**Um algoritmo quase-linear para árvores PQR e um esquema para *clustering* de seqüências expressas de cana-de-açúcar**

*Guilherme Pimentel Telles*

**Tese de Doutorado**

Instituto de Computação
Universidade Estadual de Campinas

# Um algoritmo quase-linear para árvores PQR e um esquema para *clustering* de seqüências expressas de cana-de-açúcar

**Guilherme Pimentel Telles**[1]

fevereiro de 2003

**Banca Examinadora:**

- Prof. Dr. João Meidanis (Orientador)
  Instituto de Computação – UNICAMP
- Prof. Dr. Eduardo Laber
  Departamento de Informática – PUC-Rio
- Prof. Dr. Arnaldo Mandel
  Instituto de Matemática e Estatística – USP
- Prof. Dr. Arnaldo Moura
  Instituto de Computação – UNICAMP
- Prof. Dr. Jorge Stolfi
  Instituto de Computação – UNICAMP
- Prof. Dr. Oscar Porto (Suplente)
  Departamento de Engenharia Elétrica – PUC-Rio
- Prof. Dr. Flávio Miyazawa (Suplente)
  Instituto de Computação – UNICAMP

[1]**Financiado pelo CNPq e pela FAPESP.**

Substituir pela ficha catalografica

# Um algoritmo quase-linear para árvores PQR e um esquema para *clustering* de seqüências expressas de cana-de-açúcar

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida por Guilherme Pimentel Telles e aprovada pela banca examinadora.

Campinas, 12 de dezembro de 2002.

Prof. Dr. João Meidanis (Orientador)
Instituto de Computação – UNICAMP

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

Substituir pela página de assinaturas

# Agradecimentos

Ao meu orientador, o Prof. João Meidanis, por ele ter me aturado esse tempo todo.

Ao CNPq e à Fapesp pelo apoio financeiro.

Ao Dr. Felipe R. da Silva pelas aulas de biologia molecular e pela lição de bioinformática mais importante que eu já recebi.

Ao Prof. Jorge Stolfi por ele ter apontado um erro nas nossas análises do algoritmo PQR e por ter indicado a forma de contorná-lo.

À Profa. Cecília Rubira pelas sugestões na modelagem das classes para implementar as árvores PQR.

Por último, mas importantíssimo, quero agradecer a todas as pessoas que se preocuparam comigo e se interessaram pelo desenrolar deste trabalho. Não vou me atrever a nomeá-las para não esquecer nenhuma. Mas a regra é simples: se alguma vez você me encontrou e perguntou "E a tese?", obrigado!

"Olhe, eu podia mesmo contar-lhe a minha vida inteira, em que há outras coisas interessantes, mas para isso era preciso tempo, ânimo e papel, e eu só tenho papel; o ânimo é frouxo, e o tempo assemelha-se à lamparina de madrugada. Não tarda o sol do outro dia, um sol dos diabos, impenetrável como a vida. Adeus, meu caro senhor, leia isto e queira-me bem; perdoe-me o que lhe parecer mau, e não maltrate muito a arruda, se lhe não cheira a rosas. Pediu-me um documento humano, ei-lo aqui. Não me peça também o império do Grão-Mogol, nem a fotografia dos Macabeus; peça, porém, os meus sapatos de defunto e não os dou a ninguém mais."

(Machado de Assis, *O Enfermeiro*)

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Resumo

Nesta tese apresentamos um algoritmo quase-linear para construir árvores PQR e o esquema para *clustering* de seqüências expressas que foi usado no projeto SUCEST (*Sugarcane EST project*).

Uma árvore PQR é uma estrutura de dados capaz de resolver o problema dos uns consecutivos e outros problemas, como mapeamento físico de DNA, reconhecimento de grafos intervalo, otimização de circuitos lógicos e recuperação de dados. As árvores PQR são uma generalização das árvores PQ de Booth e Lueker e estão fundamentadas em propriedades algébricas sólidas. O algoritmo que apresentamos nesta tese se baseia nessas propriedades e é formado por um conjunto pequeno de padrões que modificam a árvore. Nosso algoritmo é mais eficiente que o algoritmo quadrático proposto originalmente por Meidanis e Munuera ao definir as árvores PQR e, embora não seja linear como o algoritmo para construir árvores PQ, acreditamos que ele contribui em direção a uma solução definitiva para o problema dos uns consecutivos.

*Clustering* é o problema de categorização de um conjunto de objetos quando nem o número nem a composição das categorias é conhecido antecipadamente. Seqüências expressas ou ESTs (*expressed sequence tags*) são amostras de genes ativos extraídas das células de um organismo. Em um projeto genoma EST são obtidas milhares dessas seqüências que serão usadas como fonte para investigações por cientistas interessados nos processos celulares do organismo. O *clustering* de seqüências expressas é necessário para avaliar e reduzir a redundância do conjunto de ESTs. Nesta tese apresentamos o esquema de *clustering* usado no projeto da cana-de-açúcar, que produziu aproximadamente 300.000 ESTs. O esquema que apresentamos substituiu um esquema pré-existente e se caracteriza por uma limpeza prévia intensiva dos ESTs e pelo uso de um montador de genomas para agrupá-los. Acreditamos que este esquema possa ser utilizado em outros projetos do gênero.

# Abstract

In this thesis we introduce an almost-linear algorithm for building PQR trees and the method for expressed sequence tags clustering used in the SUCEST project (Sugarcane EST Project).

A PQR tree is a structure that can be used to solve many problems, as the consecutive ones problem, DNA physical mapping, interval graphs recognition and data retrieval. PQR trees are a generalization of Booth and Lueker's PQ trees, and are founded on solid algebraic properties. The algorithm we present in this work is based on such properties and is composed by a small set of well-organized patterns. Our algorithm is more efficient than the quadratic one proposed by Meidanis and Munuera, who defined the PQR trees, and, although not linear as the algorithm for PQ trees construction, we believe that it contributes for a definite solution for the consecutive ones problem.

Clustering is the problem of categorizing a set of objects when neither the number of categories nor the composition of the categories is known. Expressed sequence tags (ESTs) are samples from active genes extracted from cells of an organism. In an EST project, thousands of ESTs are produced and used as a source for research. Clustering ESTs is necessary to reduce the redundancy in the set of sequences. In this thesis we introduce the method used in the Sugarcane EST Project, that produced almost 300,000 sequences. The method we introduce in this work replaced another one that had problems. Our scheme include an intensive trimming of the ESTs and the use of a genome assembler for the whole set of sequences. We believe that our scheme may be used in other EST projects.

# Capítulo 1

# Introdução

Esta tese é composta por duas partes que contêm nossas principais contribuições ao longo do doutoramento, que desenvolveu-se ao mesmo tempo que colaborávamos com o Laboratório de Bioinformática do Instituto de Computação da Unicamp, entre 1998 e 2002. Este laboratório foi o que supriu as necessidades de informática, incluindo sistemas de software e poder de processamento, dos projetos genoma da bactéria *Xylella fastidiosa* (o primeiro projeto genoma brasileiro), da bactéria *Xantomonas citri* e da cana-de-açúcar, financiados pela Fapesp e levados a cabo pela Rede Onsa, a rede formada por laboratórios paulistas para seqüenciamento e análise de dados genômicos.

Nossa primeira contribuição é teórica: um algoritmo quase-linear para a construção de árvores PQR. A segunda é aplicada: um esquema de *clustering* de seqüências expressas que foi usado no projeto genoma da cana-de-açúcar (SUCEST – *Sugarcane EST Project*).

### Primeira parte: algoritmo quase-linear para árvores PQR

Dada uma coleção de $m$ subconjuntos $C_1, C_2, \ldots, C_m$ de um conjunto $U$ de $n$ elementos, o *problema dos uns consecutivos* consiste em responder se há uma permutação dos elementos de $U$ que mantém os elementos de cada conjunto $C_i$ consecutivos. Se há tal permutação, então a coleção possui a *propriedade dos uns consecutivos (P1C)*.

Muitos problemas podem ser modelados como o dos uns consecutivos: mapeamento físico de DNA [31], recuperação de dados [15], reconhecimento de grafos intervalo [14], isomorfismo de grafos intervalo [22], planaridade de grafos [6] e otimização de circuitos lógicos [13, 17].

O problema dos uns consecutivos foi resolvido com o algoritmo polinomial de Fulkerson e Gross [14] em 1965. Em 1976, Booth e Leuker [6] inventaram as árvores PQ, uma estrutura de dados para resolver o problema e representar todas as permutações de $U$ sujeitas a $C_1, C_2, \ldots, C_m$ de forma compacta. Eles também apresentaram um algoritmo linear para construir as árvores PQ. Em 1995, Meidanis e Munuera [23] criaram as árvores

PQR, uma generalização das árvores PQ e apresentaram um algoritmo polinomial para construí-las. Em uma continuação desse trabalho publicada em 1998, Meidanis, Porto e Telles [24] estenderam a teoria algébrica por trás das árvores PQR.

As árvores PQR podem ser usadas para resolver os mesmos problemas que as árvores PQ, com algumas vantagens. Em primeiro lugar, o fato de que as árvores PQR estão fundamentadas em propriedades algébricas sólidas. Em segundo lugar, o fato de quando a coleção não tem a P1C, as árvores PQ são nulas mas as árvores PQR vão apontar subcoleções responsáveis pela falha [24]. Esta característica pode ser útil quando a coleção representa dados que podem conter erros. Esta situação é comum na construção de mapas físicos de DNA, o que faz com que da árvore PQR uma ferramenta com grande potencial na solução deste problema em particular. Em terceiro lugar, para construir as árvores PQR há um algoritmo recursivo que não faz casamento de padrões na árvore e que reflete várias propriedades das coleções e da P1C.

Nesta tese damos mais um passo em direção a uma solução definitiva para o problema, apresentando um algoritmo quase-linear para construir árvores PQR. Nosso algoritmo está fundamentado em propriedades teóricas sólidas das árvores e usa um conjunto pequeno de padrões bem organizados. O Capítulo 2 apresenta as definições relacionadas às árvores PQR que são necessárias nos capítulos seguintes. O Capítulo 3 define as árvores PQR marcadas e as operações de redução destas árvores que são a base do nosso algoritmo. No Capítulo 4 apresentamos o algoritmo quase-linear para construir árvores PQR, prova de correção, análise de complexidade e descrição da implementação que fizemos.

### Segunda parte: *clustering* de seqüências expressas de cana-de-açúcar

Uma seqüência expressa ou EST (*expressed sequence tag*) [1] é uma amostra de um gene ativo retirada aleatoriamente de células de um organismo. Na célula, essa seqüência dará origem a uma proteína. O interesse dos biólogos moleculares nessas seqüências é o de conhecer as proteínas produzidas pela célula e estudar os processos celulares que influenciam o organismo em foco. A partir desse conhecimento, tais cientistas tentam desenvolver terapias para combater doenças que afetam o organismo e também tentam produzir variedades melhoradas do próprio organismo.

