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

Florestas geradoras maximais de custo mínimo em grafos dinâmicos

Chung Jin Shian

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Prof.ª Dr.ª Cristina Gomes Fernandes

São Paulo
2025

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)

Agradecimentos

"Nada é impossível para aquele que persiste."
— Alexandre, o Grande

Primeiramente gostaria de agradecer aos meus amigos do IME durante toda essa jornada. O contato com cada um e a troca de experiências contribuiu significativamente para o meu desenvolvimento técnico e profissional durante o curso.

Gostaria de agradecer à minha orientadora Cristina Gomes Fernandes por ter me apresentado ao tema desse trabalho, e ter sempre sido muito atenciosa e dedicada para garantir que eu estava de fato aprendendo os conceitos do trabalho. Foi ela quem fez crescer o meu interesse por Teoria da Computação, principalmente em algoritmos e estrutura de dados.

Além da Cristina, gostaria de agradecer aos professores Carlos Eduardo Ferreira e Marcel Kenji de Carli Silva, que fomentaram o meu interesse por algoritmos em grafos, e cujos ensinamentos contribuíram para a produção deste trabalho.

Por fim, quero agradecer à minha família, que me deu bastante suporte para eu me dedicar aos estudos nos primeiros anos da graduação.

Resumo

Chung Jin Shian. **Florestas geradoras maximais de custo mínimo em grafos dinâmicos**. Monografia (Bacharelado). Instituto de Matemática, Estatística e Ciência da Computação, Universidade de São Paulo, São Paulo, 2025.

Grafos dinâmicos permitem modelar problemas em que o grafo sofre alterações ao longo do tempo. Um dos problemas fundamentais nesse contexto é a manutenção de uma árvore geradora de custo mínimo no decorrer de várias alterações no grafo. Neste trabalho, estudamos e implementamos vários algoritmos propostos por Holm, de Lichtenberg e Thorup [3] para variantes desse problema. O foco foi no algoritmo para manter uma floresta geradora maximal de custo mínimo (MSF) decremental, em que se dá suporte eficiente à remoção de arestas do grafo. Além disso, estudamos e apresentamos a ideia de um algoritmo que mantém uma floresta geradora maximal de custo mínimo (MSF) em um grafo dinâmico, em que se dá suporte eficiente à adição e remoção de arestas.

Palavras-chave: grafo dinâmico. floresta geradora maximal de custo mínimo. splay trees.

Abstract

Chung Jin Shian. **Minimum spanning forests in dynamic graphs**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Dynamic graphs allow us to model problems in which the graph changes over time. One of the fundamental problems in this context is maintaining a minimum spanning tree of the dynamic graph as it undergoes multiple updates. In this work, we study and implement several algorithms proposed by Holm, de Lichtenberg, and Thorup [3] for variants of this problem. Our main focus is the algorithm for maintaining a decremental minimum spanning forest, which efficiently supports edge deletions in the graph. In addition, we study and outline the approach for maintaining a fully dynamic minimum spanning forest, which efficiently supports both edge insertions and deletions in the graph.

Keywords: dynamic graph. minimum spanning forest. splay trees.

Sumário

Capítulo 1

Introdução

Grafos são estruturas de dados que nos permitem modelar vários problemas existentes da vida real, sejam eles estáticos ou dinâmicos. Em problemas estáticos, o grafo não sofre alterações com o passar do tempo. Podemos citar, como exemplo, o planejamento de rotas de entrega, análise de moléculas químicas e de dependências em software utilizando ordenação topológica. Entretanto, ainda existem muitas situações em que ocorre dinamicidade, como nas interações de usuários em redes sociais, monitoramento de epidemias (contatos e isolamentos) e sistemas de navegação *GPS*, onde há necessidade de recalcular rotas dependendo de condições como congestionamentos e acidentes. Para modelar tais problemas, podemos usar grafos dinâmicos.

Dessa forma, são considerados problemas em grafos completamente dinâmicos aqueles em que o grafo sofre, com o tempo, alterações como inserções e remoções de arestas. As invariantes em que se permite apenas inserção ou apenas remoção de arestas, tais problemas são chamados de parcialmente dinâmicas, conforme Holm, de Lichtenberg e Thorup [3]. Note que as operações de atualização e consulta são apresentadas de forma online, sem conhecimento algum das operações futuras.

Aqui serão tratados problemas em que o grafo dinâmico possui um conjunto fixo de vértices V, e estabelecemos $n = |V|$. Além disso, pode-se definir m como o número de arestas existentes. Na maior parte das vezes, a complexidade de tempo das operações será amortizada, o que implica que elas são calculadas como a média sobre todas as operações realizadas.

Um grafo dinâmico de ordem n é uma sequência de grafos $(G_0, G_1, ..., G_T)$, onde G_0 é o grafo inicial com n vértices e cada G_t para $1 \leq t \leq T$ é obtido a partir de G_{t-1} pela adição ou remoção de alguma aresta. Chamamos de **alteração**, **modificação** ou **atualização** quando ocorre alguma operação de adição e/ou remoção de arestas no grafo dinâmico.

Um problema em grafos dinâmicos consiste em verificar se o grafo atual G satisfaz alguma propriedade, e cada operação que realiza essa verificação é denominada **consulta**. A solução do problema depende da criação de um algoritmo que utiliza uma estrutura de dados capaz de realizar estas consultas e as alterações de forma eficiente.

Iremos tratar inicialmente do **problema de conexidade em grafos dinâmicos**, que consiste em manter um grafo dinâmico que sofre uma sequência de inserções e remoções de arestas. Entre essas modificações, realizamos consultas para verificar se dois vértices u e v estão conectados por algum caminho. Porém, antes de entrarmos em detalhes, iremos apresentar alguns conceitos importantes que constituirão a base do nosso problema.

Uma **floresta** em um grafo G é um subgrafo acíclico de G. Uma **árvore** é uma floresta conexa, ou seja, uma floresta pode consistir de várias árvores. Um subgrafo F de G é **gerador** se contém todos os vértices de G. Com isso, podemos enunciar um problema clássico em grafos chamado o **problema da árvore geradora de custo mínimo** (MST, de *Minimum Spanning Tree*). Seja $G = (V, E)$ um grafo conexo, onde V é o conjunto de vértices e E o conjunto de arestas. Para cada aresta $uv \in E$, temos um peso $w(uv)$ associado. Assim, o objetivo do problema é encontrar uma árvore geradora cujo peso total

$$w(T) = \sum_{e \in E(T)} w(e)$$

seja mínimo. A Figura 1.1 mostra um exemplo de grafo conexo com pesos nas arestas e uma árvore geradora mínima.

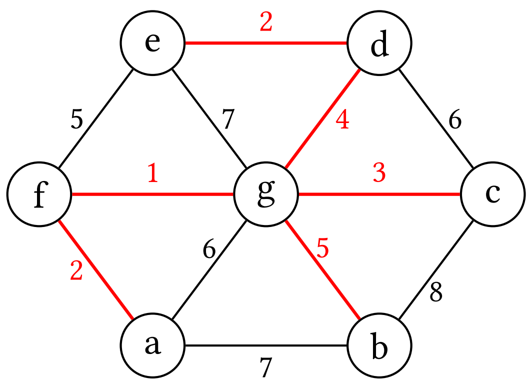

Figura 1.1: *Um grafo com sete vértices. As arestas em vermelho formam uma árvore geradora mínima (MST) com peso total 17.*

Para encontrar uma árvore geradora mínima, podemos utilizar uma abordagem gulosa para o problema. Existem dois algoritmos gulosos clássicos que resolvem este problema eficientemente, quando o grafo não é dinâmico: o algoritmo de Kruskal e o de Prim. Cada um deles estabelece uma regra específica para escolher uma aresta segura.

Se um grafo G tem n vértices e m arestas, o algoritmo de Kruskal consome $O(m \lg n)$, que é a complexidade de tempo para ordenar as arestas em ordem crescente de peso. Já o algoritmo de Prim pode ser implementado de modo a consumir $O(m + n \lg n)$, usando Fibonacci heaps. Tais algoritmos e suas implementações estão bem descritos no Capítulo 23.2 do livro de Cormen, Leiserson, Rivest e Stein [1] e, como não são o foco do nosso estudo, não iremos descrevê-los nesse estudo.

No nosso estudo, estamos interessados em grafos dinâmicos, que podem sofrer inserções e remoções de arestas, e por isso abrimos mão de exigir que o grafo seja conexo. Queremos manter uma floresta geradora maximal (MSF) de custo mínimo do grafo. Já existe uma solução para esse problema proposta por Holm, de Lichtenberg e Thorup [3] na Seção 5 do seu artigo. Essa solução se baseia num algoritmo para o problema de conexidade em grafos dinâmicos, descrito na Seção 3 de seu artigo.

No decorrer do nosso estudo, destrincharemos a solução dos autores em vários capítulos do texto. No Capítulo 2 descreveremos em detalhes o problema da conexidade em grafos dinâmicos, onde mostramos os pseudocódigos e a complexidade de tempo de cada um, e como as invariantes são mantidas no decorrer de sua execução.

No Capítulo 3, descreveremos a nossa implementação do algoritmo para o problema decremental da floresta geradora maximal de custo mínimo, que chamamos de **MSF decremental**. Como este algoritmo é baseado em vários métodos do problema de conexidade em grafos dinâmicos, realizamos alguns ajustes nos métodos deste último para adaptarmos ao contexto decremental.

No Capítulo 4, por fim, descreveremos a ideia por trás do algoritmo para o problema dinâmico da floresta geradora maximal de custo mínimo, que chamamos de **MSF dinâmica**. Este algoritmo, que dá suporte a adições e remoções de arestas, utiliza várias estruturas decrementais em sua implementação.

As implementações dos nossos algoritmos da solução de Holm, de Lichtenberg e Thorup [3] foram feitas utilizando a linguagem *C++*, e disponibilizamos o código no repositório do *GitHub* [6].

Capítulo 2

Conexidade em grafos dinâmicos

Como citado no Capítulo 1, o problema da conexidade em grafos dinâmicos visa construir um algoritmo eficiente que dê suporte a inserções e remoções de arestas e consultas de conexidade entre dois vértices. O algoritmo de Holm, de Lichtenberg e Thorup [3] para este problema de conexidade mantém $\lceil \lg n \rceil$ florestas dinâmicas do grafo G, e utiliza uma biblioteca que será descrita na próxima seção.

2.1 Conexidade em florestas dinâmicas

Rodrigues [5], em sua dissertação do mestrado, estudou, entre outros assuntos, o problema da conexidade em florestas dinâmicas e implementou o algoritmo que foi proposto na Seção 2 do artigo de Holm, de Lichtenberg e Thorup [3]. No Capítulo 2 da sua dissertação, Rodrigues descreve as rotinas principais de sua implementação, que se baseia em Euler tour trees, e realiza uma análise minuciosa da complexidade de tempo de cada rotina de sua implementação. Levando isso em conta, optamos por não apresentar uma descrição detalhada desse mesmo algoritmo, e apenas explicar brevemente o que as rotinas principais fazem, ressaltando algumas diferenças da nossa implementação em código em relação à de Rodrigues.

O problema da conexidade em florestas dinâmicas pode ser considerado uma simplificação do problema de conexidade em grafos dinâmicos, quando o grafo em questão é uma floresta. A biblioteca que usaremos contém os seguintes métodos:

- **`florestaDinâmica(`**n**`)`**: constrói e devolve uma floresta dinâmica F com n vértices e sem arestas;
- **`conectadosFD(`**F**`,`** u**`,`** v**`)`**: devolve verdadeiro se u e v estão na mesma componente da floresta F e falso caso contrário;
- **`adicioneFD(`**F**`,`** u**`,`** v**`)`**: insere uma aresta uv na floresta F;
- **`removaFD(`**u**`,`** v**`)`**: remove a aresta uv da floresta F.

A estrutura de dados principal usada neste algoritmo de Holm, de Lichtenberg e Thorup para dar suporte eficiente às rotinas acima é uma árvore binária de busca balanceada (ABBB). Uma floresta dinâmica é constituída de várias ABBBs. Rodrigues utiliza *treaps* em sua implementação, que são de natureza aleatória. Em nosso caso, utilizamos árvores `splay`, que foram desenvolvidas por *Sleator e Tarjan* [7]. Árvores `splay` são árvores binárias de busca (ABBs) que possuem uma rotina extra (além das usuais de busca, inserção e remoção) chamada `splay`, que é acionada ao final de cada operação feita na árvore, de modo que é sempre aplicada ao nó mais profundo visitado. O nó em que a operação `splay` é aplicada é trazido, por meio das tradicionais rotações usadas no balanceamento de árvores binárias de busca, para cima até chegar na raiz da árvore. Isso faz com que o custo de uma sequência de m operações (inserção, remoção ou busca) em uma árvore `splay` com n nós seja $O(m \lg n)$, ou seja, o custo amortizado por operação é $O(\lg n)$. Como também já existe bastante literatura sobre árvores `splay` [4, Lecture 12], e seu funcionamento interno afeta muito pouco a descrição dos algoritmos que descreveremos, não entraremos em detalhes de sua implementação.

O resultado é uma implementação em que `florestaDinâmica(n)` tem custo $\Theta(n)$ e os demais métodos da biblioteca têm custo amortizado $O(\lg n)$.

2.2 Biblioteca do grafo dinâmico

Implementar o grafo dinâmico resume-se à construção da seguinte biblioteca de forma eficiente:

- **`grafoDinâmico(`**n**`)`**: contrói e devolve um grafo dinâmico com n vértices e sem arestas;
- **`conectadosGD(`**G**`,`** u**`,`** v**`)`**: devolve verdadeiro se os vértices u e v estão na mesma componente de G e falso caso contrário;
- **`adicioneGD(`**G**`,`** u**`,`** v**`)`**: adiciona a aresta uv no grafo G;
- **`removaGD(`**G**`,`** u**`,`** v**`)`**: remove a aresta uv do grafo G.

Para entender como cada uma dessas rotinas funcionam, será necessário apresentar a estrutura interna da implementação do grafo para explicar como ele mantém essas estruturas e como elas deixam essas rotinas mais eficientes.

2.2.1 Fatiamento do grafo em níveis

Na Seção 3.1 do artigo de Holm, de Lichtenberg e Thorup [3], é apresentada a técnica de fatiar o grafo G em níveis. Cada aresta do grafo possui um nível entre 1 e $\lceil \lg n \rceil$, onde n é o número de vértices do grafo G. Toda vez que inserimos uma aresta em G, ela possuirá o nível $\lceil \lg n \rceil$, e ele nunca será aumentado.

Seja G um grafo com conjunto $V(G)$ de vértices e conjunto $E(G)$ de arestas. Se X é um conjunto não-vazio de arestas, dizemos que o subgrafo de G **induzido** por X é o subgrafo gerador H de G tal que $E(H) = X$. Denotamos, H como $G[X]$.

Sendo assim, seja $G_i = G[X]$ o grafo onde X é o conjunto das arestas do grafo G de nível menor ou igual a i. Para cada nível i, manteremos uma floresta maximal F_i de G_i. Além disso, vamos manter também, para cada nível i, um grafo R_i em forma de listas de adjacências, que guardam apenas arestas de nível i que não estejam em F_i.

Consequentemente, temos que $G_1 \subseteq G_2 \subseteq \cdots \subseteq G_{\lceil \lg n \rceil}$, para um grafo G de n vértices, e deduzimos que $G = G_{\lceil \lg n \rceil}$ e que $F_{\lceil \lg n \rceil}$ é uma floresta maximal de G. Dessa maneira, sempre que estivermos realizando alguma operação de consulta de conexidade em nosso grafo G, podemos realizá-la na floresta $F_{\lceil \lg n \rceil}$ de G.

Com isso, podemos estabelecer algumas invariantes, que são mantidas ao longo das modificações no grafo G:

(I) F_i é uma floresta maximal de G_i para todo $1 \leq i \leq \lceil \lg n \rceil$;

(II) $F_i \subseteq F_{i+1}$ para todo $1 \leq i \leq \lceil \lg n \rceil - 1$;

(III) Cada componente da floresta F_i possui no máximo 2^i vértices.

Na Seção 2.3, descreveremos com mais detalhes como as rotinas da biblioteca funcionam e como cada uma preserva as invariantes acima, de modo a manter a corretude do algoritmo durante toda a sua execução. Por fim, para simplificar um pouco a notação, escreveremos $L = \lceil \lg n \rceil$, isto é, $F_{\lceil \lg n \rceil}$ passa a ser escrita como F_L, da mesma forma que $G_L = G_{\lceil \lg n \rceil}$ e $R_L = R_{\lceil \lg n \rceil}$.

2.2.2 Tipos de arestas do grafo

Quando realizamos uma chamada à função `adicioneGD(`G`, `u`, `v`)`, é feita uma chamada à rotina `conectadosGD(`G`, `u`, `v`)` para verificar a conexidade de u com v em G. Se estes vértices não estiverem na mesma componente de G, então a aresta uv é inserida na floresta maximal F_L que estamos mantendo, assim ligando a árvore que contém u com a que contém v em F_L. Chamamos esse tipo de aresta de **aresta da floresta**.

