(G)$.

Na Figura 4.14, a aresta (a, c) no caminho mínimo p_{uv} é fronteira de duas faces internas (f_1 e f_2) do grafo *outerplanar* G.

Na Figura 4.15, (a, b) e (b, c) são duas arestas consecutivas no caminho mínimo p_{uv} em G que pertencem ao mesmo bloco, $\bar{b}$, em $BC(G)$.

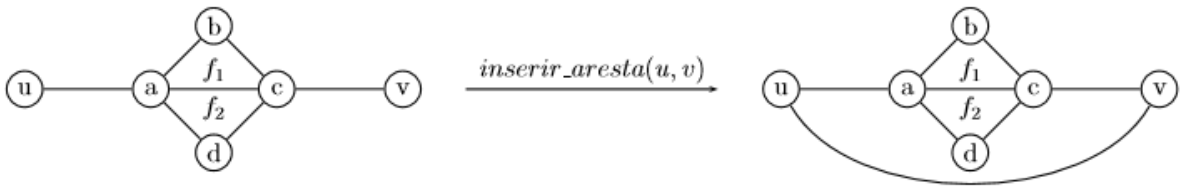


FIG. 4.14: Aresta de p_{uv} fronteira entre faces internas.

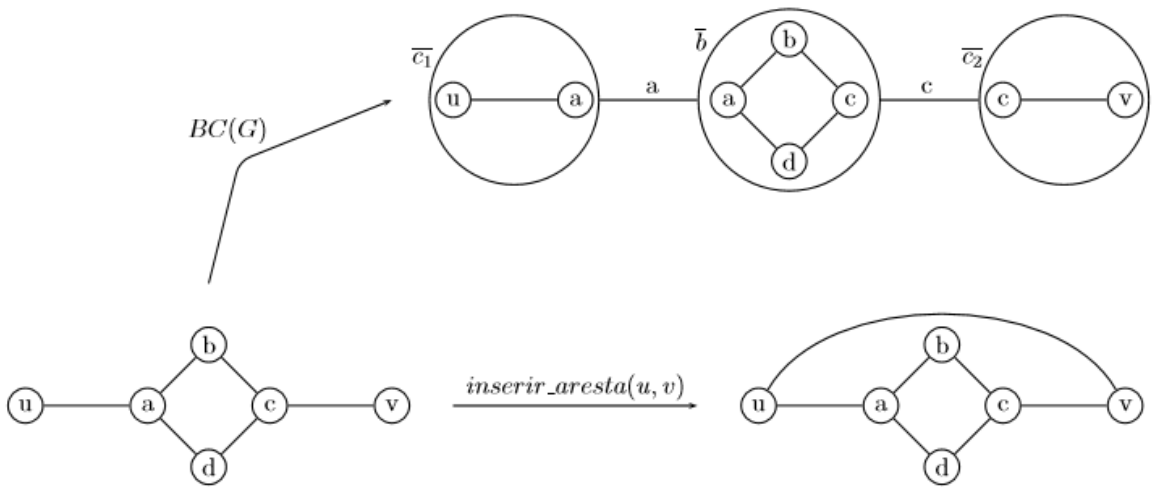


FIG. 4.15: Arestas consecutivas de p_{uv} pertencentes ao mesmo bloco.

Sendo possível a adição da aresta (u, v), faz-se necessário atualizar todas as estruturas de dados. O procedimento é realizado percorrendo-se o caminho mínimo p_{uv}. Para as conexões, a atualização é semelhante ao primeiro subcaso citado (u e v pertencem à mesma conexão). No caso de um bloco, deve-se atentar para o fato de que arestas pertencentes a p_{uv} deixam de ser hamiltonianas e passam a ser cordas. A Figura 4.16 mostra um exemplo de adição de aresta hamiltoniana, formando um novo bloco. Neste caso, o caminho mínimo em G é $p_{uv} = \{u, c, d, g, v\}$. Adicionando a aresta (u, v), as arestas (u, c) e (g, v), que antes da inserção eram hamiltonianas, passam a ser cordas.

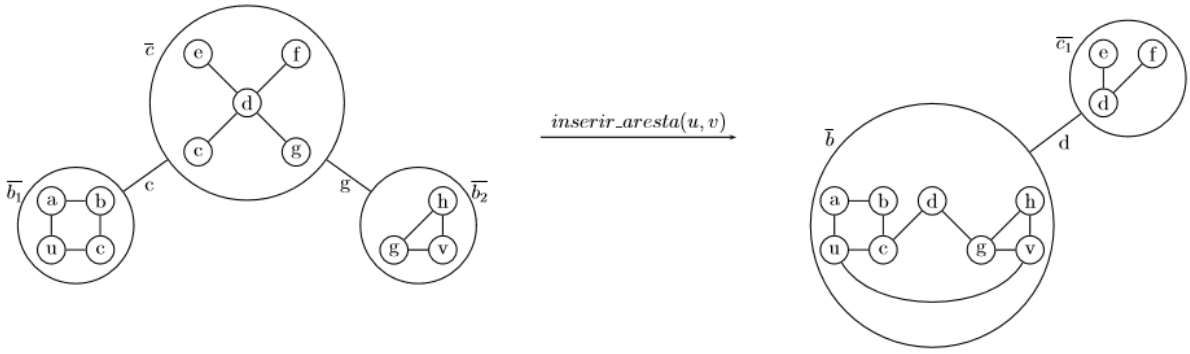


FIG. 4.16: Inserção de aresta hamiltoniana.

4.3 REMOÇÃO DE ARESTA

Conforme visto no Lema 4.1, a remoção de arestas em um grafo $G = (V, E)$ *outerplanar* mantém a propriedade da *outerplanaridade*. Sendo assim, a única preocupação é a manutenção das estruturas de dados que controlam a execução do algoritmo. Da mesma

maneira que procede-se na inserção, o algoritmo para remoção de arestas também realiza a identificação do tipo de aresta a ser removida no grafo, dividindo em três casos semelhantes. Dada uma aresta $e = (u, v), u, v \in V, e \in E$, os possíveis casos de remoção são:

- e é uma ponte e sua remoção divide uma componente conexa em G;
- e é uma corda, cuja remoção implica na fusão de faces adjacentes em um bloco;
- e é uma aresta hamiltoniana e sua remoção destrói um bloco de G.

Nas próximas seções serão descritos os detalhes para cada um dos casos possíveis de remoção de aresta.

4.3.1 REMOÇÃO DE PONTE

Para o caso de remoção de uma ponte, uma componente conexa será dividida em duas. Portanto, será necessário atualizar o grafo bloco conexão e manter a informação dos conjuntos associados a cada vértice. A Figura 4.17 ilustra um exemplo em que realiza-se uma remoção de ponte no grafo. Nesse caso, os vértices c, e, f, g, j, k, l, u e v se ligavam inicialmente à conexão $\bar{c}$. Após a remoção da aresta (u, v), a conexão é dividida e os vértices c, e, j e u passam a ser ligar à conexão $\bar{c_1}$, enquanto f, g, k, l e v ligam-se à conexão $\bar{c_2}$.

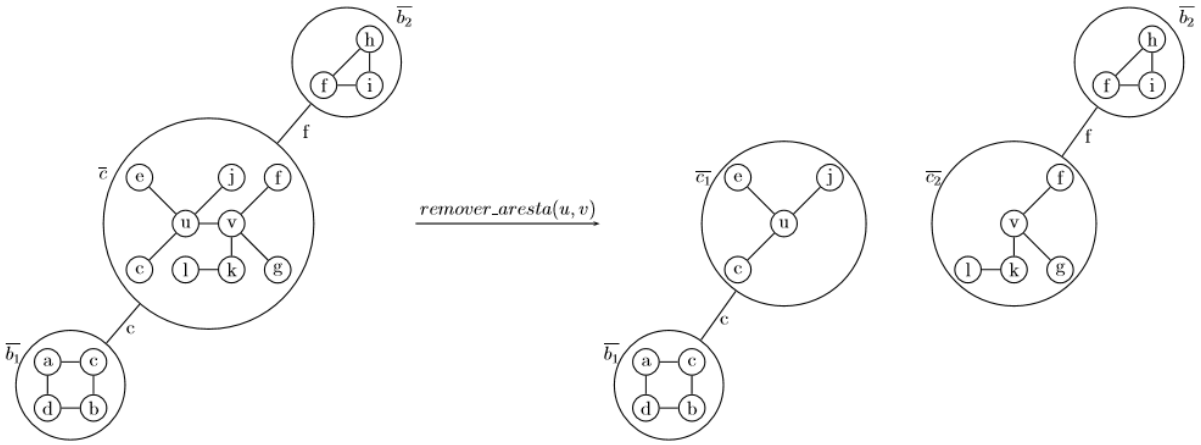


FIG. 4.17: Remoção de ponte em G.

4.3.2 REMOÇÃO DE CORDA

Assim como no caso da inserção de arestas, a remoção de uma corda em G não altera a estrutura do grafo bloco conexão. Porém, é necessário realizar a atualização do conjunto de faces associadas a um subconjunto dos vértices do ciclo hamiltoniano. A remoção de uma corda (u, v) implica na fusão de duas faces de um bloco. Sejam f_1 e f_2 as faces envolvidas

na remoção de (u, v). Como as faces serão fundidas, deve-se atualizar o conjunto de faces de todos os vértices que possuem a face f_2 em seus conjuntos para possuir a face f_1, ou o contrário. Para realizar a descoberta de quais vértices devem ser atualizados, novamente faz-se necessário o cálculo do caminho mínimo p_{uv} entre os vértices u e v após a remoção da corda (u, v). A Figura 4.18 exemplifica um caso de remoção de corda em G e as faces envolvidas no processo de remoção da aresta. Nesse caso, o caminho mínimo é $p_{uv} = \{u, f, v\}$. Portanto, deve-se remover a face f_2 da lista de faces dos vértices u, f e v, e adicionar a face f_1 à lista de faces do vértice f.

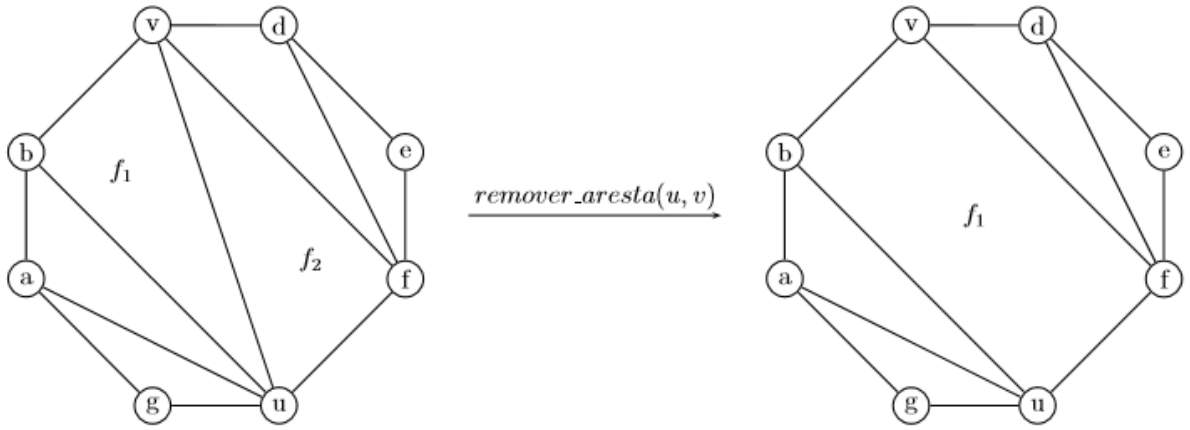


FIG. 4.18: Remoção de corda em G.

4.3.3 REMOÇÃO DE ARESTA HAMILTONIANA

A remoção de uma aresta hamiltoniana implica na modificação de um bloco $\bar{b}$. O bloco $\bar{b}$ pode ou não ser eliminado, mas o grafo bloco conexão será modificado. Além disso, com a remoção da aresta hamiltoniana, algumas arestas do bloco podem passar a ser de outros tipos (arestas hamiltonianas podem se tornar pontes e cordas podem passar a ser arestas hamiltonianas) e uma face do grafo deixa de existir.

Assim como em outras operações, para realizar a atualização das estruturas de dados é necessário o cálculo do caminho mínimo p_{uv} em G. Na análise de p_{uv} podem ser identificados três subcasos:

- Sequência de arestas hamiltonianas em p_{uv}. Nesse caso, todas as arestas da sequência passarão a ser pontes. Além disso, todos os vértices da sequência de arestas hamiltonianas passarão a fazer parte de uma única conexão. Se alguns desses vértices já faziam parte previamente de conexões, elas devem ser fundidas. A Figura 4.19 ilustra um caso de remoção de aresta hamiltoniana no qual o caminho p_{uv} é composto por uma sequência de arestas hamiltonianas. Nesse caso, tem-se que $p_{uv} = \{u, d, c, b, a, v\}$.

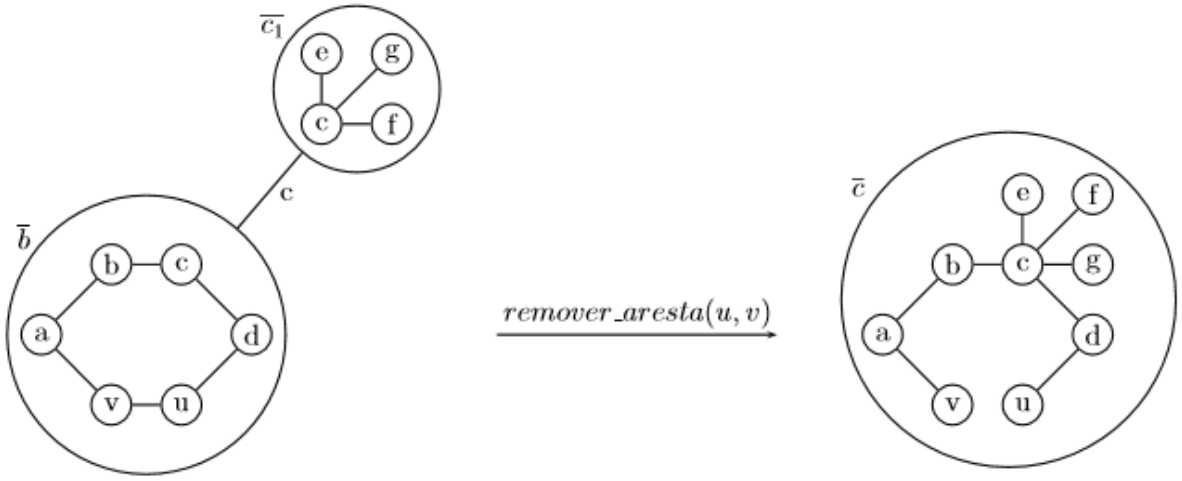


FIG. 4.19: Remoção de aresta hamiltoniana com sequência de arestas hamiltonianas em p_{uv}.

- Corda em p_{uv}. Nesse caso, a aresta passa a ser hamiltoniana, formando um novo bloco, composto por alguns dos vértices que faziam parte de $\bar{b}$. Seja a corda $(x, y), x, y, \in V, (x, y) \in p_{uv}$. O novo bloco formado pode ser encontrado realizando-se uma busca me largura a partir de x até y, percorrendo apenas arestas hamiltonianas em G. A Figura 4.20 ilustra o caso em que, após a remoção da aresta hamiltoniana (u, v), o caminho mínimo p_{uv} possui uma corda. Para o caso em questão, $p_{uv} = \{u, c, b, a, v\}$ e o caminho mínimo possui a corda (c, u).

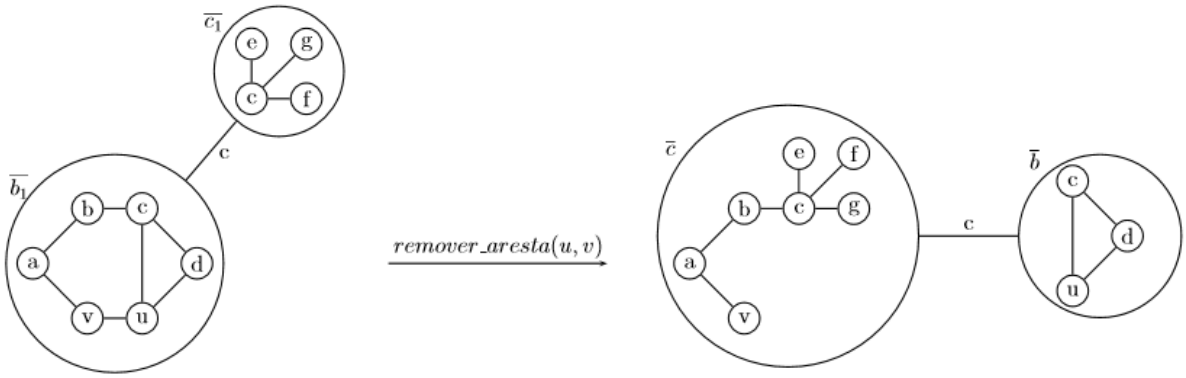


FIG. 4.20: Remoção de aresta hamiltoniana com corda em p_{uv}.

- Cordas consecutivas em p_{uv}. Nesse caso, a presença de cordas consecutivas gera um ponto de articulação em G. Isso significa que o vértice de articulação fará parte de uma conexão (é possível que o vértice já esteja previamente ligado a uma conexão, caso esse em que não é feita alteração no grafo bloco conexão para o vértice em questão). Sendo assim, caso o vértice esteja ligado ao bloco que será modificado, é necessário criar uma conexão

somente para o ponto de articulação. A Figura 4.21 exemplifica o caso em que, após a remoção da aresta hamiltoniana (u, v), o caminho mínimo p_{uv} possui cordas consecutivas. Nesse caso, $p_{uv} = \{u, c, v\}$ e as arestas (u, c) e (c, v) são cordas consecutivas no caminho mínimo.

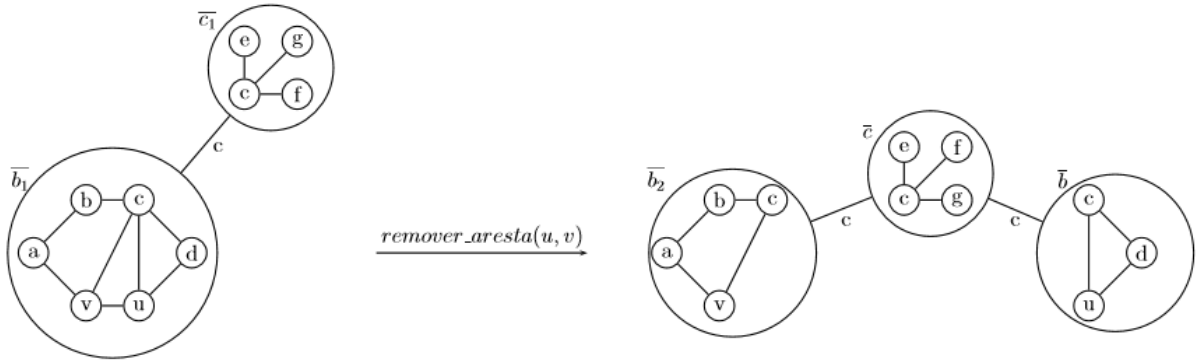


FIG. 4.21: Remoção de aresta hamiltoniana com cordas consecutivas em p_{uv}.

5 IMPLEMENTAÇÃO DO ALGORITMO

Este capítulo tem por objetivo apresentar todos os detalhes de implementação do algoritmo de manutenção da *outerplanaridade* em grafos dinâmicos discutido no capítulo anterior. Inicialmente, na Seção 5.1, será feita uma apresentação das estruturas de dados principais e auxiliares que são utilizadas no decorrer do algoritmo. Em seguida, na Seção 5.2, será feita a descrição acerca da inicialização das estruturas de dados no algoritmo de manutenção da *outerplanaridade*. Posteriormente serão apresentados os detalhes intrínsecos a cada operação, abordando inicialmente, na Seção 5.3, a operação de inserção de arestas e posteriormente, na Seção 5.4, a remoção das arestas.

Basicamente a implementação separa as estruturas em dois "níveis", guardando informações sobre o grafo G e também sobre o seu grafo bloco conexão $BC(G)$ associado. A separação das informações facilita a implementação do algoritmo.

Optou-se por utilizar a linguagem C++ para a implementação do algoritmo de manutenção da *outerplanaridade*, devido principalmente aos seguintes fatores:

- A performance é satisfatória e, em geral, a execução é mais rápida do que outras linguagens de ampla utilização (Java, Python, etc.);

- A presença de uma vasta biblioteca de templates (STL) que facilita a utilização de estruturas de dados e algoritmos amplamente conhecidos;

O algoritmo de manutenção da *outerplanaridade* tratado nesse trabalho parte de algumas premissas:

- O conjunto de vértices do grafo é fixo, isto é, não é possível inserir ou remover vértices;

- Os grafos são não direcionados (não admitindo laços e arestas múltiplas).

A entrada do algoritmo controla as premissas mencionadas anteriormente, não admitindo informações fora desse padrão.

No anexo deste trabalho é possível consultar o código em C++ implementado para o algoritmo de manutenção da *outerplanaridade*.

5.1 ESTRUTURAS DE DADOS UTILIZADAS

Nesta seção são descritas as estruturas utilizadas para a manutenção da *outerplanaridade*. Para facilitar a compreensão, apresentam-se as estruturas utilizadas para manutenção do

grafo G, as estruturas associadas ao grafo bloco conexão $BC(G)$ e as estruturas de interligação entre G e $BC(G)$.

A nível de grafo utilizam-se as seguintes estruturas:

- EG_1 Adjacências do grafo G. Vale ressaltar aqui que, devido à necessidade de alto desempenho, foi necessário utilizar uma estrutura um pouco mais complexa do que uma convencional lista de adjacências, na qual cada vértice possui uma lista encadeada com todos os vértices que possui adjacência. Nesse caso, como é preciso realizar uma série de inserções e remoções de adjacências no decorrer do algoritmo, faz-se necessária a utilização de uma estrutura que comporte essas operações numa complexidade melhor. O C++ fornece uma estrutura de template que resolve bem essa necessidade: o *unordered_set* (conjunto não ordenado). Os conjuntos não ordenados representam contêineres que armazenam elementos únicos (vértices, no caso do algoritmo de manutenção da *outerplanaridade*) em nenhuma ordem específica, que permitem rápida inserção, consulta e remoção de elementos baseado em seus valores. A implementação dessa estrutura utiliza a ideia de *hash* (com as hipóteses de colisão devidamente tratadas). É possível fornecer a função de *hash* que a estrutura utilizará, porém optou-se por utilizar a função padrão que a estrutura implementa. Na implementação do algoritmo, a estrutura utilizada é representada pela variável *adj*, que é um vetor de conjuntos não ordenados, no qual para um dado vértice v, $adj[v]$ armazena as adjacências de v.

- EG_2 Arestas hamiltonianas do grafo G. Cabe salientar aqui que, como as operações de inserção/remoção de arestas podem alterar bastante a estrutura do grafo, a informação da arestas pertecentes aos ciclos hamiltonianos podem variar com certa freqûencia. Devido a essa característica, preferiu-se utilizar a mesma estrutura das adjacências dos vértices citada anteriormente (*unordered_set*), adaptando-a para utilização de pares ordenados (arestas) ao invés de inteiros simples (vértices). Por uma questão de controle na estrutura, optou-se por padronizar a inserção de pares ordenados (u, v), de forma tal que $u < v$. Na implementação do algoritmo, as arestas hamiltonianas são armazenadas na estrutura representada pela variável *hamiltoniano*.

- EG_3 Cordas do grafo G. Nesse caso, a estrutura é semelhante àquela que armazena as arestas hamiltonianas de G. É importante observar que existe outro tipo de aresta que aparece no decorrer das operações de inserção/remoção que não é armazenado no algoritmo: as pontes. Em termos teóricos é importante ressaltar a presença desse tipo de aresta no problema de manutenção da *outerplanaridade*. Para a implementação, optou-se por não representar esse tipo de aresta por uma questão de performance, apenas. Já

que só existem 3 tipos de aresta (hamiltonianas, cordas e pontes) no algoritmo, como já são armazenados dois tipos de aresta, não se faz necessário armazenar o terceiro tipo. A estrutura que armazena as cordas é identificada pela variável *corda*.

- EG_4 Faces do grafo G. Nesse caso, armazena-se para cada vértice toda a lista de faces a que esse vértice pertence. Na execução do algoritmo, uma face é identificada por um número. A lista de faces do vértice é uma lista duplamente encadeada que deve-se manter ordenada. A necessidade de ordenação da lista também se deve a questões de performance, visto que é necessário encontrar as faces comuns a dois vértices em determinadas operações (ao invés realizar uma abordagem com complexidade quadrática para encontrar elementos comuns na lista, pode-se percorrer as listas em conjunto, visto que elas estão ordenadas). Outra observação é que existe um identificador global de faces, que é um número crescente, ou seja, caso uma face deixe de existir em algum momento e volte a ser formada em tempo futuro, essa face terá um novo identificador. A escolha por essa solução se deve ao fato de que a manutenção da invariante das listas estarem ordenadas é bastante simples, visto que para adicionar uma face a determinado vértice significa adicionar um novo número ao fim da lista encadeada. Vale ressaltar que, para efeitos de implementação, a informação da face externa não é armazenada nas estruturas do algoritmo, visto que é uma propriedade intrínseca aos grafos *outerplanares*. Portanto, o algoritmo armazena apenas informações acerca das faces internas. No algoritmo implementado, a estrutura é um vetor de listas identificado pela variável $faces$, no qual para um dado vértice v, $faces[v]$ armazena uma lista com as faces a que v pertence.

- EG_5 Componentes conexos do grafo G. Essa estrutura é um simples vetor que visa manter a informação das componentes conexas do grafo G. Sempre que uma ponte é adicionada ou removida do grafo em questão, a estrutura de conjuntos entra em ação para manutenção dessa informação. A escolha do uso de um vetor para representação da estrutura permite a rápida identificação da componente a que um vértice pertence. Vale ressaltar aqui que na implementação do algoritmo de manutenção da *outerplanaridade* incremental (suporta apenas adição de arestas), optou-se por utilizar a estrutura *Union-Find* para o armazenamento dos conjuntos. O problema na utilização dessa estrutura no algoritmo completamente dinâmico é que ela não suporta a operação de remoção de arestas. Nas duas versões do algoritmo, a estrutura é identificada pela variável *conjunto*, onde $conjunto[v]$, identifica a componenente conexa à qual v pertence.

- EG_6 Conjuntos disponíveis: essa é uma estrutura auxiliar na execução do algoritmo. Como mencionado anteriormente, o algoritmo guarda as informações acerca dos conjuntos associados a cada vértice. Inicialmente, tem-se que $conjunto[i] = i$. Caso, por

exemplo, realize-se a operação $adicionar_aresta(1, 2)$ no grafo G, o algoritmo unirá os dois vértices a um único conjunto, por exemplo o conjunto 1, enquanto o conjunto 2 nesse momento se tornará disponível para utilização. Essa estrutura é uma pilha que, conforme o passar do tempo, armazena os conjuntos que não estão sendo utilizados no momento atual de execução do algoritmo. Assim, quando for realizada uma operação de remoção de uma ponte, pode-se atribuir um conjunto disponível a uma das componentes conexas recém formada. Por questão de comodidade, o algoritmo mantém a identificação anterior de conjunto à outra componente conexa. Os conjuntos disponíveis são identificados no algoritmo pela variável *conjuntos*.

- EG_7 Predecessor dos vértices: estrutura auxiliar que armazena o vértice predecessor (ou pai) em uma árvore geradora. Mais especificamente, o algoritmo de manutenção da *outerplanaridade* necessita calcular caminhos mínimos durante sua execução. Sendo assim, optou-se, por questões de simplicidade, pela utilização do algoritmo de busca em largura (BFS) para o cálculo dos caminhos mínimos. O BFS, durante sua execução, gera uma árvore geradora em largura do grafo G. A estrutura de predecessores dos vértices é útil para percorrer os vértices do caminho mínimo. Os predecessores são armazenados em um vetor denotado *pai*, onde $pai[v]$ denota o predecessor do vértice v.

- EG_8 Vértices visitados no BFS: estrutura auxiliar que armazena a informação que um vértice já foi percorrido ou não na execução do algoritmo de busca em largura. É uma estrutura que permite que a busca controle durante a sua execução a informação de quais vértices já foram explorados em um dado momento. A estrutura de vértices visitados é um vetor identificado pela variável *vis*, onde $vis[v] = 0$ indica que o vértice v ainda não foi explorado e $vis[v] = 1$, caso contrário.

A seguir são apresentadas as estruturas que mantém a interligação entre G e $BC(G)$:

- EI_1 Índices: essa é uma estrutura importante no sentido de que representa uma ligação entre o grafo G e seu respectivo grafo bloco conexão, $BC(G)$. A estrutura de índices é um simples vetor que armazena a qual bloco ou conexão em $BC(G)$ o vértice $v \in G$ está ligado. Posteriormente será apresentada uma estrutura ($EBC(G)_1$) que permite identificar blocos e conexões em $BC(G)$ usando pares ordenados. Portanto, o vetor de índices armazena pares ordenados. Os índices são armazenados no vetor *indice* no algoritmo.