O *clustering* [11] é um processo de categorização de objetos no qual nem a quantidade de categorias nem a composição das categorias é conhecida antecipadamente. Quando os objetos são seqüências expressas, o *clustering* é importante para avaliar e reduzir a redundância do conjunto de seqüências que serão analisadas pelos biólogos e auxiliar nas análises.

Nesta tese apresentamos o esquema de *clustering* que foi usado no Projeto SUCEST, que produziu aproximadamente 300.000 seqüências expressas de cana-de-açúcar [39]. O nosso esquema de *clustering* substituiu um esquema anterior, que vinha apresentando problemas, durante o andamento do projeto. Em nosso esquema incluímos uma limpeza prévia intensiva das seqüências e usamos apenas um montador de genomas para produzir as categorias. A vantagem principal de usar apenas um montador de genomas para o *clustering* é que cada categoria tem uma seqüência consenso, que é uma superseqüência de todas as outras na categoria. Os parâmetros do nosso esquema foram ajustados com base em testes com os dados reais do projeto SUCEST. Nós acreditamos que esse esquema possa ser usado em outros projetos genoma do mesmo tipo. No Capítulo 5 apresentamos uma síntese sobre *clustering*, explicamos em mais detalhes as características principais dos ESTs e citamos os principais métodos já utilizados para o *clustering* de ESTs. Esse capítulo serve de introdução ao Capítulo 6 que apresenta o esquema de *clustering* do Projeto SUCEST.

# Capítulo 2

# Propriedade dos Uns Consecutivos e Árvore PQR

Neste capítulo apresentamos a definição de *árvore PQR*, uma estrutura de dados capaz de armazenar todas as soluções possíveis para o *problema dos uns consecutivos*. Também apresentamos outras definições e propriedades relacionadas à árvore que serão usadas ao longo desta tese. A maioria delas vem do trabalho de Meidanis, Porto e Telles [24].

## 2.1 Propriedade dos Uns Consecutivos

Chamamos um conjunto de conjuntos de **coleção** e denotamo-lo usando maiúsculas caligráficas, como $\mathcal{C}$. O **conjunto potência** do conjunto $U$ é formado por todos os subconjuntos de $U$ e é denotado $\mathcal{P}(U)$. Um conjunto $A \subseteq U$ é **trivial** se $A = \emptyset$, $|A| = 1$ ou $A = U$.

Uma **permutação** de um conjunto $U$, $|U| = n$, é uma bijeção $\alpha : \{1, 2, \ldots, n\} \to U$. Um conjunto $A \subseteq U$ é **consecutivo** em $\alpha$ se todos os elementos de $A$ aparecem em $\alpha$ consecutivamente. Uma coleção $\mathcal{A}$ é consecutiva em $\alpha$ se cada um dos conjuntos de $\mathcal{A}$ é consecutivo em $\alpha$. Por exemplo, seja $U = \{a, b, c, d, e, f\}$ e seja $\alpha = fdeacb$ uma permutação dos elementos de $U$. O conjunto $A = \{d, c, a, e\}$ é consecutivo em $\alpha$, mas o conjunto $B = \{f, e, b, c\}$ não é.

Uma coleção $\mathcal{C} \subseteq \mathcal{P}(U)$ possui a **propriedade dos uns consecutivos** (P1C) se existir uma permutação $\alpha$ de $U$ em que $\mathcal{C}$ seja consecutiva. Se existe tal permutação ela é **válida** para $\mathcal{C}$. A propriedade dos uns consecutivos é caracterizada em matrizes binárias [14], coleções de conjuntos [23], grafos bipartidos [37] e conjuntos de intervalos sobre a reta real [10].

O problema dos uns consecutivos consiste em verificar se uma coleção possui a P1C. No caso de coleções de conjuntos, o tamanho de uma instância para o problema é $n+m+r$, onde $n = |U|$, $m = |\mathcal{C}|$ e $r = \sum_{A \in \mathcal{C}} |A|$.

Para uma coleção que possui a P1C podemos definir o conjunto de todas as permutações que verificam a propriedade:

**Definição 1** *O conjunto de todas as permutações de $U$ válidas para uma coleção $\mathcal{C}$ é*

$$Valid(\mathcal{C}) = \{\alpha : \alpha \text{ é válida para } \mathcal{C}\}.$$

## 2.2 Árvore PQR

Uma árvore PQR $T$ é uma árvore enraizada, de aridade variável, na qual a ordem relativa dos filhos de um nó importa, que tem quatro tipos de nós — P, Q, R e folhas — e que deve obedecer às seguintes restrições à sua estrutura interna:

- As folhas estão em bijeção com os elementos de um conjunto $U$.
- Cada nó P tem pelo menos dois filhos.
- Cada nó Q tem pelo menos três filhos.
- Cada nó R tem pelo menos três filhos.

Na Figura 2.1 temos dois exemplos de árvores PQR. No exemplo, nós P são representados por círculos, nós Q por retângulos e nós R pela letra R circunscrita.

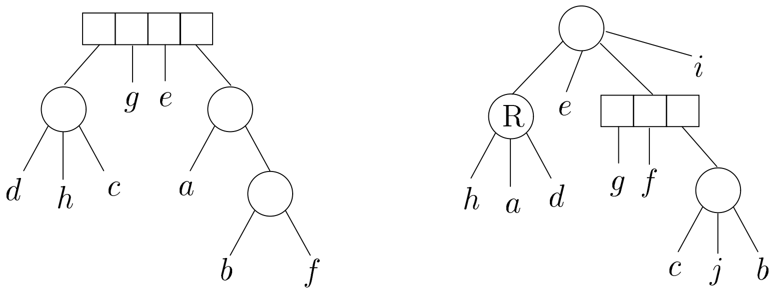



Figura 2.1: Exemplos de árvores PQR.

Uma árvore PQR pode ser transformada em outras árvores equivalentes a ela. As **transformações de equivalência** a seguir podem ser aplicadas a uma árvore PQR:

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

Dadas duas árvores PQR $T'$ e $T''$, elas são **equivalentes**, $T' \equiv T''$, se e somente se uma delas puder ser transformada na outra pela aplicação de zero ou mais transformações de equivalência. Duas árvores PQR equivalentes, $T'$ e $T''$, aparecem na Figura 2.2.

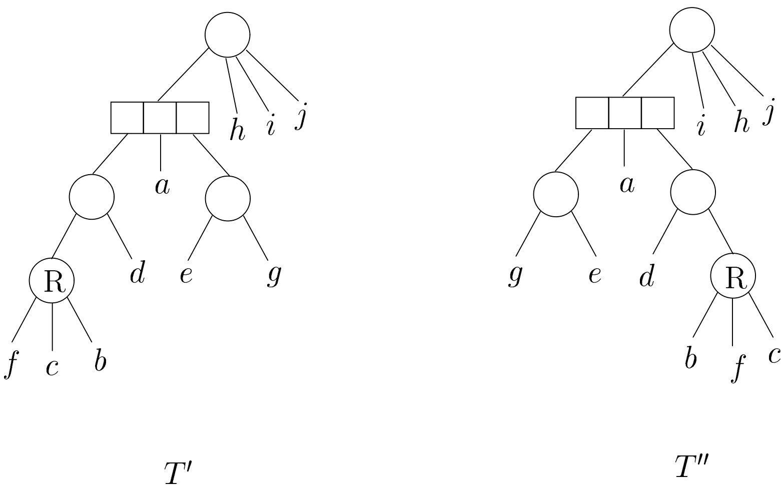



Figura 2.2: Árvores PQR equivalentes.

Uma permutação das folhas de uma árvore PQR pode ser obtida através da sua fronteira:

**Definição 2 ([24])** *A* **fronteira** *de uma árvore PQR $T$ é a permutação do conjunto $U$ obtida pela leitura das folhas da árvore da esquerda para a direita.*

Na Figura 2.2, a fronteira da árvore PQR $T'$ é *fcbdaeghij*. O conjunto de todas as fronteiras de uma árvore é o conjunto $Compat(T)$:

**Definição 3 ([24])** *Dada uma árvore PQR $T$, o conjunto de todas as fronteiras de árvores equivalentes a $T$, chamadas* **permutações compatíveis**, *é*

$$Compat(T) = \{\alpha : \alpha \text{ é a fronteira de } T' \text{ e } T' \equiv T\}.$$

Árvores PQR com apenas um nó interno recebem nomes especiais. São as árvores P-total, Q-total e R-total:

**Definição 4** *Uma árvore* **P-total**, **Q-total** *ou* **R-total** *é uma árvore PQR em que todas as folhas são filhas da raiz que é, respectivamente, do tipo P, Q ou R. Uma árvore P-total também é chamada de* **árvore universal**.

As árvores PQR são uma generalização das árvores PQ de Booth e Lueker [6]. Assim como suas antecessoras, as árvores PQR são capazes de representar todas as permutações válidas para uma coleção $\mathcal{C} \subseteq \mathcal{P}(U)$ que tem a propriedade dos uns consecutivos, através das fronteiras de árvores equivalentes. Quando $\mathcal{C}$ não tem a P1C, a árvore PQ é nula, mas a árvore PQR não é e os nós R fornecem informação sobre as subcoleções de $\mathcal{C}$ que estão obstruindo a propriedade [24].

## 2.3 P1C e Árvores PQR

Para toda coleção $\mathcal{C}$ podemos construir uma árvore PQR $T$ tal que

1. se $\mathcal{C}$ tem a P1C, então $T$ não terá nenhum nó R e as fronteiras de árvores equivalentes a $T$ serão permutações válidas para $\mathcal{C}$ e

2. se $\mathcal{C}$ não tem a P1C, então $T$ terá pelo menos um nó R.

Por outro lado, toda árvore PQR $T$ corresponde a uma coleção $\mathcal{C}$ que terá a P1C se e somente se $T$ não tem nós R.

Estas propriedades são formalizadas através das coleções $\overline{\mathcal{C}}$ e $Compl(T)$. Estas coleções são as mais importantes para caracterizar as árvores PQR e para mostrar a correção dos algoritmos para construí-las.

### 2.3.1 A coleção $\overline{\mathcal{C}}$

A coleção $\overline{\mathcal{C}}$ é obtida a partir de uma coleção $\mathcal{C} \subseteq \mathcal{P}(U)$. Ela contém todos os conjuntos de $\mathcal{P}(U)$ que podem ser obtidos a partir dos conjuntos em $\mathcal{C}$ pela aplicação de certas operações que preservam a consecutividade.

**Definição 5 ([24])** *Dada uma coleção $\mathcal{C} \subseteq \mathcal{P}(U)$, a coleção $\overline{\mathcal{C}}$ é a menor coleção que contém os conjuntos em $\mathcal{C}$ e os conjuntos triviais contidos em $U$ e que é fechada pelas seguintes operações:*

1. **interseção**: $A \cap B$.

2. **união não disjunta**: $A \uplus B = A \cup B$ *desde que* $A \cap B \neq \emptyset$.

3. **diferença não contida**: $A \between B = A \setminus B$ *desde que* $B \not\subseteq A$.