Caso u e v já estejam conectados em G, então essa aresta uv é chamada de **aresta reserva** e ela será armazenada no grafo R_L representado por listas de adjacências, que contém a seguinte biblioteca:

- **`listasDeAdjacências(`n`)`**: constrói e devolve um grafo com n vértices e sem arestas, representado por listas de adjacências;
- **`adicioneLA(`R`, `u`, `v`)`**: adiciona u na lista de adjacências de v em R e vice-versa;
- **`removaLA(`R`, `u`, `v`)`**: remove u da lista de adjacências de v em R e vice-versa.

A nossa implementação [6] de listas de adjacências possui um custo $O(n)$ ao acionar o construtor `listasDeAdjacências`, e para as rotinas `adicioneLA` e `removaLA` é garantido tempo esperado $O(1)$, visto que estamos utilizando um mapa hash da linguagem *C++* para realizar adição de um vizinho v na lista de u e remoção de v da lista de adjacências de u.

2.3 Rotinas da biblioteca do grafo dinâmico

2.3.1 Criação do grafo

Para criar um grafo dinâmico G com n vértices e inicialmente sem arestas, acionamos a rotina `grafoDinâmico(`n`)`. Nesta chamada, armazenamos o nível máximo do grafo, no caso o valor de $\lceil \lg n \rceil$, numa variável do grafo chamada *nívelMax*, onde podemos extrair e usar o valor do nível máximo chamando *G.nívelMax*. Em seguida, chamamos `florestaDinâmica`(n), que criará $\lceil \lg n \rceil$ florestas dinâmicas com n vértices, e `listasDeAdjacências`(n), que criará $\lceil \lg n \rceil$ grafos com n vértices representados por listas de adjacências.

Além disso, usamos um mapa hash para guardar e obter o nível de uma aresta uv em tempo esperado O(1). Assim, este mapa usa como chave as pontas da aresta (u e v) e armazena o valor do nível da aresta uv. Assim, podemos definir um outro método `novoMapaHash`(n) que devolve um mapa hash vazio em tempo O(1) para um grafo de n vértices. Dessa forma, se chamarmos esse método e atribuirmos o objeto devolvido a uma variável chamada *nível*, podemos realizar as seguintes operações com *nível*:

- *nível[u, v]* $\leftarrow i$: armazena i como o nível da aresta uv.
- *nível[u, v]* $\leftarrow$ `NIL`: remove a aresta uv do mapa hash.
- $x \leftarrow$ *nível[u, v]*: atribui o valor do nível da aresta uv à variável x.

Dessa forma, podemos apresentar o construtor do grafo no Programa 2.1.

Programa 2.1 `grafoDinâmico`(n)

Entrada: Recebe o número n de vértices do grafo.
Saída: Devolve um grafo dinâmico G com n vértices e sem arestas.

$L \leftarrow \lceil \lg n \rceil$
$G.nívelMax \leftarrow L$
para $i \leftarrow 1$ **até** L **faça**
    $G.F_i \leftarrow$ `florestaDinâmica`(n)
    $G.R_i \leftarrow$ `listasDeAdjacências`(n)
$G.nível \leftarrow$ `novoMapaHash`(n)
retorne G

Como ambos `florestaDinâmica`(n) e `listasDeAdjacências`(n) consomem tempo O(n), então `grafoDinâmico`(n) consome tempo O($n \lg n$), pois estamos criando $\lceil \lg n \rceil$ florestas dinâmicas e listas de adjacências. Além disso, é fácil de ver que, ao final, as três invariantes valem.

2.3.2 Consultas de conexidade

Para testar a conexidade entre dois vértices u e v no grafo G, basta chamarmos `conectadosGD`(G, u, v), que por sua vez aciona `conectadosFD`(F_L, u, v), pois F_L é uma floresta maximal de G pelo invariante (I). O Programa 2.2 mostra essa rotina.

Programa 2.2 `conectadosGD`(G, u, v)

Entrada: Recebe dois vértices u e v do grafo G.
Saída: Devolve um booleano indicando se u e v estão conectados em G.

$L \leftarrow G.nívelMax$
retorne `conectadosFD`($G.F_L$, u, v)

A rotina `conectadosFD(`F_L`, `u`, `v`)` em nossa implementação consome tempo amortizado O(lg n), e, portanto, `conectadosGD(`F_L`, `u`, `v`)` também terá o mesmo consumo de tempo amortizado. Além disso, a rotina de consulta não altera o nosso grafo, incluindo florestas e listas de adjacências, já que não há nenhuma modificação neles. Isso implica que as invariantes são mantidas.

2.3.3 Inserções de arestas

Como explicado na Seção 2.2.2, inserimos uma aresta uv no grafo G chamando a rotina `adicioneGD(`G`, `u`, `v`)` e, em seguida, testamos a conexidade de u e v chamando a rotina `conectadosGD(`G`, `u`, `v`)`, e, dependendo do resultado, uv pode ser inserida como aresta reserva ou como aresta da floresta. Assumindo que a aresta uv não exista em G no momento de sua inserção, temos dois cenários:

- Se os vértices u e v já estão conectados em G, então chamaremos a função `adicioneLA(`R_L`, `u`, `v`)`, que armazenará uv como aresta reserva de nível L de G em R_L. A Figura 2.1 ilustra esse cenário.

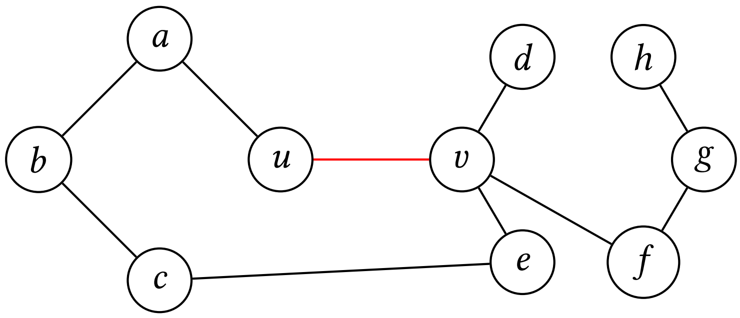

Figura 2.1: *As arestas pretas são da floresta maximal F_L do grafo. Como queremos inserir a aresta uv e os vértices u e v já estão conectados pelo caminho $u \to a \to b \to c \to e \to v$, então armazenamos uv como aresta reserva (em vermelho).*

- Se u e v não estão conectados em G, então inserimos uv como aresta da floresta F_L de nível L, chamando a função `adicioneFD(`F_L`, `u`, `v`)`. A Figura 2.2 ilustra esse cenário.

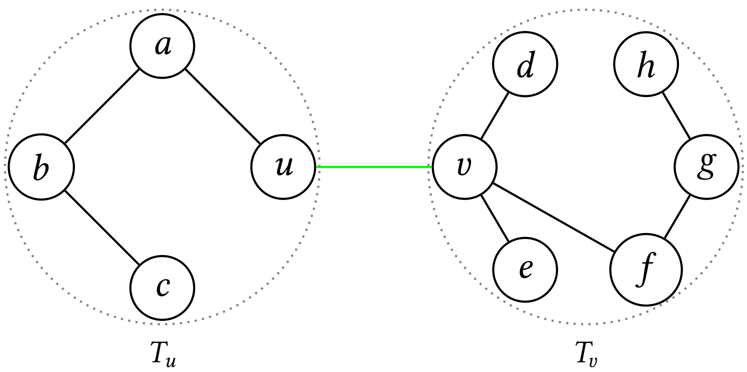

Figura 2.2: *As arestas pretas são da floresta maximal F_L do grafo. Como os vértices u e v não estão conectados em F_L, então inserimos uv como aresta da floresta (em verde), nesse caso conectando as componentes T_u e T_v de F_L.*

O Programa 2.3 ilustra uma primeira versão da rotina `adicioneGD`. Posteriormente, faremos alguns ajustes nessa rotina por causa da rotina de remoção de arestas explicada na Seção 2.3.4.

Programa 2.3 `adicioneGD`(G, u, v)

Entrada: Recebe dois vértices u e v do grafo G.
Efeito: Adiciona a aresta uv no grafo G.

$L \leftarrow G.nívelMax$
$G.nível[u, v] \leftarrow L$
se `conectadosFD`($G.F_L$, u, v) **então**
    `adicioneLA`($G.R_L$, u, v)
senão
    `adicioneFD`($G.F_L$, u, v)

Em nossa implementação, `conectadosFD` consome tempo amortizado $O(\lg n)$. A rotina `adicioneLA`, quando acionada, consome tempo esperado constante $O(1)$ em nossa implementação por conta do mapa hash. Já inserir uma aresta da floresta chamando `adicioneFD` consome tempo $O(\lg n)$ (amortizado, em nossa implementação). Portanto, a rotina `adicioneGD` tem custo de tempo amortizado $O(\lg n)$.

A invariante (I) se mantém para o nível $i = \lceil \lg n \rceil$. Como sempre adicionamos arestas com nível $\lceil \lg n \rceil$ em $F_{\lceil \lg n \rceil}$ (se não forem reservas), então as outras florestas de níveis inferiores não são afetadas, mantendo-se, assim, os invariantes (II) e (III) também.

2.3.4 Remoção de arestas

A remoção de arestas se divide em dois casos: remoção de uma aresta reserva ou remoção de uma aresta da floresta.

Quando queremos remover uma aresta uv e ela é reserva, podemos simplesmente acionar a rotina `removaLA`(R_i, u, v) onde i, obtido do mapa hash, é o nível da aresta uv e R_i é a lista de adjacências na qual uv está armazenada. Por conta disso, nenhuma das florestas maximais F_j do grafo será afetada, e as três invariantes serão mantidas. A Figura 2.3 mostra esse cenário.

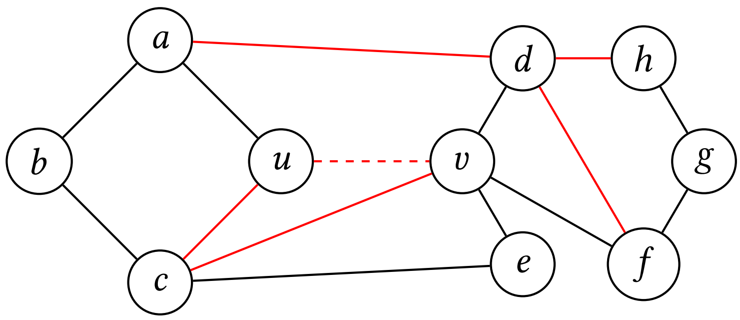

Figura 2.3: *As arestas pretas são da floresta maximal F_L do grafo, enquanto as vermelhas são arestas reserva. A aresta vermelha uv tracejada é reserva e está prestes a ser removida.*

O caso da remoção de uma aresta uv da floresta é mais complexo. Se a aresta uv tem nível i, remover uv sempre quebra uma componente de F_i em duas árvores T_u e T_v, de modo que a primeira contém o vértice u e a segunda contém o vértice v. Neste caso, precisamos verificar se existe alguma aresta reserva que ligue T_u a T_v, para que possamos garantir que a floresta F_i continue maximal em G_i. Chamamos uma tal aresta de **aresta substituta**.

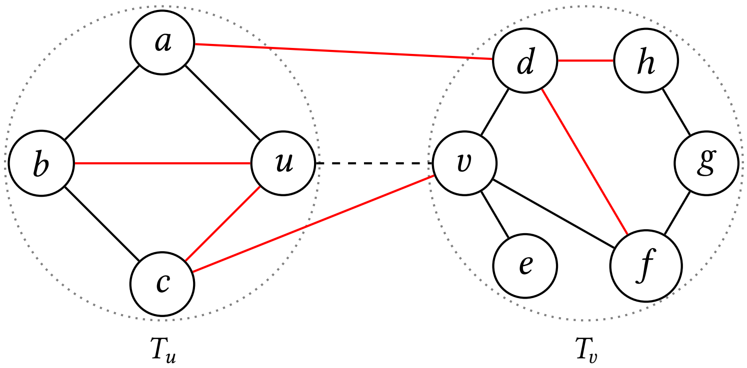

Figura 2.4: *As arestas pretas são da floresta maximal F_i do grafo G_i, enquanto as vermelhas são reservas de nível i. A aresta uv tracejada de nível i está prestes a ser removida, assim ela pode ser substituída por ad ou cv, pois qualquer uma destas liga T_u a T_v.*

Para buscar uma aresta reserva de maneira o mais eficiente possível, o algoritmo percorre cada vértice x de T_u e verifica se existe algum vértice y na lista de adjacências de x de R_i que esteja em T_v. Se $y \in V(T_v)$, então a aresta xy é uma aresta substituta de nível i, bastando apenas acionar `adicioneFD(`F_i`, x, y)` para reconectar T_u e T_v, que virariam uma única componente da floresta F_i, e acionar `removaLA(`R_i`, x, y)` já que xy se tornará uma aresta da floresta.

É por este motivo que introduzimos o fatiamento em níveis na Seção 2.2.1. A intuição por trás deste fatiamento é que, quando uma aresta de nível i da floresta é removida, não é necessário buscar por substitutas nos níveis menores que i. Isso quer dizer que começamos a busca no nível em questão, ou seja, em R_i, e caso não haja nenhuma substituta em R_i, passamos a procurar em $R_{i+1}, R_{i+2}, \ldots, R_L$. Quando não encontramos uma substituta em um certo R_i, aproveitamos para rebaixar o nível de todas as arestas percorridas em R_i para $i-1$, de modo que não precisaremos mais percorrer essas arestas quando removermos uma outra aresta de nível i, visto que elas já estariam em R_{i-1}.

Na verdade, antes de fazer esse rebaixamento, rebaixamos o nível de toda aresta de nível i de T_u para $i-1$, de modo que $T_u \subseteq F_{i-1}$. Rebaixar o nível dessas arestas significa inseri-las em F_{i-1}, pois elas passam a ser de nível $i-1$. Esse rebaixamento e as inserções em F_{i-1} se tornam necessários para preservar a invariante (I). Ao mesmo tempo, para manter também a invariante (III), esse processo deverá ser feito na menor das árvores T_u e T_v. Seja $T = T_u \cup T_v + uv$. Denotando o número de vértices de uma árvore T por $|T|$, o algoritmo garantirá que $|T_u| \leq |T_v|$. Pela invariante (III), temos que $|T| \leq 2^i$, e como $|T_u| + |T_v| = |T|$, então $|T_u| \leq 2^{i-1}$. Por isso, ao rebaixarmos todas as arestas de nível i de T_u para o nível $i-1$, preservamos a invariante (III).

Ao remover uma aresta de nível i da floresta, na verdade temos que removê-la não só de F_i, mas também de $F_{i+1}, \ldots, F_L$ pela invariante (II). Similarmente, quando encontramos uma aresta substituta de nível i, temos que acrescentá-la não só a F_i, mas também a $F_{i+1}, \ldots, F_L$ para manter as invariantes (I) e (II). A invariante (III) neste caso é mantida trivialmente.

Agora, veremos em detalhes o motivo de não precisarmos procurar uma substituta em níveis menores que i quando removemos uma aresta uv de nível i. Veja que, como a aresta uv tem nível i, ela não pertence a F_{i-1}. Logo, pela invariante (II), u e v estão em componentes distintas de F_{i-1}. Como F_{i-1} é maximal em G_{i-1}, não existe aresta reserva de nível $\leq i-1$ que conecte as componentes T_u e T_v. Portanto, só procuramos uma substituta em níveis $\geq i$. A Figura 2.5 torna a explicação mais intuitiva.

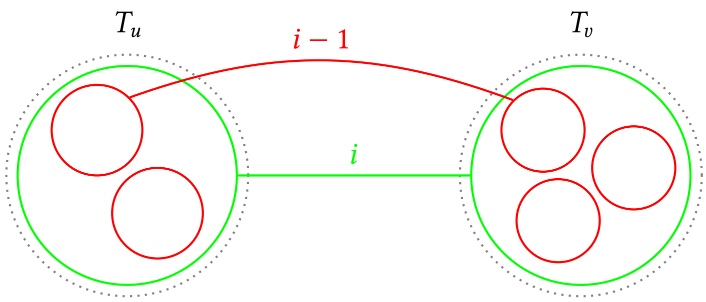

Figura 2.5: *Os círculos verdes representam as componentes T_u e T_v da floresta F_i do grafo G_i, enquanto os círculos vermelhos representam as componentes da floresta F_{i-1} do grafo G_{i-1}. Note que a aresta reserva de nível $i-1$ mostrada não deveria existir, porque senão ela violaria a invariante (I). Portanto, tais arestas de nível $i-1$ ligando componentes de F_i (como a aresta vermelha) não existem e só é necessário procurar arestas substitutas a partir de nível i (como a aresta verde) para conectar T_u e T_v.*

A seguir, demonstraremos a remoção de uma aresta uv da floresta em uma série de imagens. Na Figura 2.6, temos um grafo G de $n = 10$ vértices. Para facilitar, assumiremos que, até o momento, só houve inserções de arestas.

