Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Bacharelado em Ciência da Computação

Implementação de um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes

Thiago Henrique de Araújo Lemos

Natal-RN

Fevereiro de 2013

Thiago Henrique de Araújo Lemos

Implementação de um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes

Monografia de Graduação apresentada ao Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de bacharel em Ciência da Computação.

Orientador(a)

Prof. Dr. Marcelo Ferreira Siqueira

Universidade Federal do Rio Grande do Norte – UFRN
Departamento de Informática e Matemática Aplicada – DIMAp

Natal-RN

Fevereiro de 2013

Monografia de Graduação sob o título *Implementação de um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes* apresentada por Thiago Henrique de Araújo Lemos e aceita pelo Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. Marcelo Ferreira Siqueira
Orientador(a)
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Profa. Dra. Elizabeth Ferreira Gouvêa Goldbarg
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Prof. Dr. Marco César Goldbarg
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Profa. Sílvia Maria Diniz Monteiro Maia
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Natal-RN, 15 de fevereiro de 2013

Agradecimentos

Gostaria de agradecer, em primeiro lugar, ao orientador Marcelo Ferreira Siqueira, cujo apoio tornou este trabalho possível; aos meus avós, que sempre rezaram por mim; e aos meus pais, a quem eu devo tudo.

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

Isaac Asimov

Implementação de um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes

Autor: Thiago Henrique de Araújo Lemos
Orientador: Prof. Dr. Marcelo Ferreira Siqueira

Resumo

Este texto descreve um trabalho de conclusão de curso que consistiu em implementar um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. O algoritmo em questão é o que apresenta a melhor complexidade de tempo entre todos aqueles disponíveis na literatura até então para o tipo de grafo acima. A implementação do algoritmo exigiu a utilização de uma estrutura dinâmica de conectividade para testar, de forma eficiente, se um grafo, inicialmente conexo, permanece conexo após a remoção ou inserção de arestas. O algoritmo possui aplicações em vários problemas de geometria computacional.

Palavras-chave: Grafos, emparelhamentos perfeitos, teorema de Petersen, algoritmo.

Implementation of an algorithm for finding perfect matchings in bridgeless cubic graphs

Author: Thiago Henrique de Araújo Lemos
Advisor: Prof. Dr. Marcelo Ferreira Siqueira

Abstract

This manuscript describes the development of a senior project, which comprised the implementation of an algorithm for finding perfect matchings on bridgeless, cubic graphs. To the best of our knowledge, this algorithm has currently the best upper bound for time complexity among all matching algorithms for the same type of graph in the literature. The algorithm implementation is based on a data structure for dynamic connectivity tests. This data structure allows us to find out whether a connected graph becomes disconnected after each insertion or deletion of an edge. Finally, the algorithm implemented here can be applied to many important research problems in computational geometry and related areas.

Keywords: Graphs, perfect matchings, Petersen's theorem, algorithm.

Lista de figuras

Lista de tabelas

Lista de algoritmos

Sumário

1 Introdução

Este capítulo introduz o desenvolvimento de um trabalho de conclusão de curso de graduação. A Seção 1.1 contextualiza o problema-alvo do trabalho e a Seção 1.2 apresenta os objetivos e contribuições. Por fim, a Seção 1.3 descreve como o restante do texto está organizado.

1.1 Contextualização

A procura por emparelhamentos é um problema antigo da teoria dos grafos, com uma história que remonta ao final do século XIX, quando um brilhante matemático dinamarquês, Julius Petersen, publicou um artigo pioneiro sobre o tema (PETERSEN, 1891). Neste artigo, Petersen prova um teorema, que ficou conhecido como teorema de Petersen, que implica que todo grafo cúbico e sem pontes possui um emparelhamento perfeito. Atualmente, esse teorema é mais conhecido como um corolário do teorema de Tutte (LOVÁSZ; PLUMMER, 1986; TUTTE, 1947), caracterizando a existência de emparelhamentos perfeitos em grafos gerais.

Como a prova para o teorema de Petersen dada pelo próprio Petersen é bastante complexa, diversos autores tentaram publicar versões mais simples desde então. O problema com a maioria dessas demonstrações é que elas não fornecem um algoritmo para computar emparelhamentos perfeitos, por não serem construtivas. De particular interesse para esta monografia é a prova de Orrin Frink Jr. (FRINK JR., 1926), que é construtiva, por indução no número de vértices do grafo, e leva a um algoritmo com complexidade de tempo $\mathcal{O}(n^2)$. Uma pequena falha contida na prova dada por Frink foi corrigida por König (KÖNIG, 1990).

O problema de encontrar emparelhamentos perfeitos em *grafos cúbicos e sem arestas de corte* ocorre em muitas aplicações. As subseções a seguir ilustram algumas das mais conhecidas.

1.1.1 Iluminação de terrenos

Um dos problemas clássicos em geometria computacional é o de iluminar ou patrulhar uma área utilizando o menor número possível de lâmpadas ou guardas, respectivamente. Uma de suas instâncias mais conhecidas é a de patrulhar uma galeria de arte usando o menor número possível de câmeras e a maior parte da pesquisa nessa área foi realizada tendo em mente o caso bidimensional (O'ROURKE, 1987; SHERMER, 1992; URRUTIA, 2000). O caso tridimensional, que tem sido cada vez mais alvo de pesquisas, leva em conta o estudo de terrenos poliédricos, isto é, superfícies poliédricas que fazem interseção com cada linha vertical em no máximo um ponto. Cole e Sharir (COLE; SHARIR, 1989) já demonstraram que a versão de decisão do problema de guardar um terreno poliédrico é NP-difícil.

A maior parte do trabalho feito até então para o caso tridimensional considera os terrenos poliédricos a serem patrulhados como triangulações, e modela-os como grafos planares. Isso permite relacionar o problema geométrico de guardar o terreno com o problema de otimização combinatória subjacente de guardar um grafo planar. Um grafo planar está *vigiado* por um conjunto de guardas (posicionados em vértices ou arestas) se pelo menos um guarda é incidente em cada face do grafo. Essa relação é baseada na observação de que a região visível associada a um guarda contém a união de todas as faces incidentes àquele guarda (na verdade, este é o caso quando o poliedro é convexo). Portanto, resolvendo-se o problema de otimização combinatória subjacente é possível se obter um limite superior no número de guardas necessários para guardar um terreno poliédrico.

Utilizando o teorema das quatro cores, Bose, Shermer, Toussaint e Zhu (BOSE et al., 1997) estabeleceram que $\lfloor n/2 \rfloor$ guardas nos vértices ou pelo menos $\lfloor (4n-4)/13 \rfloor$ guardas nas arestas são sempre suficientes para patrulhar um terreno poliédrico com n vértices. Everett e Rivera-Campo (EVERETT; RIVERA-CAMPO, 1997) provaram o resultado aprimorado de que $\lfloor n/3 \rfloor$ guardas nas arestas são sempre suficientes, também utilizando o mesmo teorema. Infelizmente, não se conhece nenhum algoritmo prático para o teorema das quatro cores, e esses resultados não levam a um algoritmo para posicionar os guardas no terreno.

Uma maneira de escapar da dependência do teorema das quatro cores é recorrendo ao uso de emparelhamentos. Com essa estratégia, Bose, Kirkpatrick e Li (BOSE; KIRKPATRICK; LI, 1996) apresentaram algoritmos com complexidade de tempo $\mathcal{O}(n^{3/2})$ para vigiar uma triangulação de n vértices com $\lfloor n/2 \rfloor$ guardas nos vértices ou $\lfloor n/3 \rfloor$ guardas

nas arestas. Para tal, eles usaram o grafo dual da triangulação, que é um grafo cúbico e sem pontes. Logo, pelo teorema de Petersen, este tipo de grafo admite um emparelhamento perfeito e a complexidade dos dois algoritmos é dominada pelo tempo para encontrar esse emparelhamento. O grafo dual das triangulações considerado em (BOSE; KIRKPATRICK; LI, 1996) também é *planar*, mas a restrição de planaridade não é de interesse para esta monografia.

1.1.2 Refinamento adaptativo de malhas

Uma triangulação é uma maneira de representar um objeto geométrico contínuo como uma malha discreta de regiões mais simples, conexas, com interiores disjuntos e com formato triangular. Malhas desse tipo (e as formadas por outros polígonos convexos simples, tais como quadriláteros) possuem aplicações em diversas disciplinas, como análise numérica (CARVALHO; VELHO; GOMES, 1992), geometria computacional (GUIBAS; STOLFI, 1985), computação gráfica (WATT, 1999) e modelagem geométrica de superfícies (SCHUMAKER, 1993).

Muitas simulações numéricas envolvem a solução de equações diferenciais parciais em algum domínio geométrico. O Método dos Elementos Finitos (MEF), um dos mais populares e poderosos para resolver esse tipo de equação (na forma variacional), requer como entrada uma subdivisão do domínio do problema (JOHNSON, 2009), que, na maioria das vezes, é uma triangulação ou quadrilaterização quando o domínio é planar. Em geral, quanto maior for o número de elementos (por exemplo, triângulos) da malha e quanto menor for o tamanho deles, mais precisa será a simulação. Porém, as áreas do domínio que requerem a maior precisão são muitas vezes difíceis de prever antes da simulação ser realizada, e podem variar em função do tempo (no caso em que o fenômeno simulado varia com o tempo).

É possível resolver o problema de encontrar a malha "ideal" refinando-se adaptativamente a malha; isto é, aumentando-se, durante a realização da própria simulação, o número de triângulos nas regiões onde o erro numérico resultante do último passo da simulação é maior. Uma das maneiras de se fazer isso é marcar um vértice de cada triângulo como o *vértice mais novo* e procurar *compatibilidades* entre os triângulos, bisseccionando triângulos compatíveis de seu vértice mais novo ao ponto médio da aresta oposta ao vértice mais novo. Um triângulo é *compatível* se não possuir vizinho (outro triângulo que seja incidente à aresta oposta ao vértice mais novo) ou se seu vértice mais novo se opõe ao vértice mais novo de seu vizinho. Os vértices no ponto médio tornam-se os vértices mais

novos dos triângulos refinados. Esse procedimento é conhecido como *bissecção do vértice mais novo* (MITCHELL, 1991).

A utilidade de emparelhamentos perfeitos, nesse contexto, surge da necessidade de começar o procedimento acima com uma *atribuição compatível* de vértices mais novos (uma atribuição tal que cada triângulo seja compatível). Como o grafo dual da triangulação (modificado pelo acréscimo de alguns vértices ao longo das arestas de fronteira da triangulação, isto é, aqueles incidentes em apenas um triângulo) é cúbico e sem pontes, o grafo admite um emparelhamento perfeito. Qualquer emparelhamento desse tipo no grafo em questão resulta em uma atribuição compatível na triangulação. Logo, a complexidade de encontrar tal atribuição é dominada pela busca por um emparelhamento perfeito no grafo dual.

1.1.3 Quadrilaterizações

Nem sempre triangulações são o tipo de malha mais apropriado para certas simulações numéricas, como é o caso de algumas simulações baseadas no MEF (MALANTHARA; GERSTLE, 1997). Em tais casos, é preferível o uso de quadrilaterizações: subdivisões do domínio em regiões na forma de quadriláteros. Existem vários algoritmos para calcular quadrilaterizações de regiões planares (BLACKER, 1991; JOE, 1995; BERN; EPPSTEIN, 1997; OWEN et al., 1999; VISWANATH; SHIMADA; ITOH, 2000; RAMASWAMI et al., 2005; ATALAY; RAMASWAMI; XU, 2008), mas eles não oferecem garantias teóricas tão boas para alguns atributos da malha gerada quanto os algoritmos que calculam triangulações — o que pode ser fundamental em algumas aplicações (BERN; EPPSTEIN, 1992). Isso ocorre pois, diferente do que ocorreu com triangulações, cujas propriedades foram bem estudadas e são conhecidas há décadas, sabe-se relativamente pouco sobre as propriedades das quadrilaterizações.

A dificuldade em se gerar "boas" quadrilaterizações e o bom conhecimento das propriedades das triangulações levaram vários pesquisadores a adotar a estratégia de converter boas triangulações em quadrilaterizações, na esperança de "herdar", neste processo de conversão, as boas propriedades da malha triangular. Uma das maneiras de realizar essa conversão consiste, basicamente, em emparelhar pares de triângulos que compartilham uma aresta, originando blocos independentes de pares de triângulos, que são, em seguida, transformados em quadriláteros através da remoção da aresta comum aos dois triângulos do par. De maneira semelhante ao mostrado na subseção anterior, esse problema pode ser modelado como a busca por um emparelhamento perfeito no grafo dual da triangulação,

que é cúbico e sem pontes e que, portanto, admite o emparelhamento desejado.

1.2 Objetivo e contribuições

Diferentemente da maioria dos outros algoritmos para encontrar emparelhamentos perfeitos, que se baseiam na ideia de encontrar caminhos de aumento (EDMONDS, 1965; GABOW, 1976; LAWLER, 2001; MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991) e lidam com grafos arbitrários, o algoritmo apresentado por Biedl, Demaine, Bose e Lubiw em (BIEDL et al., 2001) se baseia na prova de Frink para o teorema de Petersen e é restrito a grafos cúbicos e sem pontes (ou, equivalentemente, 3-regulares e biconexos). Mais especificamente, esses pesquisadores identificaram dois "gargalos" no algoritmo resultante da prova de Frink, que possui complexidade de tempo $\mathcal{O}(n^2)$, onde n é o número de vértices do grafo, e propuseram um algoritmo com complexidade de tempo $\mathcal{O}(n \lg^4 n)$. Esta complexidade é assintoticamente inferior àquela de todos os algoritmos baseados em caminho de aumento (EDMONDS, 1965; GABOW, 1976; LAWLER, 2001; MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991), quando estes são aplicados a grafos cúbicos e sem pontes.

Mais recentemente, Diks e Stanczyk (DIKS; STANCZYK, 2010), levando em conta algumas propriedades dos grafos cúbicos e sem pontes e utilizando duas estruturas de dados dinâmicas, foram capazes de alterar o algoritmo em (BIEDL et al., 2001) e torná-lo ainda mais eficiente. Com uma complexidade (amortizada) de tempo $\mathcal{O}(n \lg^2 n)$, o algoritmo proposto é, até onde se saiba, o algoritmo mais eficiente para calcular emparelhamentos perfeitos em grafos cúbicos e sem pontes. O trabalho aqui exposto tem como objetivo implementá-lo.

A principal contribuição deste trabalho consiste, justamente, na implementação do algoritmo em (DIKS; STANCZYK, 2010), pois não se encontrou uma implementação disponível publicamente. Outra contribuição do trabalho é a apresentação das análises dos algoritmos que atuam sobre as várias estruturas de dados utilizadas, que são mais detalhadas e esclarecedoras do que as presentes nos próprios artigos onde tais estruturas são descritas. Finalmente, este trabalho apresentou ao autor as oportunidades de estudar um problema atual na área de processamento geométrico, aprofundar-se em um tópico importante em teoria dos grafos (isto é, emparelhamento) e de ganhar experiência em pesquisa.

1.3 Organização

O restante do texto está organizado em mais seis capítulos e dois apêndices como segue:

- o Capítulo 2 apresenta várias definições necessárias para entender a declaração formal do problema-alvo do trabalho;
- o Capítulo 3 apresenta uma revisão da bibliografia e declara o problema-alvo;
- os Capítulos 4 e 5 tratam, respectivamente, das Árvores Splay e das Árvores Dinâmicas, ambas desenvolvidas por Sleator e Tarjan;
- o Capítulo 6 tem como objeto de estudo uma estrutura de dados para o problema da conectividade dinâmica em grafos;
- o Capítulo 7 detalha as etapas envolvidas na implementação do algoritmo e a estrutura do código, apresenta os resultados dos testes realizados com o código e avalia os resultados obtidos;
- o Capítulo 8 sintetiza os pontos mais importantes do texto e destaca algumas possíveis continuações deste trabalho;
- o Apêndice A descreve, detalhadamente, a prova de Frink para o teorema de Petersen (caso particular e geral);
- o Apêndice B revisa o método do potencial para análise de complexidade amortizada.

2 Preliminares

Este capítulo apresenta a terminologia e a notação utilizadas ao longo do texto, assim como uma série de definições necessárias ao entendimento do problema-alvo do trabalho proposto. Em particular, a Seção 2.1 introduz conceitos elementares de Teoria dos Grafos relevantes para este trabalho. O propósito em introduzir tais conceitos é apresentar e uniformizar a terminologia e a notação utilizadas. A Seção 2.2 define emparelhamentos em grafos, enfatizando o tipo de emparelhamento com o qual esta monografia lida, e também discute vários resultados importantes relacionados a emparelhamentos de cardinalidade máxima e emparelhamentos perfeitos. Finalmente, a Seção 2.3 apresenta o teorema de Petersen, que é a base dos algoritmos para emparelhamento perfeito estudados no Capítulo 3. O conteúdo das seções 2.1 e 2.2 é uma compilação do material encontrado em (CLARK; HOLTON, 1991; BONDY; MURTY, 2010; WEST, 2002; DIESTEL, 2000; GALLIER, 2011).

2.1 Terminologia e notação

Esta monografia trata de algoritmos para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. Um teorema clássico de Teoria dos Grafos, conhecido como teorema de Petersen, garante a existência de emparelhamentos perfeitos neste tipo de grafo. Esta seção apresenta definições básicas relacionadas a grafos, tais como grafos cúbicos, pontes, pontos de articulação e biconectividade, e descreve alguns resultados importantes que relacionam esses conceitos com certas estruturas de um grafo, tais como caminhos disjuntos e ciclos.

Definição 2.1.1. Um *grafo* é uma tripla $G = (V, E, st)$, onde V é um conjunto de *vértices* (ou *nós*), E é um conjunto de *arestas* disjunto de V, e $st : E \to V \cup [V]^2$ é uma *função de incidência* que atribui um par de vértices em $[V]^2$ ou um vértice em V a cada aresta, onde

$$[V]^2 = \big\{\{u, v\} \in V \times V \mid u \neq v\big\}.$$

Se e é uma aresta em E tal que $st(e) = \{u, v\}$ (resp. $st(e) = u$), então se diz que e conecta u a v (resp. u a ele próprio) e que u e v são os vértices *extremos* de e (resp. u é o *extremo* de e).

Definição 2.1.2. Dado um grafo G, para cada aresta $e \in E$, tal que $st(e) = \{u, v\}$, diz-se que:

(i) Os vértices u e v são *adjacentes*;

(ii) Os vértices u e v são *incidentes* em e;

(iii) A aresta e é *incidente* nos vértices u e v;

(iv) Duas arestas, $e, e' \in E$ são *adjacentes* se, e somente se, elas incidem em um mesmo vértice.

Note que a função st não é necessariamente injetiva ou sobrejetiva, o que permite grafos com "vértices isolados" (isto é, vértices que não são extremos de nenhuma aresta) e "arestas paralelas". Duas arestas $e, e' \in E$ são ditas *paralelas* se, e somente se, $st(e) = st(e')$. Se exatamente m arestas de G incidem sobre os mesmos dois vértices ou sobre o mesmo vértice, diz-se que cada uma delas possui multiplicidade m. Arestas de multiplicidade um, dois e três são denominadas, respectivamente, arestas *simples*, *duplas* e *triplas*. A Definição 2.1.1 também permite laços em um grafo. Um *laço* é uma aresta $e \in E$ tal que $st(e) \in V$.

Definição 2.1.3. Um grafo G é dito ser *simples* se, e somente se, o grafo G não possui arestas paralelas nem laços, ou seja, se, e somente se, toda aresta de G for uma aresta simples.

Denotam-se os números de vértices e arestas de um grafo G por $v(G)$ e $e(G)$, respectivamente. Os números, $v(G)$ e $e(G)$, são denominados a *ordem* e o *tamanho* de G, respectivamente. Quando $v(G)$ e $e(G)$ são finitos, o grafo G é dito *finito*. Este trabalho lida apenas com grafos finitos. Portanto, assume-se, de agora em diante, que todo grafo é finito.

Definição 2.1.4. Seja $G = (V, E, st)$ um grafo. Então, para cada $u \in V$, o *grau*, $d_G(u)$, de u é

$$d_G(u) = |\{e \in E \mid u \in st(e)\}| + 2 \cdot |\{e \in E \mid u = st(e)\}|.$$

O grau de um vértice também é conhecido por *valência* do vértice e nada mais é do que o número de arestas que incidem no vértice, tal que cada laço (se algum) é contado duas vezes.

Definição 2.1.5. Um grafo G onde todos os vértices possuem o mesmo grau, k, é denominado *k-regular* ou, simplesmente, *regular*. Um grafo 3-regular também é chamado de *cúbico*.

Definição 2.1.6. Dados dois grafos, $G = (V, E, st)$ e $G' = (V', E', st')$, o grafo G' é dito *subgrafo* de G se, e somente se, $V' \subset V$, $E' \subset E$, e st' é a restrição de st a E'. Se G' é um subgrafo de G e $V' = V$, então G' é chamado de *subgrafo gerador* de G. Dado qualquer subconjunto V' de V, o *subgrafo induzido*, $G[V']$, de G é o grafo $G[V'] = (V', E_{V'}, st')$ tal que

$$E_{V'} = \left\{ e \in E \mid st(e) \in V' \text{ ou } st(e) \in [V']^2 \right\} .$$

Definição 2.1.7. Seja G um grafo qualquer. Um grafo G' é chamado *k-fator* de G se, e somente se, o grafo G' for k-regular e subgrafo gerador de G, onde k é um inteiro não-negativo.

Definição 2.1.8. Dados um grafo $G = (V, E, st)$ e quaisquer dois vértices $u, v \in V$, um *caminho* em G de u para v, ou simplesmente, um *caminho $u - v$* é uma sequência finita, $v_0 e_1 v_1 e_2 v_2 \cdots v_{n-1} e_n v_n$, tal que $n \in \mathbb{N}$, $v_i \in V$, $e_j \in E$, $v_0 = u$, $v_n = v$ e $e_j = \{v_{j-1}, v_j\}$, para todo $j \in \{1, \ldots, n\}$ e para todo $i \in \{0, \ldots, n\}$. O *comprimento* do caminho π é denotado por $|\pi| = n$. Quando $n = 0$, tem-se um *caminho nulo* de u para u. Se $u = v$, então o caminho π é denominado de *caminho fechado*; caso contrário, ele é denominado de *caminho aberto*. Finalmente, um caminho é dito *simples* se, e somente se, nenhum vértice do caminho ocorre duas vezes, com a possível exceção do vértice u (se o caminho for fechado).

A seguinte proposição estabelece que a existência de um caminho $u - v$ em um grafo G é suficiente para garantir que existe um caminho $u - v$ simples em G (CLARK; HOLTON, 1991):

Proposição 2.1.9. Dados quaisquer dois vértices, u e v, de um grafo G, todo caminho $u - v$ em G contém um caminho $u - v$ simples em G; isto é, dado qualquer caminho $P = u e_1 v_1 \cdots v_{n-1} e_n v$ no grafo G, após algumas remoções de vértices e arestas de P (se for necessário), obtém-se uma subsequência, Q, de P que é um caminho $u - v$ simples em G.

A seguinte proposição estabalece outro fato relacionado a caminhos em grafos. Este fato, demonstrado em (KÖNIG, 1990), é utilizado pela prova do teorema de Frink dada no Apêndice A:

Proposição 2.1.10. Sejam u, v e w três vértices distintos de um grafo G tais que há um caminho (aberto), P, em G que conecta u a v, e um caminho (aberto), Q, em G que conecta v a w. Então, há um caminho simples em G de u a w cujas arestas pertencem a P ou Q.

Definição 2.1.11. Seja G um grafo. Um *ciclo* em G é um caminho fechado em G no qual todas as arestas são distintas. O ciclo é *simples* se, e somente se, ele for um caminho fechado simples. Um ciclo de comprimento k, isto é, um ciclo com k arestas, é dito um *k-ciclo*.

Definição 2.1.12. Seja $G = (V, E, st)$ um grafo e C_G a relacão binária sobre V definida da seguinte maneira: para cada $u, v \in V$, tem-se que uC_Gv se, e somente se, existe um caminho $u - v$ em G. Quando isso ocorre, diz-se que os vértices u e v estão *conectados*. Observe que a relação C_G é uma relação de equivalência (reflexiva, simétrica e transitiva), e suas classes de equivalência são denominadas *componentes conexas* de G. O número de componentes conexas de G é denotado por $\omega(G)$. Um grafo G é *conexo* se, e somente se, existe um caminho em G entre quaisquer dois vértices, u e v, de G, ou seja, se, e somente se,

$$\omega(G) = 1\,.$$

Seja $G = (V, E, st)$ um grafo qualquer. Se e é uma aresta de G, então $G - e$ denota o grafo obtido pela remoção da aresta e de G: $G - e = (V, E - \{e\}, st_e)$, onde $st_e(f) = st(f)$, para toda aresta $f \in E - \{e\}$. De forma análoga, se v é um vértice de G, então $G - v$ denota o grafo obtido pela remoção do vértice v e de todas as arestas incidentes nele: $G - v = (V - \{v\}, E_v, st_v)$, onde $E_v = \{e \in E \mid v \notin st(e) \text{ ou } v \neq st(e)\}$ e $st_v(f) = st(f)$, para toda aresta $f \in E_v$. Finalmente, se $P = v_0e_1v_1e_2 \cdots v_{n-1}e_nv_n$ é um caminho em G e $e = \{v_n, v_{n+1}\}$ é qualquer aresta em G incidente sobre v_n, então $P + e$ denota o caminho

$$v_0e_1v_1e_2 \cdots v_{n-1}e_nv_nev_{n+1}\,,$$

que se obtém a partir de $P = v_0e_1v_1e_2 \cdots v_{n-1}e_nv_n$ com a inclusão da aresta e ao final de P.

Definição 2.1.13. Uma aresta e de um grafo G é uma *ponte* (ou *aresta de corte*) de G se, e somente se, $G - e$ possuir mais componentes conexas do que G; isto é, se, e somente

se,

$$\omega(G - e) > \omega(G)\,.$$

O seguinte teorema estabelece uma relação entre pontes e ciclos do grafo (CLARK; HOLTON, 1991):

Teorema 2.1.14. Uma aresta e de um grafo G é uma ponte se, e somente se, a aresta e não pertence a nenhum ciclo em G.

Definição 2.1.15. Um vértice v de um grafo G é denominado *ponto de articulação* de G se, e somente se, $G - v$ possuir mais componentes conexas do que G; isto é, se, e somente se,

$$\omega(G - v) > \omega(G)\,.$$

Definição 2.1.16. Seja G um grafo simples. A *conectividade (de vértices)* de G, denotada por $\kappa(G)$, é o menor número de vértices em G cuja remoção faz com que o grafo resultante tenha mais componentes conexas do que G ou que seja isomorfo ao grafo K_1 (isto é, o grafo resultante é um grafo que consiste de um único vértice e que não possui nenhuma aresta).

Definição 2.1.17. Seja G um grafo simples. Então, diz-se que G é *n-conexo* se, e somente se,

$$\kappa(G) \geq n\,,$$

onde $n \in \mathbb{N}$. Isto é, um grafo simples G é n-conexo se, e somente se, devem-se remover pelo menos n vértices de G para se obter um grafo com mais componentes conexas do que G ou K_1.

Note que um grafo simples G é 1-conexo se, e somente se, o grafo G é conexo e possui pelo menos dois vértices. Além disso, o grafo G é 2-conexo (biconexo ou biconectado) se, e somente se, o grafo G é conexo, contém pelo menos três vértices e nenhum ponto de articulação.

Definição 2.1.18. Sejam u e v dois vértices de um grafo G. Dois caminhos $u - v$, P e Q, em G são ditos *disjuntos (internamente)* se, e somente se, os únicos vértices comuns a P e Q são u e v.

Teorema 2.1.19 (**Teorema de Whitney, 1932**)**.** Seja G um grafo simples com pelo menos três vértices. Então, o grafo G é biconexo se, e somente se, para cada par de vértices distintos, u e v, de G houver dois caminhos $u - v$ em G que são disjuntos internamente.

Uma prova para o Teorema 2.1.19 pode ser encontrada em (CLARK; HOLTON, 1991). Uma consequência imediata deste teorema implica que um grafo biconexo não pode conter nenhuma ponte, pois todo par de vértices do grafo deve pertencer a um ciclo simples do grafo e uma ponte não pode pertencer a nenhum ciclo do grafo, como afirma o Teorema 2.1.14.

Corolário 2.1.20. Sejam u e v dois vértices de um grafo G biconexo. Então, há um ciclo simples em G que contém os vértices u e v.

Uma prova para o Corolário 2.1.20 também pode ser encontrada em (CLARK; HOLTON, 1991).

Os algoritmos que serão apresentados no Capítulo 3 fazem uso de uma estrutura de dados que suporta consulta, de forma eficiente, a pares de vértices de um grafo com respeito a um tipo um pouco menos restrito de conectividade do que aquela definida em 2.1.16:

Definição 2.1.21. Seja G um grafo simples com pelo menos dois vértices. Então, a *conectividade de aresta* de G, denotada por $\lambda(G)$, é o menor número de arestas em G cuja remoção gera um grafo desconexo. Em particular, tem-se que $\lambda(G) = 0$ se G não é um grafo conexo. Finalmente, o grafo G é dito *n-aresta-conexo* se, e somente se, $\lambda(G) \geq n$, onde $n \in \mathbb{N}$.

Note que qualquer grafo, G, simples que possui uma ponte é tal que $\lambda(G) = 1$. Além disso, tem-se que $\lambda(G) = 0$ se, e somente se, o grafo G não é conexo e possui mais de um vértice. Note também que se G é n-conexo, então G também é n-aresta-conexo, mas nem todo grafo n-aresta-conexo é n-conexo, como comprova o exemplo ilustrado na Figura 2.1. Em particular, pode-se mostrar que a relação $\kappa(G) \leq \lambda(G) \leq \delta(G)$ é satisfeita para todo grafo, G, simples com pelo menos dois vértices, onde $\delta(G)$ denota o grau do vértice de menor grau do grafo G (DIESTEL, 2000). O seguinte teorema fornece uma caracterização de grafos n-arestas-conexos em termos do número de caminhos que não compartilham arestas:

Teorema 2.1.22. Um grafo simples, G, com pelo menos dois vértices é n-aresta-conexo se, e somente se, para quaisquer dois vértices distintos, u e v, de G, houver pelo menos n caminhos $u - v$ (não necessariamente simples) aresta-disjuntos em G, ou seja, caminhos $u - v$ sem arestas em comum (dois a dois).

Uma prova para o Teorema 2.1.22 pode ser encontrada em (CLARK; HOLTON, 1991). Uma consequência importante deste teorema para o presente trabalho é a seguinte: se G

é um grafo 2-aresta-conexo, então G não pode possuir uma ponte, pois, de acordo com o Teorema 2.1.22, há pelo menos dois caminhos aresta-disjuntos em G unindo os dois extremos de cada aresta. Logo, toda aresta de G pertence a um ciclo (simples) no grafo G. Note que se G é cúbico e possui mais de dois vértices, então 2-aresta-conectividade implica 2-conectividade, pois se houvesse um ponto de articulação em um grafo 2-aresta-conexo, G, então uma das arestas incidentes no ponto de articulação seria uma ponte. Mas, sendo G 2-aresta-conexo, isso é impossível. Grafos 2-aresta-conexos são de grande interesse aqui.

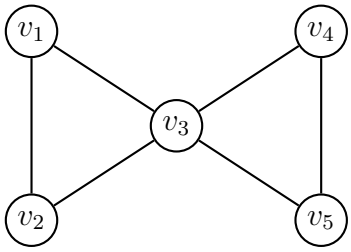


Figura 2.1: Um grafo 2-aresta-conexo que não é 2-conexo.

Finalmente, têm-se as definições de árvore, floresta e árvore geradora:

Definição 2.1.23. Uma *árvore* é um grafo conexo e acíclico (isto é, que não possui ciclos). Uma *floresta* é um grafo cujas componentes conexas são árvores. Uma *árvore geradora* de um grafo G é um subgrafo gerador de G que também é uma árvore, ou seja, conexo e acíclico.

2.2 Emparelhamentos

Como dito antes, esta monografia descreve um algoritmo para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. Esta seção é dedicada à formalização da noção de emparelhamento em grafos e alguns fatos importantes relacionados a esta noção. Informalmente, um emparelhamento envolve a formação de pares disjuntos de vértices de um grafo, "casando-os" dois a dois. O emparelhamento é perfeito quando todos os vértices do grafo estiverem "casados" de forma que cada vértice faça parte de exatamente um par.

Definição 2.2.1. Dado um grafo $G = (V, E, st)$, um *emparelhamento* M em G é um subconjunto de arestas de E tal que quaisquer duas arestas distintas em M não possuem extremos em comum (ou seja, não são adjacentes) ou, equivalentemente, tal que cada vértice $v \in V$ seja incidente em, no máximo, uma aresta em M. Diz-se que um vértice

$v \in V$ está *emparelhado* (ou *saturado com respeito a* M) se, e somente se, ele é incidente em alguma aresta em M, e é dito *não emparelhado* (*não saturado* ou *livre*) caso contrário.

A Figura 2.2 ilustra um emparelhamento em um grafo com quatro vértices e seis arestas. As arestas do emparelhamento são mostradas em vermelho e as arestas que não estão no emparelhamento são mostradas em azul. Usar-se-á esta convenção de cores daqui em diante.

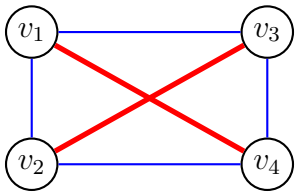


Figura 2.2: Exemplo de emparelhamento.

Um emparelhamento, M, em um grafo G é um conjunto de arestas. Logo, a *cardinalidade* (ou *tamanho*) de M corresponde ao número de arestas que ele contém. Assim sendo, uma maneira de construir um emparelhamento é escolhendo, iterativamente, arestas cujos extremos ainda não estejam saturados, até que nenhuma outra aresta possa ser adicionada. Um emparelhamento construído dessa maneira é definido por (WEST, 2002) como *maximal*, não sendo necessariamente de *cardinalidade máxima*, conforme a definição abaixo:

Definição 2.2.2. Seja $G = (V, E, st)$ um grafo, e $\mathcal{M}$ o conjunto de todos os emparelhamentos em G. Um emparelhamento M em $\mathcal{M}$ é de *cardinalidade máxima* se, e somente se,

$$|M| \geq |M'|\,,$$

para todo $M' \in \mathcal{M}$, onde $|A|$ é a notação de cardinalidade de A, para qualquer conjunto A.

Definição 2.2.3. Seja $G = (V, E, st)$ um grafo. Um emparelhamento M em G é um *emparelhamento perfeito* se todos os vértices de G estão saturados (emparelhados) com respeito a M.

O emparelhamento da Figura 2.2 é perfeito.

Observe que todo emparelhamento perfeito é um emparelhamento de cardinalidade máxima, mas nem todo emparelhamento de cardinalidade máxima é perfeito. Quando um

emparelhamento de cardinalidade máxima, M, não é perfeito, o grafo não admite um emparelhamento perfeito. Caso contrário, haveria um emparelhamento (isto é, o emparelhamento perfeito) de cardinalidade maior que $|M|$, o que contradiz o fato da cardinalidade, $|M|$, de M ser máxima. Observe também que um subgrafo gerador, G', de um grafo G, é um 1-fator de G se, e somente se, o conjunto de arestas de G' é um emparelhamento perfeito em G.

Definição 2.2.4. Seja M um emparelhamento em um grafo G. Então, um *caminho alternante com respeito a* M é um caminho simples cujas arestas estão alternadamente nos conjuntos $E - M$ e M, isto é, um caminho simples que alterna arestas livres e ocupadas com respeito a M. Caso os vértices extremos desse caminho sejam livres com respeito a M, diz-se também que o caminho é *um caminho de aumento com respeito a* M. Assim, as arestas extremas de um caminho de aumento também são livres com respeito ao emparelhamento.

O teorema a seguir estabelece uma importante relação entre emparelhamentos de cardinalidade máxima e caminhos de aumento. Esta relação é a base de praticamente todos os algoritmos para encontrar emparelhamentos de cardinalidade máxima em grafos arbitrários:

Teorema 2.2.5 (**Berge, 1957**)**.** Seja G um grafo sem laços e M um emparelhamento em G. Então, M é máximo se, e somente se, G não possui nenhum caminho de aumento com respeito a M.

A prova dada por Berge pode ser encontrada em (CLARK; HOLTON, 1991). O interessante é que a prova é construtiva e fornece um algoritmo para que um emparelhamento de cardinalidade máxima seja encontrado. De maneira geral, o algoritmo consiste em procurar por caminhos de aumento no grafo com respeito a um emparelhamento qualquer, M. Caso um caminho, P, de aumento seja encontrado, o emparelhamento M é substituído por outro,

$$M' = (M - P) \cup (P - M),$$

de maior cardinalidade. Em seguida, repete-se a busca por outro caminho de aumento no grafo com respeito a M, com $M = M'$. Caso nenhum caminho de aumento com respeito a M exista, o Teorema 2.2.5 garante que o emparelhamento atual, M, possui cardinalidade máxima.

Basicamente, todos os algoritmos conhecidos para encontrar emparelhamentos de cardinalidade máxima em um grafo qualquer, G, se distinguem pela forma como a operação

fundamental da prova de Berge, *encontrar um caminho de aumento com respeito ao emparelhamento atual*, é realizada. O primeiro algoritmo determinístico de tempo polinomial para encontrar um tal emparelhamento foi dado por Jack Edmonds (EDMONDS, 1965) e possui complexidade $\mathcal{O}(v(G)^4)$. O trabalho de Edmonds inspirou uma série de novos algoritmos e implementações computacionais robustas, que são de grande valor em aplicações práticas.

Em 1976, Gabow e Lawler mostraram, independentemente, que implementações mais cuidadosas do algoritmo de Edmonds possuem complexidade de tempo $\mathcal{O}(v(G)^3)$ (GABOW, 1976; LAWLER, 2001). A partir daí, muitas outras modificações que simplificaram e reduziram a complexidade de tempo do algoritmo foram propostas. Atualmente, os algoritmos propostos por Micali e Vazirani (MICALI; VAZIRANI, 1980), Blum (BLUM, 1990) e Gabow e Tarjan (GABOW; TARJAN, 1991), todos com complexidade de tempo $\mathcal{O}(e(G) \cdot \sqrt{v(G)})$, ainda são o que há de melhor em termos de algoritmos determinísticos para o problema de encontrar emparelhamentos de cardinalidade máxima em grafos quaisquer.

2.3 O teorema de Petersen

Como já foi dito diversas vezes, o interesse desta monografia é por grafos cúbicos e sem pontes. Tais grafos gozam de uma propriedade importante: eles admitem um emparelhamento perfeito. Este fato foi provado pelo matemático dinamarquês, Julius Peter Christian Petersen, em 1891 (PETERSEN, 1891) e ficou bastante conhecido como o teorema de Petersen:

Teorema 2.3.1. Todo grafo cúbico e sem pontes admite um emparelhamento perfeito.

Há inúmeras demonstrações deste teorema. Uma das mais conhecidas e elegantes pode ser facilmente obtida a partir de um resultado provado por William Tutte, em 1947 (DIESTEL, 2000). No entanto, esta demonstração e muitas outras não são construtivas e, portanto, não fornecem um algoritmo que possa ser utilizado para se encontrar um emparelhamento perfeito, como é o caso da prova dada por Berge para o Teorema 2.2.5. Uma exceção é a prova dada por Orrin Frink Jr. em 1926 (FRINK JR., 1926). A prova de Frink para o teorema de Petersen se baseia em um resultado, denominado aqui de teorema de Frink, que, por sua vez, faz uso de uma operação local em grafos conhecida como redução de aresta.

Seja $e = \{u, w\}$ uma aresta de um grafo qualquer, G, que 1) conecta dois vértices distintos de grau 3 cada e 2) não é uma aresta paralela, como mostra a Figura 2.3. As arestas e_1 e e_2 são incidentes no vértice u e as arestas e_3 e e_4 são incidentes no vértice w. Removem-se de G os vértices u e w e as cinco arestas, e, e_1, e_2, e_3 e e_4, neles incidentes e adicionam-se duas novas arestas, e_{13} e e_{24}, conectando o vértice x_1 ao vértice x_3 e o vértice x_2 ao vértice x_4, respectivamente. Denomina-se o grafo resultante de G_1. Se, por outro lado, as arestas adicionadas são e_{14} e e_{23}, que conectam o vértice x_1 ao vértice x_4 e o vértice x_2 ao vértice x_3, respectivamente, então o grafo resultante é denominado G_2. Diz-se que os dois grafos, G_1 e G_2, *originam-se de G a partir da redução da aresta $e = \{u, w\}$*.

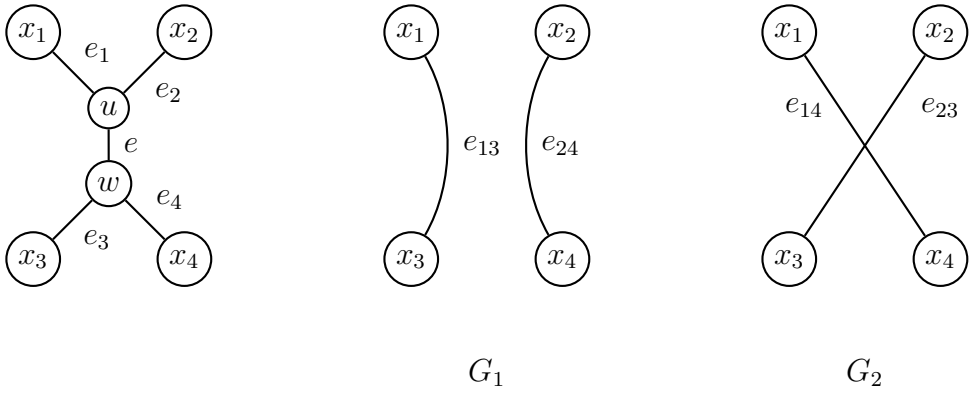


Figura 2.3: Redução da aresta $e = \{u, w\}$.

O teorema de Frink pode ser agora enunciado como segue:

Teorema 2.3.2 (**Teorema de Frink**)**.** Seja G um grafo conexo, cúbico e sem pontes e seja e uma aresta de G que não pertence a um 2-ciclo em G. Sejam G_1 e G_2 os dois grafos cúbicos que se originam de G a partir da redução da aresta e. Então, pelo menos um desses dois grafos, G_1 ou G_2, é, ao mesmo tempo, um grafo conexo, cúbico e sem pontes.

A prova de Frink para o Teorema 2.3.2 está detalhada no Apêndice A. Embora a prova deste teorema utilize o método de redução ao absurdo (ou seja, contradição), ela faz uso de uma operação construtiva: a redução de aresta. Usando esta operação e o Teorema 2.3.2, Frink provou o teorema de Petersen. Mais recentemente, Bield, Demaine, Bose e Lubiw perceberam que a operação de redução de aresta poderia ser utilizada para se obter um algoritmo para determinar emparelhamentos perfeitos em grafos cúbicos e sem pontes (BIEDL et al., 2001). Este algoritmo, que é descrito em detalhes no capítulo que segue, baseou-se também em outras passagens da prova de Frink para o teorema de Petersen.

A importância do algoritmo em (BIEDL et al., 2001) é que a complexidade de tempo dele é assintoticamente menor do que a dos algoritmos para encontrar emparelhamentos de cardinalidade máxima em grafos quaisquer. Logo, se a classe de grafos que se quer estudar for a dos grafos cúbicos e sem pontes, como é o caso aqui, os algoritmos mencionados na seção anterior, isto é, os algoritmos em (MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991), não são os mais indicados. Em 2010, Diks e Stanczyk modificaram o algoritmo em (BIEDL et al., 2001) e obtiveram um algoritmo de complexidade de tempo ainda menor (DIKS; STANCZYK, 2010). Este algoritmo é o principal objeto de estudo desta monografia.

3 Revisão Bibliográfica

Este capítulo revisa os dois principais trabalhos que motivaram o presente trabalho de final de curso. Os dois trabalhos fornecem os algoritmos mais eficientes e conhecidos na literatura para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. Como visto na Seção 2.3, tais grafos sempre admitem um emparelhamento perfeito e este fato é uma decorrência do teorema de Petersen. Os algoritmos em questão foram inspirados na prova desenvolvida por Frink (veja o Apêndice A) para o teorema de Petersen (FRINK JR., 1926).

Em particular, a prova de Frink para o teorema de Petersen depende de um teorema provado por Frink, chamado aqui de teorema de Frink (veja o Teorema 2.3.2). A prova deste teorema é inerentemente algorítmica, mas o algoritmo que se deriva diretamente dela não é mais eficiente do que os algoritmos mais eficientes para encontrar emparelhamentos de cardinalidade máxima em grafos quaisquer (MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991). No entanto, Biedl, Bose, Demaine e Lubiw (BIEDL et al., 2001) identificaram os "gargalos" do algoritmo resultante da prova de Frink e propuseram modificações que resultaram em um algoritmo com complexidade assintoticamente menor do que a dos algoritmos em (MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991).

Mais recentemente, Diks e Stanczyk (DIKS; STANCZYK, 2010) foram capazes de reduzir, ainda mais, a complexidade de tempo do algoritmo proposto por Biedl e seus colegas (BIEDL et al., 2001), obtendo um algoritmo mais eficiente para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes; de fato, o algoritmo mais eficiente (para o problema em questão) que se tem conhecimento na literatura até o presente momento.

A Seção 3.1 descreve o algoritmo resultante da prova de Frink para o teorema de Petersen, enfatizando as duas operações do algoritmo que são responsáveis pela cota inferior da complexidade de tempo de pior caso do algoritmo. A Seção 3.2 discute o

algoritmo proposto por Biedl, Demaine, Bose e Lubiw em (BIEDL et al., 2001), detalhando a estratégia usada pelos autores para reduzir a complexidade do algoritmo resultante da prova de Frink. A Seção 3.3 apresenta o algoritmo proposto por Diks e Stanczyk em (DIKS; STANCZYK, 2010), que consiste em uma melhoria do algoritmo proposto em (BIEDL et al., 2001).

3.1 O algoritmo de Frink

Seja G um grafo cúbico e 2-aresta-conexo. Como visto na Seção 2.1, se um grafo é 2-aresta-conexo isso implica que ele também é conexo e sem pontes. Por sua vez, como o grafo é cúbico, a inexistência de pontes implica a inexistência de laços, embora o grafo possa possuir arestas duplas e triplas. Mas, como o grafo G é conexo, se G possuir uma aresta tripla, então o grafo G é um grafo com apenas dois vértices e três arestas (triplas), como ilustra a Figura 3.1. Por outro lado, se o grafo G não possui aresta tripla, então G possui uma aresta simples, pois se todas as arestas do grafo fossem duplas, o grafo não seria cúbico.

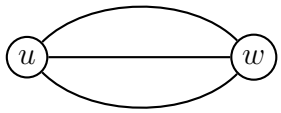


Figura 3.1: Um grafo cúbico e 2-aresta-conexo com arestas triplas que possui apenas 2 vértices.

As observações do parágrafo anterior, juntamente com o teorema de Frink (isto é, o Teorema 2.3.2), permitem a construção de um algoritmo recursivo para encontrar um emparelhamento perfeito em G. Lembre-se de que o teorema de Frink diz que *se e é uma aresta simples de G, então pelo menos um dos dois grafos, G_1 e G_2, que se originam em G a partir da redução de e é, ao mesmo tempo, conexo, cúbico e sem pontes.* Então, a idéia por trás do algoritmo baseado na prova dada por Frink para o teorema de Petersen é:

- Se G é isomorfo ao grafo da Figura 3.1, então um emparelhamento perfeito, M, de G é definido escolhendo-se uma das três arestas triplas. Este é o caso base da recursão.
- Caso contrário, sabe-se que G possui uma aresta simples, e. Aplicam-se as duas reduções possíveis de e e obtêm-se dois grafos cúbicos, G_1 e G_2, um para cada

possível escolha de redução (veja a Figura 2.3). O teorema de Frink garante que um desses dois grafos é conexo, cúbico e sem pontes (e, portanto, 2-aresta-conexo). Identifique um dos grafos, G_1 ou G_2, que é conexo, cúbico e sem pontes. Denote o grafo encontrado por G'.

- Execute o algoritmo para a entrada $G = G'$. Seja M' o emparelhamento perfeito de G' devolvido na chamada recursiva. Retire de M' as arestas de G' que não pertencem a G. Defina $M = M'$ e "estenda" M para torná-lo um emparelhamento perfeito de G.

Há duas operações críticas no algoritmo acima: (a) identificar um dos grafos, G_1 ou G_2, que é conexo, cúbico e sem pontes e (b) estender M para torná-lo um emparelhamento perfeito de G. A existência de um grafo em $\{G_1, G_2\}$ que seja conexo, cúbico e sem pontes é garantida pelo Teorema 2.3.2. Por sua vez, a existência de um emparelhamento perfeito, M, de G é garantida pelo teorema de Petersen. Logo, necessita-se detalhar como as operações (a) e (b) podem ser efetuadas e qual é a complexidade de tempo de cada uma.

De forma pouco eficiente, pode-se realizar a operação (a) com uma simples adaptação de um algoritmo para busca em profundidade em grafos, o qual conta quantas componentes biconexas há no grafo (TARJAN, 1972). Neste caso, a complexidade da operação (a) se torna

$$\mathcal{O}(v(G) + e(G)) = \mathcal{O}(v(G))\,,$$

pois $e(G) = \Theta(v(G))$ para um grafo cúbico G. Para a operação (b), suponha, sem perda de generalidade, que o grafo G_1 tenha sido o grafo identificado como conexo, cúbico e sem pontes pelo algoritmo. Isto implica que há exatamente três possíveis situações com respeito às arestas e_{13} e e_{24} originadas a partir da redução da aresta e de G e o emparelhamento M' de $G' = G_1$ (veja a Figura 3.2): (i) as duas arestas, e_{13} e e_{24}, estão em M', (ii) as duas arestas, e_{13} e e_{24}, *não* estão em M' e (iii) apenas uma das duas arestas, e_{13} e e_{24}, está em M'.

Os casos (ii) e (iii) são triviais. De fato, no caso (ii), basta inserir a aresta $\{u, w\}$ em $M = M'$ para se obter um emparelhamento perfeito de G. No caso (iii), se $e_{13} \in M'$, então basta inserir as arestas $\{u, x_1\}$ e $\{w, x_3\}$ em $M = M' - \{e_{13}\}$ para se obter um emparelhamento perfeito de G. De forma análoga, se $e_{24} \in M'$, então basta inserir as arestas $\{u, x_2\}$ e $\{w, x_4\}$ em $M = M' - \{e_{24}\}$ para se obter um emparelhamento perfeito de G. Observe que, tanto em (ii) quanto em (iii), M pode ser obtido de M' em tempo

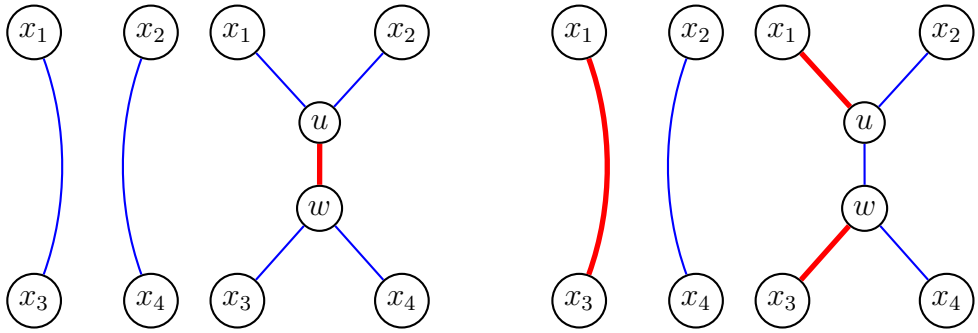


Figura 3.2: As configurações (ii) e (iii) e as respectivas reduções sendo desfeitas.

constante.

O caso "problemático" é o caso (i). Neste caso, apela-se para um resultado intermediário da prova dada por Frink para o teorema de Petersen: *se o grafo G é conexo, cúbico e sem pontes e M é um emparelhamento perfeito de G, então toda aresta, e, de G está contida em um ciclo alternante de G com respeito a M* (veja o Lema A.3.1 no Apêndice A). Então, escolhe-se uma das duas arestas, e_{13} ou e_{24}, em M e procura-se por um ciclo alternante que a contém. Uma vez que este ciclo, C, seja encontrado, redefine-se M':

$$M' = (M' - C) \cup (C - M') \,.$$

Observe que o novo M' ainda é um emparelhamento perfeito de G'. No entanto, tem-se, agora, que $e_{13}, e_{24} \notin M'$ ou exatamente uma delas está em M'. Logo, o problema é reduzido para o caso (ii) ou o caso (iii) e, a partir daí, procede-se como explicado anteriormente. O seguinte teorema diz que o ciclo C pode ser encontrado em tempo linear em $v(G)$ (BIEDL et al., 2001):

Teorema 3.1.1. Sejam G, M e e um grafo cúbico e sem pontes, um emparelhamento perfeito de G e uma aresta qualquer de G, respectivamente. Então, pode-se encontrar, em tempo $\mathcal{O}(v(G))$, um ciclo, C, alternante em G com respeito ao emparelhamento M tal que $e \in C$.

Demonstração. Há duas possíveis situações para e com respeito a M: $e \in M$ ou $e \notin M$. Se $e \in M$ então remova e de G. Pelo Lema A.3.1, sabe-se que há um ciclo, C, alternante em G com respeito a M tal que $e \in C$. Logo, há um caminho de aumento em $G - e$ com respeito a $M - \{e\}$ que conecta os vértices extremos, x e y, de e. Como todos os vértices de $G - e$, com exceção de x e y, estão emparelhados com respeito a $M - \{e\}$, tal caminho pode ser encontrado em tempo linear em $\mathcal{O}(v(G-e))$ utilizando-se, por exemplo,

o algoritmo em (GABOW; TARJAN, 1985) na entrada $(G-e, M-\{e\}, x, y)$ para encontrar caminhos de aumento conectando dois vértices quaisquer de um grafo. Por outro lado, se $e \notin M$ então sejam e_1 e e_2 as outras duas arestas incidentes em um extremo, x, de e. Uma delas, diga-se e_1, deve pertencer a M, pois M é perfeito e, portanto, o vértice x deve ser saturado com respeito a M. Logo, nenhum ciclo alternante com respeito a M pode conter e e e_2. Isto implica que um ciclo, C, alternante, com respeito a M, que contenha e tem de conter e_1. Logo, o ciclo C também é um ciclo alternante em $G - e_2$ com respeito ao emparelhamento perfeito, M, de $G-e_2$. O ciclo C pode ser encontrado se um caminho de aumento, P, com respeito a $M - \{e_1\}$, for procurado no grafo $G - \{e_1, e_2\}$, que é o grafo obtido pela remoção das arestas e_1 e e_2 de G. De fato, o caminho P tem de conectar x ao extremo de e_1 distinto de x, pois todos os demais vértices de $G - \{e_1, e_2\}$ estão saturados com respeito a $M - \{e_1\}$. Como e é a única aresta incidente em x, tem-se que P tem de conter e. Ao se incluir a aresta e_1 em P, obtém-se o ciclo C. Como P pode ser encontrado em tempo $\mathcal{O}(v(G-\{e_1, e_2\}))$, o ciclo C também pode e, portanto, a afirmação do teorema é válida. □

Observe que a prova do Teorema 3.1.1 fornece uma forma de se encontrar um ciclo alternante, com respeito ao emparelhamento M' de G', que contém e_{13} ou e_{24}. Logo, a descrição de todos os detalhes algoritmo de Frink está completa. Uma simples prova por indução no número, $v(G)$, de vértices de G fornece a corretude do algoritmo de Frink e, também, mostra que o algoritmo possui tempo $\mathcal{O}(v(G)^2)$, pois cada passo da recursão gasta $\mathcal{O}(v(G))$ unidades de tempo — *para identificar* G' *e para calcular* M *a partir de* M' — e remove dois vértices do grafo G — *para obter* G'. Como há $v(G)$ vértices no grafo G, há $\Theta(v(G))$ chamadas recursivas. Logo, a complexidade de tempo do algoritmo de Frink é $\mathcal{O}(v(G)^2)$.

3.2 O algoritmo de Biedl, Demaine, Bose e Lubiw

Utilizar o algoritmo de Frink para encontrar um emparelhamento perfeito em um grafo cúbico e sem pontes é menos eficiente do que utilizar qualquer um dos algoritmos em (MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991), pois esses encontrariam o emparelhamento desejado em tempo $\mathcal{O}(e(G)\cdot\sqrt{v(G)}) = \mathcal{O}(v(G)^{1.5})$, pois $e(G) = \Theta(v(G))$ para grafos cúbicos. Os gargalos do algoritmo de Frink, com respeito à complexidade de tempo, são as operações (a) e (b), discutidas na Seção 3.1, as quais podem requerer tempo $\Omega(v(G))$.

Biedl, Demaine, Bose e Lubiw propuseram, em (BIEDL et al., 2001), duas modificações importantes no algoritmo de Frink. Uma delas elimina, totalmente, a necessidade de se encontrar um ciclo alternante durante a operação (b). A outra modificação permite reduzir a complexidade da operação (a) para $\mathcal{O}(\lg^4 v(G))$. Com essas duas modificações, o algoritmo resultante passa a ter complexidade de tempo $\mathcal{O}(v(G) \cdot \lg^4 v(G))$ e se torna uma solução bem mais atraente do que a utilização dos algoritmos em (MICALI; VAZIRANI, 1980; BLUM, 1990; GABOW; TARJAN, 1991), que podem ser aplicados a grafos quaisquer, mas que são menos eficientes para a classe de grafos que realmente interessa ao presente trabalho.

Para eliminar a necessidade de se encontrar um ciclo alternante durante a operação (b), Biedl e colegas forneceram uma versão ligeiramente distinta do teorema de Petersen: *todo grafo cúbico e 2-aresta-conexo possui um emparelhamento perfeito que não usa uma dada aresta,* f. Agora, ao invés de escolher uma aresta arbitrária e com a qual realizar uma redução no algoritmo de Frink, deve-se escolher uma aresta simples, g, adjacente à aresta f.

A redução é realizada com g e, em seguida, a aresta que for incidente a um dos extremos de f no grafo anterior à redução passa a ser a nova f. A escolha da nova aresta, f, depende de qual redução foi utilizada. O propósito de Biedl e colegas em realizar reduções da forma descrita acima é que, no máximo, uma das arestas de G' originadas com a redução (isto é, e_{13} ou e_{24} se $G' = G_1$ e e_{14} ou e_{23} se $G' = G_2$) pertencerá a M'. Logo, um emparelhamento perfeito, M, sempre poderá ser calculado sem a necessidade de se encontrar um ciclo alternante.

Mais especificamente, seja f uma aresta qualquer de G que não se deseja ter em M. Suponha, inicialmente, que existe uma aresta simples, g, em G que é adjacente à aresta f. Fazendo referência à Figura 2.3, assuma que $f = \{x_1, u\}$ e $g = \{u, w\}$. Agora, no algoritmo de Frink, utiliza-se g para efetuar a redução. Em seguida, escolhe-se a nova f para ser $\{x_1, x_3\}$ se $G' = G_1$; caso contrário, a nova f é $\{x_1, x_4\}$. Usando um argumento indutivo, assume-se que a nova aresta f não estará presente em M'. Logo, no momento de calcular M a partir de M', os únicos casos possíveis são os casos (ii) e (iii) da operação (b).