Algumas propriedades de $\overline{\mathcal{C}}$ que serão úteis ao longo do texto estão agrupadas no próximo lema.

**Lema 6 ([24])** *Seja $\mathcal{C}$ uma coleção de conjuntos. Então*

$$\mathcal{C} \subseteq \overline{\mathcal{C}}$$
$$\overline{\overline{\mathcal{C}}} = \overline{\mathcal{C}}$$

### 2.3.2 A coleção $Compl(T)$

A coleção $Compl(T)$ é uma coleção obtida a partir de uma árvore PQR $T$, tomando-se conjuntos de folhas descendentes dos nós de $T$. Esta coleção é fechada pelas operações 1, 2 e 3 acima.

Antes de apresentar a definição de $Compl(T)$ vamos definir conjuntos e coleções de folhas descendentes de nós de uma árvore PQR.

**Definição 7 ([24])** *O* **conjunto de folhas descendentes** *de um nó $v$ em uma árvore PQR $T$ é*

$$Desc_T(v) = \{a \in U \ : \ v \text{ é ancestral de } a \text{ em } T\}.$$

*Para um conjunto $A$ de nós,*

$$Desc_T(A) = \bigcup_{v \in A} Desc_T(v).$$

**Definição 8** *A* **coleção de conjuntos de folhas descendentes** *de um nó $w$ em $T$, $\mathcal{D}_T(w)$, é definida assim:*

1. *Se $w$ é uma folha então*

$$\mathcal{D}_T(w) = \{\{w\}\}.$$

2. *Se $w$ é um nó $P$ então*

$$\mathcal{D}_T(w) = \{Desc_T(w)\}.$$

3. *Se $w$ é um nó $Q$ então*

$$\mathcal{D}_T(w) = \{Desc_T(A) : A \text{ é um conjunto de filhos consecutivos de } w\}.$$

4. *Se $w$ é um nó $R$ então*

$$\mathcal{D}_T(w) = \{Desc_T(A) : A \text{ é um conjunto qualquer de filhos de } w\}.$$

**Definição 9** *Dada uma árvore PQR $T$, a coleção $Compl(T)$ é*

$$\bigcup_{v \text{ é nó de } T} \mathcal{D}_T(v) \cup \{\emptyset\}.$$

Para ilustrar, a coleção $Compl(T')$ da árvore na Figura 2.2 é

$$\begin{array}{l}
\{\emptyset, \\
\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}, \\
\{a, b, c, d, e, f, g, h, i, j\}, \\
\{f, c, b, d, a, e, g\}, \{f, c, b, d, a\}, \{a, e, g\}, \\
\{f, c, b, d\}, \\
\{f, c, b\}, \{f, c\}, \{c, b\}, \{f, b\}, \\
\{e, g\}\}
\end{array}$$

### 2.3.3 Relação entre P1C e Árvores PQR

O próximo teorema define uma **árvore PQR para uma coleção $\mathcal{C}$**.

**Teorema 10 ([24])** *Para toda coleção $\mathcal{C} \subseteq \mathcal{P}(U)$ existe uma árvore PQR $T$ tal que*

$$Compl(T) = \overline{\mathcal{C}}.$$

Os dois próximos teoremas relacionam coleções, árvores PQR e permutações válidas. Eles mostram que o conjunto de permutações válidas para uma coleção $\mathcal{C}$ que tem a P1C é armazenado pela árvore PQR que tem $Compl(T)$ igual a $\overline{\mathcal{C}}$.

**Teorema 11 ([24])** *Se uma árvore PQR $T$ não tem nós R então*

$$Valid(Compl(T)) = Compat(T).$$

**Teorema 12 ([24])** *Sejam $\mathcal{C}$ um coleção que tem a P1C e $T$ uma árvore PQR. Se $Compl(T) = \overline{\mathcal{C}}$ então*

$$Valid(\mathcal{C}) = Compat(T).$$

## 2.4 O algoritmo de Meidanis e Munuera

Um algoritmo para construir árvores PQR pode ser de dois tipos: *off-line* e *on-line*. Um algoritmo *off-line* processa todos os conjuntos em $\mathcal{C}$ de uma única vez e constrói a árvore. Um algoritmo *on-line* começa com uma árvore universal e adiciona os conjuntos em $\mathcal{C}$ um a um, obtendo árvores intermediárias para as subcoleções processadas.

Um algoritmo *on-line* pode ser vantajoso para as árvores PQR porque possibilita *backtracking* à medida que a árvore é construída e em conjunto com a detecção de nós R pode determinar uma boa estratégia para lidar com erros nos dados que fazem com que uma coleção não tenha a P1C.

O algoritmo de Booth e Lueker para construir árvores PQ [6] é *on-line* e tem complexidade $O(n+m+r)$. Para cada conjunto, o algoritmo colore alguns nós da árvore e depois casa e substitui padrões, que levam em conta o tipo e a cor dos nós. Se a coleção $\mathcal{C}$ não tem a P1C, o algoritmo de Booth e Lueker retorna uma árvore nula.

O algoritmo de Meidanis e Munuera [23] para construir árvores PQR é *off-line*, recursivo e tem complexidade $O((m+n+r)^2n^2)$ [34]. A cada chamada recursiva, o algoritmo divide a coleção em duas, de tal forma que uma parte está relacionada a uma subárvore completa da árvore que está sendo construída e a outra parte ao restante da árvore. Para dividir a coleção o algoritmo escolhe um conjunto de folhas que descende de um nó da árvore. Conjuntos desse tipo são que estão na interseção entre as coleções $\overline{\mathcal{C}}$ e $\mathcal{C}^\perp$. A coleção $\mathcal{C}^\perp$ é definida abaixo.

**Definição 13** *Dois conjuntos $A$ e $B$ são* **ortogonais**, *$A \perp B$, se*

- *$A \subseteq B$ ou*
- *$B \subseteq A$ ou então*
- *$A \cap B = \emptyset$.*

*Dado um conjunto $A$ e uma coleção $\mathcal{C}$, dizemos que $A$ é ortogonal a $\mathcal{C}$, $A \perp \mathcal{C}$, quando $A \perp B$ para todo $B \in \mathcal{C}$. Da mesma forma, dizemos que duas coleções $\mathcal{C}$ e $\mathcal{D}$ são ortogonais, $\mathcal{C} \perp \mathcal{D}$, quando $A \perp B$ para todo $A \in \mathcal{C}$ e $B \in \mathcal{D}$.*

**Definição 14** *Dada uma coleção $\mathcal{C} \subseteq \mathcal{P}(U)$, a coleção $\mathcal{C}^\perp$ é*

$$\mathcal{C}^\perp = \{A : A \in \mathcal{P}(U), A \perp \mathcal{C}\}.$$

Algumas propriedades da coleção $\mathcal{C}^\perp$ aparecem no próximo lema.

**Lema 15 ([24])** *Para toda coleção $\mathcal{C}$,*

$$\mathcal{C}^\perp = \overline{\mathcal{C}^\perp} = \overline{\mathcal{C}}^\perp.$$

O próximo lema mostra que todo conjunto de folhas que descendem de um nó $v$ de uma árvore PQR $T$ para uma coleção $\mathcal{C}$ é um conjunto em $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$.

**Lema 16 ([24])** *Para qualquer nó $v$ de uma árvore PQR $T$,*

$$Desc_T(v) \in Compl(T) \cap Compl(T)^{\perp}.$$

*Por outro lado, qualquer conjunto em $Compl(T) \cap Compl(T)^{\perp}$ é da forma $Desc_T(v)$ para algum nó $v$ de $T$.*

Para exemplificar, sejam

$$U = \{a, b, c, d, e, f, g, h\},$$
$$\mathcal{C} = \{\{e, g\}, \{b, f\}, \{a, b, e, f\}, \{c, d, g, h\}\}.$$

Fazendo

$$\mathcal{T} = \{\emptyset, U, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}\}$$

Obtemos

$$\overline{\mathcal{C}} = \mathcal{C} \cup \mathcal{T} \cup \{\{a, b, e, f, g\}, \{c, d, e, g, h\}, \{a, b, f\}, \{c, d, h\}\},$$
$$\mathcal{C}^{\perp} = \mathcal{T} \cup \{\{b, f\}, \{c, d\}, \{c, h\}, \{d, h\}, \{a, b, f\}, \{c, d, h\}\},$$
$$\overline{\mathcal{C}} \cap \mathcal{C}^{\perp} = \mathcal{T} \cup \{\{b, f\}, \{a, b, f\}, \{c, d, h\}\}.$$

Uma árvore para essa coleção aparece na Figura 2.3a. Na Figura 2.3b aparece uma árvore para

$$U = \{a, b, c, d, e, f, g, h\}$$
$$\mathcal{C}' = \{\{a, d\}, \{a, h\}, \{d, h\}, \{f, g\}, \{c, b, f\}\}.$$

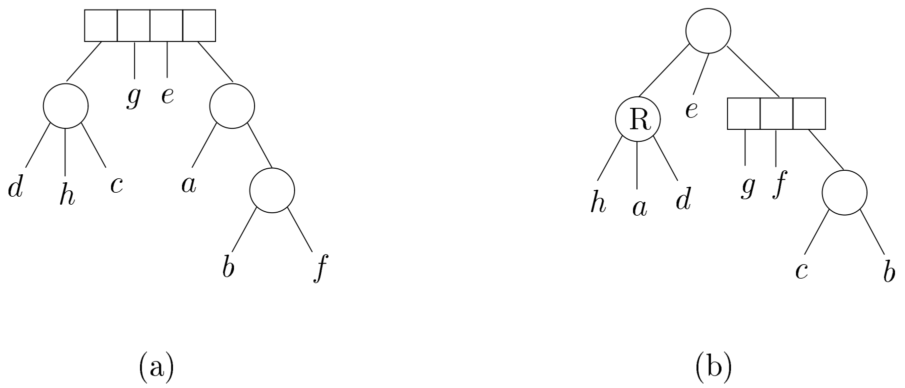



Figura 2.3: Árvores PQR para $(U, \mathcal{C})$ e $(U, \mathcal{C}')$.

O algoritmo de Meidanis e Munuera aparece na Figura 2.4. A cada chamada recursiva, um conjunto $X \in \overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$ é usado para particionar a coleção $\mathcal{C}$ e construir duas novas subinstâncias:

**Função** $PQR(U, \mathcal{C})$
  **Se** existe um conjunto não-trivial $X$ em $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$ **então**
    $T_1 \leftarrow PQR(U/X, \mathcal{C}/X)$;
    $T_2 \leftarrow PQR(X, \mathcal{C} \wedge X)$;
    **Retorne** $T_1$ com a folha $X$ substituída por $T_2$;
  **Senão**
    **Escolha**
      $\mathcal{C}$ contém apenas conjuntos triviais: **Retorne** *Nó-P*$(U)$);
      Existe permutação $\alpha$ válida em $\mathcal{C}$: **Retorne** *Nó-Q*$(\alpha)$;
      Senão: **Retorne** *Nó-R*$(U)$;
    **fim**

Figura 2.4: Algoritmo recursivo para a construção de árvores PQR.

