Algoritmos para conexidade em grafos dinâmicos

Arthur Henrique Dias Rodrigues

Dissertação apresentada ao
Instituto de Matemática e Estatística
da Universidade de São Paulo
para obtenção do título de
Mestre em Ciências

Programa: Ciência da Computação
Orientadora: Profª. Drª. Cristina Gomes Fernandes

São Paulo
Outubro de 2024

Algoritmos para conexidade em grafos dinâmicos

Arthur Henrique Dias Rodrigues

Esta é a versão original da dissertação elaborada pelo candidato Arthur Henrique Dias Rodrigues, tal como submetida à Comissão Julgadora.

Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Rodrigues, Arthur Henrique Dias
Algoritmos para conexidade em grafos dinâmicos / Arthur Henrique Dias Rodrigues; orientadora, Cristina Gomes Fernandes. – São Paulo, 2024.
68 p.: il.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão original

1. Grafos dinâmicos. 2. Conexidade. 3. Floresta maximal de peso mínimo.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Dedico esse trabalho a minha mãe e
a minha irmã, por todo amor e apoio.

Agradecimentos

Gostaria de expressar minha profunda gratidão a todas e todos que me apoiaram durante esta jornada. Primeiramente, à minha mãe, Lourdes, à minha irmã, Bruna, e ao meu namorado Eugênio por estarem sempre ao meu lado, oferecendo amor, paciência e incentivo incondicionais.

Agradeço imensamente à minha orientadora, Cris, que me recebeu de braços abertos no mestrado, me orientou com excelência e dedicação, me ensinando com meus erros e tendo muita compreensão nos meus momentos mais dificeis. Sua orientação foi essencial para o meu crescimento.

Ao meu amigo Coelho, pela inspiração nessa jornada pela Teoria dos Grafos, por todos os ensinamentos e sabedoria que vou levar pro resto da minha vida e por todos os cafés que tomamos juntos.

Aos amigos, William Gnann, Hercules Freire, Igor Texeira, Matheus Pereira, Matheus Polkorny, Rodrigo Dias e Rodrigo Ribeiro, meu sincero agradecimento pela amizade e apoio ao longo desta jornada. Obrigado por estarem ao meu lado, compreendendo minhas ansiedades e cansaço. Um agradecimento especial a Rodrigo Ribeiro, Hercules Freire e Rodrigo Dias, por compartilharem comigo os desafios de conciliar nossas ocupações duplas — o mestrado/doutorado e a atuação como pesquisadores no Instituto de Pesquisas Tecnológicas.

Aos amigos do IME, Antônio Kaique, Ariana, César, Colucci, Gabriel Morete, Heloísa, Hugo, Juliane, Pedro e Thiago Oliveira, sou muito grato pela camaradagem que tornou este percurso muito mais agradável.

A todos, meu sincero agradecimento!

Resumo

Arthur Henrique Dias Rodrigues. **Algoritmos para conexidade em grafos dinâmicos**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Essa dissertação aborda os problemas de conexidade em grafos dinâmicos e floresta maximal de peso mínimo em grafos planos ponderados dinâmicos. Para o primeiro problema é apresentado a solução proposta por Holm, de Lichtenberg e Thorup [20] junto às estruturas de dados utilizadas e a elaboração de um algoritmo que resolve esse problema restrito a floresta dinâmicas. Para o segundo problema é apresentada de forma didática a solução de Eppstein et al. [12]. Por fim é apresentado o limitante inferior de $\Omega(\lg n)$ por operação para os problemas estudados.

Palavras-chave: Grafos dinâmicos. Conexidade. Floresta maximal de peso mínimo. Euler tour trees.

Abstract

Arthur Henrique Dias Rodrigues. **Algorithms for dynamic graph connectivity**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

This dissertation addresses the problems of connectivity in dynamic graphs and the minimum weight spanning forest in dynamic weighted planar graphs. For the first problem, the solution proposed by Holm, de Lichtenberg, and Thorup [20] is presented, along with the data structures used, as well as the development of an algorithm that solves this problem restricted to dynamic forests. For the second problem, Eppstein et al. [12] solution is presented in a didactic manner. Finally, the lower bound of $\Omega(\lg n)$ per operation for the studied problems is presented.

Keywords: Dynamic graphs. Connectivity. Minimum spanning forest. Euler tour trees.

Sumário

Capítulo 1

Introdução

1.1 Motivação

Problemas em grafos servem para modelar uma série de aplicações do dia a dia, e há uma vasta e clássica literatura que aborda vários problemas centrais em grafos. Estes problemas clássicos geralmente consideram um modelo estático da situação. Ou seja, o grafo dado a priori não sofre alterações enquanto estamos resolvendo o problema.

Há no entanto aplicações em que o grafo modela uma situação menos estática. Por exemplo, em redes de dispositivos de internet das coisas, chuvas, ventos fortes ou falha na fonte de energia podem prejudicar a conexão entre dispositivos, o que pode ser representado pela remoção de uma aresta no grafo que abstrai a rede.

Algoritmos em grafos dinâmicos é o termo usado para se referir à área de projeto de algoritmos que se concentra em resolver problemas clássicos de forma eficiente nesse contexto em que o grafo está sofrendo alterações. A literatura nessa área tem mais de 40 anos e muito progresso significativo tem ocorrido recentemente. No entanto, há uma carência de material escrito – especialmente em português – nos livros de algoritmos sobre essa área tão atraente e atual.

Formalmente, um **grafo dinâmico** de ordem n é uma sequência de grafos $(G_0, G_1, \ldots, G_T)$, onde G_0 é um grafo com n vértices e cada G_t para $1 \leqslant t \leqslant T$ é obtido a partir de G_{t-1} pela adição ou remoção de uma aresta. Ou seja, $E(G_t) := E(G_{t-1}) \cup \{uv\}$, para alguma aresta $uv \notin E(G_{t-1})$; ou $E(G_t) := E(G_{t-1}) \setminus \{uv\}$, onde $uv \in E(G_{t-1})$, respectivamente. As operações de adição e remoção de arestas são chamadas de **modificações** ou **atualizações** do grafo dinâmico.

Há aplicações em que as conexões possuem um custo ou peso associado, representando a latência de comunicação entre dispositivos ou o custo de construção de uma infraestrutura cabeada, por exemplo. Para lidar com tais aplicações, associa-se um valor real a cada aresta do grafo. Nesse caso, o grafo resultante é chamado de **ponderado**. Em um problema com grafos dinâmicos ponderados também é considerada a operação de mudança de peso de uma determinada aresta como uma operação de atualização válida.

Um problema em grafos dinâmicos envolve verificar se o grafo corrente G_t satisfaz

alguma determinada propriedade. A operação de verificação dessa propriedade é chamada de **consulta**. Solucionar um tal problema envolve desenvolver um algoritmo ou estrutura de dados capaz de dar suporte às modificações e consultas de forma eficiente.

Em alguns casos, restringimos cada G_t a uma família de grafos, como, por exemplo, florestas ou **grafos planos**, isto é, um grafo que é planar, com uma imersão específica no plano.

Trabalhar com essas classes mais restritas de grafos permite o desenvolvimento de algoritmos mais simples que podem servir de etapa intermediária para se obter uma solução para o problema geral ou ser de interesse para alguma aplicação específica. Usaremos a primeira estratégia no estudo do nosso primeiro problema: O **problema de conexidade em grafos dinâmicos**, que consiste em, dado um grafo dinâmico submetido a uma sequência de inserções e remoções de arestas, responder a consultas do tipo "Os vértices u e v estão conectados por um caminho?".

Vamos primeiro tratar o caso em que o grafo dinâmico é uma floresta, isto é, trabalharemos com o **problema de conexidade em florestas dinâmicas**, para em seguida usar as estruturas de dados desenvolvidas nesse problema para solucionar o caso geral.

O segundo problema estudado também será restrito a uma classe específica de grafos. Estudaremos o **problema da floresta maximal de peso mínimo em grafos ponderados dinâmicos** restrito a grafos planos. Esse problema visa manter uma **floresta maximal de peso mínimo** (MSF) de um grafo dinâmico plano ponderado ao longo de uma sequência de inserções e remoções de arestas e de modificações nos pesos das arestas.

Começaremos esse estudo na próxima seção, em que faremos uma revisão histórica dos resultados relacionados aos problemas que serão abordados nos próximos capítulos. No Capítulo 2, estudaremos o problema de conexidade em florestas dinâmicas e uma de suas soluções, proposta por Holm, de Lichtenberg e Thorup [20], que envolve Euler tour trees. Essa estrutura de dados utiliza árvores binárias de busca de chave implícita, cuja implementação será elaborada no Capítulo 3. No Capítulo 4, expandiremos nosso estudo sobre conexidade estudando o problema de conexidade em grafos dinâmicos. A solução estuda também foi proposta por Holm, de Lichtenberg e Thorup [20]. Em nossos estudos, implementamos essa solução em Python3 e disponibilizamos o código no repósitorio git [32].

No Capítulo 5, apresentaremos um algoritmo proposto por Eppstein et al. [12] para solucionar o problema da floresta maximal de peso mínimo em grafos dinâmicos planos ponderados.

No Capítulo 6, mostraremos a demonstração de um limitante inferior de $\Omega(\lg n)$ por operação para todos os problemas estudados. Esse limitante, obtido por Patrascu e Demaine [31], mostra que os algoritmos apresentados para o problema de conexidade em florestas dinâmicas e para o problema da floresta maximal de peso mínimo em grafos dinâmicos planos ponderados são assintoticamente ótimos. No Capítulo 7 faremos considerações finais sobre nossos estudos.

1.2 Historiografia

Essa seção é inspirada na historiografia apresentada em [16, 42].

Soluções para os problemas de conexidade dinâmica e de floresta maximal de peso mínimo se desenvolveram em paralelo ao longo dos anos. Em 1975, Spira e Pan [35] atacaram o problema MSF propondo um algoritmo cuja complexidade é $O(n)$ para inserção e $O(n^2)$ para remoção de arestas, onde n é o número de vértices do grafo. Três anos depois, Chin e Houck [7] apresentaram uma solução mais simples para inserção e remoção de arestas que possui o mesmo consumo de tempo que o algoritmo de Spira e Pan.

Em 1985, Frederickson [15] reduziu a complexidade de ambas as operações de modificação para $O(\sqrt{m})$, onde m é o número de arestas do grafo no momento em que a operação é aplicada. Em 1992, Eppstein et al. [10, 11] melhoraram o consumo de tempo do algoritmo de Frederickson para $O(\sqrt{n})$.

O primeiro algoritmo poli-logarítmico para o problema de conectividade dinâmica foi apresentado por Henzinger e King [18] em 1995 e possui consumo amortizado esperado $O(\lg^3 n)$ para cada operação. Nesse artigo, elas propuseram a utilização de uma estrutura de dados chamada Euler tour trees como uma maneira eficiente de representar florestas dinâmicas. Em seguida, em 1997, o consumo amortizado esperado por operação foi reduzido a $O(\lg^2 n)$ [17] e, nesse mesmo ano, foi realizado o primeiro estudo experimental, avaliando diferentes versões do algoritmo de Frederickson [3].

Ainda em 1995, Nikoletseas, Reif, Spirakis e Yung [28] propuseram uma estrutura de dados para o problema **estocástico** de conexidade em grafos dinâmicos, em que a sequência de atualizações e consultas é escolhida ao acaso com probabilidade uniforme. Nessas condições, essa estrutura de dados permite fazer atualizações em tempo $O(\lg^3 n)$ esperado amortizado e consulta em tempo $O(1)$ esperado amortizado.

Em 1999, Fatourou, Spirakis, Zarafidis, e Zoura [13] implementaram os algoritmos de Henzinger e King [18] e de Nikoletseas et al. [28] e os compararam experimentalmente. Eles concluíram que a estrutura de dados de Nikoletseas et al. tem uma performance melhor do que a de Henzinger e King quando o grafo dinâmico estocástico é denso.

Inspirados na solução de Henzinger e King para o problema de conexidade dinâmica, em 1998, Holm, de Lichtenberg e Thorup [20] atacaram ambos os problemas de conectividade dinâmica e de MSF usando Euler tour trees. A solução deles para o problema de conexidade empata com o consumo assintótico da solução de Henzinger e King de $O(\lg^2 n)$. No entanto, se a implementação das Euler tour trees for determinística, então o algoritmo para conexidade dinâmica de Holm et al. é completamente determinístico e seu consumo é somente amortizado, enquanto que o consumo na solução de Henzinger e King é esperado e amortizado. Apresentaremos a solução de Holm et al. nesse texto com uma implementação aleatorizada de Euler tour trees. Avaliações experimentais foram feitas sobre essa solução [2, 22, 14]. Comentaremos esses experimentos no Capítulo 7.

O algoritmo de Holm et al. para MSF possui consumo amortizado $O(\lg^4 n)$ para cada operação, sendo assim a primeira estrutura de dados determinística com consumo de tempo poli-logarítmico para o problema MSF. Uma implementação eficiente deste algoritmo foi apresentada por Cattaneo et al. [5] junto a um outro algoritmo mais simples e assintoticamente pior, mas que teve bom desempenho nos experimentos práticos realizados.

Em 2000, Thorup [38] complementou o algoritmo de Holm et al. [20], adicionando uma estrutura de dados auxiliar chamada floresta estrutural, que reduz o consumo de tempo de cada operação para $O(\lg n\,(\lg\lg n)^3)$ esperado amortizado e $O(\frac{\lg n}{\lg\lg\lg n})$ para a consulta de conexidade. Ela só não é ótima por um fator de $O((\lg\lg n)^3)$, devido ao limitante inferior de $\Omega(\lg n)$ provado em 2006 por Patrascu e Demaine [31].

Em 2010, Tarjan e Werneck [36] fizeram um estudo experimental com diversas estruturas de dados para árvores dinâmicas que resolvem ambos os problemas de conexidade em florestas dinâmicas e da floresta maximal de peso mínimo, destacando qualidades e deficiências de cada estrutura de dados quando sujeitas a diferentes tipos de cargas de trabalho.

Desde 1997, os algoritmos com consumo de tempo amortizado foram predominantes na literatura, porém todos possuem alguma instância com consumo de tempo $\Theta(n)$ no pior caso. O problema de melhorar o tempo assintótico no pior caso continuou a permear a literatura sem avanços até 2013, quando Kapron, King e Mountjoy [24] apresentaram uma estrutura de dados para o problema de conexidade em grafos dinâmicos baseada no problema *cutset* que proporciona consumo $O(\lg^4 n)$ para a inserção de arestas, $O(\lg^5 n)$ para remoção e responde à consulta de conexidade em tempo $O(\frac{\lg n}{\lg\lg n})$. A estrutura responde a consultas corretamente quando a resposta é "sim" e com alta probabilidade de acerto quando a resposta é "não". A probabilidade de falso positivo, mesmo que baixa, continuou incentivando a pesquisa na área.

Em 2015, Kejlberg-Rasmussen et al. [25] apresentaram uma estrutura de dados para o problema de conexidade em grafos dinâmicos que permite fazer consultas em tempo constante, mas que possui consumo de tempo para as atualizações de $O\left(\sqrt{\frac{n(\lg\lg n)^2}{\lg n}}\right)$, um consumo de tempo assintótico consideravelmente maior do que os últimos algoritmos citados.

Em 2016, Wulff-Nilsen [41] apresentou um novo algoritmo para o problema de conexidade em grafos dinâmicos que dá suporte às atualizações com consumo $O(\frac{\lg^2 n}{\lg\lg n})$ amortizado e $O(\frac{\lg n}{\lg\lg n})$ para a consulta de conexidade.

Em 2017, Huang et al. [21] reorganizam a solução de Thorup [38] obtendo uma solução com consumo de tempo de $O(\lg n\,(\lg\lg n)^2)$ esperado amortizado para cada atualização e $O(\frac{\lg n}{\lg\lg\lg n})$ para as consultas de conexidade. Essa é atualmente a estrutura de dados com menor consumo de tempo assintótico conhecida para solucionar o problema de conexidade em grafos dinâmicos.

Em 2022, Chen et al. [6] propuseram uma heurística baseada no diâmetro do grafo dinâmico e com ela desenvolveram um algoritmo que possui bom desempenho prático quando aplicado a grafos dinâmicos extraídos de situações reais. O consumo de tempo dessa heurística é linear em função do diâmetro do grafo, portanto as operações de atualização e

de consulta possuem consumo de tempo de $O(n)$ no pior dos casos. Veja um resumo dos resultados mencionados nesta seção na Tabela 1.1

ano	inserção	remoção	consulta	tipo	ref.
1985	$O(\sqrt{m})$	$O(\sqrt{m})$	$O(1)$	determinístico; pior caso	[15]
1992	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(1)$	determinístico; pior caso	[10]
1995	$O(\lg^3 n)$	$O(\lg^3 n)$	$O(\lg n)$	aleatorizado; amortizado	[18]
1997	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(\lg n)$	aleatorizado; amortizado	[17]
1998	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(\lg n)$	determinístico; amortizado	[20]
2000	$O(\lg n(\lg\lg n)^3)$	$O(\lg n(\lg\lg n)^3)$	$O(\frac{\lg n}{\lg\lg\lg n})$	aleatorizado; amortizado	[38]
2013	$O(\lg^4 n)$	$O(\lg^5 n)$	$O(\lg n/\lg\lg n)$	determinístico; pior caso	[24]
2015	$O\left(\sqrt{\frac{n(\lg\lg n)^2}{\lg n}}\right)$	$O\left(\sqrt{\frac{n(\lg\lg n)^2}{\lg n}}\right)$	$O(1)$	determinístico; pior caso	[25]
2016	$O(\frac{\lg^2 n}{\lg\lg n})$	$O(\frac{\lg^2 n}{\lg\lg n})$	$O(\frac{\lg n}{\lg\lg n})$	determinístico; amortizado	[41]
2017	$O(\lg n(\lg\lg n)^2)$	$O(\lg n(\lg\lg n)^2)$	$O(\frac{\lg n}{\lg\lg\lg n})$	aleatorizado; amortizado	[21]
2022	$O(n)$	$O(n)$	$O(n)$	determinístico; pior caso	[6]

Tabela 1.1: *Consumo de tempo de estruturas de dados para o problema de conexidade em grafos dinâmicos.*

Capítulo 2

Conexidade em florestas dinâmicas

O problema de conexidade em grafos dinâmicos, descrito na Seção 1.1, pode ser reduzido ao caso em que o grafo é uma floresta, despertando assim o interesse no **problema de conexidade em florestas dinâmicas**. Detalharemos como essa redução é feita no Capítulo 4. Esse problema pode ser apresentado como a implementação da seguinte biblioteca da forma mais eficiente possível:

- NOVAFD(n): cria e retorna uma floresta dinâmica com n vértices isolados;
- LIGUEFD(F, u, v): adiciona a aresta uv à floresta dinâmica F;
- REMOVAFD(F, u, v): remove a aresta uv de F; e
- CONECTADOFD(F, u, v): retorna verdadeiro se u e v estão na mesma componente conexa de F e falso, caso contrário.

Há na literatura uma estrutura de dados bem conhecida chamada **link cut trees** [34] que resolve uma versão direcionada desse problema, em que as árvores da floresta são enraizadas. Com essa estrutura de dados e uma rotina adicional, que permite mudar a raiz de uma dada árvore para um de seus outros nós, as link cut trees resolvem também a versão não direcionada do problema, ou seja, o problema de conexidade em florestas dinâmicas. Na Seção 5.7.2 introduziremos as link cut trees como solução para um outro problema envolvendo grafos dinâmicos. Nessa seção apresentaremos as Euler tour trees, uma solução mais simples e tão eficiente quanto a solução com link cut trees.

2.1 Euler tour trees

Tarjan e Vishkin [37] propuseram a **representação por trilha Euleriana** de uma árvore (originalmente chamada de *Euler tour technique*). Essa representação é obtida de uma árvore T substituindo-se cada aresta por dois arcos em sentidos opostos e adicionando-se um laço a cada vértice, como pode ser visto na Figura 2.1. O digrafo resultante é **Euleriano**, ou seja, é conexo e possui uma trilha que começa e termina num mesmo vértice e que

passa por todos os arcos do digrafo exatamente uma vez. Uma tal trilha é chamada de **trilha Euleriana** do digrafo.

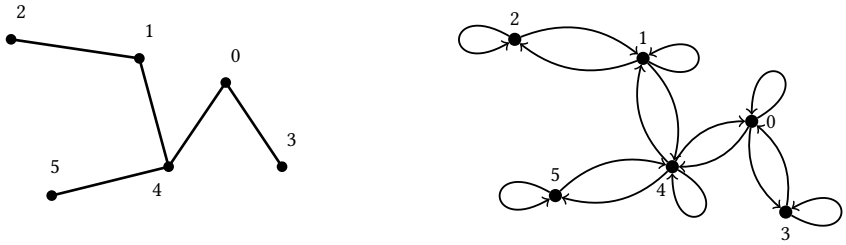


Figura 2.1: *Um exemplo de árvore e sua transformação como um digrafo a ser usado para sua representação por trilha Euleriana.*

A representação da árvore T é essencialmente uma sequência de arcos que forma uma trilha Euleriana do digrafo correspondente a T. Denotamos cada arco pelo par de vértices que o compõe. Isto é, o arco com origem no vértice u e destino no vértice v é escrito como uv. Dessa forma, um laço em v é escrito como vv. Utilizaremos esses laços como representantes dos vértices na sequência. Por exemplo, a sequência (2.1) é uma trilha Euleriana do digrafo correspondente à árvore da Figura 2.1.

$$30\ 00\ 04\ 41\ 12\ 22\ 21\ 11\ 14\ 44\ 45\ 55\ 54\ 40\ 03\ 33. \tag{2.1}$$

Note que a sequência depende do vértice inicial e da ordem em que cada vizinho de cada vértice é visitado. Chamaremos uma tal sequência de **sequência Euleriana** da árvore T.

Henzinger e King [18] propuseram armazenar uma sequência Euleriana em uma **árvore binária de busca** (ABB), que é uma árvore binária composta por nós que possuem quatro campos: chave, pai, filho esquerdo e filho direito [8].

Os campos pai e filhos descrevem a estrutura de árvore binária à ABB. Cada nó N possui até dois filhos (esquerdo e direito), o campo **pai** de cada um dos filhos aponta para N. Nenhum nó é filho de dois outros nós. Pode ocorrer de um nó não possuir algum dos filhos; nesse caso, os campos correspondentes a filhos inexistentes contêm NIL. Somente um nó não possui pai: este é chamado de **raiz** da ABB.

Para uma árvore binária ser considerada de busca, é necessário que, para todo nó N, todas as chaves da subárvore esquerda sejam menores do que a chave de N e, simetricamente, todas as chaves da subárvore direita sejam maiores do que a chave de N.

Na representação de Henzinger e King, cada nó da ABB guarda um elemento da sequência Euleriana, ou seja, um par de vértices da árvore T, em um campo adicional **info**, e armazena, no campo **chave**, um valor entre 1 e n correspondente ao índice desse elemento na sequência, onde n é o comprimento da sequência.

Por exemplo, a ABB na Figura 2.2 está armazenando a sequência Euleriana (2.1). Uma tal ABB é chamada de **Euler tour tree**. Henzinger e King propuseram representar uma floresta por uma coleção de Euler tour trees: uma para cada componente da floresta. Dessa forma, como veremos na Seção 2.2, é possível obter uma implementação de uma floresta

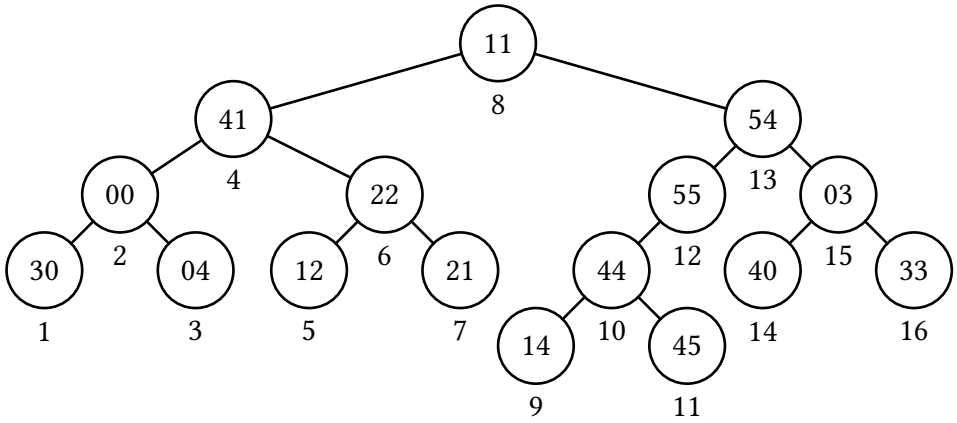


Figura 2.2: *Sequência* (2.1) *armazenada em uma ABB. Dentro do círculo mostramos o arco armazenado no nó e abaixo do círculo está sua chave.*

dinâmica em que as operações de consulta de conexidade e de inserção e remoção de aresta têm consumo esperado $O(\lg n)$, onde n é o número de vértices da floresta.

2.2 Implementação de floresta dinâmica com Euler tour trees

Para implementar a biblioteca de florestas dinâmicas com Euler tour trees, consideremos inicialmente a seguinte biblioteca.

- NOVONÓ(u, v): cria e retorna uma ABB com somente um nó que armazena o par de vértices uv;
- RAIZ($nó$): retorna a raiz da ABB que contém $nó$;

A implementação dessas rotinas será detalhada no Capítulo 3. Veremos que o consumo esperado de NOVONÓ e RAIZ será, respectivamente, $O(1)$ e $O(\lg n)$, onde n é o número de nós na ABB.

Para representar uma floresta dinâmica F, precisaremos também de um dicionário que guardará apontadores para os nós das ABBs que representam F. Note que cada nó representa um elemento uv de uma sequência Euleriana e que cada uv ocorre no máximo uma vez nas sequências. Então o par (u, v) será usado como chave e o valor correspondente a tal chave será o apontador para o nó que contém uv nas ABBs que representam F.

Para simplificar os pseudocódigos, usaremos uma representação matricial para esse dicionário, ou seja, usaremos a seguinte biblioteca.

- $F \leftarrow$ NOVODICIO(n): cria e retorna um dicionário F para uma floresta dinâmica com n vértices;
- $F[u, v] \leftarrow uv$: insere o nó que contém uv, com chave (u, v) e valor associado uv na tabela F. Se o par (u, v) já estiver presente no dicionário, então seu valor associado é substituído por uv;

- $F[u, v] \leftarrow$ NIL: remove o nó associado a (u, v) e seu valor associado do dicionário F;
- $var \leftarrow F[u, v]$: atribui o valor associado à chave (u, v) à variável var; Caso a chave (u, v) não esteja presente em F, atribui NIL a var.

Existem implementações bem conhecidas de dicionários em que a primeira rotina consome $O(n)$ e as demais consomem tempo esperado $O(1)$ [8].

No Algoritmo 1 apresentamos a implementação de NOVAFD, que cria e retorna uma nova floresta dinâmica que possui n vértices e nenhuma aresta. Já no Algoritmo 2 mostramos a implementação de CONECTADOFD, que responde à consulta de conexidade entre dois vértices u e v na floresta F.