Quando todas as arestas adjacentes a f não são simples, o argumento anterior não pode ser aplicado. Como o gráfico é cúbico, as arestas adjacentes a f devem ser todas arestas duplas. Seja $g = \{x, y\}$ uma dessas arestas. Um dos extremos de g, diga-se x, é comum a $f = \{w, x\}$, enquanto o outro extremo, y, tem de pertencer, também, a uma

aresta simples, $h = \{y, z\}$, distinta de f (veja a Figura 3.3). A modificação proposta por Biedl e colegas, neste caso, é eliminar as arestas f, g, h e a outra aresta dupla paralela a g de G e adicionar uma aresta $j = \{w, z\}$ ao grafo resultante. Este grafo passa a ser o grafo G' e o algoritmo é invocado, recursivamente, com $G = G'$ e a nova aresta f igual a j. No retorno da chamada recursiva, sabe-se, pela hipótese indutiva, que a aresta f não pertence ao emparelhamento M'. Logo, o emparelhamento perfeito, M, de G pode ser calculado a partir de M' com a inclusão de um dos lados de g em M: $M = M' \cup \{g\}$, como mostra a Figura 3.3.

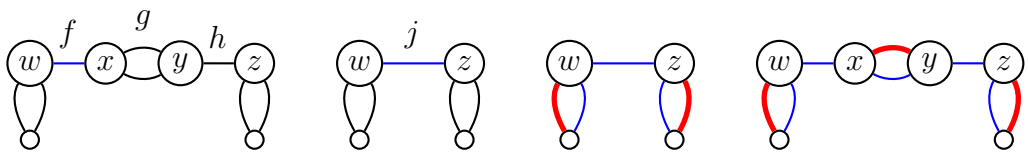


Figura 3.3: O caso em que não há arestas simples adjacentes a f.

Já para reduzir a complexidade da operação (a), foi necessário levar em conta que testar se o grafo reduzido é 2-aresta-conexo equivale a testar se existem dois caminhos aresta-disjuntos entre cada par de seus vértices (veja o Teorema 2.1.22), o que é conhecido na literatura de Teoria dos Grafos como o *problema da 2-aresta-conectividade*. Para isso, Biedl, Bose, Demaine e Lubiw fizeram uso de uma das estruturas dinâmicas descrita em (HOLM; LICHTENBERG; THORUP, 2001), capaz de suportar esse tipo de consulta (se um par de vértices é 2-aresta-conexo), além de inserções e remoções de arestas, em tempo $\mathcal{O}(\lg^4 v(G))$.

Para descobrir a redução correta, basta remover da estrutura dinâmica as arestas que seriam retiradas em uma das duas reduções e testar, para cada par de vértices em $\{x_1, x_2, x_3, x_4\}$ (um número constante, portanto), se ainda existem dois caminhos aresta-disjuntos entre eles. Se a resposta for verdadeira para todos os pares, basta escolher como G' o grafo — G_1 ou G_2 — correspondente à redução. Caso contrário, o teorema de Frink garante que a outra redução é a correta. Logo, a complexidade de tempo do algoritmo proposto por Biedl e colegas é $\mathcal{O}(v(G) \cdot \lg^4 v(G))$. Um pseudocódigo do algoritmo é exibido no Algoritmo 3.1. Assume-se que a estrutura dinâmica D já foi inicializada com as arestas de G, e que uma aresta qualquer de G foi passada como argumento para ser a primeira aresta f.

Algoritmo 3.1 BIEDL(G, D, f)

Entrada: Um grafo cúbico e sem pontes, $G = (V, E, st)$
Entrada: Estrutura de 2-aresta-conectividade, D, inicializada com as arestas de E
Entrada: Uma aresta f que não fará parte do emparelhamento
Saída: Um emparelhamento perfeito M de G

```
M ← ∅
escolha uma aresta g = ∪{x, y} ∈ E adjacente à f
se |V| = 2 então
  M ← M ∪ {g}
senão
  se g é simples então
    utilize D para identificar a redução apropriada
    G' ← grafo resultante da redução apropriada
    f ← a aresta de G' que não está em G e é adjacente à velha f
    M ← BIEDL(G', D, f)
    se apenas uma das arestas reduzidas está em M então
      sejam e1 e e2 as arestas que ligam os extremos dessa aresta aos de g
      M ← M ∪ {e1, e2}
    senão
      M ← M ∪ {g}
    fim se
  senão
    G' ← G − x, y
    j ← a aresta única colocada no lugar das removidas
    M ← BIEDL(G', D, j)
    M ← M ∪ {g}
  fim se
fim se
return M
```

3.3 O algoritmo de Diks e Stanczyk

Mais recentemente, Diks e Stanczyk propuseram uma nova modificação no algoritmo de Frink que resultou em um aperfeiçoamento significativo sobre o algoritmo desenvolvido por Biedl e colegas (DIKS; STANCZYK, 2010). De forma geral, os autores elaboraram uma nova maneira de realizar o teste para identificar a redução correta, que não utiliza a complexa estrutura de 2-aresta-conectividade adotada pelo algoritmo de Biedl e colegas. Para tal, eles levam em conta algumas propriedades dos grafos cúbicos e sem pontes e adotam duas outras estruturas de dados: a estrutura de conectividade dinâmica — coincidentemente apresentada no mesmo artigo que a adotada pelo algoritmo de Biedl e colegas em (BIEDL et al., 2001) — e as árvores dinâmicas de Sleator e Tarjan (SLEATOR; TARJAN, 1983).

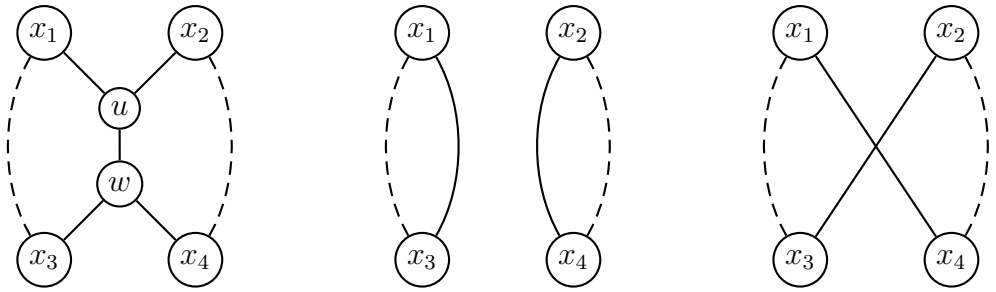
De maneira semelhante à de 2-aresta-conectividade, a primeira estrutura de dados suporta operações de inserção e remoção de arestas e consultas sobre se dois vértices de um grafo, G, estão conectados. Todas essas operações possuem complexidade amortizada $\mathcal{O}(\lg^2 v(G))$. Já a segunda estrutura mantém uma floresta dinâmica, H, que suporta, em tempo amortizado $\mathcal{O}(\lg v(H))$, a inserção e remoção de arestas e o cálculo do ancestral comum mais próximo — isto é, o de menor altura — de dois vértices, u e w, em uma árvore enraizada. Esta operação é denotada aqui por $\mathrm{LCA}(u, w)$ – do inglês *Lowest Common Ancestor*.

No algoritmo em (DIKS; STANCZYK, 2010), a estrutura de conectividade dinâmica (que, por brevidade, será denominada D daqui em diante) deve ser inicializada com as arestas do grafo G fornecido como entrada para o algoritmo. A estrutura manterá, internamente, uma floresta geradora de G cujas árvores (após serem enraizadas) são representadas utilizando árvores dinâmicas (estrutura que será chamada T). Isso é feito através da inserção, em T, de todas as arestas da árvore geradora em D. Dessa forma é possível se encontrar o ancestral comum mais próximo para quaisquer dois vértices do grafo, o que pode ser necessário na hora de determinar qual é a redução apropriada (i.e., aquela que preserva biconectividade). A maneira como esse teste é realizado é, essencialmente, a única alteração proposta por Diks e Stanczyk para o Algoritmo 3.1.

Para ser mais específico, considere o momento em que o algoritmo se prepara para determinar que redução utilizar. Primeiro, remove-se de G e da estrutura D as arestas e, e_1, e_2, e_3 e e_4 que não pertencerão a G'. A Figura 3.4(a) ilustra a estrutura de G antes das arestas serem removidas. A Figura 3.4(b) mostra a redução levando a um grafo não conectado. A Figura 3.4(c) exibe a redução levando a um grafo biconectado. Em seguida, consulta-se D para determinar se o vértice x_1 ainda está conectado aos vértices x_2, x_3 e x_4. Isso equivale a verificar se o grafo, G'', resultante da remoção das arestas e, e_1, e_2, e_3 e e_4 de G se tornou desconexo ou permaneceu conexo. Neste instante, o algoritmo considera os dois possíveis resultados da consulta à estrutura D: 1) G'' é desconexo ou 2) G'' permaneceu conexo.

Como, por hipótese, o grafo G é um grafo 2-aresta-conexo, cada uma das arestas, e, e_1, e_2, e_3 e e_4, faz parte de um ciclo em G. O fato da aresta e pertencer a um ciclo em G implica que e não é ponte de G e que existe, em G'', um caminho entre x_1 e x_3 (ou x_4) ou um caminho entre x_2 e x_4 (ou x_3). Suponha que exista um caminho entre x_1 e x_3 em G''. Afirma-se que há um caminho entre x_2 e x_4 em G'', pois se tal caminho não existisse, então as arestas e_2 e e_4 seriam pontes em G, o que contradiz a hipótese de G

ser 2-aresta-conexo. Reciprocamente, se houver um caminho entre x_1 e x_4 em G'', então há um caminho entre x_2 e x_3 em G''.



(a) (b) (c)

Figura 3.4: O caso 1 do algoritmo de Diks e Stanczyk.

Pelo exposto acima, aplicam-se dois testes de conectividade, ao invés de biconectividade, no grafo G'' para determinar qual dos dois casos, 1 ou 2, é o caso que se tem. Por exemplo, testa-se se x_1 está conectado a x_3. Em caso afirmativo, sabe-se que x_2 está conectado a x_4. Então, basta testar se x_1 também está conectado a x_4 para determinar se o grafo G'' é desconexo ou conexo. Se x_1 não está conectado a x_3, sabe-se que x_1 está conectado a x_4, x_2 está conectado a x_3 e G'' é desconexo. Logo, necessita-se de duas consultas a estrutura de dados dinâmica D, cada qual pode ser realizada em tempo amortizado $\mathcal{O}(\lg^2 v(G))$.

Observe que se o resultado dos testes indicar o caso 1, a redução apropriada também pode ser inferida do resultado, ou seja, sem a necessidade de se fazer mais testes de conectividade. De fato, se há um caminho entre x_1 e x_3 em G'', aplica-se a redução que gera o grafo G_2 da Figura 2.3, pois G_2 é 2-aresta-conexo. Caso contrário, há um caminho entre x_1 e x_4 em G'' e, portanto, deve-se aplicar a redução que gera o grafo G_1 da Figura 2.3. Em ambos os casos, o grafo G' passa a ser o grafo que se origina da redução da aresta $\{u, w\}$.

O problema surge, porém, quando se tem o caso 2 e é justamente aí que se vê a principal contribuição de Diks e Stanczyk. Como o grafo G'' é conexo, considere a árvore geradora, T'', de G'' em D. Considere a subárvore, H, de T'' que consiste de todas as arestas nos caminhos que conectam os vértices x_1, x_2, x_3 e x_4 em T''. De acordo com uma prova exposta na página 327 de (DIKS; STANCZYK, 2010), a redução que preservará a biconectividade de G' é aquela em que as arestas, f e g, resultantes da redução (isto é,

$f = e_{13}$, $g = e_{24}$ ou $f = e_{14}$, $g = e_{23}$) são tais que cada aresta de T'' pertence a algum ciclo em $T'' \cup \{f, g\}$. Para isso, é necessário avaliar os ancestrais comuns mais próximos dos vértices envolvidos, que devem se enquadrar em um dos três casos a seguir:

(i) $\text{LCA}(x_1, x_3) = x$, $\text{LCA}(x_2, x_4) = y$ e $\text{LCA}(x, y) = z$, com $z \neq x, y$;

(ii) $\text{LCA}(x_1, x_3) = \text{LCA}(x_2, x_4)$;

(iii) Este caso possui duas variantes simétricas, cada uma com dois subcasos:

  (1) $\text{LCA}(x_1, x_3) = x$, $\text{LCA}(x_2, x_4) = z$ e $\text{LCA}(x, z) = z$, com $z \neq x$:

    (a) $\text{LCA}(x_2, x) = x$ ou $\text{LCA}(x_4, x) = x$;

    (b) $\text{LCA}(x_2, x) \neq x$ e $\text{LCA}(x_4, x) \neq x$;

  (2) $\text{LCA}(x_1, x_3) = x$, $\text{LCA}(x_2, x_4) = z$ e $\text{LCA}(x, z) = x$, com $z \neq x$:

    (a) $\text{LCA}(x_1, z) = z$ ou $\text{LCA}(x_3, z) = z$

    (b) $\text{LCA}(x_1, z) \neq z$ e $\text{LCA}(x_3, z) \neq z$

A Figura 3.5 ilustra os casos acima – omitindo os subcasos (iii).(2).(a) e (iii).(2).(b), que são simétricos aos de (iii).(1) – e, também, a redução que deve ser aplicada em cada um.

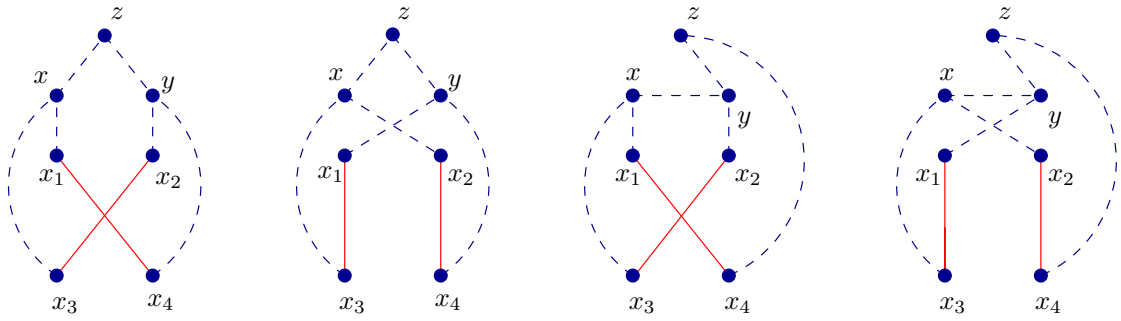


Figura 3.5: Algumas das possíveis configurações de T'' quando G'' é conexo.

Nos dois primeiros casos, basta adicionar as arestas f e g conforme especificado na Figura 3.5. O caso (iii) é um pouco mais complexo, pois são necessários mais dois testes para descobrir qual redução vai gerar ciclos que contenham as arestas entre x e z. Considere o caso (iii).(1). Se $\text{LCA}(x_2, x) = x$ ou $\text{LCA}(x_4, x) = x$ – subcaso (a) – então a redução que gera o grafo G_1, ou seja, que conecta x_1 com x_3 e x_2 com x_4, deve ser aplicada. Caso contrário, subcaso (b), a redução que gera o grafo G_2, ou seja, que conecta x_1 com x_4 e x_2

com x_3, deve ser aplicada. No caso (iii).(2), simétrico ao anterior, a única diferença é no teste a ser realizado, que se inverte. Isto é, o subcaso (a) ocorre quando $\mathrm{LCA}(x_1, z) = z$ ou $\mathrm{LCA}(x_3, z) = z$, sendo também aplicada a redução que leva ao grafo G_1. De modo semelhante, aplica-se a redução que leva a G_2 no subcaso (b). É importante notar que, em qualquer caso, são feitas, no máximo, cinco consultas à estrutura dinâmica D para se obter o LCA de dois vértices, cada uma delas com complexidade de tempo $\mathcal{O}(\lg v(G))$.

4 Árvores Splay

Este capítulo descreve um tipo autoajustável de árvore binária de busca, conhecido na literatura como *árvore splay*[1]. Esta estrutura de dados é de interesse especial para esta monografia, pois ela serve de base para a implementação da *árvore ST* descrita no próximo capítulo, que por sua vez é utilizada na implementação da principal estrutura de dados usada pelo algoritmo de emparelhamento perfeito de Diks e Stancyk (veja a Seção 3.3).

O conteúdo deste capítulo se baseia inteiramente na descrição de árvores splay dada por seus próprios criadores em (SLEATOR; TARJAN, 1985). No entanto, tentou-se fornecer aqui uma descrição mais detalhada dos algoritmos que manipulam tais árvores e, em especial, tentou-se preencher algumas lacunas deixadas pelo texto em (SLEATOR; TARJAN, 1985) no que se refere às provas da análise da complexidade amortizada de tempo desses algoritmos.

A Seção 4.1 apresenta uma visão geral das árvores splay. A Seção 4.2 descreve a heurística de autoajuste utilizada pelos algoritmos de manipulação. A Seção 4.3 descreve as operações de manipulação das árvores splay e seus aspectos de implementação. A Seção 4.4 conclui o capítulo com uma análise da complexidade dos algoritmos descritos na Seção 4.3.

4.1 Visão geral

O surgimento das árvores splay foi motivado pela observação que muitos tipos conhecidos de árvores de busca possuem várias desvantagens. Árvores balanceadas, tais como aquelas balanceadas por altura (ADELSON-VELSKII; LANDIS, 1962; GUIBAS; SEDGEWICK, 1978) ou por peso (NIEVERGELT; REINGOLD, 1973) e as árvores B (BAYER; MCCREIGHT, 1972) e suas variantes (HUDDLESTON; MEHLHORN, 1981, 1982; MAIER; SALVETER, 1981),

[1] Em português, costuma-se denominar esta árvore de *árvore de difusão*.

são estruturadas de tal forma que cada operação de manipulação possui complexidade de tempo $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore. No entanto, além dessas árvores balanceadas necessitarem de espaço extra para armazenar informação utilizada pelo processo de balanceamento, os seus algoritmos de manipulação não são tão eficientes quanto o possível *quando o padrão de acesso aos elementos da árvore não é uniforme, mas tendencioso*.

As árvores de busca ótima (HU; TUCKER, 1979; KNUTH, 1971) garantem o menor tempo médio de acesso aos elementos da árvore, mas esta garantia assume que os acessos não são correlacionados e possuem probabilidades fixas e conhecidas *a priori*. Além disso, o tempo gasto com inserções e remoções é mais alto do que nas árvores splay. Já as árvores de busca do tipo *biased* combinam o rápido tempo médio de acesso das árvores ótimas com a rápida reestruturação das árvores balanceadas, mas possuem restrições estruturais extremamente complexas e mais difíceis de manter do que as das árvores balanceadas. Finalmente, as árvores de busca do tipo *finger* (BROWN; TARJAN, 1980; HUDDLESTON; MEHLHORN, 1982; MAIER; SALVETER, 1981; KOSARAJU, 1981) permitem acesso rápido na vizinhança de um ou mais "dedos" (os *fingers*), mas também requerem espaço extra para armazenamento de apontadores.

Todas as estruturas mencionadas acima foram projetadas com a finalidade de reduzir o tempo *de pior caso* de cada operação individual. No entanto, em aplicações típicas de árvores de busca, não apenas uma, mas toda uma sequência de operações é realizada. Neste contexto, o critério de medida de tempo que realmente importa é o que leva em conta o tempo (total) de execução de todas as operações da sequência e não os tempos individuais das operações. Logo, em tais aplicações, deve-se pensar em reduzir o tempo médio de cada operação em uma *sequência de pior caso* de operações (isto é, uma sequência que maximize o tempo total de execução entre todas as sequências com o mesmo número de operações). Este tempo médio é conhecido como *tempo (de execução) amortizado* e a análise de complexidade que determina o tempo amortizado é conhecida como *análise amortizada*.

Uma das maneiras de se obter um tempo amortizado menor é efetuando algum tipo de autoajuste em cada operação sobre a árvore, modificando a estrutura interna dela com o objetivo de melhorar a eficiência de operações *futuras* da sequência de operações. Estruturas desse tipo tendem a ser mais fáceis de implementar, a necessitar de menos espaço (pois não armazenam informações de balanceamento) e a ter um desempenho melhor na prática do que seus equivalentes não-ajustáveis. Como o foco é na redução

do tempo total de execução da sequência de operações, algumas operações da sequência podem ser computacionalmente onerosas. Isto pode inviablizar o uso dessas estruturas autoajustáveis em aplicações de tempo real. Além disso, tais estruturas requerem uma quantidade maior de ajustes do que as árvores balanceadas.

As árvores splay são, justamente, árvores binárias de busca autoajustáveis que fazem uso de uma heurística de reestruturação chamada *splaying* para reduzir o tempo amortizado de uma sequência de operações. Esta heurística consiste, resumidamente, em mover determinado nó para a raiz da árvore através de uma série de rotações ao longo do caminho entre a posição original do nó em questão e a raiz. A principal desvantagem desse tipo de árvore é que, por não armazenar informação de balanceamento, sua estrutura pode acabar degenerando para uma lista linear (como ocorre caso todos os elementos sejam acessados em ordem não-decrescente). O que implica, conforme mencionado anteriormente, que algumas operações da sequência podem ser bastante lentas. Entretanto, o tempo amortizado total ainda é logarítmico no número de nós, mesmo em uma sequência de pior caso.

No trabalho ora descrito, árvores splay são utilizadas na implementação da árvore ST do Capítulo 5. Uma árvore ST é composta por uma única árvore splay ou por uma coleção delas. No último caso, as várias árvores splays se conectam entre si por arestas especiais. A árvore ST não necessita ser implementada com árvores splay. Há outras opções, como descrito em (SLEATOR; TARJAN, 1983). No entanto, a adoção de árvores splay facilita a implementação da árvore ST, além de garantir que o tempo amortizado de qualquer operação individual sobre a árvore ST está em $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore. Esta garantia não é dada por outras árvores balanceadas, tais como a AVL ou rubro-negra.

4.2 Splaying

Como mencionado na Seção 4.1, árvores splay são árvores autoajustáveis. Isto significa que a estrutura de uma árvore splay é modificada em função da operação realizada, da estrutura atual e dos itens acessados pela operação. Em particular, uma árvore splay muda sua estrutura a cada operação de acesso a um item da árvore. Para tal, uma operação denominada *splaying* é utilizada. Esta operação faz com que o item acessado se torne a raiz da árvore. *Splaying* é uma forma de se implementar uma heurística para reduzir o tempo amortizado de uma sequência de operações na árvore. A idéia por trás da heurística

é manter os nós mais frequentemente acessados próximos à raiz da árvore. A heurística se justifica pela suposição de que itens recentemente acessados possuem uma maior probabilidade de serem acessados em breve novamente, o que realmente se observa, por exemplo, no funcionamento de paginação de memória e memória cache (TANENBAUM, 2009).

A operação de *splaying* se baseia em uma primitiva de reestruturação denominada *rotacão*, que possui tempo constante e que preserva a propriedade de árvore de busca da árvore splay, como ilustrado na Figura 4.1. Seja y um nó da árvore splay que possui o nó x como filho esquerdo e C como subárvore direita. Sejam A e B as subárvores esquerda e direita de x, respectivamente. Então, a operação de *rotacão à direita* rotaciona a aresta que conecta x a y de tal forma que x se torna pai de y, y se torna filho direito de x e a subárvore B passa a ser a subárvore esquerda de y. A *rotação à esquerda* é a inversa da direita.

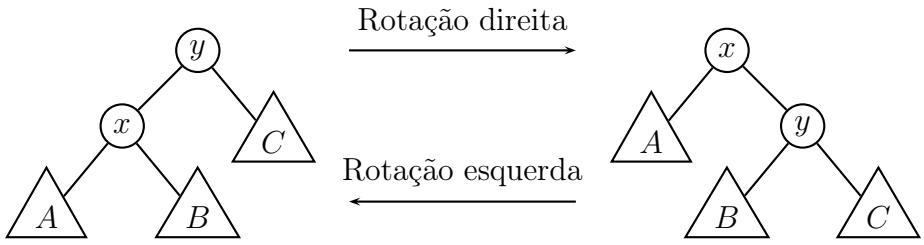


Figura 4.1: Rotação da aresta entre os nós x e y. Triângulos denotam subárvores.

Usando uma ou duas rotações de cada vez, um nó, x, qualquer da árvore pode ser movido até a raiz da árvore. Mais especificamente, uma sequência de operações de rotações é executada sobre o nó x de acordo com os três casos descritos a seguir e ilustrados na Figura 4.2:

(i) Se o pai de x for a raiz, rotacione a aresta que liga x a seu pai;

(ii) Se o pai, y, de x não for a raiz e x e y forem ambos filhos esquerdos (ou, respectivamente, filhos direitos), rotacione a aresta que liga y ao avô, z, de x e, em seguida, rotacione a aresta que liga x a y;

(iii) Se o pai, y, de x não for a raiz, x for um filho esquerdo e y for um filho direito (ou vice-versa), rotacione a aresta que liga x a y e, em seguida, rotacione a aresta que liga x a seu novo pai.

Os casos (i), (ii) e (iii) são denominados, respectivamente, *zig* (pois ocorre apenas uma

rotação à direita ou à esquerda), *zig-zig* (duas rotações à direita ou duas rotações à esquerda) e *zig-zag* (uma rotação à direita seguida por uma rotação à esquerda — ou vice-versa — do mesmo nó). As operações de rotação são aplicadas, segundo os casos acima, até que o nó x se torne a raiz da árvore. Observe que isto sempre ocorre após d rotações, onde d é o nível de x na árvore, pois cada rotação diminui o nível de x em uma unidade.

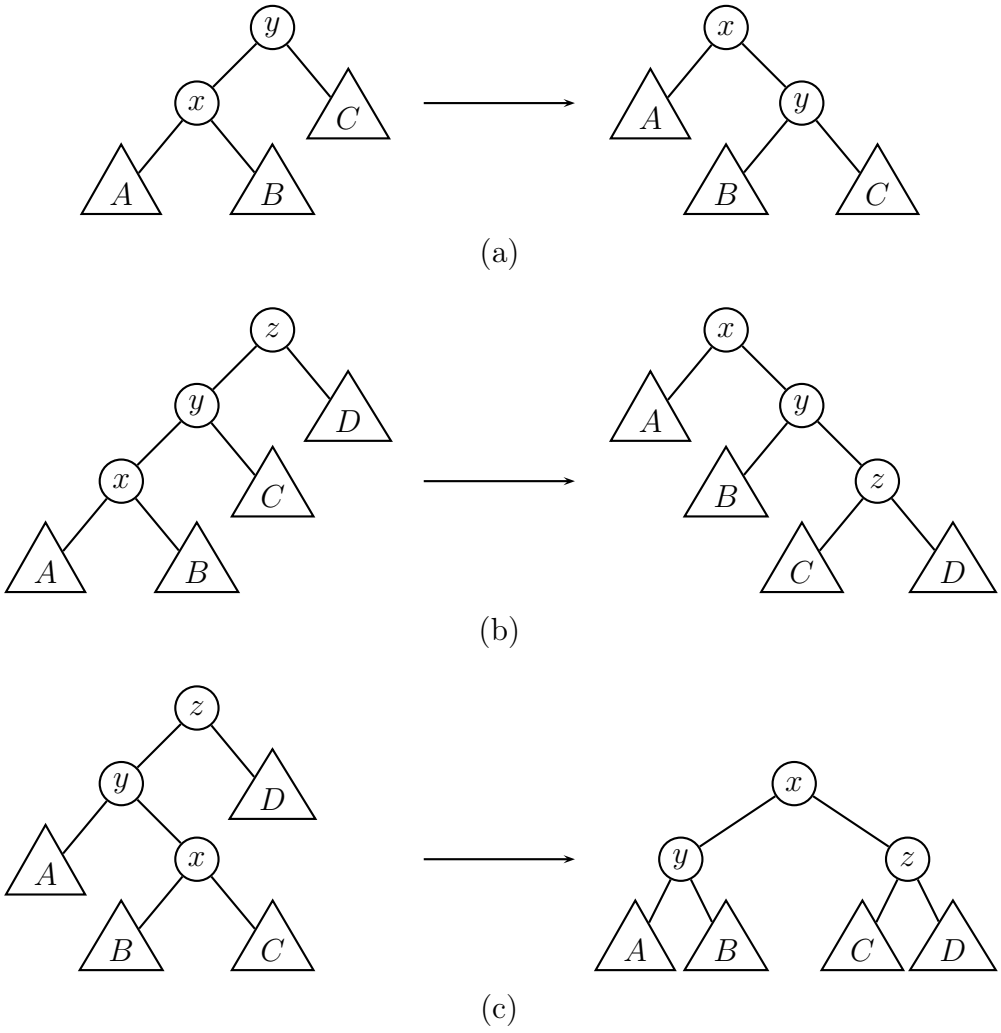


Figura 4.2: Os três possíveis casos da operação *splaying*: (a) Zig. (b) Zig-zig. (c) Zig-zag.

O Algoritmo 4.1 contém o pseudocódigo da operação *splaying*. Assume-se que cada nó da árvore possui apontadores para o filho esquerdo, filho direito e nó pai. Desta forma, as funções Esquerdo(), Direito() e Pai(), que recebem um nó da árvore e retornam o filho esquerdo, o filho direito e o pai do nó, respectivamente, são executadas em tempo constante. Como cada chamada a Rotate() faz com que a profundidade de x na árvore decresça em uma unidade e como Rotate() é executada em tempo constante, pode-se concluir que o corpo do laço **enquanto** de Splay() é executado d vezes, onde d é o nível

inicial de x na árvore, e que a complexidade de tempo de uma chamada à função SPLAY() é $\Theta(d)$. Pode-se mostrar também que SPLAY() reduz pela metade a profundidade de cada nó do caminho que vai de x à raiz (SLEATOR; TARJAN, 1983), embora isto não seja óbvio de ver.

Algoritmo 4.1 SPLAY(x)

Entrada: Um nó x em uma árvore splay T
Saída: Nenhuma

```
enquanto PAI(x) ≠ nil faça
  y ← PAI(x)
  z ← PAI(y)
  se z = nil então
    ROTATE(x)
  senão
    ambosEsquerdos ← (x = ESQUERDO(y)) e (y = ESQUERDO(z))
    ambosDireitos ← (x = DIREITO(y)) e (y = DIREITO(z))
    se ambosEsquerdos ou ambosDireitos então
      ROTATE(y)
      ROTATE(x)
    senão
      ROTATE(x)
      ROTATE(x)
    fim se
  fim se
fim enquanto
```

4.3 Operações

As árvores splay suportam todas as três operações básicas do tipo abstrato de dados (TAD) dicionário: busca, inserção e remoção de elementos. Para se manter coerente com a bibliografia sobre árvores splay, a operação de busca será denominada de operação de *acesso*. Cada uma das três operações, acesso, inserção e remoção, recebe como parâmetros de entrada o valor, i, da chave de um nó e um apontador, q, para a raiz de uma árvore splay:

- ACCESS(i, q): Se a chave i estiver na árvore enraizada em q, encontra o nó que a contém e devolve um apontador para ele; caso contrário, devolve um apontador para **nil**.
- INSERT(i, q): Insere a chave i na árvore enraizada em q, assumindo que ela não está lá.

- REMOVE(i, q): Remove a chave i da árvore enraizada em q, assumindo que ela está lá.

A função ACCESS() faz uma busca a partir da raiz, q, da árvore splay, procurando pelo nó que contém a chave i. Se tal nó for encontrado, ACCESS() executa SPLAY(x), onde x é o nó contendo i, e retorna um apontador para ele (que passa a ser a raiz da árvore). Se a chave i não estiver na árvore e a árvore não estiver vazia, ACCESS() executa SPLAY(x) no último nó (interno) da árvore visitado durante a busca. Em seguida, devolve um apontador para o endereço **nil**. A Figura 4.3 ilustra o funcionamento de ACCESS() e o Algoritmo 4.2 contém o pseudocódigo da função. É importante notar que, embora ACCESS() seja uma operação apenas de "leitura", ela modifica a estrutura da árvore.

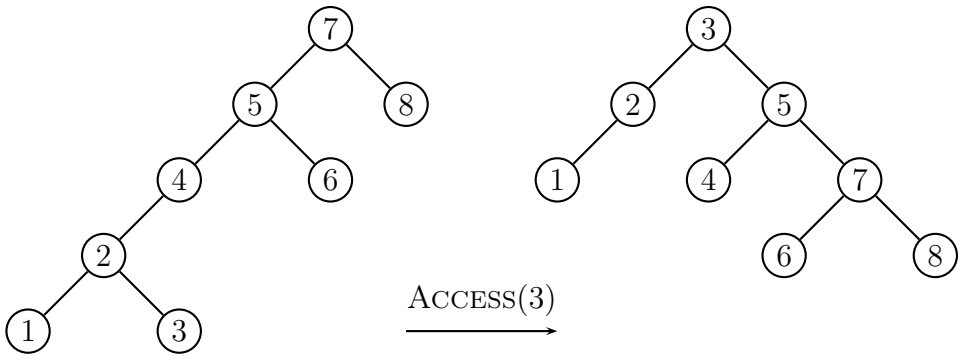


Figura 4.3: Acesso ao nó com chave 3.

As operações INSERT() e REMOVE() são implementadas com o auxílio de duas outras funções:

- JOIN(q_1, q_2): Combina as árvores splay enraizadas nos nós apontados por q_1 e q_2 em uma única árvore splay e devolve um apontador para a raiz da árvore resultante. Esta operação assume que todas as chaves da árvore enraizada em q_1 são menores do que aquelas na árvore enraizada em q_2. As árvores enraizadas em q_1 e q_2 são destruídas.

- SPLIT(i, q): Constrói duas árvores splay e devolve apontadores, q_1 e q_2, para as raízes dessas árvores. A árvore enraizada no nó apontado por q_1 consiste de todas as chaves da árvore splay enraizada em q que são menores ou iguais a i, enquanto a árvore enraizada no nó apontado por q_2 consiste de todas as chaves da árvore splay

enraizada em q que são maiores do que i. Esta operação destrói a árvore enraizada em q.

Algoritmo 4.2 ACCESS(i, q)

Entrada: Uma chave i e um apontador q para a raiz de uma árvore splay
Saída: O endereço do nó contendo i ou o endereço **nil** se i não estiver na árvore.

```
c ← q
p ← nil
enquanto c ≠ nil e CHAVE(c) ≠ i faça
  p ← c
  se CHAVE(c) < i então
    c ← ESQUERDO(c)
  senão
    c ← DIREITO(c)
  fim se
fim enquanto
se c ≠ nil então
  SPLAY(c)
  return c
senão
  SPLAY(p)
  return nil
fim se
```

Para executar JOIN(q_1, q_2), encontra-se o nó, x, da árvore enraizada em q_1 que contém a maior chave, i, entre todas as chaves armazenadas na árvore. Em seguida, executa-se SPLAY(x). Isto faz com que x se torne a raiz da árvore originalmente enraizada em q_1. Como o nó x contém a chave de maior valor da árvore, ele não possui filho direito. Então, faz-se o apontador para o filho direito de x apontar para o nó apontado por q_2, que é a raiz da segunda árvore splay. O resultado é uma árvore splay com raiz x contendo todas as chaves que estão nas árvores enraizadas em q_1 e q_2 (veja a Figura 4.4). Finalmente, as árvores enraizadas em q_1 e q_2 são destruídas (mas, não os nós!) e a árvore resultante é devolvida.

Para executar SPLIT(i, q), executa-se ACCESS(i, q), o que faz com que a raiz da árvore splay originalmente enraizada no nó apontado por q passe a ser o nó, x, contendo a chave i, se i estiver na árvore, ou a maior (resp. menor) chave menor (resp. maior) do que i se i não estiver na árvore. Em seguida, remove-se a aresta que liga a raiz da árvore à subárvore esquerda (se a chave em x for maior do que i) ou a aresta que liga a raiz da árvore à subárvore direita (se a chave em x não for maior do que i). Finalmente, apontadores, q_1 e q_2, para as duas árvores resultantes são devolvidos. A Figura 4.5 ilustra a chamada a SPLIT(i, q).

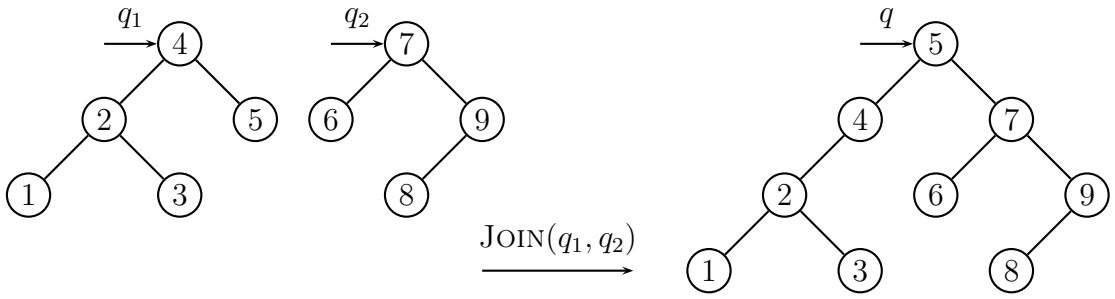


Figura 4.4: Junção das árvores apontadas por q_1 e q_2.

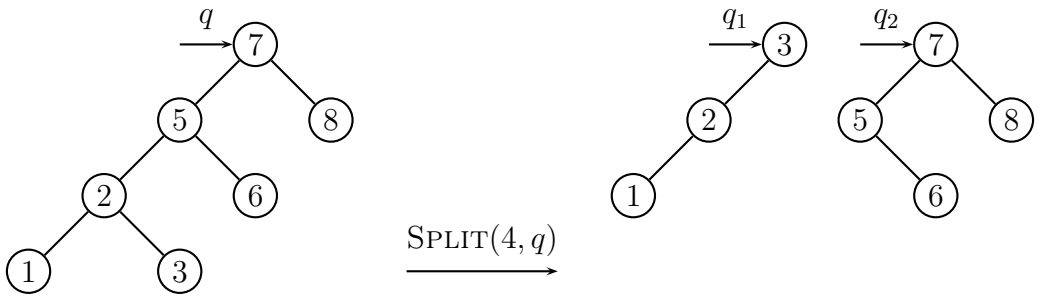


Figura 4.5: Divisão da árvore no nó com chave 4 (inexistente).

A função INSERT() inicia sua execução com a chamada SPLIT(i, q). Como visto acima, o resultado desta chamada são duas árvores splay cujas raízes são apontadas por q_1 e q_2. A árvore cuja raiz é apontada por q_1 consiste de todas as chaves da árvore original que são menores do que i, enquanto a árvore cuja raiz é apontada por q_2 consiste de todas as chaves da árvore original que são maiores do que i, pois, presumivelmente, a chave i não pertence à árvore original. Finalmente, cria-se uma nova árvore splay tal que a raiz é um nó contendo a chave i e tendo as duas árvores resultantes da chamada a SPLIT(i, q) como subárvores esquerda e direita as árvores enraizadas nos nós apontados por q_1 e q_2, respectivamente. A Figura 4.6 exemplifica INSERT() e o Algoritmo 4.3 mostra o pseudocódigo.

A função REMOVE() inicia sua execução com a chamada ACCESS(i, q). Presumindo que a chave i está árvore splay enraizada no nó apontado por q, o resultado da chamada a ACCESS() é uma árvore splay com exatamente os mesmos nós, mas enraizada no nó, x, que contém a chave i. Sejam q_1 e q_2 apontadores para as subárvores esquerda e direita do nó x. O próximo passo de REMOVE() é a chamada JOIN(q_1, q_2). Esta chamada devolve uma árvore splay que consiste exatamente dos nós das árvores cujas raízes são apontadas

por q_1 e q_2 (ou seja, todos os nós da árvore original, com exceção de x). A Figura 4.7 ilustra o funcionamento da função REMOVE() e o Algoritmo 4.4 contém o pseudocódigo da função.

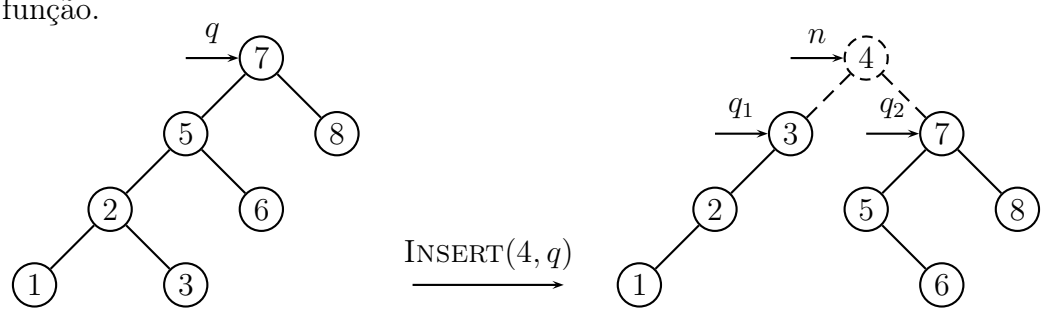


Figura 4.6: Inserção do nó com chave 4 na árvore enraizada em q.

Algoritmo 4.3 INSERT(i, q)

Entrada: Uma chave i e um apontador q para a raiz de uma árvore splay
Saída: Um apontador para a raiz da árvore splay resultante da inserção de i.

```
(q1, q2) ← SPLIT(i, q)
Crie um novo nó, n
CHAVE(n) ← i
PAI(n) ← nil
ESQUERDO(n) ← q1
DIREITO(n) ← q2
return n
```

Note que todas as três funções, ACCESS(), INSERT() e REMOVE(), executam uma chamada, direta ou indireta, a SPLAY(). Logo, a estrutura da árvore resultante sempre muda, até mesmo numa operação que não insere nem remove chaves: ACCESS(). Embora não seja de interesse imediato para este trabalho, este fato evidencia uma desvantagem das árvores splay em relação às árvores balanceadas e de busca ótima: em ambientes *multi-threaded*, há a necessidade de se implementar um controle de concorrência até mesmo para a busca.

4.4 Complexidade amortizada

Esta seção fornece uma análise da complexidade das funções ACCESS(), INSERT(), REMOVE(), JOIN() e SPLIT(). Mais especificamente, calcula-se uma cota superior para o tempo amortizado de cada uma dessas operações e, em seguida, mostra-se que o tempo total de execução de qualquer sequência de m operações em uma árvore splay inicialmente vazia, onde cada operação é um acesso, inserção ou remoção, é $\mathcal{O}(m + \sum_{i=1}^{m} \lg n_i)$, onde

n_i é o número de itens na árvore ou árvores envolvidos na i-ésima operação. Observe que este resultado implica no tempo médio de cada operação da sequência ser $\mathcal{O}(\lg n_{\max})$, onde $n_{\max} = \max_{i=1}^{m}\{n_i\}$. Isso mostra a eficiência das árvores splay quando amortização é empregada. Para realizar as análises desta seção, utilizou-se o método do potencial. De agora em diante, assume-se que o leitor está familiarizado com análise amortizada e o método do potencial. Se este não for o caso, o leitor pode se valer do conteúdo do Apêndice B.

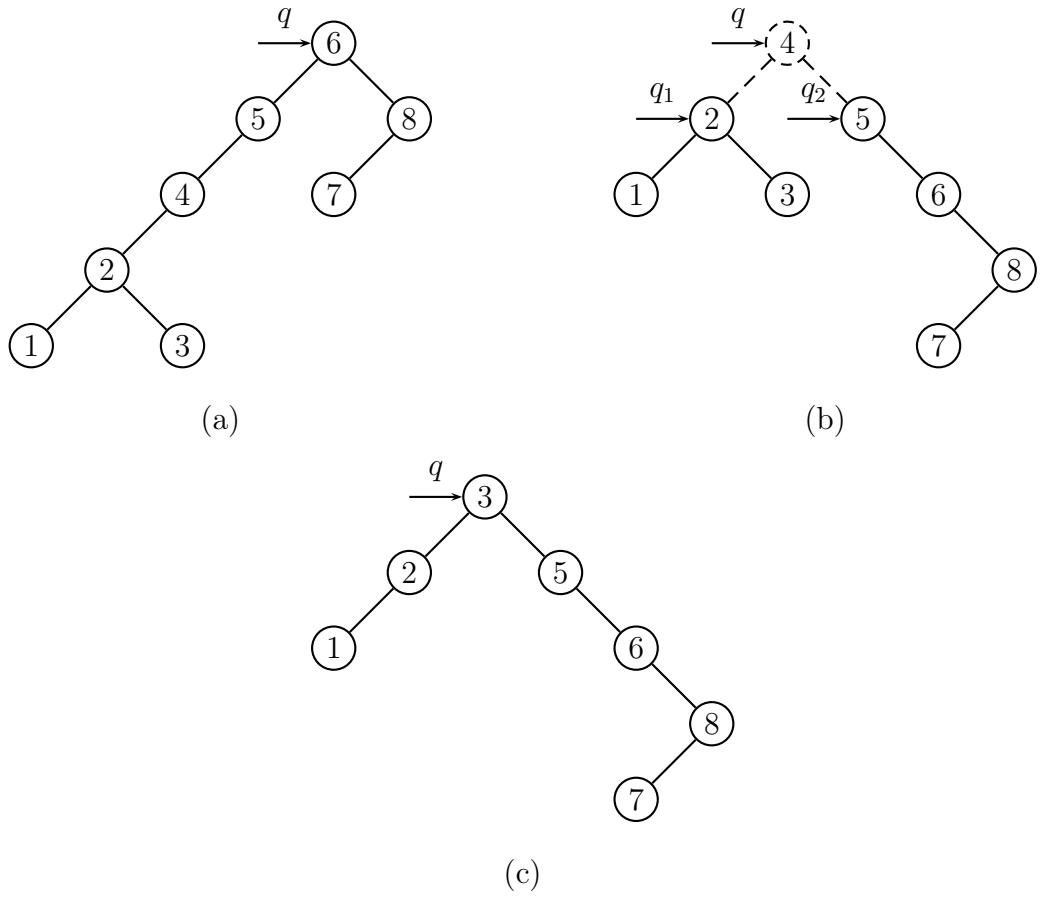


Figura 4.7: Remoção do nó com chave 4 na árvore enraizada em q. (a) Antes da remoção. (b) Após ACCESS$(4, q)$ (c) Após JOIN(q_1, q_2)

Seja S qualquer sequência de m operações de acesso, inserção e remoção em uma árvore splay, T, inicialmente vazia. Denota-se por T_i, para todo $i = 1, \ldots, m$, o *estado* (ou *configuração*) da árvore resultante da i-ésima operação. A configuração inicial, ou seja, aquela correspondente à árvore vazia, é denotada por T_0. Lembre-se de que, no método do potencial, define-se uma *função potencial*, $\Phi_T : \mathcal{T} \to \mathbb{R}$, que associa um número não-negativo, $\Phi_T(A)$, a cada configuração, A, do espaço, $\mathcal{T}$, das configurações possíveis de T.

A função Φ_T deve satisfazer a condição $\Phi_T(T_i) \geq \Phi_T(T_0)$, para todo $i = 1, \ldots, m$. Além disso, tem-se, comumente, que $\Phi_T(T_0) = 0$. Uma vez que Φ_T esteja definida, o *tempo amortizado*, $\hat{t}_i$, da i-ésima operação de S é igual a $\hat{t}_i = t_i + \Phi_T(T_i) - \Phi_T(T_{i-1})$, onde t_i é o tempo (real) gasto com a i-ésima operação e o termo $\Phi_T(T_i) - \Phi_T(T_{i-1})$ é a *diferença de potencial*.

Algoritmo 4.4 REMOVE(i, q)

Entrada: Uma chave i e um apontador q para a raiz de uma árvore splay
Saída: Um apontador para a raiz da árvore splay resultante da remoção de i.

```
ACCESS(i, q)
q1 ← ESQUERDO(q)
q2 ← DIREITO(q)
Destrua o nó apontado por q
q ← JOIN(q1, q2)
return q
```

O objetivo de uma análise amortizada da sequência S é determinar o *tempo de execução amortizado total* de todas as m operações da sequência, que nada mais é do que a soma

$$\sum_{i=1}^{m} \hat{t}_i \,.$$

Usando o fato que $\hat{t}_i = t_i + \Phi(T_i) - \Phi(T_{i-1})$, pode-se concluir que a soma acima é dada por

$$\sum_{i=1}^{m} \hat{t}_i = \sum_{i=1}^{m} t_i + \Phi(T_m) - \Phi(T_0) \,.$$

Como, por definição de Φ, tem-se que $\Phi(T_i) \geq \Phi(T_0)$, para todo $i = 1, \ldots, m$, conclui-se que

$$\sum_{i=1}^{m} \hat{t}_i \geq \sum_{i=1}^{m} t_i \,,$$

e, portanto, o tempo de execução amortizado total é uma cota superior para o *tempo de execução real* de todas as m operações de S. Logo, o *tempo de execução amortizado* de cada operação, que é a média

$$\frac{\sum_{i=1}^{m} \hat{t}_i}{m}$$

sobre as m operações de S, é uma cota superior para a média do *tempo (real) de execução*, isto é,

$$\frac{\sum_{i=1}^{m} t_i}{m} \,,$$

das m operações de S. O tempo de execução amortizado sobre todas as m operações de S é a *complexidade amortizada* de cada operação da sequência. É importante salientar que

o tempo de execução real de uma dada operação individual da sequência pode exceder, e em muito, a sua complexidade amortizada, mas o que importa é que a média do tempo de execução de todas as operações da sequência não excede a complexidade amortizada delas.

Para definir a função potencial, $\Phi_T : \mathcal{T} \to \mathbb{R}$, assume-se que cada chave α em T possui um número positivo, $w(\alpha)$, associado a ela e denominado *peso*, cujo valor é arbitrário, mas fixo. O peso, $w(\alpha)$, é um parâmetro da análise e não depende do algoritmo a ser analisado. Usando-se o peso das chaves, definem-se dois outros valores, tamanho e posto, associados a cada nó de T. A saber, se x é um nó em T, então o *tamanho*, $s(x)$, de x é definido pela soma

$$\sum_{y \in T_x} w(\alpha_y)\,,$$

onde T_x é a subárvore de T enraizada em x e i_y é a chave armazenada no nó y. Em outras palavras, o tamanho, $s(x)$, de x é a soma dos pesos dos nós pertencentes à subárvore, T_x, de T enraizada em x. Por sua vez, o *posto*, $r(x)$, do nó x é igual a $\lg s(x)$. Finalmente, tem-se:

$$\Phi_T(A) = \sum_{x \in A} r(x)\,,$$

isto é, o potencial da configuração, A, da árvore T é igual à soma dos postos de todos os seus nós.

Como exemplo, considere a configuração de árvore na Figura 4.8. Se o valor de $w(\alpha)$ for 1 para todas as chaves, então $s(c) = s(e) = s(g) = 1$, $s(b) = 2$, $s(d) = 4$, $s(f) = 6$ e $s(a) = 7$, e

$$\begin{aligned}
\Phi_T(A) &= \sum_{x \in A} r(x) \\
&= r(a) + r(f) + r(d) + r(g) + r(b) + r(e) + r(c) \\
&= \lg s(a) + \lg s(f) + \lg s(d) + \lg s(g) + \lg s(b) + \lg s(e) + \lg s(c) \\
&= \lg \left(s(a) \cdot s(f) \cdot s(d) \cdot s(g) \cdot s(b) \cdot s(e) \cdot s(c)\right) \\
&= \lg(7 \cdot 6 \cdot 4 \cdot 1 \cdot 2 \cdot 1 \cdot 1) \\
&= \lg 336 \\
&\approx 8{,}39
\end{aligned}$$

Como mencionado na Seção 4.3, os algoritmos de acesso, inserção e busca em árvores splay realizam *splaying*, direta ou indiretamente. Além disso, *splaying* é a operação

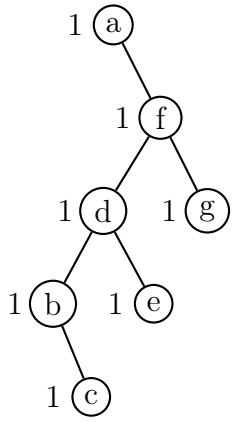


Figura 4.8: Uma configuração com potencial 8,39 se todas as chaves têm peso 1.

preponderante em termos da complexidade de tempo desses algoritmos. Ao se examinar o Algoritmo 4.1, nota-se que o número de operações primitivas executado pelo algoritmo durante o percurso na árvore é $\max\{1, c \cdot n\}$, onde n é o número de rotações e c é uma constante. Esta observação permite que se adote o *número de rotações* da operação *splaying* como o "modelo de custo" na análise da complexidade amortizada do Algoritmo 4.1, pois o tempo gasto com cada uma das outras operações primitivas pode ser embutido no tempo de uma rotação. Para os demais algoritmos vistos na Seção 4.3, deve-se contabilizar, eventualmente, o tempo para se desconectar uma subárvore ou para conectar duas subárvores a um nó para se obter uma nova árvore. Em ambos os casos, pode-se supor que o tempo gasto com uma operação de conexão (resp. desconexão) seja o mesmo de uma rotação.

O lema a seguir fornece uma cota superior para o tempo amortizado, $\hat{t}$, de *uma única* operação *splaying*. Isto é, ele fornece uma cota superior para $t + \Phi_T(B) - \Phi_T(A)$, onde t é o tempo de execução da operação *splaying*, A é a configuração da árvore splay, T, antes da operação e B é a configuração de T após a operação. A cota fornecida é dada em função do posto, $r(q)$, da raiz de A e do posto, $r(x)$, do nó x de A para o qual a operação *splaying* foi invocada. A cota superior fornecida pelo lema é usada por todas as demais análises.

Lema 4.4.1. O tempo amortizado de uma única operação *splaying* sobre um nó x de uma árvore splay cuja raiz é o nó q é, no máximo, igual a $1 + 3 \cdot (r(q) - r(x))$, que está em $\mathcal{O}\left(\lg \frac{s(q)}{s(x)}\right)$.

Demonstração. Assuma que cada rotação gasta 1 unidade de tempo. Então, se não houver nenhuma rotação (isto é, se $x = q$), então $t = 0$, $r(x) = r(q)$ e $A = B$, onde A e B são as configurações da árvore, T, antes e depois da operação *splaying* em x, respectivamente.

Logo, tem-se que $\Phi_T(A) = \Phi_T(B)$ e, portanto, $\hat{t} = t + \Phi_T(B) - \Phi_T(A) = 0 < 1 = 1 + 0 = 1 + 3 \cdot 0 = 1 + 3 \cdot (r(q) - r(x))$. Se houver pelo menos uma rotação, então considere a sequência, $A = A_0, A_1, \ldots, A_{n-1}, A_n = B$, de configurações da árvore T tal que A_j, para todo $j = 1, \ldots, n$, é a configuração obtida após o j-ésimo *passo* de *splaying*, n (com $n \geq 1$) é o total de passos e cada passo é formado pela aplicação de um dos três casos de rotação da Figura 4.2. Logo, o tempo amortizado, $\hat{t}$, pode ser escrito como uma soma:

$$\hat{t} = \sum_{j=1}^{n} \hat{t}_j \,,$$

onde $\hat{t}_j$ é o tempo amortizado para se passar de A_{j-1} para A_j. Cada passo possui exatamente uma ou duas rotações. Então, o valor de $\hat{t}_j$ depende de qual dos três casos da Figura 4.2 foi executado no j-ésimo passo da operação *splaying*. No que se segue, calcula-se uma cota superior para $\hat{t}_j$ em cada um três casos. Para tal, denota-se por s_{j-1} e s_j a função tamanho antes e depois do passo j ser executado. De forma análoga, utiliza-se r_{j-1} e r_j para a função posto. A seguir, mostra-se que $\hat{t}_j \leq 1 + 3 \cdot (r_j(x) - r_{j-1}(x))$ se o passo i aplicar o caso (a) da Figura 4.2 (ou seja, o caso zig) e que $\hat{t}_j \leq 3 \cdot (r_j(x) - r_{j-1}(x))$ se o passo j aplicar o caso (b) ou (c) da Figura 4.2 (ou seja, o caso zig-zig ou o caso zig-zag):

1) No caso zig, apenas uma rotação é realizada e, portanto, tem-se $t_j = 1$ e

$$\begin{array}{rcll}
\hat{t}_j & = & t_j + \Phi_T(A_j) - \Phi_T(A_{j-1}) & \text{por definição de } \hat{t}_j \\
& = & 1 + r_j(x) + r_j(y) - r_{j-1}(x) - r_{j-1}(y) & \text{pois, apenas } x \text{ e seu pai, } y, \text{ têm} \\
& & & \text{o posto modificado em } A_j \\
& \leq & 1 + r_j(x) - r_{j-1}(x) & \text{pois, } r_j(y) \leq r_{j-1}(y) \\
& \leq & 1 + 3 \cdot (r_j(x) - r_{j-1}(x)) & \text{pois, } r_j(x) \geq r_{j-1}(x).
\end{array}$$

2) No caso zig-zig, exatamente duas rotações são realizadas e, portanto, tem-se $t_j = 2$ e

$$\begin{array}{rcll}
\hat{t}_i & = & t_i + \Phi_T(A_j) - \Phi_T(A_{j-1}) & \text{por definição de } \hat{t}_j \\
& = & 2 + r_j(x) + r_j(y) + r_j(z) & \text{pois, apenas } x, \text{ seu pai, } y, \\
& & \quad - r_{j-1}(x) - r_{j-1}(y) - r_{j-1}(z) & \text{e seu avô, } z, \text{ têm posto} \\
& & & \text{modificado em } A_j. \\
& = & 2 + r_j(y) + r_j(z) - r_{j-1}(x) - r_{j-1}(y) & \text{pois, } r_j(x) = r_{j-1}(z) \\
& \leq & 2 + r_j(x) + r_j(z) - 2 \cdot r_{j-1}(x) & \text{pois, } r_j(x) \geq r_j(y) \text{ e} \\
& & & r_{j-1}(y) \geq r_{j-1}(x).
\end{array}$$

Afirma-se que $2 + r_j(x) + r_j(z) - 2 \cdot r_{j-1}(x) \leq 3 \cdot (r_j(x) - r_{j-1}(x))$ ou, de forma

equivalente, que $2 \cdot r_j(x) - r_{j-1}(x) - r_j(z) \geq 2$ e $r_j(z) + r_{j-1}(x) - 2 \cdot r_j(x) \leq -2$. Mas, sabe-se que

$$r_j(z) + r_{j-1}(x) - 2 \cdot r_j(x) = \lg s_j(z) + \lg s_{j-1}(x) - 2 \cdot \lg s_j(x)\,,$$

onde, por definição, $s_j(z), s_{j-1}(x), s_j(x) > 0$. Ao examinar o caso (b) na Figura 4.2, nota-se que $s_j(z) + s_{j-1}(x) \leq s_j(x)$, pois todos os nós nas subárvores A e B de A_{j-1} e todos os nós das subárvores C e D de A_j são nós da árvore, T_x, enraizada em x em A_j. Além disso, os nós x e z também fazem parte de T_x em A_j. Por outro lado, se a, b e c são quaisquer três números positivos, com $a + b \leq c$, então se tem a seguinte derivação:

$$\begin{aligned}
\left(\sqrt{a} - \sqrt{b}\right)^2 \geq 0 &\Rightarrow \sqrt{a \cdot b} \leq \frac{a+b}{2} \\
&\Rightarrow \sqrt{a \cdot b} \leq \frac{c}{2} \\
&\Rightarrow \lg\left(\sqrt{a \cdot b}\right) \leq \lg\left(\frac{c}{2}\right) \\
&\Rightarrow \frac{1}{2} \cdot (\lg a + \lg b) \leq (\lg c - 1) \\
&\Rightarrow \lg a + \lg b - 2 \cdot \lg c \geq -2\,.
\end{aligned}$$

Substituindo-se a, b e c acima por $s_j(z)$, $s_{j-1}(x)$ e $s_j(x)$, tem-se o que se desejava provar:

$$\lg s_j(z) + \lg s_{j-1}(x) - 2 \cdot \lg s_j(x) \geq -2 \Rightarrow r_j(z) + r_{j-1}(x) - 2 \cdot r_j(x) \leq -2\,.$$

Logo,

$$\hat{t}_j \leq 3 \cdot (r_j(x) - r_{j-1}(x))\ .$$

3) No caso zig-zag, exatamente duas rotações são realizadas e, portanto, tem-se $t_j = 2$ e

$$\begin{array}{rcll}
\hat{t}_j & = & t_j + \Phi_T(A_j) - \Phi_T(A_{j-1}) & \text{por definição de } \hat{t}_j \\
& = & 2 + r_j(x) + r_j(y) + r_j(z) & \text{pois, apenas } x\text{, seu pai, } y\text{,} \\
& & \quad - r_{j-1}(x) - r_{j-1}(y) - r_{j-1}(z) & \text{e seu avô, } z\text{, têm posto} \\
& & & \text{modificado em } A_j\text{.} \\
& = & 2 + r_j(y) + r_j(z) - r_{j-1}(x) - r_{j-1}(y) & \text{pois, } r_j(x) = r_{j-1}(z) \\
& \leq & 2 + r_j(y) + r_j(z) - 2 \cdot r_{j-1}(x) & \text{pois, } r_{j-1}(y) \geq r_{j-1}(x)\text{.}
\end{array}$$

Afirma-se que $2 + r_j(y) + r_j(z) - 2 \cdot r_{j-1}(x) \leq 2 \cdot (r_j(x) - r_{j-1}(x))$ ou, de forma equivalente, que $2 \cdot r_j(x) - r_j(y) - r_j(z) \geq 2$ ou $r_j(y) + r_j(z) - 2 \cdot r_j(x) \leq -2$. Usando

o fato que $s_j(y) + s_j(z) \leq s_j(x)$ e o mesmo argumento da demonstração do caso anterior, pode-se concluir que $\hat{t}_j \leq 2 \cdot (r_j(x) - r_{j-1}(x))$ e, portanto, tem-se também que

$$\hat{t}_j \leq 3 \cdot (r_j(x) - r_{j-1}(x)) \ .$$

Como o caso em que há uma única rotação (isto é, zig) só pode ser aplicado no n-ésimo passo de *splaying*, que é justamente quando x é um filho da raiz da árvore A_{n-1}, tem-se que $\hat{t}_j \leq 3 \cdot (r_j(x) - r_{j-1}(x))$, para todo $i = 1, \ldots, n-1$ e $\hat{t}_n \leq 1 + 3 \cdot (r_j(x) - r_{j-1}(x))$. Resta agora derivar a cota superior de $\hat{t}$. Como visto antes, $\hat{t} = \sum_{j=1}^{n} \hat{t}_j = \sum_{j=1}^{n-1} \hat{t}_j + \hat{t}_n$. Mas,

$$\sum_{j=1}^{n-1} \hat{t}_j \leq \sum_{j=1}^{n-1} 3 \cdot (r_j(x) - r_{j-1}(x)) = 3 \cdot (r_{n-1}(x) - r_0(x))$$

e

$$\hat{t}_n \leq 1 + 3 \cdot (r_n(x) - r_{n-1}(x)) \ .$$

Logo,

$$\hat{t} \leq 1 + 3 \cdot (r_n(x) - r_0(x)) \ .$$

Mas, $r_0(x)$ é o posto do nó x na configuração inicial, $A = A_0$, de T, enquanto $r_n(x)$ é o posto do nó x na configuração final, $B = A_n$, de T. Logo, $r_n(x) = r_0(q)$, pois x é a raiz da árvore B após a operação *splaying* terminar. Como nenhum nó foi inserido ou removido durante a operação *splaying*, deve-se ter $s_0(q) = s_n(x)$. Então, $\hat{t} \leq 1 + 3 \cdot (r(q) - r(x))$, como desejado. □

A cota superior fornecida pelo Lema 4.4.1 para o tempo amortizado de uma operação *splaying* é tudo que se precisa para deduzir uma cota superior para o tempo amortizado de cada chamada a Access(), Insert(), Remove(), Join() ou Split(). Uma vez que o tempo amortizado de uma chamada a cada uma dessas funções seja deduzido, pode-se estabelecer o tempo médio amortizado das operações de uma sequência de m operações sobre uma árvore splay inicialmente vazia, onde cada operação é um acesso, inserção ou remoção.