1. A subinstância formada pelo par $(U/X, \mathcal{C}/X)$, onde

   $U/X = (U \setminus X) \cup \{x\}$,
   $A/X = A$ se $X \cap A = \emptyset$ ou $(A \setminus X) \cup \{x\}$ caso contrário,
   $\mathcal{C}/X = \{A/X : A \in \mathcal{C}$ e $(X \subset A$ ou $X \cap A = \emptyset)\}$,
   $x$ é um elemento que não pertence a $U$.

   Esta subinstância vai dar origem à árvore cujo conjunto de folhas é $(U \setminus X) \cup \{x\}$, onde $x$ é uma nova folha e representa a subárvore determinada pela outra subinstância.

2. A subinstância formada pelo par $(X, \mathcal{C} \wedge X)$, onde

   $$\mathcal{C} \wedge X = \{A : A \in \mathcal{C}, A \subset X\}.$$

   Esta subinstância vai dar origem a uma subárvore completa cujo conjunto de folhas é $X$.

Por exemplo, sejam

$U = \{a, b, c, d, e, f\}$,
$\mathcal{C} = \{\{a, c, e, f\}, \{b, c, d, e, f\}, \{a, b, d\}, \{e, f\}, \{c, e\}\}$,
$X = \{c, e, f\}$.

Então,

$(U/X, \mathcal{C}/X) = (\{a, b, d, x\}, \{\{a, x\}, \{b, d, x\}, \{a, b, d\}\})$
$(X, \mathcal{C} \wedge X) = (\{c, e, f\}, \{\{e, f\}, \{c, e\}\})$

Quando todos os conjuntos na interseção $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$ de uma instância são triviais, dizemos que a coleção é prima. Nesse caso a instância vai dar origem a um único nó pai de todas as folhas de $U$. Se $\mathcal{C}$ tem apenas conjuntos triviais, o nó é do tipo P. Senão, ele é do tipo Q se houver uma permutação válida para $\mathcal{C}$ ou R caso contrário.

Para encontrar conjuntos em $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$, Meidanis e Munuera sugerem dois métodos. O primeiro é encontrar uniões de componentes conexas do grafo de sobreposições estritas de $\mathcal{C}$. O segundo é encontrar classes de equivalência da relação de gêmeos em $\mathcal{C}$, quando todas as uniões de componentes conexas do grafo de sobreposições estritas de $\mathcal{C}$ são triviais. O teste para verificar se há uma permutação válida quando a coleção é prima pode ser feito com base no próprio grafo de sobreposições estritas. Os detalhes sobre estes procedimentos e a prova de que a decomposição das instâncias funciona podem ser encontrados no trabalho de Meidanis, Porto e Telles [24].

# Capítulo 3

# Árvore PQR marcada

Neste capítulo definimos a *árvore PQR marcada* e apresentamos um conjunto de operações para transformá-la. Estas operações compõem nosso algoritmo quase-linear para construir árvores PQR.

## 3.1 Definições

Deste ponto em diante vamos usar a seguinte notação: dado um nó $v$ de uma árvore PQR $T$, o símbolo $v$ será usado para representar tanto o nó quanto o conjunto

$$v = Desc_T(v).$$

A seguir apresentamos a definição de árvore PQR marcada e de ancestral comum mais baixo nessa árvore.

**Definição 17** *Uma* **árvore PQR marcada** *pelo conjunto $S$ é uma árvore PQR com nós pretos, cinzas e brancos (nós sem cor) respeitando as seguintes restrições:*

- *Todo nó $v$ tal que $v \subseteq S$ é preto.*
- *Todo nó $v$ tal que $v \not\perp S$ é cinza.*
- *Todo nó $v$ tal que $v \cap S = \emptyset$ ou $S \subset v$ é branco.*

**Definição 18** *O ancestral comum mais baixo de todos os nós pretos de uma árvore PQR marcada é denotado $LCA$.*

No próximo lema apresentamos uma caracterização da marcação de uma árvore PQR com base na cor dos filhos dos nós.

**Lema 19** *Uma árvore PQR é marcada se e somente se as seguintes condições se verificam simultaneamente:*

- *Toda folha é preta ou branca.*
- *Se todos os filhos de um nó são brancos então ele é branco.*
- *Se todos os filhos de um nó são pretos então ele é preto, exceto se é ancestral de todas as folhas pretas, caso em que é branco.*
- *Caso contrário é cinza.*

A coleção $\overline{T}$, definida a seguir, é uma coleção importante nas provas dos teoremas deste e do próximo capítulos. Ela será usada para demonstrar que o nosso algoritmo para construir árvores PQR é correto.

**Definição 20** *Seja $T$ uma árvore PQR marcada pelo conjunto $S$. A coleção $\overline{T}$ é*

$$\overline{T} = \overline{Compl(T) \cup \{S\}}.$$

A partir deste ponto usaremos indistintamente as expressões *marcar* e *pintar* ou *marcado* e *pintado* para nos referirmos às ações e aos nós de uma árvore marcada.

## 3.2 Operações de Redução

Nesta seção apresentamos um conjunto de **operações de redução** que transformam as árvores PQR marcadas e mostramos que cada uma dessas operações, ao receber como entrada uma árvore PQR marcada $T'$, produz uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$. As operações são: *Preparar o LCA*, *Preparar nó P*, *Mover filhos para nó cinza*, *Mover filhos para o LCA*, *Transformar nó Q em R*, *Reverter o LCA* e *Reverter nó cinza.*

O algoritmo para construir árvores PQR do Capítulo 4 é formado por composições destas operações. As operações serão usadas para remover os nós cinzas de uma árvore PQR marcada, fundindo-os a outros, e para ajustar o LCA ao final do processamento.

Nas figuras que ilustram as operações vamos adotar as seguintes convenções: um nó que possa ser de mais de um tipo na operação será representado por um triângulo. Os nós poderão ser rotulados por letras. As cores preta, cinza e branca serão aplicadas diretamente aos nós. Um nó que possa ser, ao mesmo tempo, de mais de um tipo e de mais de uma cor na operação será representado apenas por uma letra.

### 3.2.1 Preparar o LCA

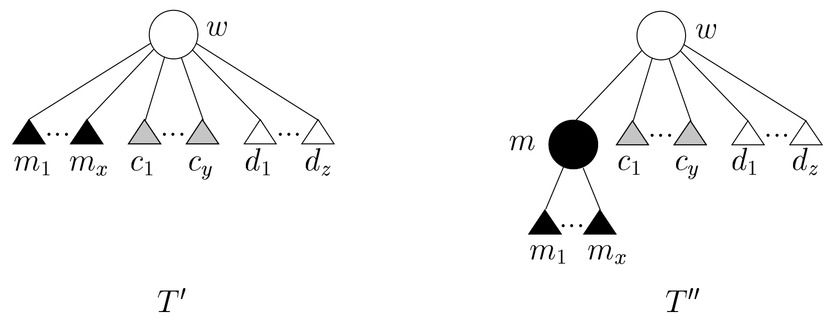
Esta operação é aplicada quando o LCA é um nó P. Ela agrupa os filhos pretos do LCA em um novo nó P.

**Entrada:** Uma árvore PQR marcada $T'$ e um nó $w$ de $T'$, tal que $w$ é o LCA e é do tipo P.

**Algoritmo:**

**Função** *Preparar_o_LCA*$(T', w)$
  Sejam $m_1, \ldots, m_x$ os filhos pretos de $w$.
  **Se** $x \geq 2$ e $x <$ *número de filhos de* $w$ **então**
    Crie um novo nó P pintado de preto e rotulado $m$.
    Torne $m_1, \ldots, m_x$ filhos de $m$.
    Torne $m$ filho de $w$.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.




**Lema 21** *Seja $T'$ uma árvore PQR marcada com um nó $w$ satisfazendo as restrições da entrada da operação "Preparar o LCA". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* Se $w$ tem no máximo um filho preto ou todos os filhos dele são pretos, então a operação não modifica $T'$. Caso contrário, $T'$ recebeu apenas um novo nó $m$ do tipo P com mais de dois filhos e $w$ ficou com pelo menos dois filhos. Então $T''$ é uma árvore PQR.

Para ver que a coloração de $T''$ é válida basta observar que:

- O nó $m$ é preto e o conjunto $m$ é igual a $m_1 \cup m_2 \cup \ldots \cup m_x$ e portanto $m \subseteq S$.

- O conjunto $w$ não foi modificado.

  Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

  $\overline{T'} = \overline{Compl(T') \cup \{S\}}$.

  Como $w$ foi o único nó afetado pela operação e $\mathcal{D}_{T'}(w) = \mathcal{D}_{T''}(w)$, por construção

  $\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(m)}$.

Então $\overline{T'} \subseteq \overline{T''}$. Para provar que $\overline{T''} \subseteq \overline{T'}$, como $\mathcal{D}_{T''}(m) = \{m\}$ basta mostrar que $m \in \overline{T'}$. Se $w$ não tem filhos cinzas, então

$$m = S.$$

Caso contrário, sejam $c_1, \ldots, c_y$ os filhos cinzas de $w$. Então podemos construir

$$m = S \mathbin{\backslash\!\!\!\circ} c_1 \mathbin{\backslash\!\!\!\circ} \ldots \mathbin{\backslash\!\!\!\circ} c_y.$$

$\square$

### 3.2.2 Preparar nó P

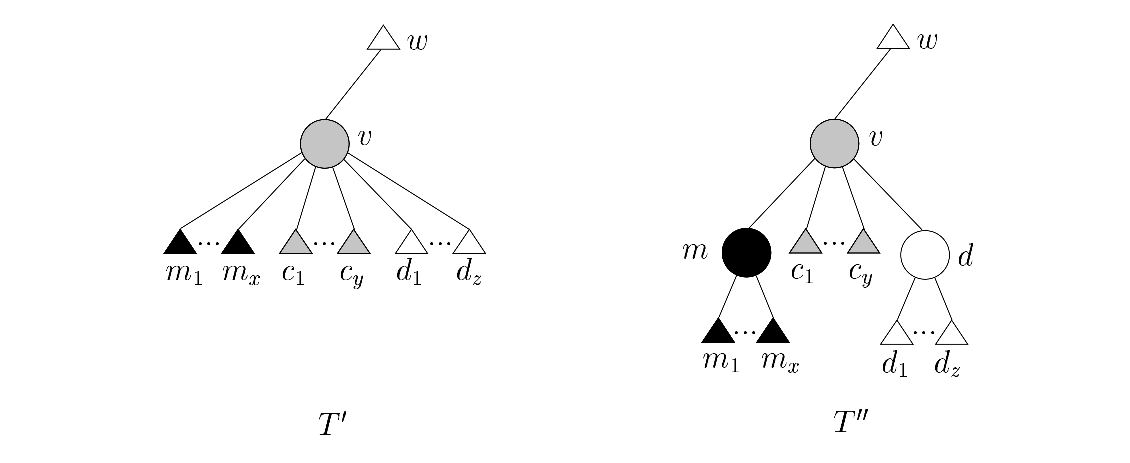
Esta operação é aplicada a um nó P cinza filho do LCA. Ela agrupa os filhos pretos e brancos desse nó P, criando novos nós P.