Algoritmo 1 NOVAFD(n)

```
1 F ← NOVODICIO(n)
2 para v ← 1 até n faça
3     F[v, v] ← NOVONÓ(v, v)
4 retorne F
```

Com essa implementação NOVAFD consume tempo $O(n)$. A rotina CONECTADOFD, descrita no Algoritmo 2, consume tempo esperado $O(\lg n)$ em uma floresta com n vértices.

Algoritmo 2 CONECTADOFD(F, u, v)

```
1 uu ← F[u, u]
2 vv ← F[v, v]
3 retorne RAIZ(uu) = RAIZ(vv)
```

Para implementar LIGUEFD e REMOVAFD, precisaremos adicionalmente das seguintes rotinas na biblioteca de Euler tour trees.

- CORTA($nó$): corta a ABB que contém um nó $nó$ em três ABBs. A primeira ABB contém todos os nós com chave estritamente menor do que a chave de $nó$, a segunda contém somente $nó$ e a última contém todos os nós com chave estritamente maior do que a chave de $nó$. Essa rotina retorna as raízes dessas três ABBs; e
- JUNTA(T, R): junta as ABBs T e R de modo que a sequência armazenada na árvore resultante é a concatenação das sequências armazenada em T e R e retorna a raiz dessa árvore resultante.

Na Seção 3.2, mostraremos a implementação dessas rotinas e que JUNTA consome $O(h)$, onde h é a soma das alturas de T e R e CORTA consome $O(\lg n)$, onde n é número de nós da árvore que contém $nó$. Nos pseudocódigos que se seguem, usaremos JUNTA em sequências $T_1, \ldots, T_k$ de ABBs. Para manter os códigos mais legíveis, vamos abreviar chamadas repetidas de JUNTA, isto é, em vez de escrever JUNTA(JUNTA(... (JUNTA(T_1, T_2), T_3) ...), T_k) escreveremos simplesmente JUNTA($T_1, \ldots, T_k$).

Para implementar a operação REMOVAFD(F, u, v), descrita no Algoritmo 3, primeiro utilizaremos o dicionário para obter ponteiros para os nós que armazenam os arcos uv e vu.

Em seguida, aplicaremos CORTA(uv) dividindo essa sequência em duas partes, nomeadas A e B, como pode ser visto na Figura 2.3.

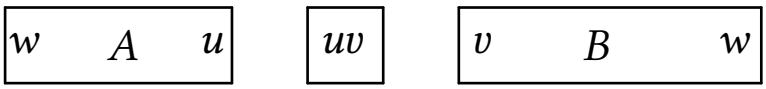


Figura 2.3: *Sequências A e B após a chamada de* CORTA(uv) *em* REMOVAFD(F, u, v).

Como a sequência original é Euleriana, sabemos que o primeiro e último vértice dessa sequência coincidem. Na Figura 2.3 chamamos esse vértice de w. Dessa forma A representa uma trilha de w até u e B representa uma trilha de v até w.

Note que não sabemos se vu está na sequência A ou na sequência B, mas não precisaremos dessa informação, pois podemos fazer a concatenação de B com A, chamando JUNTA(B, A), obtendo uma trilha de v até u que passa pelo arco vu em algum ponto.

Por fim, para concluir esse algoritmo, basta chamar CORTA(vu) para dividir essa sequência em duas. A primeira sendo a sequência Euleriana que representa a árvore que contém o vértice v e a segunda sendo a sequência Euleriana que contém o vértice u. Omitimos desse pseudocódigo a liberação dos nós da estrutura que não são mais usados.

Algoritmo 3 REMOVAFD(F, u, v)

```
1 uv ← F[u, v]
2 vu ← F[v, u]
3 A, uv, B ← CORTA(uv)
4 JUNTA(B, A)
5 CORTA(vu)
6 F[u, v] ← NIL
7 F[v, u] ← NIL
```

Notemos que as linhas 1, 2, 6 e 7 do Algoritmo 3 consumem tempo esperado constante, enquanto que as linhas 3, 4 e 5, e consequentemente REMOVAFD, possuem consumo de tempo $O(\lg n)$, onde n é o número de vértices de F.

Como exemplo, veremos o que ocorre com a sequência (2.1) durante a execução da chamada REMOVAFD(F, 1, 4). Primeiro, após a chamada de CORTA(14), obtemos árvores correspondentes às seguintes duas sequências:

$$A = 30\ 00\ 04\ 41\ 12\ 22\ 21\ 11 \qquad B = 44\ 45\ 55\ 54\ 40\ 03\ 33.$$

Nesse caso, temos que $w = 3$. Após JUNTA(B, A), temos uma árvore que representa a sequência

$$44\ 45\ 55\ 54\ 40\ 03\ 33\ 30\ 00\ 04\ 41\ 12\ 22\ 21\ 11.$$

Por fim, após CORTA(41), temos árvores que representam as duas sequências

$$44\ 45\ 55\ 54\ 40\ 03\ 33\ 30\ 00\ 04 \quad \text{e} \quad 12\ 22\ 21\ 11. \tag{2.2}$$

Note que as duas sequências resultantes representam a floresta obtida pela remoção da aresta 14, que está ilustrada na Figura 2.4.

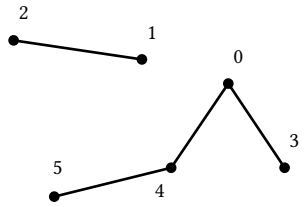


Figura 2.4: *Floresta resultante de* REMOVAFD*(F,* 1*,* 4*), representada pelas sequências em* (2.2)*.*

Para implementar LIGUEFD, descrita no Algoritmo 5, utilizamos a rotina auxiliar MOVAINÍCIO. Essa rotina recebe uma floresta F e um vértice u e restrutura a ABB que contém uu de forma que este se torne o primeiro elemento de sua sequência Euleriana, e retorna a raiz da ABB resultante.

Por exemplo, se aplicarmos MOVAINÍCIO(F,2) e MOVAINÍCIO(F,5), onde F é a floresta dinâmica ilustrada na Figura 2.4, obtemos as sequências:

$$55\ 54\ 40\ 03\ 33\ 30\ 00\ 04\ 44\ 45 \quad \text{e} \quad 22\ 21\ 11\ 12. \tag{2.3}$$

Para implementar MOVAINÍCIO(F, u), basta cortar a sequência em uu chamando CORTA(uu) e concatenar as sequências resultantes de forma apropriada com JUNTA. Note que, como CORTA remove uu da sequência, temos que adicioná-lo novamente à sequência como pode ser visto na linha 3 do Algoritmo 4.

Algoritmo 4 MOVAINÍCIO(F, u)

```
1 uu ← F[u, u]
2 A, uu, B ← CORTA(uu)
3 retorne JUNTA(uu, B, A)
```

Notemos que, devido às linhas 2 e 3 do Algoritmo 4, o consumo de tempo de MOVAINÍCIO é $O(\lg n)$. Com a rotina MOVAINÍCIO implementada, podemos elaborar LIGUEFD, descrita no Algoritmo 5.

Algoritmo 5 LIGUEFD(F, u, v)

```
1 U ← MOVAINÍCIO(F, u)
2 V ← MOVAINÍCIO(F, v)
3 uv ← NOVONÓ(u, v)
4 vu ← NOVONÓ(v, u)
5 F[u, v] ← uv
6 F[v, u] ← vu
7 JUNTA(U, uv, V, vu)
```

Primeiro, usamos MOVAINÍCIO para mover uu e vv para o início de suas sequências. Em seguida, criamos novos nós uv e vu; os adicionamos à tabela de símbolos e usamos JUNTA

pra unir todas as sequências de tal forma que a sequência resultante seja a sequência Euleriana correspondente à árvore resultante da adição da aresta uv.

Dessa forma, se quisermos adicionar uma aresta ligando os vértices 2 e 5 na floresta da Figura 2.4, obtendo assim a floresta da Figura 2.5, primeiro temos que mover 22 e 55 para o início de suas sequências, como fizemos com as sequências (2.3). Em seguida criamos os nós contendo 25 e 52 e unimos essas sequências, obtendo assim a sequência:

$$55\ 54\ 40\ 03\ 33\ 30\ 00\ 04\ 44\ 45\ 52\ 22\ 21\ 11\ 12\ 25. \tag{2.4}$$

Logo, o consumo esperado de LIGUEFD também será $O(\lg n)$.

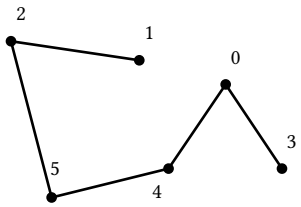


Figura 2.5: *Floresta resultante de LIGUEFD(F, 2, 5), representada pela sequência (2.4).*

Capítulo 3

Árvore binária de busca com chave implícita

O objetivo desse capítulo é mostrar uma implementação eficiente para a biblioteca das Euler tour trees:

- NOVONÓ(u, v): cria e retorna uma ABB com somente um nó que armazena o par de vértices uv;
- RAIZ($nó$): retorna a raiz da ABB que contém $nó$;
- CORTA($nó$): corta a ABB que contém um nó $nó$ em três ABBs. A primeira ABB contém todos os nós com chave estritamente menor do que a chave de $nó$, a segunda contém somente $nó$ e a última contém todos os nós com chave estritamente maior do que a chave de $nó$. Essa rotina retorna as raízes dessas três ABBs; e
- JUNTA(T, R): junta as ABBs T e R de modo que a sequência armazenada na árvore resultante é a concatenação das sequências armazenada em T e R e retorna a raiz dessa árvore resultante.

Note que cada uma das ABBs envolvidas nas Euler tour trees representa uma sequência Euleriana e os nós de cada ABB têm como chaves os inteiros de 1 a t, onde t é o número de nós da ABB.

A operação JUNTA terá o efeito de alterar a chave dos nós da árvore R para os inteiros de $t + 1, \ldots, t + r$, onde t e r são, respectivamente, o número de nós nas árvores T e R. Já a operação CORTA terá o efeito de alterar as chaves da segunda e da terceira ABBs devolvidas, para que fiquem entre 1 e t, onde t é o número de nós da ABB.

Se as chaves forem armazenadas explicitamente nos nós das ABBs, não há como garantir uma implementação muito eficiente. Aqui apresentaremos uma implementação que omite as chaves dos nós, e assim admite implementações para JUNTA e CORTA que consomem tempo esperado $O(\lg n)$, onde n é o número de nós nas ABBs envolvidas nas operações.

Para reduzir o consumo de tempo de JUNTA e CORTA, na biblioteca de Euler tour trees, as ABBs utilizadas possuirão **chave implícita**. Especificamente, substituímos o campo

chave de cada nó pelo campo **tam** que armazena o tamanho da subárvore enraizada naquele nó, isto é, o número de nós nessa subárvore. Com esse novo campo, a chave de cada nó pode ser determinada em tempo proporcional à profundidade do nó na ABB.

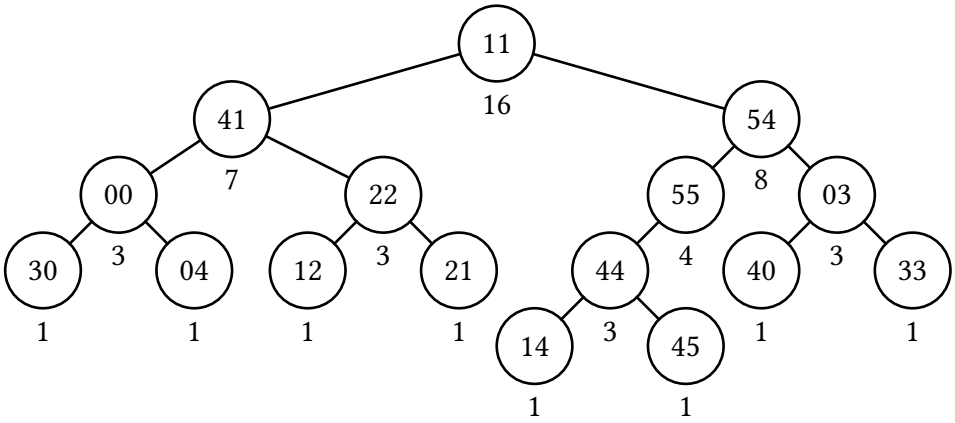


Figura 3.1: *Árvore da Figura 2.2 exibindo o valor do campo tam abaixo de cada nó.*

Com essa mudança, em JUNTA e CORTA, será necessário ajustar apenas o campo *tam* de um número bem mais reduzido de nós, o que resultará em uma implementação com o consumo de tempo desejado.

Como manipularemos muitos ponteiros a ABBs que podem conter NIL, é conveniente a adição da rotina interna TAMANHO(T), descrita no Algoritmo 6, que retorna 0 caso T seja NIL e, no caso em que T aponte para uma ABB não vazia, retorna seu tamanho. Essa rotina consome tempos O(1).

Algoritmo 6 TAMANHO(T)

```
1 se T = NIL então
2     retorne 0
3 retorne T.tam
```

Com chaves implícitas, a alteração de uma subárvore enraizada num nó x torna necessário somente a atualização do campo *tam* dos nós do caminho entre o nó x e a raiz da ABB que contém x, ou seja, o consumo de tempo dessas atualizações será assintoticamente proporcional à altura da ABB. Resta então o desafio de manter a ABB balanceada. Na próxima seção apresentaremos a estrutura de dados treap, que resolve esse desafio sem onerar o custo assintótico das operações ou adicionar demasiada complexidade aos algoritmos.

3.1 Treaps

Heaps são árvores binárias quase completas constituídas por nós que possuem quatro campos: prioridade, pai e filhos esquerdo e direito [8]. Os campos referentes aos pais e filhos dão a estrutura de árvore binária ao heap e a **prioridade** de um nó é um número real não negativo. Não apresentaremos a definição de árvore binária quase completa, pois

ela é desnecessária para a definição de treaps. Para que uma árvore binária quase completa seja considerada um heap é necessário que a prioridade de cada nó seja maior do que a de seus filhos.

Treaps são uma mescla entre árvores binárias de busca e heaps. Seus nós possuem cinco campos: *pai*, *esq*, *dir*, *chave* e *prio*. Os campos *esq* e *dir* representam os filhos esquerdo e direito de cada nó, respectivamente, e junto ao campo *pai* descrevem a estrutura de uma árvore binária. O campo *chave* satisfaz a propriedade de uma ABB enquanto o campo *prio* armazena a prioridade do nó e satisfaz a propriedade de um heap. Diferentemente de heaps, treaps não precisam ser quase completas.

Tais árvores foram inicialmente nomeadas **árvores cartesianas** [40], pois podemos representar cada nó como um par ordenado em que a primeira coordenada é a chave do nó e a segunda coordenada é sua prioridade. Ao visualizar esses pares imersos no plano cartesiano, como feito na Figura 3.2, ambas as estruturas de ABB e heap são bem representadas. Isto é, nós com maior prioridade ficam ilustrados acima de nós com menor prioridade e os nós ficam ordenados de forma crescente, da esquerda para a direita, em função de suas chaves.

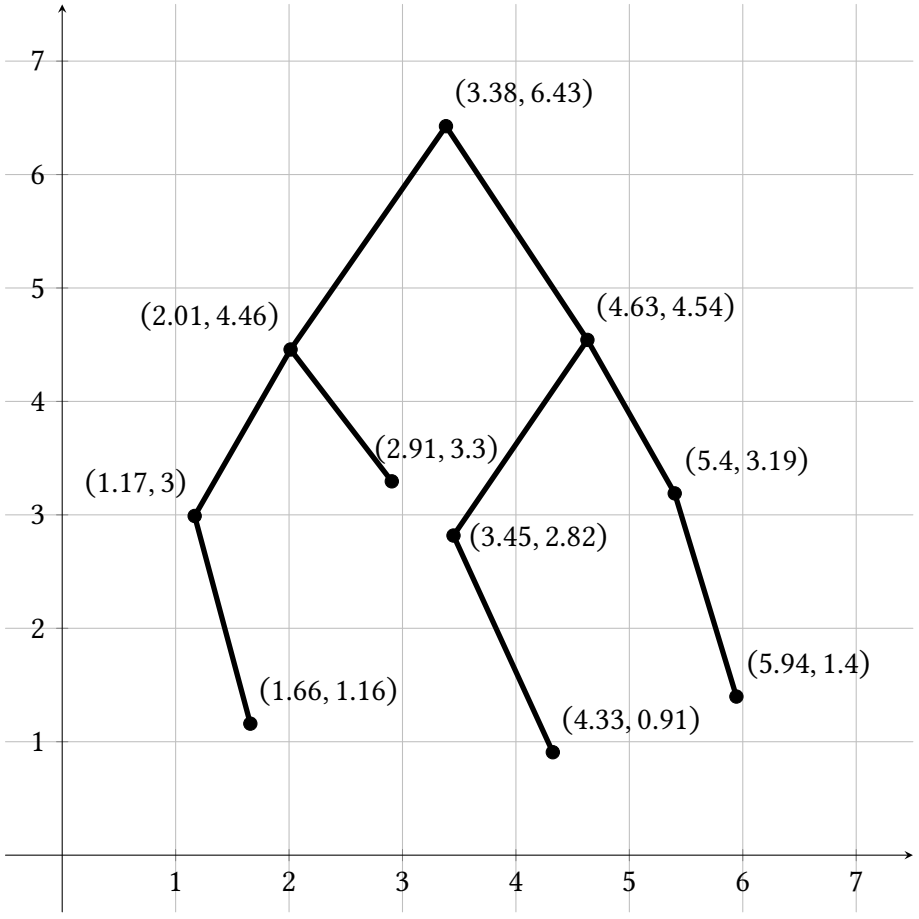


Figura 3.2: *Uma treap imersa no plano cartesiano.*

3.2 Treaps implícitas

Treaps implícitas são treaps com chaves implícitas [39].

A implementação da rotina RAIZ, apresentada no Algoritmo 7, independe da técnica que usaremos para balancear a treap, pois é uma consulta e somente usa a estrutura da ABB sem fazer modificações. Seu consumo de tempo é O(h) onde h é a altura da árvore.

Algoritmo 7 RAIZ($nó$)

```
1 raiz ← nó
2 enquanto raiz.pai ≠ NIL faça
3     raiz ← raiz.pai
4 retorne raiz
```

Para balancear uma treap e assim garantir consumo de tempo logarítmico, Aragon e Seidel [4, 33] propuseram escolher a prioridade de cada nó de forma aleatória com distribuição de probabilidade uniforme em um universo suficientemente grande para que a probabilidade de haver nós com a mesma prioridade seja próxima de 0.

Para representar essa escolha aleatória, usaremos uma função auxiliar SORTEIE() que retorna um número real entre 0 e 1 escolhido com probabilidade uniforme. Aproximações para esse tipo de função está presente nativamente em diversas linguagens de programação e a elaboração de sua implementação foge do escopo desse texto. Consideraremos que seu consumo de tempo é O(1).

Com essa técnica de balanceamento de treaps em mãos, podemos apresentar a implementação dos demais algoritmos que compõem essa biblioteca. O primeiro desses é NOVONÓ(u, v), que recebe vértices u e v, cria um nó de treap chamado $nó$, inicializa seus campos apropriadamente e o retorna. Essa rotina consome tempo O(1).

Algoritmo 8 NOVONÓ(u, v)

```
1 nó.tam ← 1
2 nó.prio ← SORTEIE()
3 nó.info ← (u, v)
4 nó.esq ← nó.dir ← nó.pai ← NIL
5 retorne nó
```

A implementação da rotina JUNTA pode ser vista no Algoritmo 9. Essa rotina junta as ABBs T e R de modo que a sequência armazenada na árvore resultante seja a concatenação das sequências armazenada em T e R e retorna a raiz dessa árvore resultante.

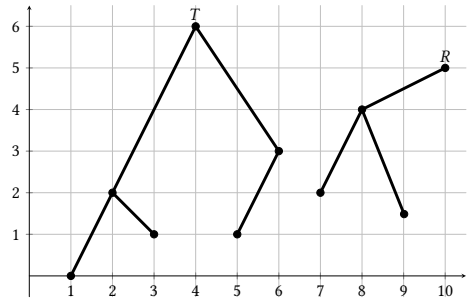
Para compreender o funcionamento de JUNTA podemos utilizar a ideia da imersão das treaps no plano cartesiano para visualizar melhor suas estruturas. Na Figura 3.3a, podemos ver um exemplo de treaps T e R.

Após tratar os casos em que T ou R podem ser NIL, precisamos definir o nó que será a raiz da junção das duas árvores. Este nó será aquele com maior prioridade, assim mantendo a propriedade de heap. Essa comparação é feita na linha 3.

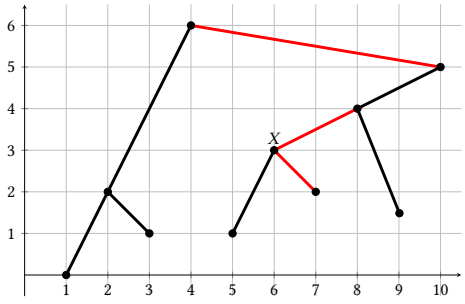
Algoritmo 9 JUNTA(T, R)

```
1  se T = NIL então retorne R
2  se R = NIL então retorne T
3  se T.prio > R.prio então
4      T.dir ← JUNTA(T.dir, R)
5      T.dir.pai ← T
6      T.tam ← T.tam + R.tam
7      retorne T
8  senão
9      R.esq ← JUNTA(T, R.esq)
10     R.esq.pai ← R
11     R.tam ← T.tam + R.tam
12     retorne R
```

Se a prioridade da raiz de T for maior do que a de R, como ilustrado na Figura 3.3a, então devemos juntar R a alguma subárvore de T. Como usamos chaves implícitas e a sequência contida em R ficará após a sequência contida em T depois da junção dessas árvores, temos que todas as chaves de R são estritamente maiores do que as chaves de T, logo temos que recursivamente juntar R com a subárvore enraizada no nó $T.dir$. Em seguida, corrigimos o campo *pai* de $T.dir$ e o campo *tam* de T.



(a) *Exemplo de duas treaps imersas no plano cartesiano.*



(b) *Resultado da junção das treaps T e R. Ponteiros modificados estão destacados em vermelho.*

Figura 3.3: *Antes e depois da junção das treaps T e R.*

Caso a prioridade de R seja maior do que a de T, então temos que juntar recursivamente T com a subárvore enraizada em $R.esq$. Esse caso é simétrico ao anterior. Podemos ver o resultado da junção das treaps T e R da Figura 3.3a na Figura 3.3b.

Notemos que JUNTA consome $O(h)$ onde h é a soma das alturas das duas árvores que são unidas. Como a altura esperada dessas árvores é $O(\lg n)$, temos que o consumo esperado de tempo dessa rotina também será logarítmico.

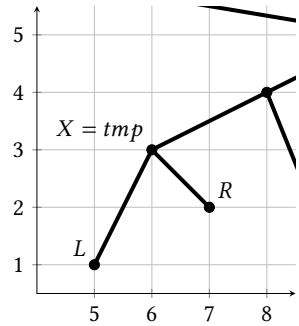
A implementação da rotina CORTA(*nó*) pode ser vista no Algoritmo 10. Nesse algoritmo percorremos o caminho de *nó* até a raiz de sua árvore reatribuindo a relação de parentesco entre os nós percorridos de forma a cortar a treap em duas: a primeira contendo todos os nós com chave menor do que a chave de *nó* e a segunda com todos os nós com chave maior do que *nó*. Para fazer isso mantemos três ponteiros: L, R e *tmp*. Os dois primeiros

apontam para as raízes das árvores resultantes desse corte e o terceiro aponta para o nó cuja relação de parentesco será modificada.

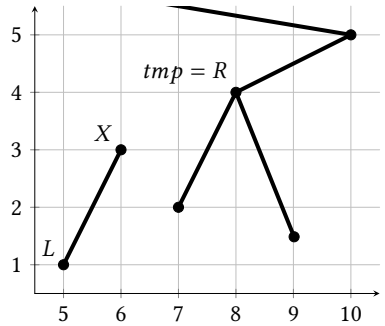
Algoritmo 10 CORTA($nó$)

```
1  R ← nó.dir; L ← nó.esq; tmp ← nó
2  enquanto tmp.pai ≠ NIL faça
3      se tmp.pai.esq = tmp então
4          tmp.pai.esq ← R
5          tmp.pai.tam ← tmp.pai.tam − TAMANHO(L)
6          se R ≠ NIL então
7              R.pai ← tmp.pai
8          R ← tmp.pai
9      senão
10         tmp.pai.dir ← L
11         tmp.pai.tam ← tmp.pai.tam − TAMANHO(R)
12         se L ≠ NIL então
13             L.pai ← tmp.pai
14         L ← tmp.pai
15     tmp ← tmp.pai
16 se L ≠ NIL então L.pai ← NIL
17 se R ≠ NIL então R.pai ← NIL
18 nó.dir ← nó.esq ← nó.pai ← NIL
19 retorne L, nó, R
```

Simularemos a execução de CORTA(X), onde X é o nó de coordenadas $(6, 3)$ da Figura 3.3b. O estado inicial dos ponteiros L e R são os filhos de X e tmp aponta para X, como mostra a Figura 3.4a.



(a) *Estado inicial dos ponteiros L, R e tmp.*



(b) *Estado após atualização dos ponteiros de parentesco e tmp.*

Figura 3.4: *Dois estados da simulação de* CORTA(X).

Se tmp for filho esquerdo de seu pai, isto é, $tmp.pai.esq = tmp$, então a chave de $tmp.pai$ é maior do que a chave de X e ele logo deve ser um nó de R. Além disso, sabemos que a prioridade e chave de $tmp.pai$ são maiores do que a prioridade e chave de R, logo R torna-se o filho esquerdo de $tmp.pai$ e $tmp.pai$ deve ser a nova raiz de R. Após atualizar os

campos de parentesco entre R, *tmp* e *tmp.pai*, atualizamos o valor de *tmp* para *tmp.pai*, obtendo o estado mostrado na Figura 3.4b. Então repetimos esse processo.

Na próxima iteração desse laço, repetimos o processo descrito no parágrafo anterior, pois temos que o nó apontado por *tmp* é filho esquerdo de seu pai. Já na próxima iteração, fazemos as operações simétricas, pois teremos que o nó apontado por *tmp* é filho direito de seu pai. Ao término dessa iteração, obtemos as treaps da Figura 3.5.

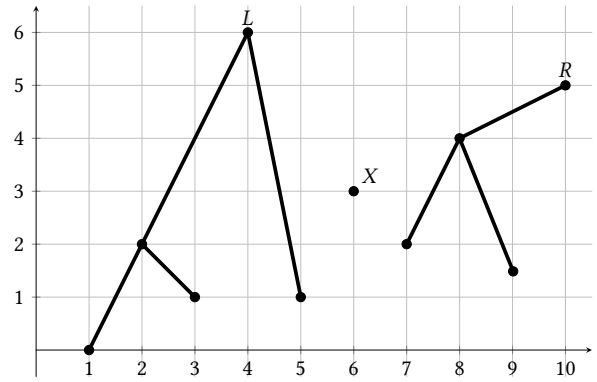


Figura 3.5: *Resultado de* CORTA*(X).*

Notemos que CORTA consiste em um laço que percorre os nós de uma treap até sua raiz e em cada nó realiza operações de custo constante, logo seu consumo de tempo é proporcional à altura da treap e, como essa é balanceada, temos que seu consumo esperado é $O(\lg n)$.

