Instituto de Computação
Universidade Estadual de Campinas

Complexidade de Construção de Árvores PQR

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por João Paulo Pereira Zanetti e aprovada pela Banca Examinadora.

Campinas, 27 de fevereiro de 2012.

João Meidanis (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Zanetti, João Paulo Pereira, 1987-
Z163c Complexidade de construção de árvores PQR / João Paulo Pereira Zanetti. – Campinas, SP : [s.n.], 2012.

Orientador: João Meidanis.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

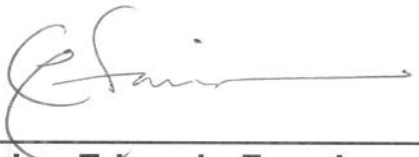
1. Algoritmos on-line. 2. Estruturas de dados (Computação). I. Meidanis, João, 1960-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Complexity of PQR tree construction
Palavras-chave em inglês:
Online algorithms
Data structures (Computer science)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
João Meidanis [Orientador]
Carlos Eduardo Ferreira
Guilherme Pimentel Telles
Data de defesa: 27-02-2012
Programa de Pós-Graduação: Ciência da Computação

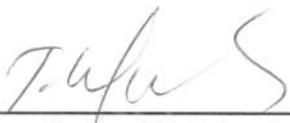
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 27 de Fevereiro de 2012, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Carlos Eduardo Ferreira
IME / USP

Prof. Dr. Guilherme Pimentel Telles
IC / UNICAMP



Prof. Dr. João Meidanis
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Complexidade de Construção de Árvores PQR

João Paulo Pereira Zanetti[1]

Fevereiro de 2012

Banca Examinadora:

- João Meidanis (Orientador)
- Carlos Eduardo Ferreira
  Instituto de Matemática e Estatística — USP
- Guilherme Pimentel Telles
  Instituto de Computação — Unicamp
- Celina Miraglia Herrera de Figueiredo (Suplente)
  Programa de Engenharia de Sistemas e Computação, COPPE — UFRJ
- Célia Picinin de Mello (Suplente)
  Instituto de Computação — Unicamp

[1]Suporte financeiro de: Bolsas do CNPq (processo 132182/2010-6) 03—08/2010 e FAPESP (processo 2010/04071-1) 09/2010—02/2012.

Resumo

As árvores PQR são estruturas de dados usadas para tratar o problema dos uns consecutivos e problemas relacionados. Aplicações incluem reconhecimento de grafos de intervalos, de grafos planares, e problemas envolvendo moléculas de DNA. A presente dissertação busca consolidar o conhecimento sobre árvores PQR e, principalmente, sua construção incremental, visando fornecer uma base teórica para o uso desta estrutura em aplicações.

Este trabalho apresenta uma descrição detalhada do projeto do algoritmo para construção online de árvores PQR, partindo de uma implementação inocente das operações sugeridas e refinando sucessivamente o algoritmo até alcançar a complexidade de tempo quase-linear. Neste projeto, lidamos com um obstáculo que surge com a utilização de estruturas de union-find que não havia sido tratado anteriormente. A demonstração da complexidade de tempo do algoritmo apresentada aqui também é nova e mais clara. Além disso, o projeto é acompanhado de uma implementação em Java dos algoritmos descritos.

Abstract

PQR trees are data structures used to solve the consecutive ones problem and other related problems. Applications include interval or planar graph recognition, and problems involving DNA molecules. This dissertation aims at consolidating existing and new knowledge about PQR trees and, primarily, their online construction, thus providing a theoretical basis for the use of this structure in applications.

This work presents a detailed description of the online PQR tree construction algorithm's design, starting with a naive implementation of the suggested operations and refining them successively, culminating with an almost-linear time complexity. In this project, we dealt with an obstacle that arises with the use of union-find structures and that has never been addressed before. The proof presented here for the time complexity is also novel and clearer. Furthermore, the project is accompanied by a Java implementation of all the algorithms described.

Agradecimentos

Eu gostaria de agradecer a todos que tornaram possível este trabalho.

Ao CNPq e à FAPESP, pelo suporte financeiro.

À Unicamp, especialmente o Instituto de Computação, e todos seus professores e funcionários que conheci durante estes anos, pelo privilegiado ambiente de estudo e trabalho.

Ao meu orientador, João Meidanis, pelo incentivo e paciência durante este tempo todo.

Aos meus pais, João e Janete, que sempre me apoiaram e acreditaram em mim.

E, por fim, a todos os meus amigos que participaram deste período tão importante da minha vida.

A cada um de vocês, muito obrigado.

Sumário

Capítulo 1

Propriedade dos uns consecutivos

O problema dos uns consecutivos pode ser descrito como segue: dados um conjunto U, de n elementos, e uma coleção $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ de m subconjuntos de U, o algoritmo deve decidir se existe uma permutação dos elementos de U em que os elementos de cada $C \in \mathcal{C}$ apareçam consecutivamente. Idealmente, o algoritmo determinaria também todas as permutações que satisfaçam estes critérios. Estas permutações são chamadas válidas. O nome do problema vem de sua caracterização original, em matrizes binárias, em que o objetivo é permutar as linhas da matriz de forma que, em cada coluna, os uns fiquem consecutivos.

Há diversas aplicações para este problema. Um exemplo é o de um sistema de recuperação de informação [9]. Um banco de dados contém um conjunto U de registros em disco. O sistema responde a consultas sobre informações contidas nestes registros. Uma consulta corresponde a um subconjunto $C \subset U$ de registros necessários para respondê-la. Este sistema visa ordenar seus registros em disco de forma que, para cada consulta C, todos os seus registros estejam armazenados consecutivamente, o que permite que cada consulta seja respondida com apenas um movimento da cabeça de leitura no disco. Esta estratégia diminui o tempo para a resposta às consultas.

Para ilustrar este exemplo, suponha $U = \{A, B, C\}$. Se as únicas consultas são $\{A\}$, $\{B\}$ e $\{C\}$, todas as seis permutações possíveis de U são válidas. Se a consulta $\{A, B\}$ for incluída, A e B têm que ser consecutivos. Portanto, passam a ser quatro as permutações válidas: ABC, BAC, CAB e CBA. Incluindo também a consulta $\{B, C\}$, somente ABC e CBA são válidas. Entretanto, se o sistema precisar responder também à consulta $\{A, C\}$, os registros não poderão ser ordenados satisfatoriamente, pois não há permutação válida.

Para instâncias maiores, o número de permutações válidas pode aumentar exponencialmente e, portanto, faz-se necessário um algoritmo mais elaborado do que testar todas as permutações. Neste mestrado, estudamos uma estrutura de dados chamada árvore

PQR para tratar este problema eficientemente.

Em 1965, Fulkerson e Gross apontaram uma aplicação para o problema dos uns consecutivos, o reconhecimento de grafos de intervalos [7]. Além disso, eles exploram várias propriedades do problema e apresentam um algoritmo para resolvê-lo.

Kendall em 1969 [14], indica uma relação entre o problema dos uns consecutivos a datação relativa de objetos em tumbas, por exemplo. Cada elemento do conjunto U corresponderia a uma tumba encontrada e cada restrição em $\mathcal{C}$ corresponderia aos objetos encontrados nas tumbas, isto é, os elementos em uma mesma restrição são tumbas próximas cronologicamente. Assim, uma permutação válida de U, que Kendall chama de *forma de Petrie*, determinaria uma ordem cronológica para as tumbas e, consequentemente, aos objetos. Entretanto, Kendall ressalta que, ao fazer esse tipo de datação, nem sempre se tem uma entrada compatível e se busca uma solução aproximada para o problema.

Ghosh, em 1972 [9], aponta a importância em sistemas de recuperação de informação da propriedade dos uns consecutivos, que ele chama de propriedade de recuperação consecutiva. Sendo U o conjunto de registros e $\mathcal{C}$ as consultas ou os arquivos do sistema, se $(U, \mathcal{C})$ tem a propriedade dos uns consecutivos, então os registros podem ser armazenados em uma mídia de armazenamento linear de forma que cada consulta possa ser feita lendo registros consecutivos, sem a duplicação de registros. Ghosh prova também alguns teoremas sobre a propriedade, relacionados à inserção de novas consultas ou à partição do conjunto de consultas.

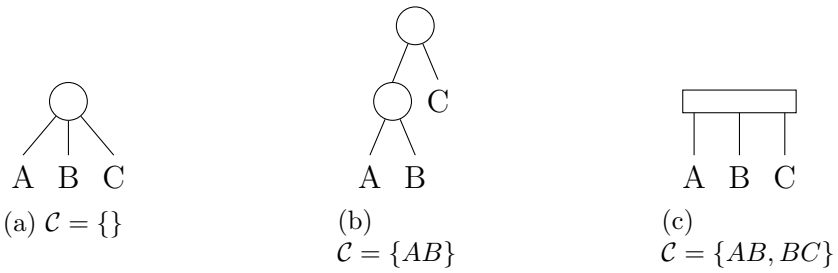
Em 1976, Booth e Lueker mostraram como, além de grafos de intervalos, é possível usar o problema dos uns consecutivos para reconhecer grafos planares [2].

Greenberg e Istrail [11] relacionam o problema dos uns consecutivos com o mapeamento genético utilizando marcadores moleculares.

Chauve e Tannier em 2008 [4] apresentaram um método baseado em árvores PQR na reconstrução de genomas ancestrais, a partir de genomas conhecidos de espécies descendentes. Os nós do tipo R são utilizados em uma estratégia de branch-and-bound para encontrar o número máximo de restrições que sejam compatíveis, que, como já foi visto, é um problema NP-difícil.

Outra aplicação é na área de visualização de dados. Silva, Melo e Meidanis [22] usaram as árvores PQR para reordenar automaticamente as linhas e colunas de matrizes de acordo com similaridades, de forma a facilitar a análise visual dos dados representados nestas matrizes.

Podemos classificar os algoritmos para resolver o problema dos uns consecutivos em duas categorias: online e offline. Um método offline recebe todas as restrições de uma vez e retorna a árvore resultante. Em contrapartida, um método online, ou incremental, recebe as restrições uma de cada vez, e a cada restrição devolve a árvore relativa às restrições já dadas. O cerne de um método online é um algoritmo que adiciona uma



(a) $\mathcal{C} = \{\}$ (b) $\mathcal{C} = \{AB\}$ (c) $\mathcal{C} = \{AB, BC\}$

Figura 1.1: Árvores PQ correspondentes ao exemplo de Ghosh.

nova restrição a uma árvore dada, no que chamamos de redução da árvore. Quando avaliamos a complexidade de um algoritmo para resolver o problema dos uns consecutivos, consideramos o tamanho da entrada $s = n + m + \sum_{i=1}^{m} |C_i|$.

Em 1976, as árvores PQ foram introduzidas por Booth e Lueker em um artigo pioneiro [2]. As árvores PQ conseguem representar todas as permutações válidas de forma eficiente e compacta. Uma árvore PQ é uma estrutura de dados em forma de árvore na qual as folhas são os elementos de U e as permutações válidas são obtidas pela ordem das folhas da árvore submetida a duas operações de equivalência: permutação dos filhos de um nó do tipo P e reversão dos filhos de um nó do tipo Q. Representamos nós do tipo P com círculos e nós do tipo Q com retângulos. Na Figura 1.1, temos as árvores PQ correspondentes às restrições do exemplo que inicia este capítulo. Note que, para $\mathcal{C} = \{AB, AC, BC\}$, não existe árvore PQ, pois não há permutação válida. Neste trabalho, Booth e Lueker apresentam o primeiro algoritmo linear para resolver o problema dos uns consecutivos, construindo uma árvore PQ incrementalmente.

Novick, em 1989 [20], desenvolveu uma estrutura derivada da árvore PQ chamada de árvore PQ generalizada, ou gPQ. Esta estrutura resolve um problema relacionado ao dos uns consecutivos. Em vez de representar permutações válidas, a árvore gPQ representa todos os subconjuntos de U que tem interseção trivial com todos os elementos de $\mathcal{C}$. Novick apresenta um algoritmo online que constrói esta árvore em $O(nm)$. Além disso, mostra como utilizar a estrutura para resolver versões restritas do problema da interseção trivial e discute a aplicação das árvores gPQ para a construção de grafos de sobreposição.

As árvores PQR foram apresentadas em 1996 por Meidanis e Munuera, com um algoritmo offline quadrático para sua construção [18, 19]. Em um artigo de 1998, Meidanis, Porto e Telles se aprofundam na teoria das árvores PQR [16]. As árvores PQR são uma generalização das árvores PQ, introduzindo um novo tipo de nó que indica que não existe uma permutação válida para as restrições dadas. Enquanto para entradas cujas restrições não sejam compatíveis, não existe uma árvore PQ, sempre existe uma árvore PQR correspondente, com nós tipo R indicando a incompatibilidade. Nós do tipo R são representados por um círculo com a letra R. Um exemplo de árvore PQR pode ser visto

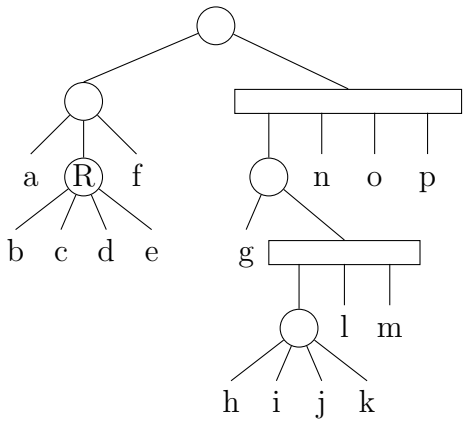


Figura 1.2: Exemplo de árvore PQR sobre $U = abcdefghijklmnop$ e com restrições $\mathcal{C} = \{abcdef, bc, cd, de, ce, ghijklmn, no, op, hijkl, lm\}$.

na Figura 1.2.

Em um trabalho de 2000, Habib e colegas demonstram como reconhecer diversas classes de grafos utilizando algoritmos baseados na Lex-BFS [12]. Além disso, exibem um algoritmo para reconhecer a propriedade dos uns consecutivos, interpretando a entrada como um grafo de cliques maximais e testando se esse grafo é de intervalos.

Em 2002, Telles propôs um algoritmo online quase-linear para a redução de árvores PQR [25]. Este algoritmo é parecido com o de Booth e Lueker, mas é mais simples, utilizando quatro operações de redução otimizadas em vez dos nove padrões de substituição de Booth e Lueker. Em adição, utiliza a estrutura de union-find para evitar manter referências para o nó pai em cada nó da árvore. Este tipo de estrutura permite armazenar tais referências, mas não em tempo constante. A complexidade das operações em estruturas de union-find foi provada por Tarjan em 1975 [23]. Uma versão mais simples dessa demonstração pode ser encontrada no livro de Cormen e colegas [5, Cap. 21], que servirá aos nossos propósitos. A complexidade amortizada de cada operação de *find* é de $O(\alpha(s))$, onde $\alpha(x)$, que cresce muito lentamente, é a inversa de uma função de Ackermann.

Mais tarde, Telles e Meidanis [24] melhoraram a descrição do algoritmo e simplificaram as operações.

Outro algoritmo importante para resolver o problema dos uns consecutivos é o teste de Hsu, de 2002 [13]. O algoritmo de Hsu é offline e, além de testar para a propriedade, constrói a árvore PQ correspondente em tempo linear. Este algoritmo divide a entrada em instâncias primas, que são então simples de se processar. Depois, para recompor a árvore PQ, substituem-se os nós referentes a uma dessas instâncias primas pela subárvore dessa instância.

McConnell, em 2004, desenvolveu independentemente uma estrutura também chamada

de árvore PQ generalizada [15], que é idêntica à árvore PQR. McConnell parece desconhecer os trabalhos anteriores de Novick sobre árvores gPQ [20] e de Meidanis, Porto e Telles sobre árvores PQR [16]. Neste trabalho, ele apresenta um algoritmo offline linear para construir uma árvore PQ generalizada, além de uma caracterização por estruturas proibidas em um grafo de incompatibilidade construído a partir da coleção de restrições. Este algoritmo utiliza o algoritmo linear de Dahlhaus [6] para a construção do grafo de sobreposição estrita, porém temos dúvidas quanto ao uso feito por Dahlhaus de ordenações em tempo linear e sua complexidade real. Em 2008, Charbit e colegas [3] apresentaram uma versão para o algoritmo de Dahlhaus que resolvem nossas dúvidas, além de ser mais simples e de se adaptar melhor ao algoritmo de McConnell.

Em 2009, em um trabalho de iniciação científica, o autor desta dissertação aprofundou o estudo de algoritmos online e offline para a construção de árvores PQR. Uma descrição detalhada das operações de transformação do algoritmo online foi redigida, juntamente com esboços da análise de sua complexidade e prova de corretude. A respeito do algoritmo offline, foram exploradas abordagens para diminuir sua complexidade de tempo, estudando a construção do grafo de sobreposição estrita e das classes de gêmeos, e quais delas podem ser aproveitadas nas chamadas recursivas [26].

A árvore PQR, pela presença do nó do tipo R, oferece mais informações do que a árvore PQ quando a entrada não tem nenhuma permutação válida. A existência de um nó R na árvore indica que não existe uma permutação válida para o conjunto de restrições coberto por aquele nó. Meidanis, Porto e Telles [16] sugerem o uso do nó R para encontrar erros em experimentos de hibridização.