**Entrada:** Uma árvore PQR marcada $T'$ e um nó $v$ de $T'$, tal que $v$ é cinza, do tipo P e filho do LCA.

**Algoritmo:**

```
Função Preparar_nó_P(T', v)
    Sejam m_1, ..., m_x os filhos pretos de v.
    Sejam d_1, ..., d_z os filhos brancos de v.
    Se x ≥ 2 então
        Crie um novo nó P pintado de preto e rotulado m.
        Torne m_1, ..., m_x filhos de m.
        Torne m filho de v.
    Se z ≥ 2 então
        Crie um novo nó P sem cor e rotulado d.
        Torne d_1, ..., d_z filhos de d.
        Torne d filho de v.
```

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.




**Lema 22** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Preparar nó P". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* Se $v$ tem no máximo um filho preto e no máximo um filho branco então a operação não modifica $T'$. Senão $T'$ recebeu apenas novos nós do tipo P com mais de dois filhos. O nó $v$ ficou com pelo menos dois filhos porque, sendo cinza, nem todos os seus filhos eram pretos e nem todos os seus filhos eram brancos. Então $T''$ é uma árvore PQR.

Para ver que a coloração de $T''$ é válida basta observar que:

- O nó $m$ é preto e o conjunto $m$ é igual a $m_1 \cup m_2 \cup \ldots \cup m_x$ e portanto $m \subseteq S$.
- O nó $d$ é branco e o conjunto $d$ é igual a $d_1 \cup d_2 \cup \ldots \cup d_z$ e portanto $d \cap S = \emptyset$.
- Ao final da operação $v$ continua não ortogonal a $S$.

Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

$$\overline{T'} = \overline{Compl(T') \cup \{S\}}$$

Como $v$ foi o único nó afetado pelo algoritmo e como $\mathcal{D}_{T'}(v) = \mathcal{D}_{T''}(v)$, por construção

$$\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(m) \cup \mathcal{D}_{T''}(d)},$$

Então $\overline{T'} \subseteq \overline{T''}$. Para provar que $\overline{T''} \subseteq \overline{T'}$, como $\mathcal{D}_{T''}(m) = \{m\}$ e $\mathcal{D}_{T''}(d) = \{d\}$, basta mostrar que $m \in \overline{T'}$ e $d \in \overline{T'}$. Se $v$ não tem filhos cinzas podemos construir

$$m = v \cap S,$$
$$d = v \between S.$$

Caso contrário, sejam $c_1, \ldots, c_y$ os filhos cinzas de $v$. Então podemos construir

$$
\begin{aligned}
m &= (v \cap S) \mathbin{\circ\!\!\!\backslash} c_1 \mathbin{\circ\!\!\!\backslash} \ldots \mathbin{\circ\!\!\!\backslash} c_y, \\
d &= v \mathbin{\circ\!\!\!\backslash} S \mathbin{\circ\!\!\!\backslash} c_1 \mathbin{\circ\!\!\!\backslash} \ldots \mathbin{\circ\!\!\!\backslash} c_y.
\end{aligned}
$$

□

### 3.2.3 Transformar nó P em Q

Esta operação transforma um nó P cinza filho do LCA em um nó Q. Depois da aplicação desta operação, pode ser que um nó Q fique com dois filhos, contrariando a definição de árvore PQR. Poderíamos contornar este problema com um número maior de operações e subcasos nas demonstrações das propriedades das operações que já existem. No entanto, optamos por modificar ligeiramente a definição de árvore PQR e aceitar nós Q com dois filhos, primeiro porque as transformações de equivalência e o conjunto $Desc_T$ para um nó P com dois filhos e para um nó Q com dois filhos são iguais e depois porque esta operação será usada em composições que farão com que nós Q com dois filhos sejam fundidos a outros.

Nas operações a seguir, vamos considerar então que as árvores PQR obedecem às seguintes restrições à sua estrutura interna:

- As folhas estão em bijeção com os elementos de um conjunto $U$.
- Cada nó P tem pelo menos dois filhos.
- Cada nó Q tem pelo menos dois filhos.
- Cada nó R tem pelo menos três filhos.

e que as transformações de equivalência que podem ser aplicadas a uma árvore PQR são:

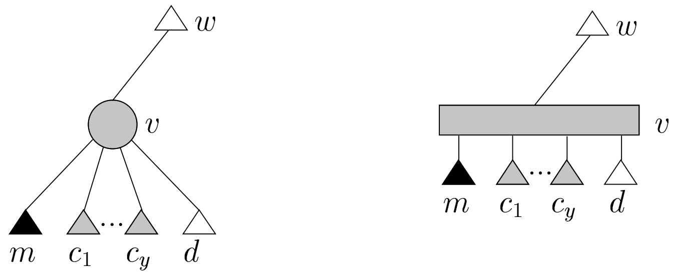
- Permutações arbitrárias dos filhos de um nó P.
- Reversão dos filhos de um nó Q.
- Permutações arbitrárias dos filhos de um nó R.
- Troca do tipo de um nó com dois filhos.

**Entrada:** Uma árvore PQR marcada $T'$ e um nó $v$ de $T'$, tal que $v$ é um filho cinza do LCA, do tipo P, com no máximo um filho preto e no máximo um filho branco.

**Algoritmo:**

**Função** *Transformar_nó_P_em_Q*$(T', v)$
Seja $m$ o filho preto de $v$, se houver.
Seja $d$ o filho branco de $v$, se houver.
Sejam $c_1, \ldots, c_y$ os filhos cinzas de $v$, se houver.
Transforme $v$ em um nó Q com os filhos na seguinte ordem:
$m, c_1, \ldots, c_y, d$

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.




$T'$ $T''$

**Lema 23** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Transformar nó P em Q". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* É fácil ver que $T''$ é uma árvore PQR com coloração válida.

Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

$$\overline{T'} = \overline{Compl(T') \cup \{S\}}.$$

Como $v$ foi o único nó afetado pela operação e como $\mathcal{D}_{T'}(v) \subseteq \mathcal{D}_{T''}(v)$, por construção

$$\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(v)}.$$

Então $\overline{T'} \subseteq \overline{T''}$. Para provar que $\overline{T''} \subseteq \overline{T'}$ é suficiente mostrar que todo par de filhos consecutivos de $v$ está em $\overline{T'}$. Podemos construir

$$m \cup c_1 = ((v \cap S) \between c_2 \between \ldots \between c_y) \uplus c_1,$$
$$m \cup c_1 \cup c_2 = ((v \cap S) \between c_3 \between \ldots \between c_y) \uplus c_1 \uplus c_2,$$
$$\ldots$$
$$m \cup c_1 \cup \ldots \cup c_y = (v \cap S) \uplus c_1 \uplus \ldots \uplus c_y,$$

$$
\begin{aligned}
&c_y \cup d = ((v \between S) \between c_1 \between \ \ldots \ \between c_{y-1}) \uplus c_y,\\
&c_{y-1} \cup c_y \cup d = ((v \between S) \between c_1 \between \ \ldots \ \between c_{y-2}) \uplus c_{y-1} \uplus c_y,\\
&\ldots\\
&c_1 \cup \ldots \cup c_y \cup d = (v \between S) \uplus c_1 \uplus \ldots \uplus c_y.
\end{aligned}
$$

Por fim, se $y \geq 2$, podemos construir

$$c_i \cup c_{i+1} = (m \cup c_1 \cup \ldots \cup c_i \cup c_{i+1}) \cap (c_i \cup c_{i+1} \cup \ldots \cup c_y \cup d),$$

para todo $1 \leq i \leq y-1$, o que conclui a prova. □

### 3.2.4 Mover filhos para nó cinza

Nesta operação um filho cinza do LCA passa a ser o pai dos seus irmãos pretos e cinzas. Antes de apresentar a operação, definimos ordens especiais para os filhos de nós Q que são usados nesta e na próxima operações.

**Definição 24** *Um nó $w$ do tipo Q e com filhos rotulados $v_1 \ldots v_t$ da esquerda para a direita, está em* **ordem adequada** *se*

- *$v_1$ é preto ou*
- *$v_1$ é cinza e $v_t$ é branco ou cinza ou*
- *$v_1$ é branco e $v_t$ é branco.*

*Um nó $w$ do tipo Q com filhos rotulados $v_1 \ldots v_t$ da esquerda para a direita, está em* **ordem adequada em relação ao seu filho $v_i$** *se*

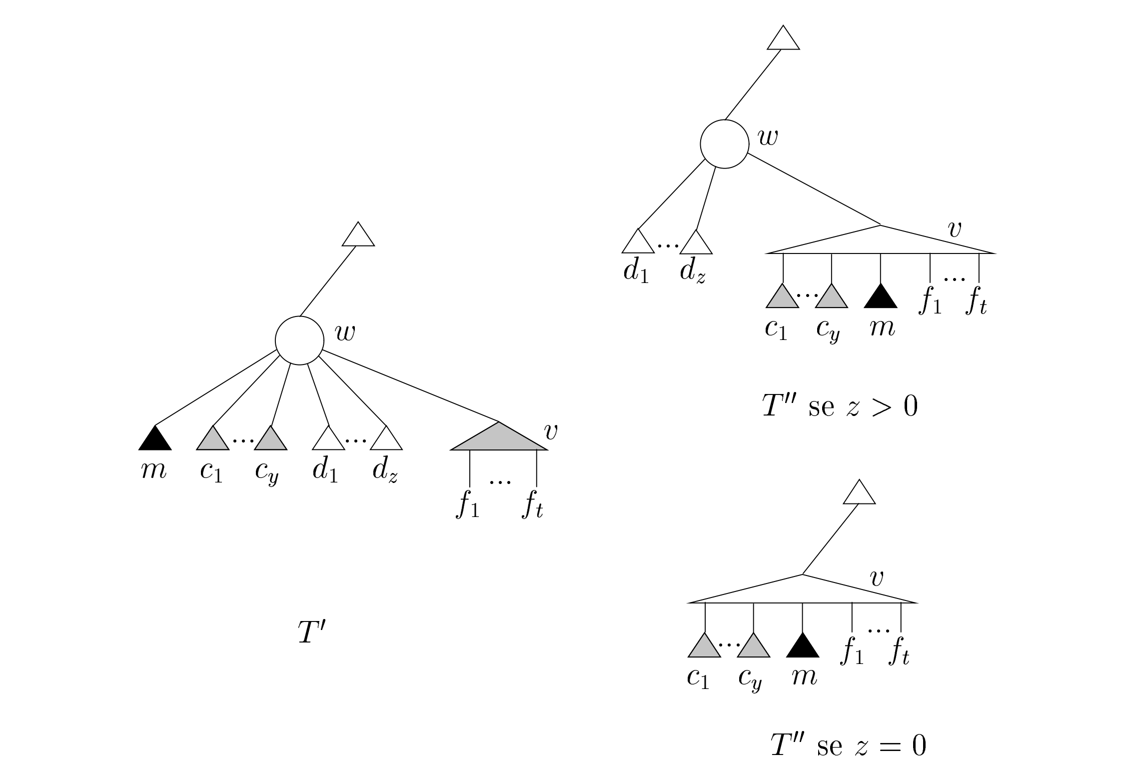
- *$i = 1$ e $v_2$ é branco ou*
- *$1 < i < t$ e $v_{i-1}$ é preto ou*
- *$1 < i < t$ e $v_{i-1}$ é cinza e $v_{i+1}$ é branco ou cinza ou*
- *$1 < i < t$ e $v_{i-1}$ é branco e $v_{i+1}$ é branco ou*
- *$i = t$ e $v_{i-1}$ é preto ou cinza.*

**Entrada:** Uma árvore PQR marcada $T'$ e nós $w$ e $v$ de $T'$, tais que $w$ é o LCA do tipo P com no máximo um filho preto e $v$ é um filho cinza de $w$ que é do tipo Q em ordem adequada ou do tipo R.