Nos resultados a seguir, assume-se que todas as chaves pertencem a um universo, U, e que cada chave α em U possui um peso, $w(\alpha)$, associado a ela. Este peso é arbitrário, mas uma vez fixado, não pode mudar. Assume-se também que cada chave de U pertence, inicialmente, a uma árvore com um único nó e que T, a árvore sobre a qual as operações serão executadas, está vazia. Define-se o *potencial de uma coleção de árvores* como a soma dos potenciais individuais de cada árvore da coleção. Portanto, inicialmente, tem-

se que o potencial da coleção formada por T e as árvores que contêm as chaves de U é $\sum_{\alpha \in U} \lg w(\alpha)$. Durante as análises, se x é um nó de T contendo a chave α, então se denotará por α_- e α_+ as chaves dos nós antecessor e sucessor de x, respectivamente, em um percurso em ordem simétrica em T. Se o nó x não possuir antecessor (resp. sucessor), então $w(\alpha_-) = \infty$ (resp. $w(\alpha_+) = \infty$).

Lema 4.4.2. O tempo amortizado, $\hat{t}$, da chamada ACCESS(α, q) é tal que

$$\hat{t} \leq \begin{cases} 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + 1, & \text{se } \alpha \text{ está na árvore} \\ 3 \cdot \lg\left(\dfrac{W}{\min\{w(\alpha_-), w(\alpha_+)\}}\right) + 1, & \text{caso contrário,} \end{cases}$$

onde q é um apontador para a raiz de uma árvore splay, α é uma chave (que pode ou não pertencer a árvore), W é a soma dos pesos dos nós da árvore, $w(\alpha)$ é o peso associado a α e $w(\alpha_-)$ (resp. $w(\alpha_+)$) é o peso associado à chave, α_- (resp. α_+), que precede (resp. sucede) α em um percurso em ordem simétrica pela árvore caso α esteja na árvore ou, em caso contrário, α_- (resp. α_+) precederia (resp. sucederia) α no mesmo percurso se α estivesse na árvore.

Demonstração. Considere os dois possíveis cenários: (1) a chave α está na árvore e (2) a chave α não está na árvore. No cenário 1, o Algoritmo 4.2 encontra o nó, x, que contém α, executa SPLAY() e retorna um apontador para x. Assumindo que cada rotação gasta 1 unidade de tempo, o Lema 4.4.1 diz que o tempo amortizado, $\hat{t}$, da chamada ACCESS(α, q), onde q é um apontador para a raiz da árvore, é tal que $\hat{t} \leq 1 + 3 \cdot (r(q) - r(x))$. Mas, por definição de W e pelo fato de α está na árvore, tem-se que $r(q) = \lg W$. Por sua vez, sabe-se que $s(x) \geq w(\alpha)$, pois $w(\alpha)$ é o menor valor de tamanho que o nó x pode ter e $s(x) = w(\alpha)$ exatamente quando x for um nó folha. Então, $r(x) \geq \lg w(\alpha)$ e, portanto, tem-se

$$\hat{t} \leq 1 + 3 \cdot (r(q) - r(x)) \leq 1 + 3 \cdot (\lg W - \lg w(\alpha)) = 1 + 3 \cdot \lg\left(\frac{W}{w(\alpha)}\right).$$

No cenário 2, a busca realizada por ACCESS(α, q) pela chave α termina no nó, x, que contém a chave, α_- (resp. α_+), que precederia (resp. sucederia) a chave α em um percurso em ordem simétrica pela árvore. Assim como no cenário anterior, o Algoritmo 4.2 executa SPLAY() em x, mas retorna um apontador para o endereço **nil**. Logo, o tempo amortizado, $\hat{t}$, da chamada ACCESS(α, q), onde q é um apontador para a raiz da árvore, também é limitado superiormente por $1 + 3 \cdot (r(q) - r(x))$, de acordo com o Lema 4.4.1. No entanto, desta vez, o nó x é aquele que contém a chave α_- ou a chave α_+, dependendo de onde

a busca terminar. Logo, sabe-se que $s(x) \geq \min\{w(\alpha_-), w(\alpha_+)\}$ e, como no cenário 1, tem-se

$$\begin{aligned}\hat{t} &\leq 1 + 3 \cdot (r(q) - r(x)) \\ &\leq 1 + 3 \cdot (\lg W - \lg \min\{w(\alpha_-), w(\alpha_+)\}) \\ &= 1 + 3 \cdot \lg \left(\frac{W}{\min\{w(\alpha_-), w(\alpha_+)\}} \right) .\end{aligned}$$

□

As operações SPLIT(), JOIN(), INSERT() e REMOVE() modificam o estado de mais de uma árvore da coleção de árvores (isto é, da floresta). Logo, nos lemas que se seguem, é importante perceber que o tempo amortizado, $\hat{t}$, de qualquer uma dessas operações é igual a

$$\hat{t} = t + \Phi_F(B) - \Phi_F(A) ,$$

onde A e B são os estados da floresta imediatamente antes e imediatamente depois da operação ser executada, respectivamente, e $\Phi_F : \mathcal{F} \to \mathbb{R}$ é uma função potencial que associa um número não-negativo a cada configuração do conjunto, $\mathcal{F}$, de todas as possíveis configurações. Pela definição vista anteriormente de potencial de uma coleção de árvores, tem-se

$$\Phi_F(A) = \sum_{Q \in A} \Phi_T(Q) \qquad \text{e} \qquad \Phi_F(B) = \sum_{R \in B} \Phi_T(R)$$

onde Q (resp. R) denota uma árvore da configuração A (resp. B) da floresta imediatamente antes (resp. depois) da operação e $\Phi_T : \mathcal{T} \to \mathbb{R}$ é a já conhecida função potencial que associa um valor não-negativo a cada árvore do conjunto, $\mathcal{T}$, de todas as possíveis árvores splay.

Com base no exposto acima, consideram-se a seguir duas situações que serão exploradas nas provas dos lemas 4.4.3-4.4.6. Na primeira delas, supõe-se uma árvore T_1 cujo nó raiz possui E e D como subárvores esquerda e direita, respectivamente (veja a Figura 4.9). Suponha que a aresta que liga a raiz de T_1 à subárvore D foi retirada de T_1, dando origem a duas novas árvores, T_1' e D, tal que T_1' é a árvore T_1 sem a subárvore D. Em seguida, a árvore T_1 é destruída (mas, não os nós que estão em T_1' e D). Usando a notação do parágrafo anterior, denota-se por A a coleção de árvores que se tinha antes da retirada da aresta de T_1 e, por B, a coleção de árvores obtida com a retirada da aresta. Observe que a única diferença entre A e B é que A possui T_1 mas não possui T_1' nem D, e vice-versa. Mas, o que se pode dizer da diferença, $\Phi_F(B) - \Phi_F(A)$, dos potenciais de A e B?

Figura 4.9: A árvore T_1 (esquerda) e as árvores $T_1' = \{x\} \cup E$ e D (direita).

Levando-se em conta as únicas diferenças existentes entre A e B, tem-se que

$$\Phi_F(B) - \Phi_F(A) = \sum_{Q \in A} \Phi_T(Q) - \sum_{R \in B} \Phi_T(R) = \Phi_T(T_1') + \Phi_T(D) - \Phi_T(T_1)\,.$$

Mas,

$$\Phi_T(T_1) = \sum_{x \in T_1} r(x) = r(q) + \sum_{x \in E} r(x) + \sum_{x \in D} r(x) = r(q) + \Phi_T(E) + \Phi_T(D)\,,$$

onde q é o nó raiz de T_1 e $r(x)$ denota o posto de x, para todo nó x em T_1. Por sua vez, tem-se

$$\Phi_T(T_1') = \sum_{x \in T_1'} r'(x) = r'(q) + \sum_{x \in E} r'(x) = r'(q) + \sum_{x \in E} r(x) = r'(q) + \Phi_T(E)\,,$$

onde q também é o nó raiz de T_1', $r'(x)$ denota o posto de x, para todo nó x em T_1' e $r'(x) = r(x)$ para todo x na subárvore esquerda, E, de q em T_1', pois ela é a mesma da subárvore esquerda de q em T_1. Logo, $\Phi_F(B) - \Phi_F(A) = r'(q) - r(q) = \lg s'(q) - \lg s(q) = \lg \frac{s'(q)}{s(q)}$, onde $s'(q)$ e $s(q)$ são os tamanhos do nó q em T_1' e T_1, respectivamente. Como $s(q) = s'(q) + s(q_d)$, onde q_d é o nó raiz da subárvore D e $s(q_d)$ é o tamanho de D, tem-se que

$$\Phi_F(B) - \Phi_F(A) = \lg \left(\frac{s'(q)}{s'(q) + s(q_d)} \right)\,.$$

Observe que o valor acima é menor ou igual a zero, sendo zero apenas se a subárvore D for vazia. Se este não for o caso, a operação de remoção da aresta causa um decréscimo de potencial.

Na segunda situação, supõe-se a existência de três árvores, E, D e T_1, onde T_1 contém um único nó e todas as chaves associadas aos nós de E (resp. D) são menores (resp. maiores) ou iguais do que aquela associada ao único nó de T_1, isto é, o nó raiz de T_1. Então, constrói-se uma nova árvore, T_1', conectando as árvores E e D ao nó raiz de T_1 de tal forma que E (resp. D) se torna a subárvore esquerda (resp. direita) deste nó (veja a Figura 4.10). Em seguida, as árvores E, D e T_1 são destruídas (mas, não os seus nós, que estão em T_1'). Assim como antes, denota-se por A a coleção de árvores que se tinha

antes da construção de T_1' e, por B, a coleção de árvores obtida com após a construção. Observe que a única diferença entre A e B é que A possui E, D e T_1 mas não possui T_1', e vice-versa. O que se quer é determinar a diferença, $\Phi_F(B) - \Phi_F(A)$, entre os potenciais de A e B.

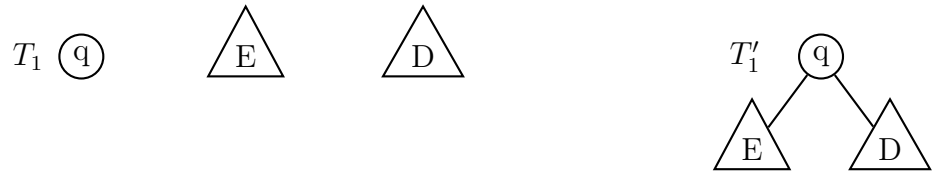


Figura 4.10: As árvores T_1, E e D (esquerda) e a árvore $T_1' = \{q\} \cup E \cup D$ (direita).

Levando-se em conta, novamente, as únicas diferenças existentes entre A e B, tem-se que

$$\Phi_F(B) - \Phi_F(A) = \sum_{Q \in A} \Phi_T(Q) - \sum_{R \in B} \Phi_T(R) = \Phi_T(T_1') - \Phi_T(D) - \Phi_T(E) - \Phi_T(T_1)\,.$$

Mas,

$$\begin{aligned}
\Phi_T(T_1') &= \sum_{x \in T_1'} r'(x) \\
&= r'(q) + \sum_{x \in E} r'(x) + \sum_{x \in D} r'(x) \\
&= r'(q) + \sum_{x \in E} r(x) + \sum_{x \in D} r(x) \\
&= r'(q) + \Phi_T(E) + \Phi_T(D)\,,
\end{aligned}$$

onde q é o nó raiz de T_1 e T_1', $r'(x)$ e $r(x)$ denotam o posto de x em T_1 e T_1', para todo nó x em T_1 e T_1', respectivamente, e $r(x) = r'(x)$ para todo nó x em T_1', com exceção do nó raiz, q, pois as árvores E e D são iguais às subárvores esquerda e direita de q em T_1'. Logo,

$$\Phi_F(B) - \Phi_F(A) = r'(q) - \Phi_T(T_1) = r'(q) - r(q) = \lg s'(q) - \lg s(q) = \lg\left(\frac{s'(q)}{s(q)}\right)\,,$$

onde $s'(q)$ e $s(q)$ são os tamanhos do nó q em T_1' e T_1, respectivamente. Como $s(q) = w(\alpha)$, onde α é a chave associada ao nó q, tem-se

$$\Phi_F(B) - \Phi_F(A) = \lg\left(\frac{s'(q)}{w(\alpha)}\right)\,.$$

Observe que o valor acima é maior ou igual a zero, sendo zero apenas se as subárvores E e D são vazias. Se este não for o caso, a operação que constrói T_1' causa um acréscimo de

potencial.

Nas provas dos lemas 4.4.3-4.4.6, as duas situações acima e suas "inversas" ocorrem. Em cada ocorrência, a diferença de potencial, $\Phi_F(B) - \Phi_F(A)$, entre as coleções A e B de árvores antes e após alguma operação foi expressa, *diretamente*, em termos da função Φ_T e das árvores que configuram a diferença entre A e B. Isto porque a lógica por trás de cada dedução é a mesma das deduções realizadas acima. Portanto, tentou-se evitar redundância.

Lema 4.4.3. O tempo amortizado, $\hat{t}$, da chamada SPLIT(α, q) é tal que

$$\hat{t} \leq \begin{cases} 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + \mathcal{O}(1), & \text{se } \alpha \text{ está na árvore} \\ 3 \cdot \lg\left(\dfrac{W}{\min\{w(\alpha_-), w(\alpha_+)\}}\right) + \mathcal{O}(1), & \text{caso contrário,} \end{cases}$$

onde q é um apontador para a raiz de uma árvore splay, α é uma chave (que pode ou não pertencer a árvore), W é a soma dos pesos dos nós da árvore, $w(\alpha)$ é o peso associado a α e $w(\alpha_-)$ (resp. $w(\alpha_+)$) é o peso associado à chave, α_- (resp. α_+), que precede (resp. sucede) α em um percurso em ordem simétrica pela árvore caso α esteja na árvore ou, em caso contrário, α_- (resp. α_+) precederia (resp. sucederia) α no mesmo percurso se α estivesse na árvore.

Demonstração. De acordo com a descrição de SPLIT() na Seção 4.3, pode-se escrever $\hat{t}$ como

$$\hat{t} = \hat{t}_a + \hat{t}_b \,,$$

onde $\hat{t}_a$ é o tempo amortizado da chamada a ACCESS() e $\hat{t}_b$ é o tempo amortizado das operações que sucedem a chamada a ACCESS(). De acordo com o Lema 4.4.2, sabe-se que

$$\hat{t}_a \leq \begin{cases} 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + 1, & \text{se } \alpha \text{ está na árvore} \\ 3 \cdot \lg\left(\dfrac{W}{\min\{w(\alpha_-), w(\alpha_+)\}}\right) + 1, & \text{caso contrário.} \end{cases}$$

Após a chamada a ACCESS(), a operação SPLIT() cria duas novas árvores splay, T_1' e T_2', eliminando a aresta que liga a raiz da árvore, T', resultante de ACCESS() a T_1' ou T_2'. Em seguida, a árvore T' é destruída. Seja q' um apontador para a raiz de T', e sejam q_1 e q_2 apontadores para as raízes de T_1' e T_2', respectivamente. Sem perda de generalidade, assuma que T_1' é a subárvore desconectada por SPLIT(). O valor de $\hat{t}_b$ é igual ao tempo gasto, t, para desconectar T_1' de T' mais a diferença de potencial das coleções de árvores

imediatamente antes e imediatamente depois da desconexão de T_1'. O valor de t é constante, pois se deve considerar que a desconexão é igual a uma constante vezes o tempo gasto em uma rotação, que se assumiu ser 1. Por sua vez, a diferença entre as duas coleções de árvores está em que uma contém T' mas não contém T_1' e $T_2' \cup \{x\}$ e vice-versa, onde x é o nó raiz de T' e $T_2' \cup \{x\}$ é a árvore resultante de T' após a desconexão de T_1'. Pelas definições de potencial de uma árvore e potencial de uma coleção de árvores, tem-se que a diferença de potencial das duas coleções acima é igual a $\Phi_T(T_1') + \Phi_T(T_2' \cup \{x\}) - \Phi_T(T')$, que é igual a $r'(x) - r(x) = \lg s'(x) - \lg s(x)$, onde $r'(x)$ (resp. $s'(x)$) é o posto (resp. tamanho) do nó x em $T_2' \cup \{x\}$ e $r(x)$ (resp. $s(x)$) é o posto (resp. tamanho) do nó x em T'. Mas, $s(x) = s'(x) + s(q_1)$, onde $s(q_1)$ é o tamanho do nó raiz de T_1'. Logo, conclui-se que

$$r'(x) - r(x) = \lg s'(x) - \lg s(x) = \lg\left(\frac{s'(x)}{s(x)}\right) = \lg\left(\frac{s'(x)}{s'(x) + s(q_1)}\right) \in (-\infty, 0]\,.$$

Consequentemente, tem-se que $1 + r'(x) - r(x) \leq 1$ e, portanto, o tempo amortizado, $\hat{t}$, é tal que

$$\hat{t} = \hat{t}_a + \hat{t}_b \leq \begin{cases} 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + \mathcal{O}(1), & \text{se } \alpha \text{ está na árvore} \\ 3 \cdot \lg\left(\dfrac{W}{\min\{w(\alpha_-), w(\alpha_+)\}}\right) + \mathcal{O}(1), & \text{caso contrário,} \end{cases}$$

onde $\mathcal{O}(1)$ se refere ao valor de $\hat{t}_b$, que é menor ou igual a $1 + r'(x) - r(x) + t \leq 1 + t \in \mathcal{O}(1)$. □

Lema 4.4.4. O tempo amortizado, $\hat{t}$, da chamada JOIN(q_1, q_2) é tal que

$$\hat{t} = 3 \cdot \lg\left(\frac{W}{w(\alpha)}\right) + \mathcal{O}(1)\,,$$

onde q_1 e q_2 são as raízes de duas árvores splay, T_1 e T_2, respectivamente, α é a chave de maior valor em T_1, W é a soma dos pesos dos nós de T_1 e T_2 e $w(\alpha)$ é o peso associado à chave α.

Demonstração. De acordo com a descrição de JOIN() na Seção 4.3, tem-se que o primeiro passo desta operação é encontrar a chave, α, de maior valor na árvore T_1. O passo seguinte é executar SPLAY() no nó x que contém α. Isto faz com que o nó x se torne a raiz da nova árvore, T_1'. Como α é a chave de maior valor, o nó x não possui filho direito e, portanto, pode-se conectar T_1' a T_2, fazendo com que T_2 seja a subárvore direita da raiz, x, de T_1'. A árvore resultante é $T' = T_1' \cup T_2$. Obviamente, assume-se que todas as chaves em T_2

são maiores ou iguais a α. Caso contrário, a árvore T não seria uma árvore de busca. Finalmente, as árvores T_1' e T_2 são destruídas, enquanto a árvore T permanece. Logo, assim como na prova do Lema 4.4.3, o tempo amortizado, $\hat{t}$, de JOIN(q_1, q_2) é igual à soma

$$\hat{t} = \hat{t}_a + \hat{t}_b \,,$$

onde $\hat{t}_a$ é o tempo amortizado da chamada a SPLAY() e $\hat{t}_b$ é o tempo amortizado das operações que sucedem a chamada a SPLAY(). De acordo com o Lema 4.4.1, sabe-se que

$$\hat{t}_a \leq 1 + 3 \cdot (r(q_1) - r(x)) = 1 + 3 \cdot (\lg s(q_1) - \lg s(x)) \leq 1 + 3 \cdot (\lg s(q_1) - \lg w(\alpha)) \;, \quad (4.1)$$

onde q_1 é o nó raiz de T_1, x é o nó de T_1 que contém α e $s(x)$ é o tamanho do nó x em T_1, com

$$s(x) \geq w(\alpha) \,.$$

Observe que se o tempo gasto com uma rotação for duplicado, então o tempo gasto com a busca pela chave de maior valor, α, pode ser embutido no tempo gasto com as rotações executadas por SPLAY(), não havendo, portanto, a necessidade de levar em conta, direta e explicitamente, o tempo gasto com as visitas aos nós de T_1 durante a busca. Por sua vez, o valor de $\hat{t}_b$ é igual ao tempo gasto, t, para conectar T_1' a T_2, gerando T', mais a diferença de potencial das coleções de árvores imediatamente antes e imediatamente depois da operação de conexão. O valor de t é constante. Já a diferença entre as duas coleções de árvores está em que uma contém T_1' e T_2 mas não contém T e vice-versa. Pelas definições de potencial de uma árvore e potencial de uma coleção de árvores, tem-se que a diferença de potencial das duas coleções acima é igual a $\Phi_T(T') - \Phi_T(T_1') - \Phi_T(T_2)$, que é igual a $r''(x) - r'(x) = \lg s''(x) - \lg s'(x)$, onde $r''(x)$ (resp. $s''(x)$) é o posto (resp. tamanho) do nó raiz, x, em T' e $r'(x)$ (resp. $s'(x)$) é o posto (resp. tamanho) do nó raiz x em T_1'. Observe que o nó x é a raiz de T' e T_1' e que, em ambos os casos, o nó x contém a chave α. Observe também que $s''(x) = s'(x) + s(q_2)$, onde $s(q_2)$ é o tamanho do nó raiz de T_2. Logo,

$$r''(x) - r'(x) = \lg s(q_2) = \lg (s'(x) + s(q_2) - s'(x)) = \lg (s''(x) - s'(x)) \leq \lg (W - s'(x)) \;,$$

pois $W = s''(x)$. Pela Eq. (4.1), tem-se que $\hat{t}_a \leq 1 + 3 \cdot (\lg s(q_1) - \lg w(\alpha))$. Mas, $s(q_1) = s'(x)$, pois os nós de T_1 e T_1' são exatamente os mesmos e os pesos dos nós não variam.

Logo,

$$\begin{aligned}
\hat{t} &= \hat{t}_a + \hat{t}_b \\
&\leq 1 + 3 \cdot (\lg s(q_1) - \lg w(\alpha)) + t + \lg (W - s'(x)) \\
&\leq 1 + 3 \cdot (\lg s'(x) - \lg w(\alpha)) + t + 3 \cdot \lg (W - s'(x)) \\
&= 3 \cdot \lg (W - w(\alpha)) + 1 + t \\
&= 3 \cdot \lg (W - w(\alpha)) + \mathcal{O}(1) \,.
\end{aligned}$$

□

Lema 4.4.5. O tempo amortizado, $\hat{t}$, da chamada $\textsc{Insert}(\alpha, q)$ é tal que

$$\hat{t} \leq 3 \cdot \lg \left(\frac{W - w(\alpha)}{\min\{w(\alpha_-), w(\alpha_+)\}} \right) + \lg \left(\frac{W}{w(\alpha)} \right) + \mathcal{O}(1) \,,$$

onde q é a raiz da árvore splay, T', na qual a chave α é inserida, W é a soma dos pesos dos nós da árvore splay T'' resultante da inserção de α em T' e $w(\alpha)$ é o peso associado à chave α.

Demonstração. Lembre-se de que toda chave do universo U que não está em T' pertence, por convenção, a uma árvore da coleção de árvores que possui apenas o nó raiz. Logo, a chave α está, inicialmente, em uma árvore, T_3, que contém apenas o nó raiz e α está associada a este nó. De acordo com o Algoritmo 4.3, o primeiro passo de $\textsc{Insert}(\alpha, q)$ consiste em executar $\textsc{Split}(\alpha, q)$, que devolve duas árvores splay, T_1 e T_2, tal que todas as chaves em T_1 são menores do que i e todas as chaves em T_2 são maiores do que α. O segundo e último passo constrói uma nova árvore splay, T'', tal que a raiz de T'' é um nó x contendo a chave α e as subárvores esquerda e direita de x são T_1 e T_2, respectivamente. Logo, o tempo amortizado, $\hat{t}$, de $\textsc{Insert}(\alpha, q)$ pode ser escrito como $\hat{t} = \hat{t}_a + \hat{t}_b$, onde $\hat{t}_a$ é o tempo amortizado do primeiro passo e $\hat{t}_b$ é o tempo amortizado do segundo passo. Pelo Lema 4.4.3,

$$\hat{t}_a \leq 3 \cdot \lg \left(\frac{W - w(\alpha)}{\min\{w(\alpha_-), w(\alpha_+)\}} \right) + \mathcal{O}(1) \,,$$

pois se assume que a chave α não está em T', o que implica que $s(q) = W - w(\alpha)$, onde $s(q)$ é o tamanho do nó raiz, q, de T' e W é a soma dos pesos de todos os nós em T' mais o peso $w(\alpha)$. Por sua vez, $\hat{t}_b = t + \Phi_T(T'') - (\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3))$, onde t é o tempo gasto para criar x, conectar T_1 e T_2 a x para gerar T'' e destruir T_1, T_2 e T_3, enquanto o termo $\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3)$ é a diferença de potencial entre as coleções de árvore imediatamente antes e imediatamente depois do segundo passo. Em particular, $\Phi_T(T'')$ é o potencial de T'', $\Phi_T(T_1)$ é o potencial de T_1, $\Phi_T(T_2)$ é o potencial de T_2, $\Phi_T(T_3)$ é o

potencial de T_3. Note que a diferença de potencial entre as duas coleções se deve apenas ao fato das árvores T_1, T_2 e T_3 existirem antes do segundo passo, mas não depois dele; por outro lado, a árvore T'' existe após o segundo passo, mas não antes dele. Por definição de potencial de árvore, $\Phi_T(T_3) = \lg w(\alpha)$. Por sua vez, $\Phi_T(T'') - \Phi_T(T_1) - \Phi_T(T_2) = s''(x)$, onde x é o nó raiz de T'' (isto é, aquele que contém a chave α) e $s''(x)$ é o tamanho de x em T''. Por definição de W, tem-se $W = s''(x)$, o que implica que o valor do termo $\hat{t}_b$ é igual a

$$t + \Phi_T(T'') - (\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3)) = t + \lg W - \lg w(\alpha) = t + \lg\left(\frac{W}{w(\alpha)}\right).$$

Logo,

$$\hat{t} = \hat{t}_a + \hat{t}_b \leq 3 \cdot \lg\left(\frac{W - w(\alpha)}{\min\{w(\alpha_-), w(\alpha_+)\}}\right) + \lg\left(\frac{W}{w(\alpha)}\right) + \mathcal{O}(1)\,,$$

pois t é uma constante. □

Lema 4.4.6. O tempo amortizado, $\hat{t}$, da chamada REMOVE(α, q) é tal que

$$\hat{t} \leq \begin{cases} 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + 3 \cdot \lg\left(\dfrac{W - w(\alpha)}{w(\alpha_-)}\right) + \mathcal{O}(1) & \text{se a chave } \alpha_- \text{ está em } T' \\ 3 \cdot \lg\left(\dfrac{W}{w(\alpha)}\right) + \mathcal{O}(1) & \text{caso contrário,} \end{cases}$$

onde q é a raiz da árvore splay, T', da qual a chave α é removida, W é a soma dos pesos dos nós da árvore splay T', $w(\alpha)$ é o peso associado à chave α e $w(\alpha_-)$ é o peso associado à chave α_-, que precede à chave α em um percurso em ordem simétrica por T', se α_- está em T'. Caso contrário, a chave α_- é indefinida e o valor atribuído a $w(\alpha_-)$ é ∞, por convenção.

Demonstração. De acordo com o Algoritmo 4.4, o primeiro passo de REMOVE(α, q) consiste em executar ACCESS(α, q), que modifica a estrutura da árvore T' de tal forma que o nó, x, contendo a chave α se torne a raiz da árvore resultante, T''. O segundo passo consiste em desconectar as subárvores, T_1 e T_2, do nó x em T'', gerando as árvores T_1, T_2 e T_3, onde T_3 é a árvore que contém apenas o nó x associado à chave i. A árvore T'' é destruída. O terceiro passo executa JOIN(q_1, q_2), onde q_1 e q_2 são as raízes de T_1 e T_2, respectivamente. Este passo faz com que uma nova árvore, T''', seja construída com os nós de T_1 e T_2 apenas. Então, o tempo amortizado, $\hat{t}$, de REMOVE(α, q) pode ser escrito como $\hat{t} = \hat{t}_a + \hat{t}_b + \hat{t}_c$, onde $\hat{t}_a$, $\hat{t}_b$ e $\hat{t}_c$ são os tempos amortizados do primeiro, segundo e terceiro passo, respectivamente. O Lema 4.4.2 diz que $\hat{t}_a$ é limitado superiormente pela

quantia

$$3 \cdot \lg \left(\frac{W}{w(\alpha)} \right) ,$$

onde W é a soma dos pesos de todos os nós existentes na árvore splay inicial, T', e $w(\alpha)$ é o peso associado à chave α, que se assume estar em T'. O segundo passo gasta um tempo constante, t, para desconectar as subárvores esquerda, T_1, e direita, T_2, do nó raiz da árvore T'' resultante do primeiro passo e para destruir a árvore T''. Ressalta-se que a destruição de T'' significa a remoção de T'' da coleção de árvores (o que é feito em tempo constante) e não a destruição de seus nós, que passam a ser nós de T_1, T_2 e T_3. Logo, tem-se

$$\hat{t}_b = t + (\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3)) - \Phi_T(T'') ,$$

onde $(\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3)) - \Phi_T(T'')$ é a diferença de potencial das coleções de árvores imediatamente antes e imediatamente depois do segundo passo. Em particular, $\Phi_T(T_1)$ é o potencial de T_1, $\Phi_T(T_2)$ é o potencial de T_2, $\Phi_T(T_3)$ é o potencial de T_3 e $\Phi_T(T'')$ é o potencial de T''. Note que a diferença de potencial entre as duas coleções se deve apenas ao fato da árvore T'' existir antes do segundo passo, mas não depois dele, enquanto as árvores T_1, T_2 e T_3 existem após o segundo passo, mas não antes dele. Por definição,

$$(\Phi_T(T_1) + \Phi_T(T_2) + \Phi_T(T_3)) - \Phi_T(T'') = \lg w(\alpha) - \lg W .$$

No terceiro passo, a árvore T_1 é transformada em uma árvore T_1' pela chamada Join(q_1, q_2). A raiz, q_1', de T_1' é o nó de T' associado com a chave α_-, ou seja, a chave que precede α em um percurso em ordem simétrica por T'. Observe que tal chave não existe se, e somente se, a árvore T_1 é vazia. Por enquanto, suponha que T_1 não é vazia. Então, o resultado de Join(q_1, q_2) é a árvore T''' que contém, exatamente, todos os nós de T_1 (ou, equivalentemente, T_1') e T_2. O Lema 4.4.4 estabelece que $\hat{t}_c$ é limitado superiormente pela quantia

$$3 \cdot \lg \left(\frac{W - w(\alpha)}{w(\alpha_-)} \right) + \mathcal{O}(1) ,$$

pois a chave α não está em nenhuma das duas árvores, T_1' e T_2, e a operação de *splaying* em Join(q_1, q_2) é executada no nó contendo α_-. Logo, tem-se que $\hat{t} = \hat{t}_a + \hat{t}_b + \hat{t}_c$ é tal que

$$\hat{t} \leq 3 \cdot \lg \left(\frac{W}{w(\alpha)} \right) + t + \lg w(\alpha) - \lg W + 3 \cdot \lg \left(\frac{W - w(\alpha)}{w(\alpha_-)} \right) + \mathcal{O}(1) .$$

Como t é uma constante e $\lg w(\alpha) - \lg W \leq 0$, pode-se concluir que

$$\hat{t} \leq 3 \cdot \lg \left(\frac{W}{w(\alpha)} \right) + 3 \cdot \lg \left(\frac{W - w(\alpha)}{w(\alpha_-)} \right) + \mathcal{O}(1) .$$

Quando T_1 for uma árvore vazia, JOIN(q_1, q_2) devolve T_2 e as coleções de árvores antes e depois desta chamada são as mesmas. Neste caso, o valor de $\hat{t}_c$ é constante e o valor de $\hat{t}$ é tal que

$$\hat{t} \leq 3 \cdot \lg\left(\frac{W}{w(\alpha)}\right) + \mathcal{O}(1)\,.$$

□

Os lemas anteriores permitem que se estabeleça a complexidade amortizada das operações de uma sequência S de m operações de acesso, inserção e remoção em uma árvore splay inicialmente vazia. De fato, se W_i denota a quantidade de nós envolvidos na árvore ou árvores envolvidas na i-ésima operação de S, então o tempo amortizado total, $\hat{t}_S$, de S é

$$\hat{t}_S = \sum_{i=1}^{m} \hat{t}_i \in \mathcal{O}(m + \sum_{i=1}^{m} \lg W_i)\,,$$

onde $\hat{t}_i$ é o tempo amortizado da i-ésima operação de S. Para obter a cota superior acima, basta atribuir o valor 1 a $w(\alpha)$, para toda chave α em U. Daí, pelos Lemas 4.4.2, 4.4.5 e 4.4.6,

$$\hat{t}_i \leq k \cdot (\lg W_i + 1)\ ,$$

para todo $i = 1, \ldots, m$, onde k é uma constante positiva. Isto implica que $\sum_{i=1}^{m} \hat{t}_i$ é tal que

$$\sum_{i=1}^{m} \hat{t}_i \leq k \cdot \sum_{i=1}^{m} \lg W_i + k \cdot m \in \mathcal{O}(m + \sum_{i=1}^{m} \lg W_i)\,.$$

Note que se $n = \max_{i=1}^{m}\{W_i\}$, então o tempo amortizado de cada operação de S está em $\mathcal{O}(\lg n)$, o que mostra que árvores splays são tão eficientes quanto árvores balanceadas em uma perspectiva de complexidade amortizada. O teorema a seguir conclui a discussão acima:

Teorema 4.4.7 (Teorema do Equilíbro com Atualizações (SLEATOR; TARJAN, 1985))**.** Seja S uma sequência de m operações de acesso, inserção e busca em uma árvore splay inicialmente vazia. Então, o tempo amortizado total de execução das operações de S é $\mathcal{O}(m + \sum_{i=1}^{m} \lg W_i)$, onde W_i é o número de nós na árvore ou árvores envolvidas na i-ésima operação de S.

5 Árvores ST

Este capítulo descreve uma *árvore dinâmica* (ou *árvore link-cut*), denominada *árvore ST* (do inglês *ST-tree*), que foi proposta por Sleator e Tarjan em (SLEATOR; TARJAN, 1983, 1985), para resolver o *problema da árvore dinâmica*. Neste trabalho, a árvore ST é usada na implementação da estrutura de conectividade dinâmica descrita no Capítulo 6. Esta, por sua vez, é empregada na implementação dos testes de conectividade e de ancestral comum mais próximo realizados pelo principal objeto de estudo desta monografia: o algoritmo de Diks e Stanczyk para calcular emparelhamentos perfeitos em grafos cúbicos e sem pontes (veja a Seção 3.3).

A Seção 5.1 define o problema da árvore dinâmica, apresenta algumas de suas aplicações e cita as soluções mais conhecidas (entre elas, a solução baseada na árvore ST). A Seção 5.2 descreve a representação interna da árvore ST, que é baseada nas árvores splay introduzidas no Capítulo 4. As Seções 5.3 e 5.4 apresentam, respectivamente, as operações internas e as operações externas da árvore ST. As operações externas são visíveis ao usuário e são implementadas com base nas operações internas. Finalmente, a Seção 5.5 apresenta uma análise da complexidade amortizada das operações externas descritas na Seção 5.4.

5.1 O problema da árvore dinâmica

O *problema da árvore dinâmica* pode ser resumidamente enunciado como aquele de manter uma coleção de nós distintos representada por uma floresta e sujeita às seguintes operações:

- MAKETREE(k): Cria um nó com chave k, que passa a ser raiz de uma nova árvore contendo este nó;
- LINK(v, w): Adiciona uma aresta do nó v para o nó w, tornando v um filho de w na floresta e combinando suas duas árvores em uma. Assume que v e w estão em

árvores diferentes e que v é a raiz da árvore que o contém;

- CUT(v): Remove a aresta do nó v para seu pai, dividindo sua árvore em duas e tornando v a raiz da nova árvore. Assume que v não é uma raiz da árvore que o contém;
- FINDROOT(v): Encontra e devolve o nó raiz da árvore que contém v;
- FINDPARENT(v): Encontra e devolve o pai do nó v na árvore que o contém. Se v for a raiz da árvore, então o seu pai é o nó nulo.

No problema da árvore dinâmica, assume-se que a coleção de nós está inicialmente vazia e que cada nó, ao ser adicionado à coleção, é representado por uma árvore enraizada e contendo apenas o nó, que é obviamente a raiz da árvore. As operações descritas acima modificam ou consultam a floresta que representa a coleção de nós. Em particular, a operação LINK() combina duas árvores, enquanto a operação CUT() divide uma árvore em duas. Uma árvore que suporta as operações acima é denominada de *árvore dinâmica*. É importante ressaltar que as árvores enraizadas que compõem a floresta dinâmica são, em qualquer instante de tempo, disjuntas; isto é, elas não compartilham vértices nem arestas.

A Figura 5.1 ilustra as operações suportadas pelas árvores dinâmicas. Na Figura 5.1(a), tem-se uma floresta com cinco árvores, cada qual criada com uma chamada à operação MAKETREE(). Note que cada nó é a raiz de sua própria árvore. Na Figura 5.1 (b), tem-se a floresta resultante das operações LINK(b, a), LINK(c, a), LINK(d, a) e LINK(e, b) sobre as árvores em (a). Neste ponto, todos os nós fazem parte de uma única árvore enraizada em a. Observe que os nós resultantes das chamadas FINDPARENT(e), FINDPARENT(b), FINDPARENT(c) e FINDPARENT(d) são, respectivamente, b, a, a e a. Finalmente, na Figura 5.1 (c), tem-se a floresta obtida após as execuções de CUT(c) e CUT(d) sobre a floresta em (b). Note que c e d voltam a ser raízes de suas respectivas árvores, como em (a).

Por razões que se tornarão evidentes mais adiante, o pai e os filhos de qualquer nó de uma árvore dinâmica são definidos em relação ao nó raiz da árvore. Além disso, as arestas da árvore dinâmica são consideradas como sendo "arcos orientados" do nó filho para o nó pai.

Uma maneira óbvia de representar árvores dinâmicas é por uma estrutura de dados do tipo árvore em que cada nó contém um apontador para o nó pai. Com esta representação, cada execução de LINK(), CUT() e FINDPARENT() é realizada em tempo constante,

enquanto uma execução de FINDROOT() leva, *no pior caso*, $\Theta(n)$ unidades de tempo em uma floresta com n nós. Com o intuito de reduzir a complexidade de pior caso das operações sobre árvores dinâmicas, Sleator e Tarjan propuseram em (SLEATOR; TARJAN, 1983) uma nova estrutura de dados para representar árvores dinâmicas. Esta estrutura vem em duas versões.

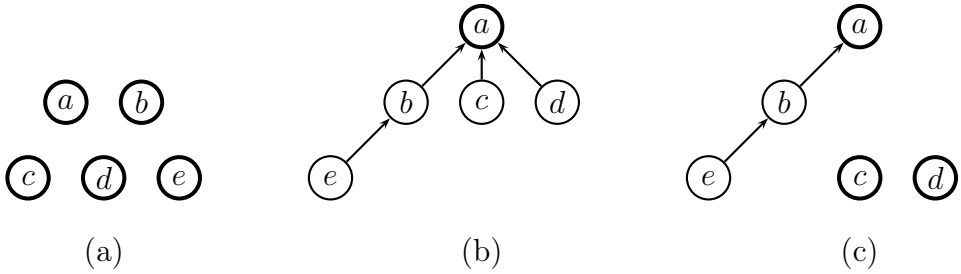


Figura 5.1: Operações em uma árvore dinâmica com 5 nós. As raízes estão destacadas.

A primeira versão da estrutura de dados proposta por Sleator e Tarjan em (SLEATOR; TARJAN, 1983) possibilita que cada operação sobre a floresta dinâmica tenha complexidade amortizada $\mathcal{O}(\lg n)$, onde n é o número total de nós das árvores envolvidas em uma sequência de pior caso de operações. A segunda versão faz com que a complexidade (não amortizada) de cada operação individual seja $\mathcal{O}(\lg n)$, onde n é o número total de nós das árvores envolvidas na operação. A grande desvantagem de ambas é que, devido à quantidade de detalhes envolvidos e à pouca clareza da descrição, elas são extremamente difíceis de implementar na prática. Felizmente, os autores apresentaram uma terceira versão em (SLEATOR; TARJAN, 1985), muito mais simples e baseada nas árvores splay do Capítulo 4.

As operações que atuam sobre a terceira versão apresentam a mesma complexidade amortizada das que atuam sobre a primeira. Mais especificamente, a partir de uma floresta vazia, qualquer sequência de m operações da árvore dinâmica levará tempo (de pior caso) $\Omega(m \cdot \lg n)$, mas uma operação individual pode levar mais do que $\mathcal{O}(\lg n)$ unidades de tempo, onde n é o número total de nós distintos envolvidos na sequência de operações. A terceira versão da árvore ST é aquela efetivamente implementada e discutida neste trabalho.

Várias outras estruturas de dados para resolver o problema da árvore dinâmica foram propostas após o surgimento da árvore ST. Entre elas estão a *ET-tree* (MILTERSEN et al.,

1994; TARJAN, 1997), *topology tree* (FREDERICKSON, 1985, 1997a, 1997b), *top tree* (ALSTRUP et al., 1997, 2005; TARJAN; WERNECK, 2005) e *RC-tree* (ACAR et al., 2004; ACAR; BLELLOCH; VITTES, 2005). Uma comparação experimental detalhada entre todas essas estruturas e a própria árvore ST pode ser encontrada em (WERNECK, 2006; TARJAN; WERNECK, 2010).

Todas as estruturas de dados citadas acima permitem que o problema da árvore dinâmica seja resolvido em tempo amortizado $\mathcal{O}(\lg n)$ por operação, onde n é o número total de nós das árvores envolvidas em uma sequência de pior caso de operações. Além disso, todas elas obtêm tal complexidade usando a mesma estratégia: mapear a árvore a ser representada em uma árvore balanceada, que é efetivamente a árvore armazanada na memória do computador. A diferença entre elas reside na técnica utilizada para realizar este mapeamento (WERNECK, 2006). As três versões da árvore ST propostas por Sleator e Tarjan utilizam a técnica de mapeamento conhecida por *decomposição em caminhos* (veja Seção 5.2).

Em (PĂTRAŞCU; DEMAINE, 2004), os autores mostraram, usando o modelo de computação denominado *cell probe*, que o tempo de pior caso de uma operação sobre uma árvore dinâmica está em $\Omega(\lg n)$. Esta cota inferior não se aplica à versão "estática" problema da conectividade, ou seja, quando arestas não podem ser adicionadas ou removidas. Logo, do ponto de vista da complexidade amortizada, as soluções baseadas nas árvores em (SLEATOR; TARJAN, 1983, 1985; MILTERSEN et al., 1994; TARJAN, 1997; FREDERICKSON, 1985, 1997a, 1997b; ALSTRUP et al., 1997, 2005; TARJAN; WERNECK, 2005) são ótimas.

O problema da árvore dinâmica pode também ser estendido para considerar outras operações. A principal delas é a operação EVERT(), que faz com que determinado nó se torne a raiz de sua árvore invertendo a orientação das arestas no caminho desse nó até a raiz original (veja a Figura 5.2). Esta operação, que foi indispensável para a realização deste trabalho – *por motivos que se tornarão evidentes no capítulo seguinte*, requer alterações especiais para operar sobre a estrutura de dados descrita em (SLEATOR; TARJAN, 1985). Embora os autores sejam extremamente vagos quanto aos detalhes de implementação da operação EVERT(), as informações necessárias podem ser encontradas em (RADZIK, 1998). Neste artigo, o autor fornece não só os detalhes para uma implementação completa da operação EVERT(), como também outras informações importantes, que serão discutidas no Capítulo 7.

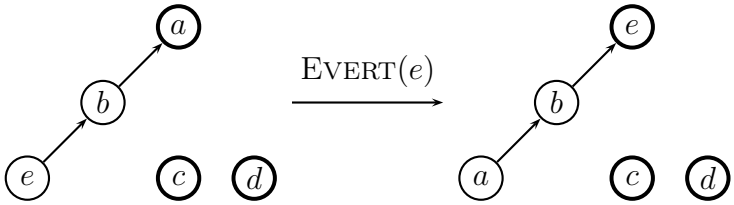


Figura 5.2: EVERT() sendo executada na árvore da Figura 5.1(c).

Outras extensões para o problema da árvore dinâmica levam em conta operações relacionadas ao custo de nós ou arestas da floresta. São elas: FINDCOST(v), que retorna o custo associado ao nó v (ou, respectivamente, à aresta que o liga ao seu pai), FINDMIN(v), que retorna o nó de menor custo no caminho entre v e a raiz de sua árvore, e ADDCOST(v, x), que acrescenta o valor x ao custo de cada nó no caminho de v à raiz da sua árvore.

A árvore ST com custos possui importantes aplicações em problemas de otimização de redes, melhorando os limites assintóticos dos tempos de execução de algoritmos para encontrar fluxos máximos, de custo mínimo ou acíclicos (AHUJA; ORLIN; TARJAN, 1989; GOLDBERG; TARJAN, 1989, 1990; GOLDBERG; TARDOS; TARJAN, 1990), além de poderem ser utilizadas para resolver o problema da árvore geradora mínima (FREDERICKSON, 1985). Porém, como neste trabalho não há preocupação com custos, tanto nos vértices quanto nas arestas, as operações acima que envolvem custo não serão consideradas.

As duas aplicações da árvore ST que interessam para este trabalho são (i) manter uma árvore geradora (qualquer) de um grafo dinâmico (isto é, sujeito a inserções e remoções de vértices e arestas) e (ii) encontrar o ancestral comum mais próximo (LCA, do inglês *Lowest Common Ancestor*) de dois vértices nessa árvore em tempo $\mathcal{O}(\lg n)$ (onde n é o número de nós na árvore). A maneira exata como a árvore geradora é mantida e os conceitos relacionados a algoritmos dinâmicos em grafos serão explicados em detalhes no Capítulo 6, enquanto o algoritmo para encontrar o LCA de dois vértices será visto no Capítulo 7.

5.2 Representação interna

Como mencionado na seção anterior, uma árvore ST é representada, no computador, por uma árvore balanceada. Esta representação "interna" não é percebida pelo "usuário",

que vê a árvore como sendo aquela resultante das operações LINK(), CUT() e EVERT(), como nas Figuras 5.1 e 5.2. No entanto, todas as operações atuam, de fato, sobre a representação interna. O mapeamento entre a árvore percebida pelo usuário e a sua representação interna por uma árvore balanceada é realizada por uma partição da primeira em caminhos disjuntos (por vértices). Cada caminho é mapeado para uma árvore balanceada. As árvores balanceadas que representam os diversos caminhos são conectadas para formar uma única árvore, que é a representação interna da árvore ST percebida pelo usuário.

Sleator e Tarjan denominaram a árvore percebida pelo usuário de *árvore real*, enquanto a sua representação interna foi denominada de *árvore virtual*[1] (SLEATOR; TARJAN, 1985). Estas denominações são confusas, pois elas são o oposto da analogia tradicional que se faz com os termos imagem real e imagem virtual em física óptica. De fato, o que o usuário percebe – a chamada árvore real – não é a representação concreta da árvore – a chamada árvore virtual. Logo, seria mais natural denominar de "virtual" o que o usuário percebe e, de "real" a representação interna da árvore. No entanto, para se manter coerente com a literatura da área, seguir-se-á, neste texto, a mesma convenção adotada por Sleator e Tarjan.

A *árvore virtual* contém exatamente os mesmos n nós que a árvore real correspondente, mas a estrutura na qual eles estão armazenados é diferente. Cada nó de uma árvore virtual possui apontadores para o nó pai e para seus filhos esquerdo e direito (apontadores que podem ou não ser nulos). Além disso, cada nó de uma árvore virtual pode estar associado a zero ou mais *filhos do meio* (isto é, filhos que apontam para ele, mas que não são filhos esquerdos nem direitos). Diferentemente do que ocorre nas árvores reais, existem dois tipos de arestas nas árvores virtuais: as arestas *tracejadas*, que ligam um nó a seus filhos do meio, e as arestas *sólidas*, que ligam um nó a seus filhos esquerdo e direito (veja a Figura 5.3).

As componentes conexas formadas por arestas sólidas da árvore virtual são denominadas *(sub)árvores sólidas* e são representadas utilizando as árvores splay do Capítulo 4. Dessa maneira, a árvore virtual pode ser vista como uma hierarquia de árvores splay conectadas entre si por arestas tracejadas. Por exemplo, a árvore virtual da Figura 5.3 possui seis subárvores sólidas, que são compostas pelos vértices: 1) q, l, i, f, c, b e a; 2) p; 3) o, k, h e e; 4) n; 5) r, m, j, g e d; e 6) w, v, u, t e s. A raiz de uma árvore sólida é uma *raiz sólida* e, para cada nó v em uma árvore virtual, denota-se por RAIZSÓLIDA(v) a raiz da árvore sólida que contém v. Por exemplo, RAIZSÓLIDA(d) = j na árvore da Figura 5.3.

[1] Também denominada árvore "sombra" (do inglês, *shadow tree*.)

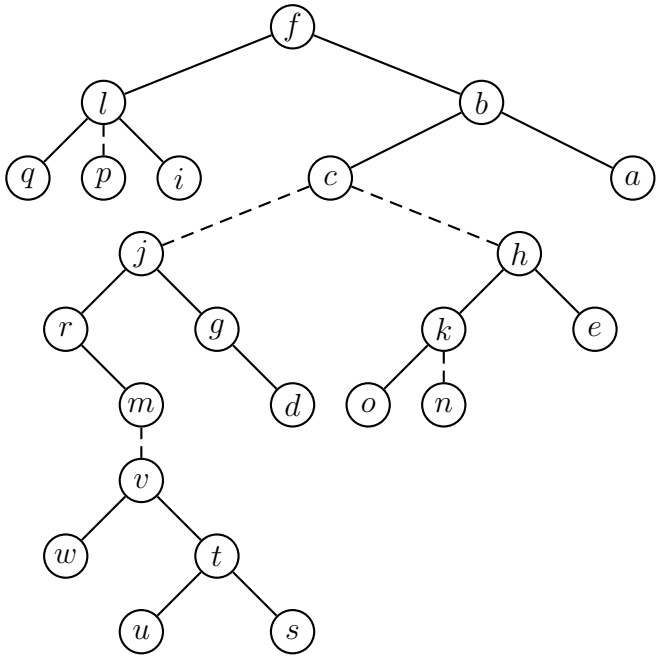


Figura 5.3: Uma árvore virtual que equivale à árvore real da Figura 5.4.

Cada subárvore sólida de uma árvore virtual representa um caminho da árvore real correspondente. Portanto, uma árvore virtual induz uma decomposição em caminhos da árvore real. Por exemplo, a árvore real representada pela árvore virtual da Figura 5.3 é aquela da Figura 5.4. A Figura 5.5 ilustra uma decomposição em caminhos da árvore real da Figura 5.4. Esta decomposição é a induzida pela árvore virtual da Figura 5.3. As arestas de um mesmo caminho são representadas por arcos sólidos orientados. As demais arestas são segmentos de reta pontilhados que conectam os diversos caminhos da decomposição.

Ao se comparar as Figuras 5.3 e 5.5, pode-se perceber que cada subárvore sólida da árvore virtual corresponde a um caminho distinto da decomposição em caminhos da árvore real. Em particular, a relação entre uma árvore real e sua representação por uma árvore virtual pode ser definida formalmente. Uma árvore virtual, V, representa uma árvore real, T, se, e somente se, T e V possuem o mesmo conjunto de nós e, para cada nó x,

$$\textsc{Pai}_T(x) = \begin{cases} \textsc{Sucessor}_V(x) & \text{se o sucessor de } x \text{ existe} \\ \textsc{Pai}_V(\textsc{RaizSólida}(x)) & \text{caso contrário} \end{cases} \tag{5.1}$$

onde $\textsc{Pai}_T(x)$ e $\textsc{Sucessor}_V(x)$ são, respectivamente, o nó pai de x em T e o nó sucessor de x em V em um percurso em ordem simétrica na subárvore sólida de V que contém o nó x.

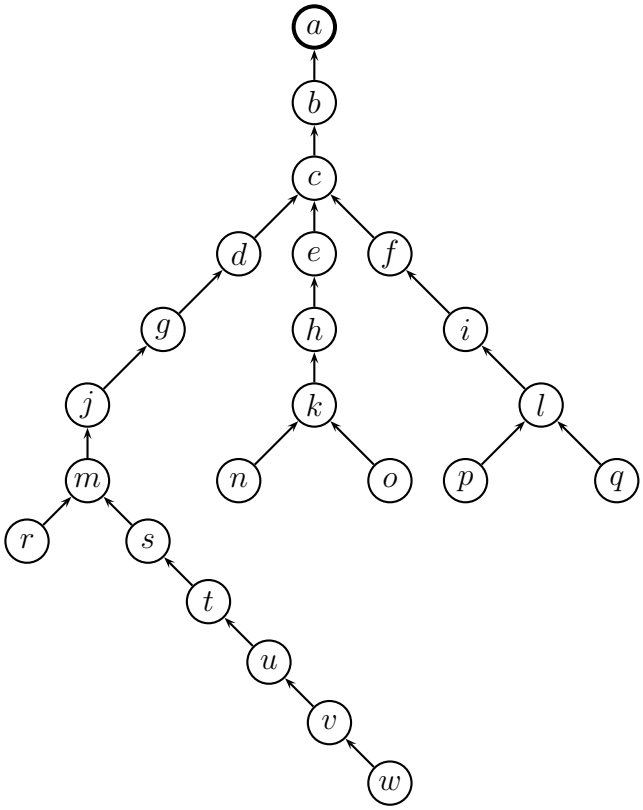


Figura 5.4: Uma árvore real retirada de (SLEATOR; TARJAN, 1985). Raiz em destaque.

Informalmente, a relação da Eq. (5.1) diz que o pai de um nó, x, na árvore real equivale ao sucessor de x, em um percurso em ordem simétrica, na sua subárvore sólida na árvore virtual. Se x não possuir sucessor (i.e., se x for o último nó da sequência resultante do percurso), seu pai na árvore real é o pai da raiz de sua subárvore sólida na árvore virtual. Para tornar esta relação mais evidente, considere o percorrimento em ordem simétrica de cada subárvore sólida da árvore virtual da Figura 5.3. Em tais percursos, obtêm-se as sequências

- q, l, i, f, c, b, a;
- p;
- o, k, h, e;
- n;
- r, m, j, g, d; e

- w, v, u, t, s.