Sabe-se que diversas variantes do problema destinadas a aproximar uma solução são NP-difíceis, por exemplo, encontrar o máximo subconjunto dos elementos que admita uma solução, encontrar o número mínimo de mudanças de 0 para 1 na matriz que produza uma solução [8], ou minimizar o número de blocos de uns em uma matriz [10]. Não se sabe a complexidade de encontrar o maior número de restrições que sejam compatíveis (ou seja, resultem em uma árvore PQ não nula). É possível que árvores PQR possam ajudar a resolver ou aproximar estas variantes.

As árvores PQR tencionam também ser de fácil implementação em relação às árvores PQ. Enquanto o algoritmo de Booth e Lueker exige a verificação de nove padrões, as árvores PQR utilizam cinco padrões.

As principais contribuições deste trabalho são:

- Uma descrição detalhada do projeto do algoritmo online, com implementação.
- Correção a trabalhos anteriores sobre o uso de estruturas de union-find.
- Crítica ao uso de radix sort nos trabalhos de McConnell [15] e Dahlhaus [6].

O resto desta dissertação está organizado como segue. No Capítulo 2, apresentamos algumas definições e teoremas importantes para o trabalho. A seguir, no Capítulo 3, apresentamos uma versão simples do algoritmo online, que utilizamos para ilustrar a redução de árvores PQR e provar a corretude das operações, apesar de sua complexidade de tempo ser maior que a desejada. No Capítulo 4, discutimos como o algoritmo online pode se tornar mais eficiente. No Capítulo 5, reproduzimos o algoritmo online final e provamos sua complexidade de tempo quase-linear. O Capítulo 6 apresenta nossas conclusões.

Capítulo 2

Fundamentação teórica

Neste capítulo, reunimos algumas definições básicas e teoremas utilizados ao longo da dissertação. A maior parte das informações deste capítulo vêm dos trabalhos de Meidanis, Porto e Telles [16] e de Telles e Meidanis [24], e mais propriedades das árvores PQR podem ser encontradas naqueles artigos.

2.1 Definições relativas a conjuntos

Neste trabalho, utilizamos o termo **coleção** para designar conjuntos de conjuntos. Mais especificamente, tratamos somente de dois tipos de conjuntos nesta dissertação: subconjuntos de U e coleções de subconjuntos de U.

A **união não-disjunta** de dois conjuntos A e B é uma operação que só pode ser efetuada se $A \cap B \neq \emptyset$. Neste caso, o resultado é igual a $A \cup B$. Denotamos a união não-disjunta de A e B por $A \uplus B$, onde o símbolo '+' serve para lembrar-nos que A e B não podem ser disjuntos.

De forma semelhante, a **diferença não-contida** de dois conjuntos A e B é uma operação que só pode ser efetuada se $B \not\subseteq A$, quando seu resultado é igual a $A \setminus B$. Denotamos a diferença não-contida de A e B por $A \circledcirc B$, com o símbolo '$\circ$' para lembrar-nos que B não pode estar contido em A.

Dado um conjunto U, seus **subconjuntos triviais** são U, o conjunto vazio e todos os conjuntos $\{x\}$ tais que $x \in U$.

Uma coleção $\mathcal{C}$ de subconjuntos de U é **completa** se contém todos os subconjuntos triviais de U e é fechada para as operações de união não-disjunta, intersecção e diferença não-contida. O **fecho** $\overline{\mathcal{C}}$ de $\mathcal{C}$ é a menor coleção completa que contém $\mathcal{C}$.

Dois conjuntos A e B se **sobrepõem estritamente** quando $A \cup B \neq \emptyset$ e nenhum dos conjuntos está contido no outro.

Dizemos que dois conjuntos A e B são **ortogonais**, denotado por $A \perp B$, quando A

e B não se sobrepõem estritamente, isto é, quando $A \subseteq B$, $B \subseteq A$, ou $A \cup B = \emptyset$. Nós chamamos $\mathcal{C}^{\perp}$ a coleção dos conjuntos ortogonais a todos os conjuntos em $\mathcal{C}$.

2.2 Árvores PQR

Uma **árvore PQR** sobre um conjunto U é uma árvore enraizada cujas folhas são os elementos de U, sem repetição, e com três tipos de nós internos: P, Q e R, sujeitos às seguintes restrições:

- Cada nó P tem no mínimo dois filhos.
- Cada nó Q tem no mínimo três filhos.
- Cada nó R tem no mínimo três filhos.

Duas árvores PQR são **equivalentes** se e somente se uma pode ser transformada na outra através da aplicação de zero ou mais das seguintes operações de equivalência:

- Permutações arbitrárias dos filhos de um nó P.
- Reversão dos filhos de um nó Q.
- Permutações arbitrárias dos filhos de um nó R.

Em geral representamos árvores PQR indicando o tipo do nó pelo seu formato: nó P por um círculo, nó Q por um retângulo e nó R por um círculo com a letra R.

Uma **árvore PQ** é uma árvore PQR sem nós R e representa todas as permutações válidas para um conjunto U e uma coleção $\mathcal{C}$ de subconjuntos de U.

A **fronteira** de uma árvore PQR (ou PQ) é a permutação de U obtida lendo as folhas da árvore da esquerda para a direita. Dada uma árvore PQ T, a classe de permutações válidas que T representa é a classe de permutações obtida a partir das fronteiras de T e de todas as árvores equivalentes a T.

O **conjunto de folhas descendentes** de um nó v é denotado $\hat{v}$.

A **subárvore pertinente** em relação a um conjunto C é a subárvore de altura mínima cuja fronteira contém todos os elementos de C.

Assim como para coleções, podemos definir o **fecho** de uma árvore PQR T, denotado por $\overline{T}$. Para isso, antes precisamos definir três coleções relativas a um nó v de T:

- $\mathit{Trivial}(v)$, que contém os subconjuntos triviais de $\hat{v}$.
- $\mathit{Consec}(v)$, que contém todos os conjuntos da forma $\bigcup_{v \in S} \hat{v}$ tais que S é um conjunto de filhos consecutivos de v.

- $Pot(v)$, que contém todos os conjuntos da forma $\bigcup_{v \in S} \hat{v}$ tais que S é um conjunto arbitrário de filhos de v.

Com estas definições, definimos o fecho $\overline{v}$ de um nó v como na Equação (2.1)

$$\overline{v} = \begin{cases} \mathit{Trivial}(v) & \text{se } v \text{ é do tipo P,} \\ \mathit{Trivial}(v) \cup \mathit{Consec}(v) & \text{se } v \text{ é do tipo Q,} \\ \mathit{Trivial}(v) \cup \mathit{Pot}(v) & \text{se } v \text{ é do tipo R.} \end{cases} \tag{2.1}$$

e, então, definimos o fecho $\overline{T}$ de uma árvore PQR T como na Equação (2.2).

$$\overline{T} = \bigcup_{v \in T} \overline{v} \tag{2.2}$$

É importante notar que $\overline{T}$ é o que foi chamado de $\mathit{Compl}(T)$ no artigo de Meidanis, Porto e Telles [16], apesar de ser definido de forma diferente.

Uma árvore PQR representa a coleção $\overline{\mathcal{C}}$. Dada uma árvore PQR T, sujeita a uma coleção de restrições $\mathcal{C}$, temos que $\overline{T} = \overline{\mathcal{C}}$. Para cada nó v de T, o conjunto $\hat{v}$ é um elemento de $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$.

Durante o processo de adicionar uma nova restrição C a uma árvore PQR T, que chamamos de redução, os nós são coloridos da seguinte forma:

- Um nó v é **negro** quando $\hat{v} \subset C$.
- Um nó v é **cinza** quando $\hat{v} \not\perp C$.
- Um nó v é **branco** quando $\hat{v} \cap C = \emptyset$ ou $C \subseteq \hat{v}$.

Em alguns momentos, chamamos de coloridos os nós negros ou cinzas e também nos referimos ao ato de colorir um nó de branco como descolorir um nó. Dado um nó v de uma árvore PQR, denominamos os conjuntos de filhos negros, cinzas e brancos de v por $B(v)$, $G(v)$ e $W(v)$, respectivamente.

Ao adicionar uma restrição C à coleção $\mathcal{C}$, a coleção $\overline{\mathcal{C}}$ cresce, enquanto $\mathcal{C}^{\perp}$ decresce. Neste processo, tanto nós negros quanto brancos continuam em $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$. Já os nós cinzas não estão em $\mathcal{C}^{\perp}$, portanto eles devem ser reestruturados. Este processo será detalhado no próximo capítulo.

Para garantir a corretude de uma operação durante a redução de uma árvore PQR T em relação ao conjunto C, é necessário provar que

$$\overline{\overline{T'} \cup \{C\}} = \overline{\overline{T''} \cup \{C\}},$$

onde T' e T'' são as árvores antes e depois da operação, respectivamente. Provamos isto separadamente para o fecho de cada nó das árvores e, para facilitar tal trabalho, utilizamos o conceito a seguir.

Dadas duas árvores PQR T' e T'', um nó v' em T' é **equivalente** a um nó v'' em T'' quando têm o mesmo tipo e há uma correspondência biunívoca entre os filhos de v' e v'' que mantém ordem e as folhas descendentes, ou seja, para todo i entre 1 e k, temos $\hat{f'}_i = \hat{f}''_i$, onde f'_i e f''_i são, respectivamente, os i-ésimos filhos de v' e v'' e k é o número de filhos de v' (e de v'').

Os fechos de nós equivalentes apresentam a seguinte propriedade:

Teorema 1. *Se v' é equivalente a v'', então $\overline{v'} = \overline{v''}$.*

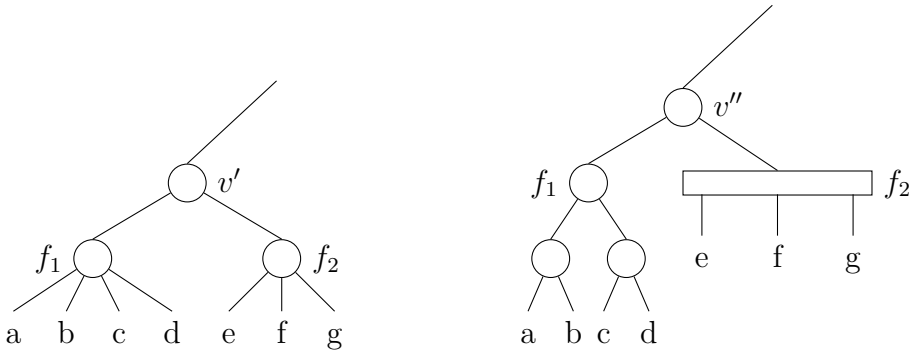
Demonstração. Como v' é equivalente a v'', ambos têm o mesmo tipo. Assim, basta provar que $\mathit{Trivial}(v') = \mathit{Trivial}(v'')$, $\mathit{Consec}(v') = \mathit{Consec}(v'')$ e $\mathit{Pot}(v') = \mathit{Pot}(v'')$, já que tanto v' quanto v'' caem no mesmo caso da Equação (2.1).

Sabemos que, para todo nó f' filho de v', existe um filho f'' de v'' tal que $\hat{f'} = \hat{f}''$. Como $\hat{v'}$ é a união de todos os $\hat{f'}$ e o mesmo vale para v'', temos que $\mathit{Trivial}(v') = \mathit{Trivial}(v'')$.

Como a correspondência entre os filhos de v' e v'' mantém, além do conjunto de folhas, a ordem dos filhos, também temos que $\mathit{Consec}(v') = \mathit{Consec}(v'')$.

De maneira similar, podemos demonstrar que $\mathit{Pot}(v') = \mathit{Pot}(v'')$, pois todo filho de v' tem um correspondente com mesmo conjunto de folhas descendentes em v'' e vice-versa. □

Note que, se v' e v'' são nós equivalentes, isto não necessariamente implica em serem equivalentes as subárvores com raízes em v' e v'', isto é, tomando a subárvore de T' formada por v' e todos os seus descendentes, ela não é necessariamente equivalente à subárvore de T'' formada por v'' e todos os seus descendentes. Um contra-exemplo é mostrado na Figura 2.1.



(a) T' (b) T''

Figura 2.1: Exemplo de nós equivalentes com raízes de subárvores não equivalentes: os nós v' e v'' são equivalentes, mas as subárvores sobre $\hat{v}' = \hat{v}''$ com raízes neles não são equivalentes.

Capítulo 3

Algoritmo online ingênuo

Neste capítulo, apresentamos uma versão simples do algoritmo para a construção incremental de árvores PQR, que então usamos para explicar a redução de uma árvore PQR e provar a corretude de suas operações. A implementação desta versão do algoritmo está disponível no endereço eletrônico `http://www.ic.unicamp.br/~meidanis/PUB/Mestrado/2010-Zanetti/zanetti-PQR/`.

3.1 Algoritmo ingênuo

O algoritmo ingênuo é um algoritmo online e, portanto, processa os conjuntos em $\mathcal{C}$ um por vez. No início da execução, temos uma árvore universal, que tem uma raiz do tipo P com todas as folhas como filhos, representando o menor $\overline{\mathcal{C}}$ possível, apenas contendo $\mathit{Trivial}(t)$, onde t é a raiz da árvore. A seguir, a árvore é reduzida em relação a cada um dos conjuntos em $\mathcal{C}$, usando o Algoritmo 2.

Algoritmo 1: Redução de árvores PQR

```
1 Inicializar T como uma árvore universal
2 para cada C ∈ 𝒞 faça
3 |   Adicionar C a T (Algoritmo 2)
```

Reduzir uma árvore PQR em relação a um conjunto é uma atividade que nós dividimos no Algoritmo 2 em cinco passos. Se o conjunto C tem menos que dois elementos, pode ser ignorado, pois a árvore PQR não sofrerá alterações. Portanto, assumimos que $|C| \geq 2$.

Algoritmo 2: Algoritmo para adicionar um conjunto C a uma árvore T.

```
1 Colorir a árvore em relação a C
2 Encontrar a raiz da subárvore pertinente
3 Reestruturar a árvore (Algoritmo 3)
4 Ajustar a raiz
5 Descolorir a árvore
```

Como já foi dito, a cada restrição que é acrescentada, cresce a coleção $\overline{\mathcal{C}}$ e diminui o número de conjuntos em $\mathcal{C}^{\perp}$. Portanto, a árvore pode acabar sendo alterada.

Os nós brancos e os nós negros são ortogonais ao conjunto C, isto é, nós brancos não têm interseção com C ou contêm C e nós negros estão contidos propriamente em C. Por isso, estes nós têm correspondentes em T após a redução com relação a C. Por outro lado, os nós cinzas têm fronteiras que se sobrepõem estritamente a C e devem ser reparados, ganhando ou perdendo filhos para se tornarem brancos, ou até mesmo sendo removidos da árvore.

Assim, o terceiro passo da redução é tratar os nós cinzas da árvore, um por vez, utilizando o Algoritmo 3, até que a árvore não tenha nenhum nó cinza. Este é o passo mais elaborado do algoritmo e usa cinco diferentes operações, de acordo com o tipo da raiz da subárvore pertinente r e do nó cinza v (ou v_i, se r for do tipo Q ou R) sendo processado.

Algoritmo 3: Algoritmo para reestruturar a árvore. (Passo 3 do Algoritmo 2)

```
1  enquanto existe um filho cinza v da raiz r faça
2      se v é do tipo P então
3          transformar nó P em nó Q
4      se r é do tipo P então
5          unir filhos negros
6          mover filhos para fora da raiz
7      senão
8          se v é do tipo Q então
9              reverter condicionalmente nó cinza
10         fundir com a raiz
```

Como vimos no Capítulo 2, a corretude dos algoritmos para as cinco operações do terceiro passo é provada seguindo a técnica apresentada na Seção 5.1 do trabalho de Telles e Meidanis [24]. Esta técnica consiste de provar que, se T' e T'' são as árvores antes e depois cada passo do algoritmo e C é o conjunto sendo adicionado, então

$$\overline{\overline{T'} \cup \{C\}} = \overline{\overline{T''} \cup \{C\}}.$$

Para isso, basta escrever $\overline{u}$ em função de conjuntos em $\overline{T''} \cup \{C\}$, para cada nó u de T' que não tem equivalente em T'', e $\overline{v}$ em função de conjuntos em $\overline{T'} \cup \{C\}$, para cada nó v de T'' que não tem equivalente em T', sempre usando apenas as operações de interseção, união não-disjunta e diferença não contida.

Note que, como T' e T'' são árvores PQR sobre um mesmo conjunto, para todo nó v de uma dessas árvores, todos os conjuntos em $\mathit{Trivial}(v)$, exceto possivelmente $\hat{v}$, pertencem ao fecho da outra árvore. Assim, podemos nos limitar a escrever somente o conjunto $\hat{v}$ para v do tipo P, e as coleções $\mathit{Consec}(v)$ para v do tipo Q e $\mathit{Pot}(v)$ para v do tipo R em função de nós da outra árvore e de C.