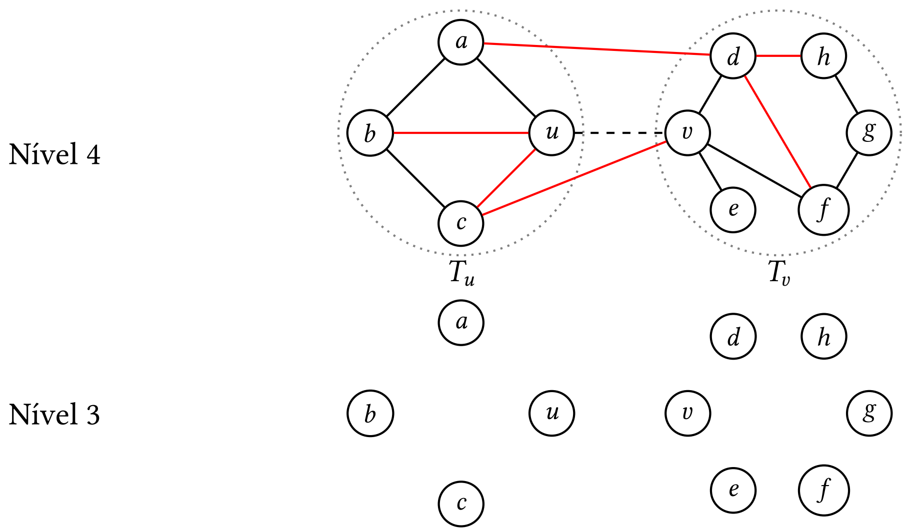

Figura 2.6: *Um grafo G de 10 vértices, onde as arestas pretas são da floresta F_4, enquanto as vermelhas são reservas. A aresta uv está prestes a ser removida. A floresta F_4 de G de cima contém todas as arestas pretas recém-inseridas e as arestas vermelhas estão em R_4. A floresta de baixo é a F_3, com os vértices isolados, e R_3 também não tem nenhuma aresta.*

Temos também que $n = 10$ e $\lceil \lg 10 \rceil = 4$, logo o nível máximo L da floresta é 4 e, consequentemente, $G = G_4$. Como todas as inserções só acontecem no nível L, no momento, em F_4, só temos arestas da floresta de nível 4, enquanto F_3 contém apenas vértices isolados. Neste cenário, note que a remoção da aresta uv da floresta, representada por uma linha tracejada na figura, acaba quebrando a única componente da floresta F_4 em duas, T_u e T_v. Como F_4 é a floresta maximal de nível máximo de G, então removemos somente a uv de F_4.

O próximo passo é rebaixar todas as arestas de nível 4 em T_u para o nível 3. Dessa forma, as arestas de T_u passam a estar em F_3, como se pode ver na Figura 2.7, pois agora elas passam a ser de nível 3.

Perceba que, devido à invariante (II), podemos ter arestas de diferentes níveis em uma mesma floresta. Assim, percorrer todas as arestas de T_u e selecionar apenas as de nível i para rebaixar pode se tornar demorado quando o grafo possui uma grande quantidade de vértices. A forma como o algoritmo procura as arestas de nível i de T_u será descrita de maneira mais detalhada na Seção 2.4.3. No momento, só precisamos entender como este rebaixamento funciona.

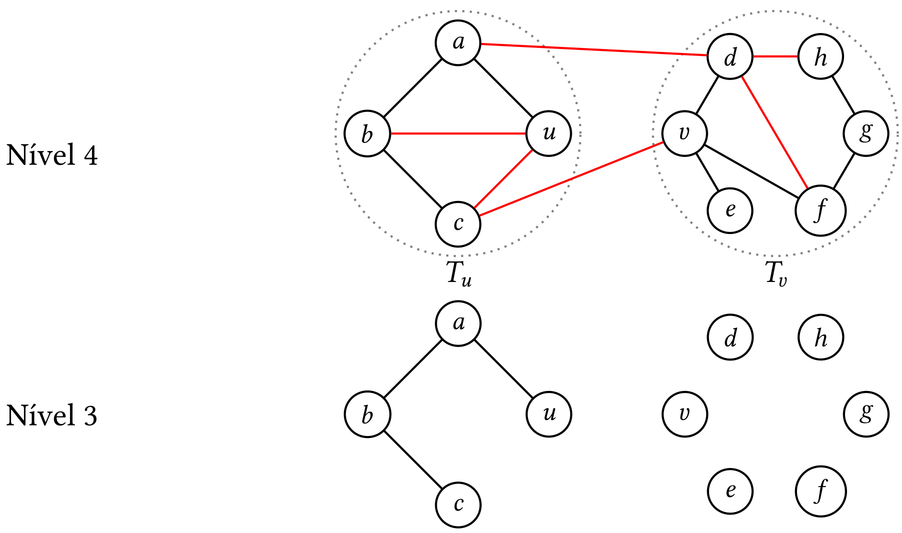

Figura 2.7: *Representação da remoção da aresta uv em G. As arestas de nível* 4 *de T_u foram rebaixadas para o nível* 3*, o que pode ser visto na floresta F_3.*

Como T_u e T_v em F_4 ficaram separadas após a remoção de uv, precisamos encontrar, se existir, uma aresta reserva que possa reconectá-las. Note que percorrer todas as arestas reserva para achar uma substituta que ligue T_u a T_v pode ser ineficiente quando temos muitas arestas reserva. Por isso, explicaremos como implementar essa busca por uma substituta de forma eficiente na Seção 2.4.4.

Na Figura 2.8, percorremos as arestas reserva em R_4 que tenham uma das pontas em T_u. Para cada aresta percorrida, verificamos se a outra ponta dela incide em algum vértice de T_v. Caso não incida, a aresta tem duas pontas em T_u, pois F_4 era maximal antes da remoção de uv, e logo a aresta não é uma substituta. Então a rebaixamos para o nível 3, ou seja, movemos de R_4 para R_3 as arestas percorridas que não são substitutas.

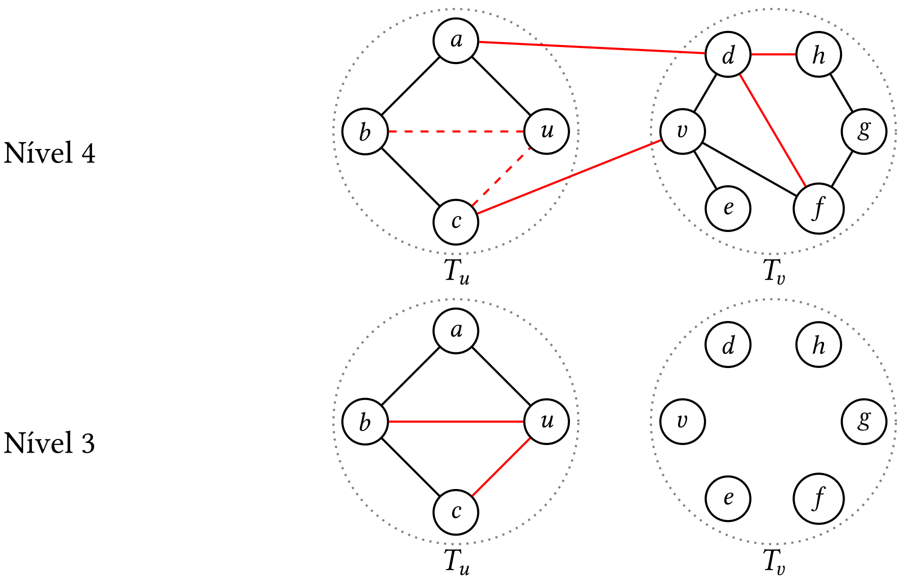

Figura 2.8: *Representação da busca por uma aresta substituta em R_4. As arestas reserva de nível* 4 *que estão tracejadas foram percorridas e estão prestes a ser removidas de R_4, pois foram rebaixadas para o nível* 3*, como se pode ver em R_3.*

Supondo que achamos a aresta ad como substituta de nível 4 antes de cv, conectamos T_u e T_v chamando `adicioneFD(`F_4`, a, d)` e ad passa a ser uma aresta da floresta, ou seja, é removida de R_4. Como $i = 4$ é o nível máximo do grafo nesse exemplo, não precisamos chamar esta rotina para os níveis superiores e então terminamos a execução do algoritmo. A Figura 2.9 ilustra essa etapa do algoritmo.

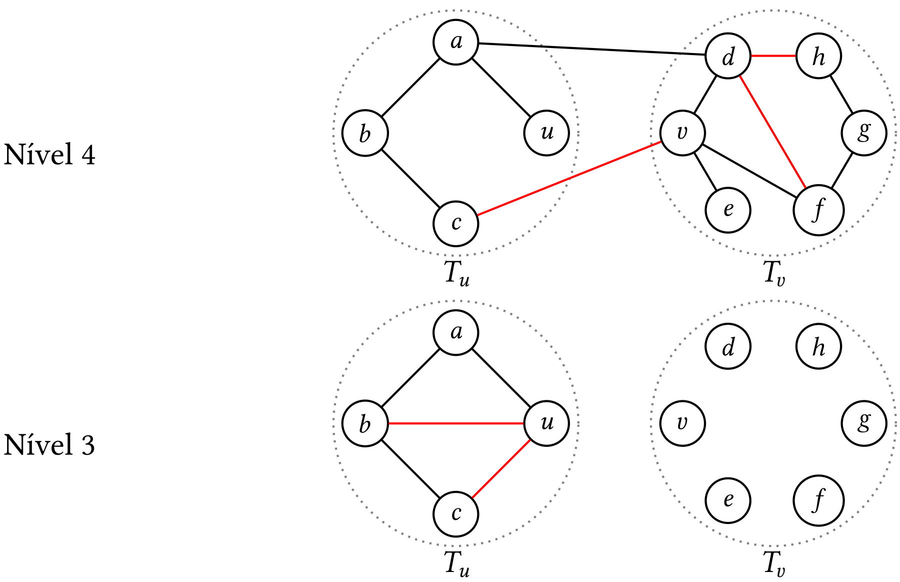

Figura 2.9: *Representação do grafo com a aresta substituta ad escolhida para conectar T_u a T_v, tornando-se uma aresta da floresta F_4.*

Quando removemos uma aresta da floresta de nível i que quebra uma componente da floresta F_i em duas, T_u e T_v, com $|T_u| \leq |T_v|$, note que ao procurar por uma aresta substituta não necessariamente percorreremos todas as arestas reserva em R_i incidentes a T_u. Então nem sempre todas as arestas reserva de R_i incidentes a T_u são rebaixadas, visto que o algoritmo será finalizado no momento em que encontrarmos uma substituta.

O Programa 2.4 apresenta uma primeira versão da rotina `removaGD`. Ela contém uma chamada à rotina `substituaAresta` que será explicada Seção 2.4.7. Alguns ajustes em `removaGD` serão necessários devido à implementação da rotina `substituaAresta`. Em particular, essa rotina deve percorrer as arestas da floresta e as arestas reserva de maneira eficiente. Apresentaremos a estratégia usada para isso nas Seções 2.4.3 e 2.4.4.

Programa 2.4 `removaGD`(G, u, v)

Entrada: Recebe dois vértices u e v adjacentes do grafo G.
Efeito: Remove a aresta uv do grafo G.

```
i ← G.nível[u,v]
nível[u, v] ← NIL                          ▷ marcamos uv como removida
L ← G.nívelMax
se uv ∈ G.F_L então                        ▷ uv é aresta da floresta
    para j ← i até L faça
        removaFD(G.F_j, u, v)              ▷ remove uv da floresta F_j
    substituaAresta(G, i, u, v)
senão                                      ▷ uv é aresta reserva
    removaLA(G.R_i, u, v)                  ▷ remove uv do grafo R_i
```

Note que `removaGD` consome $O(\lg^2 n)$ mais o tempo do `substituaAresta`, que será descrito mais adiante.

2.4 Estrutura interna do grafo dinâmico

Para explicar a rotina `substituaAresta`, precisamos saber mais detalhes sobre como as árvores de cada floresta F_i são armazenadas. Apresentaremos esses detalhes a seguir.

2.4.1 Euler tour trees

A Seção 2.1 do artigo de Holm, de Lichtenberg e Thorup [3] propõe o uso de Euler tour trees, que é uma técnica utilizada para representar uma árvore. 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 2.10.

O digrafo resultante de T é Euleriano, ou seja, é conexo e o grau de entrada de cada vértice é igual ao grau de saída. Consequentemente, há uma trilha que começa e termina num mesmo vértice, passando por todos os arcos do digrafo somente uma vez. Tal trilha é chamada de **ciclo Euleriano**.

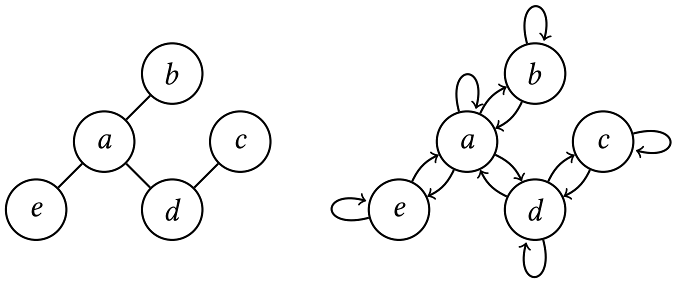

Figura 2.10: *À esquerda, temos uma árvore T e, à direita, temos o digrafo Euleriano de T.*

A representação da árvore T é basicamente a sequência de arcos que forma um ciclo Euleriano do digrafo correspondente a T. Denotamos cada arco pelo par de vértices que o compõe. Dessa forma, se o arco parte de u para v, ele será denotado como uv. Para o caso do laço em um vértice u, então o arco será escrito como uu. Assim, no exemplo da Figura 2.10, um possível ciclo Euleriano poderia ser o seguinte:

$$ee\ ea\ aa\ ab\ bb\ ba\ ad\ dd\ dc\ cc\ cd\ da\ ae. \tag{2.1}$$

A sequência dos arcos obtida de T depende do vértice inicial e da ordem em que os vizinhos de cada vértice são visitados. Uma tal sequência é chamada **sequência Euleriana** de T.

Henzinger e King [2] propuseram armazenar uma sequência Euleriana em uma árvore binária de busca balanceada, usando como chave a posição de cada elemento na sequência. Tomando como base o nosso exemplo da árvore da Figura 2.10 e sua sequência Euleriana dada em (2.1), podemos ilustrar uma possível árvore binária de busca balanceada para ela na Figura 2.11.

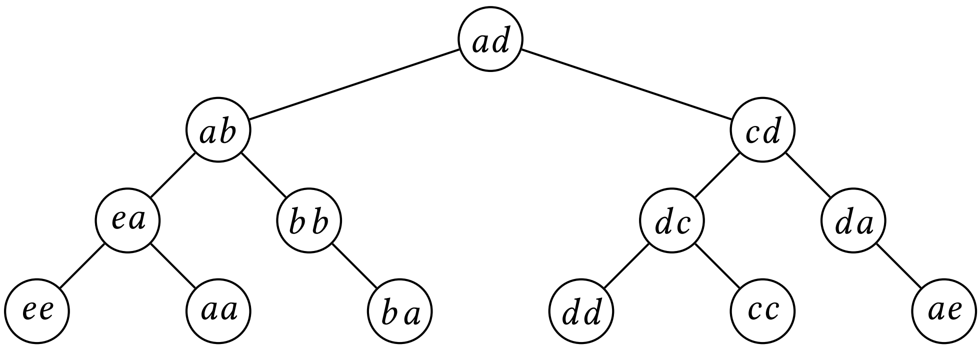

Figura 2.11: *Uma árvore binária de busca balanceada para um ciclo Euleriano da árvore da Figura 2.10. Note que se percorrermos os nós da árvore acima em inorder, obtemos a sequência Euleriana de (2.1).*

Além disso, Henzinger e King [2] propuseram representar uma floresta pela coleção de sequências Eulerianas de cada componente da floresta. Assim, é possível implementar as operações de consulta de conexidade e de alteração na floresta com consumo esperado de tempo O(lg n) (amortizado em nossa implementação), onde n é o número de nós da floresta. O algoritmo de Holm, de Lichtenberg e Thorup [3] armazena cada floresta F_i em uma estrutura dessas.

2.4.2 Nós das florestas

Os nós das árvores binárias de busca que representam as sequências Eulerianas que são mantidas pelo algoritmo serão chamados de **nós das florestas**. Cada tal nó pode representar um vértice u do grafo, se o elemento armazenado no nó for uu, ou pode representar uma aresta uv do grafo, se o elemento armazenado no nó for uv ou vu, com $u \neq v$. O primeiro tipo de nó é chamado de **nó de vértice** e o segundo, de **nó de aresta**. Em nossa implementação, para cada floresta F_i, usaremos um mapa hash *nó* que armazena, para cada par de vértices (u, v), um apontador para o nó do elemento uv na floresta F_i, se tal nó existe (ou `NIL` caso não exista).

Como representamos uma Euler tour tree por uma árvore binária de busca, cada nó p possui apontadores para o filho esquerdo, filho direito e seu pai. Na descrição de nossa implementação, denotamos tais apontadores por *p.esq*, *p.dir* e *p.pai*, respectivamente. O motivo de usar o apontador para o pai é por conta da operação `splay`.

Além disso, para extrairmos as pontas de um nó de aresta p que representa xy, usamos os campos *p.vértice1* e *p.vértice2*, que guardam os valores x e y, respectivamente. Em um nó de vértice q que representa xx, *q.vértice1* guarda o mesmo valor do *q.vértice2*, ou seja, ambos armazenam x. Podemos extrair ambas as pontas do nó p chamando $p.vértices$. Extrair as pontas dos nós será importante como veremos em vários métodos posteriormente.