Capítulo 4

Conexidade em grafos dinâmicos

Retomemos o problema de conexidade em grafos dinâmicos, apresentado na Seção 1.1, que é um dos problemas principais que vamos estudar. Ele consiste na busca por uma implementação tão eficiente quanto possível para a seguinte biblioteca:

- NOVOGD(n): cria e devolve um grafo dinâmico com n vértices isolados;
- LIGUEGD(G, u, v): adiciona a aresta uv ao grafo dinâmico G;
- REMOVAGD(G, u, v): remove a aresta uv de G; e
- CONECTADOGD(G, u, v): retorna verdadeiro se u e v estão na mesma componente conexa de G e falso, caso contrário.

Note que dois vértices estão na mesma componente conexa de G se e somente se estão na mesma componente de alguma floresta maximal de G. Dessa forma, a ideia que usaremos para responder a consulta de conexidade em nosso grafo dinâmico é manter, ao longo da sequência de inserções e remoções de arestas, uma floresta dinâmica F que seja maximal em G e, quando ocorrer a consulta CONECTADOGD, chamamos a consulta de conexidade na floresta F.

Veremos como manter F em cada operação da biblioteca de grafos dinâmicos. A rotina NOVOGD retorna um grafo composto por n vértices isolados. Logo uma floresta maximal para esse grafo consiste também em n vértices isolados.

Em uma chamada LIGUEGD(G, u, v), primeiro testamos a conexidade de u com v em G. Se esses vértices não estiverem conectados em G, então inserimos a aresta uv na floresta maximal que estamos mantendo, assim ligando as duas árvores que contém u e v nessa floresta. Essas arestas serão chamadas de **arestas da floresta** ou **arestas titulares**.

Já no caso em que u e v estiverem conectados em G, então a aresta sobressalente uv será chamada de **aresta reserva** e iremos armazená-la em um grafo auxiliar R, armazenado em listas de adjacências. Para manusear R, usamos a seguinte biblioteca:

- NOVOGRAFO(n): cria e devolve um grafo mantido com listas de adjacências com n vértices isolados.
- LIGUEGLA(R, u, v): adiciona u na lista de adjacências de v em R e vice-versa.
- REMOVAGLA(R, u, v): remove u da lista de adjacências de v em R e vice-versa.

É possível implementar as listas de adjacências de modo que o NOVOGRAFO tenha custo $O(n)$, LIGUEGLA tenha custo $O(1)$ e o REMOVAGLA, desde que de posse da ocorrência de v na lista de u, tenha custo $O(1)$. Na nossa implementação [32], que é em Python 3, usamos um dicionário para cada lista, que garante tempo esperado $O(1)$ para essas operações mesmo sem ter em mãos uma ocorrência de v na lista de u.

Vimos, na Seção 2, que a inserção de arestas em uma floresta dinâmica com n vértices custa tempo esperado $O(\lg n)$. Assim o consumo esperado de LIGUEGD será $O(\lg n)$.

Para remover uma aresta reserva, basta removê-la das listas de adjacências. Já remover de G uma aresta uv da floresta F é bem mais complexo, pois sua remoção de uma componente T de F gera duas árvores T_u e T_v, que contêm os vértices u e v, respectivamente. Esse cenário pode ser visto na Figura 4.1. Para manter a propriedade de F ser maximal em G, é necessário verificar se existe alguma aresta reserva em R que liga T_u a T_v. Tal aresta é chamada de **aresta substituta**.

Sem perda de generalidade, podemos supor que o número de vértices em T_u é menor ou igual ao número de vértices de T_v. Então, para encontrar uma aresta substituta caso tal aresta exista, podemos percorrer cada vértice t de T_u verificando se existe algum vértice w na lista de adjacências de t que não seja vértice de T_u. Caso w não seja vértice de T_u, como F era maximal, teremos que w necessariamente é um vértice de T_v e assim a aresta tw é uma substituta para a aresta uv que foi removida.

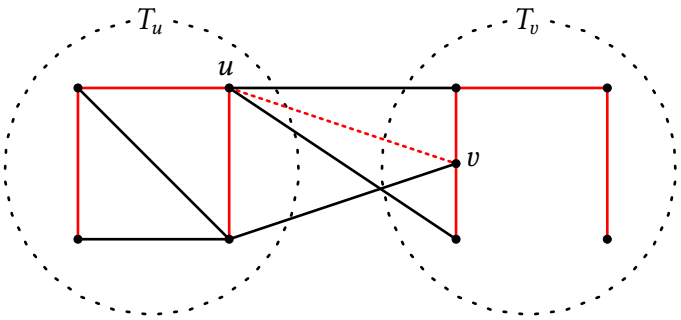


Figura 4.1: *Exemplo de grafo dinâmico com as arestas da árvore T coloridas de vermelho, enquanto que as arestas reservas estão pintadas de preto. A aresta uv removida está pontilhada.*

Para testar rapidamente se t e w estão na mesma árvore em F depois da remoção de uv, basta acionar a rotina CONECTADOFD(F, t, w), que vimos que consome tempo esperado $O(\lg n)$. A rotina CONECTADOFD, no pior dos casos, em que não há aresta substituta e T_u tem $\Theta(n)$ vértices, será chamada $\Theta(n^2)$ vezes, o que implica em um consumo esperado de pior caso de $O(n^2 \lg n)$ para REMOVAGD implementado dessa forma.

Notemos que somente a remoção de arestas de F é lenta e isso se deve à necessidade de busca por uma aresta substituta. Para obter uma implementação melhor, é necessário

reduzir o número de arestas testadas para encontrar uma substituta. A técnica que apresentaremos no restante desse capítulo deve-se a Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup [20] e tem como objetivo a redução deste número de forma amortizada. Após uma rápida introdução de notação, poderemos implementar NOVOGD, LIGUEGD e CONECTADOGD como descrito anteriormente, sendo necessários mais artifícios para obtermos uma implementação eficiente de REMOVAGD.

4.1 Fatiamento do grafo em níveis

Cada aresta do grafo possuirá um **nível** entre 1 e $\lceil \lg n \rceil$, onde n é o número de vértices do grafo. O valor inicial do nível de uma aresta recém inserida é $\lceil \lg n \rceil$ e é decrementado toda vez que percorremos a aresta em busca de uma aresta substituta. O nível de uma aresta nunca aumentará, somente diminuirá.

Dado um grafo G, podemos definir $G_{\leqslant i}$ como o grafo induzido pelas arestas de nível menor ou igual a i. Para cada nível i, manteremos uma floresta maximal $F_{\leqslant i}$ de $G_{\leqslant i}$ e o grafo R_i, guardado com listas de adjacências, dado pelas arestas reservas de nível i. Trivialmente temos que $G = G_{\leqslant \lceil \lg n \rceil}$ e portanto $F_{\leqslant \lceil \lg n \rceil}$ é uma floresta maximal de G. Ao longo da sequência de modificações realizadas no grafo, manteremos algumas invariantes importantes:

(i) $F_{\leqslant i}$ é uma floresta maximal de $G_{\leqslant i}$ para todo $1 \leqslant i \leqslant \lceil \lg n \rceil$; e

(ii) $F_{\leqslant i} \subseteq F_{\leqslant i+1}$ para todo $1 \leqslant i \leqslant \lceil \lg n \rceil - 1$;

(iii) Cada componente de $F_{\leqslant i}$ possui no máximo 2^i vértices.

A intuição por trás desse fatiamento é que, quando uma aresta titular de nível i é removida, não é necessário buscar por substitutas nos níveis menores que i. Começamos a busca no próprio nível i, ou seja, em R_i e caso não encontremos uma substituta em R_i, passamos a buscar em $R_{i+1}, R_{i+2}, \ldots, R_{\lceil \lg n \rceil}$. Quando não encontramos uma substituta em um certo R_j, aproveitamos para rebaixar as arestas percorridas de R_j, de modo a postergar futuros percursos a estas arestas. Tudo isso será detalhado e melhor explicado adiante.

Na próxima seção iremos, para cada rotina da biblioteca de grafos dinâmicos, descrever como a rotina opera, como ela preserva as invariantes acima e por fim calcular seu consumo de tempo.

4.2 Implementação

4.2.1 Criação de um grafo dinâmico

A implementação de NOVOGD pode ser vista no Algoritmo 11. Nessa implementação, simplesmente inicializamos cada $F_{\leqslant i}$ e R_i de G usando NOVAFD e NOVOGRAFO, respectivamente. Notemos que essas rotinas retornam a floresta e o grafo dados por n vértices isolados. Dessa forma $F_{\leqslant \lceil \lg n \rceil}$ e $R_{\lceil \lg n \rceil}$ juntos representam um grafo dinâmico vazio, que é exatamente o grafo que queremos construir com NOVOGD.

Será útil obter o nível de uma aresta uv em tempo esperado constante. Assim manteremos um dicionário G.NÍVEL, com as arestas de G, usando como chave para a aresta uv o par $\{u, v\}$ e como valor o nível da aresta uv. Para manusear esse dicionário, usaremos uma biblioteca análoga à usada na Seção 2.2, que relembramos a seguir.

- NÍVEL ← NOVODICIO(n): cria e retorna um dicionário NÍVEL para arestas de um grafo com n vértices;
- NÍVEL$[u, v] \leftarrow i$: insere a aresta uv com valor associado i. Se a aresta uv já estiver presente no dicionário, então seu valor associado é substituído por i;
- NÍVEL$[u, v]$ ← NIL: remove a aresta uv e seu valor associado do dicionário;
- $var \leftarrow$ NÍVEL$[u, v]$: atribui o valor associado à aresta uv à variável var.

Quando for claro, nos referenciaremos a esse dicionário no pseudocódigo somente por NÍVEL.

Algoritmo 11 NOVOGD(n)

```
1 para i ← 1 até ⌈lg n⌉ faça
2     G.F_{≤i} ← NOVAFD(n)
3     G.R_i ← NOVOGRAFO(n)
4 G.NÍVEL ← NOVODICIO(n)
5 retorne G
```

Notemos que as invariantes são mantidas por NOVOGD, pois o grafo gerado por essa rotina são n vértices isolados. Considerando o consumo de tempo de NOVAFD, NOVOGRAFO e NOVODICIO, o consumo de tempo dessa rotina será $O(n \lg n)$.

4.2.2 Consulta de conexidade

Para realizar a rotina CONECTADOGD, apresentada no Algoritmo 12, somente retornamos a resposta da consulta de conexidade feita em $F_{\leqslant \lceil \lg n \rceil}$. A corretude desse algoritmo se deve ao invariante (i), pois esse invariante garante que $F_{\leqslant \lceil \lg n \rceil}$ é uma floresta maximal de G, logo consultas de conectividade entre os vértices u e v em G e em $F_{\leqslant \lceil \lg n \rceil}$ devem possuir a mesma resposta.

Algoritmo 12 CONECTADOGD(G, u, v)

```
1 retorne CONECTADOFD(G.F_{≤⌈lg n⌉}, u, v)
```

A rotina CONECTADOGD claramente não interfere nas invariantes, já que é somente uma consulta que não modifica o grafo nem as florestas que estamos mantendo.

Como o consumo esperado de CONECTADOFD é $O(\lg n)$, é imediato ver que o consumo esperado de tempo de CONECTADOGD também é $O(\lg n)$.

4.2.3 Adição de arestas

Para inserir uma nova aresta uv em G usando a rotina LIGUEGD, implementada no Algoritmo 13, primeiro verificamos se os vértices u e v estão conectados em G, acionando CONECTADOFD($G.F_{\leqslant\lceil\lg n\rceil}$, u, v). Caso u e v estejam conectados em G, então a aresta uv é uma aresta reserva e será inserida em $R_{\lceil\lg n\rceil}$. Caso u e v não estejam conectados, então a aresta uv deve ser inserida em $F_{\leqslant\lceil\lg n\rceil}$.

Algoritmo 13 LIGUEGD(G, u, v)

```
1 NÍVEL[u,v] ← ⌈lg n⌉
2 se CONECTADOFD(G.F_{≤⌈lg n⌉}, u, v) então
3     LIGUEGLA(G.R_{⌈lg n⌉}, u, v)
4 senão
5     LIGUEFD(G.F_{≤⌈lg n⌉}, u, v)
```

Note que a invariante (i) é mantida para $i = \lceil\lg n\rceil$ e as demais invariantes também se mantêm, pois somente o nível $\lceil\lg n\rceil$ da nossa estrutura de dados foi modificado, já que a nova aresta é inserida no nível $\lceil\lg n\rceil$.

O consumo esperado de tempo de CONECTADOFD e LIGUEFD é $O(\lg n)$ e o consumo esperado de LIGUEGLA é $O(1)$. Logo o consumo esperado de tempo de LIGUEGD também é $O(\lg n)$.

4.2.4 Remoção de arestas

A complexidade da remoção de uma aresta em um grafo dinâmico vem da busca por uma aresta substituta para a aresta removida. Podemos encapsular essa busca em uma rotina própria chamada SUBSTITUAGD(G, u, v, niv), que recebe como parâmetros um grafo dinâmico G, dois vértices u e v e um inteiro niv com $1 \leqslant niv \leqslant \lceil\lg n\rceil$.

Essa rotina é chamada assim que a aresta uv, que possui nível niv, foi removida de G e das florestas $F_{\leqslant j}$ para $j \geqslant niv$ e encontra, caso exista, uma aresta substituta em G com nível mínimo, e a insere na floresta deste nível e nas de nível acima. Dessa forma, a implementação de REMOVAGD, descrita no Algoritmo 14, pode ser feita em poucas linhas.

Algoritmo 14 REMOVAGD(G, u, v)

```
1 i ← NÍVEL[u, v]
2 NÍVEL[u, v] ← NIL
3 se uv ∈ G.F_{≤⌈lg n⌉} então
4     para j ← i até ⌈lg n⌉ faça
5         REMOVAFD(G.F_j, u, v)
6     SUBSTITUAGD(G, u, v, i)
7 senão
8     REMOVAGLA(G.R_i, u, v)
```

Para remover uma aresta uv de nível i de G, primeiro precisamos determinar se ela pertence à floresta $F_{\leqslant\lceil\lg n\rceil}$ ou não, o que é feito na linha 3 do Algoritmo 14. Para realizar

esse teste, é feita uma consulta no dicionário $F_{\leqslant \lceil \lg n \rceil}$ verificando se há alguma entrada associada à chave (u, v).

Caso a aresta uv não seja titular, ela é uma aresta de R_i e somente a removemos de R_i, o que é feito na linha 8. Caso uv seja titular, então precisamos removê-la das florestas $F_i, F_{i+1}, \ldots, F_{\lceil \lg n \rceil}$. Devido ao invariante (ii), sabemos que uv está em todas essas florestas e somente nelas, então essa remoção é feita no laço da linha 4. A rotina SUBSTITUAGD faz as alterações devidas à busca de uma aresta substituta, e sua eventual inclusão nas florestas apropriadas de forma a manter as invariantes. Veremos que o SUBSTITUAGD garante que cada $F_{\leqslant j}$ é maximal em $G_{\leqslant j}$, isto é, preserva a invariante (i) e preserva também as invariantes (ii) e (iii).

O laço da linha 4 do Algoritmo 14 terá consumo esperado $O(\lg^2 n)$, pois executa no pior dos casos $\lceil \lg n \rceil$ vezes a rotina REMOVAFD, que possui consumo esperado $O(\lg n)$. Veremos que SUBSTITUAGD possui consumo amortizado $O(\lg n^2)$. Assim concluímos que o consumo de tempo esperado amortizado de REMOVAGD será $O(\lg^2 n)$.

Agora passamos a descrever o algoritmo SUBSTITUAGD, elaborado no Algoritmo 15. Para ajudar na compreensão desse algoritmo, vamos retomar o processo de busca de uma aresta substituta que foi comentado no início dessa seção, mas agora agregando a estrutura de níveis e as invariantes que definimos. Para tal, ilustramos, na Figura 4.2, o grafo da Figura 4.1 com a estrutura de níveis e com a aresta uv já removida. Consideramos que todas as arestas desse grafo são de nível i e também que não há aresta de nível $i-1$ imediatamente antes da chamada de SUBSTITUAGD(G, u, v, i).

Algoritmo 15 SUBSTITUAGD(G, u, v, niv)

```
1  para i ← niv até ⌈lg n⌉ faça
2      T_v ← RAIZ(F_i[v, v])
3      T_u ← RAIZ(F_i[u, u])
4      se TAMANHO(T_v) < TAMANHO(T_u) então          ▷ Garantimos que |T_v| ⩾ |T_u|
5          u ↔ v
6          T_u ↔ T_v
7      para xy em T_u com nível = i faça             ▷ Move T_u para o nível i − 1
8          NÍVEL[x, y] ← i − 1
9          LIGUEFD(G.F_{i−1}, x, y)
10     para xy em G.R_i com x em T_u faça            ▷ Procura substituta para uv
11         REMOVAGLA(G.R_i, x, y)
12         se y ∈ T_v então
13             para j ← i até ⌈lg n⌉ faça
14                 LIGUEFD(G.F_j, x, y)
15             retorne
16         senão
17             NÍVEL[x, y] ← i − 1
18             LIGUEGLA(G.R_{i−1}, x, y)
```

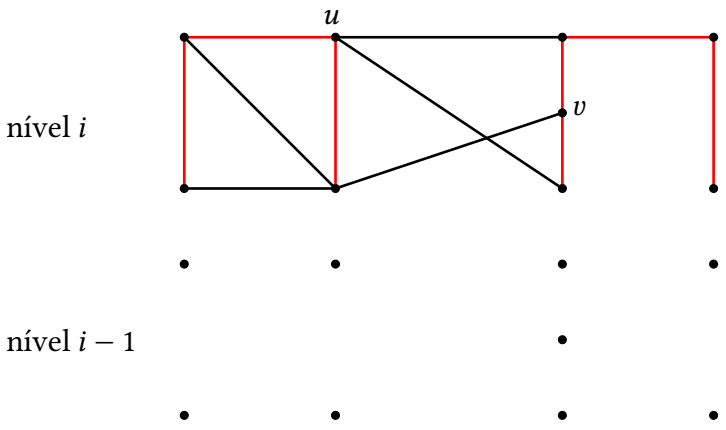


Figura 4.2: *Grafo da Figura 4.1 imerso no nível i com aresta uv removida.*

Após a remoção de uma aresta uv de nível i de uma componente T da floresta $F_{\leqslant i}$, a árvore T é dividida em duas árvores, T_u e T_v, que contém u e v, respectivamente. Denotamos por $|T|$ o número de vértices de uma árvore T. Podemos supor que $|T_u| \leqslant |T_v|$. Pela invariante (iii), vale que $|T| \leqslant 2^i$ e, como $|T_u| + |T_v| = |T|$, concluímos que $|T_u| \leqslant 2^{i-1}$. Logo podemos mover todas as arestas de nível i de T_u para o nível $i - 1$ sem infringir a invariante (iii). Esse rebaixamento é feito no laço da linha 7 do Algoritmo 15 e ilustramos a estrutura resultante dele na Figura 4.3.

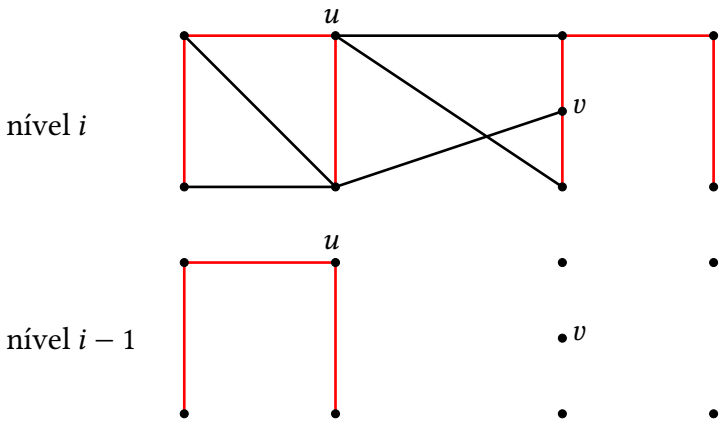


Figura 4.3: *Grafo dinâmico após o rebaixamento de T_u.*

Agora percorreremos as arestas reservas em busca de uma substituta. Notemos que, como consequência das invariantes (i) e (ii), se há uma aresta substituta para uv, então seu nível é maior ou igual a i. Provaremos esse fato por contradição, supondo que exista uma aresta xy substituta a uv com nível $j < i$. Como xy é uma aresta de nível j, temos que xy é uma aresta de $G_{\leqslant j}$, logo x e y estão na mesma componente de $G_{\leqslant j}$. Portanto, pela invariante (i), x e y estão na mesma componente conexa de $F_{\leqslant j}$. Vamos analisar o caminho P em $F_{\leqslant j}$ que liga x a y. Pela invariante (ii), temos que $F_{\leqslant j} \subseteq F_{\leqslant i}$ e assim as arestas de P são arestas de $F_{\leqslant i}$. Como x e y passam a estar em componentes distintas de G quando removemos uv, temos que uv é uma aresta de P e assim uv está em $F_{\leqslant j}$, o que é uma contradição com o fato de uv ser uma aresta de nível i.

Percorremos as arestas reservas de nível i incidentes a T_u procurando uma aresta substituta. Essas arestas estão pintadas de preto na Figura 4.3. Cada aresta percorrida desta

forma e que não seja uma substituta de uv tem seus dois extremos em T_u e também será rebaixada. Rebaixamos tais arestas de nível, pois elas deixam de ser candidatas a substitutas de arestas de nível i, uma vez que rebaixamos as arestas de T_u de nível i para o nível $i-1$. Ademais esse rebaixamento não infringe a invariante (i). Caso encontremos uma substituta no nível i, então a inserimos nas florestas dos níveis i até $\lceil \lg n \rceil$. Caso não encontremos uma substituta no nível i, teremos rebaixado para o nível $i-1$ todas as arestas incidentes a T_u, e repetimos a busca no nível $i+1$, eventualmente rebaixando arestas de nível $i+1$ incidentes a T_u para o nível i, até encontrarmos uma aresta substituta ou terminarmos a busca no nível $\lceil \lg n \rceil$. Podemos ver o resultado desses rebaixamentos na Figura 4.4.

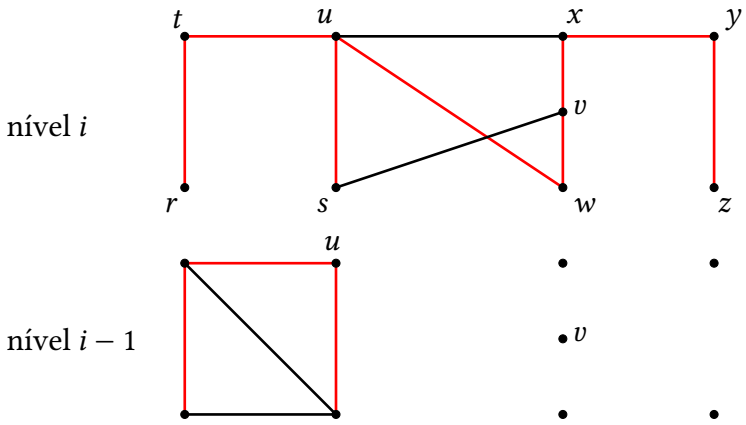


Figura 4.4: *Grafo dinâmico após encontrar uma aresta substituta para uv.*

A rotina SUBSTITUAGD preserva cada uma das invariantes que definimos. Como somente rebaixamos arestas da árvore e arestas reservas cujas duas extremidades estão em T_u, a componente resultante de $F_{\leqslant i-1}$ é maximal em $G_{\leqslant i-1}$. Portanto a invariante (i) é preservada. A invariante (ii) é preservada, pois rebaixar arestas de nível mantém essa invariante e ao inserir uma aresta que foi descoberta como substituta, garantimos que ela foi inserida em todos os níveis maiores do que i (laço da linha 13). Como garantimos que $|T_u| \leqslant 2^{i-1}$ antes de rebaixar as arestas de T_u, a invariante (iii) também é preservada.

Antes de analisar o consumo de tempo de SUBSTITUAGD, é necessário elaborar mais alguns detalhes sobre a implementação dessa rotina. Especificamente, analisaremos como realizamos os laços das linhas 7 e 10.

No laço da linha 7, percorremos o conjunto das arestas de nível i de T_u. Para acessar esse conjunto eficientemente, adicionaremos dois novos campos aos nós das Euler tour trees. O primeiro será um campo booleano, chamado *niv*, que valerá 1 somente se a aresta representada pelo nó for de nível i. Caso contrário, esse campo valerá 0. O segundo campo, chamado *cniv*, armazena o número de arestas de nível i na subárvore enraizada no nó. Podemos ver na Figura 4.5, como exemplo, a floresta $F_{\leqslant i}$ da Figura 4.4 representada por uma Euler tour tree.

Com esses dois campos, podemos adicionar uma nova rotina, chamada ARESTASDENÍVEL e descrita no Algoritmo 16, que percorre a Euler tour tree que armazena T_u evitando subárvores que não possuam arestas de nível i e constrói o conjunto de todas as arestas de nível i de T_u.

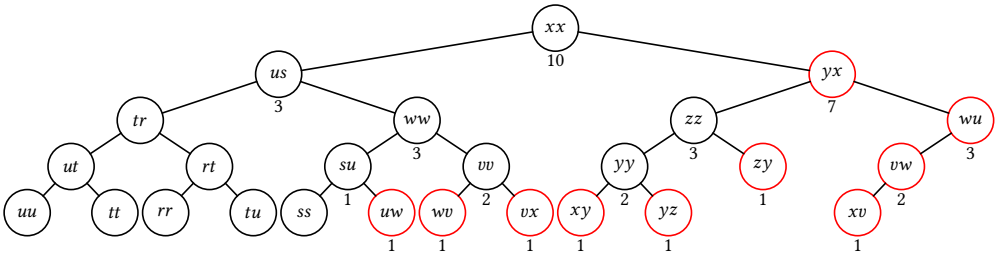


Figura 4.5: *Euler tour tree que armazena a floresta $F_{\leqslant i}$ da Figura 4.4. Os nós que representam arestas de nível i de $F_{\leqslant i}$ estão pintados de vermelho e o valor do campo cniv está denotado abaixo de cada nó, quando esse campo não for nulo.*

Algoritmo 16 ARESTASDENÍVEL(*nó*)

```
1 se nó = NIL ou nó.cniv = 0 então
2     retorne ∅
3 L ← ∅
4 L ← L ∪ ARESTASDENÍVEL(nó.esq)
5 se nó.niv = 1 então
6     L ← L ∪ {nó}
7 L ← L ∪ ARESTASDENÍVEL(nó.dir)
8 retorne L
```

Notemos que ARESTASDENÍVEL não altera o grafo nem as florestas ou nível das arestas, logo preserva as invariantes. Seu consumo de tempo esperado é $O(k \lg n)$, onde k é o número de arestas de nível i na árvore de $F_{\leqslant i}$ que contém o *nó*, que corresponde aos k percursos da raiz da Euler tour tree até cada nó com campo *niv* igual a 1. Como a Euler tour tree é balanceada, o consumo esperado de tempo de cada percurso é $O(\lg n)$.