Em todas as operações tratadas a seguir, observe que todos os nós não representados nas figuras, bem como todos os nós representados por losangos nas figuras, têm equivalente na outra árvore e portanto não precisam ser tratados. Para os outros nós, usaremos v' e v'' para indicar as versões de um nó v em T' e T'', respectivamente.

Esta técnica para demonstrar a corretude das operações é complicada, mas é a melhor que encontramos para este propósito, em contraste ao trabalho de Booth e Lueker, em que não há prova de corretude das transformações.

3.1.1 Colorir a árvore

Aqui, os nós são coloridos de forma um pouco diferente da definida no Capítulo 2: um nó v é **negro** quando $\hat{v} \subseteq C$, **cinza** quando $\hat{v} \not\subseteq C$ e $\hat{v} \cap C \neq \emptyset$ e **branco** nos outros casos, isto é, quando $\hat{v} \cap C = \emptyset$. A diferença entre esta coloração e a definida anteriormente é que a raiz da subárvore pertinente e seus ancestrais, que seriam brancos, aqui são coloridos. Isto será corrigido no próximo passo.

Uma maneira simples de colorir a árvore deste modo é fazê-lo recursivamente, percorrendo a árvore em pós-ordem e pintando cada nó de acordo com as cores de seus filhos.

Algoritmo 4: `colorir(`v, C`)` Algoritmo para colorir um nó da árvore.

```
1  se v é folha e v ∈ C então
2  |  pintar v de preto
3  senão
4  |  para cada filho u de v faça
5  |  |  colorir(u, C)
6  |  se todos os filhos de v são pretos então
7  |  |  pintar v de preto
8  |  senão se todos os filhos de v são brancos então
9  |  |  manter v descolorido
10 |  senão
11 |  |  pintar v de cinza
```

3.1.2 Encontrar raiz da subárvore pertinente

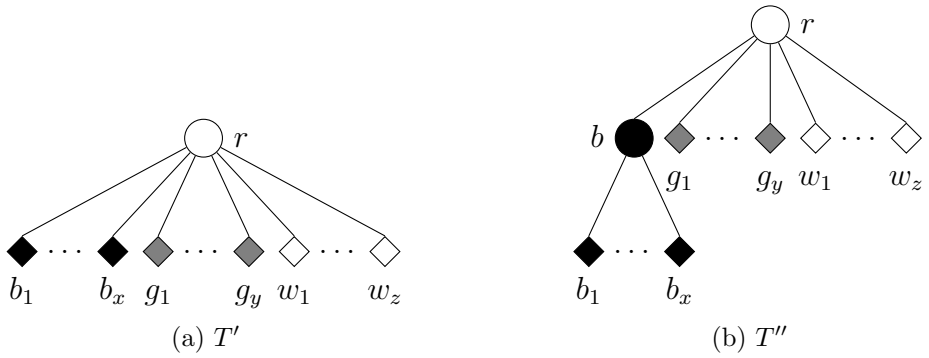
Para encontrar a raiz da subárvore pertinente de uma árvore que já foi colorida, basta encontrar o nó mais próximo da raiz da árvore que tenha mais de um filho colorido. Também utilizamos este passo para corrigir a coloração da árvore, pintando de branco os nós que contêm C. Para encontrar a raiz da subárvore pertinente de T, deve-se chamar o Algoritmo 5 como `LCA(`t`)`, onde t é a raiz de T.

Algoritmo 5: `LCA(`v`)` Algoritmo para encontrar a raiz da subárvore pertinente.

```
1  descolorir v
2  se v é folha então
3  |  retornar v
4  senão
5  |  se v tem pelo menos dois filhos coloridos então
6  |  |  retornar v
7  |  senão
8  |  |  w ← único filho colorido de v
9  |  |  retornar LCA(w)
```

3.1.3 Unir filhos negros

Quando um nó P tem dois ou mais filhos negros, no caso em que há permutação válida antes da redução em relação a C, tais nós devem ficar consecutivos para qualquer permutação válida após a redução. No outro caso, em que não há permutação válida, a união das fronteiras dos filhos negros é um elemento em $\overline{\mathcal{C}}$, como vemos a seguir. Esta operação



(a) T' (b) T''

Figura 3.1: Operação "unir filhos negros".

equivale ao padrão P2 de Booth e Lueker.

Quando a raiz r da subárvore pertinente é do tipo P, isso é feito movendo todos os filhos negros de r para um novo nó do tipo P, filho de r, como ilustrado na Figura 3.1. A operação é executada conforme o Algoritmo 6.

Algoritmo 6: Unir filhos negros

```
1 se r tem 2 ou mais filhos negros então
2 |   criar nó negro b do tipo P filho de r
3 |   para cada filho negro n de r faça
4 |   |   mover n para b
```

Nós em T' sem equivalentes em T'': r'. Porém, r' é do tipo P e $\hat{r}' = \hat{r}''$.

Nós em T'' sem equivalentes em T': r'' e b''. Para o nó r'' temos $\hat{r}'' = \hat{r}'$. Para b'', temos $\hat{b}'' = (\hat{r}' \cap C) \setminus \hat{g}'_1 \setminus \ldots \setminus \hat{g}'_y$.

3.1.4 Transformar nó P em nó Q

Se um nó v do tipo P é pintado de cinza, ele não é ortogonal a C, portanto não pode ser nó na nova árvore. No entanto, tanto $\hat{v} \cap C$ quanto $\hat{v} \setminus C$ e $C \setminus \hat{v}$ são ortogonais a C. O conjunto $C \setminus \hat{v}$ só tem elementos fora de v e será tratado em outra parte da árvore. Mas os conjuntos $\hat{v} \cap C$ e $\hat{v} \setminus C$ devem ser tratados aqui, pois estão contidos em $\hat{v}$.

Tomemos o conjunto $\hat{v} \cap C$. Ele não é ortogonal aos nós da árvore antiga, por isto não será um nó da nova árvore. Mais especificamente, $\hat{v} \cap C$ não é ortogonal aos nós g_i, embora seja ortogonal tanto aos b_i quanto aos w_i. Para remediá-lo em relação a g_1, por exemplo, podemos fazer $(\hat{v} \cap C) \setminus g_1$ ou $g_1 \setminus (\hat{v} \cap C)$ ou $(\hat{v} \cap C) \cap g_1$. Os dois últimos conjuntos não nos interessam, pois estão contidos em g_1 e serão tratados quando analisarmos g_1. Resta-nos $(\hat{v} \cap C) \setminus g_1$. Observe que este conjunto, apesar de ortogonal a g_1, ainda não é ortogonal a g_2. Fazendo uma análise semelhante à feita para g_1, concluímos que devemos tomar

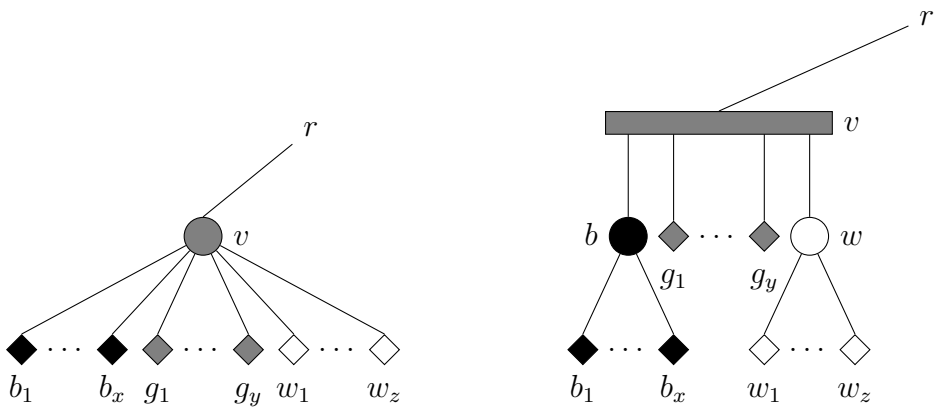


Figura 3.2: Operação "transformar nó P em nó Q".

$(\hat{v} \cap C) \oint g_i \oint g_2$. Continuando assim, chegamos ao conjunto $(\hat{v} \cap C) \oint g_i \oint g_2 \oint \ldots \oint g_y$, que nada mais é que o conjunto $\hat{b}$ da Figura 4.2 (b). Assim, concluímos que precisamos ter a união dos filhos negros nesta operação.

Tomemos agora $\hat{v} \oint C$. Este conjunto também não é ortogonal aos g_i. Fazendo como no parágrafo anterior, vem que $(\hat{v} \oint C) \oint g_i \oint g_2 \oint \ldots \oint g_y$ é um nó da nova árvore. Este conjunto corresponde ao nó w da Figura 4.2 (b), mostrando que a união dos filhos brancos também é necessária nesta operação.

Desta forma, a união das fronteiras dos filhos negros de v é ortogonal a C e o mesmo acontece com os filhos brancos. Por outro lado, os filhos cinzas de v devem ser colocados em um nó do tipo Q, entre os dois nós criados para agrupar os filhos negros e brancos de v, para que $\overline{T}$ contenha os novos conjuntos que C adiciona a $\overline{\mathcal{C}}$, como veremos na prova de corretude para esta operação. Esta operação faz o papel dos padrões P3 e P5 de Booth e Lueker.

Note que, quando o nó sendo processado tem mais de dois filhos cinzas ou pelo menos um filho negro e mais de um cinza, não existe permutação válida dos filhos deste nó. Este nó será, portanto, transformado em R no quarto passo da redução.

A transformação de P em Q é ilustrada na Figura 4.2 e executa-se segundo o Algoritmo 7.

Algoritmo 7: Transformar nó P em nó Q

1 unir filhos negros de v
2 unir filhos brancos de v
3 mudar tipo de v para Q
4 ordenar os filhos de v em ordem decrescente de cor

Vamos mostrar agora que esta operação satisfaz ao critério $\overline{\overline{T'} \cup \{C\}} = \overline{\overline{T''} \cup \{C\}}$.

Nós em T' sem equivalentes em T'': v'. Note que r' é equivalente a r'', visto que $\hat{v}' = \hat{v}''$, e isto também garante que $\hat{v}' \in \overline{T''}$.

Nós em T'' sem equivalentes em T': b'', w'' e v''.

Para os dois primeiros, temos

$$\hat{b}'' = (\hat{v}' \cap C) \oint \hat{g}'_1 \oint \ldots \oint \hat{g}'_y$$

e

$$\hat{w}'' = (\hat{v}' \oint C) \oint \hat{g}'_1 \oint \ldots \oint \hat{g}'_y.$$

Agora vamos tratar v''. Conjuntos S consecutivos de seus filhos são de uma das seguintes formas:

1. $\{b'', g''_1, \ldots, g''_y, w''\}$.
2. $\{b'', g''_1, \ldots, g''_j\}$, para $j = 0 \ldots y$.
3. $\{g''_i, \ldots, g''_y, w''\}$, para $i = 1 \ldots y + 1$.
4. $\{g''_i, \ldots, g''_j\}$, para $1 \leq i \leq j \leq y$.

No primeiro caso, temos $\bigcup_{x \in S} \hat{x} = \hat{v}'' = \hat{v}'$.

No segundo caso, escrevemos:

$$\bigcup_{x \in S} \hat{x} = (\hat{v}' \cap C) \uplus \hat{g}'_1 \uplus \ldots \uplus \hat{g}'_j \oint \hat{g}'_{j+1} \oint \ldots \oint \hat{g}'_y.$$

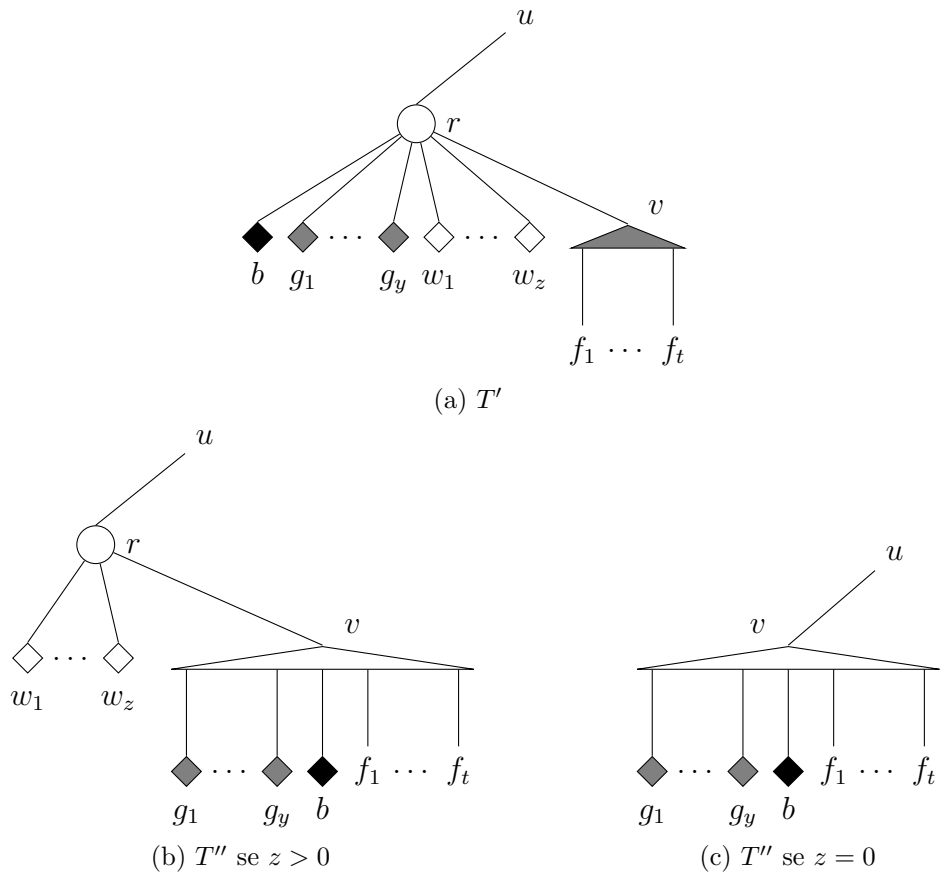
No terceiro caso, escrevemos:

$$\bigcup_{x \in S} \hat{x} = (\hat{v}' \oint C) \oint \hat{g}'_1 \oint \ldots \oint \hat{g}'_{i-1} \uplus \hat{g}'_i \uplus \ldots \uplus \hat{g}'_y.$$

E no quarto caso:

$$\bigcup_{x \in S} \hat{x} = [(\hat{v}' \cap C) \uplus \hat{g}'_i \uplus \ldots \uplus \hat{g}'_j] \cap [(\hat{v}' \oint C) \uplus \hat{g}'_i \uplus \ldots \uplus \hat{g}'_j].$$

Assim, $Consec(v'') \subseteq \overline{\overline{T'} \cup \{C\}}$.



(a) T'

(b) T'' se $z > 0$

(c) T'' se $z = 0$

Figura 3.3: Operação "mover para fora da raiz". A situação anterior à operação é ilustrada em (a), enquanto (b) e (c) são as duas possibilidades para a situação posterior.

3.1.5 Mover filhos para fora da raiz

Se a raiz da subárvore pertinente é do tipo P, ela não é transformada em um nó Q, pois r é ortogonal a C e, portanto, continuará sendo um nó na nova árvore. Neste caso, apenas se movem para um filho cinza v (que a essa altura já é com certeza do tipo Q ou R) todos os outros filhos coloridos dessa raiz. Isto faz com que v se torne a nova raiz da subárvore pertinente e, consequentemente, fique branco, eliminando assim um nó cinza da árvore. Esta operação, ilustrada na Figura 3.3 e descrita no Algoritmo 8, se assemelha aos padrões P4 e P6 de Booth e Lueker.

Algoritmo 8: Mover para fora da raiz

```
se f_1.cor < f_t.cor então
    reverter v
se r tem um filho negro b então
    mover b para o início de v
para cada filho cinza c de r diferente de v faça
    mover c para o início de v
se r tem apenas um filho então
    se r é a raiz da árvore então
        v torna-se a raiz da árvore
    senão
        mover v para r.pai
    pintar v de branco
    destruir r
```

Nós em T' sem equivalentes em T'': r' e v'. Porém, note que $\overline{v'} \subseteq \overline{v''}$ e

$$\hat{r}' = \begin{cases} \hat{r}'' & \text{se } z > 0, \\ \hat{v}'' & \text{se } z = 0, \end{cases}$$

ou seja, $\hat{r}' \in \overline{T}''$ em qualquer caso.

Nós em T'' sem equivalentes em T': r'' e v''. Entretanto, r'' existe apenas quando $z > 0$ e, neste caso, temos $\hat{r}'' = \hat{r}'$. Em relação ao nó v'', os conjuntos em $Consec(v'') \setminus Consec(v')$ são de uma das seguintes formas:

1. $\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1 \cup \ldots \cup \hat{f}_j$, para $1 \leq i \leq y+1$ e $0 \leq j \leq t$.
2. $\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b}$, para $1 \leq i \leq y+1$.
3. $\hat{g}_i \cup \ldots \cup \hat{g}_j$, para $1 \leq i \leq j \leq y$.