A seguir, descreveremos a funcionalidade de cada tipo de nó, bem como outros atributos relevantes que lhe pertencem.

2.4.3 Nó de aresta

Como já observamos, na floresta F_i, há arestas de nível $\leq i$. Para percorrermos as arestas de nível i de uma componente de F_i eficientemente, os nós da floresta F_i têm um atributo extra booleano chamado *éNível*, que, em caso de um nó de aresta, indica se tal aresta da floresta F_i é de nível i.

Além disso, todos os nós da floresta armazenam um contador chamado *arestasDeNível*, com a quantidade de nós em sua subárvore que têm o atributo *éNível* verdadeiro. Sempre que modificarmos alguma das árvores binárias da floresta F_i, devemos manter este contador com o valor correto. Na nossa implementação, a atualização deste contador é feita na operação `splay` sempre que essa executa alguma rotação.

A rotina abaixo do Programa 2.5 é usada para alterar para b o valor do atributo *éNível* para uma aresta uv. Ela é acionada sempre que adicionamos uma aresta ao grafo, e também quando uma aresta é rebaixada. Tal rotina utiliza um método auxiliar chamado *atualizeArestasDeNível*, descrito no Programa 2.6.

Programa 2.5 `atualizeÉNível(F, u, v, b)`

Entrada: Recebe uma floresta F, pontas u e v de uma aresta de G, e um booleano b.
Efeito: Atualiza o atributo *éNível* do nó *uv* da floresta F e o contador *arestasDeNível*.

```
arestaUV ← F.nó[u, v]
splay(arestaUV)
arestaUV.éNível ← b
atualizeArestasDeNível(arestaUV)
```

Programa 2.6 `atualizeArestasDeNível(p)`

Entrada: Recebe um nó p.
Efeito: Atualiza o contador *arestasDeNível* de p.

```
c ← 0
se p.esq ≠ NIL então
    c ← c + p.esq.arestasDeNível
se p.dir ≠ NIL então
    c ← c + p.dir.arestasDeNível
se p.éNível então
    c ← c + 1
p.arestasDeNível ← c
```

Como se pode ver, o Programa 2.6 consome tempo $O(1)$. Já Programa 2.5 consome tempo amortizado $O(\lg n)$ por conta da operação `splay`. Veja que ambos os métodos não alteram a floresta F_i, alteram apenas uma das árvores binárias que a representam. Portanto, todas as três invariantes são preservadas.

Usando o mesmo exemplo da Figura 2.11 na Seção 2.4.1, podemos ilustrar como estaria o atributo *arestasDeNível* de cada nó na árvore. Na nossa implementação, os vértices são identificados por inteiros de 1 a n e, para uma aresta *uv*, usamos o atributo *éNível* apenas para o nó de *uv* com $u < v$.

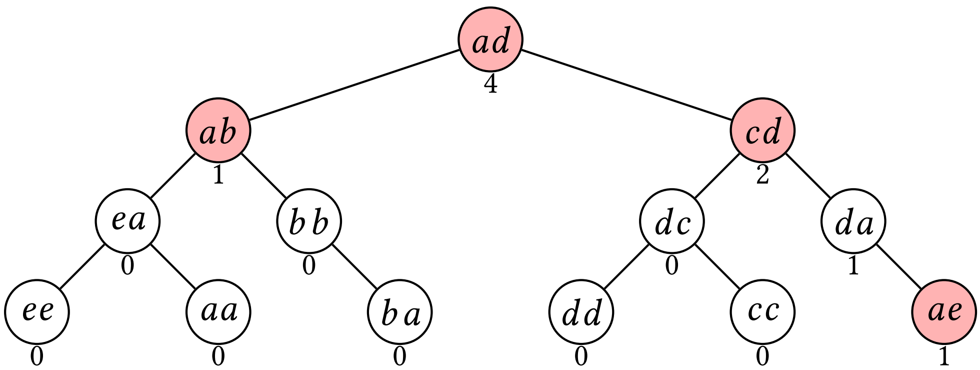

Figura 2.12: *Árvore de uma das componentes da floresta F_i, onde nós pintados em vermelho indicam arestas de nível i, e logo possuem o atributo éNível verdadeiro. Embaixo de cada nó temos o valor do contador arestasDeNível.*

Na Figura 2.12, note que os nós *ab* e *ba* representam a mesma aresta. Assim, para evitar a duplicação do atributo *éNível*, optamos por colocar este atributo como verdadeiro somente nos nós de aresta cujos vértices estão em ordem lexicográfica. Então, no nosso exemplo, nós do tipo *ba*, *da*, *dc* e *ea* vão ter este atributo falso.

Lembre-se que a remoção de uma aresta uv de nível i da floresta F_i quebra uma componente de F_i em duas, T_u e T_v. Sendo T_u a menor das duas, a rotina `substituaAresta` realiza o rebaixamento das arestas de nível i de T_u para $i-1$. Para fazer isso de forma eficiente, introduzimos um método auxiliar chamado `procureArestaDeNível`. Esse método utiliza o atributo *arestasDeNível* para encontrar um a um, numa árvore da floresta F_i, os nós de arestas de nível i.

Programa 2.7 `procureArestaDeNível(`p`)`

Entrada: Recebe um nó p de uma floresta com o contador *arestasDeNível* > 0.
Saída: Devolve um nó de aresta da subárvore do nó com *éNível* verdadeiro.

se *p.éNível* **então**
    retorne p
se *p.esq* $\neq$ `NIL` **e** *p.esq.arestasDeNível* > 0 **então**
    retorne `procureArestaDeNível(`*p.esq*`)`
senão
    retorne `procureArestaDeNível(`*p.dir*`)`

Veja que o Programa 2.7 não altera o grafo, e, portanto, as invariantes são preservadas. Como a Euler tour tree é balanceada, então o consumo de tempo de cada percurso é $O(\lg n)$ (amortizado em nossa implementação, onde sempre realizamos um `splay` no nó devolvido). Assim, se temos k arestas de nível i da árvore a serem rebaixadas, então a busca por essas k arestas custará tempo $O(k \lg n)$.

A rotina `procureArestaDeNível` será usada na rotina `substituaAresta`, que será descrita na Seção 2.4.7.

2.4.4 Nó de vértice

Na rotina `substituaAresta`, para percorrermos as arestas reserva de nível i incidentes a vértices da árvore T_u em busca de uma aresta substituta, cada nó da floresta F_i possui um booleano chamado *incideArestaReservaDeNível*, que é verdadeiro somente se o nó é um nó de vértice, e o vértice em questão é ponta de alguma aresta reserva de nível i. Dessa forma, se uv é aresta reserva de nível 2, então os nós de vértice de u e de v em F_2 terão o atributo *incideArestaReservaDeNível* como verdadeiro.

Cada nó das florestas também guardará um contador *arestasReservasDeNível*, que armazena a quantidade de nós em sua subárvore com o atributo *incideArestaReservaDeNível* verdadeiro. Esse contador deve ser mantido atualizado quando é feita qualquer alteração em uma das árvores binárias que representam as florestas. Na nossa implementação, a atualização é feita na operação `splay`, sempre que essa executa uma rotação.

O campo *incideArestaReservaDeNível* de um nó de vértice u pode mudar de valor quando houver inserções e remoções de arestas reserva que possuem como uma da suas pontas o vértice u. Portanto, mostraremos dois métodos, `incrementeArestasReservasDeNível` e `decrementeArestasReservasDeNível`, que modificam este campo e atualizam o contador *arestasReservasDeNível* chamando `atualizeArestasReservasDeNível`, descrito no Programa 2.10 abaixo.

O método `decrementeArestasReservasDeNível` do Programa 2.8 atualiza o campo *incideArestaReservaDeNível* de um nó de vértice u para falso quando ele não tem mais elementos em sua lista de adjacências das arestas reserva, isto é, quando não há mais nenhuma aresta reserva do nível da floresta incidente nele. A assinatura `R[u]` retorna o conjunto de vizinhos da lista de adjacências de u.

Programa 2.8 `decrementeArestasReservasDeNível(`F, R, u`)`

Entrada: Recebe um vértice u da floresta F e uma lista de adjacências R.
Efeito: Atualiza o campo *incideArestaReservaDeNível* para falso se necessário.

```
vérticeU ← F.nó[u, u]
se R[u] = ∅ então
    splay(vérticeU)
    vérticeU.incideArestaReservaDeNível ← falso
    atualizeArestaReservaDeNível(vérticeU)
```

Já o método `incrementeArestasReservasDeNível` do Programa 2.9 atualiza o atributo *incideArestaReservaDeNível* de um nó de vértice u para verdadeiro quando adicionamos um vértice v na lista de adjacências de u e v é o primeiro elemento de sua lista de adjacências, pois isso indica que u passa a ser incidente à aresta reserva uv.

Programa 2.9 `incrementeArestasReservasDeNível(`F, R, u`)`

Entrada: Recebe um vértice u da floresta F e uma lista de adjacências R.
Efeito: Atualiza o campo *incideArestaReservaDeNível* para verdadeiro se necessário.

```
vérticeU ← F.nó[u, u]
se |R[u]| = 1 então
    splay(vérticeU)
    vérticeU.incideArestaReservaDeNível ← verdadeiro
    atualizeArestaReservaDeNível(vérticeU)
```

Programa 2.10 `atualizeArestasReservasDeNível(`p`)`

Entrada: Recebe um nó p.
Efeito: Atualiza o contador *arestasReservasDeNível* de p.

```
c ← 0
se p.esq ≠ NIL então
    c ← c + p.esq.arestasReservasDeNível
se p.dir ≠ NIL então
    c ← c + p.dir.arestasReservasDeNível
se p.incideArestaReservaDeNível então
    c ← c + 1
p.arestasReservasDeNível ← c
```

O Programa 2.10 consome tempo O(1). Já os Programas 2.8 e 2.9 consomem tempo O(lg n), amortizado em nossa implementação por conta das operações `splay`. Como estes três métodos não alteram a floresta F_i, alteram apenas uma da árvores uma das árvores binárias que a representam, então as três invariantes são preservadas.

Usando o mesmo exemplo da Figura 2.11 na Seção 2.4.1, podemos ilustrar como estaria o atributo *arestasReservasDeNível* de cada nó na árvore.

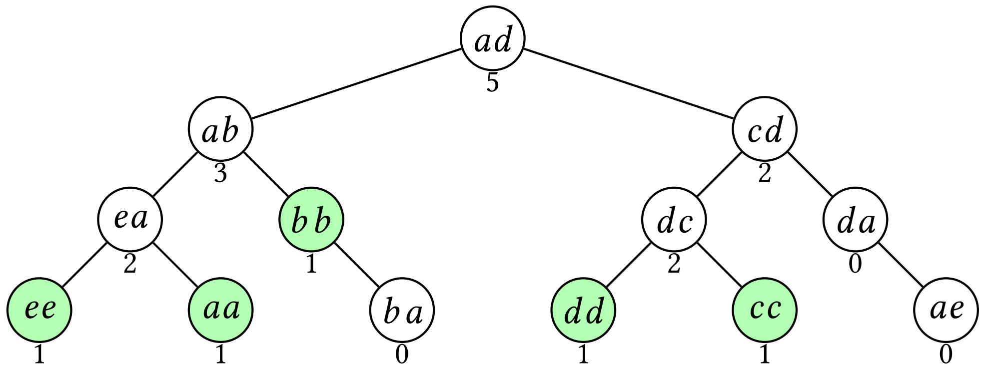

Figura 2.13: *Árvore de uma das componentes da floresta F_i, onde nós pintados em verde indicam nós de vértices com o atributo incideArestaReservaDeNível verdadeiro. No nosso exemplo, todos os vértices são ponta de alguma aresta reserva de nível i. Embaixo de cada nó temos o contador arestasReservasDeNível.*

No `substituaAresta` acionado na floresta F_i, após rebaixarmos as arestas de nível i de T_u, precisamos procurar por uma aresta substituta. Para isso, precisamos buscar por uma aresta reserva de nível i, com uma ponta em T_u e outra em T_v, para que consigamos reconectar as duas componentes de F_i separadas pela remoção de uv.

Para resolver este problema de maneira eficiente, usa-se uma estratégia semelhante à que usamos para buscar arestas de nível i em T_u. Introduzimos um método auxiliar chamado `procureNóIncideArestaReservaDeNível`, que devolveria um vértice de T_u que incide em alguma aresta reserva de nível i. Assim, podemos percorrer cada vizinho desse vértice em R_i para verificar se a aresta entre eles liga T_u a T_v.

Programa 2.11 `procureNóIncideArestaReservaDeNível`(p)

Entrada: Recebe um nó da floresta p com o contador *arestasReservasDeNível* > 0.
Saída: Devolve um nó de vértice da subárvore de p com *incideArestaReservaDeNível* verdadeiro.

se *p.incideArestaReservaDeNível* **então**
    retorne p
se *p.esq* $\neq$ `NIL` **e** *p.esq.arestasReservasDeNível* > 0 **então**
    retorne `procureNóIncideArestaReservaDeNível`(*p.esq*)
senão
    retorne `procureNóIncideArestaReservaDeNível`(*p.dir*)

Veja que o Programa 2.11 não altera o grafo, e, portanto, as invariantes são preservadas. Como a Euler tour tree é balanceada, o consumo de tempo do Programa 2.11 é $O(\lg n)$ (amortizado em nossa implementação, que sempre aciona a rotina `splay` no nó devolvido).

A rotina `procureNóIncideArestaReservaDeNível` será usada na rotina `substituaAresta`, que será descrita na Seção 2.4.7.

2.4.5 Versão completa da rotina de adição de arestas

Nesta seção, apresentamos uma versão completa da rotina `adicioneGD` mostrada na Seção 2.3, para incorporar os atributos descritos nas seções anteriores.

Programa 2.12 `adicioneGD(G, u, v)`

Entrada: Recebe dois vértices u e v do grafo G, com $u < v$.
Efeito: Adiciona a aresta uv no grafo G.

```
1  L ← G.nívelMax
2  G.nível[u,v] ← L
3  se conectadosFD(G.F_L, u, v) então                          ▷ uv é aresta reserva
4      adicioneLA(G.R_L, u, v)
5      incrementeArestasReservasDeNível(G.F_L, G.R_L, u)
6      incrementeArestasReservasDeNível(G.F_L, G.R_L, v)
7  senão
8      adicioneFD(G.F_L, u, v)
9      atualizeÉNível(G.F_L, u, v, verdadeiro)
```

Nesta versão final, veja que as linhas 5 e 6 são necessárias para que o campo *incideArestaReservaDeNível*, assim como o campo de todos os nós da floresta dos vértices u e v estejam corretos. Já a linha 9 define o atributo *éNível* do nó de aresta uv como verdadeiro em F_L, quando ela é aresta da floresta, e atualiza o campo *arestasDeNível* de todos os nós da floresta.

Atualizamos os atributos desses nós que descrevemos para fazermos a remoção eficiente de arestas, cuja rotina será descrita também com ajustes na Seção 2.4.6. Note que o método `adicioneGD` do Programa 2.12 continua tendo o mesmo consumo de tempo do Programa 2.3 após esses ajustes, ou seja, $O(\lg n)$ amortizado.

2.4.6 Versão completa da rotina de remoção de arestas

Nessa seção, apresentamos uma versão completa da rotina `removaGD` apresentada no Programa 2.4, incorporando os atributos descritos na Seção 2.4.4.

Programa 2.13 `removaGD(G, u, v)`

Entrada: Recebe dois vértices adjacentes u e v do grafo G.
Efeito: Remove a aresta uv do grafo G.

```
L ← G.nívelMax
i ← G.nível[u,v]
G.nível[u,v] ← NIL                                   ▷ marcamos uv como removida
se uv ∈ G.F_L então                                      ▷ uv é aresta da floresta
    para j ← i até L faça
        removaFD(G.F_j, u, v)
    substituaAresta(G, i, u, v)
senão                                                   ▷ uv é aresta reserva
    removaLA(G.R_i, u, v)
    decrementeArestasReservasDeNível(G.F_i, G.R_i, u)
    decrementeArestasReservasDeNível(G.F_i, G.R_i, v)
```

A diferença entre as duas versões de `removaGD` é que a segunda tem as linhas 10 e 11 a mais. Tais linhas são necessárias para que o campo *incideArestaReservaDeNível* e o contador *arestasReservasDeNível* dos nós estejam corretos, visto que, ao remover uma aresta reserva, precisamos verificar se os vértices incidentes a ela ainda são incidentes a alguma outra aresta reserva de R.

Agora falta descrever a rotina `substituaAresta`. Da mesma forma que a primeira versão, nesta segunda versão `removaGD` também consumirá tempo amortizado $O(\lg^2 n)$ mais o custo do `substituaAresta`.

2.4.7 Rotina de substituição de aresta

A rotina de substituição da aresta `substituaAresta` está descrita no Programa 2.14 abaixo, onde usaremos vários dos métodos auxiliares apresentados nas Seções 2.4.3 e 2.4.4. Além disso, usaremos dois métodos auxiliares chamados `rebaixeNívelDaAresta` e `testeSubstituta`, que serão explicados de maneira detalhada no momento em que estivermos explicando cada trecho do código de `substituaAresta`. O atributo *tam*, que existe em todos os nós da floresta, guarda o número de nós na subárvore de cada nó.