- EI_2 Índices revertidos: essa é uma estrutura auxiliar na execução do algoritmo e visa responder com maior eficiência à pergunta: quais os vértices pertencentes a determinado bloco ou conexão? Algumas operações no algoritmo pedem essa resposta eficientemente.

Para representar essa estrutura, utilizam-se duas estruturas aninhadas. Uma delas é a *unordered_set*, já mencionada anteriormente, que armazena os vértices ligados a determinado bloco ou conexão. A outra estrutura é uma mapeamento não ordenado, em C++ *unordered_map*, com uma implementação semelhante ao *unordered_set*, baseando em funções de *hash*, porém essa estrutura é um contêiner associativo que armazena elementos formados pela combinação de uma chave e um valor mapeado, e que permite um rápido retorno dos elementos baseados em suas chaves. No caso do algoritmo de manutenção da *outerplanaridade*, a chave da estrutura seria o par ordenado que identifica o bloco ou conexão e o valor associado é o conjunto não ordenado de vértices ligados a esse bloco ou conexão. Em termos de implementação, a estrutura seria instanciada como $unordered_map < pair < int, int >, unordered_set < int >> indice_rev$.

A nível de grafo bloco conexão são utilizadas as estruturas a seguir:

- $EBC(G)_1$ Blocos e conexões: para modelar a representação de cada bloco e cada conexão utiliza-se um par ordenado. O primeiro elemento do par é uma constante que identifica um bloco ou uma conexão (padronizou-se no algoritmo utilizar a constante 0 para representar um vértice do tipo bloco e a constante 1 para a representação de uma conexão). O segundo elemento do par ordenado é um inteiro identificador do bloco ou da conexão. Aqui utiliza-se o fato de que o limite superior para os blocos e conexões é o próprio número de vértices do grafo. Assim, ao se formar um novo bloco, por exemplo, atribui-se a ele um identificador no intervalo $[1, n]$. Desse modo, a representação de uma conexão, com identificador 6, por exemplo, é o par ordenado $(1, 6)$, enquanto a do bloco com identificação de número 2 é o par $(0, 2)$. Na implementação do algoritmo, para um dado vértice $\bar{v}$, $\bar{v}.first$ indica o tipo do vértice (bloco ou conexão) e $\bar{v}.second$ armazena o identificador do bloco ou conexão.

- $EBC(G)_2$ Adjacências do grafo bloco conexão: tendo definido uma representação para os blocos e as conexões pode-se elaborar uma estrutura de dados que armazena as adjacências do grafo $BC(G)$. A estrutura final é um conjunto não ordenado (*unordered_set*), semelhante ao utilizado nas adjacências do grafo G, porém modificada para armazenar mais informação. Os elementos dos conjuntos não ordenados (um para cada bloco ou conexão) são pares ordenados, onde o primeiro elemento do par representa o atributo da aresta (vértice que interliga o bloco à conexão e vice-versa). O segundo elemento do par é um par ordenado que identifica o bloco ou conexão ao qual o vértice se liga. Portanto, a estrutura final é do tipo $unordered_set < pair < int, pair < int, int >>>$. No algoritmo a estrutura é identificada pela variável *bc*, onde $bc[tipo][\bar{v}]$ identifica o conjunto

não ordenado relativo às adjacências do vértice $\bar{v}$, que é do *tipo* bloco (0) ou conexão (1).

- $EBC(G)_3$ Blocos disponíveis: de forma semelhante à estrutura que guarda os conjuntos disponíveis a nível de grafo, essa estrutura de dados é uma pilha que armazena os identificadores de blocos que não estão sendo utilizados no momento. Como mencionado anteriormente, utiliza-se o limite superior de número de vértices no grafo G para o número de blocos. Sendo assim, caso em algum instante de tempo tenha-se um grafo vazio, por exemplo, a pilha de blocos disponíveis conterá os n identificadores de blocos que foram reservados. Os blocos disponíveis são armazenados na variável *bloco*.

- $EBC(G)_4$ Conexões disponíveis: assim como se faz necessário armazenar os blocos disponíveis, também é importante controlar as conexões que não estão sendo utilizadas. A estrutura é semelhante à pilha utilizada para os blocos disponíveis. As conexões disponíveis são armazenadas na variável *conexao*.

- $EBC(G)_5$ Predecessor dos vértices do grafo bloco conexão: durante a execução do algoritmo também é necessário calcular caminhos mínimos no grafo bloco conexão. Devido à propriedade de que o grafo $BC(G)$ é uma árvore, tem-se que esse caminho é único (se existir). Sendo assim, optou-se, por questões de simplicidade, em calcular o caminho utilizando-se uma busca em profundidade (DFS). Na execução do DFS, é gerada uma árvore em profundidade e, para realizar o percurso nessa árvore, a fim de percorrer os caminhos no grafo bloco conexão faz-se necessário utilizar a estrutura auxiliar de predecessores, com ideia semelhante à utilizada a nível de grafo. Como o grafo $BC(G)$ possui algumas informações adicionais, deve-se estender a estrutura. Sendo assim, a informação do predecessor de um dado vértice é um par ordenado, onde o primeiro elemento é o atributo de aresta (que representa a aresta que foi utilizada para chegar ao vértice atual). O segundo elemento é um par ordenado que representa o bloco ou conexão que se passou para a chegada no vértice atual. A estrutura final é do tipo $pair < int, < int, int >>$. No algoritmo, a estrutura que armazena os predecessores no grafo bloco conexão é identificada pela variável *pai_bc*, onde $pai_bc[tipo][\bar{v}]$ contém o par (atributo de aresta, bloco/conexão) que é o predecessor no DFS do vértice $\bar{v}$, cujo *tipo* é bloco (0) ou conexão (1).

Com as estruturas de dados apresentadas, será possível descrever os detalhes de implementação das operações de inserção ou remoção de arestas.

5.2 INICIALIZAÇÃO DAS ESTRUTURAS DE DADOS

Ao iniciar sua execução, o algoritmo deve ter algumas de suas estruturas de dados inicializadas para que as informações futuras possam ser corretamente armazenadas. A fim de que as estruturas de dados sejam iniciadas, é necessário passar a informação como entrada do algoritmo, da quantidade de vértices, n, do grafo $G = (V, E)$. Como mencionado anteriormente, o conjunto de vértices do grafo é fixo durante toda a execução do algoritmo. O algoritmo, por padrão, executa a partir do grafo inicial vazio, com n vértices. Sendo assim, tem-se que o grafo bloco conexão $BC(G)$ inicialmente possuirá n conexões, situação na qual cada vértice do grafo G estará ligado a uma conexão distinta em $BC(G)$. Vale ressaltar que o algoritmo suporta receber um grafo *outerplanar* não vazio de entrada (após a informação do número de vértices do grafo G), porém será necessário realizar as operações de inserção de aresta a partir do grafo vazio até que sejam inseridas todas as arestas do grafo de entrada.

Após a informação do número de vértices do grafo G, o algoritmo de manutenção da *outerplanaridade* inicializa as seguintes estruturas de dados:

- *conjunto*. Como o grafo G é inicialmente vazio, cada vértice pertencerá a uma componente conexa distinta. Sendo assim, para cada vértice v de G, $conjunto[v] = v$;

- *indice*. Como mencionado anteriormente, cada vértice estará ligado a uma conexão distinta, visto que o grafo inicial é vazio. Como as conexões são identificadas por pares ordenados do tipo $(1, identificador)$, tem-se inicialmente que, para cada vértice de v, $indice[v] = (1, v)$;

- *indice_rev*. Já que a estrutura de índices deve ser inicializada, os índices revertidos também o são, consequentemente. Se, para um dado vértice v, $indice[v] = (1, v)$, então para o índice revertido da conexão $(1, v)$, $indice_rev[(1, v)]$, deve-se adicionar o vértice v;

- *bloco*. Como o grafo bloco conexão é composto inicialmente por conexões, os blocos devem ser armazenados na pilha de blocos disponíveis. O algoritmo então adiciona os blocos com identificadores no intervalo $[1, n]$ na pilha.

O algoritmo processa entradas do tipo (id, u, v), $u, v \in V$ e id é um identificador do tipo da operação a ser realizada, sendo o valor 1 para inserção de aresta e 2 para remoção. Após a inicialização das estruturas, o algoritmo prossegue podendo receber um grafo *outerplanar* como entrada ou processando as operações de inserção ou remoção de aresta. Caso o algoritmo receba um grafo *outerplanar* como entrada, as arestas são inseridas uma a uma, de acordo com procedimentos que serão descritos na Seção 5.3.

5.3 INSERÇÃO DE ARESTA

Conforme visto na Seção 4.2, a inserção de arestas no grafo pode ser dividida em três casos, a depender do tipo de aresta que será inserida:

- Adição de ponte, que realizará a união de componentes conexas distintas;

- Inserção de aresta hamiltoniana, que será responsável pela formação de novos blocos no grafo, caso seja possível;

- Adição de corda em um subgrafo que possui um ciclo hamiltoniano, que ligará vértices diretamente no ciclo, caso a adição não invalide a *outerplanaridade*.

Desse modo, é possível identificar em que caso a inserção de aresta se encaixa e realizar as devidas operações para a atualização das informações a nível de grafo e a nível de grafo bloco conexão. As implementações das operações de verificação dos casos também serão descritas nas subseções específicas.

Ao finalizar todos os casos de inserção de aresta, na Subseção 5.3.4, será apresentado o pseudocódigo correspondente.

5.3.1 ADIÇÃO DE PONTE

A verificação se a aresta a ser inserida é uma ponte deve ser a primeira a ser realizada, visto ser uma operação relativamente simples. Caso contrário deve-se analisar os outros dois casos, o que requer uma análise um pouco mais complexa.

Para verificar se a aresta (u, v) a ser inserida é uma ponte, deve-se consultar a estrutura de componentes conexos do grafo G. Dessa forma, é fácil determinar a quais componentes conexos os vértices u e v pertencem. Caso os dois vértices pertençam a componentes conexos distintos, tem-se que a aresta (u, v) é uma ponte e está conectando componentes conexas distintas.

Vale ressaltar que no caso de trabalhar unicamente com a versão incremental do algoritmo, pode ser utilizada a estrutura Union-Find. O uso do Union-Find não é possível no algoritmo completamente dinâmico, visto que a estrutura não suporta eficientemente operações de remoção de arestas.

Uma vez concluída a análise de que a aresta a ser adicionada é uma ponte, deve-se atualizar as estruturas associadas. No caso de inserção de pontes não há necessidade de preocupação com a invalidação da *outerplanaridade* pois a adição desse tipo de aresta sempre é possível.

A nível de grafo, realizam-se a seguintes operações:

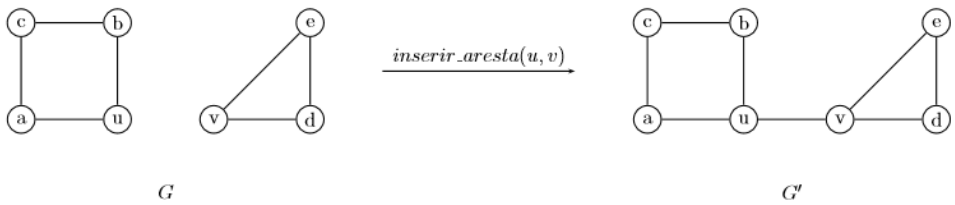
- Atualizar as adjacências do grafo G (adj), inserindo o vértice v às adjacências de u e vice-versa;

- Atualização dos componentes conexos do grafo G. Nesse caso, escolhe-se um vértice, por exemplo v, e modificam-se todos os vértices tais que $conjunto[i] = conjunto[v], i \in V$ para o novo valor $conjunto[u]$ (essa operação é linear, pois como a estrutura é um vetor, só será necessário procurar pelos vértices que pertencem ao mesmo componente conexo de v para fazer a atualização);

- Atualização da estrutura de conjuntos disponíveis, $conjuntos$. Como realizou-se a transferência de todos os vértices que possuíam o valor $conjunto[v]$, tem-se que esse identificador agora não é mais utilizado e agora se tornou disponível, logo ele é adicionado à estrutura para que possa ser utilizado futuramente.

A nível de grafo bloco conexão deve-se identificar a quais tipos de vértice u e v estão ligados em $BC(G)$. Existem três possibilidades de pares de vértices $\bar{u}$ e $\bar{v}$ em $BC(G)$ aos quais os vértices u e v são ligados, respectivamente:

- $\bar{u}$ e $\bar{v}$ são blocos. Nesse caso deve-se criar um novo vértice no grafo $BC(G)$ que será uma nova conexão que conterá os vértices u e v e a aresta recém inserida (u, v). Para fazer isso, deve-se procurar uma conexão livre (que pode ser obtida a partir da pilha de conexões disponíveis, $conexao$). Além disso, deve-se criar as adjacências referentes à nova conexão, denotada $\bar{c}$, e os dois blocos, incluindo os atributos de aresta, que serão u para a aresta $(\bar{c}, \bar{u})$ e v para a aresta $(\bar{c}, \bar{v})$ em $BC(G)$. Finalmente, deve-se atualizar as estruturas de índices e índices revertidos. Como u e v pertencerão à nova conexão $\bar{c}$, a estrutura de índices deve ser atualizada, ou seja, $indice[u] = \bar{c}$ e $indice[v] = \bar{c}$. Vale mencionar que sempre que algum índice for alterado e necessário atualizar a estrutura de índices revertidos para manutenção da validade das informações. Sendo assim, caso o vértice u tenha seu índice modificado, deve-se remover o valor antigo de $indice[u]$ no indice revertido associado, ou seja, $indice_rev[indice[u]]$, e adicionar o novo valor em seu índice revertido associado. A Figura 5.1 ilustra uma situação simples de adição de ponte neste caso;

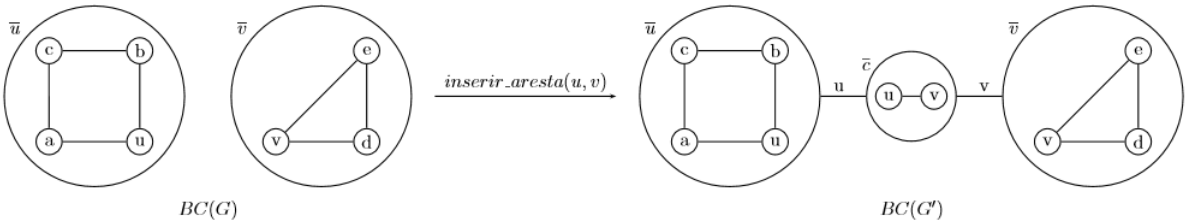


(a) Grafos G e G'

G	adj
a	c, u
b	u, c
c	a, b
d	v, e
e	d, v
u	b, a
v	d, e

G'	adj
a	c, u
b	u, c
c	a, b
d	v, e
e	d, v
u	b, v, a
v	d, e, u

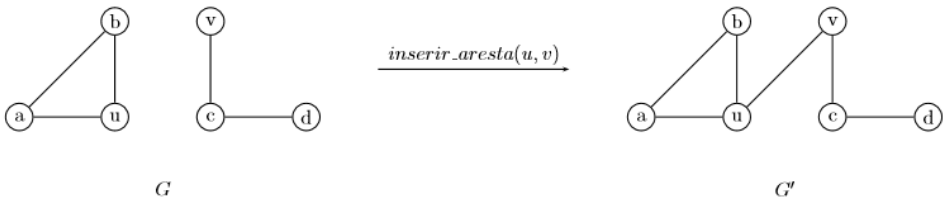
(b) Adjacências de G e G'



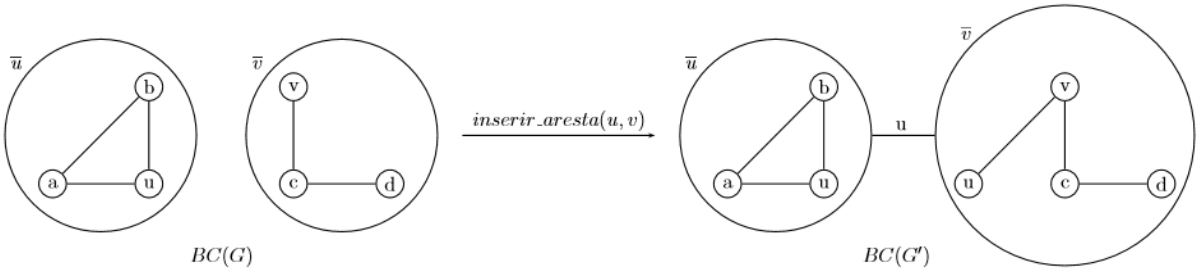
(c) Grafos $BC(G)$ e $BC(G')$

FIG. 5.1: Adição de ponte no caso em que dois blocos são conectados.

- $\bar{u}$ é bloco e $\bar{v}$ é conexão (o caso contrário é semelhante e não será analisado). Nesse caso, não há a necessidade de criação de um novo vértice para $BC(G)$. A única mudança em $BC(G)$ é que u deixa de ser ligado a $\bar{u}$ e passa a ser ligado a $\bar{v}$. Portanto, deve-se atualizar $indice[u]$ (e consequentemente os índices revertidos), apontando para $\bar{v}$. Além disso, deve-se criar a nova adjacência que liga o bloco e a conexão. O atributo de aresta é o vértice u, que será comum a ambos. A Figura 5.2 ilustra a operação de adição de ponte para esse caso;



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

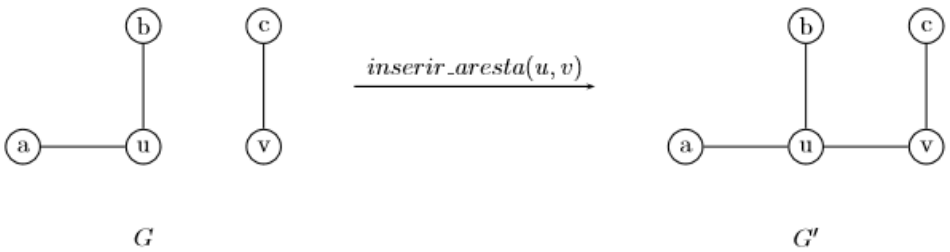
$BC(G)$	bc
$\bar{u}$	-
$\bar{v}$	-

$BC(G')$	bc
$\bar{u}$	(u, $\bar{v}$)
$\bar{v}$	(u, $\bar{u}$)

(c) Adjacências de $BC(G)$ e $BC(G')$

FIG. 5.2: Adição de ponte no caso em que um bloco e uma conexão são conectados.

- $\bar{u}$ e $\bar{v}$ são conexões. Nesse caso também não há necessidade da criação de um novo vértice em $BC(G)$, visto que a união entre conexões gera uma nova conexão. Porém, deve haver a preocupação em atualizar todas as informações, já que uma das conexões deixará de existir ao realizar a adição da aresta. Suponha, sem perda de generalidade, que a conexão $\bar{v}$ será descartada ao fim da operação de inserção. Sendo asssim, deve-se atualizar as informações envolvendo todos os vértices em G que estão ligados a $\bar{v}$. Para realizar a atualização desses vértices (que passam a ser ligados a $\bar{u}$) pode-se utilizar a estrutura de índices revertidos (*indice_ rev[indice[v]]* contém todos os vértices ligados a $\bar{v}$). Uma vez atualizadas as informações de índice, deve-se adicionar $\bar{v}$ a *conexao* (pilha com conexões disponíveis), permitindo que a conexão seja utilizada futuramente. Finalmente, precisa-se atualizar as adjacências do grafo bloco conexão. Deve-se percorrer todas as adjacências de $\bar{v}$ e atualizá-las para $\bar{u}$. Além disso, é necessário remover as adjacências de $\bar{v}$, visto que esse vértice deixou de existir em $BC(G)$. A Figura 5.3 ilustra um caso simples de adição de ponte entre conexões.

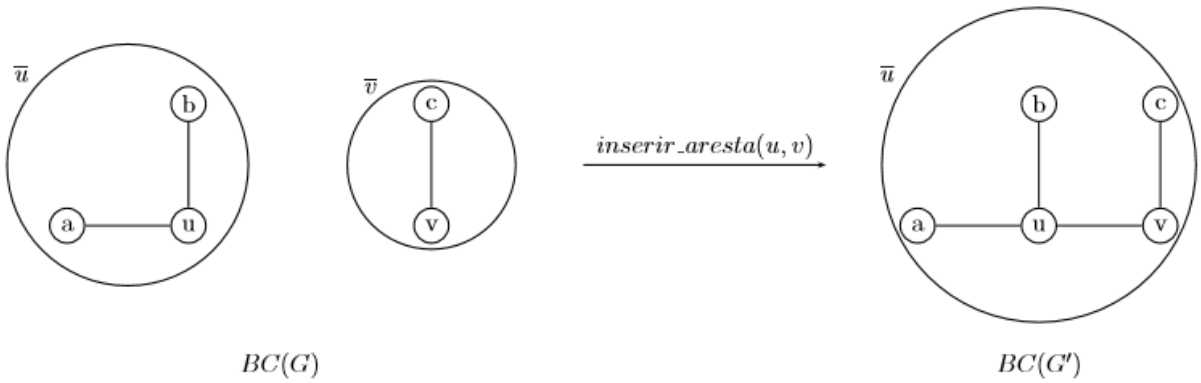


(a) Grafos G e G'

G	*conjunto*
a	1
b	1
c	2
u	1
v	2

G'	*conjunto*
a	1
b	1
c	1
u	1
v	1

(b) Componentes conexos de G e G'



(c) Grafos $BC(G)$ e $BC(G')$

FIG. 5.3: Adição de ponte no caso em que duas conexões são conectadas.

5.3.2 ADIÇÃO DE CORDA

Caso u e v pertençam à mesma componente conexa, deve-se descobrir se a aresta a ser adicionada é de tipo hamiltoniana ou corda. Primeiramente é calculado o caminho mínimo em $BC(G)$ entre os vértices que ligam u e v, ou seja, $\bar{u}$ e $\bar{v}$, respectivamente. Como mencionado anteriormente, utilizando a propriedade de que o grafo bloco conexão é uma árvore, calcula-se o caminho mínimo utilizando uma busca em profundidade (DFS). O vetor de vértices predecessores do grafo bloco conexão é responsável pela reconstrução dos caminhos entre os vértices na árvore de profundidade. A DFS retorna uma informação de grande relevância para a análise do tipo da aresta a ser inserido: a distância entre o vértice de origem e o vértice de destino da busca ($\bar{u}$ e $\bar{v}$, respectivamente). Existem três valores dessa distância que correspondem à adição de uma corda, a citar (com os respectivos detalhes de implementação):

- A distância entre $\bar{u}$ e $\bar{v}$ é igual a zero (ou seja, u e v estão ligados ao mesmo vértice em $BC(G)$) e $\bar{u}$ é bloco. A verificação dessa informação é simples: basta verificar se $\bar{u}$ (que pode ser obtido a partir da informação *indice[u]*) é um bloco;

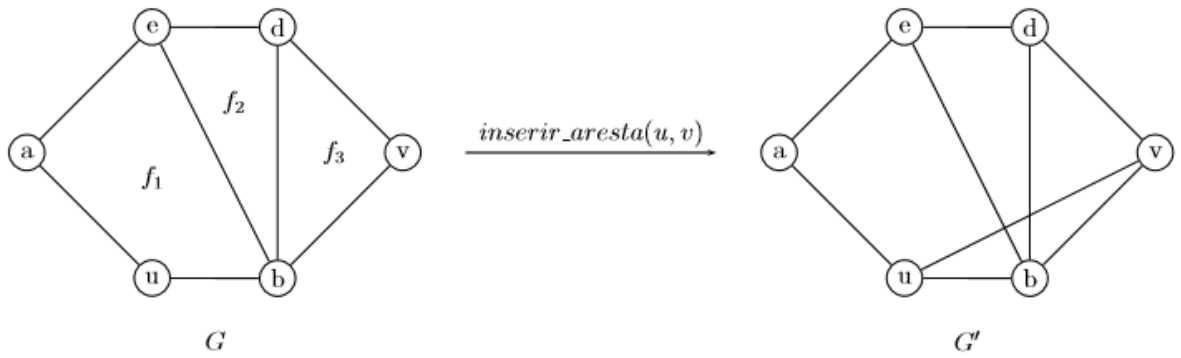
- A distância entre $\bar{u}$ e $\bar{v}$ é igual a um (ou seja, o caminho entre os vértices é $p_{\bar{u}\bar{v}} = (\bar{u}, \bar{v})$) e o atributo da única aresta no caminho é u ou v. Essa informação pode ser obtida a partir dos predecessores obtidos pela DFS. O predecessor associado a $\bar{v}$ é $\bar{u}$, por ser o comprimento do caminho entre eles igual a um neste caso. Portanto, será suficiente consultar qual o atributo de aresta armazenado $pai_bc[\bar{v}.first][\bar{v}.second]$;

- A distância entre $\bar{u}$ e $\bar{v}$ é igual a dois. Nesse caso, há um caminho da forma: $p_{\bar{u}\bar{v}} = (\bar{u}, \bar{x}, \bar{v})$. Então deve-se obter os dois atributos de aresta, de forma semelhante à comentada anteriormente (utilizando os predecessores obtidos pela DFS). Sejam os atributos: $atr_{\bar{u}\bar{x}}$, o atributo de aresta relativo à ligação $(\bar{u}, \bar{x})$, e $atr_{\bar{x}\bar{v}}$, o atributo de aresta relativo à ligação $(\bar{x}, \bar{v})$. A aresta a ser adicionada é uma corda caso $atr_{\bar{u}\bar{x}} = u$ e $atr_{\bar{x}\bar{v}} = v$ ou então $atr_{\bar{u}\bar{x}} = v$ e $atr_{\bar{x}\bar{v}} = u$.

Concluídos os detalhes de implementação dos casos em que se encaixa a adição de uma corda em G, prosseguir-se-á com a manutenção das estruturas ao fazer a adição dessa aresta em G. Diferentemente do caso em que é adicionada uma ponte, a adição de uma corda pode não ser possível, se essa adição invalida a *outerplanaridade*. Nos casos em que isso acontece, o algoritmo determina que é invalida a inserção e informa ao usuário que a adição não é possível pois invalida a propriedade da *outerplanaridade*, pois produziria cruzamento de arestas.

A primeira verificação na adição da corda é se a mesma invalida ou não a *outerplanaridade*. A adição da aresta é impossível caso os vértices u e v não possuam faces em comum (exceto a face externa). Para tal fim, foi implementada a função *encontrar_face_comum*, que recebe três parâmetros: o vértice u, o vértice v e uma posição i, e retorna a i-ésima face comum aos vértices u e v. Por exemplo, seja a lista de faces de u, $faces[u] = [2, 5, 8, 10]$ (como já mencionado, a lista de faces deve manter a invariante de que está sempre ordenada), e a lista de faces de v, $faces[v] = [5, 7, 8]$. Portanto, $encontrar_face_comum(u, v, 1)$ retorna 5, enquanto $encontrar_face_comum(u, v, 2)$ retorna 8. Caso a função não encontre a i-ésima face comum, o valor 0 é retornado. Vale lembrar aqui que, para o caso dos grafos *outerplanares*, a quantidade máxima de faces comuns para um par de vértices é dois. A implementação desse método baseia-se na invariante de que as listas de face estão ordenadas, garantindo uma complexidade linear para a implementação do método. Basta manter um ponteiro apontando para uma posição da lista de faces de u e outro ponteiro que aponta para uma posição da lista de faces de

v. Inicialmente os ponteiros apontam para os elementos iniciais de cada lista. Para o elemento da lista de u apontado pelo ponteiro, caso seu valor seja inferior ao apontado pelo ponteiro de $faces[v]$, incrementa-se a posição do ponteiro de $faces[u]$, e vice-versa. Caso os valores sejam iguais, encontrou-se uma face comum. O algoritmo prossegue até encontrar a i-ésima face em comum ou então chegar ao fim de uma das listas de face, caso no qual a face não foi encontrada. A Figura 5.4 exemplifica um caso simples em que a adição de uma corda é impossível. Nessa situação, a adição da aresta (u, v) invalida a *outerplanaridade*, visto que a lista de faces de u ($\{f_1\}$) não possui interseção com a lista de faces de v ($\{f_3\}$).



(a) Grafos G e G'

G	$faces$
a	f_1
b	f_1, f_2, f_3
d	f_2, f_3
e	f_1, f_2
u	f_1
v	f_3

(b) Lista de faces de G

FIG. 5.4: Adição inválida de uma corda.