No primeiro caso, seja c o menor índice tal que f_c não seja branco. Além disso, utilizamos o seguinte conjunto, pertencente a $Consec(v')$:

$$F_{[i,j]} = \bigcup_{i \leq k \leq j} \hat{f}_k.$$

Assim, podemos escrever:

$$\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1 = (C \setminus F_{[c+1,t]} \setminus \hat{g}_1 \setminus \ldots \setminus \hat{g}_{i-1} \uplus F_{[1,c]} \uplus \hat{g}_i \ldots \uplus \hat{g}_y) \setminus F_{[2,t]}$$

e, a partir deste resultado, todos os outros conjuntos desejados, da seguinte forma:

$$\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1 \cup \ldots \cup \hat{f}_j = (\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1) \uplus F_{[1,j]}.$$

No segundo caso, temos:

$$\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} = C \setminus \hat{v}' \setminus \hat{g}_1 \setminus \ldots \setminus \hat{g}_{i-1} \uplus \hat{g}_i \ldots \uplus \hat{g}_y.$$

Para o terceiro e último caso, escrevemos:

$$\hat{g}_i \cup \ldots \cup \hat{g}_j = [(C \setminus \hat{v}' \setminus \hat{g}_1 \setminus \ldots \setminus \hat{g}_{i-1} \uplus \hat{g}_i \uplus \ldots \uplus \hat{g}_j \setminus \hat{g}_{j+1} \setminus \ldots \setminus \hat{g}_y) \setminus C] \uplus \hat{g}_i \uplus \ldots \uplus \hat{g}_j.$$

3.1.6 Reverter condicionalmente nó cinza

Após garantir-se que o nó cinza v sendo processado seja do tipo Q ou R, se a raiz da subárvore pertinente r for também Q ou R, v será fundido a r, como será visto na próxima seção. Mas, para garantir que tal fusão preserve a invariante $\overline{\overline{T'} \cup \{C\}} = \overline{\overline{T''} \cup \{C\}}$, o nó v deve ser orientado de forma que seu extremo mais escuro fique do mesmo lado que o vizinho mais escuro de v. Uma das situações em que isso é relevante é ilustrada na Figura 3.4 e todas as condições para a reversão podem ser vistas no Algoritmo 9, lembrando que as cores têm as seguintes representações numéricas: 0=branco, 1=cinza e 2=negro.

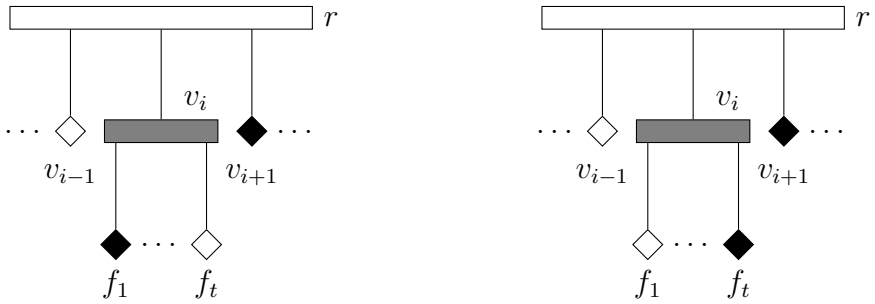
Para justificar formalmente a necessidade desta operação, começamos com um lema.

Lema 1. *Seja v um nó de cor cinza. Então existem sempre filhos distintos, f_w e f_b, de v tais que f_w tem folha descendente branca e f_b tem folha descendente negra. Ou seja, $\hat{f}_w \not\subseteq C$ e $\hat{f}_b \cap C \neq \emptyset$.*

Demonstração. Como v é cinza, não tem todos os filhos negros nem todos os filhos brancos. Então v tem um filho branco e um filho negro ou tem pelo menos um filho cinza.

Se v tem um filho branco f_w e um filho negro f_b, temos que $\hat{f}_w \not\subseteq C$ e $\hat{f}_b \cap C \neq \emptyset$, concluindo a prova.

Se v tem pelo menos um filho cinza f_c, este filho tem folhas descendentes negras e brancas. Tome um outro filho qualquer f_x de v. Se f_x for branco, tome $f_w = f_x$ e $f_b = f_c$. Se f_x for negro ou cinza, tome $f_w = f_c$ e $f_b = f_x$. □



(a) T' (b) T''

Figura 3.4: Operação "reverter condicionalmente nó cinza".

Em seguida, observamos que, se v_i é um filho cinza da raiz r da subárvore pertinente, então r deve ter algum outro filho v_d que não é branco, caso contrário v_i seria a raiz da subárvore pertinente em vez de r.

O objetivo desta operação é garantir que, existam filhos f_w e f_b do nó cinza v_i, satisfazendo as condições do Lema 1 que estejam ordenados de forma que f_b (que tem uma folha descendente negra) esteja mais próximo deste irmão v_d de v_i do que f_w. A motivação para tal operação é permitir que a operação seguinte, fundir com a raiz, mantenha os filhos mais escuros juntos, se isto for possível.

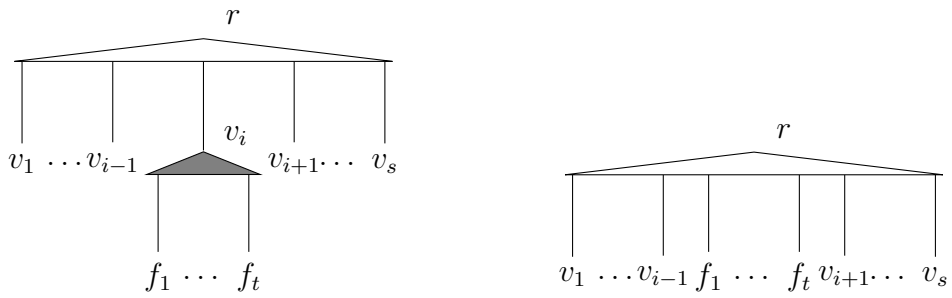
Um problema que temos com o Lema 1 é que encontrar f_w e f_b pode levar tempo não constante. Porém, é possível realizar uma operação de tempo constante que tenha um efeito suficiente para nossos propósitos.

Como esta operação deve ser realizada em tempo constante, implementamo-la como segue. Um nó cinza v_i, do tipo Q, é revertido quando as cores de seus filhos extremos f_1 e f_t não estão alinhadas com as cores dos vizinhos de v_i. Se v_i só tem um vizinho, v_i é revertido se seu filho mais próximo do vizinho existente é mais claro que o outro filho extremo de v_i.

Algoritmo 9: Reverter condicionalmente nó cinza

```
1 se v_{i-1} não existe e f_1.cor > f_t.cor então
2 |   reverter v
3 senão se v_{i+1} não existe e f_1.cor < f_t.cor então
4 |   reverter v
5 senão se (v_{i-1}.cor − v_{i+1}.cor)(f_1.cor − f_t.cor) < 0 então
6 |   reverter v
```

Se f_1 e f_t têm cores diferentes e v_{i-1} e v_{i+1} também não têm a mesma cor, o algoritmo sempre alinha v_i de forma que o nó f_b mais escuro entre f_1 e f_t fique junto do nó v_d mais escuro entre v_{i-1} e v_{i+1}.



(a) T' (b) T''

Figura 3.5: Operação "fundir com a raiz".

Quando f_1 e f_t têm a mesma cor, v_i não é revertido. Isto justifica-se, pois é sempre possível determinar f_w e f_b convenientes. Se f_1 e f_t são brancos, existe um f_b cinza ou negro, com $1 < b < t$ e f_w seria o nó extremo mais distante de v_d. Se f_1 e f_t são cinzas, o mais próximo de um nó não branco é f_b e o outro, f_w. Se f_1 e f_t são negros, existe f_w não negro, com $1 < a < t$ e f_b seria o nó extremo mais próximo do irmão não branco v_d.

O nó v_i também não é revertido se v_{i-1} e v_{i+1} têm a mesma cor. Se v_{i-1} e v_{i+1} são ambos negros ou cinzas, qualquer um deles pode ser escolhido como v_d e, portanto, qualquer orientação de v_i é satisfatória. Se v_{i-1} e v_{i+1} são brancos, a árvore já está comprometida e não será possível manter juntos todos os nós escuros, então não adianta reverter. Este é o único caso em que a árvore após a execução do Algoritmo 9 não necessariamente satisfaz $(a - b)(i - d) \geq 0$.

Portanto, depois desta operação, com certeza existem nós f_w, f_b e v_d como descritos tais que $(a-b)(i-d) \geq 0$ a não ser que v_{i-1} e v_{i+1} sejam brancos. Estes nós serão usados na próxima operação, fundir com a raiz.

Como a única transformação feita nesta operação é uma transformação de equivalência, pelo Teorema 36 de [16], temos

$$\overline{T'} = \overline{T''}$$

e portanto

$$\overline{\overline{T'} \cup \{C\}} = \overline{\overline{T''} \cup \{C\}}.$$

3.1.7 Fundir com a raiz

Com v_i devidamente orientado, procedemos à fusão, que remove o nó v_i. Lembrando que, antes desta operação, os nós r e v_i são do tipo R ou Q. Esta operação, junto com a reversão condicional, cumpre o papel do padrão Q2 de Booth e Lueker.

A fusão está ilustrada na Figura 3.5 e é executada conforme o Algoritmo 10.

Algoritmo 10: Fundir com a raiz

```
1 mover filhos de v_i para r após v_i
2 se r é do tipo Q e v_i é do tipo R então
3 |   alterar tipo de r para R
4 destruir v_i
```

Nós em T' sem equivalentes em T'': r' e v'_i.

Se v'_i é tipo Q, temos que $Consec(v'_i) \subseteq \overline{r''}$, pois r'' é do tipo Q ou R. Se v'_i é do tipo R, então necessariamente r'' será do tipo R e, portanto, $Pot(v'_i) \subseteq \overline{r''}$. Quanto ao nó r', temos sempre $\overline{r'} \subseteq \overline{r''}$, pois basta tomar subconjuntos de $\overline{r''}$ que contêm todos os $\hat{f}_i$ ou nenhum dos $\hat{f}_i$.

Nós em T'' sem equivalentes em T': r''.

Aqui precisamos definir mais um conjunto pertencente a $\overline{T'}$ que será útil:

$$V_{[x,y]} = \bigcup_{x \leq k \leq y} \hat{v}_k,$$

além do conjunto que já foi definido anteriormente:

$$F_{[x,y]} = \bigcup_{x \leq k \leq y} \hat{f}_k.$$

Estes conjuntos estão em $\overline{T'}$ pois r e v_i são nós Q ou R.

Primeiramente, demonstramos que $Consec(r'') \subseteq \overline{\overline{T'} \cup C}$. Para tanto, vamos usar o seguinte fato, facilmente verificável:

$$\overline{Consec(r') \cup Consec(v'_i) \cup \{\hat{v}_{i-1} \cup \hat{f}_1\} \cup \{\hat{v}_{i+1} \cup \hat{f}_t\}} = Consec(r'').$$

Portanto, basta provar que $\{\hat{v}_{i-1} \cup \hat{f}_1, \hat{v}_{i+1} \cup \hat{f}_t\} \subseteq \overline{\overline{T'} \cup C}$ para provar que $Consec(r'') \subseteq \overline{\overline{T'} \cup C}$. Além disso, $\hat{v}_{i+1} \cup \hat{f}_t$ pode ser escrito em função de $\hat{v}_{i-1} \cup \hat{f}_1$ e de outros conjuntos em $\overline{T'}$, como segue:

$$\hat{v}_{i+1} \cup \hat{f}_t = V_{[i,i+1]} \natural (\hat{v}_{i-1} \cup \hat{f}_1) \natural F_{[1,t-1]}.$$

Com isso, precisamos apenas de $\hat{v}_{i-1} \cup \hat{f}_1$ e este conjunto ainda pode ser escrito como

$$\hat{v}_{i-1} \cup \hat{f}_1 = (F_{[1,b]} \cup V_{[d,i-1]}) \cap V_{[i-1,i]} \natural F_{[2,t]}.$$

Vamos agora à parte principal da prova. Se v_{i-1} e v_{i+1} não são ambos brancos, a operação de reverter condicionalmente nó cinza orientou o nó v_i de forma que temos nós f_w e f_b, filhos de v_i, e v_d, irmão de v_i, com f_w não negro, f_b e v_d não brancos e f_b mais próximo de v_d que f_w (veja a Figura 3.4, onde f_w é f''_1, f_b é f''_t e v_d é v_{i+1}).

Então, há dois casos. Lembrando que as operações são sempre executadas da esquerda para a direita, se $a < b$ (que implica em $i < d$), temos:

$$F_{[b,t]} \cup V_{[i+1,d]} = C \cap V_{[i,d]} \upharpoonright F_{[1,b-1]} \uplus V_{[i+1,d]} \uplus F_{[b,t]}. \tag{3.1}$$

E se $b < a$ (que implica em $d < i$), temos:

$$F_{[1,b]} \cup V_{[d,i-1]} = C \cap V_{[d,i]} \upharpoonright F_{[b+1,t]} \uplus V_{[d,i+1]} \uplus F_{[1,b]}. \tag{3.2}$$

Se v_{i-1} e v_{i+1} são brancos, mas f_1 e f_t têm cores iguais, as fórmulas (3.1) e (3.2) continuam válidas, com escolhas aproriadas de f_w e f_b, conforme descrito na seção anterior. Portanto, só precisamos nos preocupar com o caso em que v_{i-1} e v_{i+1} são brancos e f_1 e f_t têm cores diferentes. Se f_1 é o mais claro dos dois, ou seja, f_1 não é negro, temos:

$$\hat{v}_{i-1} \cup \hat{f}_1 = V_{[i-1,i]} \upharpoonright C \upharpoonright F_{[2,t]} \uplus \hat{f}_1.$$

Se f_1 tem a cor negra, então f_t não é negro e, portanto, podemos obter $\hat{v}_{i+1} \cup \hat{f}_t$ de forma similar:

$$\hat{v}_{i+1} \cup \hat{f}_t = V_{[i,i+1]} \upharpoonright C \upharpoonright F_{[1,t-1]} \uplus \hat{f}_t.$$

Para o caso em que r'' é do tipo R, precisamos também demonstrar que $\overline{Pot(r'')} \subseteq \overline{\overline{T'} \cup C}$. Isso ocorre quando v'_i ou r' é do tipo R. Já sabemos que $Consec(r'') \subseteq \overline{\overline{T'} \cup C}$.

Se v'_i é do tipo R, temos

$$\hat{f}_1 \cup \hat{f}_t \in \overline{\overline{T'} \cup C}$$
$$Consec(r'') \cup \{\hat{f}_1 \cup \hat{f}_t\} \subseteq \overline{\overline{T'} \cup C}$$
$$\overline{Consec(r'') \cup \{\hat{f}_1 \cup \hat{f}_t\}} \subseteq \overline{\overline{T'} \cup C}$$
$$Pot(r'') \subseteq \overline{\overline{T'} \cup C}$$

Já se r' é do tipo R, temos

$$\hat{v}_1 \cup \hat{v}_s \in \overline{\overline{T'} \cup C}$$
$$Consec(r'') \cup \{\hat{v}_1 \cup \hat{v}_s\} \subseteq \overline{\overline{T'} \cup C}$$
$$\overline{Consec(r'') \cup \{\hat{v}_1 \cup \hat{v}_s\}} \subseteq \overline{\overline{T'} \cup C}$$
$$Pot(r'') \subseteq \overline{\overline{T'} \cup C}$$

3.1.8 Ajustar a árvore

Após reparar todos os nós cinzas, o quarto passo da redução consiste em fazer alguns ajustes à arvore se necessário. Duas verificações são feitas com a raiz de subárvore pertinente r: se r é do tipo P, seus filhos negros são unidos (se forem dois ou mais), utilizando o mesmo procedimento do Algoritmo 6, da Seção 3.1.3; se r é um nó Q, é necessário verificar se todos os seus filhos negros são consecutivos e, se não forem, mudar o tipo de r para R. Esta operação pode ser vista em detalhes nos Algoritmos 11 e 12 a seguir, além do Algoritmo 6 já visto anteriormente para a operação unir filhos negros.

Algoritmo 11: Ajustar a raiz.

```
1  se r é do tipo P então
2  |  unir filhos negros de r
3  senão se r é do tipo Q então
4  |  ajustar nó Q
```

Algoritmo 12: Ajustar nó Q.

```
1  v ← primeiro filho de r
2  enquanto v ≠ null faça
3  |  se v é negro então
4  |  |  se block_found = false então
5  |  |  |  block_found ← true
6  |  |  senão se block_ended = true então
7  |  |  |  alterar o tipo de r para R
8  |  |  |  sair
9  |  senão
10 |  |  se block_ended = false então
11 |  |  |  block_ended ← true
12 |  v ← v.next
```

No caso em que o tipo de r é alterado de Q para R, é necessário provar a corretude da operação.

Nós em T' sem equivalentes em T'': r'. Note porém que $Consec(r') \subseteq Pot(r'') \subseteq \overline{T''}$

Nós em T'' sem equivalentes em T': r''. Neste ponto do algoritmo, C é igual à união das fronteiras de um conjunto não consecutivo de filhos de r. Assim, temos que $Pot(r'') = \overline{Consec(r') \cup \{C\}}$.

Tabela 3.1: Complexidade de tempo para cada operação para reparar nó cinza no algoritmo ingênuo.