**Algoritmo:**

**Função** *Mover_filhos_para_nó_cinza*$(T', w, v)$
Seja $m$ o filho preto de $w$, se houver.
Sejam $c_1, \ldots, c_y$ os filhos cinzas de $w$, se houver, excluído $v$.
Sejam $f_1, \ldots, f_t$ os filhos de $v$ rotulados da esquerda para a direita.
Torne os filhos preto e cinzas de $w$ filhos de $v$, de tal forma que os filhos de $v$ fiquem na seguinte ordem:
$c_1, \ldots, c_y, m, f_1, \ldots, f_t$
Pinte $v$ de branco.
**Se** $w$ tem apenas um filho **então**
**Se** $w$ é a raiz de $T'$ **então**
Torne $v$ a raiz de $T'$.
**Senão**
Substitua $w$ por $v$.
Remova $w$ de $T'$.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.



**Lema 25** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Mover filhos para nó cinza". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* É fácil perceber que $T''$ é uma árvore PQR, já que $w$ é removido quando fica com apenas um filho. Para ver que a coloração de $T''$ continua válida basta observar que

1. O nó $v$ passa a ser o LCA, já que $S \subset v$, e então é pintado de branco.

2. Se $w$ não é removido de $T'$, continua verdade que $S \subset w$, e $w$ continua branco.

3. O pai de $w$, se existir, é branco em $T'$ porque contém $S$. A remoção eventual de $w$ não muda esta situação.

   Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

   $\overline{T'} = \overline{Compl(T') \cup \{S\}}$.

   Os nós $w$ e $v$ foram os únicos nós afetados pela operação. Além disto, se $w$ não é removido de $T'$, como $w$ é do tipo P, $\mathcal{D}_{T'}(w) = \mathcal{D}_{T''}(w)$. Caso contrário, $\mathcal{D}_{T'}(w) = \mathcal{D}_{T''}(v)$. Finalmente, como a ordem relativa dos filhos de $v$ não mudou e $v$ é um nó Q ou R, $\mathcal{D}_{T'}(v) \subseteq \mathcal{D}_{T''}(v)$. Logo, por construção

   $\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(v)}$.

Então $\overline{T'} \subseteq \overline{T''}$. Resta mostrar que $\overline{T''} \subseteq \overline{T'}$. Para tanto, é suficiente mostrar que

1. se $v$ é um nó Q e $m = \emptyset$ então os conjuntos

   $$\begin{array}{l} c_i \cup c_{i+1} \\ c_y \cup f_1 \end{array}$$

   estão em $\overline{T'}$ para $1 \leq i \leq y-1$. Caso contrário, se $m \neq \emptyset$, então os conjuntos

   $$\begin{array}{l} c_i \cup c_{i+1} \\ c_y \cup m \\ m \cup f_1 \end{array}$$

   estão em $\overline{T'}$ para $1 \leq i \leq y-1$ e

2. se $v$ é um nó R todo conjunto formado por um par de filhos de $v$ está em $T'$.

São esses os fatos que demonstraremos agora.

***v* é do tipo Q** Vamos supor primeiro que $v$ é um nó Q e demonstrar que $(c_i \cup c_{i+1}) \in \overline{T'}$, $1 \leq i \leq y-1$. Se $y \geq 2$, podemos construir

$$\begin{aligned}
&c_i \cup c_{i+1} \cup m = (S \obslash v \obslash c_1 \obslash \ldots \obslash c_{i-1} \obslash c_{i+2} \obslash \ldots \obslash c_y) \uplus c_i \uplus c_{i+1}, \\
&c_i \cup c_{i+1} = ((c_i \cup c_{i+1} \cup m) \obslash S) \uplus c_i \uplus c_{i+1}.
\end{aligned}$$

Agora vamos mostrar que se $m = \emptyset$ então $(c_y \cup f_1) \in \overline{T'}$. São três as possibilidades para $f_1$:

1. $f_1$ é preto: Como $f_1$ é preto e $v$ é cinza, $v$ tem algum filho que é cinza ou branco. Então o conjunto

   $$A = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$c_y \cup f_1 = (S \obslash c_1 \obslash \ldots \obslash c_{y-1} \obslash A) \uplus c_y.$$

2. $f_1$ é cinza: Como $v$ está em ordem adequada, $f_t$ é cinza ou branco. Então o conjunto

   $$A = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$c_y \cup f_1 = (S \obslash c_1 \obslash \ldots \obslash c_{y-1} \obslash A) \uplus c_y \uplus f_1.$$

3. $f_1$ é branco: Como $v$ está em ordem adequada, $f_t$ é branco. Seja $f_i$ um filho cinza ou preto de $v$. Os conjuntos

   $$\begin{aligned}
   A &= f_1 \cup \ldots \cup f_i \\
   B &= f_2 \cup \ldots \cup f_t
   \end{aligned}$$

   são tais que $B \not\subseteq (A \cup S)$ e então podemos construir

   $$c_y \cup f_1 = (((S \obslash c_1 \obslash \ldots \obslash c_{y-1}) \uplus A) \obslash B) \uplus c_y.$$

   Se $m \neq \emptyset$, então podemos construir

   $$c_y \cup m = (S \obslash c_1 \obslash \ldots \obslash c_{y-1} \obslash v) \uplus c_y.$$

Ainda resta mostrar que se $m \neq \emptyset$ então $(m \cup f_1) \in \overline{T'}$. São três as possibilidades para $f_1$:

1. $f_1$ é preto: Como $v$ é cinza, ele tem pelo menos um filho que é branco ou cinza. Então o conjunto

   $$A = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$m \cup f_1 = S \between c_1 \between \ldots \between c_y \between A.$$

2. $f_1$ é cinza: Como $v$ está em ordem adequada, $f_t$ é cinza ou branco. Sabemos que

   $$A = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$m \cup f_1 = (S \between c_1 \between \ldots \between c_y \between A) \uplus f_1.$$

3. $f_1$ é branco: Como $v$ está em ordem adequada, $f_t$ é branco. Seja $f_i$ um filho cinza ou preto de $v$. Então os conjuntos

   $$A = f_1 \cup \ldots \cup f_i,$$
   $$B = f_2 \cup \ldots \cup f_t,$$

   são tais que $B \not\subseteq (A \cup S)$ e então podemos construir,

   $$m \cup f_1 = ((S \between c_1 \between \ldots \between c_y) \uplus A) \between B.$$

***v* é do tipo R** Agora vamos supor que $v$ é um nó R e mostrar que todo conjunto formado por um par de seus filhos está em $\overline{T'}$. Por argumentos similares aos usados quando $v$ é um nó Q (permutando os filhos de $v$ se necessário) podemos mostrar que os conjuntos

$$c_k \cup c_{k+1},$$
$$c_y \cup f_1$$

para $1 \leq k \leq y - 1$ estão em $\overline{T'}$ se $m = \emptyset$ e os conjuntos

$$c_k \cup c_{k+1},$$
$$c_y \cup m,$$
$$m \cup f_1$$

para $1 \leq k \leq y-1$ estão em $\overline{T'}$ se $m \neq \emptyset$.

Por hipótese os conjuntos

$$f_i \cup f_j$$

estão em $\overline{T'}$ para todo $1 \leq i \leq t$ e $1 \leq j \leq t$.

Se $m = \emptyset$ então podemos construir

$$\begin{aligned}
&c_y \cup f_1 \cup f_i = (c_y \cup f_1) \uplus (f_1 \cup f_i),\\
&c_y \cup f_i = (c_y \cup f_1 \cup f_i) \obslash (f_1 \cup f_j)
\end{aligned}$$

para todo $2 \leq i \leq t$ onde $j$ é qualquer índice diferente de 1 e $i$. Sabemos que $j$ existe pois um nó R tem pelo menos três filhos. Caso contrário, se $m \neq \emptyset$, podemos construir

$$\begin{aligned}
&m \cup f_1 \cup f_i = (m \cup f_1) \uplus (f_1 \cup f_i),\\
&m \cup f_i = (m \cup f_1 \cup f_i) \obslash (f_1 \cup f_j),
\end{aligned}$$

$$\begin{aligned}
&c_y \cup m \cup f_i = (m \cup f_i) \uplus (c_y \cup m),\\
&c_y \cup f_i = (c_y \cup m \cup f_i) \obslash (m \cup f_j)
\end{aligned}$$

para todo $2 \leq i \leq t$ onde, novamente, $j \neq 1, i$. Podemos construir também

$$\begin{aligned}
&c_{y-1} \cup c_y \cup f_i = (c_y \cup f_i) \uplus (c_{y-1} \cup c_y)\\
&c_{y-1} \cup f_i = (c_{y-1} \cup c_y \cup f_i) \obslash (c_y \cup f_j)
\end{aligned}$$

$$\begin{aligned}
&c_{y-2} \cup c_{y-1} \cup f_i = (c_{y-1} \cup f_i) \uplus (c_{y-1} \cup c_{y-2})\\
&c_{y-2} \cup f_i = (c_{y-2} \cup c_{y-1} \cup f_i) \obslash (c_{y-1} \cup f_j)
\end{aligned}$$

$$\ldots$$

$$\begin{aligned}
&c_1 \cup c_2 \cup f_i = (c_2 \cup f_i) \uplus (c_1 \cup c_2)\\
&c_1 \cup f_i = (c_1 \cup c_2 \cup f_i) \obslash (c_2 \cup f_j)
\end{aligned}$$

Finalmente, se $m \neq \emptyset$ podemos construir

$$\begin{aligned}
&c_k \cup m \cup f_i = (c_k \cup f_i) \uplus (m \cup f_i),\\
&c_k \cup m = (c_k \cup m \cup f_i) \obslash (f_i \cup f_j)
\end{aligned}$$

para todo $1 \leq k \leq y$, o que conclui a prova. $\square$

### 3.2.5 Mover filhos para o LCA

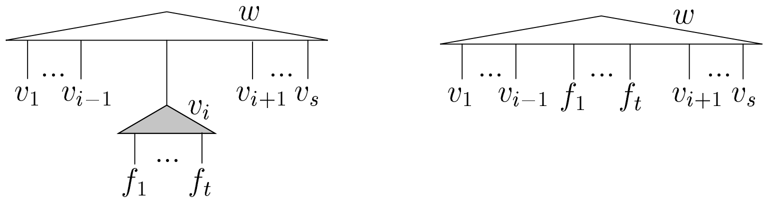
Nesta operação um filho cinza do LCA perde seus filhos, que são transferidos para o LCA, e depois é removido da árvore.