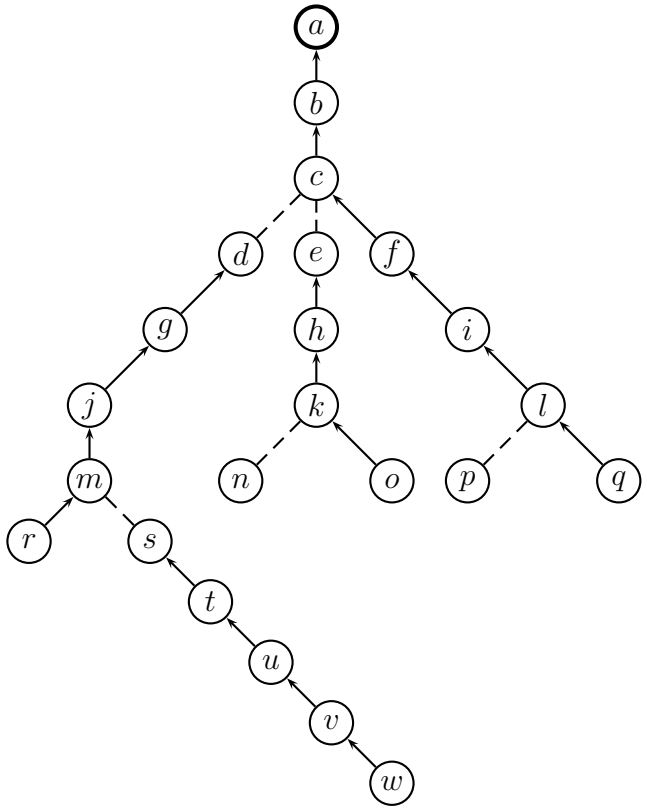


Figura 5.5: Decomposição em caminhos da árvore real da Figura 5.4.

Cada uma das sequências acima corresponde a um caminho dirigido "para cima" na árvore real. O último nó de cada caminho (isto é, aquele que não possui sucessor no caminho) está conectado, na árvore real, ao pai da raiz de sua árvore sólida na árvore virtual correspondente (veja as Figuras 5.3 e 5.5). Este nó pai é nulo ou é outro nó da árvore virtual conectado através de uma aresta pontilhada. Para um exemplo do primeiro caso, considere o nó a. Observe que, neste caso, o nó é a raiz da árvore real. Para um exemplo do segundo caso, considere o nó s. Este nó está conectado, na árvore real, ao nó m, que é o pai do nó v na árvore virtual. Por sua vez, o nó v é o nó raiz da árvore sólida que contém s.

Cada nó da árvore virtual possui um apontador para o nó pai (na própria árvore virtual), um apontador para o filho esquerdo e um apontador para o filho direito. Não há apontador de um nó para seu filho do meio. Isto é importante, pois um mesmo nó pode possuir $\Theta(n)$ filhos do meio, mas no máximo um esquerdo e um direito, numa árvore de n nós.

A principal vantagem em se utilizar árvores balanceadas na composição das árvores virtuais está no fato delas representarem caminhos arbitrariamente longos por árvores binárias com altura possivelmente menor. Cada caminho da árvore real poderia, em princípio, ser representado na árvore virtual por qualquer tipo de árvore balanceada; por exemplo, a árvore rubro-negra (CORMEM et al., 2009). Infelizmente, o fato de cada subárvore sólida ser balanceada não implica que a árvore virtual inteira seja. Em particular, se a árvore rubro-negra fosse utilizada, o tempo amortizado de uma operação individual sobre a árvore virtual pode ser $\Theta(\lg^2 n)$, como provado por Sleator e Tarjan em (SLEATOR; TARJAN, 1983), onde n é o número de nós da árvore virtual. Esta foi a principal motivação para a criação de novas árvores "balanceadas", tais como a árvore splay do Capítulo 4. A adoção da árvore splay na representação de caminhos não apenas garante que o tempo amortizado de uma operação individual sobre qualquer subárvore sólida é $\mathcal{O}(\lg n)$, mas também que a mesma operação sobre a árvore virtual inteira leva tempo amortizado $\mathcal{O}(\lg n)$.

Após esta visão geral da representação interna de uma árvore ST, pode-se explicar os detalhes que fazem com que tal representação suporte a operação EVERT(). Lembre-se de que esta operação faz com que um nó, v, torne-se a raiz de sua árvore real através da inversão da orientação do caminho entre v e a atual raiz da árvore. Para implementar tal operação, acrescenta-se um *bit* a cada nó w da árvore, denominado *bit de inversão* e denotado por $\text{B}(w)$, que tem como propósito indicar se os significados de "filho esquerdo" e "filho direito" do nó estão invertidos na subárvore sólida enraizada em w (veja a Figura 5.6).

A única mudança causada na relação estabelecida pela Eq. (5.1) é uma possível alteração na ordem simétrica dos nós. Para encontrar a *verdadeira ordem simétrica* em uma subárvore sólida com bits de inversão, é preciso considerar quem são os *verdadeiros filho esquerdo* e *filho direito* de cada nó. Seja $X(v)$ o resultado da operação binária *ou-exclusivo* sobre os bits de inversão no caminho entre o nó v e a raiz de sua árvore sólida. Um filho, u, de um nó v é o *verdadeiro filho esquerdo* de v se u é o filho esquerdo de v e $X(v) = 0$ ou se u é o filho direito de v e $X(v) = 1$. Caso contrário, u é o *verdadeiro filho direito de* v.

As duas próximas seções se destinam a explicar como as operações que atuam sobre árvores dinâmicas são implementadas em árvores ST. O importante, neste ponto, é ressaltar que a estrutura de dados descrita nesta seção foi desenvolvida com o intuito de garantir que, em qualquer sequência de operações discutidas anteriormente sobre uma floresta com n nós, a complexidade amortizada de qualquer uma dessas operações seja

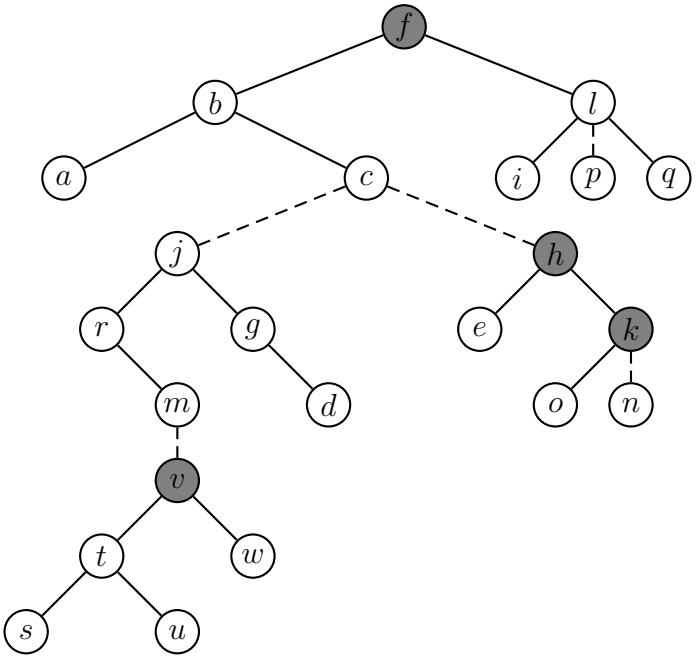


Figura 5.6: Uma árvore virtual correspondente à árvore real da Figura 5.4. Nós com bits de inversão iguais a 1 são destacados em cinza.

$\mathcal{O}(\lg n)$.

5.3 Operações primitivas

O subconjunto de operações que atuam sobre uma árvore dinâmica e relevante para o trabalho ora descrito é composto das operações: MakeTree(), Link(), Cut(), FindRoot(), FindParent() e Evert(). Essas operações são aquelas visíveis ao usuário da árvore dinâmica. Na árvore ST, tais operações são implementadas com o auxílio de quatro operações primitivas de reestruturação, que são invisíveis ao usuário da árvore; a saber:

- Rotate(v): Funciona da mesma maneira que nas árvores splay, pois ignora os filhos do meio de um nó. Isto é, apenas arestas sólidas são rotacionadas e filhos do meio não mudam de pai durante a rotação. Logo, v deve ser um filho esquerdo ou direito de seu pai. Esta operação presume que os bits de inversão de v e seu pai são iguais a 0;

- Unreverse(v): Restabelece a orientação "esquerda-direita" localmente correta no nó v verificando se seu bit de inversão é igual a 1 e, em caso positivo, atribuindo-lhe

valor 0, trocando de posição os filhos esquerdo e direito de v e invertendo seus bits de inversão;

- SPLICE(v): Faz com que o nó v, que deve ser um filho do meio de seu pai, u, passe a ser um filho esquerdo e o antigo filho esquerdo de u, se havia algum, passe a ser um filho do meio (veja a Figura 5.8). Presume que o bit de inversão de u é igual a 0; e
- SWITCHBIT(v): Muda o valor do bit de inversão do nó v para o oposto e, consequentemente, inverte a ordem simétrica na subárvore sólida com raiz em v.

Diferentemente de SWITCHBIT(v), que tem como efeito inverter a direção das arestas do caminho (da árvore real) representado pela subárvore sólida enraizada em v, as funções ROTATE(), UNREVERSE() e SPLICE(), alteram apenas a árvore virtual, não acarretando nenhuma modificação estrutural na árvore real. Como ROTATE() é basicamente a mesma operação do Capítulo 4, a explicação não será repetida. A Figura 5.7 ilustra a operação UNREVERSE().

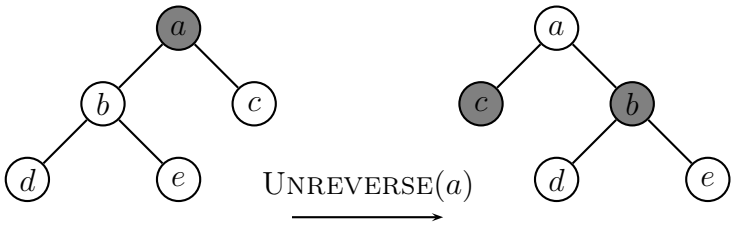


Figura 5.7: Exemplo de UNREVERSE().

A operação SWITCHBIT() é utilizada pela operação UNREVERSE(), como pode ser visto no Algoritmo 5.1. Finalmente, há um bom motivo para sempre trocar a subárvore enraizada no filho esquerdo, b, do pai, a, do nó c, ao invés da subárvore enraizada no filho direito, pela subárvore enraizada no filho do meio, c, na chamada SPLICE(c) (refira-se à Figura 5.8): o subcaminho da árvore real representado pela subárvore esquerda do pai de c está mais distante da raiz da árvore *real* do que aquele representado pela subárvore direita.

5.3.1 Splaying

As quatro operações primitivas são utilizadas em uma versão estendida da heurística de *splaying*, denominada VIRTUALSPLAY(), que ainda funciona movendo determinado nó

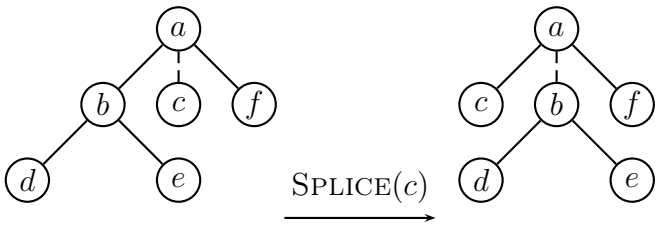


Figura 5.8: Exemplo de SPLICE().

Algoritmo 5.1 UNREVERSE(x)

Entrada: Um nó x em uma árvore virtual V
Saída: Nenhuma
  se B(x) = 1 **então**
    $t \leftarrow$ ESQUERDO(x)
    ESQUERDO(x) $\leftarrow$ DIREITO(x)
    DIREITO(x) $\leftarrow t$
    se ESQUERDO(x) $\neq$ **nil então**
      SWITCHBIT(ESQUERDO(x))
    fim se
    se DIREITO(x) $\neq$ **nil então**
      SWITCHBIT(DIREITO(x))
    fim se
  fim se

até a raiz, mas que foi adaptada para operar sobre a estrutura da árvore virtual descrita na seção anterior. Embora seja possível implementar a operação VIRTUALSPLAY() como uma única passada de baixo para cima na árvore virtual, neste trabalho será descrita a maneira mais detalhada (e didática) que consiste de três passadas distintas de baixo para cima.

Para facilitar a descrição da operação VIRTUALSPLAY(), utiliza-se uma versão modificada da operação SPLAY(), denominada SOLIDSPLAY(), que move um nó somente até a raiz de sua árvore sólida, aplicando UNREVERSE() antes de cada rotação (veja o Algoritmo 5.2). Assume-se que a função ISSOLIDROOT() retorna verdadeiro se, e somente se, um nó não possui pai ou é um filho do meio de seu pai (isto é, ele é a raiz de uma árvore sólida).

Considere que x é o nó que se deseja mover para a raiz da árvore virtual. Durante a primeira passada, prossegue-se subindo em direção à raiz, realizando-se operações SOLIDSPLAY() em cada árvore sólida encontrada. Isto é, são feitas rotações quase exatamente como especificado na Seção 4.2, acrescentando apenas as duas seguintes regras: 1) antes

Algoritmo 5.2 SolidSplay(x)

Entrada: Um nó x em uma árvore virtual V
Saída: Nenhuma

```
enquanto ¬IsSolidRoot(x) faça
  y ← Pai(x)
  z ← Pai(y)
  se z ≠ nil então
    Unreverse(z)
  fim se
  Unreverse(y)
  Unreverse(x)
  se IsSolidRoot(y) então
    Rotate(x)
  senão
    ambosEsquerdos ← (x = Esquerdo(y)) e (y = Esquerdo(z))
    ambosDireitos ← (x = Direito(y)) e (y = Direito(z))
    se ambosEsquerdos ou ambosDireitos então
      Rotate(y)
      Rotate(x)
    senão
      Rotate(x)
      Rotate(x)
    fim se
  fim se
fim enquanto
```

de qualquer chamada a ROTATE(), todos os nós envolvidos devem ser "desinvertidos" (se necessário for) com UNREVERSE(), do mais alto para o mais baixo na árvore, e 2) toda vez que x se tornar um filho do meio, a operação continua a partir do pai de x. Ao final dessa primeira passada, o caminho de x para a raiz da árvore virtual consiste inteiramente de arestas tracejadas. Além disso, todos os nós nesse caminho estão com o bit de inversão igual a zero.

A Figura 5.9 exemplifica o efeito da primeira passada de VIRTUALSPLAY().

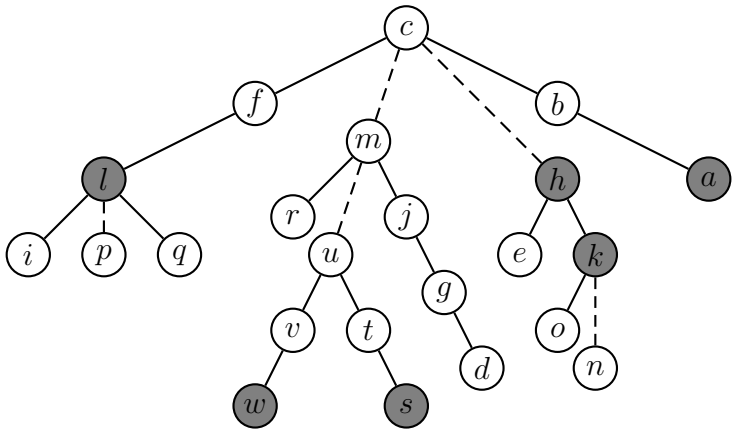


Figura 5.9: Árvore da Figura 5.6 após a primeira passada de VIRTUALSPLAY(u).

Na segunda passada de VIRTUALSPLAY(), aplica-se SPLICE() a x e a seus ancestrais para tornar todos, com exceção da raiz da árvore virtual, filhos esquerdos. O que, consequentemente, faz com que x e seus ancestrais se tornem parte de uma mesma subárvore sólida.

A Figura 5.10 exemplifica o efeito da segunda passada de VIRTUALSPLAY().

Por fim, a terceira passada consiste em repetir o procedimento comum de *splaying*, fazendo com que x se torne raiz da árvore virtual. Note que, durante a terceira passada, só é possível a ocorrência de casos *Zig* e *Zig-Zig*, pois todos os nós são filhos esquerdos (veja Seção 4.2).

A Figura 5.11 exemplifica o efeito da terceira passada de VIRTUALSPLAY().

O Algoritmo 5.3 contém o pseudocódigo da operação *splaying* para árvores virtuais. Além de fazer com que os principais nós envolvidos nas chamadas tornem-se raízes de suas árvores virtuais, a operação VIRTUALSPLAY() possui o papel importantíssimo de balancear implicitamente as árvores virtuais, o que contribui para reduzir o tempo amortizado

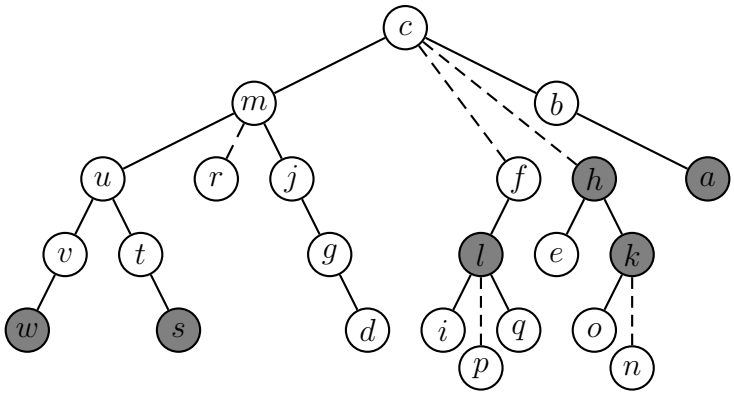


Figura 5.10: Árvore da Figura 5.6 após a segunda passada de VirtualSplay(u).

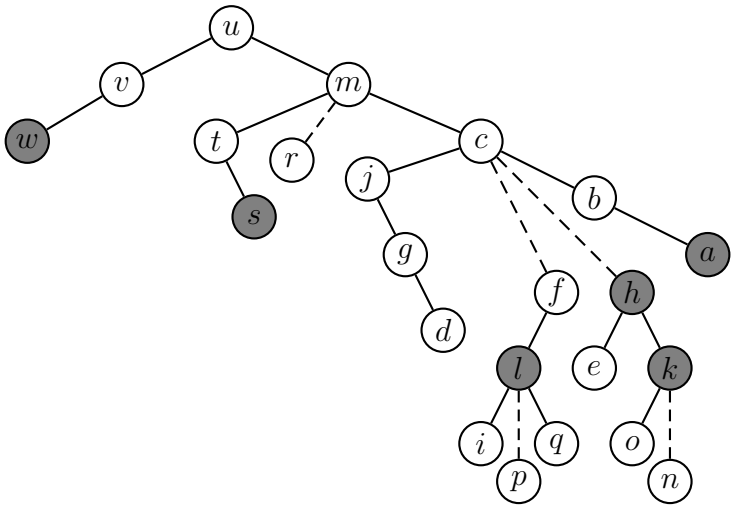


Figura 5.11: Árvore da Figura 5.6 após a terceira passada de VirtualSplay(u).

de operações subsequentes. Esta operação serve de base para as operações principais.

Toda árvore real contém um vértice, x, denominado *vértice exposto*. Este vértice é único e se caracteriza pelo fato do caminho de x até a raiz possuir apenas arestas sólidas e não haver nenhuma aresta *sólida* conectando outro vértice a x no sentido de percorrimento do caminho (isto é, de baixo para cima). Este é o caso, por exemplo, do vértice q da decomposição de caminhos da árvore da Figura 5.5, que é, como esperado, o único vértice exposto.

Para tornar u o vértice exposto, as arestas $\{s, m\}$ e $\{d, c\}$ teriam de se tornar sólidas, enquanto as arestas $\{v, u\}$, $\{r, m\}$ e $\{f, c\}$ teriam de ser tornar tracejadas. Isto é justamente o que ocorre quando se executa VirtualSplay(u) na árvore virtual da Figura 5.3, como pode ser visto na árvore virtual resultante da chamada na Figura 5.11. Na árvore real correspondente àquela (ou seja, a árvore da Figura 5.5), o vértice q é o vértice exposto, mas na árvore real correspondente à árvore virtual da Figura 5.11, o vértice u é o vértice exposto. A conversão de arestas tracejadas em arestas sólidas (e vice-versa), na árvore virtual, é realizada pelas chamadas à operação Splice(), como pode ser visto no Algoritmo 5.3.

Algoritmo 5.3 VirtualSplay(x)

Entrada: Um nó x em uma árvore virtual V
Saída: Nenhuma

```
v ← x                                  # primeira passada
enquanto Pai(v) ≠ nil faça
  p ← Pai(v)
  se IsSolidRoot(v) então
    v ← p
  senão
    SolidSplay(v)
  fim se
fim enquanto
v ← x                                  # segunda passada
enquanto Pai(v) ≠ nil faça
  Splice(v)
  v ← Pai(v)
fim enquanto
SolidSplay(x)                          # terceira passada
```

5.4 Operações sobre árvores dinâmicas

A chamada CUT(v) inicia com VIRTUALSPLAY(v), o que faz com que v se torne a raiz da árvore virtual. Em seguida, a ligação entre v e seu filho direito, w, é rompida, atribuindo-se **nil** a DIREITO(v) e PAI(w) (veja o pseudocódigo no Algoritmo 5.4). Observe que o filho direito, w, de v deve existir; caso contrário, v seria a raiz de uma árvore real, o que violaria a pré-condição de CUT(). A Figura 5.12 exemplifica o efeito da chamada CUT(m) na árvore real da Figura 5.4, enquanto a Figura 5.13 mostra a árvore virtual imediatamente após a chamada a VIRTUALSPLAY(m) e antes da conexão entre m e c ser rompida.

Algoritmo 5.4 CUT(v)

Entrada: Um nó, v, que não pode ser a raiz de uma árvore real T
Saída: Nenhuma
VIRTUALSPLAY(v)
$w \leftarrow$ DIREITO(v)
PAI(w) $\leftarrow$ **nil**
DIREITO(v) $\leftarrow$ **nil**

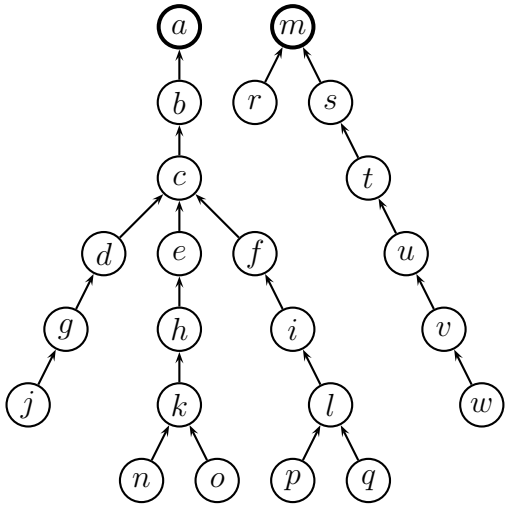


Figura 5.12: O efeito de CUT(m) na árvore da Figura 5.4.

A chamada EVERT(v), assim como CUT(v), inicia com a execução de VIRTUALSPLAY(v). Quando v se torna a raiz da árvore virtual, todos os nós na sua subárvore direita correspondem a nós acima de v na árvore *real*. Logo, a subárvore direita de v deve conter pelo menos um elemento (caso contrário, v seria a raiz de uma árvore real, o que

violaria a pré-condição desta operação). Seja w o filho direito de v. Para tornar v a raiz da árvore real, basta inverter a direção das arestas entre v e a raiz atual da árvore real, com a execução de SwitchBit(w), e redefinir w como um filho do meio de v, atribuindo **nil** a Direito(v). Este último passo modifica a decomposição em caminhos da árvore real e altera a ordem simétrica da árvore virtual, colocando todos os nós na subárvore (virtual) enraizada em w "à esquerda" do nó v, o que equivale a colocá-los abaixo de v na árvore real.

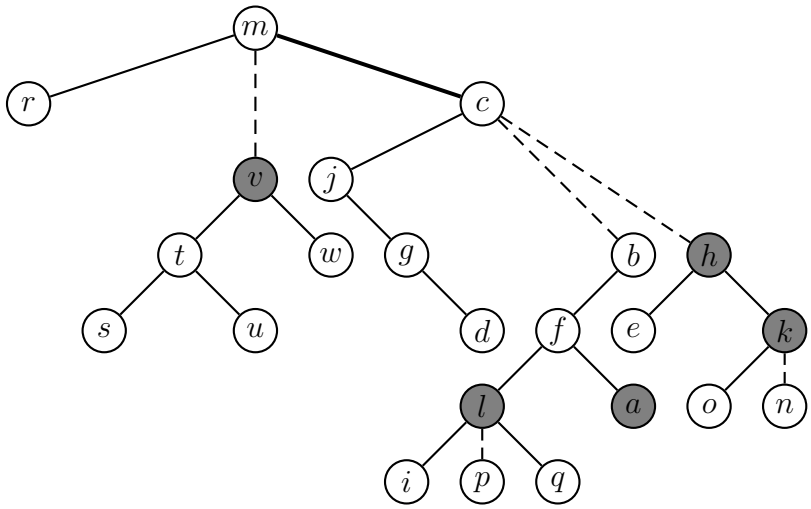


Figura 5.13: Árvore da Figura 5.6 após a chamada VirtualSplay(m) por Cut(m). A aresta exibida em destaque será removida logo em seguida, originado duas árvores disjuntas.

O Algoritmo 5.5 contém o pseudocódigo da operação Evert().

Algoritmo 5.5 Evert(v)

Entrada: Um nó, v, que não pode ser a raiz de uma árvore real T
Saída: Nenhuma
VirtualSplay(v)
$w \leftarrow$ Direito(v)
Direito(v) $\leftarrow$ **nil**
SwitchBit(w)

A Figura 5.14 exemplifica o efeito da chamada Evert(w) na árvore real da direita da Figura 5.12. A Figura 5.15 mostra o que ocorre na árvore virtual durante a execução de Evert(w).

A chamada Link(v, w) inicia sua execução com as chamadas VirtualSplay(v) e

VirtualSplay(w), de forma que os nós v e w tornam-se raízes de suas respectivas árvores virtuais. Em seguida, ela faz com que v se torne um filho do meio de w definindo Pai(v) como w.

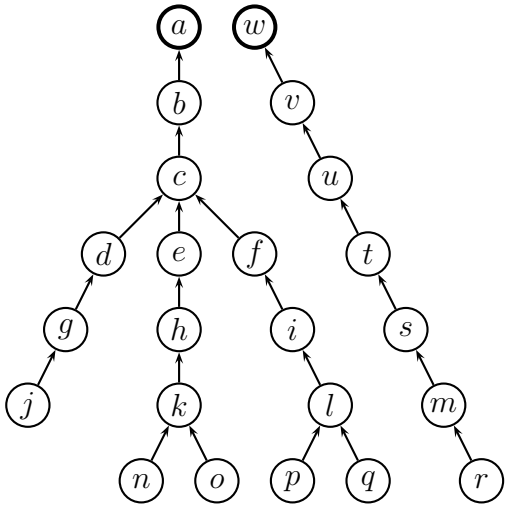


Figura 5.14: O efeito de Evert(w) na árvore da Figura 5.12.

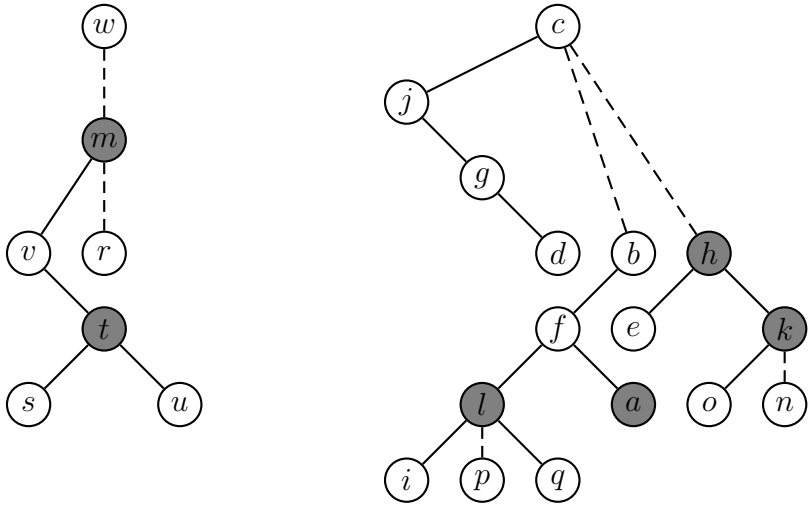


Figura 5.15: Árvore da Figura 5.6 após Cut(m), VirtualSplay(w) e SwitchBit(m).

O pseudocódigo da operação Link() está no Algoritmo 5.6.

A Figura 5.16 exemplifica o efeito da chamada Link(w, a) sobre as duas árvores reais da floresta exibida na Figura 5.14. A Figura 5.17 mostra a árvore virtual resultante da operação.

Algoritmo 5.6 LINK(v, w)

Entrada: Um nó, v, que deve ser a raiz de uma árvore real T
Entrada: Um nó, w, que esteja em uma árvore diferente da de v
Saída: Nenhuma
VIRTUALSPLAY(v)
VIRTUALSPLAY(w)
PAI$(v) \leftarrow w$

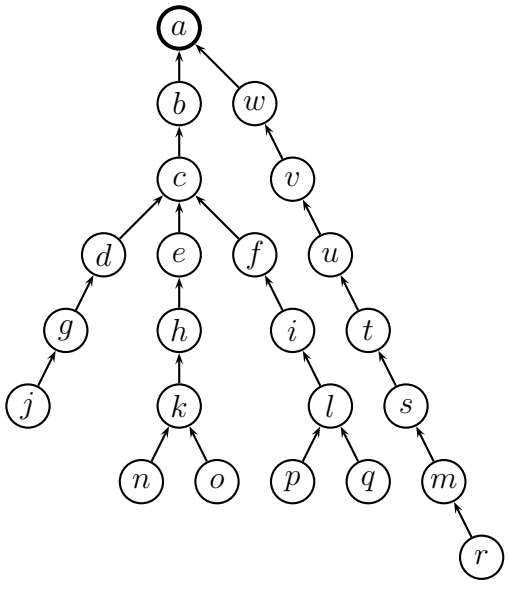


Figura 5.16: O efeito de LINK(w, a) na árvore da Figura 5.14.

As operações LINK(), CUT() e EVERT(), juntamente com MAKETREE(), são as únicas operações[2] sobre árvores dinâmicas que modificam a floresta. Note que MAKETREE() não foi descrita antes, pois envolve simplesmente a criação e inicialização dos campos de um nó. As demais operações, FINDROOT() e FINDPARENT(), são operações de consulta. Essas duas operações não modificam a árvore real, mas também modificam a estrutura da árvore virtual.

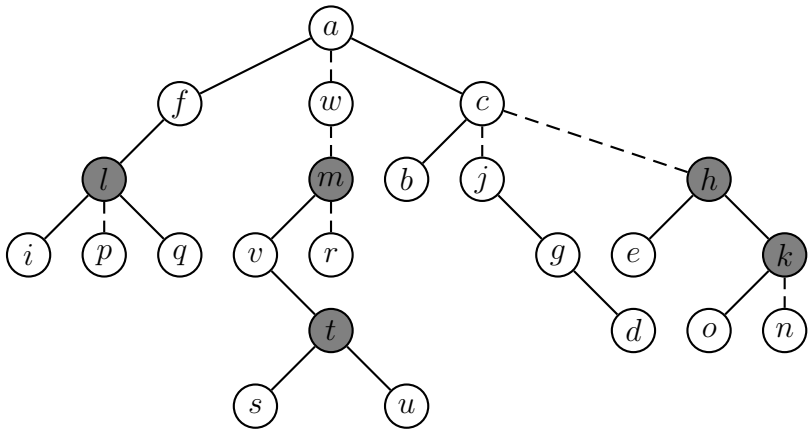


Figura 5.17: Árvore virtual correspondente à arvore da Figura 5.16.

A chamada FINDROOT(v) inicia com a execução de VIRTUALSPLAY(v) e, em seguida, procura pelo último nó, r, da sequência de nós resultante de um percurso em ordem simétrica na subárvore sólida que contém v. Neste percurso, a verdadeira ordem simétrica dos nós da árvore sólida é considerada. O nó r é a raiz da árvore real. A operação é finalizada com a execução de VIRTUALSPLAY(r), o que faz com que r se torne a raiz da árvore virtual.

O pseudocódigo da operação FINDROOT() é apresentado no Algoritmo 5.7.

A implementação de FINDPARENT() é bastante semelhante à de FINDROOT(). A principal diferença reside no fato de que o nó procurado, r, não é o último, mas sim o sucessor de v em um percurso em ordem simétrica (ou seja, o nó mais à esquerda na subárvore direita de v). Note que se o nó v for a raiz de sua árvore real, então ele não possui sucessor, e a função retorna um apontador para o endereço nulo, **nil** (veja o Algoritmo 5.8).

[2]Pelo menos entre as operações consideradas neste trabalho.

Algoritmo 5.7 FindRoot(v)

Entrada: Um nó, v, pertencente a uma árvore real T
Saída: Um apontador para a raiz de T, o nó r

```
VirtualSplay(v)
r ← v
enquanto Direito(r) ≠ nil faça
  r ← Direito(r)
  Unreverse(r)
fim enquanto
VirtualSplay(r)
return r
```

Algoritmo 5.8 FindParent(v)

Entrada: Um nó, v, pertencente a uma árvore real T
Saída: Um apontador para o pai de v em de T, o nó r

```
VirtualSplay(v)
r ← v
se Direito(r) ≠ nil então
  r ← Direito(r)
  Unreverse(r)
  enquanto Esquerdo(r) ≠ nil faça
    r ← Esquerdo(r)
    Unreverse(r)
  fim enquanto
  VirtualSplay(r)
  return r
senão
  return nil
fim se
```

5.5 Complexidade amortizada

Esta seção fornece uma análise da complexidade (amortizada) da operação VirtualSplay(). De maneira semelhante à Seção 4.4, calcula-se uma cota superior para o tempo amortizado dessa operação. Em seguida, conclui-se que esta cota superior é a mesma para as operações Link(), Cut(), Evert(), FindRoot() e FindParent() e, portanto, pode-se mostrar que o tempo total de execução de qualquer sequência de m dessas operações (juntamente com MakeTree()) em uma floresta inicialmente vazia, é $\mathcal{O}(m+\sum_{i=1}^{m} \lg n_i)$, onde n_i é o número de itens na árvore ou árvores envolvidas na i-ésima operação. Logo, o tempo médio de cada operação da sequência também é $\mathcal{O}(\lg n_{\max})$, onde $n_{\max} = \max_{i=1}^{m}\{n_i\}$.

A análise feita aqui usa uma extensão do argumento exposto na Seção 4.4, então os conceitos de *configuração* de uma árvore V, *peso*, *tamanho* e *posto* de um nó x, pertencente a V, são definidos exatamente da mesma forma que numa árvore splay. Note que a definição de tamanho dada anteriormente não distingue entre filhos esquerdos, direitos e do meio. Logo, não é necessário alterá-la. Já a função potencial, denotada aqui por $\Psi_V : \mathcal{V} \to \mathbb{R}$, onde $\mathcal{V}$ é o espaço das configurações possíveis de V, é definida pela seguinte expressão:

$$\Psi_V(A) = 3 \cdot \sum_{x \in A} r(x)\,, \tag{5.2}$$

onde A é uma dada configuração da árvore V. Isto é, o potencial, $\Psi_V(A)$, da configuração, A, da árvore V é igual a três vezes a soma dos postos de todos os seus nós. Por exemplo, considere a configuração de árvore na Figura 5.18, onde os tamanhos estão assinalados junto a cada nó (assumindo que o valor de $w(\alpha)$ é igual a 1 para todas as chaves) e o potencial é:

$$\begin{aligned}
\Psi_V(A) &= 3 \cdot \sum_{x \in A} r(x) \\
&= 3 \cdot (r(a) + r(b) + r(c) + \cdots + r(u) + r(v) + r(w)) \\
&= 3 \cdot (\lg s(a) + \lg s(b) + \lg s(c) + \cdots + \lg s(u) + \lg s(v) + \lg s(w)) \\
&= 3 \cdot (\lg (s(a) \cdot s(b) \cdot s(c) \cdots s(u) \cdot s(v) \cdot s(w))) \\
&= 3 \cdot (\lg(1 \cdot 18 \cdot 16 \cdot 1 \cdot 1 \cdot 23 \cdot 2 \cdot 5 \cdot 1 \cdot 10 \cdot 3 \cdot 4 \cdot 6 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 7 \cdot 1 \cdot 3 \cdot 1 \cdot 5 \cdot 1)) \\
&= 3 \cdot (\lg(18 \cdot 16 \cdot 23 \cdot 2 \cdot 5 \cdot 10 \cdot 3 \cdot 4 \cdot 6 \cdot 7 \cdot 3 \cdot 5)) \\
&= 3 \cdot \lg 5.007.744.000 \\
&\approx 96{,}66
\end{aligned}$$

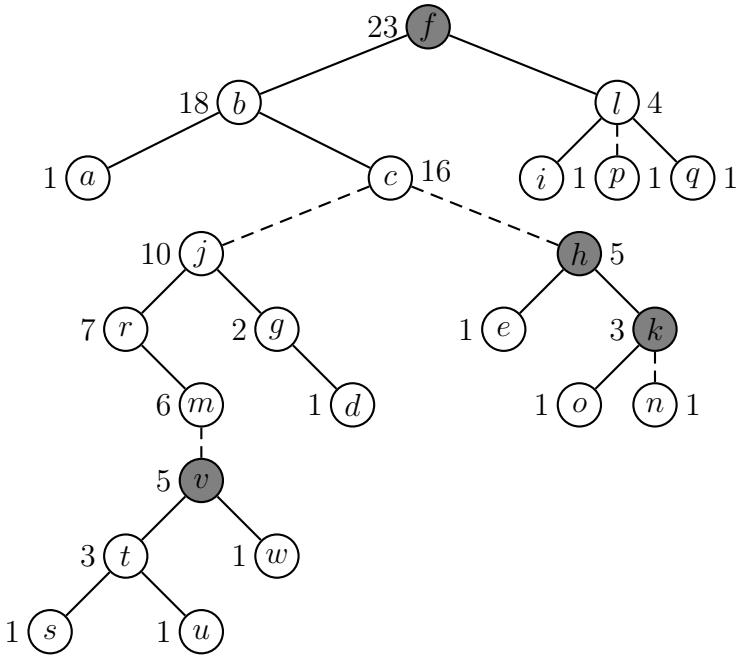


Figura 5.18: Uma configuração com potencial 96,66.

Por definição, tem-se que $\Psi_V(A) = 3{\cdot}\Phi_V(A)$, onde $\Phi_V(A)$ é o potencial da configuração A da árvore V com respeito à função potencial Φ_V definida no Capítulo 4. Observe, no entanto, que Ψ_V será aplicada a uma árvore virtual, V, por inteiro, enquanto Φ_V foi aplicada, no Capítulo 4, a uma única árvore splay. O lema a seguir estabelece uma cota superior para $\hat{t} = t + \Psi_V(B) - \Psi_V(A)$, onde t é o tempo de execução da operação VirtualSplay(), A é a configuração da árvore virtual, V, antes da operação e B é a configuração de V após a operação. A cota superior é dada em função do número, n, de nós de V.

Lema 5.5.1. O tempo amortizado de uma única operação *virtual splaying* sobre um nó x de uma árvore virtual está em $\mathcal{O}(\lg n)$, onde n é o número total de nós da árvore virtual.

Demonstração. Seja V a árvore virtual contendo x. Como visto no Algoritmo 5.3, VirtualSplay() consiste de três passadas distintas subindo por V, do nó x até o nó raiz, q, de V. A primeira passada consiste em realizar a operação SolidSplay() em cada árvore sólida encontrada no caminho até q; a segunda, em aplicar Splice() a x e todos os seus ancestrais (com exceção de q); e a terceira, em chamar SolidSplay(x). Denote por $\hat{t}_j$ e t_j os tempos amortizado e real, respectivamente, da j-ésima passada, para $j = 1, 2, 3$. Logo, o tempo amortizado, $\hat{t}$, de uma única operação de *splaying* em V é igual a $\sum_{j=1}^{3} \hat{t}_j$.

Por definição,

$$\hat{t}_j = t_j + \Psi_V(B_j) - \Psi_V(A_j)\,,$$

onde A_j é a configuração da árvore virtual, V, antes da j-ésima passada e B_j é a configuração de V após a j-ésima passada. Note que $A = A_1$ e $B = B_3$. Agora, considere cada passada:

1) Seja $t = \textsc{RaizSólida}(x)$ a raiz da subárvore sólida, T, de V contendo o nó x. Esta subárvore é uma árvore splay. Lembre-se de que o tempo amortizado de $\textsc{Splay}(x)$ é $3 \cdot (r(t) - r(x)) + 1$ quando a função potencial, Φ_T, é aplicada (veja o Lema 4.4.1). Como $\Psi_T(\cdot) = 3 \cdot \Phi_T(\cdot)$, tem-se que cada chamada a $\textsc{SolidSplay}(v)$ na primeira passada da operação $\textsc{VirtualSplay}()$ tem tempo amortizado limitado superiormente por $9 \cdot (r(\textsc{RaizSólida}(v)) - r(v)) + 1$, onde o $+1$ corresponde ao tempo do último *Zig*, se algum, e não precisa ser multiplicado por 3. Então, seja k o número de arestas tracejadas que separam x da raiz, q, da árvore virtual V. Observe que k é exatamente igual ao número de vezes que $\textsc{SolidSplay}()$ é executada na primeira passada de $\textsc{VirtualSplay}()$. Mais especificamente, $\textsc{SolidSplay}()$ é executada nos nós

$$x, \textsc{Pai}_V(\textsc{RaizSólida}(x)), \ldots, (\textsc{Pai}_V \circ \textsc{RaizSólida})^{k-1}(x)\,.$$

Logo,

$$\begin{aligned}\hat{t}_1 &\leq \sum_{l=0}^{k-1} 9 \cdot \big(r(\mathrm{RS}((\mathrm{P}_V \circ \mathrm{RS})^l(x))) - (r((\mathrm{P}_V \circ \mathrm{RS})^l(x))\big) + 1 \\ &= 9 \cdot (r(q) - r(x)) + k\,,\end{aligned}$$

onde $\textsc{RaizSólida}$ e $\textsc{Pai}_V$ foram abreviadas para RS e P_V, respectivamente, para tornar a expressão mais clara. Assim, tem-se que $\hat{t}_1$ é limitado por $\mathcal{O}(\lg n) + k$, pois $s(q) = n$ e, no pior caso, $s(x) = 1$. Como $r(v) = \lg s(v)$, para todo nó v em V, o resultado segue.

2) Na segunda passada, $\textsc{Splice}()$ é aplicada a x e a todos os seus ancestrais, com exceção de q. Nenhuma rotação é realizada e o potencial da árvore permanece inalterado, pois todos os nós continuam com os mesmos pais. Logo, $\Psi_V(A_2) = \Psi_V(B_2)$ e, portanto, $\hat{t}_2 = t_2 + \Psi_V(B_2) - \Psi_V(A_2) = t_2$. Como x possui k ancestrais e o tempo (real) de execução de cada $\textsc{Splice}()$ é igual a uma unidade, tem-se que $\hat{t}_2 = t_2 = k$.

3) Na terceira passada, a operação $\textsc{SolidSplay}()$ é executada para o nó x, que está a uma distância de k nós da raiz, s, atual de V. Logo, o tempo (real), t_3, de execução

da terceira passada é k. Então, $\hat{t}_3 = t_3 + \Psi_V(B_3) - \Psi_V(A_3) = k + \Psi_V(B_3) - \Psi_V(A_3)$. Note que

$$\Psi_V(B_3) - \Psi_V(A_3) = 3 \cdot (\Phi_V(B_3) - \Phi_V(A_3)) .$$

Além disso, como as rotações são restritas a uma única subárvore sólida de V, o Lema 4.4.1 pode ser invocado para concluir que $k + \Phi_V(B_3) - \Phi_V(A_3) \leq 3 \cdot (r(s) - r(x)) + 1$, que está em $\mathcal{O}(\lg n)$, onde n é o número de nós de V. Multiplicando a expressão

$$k + \Phi_V(B_3) - \Phi_V(A_3)$$

por 3, obtém-se

$$3 \cdot k + 3(\Phi_V(B_3) - \Phi_V(A_3)) = 3 \cdot k + \Psi_V(B_3) - \Psi_V(A_3) ,$$

que está em $\mathcal{O}(\lg n)$, o que implica que $\hat{t}_3 = k + \Psi_V(B_3) - \Psi_V(A_3)$ também está em $\mathcal{O}(\lg n)$.

A dedução realizada para a terceira passada também implica que $3 \cdot k + \Psi_V(B_3) - \Psi_V(A_3) + \mathcal{O}(\lg n)$ está em $\mathcal{O}(\lg n)$, pois $\mathcal{O}(\lg n) + \mathcal{O}(\lg n)$ está em $\mathcal{O}(\lg n)$. Agora, observe que $\hat{t} = \hat{t}_1 + \hat{t}_2 + \hat{t}_3$ é exatamente igual a $(k + \mathcal{O}(\lg n)) + k + (k + \Psi_V(B_3) - \Psi_V(A_3))$. Daí, tem-se

$$\hat{t} \in \mathcal{O}(\lg n) .$$

□

A cota superior fornecida pelo Lema 5.5.1 para o tempo amortizado de uma operação *splaying* numa árvore virtual é a mesma para o tempo amortizado de cada chamada a Link(), Cut(), Evert(), FindRoot() ou FindParent(). Esta conclusão é obtida imediatamente a partir do pseudocódigo dessas operações. Logo, pode-se afirmar que o tempo amortizado de qualquer operação em uma sequência de m operações sobre uma floresta de árvores ST inicialmente vazia, onde cada operação é uma das mencionadas ou MakeTree() (que executa em tempo real constante), está em $\mathcal{O}(\lg n)$, onde n é o número de nós distintos incluídos na floresta durante toda a execução da sequência de operações.

6 Conectividade Dinâmica

Este capítulo descreve uma solução desenvolvida por Holm, Lichtenberg e Thorup para o *problema da conectividade em grafos dinâmicos* (HOLM; LICHTENBERG; THORUP, 2001). Neste problema, um dado grafo, G, com um número, $v(G)$, fixo de vértices é submetido a uma sequência de m operações, tal que cada operação insere ou remove uma aresta de G ou determina se dois vértices de G estão conectados (por um caminho) em G. As m operações são apresentadas e executadas em ordem sequencial e cada uma delas é executada sem nenhum conhecimento prévio de quais são as operações que a sucedem na sequência. A solução aqui descrita é, na verdade, um tipo abstrato de dados (TAD) que pode ser implementado com árvores dinâmicas; em particular, a árvore ST estudada no Capítulo 5. Esta solução é crucial para uma implementação eficiente do algoritmo de Diks e Stanczyk para calcular emparelhamentos perfeitos em grafos cúbicos e sem pontes. Mais especificamente, a solução é usada para determinar se o grafo resultante da remoção de cinco arestas e dois vértices de um dado grafo conexo é ainda um grafo conexo (veja a Seção 3.3).

A Seção 6.1 faz uma descrição mais detalhada do problema da conectividade em grafos dinâmicos e inclui referências para as principais soluções conhecidas e suas respectivas complexidades. A Seção 6.2 discute o TAD HLT de forma independente da estrutura de dados usada para representar suas informações e implementar suas operações. Uma análise da complexidade amortizada das operações do TAD HLT também é apresentada. A Seção 6.3 descreve, em alto-nível, a implementação do TAD HLT feita neste trabalho e discute as limitações desta implementação com relação à complexidade amortizada de tempo.

6.1 O problema da conectividade dinâmica

Em (FREDERICKSON, 1985), Fredrickson introduziu uma estrutura de dados, conhecida como *árvore topológica* (do inglês, *topology tree*), que pode ser usada para resolver o

problema da conectividade dinâmica com tempos de pior caso em $\Theta(\sqrt{e(G)})$ e em $\Theta(1)$, respectivamente, por atualização (isto é, inserção ou remoção de arestas) e por consulta (isto é, determinar se dois vértices dados estão conectados), onde $e(G)$ é o número de arestas do grafo, G. O tempo de pior caso para atualizações pode ser reduzido para $\Theta(\sqrt{v(G)})$ usando a técnica de *esparsificação* em (EPPSTEIN et al., 1997). Obviamente, esta redução em complexidade só faz sentido quando G é um grafo denso (isto é, $v(G) \in o(e(G))$). Até onde se saiba, uma solução com tempo de "pior caso" em $\mathcal{O}(\lg^k v(G))$ para atualizações e consultas a G, onde k é um inteiro positivo, permanece um problema em aberto.

Cotas superiores mais justas podem ser obtidas se a complexidade amortizada for levada em consideração. Para tal, reduz-se o problema da conectividade dinâmica àquele de manter uma uma floresta, F, geradora do grafo, G, dinâmico (isto é, uma floresta com uma árvore geradora para cada componente conexa do grafo). A idéia é utilizar F para responder às consultas sobre conectividade em G. Em particular, dois vértices, u e v, de G estão conectados em G se, e somente se, eles fazem parte de uma mesma árvore geradora em F. Sabe-se, por exemplo, que se a floresta F for representada por árvores ST, a consulta se resume a comparar os nós devolvidos por FINDROOT(u) e FINDROOT(v), o que pode ser feito em tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós de F, que é igual a $v(G)$.

Para que a redução de problemas acima "funcione", a floresta F deve ser atualizada sempre que G o for. Em particular, quando uma aresta e é inserida em G, ela é inserida em F se, e somente se, ela conecta duas árvores (disjuntas) de F ou, de forma equivalente, se, e somente se, ela conecta duas componentes conexas de G. Por sua vez, se uma aresta e é removida de G, então dois casos devem ser considerados. Se e não é uma aresta de F, então nada precisa ser feito em F, pois a remoção de e de G não desconecta nenhuma componente conexa de G. No entanto, se e é uma aresta de F, então a remoção de e de F desconectará uma árvore geradora de F (independentemente desta remoção desconectar uma componente de G ou não). Neste caso, uma aresta, f, de G deve ser procurada para substituir e em F e, em seguida, inserida em F. Obviamente, tal aresta f só poderá existir se a remoção de e de G não desconectar uma componente conexa de G. Como será visto mais adiante, cada inserção ou remoção de aresta pode ser realizada em tempo amortizado $\mathcal{O}(\lg^2 n)$.

Em (HENZINGER; KING, 1999), Henzinger e King fizeram uso de aleatorização para manter uma floresta geradora com tempos amortizados *esperados* $\mathcal{O}(\lg^3 n)$ e $\mathcal{O}\left(\frac{\lg n}{\lg \lg n}\right)$

por atualização e consulta, respectivamente. Em seguida, Henzinger e Thorup reduziram, em (HENZINGER; THORUP, 1997), o tempo amortizado esperado de uma atualização para $\mathcal{O}(\lg^2 n)$. Holm, Lichtenberg e Thorup eliminaram, em (HOLM; LICHTENBERG; THORUP, 2001), o processo de aleatorização proposto em (HENZINGER; KING, 1999; HENZINGER; THORUP, 1997) e deram uma solução determinística para manter uma floresta dinâmica com tempos amortizados $\mathcal{O}(\lg^2 n)$ e $\mathcal{O}\left(\frac{\lg n}{\lg \lg n}\right)$ por atualização e consulta, respectivamente. Recentemente, o tempo amortizado por atualização foi reduzido para $\mathcal{O}\left(\frac{\lg^2 n}{\lg \lg n}\right)$ (WULFF-NILSEN, 2013). Esta solução se baseou na solução dada por Thorup em (THORUP, 2000), que também reduz a complexidade das operações de atualização, mas usando aleatorização.

De forma geral, se for possível manter a floresta F em tempo amortizado $\mathcal{O}(t(n) \cdot \lg n)$ por inserção ou remoção de arestas, então — usando árvores dinâmicas — é possível responder consultas sobre conectividade em tempo amortizado $\mathcal{O}\left(\frac{\lg n}{\lg t(n)}\right)$ por consulta (THORUP, 2000; HOLM; LICHTENBERG; THORUP, 2001), onde n é o número (fixo) de nós de F. Por outro lado, Pătraşcu e Demaine mostraram em (PĂTRAŞCU; DEMAINE, 2004) que qualquer solução para o problema da conectividade possui como cota inferior tempo amortizado $\Omega(\lg n)$ por operação. Além disso, eles mostraram uma interdependência entre os tempos amortizados de atualização e consulta. Quando se reduz a cota superior de um, aumenta-se a cota inferior de outro. Mais especificamente, sejam $t_u(n)$ e $t_q(n)$ os tempos amortizados de uma atualização e de uma consulta, respectivamente. Então, tem-se

$$t_q(n) \cdot \lg\left(\frac{t_u(n)}{t_q(n)}\right) \in \Omega(\lg n) \qquad \text{e} \qquad t_u(n) \cdot \lg\left(\frac{t_q(n)}{t_u(n)}\right) \in \Omega(\lg n)\,.$$

É importante ressaltar que as cotas superiores para $t_u(n)$ e $t_q(n)$ das soluções em (HOLM; LICHTENBERG; THORUP, 2001; WULFF-NILSEN, 2013) não ultrapassam as cotas inferiores das relações acima. Logo, deste ponto de vista, pode-se dizer que ambas as soluções são "ótimas".

Neste trabalho, adotou-se, para o problema da conectividade dinâmica, a solução apresentada em (HOLM; LICHTENBERG; THORUP, 2001), pois a solução descrita em (WULFF-NILSEN, 2013) só se tornou conhecida no período de escrita deste texto, após a implementação da solução adotada e a geração de resultados terem sido finalizadas. No entanto, tanto uma solução quanto a outra fazem com que a implementação do algoritmo de Diks e Stanczyk apresente a mesma complexidade amortizada. A diferença mais significativa entre as duas soluções está na complexidade de espaço. Enquanto a solução dada por Holm, Lichtenberg e Thorup requer $\mathcal{O}(h + n \lg n)$ endereços de memória, onde h é o número,

$e(G)$, de arestas do grafo G, a solução dada por Wulff-Nilsen requer espaço linear em n.

6.2 O TAD de Holm, Lichtenberg e Thorup

Esta seção descreve a solução apresentada por Holm, Lichtenberg e Thorup para o problema da conectividade em grafos dinâmicos (HOLM; LICHTENBERG; THORUP, 2001). Esta solução é, de fato, um tipo abstrato de dados (TAD), denotado pela sigla HLT — *as iniciais dos sobrenomes*, que representa um grafo G dinâmico e que suporta as seguintes operações:

- CONNECTED(u, v): determina se os vértices u e v estão conectados em G,
- INSERT(e): insere a aresta e em G e
- REMOVE(e): remove a aresta e de G.

Como dito na Seção 6.1, cada uma dessas operações pode ser realizada em tempo amortizado $\mathcal{O}(\lg^2 v(G))$. No entanto, para que isso seja possível, deve-se manter uma floresta geradora, F, do grafo G que, por sua vez, deve ser representada por árvores dinâmicas. Em princípio, qualquer árvore dinâmica que possa ser modificada ou consultada em tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore, pode ser utilizada. No entanto, a solução requer certas operações especiais sobre árvores dinâmicas, algumas das quais são mais natural e eficientemente implementadas em alguns tipos de árvore do que em outros.

Usando a floresta geradora, F, representada por árvores dinâmicas, pode-se facilmente executar CONNECTED(u, v). De fato, esta operação se reduz a duas chamadas a FINDROOT(): uma para o vértice u e outra para o vértice v. Se ambas devolverem a mesma raiz em F, os vértices u e v estão conectados em G, pois eles pertencem a uma mesma árvore geradora (resp. componente conexa) de F (resp. G); caso contrário, eles não estão conectados, pois eles pertencem a árvores geradoras (resp. componentes conexas) distintas em F (resp. G).

A execução de INSERT(e) também é simples. Primeiro, insere-se $e = \{u, v\}$ em G. Depois, verifica-se, usando CONNECTED(), se u e v estão em uma mesma árvore de F. Se estiverem, então a floresta não precisa ser modificada, pois a inclusão de e em F produziria um ciclo; caso contrário, insere-se e em F de forma que as árvores que contêm u e v são unidas. Esta união pode ser realizada com a chamada LINK(u, v), mas como u não é

necessariamente a raiz de sua árvore em F, a chamada LINK(u, v) deve ser precedida pela chamada EVERT(u), que faz com que u se torne a raiz de sua árvore em F (veja a Figura 6.1). Esta é a razão pela qual a operação EVERT() é imprescindível para este trabalho.

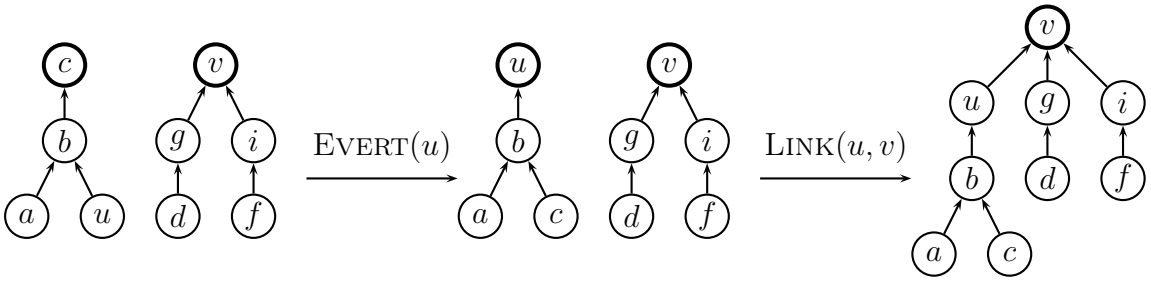


Figura 6.1: Execução de INSERT(e), com $e = \{u, v\}$. As raízes estão em destaque.

Note que a complexidade amortizada de cada execução de CONNECTED() ou INSERT() é claramente $\mathcal{O}(\lg v(G))$, pois cada uma dessas operações executa um número pequeno e constante de chamadas às operações FINDROOT(), LINK() e EVERT(). Obviamente, assume-se aqui que tais operações possuem tempo amortizado $\mathcal{O}(\lg \eta)$ cada, onde η é o número total de nós da(s) árvore(s) envolvidas na operação e $\eta \leq v(G)$. Este é, como já se sabe, o caso se a árvore dinâmica empregada for a árvore ST estudada no capítulo anterior.

A execução de REMOVE(e) é um pouco mais complicada. Há, na verdade, dois casos mutuamente exclusivos: (1) a aresta e não pertence a F e (2) a aresta e pertence a F. No primeiro caso, diz-se que e é uma *aresta de reserva*. No segundo caso, diz-se que e é uma *aresta de árvore*. Ambos os casos são precedidos, na execução de REMOVE(e), pela remoção de e de G. O primeiro caso é trivial, pois se e não está em F, então não há nada mais a fazer. O segundo caso, ao contrário, é mais complicado, pois a remoção de e de F faz com que uma árvore geradora, T, de F se divida em duas, T_u e T_v, mesmo que a componente conexa de G correspondente a T não seja. Se não for, uma aresta, f, de reserva em G deve ser encontrada para reconectar T_u e T_v. Caso contrário, tal aresta não existe.