Operação	Complexidade
Unir filhos negros	$O(\|r\|^2)$
Transformar nó P em nó Q	$O(\|v\|^2)$
Mover filhos para fora da raiz	$O(\|r\|^2 + \|v\|)$
Reverter condicionalmente nó cinza	$O(\|v\|)$
Fundir com a raiz	$O(\|v\|(\|v\| + \|r\|))$

3.1.9 Descolorir a árvore

Para descolorir a árvore, basta descolorir todos os seus nós, assim como zerar todos os contadores de filhos e folhas pertinentes.

3.2 Análise de complexidade do algoritmo ingênuo

Ao utilizar uma estrutura de lista ligada separada da árvore para armazenar os filhos de um nó, remover um filho aleatório de um nó v tem complexidade de tempo $O(|v|)$, pois é necessário localizar o nó da lista que tem a referência para o nó da árvore que se quer remover. O mesmo ocorre quando se quer determinar um vizinho de um nó ou inserir um nó em uma posição específica. Com isso, as operações vistas têm a complexidade de tempo da Tabela 3.1.

Com estes tempos, a complexidade de tempo para reparar um nó cinza, no Algoritmo 3, é de $O(|r|^2 + |v|^2)$. Como são $O(n)$ nós cinzas, temos que o Algoritmo 3 tem uma complexidade de $O(n^3)$.

Os outros quatro passos do Algoritmo 2 são $O(n)$, pois no pior caso passam por todos os nós da árvore. Portanto, reduzir uma árvore PQR T em relação a um conjunto C tem complexidade de tempo $O(n^3)$.

No total, como a coleção $\mathcal{C}$ tem m conjuntos, determinar a árvore PQR relativa à entrada, utilizando o algoritmo visto nesta seção, tem complexidade de tempo $O(mn^3)$.

Nas próximas seções, veremos como tal complexidade pode ser reduzida, sofisticando nossa estrutura de dados.

Capítulo 4

Melhorias ao algoritmo

Neste capítulo, discutimos como melhorar a complexidade de tempo do algoritmo online, com o objetivo de chegar a um algoritmo quase-linear.

Assim como o algoritmo ingênuo do capítulo anterior, a implementação de cada versão do algoritmo neste capítulo está disponível no endereço `http://www.ic.unicamp.br/~meidanis/PUB/Mestrado/2010-Zanetti/zanetti-PQR/`.

4.1 Melhorias ao algoritmo

Uma primeira e mais óbvia melhoria possível é implementar as listas de filhos de forma a poder remover qualquer filho de um nó em tempo constante.

Isso pode ser feito atribuindo a uma mesma estrutura de dados os papéis de nó da árvore e de nó da lista de filhos. Isto quer dizer que o mesmo nó tem campos com referências para seu pai e seus eventuais filhos (no caso, o primeiro e o último), assim como referências para seus dois vizinhos diretos. Com isso, não é mais necessário procurar em que posição da lista um dado nó filho se encontra para removê-lo ou buscar seus vizinhos. Basta utilizar as próprias referências armazenadas no nó.

Tal alteração permitiria executar a remoção e a inserção de nós em posições fora dos extremos das listas em tempo constante, assim como acessar os vizinhos imediatos de um nó. Esta primeira melhoria traria os tempos das operações para os que podemos ver na Tabela 4.1.

Com esse tempos, a complexidade de tempo necessária para processar um nó cinza se torna $O(|r| + |v|)$ e o total para o Algoritmo 3 passa a ser $O(n^2)$.

Como o terceiro passo ainda domina o tempo para a redução, temos que $O(n^2)$ é também a complexidade para reduzir a árvore em relação a um conjunto, o que leva a complexidade total para m restrições para $O(mn^2)$

Esta melhoria, portanto, leva a uma redução significativa da complexidade de tempo

Tabela 4.1: Complexidade de tempo para cada operação com remoção em tempo constante.

Operação	Complexidade
Unir filhos negros	$O(\|r\|)$
Transformar nó P em nó Q	$O(\|v\|)$
Mover filhos para fora da raiz	$O(\|r\| + \|v\|)$
Reverter condicionalmente nó cinza	$O(\|v\|)$
Fundir com a raiz	$O(\|v\|)$

do algoritmo. Porém, mudanças ainda mais sofisticadas podem ser feitas no algoritmo para que a complexidade caia ainda mais, como veremos nas próximas seções.

4.2 Evitar nós brancos

Outra medida que aumenta a eficiência do algoritmo é evitar processar os nós brancos da árvore. A intenção disto é fazer com que a complexidade de tempo da redução em relação a C deixe de ser função de n e passe a ser função de $|C|$. Além de $|C|$ ser sempre menor do que n, tal mudança faz com que a complexidade total da redução seja obtida somando os tamanhos de todos os conjuntos em $\mathcal{C}$, ou seja, $f = \sum_{i=1}^{m} |C_i|$, em vez de mn.

Para que nós brancos não sejam visitados durante o terceiro passso da redução, cada nó precisa saber quem são os seus filhos coloridos. Logo, listas de filhos negros e cinzas devem ser criadas durante a fase de coloração dos nós da árvore e administradas durante todo o resto da redução. A própria coloração deve ser feita de forma diferente, também evitando visitar nós brancos, começando das folhas pertinentes e subindo a árvore em direção à raiz da subárvore pertinente. Este procedimento se torna assim muito mais elaborado que o anterior e é descrito no Algoritmo 13. Este algoritmo é baseado no procedimento de bubble de Booth e Lueker para a redução de árvores PQ [2]. Porém, algumas diferenças existem porque não existem nós bloqueados em nosso algoritmo e nem é possível declarar a árvore irredutível, já que a redução é sempre possível.

Por outro lado, o algoritmo faz duas passagens pela árvore, sempre subindo das folhas em direção à raiz da subárvore pertinente, em vez de uma só, como no algoritmo de Booth e Lueker. A primeira passagem (linhas 3 a 16) serve para contar quantos filhos cada nó tem na subárvore pertinente. A segunda passagem (linhas 18 a 36) serve para colorir os nós, contar quantas folhas em C a fronteira de cada nó contém e, consequentemente, encontrar a raiz da subárvore pertinente, que é o nó mais profundo cuja fronteira contém C. A segunda passagem utiliza a contagem da primeira passagem. O algoritmo de Booth e Lueker executa somente a primeira passagem, e a coloração é feita na fase de casamento

de padrões.

Para cada nó, além das listas de filhos negros e cinzas, três váriáveis são criadas, *visited*, *pertinent_children* e *pertinent_leaves*. A primeira, *visited*, indica se o nó foi colocado na fila durante a primeira passagem. Já a variável *pertinent_children* armazena o número de filhos do nó que estão na subárvore pertinente e ainda não foram coloridos, sendo usada também para saber quando todos os filhos pertinentes de um nó já foram coloridos, o que permite que o nó seja colorido também. A última variável, *pertinent_leaves*, armazena o número de folhas pertinentes na fronteira do nó. Esta variável é usada para identificar a raiz da subárvore pertinente, que é o primeiro nó encontrado enquanto se sobe a subárvore pertinente que tem $pertinent\_leaves = |C|$.

Como a raiz da subárvore pertinente é o primeiro nó em comum entre todos os caminhos das folhas pertinentes à raiz da árvore PQR, poderíamos pensar em parar a primeira subida quando só houver um nó na fila, e este seria a raiz da subárvore pertinente. Porém, quando se inicia a subida de folhas em alturas diferentes da árvore, é possível que a raiz da árvore seja atingida a partir da folha mais próxima antes que a raiz da subárvore pertinente seja alcançada a partir das folhas mais distantes. Como a raiz não tem pai, neste caso, podemos ter único nó na fila sem que ele seja a raiz da subárvore pertinente. Assim, se faz necessária a variável *off_the_top*, que indica se a computação já passou pela raiz da árvore. Esta variável pode ser interpretada como indicadora de um nó virtual acima da raiz na fila, permitindo que a primeira passagem alcance toda a subárvore pertinente antes de parar.

Veja os exemplos da Figura 4.1, em que os nós a e b são as folhas pertinentes e r é a raiz da subárvore pertinente. Na Figura 4.1(a), tanto a quanto b estão à mesma distância da raiz da subárvore pertinente r e a computação deve parar antes de chegar à raiz da árvore. Já no caso da Figura 4.1(b), como o algoritmo não para enquanto r não é alcançado a partir de b, o caminho iniciado em a passa da raiz da árvore e *off_the_top* assume o valor 1 para que a computação não pare antes que r seja alcançado por todas as folhas pertinentes.

Assim, quando se precisa obter um filho negro ou cinza de um determinado nó, basta olhar na lista correspondente deste nó. Isto é o suficiente para encontrar um filho cinza de r enquanto se reestrutura a árvore (Algoritmo 3) em tempo constante, para unir os filhos negros de r (Algoritmo 6) em $O(|B(r)|)$ e também para realizar a operação de mover para fora da raiz (Algoritmo 8) em $O(|B(r)| + |G(r)|)$. Como definido no Capítulo 2, os conjuntos $B(r)$ e $G(r)$ são os conjuntos de filhos negros e cinzas do nó r, respectivamente.

Já a operação de transformar nó P em nó Q que vimos no Algoritmo 7 não se beneficia desta alteração, pois ele move os filhos brancos para um novo nó. Porém, é possível realizar esta operação em tempo $O(|B(v)| + |G(v)|)$, criando um novo nó para tomar o lugar de v e transformando v em filho deste novo nó, movendo-o em uma única operação e levando

Algoritmo 13: Colorir a árvore e encontrar a raiz da subárvore pertinente.

```
1  Q ← ∅
2  off_the_top ← 0
3  para cada x ∈ C faça
4  |  v ← folha de T correspondente a x
5  |  v.visited ← true
6  |  adicionar v a Q
7  enquanto Q.length + off_the_top > 1 faça
8  |  v ← remover elemento de Q
9  |  p ← v.pai
10 |  se p ≠ null então
11 |  |  p.pertinent_children ← p.pertinent_children + 1
12 |  |  se p.visited = false então
13 |  |  |  p.visited ← true
14 |  |  |  adicionar p a Q
15 |  senão
16 |  |  off_the_top ← 1
17 Q ← ∅
18 para cada x ∈ C faça
19 |  v ← folha de T correspondente a x
20 |  v.pertinent_leaves ← 1
21 |  adicionar v a Q
22 enquanto Q.length > 0 faça
23 |  v ← remover elemento de Q
24 |  se v.pertinent_leaves = |C| então
25 |  |  retornar v
26 |  p ← v.pai
27 |  p.pertinent_children ← p.pertinent_children − 1
28 |  p.pertinent_leaves ← p.pertinent_leaves + v.pertinent_leaves
29 |  se v é folha ou todos os filhos de v são negros então
30 |  |  pintar v de preto
31 |  |  adicionar v à lista de filhos negros de p
32 |  senão
33 |  |  pintar v de cinza
34 |  |  adicionar v à lista de filhos cinzas de p
35 |  se p.pertinent_children = 0 então
36 |  |  adicionar p a Q
```

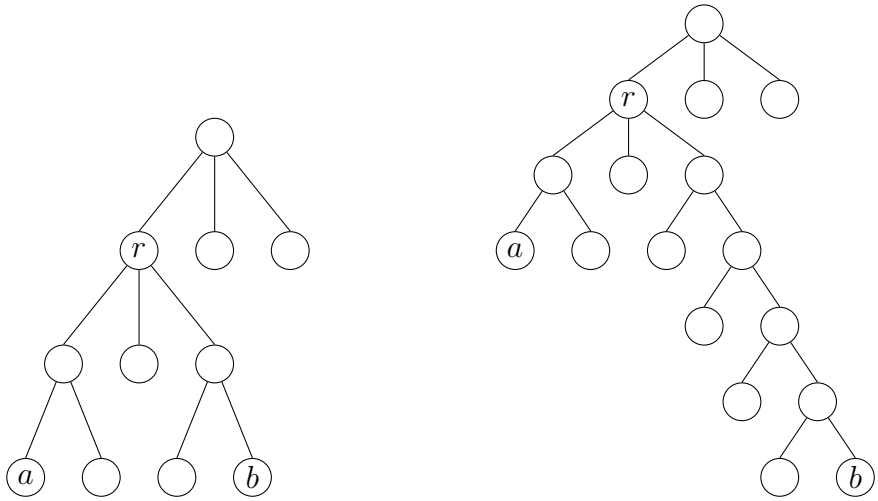
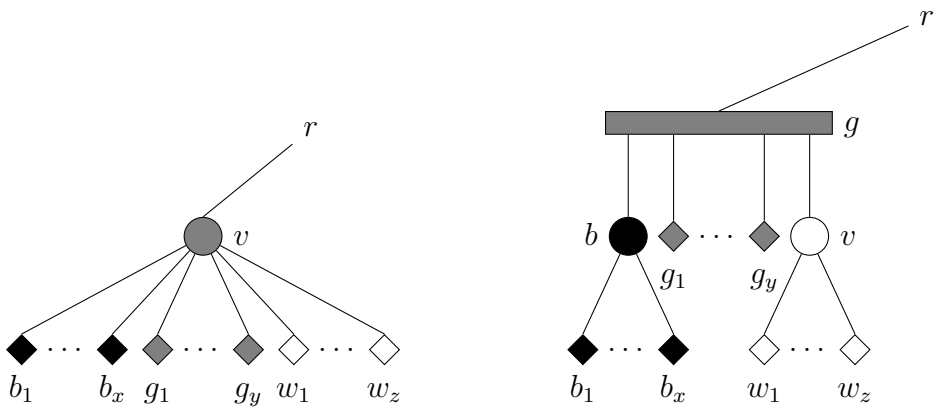


Figura 4.1: Exemplos para o comportamento da variável *off_the_top*. Na Figura (a), o caso em que *off_the_top* = 0 e, na Figura (b), o caso em que *off_the_top* = 1.

com ele todos os filhos brancos. Tal procedimento é descrito no Algoritmo 14 e ilustrado na Figura 4.2. O resultado final é o mesmo do Algoritmo 7, mas a complexidade pode ser bem menor.

Algoritmo 14: Transformar nó P em nó Q.

```
criar nó cinza g do tipo Q filho de r após v
se v tem 2 ou mais filhos negros então
    criar nó negro b do tipo P filho de g
    para cada filho negro n de v faça
        mover n para b
senão
    para cada filho negro n de v faça
        mover n para g
para cada filho cinza c de v faça
    mover c para g
se v tem 2 ou mais filhos então
    mover v para g
    pintar v de branco
senão
    se v tem um filho w então
        mover w para g
    destruir v
```



(a) T' (b) T''

Figura 4.2: Operação "transformar nó P em nó Q" que não itera sobre nós brancos. Observe a extrema semelhança com a Figura 3.2. A única diferença é nos rótulos dos nós g e v.

Após o terceiro passo da redução, o passo em que se ajusta a raiz também deve ser alterado. Quando r é do tipo P, continua passando pela operação de unir filhos negros, que já vimos. Já no caso em que r é um nó Q, é possível verificar se os filhos negros da raiz são consecutivos sem iterar por todos os nós, usando o Algoritmo 15. Isto é feito apenas visitando os vizinhos dos nós negros, cujas referências estão armazenadas. Se todos os filhos negros de v são consecutivos, todos os nós negros formam um só bloco e apenas dois vizinhos não negros (isto é, brancos ou nulos) serão encontrados. Este procedimento permite executar o quarto passo da redução em $O(|B(r)|)$, que é a nova complexidade de tempo tanto da operação de unir filhos negros quanto da operação de ajustar nó Q.

Algoritmo 15: Ajustar nó Q.

```
1 white_count ← 0
2 para cada filho negro b de r faça
3 |   white_count ← white_count+ número de irmãos brancos ou nulos de b
4 se white_count > 2 então
5 |   alterar tipo de r para R
```

Aqui também aproveitamos para descolorir a árvore em tempo $O(|C|)$, utilizando o Algoritmo 16. Além das cores dos nós, as variáveis utilizadas no Algoritmo 13 devem ser zeradas e as listas de filhos coloridos devem ser esvaziadas.

Tabela 4.2: Complexidade de tempo para cada operação com remoção em tempo constante.

Operação	Complexidade
Unir filhos negros	$O(\|B(r)\|)$
Transformar nó P em nó Q	$O(\|B(v)\| + \|G(v)\|)$
Mover filhos para fora da raiz	$O(\|G(r)\|)$
Reverter condicionalmente nó cinza	$O(\|v\|)$
Fundir com a raiz	$O(\|v\|)$

Algoritmo 16: Descolorir a árvore.

```
1  Q ← ∅
2  para cada x ∈ C faça
3  |  v ← folha de T correspondente a x
4  |  adicionar v a Q
5  enquanto Q.length > 0 faça
6  |  v ← remover elemento de Q
7  |  zerar v.pertinent_children, v.pertinent_leaves, v.visited
8  |  descolorir v
9  |  esvaziar listas de filhos
10 |  se v.pai.pertinent_leaves > 0 então
11 |  |  adicionar v.pai a Q
```

Também é importante inicializar todas estas variáveis quando um nó é pintado de branco, caso do nó v no Algoritmo 14, se v tiver dois ou mais filhos brancos.