Programa 2.14 `substituaAresta(G, i, u, v)`

Entrada: Recebe dois vértices u e v do grafo G, e o nível i da aresta uv.
Efeito: Adiciona uma aresta substituta no grafo, se ela existir.

```
1   L ← G.nívelMax
2   para j ← i até L faça
3       T_u ← splay(G.F_j.nó[u, u])                          ▷ torna o nó uu raiz de T_u
4       T_v ← splay(G.F_j.nó[v, v])                          ▷ torna o nó vv raiz de T_v
5       se T_u.tam > T_v.tam então
6           T_u ↔ T_v
7       enquanto T_u.arestasDeNível > 0 faça
8           nóXY ← procureArestaDeNível(T_u)
9           T_u ← splay(nóXY)
10          rebaixeNívelDaAresta(G, nóXY, j)
11      enquanto T_u.arestasReservasDeNível > 0 faça
12          nóXX ← procureNóIncideArestaReservaDeNível(T_u)
13          T_u ← splay(nóXX)
14          (x, x) ← nóXX.vértices
15          para y ∈ G.R_j[x] faça
16              se testeSubstituta(G, x, y, j) então
17                  retorne
```

Para explicar o `substituaAresta` do Programa 2.14, descreveremos a função de cada trecho do código. Vamos assumir de início que estamos aplicando as operações de remoção em um grafo G de n vértices. Lembre-se que, ao removermos uma aresta uv da floresta de nível i, precisamos procurar uma substituta partindo de R_i, e se não encontrarmos, passamos a buscar em $R_{i+1}, \dots, R_L$. A linha 2 faz exatamente essa iteração sobre os níveis i até L. Suponha que estamos na iteração j da linha 2, ou seja, já removemos uv de $F_i, \dots, F_{j-1}$ e não encontramos aresta substituta em $R_i, \dots, R_{j-1}$.

Nas linhas 3 e 4, obtemos as árvores T_u e T_v, que foram derivadas da floresta F_j, já que uma componente desta foi quebrada em duas após a remoção de uv. As operações `splay` puxam os nós u e v para raiz de T_u e T_v, respectivamente. As linhas 5 e 6 garantem que $|T_u| \leq |T_v|$ como já discutido na Seção 2.3.4.

Nas linhas 7 a 10, realizamos o processo de rebaixar as arestas de T_u de nível j. Na linha 7, temos um laço que terminará quando todas as arestas de nível j tiverem sido rebaixadas, isto é, quando o contador *arestasDeNível* do nó raiz de T_u estiver nulo. Na linha 8, utilizamos o método auxiliar `procureArestaDeNível`, que retornará um nó de aresta de nível j, ou seja, que possui o *éNível* verdadeiro. Para cada nó de aresta retornado, acionamos `splay` nele e atualizamos T_u. Em seguida, o rebaixamos chamando o método `rebaixeNívelDaAresta`, descrito abaixo.

Programa 2.15 `rebaixeNívelDaAresta(`G`, `p`, `j`)`

Entrada: Recebe o grafo G, um nó de aresta p da floresta F_j que é raiz de uma árvore e tem nível j.

Efeito: Rebaixa o nível do nó de aresta p.

```
(x, y) ← p.vértices
G.nível[x, y] ← j − 1
atualizeÉNível(G.F_j, x, y, falso)
adicioneFD(G.F_{j−1}, x, y)
atualizeÉNível(G.F_{j−1}, x, y, verdadeiro)
```

No Programa 2.15, atualizamos o nível do nó de aresta xy de j para $j - 1$. Note que o nó p é raiz de sua árvore pois foi feito um splay neste nó antes da chamada a `rebaixeNívelDaAresta`, na linha 9 do Programa 2.14. Ademais, como rebaixamos xy de F_j para F_{j-1}, então em F_j atualizamos o seu atributo *éNível* para falso, e em F_{j-1} atualizamos este atributo para verdadeiro. Veja que `rebaixeNívelDaAresta` consome tempo amortizado $O(\lg n)$.

Voltando ao Programa 2.14, nas linhas 11 a 17, procuramos por uma aresta substituta de nível j. Similarmente à linha 7, a linha 11 é um laço que terminará quando não existirem mais arestas reserva de nível j (ou seja, quando o contador *arestasReservasDeNível* do nó raiz de T_u estiver nulo) ou quando achamos uma aresta substituta.

Na linha 12, acionamos `procureNóIncideArestaReservaDeNível`, que retornará um nó de vértice xx que incide em alguma aresta reserva de nível j. Depois, acionamos `splay` em xx e atualizamos T_u. Na linha 14, obtemos o vértice x do nó xx e percorremos todos os vizinhos y de x na lista de adjacências na linha 15, pois queremos testar se xy é uma aresta substituta.

Para isso, chamamos o método auxiliar `testeSubstituta` na linha 16, descrito no Programa 2.16. Como o nome sugere, o método testa se xy é uma aresta substituta, devolvendo verdadeiro se xy for e falso caso contrário. Quando o método devolve verdadeiro, chegamos à linha 17 do Programa 2.14, terminando o algoritmo. Caso contrário, continuamos percorrendo os vizinhos y da lista de adjacências de x e chamando o mesmo método várias vezes.

Programa 2.16 `testeSubstituta`(G, x, y, j)

Entrada: Recebe o grafo G, as pontas x e y do nó xy e o nível j.
Saída: Devolve verdadeiro se a aresta xy é substituta e falso caso contrário.

1 `removaLA`($G.R_j$, x, y)
2 **se** `conectadosGD`(G, x, y) **então** ▷ a aresta xy não é substituta
3     $G.nível[x, y] \leftarrow j - 1$
4   `adicioneLA`($G.R_{j-1}$, x, y)
5   `incrementeArestasReservasDeNível`($G.F_{j-1}$, $G.R_{j-1}$, x)
6   `incrementeArestasReservasDeNível`($G.F_{j-1}$, $G.R_{j-1}$, y)
7   **retorne falso**
8 **senão** ▷ a aresta xy é substituta
9     $L \leftarrow G.nívelMax$
10   **para** $k \leftarrow j$ **até** L **faça**
11   `adicioneFD`($G.F_k$, x, y)
12   **se** $x > y$ **então**
13         $x \leftrightarrow y$
14   `atualizeÉNível`($G.F_j$, x, y, **verdadeiro**)
15   **retorne verdadeiro**

No Programa 2.16, a linha 1 remove a aresta reserva xy de R_j, independentemente se tal aresta é substituta ou não, pois ou ela será rebaixada, ou será uma aresta substituta, que será incluída em F_j. Se xy não é substituta, então ela é rebaixada para R_{j-1}, como visto na Seção 2.3.4. Se ela é substituta, então se tornará uma aresta da floresta de nível j conectando T_u a T_v.

As linhas 2 a 7 englobam o caso em que os vértices x e y estão em T_u, isto é, quando `conectadosGD(G, x, y)` retorna verdadeiro. Isso quer dizer que xy não é uma aresta substituta, e por isso precisamos rebaixá-la de R_j para R_{j-1}, além de atualizar os atributos dos nós de vértice x e y em F_{j-1} ao acionar `incrementeArestasReservasDeNível`. Devolvemos falso porque, neste caso, xy não é aresta substituta.

Veja que `conectadosGD(G, x, y)` só devolverá falso quando x está em T_u e y está em T_v, pois assim os dois vértices estariam em componentes separadas da floresta F_j. Este caso, abordado nas linhas 8 a 15 do Programa 2.16, mostra que encontramos xy como uma aresta substituta, e finalizamos o algoritmo incluindo xy em todas as florestas F_k, para $k = j, \dots, L$, para manter a invariante (II). Além disso, se a aresta substituta encontrada é de nível j, precisamos atualizar o atributo *éNível* do nó desta aresta em F_j para verdadeiro quando a adicionamos na floresta F_j, como acontece na linha 14. Por fim, devolvemos verdadeiro pois, neste caso, achamos uma aresta substituta.

Veja que, no Programa 2.16, as linhas 2 a 7 possuem custo amortizado $O(\lg n)$. Isso quer dizer que, enquanto as arestas que estamos testando não forem substitutas, o método `testeSubstituta` será acionado várias vezes com esse custo de tempo. No momento em que encontrarmos uma substituta, `testeSubstituta` será acionado uma única vez e consumirá tempo amortizado $O(\lg^2 n)$ por causa das linhas 10 e 11, e assim o algoritmo será finalizado.

Agora, explicaremos o custo da rotina `substituaAresta`. No pior caso, uma execução desta rotina pode consumir muito tempo. Por exemplo, se o grafo já está com $m = \Theta(n^2)$

arestas inseridas, todas de nível L, pode ocorrer uma remoção que aciona o `substituaAresta` e que acarreta o rebaixamento de $\Theta(n^2)$ arestas, a um custo $\Omega(n^2 \lg n)$.

No entanto, para chegar a essa situação, teriam ocorrido $\Theta(n^2)$ inserções, cada uma com um custo bem mais barato, de $O(\lg n)$. Isso sugere que possivelmente uma análise amortizada do custo das operações leve a um custo por operação mais baixo.

Agora mostraremos que, se ocorreram t operações de inserção e remoção de arestas desde a criação do grafo, então o custo total de tal sequência de operações é $O(t \lg^2 n)$, o que resulta em um custo amortizado por operação de $O(\lg^2 n)$.

Para tanto, cada inserção será responsável não apenas pelo custo da inserção de uma aresta e, mas também pelo custo de todos os rebaixamentos sofridos por e no decorrer de todas as remoções que ocorrerem após a inserção de e. Isso quer dizer que a inserção da aresta e vai pagar por cada execução das linhas 7 a 10 do Programa 2.14 e das linhas 2 a 7 do Programa 2.16 que processa a aresta e. Como a inserção custa $O(\lg n)$ e essas linhas custam $O(\lg n)$ e são executadas $O(\lg n)$ vezes, pois e pode ser rebaixada no máximo $\lceil \lg n \rceil$ vezes, o custo pago por uma inserção é $O(\lg^2 n)$.

Já uma remoção de aresta, executada pelo Programa 2.13, custa $O(\lg^2 n)$ mais o custo do `substituaAresta`. O custo do `substituaAresta` é $O(\lg^2 n)$ excluindo-se as execuções das linhas 7 a 10 do Programa 2.14, como também as linhas 2 a 7 do Programa 2.16. Isso porque, desconsiderando estas linhas onde ocorrem rebaixamento de aresta, na linha 2 do Programa 2.14 acaba tendo custo $O(\lg n)$ nas linhas restantes. Apesar de as linhas 10 e 11 do Programa 2.16 consumirem tempo $O(\lg^2 n)$, este trecho do código será executado uma única vez quando encontrarmos a substituta e o algoritmo será finalizado. Assim, como a linha 2 pode ser executada em no máximo $O(\lg n)$ vezes, temos que `substituaAresta` consome tempo amortizado $O(\lg^2 n)$ por operação de remoção.

Com isso, concluímos que o custo total da sequência de t inserções e remoções é $O(t \lg^2 n)$, e assim cada inserção e remoção consome tempo amortizado $O(\lg^2 n)$.

Capítulo 3

Algoritmo para MSF decremental

Neste capítulo, estudaremos o problema da árvore geradora mínima em grafos dinâmicos. Dado um grafo conexo G com um custo associado a cada uma de suas arestas, o problema da árvore geradora mínima consiste em determinar uma árvore geradora de G com custo mínimo, onde o custo de uma árvore é a soma dos custos de suas arestas. Como estamos interessados em grafos dinâmicos, é natural remover a restrição de que o grafo seja conexo, e neste caso considerar florestas geradoras maximais de custo mínimo (MSF, do inglês, *minimum spanning forest*). Chamamos um grafo com um custo associado a cada aresta de **grafo ponderado**.

O problema da árvore geradora mínima em grafos ponderados (conexos) estáticos pode ser resolvido eficientemente, por exemplo, pelos algoritmos de Kruskal e de Prim. O algoritmo de Kruskal utiliza uma estrutura de dados clássica conhecida como union-find, enquanto que o algoritmo de Prim utiliza uma fila de prioridades. Não há na literatura uma versão destes algoritmos para grafos dinâmicos. Isso talvez se deva à característica essencialmente sequencial destes algoritmos, que modificam suas estruturas internas conduzidos por uma ordem de eventos. Uma alteração no grafo poderia levar a uma alteração em toda a sequência de eventos nesses algoritmos a partir de um certo ponto, e com isso não há uma versão eficiente deles que acomode alterações no grafo.

Por outro lado, Holm, de Lichtenberg e Thorup [3] propuseram uma adaptação do seu algoritmo para conexidade em grafos dinâmicos, apresentado no Capítulo 2, para que este mantenha, de maneira eficiente, uma floresta geradora maximal de custo mínimo em um grafo ponderado que pode sofrer remoções de arestas. Ou seja, eles propuseram um algoritmo que resolve de maneira eficiente o problema que chamamos de **MSF decremental**. Neste capítulo, descreveremos esse algoritmo, que é uma adaptação do algoritmo descrito no Capítulo 2 para que este passe a resolver o problema da MSF decremental.

3.1 Biblioteca da MSF decremental

Implementar o algoritmo decremental para florestas geradoras maximais de custo mínimo resume-se à construção da seguinte biblioteca de forma eficiente:

- **`MSFDecremental(`**n**`,`** E**`)`**: contrói e devolve um grafo ponderado G com n vértices e as arestas ponderadas dadas no conjunto E;

- **`consultePesoMSF(`**G**`)`**: devolve o peso de uma MSF do grafo ponderado G;
- **`removaMSF(`**G**`,`** u**`,`** v**`)`**: remove a aresta uv do grafo ponderado G.

Note que, diferente da biblioteca do algoritmo de conexidade em grafos dinâmicos, apresentada na Seção 2.3, na MSF decremental não temos um método equivalente a `adicioneGD` disponível para o usuário. Em nossa implementação [6], para criarmos um grafo G de n vértices e m arestas ponderadas dadas em E, acionamos `MSFDecremental(`n`,` E`)`, onde criamos, como no problema da conexidade em grafos dinâmicos, $\lceil \lg n \rceil$ florestas dinâmicas e $\lceil \lg n \rceil$ listas de adjacências, com n vértices isolados. Em seguida, ordenamos e inserimos as m arestas de E em ordem crescente de peso, usando uma biblioteca pronta do C++ para ordená-las, que consome tempo esperado $O(m \lg n)$. Estas m arestas são inseridas uma a uma acionando uma rotina que chamamos de `adicioneMSF(u, v, w)`, onde u e v são pontas da aresta e w é o peso dela.

A rotina `adicioneMSF` é acionada somente dentro do construtor e tem custo amortizado $O(\lg n)$. Ela é uma versão da `adicioneGD` que acomoda os pesos das arestas como veremos adiante. Por ser uma rotina privada, ou seja, não está disponível para o usuário, após a inserção destas arestas, não é permitido mais operações de inserção, somente de remoção de arestas. Para o usuário, então, só estarão disponíveis as rotinas `consultePesoMSF` e `removaMSF`. A versão totalmente dinâmica, que inclui a rotina `adicioneMSF` para o usuário, será estudada posteriormente no Capítulo 4.

O construtor `MSFDecremental`, devido à ordenação de arestas e à chamada ao método `adicioneMSF`, possui consumo de tempo $O(m \lg n)$. Já a rotina `consultePesoMSF` possui consumo de tempo $O(1)$. Como estes dois métodos são mais simples, passaremos brevemente sobre eles, e detalharemos mais a rotina `removaMSF`, que possui a rotina auxiliar `substituaArestaMSF` implementada de maneira diferente do `substituaAresta` do algoritmo de conexidade em grafos dinâmicos.

Usaremos várias definições já apresentadas no algoritmo de conexidade em grafos dinâmicos, incluindo as mesmas invariantes apresentadas na Seção 2.2.1, os mesmos tipos de arestas da Seção 2.2.2 e nós das florestas apresentados na Seção 2.4.2. A seguir, apresentaremos as rotinas da MSF decremental e alguns ajustes a serem feitos.

3.1.1 Listas de adjacências

Na Seção 2.2.2, apresentamos a biblioteca de `listasDeAdjacências`, onde usamos um mapa hash para inserir ou remover um vértice v da lista de u, além de percorrer os vizinhos da lista de u. No algoritmo da MSF decremental, quando removemos uma aresta de nível i da floresta F_i, uma componente desta será quebrada em duas, T_u e T_v, da mesma forma que no algoritmo de conexidade em grafos dinâmicos. A diferença é que, no caso da MSF decremental, precisamos buscar por uma aresta substituta que tenha o menor peso e que ligue T_u a T_v. Não podemos simplesmente percorrer todos os vizinhos y de cada vértice x em T_u, verificar se xy reconecta as componentes separadas e se é de menor peso dentre todas as substitutas, já que isso seria ineficiente.