Observe que a remoção da aresta $e = \{u, v\}$ da floresta F pode ser realizada facilmente com a chamada CUT(u) ou a chamada CUT(v), dependendo se v é o pai de u em F ou vice-versa. A Figura 6.2 ilustra a execução da chamada CUT(u). As arestas de reserva são representadas por arcos pontilhados. Observe que a componente, G', de G correspondente

à árvore T não foi desconectada. Há, no entanto, duas tarefas mais complicadas pela frente.

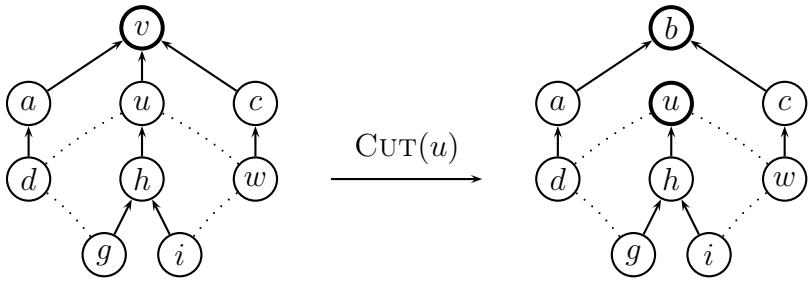


Figura 6.2: Execução de CUT(u). Arestas de reserva são exibidas com arcos pontilhados.

A primeira delas é determinar se G' também é desconectada com a remoção da aresta e do grafo G. A segunda está condicionada ao fato de G' não ter sido desconectada. Se não foi, tem-se a outra tarefa complicada de se encontrar uma aresta, f, de reserva para reconectar as árvores T_u e T_v. A principal contribuição de (HOLM; LICHTENBERG; THORUP, 2001) é uma estratégia eficiente para realizar as duas tarefas. Esta é a estratégia que permite que REMOVE() execute em tempo amortizado $\mathcal{O}(\lg^2 v(G))$, pois se uma busca "menos cuidadosa" por f for utilizada, o tempo amortizado de REMOVE() será $\Omega(v(G) \cdot \lg v(G))$.

6.2.1 Uma busca mais eficiente

A idéia central da estratégia em (HOLM; LICHTENBERG; THORUP, 2001) é manter uma hierarquia de $lmax + 1$ subgrafos do grafo G, denotados por $G_0, G_1, \ldots, G_{lmax}$, onde $lmax = \lceil \lg v(G) \rceil$. Para cada $i \in \{0, 1, \ldots, lmax\}$, o conjunto, V_i, de vértices do grafo G_i, que é o subgrafo de G no i-ésimo nível da hierarquia, é exatamente igual ao conjunto, V, de vértices do grafo G. A principal diferença entre os subgrafos da hierarquia reside nos conjuntos de arestas. Cada aresta, e, do conjunto, E, de arestas de G está associada a um valor, $l(e)$, denominado *nível* de e, tal que $l(e) \in \{0, 1, \ldots, lmax\}$. O conjunto, E_i, de arestas do subgrafo G_i consiste exatamente das arestas em E com nível menor ou igual a i:

$$E_i = \{e \in E \mid l(e) \leq i\}\,.$$

Logo,

$$E_0 \subseteq E_1 \subseteq \cdots E_{lmax} \quad \text{e} \quad G_{lmax} = G\,.$$

A Figura 6.3 contém uma hierarquia de subgrafos de um grafo G com 12 vértices e 21 arestas. A hierarquia possui $\lceil \lg 12 \rceil + 1 = 4 + 1 = 5$ níveis, cada qual com um subgrafo de G.

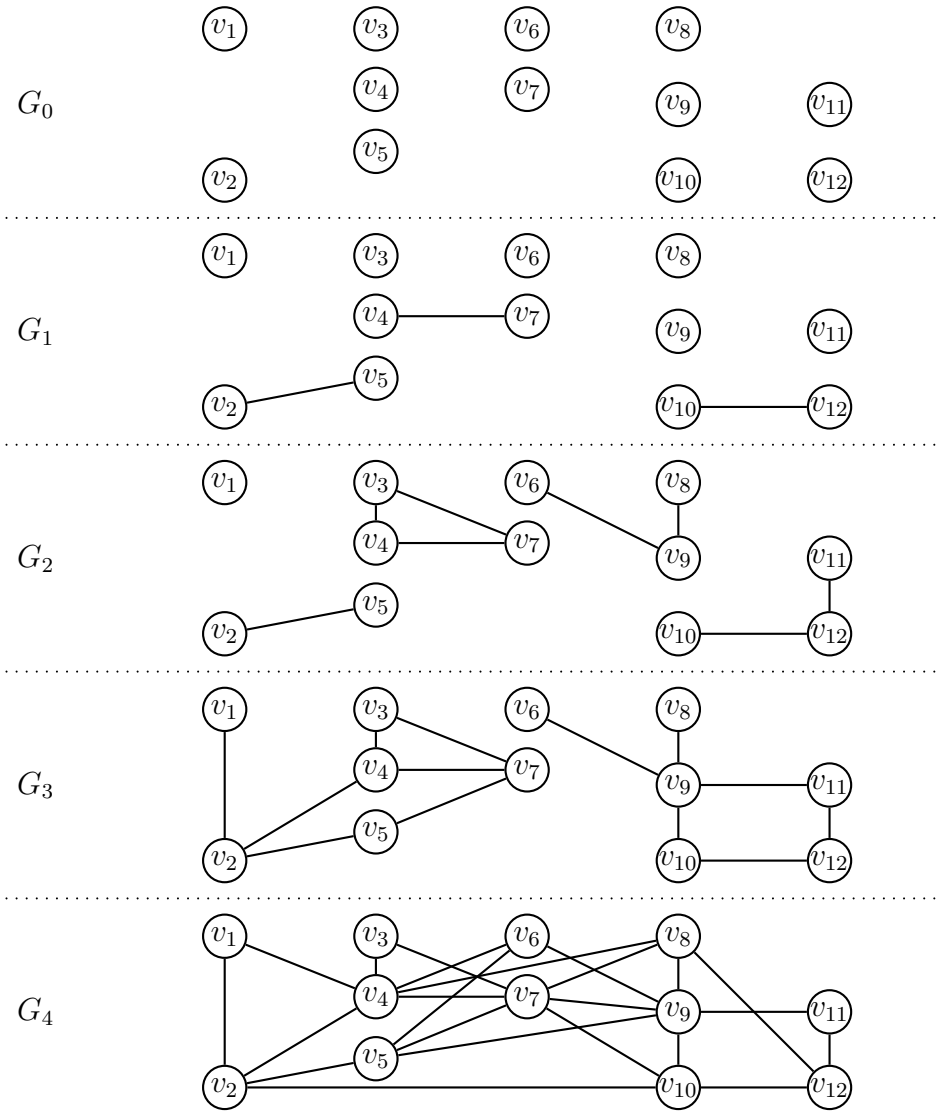


Figura 6.3: Uma hierarquia com 5 subgrafos de um grafo com 12 vértices e 21 arestas.

Lembre-se de que, no problema da conectividade dinâmica, assume-se que o conjunto, V, de vértices de G é fixo. Já o conjunto, E, de arestas de G é mutável, pois arestas podem ser inseridas e removidas de G. No contexto das aplicações que envolvem o problema da conectividade dinâmica, o primeiro passo consiste sempre em "construir" o grafo G, o que requer a inicialização das estruturas de dados. Neste contexto, supõe-se que o conjunto, E, de arestas do grafo é, inicialmente, o conjunto vazio. Durante a construção de G, as

arestas são inseridas, uma a uma, em E. Após ser construído, ele pode sofrer alterações através de remoções e inserções de novas arestas de E. Por convenção, *assume-se que o nível, $l(e)$, de toda aresta e que ainda não foi inserida em G é igual lmax*. Então, quando e é inserida em G, tem-se $l(e) = lmax$ e, portanto, a aresta faz parte apenas do subgrafo G_{lmax}.

Como será visto mais adiante, o nível $l(e)$ pode ser decrementado, mas jamais incrementado. Mais especificamente, o nível $l(e)$ pode ser decrementado em uma unidade por vez e apenas quando alguma aresta, que não é a aresta e, for removida de G. Observe que decrementar o valor de $l(e)$ em uma unidade equivale a inserir e no subgrafo G_i, com $i = l(e)$. Esta é a única forma pela qual arestas são inseridas nos subgrafos $G_0, G_1, \ldots, G_{lmax-1}$. Além disso, o valor $l(e)$ jamais se torna negativo. Isto significa que o nível de cada aresta só pode ser decrementado $lmax$ vezes, no máximo. Quando isto ocorrer a uma aresta em particular, a aresta terá sido inserida em todos os subgrafos da hierarquia.

Para construir e modificar a hierarquia com os subgrafos $G_0, G_1, \ldots, G_{lmax}$, a estratégia proposta em (HOLM; LICHTENBERG; THORUP, 2001) constrói e mantém uma hierarquia de florestas, $F_0, F_1, \ldots, F_{lmax}$, tal que F_i é uma floresta geradora do subgrafo G_i, para todo $i \in \{0, 1, \ldots, lmax\}$. Usando esta hierarquia de florestas, cada operação de consulta e atualização sobre G é realizada em tempo amortizado $\mathcal{O}(\lg^2 v(G))$. Para tal, duas invariantes devem ser mantidas antes e depois de cada inserção ou remoção de arestas de G:

(i) Toda componente conexa de G_i possui, no máximo, 2^i vértices.

(ii) Para todo $i \in \{0, 1, \ldots, lmax - 1\}$, tem-se que toda aresta de F_i é uma aresta de F_{i+1}. Em outras palavras, $F_i = F \cap G_i$ e F é uma floresta geradora *mínima* de G, onde $F = F_{lmax}$, com respeito ao nível das arestas (isto é, o custo de uma aresta e é $l(e)$).

A Figura 6.4 exibe uma hierarquia de 5 florestas geradoras, F_0, F_1, F_2, F_3 e F_4, dos subgrafos da hierarquia exibida na Figura 6.3. As hierarquias respeitam as invariantes (i) e (ii).

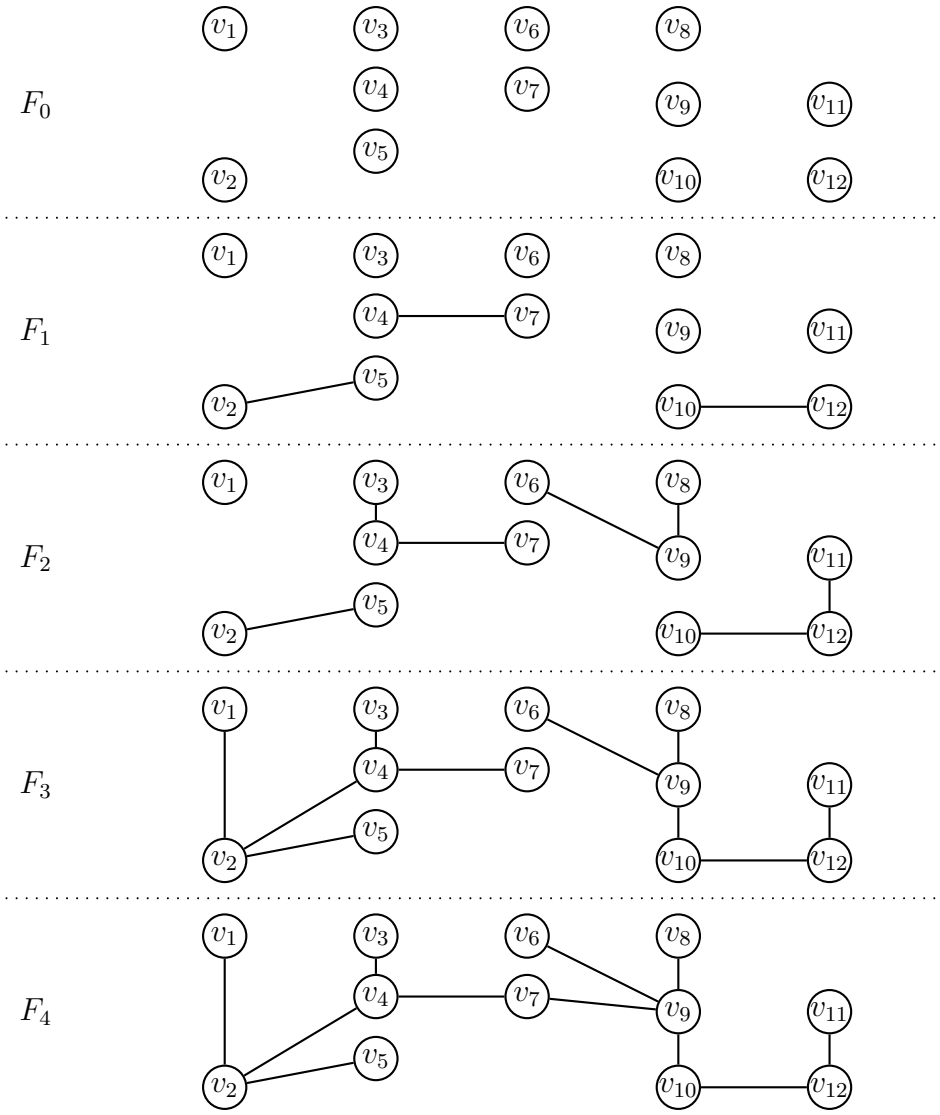


Figura 6.4: Uma hierarquia de florestas geradoras dos subgrafos da Figura 6.3.

A hierarquia de florestas é construída à medida que a hierarquia de subgrafos é construída. Na prática, apenas a hierarquia de florestas é, de fato, representada no computador, pois a estrutura de dados usada para representar o TAD HLT armazena as arestas de reserva nos próprios nós das florestas $F_0, F_1, \ldots, F_{lmax}$. Inicialmente, cada floresta F_i consiste apenas dos vértices do grafo G, assim como cada subgrafo G_i. Quando o grafo G é construído, as arestas inseridas em E são inseridas na estrutura de dados que representa F_{lmax} usando a nova versão da função INSERT() descrita mais adiante. Algumas dessas arestas se tornam arestas das árvores geradoras de F_{lmax}, enquanto as demais se tornam arestas de reserva e são armazenadas nos nós de F_{lmax}. Em particular, duas cópias da aresta, $e = \{u, v\}$, de reserva são armazenadas em F_{lmax}: uma no nó u e outra no nó v de

F_{lmax}.

A Figura 6.5 exibe a mesma hierarquia de florestas da Figura 6.4, mas com as arestas de reserva exibidas como arcos pontilhados. Tais arestas só aparecem em seus respectivos níveis.

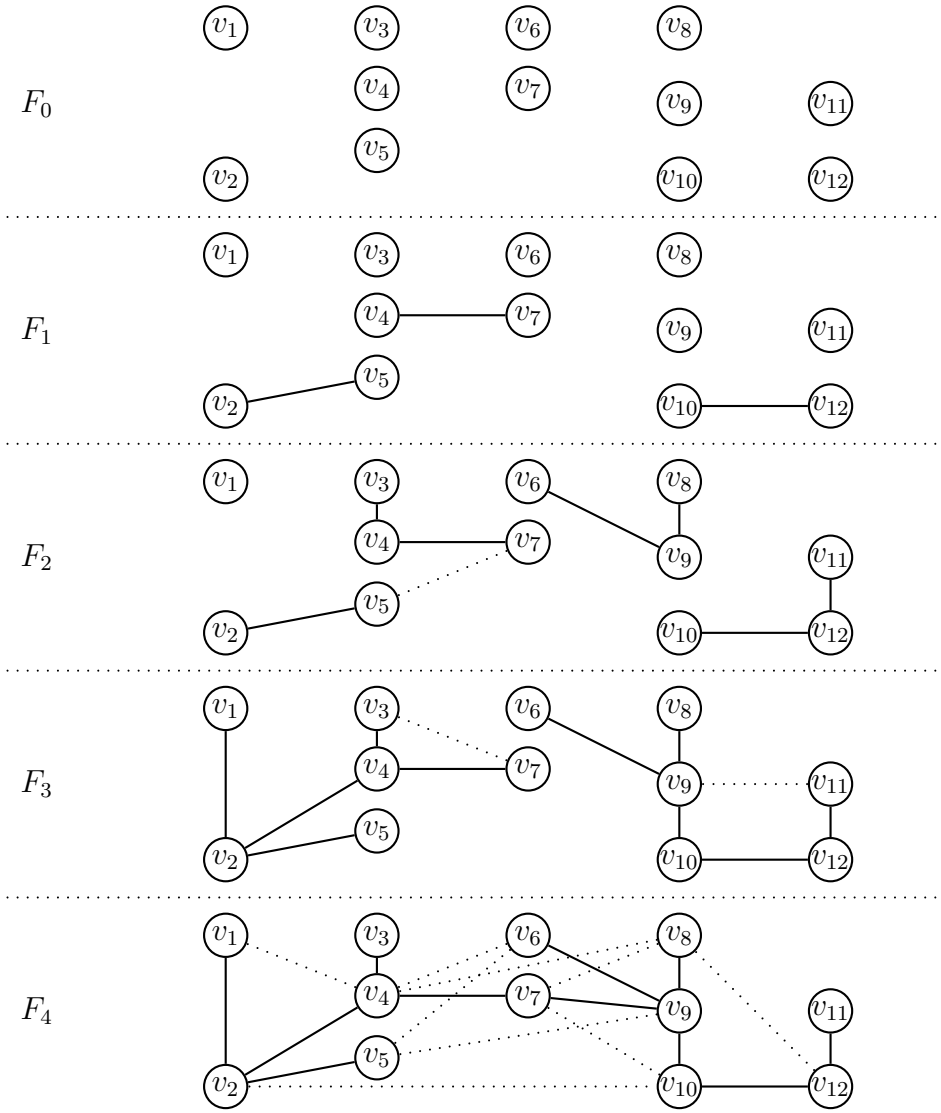


Figura 6.5: Uma hierarquia de florestas geradoras dos subgrafos da Figura 6.3. Arestas de árvore e de reserva são exibidas como arcos sólidos e pontilhados, respectivamente.

Após a construção de G (e imediatamente antes de G sofrer qualquer alteração), as componentes conexas de F_{lmax} são árvores geradoras mínimas das componentes conexas de $G = G_{lmax}$, pois todas as arestas de E possuem o mesmo custo. Além disso, os subgrafos, $G_0, G_1, \ldots, G_{lmax-1}$, e suas respectivas florestas geradoras, $F_0, F_1, \ldots, F_{lmax-1}$, não pos-

suem nenhuma aresta. Como cada componente conexa de $G_{lmax} = G$ possui, no máximo, $v(G)$ vértices e $v(G) \leq 2^{\lceil \lg v(G) \rceil}$, conclui-se que as invariantes (i) e (ii) são respeitadas imediatamente após a construção de G, ou seja, imediatamente após a inicialização da estrutura de dados. No que segue, discutem-se as operações de consulta e atualização do grafo G.

A execução de CONNECTED(u, v) é praticamente a mesma que se discutiu antes da estratégia proposta em (HOLM; LICHTENBERG; THORUP, 2001). Em particular, a consulta continua sendo realizada na floresta F, que é a floresta geradora mínima, F_{lmax}, do subgrafo $G_{lmax} = G$. Lembre-se de que a complexidade amortizada de uma execução da operação CONNECTED() é $\mathcal{O}(\lg n)$, onde n é o número de nós da floresta F, que é exatamente igual a $v(G)$.

A execução de INSERT(e) é ligeiramente modificada. Assim como antes, o primeiro passo consiste em inserir $e = \{u, v\}$ em G. Em seguida, atribui-se $lmax$ a $l(e)$. Depois, verifica-se, usando CONNECTED(), se u e v estão em uma mesma árvore geradora mínima de $F = F_{lmax}$. Se não estiverem, então insere-se e em F_{lmax} de forma que as árvores geradoras mínimas que contêm u e v são unidas. Como $l(e) = lmax$, a árvore resultante da união é uma árvore geradora mínima da respectiva componente de G_{lmax} contendo u e v. Observe que e se torna uma aresta de árvore. Por outro lado, se u e v estiverem em uma mesma árvore de F_{lmax}, então F_{lmax} não precisa ser modificada. Mas, uma cópia da aresta e é armazenada em cada um dos nós, u e v, de F_{lmax} e e se torna uma aresta de reserva. Note que o tempo amortizado de uma execução da operação INSERT() continua $\mathcal{O}(\lg v(G))$.

Como exemplo, considere a inserção da aresta $e = \{v_3, v_6\}$ no grafo G representado pela hierarquia de florestas da Figura 6.5. Como v_3 e v_6 estão em uma mesma árvore geradora de $F_{lmax} = F_4$, a aresta e se torna uma aresta de reserva em F_4 uma vez que $l(e) = 4$.

Note que as operações CONNECTED() e INSERT() dependem apenas do subgrafo G_{lmax} e de sua floresta geradora mínima, F_{lmax}, e são praticamente as mesmas operações descritas antes. Note também que as duas operações não violam as invariantes (i) e (ii), pois CONNECTED() não modifica o estado da estrutura de dados, enquanto INSERT() modifica apenas G_{lmax} e F_{lmax} sem adicionar nem remover vértices (nós). Logo, se (i) e (ii) são respeitadas antes da execução de INSERT(), elas são respeitadas imediatamente após, pois nenhuma componente de G_{lmax} possui mais do que $v(G)$ vértices e nenhum outro subgrafo é modificado.

A contribuição da estratégia proposta em (HOLM; LICHTENBERG; THORUP, 2001) reside na nova forma de executar REMOVE(e). Assim como antes, o primeiro passo da operação é remover a aresta $e = \{u, v\}$ de G. Em seguida, determina-se se e está ou não em F_{lmax}. Se não estiver, então e é uma aresta de reserva e uma cópia de e está armazenada em cada um dos dois nós, u e v, da floresta $F_{l(e)}$. Ambas as cópias são removidas de $F_{l(e)}$. Assume-se que u e v podem ser acessados em $F_{l(e)}$ em tempo constante a partir de e e $l(e)$.

Como exemplo, considere a remoção da aresta $e = \{v_5, v_7\}$ do grafo G representado pela hierarquia de florestas da Figura 6.5. A operação REMOVE() procura por e em F_4 e determina que ela não está lá. Isto significa que e é uma aresta de reserva armazenada nos nós v_5 e v_7 da floresta $F_{l(e)} = F_2$. Em seguida, os nós, v_5 e v_7, de F_2 são acessados e as respectivas cópias de e armazenadas nesses dois nós são removidas. Uma outra alternativa é usar um campo "sentinela" para indicar o status da aresta, que passa a ser "removida" após a remoção. Esta alternativa dispensa a remoção física das cópias da aresta nos dois nós.

Quando a aresta e está em F_{lmax}, ela é uma aresta de árvore e, pela invariante (ii), ela faz parte das florestas $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$, pois, de acordo com a invariante, todas as arestas de F_i estão em F_{i+1}, para todo $i \in \{0, 1, \ldots, lmax - 1\}$. Por exemplo, se e é a aresta $\{v_4, v_7\}$ do grafo G representado pela hierarquia de florestas da Figura 6.5, então e pertence às florestas F_1, F_2, F_3 e F_4, pois $l(e) = 1$. Assume-se que as cópias da aresta e em $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$ podem ser acessadas sequencialmente, a partir da cópia em F_{lmax}, o que permite que *todas* elas sejam acessadas e removidas de $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$ em tempo de pior caso $\mathcal{O}(lmax)$. Por exemplo, pode-se armazenar, em uma mesma lista simplesmente encadeada, as arestas de árvore que conectam os mesmos dois nós em florestas distintas.

A remoção de e da floresta F_j faz com que a árvore geradora mínima, T_j, de F_j, contendo u e v, seja dividida em duas, T_j^u e T_j^v, para todo $j \in \{l(e), l(e) + 1, \ldots, lmax\}$. Como toda aresta de F_i está em F_{i+1}, toda aresta de T_k^u (resp. T_k^v) está em T_{k+1}^u (resp. T_{k+1}^v), para todo $k \in \{l(e), l(e) + 1, \ldots, lmax - 1\}$. Por exemplo, se e é a aresta $\{v_4, v_7\}$ do grafo G representado pela hierarquia de florestas da Figura 6.5, então a remoção de e de F_1 faz com que a árvore, T_1, consistindo apenas dos nós v_4 e v_7 seja dividida em duas árvores, $T_1^{v_4}$ e $T_1^{v_7}$, uma contendo apenas v_4 e a outra contendo apenas v_7. Já em F_2, a remoção de e divide a árvore T_2, consistindo dos nós v_3, v_4 e v_7, nas árvores $T_2^{v_4}$ e $T_2^{v_7}$, uma contendo v_3 e v_4 e a outra contendo apenas v_7. Finalmente, note que as arestas de

$T_1^{v_4}$ estão em $T_2^{v_4}$ e que $T_1^{v_7}$ e $T_2^{v_7}$ são árvores isomorfas (contendo apenas uma cópia do nó v_7 cada).

O próximo passo é o mais "crítico" e o que motivou o desenvolvimento da estratégia de busca proposta em (HOLM; LICHTENBERG; THORUP, 2001): reconectar as árvores T_j^u e T_j^v, para todo $j \in \{l(e), l(e)+1, \ldots, lmax\}$, *caso a componente conexa de* $G_{lmax} = G$ *correspondente a* T_{lmax} *não seja desconectada pela remoção da aresta* e *de* G. Para tal, uma aresta, f, de reserva deve ser encontrada. Obviamente, tal aresta só pode existir se a componente conexa correspondente a T_{lmax} no grafo G_{lmax} continuar conexa em $G_{lmax} - e$. A existência de f é determinada, aqui, pela própria busca. Em outras palavras, se a busca por f falhar, então se conclui que f não pode existir. Caso contrário, tem-se uma aresta, f, de reserva para reconectar todos os $lmax - l(e) + 1$ pares, T_j^u e T_j^v, de árvores geradoras mínimas.

A busca por uma aresta, f, de reserva é realizada na sequência $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$ de florestas geradoras mínimas e nesta ordem. Logo, apenas uma aresta, f, de reserva com nível, $l(f)$, maior ou igual a $l(e)$ pode ser encontrada. Além disso, se tal aresta f existir, ela é uma aresta de reserva de menor nível — *entre todas as arestas de reserva em* $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$ — que reconecta os pares de árvores T_j^u e T_j^v, para todo $j \in \{l(f), l(f)+1, \ldots, lmax\}$. Observe que é crucial se restringir a busca por uma aresta, f, de reserva às florestas $F_{l(e)}$, $F_{l(e)+1}$, ..., F_{lmax}, pois se $l(f)$ fosse menor do que $l(e)$, então a floresta $F_{l(e)}$ não seria uma floresta geradora mínima *antes* de e ser removida. Logo, a invariante (ii) seria violada. Por outro lado, a busca como descrita acima mantém esta invariante.

O algoritmo que detalha os passos da busca se resume a um laço no qual um contador, k, varia de $l(e)$ a $lmax$. O laço é repetido até que uma aresta, f, de reserva que reconecta T_k^u a T_k^v seja encontrada ou k seja igual a $lmax+1$. No primeiro caso, a busca termina com sucesso; no segundo, a busca falha. Dentro do corpo do laço, determina-se qual das duas árvores, T_k^u ou T_k^v, possui o menor número de nós. Sem perda de generalidade, assuma que seja T_k^u. Então, para cada aresta, $f = \{x, y\}$, de reserva armazenada em um nó x de T_k^u, determina-se se o outro extremo, y, da aresta é um nó em T_k^v. Se ele for, então $f = \{x, y\}$ reconecta T_k^u a T_k^v. Logo, ela é inserida nas florestas $F_{l(f)}, F_{l(f)+1}, \ldots, F_{lmax}$ e a busca termina com sucesso. Se y não for um nó de T_k^v, então $f = \{x, y\}$ *não* reconecta T_k^u a T_k^v. No entanto, após este fato ser constatado, o nível, $l(f)$, de f é decrementado em uma unidade. Note que isto equivale a mover a aresta f da floresta F_k para a floresta F_{k-1}.

Os passos da operação de remoção quando $e = \{u, v\} \in F_{lmax}$ estão abaixo:

(1) Remova e de $F_{l(e)}, F_{l(e)+1}, \ldots, F_{lmax}$

(2) Para todo k de $l(e)$ a $lmax$ faça:

   (a) Determine que árvore, T_k^u ou T_k^v, possui o menor número de nós. Sem perda de generalidade, suponha que a árvore com o menor número de nós seja T_k^u.

   (b) Para cada aresta, $f = \{x, y\}$, de reserva em T_k^u, com $x \in T_k^u$, faça:

      (i) Se y pertence a T_k^v, então insira f em $F_k, F_{k+1}, \ldots, F_{lmax}$ e pare.

      (ii) Caso contrário, faça $l(f) \leftarrow k - 1$, o que equivale a remover a aresta, f, de reserva da floresta F_k e inseri-la, como aresta de reserva, na floresta F_{k-1}.

Como exemplo da execução do algoritmo acima, considere a remoção da aresta $e = \{v_4, v_7\}$ do grafo G representado pela hierarquia de florestas da Figura 6.5. O passo (1) do algoritmo acima faz com que e seja removida das florestas F_1, F_2, F_3 e F_4, pois $l(e) = 1$. A hierarquia de florestas resultante da remoção da aresta e é exibida na Figura 6.6. O passo (2) é iniciado com a execução do laço para $k = l(e) = 1$. Na iteração $k = 1$, as árvores $T_1^{v_4}$ e $T_1^{v_7}$ possuem ambas o mesmo número de nós e nenhuma aresta de reserva. Logo, o laço em (2)(b) não é sequer executado e o valor de k é incrementado, tornando-se 2. Na iteração $k = 2$, as árvores $T_2^{v_4}$ e $T_2^{v_7}$ possuem 2 nós e 1 nó, respectivamente. Logo, o passo (2)(b) é executado para a árvore $T_2^{v_7}$. Esta árvore possui apenas uma única aresta de reserva armazenada em seus nós: a aresta $f = \{v_5, v_7\}$. Como $v_7 \in T_2^{v_7}$, o nó y do algoritmo é v_5. Já que $v_5 \notin T_2^{v_4}$, o passo (2)(b)(ii) é executado ao invés de (2)(b)(i). Consequentemente, a aresta $f = \{v_5, v_7\}$ (de reserva) é removida de F_2 e inserida em F_1. A busca continua com a iteração $k = 3$. As árvores $T_3^{v_4}$ e $T_3^{v_7}$ possuem 5 nós e 1 nó, respectivamente. Logo, o passo (2)(b) é executado para a árvore $T_3^{v_7}$. Esta árvore possui uma única aresta de reserva armazenada em seus nós: a aresta $f = \{v_3, v_7\}$. Como $v_7 \in T_3^{v_7}$, o nó y do algoritmo é v_3. Como $v_3 \in T_3^{v_4}$, o passo (2)(b)(i) é executado, o que resulta na inserção da aresta $f = \{v_3, v_7\}$, como aresta de árvore, nas florestas F_3 e F_4, finalizando a busca. A hierarquia de florestas geradoras mínimas resultante está na Figura 6.7.

Para analisar a complexidade amortizada da operação REMOVE(), deve-se assumir que as árvores da hierarquia de florestas são representadas por estruturas de dados que suportam todas as operações das árvores dinâmicas em tempo amortizado $\mathcal{O}(\lg n)$ cada, onde n é o número de nós das árvores envolvidas na operação, além de outras operações descritas mais adiante. Como já foi dito, este é o caso das árvores ST estudadas no Capítulo 5.

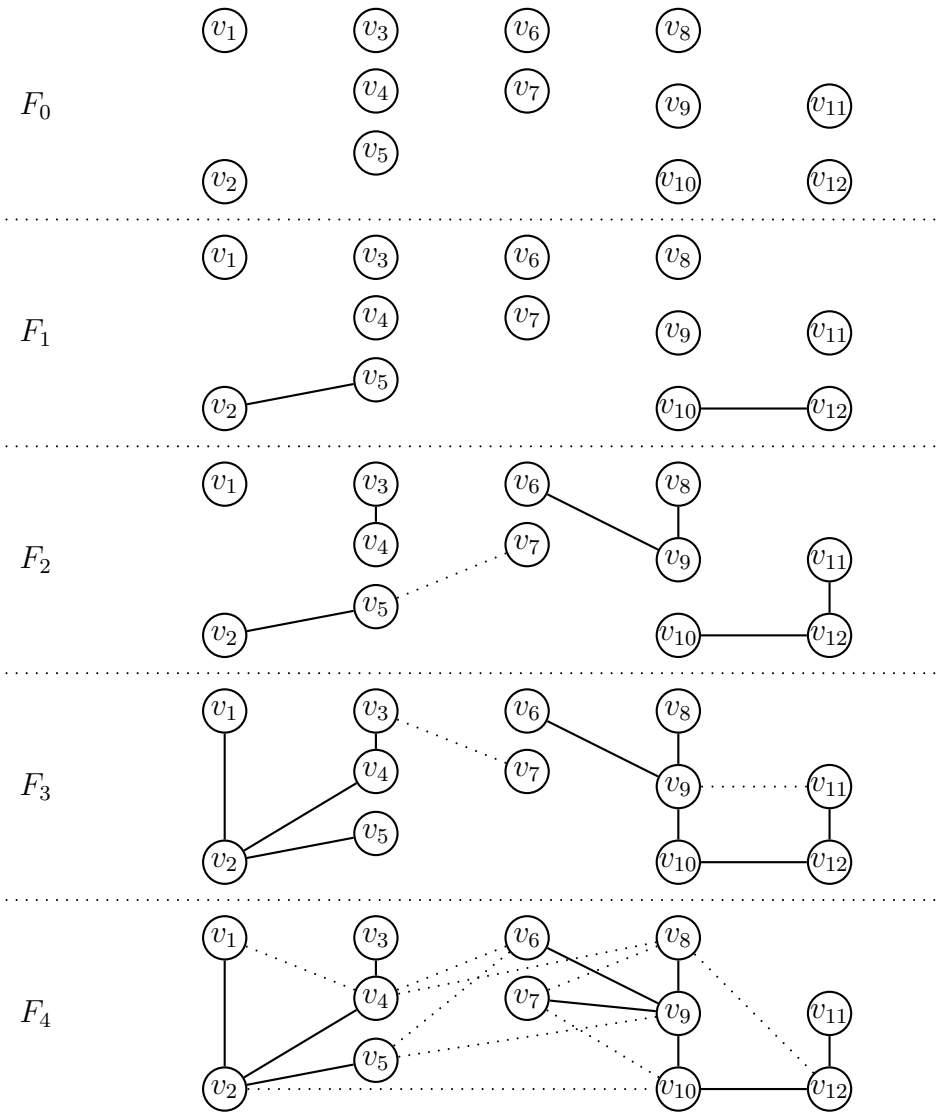


Figura 6.6: A hierarquia de florestas da Figura 6.5 após a remoção da aresta $\{v_4, v_7\}$.

A execução de REMOVE(e) começa com a verificação do status de aresta e: esta aresta é uma aresta de árvore ou de reserva? Se um campo de status (aresta de árvore ou de reserva) for adicionado à estrutura de dados que representa uma aresta, esta determinação pode ser feita em tempo constante. Se a aresta for de árvore, então $e \in F_{lmax}$ e os passos do algoritmo que se acabou de ver devem ser executados. Caso contrário, deve-se remover e da floresta $F_{l(e)}$, o que leveria tempo amortizado $\mathcal{O}(\lg v(G))$, ou simplesmente mudar o status de e para aresta "removida", o que levaria tempo constante. Então, pode-se considerar que determinar se $e \in F_{lmax}$ e tratar o caso $e \notin F_{lmax}$ consome, no total, tempo amortizado $\mathcal{O}(\lg v(G))$.

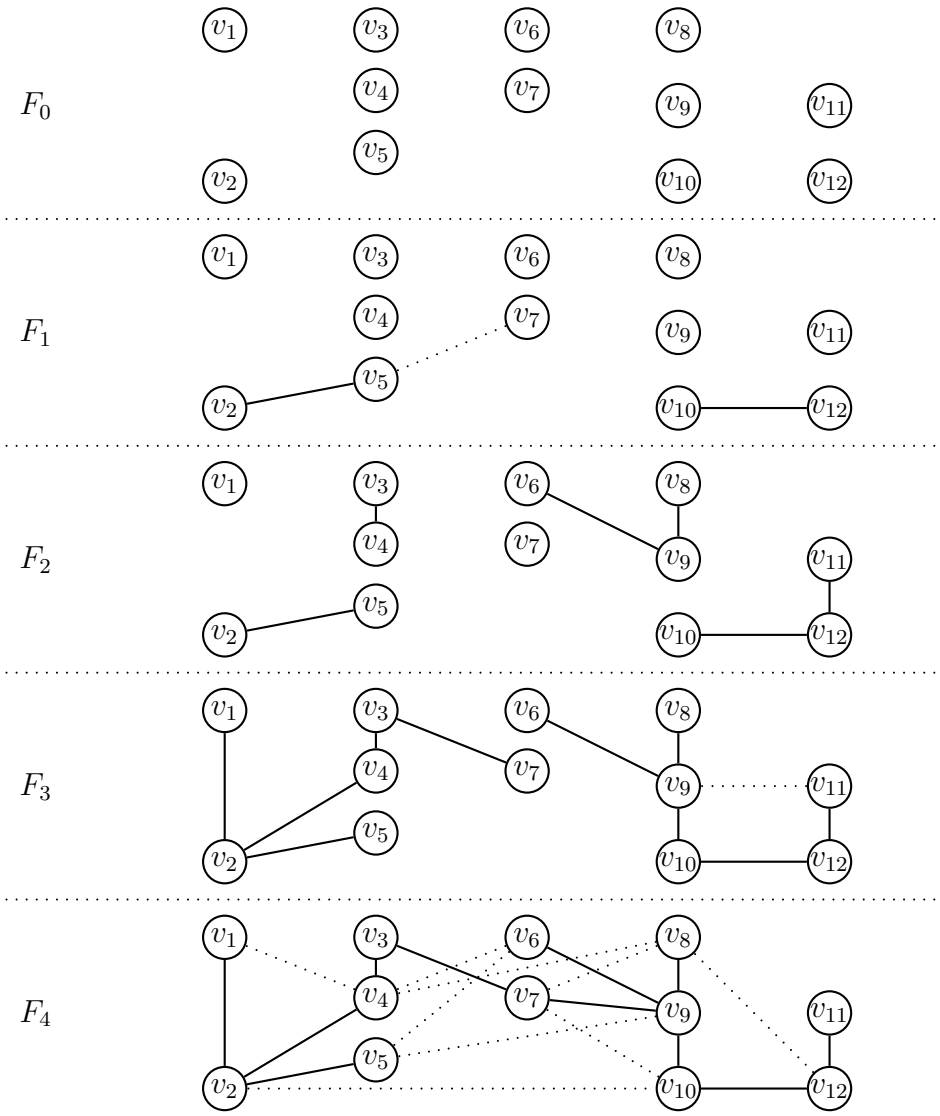


Figura 6.7: A hierarquia de florestas resultante da operação de remoção da aresta $\{v_4, v_7\}$.

Considere o caso $e \in F_{lmax}$. O passo (1) do algoritmo remove e das florestas $F_{l(e)}$, $F_{l(e)+1}$, ..., F_{lmax}. No pior caso, tem-se $l(e) = 1$, pois F_0 não pode possuir arestas. Logo, o passo (1) requer, no pior caso, *lmax* remoções da aresta e tal que cada remoção é realizada em uma árvore de cada uma das florestas $F_1, F_2, \ldots, F_{lmax}$. Usando árvores ST para representar as florestas, as *lmax* remoções se reduzem a *lmax* execuções de CUT(), o que implica que o passo (1) consome tempo amortizado $\mathcal{O}(lmax \cdot \lg v(G))$. Como $lmax = \lceil \lg v(G) \rceil$, conclui-se que a complexidade amortizada de tempo do passo (1) é $\mathcal{O}(\lg^2 v(G))$.

Para a análise do passo (2), deve-se levar em contar dois detalhes que não são fornecidos no próprio artigo que descreve o TAD HLT (HOLM; LICHTENBERG; THORUP, 2001), a saber:

- A determinação de qual árvore, T_k^u ou T_k^v, possui o menor número de nós pode ser realizada em tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós de T_k^u e T_k^v juntas.
- A "próxima" aresta, $f = \{x, y\}$, de reserva em T_k^u pode ser encontrada em tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós de T_k^u. Logo, o tempo amortizado de cada iteração do laço do passo (2)(b) deve ser acrescido do fator $\mathcal{O}(\lg n)$, que leva em conta o tempo amortizado para encontrar a próxima aresta, f, reserva na árvore T_k^u.

Ambas as operações acima não fazem parte da interface canônica das árvores dinâmicas. Felizmente, a primeira delas pode ser disponibilizada na interface das árvores ST e executada em tempo *constante* sem acarretar um aumento da complexidade das demais operações da interface, como será descrito no Capítulo 7. Por outro lado, a segunda operação se constituiu na principal dificuldade deste trabalho, pois ela não pode ser "naturalmente" implementada em árvores ST, embora os autores do artigo (HOLM; LICHTENBERG; THORUP, 2001) afirmem, *categoricamente* e *repetidamente*, que o TAD HLT pode ser implementado com a árvore ST e que a complexidade da operação REMOVE() será $\mathcal{O}(\lg^2 v(G))$.

Para continuar com a análise, assume-se que as demandas de complexidade dos dois detalhes acima são satisfeitos (independentemente da árvore dinâmica usada para implementar o TAD HLT). Logo, pode-se considerar que o tempo de cada execução do passo (2)(a) é $\mathcal{O}(\lg^2 v(G))$. No pior caso, este passo é executado *lmax* vezes para cada execução de REMOVE(), o que resulta em tempo amortizado $\mathcal{O}(\lg^2 v(G))$. Observe que a escolha da

árvore de menor número de nós, entre T_k^u e T_k^v, garante que a árvore selecionada possui, no máximo, 2^{k-1} nós, pois, de acordo com a invariante (i), a soma dos números de nós de T_k^u e T_k^v não pode exceder 2^k. A escolha da árvore de menor número de nós é uma heurística para acelerar a busca por f no passo (2), mas que *não* reduz a complexidade da busca.

A análise dos passos (2)(b)(i) e (2)(b)(ii) é feita de forma "agregada". Esses passos são mutuamente exclusivos. O passo (2)(b)(i) é realizado, no máximo, uma vez por execução de REMOVE(), enquanto o passo (2)(b)(ii) pode ser executado um certo número de vezes antes da única execução do passo (2)(b)(i) ou até o contador k atingir o valor $lmax + 1$. Sejam t_{caso_i} e $t_{\text{caso}_{ii}}$ os tempos amortizados gastos com os passos (2)(b)(i) e (2)(b)(ii) em uma única execução de REMOVE()[1]. Então, pelo que foi discutido nos quatro parágrafos anteriores, o tempo amortizado da operação REMOVE() é limitado superiormente pela seguinte expressão:

$$t_{\text{caso}_i} + t_{\text{caso}_{ii}} + \mathcal{O}(\lg^2 v(G))\,.$$

Se o passo (2)(b)(i) for executado, haverá $lmax - k + 1$ inserções da aresta, $f = \{x, y\}$, de reserva nas florestas F_k, $F_{k+1}, \ldots, F_{lmax}$. Cada inserção faz com que f se torne uma aresta de árvore na respectiva floresta. Usando árvores ST, cada inserção pode ser feita com a execução da operação LINK(), que leva tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós das duas árvores envolvidas. Logo, o tempo amortizado da única execução do passo (2)(b)(i) é $\mathcal{O}((lmax - k + 1) \cdot \lg v(G))$. No pior caso, tem-se $k = 1$, o que resulta em tempo amortizado $\mathcal{O}(\lg^2 v(G))$. Como a execução do passo (2)(b)(i) é precedida pela busca pela "próxima" f no passo (2)(b), tem-se que t_{caso_i} é limitado superiormente por uma função em

$$\mathcal{O}(\lg^2 v(G)) + \mathcal{O}(\lg v(G)) = \mathcal{O}(\lg^2 v(G))\,.$$

Cada execução do passo (2)(b)(ii) decrementa, em uma unidade, o nível, $l(f)$, da aresta, $f = \{x, y\}$, de reserva. Isto equivale a remover f da floresta F_k e a inserir f na floresta F_{k-1}, como aresta de reserva também. Tanto a inserção quanto a remoção não alteram as árvores dinâmicas, pois a aresta é armazenada nos nós x e y de F_{k-1}. Assumindo que esses nós são acessados em tempo constante, uma execução do passo (2)(b)(ii) consome tempo $\Theta(1)$. No entanto, cada execução deste passo é precedida pela busca pela "próxima" f no passo (2)(b). Logo, o tempo amortizado de uma execução do passo (2)(b)(ii), incluindo o tempo gasto com a busca em (2)(b), é uma função em $\mathcal{O}(\lg v(G))$.

[1]Mais precisamente, t_{caso_i} e t_{ii} são funções de $v(G)$.

Para concluir a análise, note que uma aresta, f, de reserva só pode mudar de "nível" na hierarquia de florestas — isto é, passar de F_k para F_{k-1} quando o valor de $l(f)$ é decrementado em uma unidade — $lmax$ vezes, no máximo. Isto decorre do fato de quando uma aresta de reserva é colocada em F_0, ela não é mais considerada pelo algoritmo, pois não existem arestas de árvore em F_0. Logo, o nível, $l(e)$, de qualquer aresta de árvore considerada para remoção é, pelo menos, igual a 1. Então, o valor de k no passo (2) não pode nunca ser menor do que 1. Note também que a mudança do nível $l(f)$ para $l(f) - 1$ de uma *mesma* aresta, f, de reserva só pode ocorrer uma única vez por execução de REMOVE().

Lembre-se de que o conjunto, E, de arestas do grafo G está "vazio" antes da inicialização do grafo e de que o tempo amortizado de cada execução de INSERT() é $\mathcal{O}(\lg v(G))$. Logo, toda aresta, e, removida de G tem de ser inserida em G antes de ser removida. Suponha, portanto, que o tempo amortizado da operação INSERT() seja "modificado" para $\Theta(\lg^2 v(G))$, isto é, a operação se tornou mais cara de forma "justa". Intuitivamente, isto significa que há uma "sobra" de tempo para ser deduzida de outras operações, que pode ser vista como um "pagamento adiantado" por uma operação futura. No caso específico desta análise, deseja-se gastar o tempo futuro, de forma antecipada, do passo (2)(b)(ii). Em particular, cada vez que o passo (2)(b)(ii) for executado para uma *mesma* aresta, f, de reserva, deduz-se o tempo gasto com esta operação, que é $\mathcal{O}(\lg v(G))$, da folga creditada à aresta na operação de inserção. Como o passo (2)(b)(ii) só pode ser executado, para esta mesma aresta f, $\mathcal{O}(\lg v(G))$ vezes, o tempo gasto total não excede $\Theta(\lg^2 v(G))$ e é coberto pelo "pagamento adiantado" realizado durante a operação INSERT() que adicionou f ao grafo G.

A observação do parágrafo anterior permite que se conclua que $t_{\text{caso}_{ii}}$ está em $\mathcal{O}(\lg^2 v(G))$, pois

$$t_{\text{caso}_{ii}} = \frac{m_e \cdot \mathcal{O}(\lg^2 v(G))}{\eta}\,,$$

onde η é o número de vezes em que INSERT() é executada para as m_e arestas de reserva que mudaram de nível ao longo das execuções de REMOVE(). Como $m_e = \eta$, o resultado segue. Observe que o número de inserções, ao invés do número de remoções, deve ser mesmo usado na fórmula acima, pois são as operações de inserção que pagam pelas $m_e \cdot \mathcal{O}(\lg^2 v(G))$ excuções do passo (2)(b)(ii). Logo, o tempo amortizado de uma única execução de REMOVE(), que é $t_{\text{caso}_i} + t_{\text{caso}_{ii}} + \mathcal{O}(\lg^2 v(G))$, está de fato em $\mathcal{O}(\lg^2 v(G))$, pois se tem que

$$t_{\text{caso}_i}, t_{\text{caso}_{ii}} \in \mathcal{O}(\lg^2 v(G))\,.$$

Finalmente, tem-se que a complexidade de espaço da estratégia proposta é assintoticamente maior do que a da solução que não se utiliza de uma hierarquia de florestas. De fato, cada vértice de G ocorre em cada um dos $lmax$ níveis da hierarquia. Logo, gasta-se $\Theta(v(G) \cdot \lg v(G))$ unidades de memória com o armazenamento de vértices. Por fim, cada aresta de árvore (há, no máximo, $v(G) - 1$ delas) pode aparecer em todos os níveis, o que também requer $\mathcal{O}(v(G) \cdot \lg v(G))$ unidades de memória. Há $\mathcal{O}(e(G))$ arestas de reserva, pois elas ocorrem apenas uma vez em toda a hierarquia. Logo, a hierarquia de florestas (e, portanto, o TAD HLT) requer $\mathcal{O}(e(G) + v(G) \cdot \lg v(G))$ unidades de memória para ser representada.

6.3 Implementação do TAD HLT

Esta seção descreve os detalhes de alto-nível da implementação do TAD HLT que foi realizada neste trabalho. Os detalhes de baixo-nível são objeto do Capítulo 7. A estrutura de dados utilizada aqui para implementar o TAD HLT se baseia na árvore ST do Capítulo 5.

Como mencionado na Seção 6.2, para que o tempo amortizado de cada operação de consulta ou alteração do grafo G esteja em $\mathcal{O}(\lg^2 v(G))$, a busca por arestas de reserva para reconectar árvores geradoras deve seguir a estratégia descrita na Seção 6.2.1, que foi proposta em (HOLM; LICHTENBERG; THORUP, 2001), ou a alternativa proposta em (WULFF-NILSEN, 2013), que é ainda mais eficiente e que não foi discutida aqui. Quando o trabalho ora descrito foi iniciado, conhecia-se apenas a primeira delas e, obviamente, procurou-se implementá-la.

De acordo com os autores, Holm, Lichtenberger e Thorup, da estratégia de busca em (HOLM; LICHTENBERG; THORUP, 2001), qualquer árvore dinâmica poderia ser utilizada na implementação do TAD HLT. Em particular, os autores citaram, no referido artigo, a árvore ST do Capítulo 5 em mais de uma ocasião. Como esta árvore é mais "popular" do que as demais árvores dinâmicas conhecidas, optou-se por adotá-las na implementação do TAD HLT.

A estratégia de busca da Seção 6.2.1, no entanto, exige que a interface da árvore dinâmica utilizada na implementação do TAD HLT seja acrescida de duas funções. A primeira função possibilita que o número de nós da subárvore enraizada em qualquer dado nó de uma árvore seja determinado em tempo $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore. A segunda função possibilita que a "próxima" aresta de reserva armazenada em algum nó

da árvore seja obtida em tempo $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore. A primeira e a segunda funções são utilizadas pelos passos (2)(a) e (2)(b), respectivamente, do algoritmo da operação REMOVE(), como descrito na Seção 6.2.1. O acréscimo da interface da árvore dinâmica com essas duas funções não pode afetar a complexidade das demais funções da interface.

A primeira função foi facilmente incorporada à interface da árvore ST e sua execução consome tempo *constante* (no pior caso) por chamada. Os detalhes de implementação desta função estão no Capítulo 7. Infelizmente, a segunda função não pode ser "naturalmente" implementada em árvores ST. Mais precisamente, *o autor deste trabalho* não sabe como fazer isso sem afetar a complexidade das demais operações da interface da árvore e nem conhece nenhum trabalho disponível na literatura pesquisada que descreva como se faz.

O problema é que a afirmação do artigo (HOLM; LICHTENBERG; THORUP, 2001) de que "qualquer" árvore dinâmica pode ser usada e, em particular, a árvore ST fez com que o autor deste trabalho investisse tempo e esforço em implementar o TAD HLT com tal árvore. Através de uma vídeo-aula[2], ministrada em 3 maio de 2012 pelo Prof. Erik Demaine do *Massachussetts Institute of Technology* (MIT), o autor deste trabalho tomou conhecimento de que a árvore ST não era mesmo apropriada para ter sua interface acrescida da segunda função. Na mesma aula, o professor Demaine menciona que a árvore dinâmica adequada para se implementar o TAD HLT é a ET *tree* (MILTERSEN et al., 1994; TARJAN, 1997).

Surpreendentemente, a informação fornecida pelo Prof. Erik Demaine não está escrita em nenhum artigo encontrado pelo autor deste trabalho. Nem mesmo no único artigo disponível na literatura que descreve uma implementação do TAD HLT (IYER JR. et al., 2001), que não por acaso utiliza a árvore ET. Para piorar, a dificuldade em se utilizar uma dada árvore dinâmica reside na forma de se decompor a árvore real para que ela seja representada pela árvore virtual. No caso da árvore ST, a decomposição é por "caminhos" (veja o Capítulo 5) e este tipo de decomposição não se presta, natural e eficientemente, para manter as informações necessárias à execução da segunda função sem afetar a complexidade das demais funções da interface da árvore. Isto não significa "imposibilidade" de solução do problema, mas sim que o problema da incorporação não deve ser simples de ser resolvido.

De acordo com Werneck em (WERNECK, 2006), a árvore ST foi desenvolvida com

[2]http://www.youtube.com/watch?v=fKlPmxFpYDk

uma aplicação em mente: encontrar arestas de custo mínimo em caminhos disjuntos. Obviamente, há outras aplicações às quais elas se aplicam. No entanto, para se utilizar tal árvore em uma dada aplicação, necessita-se (1) definir um novo conjunto de valores (isto é, "custos") a ser armazenado em cada nó da árvore, (2) certificar-se de que esses valores são atualizados apropriadamente quando a árvore virtual muda e (3) definir regras para percorrer a árvore durante as operações de consulta. No contexto deste trabalho, o ponto (3) é o que interessa. Devido à decomposição por caminhos, a árvore ST não se presta naturalmente para operações que necessitem agregar valores em subárvores *quaisquer* da árvore virtual. Logo, não parece trivial determinar em tempo logarítmico no número de nós, por exemplo, se a subárvore enraizada em um dado nó da árvore virtual contém uma aresta de reserva (e se houver, acessar o nó que a contém), pois tal subárvore pode agregar vários caminhos.

Infelizmente, o autor deste trabalho só tomou conhecimento da confirmação da limitação da árvore ST[3] quando o prazo para término deste trabalho já estava muito próximo e a implementação realizada até o momento já estava em um estágio bastante avançado. Então, decidiu-se finalizar a implementação do TAD HLT sem utilizar a estratégia de busca da Seção 6.2.1, mas sim a busca "menos cuidadosa" descrita no início da Seção 6.2. Consequentemente, a complexidade amortizada da operação REMOVE() implementada aqui é assintoticamente maior do que $\Theta(\lg^2 v(G))$. Por outro lado, devido ao fato do algoritmo de Diks e Stanczyk (veja a Seção 3.3) ser aplicado apenas a grafos cúbicos, todo nó da árvore virtual — com exceção do nó raiz — possui, no máximo, dois filhos. Isto possibilitou ao *autor deste trabalho* estender a interface desta árvore ST "restrita" de forma que a complexidade desejada para a remoção (isto é, $\mathcal{O}(\lg^2 v(G))$) seja atingida. Esta *nova* solução não será descrita aqui, mas a implementação dela já se encontra em andamento. É importante ressaltar que *esta solução só se aplica ao caso em que G é um grafo cúbico.*

6.3.1 O uso da árvore ST

A implementação do TAD HLT consiste na implementação das três operações da interface do TAD: CONNECTED(), INSERT() e REMOVE(). A primeira delas recebe um par, u e v, de vértices do grafo G como entrada, enquanto a segunda e a terceira recebem uma aresta, e, de G. Como descrito no início da Seção 6.2, as três operações são executadas em uma estrutura de dados que representa G e numa única floresta, F, geradora de G

[3]Usando a referida vídeo-aula.

(ou seja, não se utiliza uma hierarquia de florestas). As árvores da floresta, F, são árvores ST.

Inicialmente, constrói-se o grafo G apenas com o seu conjunto, $v(G)$, de vértices, que se supõe ser fixo. Para tal, cria-se a floresta F com $v(G)$ árvores, cada qual consistindo de um vértice distinto de G. Isto é feito com $v(G)$ chamadas à função MakeTree(). A partir daí, as arestas de G são inseridas em G e em F. Isto é feito com uma sequência de chamadas à função Insert(), uma para cada aresta. Após a construção do grafo G, a estrutura de dados estará pronta para responder as consultas sobre conectividade de vértices, realizadas por chamadas à função Connected(), e para atualizar o grafo mediante inserções e remoções de arestas, realizadas com chamadas às funções Insert() e Remove().

Por brevidade, o pseudocódigo de Connected() não é mostrado aqui, já que esta função simplesmente compara o resultado de duas chamadas a FindRoot(). O pseudocódigo da função Insert() é mostrado no Algoritmo 6.1. Uma explicação deste código já foi dada no início da Seção 6.2 e, portanto, não será repetida aqui. É importante ressaltar, no entanto, que se os vértices u e v da aresta e a ser inserida em G já estiverem conectados em F (isto é, se fizerem parte de uma mesma árvore geradora de F), então a aresta e se torna uma aresta de reserva. Isto significa que uma cópia desta aresta é armazenada no nó u e outra, no nó v. Se a aresta e não estiver em F, então ela é inserida em F como uma aresta de árvore. A Figura 6.1 ilustra a execução de uma chamada à função.

Algoritmo 6.1 Insert(e)

Entrada: Uma aresta $e = \{u, v\}$
Saída: Nenhuma

```
adicione e no grafo G
se Connected(u, v) então
    adicione e como uma aresta de reserva em u e em v
senão
    se u ≠ FindRoot(u) então
        Evert(u)
    fim se
    Link(u, v)
fim se
```

O pseudocódigo para a função Remove() está no Algoritmo 6.2. Esta função determina, primeiramente, se a aresta, $e = \{u, v\}$, a ser removida do grafo G é aresta de árvore ou aresta de reserva. Se e for uma aresta de reserva, duas cópias dela estão armazenadas em F, uma no nó u e outra no nó v. Então, tais cópias são removidas de F e a função é encerrada, pois nenhuma árvore de F precisa ser modificada. Se e for uma aresta de

árvore, então e deve ser removida da árvore, T, que a contém em F. Isto é feito com uma chamada à função CUT(). Mas, como dito no início da Seção 6.2, antes de CUT() ser chamada para retirar e de T, uma chamada a FINDPARENT() deve ser feita para se determinar se u é o pai de v em T ou vice-versa. Uma vez que se saiba quem é o pai entre u e v, a aresta e pode ser seguramente removida com uma chamada à função CUT(). Isso acarreta a divisão de T em duas outras árvores, T_u e T_v, uma contendo u e outra contendo v, como ilustrado na Figura 6.2. Porém, a componente correspondente em G pode não estar dividida, o que significa que possivelmente existe uma aresta de reserva que reconecta T_u e T_v. Para saber com certeza, realiza-se uma busca executando a chamada REPLACE(e).