Com essa nova rotina em mãos, realizar o laço da linha 7 do Algoritmo 15 se torna fácil. Primeiro construímos esse conjunto de arestas titulares de nível i e então o percorremos, rebaixando cada aresta para o nível $i - 1$.

A técnica utilizada para o laço da linha 10 é análoga, só que agora a informação que queremos extrair das Euler tour trees é o conjunto de vértices incidentes a alguma aresta reserva de nível i. Logo, adicionaremos dois outros novos campos aos nós das nossas Euler tour trees. O primeiro, chamado *res*, que é igual a 1 se v é incidente a alguma aresta reserva de nível i, caso contrário, valerá 0. O segundo será um contador, chamado de *cres*, que armazena o número de nós que satisfazem $res = 1$ na subárvore enraizada pelo nó. Podemos ver, na Figura 4.6, o valor desses campos para o grafo dinâmico de nível i da Figura 4.4. A manutenção desses campos é análoga à manutenção feita nos campos *niv* e *cniv*.

Podemos fazer a manutenção desses novos campos sem onerar o custo das outras rotinas. Toda nova aresta inserida em algum nível possui o campo *niv* igual a 1. Esse campo se torna 0 somente quando rebaixamos essa aresta de nível, o que ocorre somente no laço da linha 7 do Algoritmo 15. Nessa ocasião, precisamos percorrer o caminho do

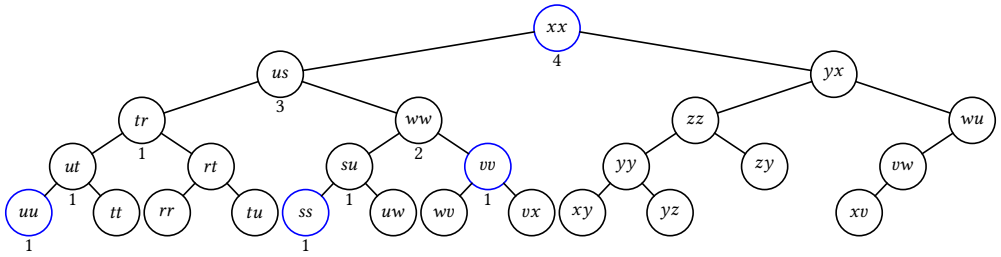


Figura 4.6: *Euler tour tree que armazena o grafo de nível i da Figura 4.4. Os nós que representam vértices incidentes a arestas reservas de nível i estão pintados de azul e o valor do campo cres está denotado abaixo de cada nó, quando esse campo não for nulo.*

nó que representa a aresta rebaixada até sua raiz decrementando *cniv*, o que consome tempo esperado $O(\lg n)$. Como cada iteração desse laço já consome tempo esperado $O(\lg n)$, manter esse campo não prejudica o consumo assintótico de tempo da rotina.

Além disso, precisamos manter *cniv* e *cres* atualizados nas operações CORTA e JUNTA da biblioteca de treaps, pois essas rotinas modificam a estrutura das treaps. A atualização desses campos consome tempo constante para cada nó da treap afetado pela modificação. Assim não onera o consumo de tempo dessas rotinas.

Além dessas ocasiões, o contador *cres* exige atualizações em situações adicionais. Toda vez que adicionamos ou removemos uma aresta reserva, devemos percorrer o caminho desse nó até a raiz atualizando o campo *cres*, o que consome tempo esperado $O(\lg n)$.

Para concluir essa seção, argumentaremos como ocorre a amortização de custo de REMOVAGD. A amortização ocorre em uma sequência arbitrária válida de m inserções e t remoções de arestas. Mostraremos que o custo esperado total destas $m + t$ operações é $O((m + t) \lg^2 n)$, o que resulta num custo esperado amortizado por operação de $O(\lg^2 n)$. Atribuiremos o custo de todos os rebaixamentos sofridos por uma aresta à operação de inserção dessa aresta. Precisamente, os custos das linhas 7, 8, 9, 17 e 18 do Algoritmo SUBSTITUAGD são pagos pela operação LIGUEGD correspondente.

Como cada uma das m arestas inseridas durante essa sequência pode ser rebaixada $\lceil \lg n \rceil$ vezes e cada rebaixamento (o custo das linhas mencionadas anteriormente) tem custo esperado $O(\lg n)$, o custo esperado total induzido pelas inserções de arestas é $O(m \lg^2 n)$. Descontadas essas linhas, SUBSTITUAGD, e consequentemente REMOVAGD, tem custo esperado $O(\lg^2 n)$. Portanto, todas as t remoções juntas têm custo esperado $O(t \lg^2 n)$ e assim o custo esperado total da sequência é $O((m + t) \lg^2 n)$, como queríamos mostrar.

Capítulo 5

Floresta maximal de peso mínimo em grafos planos

O problema da floresta maximal de peso mínimo em grafos dinâmicos planos foi inicialmente apresentado na Seção 1.1 e é o segundo problema que iremos discutir. O objetivo desse capítulo é apresentar o algoritmo de David Eppstein, Giuseppe Italiano, Roberto Tamassia, Robert Tarjan, Jeffery Westbrook e Moti Yung [12] que resolve esse problema para grafos planos que podem ter laços e arestas paralelas.

O estudo desse problema se inicia com uma revisão de conceitos de planaridade e dualidade apresentados nas Seções 5.1 e 5.2 respectivamente. Esses conceitos são bem estabelecidos na literatura e essas seções são baseadas na Seção 4.2 do livro *Graph Theory* de Reinhard Diestel [9].

Os conceitos apresentando nessas seções serão usados nas Seções 5.3 para definir formalmente o problema de floresta maximal de peso mínimo em grafos ponderados planos dinâmicos e na Seção 5.4 para fundamentar a estrutura de dados chamada árvores dinâmicas planas proposta por Eppstein et al. para solucionar esse problema. Em seguida, a biblioteca das árvores dinâmicas planas é definida na Seção 5.5 e na Seção 5.6 é mostrado como usar essa estrutura de dados para solucionar o problema MSF.

Por fim, na Seção 5.7 são apresentadas estruturas de dados auxiliares que serão usadas para implementar as árvores dinâmicas planas. Essa implementação é feita na Seção 5.8, que concluí o presente capítulo.

5.1 Planaridade

Intuitivamente, um grafo é dito plano se ele está desenhado em uma folha de papel de forma que suas arestas não se cruzem. Formalmente, um **grafo plano** [9] é um par de conjuntos finitos $G = (V, E)$ com as seguintes propriedades:

1. $V \subset \mathbb{R}^2$;
2. Toda aresta é um arco entre dois vértices;

3. O interior de uma aresta não contém vértices nem intersecta outras arestas.

Quando conveniente, G será usado tanto para o par (V, E) quanto para o conjunto $V \cup \bigcup E$.

Para cada grafo plano G, ao remover G do plano $\mathbb{R}^2$, é obtido um conjunto finito $F(G)$ de regiões conexas que é chamado de **conjunto de faces** de G. Naturalmente, cada uma dessas regiões é chamada de uma **face** de G. Uma dessas faces é ilimitada. Essa face é chamada de **face exterior**. A Figura 5.1 mostra um grafo plano G e suas as faces.

aresta	peso
a	2
b	7
c	3
d	1
f	2
g	4

Tabela 5.1: *Tabela de pesos*

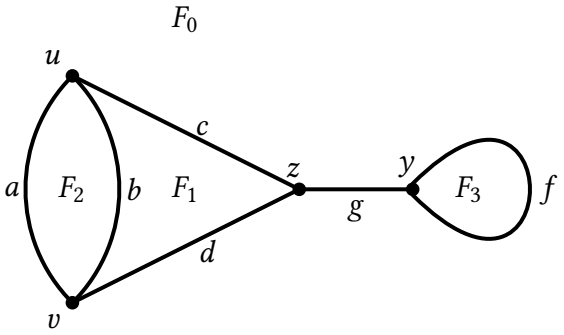


Figura 5.1: *Um grafo ponderado plano e suas faces.*

Lema 1 (Lemma 4.2.2 [9])**.** *Seja G um grafo plano e e uma de suas arestas.*

1. *Se X é a fronteira de uma face de G, então ou $e \subseteq X$ ou a intersecção de X com o interior de e é vazia.*

2. *Se e pertence a um ciclo de G, então e pertence à fronteira de exatamente duas faces distintas de G;*

3. *Se não existe ciclo que contém e, então e pertence à fronteira de uma única face de G.*

Para cada vértice v de um grafo plano G, é possível construir uma ordenação cíclica $D(v)$ das aresta incidentes a v [29]. Para obter essa ordenação, percorremos as arestas incidentes a v em sentido anti-horário até retornar à aresta inicial do percurso. Uma aresta é dita **sucessora** de outra aresta se a primeira é sucessora da segunda nesse percurso. Uma lista de todas essas ordens cíclicas é chamada de **descrição combinatória plana** de G.

A descrição combinatória plana é essencialmente uma lista de adjacências do grafo plano em que a ordem das células é relevante. Para o algoritmo que vamos descrever nesse capítulo, cada aresta possui um identificador e armazenamos também nas células das listas o identificador das arestas. Observe que cada aresta corresponde a duas células nas listas de adjacência do grafo: Se os extremos de uma aresta são u e v, há uma célula com u na lista de v e uma célula com v na lista de u, ambos representando essa aresta. Como estamos admitindo arestas paralelas, o número de ocorrências de um vértice u na lista de um vértice v será exatamente o número de arestas paralelas entre u e v. Um laço num vértice u corresponderá a duas ocorrências de u na sua lista de adjacências. Abaixo

temos uma descrição combinatória plana do grafo da Figura 5.1.

$$D(u) = \langle (a, v), (b, v), (c, z) \rangle$$
$$D(v) = \langle (a, u), (d, z), (b, u) \rangle$$
$$D(y) = \langle (e, z), (f, y), (f, y) \rangle$$

Pensamos nessas células como representações das duas possíveis orientações da aresta: de u para v (algumas vezes denotada por uv) e de v para u (algumas vezes denotada por vu). Essas duas células terão também um apontador *sym*, referenciando à outra e vice-versa.

5.2 Dualidade

Dado um grafo plano G, o grafo **dual** de G [9] é o grafo $G^\star = (F, E^\star)$ cujo conjunto de vértices é o conjunto $F = F(G)$ de faces de G e o conjunto de arestas $E^\star$ é construído a partir das arestas de G. Para cada aresta $e \in E$, existe exatamente uma aresta $e^\star$ no conjunto $E^\star$. A aresta $e^\star$ é chamada de **aresta dual** de e e os vértices incidentes a essa aresta são as duas faces (não necessariamente distintas) cujas fronteiras contém e. Nesse contexto, chamamos G de grafo **primal**. O grafo dual também é um grafo plano. Para ilustrar a relação entre cada aresta e seu dual, geralmente desenhamos cada aresta do grafo dual cruzando a correspondente aresta do grafo primal, como mostra a Figura 5.2.

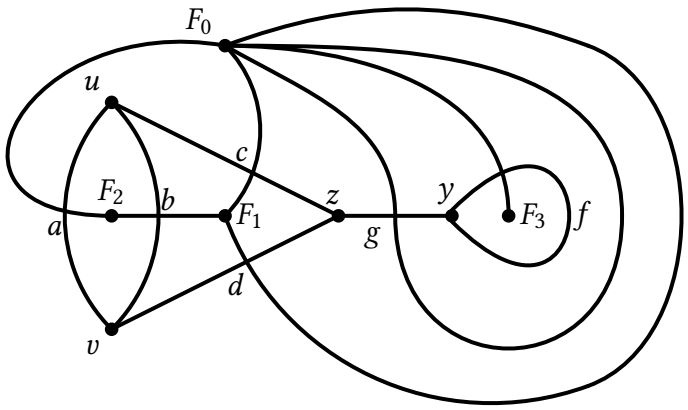


Figura 5.2: *Um grafo ponderado plano e seu dual.*

Teorema 2 (Fórmula de Euler, Teorema 4.2.9 [9])**.** *Seja $G = (V, E)$ um grafo planar conexo com n vértices e m arestas. Toda imersão de G no plano possui o mesmo número f de faces e*

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

Se G é um grafo plano com n vértices e m arestas, o Teorema 2 mostra que o número de vértices do seu grafo dual é $\Theta(n + m)$.

Teorema 3 ([12]). *Seja T uma árvore geradora de um grafo plano conexo G. O conjunto*

$$T^\star = \{e^\star : e \notin T\}$$

é uma árvore geradora de $G^\star$. Além disso, se G for ponderado e adotarmos $w(e^\star) = w(e)$, então T será de peso mínimo em G se e somente se $T^\star$ for de peso máximo em $G^\star$ e vice versa.

A Figura 5.3 mostra T e $T^\star$ dos grafos da Figura 5.2.

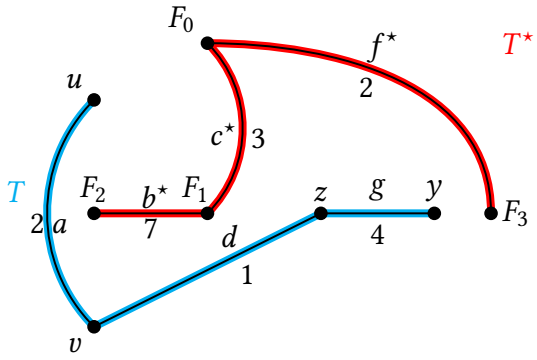


Figura 5.3: *Árvore geradora T, em azul, de peso mínimo do grafo ponderado da Figura 5.2 e a árvore correspondente $T^\star$ do seu grafo dual, em vermelho.*

5.3 Definição do problema

O problema da floresta maximal de peso mínimo (MSF) em grafos dinâmicos planos ponderados consiste na busca por uma implementação tão eficiente quanto possível para a seguinte biblioteca:

- NOVOGDP(n): Cria e devolve um grafo plano ponderado G com n vértices isolados.
- LIGUEGDP(G, e, u, e_u, v, e_v, w): Insere em G uma nova aresta e com peso w ligando os vértices u e v. A nova aresta e é sucessora das arestas e_u e e_v nas ordenações cíclicas de u e v, respectivamente.
- REMOVAGDP(G, e): Remove a aresta e de G.
- MUDAPESOGDP(G, e, w): Altera o peso da aresta e de G para o valor w.
- PESOGDP(G): Devolve o peso de uma MSF de G.

5.4 Árvores dinâmicas planas

Para resolver o problema da floresta geradora de peso mínimo em grafos planos ponderados dinâmicos, Eppstein et al. [12] propuseram o uso de uma estrutura de dados que chamaremos de **árvores dinâmicas planas** (ADP). Originalmente os autores denominaram essa estrutura de *edge ordered dynamic tree*. Com essa estrutura, eles obtém implementações de NOVOGDP e PESOGDP que consomem tempo constante e LIGUEGDP,

removaGDP e mudaPesoGDP que consomem tempo $O(\lg m)$ esperado amortizado, onde m é o número de arestas no grafo corrente.

Cada componente conexa do grafo plano dinâmico ponderado G é representada em um plano distinto. Consequentemente, a face exterior de cada componente conexa é considerada distinta das faces exteriores das outras componentes. Cada componente C de G é representada por duas árvores com pesos em seus *vértices*, sendo a primeira construída a partir de uma árvore geradora de peso mínimo T de C e a segunda a partir da correspondente dual $T^\star$. A Figura 5.4 ilustra essas duas árvores cuja construção será detalhada no próximo parágrafo.

A primeira dessas árvores, denotada por $\hat{T}$, é essencialmente a árvore T submetida a duas transformações: cada uma de suas arestas é subdivida; e é adicionada uma folha a cada vértice incidente a uma aresta de C que não esteja em T. Mais precisamente, $\hat{T}$ possui um vértice $\hat{v}$, de peso $-\infty$, para cada vértice v de C, um vértice $\hat{e}$, com o peso de e, para cada aresta e de T e dois vértices $\hat{d}_0$ e $\hat{d}_2$, ambos com o peso de d, para cada aresta d de C que não é uma aresta de T. Se uma aresta e de T é incidente aos vértices v e u de C, então o vértice $\hat{e}$ é ligado aos vértices $\hat{v}$ e $\hat{u}$. Se uma aresta d de C não está em T e é incidente aos vértices v e u de C, então $\hat{d}_0$ e $\hat{d}_2$ são ligados a $\hat{v}$ e $\hat{u}$, respectivamente. Note que $\hat{d}_0$ e $\hat{d}_2$ são folhas da árvore $\hat{T}$. Na Figura 5.4, a árvore $\hat{T}$ está colorida em azul.

A segunda árvore, denotada por $\hat{T}^\star$, possui construção análoga à primeira, mas é baseada em $T^\star$ em vez de T. Ela possui um vértice $\hat{F}$ para cada vértice F de $C^\star$ (isto é, para cada face de C), um vértice $\hat{e}^\star$ para cada aresta $e^\star$ de $T^\star$ e dois vértices $\hat{d}_1$ e $\hat{d}_3$ para cada aresta d de C que não está em $T^\star$. O peso do vértice $\hat{F}$ é ∞, enquanto que o peso de $\hat{e}^\star$ é o peso de e e o peso de $\hat{d}_1$ e $\hat{d}_3$ é o peso de d. Se uma aresta $e^\star$ de $T^\star$ é incidente às faces F_1 e F_2 de C, então o vértice $\hat{e}^\star$ é ligado aos vértices $\hat{F}_1$ e $\hat{F}_2$. Se uma aresta $d^\star$ de $C^\star$ não está em $T^\star$ e é incidente às faces F_1 e F_2 de C, então $\hat{d}_1$ e $\hat{d}_3$ são ligados a $\hat{F}_1$ e $\hat{F}_2$ respectivamente. Na Figura 5.4, a árvore $\hat{T}^\star$ está colorida em vermelho.

vértices	pesos
$\hat{a}, \hat{a}_1, \hat{a}_3$	2
$\hat{b}, \hat{b}_0, \hat{b}_2$	7
$\hat{c}, \hat{c}_0, \hat{c}_2$	3
$\hat{d}, \hat{d}_0, \hat{d}_2$	2
$\hat{f}, \hat{f}_1, \hat{f}_3$	1
$\hat{g}, \hat{g}_1, \hat{g}_3$	4
$\hat{u}, \hat{v}, \hat{y}, \hat{z}$	$-\infty$
$\hat{F}_0, \hat{F}_1, \hat{F}_2, \hat{F}_3$	∞

Tabela 5.2: *Tabela de pesos*

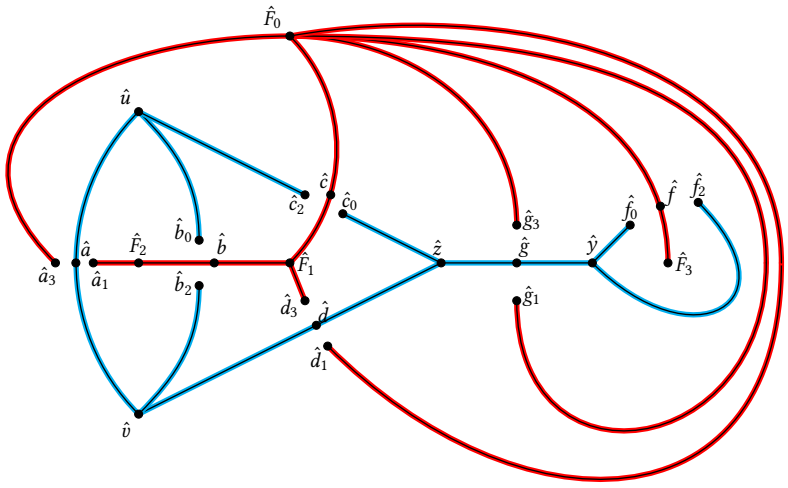


Figura 5.4: *As árvores $\hat{T}$ e $\hat{T}^\star$ construídas a partir das árvores da Figura 5.3.*

Note que há uma correspondência entre as arestas incidentes a um vértice v de G e os vértices vizinhos de $\hat{v}$ em $\hat{T}$. Cada aresta e na vizinhança de um vértice v de G corresponde ou a $\hat{e}$, $\hat{e}_0$ ou $\hat{e}_2$. Logo é possível construir, para cada vértice v de G, uma ordem cíclica dos vizinhos de $\hat{v}$ a partir da descrição combinatória plana de G. Por exemplo, se temos $D(u) = \langle (a, v), (b, v), (c, z) \rangle$, então a respectiva ordem cíclica de $\hat{u}$ é $\langle \hat{a}, \hat{b}_0, \hat{c}_2 \rangle$.

A implementação de árvores dinâmicas planas utiliza link cut trees e árvores binárias de busca com chave implícita. Árvores binárias de busca com chave implícita foram tratadas em detalhes no Capitulo 3 e sua biblioteca será expandida na Seção 5.7.1 para poder ser usada para implementar árvores dinâmicas planas. Já link cut trees serão brevemente abordadas na Seção 5.7.2.

Cada vértice $\hat{v}$ de $\hat{T}$ e $\hat{T}^\star$ é implementado por um conjunto de nós de link cut trees. Esse conjunto possui um nó e_v para cada aresta e na ordem cíclica de $\hat{v}$. Cada nó e_v é ligado aos nós que representam a aresta sucessora e predecessora de e na ordem cíclica de $\hat{v}$, com exceção do primeiro e do último na ordem, que são ligados somente ao seu sucessor e predecessor, respectivamente. Dessa forma, a ordem cíclica é representada por um caminho na link cut tree. Além disso, se a aresta e liga os nós $\hat{v}$ e $\hat{u}$, então e_v é ligado ao nó e_u que representa e na ordem cíclica de $\hat{u}$. A Figura 5.5 ilustra as duas link cut trees que representam as árvores $\hat{T}$ e $\hat{T}^\star$ ilustradas na Figura 5.4.

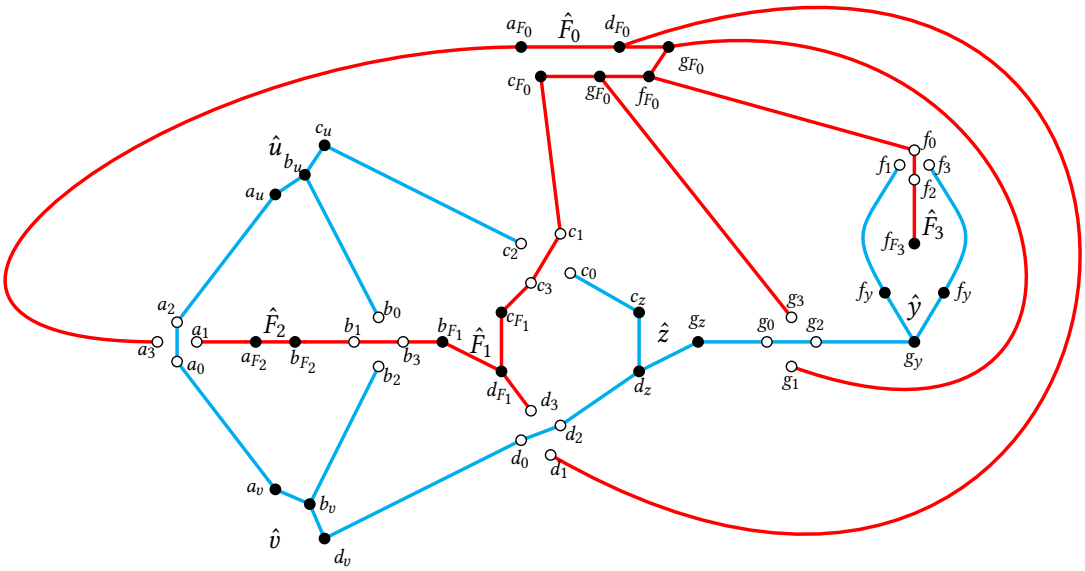


Figura 5.5: *Link cut trees que implementam $\hat{T}$ e $\hat{T}^\star$ ilustradas na Figura 5.4.*

Cada aresta e de G, que liga vértices u e v e cuja dual $e^\star$ liga vértices F_1 e F_2 de $G^\star$, é implementada por oito nós de link cut tree. Os quatro primeiros, denotados por e_0, e_1, e_2 e e_3, representam e. Os nós e_0, e_2 são nós de $\hat{T}$, enquanto que e_1, e_3 são nós de $\hat{T}^\star$. Os nós e_0 e e_2 representam as orientações uv e vu de e. Nesse caso se F_1 é a face à esquerda de e quando vamos de u para v, o nó e_1 representa a orientação F_1F_2 de $e^\star$ enquanto que e_3 representa a orientação F_2F_1 de $e^\star$. Isso pode ser visto na Figura 5.5.

Caso e seja uma aresta de T, então o vértice $\hat{e}$ de $\hat{T}$ é implementado pelos nós e_0, e_2 e a aresta entre eles. Já os nós e_1, e_3 individualmente formam as duas folhas $\hat{e}_1$, $\hat{e}_3$ de $\hat{T}^\star$. Caso d

seja uma aresta de $T^\star$, então o vértice $\hat{d}$ de $\hat{T}^\star$ é implementado pelos nós d_1, d_3 e a aresta entre eles. Já os nós d_0, d_2 individualmente formam as duas folhas $\hat{d}_0$, $\hat{d}_2$ de $\hat{T}$.

Os quatro últimos nós que representam e na link cut tree são e_u, e_v, $e^\star_{F_1}$ e $e^\star_{F_2}$, que mantém a posição de e e $e^\star$ nas ordens cíclicas de u, v, F_1 e F_2, respectivamente. Por exemplo, a ordem cíclica $D(u)$ é representada pela sequência de nós $\langle a_u, b_u, c_u \rangle$. No restante desse capítulo, será usado $D(u)$ para referenciar a sequência de nós de link cut tree que representam essa ordem cíclica. À frente, nos referiremos à óctupla $(e_0, e_1, e_2, e_3, e_u, e_v, {e^\star}_{F_1}, {e^\star}_{F_2})$ como a **óctupla** de e. A Figura 5.6 ilustra a óctupla das arestas b e g.

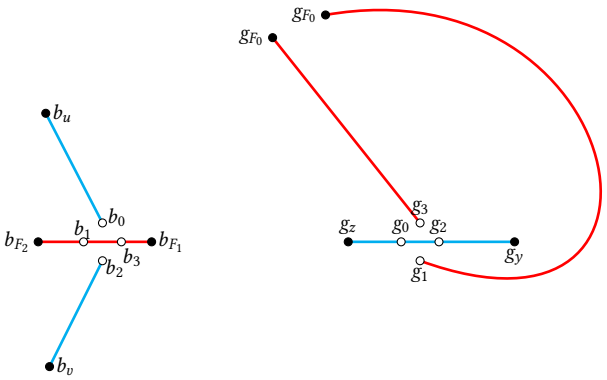


Figura 5.6: *Óctuplas das arestas b e g.*

Ademais, a sequência dos nós de link cut tree que representam as arestas de G incidentes a v é armazenada em uma árvore binária de busca com chave implícita. A ABB é utilizada para obter o primeiro e o último nó da ordem cíclica, o predecessor de um dado nó e identificar se dois nós estão na mesma ordem cíclica. A raiz dessa ABB possui um campo *nó* que armazena o identificador v.