Assim, fica claro que seria bom percorrer as arestas reserva em ordem crescente de peso e testar se alguma é substituta nesta ordem. Por isso, em vez de usar um mapa hash para armazenar os vizinhos de cada vértice, usaremos um min-heap. Na verdade, como estamos trabalhando com nós de vértice e de aresta, cada nó de vértice u guardará um min-heap com os vizinhos de u em R_i, onde a chave dessa estrutura de dados para um vizinho v será o peso da aresta uv. Nós de aresta também guardarão um min-heap, porém vazio.

Os métodos principais (remoção, inserção e extração do vértice de chave mínima) que usamos no min-heap consomem tempo $O(\lg n)$ usando uma implementação tradicional de heap, como a descrita no Capítulo 6 de Thomas H. Cormen et al. [1]. O resto dos métodos (consulta do vértice de chave mínima, da quantidade de elementos na min-heap e se a min-heap está vazia) consomem tempo constante, e eles serão necessários para buscar a aresta substituta de peso, como descreveremos mais à frente.

Como o min-heap é uma estrutura de dados bastante conhecida, não iremos descrever a sua implementação em detalhes. O objetivo é ressaltar as diferenças entre as listas de adjacências utilizadas no algoritmo de conexidade em grafos dinâmicos e na MSF decremental, e como essa mudança afetará o comportamento do método `substituaArestaMSF` da MSF decremental.

Assim, com base na implementação clássica do min-heap, podemos definir a biblioteca das listas de adjacências da MSF decremental.

- **`listasDeAdjacênciasMSF(`n`)`**: constrói e devolve um grafo com n vértices e sem arestas, representado por listas de adjacências armazenadas em min-heaps;
- **`adicioneLAMSF(`R`, `u`, `v`, `w`)`**: adiciona o vértice u na lista de adjacências de v em R e vice-versa, considerando que o peso de uv é w;
- **`removaLAMSF(`R`, `u`, `v`)`**: remove o vértice u da lista de adjacências de v em R e vice-versa;
- **`consulteMinLAMSF(`R`, `u`)`**: retorna um par (v, w), onde v é um vértice do min-heap de u em R com chave mínima;

Uma chamada à rotina `adicioneLAMSF(`R`, `u`, `v`, `w`)` adiciona o par (u, w) no min-heap de v e também adiciona o par (v, w) no min-heap de u, consumindo tempo $O(\lg n)$. Similarmente, uma chamada à rotina `removaLAMSF(`R`, `u`, `v`)` remove o par (u, w) do min-heap de v e também remove o par (v, w) do min-heap de u, consumindo também tempo $O(\lg n)$. Já o método `consulteMinLAMSF` consome tempo $O(1)$, já que estamos apenas consultando a chave mínima do min-heap de um vértice.

3.2 Ajustes nas invariantes

Como agora estamos tratando de florestas geradoras maximais de custo mínimo (MSFs), ajustaremos somente a primeira invariante, onde substituímos o termo floresta maximal por MSF, como se pode ver abaixo.

(I) F_i é uma MSF de G_i para todo $1 \le i \le \lceil \lg n \rceil$;

(II) $F_i \subseteq F_{i+1}$ para todo $1 \le i \le \lceil \lg n \rceil - 1$;

(III) Cada componente da floresta F_i possui no máximo 2^i vértices.

A partir deste momento, usaremos estas três invariantes e mostraremos como elas são preservadas no decorrer das modificações no grafo.

3.3 Rotinas da biblioteca da MSF decremental

3.3.1 Criação do grafo

O construtor `MSFDecremental` é bem parecido com o do grafo dinâmico, descrito na Seção 2.3.1. Além das variáveis de classe existentes que criamos para o grafo G no algoritmo de conexidade em grafos dinâmicos, armazenaremos o peso da MSF numa variável chamada *pesoMSF*, que será simplesmente retornada quando consultarmos o peso da MSF decremental corrente, chamando `consultePesoMSF`.

Também incluiremos um atributo do grafo chamado *peso*, que é um mapa hash que armazena o peso das arestas. Para armazenar o peso w de uma aresta uv, basta chamarmos *G.peso[u, v]* $\leftarrow w$. O atributo *peso* será fundamental para recalcular a variável *pesoMSF* no decorrer das remoções de arestas do grafo.

Dessa forma, podemos apresentar o construtor da MSF decremental no Programa 3.1, que usa a rotina `adicioneMSF` apresentada no Programa 3.2.

Programa 3.1 `MSFDecremental(`n`, `E`)`

Entrada: Recebe o número n de vértices do grafo e um conjunto de arestas E.
Saída: Devolve um grafo G com n vértices e m arestas ponderadas.

```
L ← ⌈lg n⌉
G.nívelMax ← L
G.pesoMSF ← 0
para i ← 1 até L faça
    G.F_i ← florestaDinâmica(n)
    G.R_i ← listasDeAdjacênciasMSF(n)
G.nível ← novoMapaHash(n)
G.peso ← novoMapaHash(n)
ordene(E)                ▷ ordena as arestas do conjunto E em ordem crescente de peso
para cada aresta (u, v, w) em E faça
    adicioneMSF(G, u, v, w)
retorne G
```

Podemos notar algumas diferenças quando comparamos o construtor `MSFDecremental` com o construtor `grafoDinâmico`. Na MSF decremental, além de inicializarmos $\lceil \lg n \rceil$ listas de adjacências e $\lceil \lg n \rceil$ florestas dinâmicas, ordenamos as arestas do conjunto E em ordem crescente de peso e inserimos uma a uma chamando `adicioneMSF`, que está descrita abaixo. Note que esta é a primeira versão do método `adicioneMSF`. A versão completa dele será descrita na Seção 3.3.5.

Programa 3.2 `adicioneMSF(`G`, `u`, `v`, `w`)`

Entrada: Recebe dois vértices u e v do grafo G, com $u < v$, e o peso w da aresta uv.
Efeito: Adiciona a aresta uv de peso w no grafo G.

$L \leftarrow G.nívelMax$
$G.nível[u, v] \leftarrow L$
$G.peso[u, v] \leftarrow w$
se `conectadosFD(`$G.F_L$`, `u`, `v`)` **então** ▷ uv é aresta reserva
    `adicioneLAMSF(`$G.R_L$`, `u`, `v`, `w`)`
    `incrementeArestasReservasDeNível(`$G.F_L$`, `$G.R_L$`, `u`)`
    `incrementeArestasReservasDeNível(`$G.F_L$`, `$G.R_L$`, `v`)`
senão
    $G.pesoMSF \leftarrow G.pesoMSF + w$
    `adicioneFD(`$G.F_L$`, `u`, `v`)`
    `atualizeÉNível(`$G.F_L$`, `u`, `v`, `**verdadeiro**`)`

Como citado antes, a rotina `adicioneMSF` é acionada apenas em `MSFDecremental`. Ademais, a única diferença entre a `adicioneMSF` e a `adicioneGD` que vimos na Seção 2.4.5 é que, na primeira, estamos guardando o peso das arestas quando as inserimos no grafo. Portanto, `adicioneMSF` também consome tempo amortizado $O(\lg n)$.

Para `adicioneMSF`, a invariante (I) é preservada para o nível $i = \lceil \lg n \rceil$, já que estamos inserindo as arestas do grafo em ordem crescente de peso. Essa construção basicamente simula o algoritmo de Kruskal. Como estamos inserindo arestas de nível $\lceil \lg n \rceil$ em $F_{\lceil \lg n \rceil}$, então as florestas de níveis inferiores não são afetadas, mantendo-se, assim, as invariantes (II) e (III) também.

3.3.2 Consulta de peso da MSF

A rotina `consultePesoMSF`, que devolve o peso de uma MSF do grafo G, está descrita abaixo.

Programa 3.3 `consultePesoMSF(`G`)`

Entrada: Recebe o grafo G.
Saída: Devolve o peso de uma MSF de G.

retorne $G.pesoMSF$

É fácil ver que `consultePesoMSF` consome tempo $O(1)$. Além disso, como não estamos alterando nem o grafo nem as florestas do grafo G, então as três invariantes são preservadas.

3.3.3 Remoção de arestas

A remoção de arestas também é semelhante à do algoritmo de conexidade em grafos dinâmicos. A diferença é que a busca por alguma aresta substituta, feita na `substituaArestaMSF` agora, é dada por ordem crescente de peso das arestas reserva.

Quando removemos de F_i uma aresta uv de nível i, quebramos uma componente desta floresta em T_u e T_v, com $|T_u| \leq |T_v|$, e rebaixamos todas as arestas de T_u, da mesma forma como fazíamos antes em `substituaAresta`. Porém, agora buscamos alguma aresta reserva de peso mínimo dentre todas as arestas reserva em R_i incidente a T_u, e testamos se ela é uma substituta. Se não é, a rebaixamos para R_{i-1} e buscamos a próxima de peso mínimo em R_i incidente a T_u. Assim, quando achamos uma substituta, conseguimos manter o peso da MSF do grafo, reconectando as duas componentes separadas devido à remoção de uv.

Para facilitar o entendimento da substituição de aresta na MSF decremental, demonstraremos a remoção de uma aresta uv da floresta em uma série de imagens. Na Figura 3.1, temos um grafo ponderado G e assumiremos que essa é a primeira remoção depois da criação do grafo.

No nosso exemplo, G tem $n = 10$ vértices. Sabemos que $\lceil \lg 10 \rceil = 4$, logo o nível máximo L da floresta é 4 e, consequentemente, $G = G_4$. Como, na construção, todas as inserções ocorrem no nível L em F_4, só temos arestas da floresta de nível 4, enquanto F_3 contém apenas vértices isolados. Neste cenário, note que a remoção da aresta uv da floresta, representada por uma linha tracejada na figura, acaba quebrando a única componente da floresta F_4 em duas, T_u e T_v. Como F_4 é a floresta maximal de nível máximo de G, então removemos a uv somente de F_4.

Nível 4

Nível 3

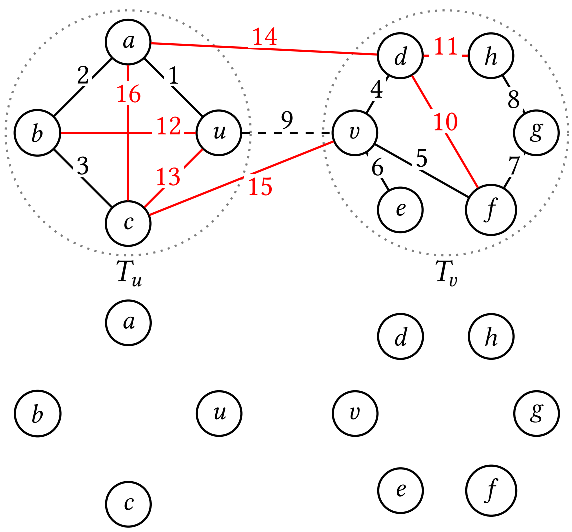

Figura 3.1: *Um grafo ponderado G de 10 vértices, onde as arestas pretas são da floresta F_4, enquanto as vermelhas são reservas. A aresta uv está prestes a ser removida. A floresta F_4 de G de cima contém todas as arestas pretas recém-inseridas e as arestas vermelhas estão em R_4. A floresta de baixo é a F_3, com os vértices isolados, e R_3 também não tem nenhuma aresta.*

O próximo passo é rebaixar todas as arestas de nível 4 em T_u para o nível 3. Dessa forma, as arestas de T_u passam a estar em F_3, como se pode ver na Figura 3.2, pois agora elas passam a ser de nível 3. Como T_u e T_v em F_4 ficaram separadas após a remoção de uv, precisamos encontrar, se existir, uma aresta reserva que possa reconectá-las. Note que agora precisamos percorrer as arestas reserva em ordem de peso. Entretanto, percorrer todas as arestas reserva de R_4 incidentes a T_u e selecionar a de menor peso é ineficiente. Isso porque, se a aresta de peso mínimo não é uma substituta, teremos que buscar a próxima de menor peso e fazer esse processo novamente, o que acaba comprometendo com a performance do algoritmo. Por isso, explicaremos como implementar essa busca eficiente por uma aresta substituta de menor peso na Seção 3.3.4.

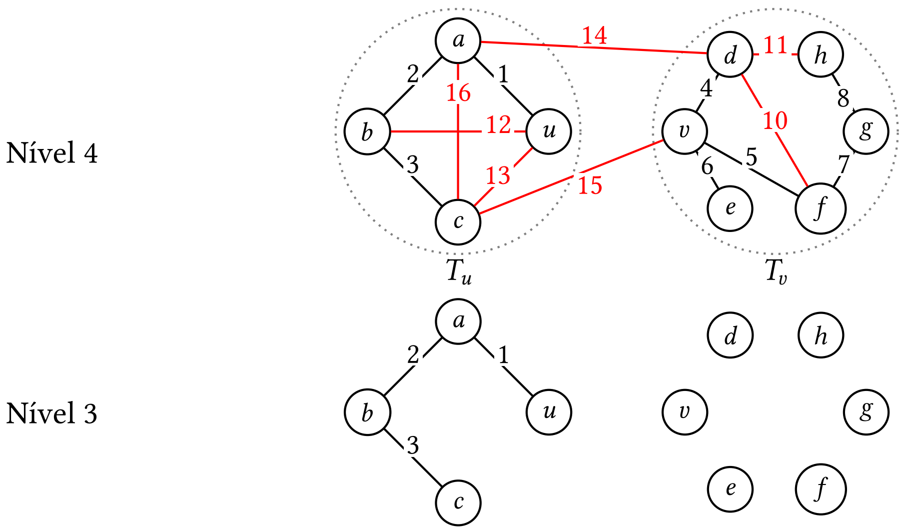

Figura 3.2: *Representação da remoção da aresta uv em G. As arestas de nível* 4 *de T_u foram rebaixadas para o nível* 3*, o que pode ser visto na floresta F_3.*

Na Figura 3.3, percorremos as arestas reserva em ordem de peso em R_4 que tenham uma das pontas em T_u. Para cada aresta percorrida, verificamos se a outra ponta dela incide em algum vértice de T_v. No nosso exemplo, olhamos para as arestas reserva em R_4, antes de encontrarmos a substituta, nesta ordem: bu (peso 12) e uc (peso 13). Veja que as rebaixamos para R_3 por não serem substitutas.

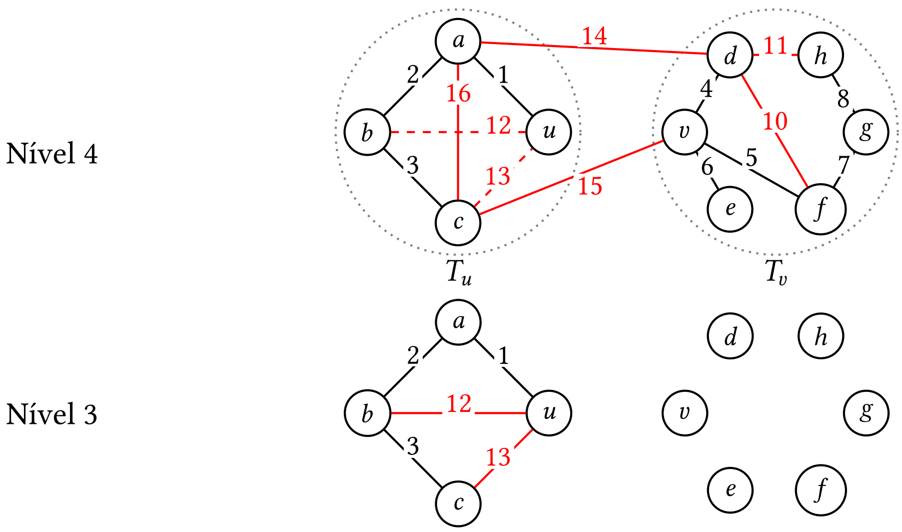

Figura 3.3: *Representação da busca por uma aresta substituta em R_4. As arestas reserva de nível* 4 *que estão tracejadas foram percorridas em ordem crescente de peso e estão prestes a serem removidas de R_4, pois foram rebaixadas para o nível* 3*, como se pode ver em R_3.*

Assim, a próxima aresta reserva de menor peso em R_4 que olharemos é a ad, de peso 14. Como ela conecta T_u a T_v, chamamos `adicioneFD(`F_4`,` a`,` d`)` e ad passa a ser uma aresta da floresta, ou seja, é removida de R_4. Como $i = 4$ é o nível máximo do grafo nesse exemplo, não precisamos chamar esta rotina para os níveis superiores e então terminamos a execução do algoritmo. A Figura 3.4 ilustra essa etapa do algoritmo.

Nível 4

Nível 3

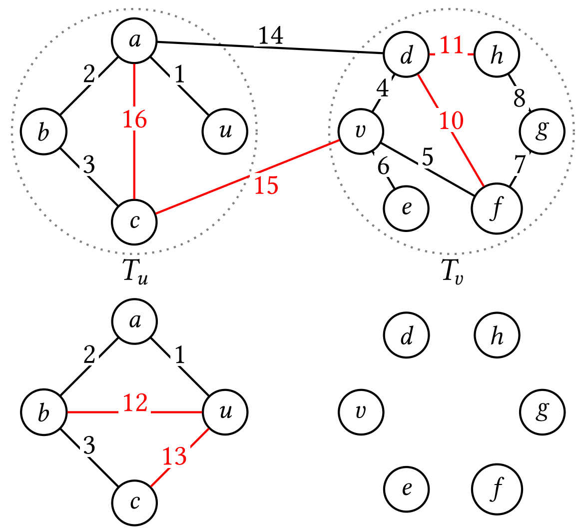

Figura 3.4: *Representação do grafo com a aresta substituta ad por ser a de menor peso em R_4 que conecta T_u a T_v, tornando-se uma aresta da floresta F_4.*

A partir dessas imagens, percebe-se que o método `removaMSF`, descrito abaixo, é bem semelhante ao método `removaGD`, exceto que no primeiro precisamos recalcular o peso da MSF de G quando removemos uma aresta da floresta. Note que o `removaMSF` descrito abaixo é a primeira versão deste método. Descreveremos a sua versão completa na Seção 3.3.6

Programa 3.4 `removaMSF(`G`, `u`, `v`)`

Entrada: Recebe dois vértices adjacentes u e v do grafo G.
Efeito: Remove a aresta uv do grafo G.