Algoritmo 6.2 REMOVE(e)

Entrada: Uma aresta $e = \{u, v\}$
Saída: Nenhuma

```
remova e do grafo G
se e é uma aresta de árvore então
    paiU ← FINDPARENT(u)
    se v = paiU então
        CUT(u)
    senão
        CUT(v)
    fim se
    REPLACE(e)
senão
    remova e como uma aresta de reserva em u e em v
fim se
```

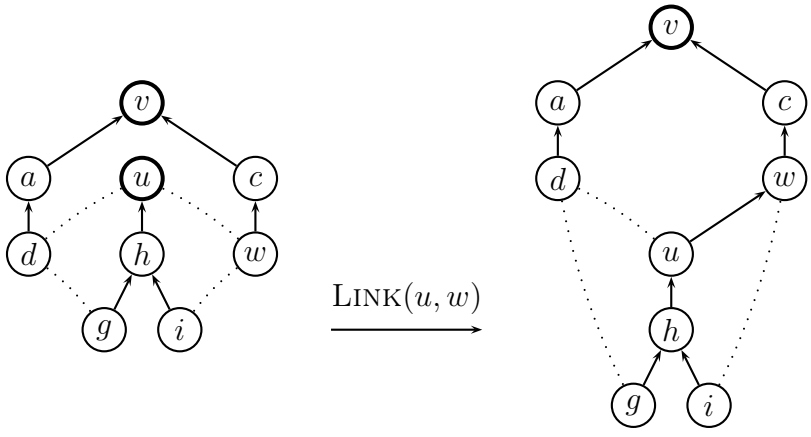


Figura 6.8: Execução de REPLACE(e), com $e = \{u, v\}$. As raízes estão em destaque

A função REPLACE() faz uma busca por uma das duas árvores, T_u ou T_v, resultantes da divisão de T, procurando por uma aresta, $f = \{x, y\}$, de reserva tal que x esteja na mesma árvore que u e y na mesma que v (ou vice-versa). Convenciona-se, no pseudocódigo de REPLACE() (veja o Algoritmo 6.3), realizar a busca pela subárvore T_u. Se tal aresta f existe, a busca é interrompida, as duas cópias de f nos nós x e y de F são removidas (pois, f é uma aresta de reserva) e, em seguida, a aresta f é inserida em F, como aresta de árvore.

Algoritmo 6.3 REPLACE(e)

```
Entrada: Uma aresta e = {u, v}
Saída: Nenhuma
  raizU ← FINDROOT(u)
  raizV ← FINDROOT(v)
  encontrado ← falso
  enquanto ¬encontrado e houver vértices não visitados em T_u faça
    x ← próximo vértice de T_u
    para cada aresta de substituição f = {x, y} armazenada em x faça
      raizX ← FINDROOT(x)
      raizY ← FINDROOT(y)
      se (raizX = raizU e raizY = raizV) ou (raizX = raizV e raizY = raizU)
      então
        encontrado ← verdadeiro
        se x ≠ raizX então
          EVERT(x)
        fim se
        LINK(x, y)
        remova f como uma aresta de reserva em x e y
      fim se
    fim para
  fim enquanto
```

Mais especificamente, verifica-se se x é a raiz de sua árvore real (chamando EVERT(x) caso não seja) e chama-se LINK(x, y). Se, ao final da busca, nenhuma aresta for encontrada, não existe aresta de reserva para substituir a aresta e em F. Isto implica que o grafo G possui uma componente a menos do que o grafo $G - e$, ou seja, que a remoção de e do grafo G também desconectou uma componente conexa de G. Neste caso, as árvores T_u e T_v são geradoras das duas componentes resultantes. A Figura 6.8 ilustra a execução de uma chamada a REPLACE() após a remoção da aresta $\{u, v\}$ na Figura 6.2. A busca é iniciada no nó u, que é raiz, e encontra a aresta de reserva que o liga ao vértice w. Como essa aresta atende ao critério de reconectar a árvore dividida, ela é inserida através de um LINK(u, w).

A operação Replace(), como descrita no Algoritmo 6.3, pode ser bastante onerosa, pois, no pior caso, $\Omega(v(G))$ nós de F podem ser visitados até que uma aresta de substituição seja encontrada ou se constate que tal aresta não existe. Ao contrário da estratégia descrita na Seção 6.2.1, este pior caso pode ocorrer em cada chamada de Replace(), de forma que o mesmo argumento de amortização não pode ser utilizado aqui. Logo, o tempo amortizado de de uma chamada à operação Replace() pode, no pior caso, estar em $\Theta(v(G) \cdot \lg v(G))$.

7 Implementação e Resultados

Este capítulo discute as decisões tomadas e os problemas encontrados durante a fase de codificação do algoritmo de Diks e Stanczyk (veja Seção 3.3) e das estruturas de dados discutidas nos Capítulos 4 a 6. A Seção 7.1 descreve a estrutura do código e as interfaces das classes envolvidas. A Seção 7.2 apresenta os resultados do código implementado com respeito à sua execução em grafos oriundos de uma base de objetos geométricos. Por último, a Seção 7.3 oferece uma discussão sobre os resultados obtidos com os testes da Seção 7.2.

7.1 Código

A etapa de implementação foi iniciada após o estudo detalhado da bibliografia sobre cada um dos assuntos descritos nos Capítulos 3 a 6. O código foi escrito "de dentro para fora", isto é, começando pelas árvores splay, passando pela árvore ST e a estrutura de conectividade dinâmica e chegando, finalmente, ao algoritmo de emparelhamentos de Diks e Stanczyk. Durante o processo, tentou-se utilizar bibliotecas já prontas que modelassem grafos, vértices e arestas (tais como a LEMON ou a Boost), mas as necessidades específicas do projeto motivaram a codificação de classes *ad-hoc*, mais simples e fáceis de utilizar. Todo o código foi escrito na linguagem C++, em parte para se adaptar a uma biblioteca já existente responsável por realizar a leitura de uma malha de triângulos de um arquivo `OFF` (utilizado na etapa de testes) e em parte por maior familiaridade do autor com a linguagem.

7.1.1 A estrutura de dados do grafo

Existem duas representações bastante conhecidas para modelar grafos como estruturas de dados: *matrizes de adjacência* e *listas de adjacência*. Porém, nenhuma das duas é perfeitamente adequada às necessidades deste projeto. Como o único interesse aqui é em

grafos cúbicos e sem pontes, que são bastante esparsos, a representação com matrizes de adjacência desperdiçaria uma quantidade significativa de memória, reservando espaço para arestas que nunca vão existir. Além disso, uma das consultas mais utilizadas no algoritmo para calcular emparelhamentos perfeitos é a iteração sobre todos os vizinhos de um vértice (e as arestas que o ligam a esses vizinhos), uma operação bastante custosa em uma matriz de adjacência (visto que toda uma linha deve ser percorrida). A lista de adjacência, embora resolva essas duas dificuldades, apresenta dois outros problemas: não há maneira óbvia de se representar laços ou arestas múltiplas e não há onde se armazenar informações associadas às arestas (como um identificador para cada aresta, por exemplo).

A solução encontrada foi utilizar uma variante da lista de adjacência, conhecida na literatura como *lista de incidência*, na qual cada vértice armazena uma lista de referências para objetos que representam arestas incidentes sobre o vértice. Cada aresta, em contrapartida, possui referências para seus dois vértices extremos. Embora os objetos extras causem um maior consumo de memória do que a versão original da lista de adjacência, eles fornecem um lugar conveniente onde armazenar, por exemplo, um campo identificador que torne possível distinguir entre dois laços ou duas arestas múltiplas diferentes (novamente, algo necessário ao algoritmo de emparelhamento perfeito). Em particular, as classes `Edge`, `Vertex` e `Graph` foram criadas para representar, respectivamente, arestas, vértices e grafos.

Cada objeto da classe `Edge` possui um identificador na forma de um inteiro sem sinal (o campo `_id`) e referências para os dois objetos da classe `Vertex` que são seus extremos (os campos `_v` e `_w`). Note que, como o grafo em questão não é dirigido, não há uma ordem definida entre os dois extremos. A Figura 7.1 mostra a interface pública da classe `Edge` e seus campos privados. Os métodos `getRandomAdjacentEdge`() e `isSimple`() foram adicionados para aumentar a legibilidade do código do algoritmo de emparelhamento perfeito e retornam, respectivamente, a primeira aresta incidente em `_v` ou `_w` cujo identificador seja diferente de `_id` (ou **n**il caso não exista nenhuma outra aresta) e um valor booleano indicando se a aresta em questão é simples ou não. Eles realizam, respectivamente, a *escolha uma aresta* $g = \cup\{x, y\} \in E$ *adjacente à* f na segunda linha do Algoritmo 3.1 e o *teste de simplicidade* realizado na sexta linha do mesmo algoritmo.

Da mesma maneira que as arestas, os objetos da classe `Vertex` também possuem um inteiro sem sinal como identificador. A diferença é que cada vértice armazena também uma lista, na forma de um `vector` da STL, de referências para as arestas incidentes em si, conforme exibido na Figura 7.2. Os métodos `getDegree`() e `getIncidentEdge`() são

métodos auxiliares utilizados, respectivamente, na validação do grafo recebido como entrada pelo algoritmo de emparelhamento (para garantir que ele é cúbico) e na remoção de arestas com base apenas no identificador (algo necessário em certos momentos do cálculo do emparelhamento perfeito, quando não há referências diretas para determinada aresta). Já o tipo `EdgeIterator` é simplesmente uma renomeação do iterator sobre o arranjo de apontadores para objetos `Edge`, sendo possível criar um laço para iterar sobre a lista de arestas incidentes utilizando os métodos `incidentEdgesBegin`() e `incidentEdgesEnd`().

```
class Edge {
    public:
        Edge(                                    ) ;
        Edge( unsigned id, Vertex* v, Vertex* w ) ;
       ~Edge(                                    ) ;

        unsigned getID                 (             ) ;
        Vertex*  getV                  (             ) ;
        Vertex*  getW                  (             ) ;
        Edge*    getRandomAdjacentEdge (             ) ;
        bool     isSimple              (             ) ;
        void     setID                 ( unsigned id ) ;
        void     setV                  ( Vertex*  v  ) ;
        void     setW                  ( Vertex*  w  ) ;

    private:
        unsigned _id ;
        Vertex*  _v  ;
        Vertex*  _w  ;

};
```

Figura 7.1: A classe `Edge`.

Por fim, a classe `Graph` é a responsável por gerenciar as arestas e armazenar os vértices. Tanto vértices quanto arestas são alocados no *heap* de memória, criados com `new` e desalocados com `delete`. Durante o período de existência delas, as referências para as arestas são armazenadas nas listas de incidência em seus dois vértices extremos. As referências para os vértices, por sua vez, são armazenadas em um *container* `map` da STL dentro de um objeto da classe `Graph`, associando cada identificador de vértice com um apontador para o mesmo (veja a Figura 7.3). O método `getRandomEdge`(), utilizado para escolher a aresta f que será passada como entrada para o Algoritmo 3.1, seleciona uma aresta qualquer dentre as que estão no grafo. Já `degreeEquals`() e `isBridgeless`() são utilizados para validação da entrada, determinando se o grafo em questão é cúbico e sem

pontes.

```
class Vertex {
    public:
        Vertex(                ) ;
        Vertex( unsigned id    ) ;
       ~Vertex(                ) ;

        unsigned      getID                (                ) ;
        unsigned      getDegree            (                ) ;
        Edge*         getIncidentEdge      ( unsigned id    ) ;
        EdgeIterator  incidentEdgesBegin   (                ) ;
        EdgeIterator  incidentEdgesEnd     (                ) ;
        void          setID                ( unsigned id    ) ;
        void          addIncidentEdge      ( Edge*    e     ) ;
        void          removeIncidentEdge   ( Edge*    e     ) ;

    private:
        unsigned                  _id             ;
        std::vector< Edge* >      _incidentEdges  ;

};
```

Figura 7.2: A classe `Vertex`.

Outro método que merece uma explicação especial é `getEdgeCount()`, que simplesmente retorna o número de inserções de arestas desde a criação do grafo (valor armazenado no campo `_edgeCount`, que inicia em zero e vai sendo incrementado). Ele é importante para garantir que não haverá conflitos na atribuição de identificadores distintos às arestas conforme elas vão sendo reduzidas e novas arestas acrescentadas durante o cálculo do emparelhamento.

7.1.2 A árvore splay

A implementação das árvores splay segue, em grande medida, o padrão para árvores binárias numa linguagem orientada a objetos. Isto é, existe uma classe para representar nós (`SplayNode`) e outra para representar árvores (`SplayTree`). A classe `SplayNode` possui quatro campos, sendo uma chave identificadora (também um inteiro sem sinal) e três apontadores: um para o pai, outro para o filho esquerdo e o último para o filho direito de um nó (veja a Figura 7.4). Sua interface pública apresenta apenas construtores, destrutores e métodos *get* e *set* para esses campos e, portanto, não será explicada em maiores detalhes.

Já a classe `SplayTree`, cuja interface pode ser vista na Figura 7.5, foi, de longe, a mais

```
class Graph {
    public:
        Graph() ;
       ~Graph() ;

        Vertex*  getVertex     ( unsigned id ) ;
        Edge*    getEdge       ( unsigned id ) ;
        Edge*    getRandomEdge (             ) ;
        bool     degreeEquals  ( unsigned d  ) ;
        bool     isBridgeless  (             ) ;
        unsigned size          (             ) ;
        unsigned getEdgeCount  (             ) ;
        Vertex*  insertVertex  ( unsigned id ) ;
        void     removeVertex  ( Vertex*  x  ) ;
        void     removeEdge    ( Edge*    e  ) ;
        void     clear         (             ) ;
        Edge*    insertEdge    (unsigned id, Vertex* v, Vertex* w) ;
        void     removeEdge    (unsigned id, Vertex* v, Vertex* w) ;

    private:
        std::map< unsigned, Vertex* > _vertices  ;
        unsigned                      _edgeCount ;
};
```

Figura 7.3: A classe `Graph`.

fácil de implementar entre todas as outras classes neste projeto. O código segue à risca uma descrição alternativa dada em (SLEATOR; TARJAN, 1985), que é apenas ligeiramente diferente do que foi exposto no Capítulo 4 (pois não utiliza diretamente JOIN() ou SPLIT()) e, de acordo com Sleator e Tarjan, possui uma complexidade amortizada ligeiramente melhor do que a versão original. Para implementar INSERT(i), o primeiro passo é realizar uma busca por i, da mesma forma que em ACCESS(). O apontador nulo encontrado na busca (já que a chave i não deve existir na árvore) é substituído por um novo nó com chave i, que é então levado até a raiz com uma chamada à SPLAY(i). De modo semelhante, REMOVE(i) inicia com uma busca pelo nó, x, com chave i, que é substituído pela junção (utilizando uma implementação *inline* de JOIN()) de suas subárvores esquerda e direita. O procedimento é completado realizando um SPLAY() no nó y, pai de x. A operação ACCESS() é a única que não muda, sendo implementada exatamente como descrita no Capítulo 4.

```
class SplayNode {
    public:
        SplayNode(              ) ;
        SplayNode(unsigned key) ;
       ~SplayNode(              ) ;

        unsigned      getKey    (                    ) ;
        SplayNode*    getParent (                    ) ;
        SplayNode*    getLeft   (                    ) ;
        SplayNode*    getRight  (                    ) ;
        void          setKey    ( unsigned   key     ) ;
        void          setParent ( SplayNode* parent  ) ;
        void          setLeft   ( SplayNode* left    ) ;
        void          setRight  ( SplayNode* right   ) ;
        void          clear     (                    ) ;

    private:
        unsigned      _key    ;
        SplayNode*    _parent ;
        SplayNode*    _left   ;
        SplayNode*    _right  ;
};
```

Figura 7.4: A classe `SplayNode`.

7.1.3 A árvore ST

As classes `SplayNode` e `SplayTree`, embora tenham sido o ponto de partida da fase de codificação do projeto (já que são a estrutura mais interna), não são utilizadas diretamente na implementação de `STNode` e `STTree`. Gerenciar as árvores apenas a partir de ponteiros para uma raiz, conforme feito pelo campo `_root` de `SplayTree`, é algo que não se encaixa muito bem com as árvores ST. Então, ao invés de utilizar algum tipo de herança, optou-se por reescrever o código fazendo as alterações necessárias. Veja as Figuras 7.6 e 7.7. De maneira semelhante aos objetos da classe `SplayNode`, os nós numa árvore ST possuem um campo chave e apontadores para seu pai, filho esquerdo e filho direito. Mas, uma das primeiras diferenças que se nota é a presença de apontadores para dois filhos do meio. Para explicar sua existência é preciso voltar ao Algoritmo 6.3.

No pseudocódigo de REPLACE() (veja o Algoritmo 6.3), há uma busca a ser realizada por todos os nós da árvore virtual T_u. Essa busca, para ser completa, exige uma maneira de "descer" pelas arestas tracejadas da árvore virtual. Caso contrário, apenas a subárvore sólida contendo a raiz (virtual) de T_u seria considerada. Foram encontradas duas soluções para esse problema durante o projeto. A primeira, que pode ser chamada de *ingênua*,

consiste em armazenar em cada nó de uma árvore ST uma lista não-ordenada de referências para seus filhos do meio. Seja k o maior grau de um nó em uma árvore real. A necessidade de manter essas referências consistentes conforme a árvore virtual fosse mudando faria com que o tempo amortizado das operações primitivas ROTATE() e SPLICE() pudesse estar em $\Theta(k)$. Isso faria com que a complexidade amortizada de uma operação VIRTUALSPLAY(), e consequentemente o tempo das operações principais, pudesse estar em $\Theta(k \cdot \lg n)$.

```
class SplayTree {
    public:
        SplayTree() ;
       ~SplayTree() ;

        SplayNode* access      ( unsigned    i ) ;
        void       insert      ( unsigned    i ) ;
        void       remove      ( unsigned    i ) ;
        void       clear       (               ) ;

    private:
        void       splay       ( SplayNode*  x ) ;
        void       rotateLeft  ( SplayNode*& y ) ;
        void       rotateRight ( SplayNode*& y ) ;

        SplayNode* _root ;
};
```

Figura 7.5: A classe `SplayTree`.

A segunda alternativa, exposta em (RADZIK, 1998), consiste em utilizar árvores virtuais para representar apenas árvores dinâmicas nas quais cada nó possui, no máximo, dois filhos. Essas árvores, denominadas *árvores dinâmicas restritas*, são utilizadas internamente para suportar a interface de árvore dinâmica (*irrestrita*), conforme descrito na Seção 5.1, mantendo a complexidade $\mathcal{O}(\lg n)$. O problema com essa abordagem, além da adição de mais um nível na hierarquia de composição de classes, é a complexidade da descrição em (RADZIK, 1998), tanto no tocante às árvores irrestritas quanto à sua utilização na implementação das árvores irrestritas. Outro motivo que favoreceu a aplicação da abordagem ingênua, além da sua simplicidade, foi o fato de todos os grafos de interesse para este trabalho serem *cúbicos*, o que implica em $k \leq 3$ para qualquer nó em qualquer caso. Em outras palavras, a abordagem ingênua, neste caso, não afeta a complexidade das operações.

Uma primeira implementação desse mecanismo utilizou um `vector` da STL para ar-

```
class STNode {
    public:
        STNode(              ) ;
        STNode(unsigned key) ;
       ~STNode(              ) ;

        unsigned      getKey           () const ;
        STNode*       getParent        () const ;
        STNode*       getLeft          () const ;
        STNode*       getRight         () const ;
        STNode*       getFirstMiddle   () const ;
        STNode*       getSecondMiddle  () const ;
        bool          isReversed       () const ;
        bool          isVisited        () const ;
        bool          isVirtualRoot    () const ;
        bool          isSolidRoot      () const ;
        int           getSize          () const ;

        void          setKey           ( unsigned     key      ) ;
        void          setParent        ( STNode* parent   ) ;
        void          setLeft          ( STNode* left     ) ;
        void          setRight         ( STNode* right    ) ;
        void          addMiddle        ( STNode* middle   ) ;
        void          removeMiddle     ( STNode* middle   ) ;
        void          setReversed      ( bool         reversed ) ;
        void          setVisited       ( bool         visited  ) ;
        void          setSize          ( int          size     ) ;
        void          addSize          ( int          size     ) ;
        void          flipReversed     (                       ) ;
        void          swapChildren     (                       ) ;
        void          clear            (                       ) ;

    private:
        unsigned      _key      ;
        STNode*       _parent   ;
        STNode*       _left     ;
        STNode*       _right    ;
        STNode*       _first    ;
        STNode*       _second   ;
        bool          _reversed ;
        bool          _visited  ;
        int           _size     ;
};
```

Figura 7.6: A classe `STNode`.

mazenar apontadores para os filhos do meio de um nó. Porém, durante a execução de testes, verificou-se algo que já era esperado: em nenhum momento algum nó teve mais do que 2 filhos do meio. O balanceamento que ocorre na árvore acaba sempre fazendo com que os outros apontadores (pai, filho esquerdo e filho direito) sejam utilizados. Isso foi visto como uma excelente oportunidade de aprimoramento do código, pois um *container* da STL que armazena apenas 2 elementos certamente é um desperdício de memória, além de existir um *overhead* considerável devido aos iteradores. Finalmente, é por esse motivo que existem apenas duas referências para os filhos do meio, *hard-coded* como campos de cada nó.

O campo `_reversed` corresponde exatamente ao bit de inversão conforme descrito ao final da Seção 5.2, então ele não será detalhado novamente aqui. Mas, antes de explicar os dois campos remanescentes, é preciso discutir um pouco mais a estrutura da classe `STTree`. O construtor, que recebe como argumento um inteiro, n, sem sinal representando quantos nós serão inseridos na floresta, reserva espaço em um arranjo unidimensional (`_forest`) que armazenará os n nós. Como cada nó possui uma chave identificadora que vai de 0 até $n-1$, o nó com chave i ficará armazenado em `_forest[i]` (a i-ésima posição do arranjo `_forest`).

A constante especial,

`MAX_UNSIGNED = std::numeric_limits<unsigned int>::max()`,

é utilizada para indicar um índice inválido. O valor n é armazenado no campo `_capacity`, que representa a capacidade total do arranjo `_forest`, enquanto o campo `_numberOfNodes` representa o número de nós que estão atualmente na floresta. Inicialmente zero, o número de nós é incrementado com cada chamada a `addNode`() (que é a MakeTree()) e decrementado em `removeNode`().

Quase todas as funções recebem, ao invés de apontadores para os nós, valores inteiros sem sinal que representam um índice no arranjo `_forest`. Note que há uma correspondência entre esses valores e os identificadores associados aos objetos da classe `Vertex`, o que facilita a comunicação com a estrutura de conectividade dinâmica (que faz a ponte entre a árvore ST e o grafo). Nos casos em que é necessário realmente trabalhar com o apontador, basta utilizar a função `getNode`(), que recebe como parâmetro um identificador e retorna um apontador para aquela posição no arranjo `_forest`. Tendo esclarecido esses pontos principais é possível agora explicar o campo `_visited` da classe `STNode`.

O campo `_visited` é uma *flag* booleana utilizada dentro da função `findLCA`(), que encontra o ancestral comum mais próximo (isto é, mais distante da raiz) de dois nós.

O código, que pode ser visto na Figura 7.8, consiste em subir pela árvore (real) a partir dois dois nós passados como parâmetros com chamadas sucessivas à função `findParent`(). Cada nó encontrado durante a busca é marcado como visitado e um apontador para o mesmo é armazenado em um vetor (para ser desmarcado ao final da busca). Se, em algum momento na subida, a função `findParent`() retornar um nó já visitado, o ancestral comum mais próximo foi encontrado. Caso contrário, não existe ancestral comum e os nós estão em árvores (reais) diferentes.

```
class STTree {
  public:
    STTree(unsigned n) ;
   ~STTree(          ) ;

    void          addNode       ( unsigned v                ) ;
    void          removeNode    ( unsigned v                ) ;
    void          link          ( unsigned v, unsigned w    ) ;
    void          cut           ( unsigned v                ) ;
    void          evert         ( unsigned v                ) ;
    STNode*       getNode       ( unsigned v                ) ;
    STNode*       findVirtualRoot ( unsigned v              ) ;
    unsigned      findRealRoot  ( unsigned v                ) ;
    unsigned      findParent    ( unsigned v                ) ;
    unsigned      findLCA       ( unsigned v, unsigned w    ) ;
    bool          connected     ( unsigned v, unsigned w    ) ;
    unsigned      size          (                           ) ;
    void          clear         (                           ) ;

  private:
    void          unreverse     ( STNode*& v ) ;
    void          rotate        ( STNode*& v ) ;
    void          splice        ( STNode*& v ) ;
    void          switchBit     ( STNode*& v ) ;
    void          virtualSplay  ( STNode*& v ) ;

    STNode*       _forest          ;
    unsigned      _numberOfNodes   ;
    unsigned      _capacity        ;
};
```

Figura 7.7: A classe `STTree`.

O último campo da classe `STNode` que resta explicar é o contador inteiro `_size`, que representa o número de elementos na subárvore virtual enraizada no nó. Inicialmente, esta informação havia sido ignorada, pois como visto na Seção 6.2.1, é irrelevante — do ponto de vista da complexidade da busca por uma aresta de reserva — em qual das duas

árvores, T_v ou T_w, a busca é realizada. Porém, conforme se observou experimentalmente, a diferença de tamanho entre T_v e T_w pode ser muito grande. Logo, a opção pela busca na árvore com o menor número de nós, pode fazer com que o código execute bem mais rapidamente.

Visto que é crucial ter alguma maneira de determinar qual das duas árvores virtuais é a menor, sem afetar a complexidade amortizada das demais operações, cogitou-se armazenar em cada nó da árvore virtual um campo, `_size`, que armazena o número de nós da árvore enraizada no nó. Para manter este campo atualizado, deve-se inserir algumas linhas de código nas funções Rotate(), Link() e Cut(). Estas são as únicas operações que podem mudar o número de nós da subárvore enraizada em um certo nó. As demais funções também podem, mas fazem isso de maneira indireta, usando uma ou mais das três funções acima. Logo, pode-se restringir a atualização do campo `_size` ao código de Rotate(), Link() e Cut().

Sejam S(v) e S'(v) o tamanho da subárvore enraizada no nó v antes e após a rotação. Seguindo a nomenclatura da Figura 4.1 (em uma rotação à direita), apenas duas atualizações são necessárias: S'(x) = S(y) e S'(y) = S(y) + S(Raiz(B)) − S(x), onde Raiz(B) se refere ao nó raiz da subárvore direita, B, de x antes da rotação. Numa rotação à esquerda, basta trocar x por y e considerar B a subárvore esquerda de y. Logo, as atualizações do campo `_size` podem ser realizadas em tempo constante em uma única execução de Rotate(). Isto implica que o número (amortizado) de atualizações deste campo em uma única chamada a VirtualSplay() está em $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore contendo o nó passado como argumento para VirtualSplay(). Logo, a complexidade amortizada de VirtualSplay(v, w) não muda, assim como não muda a complexidade das demais funções que usam VirtualSplay(), mas não modificam o campo `_size` diretamente. Este é precisamente o caso de Evert(), FindParent() e FindRoot().

Numa operação Link(v, w), executa-se VirtualSplay() para v e w antes de atribuir w como pai de v (veja o Algoritmo 5.6). Como VirtualSplay(v) faz com que v se torne a raiz de sua árvore virtual, a atualização do campo `_size` só precisa ser feita para o nó w e consiste em somar o valor atual deste campo ao valor do campo `_size` do nó v. Obviamente, outras atualizações do campo `_size` ocorreram quando da execução de VirtualSplay(), mas o número amortizado de atualizações está em $\mathcal{O}(\lg n)$, onde n é a soma do número de nós das árvores virtuais contendo v e w antes de Link(v, w) ser executada. Portanto, a complexidade amortizada da operação Link() também não muda.

Numa operação CUT(v), assume-se que o pai, w, de v existe. O primeiro passo de CUT(v) é executar VIRTUALSPLAY(v), o que faz com que v se torne a raiz da árvore virtual e que w seja seu filho direito (veja o Algoritmo 5.4). Logo, a atualização do campo `_size` só precisa ser feita para o nó v e consiste em subtrair do valor atual deste campo o valor do campo `_size` do nó w. Obviamente, outras atualizações do campo `_size` ocorreram quando da execução de VIRTUALSPLAY(v), mas o número amortizado de atualizações está em $\mathcal{O}(\lg n)$, onde n é a soma do número de nós da árvore virtual contendo v antes de LINK(v, w) ser executada. Então, a complexidade amortizada da operação CUT() também não muda.

Finalmente, observe que se um nó x é a raiz de sua árvore virtual, então o campo `_size` de x contém o número de nós da árvore virtual e da *árvore real* que o contém. Esta observação é a chave para se determinar qual das duas árvores, T_v ou T_w, contém o menor número de nós na execução de REPLACE(). Isto porque imediatemente após a execução da operação CUT() que removeu a aresta $\{v, w\}$ no Algoritmo 6.2, os nós v e w são as raízes de suas respectivas árvores virtuais. Logo, a determinação de qual das duas árvores é a menor pode ser feita em tempo constante, obtendo e comparando os valores do campo `_size` de v e w.

7.1.4 A estrutura de conectividade dinâmica

A estrutura de conectividade dinâmica, ao contrário da árvore splay, foi a mais difícil de implementar entre todas as outras estruturas envolvidas no projeto. Sua descrição original foi a mais superficial e incompleta, e a que mais omitiu os detalhes relativos à implementação. Embora os autores afirmem em (HOLM; LICHTENBERG; THORUP, 2001) que qualquer árvore dinâmica pode ser utilizada na implementação da estrutura — desde que cada operação da interface tenha tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós das árvores envolvidas na operação — a única implementação conhecida da estrutura foi realizada com a ET *tree* (IYER JR. et al., 2001).

Já a implementação desenvolvida neste trabalho, a classe `DynamicConnectivityDS` (veja a Figura 7.9), utiliza internamente um objeto da classe `STTree` para representar a floresta geradora de um grafo, além de um arranjo especial (do tipo registro `ConnectivityData`) para armazenar referências para as arestas de reserva e de árvore incidentes em cada nó.

Novamente fazendo uso do fato do grafo ser sempre cúbico, ao invés de armazenar uma lista de arestas, cada elemento do arranjo `_edges` (implementado como uma *struct*) possui

apenas cinco apontadores (três para arestas de árvore e dois para arestas de reserva) e está organizado de maneira a corresponder a um objeto `STNode` da seguinte forma: o registro na posição i do arranjo `_edges` referencia as arestas de árvore e reserva do nó associado à chave i.

Uma alternativa ao projeto de arquitetura acima poderia ser criar uma nova classe `DynamicNode`, que herdaria de `STNode`, e fazer a estrutura de conectividade dinâmica herdar de `STTree`. Embora mais elegante, essa solução não foi elaborada por falta de tempo. Mas ela não está descartada como uma alteração futura, no intuito de melhorar a qualidade do código a ser disponibilizado publicamente.

Note que é necessário armazenar também as arestas de árvore (além das arestas de reserva), pois, como são permitidas arestas múltiplas no grafo, até três arestas diferentes podem possuir os mesmos extremos e uma chamada a FindParent() não seria capaz de identificar unicamente uma aresta como de árvore ou de reserva. Logo, só porque os dois extremos de uma aresta estão conectados na árvore geradora não significa que ela corresponda ao Link() que os uniu. Isso seria um problema, por exemplo, no método `remove`(), no qual é necessário saber se a aresta está armazenada apenas como uma referência ou se é preciso um Cut() que a remova da árvore geradora.

As principais diferenças entre a classe `DynamicConnectivityDS` e o que está exposto no Capítulo 6 são: o acréscimo de métodos para inserir e remover vértices (necessários, respectivamente, na inicialização da classe e durante as reduções de aresta), funções privadas para manipular os campos do tipo registro (para facilitar o uso deles, já que se escolheu não implementá-los como classes) e uma replicação dos métodos `connected`() e `findLCA`() da classe `STTree` (de maneira a torná-los visíveis publicamente para o algoritmo de emparelhamento perfeito). Com exceção dessas alterações, há apenas alguns detalhes a acrescentar sobre o método de substituição de arestas, já que `insert`() e `remove`() seguem de maneira direta o que está exposto nos Algoritmos 6.1 e 6.2.

Em `replace`(), escolheu-se implementar a busca como uma *busca em largura* (utilizando o *container* `queue` da STL) a partir da raiz da menor árvore virtual entre as duas a considerar, T_v e T_w. É nesse momento que são utilizados os apontadores para filhos do meio, de maneira a poder fazer uma busca completa por todas as subárvores sólidas. Para saber qual das duas raízes virtuais deve ser enfileirada (i.e., qual é a menor árvore), basta chamar a função `getSize`() em ambas e comparar os valores retornados. Além disso, ao invés de um laço interno para percorrer as listas de arestas de substituição como no Algoritmo 6.3, basta fazer no máximo dois testes para cada nó, um para cada aresta de

```
unsigned STTree::findLCA(unsigned v, unsigned w) {

    unsigned lca = MAX_UNSIGNED;

    std::vector<STNode*> nodesToUnvisit;

    unsigned pV = v;
    unsigned pW = w;
    while (pV != MAX_UNSIGNED || pW != MAX_UNSIGNED) {
        if (pV != MAX_UNSIGNED) {
            STNode* n = getNode(pV);
            if (n->isVisited()) {
                lca = n->getKey();
                break;
            } else {
                n->setVisited(true);
            }
            nodesToUnvisit.push_back(n);
            pV = findParent(pV);
        }

        if (pW != MAX_UNSIGNED) {
            STNode* n = getNode(pW);
            if (n->isVisited()) {
                lca = n->getKey();
                break;
            } else {
                n->setVisited(true);
            }
            nodesToUnvisit.push_back(n);
            pW = findParent(pW);
        }
    }

    for (unsigned i = 0; i < nodesToUnvisit.size() ; ++i) {
        nodesToUnvisit[i]->setVisited(false);
    }

    return lca;
}
```

Figura 7.8: Código da funcão `findLCA`().

reserva armazenada no registro `ConnectivityData` correspondente.

```
class DynamicConnectivityDS {
    public:
        DynamicConnectivityDS(unsigned n) ;
       ~DynamicConnectivityDS(          ) ;

        void     insert    ( Vertex* v                ) ;
        void     insert    ( Edge*   e                ) ;
        void     remove    ( Vertex* v                ) ;
        void     remove    ( Edge*   e                ) ;
        bool     connected ( Vertex* v, Vertex*   w ) ;
        unsigned findLCA   ( unsigned v, unsigned w ) ;
        unsigned size      (                        ) ;
        void     clear     (                        ) ;

    private:
        void     replace           ( Edge* e              ) ;
        void     addTreeEdge       ( Edge* e, unsigned v ) ;
        void     addNonTreeEdge    ( Edge* e, unsigned v ) ;
        void     removeTreeEdge    ( Edge* e, unsigned v ) ;
        void     removeNonTreeEdge ( Edge* e, unsigned v ) ;
        bool     containsTreeEdge  ( Edge* e, unsigned v ) ;

        STTree             _spanningForest ;
        ConnectivityData* _edges           ;
};
```

Figura 7.9: A classe `DynamicConnectivityDS`.

7.1.5 O algoritmo de emparelhamento perfeito

Finalmente, fazendo uso de todo o código mostrado anteriormente, é possível implementar o algoritmo de Diks e Stanczyk para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. Porém, uma implementação direta do algoritmo — conforme descrito no Capítulo 3 — seria praticamente inútil, visto que o *overhead* da recursão certamente ocasionaria estouro da pilha até mesmo em instâncias moderadas do problema em estudo.

Para lidar com essa limitação, foi escrita uma versão iterativa do mesmo algoritmo, que utiliza uma pilha de elementos do tipo registro (`stack<ReductionData>`) para armazenar explicitamente informações sobre cada redução realizada (veja a Figura 7.10). Embora não seja praticável mostrar o código finalizado aqui (mesmo sem comentários, o seu tamanho excede 400 linhas), o pseudocódigo da versão iterativa pode ser visto no Algoritmo 7.4.

Algoritmo 7.4 DIKS-STANCZYK(G, D)

Entrada: Um grafo cúbico e sem pontes, $G = (V, E, st)$
Entrada: Estrutura de conectividade, D, inicializada com os elementos de G
Saída: Um emparelhamento perfeito M de G

```
Crie uma pilha, S, para armazenar informações sobre cada redução
Escolha uma aresta aleatória, e, de G para não fazer parte de M
enquanto D.SIZE() ≥ 2 faça
    Escolha uma aresta r = {x, y} adjacente à e
    Crie uma variável, rd, do tipo registro ReductionData
    se r é uma aresta simples então
        Sejam x1, x2 os vizinhos de x diferentes de y
        x3, x4 os vizinhos de y diferentes de x e as arestas
        e1 = {x1, x} , e2 = {x2, x} , e3 = {x3, y} , e4 = {x4, y}
        Remova e1, e2, e3, e4 e r de D e G
        Determine a redução apropriada conforme descrito na Seção 3.3
        e ← a aresta reduzida adjacente à antiga e
        Armazene em rd informações sobre a redução realizada
    senão
        Seja w o vizinho de x diferente de y
        z o vizinho de y diferente de x e as arestas
        f = {w, x} e h = {y, z}
        Remova f, h e os dois lados de r de D e G
        e ← a única aresta reduzida inserida, ligando x a y
        Armazene em rd informações sobre a redução realizada
    fim se
    Empilhe rd em S
fim enquanto
# Caso base, 2 vértices e 3 arestas
M ← ∅
Insira em M uma das duas arestas adjacentes à e
enquanto S não estiver vazia faça
    rd ← desempilhe o topo de S
    se a aresta de redução, r = {x, y}, era simples então
        Coloque de volta as arestas removidas
        se uma das arestas reduzidas faz parte de M então
            Remova-a de M
            Insira em M as duas arestas que ligam seus extremos a x e a y
        senão
            Insira a aresta de redução, r, em M
        fim se
        Remova de G as arestas reduzidas
    senão
        Remova de G a única aresta reduzida
        Coloque de volta as arestas removidas
        Adicione a M um dos lados da aresta dupla r
    fim se
fim enquanto
return M
```

```
typedef struct ReductionData {
    ReductionType type   ;
    Vertex*       x      ;
    Vertex*       y      ;
    Vertex*       x1     ;
    Vertex*       x2     ;
    Vertex*       x3     ;
    Vertex*       x4     ;
    unsigned      xyID   ;
    unsigned      e1ID   ;
    unsigned      e2ID   ;
    unsigned      e3ID   ;
    unsigned      e4ID   ;
    unsigned      enmID  ;
    unsigned      rE1ID  ;
    unsigned      rE2ID  ;
} ReductionData;
```

Figura 7.10: O registro `ReductionData`.

7.2 Resultados

Para avaliar a implementação descrita acima foi utilizada uma base de dados, de acesso público e gratuito, do Projeto AIM@SHAPE[1], que consiste em um conjunto de superfícies simpliciais (ou triangulações) representativo do tipo encontrado em trabalhos da área de processamento geométrico. Por serem superfícies simpliciais, tais superfícies possuem como grafo dual um grafo conexo, simples, cúbico e sem pontes (conforme descrito na Seção 1.1). Neste texto, como de hábito em Processamento Geométrico, cada superfície simplicial será chamada de *modelo*. Para os experimentos descritos abaixo foram escolhidos 12 (doze) modelos dentre os disponíveis no repositório, cujas características podem ser vistas na Tabela 7.1. As Figuras 7.11, 7.12, 7.13 e 7.14 mostram quatro dos doze modelos utilizados.

A criação dos grafos correspondentes a cada modelo a partir dos arquivos `OFF` que os contêm foi feita utilizando uma biblioteca escrita em C++, previamente desenvolvida pelo Prof. Marcelo Siqueira (orientador deste trabalho), que faz a leitura do arquivo e cria uma representação da malha na memória do computador. É importante ressaltar que é criado um vértice no grafo para cada *face* do modelo. Logo, nesse caso, o tamanho da entrada corresponde ao número de faces da malha. O experimento, repetido 20 vezes para

[1] http://www.aimatshape.net/

cada malha, consistiu em inicializar o grafo e a estrutura de conectividade dinâmica a partir dessa representação e, em seguida, calcular um emparelhamento perfeito. Nessas 20 execuções foram medidos os tempos de inicialização, de cálculo do emparelhamento perfeito (tempo total), para reduzir o grafo ao caso base do algoritmo (parte do tempo total de emparelhamento) e o tempo para desfazer as reduções (a outra parte do tempo total).

Tabela 7.1: Nome e característica de Euler-Poincaré dos modelos usados nos experimentos.

Modelo	#Vértices	#Faces	#Arestas	#Genus
Armadillo	171889	343774	515661	0
Bimba	14839	29674	44511	0
Botijo	20000	40016	60024	5
Cow	4315	8626	12939	0
Dinosaur	56194	112384	168576	0
Eros	197230	394456	591684	0
Fandisk	6475	12946	19419	0
Fertility	19996	40000	60000	4
Gargoyle	97130	194256	291384	0
Knot	31545	63090	94635	1
Santa	75444	150880	226320	3
Teeth	116038	232072	348108	0

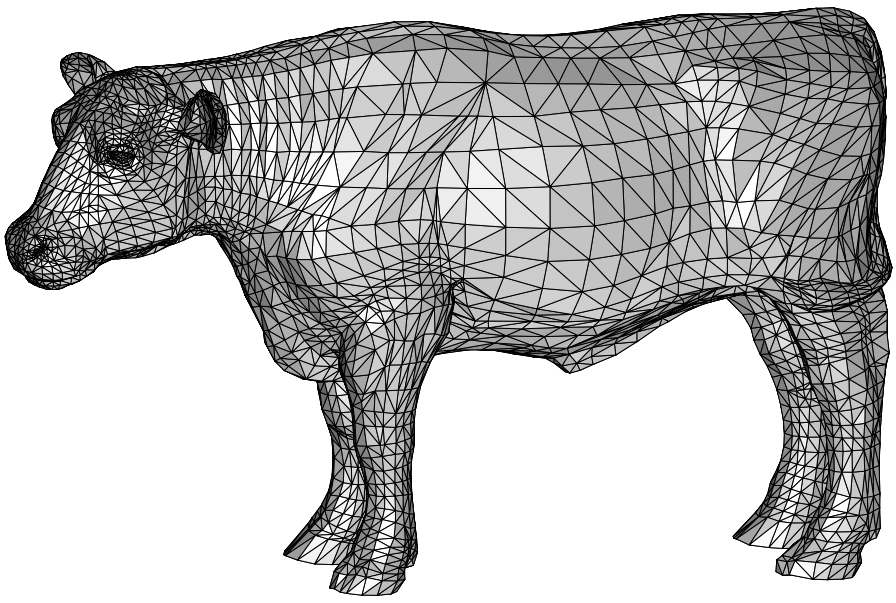


Figura 7.11: A malha *Cow*.

Para cada uma dessas medidas de tempo, observaram-se os valores mínimo, médio e máximo, além da variância e do desvio padrão. Todos os testes foram realizados num

Macbook Pro com processador Intel Core 2 Duo 2,4GHz, 4GB de memória RAM e sistema operacional Max OS X Mountain Lion. É importante ressaltar que, devido ao pouco tempo disponível para a elaboração deste texto, não foi possível mostrar aqui uma comparação com outros algoritmos de emparelhamento. Os resultados obtidos são mostrados nas Tabelas 7.2-7.5.

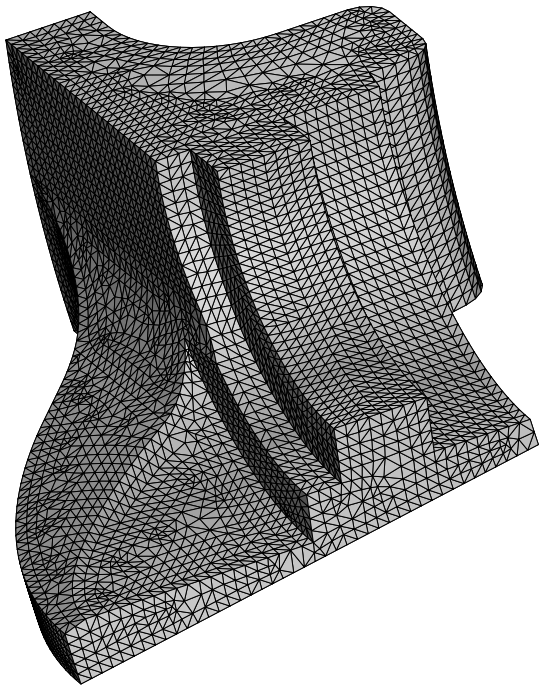


Figura 7.12: A malha *Fandisk*.

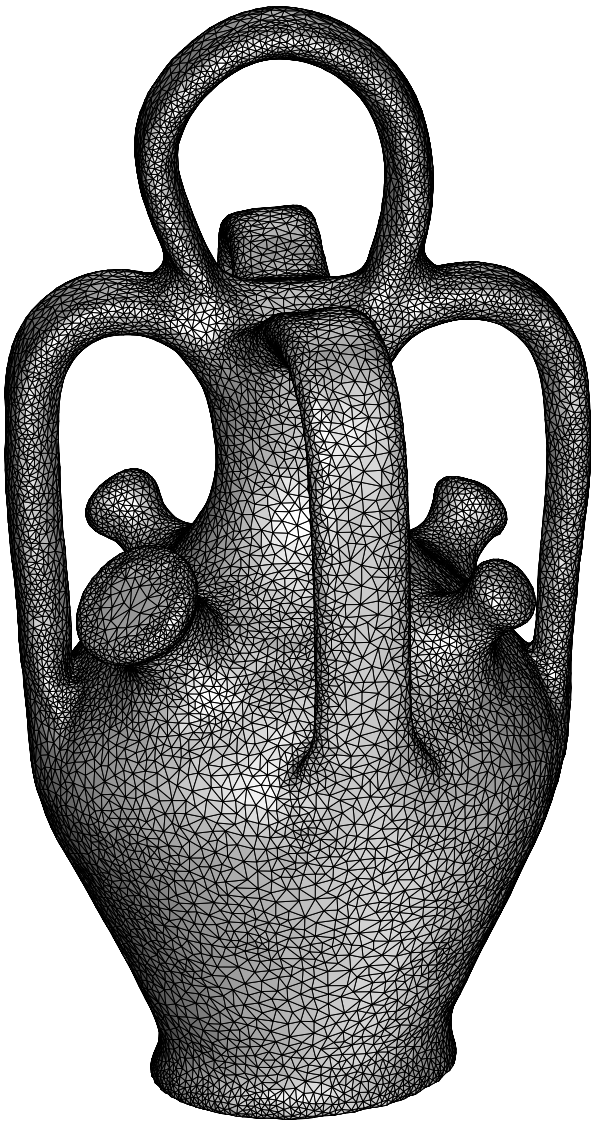


Figura 7.13: A malha *Botijo*.

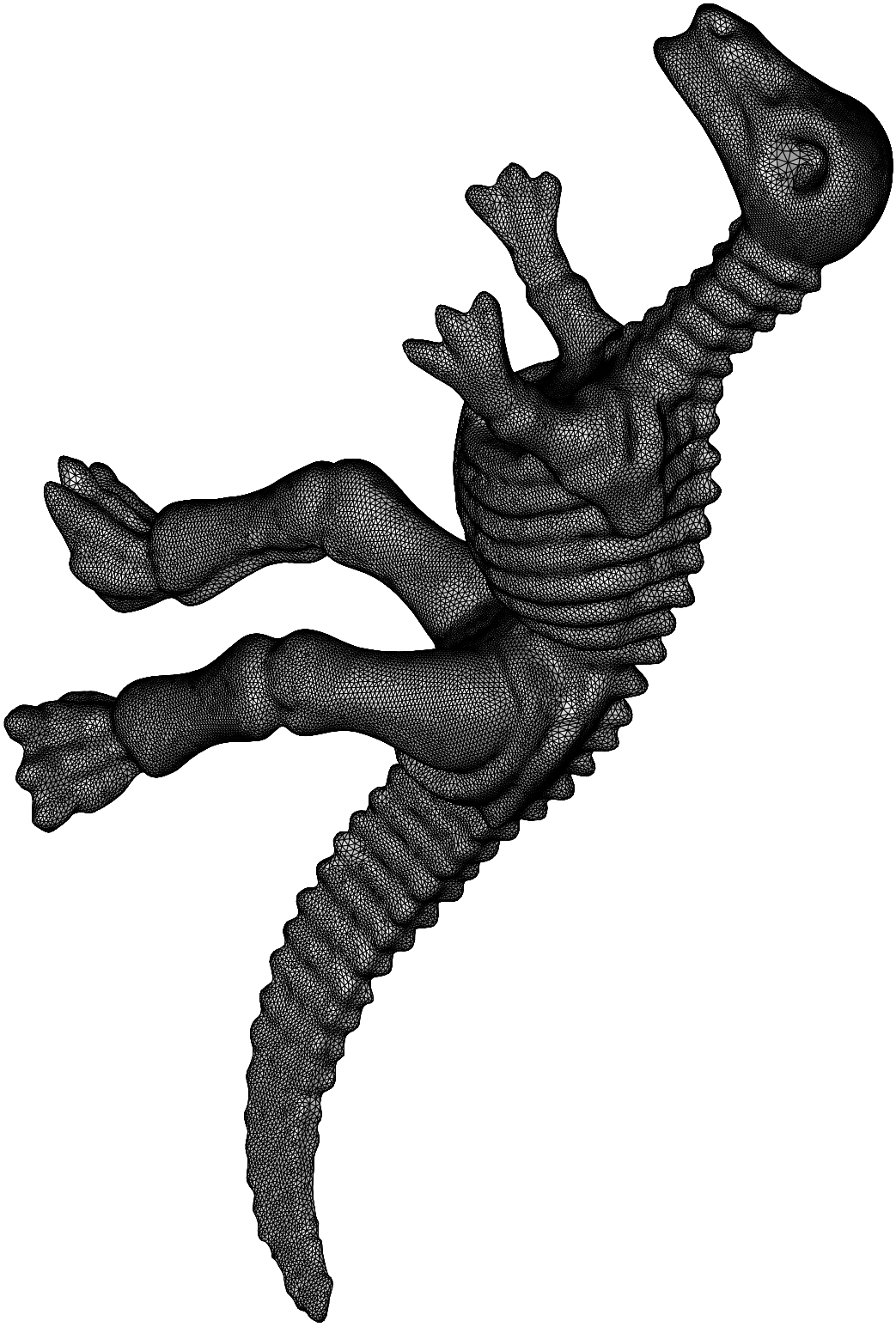


Figura 7.14: A malha *Dinosaur*.

Tabela 7.2: Tempo (em segundos) para construir o grafo e inicializar a estrutura de conectividade dinâmica a partir da malha de entrada (isto é, antes de calcular o emparelhamento perfeito).

Modelo	Mínimo	Máximo	Média	σ
Armadillo	5,57038	5,59757	5,58713	0,00653735
Bimba	0,331232	0,33978	0,332397	0,00203113
Botijo	0,502249	0,503209	0,502769	0,000237335
Cow	0,087424	0,088709	0,0876521	0,00025589
Dinosaur	1,58325	1,59496	1,59231	0,00240958
Eros	5,98449	6,00685	5,99317	0,00434926
Fandisk	0,131298	0,132928	0,131747	0,000331995
Fertility	0,437474	0,443674	0,438866	0,00153515
Gargoyle	3,48404	3,54728	3,49872	0,0129723
Knot	0,807628	0,835246	0,810141	0,00603438
Santa	2,12125	2,21942	2,14529	0,0215898
Teeth	3,61169	3,66581	3,62429	0,0108219

Tabela 7.3: Tempo total (em segundos) para calcular um emparelhamento perfeito.

Modelo	Mínimo	Máximo	Média	σ
Armadillo	50,0057	50,1319	50,0644	0,0259477
Bimba	1,29396	1,31243	1,29542	0,00394174
Botijo	2,08573	2,09017	2,0868	0,000931165
Cow	0,22795	0,228426	0,228132	0,000163787
Dinosaur	14,2788	14,325	14,2883	0,00876367
Eros	84,6421	84,7856	84,6829	0,0376005
Fandisk	0,521258	0,521987	0,521507	0,000153632
Fertility	2,15721	2,17711	2,15902	0,0042818
Gargoyle	22,8122	22,8681	22,8316	0,0138764
Knot	6,50248	6,51539	6,50454	0,00284721
Santa	16,9375	16,9893	16,9434	0,0108636
Teeth	46,0823	46,1542	46,1006	0,0224643

Tabela 7.4: Tempo (em segundos) para reduzir o grafo até atingir o caso base do algoritmo.

Modelo	Mínimo	Máximo	Média	σ
Armadillo	49,5824	49,694	49,6281	0,0239864
Bimba	1,26954	1,28626	1,27106	0,00352965
Botijo	2,04921	2,05377	2,05003	0,00108104
Cow	0,22209	0,222536	0,222257	0,000149661
Dinosaur	14,1568	14,1983	14,1626	0,00836521
Eros	84,1531	84,2894	84,1919	0,0361519
Fandisk	0,511509	0,512263	0,511752	0,000165822
Fertility	2,12276	2,14059	2,1243	0,00385843
Gargoyle	22,5684	22,6173	22,5855	0,0128929
Knot	6,44457	6,4584	6,44649	0,00305148
Santa	16,777	16,8278	16,7822	0,0107044
Teeth	45,8166	45,885	45,8335	0,0217157

Tabela 7.5: Tempo (em segundos) para desfazer todas as reduções.

Modelo	Mínimo	Máximo	Média	σ
Armadillo	0,423231	0,442963	0,436253	0,00462593
Bimba	0,024157	0,026159	0,0243467	0,000428048
Botijo	0,035839	0,036975	0,0367563	0,0002655
Cow	0,005783	0,005944	0,00582205	0,0000332979
Dinosaur	0,122005	0,127043	0,125624	0,00115023
Eros	0,484742	0,496199	0,491048	0,00323578
Fandisk	0,009641	0,009763	0,0096978	0,0000366232
Fertility	0,03438	0,036508	0,0347137	0,000429866
Gargoyle	0,240671	0,25077	0,246145	0,00253707
Knot	0,056979	0,058349	0,0580335	0,000271386
Santa	0,158094	0,164486	0,161149	0,00157681
Teeth	0,26207	0,273321	0,267034	0,00249161

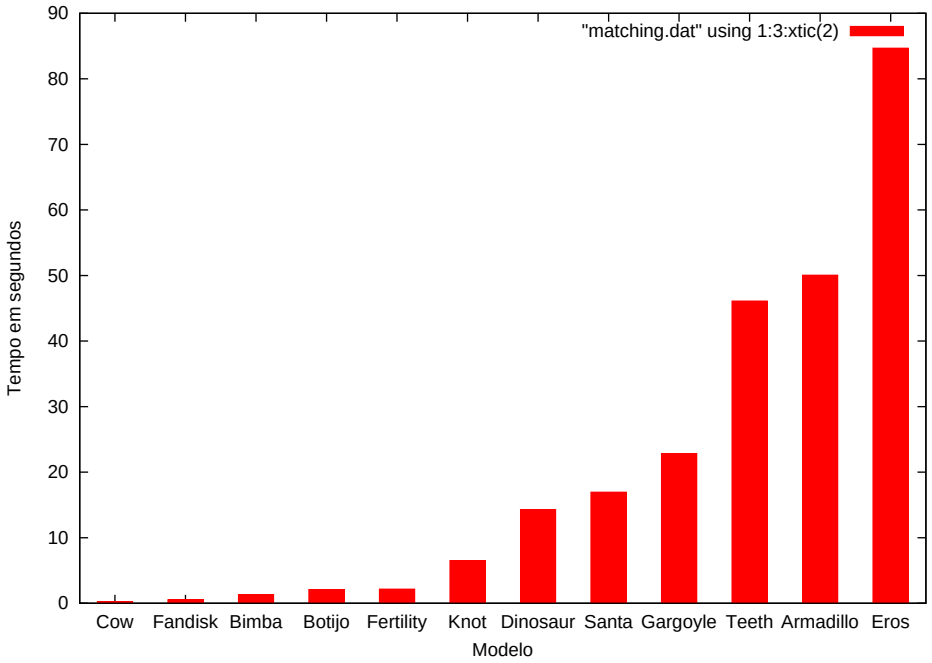


Figura 7.15: Tempos (médios) para calcular um emparelhamento perfeito em cada malha.

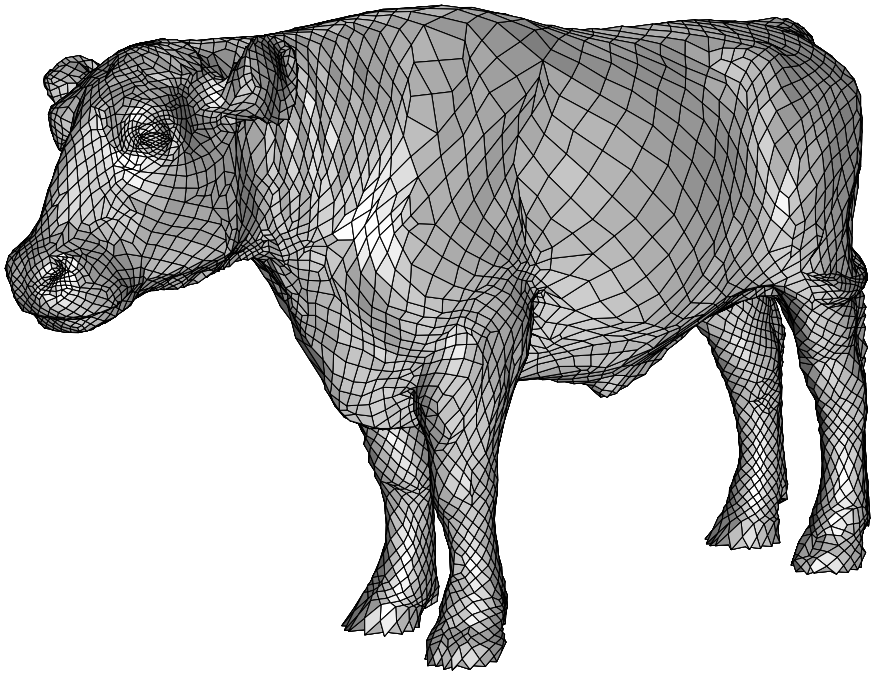


Figura 7.16: A malha *Cow* após a execução do algoritmo de emparelhamento.

7.3 Discussão

Como não foi possível comparar esta implementação com outros algoritmos de emparelhamento perfeito, é difícil analisar seu comportamento real em relação ao esperado. O que se pode observar com certeza nos resultados é que o tempo para calcular um emparelhamento é quase totalmente dominado pelas reduções de aresta que levam o grafo até o caso base (com uma aresta tripla). A criação de um emparelhamento perfeito a partir da primeira aresta escolhida nesse momento e sua extensão ao grafo original, desfazendo as reduções cujas informações ficam armazenadas numa pilha, toma apenas uma fração minúscula do tempo total. E isso é verdade mesmo em malhas consideravelmente grandes, tais como Armadillo e Eros, que possuem, respectivamente, quase 350.000 e quase 400.000 faces.

É importante ressaltar que é durante as reduções de aresta que a estrutura de dados para conectividade dinâmica é modificada por inserções e remoções de aresta, além das eventuais consultas para se determinar que redução deve ser utilizada. Após o caso base ser atingido, as reduções corretas para se reconstruir o grafo original já estão armazenadas na pilha. O uso de uma pilha se deve ao fato da implementação do algoritmo ser não-recursiva, pois uma versão recursiva seria inviável, mesmo para malhas com centenas de faces apenas.

Apesar de o experimento não ter sido repetido muitas vezes para cada modelo, pode-se concluir que não há grandes variações durante execuções diferentes do código (como indicado pelo desvio-padrão significativamente próximo de zero). A minúscula diferença entre valores mínimos e máximos pode ser atribuída à alocação de memória em endereços ligeiramente diferentes a cada execução e às pequenas variações que ocorrem internamente no sistema operacional.