Uma árvore dinâmica plana é uma link cut tree (como as ilustradas na Figura 5.5) que representa uma dada árvore em junção às ABBs que armazenam as sequências de nós que representam as ordens cíclicas de cada vértice dessa árvore. Para solucionar o problema da floresta maximal de peso mínimo em grafos dinâmicos planos, serão mantidas as duas ADPs $\hat{T}$ e $\hat{T}^\star$ e uma tabela de símbolos H, que guarda as óctuplas de cada aresta de G e usa como chave o identificador dessas arestas.

Dessa forma, G e uma floresta geradora maximal F de G de peso mínimo é representada por dois conjuntos de ADPs: o conjunto $\hat{F}$ das ADPs das componentes de F e o conjunto $\hat{F}^\star$ das ADPs do dual de cada componente de F.

Agora que a estrutura de ADPs foi apresentada, podemos introduzir na próxima seção a biblioteca dessa estrutura de dados.

5.5 Biblioteca de árvores dinâmicas planas

Árvores dinâmicas planas dão suporte à biblioteca listada a seguir. Para as descrições que se seguem, considere u e v vértices da árvore representada, p um nó da ordem

cíclica $D(u)$ e q um nó de $D(v)$.

- CREATE OCTO(e, w): Recebe um identificador e e um peso w e cria e retorna uma óctupla de nós de ADP associados a e com peso w.
- DESTROY OCTO(H, e): Recebe uma tabela de símbolos H e um identificador e e desaloca a óctupla associada a e da memória.
- CONNECTED(p, q): Retorna verdadeiro se os nós p e q estiverem na mesma ADP e falso caso contrário.
- FIND NODE(p): Recebe um nó p e retorna o vértice da $\hat{F}$ que contêm p em sua ordem cíclica.

  Por exemplo: Na Figura 5.5, FIND NODE(a_u) retorna $\hat{u}$, FIND NODE(b_{F_1}) retorna $\hat{F}_1$, FIND NODE(a_0) e FIND NODE(a_2) retornam $\hat{a}$, enquanto que FIND NODE(a_1) e FIND NODE(a_3) retornam $\hat{a}_1$ e $\hat{a}_3$, respectivamente.
- SET WEIGHT(p, w): Atribui o peso w ao vértice que contém p em sua ordem cíclica.
- FIND MAX(p, q): Retorna o nó de peso máximo no percurso entre os nós p e q. Essa operação assume que p e q são nós da mesma ADP.
- FIND MIN(p, q): Retorna o nó de peso mínimo no percurso entre os nós p e q. Essa operação assume que p e q são nós da mesma ADP.
- CYCLE(p): Permuta ciclicamente $D(u)$ de forma que o nó p seja o último na ordem. Se a ordem inicial tem forma $\alpha p \beta$, então a ordem resultante é $\beta \alpha p$.
- PRED(p): Retorna o predecessor do nó p na ordem cíclica.

É possível unir e separar nós das ADPs com as rotinas MERGE e SPLIT.

- MERGE(p, q, z): Recebe dois nós p e q de mesmo peso e une os vértices u e v que possuem p e q em suas ordens cíclicas. Se α e β são respectivamente as ordens cíclicas de u e v, então $\alpha\beta$ será a ordem cíclica do vértice resultante z. Essa operação assume que u e v estão em ADPs distintas.
- SPLIT(p, v, z): Substitui o vértice u que contém p em sua ordem cíclica por dois vértices v e z. Se a ordem cíclica $D(u)$ tem forma $\alpha p \beta$, então α e $p\beta$ serão as ordens cíclicas de v e z, respectivamente.

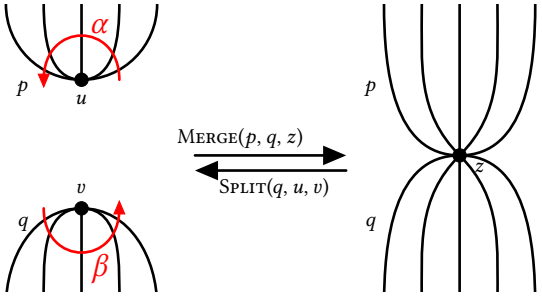


Figura 5.7: *Efeito de MERGE e SPLIT.*

A implementação dessas rotinas será apresentada na Seção 5.8. O consumo de tempo de CREATE OCTO e DESTROY OCTO é O(1) e o consumo das restantes rotinas é O(lg m) esperado amortizado, onde m é o número de arestas no grafo corrente.

5.6 Resolvendo MSF com ADPs

5.6.1 Criação de grafo plano ponderado dinâmico

Um grafo plano ponderado dinâmico G será implementado como a tabela de símbolos H descrita no final da Seção 5.4 e um inteiro p que representa o peso da floresta geradora de pesos mínimos F. Como G é inicializado vazio, então p é igual a 0. A implementação de NOVOGDP é descrita no Algoritmo 17 e seu consumo de tempo é O(1).

Algoritmo 17 NOVOGDP(n)

```
1 G.H ← NOVODICIO(n)
2 G.p ← 0
3 retorne G
```

5.6.2 Obtenção de peso

Para obter o peso da floresta geradora de pesos mínimos F, simplesmente retornamos o valor p, como pode ser visto na implementação de PESOGDP no Algoritmo 18. Seu consumo é O(1).

Algoritmo 18 PESOGDP(G)

```
1 retorne G.p
```

5.6.3 Mudança de peso

A primeira rotina de MSF que vamos apresentar é MUDAPESOGDP, cuja implementação pode ser vista no Algoritmo 19. Essa é a primeira rotina apresentada, pois não modifica a estrutura do grafo. Encorajamos a leitora ou o leitor a acompanhar a explicação do funcionamento dessa rotina antes de ler o pseudocódigo linha-a-linha. Essa explicação é feita nos próximos parágrafos e detalha o embasamento teórico utilizado pelo algoritmo, seu funcionamento e ilustra a execução de MUDAPESOGDP(G, a, 5) como exemplo, onde G é o grafo da Figura 5.5.

Dado um grafo G, um **corte** é um conjunto de arestas cuja remoção aumenta o número de componentes conexas de G. Seja F uma floresta maximal de G e uv uma aresta de F. Então existe um corte associado ao par (F, uv) dado pela união de uv com as arestas de G que reconectam as duas árvores de F geradas pela remoção de uv. Por exemplo, o corte (F, a) é o conjunto de arestas $\{a,\ b,\ c\}$.

Algoritmo 19 MUDAPESOGDP(G, e, w)

```
1  e_0, e_1, e_2, e_3, v_0, v_1, v_2, v_3 ← G.H(e)
2  para i ∈ {0, 1, 2, 3} faça
3      SET WEIGHT(e_i, w)
4  se FIND NODE(e_0) = FIND NODE(e_2) então                    ▷ e ∈ F
5      d ← FIND MIN(e_1, e_3)
6      d_0, d_1, d_2, d_3, y_0, y_1, y_2, y_3 ← G.H(d)
7      se d_0.w < w então
8          G.p ← G.p − e_0.w + d_0.w
9          CYCLE(d_3); SPLIT(d_3, d̂_1, d̂_3)
10         CYCLE(e_2); SPLIT(e_2, ê_0, ê_2 )
11         MERGE(d_0, d_2, d̂); MERGE(e_1, e_3, ê)
12 senão                                                       ▷ e* ∈ F*
13     d ← FIND MAX(e_0, e_2)
14     d_0, d_1, d_2, d_3, y_0, y_1, y_2, y_3 ← G.H(d)
15     se d_0.w > w então
16         G.p ← G.p − e_0.w + d_0.w
17         CYCLE(d_0); SPLIT(d_0, d̂_2, d̂_0)
18         CYCLE(e_1); SPLIT(e_1, ê_3, ê_1)
19         MERGE(d_1, d_3, d̂); MERGE(e_0, e_2, ê)
```

Teorema 4 (Proposição 4.6.1 [9]). *Seja G um grafo plano, F uma floresta maximal de G e uv uma aresta de F. Então o conjunto*

$$(F, uv)^\star := \{e^\star : e \in (F, uv)\}$$

forma um ciclo em $G^\star$.

Quando atualizamos o peso de uma aresta e de G para um novo peso w, precisamos eventualmente atualizar a floresta F de peso mínimo que está sendo mantida de forma que ela continue sendo maximal e de peso mínimo. Existem dois casos a serem tratados: quando e é uma aresta de F e quando $e^\star$ é uma aresta de $F^\star$.

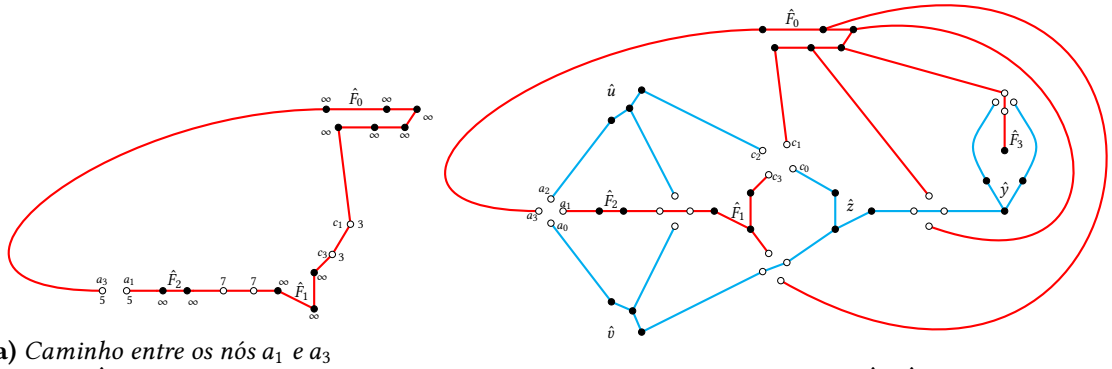
No Algoritmo 19, essa distinção de casos é feita na linha 4. Se FIND NODE(e_0) e FIND NODE(e_2) retornarem o mesmo identificador $\hat{e}$, então deduzimos que e é uma aresta de F. Caso contrário, temos que $e^\star$ é uma aresta de $F^\star$. Por exemplo, observando a Figura 5.4, notamos que FIND NODE(a_0) e FIND NODE(a_2) retornam o mesmo identificador $\hat{a}$, enquanto que FIND NODE(c_0) e FIND NODE(c_2) retornam os identificadores distintos $\hat{c}_0$ e $\hat{c}_2$.

O primeiro caso é tratado entre as linhas 5 e 11 do Algoritmo 19. Nele e é uma aresta de F e é necessário verificar se não há alguma aresta no corte (F, e) com peso menor do que novo peso w, pois se tal aresta existir, então F e $F^\star$ não serão mais de peso mínimo em G e máximo em $G^\star$, respectivamente. Nesse caso, para corrigir isso, é necessário remover a aresta e de F e adicionar em F uma aresta d de menor peso do corte (F, e). Simetricamente, também é necessário remover $d^\star$ de $F^\star$ e adicionar $e^\star$ nessa floresta.

Pelo Teorema 4, o conjunto $(F, e)^\star$ forma um ciclo em $G^\star$. Como e é a única aresta

de F cuja aresta dual $e^\star$ está em $(F, e)^\star$, então as demais arestas desse corte formam um caminho P em $F^\star$ ligando as faces incidentes a $e^\star$. Dessa forma, para obter d, basta obter o mínimo no caminho P.

As faces incidentes a $e^\star$ são acessíveis nas ADPs pelos nós e_1 e e_3. Assim, para obter d, é executado FIND MIN(e_1, e_3), que é feito na linha 5 do Algoritmo 19. A Figura 5.8a ilustra o caminho P para o corte (F, a). No exemplo, FIND MIN(a_1, a_3) retorna o identificado $\hat{c}$.



(a) *Caminho entre os nós a_1 e a_3 na árvore $\hat{T}^\star$ da Figura 5.5.*

(b) *ADPs com as arestas a e $c^\star$ cortadas de $\hat{T}$ e $\hat{T}^\star$, respectivamente.*

Figura 5.8: *Etapas da mudança de peso* MUDAPESOGDP(G, a, 5).

Em seguida, na linha 7 do Algoritmo 19, o novo peso w é comparado com o peso do vértice d. Caso w seja menor do que o peso de d, nada precisa ser feito. Caso contrário, na linha 8 do Algoritmo 19, o peso da floresta geradora de pesos mínimos é atualizada. Em seguida, na linha 10, as arestas e e $d^\star$ são removidas de F e de $F^\star$ utilizando a rotina SPLIT, que converte os nós $\hat{e}$ e $\hat{d}$ nos nós $\hat{e}_0$, $\hat{e}_2$, $\hat{d}_1$ e $\hat{d}_3$, que são as novas folhas associadas às arestas e e d. Para adicionar as arestas $e^\star$ a $F^\star$ e d a F são usadas as folhas $\hat{d}_0$, $\hat{d}_2$, $\hat{e}_1$ e $\hat{e}_3$ junto à rotina MERGE na linha 11 do Algoritmo 19.

Concluindo o exemplo, o peso de $\hat{c}$ é 3 que é menor do que o novo peso 5 de $\hat{a}$. A Figura 5.8b ilustra esse exemplo logo após a linha 10 e antes da linha 11 do Algoritmo 19. Note que há duas árvores azuis e duas vermelhas. A Figura 5.9 ilustra esse exemplo após a conclusão da chamada MUDAPESOGDP(G, a, 5).

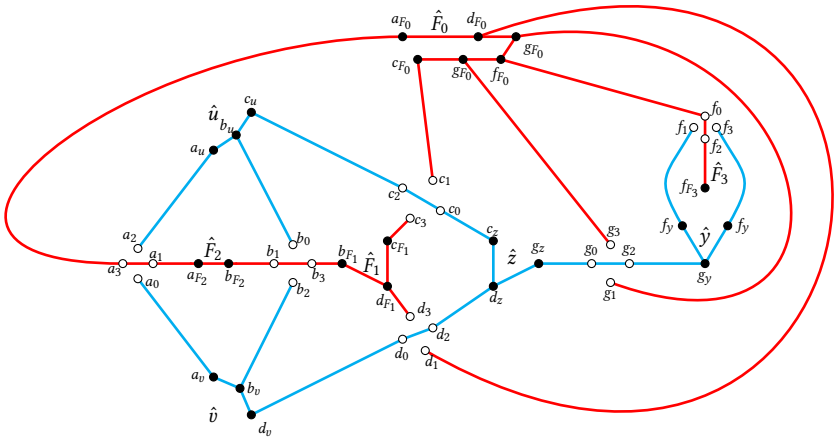


Figura 5.9: *ADPs após a execução de* MUDAPESOGDP(G, a, 5).

O segundo caso, quando e não for uma aresta de F, então $e^\star$ é uma aresta de $F^\star$. O tratamento desse caso é simétrico ao primeiro caso e é feito entre as linhas 13 e 19 do Algoritmo 19.

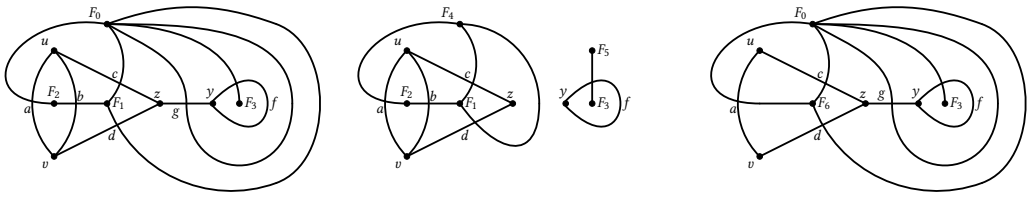
O custo de tempo assintótico de mudaPesoGDP é $O(\lg m)$ esperado amortizado, onde m é o número de arestas do grafo G corrente. As linhas 1, 6, 8, 14 e 16 do Algoritmo 19 possuem consumo de tempo constante amortizado, pois são consultas à tabela de símbolos ou operações aritméticas básicas. As linhas remanescentes são formadas por um número constante de chamadas de Set weight, Find node, Merge e Split e cada uma dessas chamadas possui consumo de tempo assintótico $O(\lg m)$ esperado amortizado, concluindo que mudaPesoGDP possui consumo de tempo assintótico $O(\lg m)$ esperado amortizado.

5.6.4 Remoção de aresta

Ao remover uma aresta e de um grafo plano ponderado dinâmico G, além de remover e de $\hat{F}$ ou $e^\star$ de $\hat{F}^\star$ é necessário também atualizar a estrutura de $\hat{F}^\star$, pois a remoção de e modifica a estrutura de faces do grafo plano. Pelo Lema 1, existem dois tipos de arestas em G: ou a aresta não está em ciclos em G e as duas faces incidentes a ela são iguais, ou essas faces são distintas e a aresta está em pelo menos um ciclo de G.

No primeiro caso, a aresta e é uma ponte e sua remoção aumenta o número de componentes conexas de G. Como consideramos que cada uma dessas componentes possui uma face exterior própria, a remoção de e implica na divisão da face exterior em duas, que serão as faces exteriores das duas componentes resultantes. Como exemplo, ao remover a aresta e do grafo G ilustrado na Figura 5.10a a face exterior F_0 é separada nas faces F_4 e F_5 da Figura 5.10b.

Já no segundo caso, a remoção de uma aresta e implica na junção das duas faces incidentes a essa aresta. No exemplo, a remoção da aresta b do grafo ilustrado na Figura 5.10a resulta na junção das faces F_1 e F_2 resultando na face F_6 ilustrada na Figura 5.10c.



(a) *Grafo da Figura 5.2.* **(b)** *Após a remoção de g.* **(c)** *Após remoção de b.*

Figura 5.10: *Exemplos de remoção de arestas.*

A implementação de removaGDP pode ser vista no Algoritmo 20. Junto com a explicação do pseudocódigo, detalharemos o processamento das chamadas removaGDP(G, g) e removaGDP(G, b) para exemplificar os dois casos de remoção de aresta.

Antes de identificar em qual caso a aresta e está, sua óctupla é obtida da tabela de símbolos e o identificado de cada nó dessa óctupla é obtido usando a rotina Find node no laço da linha 2 do Algoritmo 20.

A identificação de qual caso uma aresta e se encontra é feita na linha 5 do Algoritmos 20. Nessa linha são comparados os valores de FIND NODE(v_1) e FIND NODE(v_3). Esses valores são os identificadores das faces adjacentes a e. Caso esses identificadores sejam iguais, então as faces incidentes a e são iguais e logo e não está em ciclos de G. Caso contrário, então e está em pelo menos um ciclo. A aresta g se encontra no primeiro caso, pois ambas as chamadas de FIND NODE(g_{F_0}) retornam o identificador $\hat{F}_0$. Enquanto que b se encontra no segundo caso, já que FIND NODE(b_u) retorna $\hat{u}$, enquanto que FIND NODE(b_v) retorna $\hat{v}$.

O primeiro caso é tratado entre as linhas 6 e 10 do Algoritmo 20. Remover uma aresta e das ADPs consiste em desconectar a óctupla de e do restante da ADP. Para desconectar um nó v_i de sua ordem cíclica são chamadas as rotinas CYCLE(v_i) e SPLIT(v_i, $\hat{v}_1$, NIL) em sucessão. A rotina CYCLE(v_i) garante que a ordem cíclica que contém v_i tenha forma αv_i, enquanto que a rotina SPLIT(v_i, $\hat{v}_i$, NIL) separa essa ordem em duas: a primeira contendo todos os nós anteriores a v_i e a segunda com v_i e todos os nós sucessores, mas como a ordem tem forma αv_i, então as duas ordens resultantes são α e v_i. Ou seja, o nó v_i foi desconectado do restante da ordem cíclica. Essa rotina também atribui o identificador $\hat{v}_i$ à primeira ordem, isto é, o antigo identificador do vértice que continha v_i em sua ordem cíclica. O terceiro parâmetro da rotina SPLIT é o identificador da segunda ordem cíclica obtida por essa rotina, nesse caso a ordem é composta somente pelo nó v_i. Como essa ordem será apagada da memória, esse novo identificador atribuído por SPLIT não é relevante para a operação da rotina REMOVAGDP e para simbolizar isso no pseudocódigo, optamos por passar NIL como parâmetro nesse caso. Esse processo é feito para v_0, v_1 e v_2 nas linhas 6 a 8. Relembramos que a óctupla de g e b estão ilustradas na Figura 5.6.

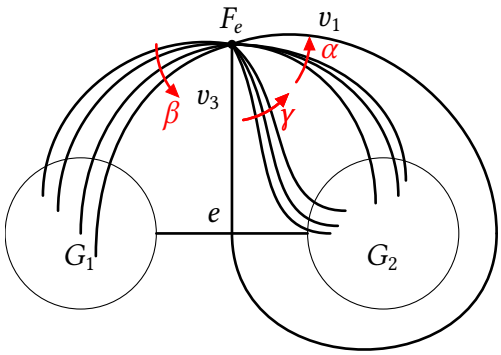
Algoritmo 20 REMOVAGDP(G, e)

```
1  e_0, e_1, e_2, e_3, v_0, v_1, v_2, v_3 ← G.H(e)
2  para i ∈ {0, 1, 2, 3} faça
3      ê_i ← FIND NODE(e_i)
4      v̂_i ← FIND NODE(v_i)
5  se v̂_1 = v̂_3 então                                  ▷ e* é um laço de G*
6      CYCLE(v_0); SPLIT(v_0, v̂_0, NIL)
7      CYCLE(v_1); SPLIT(v_1, v̂_1, NIL)
8      CYCLE(v_2); SPLIT(v_2, v̂_2, NIL)
9      SPLIT(v_3, F_0, F_1)
10     CYCLE(v_3); SPLIT(v_3, v̂_3, NIL)
11 senão                                                ▷ e está em um ciclo em G
12     MUDAPESOGDP(G, e, ∞)
13     CYCLE(e_1); SPLIT(e_1, ê_3, ê_1)
14     CYCLE(v_1); CYCLE(v_3); MERGE(v_1, v_3, F_0)
15     CYCLE(v_0); SPLIT(v_0, v̂_0, NIL)
16     CYCLE(v_1); SPLIT(v_1, F_0, NIL)
17     CYCLE(v_2); SPLIT(v_2, v̂_2, NIL)
18     CYCLE(v_3); SPLIT(v_3, F_0, NIL)
19 DESTROY OCTO(G.H, e)
```

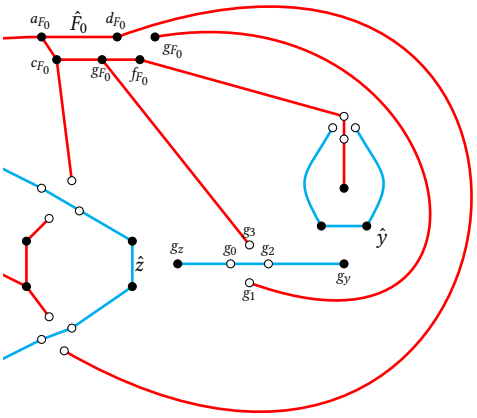
Antes de desconectar v_3 de sua ordem cíclica, a face exterior F_e que contém v_1 e v_3 em sua ordem cíclica é dividida nas duas faces exteriores das duas componentes conexas G_1 e G_2 obtidas pela remoção de e de G. As duas pontas do laço $e^\star$, representadas por v_1 e v_3 nas ADPs, dividem a ordem cíclica de F_e em duas partes. A primeira contendo todos os nós entre v_1 e v_3 e a segunda possuindo todos os nós de v_3 a v_1 seguindo a ordem. Isto é, se a ordem de F_e tem forma $\alpha v_1 \beta v_3 \gamma$, então essas duas partes têm forma $\gamma\alpha$ e β. Essas duas partes são as ordens cíclicas das futuras faces exteriores de G_1 e G_2, como ilustra a Figura 5.11a.

Após a desconexão de v_1 com as chamadas Cycle(v_1) e Split(v_1, $\hat{v}_1$, NIL), feitas na linha 7, a ordem cíclica da face $\hat{F}_e$ resultante dessas operações tem forma $\beta v_3 \gamma \alpha$. Então, para separar β de $\gamma\alpha$, é executado um Split em v_3 na linha 9, obtendo dois vértices F_0 e F_1 com ordens cíclicas β e $v_3\gamma\alpha$. É importante que F_0 e F_1 sejam identificadores únicos para as novas faces criadas, pois esses identificadores são usados na linha 5 do Algoritmos 20. Em seguida, v_3 é desconectado da segunda ordem cíclica na linha 10. Por fim, na linha 19 a óctupla de e é desalocada da memória.

Na Figura 5.11b, podemos ver as ADPs logo antes da linha 9 do Algoritmo 20. Após desconectar o nó g_{F_0} que está ligado a g_1, a ordem cíclica da face exterior $\hat{F}_0$ é $\langle d_{F_0}, a_{F_0}, c_{F_0}, g_{F_0}, f_{F_0}\rangle$. Logo, nesse caso, temos que $\gamma\alpha$ e β são respectivamente $\langle f_{F_0}\rangle$ e $\langle d_{F_0}, a_{F_0}, c_{F_0}\rangle$. Essas são as duas ordens cíclicas das faces exteriores F_5 e F_4 ilustradas na Figura 5.10b.



(a) *Ordens cíclicas da face exterior F_e.*

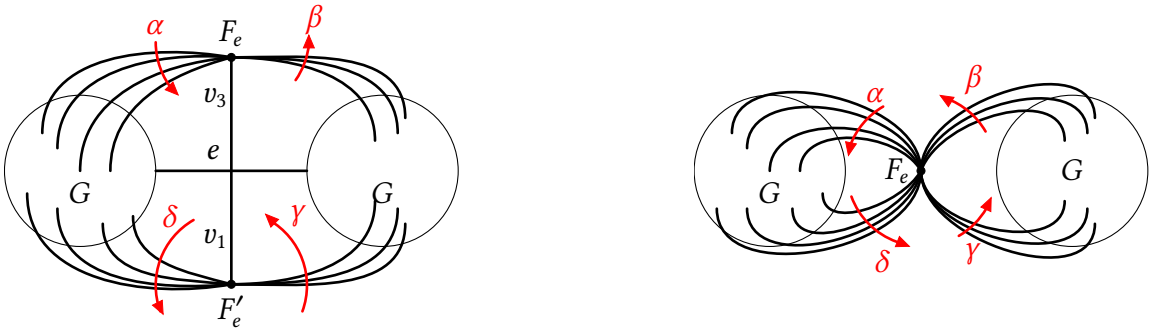


(b) *ADPs logo antes da linha 9 do Algoritmo 20.*

Figura 5.11: *Ordens cíclicas de faces exteriores. Figura 5.11a representa o caso genérico. Figura 5.11b ilustra as ADPS da Figura 5.5 ao remover a aresta e.*

O segundo caso é tratado entre as linhas 12 e 18 do Algoritmo 20. Nesse caso, a remoção de e não desconecta sua componente conexa. Dessa forma, caso e seja uma aresta de F, é necessário buscar uma aresta que a substituirá em F, assim mantendo F como maximal. Para fazer essa busca eficientemente, na linha 12 do Algoritmo 20, o peso de e é mudado para ∞, fazendo com que $e^\star$ se torne uma aresta de $F^\star$. Em nosso exemplo, note que mudaPesoGDP(G, b, ∞) não altera as estruturas das ADPs, já que $b^\star$ já é uma aresta de $\hat{T}^\star$.