Supondo que a função encontre uma face comum, a adição da corda é possível. Sendo assim, a estrutura de faces a nível de grafo G deve ser atualizada. É interessante observar que a adição da corda não altera o grafo bloco conexão. Para realizar as atualizações, deve-se calcular o caminho mínimo entre os vértices u e v. Esse caminho deve existir, visto que u e v devem pertencer ao mesmo bloco para que essa operação esteja sendo realizada. O caminho mínimo é calculado, conforme mencionado anteriormente, com uma busca em largura (BFS), utilizando-se como estruturas auxiliares o vetor vis de vértices visitados, que controla os vértices já explorados na busca, e o vetor pai de predecessores dos vértices, que ajuda a percorrer o caminho entre o vértice de origem, u, e o vértice de

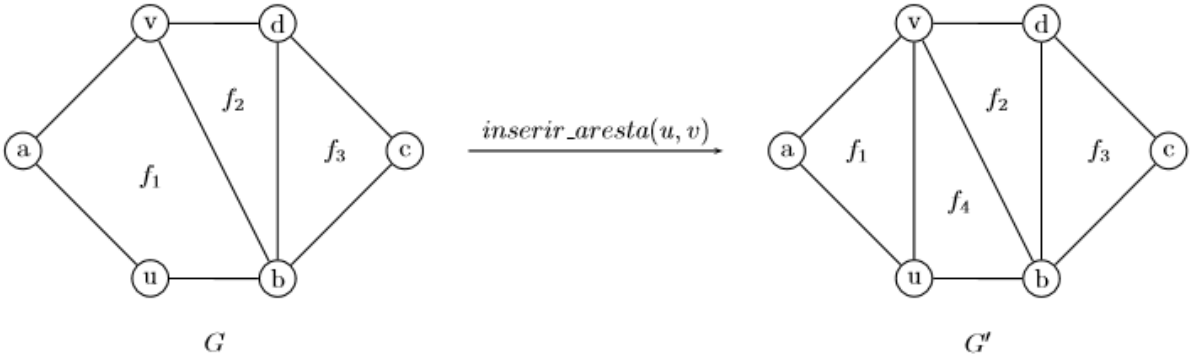
destino, v, a partir da árvore geradora em largura.

Calculado o caminho mínimo, p_{uv}, deve-se percorrer os vértices de p_{uv}. Duas operações são realizadas:

- Para todos os vértices de p_{uv} insere-se a nova face ao final da lista (que conforme mencionado anteriormente é um identificador crescente e que mantém a invariante de ordenação da lista duplamente encadeada das faces).

- Para todos os vértices de p_{uv}, exceto u e v, remove-se a face comum da lista de faces desses vértices. A retirada da face é uma remoção comum numa lista duplamente encadeada.

A Figura 5.5 ilustra uma adição válida de uma corda em um grafo. Nesse caso, u e v possuem a face f_1 em comum. Suponha, sem perda de generalidade, que o caminho mínimo calculado pelo algoritmo é $p_{uv} = \{u, b, v\}$. Então deve-se adicionar a nova face, f_4, ao fim da lista de faces de cada um dos vértices do p_{uv}, além de remover a face f_1 dos vértices intermediários de p_{uv}, no caso somente b.



(a) Grafos G e G'

G	$faces$
a	f_1
b	f_1, f_2, f_3
c	f_3
d	f_2, f_3
u	f_1
v	f_1, f_2

G'	$faces$
a	f_1
b	f_2, f_3, f_4
c	f_3
d	f_2, f_3
u	f_1, f_4
v	f_1, f_2, f_4

(b) Lista de faces de G e G'

FIG. 5.5: Adição válida de uma corda.

5.3.3 ADIÇÃO DE ARESTA HAMILTONIANA

Como existem apenas três tipos de arestas na manutenção de grafos *outerplanares* (pontes, arestas hamiltonianas e cordas), o único caso que falta abordar é o da adição de aresta hamiltoniana. Deve-se observar que a análise de cordas e arestas hamiltonianas é iniciada

em conjunto, sabendo que os vértices u e v pertencem a mesma componente conexa, e tendo já calculado o caminho mínimo entre os vértices correspondentes $\bar{u}$ e $\bar{v}$ em $BC(G)$, $p_{\bar{u},\bar{v}}$, e a distância ente $\bar{u}$ e $\bar{v}$. Existem quatro casos para os quais é possível adicionar uma aresta hamiltoniana, a citar (com os respectivos detalhes de implementação):

- A distância entre $\bar{u}$ e $\bar{v}$ é igual a zero (ou seja, u e v estão ligados ao mesmo vértice em $BC(G)$) e $\bar{u}$ é conexão. A verificação dessa informação é simples: basta verificar se $\bar{u}$ (que pode ser obtido a partir da informação *indice[u]*) é uma conexão;

- A distância entre $\bar{u}$ e $\bar{v}$ é igual a um (ou seja, o caminho entre os vértices é $p_{\bar{u}\bar{v}} = (\bar{u}, \bar{v})$) e o atributo de aresta não é u e não é v. Essa informação pode ser obtida a partir da informação dos predecessores. O predecessor associado a $\bar{v}$ é $\bar{u}$, conforme visto em $p_{\bar{u}\bar{v}}$. Com isso, basta consultar qual o atributo de aresta armazenado no predecessor;

- A distância entre $\bar{u}$ e $\bar{v}$ é igual a dois. Nesse caso, há um caminho da forma: $p_{\bar{u}\bar{v}} = (\bar{u}, \bar{x}, \bar{v})$. Então deve-se obter os dois atributos de aresta, de forma semelhante à comentada anteriormente (utilizando os predecessores). Sejam os atributos: $atr_{\bar{u}\bar{x}}$, o atributo de aresta relativo à ligação $(\bar{u}, \bar{x})$, e $atr_{\bar{x}\bar{v}}$, o atributo de aresta relativo à ligação $(\bar{x}, \bar{v})$. A aresta a ser adicionada é hamiltoniana caso $atr_{\bar{u}\bar{x}} \neq u$ ou $atr_{\bar{x}\bar{v}} \neq v$ e também $atr_{\bar{u}\bar{x}} \neq v$ ou $atr_{\bar{x}\bar{v}} \neq u$.

- A distância entre $\bar{u}$ e $\bar{v}$ é superior a dois.

Do mesmo modo feito na adição da corda, prossegue-se com a especificação da manutenção das estruturas ao fazer a adição dessa aresta em G. De forma semelhante à inserção de cordas, a adição de uma aresta hamiltoniana pode não ser possível, invalidando a *outerplanaridade*. Nos casos em que isso acontece, o algoritmo determina que a inserção é inválida e informa ao usuário que a adição não é possível pois invalida a propriedade da *outerplanaridade*, visto que um ou mais vértices não pertencerão à face externa.

A atualização das estruturas de dados afetadas pela inserção de uma aresta hamiltoniana pode ser dividida em dois casos: caso a), e mais simples, u e v são ligados a um mesmo vértice conexão em $BC(G)$; caso b), u e v são ligados a vértices distintos no grafo bloco conexão.

Inicialmente, para ambos os casos, deve-se calcular o caminho mínimo, em G, a partir do vértice de origem, u, até o vértice de destino, v. Como já mencionado, o cálculo é feito utilizando uma busca em largura e a reconstrução do caminho pode ser feita utilizando o vetor de predecessores.

Para o caso a), não há necessidade de preocupação com a invalidação da *outerplanaridade*. Isso porque está sendo adicionada uma aresta em uma conexão, o que significa que uma aresta é inserida a um subgrafo que é uma árvore, ou seja, um subgrafo dessa

árvore unido da nova aresta (u, v) irá fechar um novo ciclo (formando um novo bloco). Para verificar se o algoritmo está lidando com esse tipo de situação, basta analisar se os vértices ligados a u e v (através de $indice[u]$ e $indice[v]$) representam a mesma conexão.

Note que, para o caso da inserção de uma aresta dentro de uma conexão, deve-se separar o novo bloco que será criado e as possíveis conexões menores que resultarão após a separação, além da preservação das ligações que existiam na conexão antiga (que será desmembrada) com outros blocos em $BC(G)$.

A ideia inicial é manter em uma estrutura temporária, *temp* (*unordered_set*, por questões de performance), que armazenará todos os atributos de aresta das adjacências de $\bar{u}$ (que é obtida apenas percorrendo a estrutura de adjacências a nível de grafo bloco conexão). Em seguida, como será formado um novo bloco, deve-se identificar um bloco disponível, denotado $\bar{b}$, na pilha de blocos disponíveis, *bloco*.

Após essa etapa inicial, realiza-se a atualização das estruturas de índices e índices revertidos dos vértices do caminho p_{uv} em G, atribuindo ao índice desses vértices o bloco $\bar{b}$. Note que essa atualização ainda pode ser modificada, pois caso algum vértice de p_{uv} possua alguma adjacência para um vértice que não faça parte do novo bloco, ele constituirá um ponto de articulação e, portanto, esse vértice será ligado a uma conexão e não a $\bar{b}$. Logo, essa atualização pode ser temporária, fato que é descoberto no decorrer da função de criação de um novo bloco. O percurso dos vértices em p_{uv} é feito utilizando-se o vetor *pai*, que foi preenchido previamente no BFS.

Novamente, deve-se explorar os vértices do caminho p_{uv} para realizar a geração das novas conexões (se existirem). Em paralelo, realiza-se a atualização de duas informações a nível de grafo:

- Para cada vértice de p_{uv} deve-se adicionar a nova face, visto que está sendo formado um novo bloco, ao fim da lista de faces dos respectivos vértices;

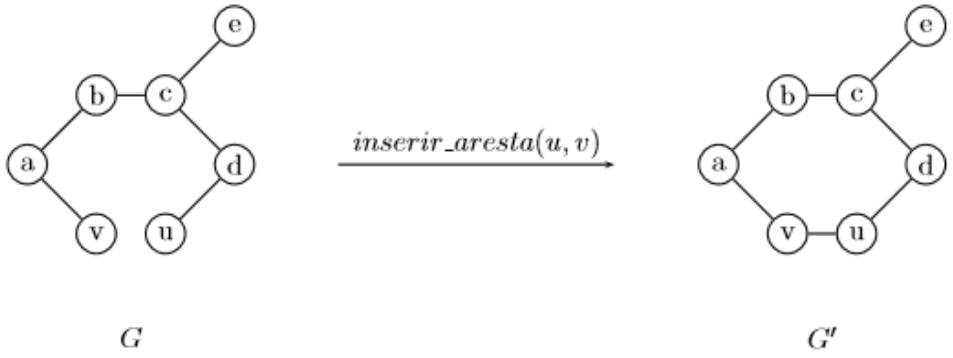
- Para cada aresta percorrida em p_{uv}, adiciona-se essa aresta à estrutura *hamiltoniano*. Como a aresta (u, v) está sendo inserida em um subgrafo que respresenta uma conexão, e portanto todas arestas são pontes, não há preocupação em retirar essas arestas da estrutura *corda*, mas caso estivesse sendo utilizada uma estrutura para armazenamento de pontes, as arestas deveriam ser removidas desta.

Durante o percurso do caminho p_{uv}, seja x o vértice atual sendo analisado. Para gerar uma possível nova conexão, deve-se explorar o vértice x. Isso é feito executando um BFS auxiliar, cuja função é explorar apenas os vértices pertencentes à conexão antiga (que está sendo modificada) e que não estejam presentes em p_{uv}, a partir da origem x. O resultado dessa busca em largura auxiliar é uma estrutura de dados temporária, *temp*

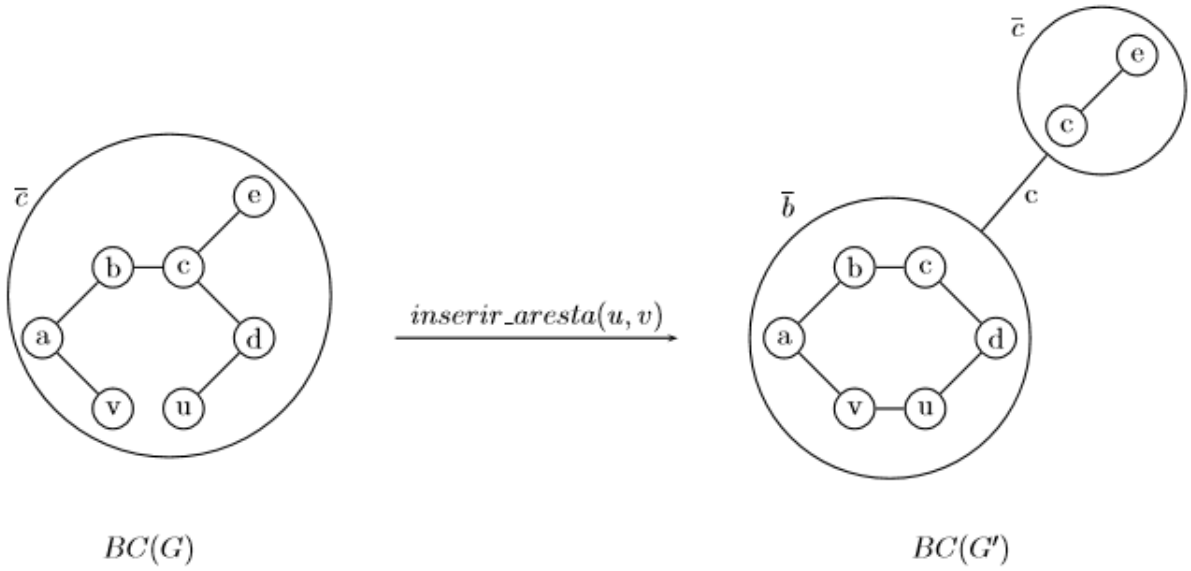
($unordered_set$), que conterá todos os vértices que foram encontrados na busca, incluindo x.

Caso a busca a partir do vértice x não encontre nenhum outro vértice além dele, e x não está presente na estrutura temporária de atributos de aresta, o algoritmo pode prosseguir para o próximo vértice no caminho p_{uv}. Caso contrário, deve-se criar uma nova conexão que se ligará com o novo bloco formado e também será adjacente aos blocos que se ligavam à antiga conexão com atributo de aresta sendo algum vértice presente na estrutura temporária de vértices explorados no BFS auxiliar. Para isso, atribui-se a todos os vértices nesta estrutura uma nova conexão, obtida a partir de *conexao* (atualizando-se as estruturas de índices e índices revertidos). A nível de grafo bloco conexão, realiza-se a atualização da estrutura *bc* para a nova conexão e o novo bloco (ligação entre novo bloco e nova conexão, ligação entre nova conexão e blocos que eram adjacentes à conexão antiga). Além disso, como a aresta é bidirecional, deve-se remover a adjacência dos blocos para a conexão antiga. Finalmente, deve-se remover a conexão antiga, $\bar{c}$ da estrutura (apagando suas adjacências e removendo as ocorrências presentes em $indice_rev[\bar{c}]$) e também adicionar a conexão antiga à pilha *conexao*s.

A Figura 5.6 ilustra um caso simples de adição de aresta hamiltoniana em G, na situação em que a aresta é adicionada numa conexão. Nesse caso, o caminho mínimo é $p_{uv} = \{u, d, c, b, a, v\}$, e todos os vértices estão ligados à mesma conexão, $\bar{c}$. Ao inserir a aresta (u, v) e criar o bloco $\bar{b}$, é necessário preservar a conexão já existente, visto que o vértice e não estará ligado ao bloco $\bar{b}$.



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\bar{c}$
b	$\bar{c}$
c	$\bar{c}$
d	$\bar{c}$
e	$\bar{c}$
u	$\bar{c}$
v	$\bar{c}$

G'	*indice*
a	$\bar{b}$
b	$\bar{b}$
c	$\bar{c}$
d	$\bar{b}$
e	$\bar{c}$
u	$\bar{b}$
v	$\bar{b}$

$BC(G)$	*indice_rev*
$\bar{c}$	a, b, c, d, e, u, v

$BC(G')$	*indice_rev*
$\bar{c}$	c, e
$\bar{b}$	a, b, d, u, v

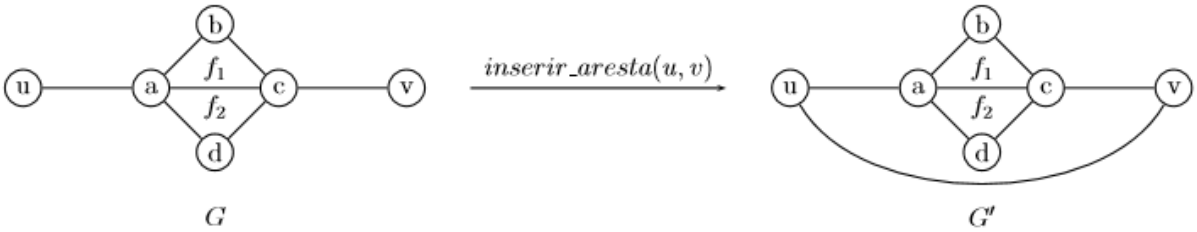
(c) Índices e índices revertidos de G e G'

FIG. 5.6: Adição de aresta hamiltoniana em uma conexão.

O caso de adição b) de atualização de estruturas afetadas pela adição de uma aresta hamiltoniana é um pouco mais complexo, porém tem implementação bastante semelhante à realizada para o caso a). Nessa situação, a aresta é adicionada entre vértices distintos do grafo bloco conexão. O primeiro fato a ser analisado é que, nesse caso, a adição da aresta pode não ser possível. Deve-se então verificar se as condições para inserção da aresta são satisfeitas. Os casos a serem verificados foram mencionados no teorema 4.2 do capítulo

4. Recapitulando:

- Nenhuma aresta no caminho mínimo pode ser fronteira entre faces internas em G. Para verificar essa condição, basta analisar se alguma aresta de p_{uv} é corda, o que pode ser feito utilziando o vetor de predecessores e a estrutura que armazena as cordas do grafo. Caso a aresta (u, v) seja fronteira entre faces internas, o algoritmo informa que a adição dessa aresta invalida a *outerplanaridade*, descarta a aresta e prossegue com sua execução. A Figura 5.7 ilustra um caso em que esse tipo de inserção não é válido. Nessa situação o caminho mínimo é $p_{uv} = \{u, a, c, v\}$. Como a aresta (a, c) é fronteira interna entre as faces f_1 e f_2, a adição da aresta (u, v) invalida a *outerplanaridade*, pelo Teorema 4.2.;



(a) Grafos G e G'

G	*hamiltoniano*
	$(a, b), (a, d), (b, c), (c, d)$

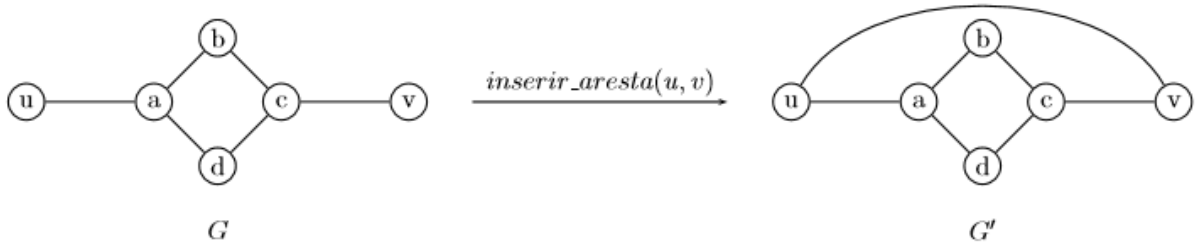
G	*corda*
	(a, c)

(b) Arestas hamiltonianas e cordas de G

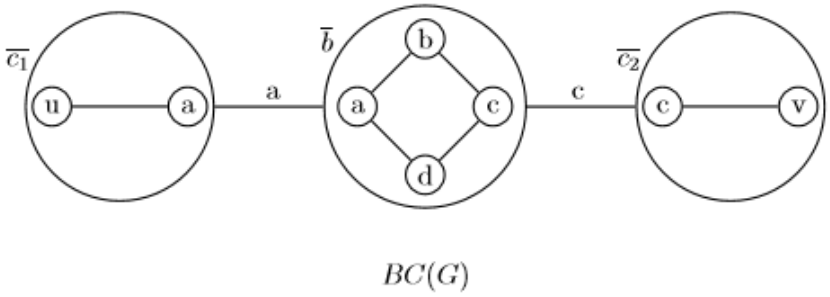
FIG. 5.7: Adição de aresta hamiltoniana inválida em que há aresta fronteira entre faces internas em p_{uv}.

- Duas arestas consecutivas no caminho mínimo não podem pertencer ao mesmo bloco de G. Para esse caso, há a necessidade de implementar uma função auxiliar, *encontrar_bloco*, que calcula o bloco comum a dois vértices que se ligam por uma aresta. Caso os blocos coincidam para as duas arestas consecutivas, há a invalidação da *outerplanaridade* e o algoritmo informa ao usuário essa informação, descartando a aresta e prosseguindo com sua execução. *encontrar_bloco* recebe como parâmetros os dois vértices a serem analisados, por exemplo x e y. Inicialmente, deve-se calcular todos os blocos a que x pertence, armazenando-os em uma estrutura do tipo *unordered_set*, denotada na implementação pela variável *blocos*. Caso x esteja ligado a um bloco ($indice[x]$ aponta para um vértice do tipo bloco), há apenas um bloco, que é indicado por $indice[x]$. Caso contrário, x está ligado a uma conexão, porém pode pertencer a vários blocos em G. Para coletar os blocos aos quais x pertence, é suficiente percorrer as adjacências de $indice[x]$ e armazenar apenas os blocos que possuem cuja aresta de ligação possui atributo x. O mesmo procedimento é utilizado com o vértice y, porém ao invés de armazenar os blo-

cos aos quais y é ligado, para cada bloco verificado, verifica-se se este está presente na estrutura que armazena os blocos a que x pertence. Caso a função encontre algum resultado, retorna o bloco em comum. Caso contrário, a função retorna o valor -1, indicando que não há bloco comum. A Figura 5.8 apresenta um caso em que esse tipo de situação ocorre ao adicionar uma aresta hamiltoniana. Suponha, sem perda de generalidade, que nessa situação o caminho mínimo calculado pelo algoritmo é $p_{uv} = \{u, a, b, c, v\}$. Como as arestas (a, b) e (b, c) são consecutivas e pertencem ao bloco $\bar{b}$, a adição da aresta (u, v) invalida a *outerplanaridade*, pelo teorema 4.2.



(a) Grafos G e G'



(b) Grafo $BC(G)$

G	*indice*
a	$\overline{c_1}$
b	$\bar{b}$
c	$\overline{c_2}$
d	$\bar{b}$
u	$\overline{c_1}$
v	$\overline{c_2}$

G	*indice_rev*
$\bar{b}$	b, d
$\overline{c_1}$	u, a
$\overline{c_2}$	c, v

(c) Índices e índices revertidos de G

FIG. 5.8: Adição de aresta hamiltoniana inválida em que há arestas consecutivas em p_{uv} pertencentes ao mesmo bloco.

Passadas as verificações da manutenção da *outerplanaridade*, é possível realizar a adição de (u, v) em G. Inicialmente, a nível de grafo, percorre-se o caminho p_{uv} com o vetor de predecessores e realiza-se a adição da nova face na lista de faces para cada vértice do caminho, assim como anteriormentes. Além disso, realiza-se a atualização dos tipos das arestas. Diferentemente do primeiro caso em que só havia pontes, nesse caso as arestas de p_{uv} que eram hamiltonianas passam a ser cordas (e portanto devem ser removidas

da estrutura de arestas hamiltonianas e adicionadas na estrutura de cordas). As outras arestas (que eram pontes) passam a ser hamiltonianas (e devem ser inseridas na estrutura de arestas hamiltonianas). Finalmente, adiciona-se o vértice v a $adj[u]$ e vice-versa.

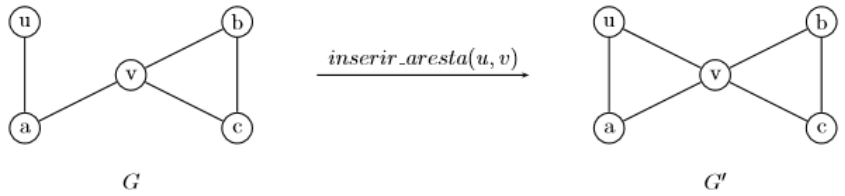
A nível de grafo bloco conexão deve-se percorrer o caminho em $BC(G)$ (com a ajuda do vetor de predecessores de $BC(G)$, que foi calculado previamente, e atualizar todos os blocos e conexões correspondentes. Para realizar as atualizações nas estruturas, devemos primeiramente identificar um bloco na pilha de blocos disponíveis, *bloco*. Esse bloco representa o novo bloco que será formado ao adicionar a aresta hamiltoniana. As atualizações são diferentes a depender do tipo do vértice (bloco ou conexão):

- O caso em que se esteja passando por uma conexão é bem semelhante ao caso de adição de uma aresta hamiltoniana em uma conexão, com a exceção de que o bloco não será formado somente com a conexão em questão mas com um ou mais blocos e uma ou mais conexões. Os detalhes de implementação são praticamente idênticos aos abordados para o primeiro caso e não serão comentados;

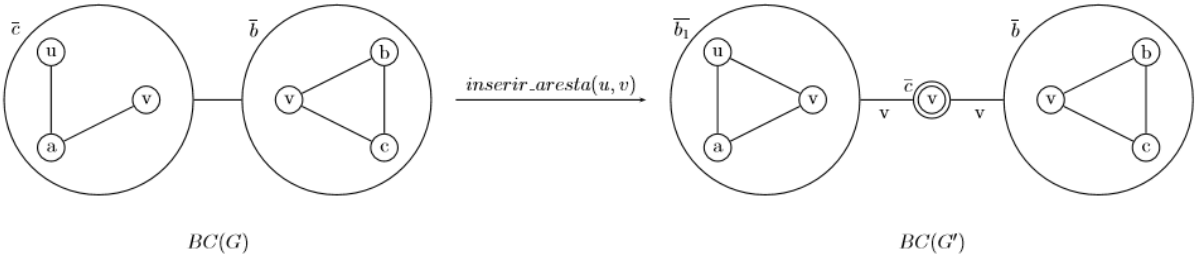
- O caso em que o vértice que se está analisando no caminho $p_{\bar{u}\bar{v}}$ é um bloco, digamos $\bar{b}$ é mais simples. Nesse caso, $\bar{b}$ deixará de existir e passará a fazer parte do novo bloco que será gerado. Todos os vértices de G ligados a $\bar{b}$ devem se ligar ao novo bloco. Isso pode ser feito percorrendo a estrutura de índices revertidos ($indice_rev[\bar{b}]$). Atualizados os índices e os índices revertidos, deve-se atualizar as adjacências a nível de grafo bloco conexão. Todas as adjacências (basta percorrer a estrutura de adjacências de $BC(G)$) de $\bar{b}$ passarão para o novo bloco, exceto à(s) conexão(ões) que também será(ão) atualizada(s) na adição da aresta (u, v). Deve-se lembrar de que, como as arestas são bidirecionais, as adjacências para o vértice $\bar{b}$ devem ser apagadas. Finalmente, é realizada a remoção do bloco $\bar{b}$ em $BC(G)$ e das informações armazenadas em seu índice revertido. Além disso, $\bar{b}$ fica disponível para ser usado e é adicionado à pilha *bloco*.

As Figuras 5.9 e 5.10 apresentam situações simples de inserção de aresta hamiltoniana. Na Figura 5.9, há uma inserção da aresta hamiltoniana (u, v), na qual ambos os vértices estão ligados à mesma conexão, $\bar{c}$. No entanto, a adição da aresta (u, v) faz com que o vértice v pertença a dois blocos distintos, $\bar{b}$ e $\bar{b_1}$, sendo necessária a criação de uma conexão $\bar{c}$ a qual apenas o vértice v se liga, visto que v se tornará uma articulação. A Figura 5.10 ilustra um caso em que a adição conecta vértices distintos em $BC(G)$. O caminho mínimo é $p_{uv} = \{u, a, b, v\}$ e o caminho a ser percorrido no grafo bloco conexão é $p_{\bar{u}\bar{v}} = \{\bar{b_1}, \bar{c}, \bar{b}$. Note que a adição da aresta hamiltoniana (u, v) gera um único bloco, $\bar{b}$ em $BC(G)$. Além disso, as arestas (u, a) e (b, v), que inicialmente eram hamiltonianas, passam a ser cordas. Por outro lado, a aresta (a, b), que antes da inserção de (u, v) era

uma ponte, passa a ser uma aresta hamiltoniana.