$L \leftarrow G.nívelMax$
$i \leftarrow G.nível[u,v]$
$G.nível[u,v] \leftarrow$ `NIL` ▷ marcamos uv como removida
se $uv \in G.F_L$ **então** ▷ uv é aresta da floresta
    $w \leftarrow G.peso[u,v]$
    $G.pesoMSF \leftarrow G.pesoMSF - w$
    para $j \leftarrow i$ **até** L **faça**
        `removaFD(`$G.F_j$`, `u`, `v`)`
    `substituaArestaMSF(`G`, `i`, `u`, `v`)`
senão ▷ uv é aresta reserva
    `removaLAMSF(`$G.R_i$`, `u`, `v`)`
    `decrementeArestasReservasDeNível(`$G.F_i$`, `$G.R_i$`, `u`)`
    `decrementeArestasReservasDeNível(`$G.F_i$`, `$G.R_i$`, `v`)`

O método `substituaArestaMSF`, que é uma versão ajustada de `substituaAresta`, será descrito mais adiante. Por enquanto, sabemos que `removaMSF` consome tempo $O(\lg^2 n)$ mais o custo de `substituaArestaMSF`.

3.3.4 Ajustes em nós das florestas

No algoritmo de conexidade em grafos dinâmicos, vimos que os nós da floresta guardam dois campos, *incideArestaReservaDeNível* e *éNível*, além de dois contadores, *arestasDeNível* e *arestasReservasDeNível*. Mostramos também alguns métodos que atualizam e utilizam estes campos para realizar a busca eficiente de uma aresta substituta.

Para o algoritmo da MSF decremental, além destes campos apresentados, precisaremos de dois campos extras para cada nó da floresta: *peso* e *pesoMínimo*. O primeiro campo armazena o peso de um nó de aresta (nós de vértice guardam ∞ neste campo). Na floresta F_i, cada nó de vértice sabe facilmente o peso mínimo de uma aresta reserva de R_i incidente nele. Assim, o campo *pesoMínimo* de cada nó p de floresta guarda o peso mínimo de uma aresta reserva de R_i incidente a algum vértice cujo nó está na subárvore de p.

Como o peso de cada aresta ponderada nunca muda, então não precisamos atualizar o seu peso. Entretanto, à medida que vamos removendo arestas da floresta F_i, quebramos alguma componente dela em duas e precisamos buscar alguma aresta substituta para reconectar as duas componentes separadas. Assim, quando procuramos por alguma aresta substituta em R_i, podemos neste processo rebaixar algumas arestas de R_i para R_{i-1} e o *peso-Mínimo* dos nós em F_i e em F_{i-1} precisa ser atualizado. Se em R_i acharmos uma substituta, ela se tornará uma aresta da floresta F_i e precisamos também atualizar o *pesoMínimo* de alguns nós em F_i, que agora será o peso mínimo dentre as arestas reserva restantes em R_i.

Por isso, fica claro que precisamos de um método que atualize o campo *pesoMínimo* dos nós. Para isso, criamos o método `atualizePesoMínimo`, que está descrito abaixo. Na nossa implementação, ele é usado em métodos quando estamos fazendo alguma alteração em R_i, e também é usado ao acionarmos as operações `splay`, sempre que essa executa uma rotação.

Programa 3.5 `atualizePesoMínimo(F, R, u)`

Entrada: Recebe um vértice u, as listas de adjacências R e a floresta F.
Efeito: Atualiza o atributo *pesoMínimo* do nó de vértice u.

```
nóUU ← F.nó[u, u]
splay(nóUU)
c ← ∞
se nóUU.esq ≠ NIL e nóUU.esq.pesoMínimo < c então
    c ← nóUU.esq.pesoMínimo
se nóUU.dir ≠ NIL e nóUU.dir.pesoMínimo < c então
    c ← nóUU.dir.pesoMínimo
se R[u] ≠ ∅ então
    (v, w) ← consulteMinLAMSF(R, u)
    se w < c então
        c ← w
nóUU.pesoMínimo ← c
```

Como se pode ver, o Programa 3.5 consome tempo amortizado $O(\lg n)$ por conta da operação `splay`. Além disso, ele não altera a floresta F, altera somente uma das árvores binárias que a representam. Portanto, todas as três invariantes são preservadas.

Para entendermos como estes dois campos extras aparecem em cada nó da floresta, usaremos um exemplo de um grafo ponderado G de 5 vértices e 7 arestas ponderadas, como se pode ver na Figura 3.5. A Figura 3.6 mostra estes campos nos nós da floresta F_L de G.

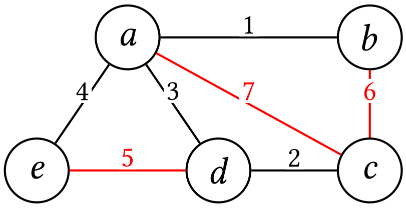

Figura 3.5: *Grafo ponderado G de 5 vértices e 7 arestas ponderadas. Arestas pretas são da floresta e formam a MSF de G, enquanto as vermelhas são arestas reserva.*

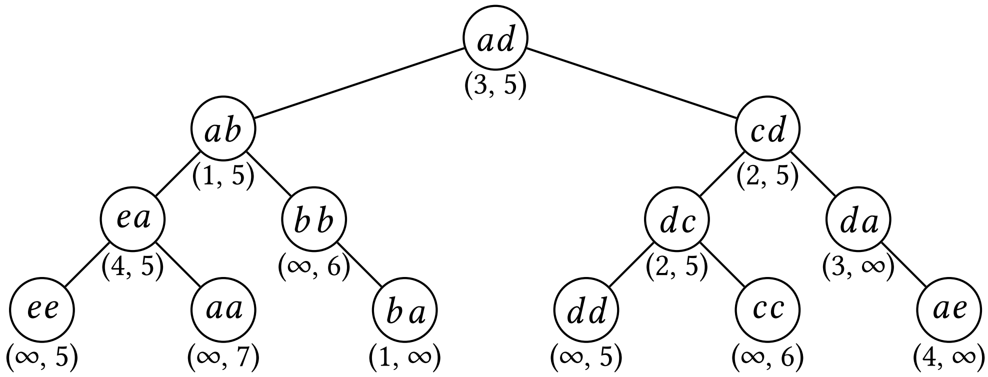

Figura 3.6: *Árvore da única componente da floresta F_L do grafo G da Figura 3.5, onde embaixo de cada nó há um par de números. O primeiro número indica o atributo peso do nó, enquanto o segundo número indica o atributo pesoMínimo, calculado através dos nós em sua subárvore.*

A seguir, o Programa 3.6 apresenta o método `procureNóIncideArestaDePesoMínimo`, que procura e retorna o nó de vértice que incide em uma aresta reserva de peso mínimo. Ele será usado no método `substituaArestaMSF`, que descreveremos na Seção 3.3.7.

Programa 3.6 `procureNóIncideArestaDePesoMínimo`$(R,\ p)$

Entrada: Recebe um nó p de uma floresta com o atributo *arestasReservasDeNível* > 0 e as listas de adjacências R.
Saída: Devolve um nó de vértice incidente a uma aresta reserva de peso mínimo.

```
c ← ∞
(x, y) ← p.vértices
se x = y e R[x] ≠ ∅ então                    ▷ verificamos se p é um nó de vértice
    (v, w) ← consulteMinLAMSF(R, p)
    c ← w
se c ≠ ∞ e p.pesoMínimo = c então
    retorne p
se p.esq ≠ NIL e p.esq.pesoMínimo = p.pesoMínimo então
    retorne procureNóIncideArestaDePesoMínimo(R, p.esq)
senão
    retorne procureNóIncideArestaDePesoMínimo(R, p.dir)
```

Veja que o Programa 3.6 não altera o grafo, e, portanto, as invariantes são preservadas. Como a Euler tour tree é balanceada, então o consumo de tempo de cada percurso é O(lg n) (amortizado em nossa implementação, onde sempre realizamos um `splay` no nó devolvido).

3.3.5 Versão completa da rotina de adição de arestas

A versão completa do método `adicioneMSF` está descrita abaixo. Ao inserirmos uma aresta reserva uv em R_L, precisamos atualizar o atributo *pesoMínimo* dos nós de vértice u e v de F_L, como se pode ver nas linhas 6 e 7. Isso porque se o peso de uv é o menor dentre todas as arestas reserva inseridas em R_L até o momento, o campo *pesoMínimo* de u e de v então passa a ser o peso de uv. Assim, a complexidade de tempo da versão final de `adicioneMSF` continua sendo O(lg n).

Programa 3.7 `adicioneMSF`(G, u, v, w)

Entrada: Recebe dois vértices u e v do grafo G, com $u < v$, e o peso w da aresta uv.
Efeito: Adiciona a aresta uv de peso w no grafo G.

1 $L \leftarrow G.nívelMax$
2 $G.nível[u, v] \leftarrow L$
3 $G.peso[u, v] \leftarrow w$
4 **se** `conectadosFD`($G.F_L$, u, v) **então** ▷ uv é aresta reserva
5   `adicioneLAMSF`($G.R_L$, u, v, w)
6   `atualizePesoMínimo`($G.F_L$, $G.R_L$, u)
7   `atualizePesoMínimo`($G.F_L$, $G.R_L$, v)
8   `incrementeArestasReservasDeNível`($G.F_L$, $G.R_L$, u)
9   `incrementeArestasReservasDeNível`($G.F_L$, $G.R_L$, v)
10 **senão**
11     $G.pesoMSF \leftarrow G.pesoMSF + w$
12   `adicioneFD`($G.F_L$, u, v)
13   `atualizeÉNível`($G.F_L$, u, v, **verdadeiro**)

3.3.6 Versão completa da rotina de remoção de arestas

A versão completa do método `removaMSF` está descrita abaixo.

Programa 3.8 `removaMSF`(G, u, v)

Entrada: Recebe dois vértices adjacentes u e v do grafo G.
Efeito: Remove a aresta uv do grafo G.

$L \leftarrow G.nívelMax$
$i \leftarrow G.nível[u, v]$
$G.nível[u, v] \leftarrow$ `NIL` ▷ marcamos uv como removida
se $uv \in G.F_L$ **então** ▷ uv é aresta da floresta
    $w \leftarrow G.peso[u, v]$
    $G.pesoMSF \leftarrow G.pesoMSF - w$
    para $j \leftarrow i$ **até** L **faça**
        `removaFD`($G.F_j$, u, v)
    `substituaArestaMSF`(G, i, u, v)
senão ▷ uv é aresta reserva
    `removaLAMSF`($G.R_i$, u, v)
    `atualizePesoMínimo`($G.F_i$, $G.R_i$, u)
    `atualizePesoMínimo`($G.F_i$, $G.R_i$, v)
    `decrementeArestasReservasDeNível`($G.F_i$, $G.R_i$, u)
    `decrementeArestasReservasDeNível`($G.F_i$, $G.R_i$, v)

Ao removermos uma aresta reserva uv de R_i, precisamos atualizar o atributo *pesoMínimo* dos nós de vértice u e v de F_i, como se pode ver nas linhas 12 e 13. Isso porque se o peso de uv era o menor dentre todas as arestas reserva restantes, então o atributo `pesoMínimo` de u e de v passa a ser o peso da aresta reserva de segundo menor peso em R_i. Assim, a complexidade de tempo da versão final de `removaMSF` continua sendo $O(\lg^2 n)$ mais o custo da rotina `substituaArestaMSF`, que será descrita na seção seguinte.

3.3.7 Rotina de substituição de aresta

Para descrever a rotina `substituaArestaMSF` do Programa 3.9, usaremos vários dos métodos auxiliares já apresentados, além de alguns métodos e campos novos adicionados aos nós das florestas, como vimos na Seção 3.3.4. Mostraremos também a rotina `testeSubstitutaMSF`, que é uma versão ajustada da rotina `testeSubstituta` que vimos antes.

Programa 3.9 `substituaArestaMSF`(G, i, u, v)

Entrada: Recebe dois vértices u e v do grafo G, e o nível i da aresta uv.
Efeito: Adiciona uma aresta substituta de peso mínimo em G, se ela existir.

1 $L \leftarrow G.nívelMax$
2 **para** $j \leftarrow i$ **até** L **faça**
3     $T_u \leftarrow$ `splay`($G.F_j.nó[u,u]$) ▷ torna o nó uu raiz de T_u
4     $T_v \leftarrow$ `splay`($G.F_j.nó[v,v]$) ▷ torna o nó vv raiz de T_v
5   **se** $T_u.tam > T_v.tam$ **então**
6         $T_u \leftrightarrow T_v$
7   **enquanto** $T_u.arestasDeNível > 0$ **faça**
8         $nóXY \leftarrow$ `procureArestaDeNível`(T_u)
9         $T_u \leftarrow$ `splay`($nóXY$)
10   `rebaixeNívelDaAresta`(G, $nóXY$, `j`)
11   **enquanto** $T_u.arestasReservasDeNível > 0$ **faça**
12         $nóXX \leftarrow$ `procureNóIncideArestaDePesoMínimo`(R_j, T_u)
13         $T_u \leftarrow$ `splay`($nóXX$)
14         $(x, x) \leftarrow nóXX.vértices$
15         $(y, w) \leftarrow$ `consulteMinLAMSF`(R_j, x)
16   **se** `testeSubstitutaMSF`(G, x, y, j) **então**
17   **retorne**

Veja que já vimos as linhas 1 a 10 do Programa 3.9, pois elas são exatamente iguais a esse mesmo trecho do código do Programa 2.14. Isso quer dizer que o rebaixamento de arestas da floresta acontece da mesma forma que no algoritmo de conexidade em grafos dinâmicos. O que muda é somente a forma como procuramos por alguma aresta substituta.

Sendo assim, explicaremos o código da linha 11 em diante. A linha 11, como também já visto, é um laço que terminará quando não existirem mais arestas reserva de nível j (ou seja, quando o contador *arestasReservasDeNível* do nó raiz de T_u estiver nulo) ou quando achamos uma aresta substituta.

A linha 12 é onde acionamos o método `procureNóIncideArestaDePesoMínimo`, para obtermos um nó de vértice incidente a uma aresta reserva de peso mínimo. Assim, obtemos esta aresta chamando `consulteMinLAMSF` na linha 15, onde obtemos um par (y, w).

Em seguida, basta testarmos se a aresta xy de peso mínimo w é uma aresta substituta que reconecta T_u a T_v. Para isso, na linha 16 chamamos `testeSubstitutaMSF`, descrita no Programa 3.10.

Programa 3.10 `testeSubstitutaMSF(`G`, `x`, `y`, `j`)`

Entrada: Recebe o grafo G, as pontas x e y da aresta xy e o nível j.
Saída: Remove xy de R_j e, caso xy seja substituta, adiciona xy a R_j e devolve verdadeiro. Caso contrário, adiciona xy a R_{j-1} e devolve falso.

1 `removaLAMSF(`$G.R_j$`, `x`, `y`)`
2 `atualizePesoMínimo(`$G.F_j$`, `$G.R_j$`, `x`)`
3 `atualizePesoMínimo(`$G.F_j$`, `$G.R_j$`, `y`)`
4 **se** `conectadosFD(`$G.F_j$`, `x`, `y`)` **então** ▷ a aresta xy não é substituta
5 $\quad G.nível[x, y] \leftarrow j - 1$
6 $\quad w \leftarrow G.peso[x, y]$
7 $\quad$ `adicioneLAMSF(`$G.R_{j-1}$`, `x`, `y`, `w`)`
8 $\quad$ `atualizePesoMínimo(`$G.F_{j-1}$`, `$G.R_{j-1}$`, `x`)`
9 $\quad$ `atualizePesoMínimo(`$G.F_{j-1}$`, `$G.R_{j-1}$`, `y`)`
10 $\quad$ `incrementeArestasReservasDeNível(`$G.F_{j-1}$`, `$G.R_{j-1}$`, `x`)`
11 $\quad$ `incrementeArestasReservasDeNível(`$G.F_{j-1}$`, `$G.R_{j-1}$`, `y`)`
12 $\quad$ **retorne falso**
13 **senão** ▷ a aresta xy é substituta
14 $\quad L \leftarrow G.nívelMax$
15 $\quad G.pesoMSF \leftarrow G.pesoMSF + w$
16 $\quad$ **para** $k \leftarrow j$ **até** L **faça**
17 $\quad\quad$ `adicioneFD(`$G.F_k$`, `x`, `y`)`
18 $\quad$ **se** $x > y$ **então**
19 $\quad\quad x \leftrightarrow y$
20 $\quad$ `atualizeÉNível(`$G.F_j$`, `x`, `y`, ` **verdadeiro**`)`
21 $\quad$ **retorne verdadeiro**

Na rotina `testeSubstitutaMSF`, precisamos atualizar a variável *pesoMSF* de G quando encontramos uma aresta substituta, além de chamar os devidos métodos auxiliares (acionamos `removaLAMSF` e `adicioneLAMSF` em vez de `removaLA` e `adicioneLA`). Além disso, como estamos rebaixando arestas reserva, precisamos atualizar o atributo *pesoMínimo* dos nós de vértice afetados, acionando a rotina `atualizePesoMínimo` nas linhas 2, 3, 8 e 9 do Programa 3.10.