**Entrada:** Uma árvore PQR marcada $T'$ e nós $w$ e $v$ de $T'$, tais que $w$ é o LCA do tipo Q em ordem adequada em relação a seu filho $v$ ou do tipo R e o nó $v$ é do tipo Q em ordem adequada ou do tipo R.

**Algoritmo:**

**Função** *Mover_filhos_para_o_LCA*$(T', w, v)$
Sejam $v_1, \ldots, v_s$ os filhos de $w$ rotulados da esquerda para a direita.
Sejam $f_1, \ldots, f_t$ os filhos de $v_i$ rotulados da esquerda para a direita.
Torne $f_1, \ldots, f_t$ filhos de $w$, de tal forma que $v_{i-1}$ e $f_1$ sejam vizinhos e que a ordem relativa dos filhos de $v_i$ seja mantida.
Remova $v_i$ de $T'$.
**Se** $w$ é um nó Q e $v_i$ é um nó R **então**
Transforme $w$ em um nó R.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.




$T'$ $T''$

**Lema 26** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v_i$ satisfazendo as restrições da entrada da operação "Mover filhos para o LCA". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* É fácil ver que $T''$ é uma árvore PQR e que a coloração de $T''$ é válida.

Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

$$\overline{T'} = \overline{Compl(T') \cup \{S\}}.$$

Como $w$ e $v_i$ foram os únicos nós afetados pela operação e $w$ é nó Q ou R, então $\mathcal{D}_{T'}(v_i) \subseteq \mathcal{D}_{T''}(w)$ e $\mathcal{D}_{T'}(w) \subseteq \mathcal{D}_{T''}(w)$. Logo, por construção

$$\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(w)},$$

Então $\overline{T'} \subseteq \overline{T''}$. Para mostrar que $\overline{T''} \subseteq \overline{T'}$ vamos considerar os tipos de $w$ e $v_i$.

**$w$ e $v_i$ são do tipo Q** Para mostrar que $\mathcal{D}_{T''}(w) \subseteq \overline{T'}$ devemos mostrar que todos os pares de filhos consecutivos de $w$ em $T''$ estão em $\overline{T'}$.

Vamos começar supondo que $2 \leq i \leq s-1$. Nesse caso os conjuntos

$$v_{i-1} \cup v_i$$
$$v_i \cup v_{i+1}$$
$$f_j \cup \ldots \cup f_k$$

estão em $\overline{T'}$ para $1 \leq j \leq k \leq t$. Vamos supor que $v_{i-1}$ é preto ou cinza, considerar as possibilidades para $f_1$ e mostrar que $(v_{i-1} \cup f_1) \in \overline{T'}$:

1. $v_{i-1}$ é preto ou cinza e $f_1$ é preto: Como $v_i$ é cinza, $f_1$ tem pelo menos um irmão branco ou cinza. Então o conjunto

   $$B = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$v_{i-1} \cup f_1 = (((v_{i-1} \cup v_i) \cap S) \between B) \uplus v_{i-1}.$$

2. $v_{i-1}$ é preto ou cinza e $f_1$ é cinza: Como $v_i$ está em ordem adequada, $f_t$ é cinza ou branco. Então o conjunto

   $$B = f_2 \cup \ldots \cup f_t$$

   não está contido em $S$ e podemos construir

   $$v_{i-1} \cup f_1 = (((v_{i-1} \cup v_i) \cap S) \between B) \uplus v_{i-1} \uplus f_1.$$

3. $v_{i-1}$ é preto ou cinza e $f_1$ é branco: Como $v_i$ é cinza e está em ordem adequada, ele tem um filho preto ou cinza de índice $j < t$. Então os conjuntos

$$B = f_1 \cup \ldots \cup f_j,$$
$$C = f_2 \cup \ldots \cup f_t$$

são tais que $C \not\subseteq B \cup S$ e podemos construir

$$v_{i-1} \cup f_1 = ((((v_{i-1} \cup v_i) \cap S) \uplus B) \not\between C) \uplus v_{i-1}.$$

Agora vamos supor que $v_{i-1}$ é branco. Neste caso, por conta da ordem adequada, $v_{i+1}$ também é branco. Vamos considerar as possibilidades para $f_t$ e mostrar que $(f_t \cup v_{i+1}) \in \overline{T'}$:

1. $v_{i+1}$ é branco e $f_t$ é preto: Como $v_i$ é um nó Q em ordem adequada, $f_1$ também é preto. Como $v_i$ é cinza, ele tem um filho $f_j$ que é cinza ou branco. Então os conjuntos

$$B = f_j \cup \ldots \cup f_t,$$
$$C = f_1 \cup \ldots \cup f_{t-1}$$

são tais que $C \not\subseteq (v_i \setminus S) \cup B$ e podemos construir

$$f_t \cup v_{i+1} = (((v_i \cup v_{i+1}) \not\between S) \uplus B) \not\between C.$$

2. $v_{i+1}$ é branco e $f_t$ é cinza: Como $v_i$ é um nó Q em ordem adequada, $f_1$ é preto ou cinza. Então podemos construir

$$f_t \cup v_{i+1} = (((v_i \cup v_{i+1}) \not\between S) \not\between (f_1 \cup \ldots \cup f_{t-1})) \uplus f_t.$$

3. $v_{i+1}$ é branco e $f_t$ é branco: Como $v_i$ é cinza, ele tem um filho $f_j$, com $j < t$, que é cinza ou preto. Então podemos construir

$$f_t \cup v_{i+1} = ((v_i \cup v_{i+1}) \not\between S) \not\between (f_1 \cup \ldots \cup f_{t-1})$$

Até este ponto mostramos que quando $v_{i-1}$ é preto ou cinza então $(v_{i-1} \cup f_1)$ pertence a $\overline{T'}$ e mostramos que quando $v_{i+1}$ é branco então $(f_t \cup v_{i+1})$ pertence a $\overline{T'}$. Vamos complementar estes casos agora. Se

$$\begin{aligned}&v_k \cup v_{k+1},\\&f_j \cup f_{j+1},\\&v_{i-1} \cup f_1,\end{aligned}$$

para $1 \leq k \leq s-1$ e $1 \leq j \leq t-1$ estão em $\overline{T'}$, então podemos construir

$$f_t \cup v_{i+1} = (v_i \cup v_{i+1}) \between ((v_{i-1} \cup f_1) \uplus (f_1 \cup f_2) \uplus \ldots \uplus (f_{t-2} \cup f_{t-1})).$$

E se

$$\begin{aligned}&v_k \cup v_{k+1},\\&f_j \cup f_{j+1},\\&f_t \cup v_{i+1},\end{aligned}$$

para $1 \leq k \leq s-1$ e $1 \leq j \leq t-1$ estão em $\overline{T'}$, então podemos construir

$$v_{i-1} \cup f_1 = (v_{i-1} \cup v_i) \between ((f_2 \cup f_3) \uplus \ldots \uplus (f_{t-1} \cup f_t) \uplus (f_t \cup v_{i+1})).$$

Se $i = 1$, como $w$ está em ordem adequada em relação a $v_i$, então $v_2$ é branco e podemos usar argumentos análogos aos usados quando $v_{i+1}$ é branco para mostrar que $(f_t \cup v_2) \in \overline{T'}$. E se $i = s$, $v_{s-1}$ é preto ou é cinza e podemos usar argumentos análogos aos usados quando $v_{i-1}$ é preto ou é cinza para mostrar que $(v_{s-1} \cup f_1) \in \overline{T'}$.

**$w$ é do tipo Q e $v_i$ é do tipo R** Neste caso, $w$ é um nó R em $T''$. Precisamos mostrar que $\mathcal{D}_{T''}(w)$ está contido em $\overline{T'}$, e para isso é suficiente mostrar que os todos os conjuntos formados por pares de filhos de $w$ estão em $\overline{T'}$.

Podemos reordenar os filhos de $v_i$ livremente sem alterar $\mathcal{D}_{T'}(v_i)$. Então se colocarmos os filhos de $v_i$ em uma ordem compatível com uma ordem adequada de um nó Q, podemos, da mesma forma que no caso anterior em que $v_i$ é do tipo Q, concluir que os conjuntos

$$\begin{aligned}&v_{i-1} \cup f_1\\&f_t \cup v_{i+1}\end{aligned}$$

estão em $\overline{T'}$ (ressalvando os casos $i = 1$ ou $i = s$).

Então, como todo par de filhos de $v_i$ pertence a $\overline{T'}$ e ele tem pelo menos três filhos, podemos construir

$$v_{i-1} \cup f_k = ((v_{i-1} \cup f_1) \uplus (f_1 \cup f_k)) \obslash (f_1 \cup f_l)$$

para todo $1 \leq k \leq t$ onde $l$ é um índice diferente de 1 e de $k$,

$$v_{i+1} \cup f_k = ((f_t \cup v_{i+1}) \uplus (f_t \cup f_k)) \obslash (f_t \cup f_l)$$

para todo $1 \leq k \leq t$ onde, novamente, $l \neq t, k$. Além disso,

$$\begin{aligned}
&v_{i-2} \cup f_k = ((v_{i-1} \cup f_k) \uplus (v_{i-1} \cup v_{i-2})) \obslash (v_{i-1} \cup f_l)\\
&v_{i-3} \cup f_k = ((v_{i-2} \cup f_k) \uplus (v_{i-2} \cup v_{i-3})) \obslash (v_{i-2} \cup f_l)\\
&\ldots\\
&v_1 \cup f_k = ((v_2 \cup f_k) \uplus (v_1 \cup v_2)) \obslash (v_2 \cup f_l)
\end{aligned}$$

$$\begin{aligned}
&v_{i+2} \cup f_k = ((v_{i+1} \cup f_k) \uplus (v_{i+1} \cup v_{i+2})) \obslash (v_{i+1} \cup f_l)\\
&v_{i+3} \cup f_k = ((v_{i+2} \cup f_k) \uplus (v_{i+2} \cup v_{i+3})) \obslash (v_{i+2} \cup f_l)\\
&\ldots\\
&v_s \cup f_k = ((v_{s-1} \cup f_k) \uplus (v_{s-1} \cup v_s)) \obslash (v_{s-1} \cup f_l)
\end{aligned}$$

para todo $1 \leq k \leq t$, onde $l$ é escolhido convenientemente,e

$$v_m \cup v_n = ((v_m \cup f_k) \uplus (v_n \cup f_k)) \obslash (f_k \cup f_l)$$

para todo $1 \leq m < n \leq s$, com $m, n \neq i$ onde $k$ e $l$ são escolhidos convenientemente.