Apesar de todas estas mudanças, a complexidade de tempo de adicionar uma restrição continua sendo $O(n^2)$, pois as operações de reverter um nó e fundir com a raiz continuam necessitando de tempo linear em relação ao número de filhos do nó. Veremos a seguir como é possível melhorar também estas operações para obter maiores ganhos em eficiência.

4.3 Estrutura de union-find

A seguir, podemos melhorar a operação de fundir com a raiz.

É possível fundir duas listas ligadas em tempo constante (veja Algoritmo 17), porém não podemos aproveitar esta característica das listas porque é necessário atualizar as referências ao nó pai de cada filho. Uma maneira de evitar ter que administrar todas estas referências durante a fusão é utilizar estruturas de union-find, como veremos a seguir.

Algoritmo 17: Inserir todos os nós de uma lista b após o nó v.

```
1 b.first.prev ← v
2 b.last.next ← v.next
3 v.next.prev ← b.last
4 v.next ← b.first
```

Façamos uma breve recordação do funcionamento da estrutura de union-find.

Este tipo de estrutura de dados serve para, dado um conjunto de elementos particionado em subconjuntos disjuntos, determinar eficientemente a qual subconjunto um elemento pertence. Um elemento de cada subconjunto é escolhido para representar o subconjunto.

Três operações básicas são suportadas: *make-set*, que cria um conjunto com um único elemento; *union*, que une dois conjuntos, dados os seus representantes, e retorna o representante do conjunto unido; e *find*, que determina a qual conjunto um elemento pertence (retorna seu representante).

Os algoritmos de union-find mais eficientes permitem operações de *union* em tempo constante e *find* em tempo médio $O(\alpha(n_e))$, onde $\alpha(x)$ é a função inversa de Ackermann e cresce muito lentamente e n_e é o número de elementos na estrutura de union-find [5]. Este é tambem o limitante inferior de tempo para estes algoritmos [21].

Nas árvores PQR, cada subconjunto é formado pelos filhos de um nó do tipo Q ou R. Como nós do tipo P não são fundidos, filhos de nós P continuam tendo referências para seus pais normalmente. Mas entre os filhos de nós Q ou R, somente o nó representante tem a referência para o pai.

Com o uso dessa estrutura, determinar o pai de um nó passa a ser um pouco mais complicado que apenas ler um campo. Para nós que não tem o campo *pai* definido, que é o caso de filhos de nós Q ou R que não são representantes, deve-se utilizar a operação de *find* para descobrir o irmão representante e obter deste nó a referência para seu pai. Este procedimento para determinar o pai de um nó é detalhado no Algoritmo 18.

Algoritmo 18: `pai(`v`)` Algoritmo para determinar o pai de um nó v, utilizando estruturas de union-find.

```
1 se v tem a referência direta para seu pai então
2 |   retornar v.pai
3 senão
4 |   retornar pai(find(v))
```

Para inserir um novo filho que anteriormente tinha pai do tipo P em um nó do tipo Q ou R, basta fazer a união do conjunto de filhos do nó Q ou R com o conjunto unitário contendo apenas o novo filho. Na maioria das aplicações da estrutura de conjuntos disjuntos, novos

elementos não são criados durante a execução. Nas árvores PQR isto acontece, mas não há problema, pois basta executar *make-set* para o novo elemento antes da união.

Tal estrutura permite, além da inserção de um novo filho, a fusão com a raiz em tempo constante. Para tal, é feita a fusão das listas de filhos e a união dos conjuntos de filhos dos dois nós. Com isso, a fusão é realizada em tempo constante. Esta nova operação é descrita no Algoritmo 19.

Algoritmo 19: Fundir com a raiz.

```
1 r.representante ← union(v_i.representante, r.representante)
2 unir os conjuntos de filhos de v_i e r
3 inserir filhos de v_i em r após v_i
4 se r é do tipo Q e v_i é do tipo R então
5 |   alterar tipo de r para R
6 destruir v_i
```

Para garantir que a fusão seja feita em tempo constante, é necessário que cada nó Q ou R saiba qual dentre seus filhos é o representante. Esta informação é fácil de manter, já que o nó representante de um conjunto só é alterado durante a união, situação em que o pai dos nós pertencentes aos conjuntos sendo unidos é conhecido.

O uso de estruturas de union-find permite realizar a fusão com a raiz com uma complexidade de tempo constante, mas, para isso, há um preço a ser pago. Enquanto a fusão é realizada muito mais eficientemente, encontrar o pai de um nó passa a requerer uma operação de *find*, se o pai é do tipo Q ou R, deixando de ser executada em tempo constante.

Utilizando o algoritmo de florestas, com união por posto e compressão de caminhos, a complexidade amortizada de um *find* é de $O(\alpha(n_e))$, onde n_e é o número de elementos na estrutura de union-find e a função $\alpha(x)$ é a função inversa de Ackermann, definida como segue [5, Sec. 21.4]:

$$A_k(j) = \begin{cases} j+1 & \text{se } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{se } k \geq 1 \end{cases}$$

para inteiros $k \geq 0$ e $j \geq 1$, e

$$\alpha(x) = \min\{k : A_k(1) \geq x\}$$

para um inteiro $x \geq 0$.

A expressão $A_{k-1}^{(j+1)}(j)$ utiliza uma notação de iteração funcional, em que $A_{k-1}^{(0)}(j) = j$ e $A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$, para $i \geq 1$.

A função de Ackermann é conhecida por ter um crescimento muito rápido. Neste caso, temos que $A_0(1) = 2$, $A_1(1) = 3$, $A_2(1) = 7$, $A_3(1) = 2047$ e $A_4(1) > 2^{2048}$. Como

estima-se que o número de átomos no universo seja cerca de 10^{80}, diz-se que $\alpha(x)$ é no máximo 4 para qualquer entrada relevante na prática, apesar de não ser assintoticamente constante.

Como vimos, a complexidade amortizada de uma única operação de *find* é de $O(\alpha(n_e))$. Precisamos portanto calcular o número possível de elementos na union-find em função do nosso tamanho de entrada s para analisar a complexidade do uso de estruturas de union-find em nosso algoritmo. Como não removemos nós da estrutura de union-find, o limitante superior que buscamos é o número total de nós criados durante toda a execução do algoritmo.

No começo da execução do algoritmo, a árvore PQR tem $n + 1$ nós: as n folhas mais a raiz do tipo P. A cada redução, um número $O(n)$ de nós é pintado de cinza. Durante o processamento de cada um desses nós cinzas, um número constante de nós é criado. Assim, o número total de nós criados durante a execução do algoritmo é $O(nm)$ e, portanto, $O(s^2)$.

Com isto, cada *find* teria complexidade $O(\alpha(s^2))$, mas, explorando as propriedades assintóticas da função de Ackermann, podemos demonstrar que $\alpha(s^2) \leq 1+\alpha(s)$ e chegar a uma complexidade de $O(\alpha(s))$. Para tal, basta provar que, dado $k \geq 3$, temos que $A_k(1) \geq (A_{k-1}(1))^2$.

Lema 2. *Para todo $k \geq 2$ e $x \geq 1$, $A_k(x) \geq x^2$.*

Demonstração. Cormen e colegas [5, Sec. 21.4] dão expressões para $A_1(x) = 2x + 1$ e $A_2(x) = 2^{x+1}(x + 1) - 1$. A partir da expressão para $k = 2$, chegamos a

$$A_2(x) = 2^{x+1}(x + 1) - 1 \geq x2^{x+1} \geq x^2.$$

Como a função $A_k(x)$ é crescente em relação a k [5, Sec. 21.4], temos $A_k(x) \geq x^2$ para todo $k \geq 2$. □

Teorema 2. *Para todo $k \geq 3$, temos que $A_k(1) \geq (A_{k-1}(1))^2$.*

Demonstração. Pela definição da função de Ackermann, temos

$$A_k(1) = A^{(2)}_{k-1}(1) = A_{k-1}(A_{k-1}(1)).$$

Como $k \geq 3$, pelo Lema 2, chegamos a $A_{k-1}(A_{k-1}(1)) \geq (A_{k-1}(1))^2$. □

Portanto, cada acesso ao pai de um nó e, por consequência, cada movimentação de nó passam a ter, no pior caso, complexidade de tempo $O(\alpha(s))$. Entretanto, para as movimentações de nó, detalhadas no Algoritmo 20, note que a complexidade depende do tipo do pai anterior q do nó v sendo movido. No caso em que q é um nó do tipo P, a

Algoritmo 20: Mover v para p

```
1 q ← pai(v)
2 remover v de todas as listas de filhos de q
3 atualizar número de filhos de q
4 se p é do tipo P então
5 |  v.pai ← p
6 senão
7 |  p.representante ← union(v, p.representante)
8 inserir v no início da lista de filhos de p
9 atualizar número de filhos de p
```

operação `pai(`v`)` é realizada em tempo constante e, consequentemente, a movimentação é executada em tempo $O(1)$.

Uma limitação deste tipo de estrutura, em sua implementação mais eficiente, é que um nó não pode ser removido dela eficientemente. Uma consequência disto é que não se pode destruir um nó removido da árvore PQR. Um nó removido da árvore PQR que seja ou tenha sido filho de nó Q ou R deve ser conservado na union-find e apenas marcado como destruído, pois ainda pode ser necessário.

A segunda consequência, ainda mais grave, é que a estrutura de union-find exige que, uma vez que dois nós são filhos de um mesmo nó Q ou R, eles não podem deixar de ser irmãos e o pai comum tem que continuar a ser do tipo Q ou R. Isto é verdade para os algoritmos de todas as operações vistas até aqui, exceto uma. Na versão da operação de transformar nó P em nó Q que evita mover nós brancos, do Algoritmo 14, o nó v deixa de ser filho de r, que, por sua vez, pode ser do tipo Q ou R, o que fere a exigência feita. Este problema não ocorreria na versão ingênua desta operação, no Algoritmo 7, porque a movimentação que o provoca é justamente o que permite evitar movimentar os nós brancos. Para lidar com este problema, é importante notar que v vai voltar a ser filho de r, pois, neste caso, depois da operação de transformar nó P em Q, será realizada a fusão com a raiz. A sequência de operações em que ocorre este conflito é ilustrada na Figura 4.3.

A solução encontrada é então criar uma nova operação para o caso em que r é do tipo Q ou R e v é do tipo P. Esta nova operação é basicamente a união das operações de transformar nó P em nó Q, reverter condicionalmente nó cinza e fundir com a raiz, isto é, as operações que eram realizadas neste caso até agora. A diferença entre esta nova operação, que chamamos de "fundir nó P", e as anteriores, é que não é criado o nó g e, portanto, o nó v não é movido, e seus filhos coloridos são movidos diretamente para o nó r. Como o nó v não é movido, ele não precisa ser removido de nenhum subconjunto na estrutura de union-find.

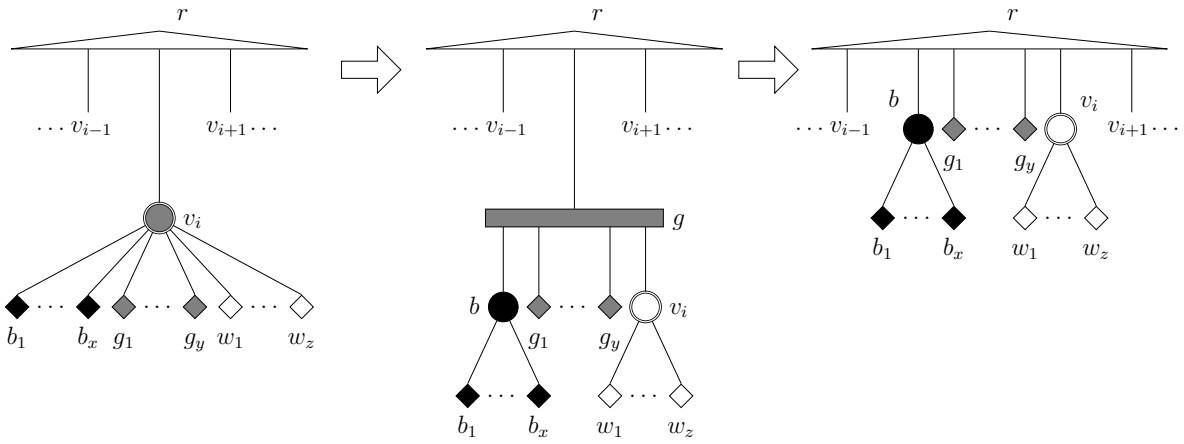


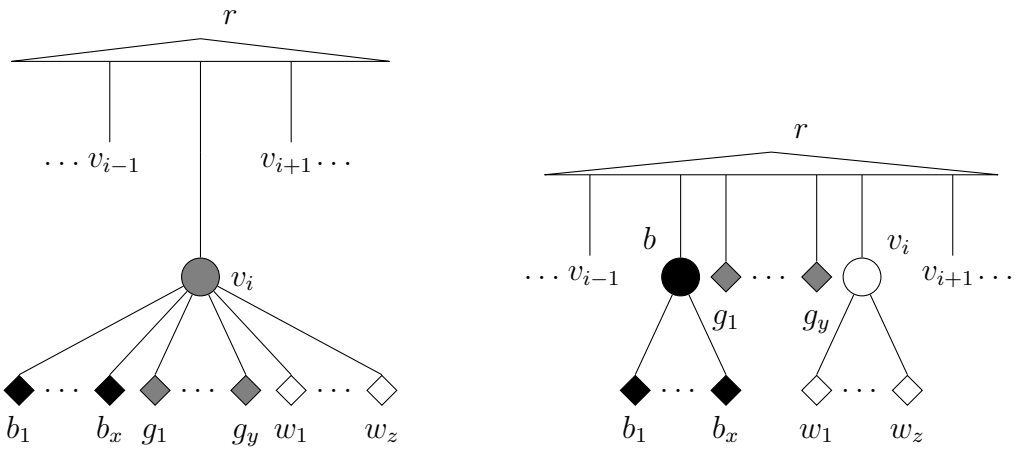
Figura 4.3: Sequência de operações em que surge o conflito com as estruturas de union-find.

Ao projetar a operação de fundir nó P, também é importante ter em mente como posicionar os nós que serão movidos, já que, como não há um nó intermediário, também não há o passo que o reverte para alinhá-lo devidamente. Por isso, os nós coloridos de v devem ser inseridos em r sempre entre v e seu vizinho mais escuro, primeiro o nó negro (todos os filhos negros de v são unidos primeiro) e depois os filhos cinzas. Assim, o resultado final desta operação é o mesmo alcançado pela execução das três operações anteriores em sequência. A nova operação é descrita no Algoritmo 21 e ilustrada na Figura 4.4.

Algoritmo 21: Fundir nó P.

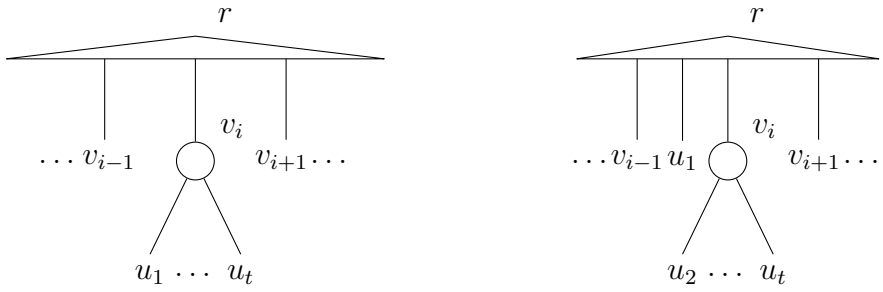
```
1  unir filhos negros de v
2  determinar direção do vizinho mais escuro de v
3  se v tem um filho negro b então
4  |  mover b para r do lado determinado de v
5  para cada filho cinza c de v faça
6  |  mover c para r do lado determinado de v
7  descolorir v
8  se v tem no máximo 1 filho então
9  |  se v tem um filho w então
10 |  |  mover w para r do lado determinado de v
11 |  destruir v
```

O vizinho mais escuro de v_i é determinado de maneira semelhante à operação anterior de reverter condicionalmente nó cinza, na Seção 3.1.6. Se v_i é um filho extremo de r, isto é, tem somente um vizinho, este vizinho é escolhido. Já se v_i tem dois vizinhos de mesma



(a) T' (b) T''

Figura 4.4: Operação "fundir nó P".



(a) T' (b) T''

Figura 4.5: Iteração da operação "fundir nó P". As cores dos nós não são representadas.

cor, qualquer um deles pode ser escolhido.

Para provar a corretude desta operação, utilizamos uma operação mais simples, de mover apenas um filho de v para r, como visto na Figura 4.5. Esta operação representa cada movimento que ocorre entre as linhas 3 e 6 do Algoritmo 21. Como a operação da primeira linha do algoritmo equivale à operação de unir filhos negros da Seção 3.1.3, basta provar a corretude da operação da Figura 4.5, que é repetida para cada filho pertinente de v_i.

Nós em T' sem equivalentes em T'': v'_i e r'.

Para o nó v'_i, temos que $\hat{v}'_i = \hat{v}''_i \cup \hat{u}_1$, que é um conjunto em $Consec(r'') \subset \overline{T}''$.