8 Conclusão

Este capítulo conclui a presente monografia. A Seção 8.1 apresenta um pequeno resumo dos aspectos mais importantes do trabalho. A Seção 8.2 revela as principais dificuldades enfrentadas ao longo do desenvolvimento do trabalho. Por fim, a Seção 8.3 destaca algumas possíveis extensões para trabalhos futuros e descreve alguns problemas de pesquisa.

8.1 Sobre o trabalho desenvolvido

Este texto descreveu um trabalho de conclusão de curso de graduação no qual foi implementado um algoritmo, proposto por Diks e Stanczyk (DIKS; STANCZYK, 2010), para calcular emparelhamentos perfeitos em grafos cúbicos e sem pontes. Para esta classe de grafos, o algoritmo em questão possui a menor complexidade de tempo entre todos os algoritmos conhecidos: $\mathcal{O}(n \lg^2 n)$, onde n é o número de vértices do grafo. Esta complexidade só é obtida graças à utilização do TAD HLT em (HOLM; LICHTENBERG; THORUP, 2001) descrito no Capítulo 6, utilizado no teste de conectividade dinâmica e implementado com árvores dinâmicas, tais como a árvore ST (SLEATOR; TARJAN, 1983, 1985) descrita no Capítulo 5.

Após rever a terminologia necessária e revisar a literatura sobre emparelhamento em grafos, os tipos e estruturas de dados utilizados neste trabalho foram estudadas em detalhes. Em particular, deu-se atenção especial à análise da complexidade amortizada das operações que atuam sobre as estruturas de dados estudadas, que foi realizada com o método do potencial (descrito no Apêndice B). A descrição fornecida teve como objetivo fornecer provas mais claras e mais didáticas do que aquelas que os próprios autores apresentam nos artigos correspondentes. Por fim foi descrita toda a etapa de codificação das estruturas de dados e do algoritmo em (DIKS; STANCZYK, 2010), além do mecanismo de testes utilizado para medir, experimentalmente, a eficiência da implementação realizada aqui.

8.2 Dificuldades Encontradas

Um bom número de dificuldades foi encontrado ao longo do desenvolvimento deste trabalho. Grande parte dessas dificuldades se deveu, diretamente, à falta de clareza dos principais artigos que serviram de base para o trabalho. Detalhes cruciais para a implementação do que está sendo descrito foram muitas vezes omitidos, descartados como secundários. Por exemplo, a operação EVERT() só é necessária na árvore ST devido a um caso particular da inserção de aresta na estrutura de conectividade dinâmica (o caso em que nenhum dos dois extremos é raiz de uma árvore, então não é possível fazer um LINK() antes de fazer com que pelo menos um dos extremos em questão se torne raíz de sua árvore). Esse problema só foi descoberto numa fase avançada do trabalho, quando já existia uma implementação da árvore ST, e seria muito difícil resolvê-lo não fosse a explicação minuciosa presente em (RADZIK, 1998) e a ajuda inestimável do Dr. Renato Werneck[1].

Um problema similar, cuja solução também foi retirada de (RADZIK, 1998), ocorreu com o procedimento REPLACE(), no qual há necessidade de um mecanismo que permita realizar uma busca em largura a partir da raiz virtual de uma árvore ST (isto é, "descer" pelos filhos do meio de cada nó). Outra dificuldade ocorreu durante as tentativas de integrar os código-fontes do projeto às bibliotecas LEMON e Boost, ambas resultando em falhas e tempo perdido. A grande complexidade envolvida na utilização dessas bibliotecas foi o que motivou a criação de classes *ad-hoc* mais simples para representar grafos, vértices e arestas. Finalmente, a falta de familiaridade do autor deste trabalho com os conceitos de análise de complexidade amortizada dificultou muito a escrita de certos trechos da monografia, embora o trabalho tenha sido uma excelente oportunidade para aprender o assunto.

8.3 Trabalhos Futuros

A primeira extensão natural deste trabalho é implementar a solução, que não foi descrita aqui, e que acrescenta à interface da árvore ST as funções para obter a "próxima" aresta de reserva durante a busca em tempo amortizado $\mathcal{O}(\lg n)$, onde n é o número de nós da árvore. Esta modificação não afeta a complexidade amortizada das demais operações da interface, o que faz com que a implementação do algoritmo em (DIKS; STANCZYK, 2010)

[1] Pesquisador da Microsoft, nos EUA, que cedeu sua própria implementação da árvore ST para este autor.

possa realmente ter a complexidade amortizada, $\mathcal{O}(v(G) \cdot \lg^2 v(G))$, prevista no referido artigo.

Uma outra tarefa futura e que depende da primeira é realizar uma avaliação experimental do algoritmo mais completa e rigorosa do que a que foi apresentada no Capítulo 7. O ideal seria traçar um perfil do tempo de execução do algoritmo de emparelhamento perfeito para uma sequência de modelos com números de faces que se diferenciem em ordens de magnitude (isto é, 10, 10^2, 10^3, 10^4, 10^5, ..., faces). Isto pode ser feito com o refinamento da malha de um toro, por exemplo. Depois disso, dever-se-ía comparar o tempo de execução da implementação realizada com aqueles obtidos por alguma implementação do algoritmo de Micali e Vazirani (MICALI; VAZIRANI, 1980). Finalmente, dever-se-ía também comparar o tempo de execução da implementação realizada com aqueles obtidos por alguma outra implementação do mesmo algoritmo com base em outro tipo de árvore dinâmica, tal como a árvore ET. Uma opção para comparação é a dada em (IYER JR. et al., 2001).

Embora não faça parte do escopo desta monografia, existe também um problema de pesquisa no contexto deste trabalho que poderia ser objeto de estudo numa pós-graduação. Ele trata do desenvolvimento de um novo algoritmo para resolver o problema de se converter uma triangulação de uma superfície sem bordo em $\mathbb{R}^3$ em uma quadrilaterização da mesma superfície.

Conforme discutido no Capítulo 1, o referido problema pode ser convertido em um problema de emparelhamento em grafos. Em particular, define-se o grafo dual de uma triangulação como sendo o grafo que possui um vértice para cada triângulo da triangulação e uma aresta conectando dois vértices se, e somente se, os triângulos duais desses vértices compartilham uma aresta. Dado um emparelhamento perfeito do grafo dual, pode-se obter uma quadrilaterização da mesma superfície através da eliminação da aresta da triangulação que é comum a cada par de triângulos cujos os vértices duais estão emparelhados no grafo.

No artigo (BIEDL et al., 2001), além do Algoritmo descrito no Capítulo 3, há também um algoritmo para calcular emparelhamentos perfeitos em grafos cúbicos, sem pontes e *planares*. Este algoritmo possui complexidade de tempo linear no número n de triângulos da triangulação e é ótimo. No entanto, o fato do grafo ser planar é crucial. Como imersões em superfícies são, de certa forma, uma generalização de imersões planares para espaços mais complexos (isto é, as superfícies), seria bastante interessante investigar a possibilidade de estender o algoritmo em (BIEDL et al., 2001) para grafos duais não planares de tal

forma que a complexidade de tempo do algoritmo resultante seja assintoticamente menor do que $\mathcal{O}(n \lg^2 n)$, que é a complexidade do algoritmo apresentado por Diks e Stanczyk em (DIKS; STANCZYK, 2010). Essa possibilidade é particularmente digna de investigação, pois os elementos da solução apresentada por Biedl et al. em (BIEDL et al., 2001) são bem diferentes daqueles apresentados por Diks e Stancyk em (DIKS; STANCZYK, 2010). Mais especificamente, a solução deste se apóia na utilização de uma estrutura de dados especial, enquanto a solução daquele faz uso de elementos topológicos e geométricos da triangulação.

Referências

ACAR, U. A. et al. Dynamizing static algorithms, with applications to dynamic trees and history independence. In: *Proceedings of the 15th annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004. (SODA'04), p. 531–540.

ACAR, U. A.; BLELLOCH, G. E.; VITTES, J. L. An experimental analysis of change propagation in dynamic trees. In: *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*. [S.l.: s.n.], 2005. p. 41–54.

ADELSON-VELSKII, G. M.; LANDIS, E. M. An algorithm for the organization of information. *Soviet Mathematics Doklady*, n. 3, p. 1259–1262, 1962.

AHUJA, R. K.; ORLIN, J. B.; TARJAN, R. E. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, n. 18, p. 939–954, 1989.

ALSTRUP, S. et al. Minimizing diameters of dynamic trees. In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*. [S.l.]: Springer-Verlag, 1997. p. 270–280.

ALSTRUP, S. et al. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, ACM, New York, NY, USA, v. 2, n. 1, p. 243–264, 2005.

ATALAY, F. B.; RAMASWAMI, S.; XU, D. Quadrilateral meshes with bounded minimum angle. In: *Proceedings of the 17th International Meshing Roundtable (IMR)*. Pittsburgh, PA, USA: [s.n.], 2008. p. 73–91.

BAYER, R.; MCCREIGHT, E. Organization of large ordered indexes. *Acta Informatica*, n. 1, p. 173–189, 1972.

BERN, M.; EPPSTEIN, D. Computing euclidean geometry. In: HWANG, F.; DU, D.-Z. (Ed.). [S.l.]: World Scientific, 1992. cap. Mesh generation and optimal triangulation, p. 23–90.

BERN, M.; EPPSTEIN, D. Quadrilateral meshing by circle packing. In: *Proceedings of the 6th International Meshing Roundtable (IMR)*. Park City, Utah, USA: [s.n.], 1997. p. 7–19.

BIEDL, T. C. et al. Efficient algorithms for Petersen's matching theorem. *Journal of Algorithms*, Duluth, MN, USA, v. 38, n. 1, p. 110–134, 2001.

BLACKER, T. Paving: a new approach to automated quadrilateral mesh generation. *International Journal For Numerical Methods in Engineering*, v. 32, n. 4, p. 811–847, 1991.

BLUM, N. A new approach to maximum matching in general graphs. In: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*. New York, NY, USA: Springer-Verlag, 1990. p. 586–597.

BONDY, J. A.; MURTY, U. S. R. *Graph theory*. New York, NY, USA: Springer, 2010. (Graduate Texts in Mathematics, v. 244).

BOSE, P.; KIRKPATRICK, D.; LI, Z. Proceedings of the 8th canadian conference on computational geometry (cccg). In: _____. [S.l.]: Carleton University Press, 1996. (International Informatics Series, v. 5), cap. Efficient algorithms for guarding or illuminating the surface of a polyhedral terrain, p. 217–222.

BOSE, P. et al. Guarding polyhedral terrains. *Computational Geometry: Theory and Applications*, v. 7, n. 3, p. 173–195, 1997.

BROWN, M. R.; TARJAN, R. E. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, v. 9, n. 3, p. 594–614, 1980.

CARVALHO, P. C. P.; VELHO, L.; GOMES, J. de M. *Spatial decompositions: theory and practice*. Rio de Janeiro, RJ, Brasil: [s.n.], 1992.

CLARK, J.; HOLTON, D. A. *A first look at graph theory*. First. Singapore: World Scientific Publishing Co. Pte. Ltd, 1991.

COLE, R.; SHARIR, M. Visibility problems for polyhedral terrains. *Journal of Symbolic Computation*, v. 7, n. 1, p. 11–30, 1989.

CORMEM, T. H. et al. *Introduction to Algorithms*. Third. Cambridge, Massachusetts, USA: The MIT Press, 2009.

DIESTEL, R. *Graph Theory*. Second. New York, NY, USA: Springer, 2000.

DIKS, K.; STANCZYK, P. Perfect matching for biconnected cubic graphs in $\mathcal{O}(n\,log^2\,n)$ time. In: *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'10)*. Germany: Springer-Verlag, 2010. p. 321–333.

EDMONDS, J. Paths, trees, and flowers. *Canadian Journal of Mathematics*, v. 17, p. 449–467, 1965.

EPPSTEIN, D. et al. Sparsification – a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, ACM, New York, NY, USA, v. 44, n. 5, p. 669–696, September 1997.

EVERETT, H.; RIVERA-CAMPO, E. Edge guarding polyhedral terrains. *Computational Geometry: Theory and Applications*, v. 7, n. 3, p. 201–203, 1997.

FREDERICKSON, G. N. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, v. 4, n. 14, p. 781–798, 1985.

FREDERICKSON, G. N. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, v. 2, n. 26, p. 484–538, 1997.

FREDERICKSON, G. N. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, Academic Press, Inc., Duluth, MN, USA, v. 1, n. 24, p. 37–65, 1997.

FRINK JR., O. A proof of Petersen's theorem. *The Annals of Mathematics*, v. 27, n. 4, p. 491–493, June 1926.

GABOW, H. N. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, v. 23, n. 2, p. 221–234, 1976.

GABOW, H. N.; TARJAN, R. E. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, v. 30, n. 2, p. 209–221, 1985.

GABOW, H. N.; TARJAN, R. E. Faster scaling algorithms for general graph matching problems. *Journal of the ACM*, v. 38, n. 4, p. 815–853, 1991.

GALLIER, J. *Discrete Mathematics*. New York, NY, USA: Springer, 2011.

GOLDBERG, A. V.; TARDOS, E.; TARJAN, R. E. Network flow algorithms. In: KORTE, B. et al. (Ed.). *Paths, Flows and VLSI-layout*. [S.l.]: Springer-Verlag, 1990. p. 101–164.

GOLDBERG, A. V.; TARJAN, R. E. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, n. 36, p. 388–397, 1989.

GOLDBERG, A. V.; TARJAN, R. E. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, n. 15, p. 430–466, 1990.

GUIBAS, L.; STOLFI, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, v. 4, n. 2, p. 74–123, 1985.

GUIBAS, L. J.; SEDGEWICK, R. A dichromatic framework for balanced trees. In: *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Ann Arbor, MI, USA: IEEE, 1978. p. 8–21.

HENZINGER, M. R.; KING, V. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, ACM, New York, NY, USA, v. 46, n. 4, p. 502–516, July 1999.

HENZINGER, M. R.; THORUP, M. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures & Algorithms*, Wiley Subscription Services, Inc., A Wiley Company, v. 11, n. 4, p. 369–379, 1997.

HOLM, J.; LICHTENBERG, K. D.; THORUP, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. *Journal of the ACM*, New York, NY, USA, v. 48, n. 4, p. 723–760, 2001.

HONSBERGER, R. *Mathematical Gems II*. EUA: The Mathematical Association of America, 1976. (Dolciani Mathematical Expositions, v. 2).

HU, T. C.; TUCKER, A. C. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, n. 37, p. 246–256, 1979.

HUDDLESTON, S.; MEHLHORN, K. Robust balancing in b-trees. In: DEUSSEN, P. (Ed.). *Theoretical Computer Science*. [S.l.]: Springer, 1981, (Lecture Notes in Computer Science, v. 104). p. 234–244. ISBN 978-3-540-10576-3.

HUDDLESTON, S.; MEHLHORN, K. A new data structure for representing sorted lists. *Acta Informatica*, v. 17, n. 2, p. 157–184, 1982.

IYER JR., R. D. et al. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *Journal of Experimental Algorithmics*, ACM, New York, NY, USA, v. 6, n. 4, December 2001.

JOE, B. Quadrilateral mesh generation in polygonal regions. *Computer-Aided Design*, v. 27, n. 3, p. 209–222, 1995.

JOHNSON, C. *Numerical solution of partial differential equations by the finite element method*. Mineola, NY, USA: Dover Publications Inc., 2009.

KNUTH, D. E. Optimum binary search trees. *Acta Informatica*, n. 1, p. 14–25, 1971.

KÖNIG, D. *Theory of Finite and Infinite Graphs*. Ann Arbor, MI, EUA: Birkhäuser Boston, 1990. Tradução do original em alemão, *Theorie der endlichen und undelichen Graphen*, publicado pela *Akademische Verlagsgesellschaft*, Leipzig, Alemanha, em 1936.

KOSARAJU, S. R. Localized search in sorted lists. In: *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1981. (STOC'81), p. 62–69.

LAWLER, E. L. *Combinatorial Optimization: Networks and Matroids*. Mineola, NY, USA: Dover Publications, Inc., 2001.

LOVÁSZ, L.; PLUMMER, M. D. *Matching Theory*. First. Amsterdam: North-Holland Publishing Company, 1986.

MAIER, D.; SALVETER, S. C. Hysterical b-trees. *Information Processing Letters*, v. 12, n. 4, p. 199–202, 1981.

MALANTHARA, A.; GERSTLE, W. Comparative study of unstructured meshes made of triangles and quadrilaterals. In: *Proceedings of the 6th International Meshing Roundtable (IMR)*. Park City, Utah, USA: [s.n.], 1997. p. 437–447.

MICALI, S.; VAZIRANI, V. V. An $\mathcal{O}(|e| \cdot \sqrt{|V|})$ algorithm for finding maximum matchings in general graphs. In: *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Washington, DC, USA: IEEE Computer Society, 1980. p. 17–27.

MILTERSEN, P. B. et al. Complexity models for incremental computation. *Theoretical Computer Science*, v. 130, n. 1, p. 203–236, August 1994.

MITCHELL, W. F. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of Computational and Applied Mathematics*, v. 36, n. 1, p. 65–78, 1991.

MULDER, H. M. Julius Petersen's theory of regular graphs. *Discrete Mathematics*, v. 100, n. 1-3, p. 157–175, May 1992.

NIEVERGELT, J.; REINGOLD, E. M. Binary search trees of bounded balance. *SIAM Journal on Computing*, n. 2, p. 33–43, 1973.

O'ROURKE, J. *Art Gallery Theorems and Algorithms*. [S.l.]: Oxford University Press, 1987.

OWEN, S. J. et al. Q-morph: an indirect approach to advancing front quad meshing. *International Journal for Numerical Methods in Engineering*, v. 44, n. 9, p. 1317–1340, 1999.

PĂTRAŞCU, M.; DEMAINE, E. D. Lower bounds for dynamic connectivity. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 2004. (STOC'04), p. 546–553.

PETERSEN, J. P. C. Die theorie der regulären graphs (the theory of regular graphs). *Acta Mathematica*, v. 15, n. 1, p. 193–220, 1891.

RADZIK, T. Implementation of dynamic trees with in-subtree operations. *Journal of Experimental Algorithmics*, ACM, New York, NY, USA, v. 3, September 1998. ISSN 1084-6654.

RAMASWAMI, S. et al. Constrained quadrilateral meshes of bounded size. *International Journal of Computational Geometry & Applications*, v. 15, n. 1, p. 55–98, 2005.

SCHUMAKER, L. L. Triangulations in cagd. *IEEE Computer Graphics and Applications*, v. 13, n. 1, p. 47–52, 1993.

SHERMER, T. C. Recent results in art galleries. *Proceedings of the IEEE*, v. 80, n. 9, p. 1384–1399, 1992.

SLEATOR, D. D.; TARJAN, R. E. A data structure for dynamic trees. *Journal of Computer and System Sciences*, Orlando, FL, USA, v. 26, n. 3, p. 362–391, 1983.

SLEATOR, D. D.; TARJAN, R. E. Self-adjusting binary search trees. *Journal of the ACM*, New York, NY, USA, v. 32, n. 3, p. 652–686, 1985.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 3. ed. São Paulo, SP, Brasil: Pearson Prentice Hall, 2009.

TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, SIAM, v. 1, n. 2, p. 146–160, 1972.

TARJAN, R. E. Amortized computional complexity. *SIAM Journal on Algebraic and Discrete Methods*, Philadelphia, PA, USA, v. 6, n. 2, p. 306–318, 1985.

TARJAN, R. E. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming*, n. 78, p. 169–177, 1997.

TARJAN, R. E.; WERNECK, R. F. Self-adjusting top trees. In: *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005. (SODA'05), p. 813–822.

TARJAN, R. E.; WERNECK, R. F. Dynamic trees in practice. *Journal of Experimental Algorithmics*, ACM, New York, NY, USA, v. 14, p. 5:4.5–5:4.23, January 2010. ISSN 1084-6654.

THORUP, M. Near-optimal fully-dynamic graph connectivity. In: *Proceedings of the 32rd Annual ACM Symposium on Theory of computing*. New York, NY, USA: ACM, 2000. (STOC'00), p. 343–350.

TUTTE, W. T. The factorization of linear graphs. *Journal of the London Mathematical Society*, v. 22, p. 107–111, 1947.

URRUTIA, J. Handbook on computational geometry. In: _____. [S.l.]: Elsevier Science B.V., 2000. (Lecture Notes in Economics and Mathematical Systems, v. 22), cap. Art Gallery and illumination problems, p. 973–1027.

VISWANATH, N.; SHIMADA, K.; ITOH, T. Quadrilateral meshing with anisotropy and directionality control via close packing of rectangular cells. In: *Proceedings of the 9th International Meshing Roundtable (IMR)*. New Orleans, Louisiana, USA: [s.n.], 2000. p. 217–225.

WATT, A. *3D Computer Graphics*. Essex, England: Addison-Wesley Publishing Company Inc., 1999.

WERNECK, R. F. *Design and Analysis of Data Structures for Dynamic Trees*. Tese (Doutorado) — Princeton University, 2006.

WEST, D. B. *Introduction to Graph Theory*. Patparganj, Delhi, India: Pearson Education, 2002.

WULFF-NILSEN, C. Faster deterministic fully-dynamic grapg connectivity. In: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2013. (SODA'13), p. 1757–1769.

APÊNDICE A – A prova de Frink

Este apêndice detalha a prova dada por Orrin Frink Jr. em (FRINK JR., 1926) para o Teorema de Petersen (PETERSEN, 1891). A prova, como apresentada em (FRINK JR., 1926) contém alguns erros. Uma versão corrigida da prova e enriquecida com mais detalhes pode ser encontrada em um livro clássico de Teoria dos Grafos escrito por Dénes König em 1936 e traduzido para inglês em 1990 (KÖNIG, 1990). A apresentação dada aqui é baseada na versão traduzida para inglês do livro de König. Tentou-se, na medida do possível, adequar a terminologia da época àquela utilizada nos demais capítulos do texto. Além disso, procurou-se organizar os vários e longos passos da demonstração de uma outra forma e enriquecer a descrição dos passos mais cruciais com a adição de mais detalhes e figuras.

A.1 Considerações iniciais

O teorema de Petersen estabelece que *todo grafo cúbico e com, no máximo, duas folhas contém um* 1-*fator* (PETERSEN, 1891; MULDER, 1992). Uma *folha* é uma componente conexa que não possui nenhuma ponte e que surge após a remoção de alguma ponte do grafo. Por definição, as arestas de um 1-fator (veja Definição 2.1.7) consistem em um emparelhamento perfeito no grafo. Um caso particular do teorema de Petersen é o que realmente importa para o presente trabalho: *todo grafo cúbico e sem pontes contém um* 1-*fator*. De forma equivalente, *todo grafo cúbico e sem pontes admite um emparelhamento perfeito*.

Observe que os enunciados dos casos geral e particular do teorema de Petersen não restringem o grafo a um grafo simples, embora um grafo cúbico que possua um laço tenha de conter, necessariamente, uma ponte, pois o grau de todo vértice é igual a 3 e um laço conta como duas arestas incidentes sobre um mesmo vértice. De agora em diante, ao encontrar o termo grafo, deve-se entendê-lo como um grafo no sentido mais geral (isto é, possivelmente com laços e aresta paralelas). Caso se deseje considerar o grafo em questão

como um grafo simples, mencionar-se-á isso de forma explícita e como um caso excepcional.

Há várias provas para o teorema de Petersen. Uma das mais simples se baseia em um teorema provado por Tutte em 1947, para o qual Lovász forneceu uma das mais belas demonstrações já vistas, de acordo com Honsberger (HONSBERGER, 1976). O problema com esta e a maioria das provas que se conhecem é que elas se baseiam em argumentos não construtivos (por exemplo, contradição). Como consequência, não se pode derivar, diretamente, um algoritmo a partir delas para construir um emparelhamento perfeito no grafo.

A prova que se apresenta aqui, dada por Frink (FRINK JR., 1926), também se vale de contradição, mas ela possui um passo construtivo — denominado de *redução* — que permitiu o desenvolvimento de dois algoritmos recursivos para a construção de emparelhamentos perfeitos em grafos cúbicos e sem pontes (BIEDL et al., 2001; DIKS; STANCZYK, 2010), como visto no Capítulo 3. Esses algoritmos são mais eficientes do que o algoritmo mais eficiente que se conhece, atualmente, para construir emparelhamentos de cardinalidade máxima em grafos arbitrários, ou seja, não necessariamente cúbicos ou sem pontes (MICALI; VAZIRANI, 1980).

A.2 O teorema de Frink

A prova de Frink para o teorema de Petersen depende de vários resultados intermediários e de um teorema atribuído ao próprio Frink. Esta seção descreve esses resultados e o teorema.

Proposição A.2.1. Seja G um grafo conexo. Se $e = \{u, v\}$ é uma ponte de G, então o grafo, $G - e$, resultante da remoção de e, possui exatamente duas componentes conexas.

Demonstração. De acordo com a definição de ponte (veja Definição 2.1.13), o grafo $G - e$ não pode ser conexo. Logo, o grafo $G - e$ possui, pelos menos, duas componentes conexas. Note que uma delas contém v e outra contém u, pois se v e u estivessem na mesma componente, então haveria um caminho, P, nesta componente, conectando u a v e que não contém e. Mas, se a aresta e for adicionada a P, obtém-se um ciclo simples, $P + e$, em G contendo e, o que contradiz a hipótese de e ser uma ponte de G. Afirma-se, agora, que $G - e$ possui exatamente duas componentes. Como G é conexo, se w é qualquer vértice de G, então há um caminho, $(w, e_1, x_1, \ldots, x_{n-1}, e_n, v)$, em G entre w e v. Se $e_n \neq e$, então

este caminho também é um caminho em $G-e$. Mas, se $e_n = e$, então $x_{n-1} = u$ e, portanto, $(w, e_1, x_1, \ldots, x_{n-2}, e_{n-1}, x_{n-1})$ é um caminho em $G - e$ conectando w a $u = x_{n-1}$. Isto implica que, para todo vértice w em $G - e$, há um caminho conectando w a v ou w a u. Logo, todo vértice, w, de G pertence à componente de $G - e$ que contém v ou à que contém u, o que, por sua vez, implica que $G - e$ possui exatamente duas componentes conexas. □

As duas componentes conexas, U_1 e U_2, às quais se refere a Proposição A.2.1 são conhecidas como as duas *margens da ponte* $e = \{v, u\}$ *com respeito a* G. Diz-se que uma margem *se origina a partir do vértice* v *da ponte*, enquanto a outra *se origina a partir do vértice* u. Essa denominação também se aplica ao caso em que G não é conexo. Isto é, se a ponte e estiver contida em uma componente conexa, U, de G, então as margens, U_1 e U_2, de e com respeito a U são também denominadas de margens de e com respeito a G. Uma margem é uma *folha* do grafo G se, e somente se, ela não contém nenhuma ponte de G.

A Figura A.1 ilustra as noções de margem e folha.

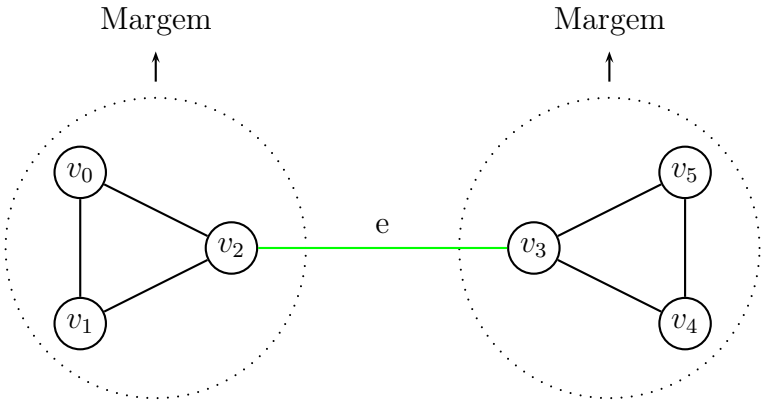


Figura A.1: As duas margens de uma ponte, e, com respeito ao grafo conexo.

Os dois seguintes corolários são consequências imediatas da Proposição A.2.1:

Corolário A.2.2. Seja G um grafo. Então, se a ponte $e = \{v, u\}$ de G não está contida em um caminho, em G, que conecta um vértice w de G a v, então os vértices e arestas deste caminho devem pertencer, obrigatoriamente, à margem de e que se origina a partir de v.

Corolário A.2.3. Seja G um grafo conexo. Então, todo subgrafo conexo de G que não contém uma ponte e de G é um subgrafo de uma das margens de e com respeito ao grafo

G.

A seguinte proposição se vale do fato de G ser finito, ao contrário dos resultados anteriores. Lembre-se que, por hipótese, todos os grafos em questão neste trabalho são finitos.

Proposição A.2.4. Seja G um grafo. Se a aresta e é uma ponte de G, então cada uma das duas margens de e com respeito ao grafo G contém um subgrafo que é uma folha de G.

Demonstração. Se G não possui nenhuma ponte, então a afirmação é trivialmente verdadeira. Logo, assuma que G possui pelo menos uma ponte, e. Assuma, neste momente, que G é conexo. Seja U qualquer uma das duas margens de e com respeito a G. Se U é uma folha, escolhe-se a outra margem como U. Se, ainda assim, U é uma folha, a afirmação é verdadeira. Então, assuma que U contém uma ponte, e_1, de G. Seja U_1 a margem de e_1 com respeito a G que não contém e. Afirma-se que U_1 é um subgrafo de U. Por definição, U_1 é um subgrafo conexo de G. Logo, há um caminho, em U_1, de cada um de seus vértices a um dos extremos de e_1. Se e_1 for adicionada a este caminho, obtém-se um caminho, P. Pelo Corolário A.2.3, todos os vértices e arestas de P devem pertencer, inteiramente, a uma margem da ponte e. Mas, como P contém a ponte e_1, esta margem só pode ser U. Pelo Corolário A.2.3 novamente, U_1 é um subgrafo de uma margem de e. Mas, como U_1 possui pelo menos um vértice em comum com W, esta margem só pode ser U também. Consequentemente, a margem U_1 é um subgrafo de U. Há, agora, duas possibilidades: U_1 não possui nenhuma ponte e, sendo assim, a proposição é verdadeira (para G conexo), ou U_1 contém uma ponte de G. Assuma que U_1 possui uma ponte, e_2, de G e denomine por U_2 a margem desta ponte que não contém e_1. Usando o mesmo argumento de antes, obtém-se uma sequência, $U, U_1, U_2, \ldots$, de subgrafos onde todo elemento, exceto U, é um subgrafo de seu predecessor. Mas, como a ponte e_i não faz parte de U_{i+1}, para todo $i = 1, 2, \ldots$, tem-se que U_{i+1} contém, pelo menos, uma aresta a menos que U_i. Mas, como G é finito, a sequência $U, U_1, U_2, \ldots$ não pode ser infinita, o que implica que há uma margem, U_j, contida em U, para algum j finito, que deve ser uma folha de G. Logo, a proposição é válida quando G é conexo. Se G não é conexo, aplica-se o mesmo argumento para cada componente conexa de G que contém uma ponte, uma por vez, e se chegará à mesma contradição. □

Proposição A.2.5. Se um grafo, G, contém exatamente duas folhas, F_1 e F_2, então todo caminho, P, em G que conecta um vértice de F_1 a um vértice de F_2 contém todas as pontes de G.

*Demonstração.*Assuma, inicialmente, que G é conexo e, objetivando uma contradição, que há uma ponte, e, em G que não pertence ao caminho P. Seja H o subgrafo de G que consiste de todas as arestas (e seus respectivos vértices) de F_1, F_2 e P. Note que o subgrafo H é conexo, pois F_1 e F_2 são conexos e P conecta um vértice de cada. Note também que o subgrafo H não contém a ponte e. Logo, pelo Corolário A.2.3, o subgrafo H tem de pertencer, inteiramente, a uma das margens de e com respeito a G. Mas, isto implicaria a existência de uma terceira folha em G, o próprio subgrafo H, o que contradiz a hipótese de existência de apenas duas folhas. Logo, G não pode ser conexo. Pela Proposição A.2.4, as duas folhas de G têm de pertencer a uma mesma componente conexa de G, pois um grafo não pode possuir uma única folha[1]. Mas, como G possui exatamente duas folhas, as demais componentes conexas de G não possuem nenhuma ponte e, consequentemente, o problema se reduz ao caso de um grafo conexo novamente. Como este caso leva a uma contradição, a afirmação é verdadeira e, portanto, o caminho P contém todas as pontes de G. □

O lema a seguir faz uso de uma operação envolvendo duas arestas de um grafo. Esta operação é utilizada em um dos passos da prova do teorema de Petersen e pode ser definida como segue: sejam $e_1 = \{x_1, y_1\}$ e $e_2 = \{x_2, y_2\}$ duas arestas de um grafo G. Um novo grafo, H, é definido a partir de G, e_1 e e_2 através da remoção das arestas e_1 e e_2 e da introdução de dois novos vértices, x e y, e de cinco arestas, $\{x_1, x\}$, $\{y_1, x\}$, $\{x_2, y\}$, $\{y_2, y\}$ e $\{x, y\}$, como ilustrado na Figura A.2. A operação que transforma o grafo G no grafo H é denominada *conexão das arestas* e_1 *e* e_2. Diz-se que H se *origina* da conexão das arestas e_1 e e_2.

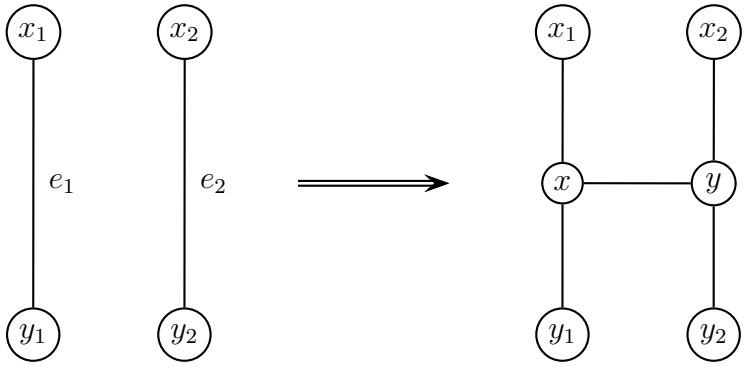


Figura A.2: Conexão das arestas e_1 e e_2.

[1]Esta afirmação é válida apenas para grafos finitos.

Lema A.2.6. Se um grafo G possui exatamente duas folhas, F_1 e F_2, e uma aresta e_1 de F_1 é conectada a uma aresta e_2 de F_2, então o grafo, H, resultante não possui nenhuma ponte.

Demonstração. A segunda parte da demonstração da Proposição A.2.5 implica que a prova da afirmação acima pode se restringir a um grafo conexo, pois as duas folhas têm, necessariamente, de pertencer a uma mesma componente conexa. Então, há um caminho, P, que conecta um vértice de $e_1 = \{x_1, y_1\}$ a um vértice de $e_2 = \{x_2, y_2\}$. De acordo com a Proposição A.2.5, o caminho P contém todas as pontes de G. Pode-se assumir que P não possui e_1 nem e_2, pois tal caminho pode ser obtido com a possível remoção de um ou dois vértices extremos de P (e de suas arestas incidentes em P). Logo, o caminho P, que conecta, por exemplo, x_1 a x_2, é um caminho em H também. Se o caminho (x_1, x, y, x_2) — *as arestas foram omitidas* — for adicionado a P, então um ciclo simples em H que contém todas as pontes de G e, também, $\{x, y\}$, é obtido. Isto implica que nem $\{x, y\}$ e nem as pontes de G são pontes de H. Logo, resta provar a afirmação para as arestas de G que não são pontes e para as arestas $\{x_1, x\}$, $\{x, y_1\}$, $\{x_2, x\}$ e $\{x, y_2\}$. Se e não é uma ponte de G, então e não pode ser uma ponte de H. De fato, por não ser ponte, há um ciclo em G que contém e. Este ciclo é um ciclo de H ou se torna um ciclo de H se a aresta e_1 for substituída pelas arestas $\{x_1, x\}$ e $\{x, y_1\}$ e a aresta e_1 for substituída pelas arestas $\{x_2, x\}$ e $\{x, y_2\}$. Finalmente, as arestas $\{x_1, x\}$, $\{x, y_1\}$, $\{x_2, x\}$ e $\{x, y_2\}$ também não são pontes de H. Como uma aresta de uma folha de G, a aresta $e_1 = \{x_1, y_1\}$ não é uma aresta de G e, portanto, pertence a um ciclo simples de G. Se, neste ciclo, a aresta e_1 for substituída pelas arestas $\{x_1, x\}$ e $\{x, y_1\}$, então um ciclo simples em H é obtido. Logo, as arestas $\{x_1, x\}$ e $\{x, y_1\}$, que pertencem a este ciclo, não são pontes de H. O mesmo argumento pode ser utilizado para provar que $\{x_2, x\}$ e $\{x, y_2\}$ também não são pontes de H. Logo, nenhuma aresta de H pode ser uma ponte e, portanto, a afirmação é verdadeira. □

Finalmente, pode-se enunciar e provar o importante teorema de Frink. Este teorema faz uso de uma operação, denominada *redução*, que é a base dos algoritmos em (BIEDL et al., 2001; DIKS; STANCZYK, 2010) para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes. Antes de se enunciar o teorema, define-se e ilustra-se a operação de redução. Seja $e = \{u, w\}$ uma aresta de um grafo arbitrário, G, que 1) conecta dois vértices distintos de grau 3 cada e 2) não é uma aresta paralela, como mostra a Figura A.3. As arestas e_1 e e_2 são incidentes no vértice u e as arestas e_3 e e_4 são incidentes no vértice w. Removem-se de G os vértices u e w e as cinco arestas, e, e_1, e_2, e_3 e e_4, incidentes

neles e adicionam-se duas novas arestas, e_{13} e e_{24}, conectando o vértice x_1 ao vértice x_3 e o vértice x_2 ao vértice x_4, respectivamente. Denomina-se o grafo resultante de G_1. Se, por outro lado, as arestas adicionadas são e_{14} e e_{23}, que conectam o vértice x_1 ao vértice x_4 e o vértice x_2 ao vértice x_3, respectivamente, então o grafo resultante é denominado G_2. Diz-se que os dois grafos, G_1 e G_2, *originam-se de G a partir da redução da aresta* $e = \{u, w\}$.

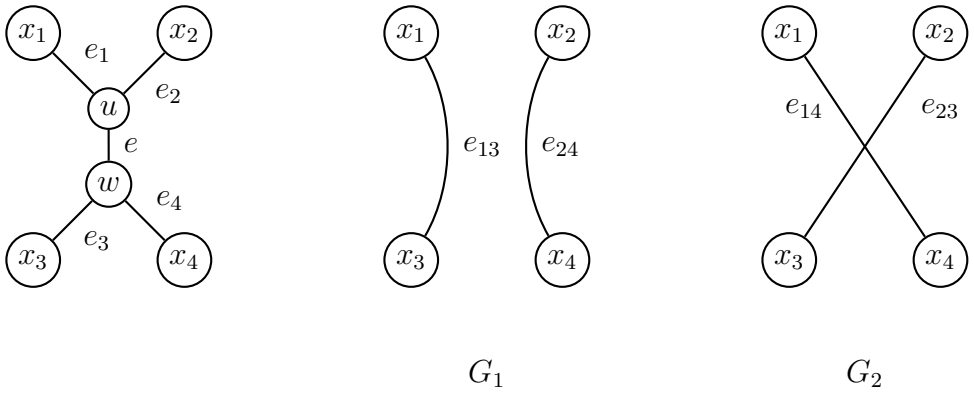


Figura A.3: Redução da aresta $e = \{u, w\}$.

Há algumas sutilezas na operação de redução. Para discuti-las de forma mais objetiva, assume-se que G é um grafo cúbico — o caso que realmente importa neste trabalho. As sutilezas são casos especiais que surgem quando os vértices x_1, x_2, x_3 e x_4 não são todos distintos. Mais especificamente, como se assumiu que o grafo é cúbico, tem-se que até três desses vértices podem ser os mesmos, mas os quatro não podem. Por isso, com exceção de simetria ou troca de índices dos vértices, os casos especiais se reduzem a um dos cinco abaixo:

(a) $x_1 = x_3$ e $x_2 \neq x_4$,

(b) $x_1 = x_3$ e $x_2 = x_4$,

(c) $x_1 = x_2$ e $x_3 \neq x_4$,

(d) $x_1 = x_2$ e $x_3 = x_4$ e

(e) $x_1 = x_2 = x_3$ e $x_1 \neq x_4$.

O grafo G dos casos (c), (d) e (e) possui, obrigatoriamente, arestas paralelas. Os casos (a), (b), (d) e (e) originam grafos, G_1 e G_2, que possuem laços ou arestas paralelas dentre

aquelas arestas introduzidas pela operação de redução. Já o caso (d) origina grafos, G_1 e G_2, que possuem apenas arestas paralelas dentre aquelas arestas introduzidas pela operação de redução. As Figuras de A.4 a A.8 ilustram cada um dos cinco casos especiais acima.

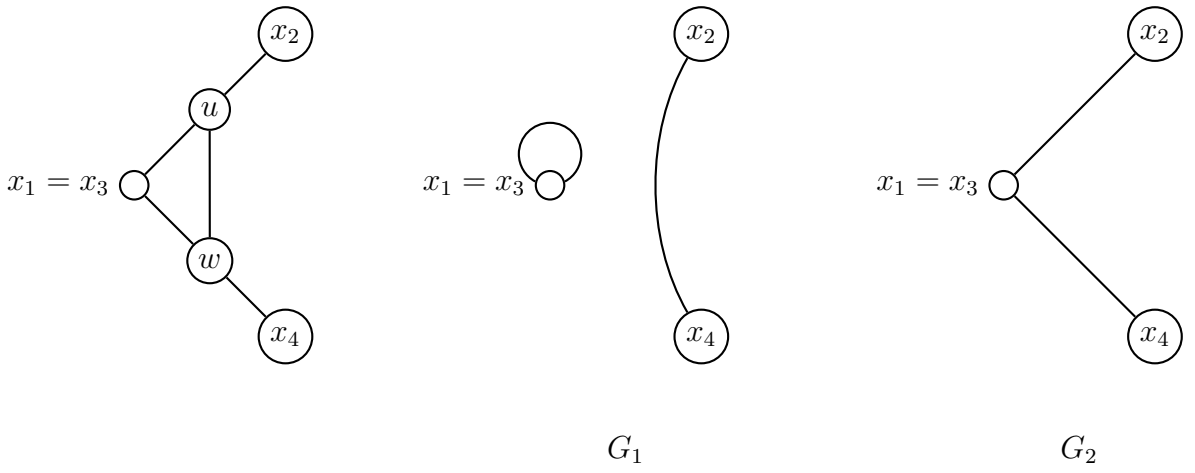


Figura A.4: Redução da aresta $e = \{u, w\}$ quando $x_1 = x_3$ e $x_2 \neq x_4$.

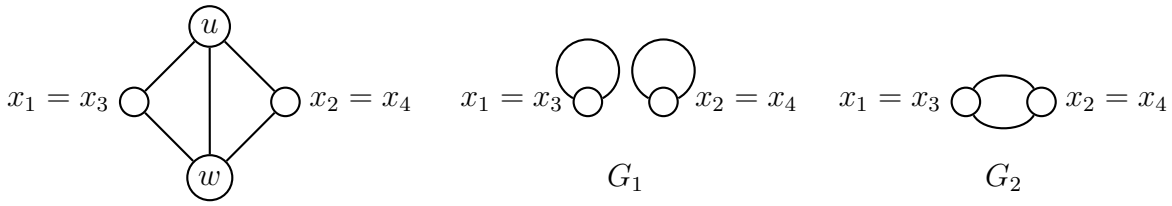


Figura A.5: Redução da aresta $e = \{u, w\}$ quando $x_1 = x_3$ e $x_2 = x_4$.

É importante ressaltar que o grafo H, originado a partir de um grafo G cúbico por uma redução de aresta, continua sendo um grafo cúbico, pois o grau de cada um dos vértices, x_1, x_2, x_3 e x_4, em H, é sempre 3, independentemente da redução envolver casos especiais. Além disso, o grafo H possui *dois vértices a menos* do que G. Essas duas observações são fundamentais para o desenvolvimento dos algoritmos em (BIEDL et al., 2001; DIKS; STANCZYK, 2010), que encontram emparelhamentos perfeitos em grafos cúbicos e sem pontes.

Os dois algoritmos se baseiam na mesma idéia de aplicar sucessivas reduções de aresta a um grafo conexo, cúbico, sem pontes e, inicialmente, simples. Após cada redução, os

dois grafos, G_1 e G_2, resultantes também são cúbicos e, como estabelece o teorema de Frink a seguir, *pelo menos um deles é conexo, sem pontes e sem nenhum laço* (embora arestas paralelas possam existir). Os algoritmos se valem deste fato para reduzir o grafo inicial, recursivamente, a um grafo que contém apenas dois vértices e que consiste no caso base da recursão. Em cada passo, apenas um grafo conexo e sem pontes e laços, resultante da redução, é considerado — o outro grafo é descartado. Quando o caso base é atingido, um emparelhamento perfeito trivial é calculado e, no retrocesso da recursão, o emparelhamento é "aumentado" à medida que as reduções de aresta são desfeitas para restaurar o grafo do passo atual. Esta operação é local e envolve apenas as arestas da redução.

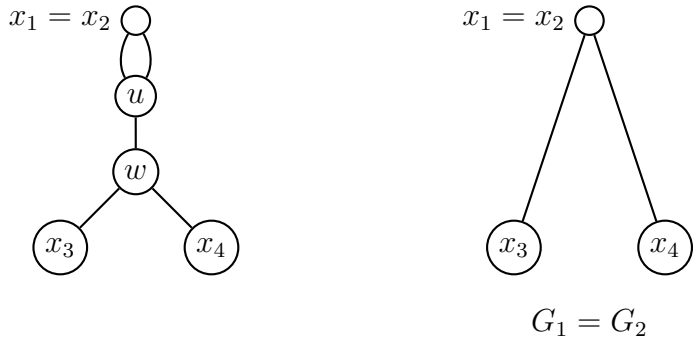


Figura A.6: Redução da aresta $e = \{u, w\}$ quando $x_1 = x_2$ e $x_3 \neq x_4$.

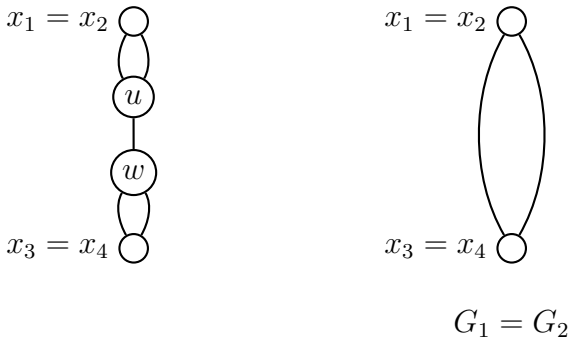


Figura A.7: Redução da aresta $e = \{u, w\}$ quando $x_1 = x_2$ e $x_3 = x_4$.

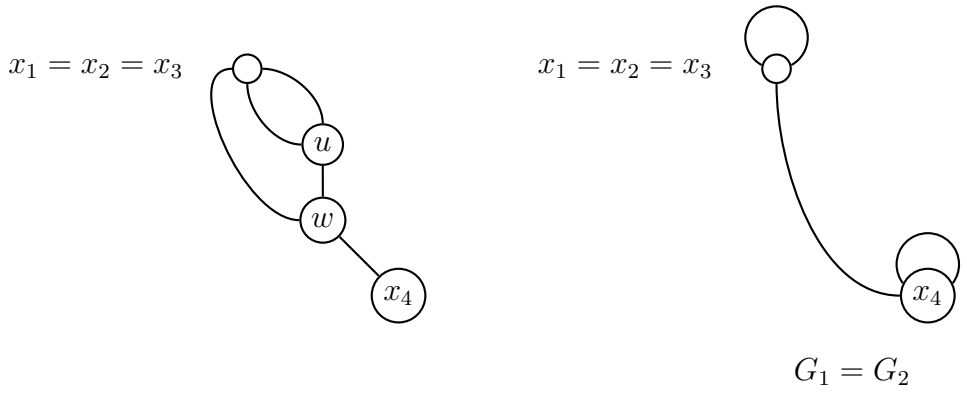


Figura A.8: Redução da aresta $e = \{u, w\}$ quando $x_1 = x_2 = x_3$ e $x_1 \neq x_4$.

Teorema A.2.7 (**Teorema de Frink**)**.** Seja G um grafo conexo, cúbico e sem pontes e seja e uma aresta de G que não pertence a um 2-ciclo em G. Sejam G_1 e G_2 os dois grafos cúbicos que se originam de G a partir da redução da aresta e. Então, pelo menos um desses dois grafos, G_1 ou G_2, é, ao mesmo tempo, um grafo conexo, cúbico e sem pontes.

Demonstração. Como G é cúbico e sem pontes, não pode haver nenhum laço em G, pois a existência de um laço incidente em um vértice, v, de G implicaria na existência de uma outra aresta, f, incidente em v e em outro vértice, u, de G, que não é aresta paralela de G, pois v possui grau 3. A remoção de f geraria um grafo, $G - f$, no qual o vértice v se torna um vértice isolado dos demais vértices de $G - f$, o que implica que f é uma ponte de G, contradizendo a hipótese de que G não possui pontes. No entanto, o grafo G pode possuir arestas paralelas. Por definição, a operação de redução de aresta só pode ser aplicada a uma aresta que *não* é paralela. Suponha que G possui uma aresta não paralela, e, tal que a hipótese do teorema seja satisfeita. A demonstração do teorema possui cinco partes. As descrições de algumas das cinco partes se referem à Figura A.3, que ilustra a redução de e.

(a) A primeira parte da prova destina-se a mostrar que *se o grafo G_1 não é conexo, então ele possui exatamente duas componentes conexas: uma contendo a aresta e_{13} e outra contendo a aresta e_{24}*. Assuma, por contradição, que uma componente conexa, U, de G_1 não contém nem e_{13} nem e_{24}. Então, a componente U é um subgrafo de G. Seja z um vértice qualquer de U. Como, por hipótese, o grafo G é conexo, há um caminho, P, em G que conecta z a u. O caminho P deve, obrigatoriamente, conter o vértice x_1, x_2, x_3 ou x_4. Sem perda de generalidade, assuma que x_1 é o

primeiro desses quatro vértices em P na direção de z a u. Então, uma parte de P é um caminho que conecta a com x_1 e não contém nenhuma das arestas, e_1, e_2, e_3, e_4 e e, de G que não estão em G_1. Logo, o caminho P também é um caminho em G_1. Mas, neste caso, a aresta e_{13} tem de pertencer à componente U, o que contradiz a hipótese de que U não contém nem e_{13} nem e_{24}. Se, ao invés de x_1, o vértice x_2, x_3 ou x_4 fosse o primeiro dos quatro vértices a ocorrer em P na direção de z a u, então o mesmo argumento poderia ser usado para mostrar que e_{13} ou e_{24} pertence a U. Portanto, a componente U possui uma das duas arestas. Além disso, se ela possuir as duas arestas, então outra componente de G_1 não poderia possuir nenhuma das duas, o que é impossível pelo que se acabou de mostrar. Isto implica que o grafo G_1 possui exatamente duas componentes conexas, uma contendo e_{13} e outra contendo e_{24}.

(b) A segunda parte da prova destina-se a mostrar que *nem* e_{13} *nem* e_{24} *é uma ponte de* G_1. Assuma, por contradição, que e_{13} é uma ponte de G_1. Logo, não pode existir nenhum caminho, Q, de x_1 a x_3 em G_1 que não contenha a aresta e_{13}, pois se tal caminho existisse, então se poderia definir um ciclo simples, $Q+e_{13}$, em G_1 contendo e_{13}, o que implicaria em e_{13} não ser uma ponte de G_1: uma contradição. Mas, como e_1 não é uma ponte de G — pois o grafo G não possui pontes, por hipótese — há um caminho, R, em G que conecta x_1 a u, mas não contém e_1. Obviamente, o caminho R tem de conter x_2, x_3 ou x_4, e um desses três vértices é o primeiro a ocorrer em R na direção de x_1 a u. Se for x_3, então uma parte de R seria um caminho em G_1 que conecta x_1 a x_3 e não contém e_{13} e, portanto, a aresta e_{13} não seria uma ponte em G_1: uma contradição. Então, o primeiro vértice deve ser x_2 ou x_4. Usando o mesmo argumento — mas, esta vez, considerando o vértice x_3 e a aresta e_3 ao invés de x_1 e e_1 — obtém-se um caminho, S, de x_3 para x_2 ou x_4 que não contém e_{13}. Certas arestas de R e S definem um caminho em G_1, possivelmente contendo a aresta e_{24}, que não contém a aresta e_{13} e conecta x_1 a x_3 (veja a Proposição 2.1.10). Isto implica que e_{13} não é uma ponte em G_1: uma contradição. Usando o mesmo argumento, mas, desta vez, considerando a aresta e_{24} como uma ponte de G_1, conclui-se, por contradição, que e_{24} não é uma ponte em G_1. Logo, nem e_{13} nem e_{24} é uma ponte de G_1.

(c) A terceira parte da prova destina-se a mostrar que *se* $f = \{z_1, z_2\}$ *é uma ponte de* G_1*, então uma margem de* f *com respeito a* G_1 *contém* e_{13} *e a outra contém* e_{24}. Pelo item (b), pode-se concluir que f pertence a G e, portanto, está contida em um ciclo simples, K, de G. O ciclo K deve conter uma das arestas, e_1, e_2, e_3 ou e_4, pois, caso contrário, a aresta e também não faria parte de K, o que implica que o ciclo

K seria um ciclo simples em G_1 que contém a aresta f: uma contradição com o fato de f ser ponte de G_1. Portanto, o ciclo K deve, obrigatoriamente, conter um dos vértices x_1, x_2, x_3 e x_4. Considere o caminho formado pelos vértices e arestas de K na direção que não leva z_1 imediatamente para z_2. Seja x_1 o primeiro vértice dos quatro acima que ocorre neste caminho. A parte deste caminho que vai de z_1 a x_1 é um caminho, P, em G_1 que não contém a ponte f. Pelo Corolário A.2.3, o vértice x_1 — e, portanto, a aresta e_{13} — pertence à margem de f que se origina a partir de z_1. De forma análoga, considere o caminho, Q, formado pelos vértices e arestas de K na direção que não leva z_2 imediatamente para z_1. Note que x_2 ou x_4 deve ser o primeiro vértice dos quatro acima que ocorre em Q. Caso contrário, a parte de Q que vai de z_2 a x_2 (ou x_4), juntamente com f e o caminho P e, possivelmente com a adição de e_{13}, formariam um ciclo simples contendo f, o que contradiz a hipótese de f ser uma ponte de G_1. Logo, pelo Corolário A.2.3 novamente, o vértice x_2 (ou x_4) — e, portanto, a aresta e_{24} — pertence à margem de f que se origina a partir de z_2. Portanto, conclui-se que e_{13} e e_{24} estão em margens distintas da ponte f do grafo G_1.

(d) Do item (b), pode-se concluir que há um ciclo, K_1, simples em G_1 que contém a aresta e_{13} e outro ciclo simples, K_2, em G_1 que contém a aresta e_{24}. A quarta parte da prova destina-se a mostrar que *se G_1 não é, ao mesmo tempo, conexo e sem pontes, então os ciclos K_1 e K_2 são mutuamente disjuntos, ou seja, eles não possuem nem arestas e nem vértices em comum.* Assuma que G_1 não é conexo. Então, cada um dos ciclos, K_1 e K_2, pertence, completamente, a uma componente conexa de G_1. Do item (a), pode-se concluir que G_1 possui exatamente duas componentes conexas e que uma delas contém K_1 e a outra contém K_2. Logo, esses dois ciclos são mutuamente disjuntos *se G_1 não for conexo.* Por outro lado, se G_1 for conexo, então, para que a hipótese da afirmação seja verdadeira, deve-se assumir que G_1 possui uma ponte, f. Por definição, a ponte f não pode pertencer a K_1 nem a K_2. Neste caso, o Corolário A.2.3 diz que cada um dos ciclos, K_1 e K_2, deve pertencer inteiramente a uma mesma margem de f com respeito a G_1. Do item (c), pode-se concluir que essas margens são distintas, o que implica que K_1 e K_2 são ciclos simples disjuntos.

(e) A quinta parte da prova combina as afirmações dos itens anteriores para concluir a prova do teorema de Frink. Assuma que G_1 não é, ao mesmo tempo, conexo e sem pontes. Pretende-se mostrar que G_2 é conexo, sem pontes e sem laços. Para tal, suponha que as arestas e_{13} e e_{24} sejam removidas de G_1. Então, pela demonstração do item (d), os ciclos K_1 e K_2 dão surgimento a dois caminhos, P_1 e P_2, que conectam

os vértices x_1 a x_3 e x_2 a x_4, respectivamente. De acordo com a afirmação provada em (d), os caminhos, P_1 e P_2, são disjuntos. Observe que P_1 e P_2 também são caminhos de G e G_2. Se as arestas, e_{14} e e_{23}, e as arestas de P_2 forem adicionadas ao caminho P_1, então um ciclo simples em G_2 é obtido (veja a Figura A.9). Isto implica que e_{14} e e_{23} não são pontes de G_2. Logo, se a contrapositiva da afirmação em (d) for aplicada a G_2, substituindo-se e_{13} e e_{24} por e_{14} e e_{23}, conclui-se que G_2 é conexo e sem pontes. Resta, agora, mostrar que se G_2 é conexo e sem pontes, então G_2 também não possui nenhum laço. Suponha, por contradição, que há um laço incidente em um vértice v de G_2. Como G não possui laços, este laço foi criado durante a redução da aresta e que deu origem a G_2. Logo, uma das arestas, e_{14} ou e_{23}, é o laço incidente em v. Como G_2 é um grafo cúbico, deve haver outra aresta, f, incidente em v. Mas a remoção de f de G_2 isolaria v dos demais vértices de G_2, o que implica que f é uma ponte, contradizendo a hipótese de que G_2 não possui pontes.

□

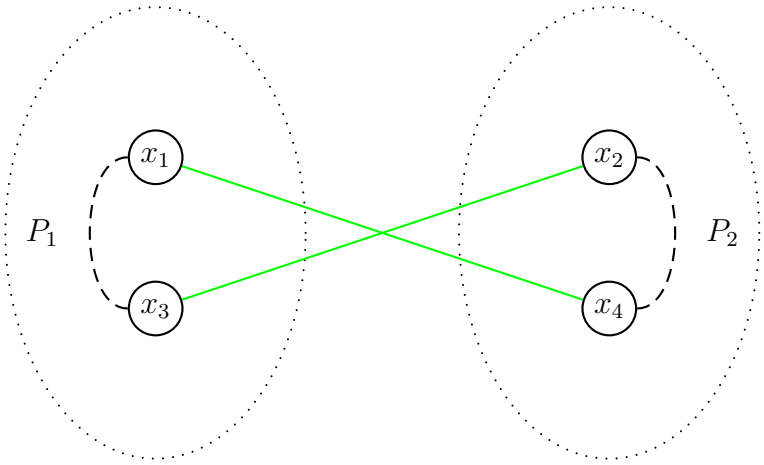


Figura A.9: Ilustração da parte (e) da prova do teorema de Frink.