Em seguida, na linha 13 do Algoritmo 20, o vértice $\hat{e}^\star$ é dividido em dois vértices $\hat{e}_1$ e $\hat{e}_3$. Essa operação quebra a ADP $\hat{T}^\star$ em duas, assim tornando possível juntar as faces F_e e F'_e incidentes a e sem criar ciclos em $\hat{T}^\star$. A junção dessas faces ocorre na linha 14 do Algoritmo 20 e é feita de forma que a descrição combinatória resultante se mantenha plana, isto é, se $\alpha v_1 \beta$ e $\gamma v_3 \delta$ são as ordens cíclicas de $\hat{F}_e$ e $\hat{F}'_e$, respectivamente, então $\alpha\delta\gamma\beta$ é a ordem cíclica da face resultante. Como ilustra a Figura 5.12a. Em mais detalhes, as chamadas Cycle(v_1) e Cycle(v_3) garantem que as ordens cíclicas de F'_e e F_e tenham forma $\delta\gamma v_1$ e $\beta\alpha v_3$. Em seguida, a chamada Merge(v_1, v_3, F_0) junta as faces F'_e e F_e criando uma nova face F_0 com ordem cíclica $\delta\gamma v_1\beta\alpha v_3$. Entre as linhas 15 e 18, cada nó v_i é desconectado de sua ordem cíclica com um par de chamadas Cycle(v_i) e Split(v_i, $\hat{v}_i$, NIL) como feito no caso anterior. Em particular, os nós v_1 e v_3 são desconectado da ordem cíclica $\delta\gamma v_1\beta\alpha v_3$ obtendo a ordem cíclica $\delta\gamma\beta\alpha$ como desejado. Por fim, a óctupla é desalocada da memória.



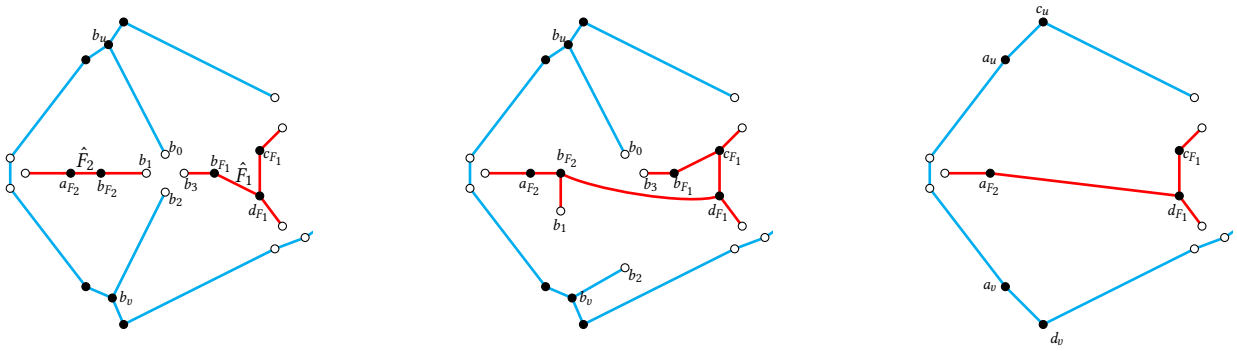
(a) *Antes de remover e.*

(b) *Depois de remover e.*

Figura 5.12: *Ordens cíclicas das faces incidentes à aresta e antes e depois de sua remoção.*

Na simulação de removaGDP(G, b), após cortar o vértice $\hat{b}$ em dois, obtém-se duas ADPs, uma contendo somente o vértice $\hat{F}_2$ e as duas folhas $\hat{b}_1$ e $\hat{a}_1$, como pode ser visto na Figura 5.13a. Na linha 14 do Algoritmo 20 as ordens cíclicas de $\hat{F}_1$ e $\hat{F}_2$ são unidas. Primeiro, Cycle(b_{F_2}) e Cycle(b_{F_1}) rearranjam as ordens de $\hat{F}_2$ e $\hat{F}_1$ para terem forma $\langle a_{F_2}, b_{F_2}\rangle$ e $\langle d_{F_1}, c_{F_1}, b_{F_1}\rangle$, respectivamente. Em seguida, esses vértices são juntados com Merge(b_{F_2}, b_{F_1}), obtendo o vértice com ordem $\langle a_{F_2}, b_{F_2}, d_{F_1}, c_{F_1}, b_{F_1}\rangle$, como ilustrado na Figura 5.13b. Por fim, a óctupla de b é desconectada das ordens cíclicas e desalocada da memória. A ordem da face resultante é $\langle a_{F_2}, d_{F_1}, c_{F_1}\rangle$, como ilustrado na Figura 19. Note que esse vértice de ADP representa a face F_6 da Figura 5.10c.

As linhas 1 e 19 do Algoritmo 20 possuem consumo de tempo constante, pois são uma consulta à tabela de símbolos $G.H$ e uma chamada da rotina Destroy Octo. As linhas remanescentes são formadas por um número constante de chamadas de Find node, Cycle, Merge, Split e mudaPesoGDP e cada uma dessas chamadas possui consumo de tempo assintótico igual a $O(\lg m)$ esperado, onde m é o número de arestas no grafo corrente. Dessa forma, removaGDP também possui consumo de tempo assintótico $O(\lg m)$ esperado.



(a) *Após linha 13.* **(b)** *Após linha 14.* **(c)** *Após linha 19.*

Figura 5.13: *Diferentes momentos da execução de* REMOVAGDP(G, b).

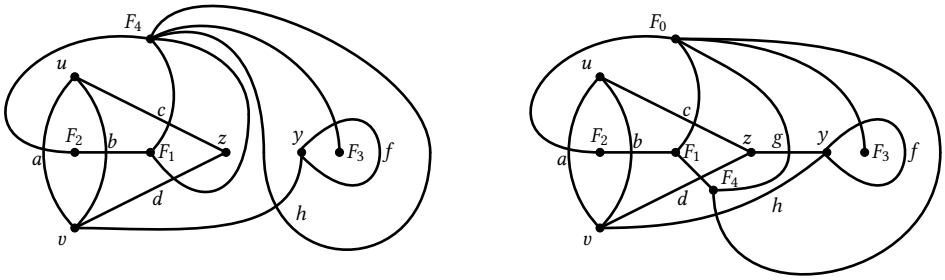
5.6.5 Adição de aresta

A última rotina de MSF elaborada é LIGUEGDP, cuja implementação é descrita no Algoritmo 21. Como em REMOVAGDP, existem dois casos a serem tratados: O primeiro é quando a nova aresta inserida no grafo plano ponderado dinâmico G liga duas componentes conexas distintas, isto é, a nova aresta é uma ponte. Nesse caso, as faces exteriores das duas componentes conexas são unidas em uma única face exterior. O segundo caso é quando a nova aresta liga dois vértices de uma mesma componente conexa. Nesse caso, a rotina LIGUEGDP assume que as arestas f e g passadas como parâmetros compartilham uma face F_{fg}, pois a inserção de uma aresta que não obedece essa hipótese não estaria de acordo com a imersão corrente do grafo G. Nesse caso a face F_{fg} é dividida em duas faces e será elaborado adiante como F_{fg} é encontrada.

Algoritmo 21 LIGUEGDP(G, e, u, f, v, g, w)

```
1  f_0, f_1, f_2, f_3, u_0, u_1, u_2, u_3 ← G.H(f)
2  g_0, g_1, g_2, g_3, v_0, v_1, v_2, v_3 ← G.H(g)
3  se FIND NODE(u_0) = u então i ← 0 senão i ← 2
4  se FIND NODE(v_0) = v então j ← 0 senão j ← 2
5  G.H(e) ← e_0, e_1, e_2, e_3, s_0, s_1, s_2, s_3 ← CREATE OCTO(e, w)
6  CYCLE(u_i); MERGE(u_i, s_0, u)
7  CYCLE(v_j); MERGE(v_j, s_2, v)
8  CYCLE(PRED(u_{i-1})); MERGE(u_{i-1}, s_3, FIND NODE(u_{i-1}))
9  CYCLE(PRED(v_{j-1})); MERGE(v_{j-1}, s_1, FIND NODE(v_{j-1}))
10 se CONNECTED(u, v) então
11     SPLIT(u_{i-1}, F_u, F'_u)
12     MUDAPESOGDP(G, e, w)
13 senão
14     MERGE(u_{i+1}, v_{j+1}, F_uv)
15     MERGE(e_0, e_2, ê)
```

Como exemplos, vamos inserir uma nova aresta h ligando os vértices v e y em dois grafos. O primeiro grafo está ilustrado na Figura 5.10b e possui duas componentes que são ligadas por h. A nova aresta é inserida após as arestas a e f nas ordens cíclicas de v e y, respectivamente, logo a chamada para essa rotina é LIGUEGDP(G, h, v, a, y, f, 7). O segundo grafo é o da Figura 5.2 e h é inserida após as aresta a e g nas ordens cíclicas, logo a chamada feita nesse exemplo é LIGUEGDP(G, h, v, a, y, g, 7). A face comum que será dividida em duas é F_0. A Figura 5.14 ilustra os grafos obtidos por essas inserções.



(a) *Após a adição da ponte h.* **(b)** *Após a adição de h.*

Figura 5.14: *Exemplos de adição de arestas.*

Note que adicionar uma aresta e a G como sucessora de f e g implica adicionar $e^\star$ a $G^\star$ como predecessora de $f^\star$ e $g^\star$ na ordem cíclica das faces apropriadas. Na Figura 5.14b é possível ver que $h^\star$ é predecessora de $a^\star$ e $g^\star$ na ordem cíclica de F_0 e F_4, respectivamente.

O processamento da rotina LIGUEGDP começa obtendo as óctuplas de f e g e comparando o vértice u com o identificador retornado por FIND NODE(u_0). Esse teste é feito para descobrir qual nó, u_0 ou u_2, pertence à ordem cíclica de u. O índice (0 ou 2) é salvo em uma variável i para ser usado futuramente. Esse mesmo processo é feito com v. Então é criado uma nova óctupla com CREATE OCTO, que é guardada na tabela de símbolos H com chave associada e.

As linhas 6 e 7 executam CYCLE e MERGE para juntar os nós s_0 e s_2 às ordens cíclicas dos vértices u e v de $\hat{T}$. Ao executar CYCLE(u_i), a ordem cíclica de u obtém forma αu_i, assim a operação MERGE(u_i, s_0, u) resulta na ordem cíclica $\alpha u_i s_0$. Analogamente, CYCLE(v_j) seguido de MERGE(v_j, s_2, v) resulta na ordem cíclica $\beta v_j s_2$ para o vértice v. Isto é, a nova aresta é inserida após f e g nas ordens cíclicas.

As linhas 8 e 9 fazem o mesmo para os nós s_1 e s_3 nas ordens cíclicas dos vértices de $\hat{T}^\star$. Note que, como a nova aresta e é inserida após f e g em suas ordens cíclicas, então $e^\star$ é inserida nas faces obtidas por FIND NODE(u_{i-1}) e FIND NODE(v_{j-1}). A única diferença é que esses nós são inseridos como predecessores de u_{i-1} e v_{j-1}. Para tal, a rotina CYCLE é chamada para o predecessor de u_{i-1} obtido com a rotina PRED.

Em seguida, o algoritmo acerta a estrutura da face com base nos dois casos comentados anteriormente. A detecção de qual caso a aresta e se encontra é feita na linha 10 com um teste de conexidade entre os vértices u e v. Se os dois vértices estiverem conectados, então é necessário aplicar SPLIT para dividir a face FIND NODE(u_{i-1}) em duas e o peso de e é atualizado com MUDAPESOGDP(G, e, w), que também determina se a nova aresta e será

uma aresta de $\hat{T}$ ou de $\hat{T}^\star$. Caso contrário, é necessário aplicar um MERGE para juntar as faces exteriores das duas componentes conexas e como a aresta é uma ponte, então necessariamente $\hat{e}$ é um vértice de $\hat{T}$, logo é feito um MERGE(e_0, e_2, $\hat{e}$).

Simularemos primeiro a inserção da aresta h no caso em que ela é uma ponte, isto é, a chamada LIGUEGDP(G, h, v, a, y, f, 7) comentada anteriormente. A Figura 5.15 ilustra as ADPs das duas componentes conexas do grafo da Figura 5.10b.

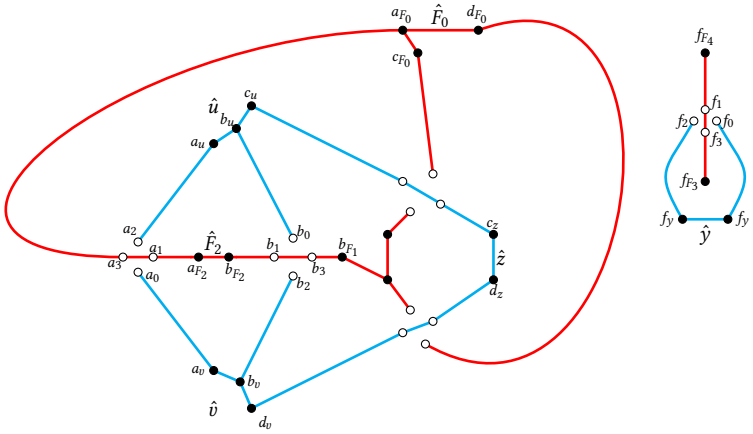


Figura 5.15: *ADPs antes da adição de h.*

Após obter as óctuplas de a e f, o algoritmo compara v com o identificador retornado por FIND NODE(a_v), que também é v e assim é atribuído o valor 0 à variável i. O algoritmo faz o mesmo processo com y e atribui 0 à j. Em seguida é criada uma nova óctupla associada a h e os nós s_i são adicionados nas respectivas ordens cíclicas.

Primeiro é feito CYCLE(a_v) e MERGE(a_v, h_0, v). A rotina CYCLE(a_v) não modifica a estrutura dessa ordem cíclica, já que o nó a_v já é o último nó de sua ordem cíclica. Por sua vez, a rotina MERGE(a_v, s_0, v) resulta na ordem cíclica de nós $\langle d_v, b_v, a_v, s_0\rangle$.

Depois é feito CYCLE(f_y) e MERGE(f_y, h_y, y). A rotina CYCLE(f_y) não modifica a estrutura dessa ordem cíclica, já que o nó f_y já é o último nó de sua ordem cíclica. A rotina Merge(f_y, h_y, y) por sua vez resulta na ordem cíclica de nós $\langle f_y, f_y, h_y\rangle$.

Como $i = 0$, temos que $i - 1 = 3$ módulo 4. Assim o nó u_{i-1} em que é calculado o predecessor na linha 8 do Algoritmo 21 é o nó a_{F_0}, pois esse está ligado ao nó a_3. O predecessor de a_{F_0} em sua ordem cíclica é d_{F_0}. Assim CYCLE(PRED(a_{F_0})) reestrutura a ordem cíclica de $\hat{F}_0$ para ser $\langle a_{F_0}, c_{F_0}, d_{F_0}\rangle$ e MERGE(a_{F_0}, h_{F_0}, F_0) resulta na ordem $\langle a_{F_0}, c_{F_0}, d_{F_0}, h_{F_0}\rangle$.

Como $j = 0$, temos que $j - 1 = 3$ módulo 4. Assim o nó v_{i-1} em que é calculado o predecessor na linha 9 do Algoritmo 21 é o nó f_{F_3}, pois esse está ligado ao nó f_3. O nó f_{F_3} é o próprio predecessor. Assim CYCLE(PRED(f_{F_3})) não altera a estrutura da ordem cíclica de $\hat{F}_3$ e MERGE(f_{F_3}, h_{F_3}, F_3) resulta na ordem $\langle f_{F_3}, h_{F_3}\rangle$. A Figura 5.16 ilustra as ADPs após esse bloco de execuções de CYCLE e MERGE.

Em seguida é feito um teste de conexidade entre v e y que retorna falso, assim identificando que a nova aresta é uma ponte. Assim é feito um Merge(a_{F_0}, f_{F_3}, F_0), correspondente à linha 14 do Algoritmo 21, obtendo assim a ordem cíclica $\langle a_{F_0}, c_{F_0}, d_{F_0}, h_{F_0}, f_{F_4}, h_{F_4}\rangle$ corres-

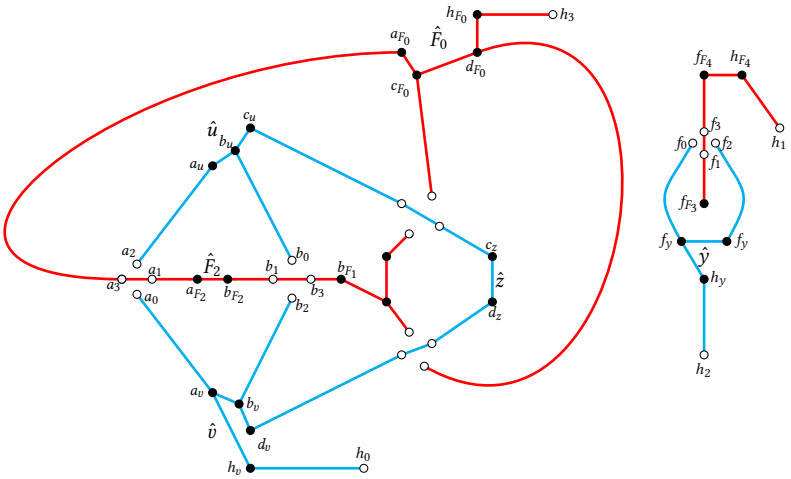


Figura 5.16: *ADPs após adição da óctupla de h.*

pondente à nova face exterior do grafo. Por fim, é feito Merge(h_0, h_2, $\hat{h}$). A Figura 5.17 ilustra o grafo resultante dessa rotina.

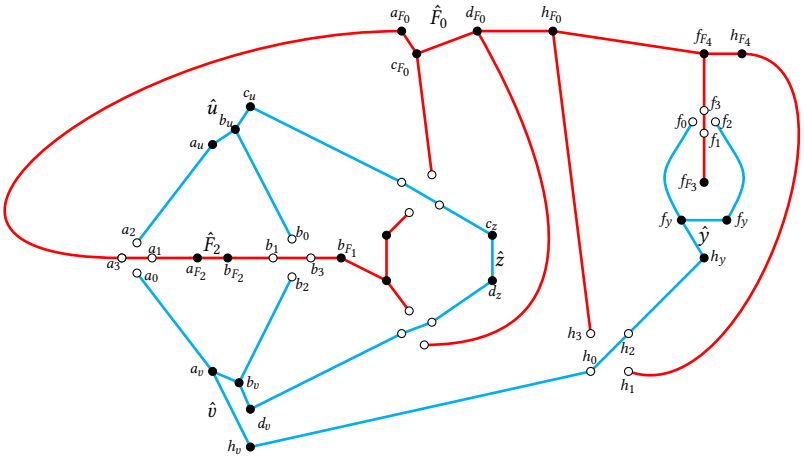


Figura 5.17: *ADPs obtidas por* ligueGDP*(G, h, v, a, y, f, 7).*

Agora a chamada ligueGDP(G, h, v, a, y, g, 7) será simulada. O processo é análogo ao caso anterior até a linha 10 do Algoritmo 21. Os vértices v e y são comparados com Find node(a_v) e Find node(g_y) e são atribuídos 0 à variável i e 2 a j. Uma nova óctupla é criada e associada a h e os nós s_i são adicionados nas respectivas ordens cíclicas. As chamadas Cycle(a_v) e Merge(a_v, h_0, $\hat{v}$), correspondentes à linha 6 do Algoritmo 21, reestruturam a ordem cíclica de v para ser $\langle d_v, b_v, a_v, h_v\rangle$. Já as chamadas Cycle($g_y$) e Merge($g_y$, h_y, $\hat{y}$), correspondentes à linha 7 da mesma rotina, reestruturam a ordem cíclica de y para ser $\langle f_y, f_y, g_y, h_y\rangle$.

As linhas 8 e 9 fazem o mesmo para os nós s_1 e s_3 nas ordens cíclicas dos vértices de $\hat{F}^\star$. Como $i = 0$, temos que $i - 1 = 3$ módulo 4. Assim o nó u_{i-1} em que é calculado o predecessor na linha 8 do Algoritmo 21 é o nó a_{F_0}, pois esse está ligado ao nó a_3. O predecessor de a_{F_0} em sua ordem cíclica é d_{F_0}. A ordem cíclica de F_0 antes da execução dessa

rotina é $\langle c_{F_0}, g_{F_0}, f_{F_0}, g_{F_0}, d_{F_0}, a_{F_0}\rangle$. Assim Cycle(Pred($a_{F_0}$)) reestrutura a ordem cíclica de $\hat{F}_0$ para ser $\langle a_{F_0}, c_{F_0}, d_{F_0}\rangle$ e Merge(a_{F_0}, h_{F_0}) resulta na ordem $\langle a_{F_0}, c_{F_0}, d_{F_0}, h_{F_0}\rangle$. Em seguida, como $j = 2$, temos que $j - 1 = 1$ módulo 4. Assim o nó v_{i-1} em que é calculado o predecessor na linha 9 do Algoritmo 21 é o nó g_{F_0}, pois esse está ligado ao nó g_1. O predecessor de g_{F_0} é f_{F_0}, assim Cycle(Pred(a_{F_0})) reestrutura a ordem cíclica de $\hat{F}_0$ para ser $\langle g_{F_0}, d_{F_0}, h_{F_0}, a_{F_0}, c_{F_0}, g_{F_0}, f_{F_0}\rangle$ e Merge(a_{F_0}, h_{F_0}) resulta na ordem $\langle g_{F_0}, d_{F_0}, h_{F_0}, a_{F_0}, c_{F_0}, g_{F_0}, f_{F_0}, h_{F_0}\rangle$. A Figura 5.18 ilustra as ADPs após essas modificações.

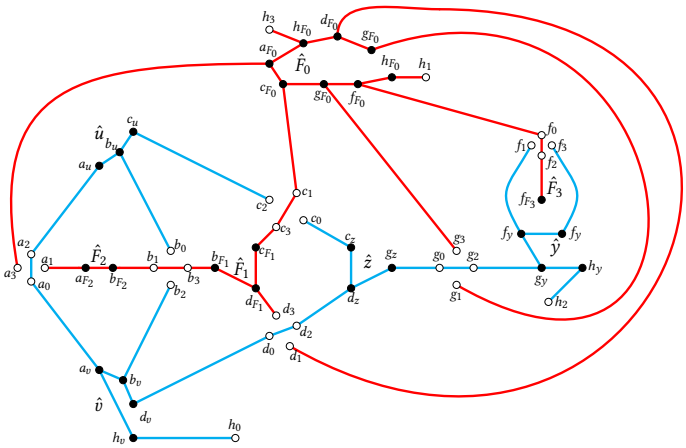


Figura 5.18: *ADPs após adição da óctupla de h.*

Após essas modificações, é feita a consulta de conexidade para verificar em qual caso a aresta h se encontra. Nesse caso esse teste retorna positivo e assim constata-se que h está no segundo caso. Assim é executado Split(a_{F_0}, F_4, F_0), correspondente à linha 11 do Algoritmo 21, que corta a ordem cíclica em duas,a primeira sendo $\langle g_{F_0}, d_{F_0}, h_{F_0}\rangle$ e a segunda $\langle a_{F_0}, c_{F_0}, g_{F_0}, f_{F_0}, h_{F_0}\rangle$. A primeira ordem cíclica corresponde à face F_4 da Figura 5.19, enquanto que a segunda se mantém sendo a face exterior F_0. Por fim, é chamado mudaPesoGDP(G, h, 7) que atesta que essa aresta pertence a $\hat{F}^\star$ e internamente executa Merge($h_1, h_3, \hat{h}$).

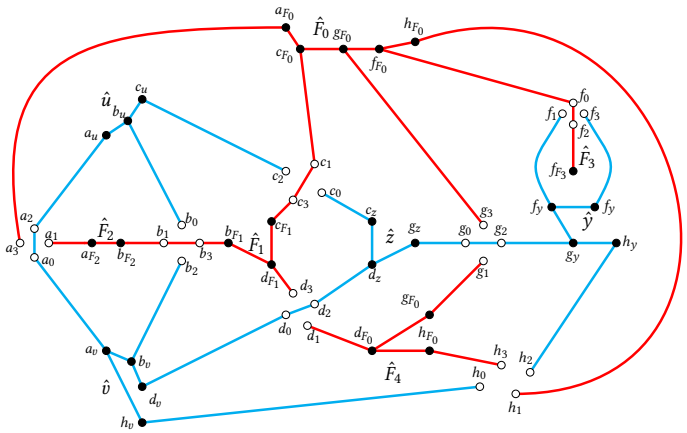


Figura 5.19: *ADPs após o corte da face exterior.*

As linhas 1, 2 e 5 do Algoritmo 21 possuem consumo de tempo constante, pois são consultas a tabela de símbolos $G.H$ e uma chamada da rotina CREATE OCTO. As linhas remanescentes são formadas por um número constante de chamadas de FIND NODE, CYCLE, MERGE, SPLIT, MUDAPESOGDP e CONNECTED e cada uma dessas chamadas possui consumo de tempo assintótico igual a $O(\lg m)$ esperado amortizado, onde m é o número de arestas no grafo corrente. Dessa forma, LIGUEGDP também possui consumo esperado de tempo assintótico $O(\lg m)$ amortizado.

5.7 Estruturas auxiliares

A implementação da biblioteca das árvores dinâmicas planas será detalhada na próxima seção. Para construir essa estrutura de dados, são necessárias duas estruturas fundamentais: a primeira é a árvore binária de busca com chave implícita, discutida inicialmente em detalhes no Capítulo 3 e a segunda são as link cut trees, que serão abordadas brevemente na Seção 5.7.2.

5.7.1 Árvores binárias de busca com chave implícita

Árvores binárias de busca com chave implícita são particularmente úteis para armazenar sequências de objetos. No Capítulo 3 elas foram utilizadas para armazenar uma sequência de arestas que representa uma árvore. No presente capítulo, elas serão usadas para armazenar a sequência de nós de link cut tree que representa a ordem cíclica de cada vértice. Para implementar as rotinas presentes na biblioteca de ADPs, é necessário complementar a biblioteca de ABBs de chave implícita com as seguintes rotinas:

- PRIMEIRO(*nó*): Retorna o nó de menor chave da ABB que contém *nó*.
- ÚLTIMO(*nó*): Retorna o nó de maior chave da ABB que contém *nó*.
- CORTADIREITA(*nó*): Corta a ABB que contém um nó *nó* em duas ABBs. A primeira ABB contém todos os nós com chave menor ou igual a chave de *nó* e a segunda contém todos os nós com chave estritamente maior do que a chave de *nó*. Essa rotina retorna as raízes dessas duas ABBs.