Veja que as linhas 4 a 12 do Programa 3.10 possuem custo amortizado $O(\lg n)$. Isso quer dizer que, enquanto as arestas que estamos testando não forem substitutas, o método `testeSubstitutaMSF` será acionado várias vezes com esse custo de tempo. No momento em que encontrarmos uma substituta, `testeSubstitutaMSF` será acionado uma única vez e consumirá tempo amortizado $O(\lg^2 n)$ por causa das linhas 16 e 17, e assim o algoritmo será finalizado.

Agora, explicaremos o custo da rotina `substituaArestaMSF`. Usaremos o mesmo argumento da amortização da rotina `substituaAresta`, apresentado na Seção 2.4.7.

No pior caso, uma execução da rotina `substituaArestaMSF` pode consumir muito tempo. Por exemplo, se o grafo já está com $m = \Theta(n^2)$ arestas inseridas, todas de nível L, pode ocorrer uma remoção que aciona o `substituaArestaMSF` e que acarreta o rebaixamento de $\Theta(n^2)$ arestas, a um custo $\Omega(n^2 \lg n)$.

No entanto, para chegar a essa situação, teriam ocorrido $\Theta(n^2)$ inserções, cada uma com um custo bem mais barato, de $O(\lg n)$. Isso sugere que possivelmente uma análise amortizada do custo das operações leve a um custo por operação mais baixo.

Agora mostraremos que, se ocorreram t operações de inserção e remoção de arestas desde a criação do grafo, então o custo total de tal sequência de operações é $O(t \lg^2 n)$, o que resulta em um custo amortizado por operação de $O(\lg^2 n)$.

Para tanto, cada inserção será responsável não apenas pelo custo da inserção de uma aresta e, mas também pelo custo de todos os rebaixamentos sofridos por e no decorrer de todas as remoções que ocorrerem após a inserção de e. Isso quer dizer que a inserção da aresta e vai pagar por cada execução das linhas 7 a 10 do Programa 3.9 e das linhas 4 a 12 do Programa 3.10 que processa a aresta e. Como a inserção custa $O(\lg n)$ e essas linhas custam $O(\lg n)$ e são executadas $O(\lg n)$ vezes, pois e pode ser rebaixada no máximo $\lceil \lg n \rceil$ vezes, o custo pago por uma inserção é $O(\lg^2 n)$.

Já uma remoção de aresta, executada pelo Programa 3.8, custa $O(\lg^2 n)$ mais o custo do `substituaArestaMSF`. O custo do `substituaArestaMSF` é $O(\lg^2 n)$ excluindo-se as execuções das linhas 7 a 10 do Programa 3.9, como também as linhas 4 a 12 do Programa 3.10. Isso porque, desconsiderando estas linhas onde ocorrem rebaixamento de aresta, na linha 2 do Programa 3.9 acaba tendo custo $O(\lg n)$ nas linhas restantes. Apesar de as linhas 16 e 17 do Programa 3.10 consumirem tempo $O(\lg^2 n)$, este trecho do código será executado uma única vez quando encontrarmos a substituta e o algoritmo será finalizado. Assim, como a linha 2 pode ser executada em no máximo $O(\lg n)$ vezes, temos que `substituaAresta` consome tempo amortizado $O(\lg^2 n)$ por operação de remoção.

Com isso, concluímos que o custo total da sequência de t inserções e remoções é $O(t \lg^2 n)$, e assim cada inserção e remoção consome tempo amortizado $O(\lg^2 n)$.

Capítulo 4

MSF totalmente dinâmica

Neste capítulo, consideraremos o problema da **MSF dinâmica**, onde queremos manter uma MSF em um grafo dinâmico. Estudaremos um algoritmo para MSF dinâmica. Como este algoritmo não foi implementado em nosso estudo, apresentaremos apenas a ideia por trás dele, de como podemos manter o peso mínimo de uma MSF de um grafo G dando suporte à adição e remoção de arestas. Inicialmente, será descrito o funcionamento das **top trees**, estruturas de dados que serão usadas na manutenção da MSF dinâmica. Estas estruturas estão descritas na Seção 2.2 do artigo de Holm, de Lichtenberg e Thorup [3].

4.1 Top trees

Antes de definir uma top tree, precisamos definir alguns conceitos. Seja um par $(T, \partial T)$, onde T é uma árvore e ∂T é um conjunto que contém no máximo dois vértices de T, que são chamados **vértices da fronteira** (*external boundary vertices*), como se pode ver na Figura 4.1.

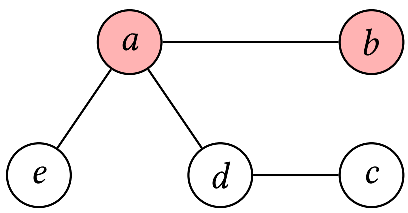

Figura 4.1: *Uma árvore T com um conjunto ∂T destacado em vermelho*

Para qualquer subárvore C de T, seja um conjunto $\partial_{(T,\partial T)}C$, que são vértices de C que estão ou em ∂T ou são incidentes a uma aresta de T saindo de C. Chamamos o conjunto $\partial_{T,\partial T}C$ de vértices da fronteira de C. Veja o exemplo da Figura 4.2.

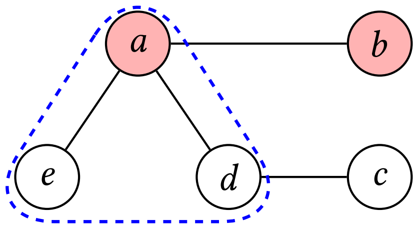

Figura 4.2: *Uma árvore T com um conjunto ∂T marcado em vermelho. A região demarcada em azul representa uma subárvore C contendo os vértices a, e e d e as arestas ae e ad. Neste exemplo, note que os vértices da fronteira de C são a e d.*

Uma subárvore C é chamada de **cluster** de $(T, \partial T)$ se o número de vértices da fronteira de C é no máximo dois. Assim, por definição, T é um cluster dele mesmo, com $\partial_{(T,\partial T)} T = \partial T$. Além disso, qualquer aresta de T é um cluster de $(T, \partial T)$. A subárvore C da Figura 4.2 acima é um cluster, pois ela possui apenas dois vértices da fronteira.

Além disso, se R é uma subárvore de C, então R é um cluster de $(C, \partial_{(T,\partial T)} C)$ se, e somente se, R é um cluster de $(T, \partial T)$. Podemos ver essa situação no exemplo da Figura 4.3. Para simplificar, denotaremos $\partial_{(T,\partial T)}$ como ∂ a partir de agora.

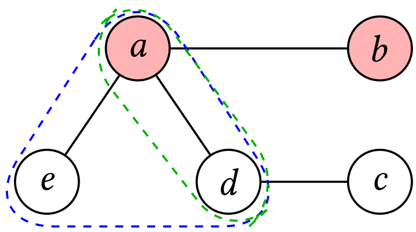

Figura 4.3: *A região demarcada em azul representa a subárvore C de T, e é cluster de $(T, \partial T)$. Já a região demarcada em verde representa uma subárvore R de C, e é cluster de $(C, \partial C)$ e de $(T, \partial T)$.*

Dizemos que clusters A e B são vizinhos se eles compartilham um único vértice e se $A \cup B$ é um cluster. Finalmente, definimos uma **top tree** $\mathcal{T}$ do par $(T, \partial T)$ como sendo uma árvore binária que respeita o seguinte:

- Os vértices de $\mathcal{T}$ são os clusters de $(T, \partial T)$;
- As folhas de $\mathcal{T}$ são as arestas de T;
- Se C é pai de A e de B em $\mathcal{T}$, então $C = A \cup B$ e A e B são vizinhos;
- A raiz de $\mathcal{T}$ é a própria árvore T.

Para um cluster C, os vértices em $C \backslash \partial C$ são chamados de **vértices internos**. Se a e b são vértices da fronteira de C, o caminho entre a e b em C se chama **caminho de cluster** e o denotamos por $\pi(C)$. Se $a \neq b$, o cluster é chamado de **cluster-caminho**.

Dizemos que o cluster C é **ancestral de caminho** do cluster D e D é chamado de **descendente de caminho** de C se ambos forem cluster-caminhos e $\pi(D) \subseteq \pi(C)$. Note que cada aresta $e \in \pi(C)$ é um descendente de caminho de C. Um filho que é descendente de caminho é um filho de caminho. Na Figura 4.4, apresentamos os quatro casos quando fazemos a união de dois clusters. Em (1), temos dois filhos de caminho; em (2), temos um filho de caminho; e em (3) e (4), temos nenhum filho de caminho.

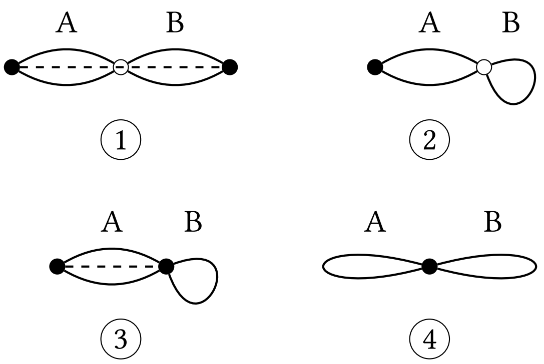

Figura 4.4: *Representação da união de dois clusters A e B formando um cluster C, e seus quatro diferentes casos. Os vértices pretos são vértices da fronteira de C após a união de A e B. Já os vértices brancos são vértices da fronteira de A e de B, mas que não tornaram vértices da fronteira de C após a união. A linha tracejada é o cluster path formado entre os vértices da fronteira de C.*

Ademais, se a é um vértice de fronteira de C e C possui dois filhos A e B, então A é considerado o mais próximo de a se $a \notin B$. Caso $\partial C = \partial A = \partial C$, então o cluster mais próximo é escolhido arbitrariamente. Dada essas definições, podemos ilustrar como ficaria uma *top tree* para a árvore T da Figura 4.5.

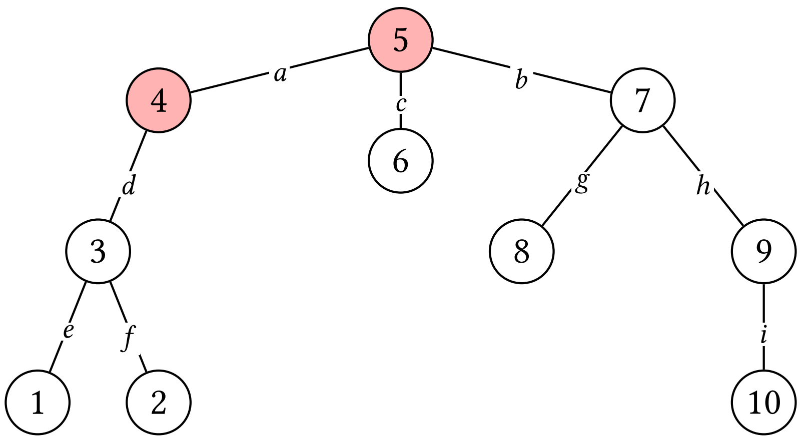

Figura 4.5: *Árvore T com 10 vértices. Os vértices 4 e 5 em vermelho correspondem aos vértices da fronteira de T.*

A partir da árvore T, podemos derivar uma *top tree*, como se pode ver na Figura 4.6 abaixo.

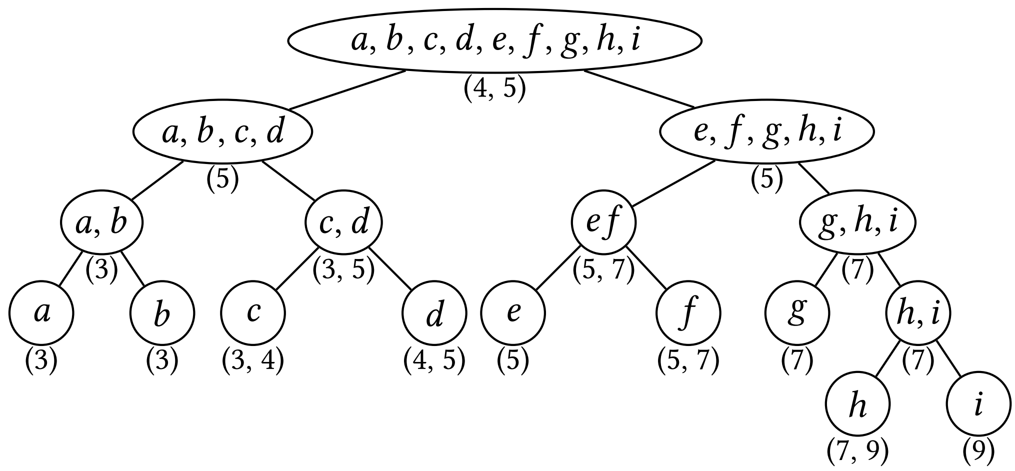

Figura 4.6: *Uma top árvore $\mathcal{T}$ derivada da árvore T. Cada vértice (cluster) de $\mathcal{T}$, exceto as folhas, guarda a união dos vértices dos filhos, que também são clusters. Embaixo de cada vértice p de $\mathcal{T}$, temos os vértices da fronteira do p.*

4.2 Rotinas da biblioteca de top trees

Na construção das florestas dinâmicas, nas quais o algoritmo de conexidade e a MSF decremental foram baseadas, usamos Euler Tour trees para armazenar uma sequência Euleriana em uma árvore binária de busca balanceada. Agora, usaremos top trees na construção da floresta dinâmica, que será utilizada na MSF dinâmica.

Como não iremos implementar a biblioteca, apresentaremos o que cada rotina faz e o seu consumo de tempo de forma breve.

- **`adicioneTopTree(`**$\mathcal{T}$**`,`** u**`,`** v**`)`**: liga o vértice u ao vértice v se ambos estão em top trees diferentes na top tree $\mathcal{T}$, criando a aresta (u, v);
- **`removaTopTree(`**$\mathcal{T}$**`,`** e**`)`**: remove a aresta e da top tree $\mathcal{T}$ da floresta dinâmica;
- **`exponhaTopTree(`**$\mathcal{T}$**`,`** u**`,`** v**`)`**: retorna `NIL` se u e v não estão na mesma árvore. Caso contrário, u e v se tornarão vértices da fronteira da top tree $\mathcal{T}$ que os contém e retorna uma nova cluster raiz (*root cluster*).

As alterações feitas em top trees usando as rotinas mencionadas acima utilizam uma sequência de operações `Merge` e `Split`, definidas abaixo:

- **`Merge(`**A**`,`** B**`)`**: Cria um novo cluster $C = A \cup B$, onde A e B são clusters vizinhos e raízes das top trees $\mathcal{T}_A$ e $\mathcal{T}_B$. O cluster C se torna raiz com os filhos $\mathcal{T}_A$ e $\mathcal{T}_B$, e por fim retornamos C.
- **`Split(`**C**`)`**: Remove a cluster raiz C da top tree $\mathcal{T}$, que contém os clusters filhos A e B. Assim, a remoção de C de $\mathcal{T}$ gera as top trees $\mathcal{T}_A$ e $\mathcal{T}_B$.

Bibliografia

[1] Thomas H. Cormen et al. *Introduction to Algorithms.* 3ª ed. Cambridge, MA: MIT Press, 2009. ISBN: 978-0262033848 (ver pp. 2, 29).

[2] Monika Rauch Henzinger e Valerie King. "Randomized dynamic graph algorithms with polylogarithmic time per operation". Em: *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 (ver p. 16).

[3] Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". Em: *Journal of the ACM* 48.4 (2001), pp. 723–760. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1f63499a9cb43f0f4d6a56b37de551c7e0c94971 (ver pp. v, vii, 1–3, 5, 6, 15, 17, 27, 41).

[4] Dexter C. Kozen. *The Design and Analysis of Algorithms.* New York, NY: Springer-Verlag, 1991. ISBN: 0-387-97687-6 (ver p. 6).

[5] Arthur Henrique Dias Rodrigues. "Algoritmos para conexidade em grafos dinâmicos". Master's thesis. São Paulo, Brazil: Universidade de São Paulo, 2024 (ver p. 5).

[6] Chung Jin Shian. *Repositório Git.* 2025. URL: https://github.com/cjinshian27/TCC (acesso em 26/06/2025) (ver pp. 3, 7, 28).

[7] Daniel D. Sleator e Robert E. Tarjan. "Self-adjusting binary search trees". Em: *Journal of the ACM* 32.3 (1985), pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835 (ver p. 6).