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\bar{c}$
b	$\bar{b}$
c	$\bar{b}$
u	$\bar{c}$
v	$\bar{c}$

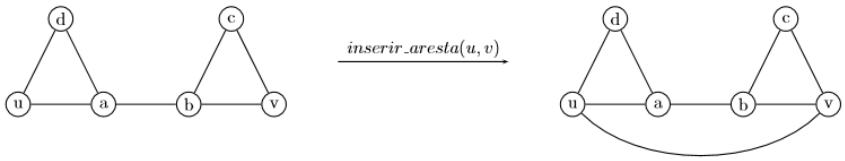
G'	*indice*
a	$\overline{b_1}$
b	$\bar{b}$
c	$\bar{b}$
u	$\overline{b_1}$
v	$\bar{c}$

$BC(G)$	*indice_rev*
$\bar{c}$	u, a, v
$\bar{b}$	b, c

$BC(G')$	*indice_rev*
$\bar{c}$	v
$\bar{b}$	c, b
$\overline{b_1}$	a, u

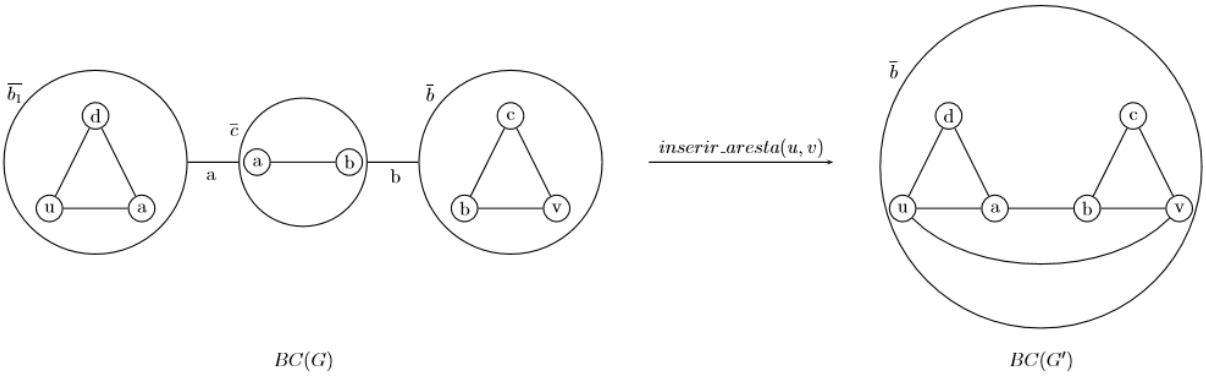
(c) Índices e índices revertidos de G e G'

FIG. 5.9: Adição de aresta hamiltoniana com formação de articulação.



G G'

(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\overline{c}$
b	$\overline{c}$
c	$\overline{b}$
d	$\overline{b_1}$
u	$\overline{b_1}$
v	$\overline{b}$

G'	*indice*
a	$\overline{b}$
b	$\overline{b}$
c	$\overline{b}$
d	$\overline{b}$
u	$\overline{b}$
v	$\overline{b}$

$BC(G)$	*indice_rev*
$\overline{c}$	b, a
$\overline{b}$	v, c
$\overline{b_1}$	d, u

$BC(G')$	*indice_rev*
$\overline{b}$	d, a, c, b, v, u

(c) Índices e índices revertidos de G e G'

FIG. 5.10: Adição de aresta hamiltoniana ligando vértices distintos em $BC(G)$.

5.3.4 PSEUDOCÓDIGO PARA INSERÇÃO DE ARESTA

Denotamos por u um vértice no grafo G, $\bar{u}$ um vértice em $BC(G)$ e $S_{\bar{u}}$ denota o subgrafo de G associado ao bloco ou conexão representado pelo vértice $\bar{u}$ em $BC(G)$. O Algoritmo 1 apresenta as instruções correspondentes ao caso de inserção de uma aresta, segundo as descrições das Subseções 5.3.1, 5.3.2 e 5.3.3.

Algoritmo 1: Inserir Aresta

Entrada: $(u, v), u, v \in V$

1. $C_1 = componente_conexa(u)$;
2. $C_2 = componente_conexa(v)$;
3. **Se** $C_1 \neq C_2$ **então**
4. fundir as componentes C_1 e C_2;
5. inserir a ponte (u, v) em G;
6. atualizar o grafo $BC(G)$; (Seção 5.3.1)
7. **senão**
8. calcular o caminho $p_{\bar{u}\bar{v}}$ entre os vértices $\overline{u}$ ligado a u e $\overline{v}$ ligado a v em $BC(G)$;
9. calcular o caminho p_{uv} em G;
10. **caso** comprimento de $p_{\bar{u}\bar{v}}$:
11. 0:
12. $p_{\bar{u}\bar{v}} = \{\overline{v_1}\}$;
13. **Se** $\overline{v_1}$ é bloco **então** inserir corda (u, v) em $S_{\overline{v_1}}$; (Seção 5.3.2)
14. **senão**
15. inserir aresta hamiltoniana (u, v) em $S_{\overline{v_1}}$;
16. criar novo bloco com base em $p_{\bar{u}\bar{v}}$; (Seção 5.3.3)
17. 1:
18. $p_{\bar{u}\bar{v}} = \{\overline{v_1}, \overline{v_2}\}$;
19. **Se** atributo $(\overline{v_1}, \overline{v_2})$ é u ou v **então**
20. **Se** $\overline{v_1}$ é bloco **então** inserir corda (u, v) em $S_{\overline{v_1}}$; (Seção 5.3.2)
21. **senão** inserir corda (u, v) em $S_{\overline{v_2}}$; (Seção 5.3.2)
22. **senão**
23. inserir aresta hamiltoniana (u, v) em $S_{\overline{v_1}} \cup S_{\overline{v_2}}$;
24. criar novo bloco com base em $p_{\bar{u}\bar{v}}$; (Seção 5.3.3)
25. 2:
26. $p_{\bar{u}\bar{v}} = \{\overline{v_1}, \overline{v_2}, \overline{v_3}\}$;
27. **Se** atributo $(\overline{v_1}, \overline{v_2})$ é u e atributo $(\overline{v_2}, \overline{v_3})$ é v **então** inserir corda (u, v) em $S_{\overline{v_2}}$; (Seção 5.3.2)
28. **senão**
29. inserir aresta hamiltoniana (u, v) em $S_{\overline{v_1}} \cup S_{\overline{v_2}} \cup S_{\overline{v_3}}$;
30. criar novo bloco com base em $p_{\bar{u}\bar{v}}$; (Seção 5.3.3)
31. **senão**:
32. $p_{\bar{u}\bar{v}} = \{\overline{v_1}, \overline{v_2}, ..., \overline{v_k}\}, k \geq 4$;
33. inserir aresta hamiltoniana (u, v) em $S_{\overline{v_1}} \cup S_{\overline{v_2}} ... \cup S_{\overline{v_k}}$;
34. criar novo bloco com base em $p_{\bar{u}\bar{v}}$; (Seção 5.3.3)

Dessa forma, todos os detalhes de implementação acerca da inserção de arestas no algoritmo de manutenção da *outerplanaridade* foram descritos. Serão apresentados a seguir os detalhes na implementação da remoção de arestas.

5.4 REMOÇÃO DE ARESTA

De modo semelhante à inserção, a operação de remoção de arestas no grafo pode ser dividida em três casos, a depender do tipo de aresta que será removida:

- Remoção de ponte, que separará o subgrafo em componentes conexas distintas;

- Remoção de aresta hamiltoniana, que será responsável pela desconfiguração de um bloco no grafo;

- Remoção de corda em um subgrafo que possui um ciclo hamiltoniano.

Deve-se notar que a remoção de arestas no algoritmo de manutenção da *outerplanaridade* sempre pode ser executada, independentemente do tipo de aresta. Portanto será necessário apenas fazer a manutenção das estruturas de dados afetadas.

Como a remoção de arestas do grafo *outerplanar* é sempre possível, o primeiro passo é remover a aresta (u, v) na estrutura de adjacências, ou seja, remove-se v das adjacências de u e vice-versa. Posteriormente deve-se analisar o tipo de aresta que está sendo removida. A verificação nesse caso é mais simples do que no caso da inserção. Sejam $\bar{u} = indice[u]$ e $\bar{v} = indice[v]$. Se $\bar{u}$ e $\bar{v}$ são conexões, tem-se um caso de remoção de ponte. Caso contrário, é suficiente analisar a estrutura que armazena as cordas e as arestas hamiltonianas. Se (u, v) está presente na estrutura de corda, naturalmente é uma aresta do tipo corda. Do contrário, é uma aresta hamiltoniana.

Nas próximas subseções será feita uma descrição detalhada da implementação para cada um dos casos citados, analisando-se principalmente as estruturas envolvidas na operação.

Ao finalizar todos os casos de remoção de aresta, na Subseção 5.4.4, será apresentado o pseudocódigo correspondente.

5.4.1 REMOÇÃO DE PONTE

No caso da remoção de uma ponte num grafo *outerplanar* armazenado segundo descrito nos capítulos anteriores, tem-se que os vértices u e v estão ligados à mesma conexão em $BC(G)$. Vale observar que a remoção de uma aresta que faz parte de uma conexão gera duas conexões. Portanto, será necessário encontrar uma conexão utilizando a pilha de conexões disponíveis, *conexao*. No algoritmo, a conexão $\bar{v}$ que já existia será mantida de tal forma que o vértice v ainda será ligado a $\bar{v}$. A nova conexão estará ligada ao vértice u.

Uma mudança que deve ser feita a nível de grafo é a atualização da estrutura de conjuntos, visto que será criada uma nova componente conexa. Sendo assim, deve-se obter um novo conjunto a partir da estrutura de conjuntos livres. Como decidiu-se preservar a existência de $\bar{v}$, a componente conexa induzida pelo vértice v continuará sendo representada pelo mesmo conjunto, logo, a componente induzida pelo vértice u após a remoção da ponte deve se associar a um novo conjunto.

Em seguida, de modo semelhante ao realizado na inserção de arestas, cria-se uma outra estrutura temporária, *temp*, que armazenará todos os vértices da conexão atual, com raiz

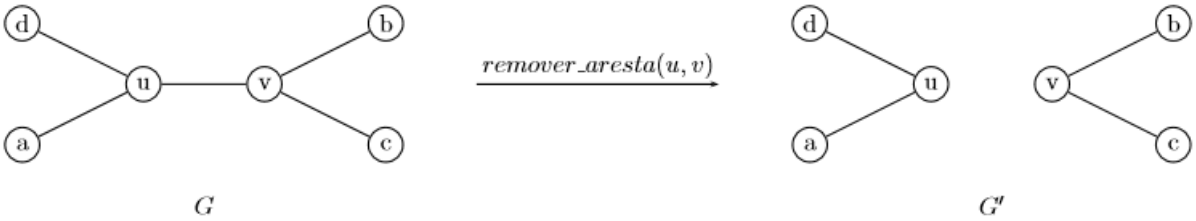
em u e que não pertencem à subárvore cuja raiz é v. Pode-se montar essa estrutura e atualizar as informações dos conjuntos simultaneamente executando uma busca em largura a partir de u, que exclui a subárvore com raiz em v. O BFS explora todos os vértices que terão seu conjunto modificado e, caso o vértice pertença à conexão atual (o que pode ser verificado a partir do vetor de índices), $\bar{v}$, ele é adicionado à estrutura temporária.

Com a estrutura temporária de vértices preenchida, deve-se atualizar o índice e o índice revertido para ocada um dos vértices que estão presentes na estrutura, ligando-os à nova conexão, $\bar{u}$.

Notar que, a nível de grafo bloco conexão, deve-se atualizar as adjacências. Isso acontece porque a conexão foi segmentada, e alguns dos blocos (caso houver) devem passar a ser adjacentes à nova conexão, $\bar{u}$. Sendo assim, deve-se percorrer as adjacências da conexão antiga. Caso o atributo de aresta obtido a partir das adjacências esteja presente na estrutura temporária, isso significa que essa aresta deve deixar a conexão antiga e passar para a nova conexão $\bar{u}$ (portanto cria-se a adjacência do bloco com $\bar{u}$ e remove-se essa adjacência de $\bar{v}$).

Conforme mencionado anteriormente, o restante dos vértices continuam pertencendo ao conjunto antigo e ligados à conexão $\bar{v}$. As adjacências restantes também continuam armazenadas na estrutura de adjacências de $\bar{v}$.

As figuras a seguir ilustram casos de remoção de ponte. A Figura 5.11 ilustra um caso mais simples. A Figura 5.12 mostra um caso em que há blocos adjacentes à conexão da qual é removida uma aresta.



(a) Grafos G e G'

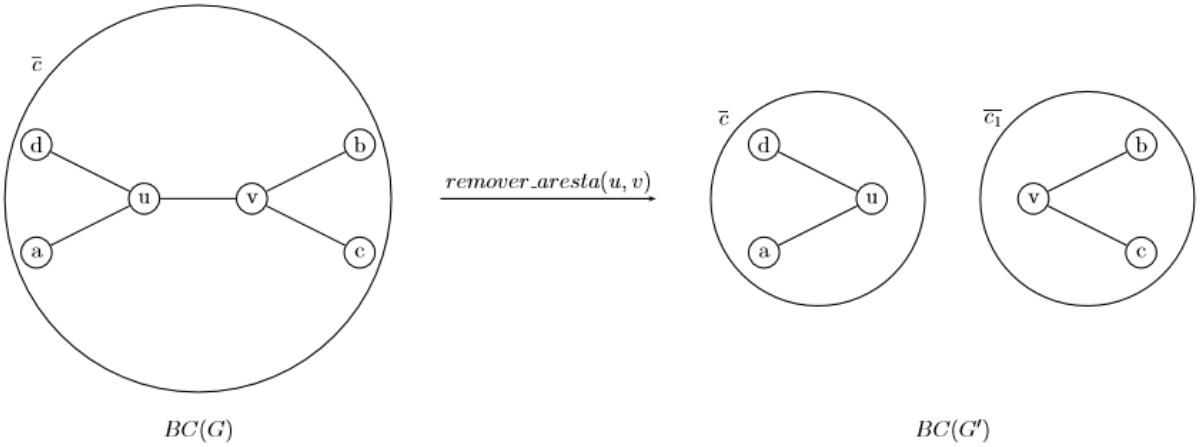
G	*conjunto*
a	1
b	1
c	1
d	1
u	1
v	1

G'	*conjunto*
a	1
b	2
c	2
c	1
u	1
v	2

(b) Componentes conexos de G e G'

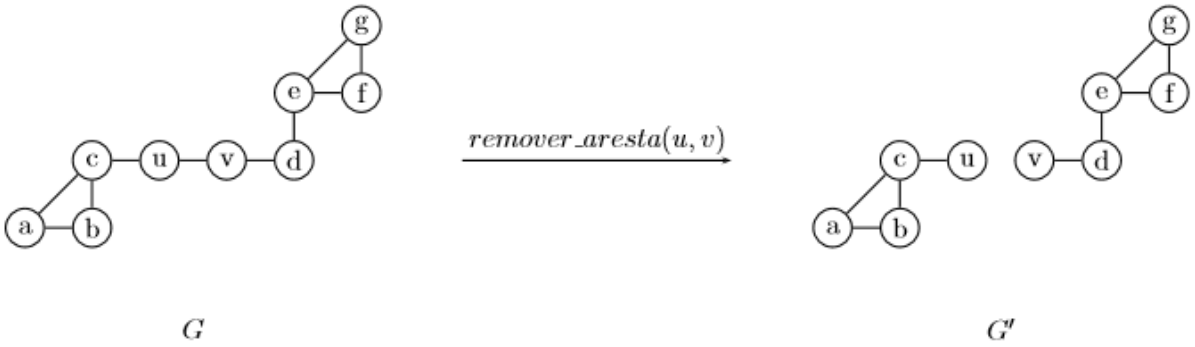
temp
u
a
d

(c) Estrutura temporária *temp*

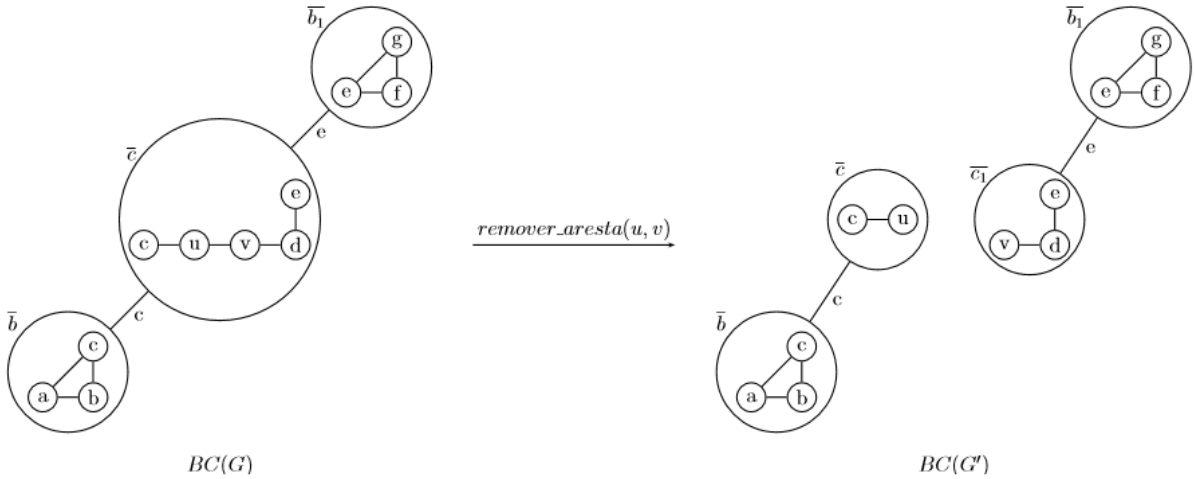


(d) Grafos $BC(G)$ e $BC(G')$

FIG. 5.11: Remoção de ponte em G.



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\overline{b}$
b	$\overline{b}$
c	$\overline{c}$
d	$\overline{c}$
e	$\overline{c}$
f	$\overline{b_1}$
g	$\overline{b_1}$
u	$\overline{c}$
v	$\overline{c}$

G'	*indice*
a	$\overline{b}$
b	$\overline{b}$
c	$\overline{c}$
d	$\overline{c_1}$
e	$\overline{c_1}$
f	$\overline{b_1}$
g	$\overline{b_1}$
u	$\overline{c}$
v	$\overline{c_1}$

$BC(G)$	*indice_rev*
$\overline{c}$	u, d, c, v, e
$\overline{b}$	a, b
$\overline{b_1}$	g, f

$BC(G')$	*indice_rev*
$\overline{c}$	u, c
$\overline{c_1}$	d, v, e
$\overline{b}$	a, b
$\overline{b_1}$	g, f

(c) Índices e índices revertidos de G e G'

FIG. 5.12: Remoção de ponte em G.

5.4.2 REMOÇÃO DE CORDA

Caso a aresta a ser removida não seja uma ponte, procede-se de forma semelhante à realizada na inserção de arestas. Antes de verificar se a aresta a ser removida é uma corda uma aresta hamiltoniana, calcula-se o caminho mínimo, p_{uv} entre u e v em G utilizando uma busca em largura. E será mantido o vetor de predecessores a partir do qual é possível reconstruir o caminho. É importante lembrar que, nesse ponto do algoritmo, a aresta (u, v) já foi removida e o caminho mínimo p_{uv} não será a aresta (u, v), mas uma sequência de

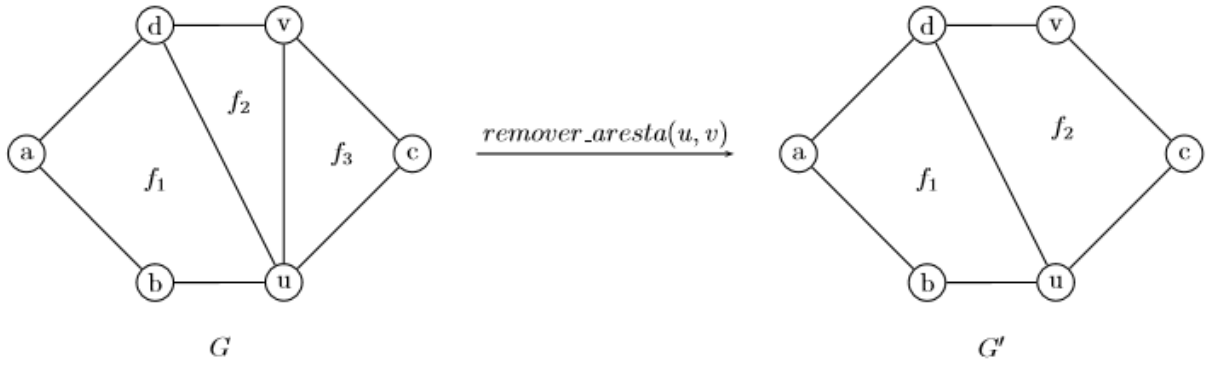
pelo menos três vértices.

Se a estrutura que armazena as cordas contiver a aresta (u, v), tem-se um caso de remoção de corda, naturalmente. A remoção de uma corda do grafo *outerplanar* é uma operação relativamente simples, visto que, assim como no caso da inserção, não altera a estrutura do grafo bloco conexão. Portanto, apenas deve-se atualizar apenas as estruturas a nível de grafo.

Inicialmente é realizada a remoção da corda da estrutura que armazena as cordas do grafo. Como a aresta (u, v) existia e os vértices u e v faziam parte do mesmo bloco, necessariamente eles devem possuir duas faces em comum (além da face externa). O algoritmo então deve encontrar as duas faces em comum, digamos f_1 e f_2. Essas faces podem ser encontradas utilizando-se o método *encontrar_face_comum*, apresentado no caso de inserção de arestas (Subseção 5.3.2).

Para realizar a atualização da estrutura de faces, que é a única a ser modificada nesse caso, deve-se utilizar um vértice intermediário do caminho p_{uv}. Usando a estrutura de predecessores, obtém-se um dos vértices. No caso do algoritmo, toma-se x como sendo o vértice predecessor de v em p_{uv}. Caso x também possua f_1 em sua estrutura de faces, remove-se f_1 de todos os vértices do caminho p_{uv} e adiciona-se f_2 à estrutura de faces de todos os vértices de p_{uv}, exceto u e v. Caso contrário, x possuirá em sua estrutura de faces a face f_2. Sendo assim, remove-se f_2 de todos os vértices do caminho p_{uv} e adiciona-se a face f_1 à estrutura de faces de todos os vértices de p_{uv}, exceto u e v.

A Figura 5.13 apresenta um caso simples de remoção de corda. Nessa situação, tem-se que u e v possuem as faces f_2 e f_3 em comum. Sem perda de generalidade, considere que o caminho mínimo seja $p_{uv} = \{u, c, v\}$ (outra possibilidade seria o caminho $\{u, d, v\}$)). O vértice c, intermediário, possui a face f_3 em sua lista de faces. Logo, deve-se remover f_3 das listas de u, c e v e adicionar a face f_2 à lista de faces de c.



(a) Grafos G e G'

G	$faces$
a	f_1
b	f_1
c	f_3
d	f_1, f_2
u	f_1, f_2, f_3
v	f_2, f_3

G'	$faces$
a	f_1
b	f_1
c	f_2
d	f_1, f_2
u	f_1, f_2
v	f_2

(b) Lista de faces de G e G'

FIG. 5.13: Remoção de corda em G.

5.4.3 REMOÇÃO DE ARESTA HAMILTONIANA

O último caso trata da remoção de uma aresta hamiltoniana. Neste caso um vértice correspondente a um bloco do grafo bloco conexão será eliminado e a seguir devem ser criados novos vértices. Assim como na inserção de arestas hamiltonianas, esse é o caso mais complexo e com mais possibilidades.

Inicialmente, procede-se com a atualização das estruturas a nível de grafo. Primeiramente remove-se a aresta (u, v) da estrutura que armazena as arestas hamiltonianas e, em seguida, calcula-se a face em comum, f entre os vértices u e v. Essa face pode ser encontrada utilizando a função *encontrar_face_ comum*. Tendo identificado a face comum, o algoritmo remove f da lista de faces de todos os vértices do caminho p_{uv} (utilizando o vetor de predecessores calculado utilizando a BFS). A remoção da face na lista é uma remoção comum em uma lista duplamente encadeada.

Em seguida, realizam-se as atualizações a nível de grafo bloco conexão. Inicialmente, calcula-se o bloco comum a que u e v se ligam, que será removido futuramente. A implementação do cálculo do bloco em comum foi descrita no detalhamento da inserção de arestas (Subseção 5.3.3).

Prossegue-se o algoritmo percorrendo o caminho p_{uv}. A atualização das estruturas divide-se em três casos a depender dos tipos de arestas encontrados em p_{uv}:

- Sequência contígua de arestas hamiltonianas;

- Presença de cordas consecutivas;

- Presença de corda.

O detalhamento da implementação para cada caso será descrito a seguir:

- Sequência contígua de arestas hamiltonianas. Ao percorrer o caminho p_{uv} pode-se ter várias sequências ininterruptas de arestas hamiltonianas. Nesse caso, o conjunto de vértices para cada sequência irá formar uma nova conexão. Notar que, como existia um bloco antes da remoção da aresta (u, v), é possível ter alguns desses vértices da sequência ligados a conexões, então, caso isso ocorra, deve haver a preocupação em realizar a fusão de todas essas conexões que já existem, ligadas aos vértices do caminho contíguo. Para formar a conexão única, é implementada a função *unir_conexao*. A função recebe um conjunto de vértices em G e une os mesmos, além dos respectivos vértices que estejam ligados nas respectivas conexões em $BC(G)$ a que esse conjunto de vértices possa pertencer, em uma única conexão. Além disso, a função atualiza toda a estrutura de $BC(G)$, desfazendo qualquer ligação que exista com esse bloco que deixará de existir e criando uma nova adjacência (com o respectivo atributo de aresta) caso algum bloco tenha sido criado imediatamente antes no percurso de p_{uv}. A função retorna o identificador da conexão que foi criada, que pode ser passado futuramente em cada um dos outros casos, se for necessário.

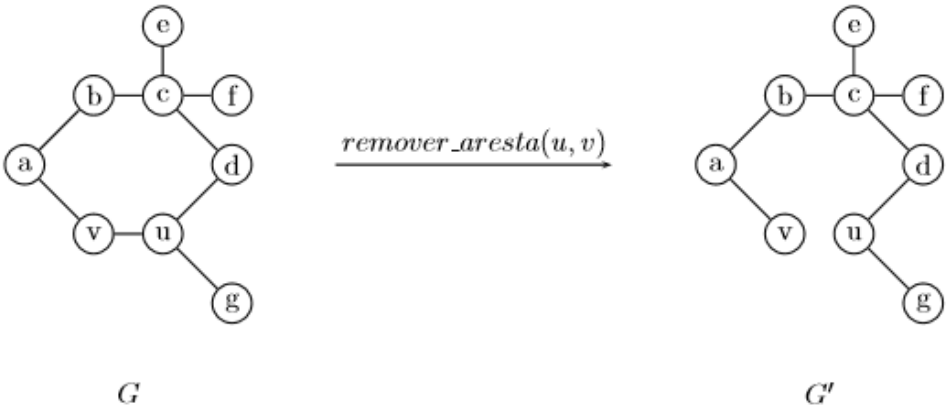
Inicialmente a função obtém uma conexão livre na pilha *conexao*. Essa será a conexão resultante da operação. Para cada vértice x da sequência contígua de arestas hamiltonianas analisam-se dois casos: caso o vértice esteja ligado a um bloco, apenas é necessário atualizar as informações de índice e índice revertido de x. Caso contrário, adiciona-se a conexão a que x está ligado a uma estrutura temporária, *conexoes_apagar*, de conexões que serão eliminadas (um vetor simples) e atribui-se todos os vértices da conexão que será eliminada à nova conexão (pode-se realizar essa operação percorrendo a estrutura de índices invertidos $indice_rev[indice[\bar{x}]]$ e atualizando os respectivos índices. Ao final da operação, remove-se o índice revertido antigo).

Em seguida, deve-se percorrer o vetor com as conexões a eliminar para que seja possível transferir todas as adjacências para a nova conexão, exceto a adjacência para o bloco que está sendo desfeito. Além disso, os blocos adjacentes às conexões que serão eliminadas deverão substituir as respectivas adjacências a essa conexões por uma à nova conexão. Todas essa operações podem ser realizadas com o auxílio da estrutura de adjacências das conexões que serão removidas. Finalmente, adicionam-se as conexões eliminadas de $BC(G)$ à pilha *conexao* para posterior disponibilização.