Já no caso de r', temos que tanto $Consec(r') \subset Consec(r'')$ quanto $Pot(r') \subset Pot(r'')$. Consequentemente, como o tipo de r não se altera, $\overline{r}' \subset \overline{r}''$ e, portanto, $\overline{r}' \subset \overline{T}''$.

Nós em T'' sem equivalentes em T': v''_i e r''.

Tabela 4.3: Complexidade de tempo para cada operação com estruturas de union-find.

Operação	Complexidade
Unir filhos negros	$O(\|B(r)\|)$
Transformar nó P em nó Q	$O(\|B(v)\| + \|G(v)\|)$
Mover filhos para fora da raiz	$O(\|G(r)\|\alpha(s))$
Reverter condicionalmente nó cinza	$O(\|v\|)$
Fundir com a raiz	$O(\alpha(s))$
Fundir nó P	$O(\|B(v)\| + \|G(v)\|)$

Se u_1 é o único filho negro de v'_i, temos

$$\hat{v}''_i = (\hat{v}'_i \setminus C) \uplus \hat{c}_1 \uplus \ldots \uplus \hat{c}_y,$$

em que $c_1, \ldots, c_y$ são os filhos cinzas de v'_i.

Se u_1 é cinza, temos

$$\hat{v}''_i = (\hat{v}'_i \setminus C) \uplus \hat{c}_1 \uplus \ldots \uplus \hat{c}_y \setminus \hat{u}_1,$$

onde $c_1, \ldots, c_y$ são os filhos cinzas de v'_i, exceto u_1.

Para o nó r'', precisamos utilizar o que vimos na Seção 3.1.6. O lado escolhido para mover os filhos coloridos de v_i (o lado do vizinho mais escuro de v_i no início da fusão) é o lado que tem um nó não branco, a não ser que tanto v_{i-1} quanto v_{i+1} sejam brancos.

Suponha que o lado escolhido seja o de v_{i-1}. Para começar, observe que:

$$\mathit{Consec}(r'') = \overline{\mathit{Consec}(r') \cup \{\hat{v}_{i-1} \cup \hat{u}_1\} \cup \{\hat{v}_{i+1} \cup \hat{v}''_i\}}.$$

Para o caso em que existe um nó pertinente v_d mais próximo de u''_1 que de v''_i, temos

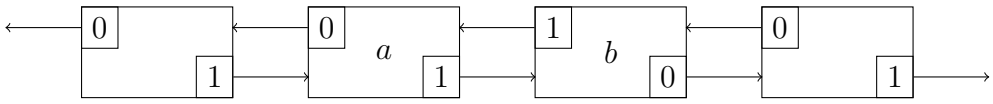
$$\hat{v}_{i-1} \cup \hat{u}_1 = (V_{[d,i]} \cap C) \uplus V_{[d,i-1]} \uplus \hat{u}_1 \setminus \hat{c}_1 \setminus \ldots \setminus \hat{c}_y \setminus V_{[1,i-2]}$$

Se v_{i-1} é branco, v_{i+1} também o é, e então podemos fazer

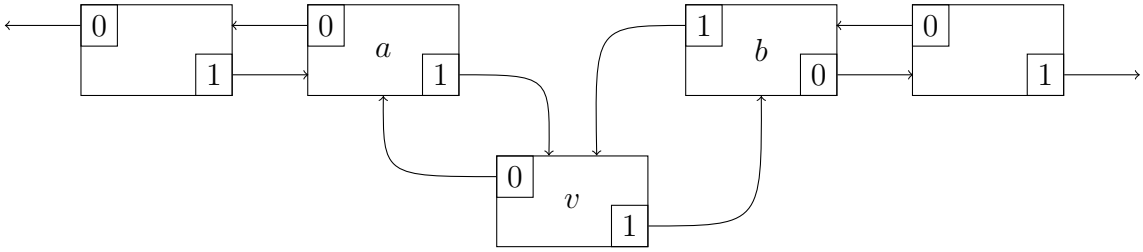
$$\hat{v}_{i+1} \cup \hat{v}''_i = V_{[i,i+1]} \setminus C \setminus \hat{u} \uplus \hat{c}_1 \uplus \ldots \uplus \hat{c}_y$$

Assim, temos um conjunto entre $(\hat{v}_{i-1} \cup \hat{u}_1)$ e $(\hat{v}_{i+1} \cup \hat{v}''_i)$ escrito a partir de conjuntos em $\overline{T'}$. Além disso, é possível derivar um do outro, da seguinte forma:

$$\hat{v}_{i-1} \cup \hat{u}_1 = V_{[i-1,i]} \setminus (\hat{v}_{i+1} \cup \hat{v}''_i)$$
$$\hat{v}_{i+1} \cup \hat{v}''_i = V_{[i,i+1]} \setminus (\hat{v}_{i-1} \cup \hat{u}_1)$$



(a) Lista antes da inserção.



(b) Nó v inserido entre os nós a e b.

Figura 4.6: Inserção de um nó v entre os nós a e b em uma lista simétrica.

4.4 Listas simétricas

O último alvo para melhorias é a operação de reverter um nó Q. Para tal, utilizamos um tipo de lista ligada chamada de lista simétrica [1], que pode ser revertida em tempo constante. O preço a pagar é que um nó em uma lista simétrica tem, em vez de referências para o próximo nó da lista e o anterior, apenas duas refrências para os vizinhos, *p[0]* e *p[1]*, sem uma direção explícita.

Devido a esta característica das listas simétricas, só é possível determinar quem é o nó seguinte em relação a um nó qualquer quando se está iterando pela lista (pois é necessário conhecer o nó anterior), o que evitamos fazer para não passar por nós brancos, como visto na Seção 4.2. Portanto, para poder utilizar as listas simétricas, outras adaptações devem ser feitas no algoritmo.

Em primeiro lugar, não é mais possível a operação de inserir um filho **após** outro filho dado. Este procedimento deve ser substituído pela operação de inserir um filho entre dois filhos consecutivos, que vemos na Figura 4.6. Note como as referências dos campos *p[0]* e *p[1]* não implicam em uma direção na lista.

Outro problema surge para fazer a reversão condicional do nó cinza. É impossível determinar qual o vizinho seguinte ou anterior para orientar devidamente o nó cinza. A solução encontrada é de realizar a verificação das cores dos vizinhos no momento em que as listas são fundidas, conforme o Algoritmo 22.

Algoritmo 22: Fundir com a raiz, utilizando listas simétricas.

unir os conjuntos de filhos de v_i e r
$r.child\_count \leftarrow r.child\_count + v_i.child\_count$
seja v_d o vizinho mais escuro de v_i
seja v_l o outro vizinho de v_i
seja f_d o filho extremo mais escuro de v_i
seja f_l o outro filho extremo de v_i
substituir em v_d a referência para v_i por f_d
substituir em v_l a referência para v_i por f_l
substituir em f_d a referência nula por v_d
substituir em f_l a referência nula por v_l
se v_i *era o primeiro filho de* r **então**
    v_l se torna o primeiro filho de r
se v_i *era o último filho de* r **então**
    v_l se torna o último filho de r
remover v_i

Capítulo 5

Algoritmo online final e complexidade

Neste capítulo, reapresentamos o algoritmo online em sua versão final e analisamos sua complexidade.

5.1 Algoritmo final

Esta seção reúne todos os algoritmos anteriores utilizados na versão final do algoritmo.

Junto aos algoritmos, analisamos a complexidade de cada operação, com um comentário em cada linha, indicando a complexidade de tempo da operação executada naquela linha. No caso de um laço, o comentário na linha da condição indica a complexidade da verificação da condição, multiplicada pelo número de vezes em que é feita. Para todas as operações dentro de um laço, a complexidade indicada é a complexidade de tempo de uma execução da operação, multiplicada pelo número de execuções. Ao fim de cada algoritmo, um comentário indica a complexidade de tempo total para a execução do algoritmo. Alguns algoritmos não tem uma análise de complexidade aqui. Estes requerem uma análise mais sofisticada, que veremos na próxima seção.

Algoritmo 23: Redução de árvores PQR

```
1 Inicializar T como uma árvore universal
2 para cada C ∈ 𝒞 faça
3 |   Adicionar C a T (Algoritmo 24)
```

Algoritmo 24: Algoritmo para adicionar um conjunto C a uma árvore T.

1 Colorir a árvore em relação a C e encontrar a raiz da subárvore pertinente
2 Reestruturar a árvore, reparando nós cinzas (Algoritmo 3)
3 Ajustar a raiz
4 Descolorir a árvore

Algoritmo 26: Algoritmo para reestruturar a árvore, reparando nós cinzas. (Passo 3 do Algoritmo 24)

```
1  enquanto existe um filho cinza v da raiz r faça
2      se r é do tipo P então
3          se v é do tipo P então
4              transformar nó P em nó Q
5          unir filhos negros
6          mover filhos para fora da raiz
7      senão
8          se v é do tipo P então
9              fundir nó P
10         senão
11             fundir com a raiz
```

Algoritmo 27: Descolorir a árvore.

```
1  Q ← ∅                                                          // O(1)
2  para cada x ∈ C faça                                           // O(|C|)
3      v ← folha de T correspondente a x                          // O(|C|)
4      adicionar v a Q                                            // O(|C|)
5  enquanto Q.length > 0 faça                                     // O(|C|)
6      v ← remover elemento de Q                                  // O(|C|)
7      zerar v.pertinent_children, v.pertinent_leaves, v.visited  // O(|C|)
8      descolorir v                                               // O(|C|)
9      esvaziar listas de filhos coloridos                        // O(|C|)
10     se pai(v) ≠ null e pai(v).pertinent_leaves > 0 então       // O(|C|α(s))
11         adicionar pai(v) a Q                                   // O(|C|α(s))
                                                                  // Total: O(|C|α(s))
```

Algoritmo 25: Colorir a árvore e encontrar a raiz da subárvore pertinente.

```
1  Q ← ∅
2  off_the_top ← 0
3  para cada x ∈ C faça
4  |  v ← folha de T correspondente a x
5  |  v.visited ← true
6  |  adicionar v a Q
7  enquanto Q.length + off_the_top > 1 faça
8  |  v ← remover elemento de Q
9  |  p ← pai(v)
10 |  se p ≠ null então
11 |  |  p.pertinent_children ← p.pertinent_children + 1
12 |  |  se not p.visited então
13 |  |  |  p.visited ← true
14 |  |  |  adicionar p a Q
15 |  senão
16 |  |  off_the_top ← 1
17 Q ← ∅
18 para cada x ∈ C faça
19 |  v ← folha de T correspondente a x
20 |  v.pertinent_leaves ← 1
21 |  adicionar v a Q
22 enquanto Q.length > 0 faça
23 |  v ← remover elemento de Q
24 |  se v.pertinent_leaves = |C| então
25 |  |  retornar v
26 |  p ← pai(v)
27 |  p.pertinent_children ← p.pertinent_children − 1
28 |  p.pertinent_leaves ← p.pertinent_leaves + v.pertinent_leaves
29 |  se v é folha ou todos os filhos de v são negros então
30 |  |  pintar v de preto
31 |  |  adicionar v à lista de filhos negros de p
32 |  senão
33 |  |  pintar v de cinza
34 |  |  adicionar v à lista de filhos cinzas de p
35 |  se p.pertinent_children = 0 então
36 |  |  adicionar p a Q
```

Algoritmo 28: Unir filhos negros

```
1  se r tem 2 ou mais filhos negros então                          // O(1)
2  |  criar nó negro b do tipo P filho de r                       // O(1)
3  |  para cada filho negro n de r faça                          // O(|B(r)|)
4  |  |  mover n para b                                          // O(|B(r)|)
                                                  // Total:  O(|B(r)|)
```

Algoritmo 29: Mover para fora da raiz

```
1  se f_1.cor < f_t.cor então                                      // O(1)
2  |  reverter v                                                  // O(1)
3  se r tem um filho negro b então                                 // O(1)
4  |  mover b para o início de v                                  // O(1)
5  para cada filho cinza c de r diferente de v faça              // O(|G(r)|)
6  |  mover c para o início de v                                  // O(|G(r)|)
7  se r tem apenas um filho então                                  // O(1)
8  |  se r é a raiz da árvore então                               // O(1)
9  |  |  v torna-se a raiz da árvore                              // O(1)
10 |  senão
11 |  |  u ← pai(v)                                              // O(α(s))
12 |  |  mover v para u                                          // O(1)
13 |  descolorir v                                                // O(1)
14 |  destruir r                                                  // O(1)
                                          // Total:  O(|G(r)| + α(s))
```

Algoritmo 30: Transformar nó P em nó Q.

```
1  criar nó cinza g do tipo Q filho de r                          // O(1)
2  se v tem 2 ou mais filhos negros então                         // O(1)
3  |  criar nó negro b do tipo P filho de g                      // O(1)
4  |  para cada filho negro n de v faça                          // O(|B(v)|)
5  |  |  mover n para b                                          // O(|B(v)|)
6  senão
7  |  para cada filho negro n de v faça                          // O(1)
8  |  |  mover n para g                                          // O(1)
9  para cada filho cinza c de v faça                              // O(|G(v)|)
10 |  mover c para g                                             // O(|G(v)|)
11 se v tem 2 ou mais filhos então                                // O(1)
12 |  mover v para g                                             // O(1)
13 |  descolorir v                                               // O(1)
14 senão
15 |  se v tem um filho w então                                  // O(1)
16 |  |  mover w para g                                          // O(1)
17 |  destruir v                                                 // O(1)
                                                  // Total: O(|B(v)| + |G(v)|)
```

Algoritmo 31: Fundir nó P.

```
1  unir filhos negros de v                                        // O(|B(v)|)
2  determinar direção do vizinho mais escuro de v                 // O(1)
3  se v tem um filho negro b então                                // O(1)
4  |  mover b para r do lado determinado de v                    // O(1)
5  para cada filho cinza c de v faça                              // O(|G(v)|)
6  |  mover c para r do lado determinado de v                     // O(|G(v)|)
7  descolorir v                                                   // O(1)
8  se v tem no máximo 1 filho então                               // O(1)
9  |  se v tem um filho w então                                  // O(1)
10 |  |  mover w para r do lado determinado de v                 // O(1)
11 |  destruir v                                                 // O(1)
                                                  // Total: O(|B(v)| + |G(v)|)
```

Algoritmo 32: Fundir com a raiz, utilizando listas simétricas.

unir os conjuntos de filhos de v_i e r // $O(1)$
$r.child\_count \leftarrow r.child\_count + v_i.child\_count$ // $O(1)$
seja v_d o vizinho mais escuro de v_i // $O(1)$
seja v_l o outro vizinho de v_i // $O(1)$
seja f_d o filho extremo mais escuro de v_i // $O(1)$
seja f_l o outro filho extremo de v_i // $O(1)$
substituir em v_d a referência para v_i por f_d // $O(1)$
substituir em v_l a referência para v_i por f_l // $O(1)$
substituir em f_d a referência nula por v_d // $O(1)$
substituir em f_l a referência nula por v_l // $O(1)$
se v_i *era o primeiro filho de* r **então** // $O(1)$
    v_l se torna o primeiro filho de r // $O(1)$
se v_i *era o último filho de* r **então** // $O(1)$
    v_l se torna o último filho de r // $O(1)$
remover v_i // $O(1)$
// Total: $O(1)$

Algoritmo 33: Ajustar a raiz.

se r *é do tipo* P **então** // $O(1)$
    unir filhos negros de r // $O(|B(r)|)$
senão se r *é do tipo* Q **então** // $O(1)$
    ajustar nó Q // $O(|B(r)|)$
// Total: $O(|B(r)|)$

Algoritmo 34: Ajustar nó Q.

$white\_count \leftarrow 0$ // $O(1)$
para cada *filho negro* b *de* r **faça** // $O(|B(r)|)$
    $white\_count \leftarrow white\_count +$ número de irmãos brancos ou nulos de b // $O(|B(r)|)$
se $white\_count > 2$ **então** // $O(1)$
    alterar tipo de r para R // $O(1)$
// Total: $O(|B(r)|)$

A seguir, a Tabela 5.1 sintetiza as complexidades de tempo para as operações de cada versão do algoritmo, em função dos parâmetros $n = |U|$, $m = |\mathcal{C}|$, $f = \sum_{C \in \mathcal{C}} |C|$ e $s = n + m + f$. Estas complexidades são calculadas em relação à execução total do

Tabela 5.1: Comparação das complexidades de tempo para cada operação em cada versão do algoritmo. Todos os valores são referentes a comportamentos assintóticos. Os parâmetros são: n, o tamanho do universo; m, o número de restrições; f, a soma dos tamanhos das restrições e $s = n + m + f$. A notação $O()$ foi suprimida por questões de espaço e clareza, assim como os nomes de algumas operações foram abreviados.