As implementações dessas rotinas estão descritas nos Algoritmos 22, 23 e 24. Seu consumo de tempo será $O(\lg t)$ esperado, onde t é o número de nós da árvore. Como descrito no final da Seção 5.4, essas ABBs são usadas para armazenar a ordem cíclica de cada vértice. Como cada nó é incidente a no máximo todas as m de arestas no grafo dinâmico, concluímos que essas rotinas consumem $O(\lg m)$ esperado.

A rotina PRIMEIRO, descrita no Algoritmo 22, começa seu processamento inicializando uma variável temporária, p, com o nó passado como argumento. Em seguida, é usado a rotina RAIZ, descrita no Algoritmo 7, para obter a raiz da ABB que contém *nó*. Depois de obter a raiz, o algoritmo desce pela árvore, movendo-se sempre para o filho esquerdo, até alcançar o nó mais à esquerda, que representa o nó com menor chave. Finalmente, o nó encontrado é retornado.

A rotina ÚLTIMO, descrita no Algoritmo 23, é análoga à rotina PRIMEIRO. Ele também começa inicializando uma variável temporária, u, com o nó passado como argumento e

Algoritmo 22 PRIMEIRO(*nó*)

```
1  p ← RAIZ(nó)
2  enquanto p.esq ≠ NIL faça
3      p ← p.esq
4  retorne p
```

em seguida, usando RAIZ, obtém a raiz da ABB que contém *nó*. Mas diferentemente de PRIMEIRO, após obter a raiz, ÚLTIMO desce pela árvore, movendo-se sempre para o filho direito, até alcançar o nó mais à direita, que representa o nó com maior chave, que é retornado.

Algoritmo 23 ÚLTIMO(*nó*)

```
1  u ← RAIZ(nó)
2  enquanto u.dir ≠ NIL faça
3      u ← u.dir
4  retorne u
```

A implementação de CORTADIREITA, que pode ser vista no Algoritmo 24, é quase idêntica a de CORTA, que foi descrito no Algoritmo 10. A única diferença entre essas rotinas é na linha 2, em que atribuímos *nó* a *L* em vez de *nó.esq*, como feito em CORTA.

Algoritmo 24 CORTADIREITA(*nó*)

```
1  R ← nó.dir
2  L ← nó
3  tmp ← nó
4  enquanto tmp.pai ≠ NIL faça
5      se tmp.pai.esq = tmp então
6          tmp.pai.esq ← R
7          tmp.pai.tam ← tmp.pai.tam − TAMANHO(L)
8          se R ≠ NIL então
9              R.pai ← tmp.pai
10         R ← tmp.pai
11     senão
12         tmp.pai.dir ← L
13         tmp.pai.tam ← tmp.pai.tam − TAMANHO(R)
14         se L ≠ NIL então
15             L.pai ← tmp.pai
16         L ← tmp.pai
17     tmp ← tmp.pai
18 se L ≠ NIL então L.pai ← NIL
19 se R ≠ NIL então R.pai ← NIL
20 nó.dir ← nó.esq ← nó.pai ← NIL
21 retorne L, R
```

5.7.2 Link cut trees

Link cut trees são uma estrutura de dados usada para representar e manipular florestas dinâmicas enraizadas. Elas foram originalmente introduzidas por Sleator e Tarjan [34] em 1983 e já são bem conhecidas na literatura [30, 23, 36]. Para um texto introdutório sobre link-cut trees, recomendamos o trabalho de Felipe Noronha e Cristina Gomes Fernandes [30]. Cada nó de link cut tree possui um campo *abb* que aponta para o nó de ABB de chave implícita que representa esse nó na ordem cíclica. Essa estrutura de dados dá suporte às seguintes operações:

- NEWLCT(): Cria e retorna um novo nó de link cut tree.
- DELLCT(v): Remove o nó v de link cut tree da memória.
- LINK(v, w): Adiciona uma aresta de v a w, tornando v um filho de w. Essa operação assume que v é a raiz de sua árvore e que w não é um nó da árvore de v.
- CUT(v): Remove a aresta de v para seu pai. Essa operação assume que v não é a raiz de sua árvore.
- EVERT(v): Torna v a raiz de sua árvore revertendo o caminho de v para a raiz original.
- MAX(v): Retorna o nó de peso máximo no caminho entre v e a raiz de sua árvore.
- MIN(v): Retorna o nó de peso mínimo no caminho entre v e a raiz de sua árvore.
- SET WEIGHT(v, w): Atribui o peso w a todos os nós entre v e a raiz da link cut tree.
- GET ROOT(v): Retorna a raiz da link cut tree que contém o nó v.

O consumo de tempo de NEWLCT é constante, o consumo de SET WEIGHT(v, w) é $O(t)$, onde t é o número de nós entre o nó v e a raiz da sua árvore. O consumo de tempo das demais rotinas dessa biblioteca consomem tempo $O(\lg t)$ amortizado, onde t é o número de nós na link cut tree. Como t é $\Theta(m)$, onde m é o número de arestas no grafo corrente, então essas rotinas consumem $O(\lg m)$ amortizado.

5.8 Implementação de árvores dinâmicas planas

Nessa seção, vamos mostrar como usar link cut trees e árvores binárias de busca de chave implícita para implementar biblioteca de árvores dinâmicas planas.

A primeira implementação elaborada é da rotina CREATE OCTO, que pode ser vista no Algoritmo 25. O processamento dessa rotina começa com um laço que para cada $0 \leqslant i \leqslant 3$ cria novos nós de link cut tree s_i e e_i com a rotina NEWLCT. Em seguida, o campo *abb* desses nós é populado com um novo nó de ABB de chave implícita e e_i é atribuído ao campo *nó* desse nó de ABB. Então s_i e e_i são ligados usando a rotina LINK e o peso de e_i é inicializado com o valor w. Por fim o peso de cada s_i é atribuído usando a rotina SET WEIGHT e a óctupla é retornada. O consumo assintótico de tempo dessa rotina é $O(1)$.

A implementação de DESTROY OCTO está descrita no Algoritmo 26. Primeiro a óctupla de e é obtida e apagada da tabela de símbolos, em seguida, cada nó de link cut tree é apagado da memória usando a rotina DELLCT. Seu consumo de tempo é constante.

Algoritmo 25 CREATE OCTO(e, w)

```
1 para i ∈ {0, 1, 2, 3} faça
2     s_i ← newLCT(); e_i ← newLCT()
3     e_i.abb ← novoNó(); s_i.abb ← novoNó()
4     e_i.abb.nó ← e
5     link(e_i, s_i)
6     set weight(e_i, w)
7 set weight(s_0, −∞); set weight(s_2, −∞);
8 set weight(s_1, ∞); set weight(s_3, ∞);
9 retorne e_0, e_1, e_2, e_3, s_0, s_1, s_2, s_3
```

Algoritmo 26 DESTROY OCTO(H, e)

```
1 e_0, e_1, e_2, e_3, s_0, s_1, s_2, s_3 ← H(e)
2 H(e) ← NIL
3 para i ∈ {0, 1, 2, 3} faça
4     delLCT(s_i)
5     delLCT(e_i)
```

A implementação de CONNECTED é descrita no Algoritmo 27. Para consultar se dois nós de link cut trees estão na mesma árvore, simplesmente comparamos a raiz das árvores que contém p e q. Como essa rotina é composta somente por duas chamadas de GET ROOT e uma comparação, então seu consumo de tempo é O(lg m) amortizado, onde m é o número de arestas no grafo corrente, que é o consumo de GET ROOT.

Algoritmo 27 CONNECTED(p, q)

```
1 retorne get Root(p) = get Root(q)
```

A implementação de FIND NODE é simples e usa a estrutura de árvores binárias de busca com chave implícita. Dado o nó s, sabemos que ele está em alguma ABB e que a raiz dessa árvore aponta para o nó que contém s em sua ordem cíclica, logo usamos RAIZ da biblioteca de ABBs para obter a raiz r da árvore e em seguida é retornado $r.nó$. O consumo de tempo de FIND NODE é o mesmo consumo de RAIZ, isto é, O(lg m) esperado, onde m é o número de arestas no grafo corrente.

Algoritmo 28 FIND NODE(s)

```
1 r ← raiz(s.abb)
2 retorne r.nó
```

A rotina SET WEIGHT, descrita no Algoritmo 29, primeiro realiza um EVERT no último nó da ordem ciclica que contém p. Dessa forma, o caminho entre os nós PRIMEIRO(p) e ÚLTIMO(p) coincide com a ordem ciclica que contém p. Em seguida, a rotina SET WEIGHT atribui o peso w para todos os nós entre PRIMEIRO(p) e a raiz da sua árvore, isto é, ÚLTIMO(p). Logo todos os nós da ordem cíclica que contém p têm seus pesos atualizados para o novo valor w. A rotina EVERT consome tempo O(lg m) amortizado, onde m é o número de arestas

no grafo corrente, enquanto ÚLTIMO e PRIMEIRO consomem tempo $O(\lg t)$ esperado, onde t é o número de nós de link cut tree na ordem cíclica que contém p e SET WEIGHT consume tempo $O(t)$. No nosso caso, a rotina SET WEIGHT só é chamada na linha 3 do Algoritmo 19 e nessa chamada, temos que $t \leqslant 2$.

Algoritmo 29 SET WEIGHT(p, w)

```
1 EVERT(ÚLTIMO(p))
2 SET WEIGHT(PRIMEIRO(p), w)
```

As implementações das rotinas FIND MAX e FIND MIN são análogas. Em ambas, é chamado EVERT(p). Em seguida, é atribuído à variável m o valor de MAX(q) (resp. MIN(q)), que corresponde ao nó com maior (resp. menor) peso entre q e a raiz da link cut tree, isto é, p. Então é retornado o valor FIND NODE(m). Veja os Algoritmos 30 e 31. O consumo de tempo das rotinas EVERT e MAX é $O(\lg m)$ amortizado, enquanto que FIND NODE consome tempo esperado $O(\lg m)$. Assim o consumo de tempo de FIND MAX e FIND MIN é $O(\lg m)$ amortizado esperado.

Algoritmo 30 FIND MAX(p, q)

```
1 EVERT(p)
2 m ← MAX(q)
3 retorne FIND NODE(m)
```

Algoritmo 31 FIND MIN(p, q)

```
1 EVERT(p)
2 m ← MIN(q)
3 retorne FIND NODE(m)
```

A implementação da rotina MERGE pode ser vista no Algoritmo 32. O processo de MERGE começa chamando a função EVERT invertendo a árvore de modo que o nó ÚLTIMO(p) se torne a nova raiz. Isso é necessário, pois a operação LINK de adição de aresta subsequente assume que ÚLTIMO(p) é a raiz de sua árvore. A função LINK é usada para adicionar uma aresta entre ÚLTIMO(p) e PRIMEIRO(q), conectando o último nó da ordem cíclica do vértice que contém p ao primeiro nó da ordem cíclica do vértice que contém q. Para garantir que a ordem cíclica das arestas seja mantida corretamente, a função JUNTA é chamada para unir as raízes das ABBs associadas aos vértices. Como PRIMEIRO, ÚLTIMO, RAIZ e JUNTA consomem tempo esperado $O(\lg m)$ e EVERT e LINK consome tempo $O(\lg m)$ amortizado, onde m é o número de arestas no grafo corrente, então o consumo de tempo de MERGE é $O(\lg m)$ esperado amortizado.

Algoritmo 32 MERGE(p, q, z)

```
1 l ← ÚLTIMO(p)
2 EVERT(l)
3 LINK(l, PRIMEIRO(q))
4 T ← JUNTA(RAIZ(p), RAIZ(q))
5 T.nó ← z
```

A implementação da rotina SPLIT pode ser vista no Algoritmo 33. Nessa implementação, primeiro, a função EVERT torna ÚLTIMO(p) a raiz da sua link cut tree, assim o caminho entre PRIMEIRO(p) e a raiz corresponde à ordem cíclica de v. Dessa forma PARENT(p) retorna o sucessor do nó p na ordem cíclica, que é armazenado na variável y e ao aplicar CUT(p)

a ordem cíclica de v é dividida em duas sequências, a primeira de PRIMEIRO(p) até p e a segunda de y até ÚLTIMO(p). Essas sequências correspondem às ordens cíclicas dos vértices resultantes desejados. Finalmente, a árvore binária de busca que contém o nó p é então dividida em duas partes: uma que mantém a ordem das arestas até p, e outra que contém o restante. Como ÚLTIMO e CORTADIREITA consomem tempo esperado O(lg m) e EVERT e CUT consomem tempo O(lg m) amortizado, onde m é o número de arestas no grafo corrente, então o consumo de tempo de SPLIT é O(lg m) esperado amortizado.

Algoritmo 33 SPLIT(p, v, z)

```
1 EVERT(ÚLTIMO(p))
2 CUT(p)
3 T, T' ← CORTADIREITA(p)
4 T.nó ← v; T'.nó ← z
```

A implementação da rotina CYCLE(p), descrita no Algoritmo 34, começa comparando p com o último nó da ordem cíclica que o contém. Caso esses nós sejam iguais, então a função termina imediatamente. Caso contrário, a função EVERT é aplicada a ÚLTIMO(p). Essa operação faz com que o pai de p seja o sucessor dele na ordem cíclica. Então é chamado CUT(p) para remover a aresta que liga esses nós. Em seguida, a função LINK adiciona uma nova aresta que conecta o último nó da ordem cíclica que contém p ao primeiro nó dessa ordem, efetivamente ajustando a ordem cíclica. Finalmente, a árvore binária de busca associada é dividida em partes, reorganizadas, e depois unidas para representar a nova ordem cíclica, garantindo que p seja, de fato, o último nó na ordem cíclica. Como PRIMEIRO, ÚLTIMO, CORTADIREITA e JUNTA consomem tempo esperado O(lg m) e EVERT, CUT e LINK consomem tempo O(lg m) amortizado, onde m é o número de arestas no grafo corrente, então o consumo de tempo de CYCLE é O(lg m) esperado amortizado.

Algoritmo 34 CYCLE(p)

```
1 l ← ÚLTIMO(p)
2 se l = p então
3     retorne
4 EVERT(l); CUT(p)
5 LINK(l, PRIMEIRO(p))
6 T, T' ← CORTADIREITA(p)
7 T ← JUNTA(T', T)
8 T.nó ← p
```

A implementação de PRED está descrita no Algoritmo 35. Seu consumo esperado é $O(\lg m)$, onde m é o número de arestas no grafo corrente.

Algoritmo 35 PRED(p)

```
1  se p = PRIMEIRO(p) então
2      retorne ÚLTIMO(p)
3  se p.abb.esq ≠ NIL então
4      tmp ← p.abb.esq
5      enquanto tmp.dir ≠ NIL faça tmp ← tmp.dir
6      retorne tmp
7  tmp ← p
8  enquanto tmp.pai.esq = tmp faça tmp ← tmp.pai
9  tmp ← tmp.pai.esq
10 enquanto tmp.dir ≠ NIL faça tmp ← tmp.dir
11 retorne tmp
```

Capítulo 6

O limitante inferior de $\Omega(\lg n)$

Nesse capítulo explicaremos o limitante inferior de consumo de tempo amortizado de $\Omega(\lg n)$ para o problema de conexidade em grafos dinâmicos com n vértices [31]. Esse limitante é incondicional e é válido mesmo para os algoritmos que usam técnicas de aleatorização, no estilo Las Vegas ou Monte Carlo. Além disso, esse limitante é válido mesmo para o problema de conexidade em grafos dinâmicos restrito a florestas ou coleção de caminhos.

Esse resultado é consequência de um limitante inferior para o problema de verificação de soma parcial em S_k (VSPS_k), que será elaborado na Seção 6.2. Para transportar o limitante de um problema para outro, reduziremos o problema VSPS_k ao problema de conexidade em grafos dinâmicos.

O limitante inferior para o problema VSPS_k usa o modelo de computação *cell-probe*. Dessa forma nosso limitante também aplica-se a esse modelo. Portanto, devemos iniciar nossa discussão na próxima seção explicando como esse modelo de computação funciona e como é mensurado o consumo de tempo de um algoritmo nesse modelo. Em seguida, definiremos VSPS_k e seu limitante inferior formalmente e, concluindo esse capítulo, faremos a redução dele para o problema de conexidade em grafos dinâmicos.

6.1 O modelo de computação cell-probe

No modelo cell-probe, a memória do computador é representada por uma coleção de células. Cada célula é composta por uma quantidade fixa de b bits e possui um identificador único, que é chamado de **endereço** da célula. Algoritmos nesse modelo podem ler e escrever dados nas células e realizar operações elementares, como aritmética básica, em uma unidade de processamento externa à memória [1].

Ao limitar o número de células para 2^b, garantimos que o endereço de qualquer célula pode ser armazenado em uma única célula, assim aproximando esse modelo abstrato à implementação de memória RAM dos computadores usuais. Lembramos que, nessa implementação de computadores, as operações aritméticas e *bit-wise* são realizadas na CPU, externas à memória e o consumo de tempo real causado por essas operações é muito menor do que o tempo de escrita e leitura da memória. O modelo *cell-probe* representa essa

disparidade de consumo de tempo, definindo o consumo de tempo de um algoritmo como sendo proporcional à quantidade de células da memória escritas ou lidas, desconsiderando assim o tempo necessário para a realização de operações que, em um computador usual, seriam realizadas pela CPU.

Muitos resultados sobre algoritmos que usam esse modelo estão em função do parâmetro b, pois esse parâmetro determina quanta informação pode ser armazenada em uma única célula. Outra informação que parametriza o consumo de tempo nesse modelo é a quantidade δ de bits necessários para representar cada parâmetro de entrada do algoritmo. É costumeiro separar esses parâmetros, pois em diversas aplicações um tende a ser assintoticamente maior do que o outro.

6.2 Verificação de soma parcial em S_k

O grupo S_k é o grupo finito formado pelo conjunto de todas as bijeções sobre o conjunto $[k] := \{1, 2, \dots, k\}$ munido da operação de composição de funções [26]. Essas bijeções também são chamadas de **permutações**. Podemos visualizar uma permutação desenhando o domínio e contradomínio em duas colunas e desenhando setas relacionando cada elemento do domínio com sua imagem, como feito na Figura 6.1.

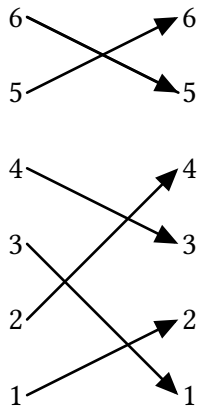


Figura 6.1: *Exemplo de representação de uma permutação com $k = 6$.*

Podemos, com a técnica de visualização empregada na Figura 6.1, ilustrar também como é feita a composição de uma sequência de permutações $\phi = (\phi_1, \phi_2, \dots, \phi_p)$, que pode ser vista na Figura 6.2. Algebricamente falando, essa composição é somente a composição de funções. Visualmente, a composição forma k caminhos direcionados constituídos pelas setas que ilustram cada permutação. Podemos calcular a permutação φ resultante da composição das permutações de ϕ percorrendo, para cada elemento $q \in [k]$, o caminho direcionado iniciado em q no domínio de ϕ_1 até chegar ao valor r no contradomínio de ϕ_p. Dessa forma, teremos $\varphi(q) = r$. Também podemos, com essa ideia de percorrer os caminhos direcionados, calcular a composição parcial de ϕ_1 até ϕ_i, percorrendo o caminho parcialmente até o contradomínio de ϕ_i, com $1 \leqslant i \leqslant p$.

Notemos que a substituição de um ϕ_i da sequência ϕ pode alterar drasticamente os k caminhos ilustrados e consequentemente alterar a permutação resultante φ. Surge então um

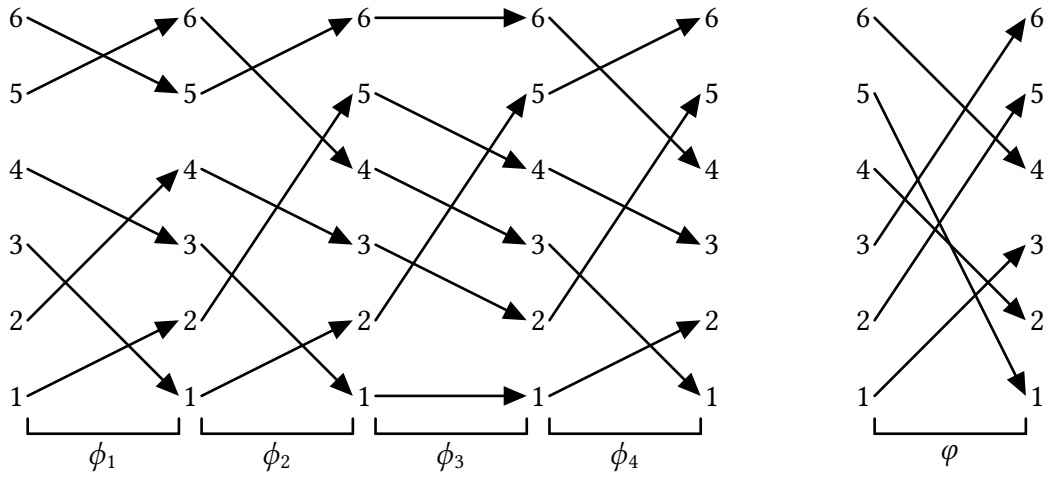


Figura 6.2: *Exemplo de uma composição de permutações adotando $p = 4$. Temos $\varphi = \phi_4 \circ \phi_3 \circ \phi_2 \circ \phi_1$.*

novo problema dinâmico, o **problema da verificação de soma parcial em S_k (VSPS_k)**, que visa manter uma sequência de p permutações $\phi = (\phi_1, \phi_2, \ldots, \phi_p)$ de forma a implementar eficientemente a seguinte biblioteca:

- SUBSTITUA(ϕ, i, φ): a i-ésima coordenada de ϕ passa a ser a permutação φ ; e
- VERIFIQUE(ϕ, i, φ): retorna verdadeiro se $\phi_i \circ \cdots \circ \phi_1 = \varphi$ e falso, caso contrário.

Mihai Patrascu e Erik D. Demaine [31] provaram o seguinte resultado:

Teorema 5. *Os consumos de tempo t_u e t_q das rotinas SUBSTITUA e VERIFIQUE, respectivamente, implementadas com qualquer estrutura de dados sob o modelo cell-probe para solucionar VSPS_k estão relacionados e limitados por*

$$\min\{t_u, t_q\} \lg \left(\frac{\max\{t_u, t_q\}}{\min\{t_u, t_q\}} \right) = \Omega \left(\frac{\delta}{b} \lg p \right),$$

onde cada célula possui b bits e são necessários δ bits para representar cada parâmetro da rotina. Esse limitante continua válido mesmo se a estrutura de dados utilizar amortização, não determinismo, aleatorização Las Vegas ou Monte Carlo com erro probabilístico $p^{-\Omega(1)}$.

A demonstração completa deste resultado é longa, complexa e foge do escopo desse texto, logo optamos por não detalhá-la aqui e somente tecer alguns comentários sobre ela.

No artigo, os autores demonstram um lema [31, Lema 5.1] cuja prova envolve a construção de uma árvore binária que modela o fluxo de células da memória lidas e escritas ao longo de uma sequência de operações da biblioteca dinâmica de VSPS_k. Esse lema limita inferiormente a quantidade esperada de leituras feitas em células escritas ao longo dessa sequência, o que implica, como os autores mostram ao fim da Seção 5.2 do artigo, em um limitante inferior de

$$\Omega \left(\frac{\delta}{b} \lg p \right) \tag{6.1}$$

amortizado para cada operação dessa sequência.

Há uma nuance na amortização presente nesse limitante. A sequência de operações é composta por ambas as rotinas SUBSTITUA e VERIFIQUE. Logo a amortização do custo da sequência por operação significa que pelo menos uma dessas rotinas, mas não necessariamente *ambas*, está limitada inferiormente por $\Omega(\frac{\delta}{b} \lg p)$. Ou seja, é possível que uma dessas rotinas consuma tempo constante e assim, nesse caso, a outra necessariamente consumirá tempo $\Omega(\frac{\delta}{b} \lg p)$. Elaboraremos um exemplo envolvendo grafos dinâmicos em que isso ocorre ao final desse capítulo.

Na Seção 5.5 do artigo, os autores explicitam essa nuance desenvolvendo um método que converte limitantes amortizados em limitantes que relacionam t_u a t_q e, ao aplicar esse método ao limitante (6.1), concluem o Teorema 5. Pontuamos que esse método se apoia em uma análise mais fina da sequência modelada pela árvore binária usada pelo lema citado anteriormente, logo discorrer detalhadamente sobre ele também foge do escopo desse trabalho.

6.3 Redução do problema VSPS_k para conexidade em grafos dinâmicos

Para fazer a redução de VSPS_k ao problema de conexidade em grafos dinâmicos, seguiremos o processo descrito na Seção 6.1 de Patrascu e Demaine [31] para converter uma instância de VSPS_k em uma instância do problema de conexidade em grafos dinâmicos.

A Figura 6.2 nos indica como vamos traduzir um problema no outro. Em essência, vamos converter cada número dessa figura em um vértice e cada seta em uma aresta, obtendo assim um grafo formado por k caminhos disjuntos, cada um de comprimento p, como pode ser visto na Figura 6.3. Formalmente, para a sequência ϕ de uma instância de VSPS_k, construiremos um grafo dinâmico $G(\phi)$ cujo conjunto de vértices consiste nos pares (x, y), com $1 \leqslant x \leqslant p+1$ e $1 \leqslant y \leqslant k$. Logo $G(\phi)$ terá $n := k \cdot (p+1)$ vértices. Cada vértice (x, y) com $1 \leqslant x \leqslant p$ será adjacente ao vértice $(x+1, \phi_x(y))$.

Em nossas rotinas, também será necessário calcular o valor $\phi_x(y)$ em O(1). Para isso, manteremos uma cópia de ϕ junto a $G(\phi)$ e, para manter o pseudocódigo mais limpo, quando for claro, denotaremos essa cópia simplesmente por ϕ em vez da descrição mais carregada $G(\phi).\phi$.

Podemos encapsular essa conversão em uma rotina chamada CONVERTA(ϕ, p, k), descrita no Algoritmo 36, que recebe a sequência ϕ, seu comprimento p e o tamanho k do domínio e contradomínio das permutações e retorna o grafo dinâmico $G(\phi)$.

Com essa conversão feita, podemos implementar a biblioteca de VSPS_k usando a biblioteca de conexidade em grafos dinâmicos. A implementação de SUBSTITUA($G(\phi)$, i, φ) pode ser vista no Algoritmo 37. Nesse algoritmo, primeiro removemos todas as arestas associadas à permutação ϕ_i. Em seguida, inserimos k novas arestas ligando (i, y) a $(i+1, \varphi(y))$, para cada $y \in [k]$.