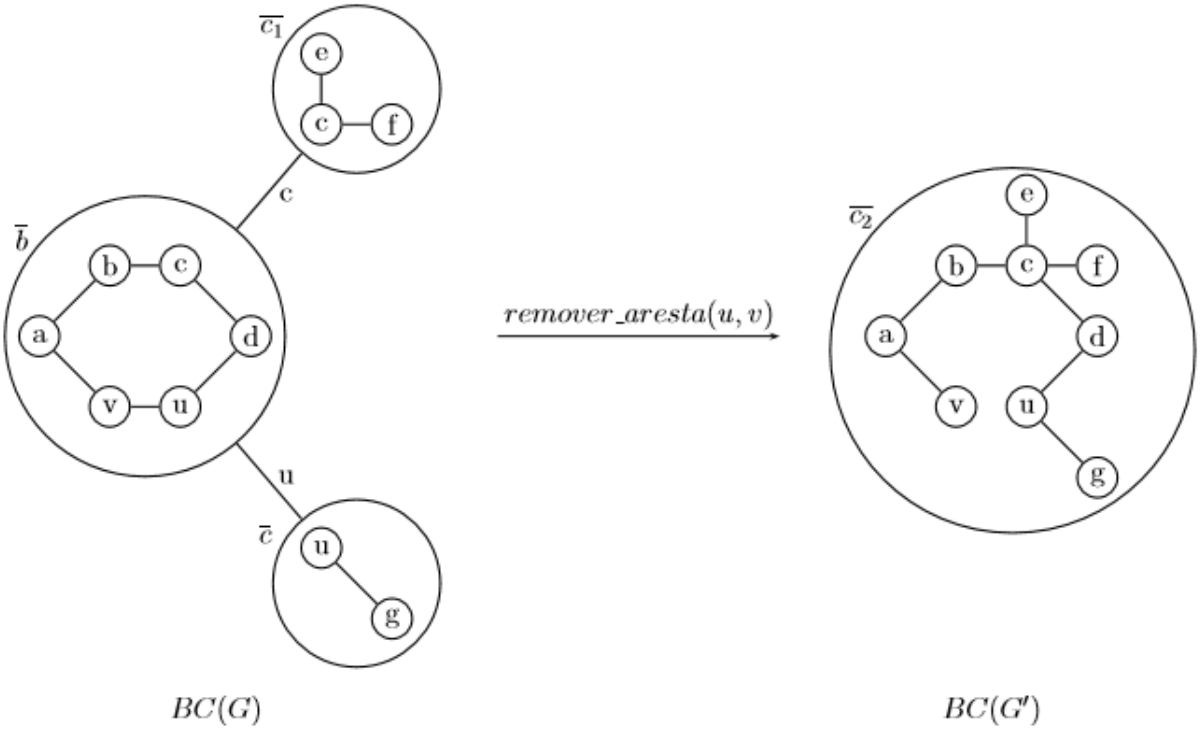
Por fim, caso algum bloco tenha sido criado imediatamente antes no percurso de p_{uv},

cria-se uma adjacência entre a nova conexão e esse bloco, com o respectivo atributo de aresta.

A Figura 5.14 ilustra um caso de remoção de aresta hamiltoniana em que o caminho mínimo é composto por uma sequência contígua de arestas hamiltonianas. Nesse caso, o caminho mínimo é $p_{uv} = \{u, d, c, b, a, v\}$, formado somente por arestas hamiltonianas. Todos os vértices que formavam um único bloco, $\bar{b}$, serão ligados a uma única conexão, $\bar{c_2}$. Notar que os vértices que faziam parte das conexãoes às quais u e c eram ligados inicialmente ($\bar{c}$ e $\bar{c_1}$, respectivamente), também passam a fazer parte da nova conexão, $\bar{c_2}$. Neste exemplo, todas as arestas hamiltonianas no bloco $\bar{b}$ passam a ser pontes depois de remover a aresta (u, v);



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

$BC(G)$	bc
$\overline{c}$	(u, $\overline{b}$)
$\overline{c_1}$	(c, $\overline{b}$)
$\overline{b}$	(c, $\overline{c_1}$), (u, $\overline{c}$)

$BC(G')$	bc
$\overline{c_2}$	-

(c) Adjacências de $BC(G)$ e $BC(G')$

conexoes_apagar
$\overline{c}$
$\overline{c_1}$

(d) Estrutura temporária *conexoes_apagar*

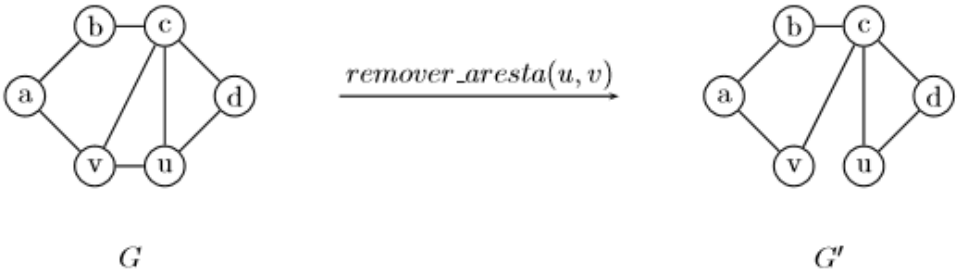
FIG. 5.14: Remoção de aresta hamiltoniana em G.

- Presença de cordas consecutivas. Esta situação pode ser verificada analisando a

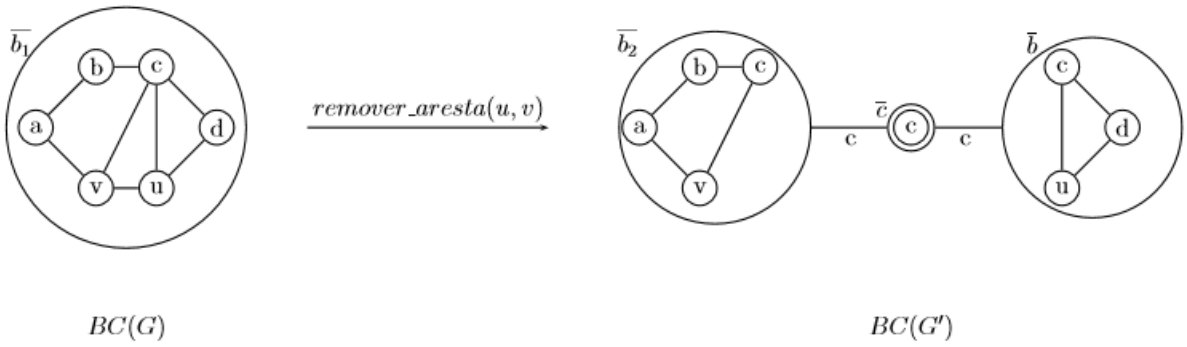
estrutura que armazena as cordas do grafo G. Nesse caso, o vértice comum às cordas consecutivas será um ponto de articulação em G após a remoção da aresta (u, v). Se esse vértice não está ligado a nenhuma conexão, deve-se criar uma conexão contendo somente esse vértice. Uma estrutura temporária do tipo *unordered_set* é criada para armazenar todos os vértices de articulação nessa operação. Essa estrutura somente é utilizada no próximo caso a ser analisado, ao se verificar cada corda individualmente. A atualização do grafo bloco conexão é implementada para esse caso na função *criar_conexao_articulacao*. Essa função cria, se necessário, uma conexão para o vértice de articulação. Além disso, desfaz todas as ligações com o bloco que será futuramente apagado. Finalmente, realiza a ligação dessa conexão com o bloco previamente criado no percurso do caminho p_{uv}, com o respectivo atributo de aresta. A função retorna o identificador ao qual o ponto de articulação está ligado.

Agora consideramos se o vértice comum às cordas consecutivas está ligado a algum bloco ou a uma conexão. Se o vértice de articulação está ligado a um bloco, deve-se ligá-lo a uma nova conexão, retirada da pilha *conexao* e atualizar o respectivo índice e índice revertido. Caso contrário, a articulação já estava ligada a uma conexão. Nesse caso, há uma aresta na estrutura de adjacências que liga essa conexão ao bloco que será removido. Deve-se então remover essa adjacência. Finalmente, deve-se atualizar o grafo bloco conexão inserindo a adjacência entre a conexão à qual o vértice de articulação está ligado e o bloco que foi previamente criado no percurso do caminho mínimo p_{uv}.

As figuras a seguir ilustram dois casos em que há remoção de aresta hamiltoniana e há presença de cordas consecutivas no caminho p_{uv}. Na Figura 5.15 o vértice que se tornará ponto de articulação está ligado a um bloco. Na Figura 5.16 o vértice que se tornará ponto de articulação está ligado a uma conexão. Em ambos os casos, o caminho mínimo é $p_{uv} = \{u, c, v\}$, composto por duas cordas consecutivas, (u, c) e (c, v), Também, em ambos os casos, todas as arestas hamiltonianas em $\bar{b_1}$ continuam sendo hamiltonianas após remover (u, v) (em $\bar{b_2}$ e $\bar{b}$);



(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\overline{b_1}$
b	$\overline{b_1}$
c	$\overline{b_1}$
d	$\overline{b_1}$
u	$\overline{b_1}$
v	$\overline{b_1}$

G'	*indice*
a	$\overline{b_2}$
b	$\overline{b_2}$
c	$\overline{c}$
d	$\overline{b}$
u	$\overline{b}$
v	$\overline{b_2}$

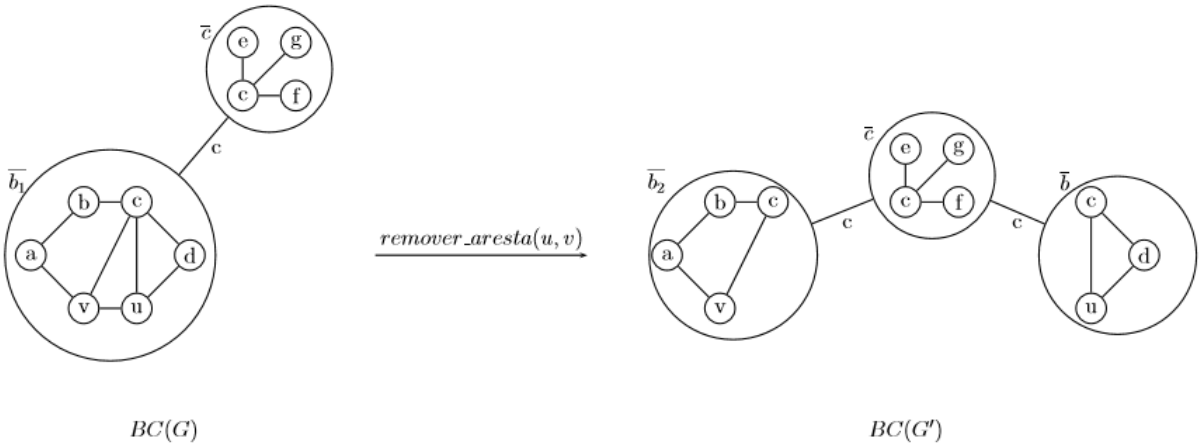
$BC(G)$	*indice_rev*
$\overline{b_1}$	a, v, d, u, b, c

$BC(G')$	*indice_rev*
$\overline{c}$	c
$\overline{b}$	d, u
$\overline{b_2}$	v, b, a

(c) Índices e índices revertidos de $BC(G)$ e $BC(G')$

FIG. 5.15: Remoção de aresta hamiltoniana em G.

(a) Grafos G e G'



(b) Grafos $BC(G)$ e $BC(G')$

G	*indice*
a	$\overline{b_1}$
b	$\overline{b_1}$
c	$\overline{c}$
d	$\overline{b_1}$
e	$\overline{c}$
f	$\overline{c}$
g	$\overline{c}$
u	$\overline{b_1}$
v	$\overline{b_1}$

G'	*indice*
a	$\overline{b_2}$
b	$\overline{b_2}$
c	$\overline{c}$
d	$\overline{b}$
e	$\overline{c}$
f	$\overline{c}$
g	$\overline{c}$
u	$\overline{b}$
v	$\overline{b_2}$

$BC(G)$	*indice_rev*
$\overline{c}$	c, g, f, e
$\overline{b_1}$	a, v, d, u, b

$BC(G')$	*indice_rev*
$\overline{c}$	c, g, f, e
$\overline{b}$	d, u
$\overline{b_2}$	v, b, a

(c) Índices e índices revertidos de $BC(G)$ e $BC(G')$

FIG. 5.16: Remoção de aresta hamiltoniana em G.

- Presença de corda. Nesse caso, a aparição de uma corda em p_{uv} gera um novo bloco. O novo bloco será formado pela corda (x, y) e todas as arestas hamiltonianas no caminho de x até y. Para realizar essa operação, utiliza-se a função *cria_ bloco*, que recebe uma corda e cria o seu bloco associado. Além disso, a função realiza as devidas atualizações em $BC(G)$, desfazendo qualquer ligação com o bloco que será futuramente eliminado. Finalmente, deve-se ligar o bloco que será criado no decorrer da função com a conexão que foi previamente criada (se houver) no percurso do caminho p_{uv}, com o respectivo atributo de aresta. A função retorna o identificador do bloco criado. Vale ressaltar que utiliza-se nesse método a estrutura temporária com os vértices de articulação que foram

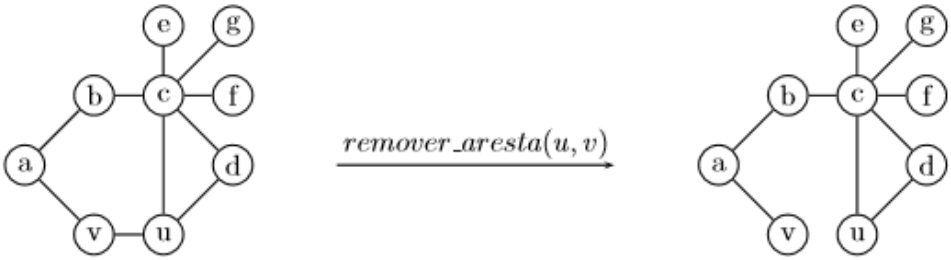
sendo descobertos com a execução da remoção da aresta (u, v). Essa estrutura é útil para não atribuir os vértices de articulação ao bloco que será criado nessa função, visto que, conforme mencionado anteriormente, as articulações serão ligadas a uma conexão.

Inicialmente deve-se procurar um bloco da pilha *bloco*, pois será formado um novo bloco. A seguir deve-se identificar quais vértices farão parte desse novo bloco. Para obter essa informação é necessário analisar o caminho p_{xy} em G percorrendo-se apenas arestas hamiltonianas. Pode-se conseguir essa informação executando uma busca em largura com origem em x, determinando o caminho p_{xy} juntamente com a informação armazenada na estrutura que armazena as arestas hamiltonianas. O BFS armazena as informações do caminho no vetor de predecessores. Prossegue-se então atualizando os índices e índices revertidos dos vértices no caminho p_{xy}. Para cada vértice do caminho, caso o mesmo não seja articulação e não esteja ligado a uma conexão, então deverá ser ligado ao novo bloco. Esses vértices serão armazenados em uma estrutura temporária do tipo *unordered_set*, que possibilitará uma rápida consulta desses itens.

Em seguida atualiza-se o grafo bloco conexão. O novo bloco deve conter as informações de adjacências do bloco a ser removido (apenas as adjacências cujo atributo de aresta esteja presente na estrutura temporária descrita anteriormente). Além disso, como as arestas são não direcionadas, também é necessário atualizar as conexões adjacentes ao bloco removido, incluindo o novo bloco (apenas as conexões cujo atributo de aresta das adjacências coincidem com os vértices armazenados na estrutura temporária).

Caso tenha sido criada uma conexão imediatamente antes no percurso de p_{uv}, deve-se então criar uma adjacência entre o novo bloco e essa conexão.

A Figura 5.17 exibe um caso de remoção de aresta hamiltoniana no qual uma das arestas do caminho p_{uv} é uma corda. Nessa situação, o caminho mínimo é $p_{uv} = \{u, c, b, a, v\}$, composto por uma corda, (u, c), e uma sequência contígua de arestas hamiltonianas. A presença da corda em questão produzirá o bloco $\bar{b}$, composto pelos vértices c, d e u. A aresta (u, c) deixa de ser corda e passa a ser hamiltoniana, enquanto as arestas da sequência contígua deixam de ser hamiltonianas e passam a ser pontes, formando a nova conexão, $\bar{c}$.



G G'

(a) Grafos G e G'

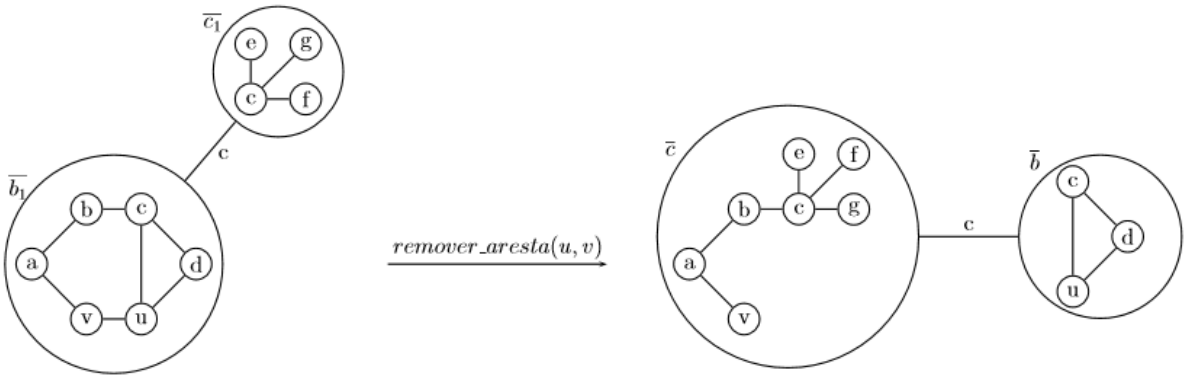
G	*hamiltoniano*
	$(a,b),(a,v),(b,c),$ $(c,d),(d,u),(u,v)$

G	*corda*
	(c,u)

G'	*hamiltoniano*
	$(c,d),(c,u),(d,u)$

G'	*corda*
	-

(b) Arestas hamiltonianas e cordas de G e G'



$BC(G)$ $BC(G')$

(c) Grafos $BC(G)$ e $BC(G')$

FIG. 5.17: Remoção de aresta hamiltoniana em G.

Em várias ocasiões que o tipo das arestas deveria ser modificado, realizava-se a alteração dessa informação a nível de grafo inicialmente, por questão de simplicidade de operação. No caso da remoção das arestas hamiltonianas não é possível realizar a operação inicialmente, pois a análise de cada um dos três casos é feita em conjunto, percorrendo o caminho p_{uv}, e a informação acerca do tipo da aresta deve ser mantida até o fim do percurso. Portanto, após realizar o percurso p_{uv}, pode-se atualizar a informação das arestas. Caso a aresta fosse uma corda inicialmente, ela passa a ser uma aresta hamiltoniana, conforme visto no terceiro caso (portanto remove-se a aresta da estrutura que armazena as cordas e insere-se a aresta na estrutura que armazena as arestas hamiltonianas). Caso

a aresta seja hamiltoniana, ela passa a ser uma ponte, conforme visto no primeiro caso, e, como as pontes não são armazenadas no algoritmo, é suficiente retirar a aresta da estrutura de arestas hamiltonianas.

5.4.4 PSEUDOCÓDIGO PARA REMOÇÃO DE ARESTA

Seguindo com a mesma notação da Seção 5.3.4, denotamos por u um vértice no grafo G, $\bar{u}$ um vértice em $BC(G)$ e $S_{\bar{u}}$ denota o subgrafo de G associado ao bloco ou conexão representado pelo vértice $\bar{u}$ em $BC(G)$. O Algoritmo 2 apresenta as instruções correspondentes ao caso de remoção de uma aresta, segundo a descrição das Subseções 5.4.1, 5.4.2 e 5.4.3.

Algoritmo 2: Remover Aresta

Entrada: $(u, v), u, v \in V$

1. $\bar{v}$ = bloco ou conexão do qual a aresta será removida;
2. remover (u, v) de $S_{\bar{v}}$;
3. **Se** $\bar{v}$ é conexão **então**
4. dividir G em novas componentes;
5. atualizar o grafo $BC(G)$; (Seção 5.4.1)
6. **senão**
7. calcular o caminho p_{uv} em G;
8. **Se** (u, v) é corda **então**
9. atualizar as faces de $S_{\bar{v}}$; (Seção 5.4.2)
10. **senão**
11. dividir o bloco associado a $S_{\bar{v}}$;
12. atualizar o grafo $BC(G)$; (Seção 5.4.3)

Dessa forma, todos os detalhes de implementação do algoritmo de manutenção da *outerplanaridade* foram abordados.

6 ANÁLISE DA COMPLEXIDADE

Neste capítulo será analisada a complexidade geral do algoritmo de manutenção da *outerplanaridade*. Conforme visto no Capítulo 5, são muitos os detalhes de implementação do algoritmo, o que acaba dificultando a visualização da complexidade de sua execução. Para facilitar a análise proceder-se-á com abordagem semelhante aos capítulos anteriores: será feita a demonstração de complexidade individualmente para as operações de inserção e remoção de arestas. Portanto, ao executar separadamente cada caso, dependendo do tipo de aresta sendo inserida/removida, a complexidade geral será majorada pelo caso com pior complexidade de execução.

A complexidade das operações correspondentes às estruturas de dados utilizadas na implementação do algoritmo descrita no Capítulo 5 podem ser encontradas no Capítulo 2.

A seguir, analisamos o tamanho das estruturas que armazenam o grafo, o grafo bloco conexão e as estruturas de interligação em função do tamanho da entrada, o número de vértices do grafo (n), e do número de arestas (m).

A nível de grafo, as respectivas estruturas que armazenam as adjacências, as arestas hamiltonianas e as cordas são representadas por conjuntos não ordenados. Para as estruturas em questão, o tamanho associado é $O(m)$, onde m é a quantidade de arestas do grafo G. Pelo Teorema 2.2, no caso de grafos *outerplanares* o número máximo de arestas é $m = 2n - 3$, então tem-se que o tamanho dessas estruturas é $O(n)$. A estrutura de faces do grafo G é representada por uma lista duplamente encadeada de tamanho $n - 2$, pois só são armazenadas as faces internas do grafo *outerplanar*, que pela fórmula de Euler (Teorema 2.1) está limitada por $n - m + 1 \leq n - 2$. Já os conjuntos disponíveis são implementados utilizando-se uma pilha com no máximo n elementos. Os componentes conexos do grafo, a estrutura de predecessores dos vértices e os vértices visitados são armazenados em vetores de tamanhos n.

Com relação às estruturas a nível de grafo bloco conexão, as adjacências são armazenadas em conjuntos não ordenados de tamanho n com a informação adicional do atributo de aresta (o que não altera a complexidade geral). Os blocos e as conexões disponíveis são armazenados em pilhas de tamanho n. Finalmente, a estrutura de predecessores de vértices do grafo bloco conexão é representada por um vetor de tamanho n.

No caso das estruturas que conectam o grafo com o grafo bloco conexão, os índices

são representados por um vetor de pares ordenados com tamanho n, enquanto os índices revertidos utilizam duas estruturas aninhadas para sua representação: mapeamentos não ordenados ($unordered_map$) e conjuntos não ordenados ($unordered_set$) de tamanhos máximos n e n, sendo que a quantidade máxima de elementos da estrutura de índices revertidos não excede n.

A Tabela 6.1 resume as estrutura de dados usadas e os correspondentes tamanhos.

	armazena	nome da estrutura	tipo	tamanho
EG	adjacências	$adj[v], \forall v \in V$	conj. não ordenados	$2m = O(n)$
	arestas hamiltonianas	$hamiltoniano$	conj. não ordenado	$m = O(n)$
	cordas	$corda$	conj. não ordenado	$m = O(n)$
	faces	$faces[v], \forall v \in V$	lista dupl. encadeada ordenada	n
	conjuntos disponíveis	$conjuntos$	pilha	n
	componentes conexos	$conjunto[v], \forall v \in V$	vetor	n
	predecessores (BFS)	$pai[v], \forall v \in V$(BFS)	vetor	n
	vértices visitados (BFS)	$vis[v], \forall v \in V$(BFS)	vetor booleano	n
$EBC(G)$	adjacências	$bc[tipo][\overline{v}]$	conj. não ordenados	$n-1$
	blocos disponíveis	$bloco$	pilha	$\lfloor \frac{n-1}{2} \rfloor$
	conexões disponíveis	$conexao$	pilha	n
	predecessores (DFS)	$pai_bc[tipo][\overline{v}]$, $\forall \overline{v} \in V(BC)$	vetor	n
EI	índice	$indice$	vetor	n
	índice revertido	$indice_rev$	map. não ordenado de conj. não ordenados	n

TAB. 6.1: Estruturas de dados utilizadas e respectivos tamanhos.

Nas próximas duas subseções será realizada uma análise da complexidade das operações de inserção e remoção de arestas apresentadas nos Capítulos 4 e 5.

6.1 INSERÇÃO DE ARESTA

6.1.1 ADIÇÃO DE PONTE

Primeiramente deve-se atualizar as adjacências de G, e como mencionado anteriormente, a inserção de um elemento nos conjuntos não ordenados pode ser feita em complexidade constante. Posteriormente, deve-se atualizar a estrutura de componentes conexos, a nível de grafo. Como os componentes conexos são implementados com um vetor, pode-se atualizar cada um deles com complexidade constante. Tendo em vista que ao realizar uma fusão de componentes conexas, deve-se atualizar cada um dos vértices de uma componente

para a outra, tem-se que a complexidade para fundir duas componentes conexas é, no pior caso, linear.

É importante observar que as estruturas de índices e índices revertidos também devem ser atualizadas. A complexidade para a atualização dos índices também é $O(n)$ no pior caso, pois quando realiza-se a adição de uma ponte entre vértices ligados a conexões forma-se uma única conexão, e consequentemente todos os vértices de uma conexão devem ser associados à outra. Para os outros casos possíveis para adição de uma ponte (ver Subseção 4.2.1), a atualização dos índices e índices revertidos pode ser realizada com complexidade constante, visto que apenas um ou dois vértices passarão a ser ligados a um novo vértice no grafo $BC(G)$.

A nível de grafo bloco conexão, o pior caso também é linear e também ocorre com a união de duas conexões. Ao realizar a fusão das conexões é necessário transferir todas as adjacências de uma delas para a outra, além de remover a conexão do grafo $BC(G)$ e adicioná-la à estrutura que armazena as conexões disponíveis. Como mencionado anteriormente, a quantidade de blocos no grafo bloco conexão é $O(n)$, e portanto a transferência de adjacências também é linear no pior caso. Para os casos em que um dos vértices envolvidos é um bloco, a atualização do grafo $BC(G)$ é mais simples, sendo necessário apenas ligar o bloco à conexão (o vértice ligado ao bloco passa a ser ligado à conexão). Quando os dois vértices são blocos a operação também é simples, sendo apenas necessário criar uma nova conexão (a nova conexão é obtida a partir da estrutura de conexões disponíveis) e ligá-la aos dois blocos envolvidos (os dois vértices passam a ser ligados à nova conexão).

6.1.2 ADIÇÃO DE CORDA

Para realizar a análise se a aresta a ser adicionada é uma corda deve-se inicialmente calcular o caminho entre os vértices aos quais u e v estão ligados no grafo bloco conexão, ou seja, $\bar{u}$ e $\bar{v}$, respectivamente, denotado $p_{\bar{u}\bar{v}}$, conforme o algoritmo da Subseção 5.3.4. Como é sabido, o grafo bloco conexão é uma floresta e sua quantidade de vértices é limitada em $O(n)$. O caminho $p_{\bar{u}\bar{v}}$ é obtido a partir de um DFS e, portanto, com complexidade linear. Identificados os casos em que é feita a adição de uma corda, conforme Subseção 5.2.2, deve-se calcular o caminho mínimo p_{uv} para posterior atualização das estruturas. BFS é utilizado para calcular p_{uv} a nível de grafo e, portanto, é obtido com complexidade linear, $O(n)$, visto que, pelo Teorema 2.2, a quantidade de arestas num grafo *outerplanar*

é no máximo $2n - 3$.

Vale ressaltar que a adição de uma corda pode não ser possível e que para realizar essa verificação, deve-se analisar se os vértices u e v possuem uma face em comum (além da face externa). O fato de que a lista de faces associadas a um dado vértice está sempre ordenada faz com que a complexidade dessa operação seja no pior caso $O(n)$.

Finalmente, deve-se inserir a nova aresta na lista de adjacências de G, adicionar a nova aresta na estrutura de cordas do grafo G ($O(1)$ para inserção no conjunto não ordenado) e também inserir a nova face gerada ao final da lista (ordenada) de faces para todos os vértices de p_{uv}, o que acarreta em uma complexidade $O(n)$ no pior caso.

6.1.3 ADIÇÃO DE ARESTA HAMILTONIANA

Para o caso da adição de uma aresta hamiltoniana é necessário utilizar o caminho $p_{\bar{u}\bar{v}}$ entre os vértices $\bar{u}$ e $\bar{v}$ em $BC(G)$ além do caminho mínimo p_{uv} entre os vértices u e v em G, ambos já calculados previamente (Subseção 6.1.2). Assim como no caso da inserção de uma corda, é possível que a adição da aresta hamiltoniana invalide a *outerplanaridade*. Sendo assim, faz-se necessário verificar se a aresta de fato pode ser inserida, utilizando as condições necessárias e suficientes da Subseção 4.2.3. Para as duas condições do teorema a verificação pode ser feita com complexidade linear. Para analisar se uma aresta é fronteira entre faces internas é necessário saber se uma dada aresta é corda. Para identificar se duas arestas pertencem a blocos consecutivos é preciso obter a informação sobre a qual bloco determinada aresta pertence (se existir algum), o que pode ser feito analisando as adjacências do bloco ou conexão no qual o vértice que está sendo analisado no momento está ligado.

A atualização das estruturas a nível de grafo bloco conexão também é realizada com complexidade $O(n)$. Conforme a Subseção 4.2.3, existem dois casos a serem analisados ao adicionar uma aresta hamiltoniana.