Operação	Ingênuo	Listas	Brancos	Union-find	Symlists
Unir filhos negros	mn^2	mn	f	f	f
Transformar P em Q	mn^3	mn^2	s^2	s	s
Mover para fora	mn^2	mn	s	$s + m\alpha(s)$	$s + m\alpha(s)$
Reverter cond.	mn^2	mn^2	mn^2	sn	-
Fundir com a raiz	mn^3	mn^2	mn^2	s	s
Fundir nó P	-	-	-	s	s
Colorir a árvore	mn	mn	s	$s\alpha(s)$	$s\alpha(s)$
Descolorir a árvore	mn	mn	f	$f\alpha(s)$	$f\alpha(s)$
Total	mn^3	mn^2	mn^2	$sn + s\alpha(s)$	$s\alpha(s)$

algoritmo, isto é, o processamento de todos os conjuntos em $\mathcal{C}$. Assim, por exemplo, a operação "unir filhos negros", em sua última versão, tem complexidade de tempo em cada execução de $O(|B(r)|)$. Como $|B(r)|$ é $O(|C|)$ e esta operação é executada no máximo uma vez a cada redução (quando r é do tipo P), a soma do tempo de todas as execuções de "unir filhos negros" é $O(f)$.

Como já foi dito, uma análise mais sofisticada do algoritmo final, que permite chegar à complexidade final de $O(s\alpha(s))$, é feita na seção seguinte.

5.2 Análise de complexidade do algoritmo final

Pelas complexidades indicadas na seção anterior, uma análise direta sugeriria que a complexidade de reduzir uma árvore PQR T em relação a um conjunto C seria quadrática em relação ao número de nós cinzas da árvore, levando ainda a um resultado de $O(n^2)$. Uma análise amortizada, porém, nos leva a um resultado mais interessante. Esta análise se baseia no fato de que algumas operações são executadas no máximo uma vez durante uma redução: transformar nó P em nó Q, unir filhos negros e mover filhos para fora da raiz. Tais operações são efetuadas somente se a raiz da subárvore pertinente r é um nó do tipo P e, depois da operação de mover filhos para fora da raiz, o nó r passa a ser do tipo Q ou R e não volta mais a ser um nó tipo P.

Em primeiro lugar, para fazer a análise amortizada da complexidade de tempo da redução, apresentamos aqui o conceito de **potencial** de uma árvore PQR, que é fundamental para o resultado principal desta seção. O potencial de uma árvore PQR T é

definido similarmente ao do trabalho de Booth e Lueker sobre árvores PQ [2]:

$$NORM(T) = \sum_{v \text{ nó P}} |filhos(v)| + \sum_{v \text{ nó Q/R}} 1.$$

O resultado a seguir nos dá uma ideia de como muda o potencial ao adicionarmos um conjunto.

Teorema 3. *Sejam T uma árvore PQR e T' a árvore obtida da redução de T em relação a C. Seja r a raiz da subárvore pertinente de T em relação a C. O ganho em potencial $\Delta NORM(T, C) = NORM(T') - NORM(T)$ satisfaz*

$$\Delta NORM(T, C) \leq |B(r)| - \sum_{v \text{ nó P cinza}} |G(v)| - \sum_{v \text{ nó Q/R cinza}} 1 + 1.$$

Demonstração. Os possíveis nós em T' que não têm equivalentes em T são:

- Um nó P com os filhos negros de r (da operação unir os filhos negros).
- Para cada nó P cinza, um nó P com seus filhos negros e outro com seus filhos brancos (da operação transformar nó P em Q).
- Um nó Q ou R filho da raiz da subárvore pertinente do tipo P (das operações transformar nó P em Q e mover filhos para fora da raiz).

Por outro lado, os nós em T que não têm equivalentes em T' são os nós cinzas.
Então, temos

$$\begin{aligned}\Delta NORM(T, C) &\leq |B(r)| + \sum_{v \text{ nó P cinza}} (|B(v)| + |W(v)|) + 1 - \\ &\quad - \sum_{v \text{ nó P cinza}} |filhos(v)| - \sum_{v \text{ nó Q/R cinza}} 1 \\ &\leq |B(r)| - \sum_{v \text{ nó P cinza}} |G(v)| - \sum_{v \text{ nó Q/R cinza}} 1 + 1.\end{aligned}$$

□

Lema 3. *Dada uma árvore PQR T e uma restrição C, o número de nós negros em T colorida em relação a C é menor que $2|C|$.*

Demonstração. Tomados isoladamente do resto da árvore, os nós negros formam uma floresta cujas folhas são as $|C|$ folhas negras de T. Como os nós internos desta floresta têm pelo menos dois filhos, são no máximo $|C| - 1$. □

Para o próximo teorema, utilizamos mais uma definição presente no trabalho de Booth e Lueker: o grafo $PRUNED(T, C)$, que é o menor subgrafo conexo de T que contém todas as folhas pertinentes em relação a C, isto é, os nós coloridos da árvore, mais a raiz da subárvore pertinente. O número de nós em $PRUNED(T, C)$ pode ser escrito de duas formas que nos serão úteis:

$$|PRUNED(T, C)| = 1 + \sum_{v \text{ nó negro}} 1 + \sum_{v \text{ nó cinza}} 1$$

ou

$$|PRUNED(T, C)| = 1 + \sum_{v \in T} (|B(v)| + |G(v)|).$$

Adotaremos ainda a convenção de que $|PRUNED(T, C)| = 1$ quando $C = \emptyset$.

Teorema 4. *A redução de uma árvore PQR T em relação a um conjunto C tem complexidade de tempo de $O(\alpha(s)|PRUNED(T, C)|)$.*

Demonstração. Dividimos a demonstração para os quatro passos da redução:

1. Colorir a árvore em relação a C e encontrar a raiz da subárvore pertinente

   Durante a primeira passagem, são visitados os nós de $PRUNED(T, C)$ mais alguns nós extras, ancestrais da raiz da subárvore pertinente. No entanto, a fila utilizada para decidir o nó a ser visitado garante que, se o nó v é visitado antes do nó u, o pai de v será visitado antes do pai de u, a não ser que o pai de u tenha sido enfileirado anteriormente, por outro filho. Cada um destes nós extras, porém, tem apenas um filho visitado durante a passagem e, portanto, o número de nós extras visitados é, no máximo, a altura de $PRUNED(T, C)$. Como cada nó é visitado somente uma vez nesta passagem e cada visita toma tempo $O(\alpha(s))$, a primeira passagem toma tempo $O(\alpha(s)|PRUNED(T, C)|)$.

   A segunda passagem visita somente os nós de $PRUNED(T, C)$. Aqui, cada nó é visitado somente uma vez, em tempo $O(\alpha(s))$. Assim, a complexidade de tempo da segunda passagem também é $O(\alpha(s)|PRUNED(T, C)|)$.

2. Reestruturar a árvore, reparando nós cinzas

   A complexidade de tempo deste passo é formada pela soma de quatro parcelas, que veremos separadamente.

   A primeira parcela vem da operação unir filhos negros. Esta operação é executada no máximo uma vez a cada redução, pois se a raiz da subárvore pertinente é do tipo P, a operação posterior de mover filhos para fora da raiz fará com que a raiz da subárvore pertinente se torne um nó Q ou R durante a primeira execução do

laço principal do algoritmo. Portanto, esta operação adiciona uma complexidade de tempo de $O(|B(r)|)$, que é $O(\alpha(s)|PRUNED(T,C)|)$.

A operação mover filhos para fora da raiz, como vimos, também só é executada no máximo uma vez a cada redução e, portanto, adiciona $O(|G(r)| + \alpha(s))$ à complexidade de tempo da redução.

Para cada nó v cinza do tipo P, é executada a operação transformar nó P em nó Q ou a operação fundir nó P. Ambas tem complexidade de $O(|B(v)| + |G(v)|)$. Assim, a complexidade total adicionada por estas operações à redução é de

$$O\left(\sum_{v \text{ nó P cinza}} (|B(v)| + |G(v)|)\right).$$

Por fim, para cada nó v cinza do tipo Q ou R, é executada a operação fundir com a raiz, em tempo constante. Portanto, a complexidade adicionada por esta operação é

$$O\left(\sum_{v \text{ nó Q/R cinza}} 1\right).$$

3. Ajustar a raiz

   Ajustar a raiz toma tempo $O(|B(r)|) = O(\alpha(s)|PRUNED(T,C)|)$.

4. Descolorir a árvore

   Descolorir a árvore é feito de forma semelhante à coloração. Entretanto, não existem nós cinzas nesta fase, então são visitados $O(|C|)$ nós, cada um em tempo $O(\alpha(s))$. Assim, descolorir a árvore é $O(|C|\alpha(s))$.

□

Teorema 5. *Existe constante $k > 0$ tal que*

$$|PRUNED(T,C)| + \Delta NORM(T,C) \leq k|C|.$$

Demonstração. Somando $|PRUNED(T,C)|$ à inequação do Teorema 3, temos

$$|PRUNED(T,C)| + \Delta NORM(T,C) \leq 1 + \sum_{v \text{ nó negro}} 1 + \sum_{v \text{ nó cinza}} 1 + |B(r)| - \\ - \sum_{v \text{ nó P cinza}} |G(v)| - \sum_{v \text{ nó Q/R cinza}} 1 + 1.$$

Simplificando, chegamos a

$$|PRUNED(T,C)| + \Delta NORM(T,C) \leq 2 + \sum_{v \text{ nó negro}} 1 + \sum_{v \text{ nó P cinza}} (1 - |G(v)|) + |B(r)|.$$

Analisemos separadamente cada parcela desta soma. Primeiro, podemos assumir que $|C| \geq 2$, pois, caso contrário, não há alteração na árvore. Para a segunda parcela da soma, sabemos, pelo Lema 3, que o número de nós negros na árvore é menor que $2|C|$.

Se v tem pelo menos um filho cinza, temos que $1 - |G(v)| \leq 0$. Assim, se v é um nó cinza de T, $1 - |G(v)|$ é igual a um se e somente se v não tem filhos cinzas, e neste caso tem pelo menos um filho negro e o resto de seus filhos são brancos. Logo,

$$\sum_{v \text{ nó P cinza}} (1 - |G(v)|) \leq \sum_{v \text{ nó negro}} 1 \leq 2|C|.$$

Finalmente, o conjunto $B(r)$ tem no máximo um nó para cada folha negra de T. Então, $|B(r)| \leq |C|$.

Portanto, temos que

$$|PRUNED(T,C)| + \Delta NORM(T,C) \leq 6|C|.$$

□

Somando o resultado deste último teorema para todos os conjuntos $C_1, C_2, \ldots, C_m$ adicionados à árvore PQR universal T_0 até obter a árvore T_m, temos

$$\sum_{i=0}^{m-1} |PRUNED(T_i, C_{i+1})| + NORM(T_m) - NORM(T_0) = O(f + m).$$

Como $NORM(T_m) \geq 0$ e $NORM(T_0) = n$, chegamos a um valor amortizado para $|PRUNED(T,C)|$ durante toda a execução do algoritmo:

$$\sum_{i=0}^{m-1} |PRUNED(T_i, C_{i+1})| = O(f + m + n) = O(s).$$

E, assim, concluímos que a complexidade amortizada do algoritmo online de construção de árvores PQR que apresentamos é de $O(\alpha(s)s)$.

Capítulo 6

Conclusões

Neste último capítulo, apresentamos nossas considerações finais a respeito do trabalho realizado neste mestrado e também sobre as árvores PQR.

6.1 Conclusões

A árvore PQR é uma alternativa para tratar problemas envolvendo o problema dos uns consecutivos. Esta estrutura de dados busca ter algoritmos mais simples e intuitivos que outros algoritmos presentes na literatura. Além disso, as árvores PQR permitem trabalhar com entradas que não tenham a propriedade dos uns consecutivos, o que é importante em muitas aplicações.

O algoritmo online estudado tem uma complexidade assintótica de tempo maior que outros algoritmos da literatura, lineares, mas na prática isto pode não ser uma desvantagem relevante, pois o termo $\alpha(s)$ cresce lentamente o suficiente para ser desprezado em aplicações práticas. Pensamos que as vantagens do método que apresentamos sejam mais importantes que o pequeno aumento na complexidade de tempo.

Como vimos na introdução, existe um algoritmo offline linear para construir árvores PQR. Existem problemas no algoritmo original de Dahlhaus para a obtenção das uniões das componentes do grafo de sobreposição estrita [6], utilizado por McConnell [15], mas foram posteriormente resolvidos por Charbit e colegas [3].

6.2 Contribuições

Neste trabalho, consolidamos o conhecimento presente nos trabalhos anteriores de Telles e Meidanis sobre a construção online de árvores PQR [17, 24, 25]. Descrevemos detalhadamente o projeto do algoritmo online quase-linear e apresentamos uma nova demonstração

da complexidade do algoritmo, mais clara, direta e mais próxima do trabalho de Booth e Lueker [2]. Além disso, lidamos também com uma questão importante sobre o uso de estruturas de union-find descoberta durante o trabalho, na Seção 4.3.

Com isso, apresentamos também uma primeira implementação completa do algoritmo online quase-linear, em Java.

6.3 Trabalhos futuros

O mais imediato desdobramento possível deste trabalho seria uma análise teórica e prática aprofundada das informações fornecidas pelos nós do tipo R. Do ponto de vista teórico, podem-se buscar heurísticas ou mesmo algoritmos aproximados para problemas NP-Difíceis, como encontrar o máximo subconjunto dos elementos que admita uma solução, encontrar o número mínimo de mudanças de 0 para 1 na matriz que produza uma solução [8], ou minimizar o número de blocos de uns em uma matriz [10]. Na prática, esse estudo envolveria uma avaliação dos resultados obtidos com o uso dessas heurísticas ou algoritmos aproximados em aplicações, tanto em relação ao tempo de computação quanto à qualidade das respostas. O nó R já é utilizado em aplicações de bioinformática [4] e visualização de dados [22].

Outra possibilidade para um estudo futuro é investigar o comportamento da árvore a cada redução. Conforme a árvore é reduzida em relação a novas restrições, novos nós surgem do tipo P, se transformam em Q e depois em R. É possível que o comportamento da árvore durante as reduções e esta progressão dos tipos de nós dê novas informações sobre o problema dos uns consecutivos.

Referências Bibliográficas

[1] Christian Bachmaier and Marcus Raitner. Improved symmetric lists. Technical report, University of Passau, 2004.

[2] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer Systems Science*, 13(3):335–379, 1976.

[3] P. Charbit, M. Habib, V. Limouzy, F. De Montgolfier, M. Raffinot, and M. Rao. A note on computing set overlap classes. *Information Processing Letters*, 108(4):186–191, 2008.

[4] C. Chauve and E. Tannier. A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genomes. *PLoS Computational Biology*, 4(11), 2008.

[5] Thomas H. Cormen, Ronald L. Rivest, Charles E. Leiserson, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2^{nd} edition, 2001.

[6] E. Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36(2):205–240, 2000.

[7] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[9] Sakti P. Ghosh. File organization: the consecutive retrieval property. *Communications of the ACM*, 15(9):802–808, 1972.

[10] P. W. Goldberg, M. C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *Journal of Computational Biology*, 2(1):139–152, 1995.

[11] David Greenberg and Sorin Istrail. Physical mapping by STS hybridization: Algorithmic strategies and the challenge of software evaluation. *Journal of Computational Biology*, 2(2):219–274, 1995.

[12] Michel Habib, Ross M. McConnell, Christophe Paul, and Laurent Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.

[13] W.L. Hsu. A simple test for the consecutive ones property. *Journal of Algorithms*, 43(1):1–16, 2002.

[14] D. G. Kendall. Incidence matrices, interval graphs and seriation in archaeology. *Pacific Journal of Mathemathics*, 28(3):565–570, 1969.

[15] R.M. McConnell. A certifying algorithm for the consecutive-ones property. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, page 777. Society for Industrial and Applied Mathematics, 2004.

[16] J. Meidanis, O. Porto, and G. P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88:325–354, 1998.

[17] J. Meidanis and G. P. Telles. Building PQR trees in almost-linear time. In *Proceedings of GRACO*, 2005. Summarized version of the Technical Report 03-26 (Institute of Computing, University of Campinas, 2003).

[18] João Meidanis and Erasmo G. Munuera. A theory for the consecutive ones property. In *Proceedings of WSP'96 - Third South American Workshop on String Processing*, pages 194–202, August 1996.

[19] Erasmo G. Munuera. Propriedade dos uns consecutivos e reconhecimento de grafos intervalo. Master's thesis, University of Campinas, 1996.

[20] Mark B. Novick. Generalized PQ-trees. Technical report, Cornell University, Ithaca, NY, USA, 1989.

[21] J. A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 34–44, New York, NY, USA, 1990. ACM.

[22] Celmar G. da Silva, Marivaldo F. de Melo, Felipi de Paula e Silva, and João Meidanis. PQR-sort — Using PQR-trees for binary matrix reorganization. Submitted for publication, 2011.

[23] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, April 1975.

[24] G. P. Telles and J. Meidanis. Building PQR trees in almost linear time. Technical Report 03-26, Institute of Computing, University of Campinas, November 2003.

[25] Guilherme Pimentel Telles. *Um algoritmo quase-linear para árvores PQR e um esquema para clustering de seqüências expressas de cana-de-açúcar*. PhD thesis, University of Campinas, 2002.

[26] João Paulo Pereira Zanetti and João Meidanis. Construção incremental de árvores PQR. Technical Report IC-10-31, Institute of Computing, University of Campinas, October 2010. In Portuguese, 18 pages.