Implementamos VERIFIQUE($G(\phi)$, i, φ) com k chamadas à consulta de conexidade em

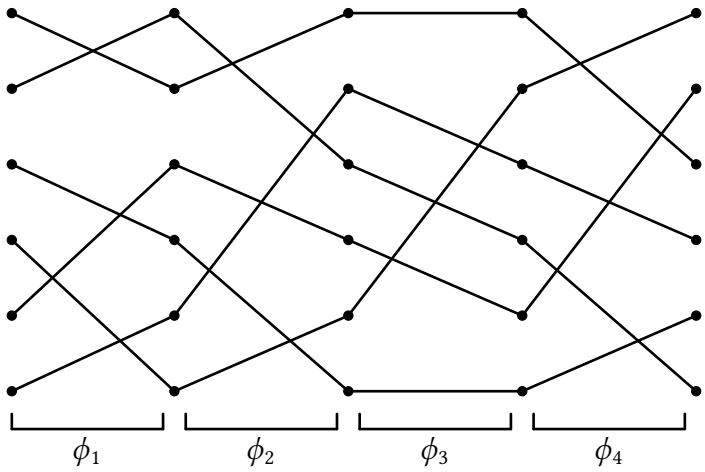


Figura 6.3: *Instância de VSPS$_k$ da Figura 6.2 convertida em uma instância do problema de conexidade em grafos dinâmicos.*

Algoritmo 36 CONVERTA(ϕ, p, k)

```
1 G(φ) ← NOVOGD((p + 1) · k)
2 G(φ).φ ← φ
3 para x ← 1 até p + 1 faça
4     para y ← 1 até k faça
5         LIGUEGD(G(φ), (x, y), (x + 1, φ_x(y)))
6 retorne G(φ)
```

Algoritmo 37 SUBSTITUA($G(\phi)$, i, φ)

```
1 para y ← 1 até k faça
2     REMOVAGD(G(φ), (i, y), (i + 1, φ_i(y)))
3 para y ← 1 até k faça
4     LIGUEGD(G(φ), (i, y), (i + 1, φ(y)))
5 φ_i ← φ
```

grafos dinâmicos, feita pela rotina CONECTADOGD. Para cada $y \in [k]$, testamos a conexidade entre os vértices $(1, y)$ e $(i+1, \varphi(y))$. O teste retorna verdadeiro se e só se existe um caminho entre esses vértices. Mas, nesse grafo, se existe um tal caminho em $G(\phi)$, então $\varphi(y) = \phi_i \circ \cdots \circ \phi_1(y)$. Caso todos os testes de conexidade retornem verdadeiro, então teremos que $\varphi = \phi_i \circ \cdots \circ \phi_1$ e VERIFIQUE deve retornar verdadeiro. Caso contrário, essa rotina deve retornar falso.

Para entender essa rotina melhor, vamos esmiuçar a execução dela em um exemplo. Na Figura 6.4, vemos um grafo $G(\phi)$ e uma permutação φ e queremos verificar se a composição de todas as cinco permutações é igual à permutação φ, também ilustrada nessa figura. Para tal, realizaremos a chamada VERIFIQUE($G(\phi)$, 5, φ). Como podemos ver no Algoritmo 38, nessa chamada a rotina VERIFIQUE testará a conexidade entre os vértices da forma $(1, y)$ e $(6, \varphi(y))$, para cada $y \in [6]$. O primeiro desses testes será entre os vértices $(1, 1)$ e $(6, 3)$, pois, como pode ser visto na figura, temos que $\varphi(1) = 3$. Como podemos constatar pela figura,

Algoritmo 38 VERIFIQUE($G(\phi)$, i, φ)

```
1 para y ← 1 até k faça
2     se não CONECTADOGD(G(φ), (1, y), (i + 1, φ(y))) então
3         retorne falso
4 retorne verdadeiro
```

existe um caminho ligando esses vértices, logo o teste de conexidade retorna verdadeiro e verificamos assim que

$$\varphi(1) = 3 = \phi_5 \circ \phi_4 \circ \phi_3 \circ \phi_2 \circ \phi_1(1).$$

Iremos então continuar testando a conexidade entre esses pares de vértices até que ou encontremos um par desconexo, que significa que $\varphi(y) \neq \phi_5 \circ \cdots \circ \phi_1(y)$, ou testamos todos, o que significa que de fato φ é igual à composição de todas as cinco permutações de ϕ. O segundo teste de conexidade é entre os vértices $(1, 2)$ e $(6, 1)$, pois $\varphi(2) = 1$, e novamente podemos observar que existe um caminho entre esses vértices, assim a consulta de conexidade retorna verdadeiro e portanto verificamos que $\varphi(2) = \phi_5 \circ \cdots \circ \phi_1(2)$. O mesmo ocorre para as duas próximas iterações do laço da rotina VERIFIQUE. Como $\varphi(3) = 4$ e $\varphi(4) = 5$, testamos sequencialmente a conexidade entre os vértices $(1, 3)$ e $(6, 4)$ e entre $(1, 4)$ e $(6, 5)$, verificando em ambos os casos que esses vértices estão interligados. Analogamente, como $\varphi(5) = 6$, testamos a conexidade entre $(1, 5)$ e $(6, 6)$, que retorna falso, pois não há caminho entre esses vértices. Isso significa que $\varphi(5) \neq \phi_5 \circ \cdots \circ \phi_1(5)$. Podemos verificar essa desigualdade pela Figura 6.4, acompanhando o caminho iniciado em $(1, 5)$. Podemos constatar que ele liga esse vértice ao vértice $(6, 2)$, o que representa que

$$\varphi_5 \circ \phi_4 \circ \phi_3 \circ \phi_2 \circ \phi_1(5) = 2 \neq 6 = \varphi(5).$$

Ao encontrar essa desconexidade, encerramos a nossa busca e retornamos falso, finalizando assim a execução da chamada VERIFIQUE($G(\phi)$, 5, φ).

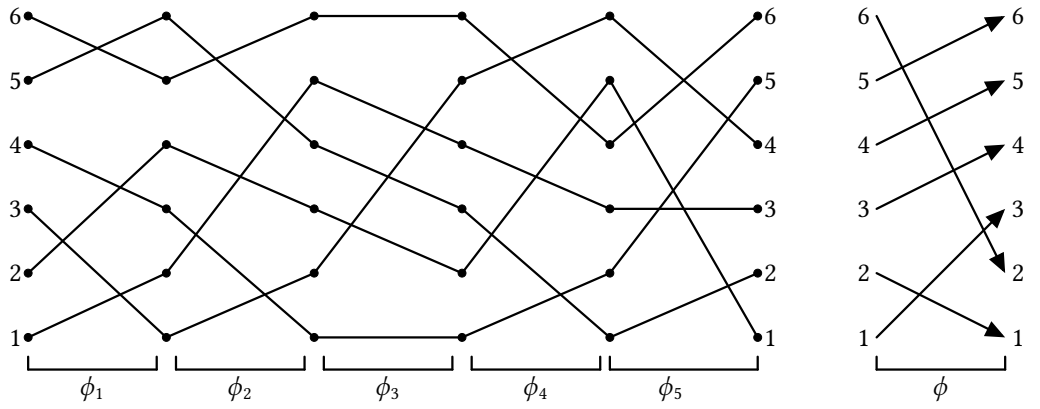


Figura 6.4: *Instância de VSPS$_k$ da Figura 6.2 convertida em uma instância do problema de conexidade em grafos dinâmicos.*

6.4 Limitante inferior para conexidade em grafos dinâmicos

Com as implementações das rotinas SUBSTITUA e VERIFIQUE descritas respectivamente pelos Algoritmos 37 e 38, podemos transferir o limitante inferior do Teorema 5 para o problema de conexidade em grafos dinâmicos.

Para fazer isso, calcularemos os valores dos parâmetros δ e b usados no enunciado do Teorema 5. Em particular, definiremos δ em termos de b. Ambas as rotinas têm como parâmetro a tripla $(G(\phi), i, \varphi)$. Primeiro notemos que $G(\phi)$ é passado por referência e logo, como comentado na Seção 6.1, um endereço de célula será armazenado em uma única célula e assim usará até b bits. O inteiro i também usa b bits, já que também assumimos que deva ser possível armazenar os inteiros com que trabalhamos em uma célula. Para implementar a permutação φ, mantemos um vetor V de inteiros com comprimento k de forma que $V[i] = \varphi(i)$. Com essa implementação, φ precisará de $k \cdot b$ bits, pois cada entrada de V consome b bits. Assim concluímos que $\delta = \Theta(k \cdot b)$. Substituindo δ no resultado do Teorema 5, obtemos que

$$\Omega\left(\frac{\delta}{b} \lg p\right) \implies \Omega\left(\frac{k \cdot b}{b} \lg p\right) \implies \Omega(k \lg p). \tag{6.2}$$

Lembremos que a igualdade $n = k \cdot (p + 1)$ relaciona os valores k e p ao número n de vértices do grafo dinâmico $G(\phi)$. Para obter um limitante exclusivamente em função de n, escolheremos uma família de instâncias de VSPS_k que permita a substituição de k e p da Equação (6.2) por n. Especificamente, escolheremos a família em que $k = p - 1$. Para essa escolha, temos

$$n = k \cdot (p + 1) = (p - 1) \cdot (p + 1) = p^2 - 1.$$

Dessa forma, teremos:

$$\Omega(k \lg p) = \Omega(k \cdot 2 \cdot \lg p) = \Omega(k \lg p^2) = \Omega(k \lg n).$$

Para concluirmos, note que $\Omega(k \lg n)$ limita inferiormente o consumo de tempo das rotinas SUBSTITUA e VERIFIQUE. O algoritmo SUBSTITUA faz k remoções e inserções de arestas e VERIFIQUE faz k consultas de conexidade. Logo se denotarmos por t_m a soma do consumo de tempo de uma execução de LIGUEGD e REMOVAGD e t_c o consumo de tempo de CONECTADOGD, então deduzimos que $t_u = kt_m$ e $t_q = kt_c$. Substituindo esses valores no Teorema 5, teremos:

$$\begin{aligned}
\min\{t_u, t_q\} \lg\left(\frac{\max\{t_u, t_q\}}{\min\{t_u, t_q\}}\right) &= \Omega\left(\frac{\delta}{b} \lg p\right) &&\implies \\
\min\{kt_m, kt_c\} \lg\left(\frac{\max\{kt_m, kt_c\}}{\min\{kt_m, kt_c\}}\right) &= \Omega(k \lg n) &&\implies \\
k \min\{t_m, t_c\} \lg\left(\frac{\max\{t_m, t_c\}}{\min\{t_m, t_c\}}\right) &= \Omega(k \lg n).
\end{aligned}$$

Portanto obtemos o seguinte limitante amortizado:

$$\min\{t_m, t_c\} \lg \left(\frac{\max\{t_m, t_c\}}{\min\{t_m, t_c\}} \right) = \Omega(\lg n)$$

para cada uma das chamadas das operações de conexidade em grafos dinâmicos.

Ressaltamos novamente que esse limitante implica que pelo menos um entre t_m e t_c é $\Omega(\lg n)$, mas não necessariamente ambos.

Como exemplo, em 2015, Kejlberg-Rasmussen *et al.* [25] apresentaram uma estrutura de dados que permite fazer consulta de conexidade em grafos dinâmicos em tempo constante e, respeitando o limite inferior elaborado aqui, possui consumo de tempo para adição e remoção de arestas de $O(\sqrt{\frac{n(\lg \lg n)^2}{\lg n}})$.

Capítulo 7

Estudos experimentais

Nessa seção, comentaremos as principais avaliações empíricas já realizadas sobre algoritmos que resolvem o problema de conectividade em grafos dinâmicos. Iniciamos com uma revisão histórica dessas avaliações já realizadas, destacando os avanços e as limitações identificadas ao longo do tempo. Em seguida, comentaremos o desenrolar da nossa própria análise empírica.

Em 1997, Alberts, Cattaneo e Italiano [2] fizeram um estudo experimental envolvendo um algoritmo simples baseado em esparsificação proposto por Eppstein et al. [10] e o algoritmo de Henzinger e King [18], sem a melhoria de Henzinger e Thorup [17].

O algoritmo de Henzinger e King, assim como o de Holm, de Lichtenberg e Thorup [20] estudado no Capítulo 4, associa um nível entre 0 e $\lceil \lg n \rceil$ a cada aresta. Nesse experimento, os autores propuseram heuristicamente truncar essa estrutura de níveis, mantendo assim somente os níveis entre k e $\lceil \lg n \rceil$, onde k é um parâmetro pré-definido. Nessa simplificação, no nível k, em vez de fazer a busca por aresta substituta proposta por Henzinger e King, o algoritmo faz uma busca exaustiva. Dessa forma, essa simplificação possui consumo de tempo assintótico pior do que o algoritmo original de Henzinger e King, porém ainda assim ela se saiu bem nos experimentos com grafos aleatórios.

Para grafos não aleatórios, o algoritmo baseado em esparsificação se saiu melhor para instâncias com poucas operações de atualização, enquanto que o algoritmo original de Henzinger e King se saiu melhor com mais operações de atualização. As implementações desenvolvidas para estes experimentos foram feitas em C++, usando a plataforma Leda [27].

Em 2002, Raj Iyer, David Karger, Hariharan Rahul e Mikkel Thorup [22] usaram como base o estudo de Alberts et al. para comparar o então recente algoritmo de Holm, de Lichtenberg e Thorup [20] com o algoritmo de Henzinger e King, considerando algumas variantes destes dois algoritmos no seu estudo. Entre outras coisas, os autores mostraram que uma das variantes de Henzinger e King considerada tem consumo de tempo $O(\lg^2 n)$ por operação de atualização.

Os autores também propõem duas heurísticas para o algoritmo de Holm, de Lichtenberg e Thorup. Essas heurísticas não invalidam as análises do consumo de tempo do algoritmo

original.

A primeira heurística usa a ideia do algoritmo de Henzinger e King de fazer um sorteio aleatório das primeiras arestas que são testadas como possíveis substitutas, em vez de percorrer sequencialmente as arestas candidatas e ir fazendo os rebaixamentos.

A segunda heurística é inspirada na heurística analisada no estudo de Alberts e outros, de truncar o número de níveis usados.

O resultado do estudo experimental em relação a estes algoritmos é que a versão do algoritmo de Holm, de Lichtenberg e Thorup com as duas heurísticas implementadas se sai melhor que o algoritmo original de Henzinger e King. As implementações desenvolvidas para estes experimentos também foram feitas em C++ usando a plataforma Leda [27].

Em 2019, David Fernández-Baca e Lei Liu [14] realizaram uma avaliação de heurísticas envolvendo o algoritmo de Holm, de Lichtenberg e Thorup focada em solucionar problemas de biologia computacional com esse algoritmo. Esse estudo reforça a eficiência das heurísticas aplicadas a esse algoritmo.

Mais recentemente, Chen et al. [6] apresentaram um outro estudo experimental, envolvendo duas heurísticas que eles propuseram e supostamente o algoritmo de Henzinger e King, entre outros. A nossa intenção inicial era incluir o algoritmo de Holm, de Lichtenberg e Thorup nesse estudo experimental. Este foi o principal motivo que nos levou a implementar o algoritmo deles em Python 3, que é a linguagem usada neste estudo experimental. No entanto, durante a fase de testes usando como base o estudo experimental de Chen e outros, notamos que a implementação do algoritmo de Henzinger e King incluída no estudo tratava-se de sua versão simplificada com níveis truncados, usada no estudo de Alberts e outros, para a qual a análise original não se aplica. Na verdade, a implementação de Chen e outros desta simplificação também executa a escolha aleatória das arestas candidatas a substitutas de uma maneira pouco eficiente, o que resulta em uma implementação com consumo de tempo muito pior que o consumo do algoritmo original de Henzinger e King. Ou seja, não é de fato uma comparação entre as heurísticas deles e o algoritmo de Henzinger e King, como é dito no artigo. Após percebermos estes problemas, desistimos de estender este estudo experimental. Ademais, nesse meio tempo, também encontramos o trabalho de Iyer, Karger, Rahul e Thorup [22] que já apresenta um excelente estudo comparativo entre o algoritmo de Henzinger e King e o algoritmo de Holm, de Lichtenberg e Thorup. De qualquer modo, produzimos uma implementação do algoritmo de Holm, de Lichtenberg e Thorup em Python3, que pode ser acessada em [32].

Referências

[1] Miklós Ajtai. “A lower bound for finding predecessors in Yao’s cell probe model”. *Combinatorica* 8 (1988), pp. 235–247. issn: 1439-6912. doi: 10.1007/BF02126797 (citado na pg. 61).

[2] David Alberts, Giuseppe Cattaneo e Giuseppe F. Italiano. “An empirical study of dynamic graph algorithms”. *ACM J. Exp. Algorithmics* 2 (1997), pp. 5–45. issn: 1084-6654. doi: 10.1145/264216.264223. url: https://doi.org/10.1145/264216.264223 (citado nas pgs. 3, 69).

[3] Giuseppe Amato, Giuseppe Cattaneo e Giuseppe F. Italiano. “Experimental analysis of dynamic minimum spanning tree algorithms”. In: *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’97. New Orleans, Louisiana, USA: Society for Industrial e Applied Mathematics, 1997, pp. 314–323. isbn: 0898713900 (citado na pg. 3).

[4] Cecilia Aragon e Raimund Seidel. “Randomized search trees”. In: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*. 1989, pp. 540–545 (citado na pg. 18).

[5] Giuseppe Cattaneo, Pompeo Faruolo, Umberto Ferraro Petrillo e Giuseppe F. Italiano. “Maintaining dynamic minimum spanning trees: an experimental study”. In: *Algorithm Engineering and Experiments*. Ed. por David M. Mount e Clifford Stein. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 111–125 (citado na pg. 4).

[6] Qing Chen, Oded Lachish e Sven Helmer Michal H. Böhlen. “Dynamic spanning trees for connectivity queries on fully-dynamic undirected graphs (extended version)”. In: *Proceedings of the Very Large Data Bases Endowment*. VLDB ’22. Very Large Data Bases, 2022, pp. 3263–3276. isbn: 21508097 (citado nas pgs. 4, 5, 70).

[7] Francis Chin e David Houck. “Algorithms for updating minimal spanning trees”. *Journal of Computer and System Sciences* 16.3 (1978), pp. 333–344. issn: 0022-0000. doi: https://doi.org/10.1016/0022-0000(78)90022-3. url: https://www.sciencedirect.com/science/article/pii/0022000078900223 (citado na pg. 3).

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. 4ª ed. The MIT Press, 2022. ISBN: 0262032937; 9780262032933; 0262531968; 9780262531962; 0070131511; 9780070131514 (citado nas pgs. 8, 10, 16).

[9] Reinhard Diestel. *Graph Theory*. 6ª ed. Graduate Texts in Mathematics. Springer, 2024. ISBN: 9783540261834; 3540261834 (citado nas pgs. 33–35, 42).

[10] David Eppstein. "Sparsification – A technique for speeding up dynamic graph algorithms". In: *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*. FOCS '92. USA: IEEE Computer Society, 1992, pp. 60–69. ISBN: 0818629002. DOI: 10.1109/SFCS.1992.267818. URL: https://doi.org/10.1109/SFCS.1992.267818 (citado nas pgs. 3, 5, 69).

[11] David Eppstein, Zvi Galil, Giuseppe F. Italiano e Amnon Nissenzweig. "Sparsification – A technique for speeding up dynamic graph algorithms". *Journal of the ACM* 44.5 (1997), pp. 669–696 (citado na pg. 3).

[12] David Eppstein *et al.* "Maintenance of a minimum spanning forest in a dynamic plane graph". *Journal of Algorithms* 13.1 (1992), pp. 33–54. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(92)90004-V (citado nas pgs. iii, v, 2, 33, 36).

[13] Panagiota Fatourou, Paul Spirakis, Panagiotis Zarafidis e Anna Zoura. "Implementation and experimental evaluation of graph connectivity algorithms using Leda". In: *Algorithm Engineering*. Ed. por Jeffrey S. Vitter e Christos D. Zaroliagis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 124–138. ISBN: 978-3-540-48318-2 (citado na pg. 3).

[14] David Fernández-Baca e Lei Liu. "Tree compatibility, incomplete directed perfect phylogeny, and dynamic graph connectivity: an experimental study". *Algorithms* 12.3 (2019). ISSN: 1999-4893. DOI: 10.3390/a12030053. URL: https://www.mdpi.com/1999-4893/12/3/53 (citado nas pgs. 3, 70).

[15] Greg N. Frederickson. "Data structures for on-line updating of minimum spanning trees, with applications". *SIAM Journal on Computing* 14 (1985), pp. 781–798 (citado nas pgs. 3, 5).

[16] Kathrin Hanauer, Monika Rauch Henzinger e Christian Schulz. "Recent advances in fully dynamic graph algorithms – A quick reference guide". *ACM J. Exp. Algorithmics* 27 (2022). URL: https://doi.org/10.1145/3555806 (citado na pg. 3).

[17] Monika R. Henzinger e Mikkel Thorup. "Sampling to provide or to bound: with applications to fully dynamic graph algorithms". *Random Structures and Algorithms* 11.4 (1997), pp. 369–379 (citado nas pgs. 3, 5, 69).

[18] Monika Rauch Henzinger e Valerie King. "Randomized dynamic graph algorithms with polylogarithmic time per operation". In: *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*. STOC '95. Las Vegas, Nevada, USA: Association for Computing Machinery, 1995, pp. 519–527. ISBN: 0897917189. DOI: 10.1145/225058.225269 (citado nas pgs. 3, 5, 8, 69).

[19] Monika Rauch Henzinger e Valerie King. "Maintaining minimum spanning trees in dynamic graphs". In: *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 594–604. ISBN: 978-3-540-69194-5.

[20] Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*. STOC '98. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 79–89. ISBN: 0897919629. DOI: 10.1145/276698.276715. URL: https://doi.org/10.1145/276698.276715 (citado nas pgs. iii, v, 2–5, 25, 69).

[21] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz e Seth Pettie. "Fully dynamic connectivity in O(log n(log log n)²) amortized expected time". In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2017, pp. 510–520. DOI: 10.1137/1.9781611974782.32 (citado nas pgs. 4, 5).

[22] Raj Iyer, David Karger, Hariharan Rahul e Mikkel Thorup. "An experimental study of polylogarithmic, fully dynamic, connectivity algorithms". *ACM J. Exp. Algorithmics* 6 (2002), pp. 4–34. ISSN: 1084-6654. DOI: 10.1145/945394.945398. URL: https://doi.org/10.1145/945394.945398 (citado nas pgs. 3, 69, 70).

[23] Ming-Yang Kao. *Encyclopedia of Algorithms*. 2ª ed. Springer Publishing Company, 2016. ISBN: 978-1-4939-2865-1 (citado na pg. 55).

[24] Bruce M. Kapron, Valerie King e Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time". In: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA' 13. New Orleans, Louisiana: Society for Industrial e Applied Mathematics, 2013, pp. 1131–1142 (citado nas pgs. 4, 5).

[25] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie e Mikkel Thorup. "Faster Worst Case Deterministic Dynamic Connectivity". In: *Proceedings of the 24th Annual European Symposium on Algorithms*. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 53:1–53:15 (citado nas pgs. 4, 5, 68).

[26] Paulo Agozzini Martin. *Grupos, Corpos e Teoria de Galois*. 1ª ed. Editora Livraria da Física, 2010. ISBN: 978-85-7861-065-4 (citado na pg. 62).

[27] Kurt Mehlhorn e Stefan Näher. "Leda: a platform for combinatorial and geometric computing". *Commun. ACM* 38.1 (1995), pp. 96–102. ISSN: 0001-0782. DOI: 10.1145/204865.204889 (citado nas pgs. 69, 70).

[28] Sotiris E. Nikoletseas, John H. Reif, Paul G. Spirakis e Moti Yung. “Stochastic graphs have short memory: fully dynamic connectivity in poly-log expected time”. In: *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*. Vol. 944. Lecture Notes in Computer Science. Springer-Verlag, 1995, pp. 159–170. isbn: 3-540-60084-1. doi: 10.1007/3-540-60084-1_71 (citado na pg. 3).

[29] Alexandre Noma. “Análise experimental de algoritmos de planaridade”. Dissertação de Mestrado. São Paulo: Instituto de Matemática e Estatística, Universidade de São Paulo, 2003 (citado na pg. 34).

[30] Felipe Castro de Noronha e Cristina Gomes Fernandes. “Link-cut trees e aplicações em estruturas de dados retroativas”. Trabalho de Conclusão de Curso. Instituto de Matemática e Estatística, Universidade de São Paulo, 2022. url: https://bdta.abcd.usp.br/directbitstream/2fbf453b-b7db-4f95-a805-e41dc8eb7b0f/3122356.pdf (citado na pg. 55).

[31] Mihai Patrascu e Erik D. Demaine. “Logarithmic lower bounds in the cell-probe model”. *SIAM Journal on Computing* 35.4 (2006), pp. 932–963. doi: 10.1137/S0097539705447256 (citado nas pgs. 2, 4, 61, 63, 64).

[32] Arthur Henrique Dias Rodrigues. *Repositório Git.* https://github.com/ArthurHDRodrigues/dynamic-connectivity/. Acessado em: 2024-09-28 (citado nas pgs. 2, 24, 70).

[33] Raimund Seidel e Cecilia Aragon. “Randomized search trees”. *Algorithmica* 16 (1996), pp. 540–545 (citado na pg. 18).

[34] Daniel D. Sleator e Robert Endre Tarjan. “A data structure for dynamic trees”. *Journal of Computer and System Sciences* 26 (1983), pp. 362–391 (citado nas pgs. 7, 55).

[35] Philip Martin Spira e Anthony Pan. “On finding and updating spanning trees and shortest paths”. *SIAM Journal on Computing* 4.3 (1975), pp. 375–380. doi: 10.1137/0204032 (citado na pg. 3).

[36] Robert E. Tarjan e Renato F. Werneck. “Dynamic trees in practice”. *ACM J. Exp. Algorithmics* 14 (2010). issn: 1084-6654. doi: 10.1145/1498698.1594231. url: https://doi.org/10.1145/1498698.1594231 (citado nas pgs. 4, 55).

[37] Robert Endre Tarjan e Uzi Vishkin. “An efficient parallel biconnectivity algorithm”. *SIAM Journal on Computing* 14.4 (1985), pp. 862–874 (citado na pg. 7).

[38] Mikkel Thorup. “Near-optimal fully-dynamic graph connectivity”. In: *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*. STOC ’00. Portland, Oregon, USA: Association for Computing Machinery, 2000, pp. 343–350. isbn: 1581131844. doi: 10.1145/335305.335345 (citado nas pgs. 4, 5).

[39] *Treap*. https://en.wikipedia.org/wiki/Treap. Acessado em: 2024-09-28 (citado na pg. 18).

[40] Jean Vuillemin. "A unifying look at data structures". *Communications of the ACM* 23.4 (1980), pp. 229–239 (citado na pg. 17).

[41] Christian Wulff-Nilsen. "Faster deterministic fully-dynamic graph connectivity". In: *Encyclopedia of Algorithms*. Ed. por Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 738–741. isbn: 978-1-4939-2864-4. doi: 10.1007/978-1-4939-2864-4_569 (citado nas pgs. 4, 5).

[42] Christos D. Zaroliagis. *Implementations and Experimental Studies of Dynamic Graph Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 229–278. isbn: 978-3-540-36383-5. doi: 10.1007/3-540-36383-1_11. url: https://doi.org/10.1007/3-540-36383-1_11 (citado na pg. 3).

Índice remissivo