Para o caso em que os vértices u e v estão ligados à mesma conexão, é necessário atualizar as possíveis novas conexões que serão adjacentes ao novo bloco que será criado. Para isso, é necessário realizar uma busca em largura nos vértices ligados à conexão a partir de cada vértice que pertencerá ao novo bloco, de modo a encontrar os vértices que farão parte de cada nova conexão. Essas buscas possuirão complexidade linear no pior caso, visto que explorarão todos os vértices da conexão uma única vez. Ao descobrir os vértices que formarão cada conexão, é necessário ainda ligá-la ao novo bloco e aos blocos

devidos que eram adjacentes à conexão antiga.

Para o caso em que os vértices estão ligados a vértices distintos em $BC(G)$ o procedimento é semelhante, porém deve-se percorrer cada bloco ou conexão no caminho entre $\bar{u}$ e $\bar{v}$. Para o caso em que um vértice é conexão, o procedimento é similar ao descrito anteriormente. Para o caso em que o vértice é bloco, faz-se necessário apenas atualizar a aresta de p_{uv} que passa a ser uma corda e anteriormente era uma aresta hamiltoniana. A complexidade total da operação é $O(n)$ no pior caso.

Em paralelo, é necessário atualizar as estruturas de interligação entre o grafo e o grafo bloco conexão. Os vértices envolvidos no procedimento citado anteriormente estarão ligados a um novo bloco ou a novas conexões, então faz-se necessário realizar a atualização do $indice$ e consequentemente do $indice_rev$.

Ao final da operação, é necessário inserir a nova aresta nas adjacências de G e também na estrutura que armazena as arestas hamiltonianas.

Notar que, para qualquer um dos possíveis casos, a complexidade no pior caso pode ser linear. Sendo assim, a complexidade para a operação de inserção de aresta é $O(n)$ no pior caso.

6.2 REMOÇÃO DE ARESTA

Pelo Lema 4.1, sabe-se que a remoção de uma aresta pertencente ao grafo G sempre é possível. Sendo assim, para qualquer um dos tipos de aresta, a primeira operação a ser realizada é a remoção da aresta (u, v) do grafo G.

6.2.1 REMOÇÃO DE PONTE

No caso da remoção de uma ponte no grafo G tem-se que a complexidade de execução da operação é também linear no pior caso. Ao remover a aresta (u, v), uma componente conexa é dividida em duas, e as estruturas associadas devem ser atualizadas. Tendo em vista a criação de uma nova componente conexa, é necessário atualizar a informação dos componentes conexo dos vértices a nível de grafo. Inicialmente, para obter os vértices que terão sua componente conexa alterada, é necessário realizar uma busca em largura no grafo. A complexidade dessa operação é, no pior caso, $O(n)$. Em seguida, com a informação dos vértices, é necessário atualizar a componente conexa associada a todos eles.

Como alguns dos vértices têm sua componente conexa alterada e, no caso em questão, se ligam a uma nova conexão, é necessário realizar a atualização dos índices e índices revertidos para os vértices que estavam ligados à conexão antiga e passam a se ligar à nova conexão que será criada. Essa operação também depende do número de vértices a serem atualizados, o que representa uma complexidade linear no pior caso.

A operação de remoção de uma ponte também altera as estruturas a nível de grafo bloco conexão. Tendo em vista a criação de uma nova conexão, visto que a componente conexa se divide em duas, é necessário atualizar as adjacências. A transferência das adjacências para a nova conexão também apresenta custo linear no pior caso pois as adjacências (blocos) cujos atributos de aresta são os vértices que passaram a se ligar à nova conexão devem passar a ser adjacentes a essa nova conexão. Como a quantidade de blocos em $BC(G)$ é $O(n)$, tem-se que essa operação também possui custo linear.

6.2.2 REMOÇÃO DE CORDA

De forma semelhante à executada no caso da inserção de arestas, caso a aresta a ser removida não seja uma ponte, é necessário inicialmente calcular o caminho mínimo p_{uv} utilizando uma BFS. Essa operação, tendo em vista as características do grafo *outerplanar*, possui complexidade linear, $O(n)$. Em seguida, a estrutura que armazena as cordas é consultada (custo $O(n)$ no pior caso). Caso a aresta a ser removida seja de fato uma corda, assim como no caso da inserção de arestas, não é necessário atualizar as estruturas de dados a nível de grafo bloco conexão, visto que a estrutura de $BC(G)$ não se modifica.

Inicialmente é feita a remoção da aresta da estrutura que armazena as cordas, o que pode ter custo linear no pior caso. Em seguida, deve-se atualizar a estrutura de faces associadas aos vértice, tendo em vista que a remoção de uma corda acarreta na fusão de duas faces. Sendo assim, é necessário percorrer os vértices do caminho p_{uv}, remover uma das faces comuns a u e v e adicionar a outra face comum na estrutura (sabe-se que além da face externa, u e v possuem duas faces em comum). A complexidade dessa operação é linear, pois é dependente do número de faces do grafo. Tendo em vista que a quantidade de faces num grafo *outerplanar* é $O(n)$, tem-se que o custo da operação é linear no pior caso.

6.2.3 REMOÇÃO DE ARESTA HAMILTONIANA

Para o caso da remoção de uma aresta hamiltoniana realiza-se inicialmente a atualização das estruturas a nível de grafo. Primeiramente remove-se a aresta (u, v) da estrutura que armazena as arestas hamiltonianas e em seguida remove-se a face comum entre os vértices u e v de todos os vértices que a possuem. Essas operações possuem complexidade linear tendo em vista que a quantidade de faces é $O(n)$ no grafo *outerplanar*.

O próximo passo é atualizar as estruturas a nível de grafo bloco conexão. Isso é feito percorrendo os vértices do caminho mínimo p_{uv} calculado previamente (Subseção 6.2.2). Em cada um dos casos (Subseção 4.3.3) deve-se atualizar $BC(G)$ de forma distinta. Para o caso em que há uma sequência contígua de arestas hamiltonianas, é necessário criar uma única conexão (se algum dos vértices da sequência já for ligado a uma conexão é preciso realizar uma fusão de conexões em uma única conexão). Para isso, deve-se percorrer todos esses vértices e atualizar os índices e índices revertidos. Tal operação é linear no pior caso.

No caso em que há cordas consecutivas, o vértice comum às cordas consecutivas será um ponto de articulação ao remover a aresta (u, v). Logo, é necessário criar uma conexão exclusiva para o vértice de articulação (se o mesmo já não for ligado a uma conexão). Basta atualizar o índice e o índice revertido, além de obter uma nova conexão disponível, se necessário. Todas as operações podem ser feitas com custo constante, porém os outros casos no percurso de p_{uv} possuem custo linear.

Finalmente, há o caso em que há uma corda em p_{uv}. Nesse caso, é necessário atualizar o tipo da aresta, que passa a ser uma aresta hamiltoniana e construir o novo bloco. Seja a corda $(x, y), x, y \in V$. Para formar o novo bloco, é preciso explorar todos os vértices que farão parte desse bloco, de modo a atualizar os índices e índices revertidos. Uma maneira de descobrir os vértices é realizar uma busca em largura de modo a encontrar o caminho mínimo p_{xy} no grafo que contém apenas arestas hamiltonianas. O caminho p_{xy} conterá apenas os vértices que farão parte do novo bloco. Com isso, é possível atualizar as estruturas de dados associadas. Tendo em vista a execução do BFS, o custo da operação é $O(n)$ no pior caso.

Note que, para qualquer um dos possíveis casos, a complexidade no pior caso pode ser linear. Sendo assim, a complexidade para a operação de remoção de aresta é $O(n)$ no pior caso.

7 CONSIDERAÇÕES FINAIS

7.1 CONCLUSÃO

Existe uma extensa bibliografia no estudo de propriedades específicas de grafos, como a conectividade ou a planaridade. Dentre elas, alguns trabalhos abordam aspectos teóricos acerca de algoritmos dinâmicos, assim como informações sobre a implementação dos mesmos. Esses algoritmos podem ser classificados de acordo com o tipo de solução obtida em determinísticos ou probabilísticos. Os algoritmos dinâmicos determinísticos são analisados em termos de complexidade do pior caso ou complexidade amortizada. Já os algoritmos dinâmicos probabilísticos são do tipo que produzem resposta que pode ser incorreta com uma certa probabilidade p. Alguns desses algoritmos dinâmicos foram apresentados no Capítulo 3.

Nesta dissertação foi proposta uma abordagem para a implementação do algoritmo determinístico e completamente dinâmico de manutenção da *outerplanaridade* baseando-se no trabalho descrito em ARAUJO (2004). A ideia proposta nesse trabalho de manter dois níveis de informação, o grafo e o grafo bloco conexão, permite analisar a validade da *outerplanaridade* do grafo obtido após a adição de arestas e mantém informação sobre a imersão do grafo *outerplanar*. No Capítulo 4 foi analisado esse algoritmo, identificando as propriedades necessárias para manter a *outerplanaridade* do grafo sujeito a inserção ou remoção de arestas. No Capítulo 5 foram descritas as estruturas de dados usadas, sua inicialização e os detalhes da implementação do algoritmo analisado no Capítulo 4 usando essas estruturas. No final desse capítulo foi apresentado o pseudocódigo das operações de inserção e remoção de arestas utilizado para implementar o algoritmo. Finalmente, o Capítulo 6 traz a análise da complexidade do pior caso do algoritmo implementado, segundo a descrição do Capítulo 5.

Conforme mencionado neste trabalho, a manutenção de algumas das estruturas de dados presentes no algoritmo implementado requerem o cálculo de caminhos mínimos e caminhos entre pares de vértices do grafo e do grafo bloco conexão. Como consequência do cálculo desses caminhos, as operações de inserção e remoção de arestas apresentam complexidade do pior caso linear no número de arestas do grafo. O fato de que a quantidade de arestas num grafo *outerplanar* é $O(n)$ garante a complexidade para o cálculo dos caminhos $O(n)$ e portanto a manutenção da *outerplanaridade* também em $O(n)$.

7.2 TRABALHOS FUTUROS

Propomos como trabalho futuro melhorar a complexidade do algoritmo abordado nesta dissertação. Para tal fim é necessário melhorar a complexidade de algumas das operações apresentadas no Capítulo 5.

Dentre as operações que apresentam complexidade linear, podemos mencionar:

- Cálculo de caminhos e caminhos mínimos num grafo;

- Atualização em massa do componente conexo associado a um determinado vértice em operações de inserção ou remoção de arestas do tipo ponte;

- Verificação de faces em comum (percorrendo a lista de faces ordenadas) nas operações de inserção ou remoção de cordas;

- Realização de buscas no grafo para atualização do grafo bloco conexão.

Sendo assim, o maior gargalo para a complexidade linear do algoritmo é a necessidade de explorar o grafo em determinadas operações.

8 REFERÊNCIAS BIBLIOGRÁFICAS

ALSTRUP, S., HOLM, J. e LICHTENBERG, K. **Maintaining information in fully dynamic trees with top trees**. *TALG*, págs. 243–264, 2005.

ARAUJO, L. H. C. **Algoritmos Dinâmicos para Manutenção de Grafos Cordais e Periplanares**. *Universidade Federal do Rio de Janeiro*, 2004.

BERNHAR, F. e KAINEN, P. C. **The Book Thickness of a Graph**. *Journal of Combinatorial Theory, Series B 27*, págs. 320–331, 1979.

BONDY, J. A. e MURTY, U. S. R. **Graph Theory**. Springer, 2010.

CAMPBELL, W. M., DAGLI, C. K. e WEINSTEIN, C. J. **Social Network Analysis with Content and Graphs**. *Lincoln Laboratory Journal, Vol. 20, No. 1*, págs. 62–81, 2013.

CHARTRAND, G. T. e HARARY, F. **Planar permutation graphs**. *Ann. Inst. Henri Poincare 3*, págs. 433–438, 1967.

CHUNG, F. R. K., LEIGHTON, F. T. e ROSENBERG, A. L. **Embedding graphs in books. A layout problem with applications to VLSI design**. *SIAM J Alg. Disc. Meth. 8*, págs. 33–58, 1987.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. e STEIN, C. **Introduction to Algorithms**, volume 2. MIT Press, 2001.

CPLUSPLUS.COM. **unordered_map - C++ Reference**, 2017a. URL `http://www.cplusplus.com/reference/unordered_map/unordered_map/`. Acessado em: 15/03/2017.

CPLUSPLUS.COM. **unordered_set - C++ Reference**, 2017b. URL `http://www.cplusplus.com/reference/unordered_set/unordered_set/`. Acessado em: 15/03/2017.

DE RIDDER, H. N. **Outerplanar Graphs**, 2017. URL `http://www.graphclasses.org/classes/gc_110.html`. Acessado em: 01/07/2017.

DI BATTISTA, G. e FRATI, F. **Small Area Drawings of Outerplanar Graphs**. *Algorithmica 54*, págs. 25–53, 2009.

DI BATTISTA, G. e TAMASSIA, R. **Incremental planarity testing**. *Foundations of Computer Science*, págs. 436–441, 1989.

DI BATTISTA, G. e TAMASSIA, R. **On-line graph algorithms with spqr-trees**. *Automata, Languages and Programming*, págs. 598–611, 1990.

DIAS, M. V. **Uma abordagem para detecção de comunidades em redes complexas baseada em informações de dominio**. *Dissertação de Mestrado, Programa de Engenharia de Defesa, Instituto Militar de Engenharia*, 2016.

EISNER, J. **State-of-the-Art Algorithms for Minimum Spanning Trees**. 1997.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F. e NISSENZWEIG, A. **Sparsification, A technique for speeding up dynamic graph algorithms**. *J. ACM 44, 5*, págs. 669–696, 1997.

FLEISCHNER, H. J., GELLER, D. P. e HARARY, F. **Outerplanar graphs and weak duals**. *J. Indian Math*, págs. 215–219, 1974.

FRATI, F. **Straight-line drawings of outerplanar graphs in O(dn**$\log n$**) area**. 2012.

FREDERICKSON, G. N. **Data structures for on-line updating of minimum spanning trees, with applications**. *SIAM J. Comput. 14, 4*, págs. 781–798, 1985.

FREDMAN, M. e HENZINGER, M. R. **Lower bounds for fully dynamic connectivity problems in graphs**. *Algorithmica 22, 3*, págs. 351–362, 1998.

GARG, A. e RUSU, A. **Area-efficient planar straight-line drawings of outerplanar graphs**. *Discrete Appl. Math. 155(9)*, págs. 1116–1140, 2007.

HENZINGER, M. R. e KING, V. **Maintaining minimum spanning trees in dynamic graphs**. *Lecture Notes in Computer Science, vol. 1256. Springer-Verlag, New York*, págs. 594–604, 1997a.

HENZINGER, M. R. e KING, V. **Randomized fully dynamic graph algorithms with polylogarithmic time per operation**. *J. ACM, 46, 4*, págs. 502–516, 1999.

HENZINGER, M. R. e THORUP, M. **Sampling to provide or to bound: With applictions to fully dynamic graph algorithms**. *Random Struct. Algorithms 11*, págs. 369–379, 1997b.

HOLM, J., LICHTENBERG, K. e THORUP, M. **Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity**. *J. ACM, 48, 4*, págs. 723–760, 2001.

HOLM, J. e ROTENBERG, E. **Dynamic Planar Embeddings of Dynamic Graphs**. *STACS*, págs. 434–446, 2015.

HONG, S. H., EADES, P., KATOH, N., LIOTTA, G., SCHWEITZER, P. e SUZUKI, Y. **A Linear-Time Algorithm for Testing Outer-1-Planarity**. *Algorithmica 72*, págs. 1033–1054, 2015.

HOPCROFT, J. e TARJAN, R. **Efficient planarity testing**. *J. ACM 21, 4*, págs. 549–568, 1974.

ITALIANO, G., GALIL, Z. e SARNAK, N. **Fully dynamic planarity testing with applications**. *J. ACM 46*, págs. 28–91, 1999.

ITALIANO, G. F., LA POUTRE, J. A. e RAUCH, M. H. **Fully dynamic planarity testing in planar embedded graphs**. *ESA*, 1993.

KAPRON, B. M., KING, V. e MOUNTJOY, B. **Dynamic graph connectivity in polylogarithmic worst case time**. *ACM-SIAM*, págs. 1131–1142, 2013.

LA POUTRE, J. A. **Alpha-algorithms for incremental planarity testing**. *STOC*, págs. 706–715, 1994.

MILTERSEN, P. B., SUBRAMANIAN, S., VITTER, J. S. e TAMASSIA, R. **Complexity models for incremental computation**. *Theoret. Comput. Sci. 130, 1*, págs. 203–236, 1994.

MITCHELL, S. L. **Linear algorithms to recognize outerplanar and maximal outerplanar graphs**. *Inf. Proc. Letters 9*, págs. 229–232, 1979.

OF NEW YORK, T. C. **NYC Open Data**, 2017. URL `https://data.cityofnewyork.us/Public-Safety/`. Acessado em: 01/07/2017.

PALUCKA, E. L., SHEPHERDB, H. e ARONOWC, P. M. **Changing climates of conflict: A social network experiment in 56 schools**. *PNAS, January 19, Vol. 113, No. 3*, págs. 566–571, 2016.

PATRASCU, M. e DEMAINE, E. D. **Lower bounds for dynamic connectivity**. *STOC*, págs. 546–553, 2005.

SLEATOR, D. D. e TARJAN, R. E. **A Data Structure for Dynamic Trees**. *Journal. Comput. Syst. Sci., 28, 3*, págs. 362–391, 1983.

SYSLO, M. **Characterization of outerplanar graphs**. *Report Nr N-3, Institute of Computer Science, University of Wroctaw*, 1976.

SYSLO, M. **Characterization of outerplanar graphs**. *Discrete Mathematics 26*, págs. 47–53, 1979.

SZWARCFITER, J. L. **Grafos e Algoritmos Computacionais**. Editora Campus, 1986.

TARJAN, R. E. **Efficiency of a Good But Not Linear Set Union Algorithm**. *J. ACM 22, 2*, págs. 215–225, 1975.

TARJAN, R. E. **Data Structures and Network Algorithms**, volume 1. Society for Industrial and Applied Mathematics, 1984.

THORUP, M. **Decremental dynamic connectivity**. *J. Algorithms 33*, págs. 229–243, 1999.

THORUP, M. **Near-optimal fully-dynamic graph connectivity**. *STOC*, págs. 343–350, 2000.

VALIANT, L. G. **Universality considerations in VLSI circuits**. *IETrans. Comput. 30(2)*, págs. 135–140, 1981.

WESTBROOK, J. **Fast incremental planarity testing**. *Automata, Languages and Programming, 623*, págs. 342–353, 1992.

WULFF-NILSEN, C. **Faster deterministic fully-dynamic graph connectivity**. *SODA*, págs. 1757–1769, 2013.