A.3 O teorema de Petersen: caso particular

A prova do caso particular do teorema de Petersen que será apresentada aqui depende, fortemente, do teorema de Frink, que foi provado na Seção A.2, e do lema apresentado a seguir:

Lema A.3.1. Seja G um grafo conexo, cúbico e sem pontes que admite um emparelhamento perfeito. Então, toda aresta, e, de G está contida em um ciclo alternante de G.

Demonstração. Suponha, por contradição, que a afirmação seja falsa. Então, existe um grafo, G, com o menor número de vértices possível para o qual existe uma aresta que não faz parte de nenhum ciclo alternante. Por ser cúbico, o grafo G não pode possuir menos do que 2 vértices. Então, suponha que G contenha exatamente dois vértices, u e w. Neste caso, o grafo G possui exatamente três arestas e cada uma delas conecta u a w (veja a Figura A.10). Se G admite um emparelhamento perfeito, então exatamente uma das três arestas pertence ao emparelhamento. Mas, isto implica que toda aresta de G faz parte de um 2-ciclo alternante em G, o que é uma contradição. Logo, o grafo G possui mais de dois vértices.

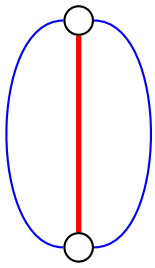


Figura A.10: Um emparelhamento perfeito em um grafo cúbico com apenas dois vértices.

Como o grafo G não pode possuir três vértices (pois a soma do grau dos vértices tem de ser um número par), suponha que G possui pelo menos mais dois vértices, x e y, além de u e w. Suponha também que há duas arestas (paralelas) conectando u a w, uma aresta conectando u a x e outra aresta conectando w a y (veja a Figura A.11). Note que $x \neq y$. Caso contrário, como x possui grau 3, haveria uma terceira aresta, f, incidente em x e que não incide em u nem em w. Se f fosse removida de G, a componente conexa contendo x, y, u e w estaria isolada do restante de $G - f$, o que implica que f é uma ponte de G. Mas, como G não possui pontes, tem-se que $x \neq y$. Agora, suponha que os vértices u e w são removidos de G, juntamente com as arestas incidentes neles, e que a aresta $\{x, y\}$ é adicionada ao grafo resultante. Seja G' o grafo obtido a partir dessas operações. O grafo G' permanece cúbico e conexo. Afirma-se que G' não possui pontes. De fato, se e é uma aresta de G', tal que $e \neq \{x, y\}$, então a aresta e pertence a G e, como G não possui pontes, há um ciclo simples, K, em G que contém e. Se K não contém nenhuma das arestas que

foram removidas de G, então K também é um ciclo simples em G'. Caso contrário, o ciclo K deve conter o caminho $(x, \{x, u\}, u, g, w, \{w, y\}, y)$ ou o inverso dele, onde g é uma das duas arestas conectando u a w. Se este caminho for substituído, em K, pela aresta $\{x, y\}$, então um ciclo simples em G', contendo e e $\{x, y\}$, é obtido. Logo, a aresta e não é uma ponte de G'. Por outro lado, se $e = \{x, y\}$, então suponha que L é qualquer ciclo simples em G que contém a aresta $\{x, u\}$. A existência de L é garantida pelo fato de G não possuir pontes. O ciclo L também tem de conter o caminho $(x, \{x, u\}, u, h, w, \{w, y\}, y)$ ou o inverso dele, onde h é uma das duas arestas conectando u a w. Usando o mesmo argumento de antes, conclui-se que e pertence a um ciclo simples em G'. Logo, o grafo G' não possui pontes. Observe também que G' não possui laços, pois a existência de um laço em um grafo cúbico implica na existência de uma ponte.

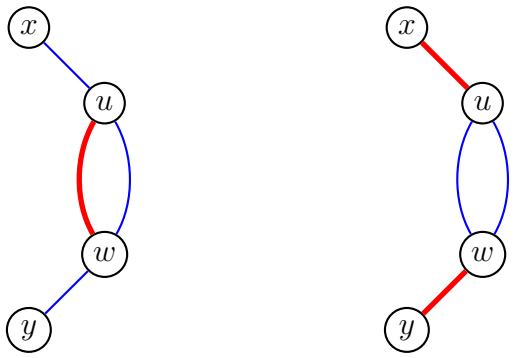


Figura A.11: Configurações de emparelhamento das arestas incidentes em u e w.

O que se quer agora é definir um emparelhamento perfeito, M', em G'. Para tal, escolhe-se qualquer emparelhamento perfeito, M, em G. Por hipótese, há sempre uma escolha. Em seguida, inclui-se no emparelhamento em G', a ser definido, todas as arestas comuns a G e G' que fazem parte de M. Finalmente, deve-se decidir se a aresta $\{x, y\}$ fará parte de M'. Esta decisão depende de quais arestas de G, incidentes em u e w, fazem parte de M. A Figura A.11 mostra as duas únicas configurações possíveis para as arestas de G, incidentes em u e w, com respeito a M. Na configuração da esquerda, tem-se o caso (a): as arestas $\{x, u\}$ e $\{w, y\}$ *não* fazem parte de M; na configuração da direita, tem-se o caso (b): as arestas $\{x, u\}$ e $\{w, y\}$ fazem parte de M. No caso (a), não se inclui $\{x, y\}$ em M'. No caso (b), inclui-se $\{x, y\}$ em M'. Em ambos os casos, têm-se que todos os vértices de G' estão cobertos por exatamente uma aresta de M' e, portanto, o emparelhamento M' é perfeito.

Pelo exposto acima, pode-se afirmar que G' é um grafo conexo, cúbico, sem pontes, sem

laços e que admite um emparelhamento perfeito. Mas, como G' possui dois vértices a menos que G, toda aresta de G' está contida, por hipótese, em um ciclo alternante, K, de G' com respeito a M'. Caso contrário, o grafo G não seria o grafo com o menor número de vértices para o qual a afirmação que se quer provar é falsa. Se $\{x, y\}$ *não* está contida em K, então K também é um ciclo alternante em G com respeito a M. Logo, a aresta $\{x, y\}$ deve pertencer a K. Mas, neste caso, pode-se substituir $\{x, y\}$ por $\{x, u\}$, g e $\{w, y\}$, onde g é uma das duas arestas conectando u a w, para se obter um ciclo simples, L, em G. Afirma-se que L é um ciclo alternante em G, com respeito a M, que contém as arestas $\{x, u\}$, g e $\{w, y\}$. De fato, se $\{x, y\} \notin M'$, então se tem o caso (a) e, se $\{x, y\} \in M'$, então se tem o caso (b). Como M e M' coincidem quando restritos às arestas comuns a G e G', tem-se que uma aresta comum a L e K está em M' se, e somente se, ela está em M. Note que a outra aresta, h, que conecta u a w, também está contida em um ciclo alternante de G com respeito a M. No caso (a), este ciclo é o 2-ciclo formado por g, h e os vértices u e w. No caso (b), o ciclo alternante é o próprio L. Logo, toda aresta de G pertence a um ciclo alternante em G com respeito a M, o que contradiz o fato de G ser um grafo conexo, cúbico, sem pontes, sem laços e com o menor número de vértices para o qual esta afirmação é falsa, *a menos que o grafo G não possua arestas paralelas como g e h.*

Assuma, portanto, que G não possui arestas paralelas. Então, G é um grafo conexo, cúbico, sem pontes, sem laços e sem arestas paralelas (ou seja, o grafo G é conexo, cúbico, sem pontes e simples). Assuma também que a afirmação que se quer provar é falsa para G e, além disso, que ela é sempre verdadeira para todo grafo conexo, cúbico, sem pontes, sem laços e com um número de vértices menor do que o de G. Este grafo, no entanto, pode possuir arestas paralelas. Usando contradição, mostra-se que toda aresta de G faz parte de um ciclo alternante em G com respeito ao emparelhamento perfeito, M, escolhido anteriormente. Seja $f = \{x_2, t\}$ uma aresta qualquer de G. Considere os dois seguintes casos:

(a) $f \notin M$

(b) $f \in M$

Considere o caso (a) e a Figura A.12. Seja $e_2 = \{u, x_2\}$ a outra aresta de G incidente em x_2 e que não está em M e seja $e = \{u, w\}$ a aresta de G que está em M e é incidente em u. As demais arestas consideradas aqui são $e_1 = \{x_1, u\}$, $e_3 = \{x_3, w\}$ e $e_4 = \{x_4, w\}$. Como a aresta e não faz parte de nenhum 2-ciclo, ela pode ser reduzida para gerar o grafo

G_1 do teorema de Frink (veja Teorema A.2.7), que contém as arestas e_{13} e e_{24} mostradas na Figura A.12. Sem perda de generalidade, assuma que G_1 é conexo e sem pontes (caso contrário, o Teorema A.2.7 garante que G_2 é e pode-se considerar G_2 aqui ao invés de G_1). Lembre-se de que G_1 também é cúbico e sem laços (pois, por hipótese, ele não possui pontes). Defina um emparelhamento perfeito, M_1, em G_1 da mesma forma que M' foi definido em G': uma aresta comum a G_1 e G está em M_1 se, e somente se, ela está em M. Finalmente, como $f \in M$, as arestas e_1, e_2, e_3 e e_4 não podem pertencer a M. Além disso, todos os vértices de G, com exceção de u e w, estão cobertos por exatamente uma aresta de $M - \{e\}$. Logo, pode-se concluir a definição de M_1 deixando e_{13} e e_{24} de fora de M_1. Como os únicos vértices de G que não estão em G_1 são u e w, o emparelhamento M_1 é perfeito.

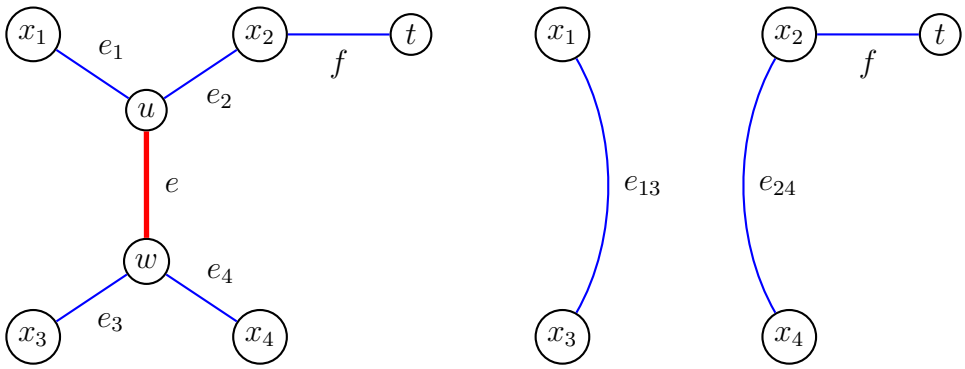


Figura A.12: As arestas e e f (esquerda) e as arestas e_{13} e e_{24} do grafo G_1.

O que se faz em seguida é mostrar, por contradição, que a aresta f pertence a um ciclo alternante em G com respeito a M. Para tal, distinguem-se duas situações. Na primeira, a aresta f pode ser igual a uma das cinco arestas 1) e, 2) e_1, 3) e_2, 4) e_3 e 5) e_4. Na segunda, a aresta f não é igual a nenhuma dessas cinco arestas. No que segue, prova-se que há um ciclo alternante em G, com respeito a M, e contendo f para a primeira situação do caso (a).

Como $e \in M$ e $f \notin M$, o caso 1 é impossível. Por definição de e_2, o caso 3 não pode ocorrer. Se $f = e_1$ então $t = u$ e $x_1 = x_2$, o que também é impossível, pois G não possui arestas paralelas, por hipótese. Logo, o caso 2 pode ser descartado. Se $f = e_4$ então $x_2 = x_4$ e $t = w$ (pois se $x_2 = w$ então G teria um 2-ciclo). Mas, isto implicaria em e_{24} ser um laço de G_1, o que também não é possível, por hipótese, e o caso 5 pode ser descartado. Logo, resta apenas o caso 4, $f = e_3$. Como $x_2 \neq w$ (caso contrário, G teria um 2-ciclo), tem-se $x_2 = x_3$ e $t = w$ (veja a Figura A.13). Então, a primeira situação do caso (a) é equivalente

ao caso 4.

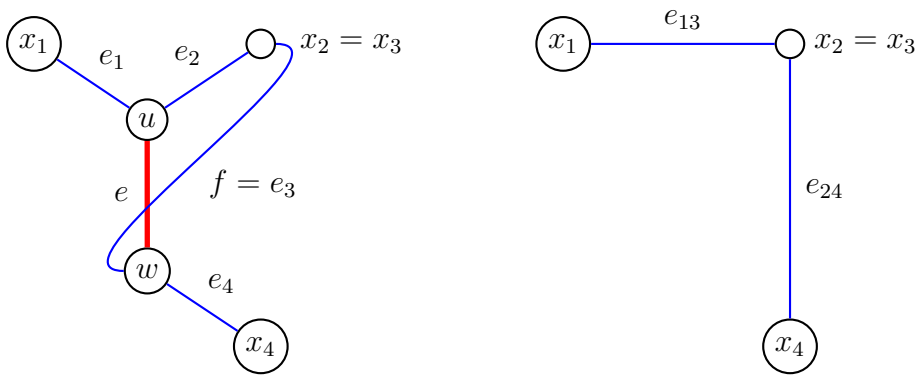


Figura A.13: O caso em que $x_2 = x_3$ e $t = w$ no grafo G e as arestas e_{13} e e_{24} do grafo G_1.

Como G_1 possui menos vértices do que G, tem de haver, em G_1, um ciclo alternante, K, com respeito a M_1 que contém a aresta e_{13}. Isto porque, por hipótese, o grafo G é o grafo conexo, cúbico, sem pontes, sem laços e com o menor número de vértices para o qual tal ciclo não existe para todas as arestas. Por definição, tem-se que e_{13} e e_{24} não pertencem a M_1. Logo, as arestas e_{13} e e_{24} não podem ser consecutivas em K. Desta forma, se a aresta e_{13} em K for substituída pelas arestas e_1, e e e_3, onde $e \in M$, obtém-se um ciclo alternante em G, com respeito a M, que contém f. A Figura A.14 mostra um exemplo de um grafo G em que as arestas f e e_3 são as mesmas e, além disso, tem-se $x_1 = x_4$. Como o ciclo resultante é um ciclo alternante em G com respeito a M que contém f, tem-se uma contradição. Logo, a primeira situação do caso (a) (isto é, $f = e_3$) também não pode ocorrer.

O que resta então é a segunda situação do caso (a): f não é igual a e, e_1, e_2, e_3 e nem e_4; ou seja, a aresta f é uma aresta do grafo G_1. Por hipótese, há um ciclo alternante, L, em G_1 com respeito a M_1 que contém f, já que G_1 é um grafo conexo, cúbico, sem pontes, sem laços e com um número de vértices menor do que o de G. Como a aresta f não pertence a M, tem-se que $f \notin M_1$ e, portanto, a aresta e_{24}, que não faz parte de M_1 e também é incidente em x_2, não pode pertencer a L. Logo, há duas possibilidades mutuamente exclusivas: L é um ciclo alternante de G com respeito a M ou L contém e_{13}. O primeiro caso leva à contradição desejada. Logo, considere o segundo caso. Se e_{13}, que não está em M_1, for substituída em L por e_1, e e e_3, obtém-se um ciclo alternante em G, com respeito a M, que contém f. Logo, chega-se à mesma contradição e, portanto, o caso (a) — isto é, $f \notin M$ — não pode ocorrer. Logo, resta apenas considerar o caso (b):

$f \in M$.

Se $f \in M$ então assuma novamente que e_2 é a aresta de G incidente em x_2 que não está no emparelhamento M. Mas, pelo que se acabou de provar, há um ciclo alternante em G, com respeito a M, que contém e_2. Como f é a única aresta de M incidente em x_2, esta aresta tem de pertencer ao mesmo ciclo, o que contradiz a hipótese de tal ciclo não existir. □

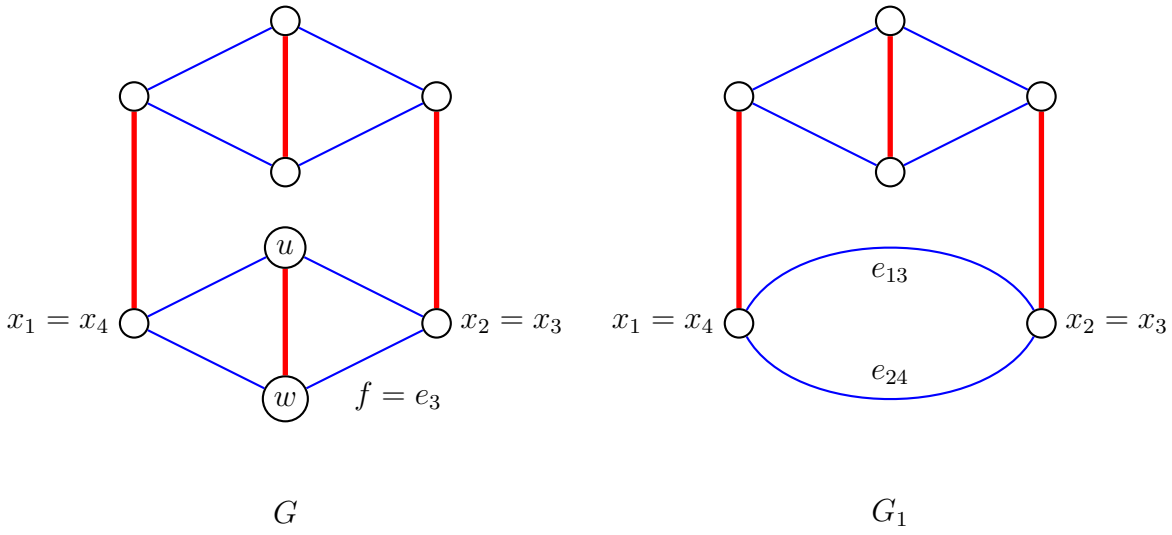


Figura A.14: Um grafo G em que $f = e_3$ e $x_1 = x_4$ (esquerda) e o grafo G_1 (direita).

Finalmente, pode-se provar o aludido caso particular do teorema de Petersen:

Teorema A.3.2. Todo grafo cúbico e sem pontes admite um emparelhamento perfeito.

Demonstração. Seja G o grafo cúbico, sem pontes e com o menor número de vértices para o qual a afirmação do teorema é falsa. Este grafo é, claro, conexo. Logo, não pode haver três arestas (paralelas) em G que conectem o mesmo par de vértices, pois se houvesse, então o grafo conteria apenas essas três arestas e os dois vértices e admitiria um emparelhamento perfeito (veja a Figura A.10). Isto implica que existe, em G, uma aresta que não é paralela. De fato, se uma aresta, $g = \{w, x\}$, é paralela, então existe uma segunda aresta, $f = \{w, x\}$, que conecta w a x e uma terceira aresta, $e = \{u, w\}$, incidente sobre w, com $u \neq x$, pois w possui grau 3 (veja a Figura A.15). Como não existe uma quarta aresta incidente no vértice w, a aresta e não está contida em um 2-ciclo e, portanto, não é paralela.

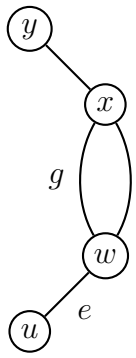


Figura A.15: A aresta g é paralela, mas e não é.

Seja $e = \{u, w\}$ uma aresta de G que não é paralela. Para facilitar a relação com as demonstrações anteriores, suponha que as demais arestas incidentes em u e w sejam $e_1 = \{x_1, u\}$, $e_2 = \{x_2, u\}$, $e_3 = \{x_3, w\}$ e $e_4 = \{x_4, w\}$, como ilustrado na Figura A.3 (esquerda). Como e não é paralela, pode-se aplicar uma redução em e. Sem perda de generalidade, assuma que o grafo G_1 da Figura A.3 (meio) é o grafo cúbico, conexo e sem pontes do Teorema A.2.7. Lembre-se de que G_1 não contém nenhum laço (pois G_1 é cúbico e sem pontes). Como G_1 possui dois vértices a menos do que G, o grafo G_1 admite (por hipótese) um emparelhamento perfeito, M_1. Mostra-se, em seguida, que este também é o caso de G, contradizendo a hipótese. Para tal, defina o emparelhamento, M, em G tal que uma aresta comum a G e G_1 está em M se, e somente se, ela está em M_1. As únicas arestas de G que não estão em G_1 são e_1, e_2, e_3, e_4 e e. Observe que o emparelhamento M não cobre os vértices u e w e, possivelmente, um ou mais vértices em $\{x_1, x_2, x_3, x_4\}$. O objetivo é aumentar M com uma ou mais dessas arestas em $\{e, e_1, e_2, e_3, e_4\}$ tal que M se torne um emparelhamento perfeito em G. Para tal, considere os três casos a seguir envolvendo e_{13} e e_{24}:

(a) As duas arestas e_{13} e e_{24} *não* fazem parte de M_1.

(b) Exatamente uma das arestas e_{13} e e_{24} faz parte de M_1.

(c) As duas arestas e_{13} e e_{24} fazem parte de M_1.

No caso (a), tem-se que $M = M_1$. Logo, inclui-se apenas e em M, pois todos os vértices de G, com exceção de u e w, já estão cobertos por exatamente uma aresta de M, já que M_1 é um emparelhamento perfeito em G_1 e e_{13} e e_{24} não cobrem nenhum vértice. No caso (b), suponha, sem perda de generalidade, que $e_{13} \in M_1$ e $e_{24} \notin M_1$. Então, os vértices x_1

e x_3 estão cobertos por e_{13}, enquanto os vértices x_2 e x_4 estão cobertos por arestas que também pertencem a M. Logo, inclui-se e_1 e e_3 em M, que juntas cobrem os vértices x_1, x_3, u e w em G. Como os demais vértices de G já estão cobertos pelas arestas comuns a M e M_1, tem-se que M é um emparelhamento perfeito em G. No caso (c), o Lema A.3.1 garante a existência de um ciclo alternante, K, em G_1, com respeito a M_1, que contém a aresta e_{13}. Então, pode-se obter um outro emparelhamento perfeito, M_1', em G_1 a partir de M_1 substituindo cada aresta de K em M_1 pela aresta consecutiva em K que não está em M_1. Obviamente, a aresta e_{13} não estará em M_1' e, se a aresta e_{24} pertence a K e a M_1, então ela também não estará em M_1'. Em qualquer um desses casos, entretanto, reduz-se o problema ao caso (a) ou ao caso (b). Logo, o grafo G admite um emparelhamento perfeito, o que contradiz a hipótese de G ser o grafo cúbico, sem pontes e com o menor número de vértices para o qual a afirmação do teorema é falsa. Logo, a afirmação do teorema é válida. □

A idéia de obter um emparelhamento perfeito em G a partir de um emparelhamento perfeito em G_1, usando a operação de redução de aresta do teorema de Frink (veja Teorema A.2.7), é a base da prova do Teorema A.3.2. A mesma idéia serviu de base para projetar o passo recursivo e garantir a corretude dos algoritmos em (BIEDL et al., 2001; DIKS; STANCZYK, 2010) para encontrar emparelhamentos perfeitos em grafos cúbicos e sem pontes.

A.4 O teorema de Petersen: caso geral

O caso geral do teorema de Petersen pode ser estabelecido como segue:

Teorema A.4.1. Todo grafo cúbico com, no máximo, duas folhas admite um emparelhamento perfeito.

Demonstração. Se o grafo não possui pontes, então o Teorema A.3.2 garante a veracidade da afirmação. Pela Proposição A.2.4, um grafo não pode conter uma única folha, pois cada margem de uma ponte contém um subgrafo que é uma folha do grafo. Logo, resta o caso em que o grafo possui exatamente duas folhas, F_1 e F_2. Seja $e_1 = \{x_1, y_1\}$ uma aresta de F_1 e seja $e_2 = \{x_2, y_2\}$ uma aresta de F_2, como ilustrado na Figura A.16 (esquerda). Conecte e_1 e e_2, como ilustrado na Figura A.16 (direita), para dar origem a um grafo, H, que é cúbico.

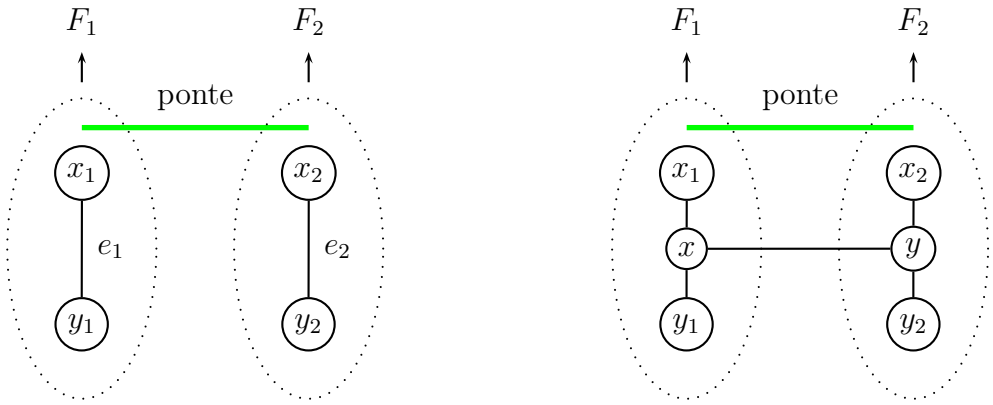


Figura A.16: Conexão das arestas e_1 e e_2.

Pelo Lema A.2.6, o grafo H não possui pontes. Então, o Teorema A.3.2 implica que H admite um emparelhamento perfeito. Afirma-se que, a partir de um emparelhamento perfeito qualquer em H, define-se um outro emparelhamento perfeito em G. De fato, se M é um emparelhamento perfeito em H, então considere o emparelhamento, M', em G tal que uma aresta comum a H e G está em M' se, e somente se, a aresta está em M. Para concluir a definição de M', distinguem-se dois casos: (a) $\{x, y\} \in M$ e (b) $\{x, y\} \notin M$. No caso (a), nota-se que todos os vértices de H, com exceção de x e y, estão cobertos em H por arestas comuns a M e M'. Como todo vértice de G é um vértice de H distinto de x e y, o emparelhamento M já é um emparelhamento perfeito em G. Logo, as arestas e_1 e e_2 não são incluídas em M. No caso (b), invoca-se o Lema A.3.1 para concluir que há um ciclo alternante, K, em H com respeito a M que contém $\{x, y\}$. Então, pode-se obter um outro emparelhamento perfeito, N, em H a partir de M substituindo cada aresta de K em M pela aresta consecutiva em K que não está em M. A aresta $\{x, y\}$ não estará em N e, portanto, o caso (b) se reduz ao caso (a). Logo, há um emparelhamento perfeito em G. □

APÊNDICE B – Análise Amortizada

A complexidade da grande maioria dos algoritmos discutidos nesta monografia foi estabelecida com um método de análise conhecido como *análise amortizada*. Uma discussão detalhada deste método pode ser encontrada em livros de projeto e análise de algoritmos (veja, por exemplo, o livro (CORMEM et al., 2009)) e ele é visto em muitos cursos de graduação em computação. No entanto, como o autor só teve contato com este método durante a realização deste trabalho, decidiu-se escrever este apêndice, de forma que leitores na mesma situação em que o autor se encontrava não precisem recorrer a outras fontes para obter o conhecimento mínimo necessário para entender as análises dos algoritmos deste texto.

B.1 Introdução

Em muitas aplicações envolvendo manipulação de dados por computador, uma sequência de operações é executada sobre uma estrutura de dados. Muito frequentemente, a avaliação da complexidade computacional desta sequência é realizada a partir de uma perspectiva bastante *pessimista*: considera-se o tempo de *pior caso* de cada operação da sequência e multiplica-se o número de operações da sequência pelo tempo de pior caso de cada operação, obtendo-se uma cota superior para o tempo gasto para executar todas as operações.

A *análise de pior caso* é muito importante na avaliação da complexidade de algoritmos utilizados em sistemas de tempo real, nos quais o tempo de resposta de cada operação é crítico. Além disso, ela é, em geral, fácil de ser realizada. No entanto, em muitas situações, somar os tempos de pior caso de cada operação de uma sequência pode ser uma forma pouco precisa de avaliação, pois ela ignora os efeitos correlacionados das operações. Como será visto nos exemplos dados nas seções vindouras, em *qualquer* sequência de operações sobre certas estruturas de dados, o tempo de cada operação não pode ser sempre o de pior caso.

Para se ter uma idéia mais realista e precisa da complexidade de tempo de uma sequência de operações sobre uma certa estrutura de dados, pode-se tentar calcular o tempo *médio* gasto com cada operação da sequência. Há, em geral, duas formas de se realizar este tipo de análise. Uma delas é conhecida como *análise de caso médio*, enquanto a outra é chamada de *análise (de complexidade) amortizada*. A diferença entre elas é bem sutil.

A análise de caso médio é destinada a cada operação individual e consiste em calcular o tempo médio gasto pela operação com relação aos tempos de execução da operação em *todas as possíveis entradas*. Para tal, assume-se uma distribuição de probabilidade para as entradas tal que cada entrada, E, possui uma probabilidade, P_E de ocorrer. Em seguida, calcula-se o *tempo médio*, $E(t)$, de execução da operação, usando a definição de valor esperado:

$$E(t) = \sum_{E \in \mathcal{E}} P_E \cdot t_E \,,$$

onde $\mathcal{E}$ é o conjunto de todas as entradas (de um mesmo tamanho tal que o tamanho é um parâmetro) da operação e t_E é o tempo gasto com a execução da operação na entrada E.

A maior dificuldade na utilização da análise de caso médio é justamente a definição de uma distribuição de probabilidade das entradas. Na prática é muito difícil garantir uma hipótese realista, pois, em geral, não se tem uma idéia precisa da frequência de ocorrência das entradas. Isto faz com que o problema de "precisão" da análise de pior caso não seja inteiramente bem resolvido com a análise de caso médio. Além disso, se esta análise for aplicada a uma sequência de operações, somar-se-ão os tempos médios individuais de cada operação. O valor resultante não é sempre igual ao tempo médio de execução da sequência *se as operações não forem independentes* (ou seja, a linearidade de E não se aplica a este caso).

A análise amortizada é uma alternativa à análise de caso médio, que é comumente aplicada ao cálculo da *complexidade amortizada* de uma sequência de operações. Neste tipo de análise, também se calcula o tempo médio, mas este tempo é igual à média aritmética dos tempos de execução das operações da *sequência de pior caso*[1]. Por exemplo, suponha que S seja *qualquer* sequência de n operações sobre uma dada estrutura de dados. Seja t o tempo gasto na execução das operações em S, na ordem dada. Note que t é simplesmente a soma dos tempos de execução de cada operação de S. Seja $t_{\max}$ o maior valor de t

[1]Uma sequência que possui tempo de execução maior ou igual ao de todas as sequências de mesmo tamanho.

entre *todas* as possíveis sequências, S, de m operações. Se S é uma sequência para a qual $t = t_{\max}$, então S é uma *sequência de pior caso*. A *complexidade amortizada* de S é igual a

$$\frac{t_{\max}}{m}.$$

Observe que o cálculo da complexidade amortizada não envolve distribuição de probabilidade das entradas do algoritmo. Ao invés disso, deve-se determinar uma sequência de pior caso e o tempo, $t_{\max}$, de execução dela. Idealmente, o tempo $t_{\max}$ não é igual à soma dos tempos de pior caso de cada operação da sequência. Caso fosse, ter-se-ía o mesmo resultado da análise de pior caso. A utilidade prática da análise amortizada reside justamente na capacidade de calcular $t_{\max}$ quando se puder mostrar que os tempos de execução das operações individuais das operações de S não são todos iguais ao de pior caso. Note também que a média de tempo dada pela complexidade amortizada é uma cota superior para a média do tempo de execução de qualquer sequência de m operações sobre a estrutura.

B.2 O método do potencial

Há três abordagens bem conhecidas para se calcular a complexidade amortizada de uma sequência de operações sobre uma dada estrutura de dados: a análise agregada, o método do contador e o método do potencial. Aqui, discute-se apenas o método potencial, pois ele é a única abordagem utilizada nas análises dos algoritmos descritos nesta monografia.

No método do potencial, toda operação da sequência possui um tempo "amortizado" de execução. Este tempo pode ser menor, igual ou maior do que o tempo gasto na execução da operação. No entanto, após qualquer operação da sequência, a soma dos tempos amortizados de todas as operações realizadas até o momento não pode ser inferior à soma dos tempos de execução das operações. Em outras palavras, a primeira soma é uma cota superior para a segunda. Se esta propriedade se mantém após a execução da última operação da sequência, então o tempo amortizado total é uma cota superior para o tempo total de execução. Além disso, se a propriedade for válida para *qualquer* sequência (em particular, para a de pior caso), então o tempo amortizado total dividido pelo número de operações da sequência de pior caso é uma cota superior para a complexidade amortizada da sequência.

De acordo com Tarjan, o método do potencial foi idealizado por Daniel Sleator (TAR-

JAN, 1985). O termo "potencial" advém de uma analogia com energia potencial. Mas, a melhor analogia para se entender a idéia por trás do método se utiliza de uma conta poupança. Neste caso, imagina-se cada operação da sequência como uma dívida. O valor da dívida é o *análogo do tempo (real) gasto pela operação da sequência*. Ao se pagar a dívida, tira-se dinheiro do bolso. A quantidade de dinheiro desembolsada para pagar a dívida é o *análogo ao tempo amortizado da operação* e ela pode ser maior ou menor do que o valor da dívida. Quando esta quantidade é maior, o excedente (ou seja, a diferença entre a quantidade desembolsada e o valor da dívida) é creditada na conta poupança. Quando ela é menor, o remanescente (ou seja, a diferença entre o valor da dívida e a quantidade desembolsada) é retirada da conta poupança. Se o valor desembolsado é igual ao valor da dívida, não há nada a creditar ou debitar da conta poupança. A quantidade de dinheiro na conta poupança é o *análogo do potencial*. Se sempre existir dinheiro (somando-se o dinheiro que se desembolsa com a quantidade que está na poupança) para pagar por cada dívida, então o total pago por todas as dívidas é sempre maior ou igual ao valor total das dívidas.

Formalmente, têm-se as seguintes definições:

Definição B.2.1. Seja D uma estrutura de dados sobre a qual m operações são executadas. Denote por D_0 o estado de D antes de qualquer operação ser executada e, por D_i, o estado de D imediatamente após a execução da i-ésima operação, onde $i = 1, \ldots, m$. Seja t_i o tempo gasto na execução da i-ésima operação, que faz com que o estado de D mude de D_{i-1} para D_i. Então, define-se uma *função potencial*, $\Phi_D : \mathcal{D} \to \mathbb{R}$, como aquela que mapeia cada possível estado, D_i, de D para um número real não-negativo, $\Phi_D(D_i)$, que se chama o *potencial associado com* D_i, onde $\mathcal{D}$ é o conjunto de todos os possíveis estados de D. O *tempo amortizado*, $\hat{t}_i$, da i-ésima operação com respeito à função Φ_D é igual a

$$\hat{t}_i = t_i + \Phi_D(D_i) - \Phi_D(D_{i-1})\,. \tag{B.1}$$

A Definição B.2.1 diz que o tempo amortizado de cada operação é igual ao tempo de execução da operação mais a diferença de potencial dos estados da estrutura de dados. Usando a Eq. (B.1), pode-se concluir que a soma do tempo amortizado das m operações

é igual a

$$\begin{aligned}\sum_{i=1}^{m}\hat{t}_i &= \sum_{i=1}^{m}\left(t_i + \Phi_D(D_i) - \Phi_D(D_{i-1})\right)\\ &= \sum_{i=1}^{m} t_i + \sum_{i=1}^{m}\Phi_D(D_i) - \sum_{i=1}^{m}\Phi_D(D_{i-1})\\ &= \sum_{i=1}^{m} t_i + \sum_{i=1}^{m-1}\Phi_D(D_i) - \sum_{i=2}^{m}\Phi_D(D_{i-1}) + (\Phi_D(D_m) - \Phi_D(D_0))\\ &= \sum_{i=1}^{m} t_i + \Phi_D(D_m) - \Phi_D(D_0)\,. \end{aligned} \tag{B.2}$$

A Eq. (B.2) mostra que se $\Phi_D(D_m) \geq \Phi_D(D_0)$, então $\sum_{i=1}^{m}\hat{t}_i \geq \sum_{i=1}^{m} t_i$, ou seja, o tempo amortizado total é uma cota superior para o tempo total de execução. Logo, o que importa é a diferença entre os potenciais da estrutura de dados antes e após as m operações. Infelizmente, na prática, nem sempre se sabe a quantidade, m, de operações que serão executadas. Como a condição $\Phi_D(D_m) \geq \Phi_D(D_0)$ *não* é suficiente para garantir que

$$\sum_{i=1}^{j}\hat{t}_i \geq \sum_{i=1}^{j} t_i\,,$$

para *todo* $j = 1, \ldots, m-1$, se existir j tal que $\sum_{i=1}^{j}\hat{t}_i < \sum_{i=1}^{j} t_i$, pode-se tomar a subsequência formada pelas j operações da sequência de m operações e se teria uma sequência de tamanho j em que o tempo amortizado total não é uma cota superior para o tempo total de execução. Fazendo uso da analogia com a conta poupança, a situação acima corresponde ao caso em que o dinheiro desembolsado somado ao saldo existente na conta poupança não é suficiente para pagar a j-ésima dívida, embora ao final dos m pagamentos, o total pago seja maior ou igual ao total das dívidas.

Uma forma de garantir que $\sum_{i=1}^{j}\hat{t}_i \geq \sum_{i=1}^{j} t_i$, para *todo* $j = 1, \ldots, m-1$, é definindo Φ_D tal que

$$\Phi_D(D_i) \geq \Phi_D(D_0)\,,$$

para todo $i = 1, \ldots, m$. Tomando $m = j$ na Eq. (B.2), pode-se concluir que, após a execução da j-ésima operação, tem-se $\sum_{i=1}^{j}\hat{t}_i \geq \sum_{i=1}^{j} t_i$. Note que esta afirmação é verdadeira para qualquer subsequência de tamanho j e para todo $j = 1, \ldots, m-1$. Fazendo uso da analogia com a conta poupança, a condição acima garante que sempre haverá dinheiro para pagar *cada* dívida, somando-se a quantidade desembolsada com aquela da poupança.

Quando o método do potencial é utilizado em uma análise amortizada, o "trabalho"

de quem realiza a análise reside em definir uma *boa* função potencial, ou seja, uma função que forneça uma cota superior, $\sum_{i=1}^{m} \hat{t}_i$, para $\sum_{i=1}^{m} t_i$, que seja tão próxima quanto possível de $\sum_{i=1}^{m} t_i$, pois a complexidade amortizada da sequência de operações é limitada superiormente por $(\sum_{i=1}^{m} \hat{t}_i)/m$. Este limite superior é o resultado da análise. Logo, a análise pelo método potencial produz um limite superior para o tempo médio de cada operação da sequência. Para garantir que o valor de Φ_D é sempre não-negativo, pode-se definir $\Phi_D(D_0) = 0$ e exigir que $\Phi_D(D_i) \geq \Phi_D(D_0)$, para todo $i = 1, \ldots, m$. Em geral, a tarefa de se determinar uma função potencial exige muita criatividade por parte de quem realiza a análise, o que quase sempre torna a análise amortizada bastante árdua para quem a utiliza.

B.3 Alguns exemplos

O primeiro exemplo de análise amortizada usando o método do potencial utiliza uma pilha que possui uma operação adicional de empilhamento. Mais especificamente, a pilha possui as operações PUSH, POP e MULTIPOP. A operação PUSH recebe como entrada um item, x, e o insere no topo da pilha. A operação POP não recebe nenhum dado de entrada e remove o item que se encontra no topo da pilha. Finalmente, a operação adicional, MULTIPOP, recebe um inteiro não-negativo, k, como entrada e remove os k itens do topo da pilha. Se k for maior ou igual ao número de itens da pilha, todos os itens da pilha são removidos.

As operações PUSH e POP são executadas em tempo constante. Já uma operação MULTIPOP gasta $\Theta(\min\{k, n\})$ unidades de tempo, onde n é o número de elementos da pilha, como pode ser facilmente constatado ao se examinar o pseudocódigo de MULTIPOP no Algoritmo B.1. A função STACK-EMPTY() executa em tempo constante e devolve o valor lógico verdadeiro se a pilha está vazia; caso contrário, a função devolve o valor lógico falso.

Algoritmo B.1 MULTIPOP(k)

```
enquanto não STACK-EMPTY() e k ≠ 0 faça
  POP()
  k ← k − 1
fim enquanto
```

Considere uma sequência de m operações sobre a pilha. Cada operação pode ser uma das três operações acima: PUSH, POP ou MULTIPOP. O tempo de pior caso de uma operação MULTIPOP na sequência é $\mathcal{O}(m)$, pois a pilha pode conter até $m - 1$ elementos

antes de MULTIPOP ser executada. Logo, se uma análise de pior caso fosse utilizada para determinar a complexidade (de pior caso) da sequência de operações, o tempo de pior caso de cada operação seria $\mathcal{O}(m)$ e, portanto, o tempo total de execução, no pior caso, seria $m \times \mathcal{O}(m) = \mathcal{O}(m^2)$. Embora essa análise esteja correta, a cota superior, $\mathcal{O}(m^2)$, não é justa. Isto porque não é possível se gastar $\Omega(m)$ unidades de tempo em cada operação da sequência.

Uma cota superior bem mais justa pode ser obtida através de uma análise amortizada. Para tal, define-se a função potencial, Φ_S, associada à pilha S de tal forma que $\Phi_S(S_i)$ é igual ao número de elementos do estado S_i de S. Antes de qualquer operação ser realizada, tem-se uma pilha vazia, representada pelo estado S_0. Por definição, tem-se $\Phi_S(S_0) = 0$. Como o número de elementos de uma pilha não pode jamais ser negativo, tem-se $\Phi_S(S_i) \geq 0 = \Phi_S(S_0)$, para todo $i = 1, 2, \ldots, m$. Logo, a função Φ_S está bem definida e, pela Eq. (B.2),

$$\sum_{i=1}^{m} \hat{t}_i \geq \sum_{i=1}^{m} t_i \,.$$

O próximo passo é usar Φ_S para calcular o tempo amortizado total das m operações sobre S.

Na análise que se segue, assume-se que o tempo gasto com as operações PUSH, POP e MULTIPOP é medido em termos de inserções e remoções de elementos do arranjo unidimensional que armazena os elementos da pilha. Neste "modelo de computação", o tempo gasto com uma operação PUSH ou POP é igual a 1 e o tempo gasto com uma operação MULTIPOP é igual a $\min\{k, n\}$, onde k é o parâmetro de entrada da função e n é o número de elementos da pilha. Se a i-ésima operação da sequência é um PUSH e a pilha contém exatamente n itens antes da operação ser realizada (isto é, S_{i-1} representa o estado de uma pilha com n itens), então a diferença de potencial é igual a $\Phi_S(S_i) - \Phi_S(S_{i-1}) = (n+1) - n = 1$. Logo, de acordo com a Eq. (B.1), o *tempo amortizado*, $\hat{t}_i$, da i-ésima operação (isto é, PUSH) é

$$\hat{t}_i = t_i + \Phi_S(S_i) - \Phi_S(S_{i-1}) = 1 + 1 = 2 \,.$$

Se a i-ésima operação da sequência é MULTIPOP(k) e se a pilha possui n itens, então o tempo gasto com a i-ésima operação é igual a $k' = \min\{k, n\}$ e a diferença de potencial é igual a

$$\Phi_S(S_i) - \Phi_S(S_{i-1}) = 0 - k' = -k' \,.$$

Note que houve um decréscimo de potencial. Isto significa que a operação MULTIPOP(k) será paga com o auxílio de recursos da "conta poupança", pois o *tempo amortizado*, $\hat{t}_i$, de

MULTIPOP) é

$$\hat{t}_i = t_i + \Phi_S(S_i) - \Phi_S(S_{i-1}) = k' - k' = 0\,.$$

Observe que $\hat{t}_i \leq t_i$, ou seja, não se paga pela operação sem os recursos da poupança. Se a i-ésima operação da sequência fosse POP, ter-se-ía $\hat{t}_i = 0$ também, pois POP é o mesmo que MULTIPOP(1). Logo, o tempo amortizado de cada uma das m operações da sequência é $\mathcal{O}(1)$.

Com o exposto acima, conclui-se que o *tempo amortizado total* de *qualquer* sequência de m operações sobre a pilha é igual a $\sum_{i=1}^{m} \hat{t}_i = m \times \mathcal{O}(1) = \mathcal{O}(m)$. Esta cota superior é bem mais justa do que aquela produzida pela análise de pior caso. Além disso, ela implica que a complexidade amortizada de cada operação da sequência é igual a $\mathcal{O}(m)/m = \mathcal{O}(1)$. Em outras palavras, algumas operações sobre a pilha podem ser "caras", mas, ao longo da sequência, haverá operações "baratas" que farão com que o tempo médio das operações da sequência seja constante! Observe que esta afirmação é válida para qualquer sequência válida.

Considere agora a implementação de um contador binário de k-bits. O contador é representado por um arranjo unidimensional, $A[0..k-1]$, de tamanho k. Um número, x, é representado em A com o bit menos significativo em $A[0]$ e o bit mais significativo em $A[k-1]$. Logo,

$$x = \sum_{l=0}^{k-1} A[l] \cdot 2^l\,.$$

Inicialmente, tem-se $x = 0$ e, portanto, $A[l] = 0$, para todo $l = 0, 1, \ldots, k-1$. Para incrementar o contador em uma unidade, utiliza-se o pseudocódigo no Algoritmo B.2. A Tabela B.1 mostra o que acontece com o contador quando ele é incrementado 16 vezes seguidas.

Algoritmo B.2 INCREMENTA(A, k)

```
l ← 0
enquanto (l < k) e (A[l] = 1) faça
   A[l] ← 0
   l ← l + 1
fim enquanto
se l < k então
   A[l] ← 1
fim se
```

No início de cada iteração do laço **enquanto** do Algoritmo B.2, deseja-se adicionar um 1 ao bit da posição l de A. Se $A[l]$ for igual a 1, então a adição de 1 a $A[l]$ muda o valor de $A[l]$ de 1 para 0 e produz um "vai-um" a ser adicionado ao bit da posição $l+1$

de A na próxima iteração do laço. Mas, se $A[l]$ for igual a 0, então o laço termina e, se $l < k$, onde k é o tamanho de A, o valor de $A[l]$ (que é igual a zero) muda para 1. Para analisar a complexidade de INCREMENTA(), assume-se que o tempo gasto por ela é igual ao número de posições do arranjo A que tiveram o valor modificado de 0 para 1 ou de 1 para 0. Observe que este tempo gasto é igual ao número de iterações do laço **enquanto** mais 1.

Tabela B.1: Um contador de 8 bits cujo valor varia de 0 a 16 através de uma sequência de 16 chamadas a INCREMENTA(). Os bits que mudam para gerar o próximo valor do contador são mostrados em negrito. O tempo gasto pela função INCREMENTA() para gerar o valor do contador na coluna mais à esquerda é mostrado na coluna mais à direita.

Valor	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	**Custo**
0	0	0	0	0	0	0	0	**0**	0
1	0	0	0	0	0	0	**0**	**1**	1
2	0	0	0	0	0	0	1	**0**	2
3	0	0	0	0	0	**0**	**1**	**1**	1
4	0	0	0	0	0	1	0	**0**	3
5	0	0	0	0	0	1	**0**	**1**	1
6	0	0	0	0	0	1	1	**0**	2
7	0	0	0	0	**0**	**1**	**1**	**1**	1
8	0	0	0	0	1	0	0	**0**	4
9	0	0	0	0	1	0	**0**	**1**	1
10	0	0	0	0	1	0	1	**0**	2
11	0	0	0	0	1	**0**	**1**	**1**	1
12	0	0	0	0	1	1	0	**0**	3
13	0	0	0	0	1	1	**0**	**1**	1
14	0	0	0	0	1	1	1	**0**	2
15	0	0	0	**0**	**1**	**1**	**1**	**1**	1
16	0	0	0	1	0	0	0	0	5

Uma única execução de INCREMENTA() gasta k unidades de tempo, *no pior caso*. Isto ocorre quando todos os bits são iguais a 1, exceto o mais significativo. Logo, em uma análise de pior caso, uma sequência de m chamadas a INCREMENTA() gasta $\Theta(m \cdot k)$ unidades de tempo. Entretanto, assim como no exemplo da pilha, o uso de análise amortizada produz uma cota superior bem mais justa para o tempo total gasto por INCREMENTA() em m operações.

A observação chave é que não é possível se ter o pior caso em *todas* as operações da sequência, pois nem todos os bits de A são mudados em cada execução de INCREMENTA(). De forma geral, para $l = 0, 1, \ldots, k-1$, o valor de $A[l]$ muda exatamente $\lfloor m/2^l \rfloor$ vezes em uma sequência de m chamadas à função INCREMENTA() *para um contador com valor*

inicial igual a zero, pois o l-ésimo bit de A só pode mudar após o valor do contador ser somado a um múltiplo de 2^l diferente de zero. Além disso, quando $l \geq k$, o bit $A[l]$ não existe e, portanto, o seu valor não pode mudar. Os fatos acima podem ser observados na Tabela B.1.

Seja $\Phi_A : \mathcal{A} \to \mathbb{R}$ a função potencial que mapeia cada estado, $A_i \in \mathcal{A}$, do contador para um número real, tal que $\Phi_A(A_i)$ é igual ao número de bits 1 do contador representado por A_i, para todo $i = 0, 1, \ldots, m$. Note que se A_i for o estado do contador após a i-ésima chamada a INCREMENTA() com um contador de valor inicial igual a zero, então o valor de $\Phi_A(A_i)$ é o mesmo que o número de bits 1 do contador *imediatamente após a i-ésima chamada a* INCREMENTA() *terminar de executar*, para todo $i = 1, \ldots, m$, e o valor de $\Phi_A(A_0)$ pode ser definido como 0. Como $\Phi_A(A_i) \geq 0 = \Phi_A(A_0)$, a função Φ_A está bem definida e

$$\sum_{i=1}^{m} \hat{t}_i \geq \sum_{i=1}^{m} t_i \,,$$

pela Eq. (B.2), o que permite que o método do potencial seja utilizado, com a função Φ_A, na determinação da complexidade amortizada de cada operação INCREMENTA() da sequência.

Suponha que a i-ésima execução de INCREMENTA() mude o valor de exatamente b_i bits de 1 para 0. Então, por definição, o tempo, t_i, de execução da i-ésima execução de INCREMENTA() é, no máximo, igual a b_i+1, pois além de mudar o valor de b_i bits de 1 para 0, INCREMENTA() muda o valor de exatamente *um* bit 0 para 1, exceto quando o contador já tiver atingido o valor máximo permitido. Então, para $i = 1, \ldots, m$, distingue-se dois casos:

1) $\Phi_A(A_i) = 0$ e

2) $\Phi_A(A_i) > 0$.

No caso 1), a i-ésima operação mudou o valor de todos os bits de A, que eram todos iguais a 1, para 0 e, portanto, tem-se que $\Phi(A_{i-1}) = b_i = k$, onde k é o tamanho de A. No caso 2), tem-se

$$\Phi_A(A_i) = \Phi_A(A_{i-1}) - b_i + 1 \,.$$

O termo $\Phi_A(A_{i-1}) - b_i$ equivale ao número de bits 1 do contador representado por A_{i-1} que não não mudam de valor *após* a i-ésima operação, ou seja, o número de bits 1 de A_{i-1} que permanecem com o mesmo valor 1 em A_i *após* a i-ésima operação ser executada. O valor $\Phi_A(A_{i-1}) - b_i$ é adicionado a 1 para se levar em conta o bit 0 que muda para

1 durante a i-ésima execução de INCREMENTA(). Nos dois casos acima, pode-se concluir que

$$\Phi_A(A_i) \leq \Phi_A(A_{i-1}) - b_i + 1\,.$$

Logo,

$$\Phi_A(A_i) - \Phi_A(A_{i-1}) \leq 1 - b_i$$

e

$$\hat{t}_i = t_i + \Phi_A(A_i) - \Phi_A(A_{i-1}) \leq (b_i + 1) + (1 - b_i) = 2\,.$$

Daí, tem-se que o tempo amortizado total das m chamadas a INCREMENTA() é dado por

$$\sum_{i=1}^{m} \hat{t}_i = \sum_{i=1}^{m} 2 = 2 \cdot m\,,$$

o que implica que a complexidade amortizada de uma chamada a INCREMENTA() é constante.

Finalmente, suponha que o contandor binário seja modificado para incluir uma operação que decrementa o valor do contador. A implementação mais óbvia do contador modificado pode ser pouco eficiente, mesmo do ponto de vista de complexidade amortizada. Isto porque se, em uma sequência de m operações, as n primeiras apenas incrementarem o contador e as $m - n$ últimas alternarem as operações de decremento e incremento, com n igual a maior potência de 2 menor ou igual a $m/2$, o valor do contador variará entre $2^h - 1$ e 2^h nas $m - k$ últimas operações, onde $h = \lg n$. Quando m é uma potência de 2, tem-se que o tempo gasto com cada uma das $m/2$ últimas operações é $m/2$. Logo, o tempo amortizado total é $\Theta(m^2)$ e, portanto, a complexidade amortizada de cada operação é $\Theta(m)$.

Uma alternativa eficiente consiste em representar o valor do contador por dois arranjos unidimensionais, F e G, de k bits cada tais que para qualquer posição l, com $l = 0, 1, \ldots, k-1$, no máximo um dos bits, $F[l]$ ou $G[l]$, é igual a 1. O valor do contador é igual a $F - G$. Os pseudocódigos das operações INCREMENTA() e DECREMENTA() estão nos Algoritmos B.3 e B.4. A função INCREMENTA() se vale do fato que $(F - G) + 1$ é igual a

$$(F - (h - 1)) - (G - h)\,,$$

onde $h < G$. Por sua vez, a função DECREMENTA() se vale do fato que $(F - G) - 1$ é igual a

$$(F - h) - (G - (h - 1))\,,$$

onde $h < G$.

Algoritmo B.3 INCREMENTA(F, G, k)

```
l ← 0
enquanto (l < k) e (F[l] = 1) faça
   F[l] ← 0
   l ← l + 1
fim enquanto
se l < k então
   se G[l] = 1 então
      G[l] ← 0
   senão
      F[l] ← 1
   fim se
fim se
```

Algoritmo B.4 DECREMENTA(F, G, k)

```
l ← 0
enquanto (l < k) e (G[l] = 1) faça
   G[l] ← 0
   l ← l + 1
fim enquanto
se l < k então
   se F[l] = 1 então
      F[l] ← 0
   senão
      G[l] ← 1
   fim se
fim se
```

Por exemplo, considere uma sequência de 6 operações sobre o contador binário duplo: INCREMENTA(), INCREMENTA(), INCREMENTA(), DECREMENTA(), DECREMENTA() e INCREMENTA() a partir dos valores iniciais $F = 10001$ e $G = 01100$. Após a primeira operação, obtém-se $F = 100\mathbf{10}$ e $G = 01100$. Após a segunda operação, obtém-se $F = 10011$ e $G = 01100$. Após a terceira operação, obtém-se $F = 100\mathbf{00}$ e $G = 01\mathbf{0}000$. Após a quarta operação, obtém-se $F = 10000$ e $G = 0100\mathbf{1}$. Após a quinta operação, obtém-se $F = 10000$ e $G = 010\mathbf{10}$. Após a sexta operação, obtém-se $F = 1000\mathbf{1}$ e $G = 01010$. Logo, o valor de F foi incrementado em uma unidade na primeira, segunda e sexta operações. Na terceira operação, os valores de F e G foram decrementados em 3 e 4 unidades, respectivamente. Na quarta e na quinta operações, o valor de G foi incrementado em uma unidade. Os bits de F e G modificados pela i-ésima operação foram mostrados acima em destaque (com negrito).

Seja $\Phi_A : \mathcal{A} \to \mathbb{R}$ a função potencial que mapeia cada estado, $(F_i, G_i) \in \mathcal{A}$, do contador duplo para o número de bits 1 dos estados F_i e G_i dos dois arranjos, F e G, respectivamente, para todo $i = 0, 1, \ldots, m$. Note que se (F_i, G_i) for o estado do contador duplo após a i-ésima operação sobre o contador duplo, em uma sequência de m operações de incremento e decremento com valor inicial igual a $(0, 0)$, então o valor de $\Phi_A(F_i, G_i)$ é o mesmo que o número de bits 1 em F e G *imediatamente após a i-ésima operação ser executada*, para todo $i = 1, \ldots, m$. Como $\Phi_A(F_i, G_i) \geq 0$, para todo $i = 1, \ldots, m$, definindo-se $\Phi_A(F_0, G_0) = 0$, tem-se que a função Φ_A está bem definida e, portanto, pela Eq. (B.2),

$$\sum_{i=1}^{m} \hat{t}_i \geq \sum_{i=1}^{m} t_i \,,$$

o que permite que o método do potencial seja utilizado, com a função Φ_A e de forma semelhante à anterior, na determinação da complexidade amortizada de cada operação da sequência.

Por definição, o tempo, t_i, gasto na i-ésima operação é igual ao número de bits cujo valor mudou (seja de 1 para 0 ou de 0 para 1) em ambos F e G. Para todo $i = 1, \ldots, m$, denote por p_i e n_i o número de bits em F e G combinados que teve o valor mudado de 0 para 1 e de 1 para 0, respectivamente, na i-ésima operação da sequência. Então, $t_i = p_i + n_i$. Além disso, tem-se que 1) $\Phi_A(F_i, G_i) = 0$ ou 2) $\Phi_A(F_i, G_i) > 0$. Se o caso 1 ocorrer, então a i-ésima operação mudou o valor de todos os bits 1 de F e G combinados para 0 e, portanto, tem-se que $\Phi(F_{i-1}, G_{i-1}) = p_i + n_i = t_i$. Se o caso 2 ocorrer, tem-se que

$$\Phi_A(F_i, G_i) = \Phi_A(F_{i-1}, G_{i-1}) + p_i - n_i \,.$$

Logo,

$$\hat{t}_i = t_i + \Phi_A(F_i, G_i) - \Phi_A(F_{i-1}, G_{i-1}) = p_i + n_i + p_i - n_i = 2 \cdot p_i \,,$$

ou seja, o tempo amortizado da i-ésima operação é igual ao número de bits cujo valor foi mudado de 0 para 1 na i-ésima operação. Mas, ao se examinar os pseudocódigos dos Algoritmos B.3 e B.4, conclui-se que há, no máximo, uma mudança de bit 0 para 1. Isto é, $p_i \leq 1$. Isto implica que $\hat{t}_i \leq 2$. Logo, a complexidade amortizada de cada operação é $\Theta(1)$.

Este último exemplo ilustra uma faceta importante da análise amortizada. Ao se tentar analisar a complexidade amortizada das operações de uma certa estrutura de dados, obtém-se um conhecimento sobre a estrutura e suas operações que, em geral, leva a otimizações da estrutura ou do código de suas operações. Nesta monografia, a análise

amortizada foi utilizada na determinação da complexidade (amortizada) das operações de manutenção de árvores splay, árvores ST e de uma estrutura de dados que suporta consultas sobre a conectividade de vértices de uma floresta de árvores geradoras, na qual arestas estão sendo inseridas e removidas entre consultas (*o problema da conectividade dinâmica*).