9 ANEXOS

ANEXO 1: IMPLEMENTAÇÃO DO ALGORITMO

```
#include <bits/stdc++.h>
#define MAXN 10000
#define mp make_pair
#define ff first
#define ss second
using namespace std;
typedef pair<int, int> pii;
typedef pair<int, pii> piii;

//Esse algoritmo resolve o problema de manutencao da outerplanaridade
//em um grafo dinamico.

const int BLOCO=0, CONEXAO=1;
const int HAMILTONIANO=1, CORDA=2;

//Numero de vertices do grafo.
int n;
//Armazena as adjacencias do vertice do grafo G em uma arvore.
set<int> adj[MAXN];
//Armazena as adjacencias do vertice do grafo BC(G) em uma arvore.
//Armazena tambem o atributo da aresta (vertice comum ao bloco e a conexao).
//(0-bloco/1-conexao)
//Informacao e uma tripla ordenada
//(atributo da aresta, bloco/conexao, id do bloco/conexao).
set<piii> bc[2][MAXN];
//Armazena as arestas (u, v), u < v, do grafo G
//que pertencem ao ciclo hamiltoniano.
set<pii> hamiltoniano;
//Armazena as arestas (u, v), u < v, do grafo G que sao cordas.
set<pii> corda;
//Armazena o conjunto de faces associada a cada vertice do grafo G.
//Invariante: a lista de faces deve estar sempre ordenada para que
//seja possivel realizar a checagem de faces comuns de modo mais eficiente.
list<int> faces[MAXN];
//Variavel que controla um numero de face disponivel para o grafo G.
//Esse numero sempre cresce. Se uma face deixa de existir, o numero
//identificador daquela face e perdido ate o fim do algoritmo. Se por um acaso
```

```
//aquela face voltar a existir, ela sera identificada com um novo numero,
//determinado por essa variavel.
int id_faces;
//Armazena a qual conjunto cada vertice pertence. E a estrutura
//utilizada na manutencao das componentes conexas para adicao de pontes.
int conjunto[MAXN];
//Estrutura utilizada caso esteja-se utilizando Union-Find.
int rank[MAXN];
//Armazena o pai de cada vertice numa busca em largura.
int pai[MAXN];
//Armazena informacao sobre vertices ja visitados na busca em largura
int vis[MAXN];
//Armazena o pai de cada vertice na busca em largura
//no grafo de arestas hamiltonianas.
int pai_h[MAXN];
//Armazena a qual bloco ou conexao o vertice esta ligado
//O par armazena a informacao
//(0-bloco/1-conexao, identificador do bloco/conexao)
pii indice[MAXN];
//Estrutura inversa do indice para controle de quais vertices pertencem a
//um dado par (bloco/conexao, id do bloco/conexao)
map<pii, set<int>> indice_rev;
//Armazena o pai de cada vertice no grafo BC(G). O grafo BC(G) e armazenado como
//uma dupla (bloco/conexao, id do bloco/conexao). A informacao do pai nesse caso
//guardara outra informacao, alem do vertice pai: o atributo da aresta, ou seja,
//o vertice comum ao bloco e a conexao que estao sendo analisados.
//pai_bc[tipo][id bloco/conexao]=
//(atributo da aresta, bloco/conexao, id do bloco/conexao).
piii pai_bc[2][MAXN];
//Armazena blocos, conexoes e conjuntos disponiveis
stack<int> bloco, conexao, conjuntos;

//Funcao de debug. Serve para analisar as estruturas com o decorrer das operacoes.
void print() {
    //Lista de adjacencias do grafo G
    for(int i=1; i<=n; i++) {
        if(adj[i].empty()) continue;
        printf("Adjacencias do vertice %d:\n", i);
        for(set<int>::iterator it=adj[i].begin(); it!=adj[i].end(); it++) {
            printf("%d ", *it);
        }
        printf("\n");
```

```
}
//Arestas hamiltonianas
if(!hamiltoniano.empty()) {
    printf("Arestas hamiltonianas:\n");
    for(set<pii>::iterator it=hamiltoniano.begin();
    it!=hamiltoniano.end(); it++) {
        printf("(%d, %d)\n", (*it).ff, (*it).ss);
    }
}
//Cordas
if(!corda.empty()) {
    printf("Cordas:\n");
    for(set<pii>::iterator it=corda.begin(); it!=corda.end(); it++) {
        printf("(%d, %d)\n", (*it).ff, (*it).ss);
    }
}
//Faces(u)
for(int i=1; i<=n; i++) {
    if(faces[i].empty()) continue;
    printf("Faces do vertice %d\n", i);
    for(list<int>::iterator it=faces[i].begin(); it!=faces[i].end(); it++) {
        printf("%d ", *it);
    }
    printf("\n");
}
//Conjunto associado a cada vertice
printf("Conjuntos:\n");
for(int i=1; i<=n; i++) printf("%d ", conjunto[i]);
printf("\n");
//Grafo BC(G)
printf("Grafo BC(G):\n");
for(int i=0; i<2; i++) for(int j=1; j<=n; j++) if(!bc[i][j].empty()) {
    printf("Adjacencias %s %d:\n", (!i ? "BLOCO": "CONEXAO"), j);
    for(set<piii>::iterator it=bc[i][j].begin(); it!=bc[i][j].end(); it++) {
        printf("(atributo=%d tipo=%s id=%d)\n",
        (*it).ff, ((*it).ss.ff==0 ? "BLOCO" : "CONEXAO"), (*it).ss.ss);
    }
}
//Indices de BC(G)
printf("Indices de BC(G):\n");
for(int i=1; i<=n; i++) {
    printf("Vertice -> %d %s %d\n", i,
```

```
        ((!indice[i].ff) ? "BLOCO" : "CONEXAO"), indice[i].ss);
    }
    //Indice rev
    printf("Indice_rev:\n");
    for(map<pii, set<int> >::iterator it=indice_rev.begin();
    it!=indice_rev.end(); it++) {
        set<int> temp=(*it).ss;
        if(temp.empty()) continue;
        printf("%s %d\n", ((*it).ff.ff==0 ? "BLOCO" : "CONEXAO"), (*it).ff.ss);
        printf("Vertices:");
        for(set<int>::iterator it1=temp.begin(); it1!=temp.end(); it1++) {
            printf(" %d", *it1);
        }
        printf("\n");
    }
}

//Realiza a inicializacao das estruturas necessarias para a execucao
//do algoritmo.
void inicializa() {
    for(int i=1; i<=n; i++) {
        conjunto[i]=i;
        //O rank e utilizado se for optado utilizar o algoritmo Union-Find.
        //rank[i]=0;
        //Associando vertice a conexao dele
        indice[i]=mp(CONEXAO, i);
        indice_rev[pii(CONEXAO, i)].insert(i);
    }
    //Adicionando os blocos disponiveis. Inicialmente todos os
    //vertices sao conexoes.
    //Adicionando na ordem reversa pois estamos usando stack.
    for(int i=n; i>=1; i--) bloco.push(-i);
    //Por uma questao de preferencia, o id inicial das faces e 1
    id_faces=1;
}

//Procedimento de busca em largura auxiliar, que armazena o menor caminho
//entre os dois vertices 's' e 'd', distintos.
//Por seguranca, retorna uma flag que indica
//se existe caminho entre os vertices.
int bfs(int s, int d) {
    queue<int> q;
```

```
    memset(vis, 0, sizeof(vis));
    //Nao possui nenhum pai, pois e o vertice de origem da busca.
    pai[s]=-1;
    vis[s]=1;
    q.push(s);
    while(!q.empty()) {
        int u=q.front();
        q.pop();
        for(set<int>::iterator it=adj[u].begin(); it!=adj[u].end(); it++) {
            int v=(*it);
            if(!vis[v]) {
                vis[v]=1;
                pai[v]=u;
                if(v==d) return 1;
                q.push(v);
            }
        }
    }
    return 0;
}

//Procedimento que calcula a distancia entre dois nos no grafo BC(G).
//Utilizando DFS pois o grafo e uma arvore e o caminho entre os nos e unico.
//O no 'u' e o vertice atual, 'p' e o vertice pai, 'd' e o vertice destino e
//attr e o atributo de aresta de BC(G) na aresta (u, p).
int dfs_bc(pii u, pii p, pii d, int attr) {
    pai_bc[u.ff][u.ss]=piii(attr, p);
    if(u.ff==d.ff && u.ss==d.ss) return 0;
    int u_tipo=u.ff;
    int u_id=u.ss;
    for(set<piii>::iterator v=bc[u_tipo][u_id].begin();
    v!=bc[u_tipo][u_id].end(); v++) {
        int attr_v=(*v).first;
        int v_tipo=(*v).second.first;
        int v_id=(*v).second.second;
        //Deve-se explorar os nos filhos apenas.
        if(v_tipo==p.ff && v_id==p.ss) continue;
        int ret=dfs_bc((*v).second, u, d, attr_v);
        if(ret!=-1) return 1+ret;
    }
    return -1;
}
```

```
//Encontra o conjunto do elemento x no Union-Find.
//OBS: So utilizar essa funcao para o caso incremental.
int encontra_conjunto(int x) {
    if(conjunto[x]==x) return x;
    return conjunto[x]=encontra_conjunto(conjunto[x]);
}

//Une conjunto do elemento y ao conjunto do elemento y no Union-Find.
//OBS: So utilizar essa funcao para o caso incremental.
void unir(int x, int y) {
    x=encontra_conjunto(x);
    y=encontra_conjunto(y);
    if(x==y) return;
    if(rank[x]>rank[y]) conjunto[y]=x;
    else {
        conjunto[x]=y;
        if(rank[x]==rank[y]) rank[y]++;
    }
}

//Verifica se dois vertices pertencem ou nao ao mesmo conjunto.
//A verificacao depende do algoritmo utilizado.
bool pertence_mesmo_conjunto(int u, int v) {
    return conjunto[u]==conjunto[v];
    //Caso esteja-se utilizando Union-Find (incremental), usar:
    //return encontra_conjunto(u)==encontra_conjunto(v);
}

//Realiza a uniao dos conjuntos que contem os elementos u e v.
//Mais especificamente, todos os elementos do conjunto que contem v
//sao transportados para o conjunto que contem u.
void unir_conjuntos(int u, int v) {
    int conj_v=conjunto[v];
    for(int i=1; i<=n; i++) if(conjunto[i]==conj_v) conjunto[i]=conjunto[u];
    //Adicionando conjunto aos conjuntos disponiveis
    conjuntos.push(-conj_v);
    //Caso esteja-se utilizando Union-Find (incremental), usar:
    //unir(u, v);
}

int encontrar_face_comum(int u, int v, int pos) {
```

```
    //Encontra a 'pos'-esima face comum aos vertices u e v.
    int ind=1;
    list<int>::iterator it_u=faces[u].begin(), it_v=faces[v].begin();
    //Nao ha face comum inicialmente
    while(1) {
        if(it_u==faces[u].end() || it_v==faces[v].end()) break;
        if((*it_u)<(*it_v)) it_u++;
        else if((*it_u)>(*it_v)) it_v++;
        else {
            //Encotramos uma face comum
            if(ind==pos) return (*it_u);
            ind++;
            it_v++;
        }
    }
    //O algoritmo nao encontrou nenhuma face em comum.
    return 0;
}

//Insere a face na posicao correta de faces(u).
//Lembrar que a estrutura faces(u) esta ordenada.
void inserir_face(int face, int u) {
    list<int>::iterator it;
    for(it=faces[u].begin(); it!=faces[u].end(); it++) {
        if((*it)>face) break;
    }
    faces[u].insert(it, face);
}

//Verifica se determinado o vertice u contem a face 'face'.
int contem_face(int u, int face) {
    return find(faces[u].begin(), faces[u].end(), face)!=faces[u].end();
}

//Insere uma nova face na lista de faces do vertice u.
void adicionar_face(int face, int u) {
    faces[u].push_back(face);
}

//Remove uma face da lista de faces do vertice u.
void remover_face(int face, int u) {
    faces[u].remove(face);
```

```
}

//Resolve caso 5.3.1
void adicionar_ponte(int u, int v) {
    //Sempre e possivel adicionar pontes sem violar a outerplanaridade.
    //Atualizacao das estruturas.
    unir_conjuntos(u, v);
    //Adicionando as arestas no grafo G.
    adj[u].insert(v);
    adj[v].insert(u);
    //Tipos dos vertices aos quais u e v estao ligados (bloco ou conexao)
    int tipo_u=indice[u].ff, tipo_v=indice[v].ff;
    int u_id=indice[u].ss, v_id=indice[v].ss;
    //Analise dos casos
    if(tipo_u==BLOCO && tipo_v==BLOCO) {
        //Gera uma nova conexao
        int conexao_disponivel=-conexao.top();
        conexao.pop();
        //Removendo aparicao de u e v do indice_rev
        indice_rev[indice[u]].erase(u);
        indice_rev[indice[v]].erase(v);
        indice[u].ff=CONEXAO;
        indice[v].ff=CONEXAO;
        indice[u].ss=indice[v].ss=conexao_disponivel;
        //Adicionando u e v no novo indice_rev
        indice_rev[indice[u]].insert(u);
        indice_rev[indice[v]].insert(v);
        //Atualiza BC(G)
        bc[CONEXAO][conexao_disponivel].insert(piii(u, pii(BLOCO, u_id)));
        bc[CONEXAO][conexao_disponivel].insert(piii(v, pii(BLOCO, v_id)));
        bc[BLOCO][u_id].insert(piii(u, pii(CONEXAO, conexao_disponivel)));
        bc[BLOCO][v_id].insert(piii(v, pii(CONEXAO, conexao_disponivel)));
    }
    else if(tipo_u==BLOCO && tipo_v==CONEXAO) {
        //Removendo aparicao de u do indice_rev
        indice_rev[indice[u]].erase(u);
        //Vertice u passa a ser da conexao a qual v e ligado
        indice[u].ff=CONEXAO;
        indice[u].ss=indice[v].ss;
        //Adicionando u no indice_rev
        indice_rev[indice[u]].insert(u);
        //Atualiza BC(G)
```

```
        bc[CONEXAO][v_id].insert(piii(u, pii(BLOCO, u_id)));
        bc[BLOCO][u_id].insert(piii(u, pii(CONEXAO, v_id)));
    }
    else if(tipo_u==CONEXAO && tipo_v==BLOCO) {
        //Removendo aparicao de v do indice_rev
        indice_rev[indice[v]].erase(v);
        //Vertice v passa a ser da conexao a qual u e ligado
        indice[v].ff=CONEXAO;
        indice[v].ss=indice[u].ss;
        //Adicionando v no indice_rev
        indice_rev[indice[v]].insert(v);
        //Atualiza BC(G)
        bc[CONEXAO][u_id].insert(piii(v, pii(BLOCO, v_id)));
        bc[BLOCO][v_id].insert(piii(v, pii(CONEXAO, u_id)));
    }
    else {
        //Ambos sao conexoes. Une as conexoes e libera a outra.
        if(indice[u].ss>indice[v].ss) swap(u, v), swap(u_id, v_id);
        for(int i=1; i<=n; i++) if(indice[i].ss==v_id) {
            //Removendo i do indice_rev
            indice_rev[indice[i]].erase(i);
            indice[i].ss=indice[u].ss;
            //Adicionando i no indice_rev
            indice_rev[indice[i]].insert(i);
        }
        conexao.push(-v_id);
        //Atualiza BC(G)
        //Passando todas as adjacencias de v_barra para u_barra.
        //Passando todas as adjacencias dos vertices
        //adjacentes a v_barra para u_barra
        for(set<piii>::iterator it=bc[CONEXAO][v_id].begin();
        it!=bc[CONEXAO][v_id].end(); it++) {
            pii x_barra=(*it).ss;
            bc[CONEXAO][u_id].insert(*it);
            bc[x_barra.ff][x_barra.ss].erase(piii((*it).ff, pii(CONEXAO, v_id)));
            bc[x_barra.ff][x_barra.ss].insert(piii((*it).ff, pii(CONEXAO, u_id)));
        }
        //Limpa conexao que foi liberada.
        bc[CONEXAO][v_id].clear();
    }
    printf("Caso 5.3.1. Aresta (%d, %d) inserida com sucesso\n", u, v);
}
```

```
//Dados dois vertices, encontra, se houver, o bloco comum aos dois vertices
int encontrar_bloco(int u, int v) {
    //Estrutura que armazena os blocos aos quais u pertence.
    set<int> blocos;
    int tipo=indice[u].ff, u_id=indice[u].ss;
    if(tipo==BLOCO) blocos.insert(u_id);
    else {
        for(set<piii>::iterator it=bc[tipo][u_id].begin();
        it!=bc[tipo][u_id].end(); it++) {
            int attr=(*it).ff;
            if(attr==u) blocos.insert((*it).ss.ss);
        }
    }
    tipo=indice[v].ff;
    int v_id=indice[v].ss;
    if(tipo==BLOCO) {
        //Encontrou um bloco comum.
        if(blocos.count(v_id)) return v_id;
    }
    else {
        for(set<piii>::iterator it=bc[tipo][v_id].begin();
        it!=bc[tipo][v_id].end(); it++) {
            int attr=(*it).ff;
            //Encontrou um bloco comum.
            if(attr==v && blocos.count((*it).ss.ss)) return (*it).ss.ss;
        }
    }
    //Nao encontrou nenhum bloco comum
    return -1;
}

//Resolve caso 5.3.3
void adicionar_bloco(int u, int v) {
    //Se os dois vertices pertencem a mesma conexao,
    //a outerplanaridade sera mantida.
    pii u_barra=indice[u], v_barra=indice[v];
    //TODO: preciso checar se os dois sao conexoes mesmo?
    //se os vertices pertencerem ao mesmo bloco nao deveria cair no caso 5.3.2?
    if(u_barra.ff==CONEXAO && v_barra.ff==CONEXAO && u_barra.ss==v_barra.ss) {
        int ret=bfs(u, v);
        //Deve haver caminho entre os vertices
```

```
assert(ret==1);
//Conexao associada aos vertices
int conexao_atual=u_barra.ss;
//Essa estrutura armazenara todos os atributos de aresta no grafo BC(G),
//ou seja, todos os vertices de G que fazem fronteira entre a conexao
//que estamos analisando e um outro bloco.
set<int> atributos;
for(set<piii>::iterator it=bc[u_barra.ff][u_barra.ss].begin();
it!=bc[u_barra.ff][u_barra.ss].end(); it++) {
    atributos.insert(it->first);
}
//Formaremos um novo bloco, entao devemos pegar um bloco disponivel.
//Todos os vertices do caminho farao parte do novo bloco.
int bloco_disponivel=-bloco.top();
bloco.pop();
//Desmarcando os vertices
memset(vis, 0, sizeof(vis));
//Marcando todos os vertices do caminho Puv
//A marcacao e feita com um identificador especial pois se houver algum
//atributo de aresta no grafo BC(G) com um dos vertices do caminho Puv,
//entao esse vertice e de articulacao e devemos criar uma nova conexao
//somente com esse vertice
for(int vert=v; vert!=-1; vert=pai[vert]) {
    vis[vert]=2;
    //Atualizando vertices do caminho para fazerem parte do novo bloco.
    //Essa atualizacao sera modificada em alguns dos vertices que forem
    //fazer parte de novas conexoes
    //Removendo vert do indice_rev
    indice_rev[indice[vert]].erase(vert);
    indice[vert].ff=BLOCO;
    indice[vert].ss=bloco_disponivel;
    //Adicionando vert no indice_rev
    indice_rev[indice[vert]].insert(vert);
}

//Guarda vertice anterior para atualizacao das arestas hamiltonianas
int ant_vert=-1;
//Devemos explorar todos os vertices do caminho
//para gerar as novas conexoes.
for(int vert=v; vert!=-1; vert=pai[vert]) {
    //Atualizacao da nova face
    adicionar_face(id_faces, vert);
```

```
//Atualizacao das arestas hamiltonianas
if(ant_vert!=-1) {
    //Toda aresta do caminho na arvore passa a ser hamiltoniana
    int temp_u=vert, temp_v=ant_vert;
    if(temp_u>temp_v) swap(temp_u, temp_v);
    hamiltoniano.insert(pii(temp_u, temp_v));
}
ant_vert=vert;

//Esta estrutura auxiliar ira armazenar todos os vertices
//que fazem parte da conexao atual e que nao fazem do caminho Puv,
//com excecao do vertice vert, raiz da arvore.
set<int> temp;
//Rodando um BFS auxiliar para encontrar os vertices
//na antiga conexao que farao parte da nova conexao
queue<int> q;
temp.insert(vert);
q.push(vert);
while(!q.empty()) {
    int u=q.front();
    q.pop();
    for(set<int>::iterator it=adj[u].begin();
    it!=adj[u].end(); it++) {
        int v=(*it);
        if(indice[v].ff==CONEXAO && indice[v].ss==conexao_atual
        && !vis[v]) {
            vis[v]=1;
            temp.insert(v);
            q.push(v);
        }
    }
}
//Se a exploracao a partir do vertice nao encontrou
//nenhum vertice alem dele e esse vertice nao e
//atributo de aresta em BC(G), podemos prosseguir.
if((int)temp.size()==1 && !atributos.count(vert)) continue;
//Caso contrario teremos que criar uma nova conexao
//que se ligara com o novo bloco e com os outros blocos
//que possuirem como atributos de aresta algum vertice que
//esteja em temp.
int conexao_disponivel=-conexao.top();
conexao.pop();
```

```
        //Atribuindo os vertices a nova conexao
        for(set<int>::iterator it=temp.begin(); it!=temp.end(); it++) {
            //Removendo it do indice_rev
            indice_rev[indice[*it]].erase(*it);
            indice[*it].ff=CONEXAO;
            indice[*it].ss=conexao_disponivel;
            //Adicionando it no indice_rev
            indice_rev[indice[*it]].insert(*it);
        }
        //Criando as adjacencias entre o novo bloco e a nova conexao
        bc[BLOCO][bloco_disponivel].insert(
        piii(vert, pii(CONEXAO, conexao_disponivel)));
        bc[CONEXAO][conexao_disponivel].insert(
        piii(vert, pii(BLOCO, bloco_disponivel)));

        //Criando as adjacencias entre a nova conexao e os blocos
        //que eram adjacentes a conexao antiga
        for(set<piii>::iterator it=bc[u_barra.ff][u_barra.ss].begin();
        it!=bc[u_barra.ff][u_barra.ss].end(); it++) {
            int attr=(*it).ff;
            int vert_tipo=(*it).ss.ff;
            int vert_id=(*it).ss.ss;
            //Garantindo que seja bloco
            assert(vert_tipo==BLOCO);
            //Se um dos vertices e atributo de aresta em BC(G),
            //devemos ligar a nova conexao com o bloco
            //que se ligava a conexao antiga.
            if(temp.count(attr)) {
                bc[CONEXAO][conexao_disponivel].insert(*it);
                bc[vert_tipo][vert_id].insert(
                piii(attr, pii(CONEXAO, conexao_disponivel)));
            }
        }
    }
    hamiltoniano.insert(pii(u, v));
    id_faces++;

    //Removendo as adjacencias a conexao_atual nos outros blocos.
    for(set<piii>::iterator it=bc[u_barra.ff][u_barra.ss].begin();
    it!=bc[u_barra.ff][u_barra.ss].end(); it++) {
        int vert_tipo=(*it).second.first;
        int vert_id=(*it).second.second;
```

```
        bc[vert_tipo][vert_id].erase(
        piii((*it).first, pii(u_barra.ff, u_barra.ss)));
    }
    //Removendo a conexao_atual e suas adjacencias
    //e atribuindo conexao_atual como conexao_disponivel
    indice_rev[u_barra].clear();
    bc[u_barra.ff][u_barra.ss].clear();
    conexao.push(-conexao_atual);

    //Adicionando as arestas no grafo G.
    adj[u].insert(v);
    adj[v].insert(u);
    printf("Caso 5.3.3.1. Aresta (%d, %d) inserida com sucesso.\n", u, v);
    return;
}

//Checando se a outerplanaridade sera mantida.
int ret=bfs(u, v);
//So faz sentido realizar a operacao se existir um caminho entre u e v.
//Isso faz total sentido, pois os vertices obrigatoriamente
//pertencem ao mesmo bloco.
assert(ret==1);
//Deve-se verificar as condicoes do teorema 4.2 para que a adicao da aresta
//seja possivel.
//1 - Nenhuma aresta no caminho minimo pode ser
//fronteira entre faces internas em G;
//2 - Duas arestas consecutivas no caminho minimo nao podem
//pertencer ao mesmo bloco de G.
//Para verificar a condicao 1: a aresta nao pode ser corda.
//Para verificar a condicao 2: duas arestas consecutivas
//nao podem ser hamiltonianas. Alem disso, encontra-se o bloco comum entre
//vert e penult_vert e entre penult_vert e ult_vert.
//Ultimo vertice e penultimo vertice
int ult_vert=-1, penult_vert=-1;
for(int vert=v; vert!=-1; vert=pai[vert]) {
    //Ordem no caminho: vert--penult_vert--ult_vert
    if(penult_vert!=-1) {
        //Verificando condicao 1
        if(corda.count(pii(min(vert, penult_vert), max(vert, penult_vert)))) {
            printf("Caso 5.3. Nao e possivel adicionar a aresta (%d, %d). ");
            printf("Aresta fronteira entre faces internas\n", u, v);
            return;
```

```
        }
        if(ult_vert!=-1) {
            //Verificando condicao 2
            if(hamiltoniano.count(
                pii(min(vert, penult_vert), max(vert, penult_vert))) &&
               hamiltoniano.count(
                pii(min(penult_vert, ult_vert), max(penult_vert, ult_vert)))) {
                int bloco1=encontrar_bloco(vert, penult_vert);
                int bloco2=encontrar_bloco(penult_vert, ult_vert);
                if(bloco1!=-1 && bloco1==bloco2) {
                    printf("Caso 5.3. Nao e possivel adicionar a aresta ");
                    printf("(%d, %d). Arestas consecutivas ");
                    printf("pertencem ao mesmo bloco\n", u, v);
                    return;
                }
            }
        }
    }
    ult_vert=penult_vert;
    penult_vert=vert;
}
//Passou pelas verificacoes de consistencia da outerplanaridade.
//Agora e possivel adicionar a aresta no grafo G mantendo a outerplanaridade.
//Pode-se proceder com a atualizacao das estruturas.
//Guarda vertice anterior para atualizacao das arestas hamiltonianas
int ant_vert=-1;
for(int vert=v; vert!=-1; vert=pai[vert]) {
    //Atualizacao da nova face
    adicionar_face(id_faces, vert);
    //Atualizacao das arestas hamiltonianas
    if(ant_vert!=-1) {
        //Se aresta era hamiltoniana previamente, deixa de ser
        //e passa a ser corda. Se aresta nao era hamiltoniana, passa a ser
        int temp_u=vert, temp_v=ant_vert;
        if(temp_u>temp_v) swap(temp_u, temp_v);
        if(hamiltoniano.count(pii(temp_u, temp_v))) {
            hamiltoniano.erase(pii(temp_u, temp_v));
            corda.insert(pii(temp_u, temp_v));
        }
        else hamiltoniano.insert(pii(temp_u, temp_v));
    }
    ant_vert=vert;
```

```
}
//Nova aresta faz parte do ciclo hamiltoniano.
hamiltoniano.insert(pii(u, v));
id_faces++;

//Atualizacao do grafo BC(G)
//Deve-se percorrer o caminho no grafo BC(G) e atualizar os blocos/conexoes
//correspondentes. Os comportamentos sao diferentes para cada tipo de vertice
//(bloco ou conexao). O comportamento para conexoes e semelhante ao tratado
//no caso em que e formado um novo bloco e os dois vertices pertencem a
//mesma conexao.

//Pegando bloco_disponivel que formara o novo bloco do grafo BC(G)
int bloco_disponivel=-bloco.top();
bloco.pop();

//Ja calculamos previamente o caminho no grafo BC(G) e tambem no grafo G.
//Percorrendo cada vertice do caminho
pii x_barra=v_barra;
//Devemos manter os vertices adjacentes a x_barra no caminho PIuv em BC(G)
//prox_barra--x_barra--ant_barra
pii ant_barra=pii(-1, -1), prox_barra=pai_bc[x_barra.ff][x_barra.ss].ss;
//Inicialmente nao ha nenhum atributo de aresta
//Deve-se manter tambem o atributo do proximo no no caminho PIuv em BC(G)
int attr=-1, prox_attr=pai_bc[x_barra.ff][x_barra.ss].ff;
while(1) {
    if(x_barra.ff==-1 && x_barra.ss==-1) break;
    if(x_barra.ff==BLOCO) {
        //Caso em que o vertice de BC(G) do caminho e um bloco. Nesse caso,
        //o bloco deixara de existir e fara parte do novo bloco_disponivel.
        //Todos os vertices de G que faziam parte do bloco, ou seja,
        //indice[vertice]=(BLOCO, x_barra.ss) devem passar para o novo
        //bloco_disponivel. Alem disso, todas as adjacencias do bloco,
        //com excecao dos vertices no caminho PIuv em BC(G), passarao a fazer
        //parte do novo bloco_disponivel.
        //Percorrendo todos os vertices ligados ao bloco x_barra
        set<int> vert_x_barra=indice_rev[x_barra];
        for(set<int>::iterator it=vert_x_barra.begin();
        it!=vert_x_barra.end(); it++) {
            indice[*it].ff=BLOCO;
            indice[*it].ss=bloco_disponivel;
            //Adicionando vertices no indice_rev
```

```
            indice_rev[indice[*it]].insert(*it);
        }
        //Apagando ocorrencias dos vertices no vertice x_barra
        indice_rev[x_barra].clear();

        //Passando adjacencias para o novo bloco
        for(set<piii>::iterator it=bc[x_barra.ff][x_barra.ss].begin();
        it!=bc[x_barra.ff][x_barra.ss].end(); it++) {
            pii vert_barra=(*it).ss;
            int attr=(*it).ff;
            //Se o vertice adjacente for um dos do caminho
            //PIuv de BC(G) nao atualizamos. Eles tambem serao/foram apagados
            if(vert_barra==ant_barra || vert_barra==prox_barra) continue;
            //Removendo adjacencia do vertice vert_barra
            bc[vert_barra.ff][vert_barra.ss].erase(*it);
            //Adicionando adjacencias nos vertices
            //vert_barra e no novo bloco_disponivel
            bc[vert_barra.ff][vert_barra.ss].insert(
            piii(attr, pii(BLOCO, bloco_disponivel)));
            bc[BLOCO][bloco_disponivel].insert(
            piii(attr, pii(vert_barra.ff, vert_barra.ss)));
        }
        //Apagando adjacencias do vertice x_barra
        bc[x_barra.ff][x_barra.ss].clear();

        //Bloco passa a ficar disponivel
        bloco.push(-x_barra.ss);
    }
    else {
        //Caso em que o vertice de BC(G) e uma conexao. Esse caso e
        //semelhante ao caso em que forma-se um novo bloco com dois
        //vertices ligados a mesma conexao. Os vertices do caminho
        //minimo puv em G farao parte do novo bloco, porem deve-se
        //analisar os vertices que nao pertencem ao caminho minimo.
        //Esses vertices, juntamente com alguns vertices do caminho
        //puv formarao novas conexoes, que deverao se ligar com o novo
        //bloco_disponivel e com os blocos adequados que se ligavam
        //com a conexao antiga.

        //Conexao associada aos vertices
        int conexao_atual=x_barra.ss;
        //Essa estrutura armazenara todos os atributos de aresta
```

```
//no grafo BC(G), ou seja, todos os vertices de G que fazem fronteira
//entre a conexao que estamos analisando e um outro bloco.
set<int> atributos;
for(set<piii>::iterator it=bc[x_barra.ff][x_barra.ss].begin();
it!=bc[x_barra.ff][x_barra.ss].end(); it++) {
    atributos.insert(it->first);
}

//Desmarcando os vertices
memset(vis, 0, sizeof(vis));
//Marcando todos os vertices do caminho Puv
//A marcacao e feita com um identificador especial pois se houver algum
//atributo de aresta no grafo BC(G) com um dos vertices do caminho Puv,
//entao esse vertice e de articulacao e devemos criar uma nova conexao
//somente com esse vertice

//Devemos  percorrer os vertices do caminho Puv na conexao que estamos
//analisando, ou seja, o vertice do fim da conexao e o atributo da
//aresta se ja passamos por algum outro vertice de BC(G), ou o proprio
//vertice v, se a conexao e o primeiro vertice pelo qual passamos em
//BC(G). Do mesmo modo, o vertice do inicio da sera algum vertice
//intermediario da conexao, ou entao o vertice fronteira, simbolizando
//que iremos prosseguir no caminho alem dessa conexao. Se o vertice
//fronteira for o objetivo, tem-se que atentar ao fato de que ele sera
//um vertice de articulacao e deve-se criar uma nova conexao com esse
//vertice

//Vertice no qual sera realizada a condicao de parada. Se devemos
//percorrer a conexao de ponta a ponta, entao o vertice de parada
//e o pai do vertice prox_attr. Caso contrario, o vertice de parada
//e o pai do vertice u.
int vertice_parada=(prox_attr==-1 ? pai[u] : pai[prox_attr]);
for(int vert=(attr==-1 ? v : attr);
vert!=vertice_parada; vert=pai[vert]) {
    vis[vert]=2;
    //Atualizando vertices do caminho para fazerem parte do novo bloco.
    //Essa atualizacao sera modificada em alguns dos vertices que forem
    //fazer parte de novas conexoes
    //Removendo vert do indice_rev
    indice_rev[indice[vert]].erase(vert);
    indice[vert].ff=BLOCO;
    indice[vert].ss=bloco_disponivel;
```

```
        //Adicionando vert no indice_rev
        indice_rev[indice[vert]].insert(vert);
    }

    //Devemos explorar todos os vertices do caminho
    //para gerar as novas conexoes.
    for(int vert=(attr==-1 ? v : attr);
    vert!=vertice_parada; vert=pai[vert]) {
        //Esta estrutura auxiliar ira armazenar todos os vertices
        //que fazem parte da conexao atual e que nao fazem do caminho Puv,
        //com excecao do vertice vert, raiz da arvore.
        set<int> temp;
        //Rodando um BFS auxiliar para encontrar os vertice
        //na antiga conexao que farao parte da nova conexao
        queue<int> q;
        temp.insert(vert);
        q.push(vert);
        while(!q.empty()) {
            int u=q.front();
            q.pop();
            for(set<int>::iterator it=adj[u].begin();
            it!=adj[u].end(); it++) {
                int v=(*it);
                if(indice[v].ff==CONEXAO && indice[v].ss==conexao_atual &&
                    !vis[v]) {
                     vis[v]=1;
                     temp.insert(v);
                     q.push(v);
                }
            }
        }
        //Se a exploracao a partir do vertice nao encontrou
        //nenhum vertice alem dele
        if((int)temp.size()==1) {
            //Se esse vertice nao e atributo de aresta em BC(G),
            //podemos prosseguir.
            if(!atributos.count(vert)) continue;
            else {
                //Tem-se aqui que tomar cuidado com as pontas da conexao,
                //coisa que nao era necessaria no caso 5.3.3.1. Nesse caso,
                //so teremos que nos preocupar com as extremidades se elas
                //forem u ou v. Se uma das extremidades e u ou v,
```

```
            //acabaremos gerando um ponto de articulacao e temos que
            //atualizar o grafo BC(G) de forma adequada. Vertices das
            //extremidades da conexao em Puv de G
            int ext1=(attr==-1) ? v : attr;
            int ext2=prox_attr;
            if(vert==ext1 || vert==ext2) {
                //Nesse caso nao precisamos mexer nas extremidades
                if(vert!=u && vert!=v) continue;
            }
        }
    }
    //Caso contrario teremos que criar uma nova conexao que se ligara
    //com o novo bloco e com os outros blocos que possuirem como
    //atributos de aresta algum vertice que esteja em temp.
    int conexao_disponivel=-conexao.top();
    conexao.pop();
    //Atribuindo os vertices a nova conexao
    for(set<int>::iterator it=temp.begin(); it!=temp.end(); it++) {
        //Removendo it do indice_rev
        indice_rev[indice[*it]].erase(*it);
        indice[*it].ff=CONEXAO;
        indice[*it].ss=conexao_disponivel;
        //Adicionando it no indice_rev
        indice_rev[indice[*it]].insert(*it);
    }
    //Criando as adjacencias entre o novo bloco e a nova conexao
    bc[BLOCO][bloco_disponivel].insert(
    piii(vert, pii(CONEXAO, conexao_disponivel)));
    bc[CONEXAO][conexao_disponivel].insert(
    piii(vert, pii(BLOCO, bloco_disponivel)));

    //Criando as adjacencias entre a nova conexao e os blocos
    //que eram adjacentes a conexao antiga
    for(set<piii>::iterator it=bc[x_barra.ff][x_barra.ss].begin();
    it!=bc[x_barra.ff][x_barra.ss].end(); it++) {
        int attr1=(*it).ff;
        pii vert_barra=(*it).ss;
        int vert_tipo=vert_barra.ff;
        int vert_id=vert_barra.ss;
        //Se o vertice adjacente for um dos do caminho PIuv
        //de BC(G) nao atualizamos. Eles tambem serao/foram apagados
        if(vert_barra==ant_barra || vert_barra==prox_barra) continue;
```

```
                //Garantindo que seja bloco
                assert(vert_tipo==BLOCO);
                //Se um dos vertices e atributo de aresta em BC(G),
                //devemos ligar a nova conexao com o bloco que se
                //ligava a conexao antiga.
                if(temp.count(attr1)) {
                    bc[CONEXAO][conexao_disponivel].insert(*it);
                    bc[vert_tipo][vert_id].insert(
                    piii(attr1, pii(CONEXAO, conexao_disponivel)));
                }
            }
        }

        //Removendo as adjacencias a conexao_atual nos outros blocos.
        for(set<piii>::iterator it=bc[x_barra.ff][x_barra.ss].begin();
        it!=bc[x_barra.ff][x_barra.ss].end(); it++) {
            int vert_tipo=(*it).second.first;
            int vert_id=(*it).second.second;
            bc[vert_tipo][vert_id].erase(
            piii((*it).first, pii(x_barra.ff, x_barra.ss)));
        }
        //Apagando ocorrencias dos vertices no vertice x_barra
        indice_rev[x_barra].clear();

        //Removendo a conexao_atual e suas adjacencias e atribuindo
        //conexao_atual como conexao_disponivel
        bc[x_barra.ff][x_barra.ss].clear();
        conexao.push(-conexao_atual);
    }
    //Andando no caminho em BC(G)
    piii vert_pai_bc=pai_bc[x_barra.ff][x_barra.ss];
    ant_barra=x_barra;
    x_barra=vert_pai_bc.ss;
    if(x_barra.ff!=-1 && x_barra.ss!=-1) {
        prox_barra=pai_bc[x_barra.ff][x_barra.ss].ss;
        prox_attr=pai_bc[x_barra.ff][x_barra.ss].ff;
    }
    attr=vert_pai_bc.ff;
}

//Adicionando as arestas no grafo G.
adj[u].insert(v);
```

```
    adj[v].insert(u);
    printf("Caso 5.3.3.2. Aresta (%d, %d) inserida com sucesso.\n", u, v);
}

//Resolve caso 5.3.2
void adicionar_corda(int u, int v) {
    //Primeiramente devemos verificar se e possivel adicionar a aresta (u, v).
    //Devemos checar se existe uma face comum entre os vertices.
    if(u>v) swap(u, v);
    int face_comum=encontrar_face_comum(u, v, 1);
    if(!face_comum) {
        //Nao ha face comum entre os vertices do bloco.
        //Nao e possivel adicionar a aresta (u, v), pois esta invalida a
        //outerplanaridade.
        printf("Caso 5.3.2. Nao e possivel adicionar a aresta (%d, %d).\n", u, v);
        return;
    }
    //Deve-se atualizar as faces do grafo G.
    //A adicao da aresta nao altera o grafo BC(G).
    int ret=bfs(u, v);
    //So faz sentido realizar a operacao se existir um caminho entre u e v.
    //Isso faz total sentido, pois os vertices obrigatoriamente
    //pertencem ao mesmo bloco.
    assert(ret==1);
    //Atualizacao das estruturas
    for(int vert=v; vert!=-1; vert=pai[vert]) {
        if(vert!=u && vert!=v) {
            //Para vertices diferentes da extremidade devemos remover a face comum,
            //pois esses vertice nao possuirao mais essa face, somente a nova.
            remover_face(face_comum, vert);
        }
        adicionar_face(id_faces, vert);
    }
    id_faces++;
    corda.insert(pii(u, v));

    //Adicionando as arestas no grafo G.
    adj[u].insert(v);
    adj[v].insert(u);
    printf("Caso 5.3.2. Aresta (%d, %d) inserida com sucesso.\n", u, v);
}
```

```
//Dado um conjunto de vertices, une-se os mesmos, alem dos respectivos
//vertices que estejam na mesma conexao que eles em uma unica conexao.
//Alem disso, atualiza toda a estrutura de BC(G). Desfaz qualquer ligacao
//com o bloco_comum, que sera futuramente apagado.
//Por fim, cria uma nova adjacencia com o novo_bloco, que foi criado
//ha pouco tempo, com atributo de aresta sendo 'atributo'. Retorna o valor
//da conexao que foi criada.
int unir_conexao(vector<int> vertices, int bloco_comum, int novo_bloco,
                 int atributo) {
    //Estrutura que armazena as conexoes a serem apagadas
    vector<int> conexoes_apagar;
    //Pegando uma conexao que esteja disponivel para ser a nova conexao.
    int conexao_disponivel=-conexao.top();
    conexao.pop();
    for(int i=0; i<vertices.size(); i++) {
        int vert=vertices[i];
        if(indice[vert].ff==CONEXAO) {
            pii temp=indice[vert];
            conexoes_apagar.push_back(indice[vert].ss);
            //Atribuindo todos os vertices da conexao de vert a nova conexao
            set<int> vertices_conexao_vert=indice_rev[indice[vert]];
            for(set<int>::iterator it=vertices_conexao_vert.begin();
            it!=vertices_conexao_vert.end(); it++) {
                indice[*it]=pii(CONEXAO, conexao_disponivel);
                indice_rev[pii(CONEXAO, conexao_disponivel)].insert(*it);
            }
            //Apagando todos os vertices da conexao que vert
            //participava de indice_rev
            indice_rev[temp].clear();
        }
        else {
            //Removendo vert de indice_rev
            indice_rev[indice[vert]].erase(vert);
            //Atualizando vert
            indice[vert]=pii(CONEXAO, conexao_disponivel);
            //Adicionando vert no novo indice_rev
            indice_rev[indice[vert]].insert(vert);
        }
    }

    //Atualizando BC(G)
    //Deve-se passar todas as adjacencias das conexoes a apagar para a nova
```

```
    //conexao_disponivel. Alem disso, devemos substituir a adjacencia nos
    //blocos adjacentes as conexoes a apagar.
    for(int i=0; i<conexoes_apagar.size(); i++) {
        int conexao_apagar=conexoes_apagar[i];
        for(set<piii>::iterator it=bc[CONEXAO][conexao_apagar].begin();
        it!=bc[CONEXAO][conexao_apagar].end(); it++) {
            int attr=(*it).ff;
            int vert_tipo=(*it).ss.ff;
            int vert_id=(*it).ss.ss;
            //Nao devemos persistir a adjacencia com o bloco_comum,
            //que sera apagado.
            if(vert_id==bloco_comum) continue;
            //Adicionando adjacencia na conexao_disponivel
            bc[CONEXAO][conexao_disponivel].insert(
            piii(attr, pii(vert_tipo, vert_id)));
            //Adicionando nova adjacencia no bloco adjacente
            bc[vert_tipo][vert_id].insert(
            piii(attr, pii(CONEXAO, conexao_disponivel)));
            //Removendo adjacencia do bloco adjacente
            bc[vert_tipo][vert_id].erase(
            piii(attr, pii(CONEXAO, conexao_apagar)));
        }
        bc[CONEXAO][conexao_apagar].clear();
        //conexao_apagar agora esta disponivel para ser usada.
        conexao.push(-conexao_apagar);
    }

    //Adicionando adjacencia ao novo_bloco
    if(novo_bloco!=-1) {
        bc[CONEXAO][conexao_disponivel].insert(
        piii(atributo, pii(BLOCO, novo_bloco)));
        bc[BLOCO][novo_bloco].insert(
        piii(atributo, pii(CONEXAO, conexao_disponivel)));
    }
    return conexao_disponivel;
}

//Cria, se necessario, uma conexao a partir do ponto de articulacao
//que aparece ao se remover uma aresta do ciclo hamiltoniano. Se o
//vertice vert ja era ligado a uma conexao, nao ha necessidade de
//fazer essa criacao. Desfaz qualquer ligacao com o bloco_comum,
//que sera futuramente apagado.Finalmente, atualiza-se BC(G) ligando
```

```
//essa conexao ao bloco recem criado, novo_bloco, com atributo de aresta vert.
//Retorna o vertice conexao ao qual vert esta ligado.
int criar_conexao_articulacao(int vert, int bloco_comum, int novo_bloco) {
    if(indice[vert].ff==BLOCO) {
        //Deve-se criar uma conexao somente para esse vertice
        int conexao_disponivel=-conexao.top();
        conexao.pop();

        //Removendo vert de indice_rev
        indice_rev[indice[vert]].erase(vert);
        indice[vert]=pii(CONEXAO, conexao_disponivel);
        //Adicionando vert ao novo indice_rev
        indice_rev[indice[vert]].insert(vert);
    }
    else {
        //Ja existia previamente uma conexao associada a vert. Nesse caso,
        //ha uma aresta em BC(G) que liga essa conexao ao bloco_comum, que
        //sera apagado futuramente. Deve-se entao remover essa adjacencia.
        bc[indice[vert].ff][indice[vert].ss].erase(
        piii(vert, pii(BLOCO, bloco_comum)));
    }

    //Atualizando BC(G)
    int vert_tipo=indice[vert].ff;
    int vert_id=indice[vert].ss;
    bc[vert_tipo][vert_id].insert(piii(vert, pii(BLOCO, novo_bloco)));
    bc[BLOCO][novo_bloco].insert(piii(vert, pii(vert_tipo, vert_id)));
    return vert_id;
}

//BFS auxiliar que percorre o grafo das arestas hamiltonianas, procurando
//um caminho entre s e d
void bfs_h(int s, int d) {
    memset(vis, 0, sizeof(vis));
    queue<int> q;
    pai_h[s]=-1;
    vis[s]=1;
    q.push(s);
    while(!q.empty()) {
        int u=q.front();
        q.pop();
        for(set<int>::iterator it=adj[u].begin(); it!=adj[u].end(); it++) {
```

```
            int v=(*it);
            if(!vis[v] && hamiltoniano.count(pii(min(u, v), max(u, v)))) {
                vis[v]=1;
                pai_h[v]=u;
                if(v==d) return;
                q.push(v);
            }
        }
    }
}

//Dada uma corda (u, v), cria um bloco. O novo bloco sera formado pela
//corda (u, v) e todas as arestas hamiltonianas no caminho de u ate v
//(no grafo de arestas hamiltonianas). Alem disso deve-se fazer as devidas
//atualizacoes em BC(G). Desfaz qualquer ligacao com o bloco_comum,
//que sera futuramente apagado.Finalmente deve-se ligar o novo
//bloco com a nova_conexao (se existir) criada anteriormente, com atributo
//de aresta 'atributo'. Retorna o bloco criado. Vertices de articulacao nao farao
//parte do novo bloco que sera criado.
int criar_bloco(int u, int v, int bloco_comum, int nova_conexao, int atributo,
                set<int> articulacao) {
    //Pegando um bloco disponivel
    int bloco_disponivel=-bloco.top();
    bloco.pop();

    if(u>v) swap(u, v);
    //BFS auxiliar percorrendo o grafo das arestas hamiltonianas
    bfs_h(u, v);

    set<int> vertices;
    //Adicionando todos os vertices que nao sao de articulacao do caminho
    //encontrado em bfs_h na estrutura 'vertices'. Esses vertices farao
    //parte do bloco_disponivel, e estarao ligados a ele se nao estiverem
    //ligados a nenhuma conexao.
    for(int vert=v; vert!=-1; vert=pai_h[vert]) {
        if(!articulacao.count(vert)) vertices.insert(vert);
    }

    for(set<int>::iterator it=vertices.begin(); it!=vertices.end(); it++) {
        int vert=(*it);
        //Atualizando vertices para o bloco_disponivel
        if(indice[vert].ff==BLOCO) {
```

```
            //Removendo vert de indice_rev
            indice_rev[indice[vert]].erase(vert);
            //Atualizando vert
            indice[vert]=pii(BLOCO, bloco_disponivel);
            //Adicionando vert no novo indice_rev
            indice_rev[indice[vert]].insert(vert);
        }
    }

    //Atualizando BC(G)
    //Deve-se passar as adjacencias do bloco_comum para o novo bloco_disponivel
    //(aquelas que compartilham atributo de aresta), ou seja, aquelas cujo
    //atributo de aresta esteja contido na estrutura vertices.
    //Alem disso, devemos substituir a adjacencia nas conexoes adjacentes pelo
    //bloco_disponivel e apagar as adjacencias ao bloco_comum nas
    //conexoes adjacentes.
    for(set<piii>::iterator it=bc[BLOCO][bloco_comum].begin();
    it!=bc[BLOCO][bloco_comum].end(); it++) {
        int attr=(*it).ff;
        int vert_tipo=(*it).ss.ff;
        int vert_id=(*it).ss.ss;
        //Se o atributo de aresta esta na estrutura vertices,
        //devemos adicionar a adjacencia ao bloco_disponivel
        if(vertices.count(attr)) {
            bc[BLOCO][bloco_disponivel].insert(*it);
            bc[vert_tipo][vert_id].insert(
            piii(attr, pii(BLOCO, bloco_disponivel)));
        }
        //Removendo adjacencia da conexao
        bc[vert_tipo][vert_id].erase(piii(attr, pii(BLOCO, bloco_comum)));
    }

    //Adicionando adjacencia a nova_conexao
    if(nova_conexao!=-1) {
        bc[BLOCO][bloco_disponivel].insert(
        piii(atributo, pii(CONEXAO, nova_conexao)));
        bc[CONEXAO][nova_conexao].insert(
        piii(atributo, pii(BLOCO, bloco_disponivel)));
    }
    return bloco_disponivel;
}
```

```
//Realiza a tentativa da adicao da aresta no grafo G
void adicionar_aresta(int u, int v) {
    if(u>v) swap(u, v);
    if(!pertence_mesmo_conjunto(u, v)) {
        //Caso 5.3.1 - Nesse caso tem-se a adicao de uma ponte no grafo G.
        adicionar_ponte(u, v);
    }
    else {
        //Nesse caso os vertices ja estao na mesma componente conexa.
        //O primeiro passo e calcular a distancia no grafo BC(G).
        pii u_barra=indice[u], v_barra=indice[v];
        int dist=dfs_bc(u_barra, pii(-1, -1), v_barra, -1);
        //Analise das distancias no grafo BC(G).
        if(!dist) {
            //Se o grafo em BC(G) e bloco, deve-se adicionar uma corda
            //Caso contrario, cria-se um novo bloco.
            if(u_barra.ff==BLOCO) adicionar_corda(u, v);
            else adicionar_bloco(u, v);
        }
        else if(dist==1) {
            //Se a distancia e 1, tem-se que o caminho e u_barra--v_barra.
            //Atributo da aresta (u_barra, v_barra).
            int attr=pai_bc[v_barra.ff][v_barra.ss].ff;
            //Se um dos vertices e atributo de aresta, entao deve-se adicionar
            //uma corda no bloco. Caso contrario, cria-se um novo bloco.
            if(attr==u || attr==v) adicionar_corda(u, v);
            else adicionar_bloco(u, v);
        }
        else if(dist==2) {
            //Nesse caso ha um caminho da forma: u_barra--x_barra--v_barra.
            pii x_barra=pai_bc[v_barra.ff][v_barra.ss].ss;
            int attr1=pai_bc[v_barra.ff][v_barra.ss].ff;
            int attr2=pai_bc[x_barra.ff][x_barra.ss].ff;
            if((attr1==u && attr2==v) || (attr1==v && attr2==u)) {
                adicionar_corda(u, v);
            }
            else adicionar_bloco(u, v);
        }
        else {
            //Para as distancias maiores que 2 deve-se sempre tentar formar um
            //novo bloco.
            adicionar_bloco(u, v);
```

```
        }
    }
}

//Realiza a remocao da aresta no grafo G
void remover_aresta(int u, int v) {
    //Para a remocao temos casos semelhantes aos da adicao de arestas no que diz
    //respeito aos tipos das arestas. Pode-se ter uma remocao de ponte, corda ou
    //aresta do ciclo hamiltoniano. No caso da remocao, nao ha necessidade de se
    //preocupar com a outerplanaridade, pois a remocao de qualquer aresta do grafo
    //mantem a outerplanaridade. Portanto, devemos nos preocupar apenas com a
    //manutencao das estruturas que controlam a outerplanaridade do grafo.
    //Remocao sempre possivel
    if(u>v) swap(u, v);
    adj[u].erase(v);
    adj[v].erase(u);
    pii u_barra=indice[u], v_barra=indice[v];
    if(u_barra.ff==CONEXAO && v_barra.ff==CONEXAO) {
        //Caso 5.4.1
        //Nesse caso estamos removendo uma ponte do grafo
        //Necessariamente os vertices devem estar ligados a mesma conexao
        assert(u_barra.ss==v_barra.ss);
        //TODO: atentar para o fato de a remocao de ponte poder gerar conexoes
        //triviais (somente um vertice). Existem casos em que isso acontece e
        //nao faz mais sentido manter o vertice isolado na conexao e ele deve
        //voltar a pertencer ao bloco. Essa versao do algoritmo, para esses tipos
        //de caso nao retorna o vertice para o bloco, o que e algo conceitualmente
        //incorreto. Mas sera que isso realmente e um problema? No momento de outra
        //adicao com esse vertice, o problema nao sera corrigido?
        //Exemplo de caso em que isso acontece:
        //Grafo G com 6 vertices (1, 2, 3, 4, 5, 6).
        //Operacoes:
        //adicionar_aresta(1, 2)
        //adicionar_aresta(1, 3)
        //adicionar_aresta(2, 3)
        //adicionar_aresta(4, 5)
        //adicionar_aresta(4, 6)
        //adicionar_aresta(5, 6)
        //adicionar_aresta(3, 4)
        //remover_aresta(3, 4)

        int conexao_atual=u_barra.ss;
```

```
//Essa estrutura armazenara todos os atributos de aresta no grafo BC(G),
//ou seja, todos os vertices de G que fazem fronteira entre a conexao
//que estamos analisando e um outro bloco.
set<int> atributos;
for(set<piii>::iterator it=bc[u_barra.ff][u_barra.ss].begin();
it!=bc[u_barra.ff][u_barra.ss].end(); it++) {
    atributos.insert(it->first);
}

//Pegando conjunto_disponivel
int conjunto_disponivel=-conjuntos.top();
conjuntos.pop();

//Esta estrutura auxiliar ira armazenar todos os vertices que fazem
//parte da conexao atual e que nao pertencem a subarvore vom raiz em v,
//visto que estamos removendo a aresta (u, v)
//TODO: nao vale a pena generalizar isso aqui? Ja usamos varias vezes
memset(vis, 0, sizeof(vis));
vis[v]=1;
set<int> temp;
//Rodando um BFS auxiliar para explorar o grafo a partir de u e
//excluindo a componente de v
queue<int> q;
temp.insert(u);
vis[u]=1;
//Atribuindo u ao novo conjunto_disponivel
conjunto[u]=conjunto_disponivel;
q.push(u);
while(!q.empty()) {
    int u=q.front();
    q.pop();
    for(set<int>::iterator it=adj[u].begin(); it!=adj[u].end(); it++) {
        int v=(*it);
        if(indice[v].ff==CONEXAO && indice[v].ss==conexao_atual) {
            temp.insert(v);
        }
        if(!vis[v]) {
            vis[v]=1;
            //Atribuindo v ao novo conjunto_disponivel
            conjunto[v]=conjunto_disponivel;
            q.push(v);
```

```
        }
    }
}

int conexao_disponivel=-conexao.top();
conexao.pop();
//Atribuindo os vertices a nova conexao
for(set<int>::iterator it=temp.begin(); it!=temp.end(); it++) {
    //Removendo it do indice_rev
    indice_rev[indice[*it]].erase(*it);
    indice[*it].ff=CONEXAO;
    indice[*it].ss=conexao_disponivel;
    //Adicionando it no indice_rev
    indice_rev[indice[*it]].insert(*it);
}

//Esse vetor auxiliar guardara as adjacencias da conexao_atual a serem
//removidas, pois essas adjacencias passaram para a nova conexao_disponivel
vector<piii> remover_adj;
//Criando as adjacencias entre a nova conexao e os blocos que eram
//adjacentes a conexao antiga
for(set<piii>::iterator it=bc[u_barra.ff][u_barra.ss].begin();
it!=bc[u_barra.ff][u_barra.ss].end(); it++) {
    int attr=(*it).ff;
    int vert_tipo=(*it).ss.ff;
    int vert_id=(*it).ss.ss;
    //Garantindo que seja bloco
    assert(vert_tipo==BLOCO);
    //Se um dos vertices e atributo de aresta em BC(G), devemos ligar a
    //nova conexao com o bloco que se ligava a conexao antiga.
    if(temp.count(attr)) {
        bc[CONEXAO][conexao_disponivel].insert(*it);
        bc[vert_tipo][vert_id].insert(
        piii(attr, pii(CONEXAO, conexao_disponivel)));
        //Nao estamos removendo as adjacencias nesse momento para nao
        //atrapalhar o andamento do iterator.
        remover_adj.push_back(*it);
    }
}

//Removendo as adjacencias da conexao_atual que passaram a nova
//conexao_disponivel.
```

```
        for(int i=0; i<remover_adj.size(); i++) {
            int attr=remover_adj[i].ff;
            pii x_barra=remover_adj[i].ss;
            bc[u_barra.ff][u_barra.ss].erase(remover_adj[i]);
            bc[x_barra.ff][x_barra.ss].erase(piii(attr, u_barra));
        }

        //O restante dos vertices/adjacencias/atributos ficara nessa conexao,
        //que correspondera a subarvore com raiz em v.
        printf("Caso 5.4.1. Aresta (%d, %d) removida com sucesso.\n", u, v);
    }
    else {
        int ret=bfs(u, v);
        //Deve existir caminho entre os vertices, mesmo apos a remocao da corda
        assert(ret==1);
        if(corda.count(pii(u, v))) {
            //Caso 5.4.2
            //A aresta a ser removida e uma corda no ciclo hamiltoniano.
            //Realiza-se a retirada da informacao de corda
            corda.erase(pii(u, v));

            //Nesse ponto a aresta (u, v) ja foi removida do grafo G e da estrutura
            //que armazena as cordas do grafo. A remocao desse tipo de aresta nao
            //altera a estrutura do grafo BC(G). Devemos nos preocupar apenas em
            //atualizar a estrutura de faces associadas aos vertices. Para isso,
            //utiliza-se um algoritmo semelhante ao do caso 5.3.2.
            //Vertice intermediario do caminho Puv
            int x=pai[v];
            //u e v devem possuir duas faces em comum. Devemos encontrar as duas
            //faces em comum digamos f1 e f2. Caso x tambem possua f1 em faces(x),
            //removemos f1 de todos os vertices do caminho Puv e adicionamos f2 em
            //todos os vertices do caminho Puv, exceto u e v. Caso contrario, x
            //possuira a segunda face em comum de u e v, f2. Entao, removemos f2
            //de todos os vertices do caminho Puv e adicionamos f1 em todos os
            //vertices do caminho Puv, exceto u e v.
            int face_comum1=encontrar_face_comum(u, v, 1);
            int face_comum2=encontrar_face_comum(u, v, 2);
            //Devem existir as duas faces comum.
            assert(face_comum1!=0);
            assert(face_comum2!=0);
            if(!contem_face(x, face_comum1)) swap(face_comum1, face_comum2);
            //Removendo as faces dos vertices do caminho Puv
```

```
        for(int vert=v; vert!=-1; vert=pai[vert]) {
            remover_face(face_comum1, vert);
            if(vert!=v && vert!=u) inserir_face(face_comum2, vert);
        }
        printf("Caso 5.4.2. Aresta (%d, %d) removida com sucesso.\n", u, v);
    }
    else {
        //Caso 5.4.3
        //Aresta deve ser hamiltoniana
        assert(hamiltoniano.count(pii(u, v))>0);
        //Removendo a aresta da estrutura de arestas hamiltonianas.
        hamiltoniano.erase(pii(u, v));
        //Inicialmente calcula-se a face em comum dos vertices u e v
        //(deve existir apenas uma).
        int face_comum=encontrar_face_comum(u, v, 1);
        //Deve existir uma face comum.
        assert(face_comum!=0);
        //Removendo a faces dos vertices do caminho Puv
        for(int vert=v; vert!=-1; vert=pai[vert]) {
            remover_face(face_comum, vert);
        }

        //Descobrindo qual o bloco comum a u e v.
        //Esse bloco sera removido futuramente.
        int bloco_comum=encontrar_bloco(u, v);
        //Deve haver um unico bloco comum
        assert(bloco_comum!=-1);

        //Agora deve-se analisar cada tipo de aresta no caminho Puv
        //Essa estrutura armazenara os vertices pertencentes a um caminho
        //contiguo de arestas hamiltonianas. Para esses vertices, deve-se
        //criar uma nova conexao, que tambem deve se fundir com possiveis
        //conexoes que ja existam, ligadas aos vertices desse caminho contiguo.
        vector<int> vertices_hamiltonianos;
        //Ultimo vertice e penultimo vertice
        int ult_vert=-1, penult_vert=-1;
        //Ultimo vertice da sequencia contigua de arestas hamiltonianas
        int ult_hamiltoniano=-1;
        //Ultimo vertice bloco ou conexao que foi criado ao percorrer o caminho
        int x_barra=-1;
        //Guarda vertices do tipo articulacao
        set<int> articulacao;
```

```
//Estrutura para fazer a atualizacao futura dos tipos das arestas
//Armazena informacao (tipo_aresta, vert1, vert2)
//1-hamiltoniano, 2-corda
vector<piii> atualizar_depois;
for(int vert=v; vert!=-1; vert=pai[vert]) {
    //Ordem no caminho: vert--penult_vert--ult_vert
    if(penult_vert!=-1) {
        //Verificando tipo de aresta (so pode ser corda ou hamiltoniana)
        if(corda.count(
            pii(min(vert, penult_vert), max(vert, penult_vert)))) {
            //Se a estrutura que armazena os vertices da sequencia de
            //arestas hamiltonianas nao for vazia nesse instante,
            //devemos inserir o penult_vert na mesma e proceder com a
            //criacao da nova conexao. Apos isso, como temos uma corda,
            //a estrutura deve ser limpa para possivelmente armazenar
            //novos elementos no futuro.
            if(!vertices_hamiltonianos.empty()) {
                //Adicionando o penult_vert nos vertices_hamiltonianos
                vertices_hamiltonianos.push_back(penult_vert);
                x_barra=unir_conexao(vertices_hamiltonianos,
                                     bloco_comum, x_barra,
                                     ult_hamiltoniano);
                vertices_hamiltonianos.clear();
            }

            //Analisando o caso em que ha cordas consecutivas.
            //Nesse caso, o vertice comum as cordas consecutivas sera
            //um ponto de articulacao. Caso ele nao pertenca a nenhuma
            //conexao, deve-se criar uma conexao somente
            //para esse vertice.
            if(ult_vert!=-1) {
                if(corda.count(
                    pii(min(ult_vert, penult_vert),
                        max(ult_vert, penult_vert)))) {
                    //Nesse caso, o vertice penult_vert
                    //e um ponto de articulacao.
                    x_barra=criar_conexao_articulacao(penult_vert,
                                                      bloco_comum,
                                                      x_barra);
                    articulacao.insert(penult_vert);
                }
            }
```

```
                //Apos o tratamento dos outros casos, agora e necessario
                //tratar o caso da corda (vert, penult_vert)
                x_barra=criar_bloco(vert, penult_vert, bloco_comum,
                                    x_barra, penult_vert, articulacao);

                //ult_hamiltoniano deixa de existir. Deve existir uma
                //sequencia contigua de arestas hamiltonianas.
                ult_hamiltoniano=-1;

                //Se aresta e corda, ela vira hamiltoniana
                atualizar_depois.push_back(
                                        piii(CORDA,
                                         pii(min(vert, penult_vert),
                                             max(vert, penult_vert))));
            }
            else {
                //Aresta deve ser hamiltoniana
                assert(hamiltoniano.count(
                pii(min(vert, penult_vert), max(vert, penult_vert)))>0);
                //A aresta hamiltoniana se torna ponte.
                atualizar_depois.push_back(
                piii(HAMILTONIANO, pii(min(vert, penult_vert),
                                    max(vert, penult_vert))));
                //Adicionando o penult_vert na estrutura dos vertices que
                //farao parte da sequencia contigua de
                //vertices hamiltonianos.
                vertices_hamiltonianos.push_back(penult_vert);
                //Adicionando o vert nos vertices_hamiltonianos
                //caso este seja o ultimo vertice no caminho
                if(pai[vert]==-1) vertices_hamiltonianos.push_back(vert);
                if(ult_hamiltoniano==-1) ult_hamiltoniano=penult_vert;
            }
        }
        ult_vert=penult_vert;
        penult_vert=vert;
    }
    //Caso a estrutura que armazena os vertices da sequencia contigua
    //de arestas hamiltonianas nao esteja vazia nesse momento, devemos
    //formar a nova conexao a partir dessa estrutura.
    if(!vertices_hamiltonianos.empty()) {
        x_barra=unir_conexao(vertices_hamiltonianos, bloco_comum,
```

```
                                    x_barra, ult_hamiltoniano);
                vertices_hamiltonianos.clear();
            }

            //Removendo o bloco_comum.
            bc[BLOCO][bloco_comum].clear();
            //bloco_comum agora esta disponivel;
            bloco.push(-bloco_comum);
            //Atualizando os tipos de aresta
            for(int i=0; i<atualizar_depois.size(); i++) {
                int tipo=atualizar_depois[i].ff;
                pii aresta=atualizar_depois[i].ss;

                if(tipo==CORDA) {
                    //Se aresta e corda, ela vira hamiltoniana
                    corda.erase(aresta);
                    hamiltoniano.insert(aresta);
                }
                else {
                    //A aresta hamiltoniana se torna ponte.
                    hamiltoniano.erase(aresta);
                }
            }
            printf("Caso 5.4.3. Aresta (%d, %d) removida com sucesso.\n", u, v);
        }
    }
}

int main() {
    //Tipo da operacao a ser realizada:
    //1 - adicao de aresta;
    //2 - remocao de aresta.
    int op;
    //Aresta a ser adicionada/removida do grafo G.
    int u, v;

    scanf("%d", &n);

    inicializa();

    while(scanf("%d %d %d", &op, &u, &v)!=EOF) {
        if(u>v) swap(u, v);
```

```
        if(u<1 || u>n || v<1 || v>n) {
            printf("Insira um vertice valido\n");
            continue;
        }
        if(op==1) {
            if(u==v || adj[u].count(v)) {
                printf("Nao e possivel adicionar aresta (%d, %d). ");
                printf("O grafo G nao pode ser um multigrafo\n", u, v);
                continue;
            }
            adicionar_aresta(u, v);
        }
        else {
            if(!adj[u].count(v)) {
                printf("Aresta (%d, %d) nao esta presente no grafo G\n", u, v);
                continue;
            }
            remover_aresta(u, v);
        }
        //DEBUG
        //print();
    }
    return 0;
}
```