***w* é do tipo R** Neste caso podemos usar o mesmo raciocínio usado no caso em que $w$ é um nó Q e $v_i$ é um nó R para mostrar que $\mathcal{D}_{T''}(w) \subseteq \overline{T'}$. □

### 3.2.6 Transformar nó Q em R

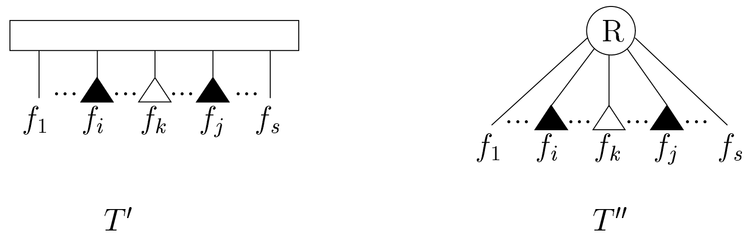
Esta operação transforma um nó Q que tenha filhos pretos separados por filhos brancos em um nó R.

| |
|---|
| **Entrada:** Uma árvore PQR marcada $T'$ e um nó $w$ de $T'$, tal que $w$ é o LCA do tipo Q, não tem filhos cinzas e dois dos seus filhos pretos têm pelo menos um irmão branco entre eles. |

**Algoritmo:**

**Função** *Transformar_nó_Q_em_R*$(T', w)$
Transforme $w$ em um nó R.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.




**Lema 27** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Transformar nó Q em R". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

*Prova:* É claro que $T''$ é uma árvore PQR e que a coloração de $T''$ é válida.

Vamos mostrar que $\overline{T'} = \overline{T''}$. Por definição

$$\overline{T'} = \overline{Compl(T') \cup \{S\}}.$$

Uma vez que $w$ foi o único nó afetado pela operação e $\mathcal{D}_{T'}(w) \subseteq \mathcal{D}_{T''}(w)$, por construção

$$\overline{T''} = \overline{Compl(T') \cup \{S\} \cup \mathcal{D}_{T''}(w)}.$$

Então $\overline{T'} \subseteq \overline{T''}$. Resta provar que $\overline{T''} \subseteq \overline{T'}$. Para tanto é suficiente mostrar que todos os pares de filhos de $w$ estão em $\overline{T'}$.

Sejam $f_i$ e $f_j$ os filhos pretos não-consecutivos de $w$ mais próximos. Vamos supor, sem perda de generalidade, que $i < j$. Se $f_i$ e $f_j$ são os únicos filhos pretos de $w$, então

$$f_i \cup f_j = S.$$

Senão sejam $d_1, \ldots, d_k$ os filhos brancos entre $f_i$ e $f_j$. O conjunto

$$B = f_i \cup d_1 \cup d_k \cup f_j,$$

pertence a $T'$ e então podemos construir

$$f_i \cup f_j = B \cap S.$$

Dado que $f_i \cup f_j$ pertence a $\overline{T'}$, por argumentos semelhantes aos usados no Lema 26 podemos concluir que todos os pares de filhos de $w$ estão em $\overline{T'}$. □

### 3.2.7 Reverter o LCA

Esta operação reverte o LCA, deixando-o em ordem adequada em relação a um de seus filhos cinzas.

**Entrada:** Uma árvore PQR marcada $T'$ e nós $w$ e $v$ de $T'$, tais que $w$ é o LCA do tipo Q e $v$ é um nó cinza filho de $w$.

**Algoritmo:**

**Função** *Reverter_o_LCA*$(T', w, v)$
  **Se** $w$ não está em ordem adequada em relação a $v$ **então**
    Reverta $w$.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.

O lema a seguir é imediato pois $T'$ e $T''$ são equivalentes.

**Lema 28** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Reverter o LCA". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

### 3.2.8 Reverter nó cinza

Esta operação reverte um nó Q cinza, colocando-o em ordem adequada.

**Entrada:** Uma árvore PQR marcada $T'$ e um nó $v$ de $T'$, tal que $v$ é do tipo Q e é filho do LCA.

**Algoritmo:**

**Função** *Reverter_nó_cinza*$(T', v)$
  **Se** $v$ não está em ordem adequada **então**
    Reverta $v$.

**Saída:** Uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.

O lema a seguir é imediato pois $T'$ e $T''$ são equivalentes.

**Lema 29** *Seja $T'$ uma árvore PQR marcada com nós $w$ e $v$ satisfazendo as restrições da entrada da operação "Reverter nó cinza". Aplicando a operação, obtemos uma árvore PQR marcada $T''$ tal que $\overline{T'} = \overline{T''}$.*

# Capítulo 4

# Algoritmo Quase-Linear para Construir Árvores PQR

Neste capítulo apresentamos um algoritmo quase-linear para construir árvores PQR, a prova de correção, a análise de complexidade, a descrição da nossa implementação e um breve histórico das nossas tentativas durante o desenvolvimento do algoritmo.

## 4.1 Algoritmo

Nosso algoritmo para construir árvores PQR é *on-line*. Depois de construir uma árvore P-total, ele processa um conjunto de $\mathcal{C} = \{S_1, \ldots, S_m\}$ de cada vez e modifica a árvore de tal forma que ao final do processamento do conjunto $S_i$, a árvore resultante é uma árvore PQR para $\{S_1, \ldots, S_i\}$.

O algoritmo PQR, na Figura 4.1, recebe como entrada um conjunto $U$ e uma coleção de conjuntos $\mathcal{C} \subseteq \mathcal{P}(U)$. O primeiro passo do algoritmo é a construção de uma árvore P-total $T$ tendo como folhas os elementos de $U$. Depois, a árvore $T$ é modificada por chamadas à função *Adicionar*, uma para cada conjunto em $\mathcal{C}$.

---

**Função** $PQR(U, \mathcal{C})$
  Construa uma árvore P-total $T$ tendo como folhas os elementos de $U$.
  **Para** todo conjunto $S \in \mathcal{C}$ **faça**
    $Adicionar(T, S)$.
  **Retorne** T.

---

Figura 4.1: Algoritmo para construir árvores PQR.

A função *Adicionar*, na Figura 4.2, recebe como entrada uma árvore PQR $T$ e um conjunto $S \subseteq U$ e faz o seguinte:

1. Marca a árvore de acordo com o conjunto $S$: a Função *Marcar*, na Figura 4.3, é uma adaptação do algoritmo de Booth e Lueker [6] para encontrar o LCA. Ela visita alguns nós da árvore duas vezes, começando pelas folhas em $S$. Na primeira visita ela conta, para cada nó interno $v$, quantas arestas incidentes a $v$ pertencem a algum caminho de $v$ a uma folha em $S$. Na segunda visita ela usa a informação armazenada na visita anterior para determinar quantas folhas em $S$ descendem de cada nó interno visitado. Além disso, o algoritmo marca os nós da árvore de acordo com a Definição 17 (Página 15). O algoritmo para marcar e encontrar o LCA pode ser bem mais simples que o que apresentamos aqui, mas para garantir a complexidade que desejamos (Seção 4.3) foi preciso lançar mão de uma solução mais sofisticada.

2. Elimina os nós cinzas da árvore: enquanto a árvore tiver nós cinzas, a Função *Reduzir*, na Figura 4.4, executa um conjunto de operações de redução de árvores PQR marcadas (Página 16 e seguintes) cujo efeito é eliminar um nó cinza filho do LCA. As operações executadas dependem do tipo do LCA e do filho cinza considerado.

3. Ajusta o tipo do LCA: a Função *Ajustar*, na Figura 4.5, ajusta o tipo do LCA depois que todos os nós cinzas foram removidos ou quando não houve nós cinzas.

4. Desmarca os nós: a Função *Desmarcar* na Figura 4.6 visita e desmarca os nós afetados pelos passos anteriores.

---

**Função** *Adicionar*$(T, S)$
    $w = Marcar(T, S)$.
    $Reduzir(T, w)$.
    $Ajustar(T, w)$.
    $Desmarcar(T, S)$.

---

Figura 4.2: Algoritmo para adicionar um conjunto $S$ a uma árvore PQR $T$.

Na Figura 4.7 aparece um exemplo da adição do conjunto $S = \{g, h, m\}$ em vários passos que ilustram as operações feitas pela função *Adicionar*.

```
Função Marcar(T, S)
    fila F = ∅.
    passou_da_raiz = 0.
    Para todo nó v em S faça
        v.visitado = 1.
        adicionar_ao_fim(F, v).
    Enquanto |F| + passou_da_raiz > 1 faça
        nó v = retirar_do_início(F).
        nó p = pai(v).
        Se p então
            p.caminhos_pertinentes = p.caminhos_pertinentes + 1.
            Se p.visitado == 0 então
                p.visitado = 1.
                adicionar_ao_fim(F, p).
        Senão
            passou_da_raiz = 1.

    nó w = null.
    F = ∅.
    Para todo nó v em S faça
        adicionar_ao_fim(F, v).
    Enquanto |F| > 0 faça
        nó v=retirar_do_início(F).
        nó p=pai(v).
        Se v é folha então
            p.caminhos_pertinentes = p.caminhos_pertinentes − 1.
            p.folhas_pertinentes = p.folhas_pertinentes + 1.
            v.cor = preto.
            Adicione v à lista de filhos pretos de p.
        Senão
            Se v.folhas_pertinentes == |S| então
                Retorne v.
            Senão
                p.caminhos_pertinentes = p.caminhos_pertinentes − 1.
                p.folhas_pertinentes = p.folhas_pertinentes + v.folhas_pertinentes.
            Se todos os filhos de v são pretos então
                v.cor = preto.
                Adicione v à lista de filhos pretos de p.
            Senão
                v.cor = cinza.
                Adicione v à lista de filhos cinzas de p.
        Se p.caminhos_pertinentes == 0 então
            adicionar_ao_fim(F, p).
```

Figura 4.3: Algoritmo para marcar uma árvore PQR e encontrar o LCA. A notação $v.x$ indica uma variável $x$ associada ao nó $v$.

```
Função Reduzir(T, w)
    Enquanto houver filho cinza v de w faça
        Escolha w:
        nó P:
            Escolha v:
            nó P: Preparar_nó_P(T, v).
                  Transformar_nó_P_em_Q(T, v).
            nó Q: Preparar_o_LCA(T, w).
                  Reverter_nó_cinza(T, v).
                  Mover_filhos_para_nó_cinza(T, w, v).
            nó R: Preparar_o_LCA(T, w).
                  Mover_filhos_para_nó_cinza(T, w, v).
            fim
        nó Q:
            Escolha v:
            nó P: Preparar_nó_P(T, v).
                  Transformar_nó_P_em_Q(T, v).
            nó Q: Reverter_o_LCA(T, w, v).
                  Reverter_nó_cinza(T, v).
                  Mover_filhos_para_o_LCA(T, w, v).
            nó R: Mover_filhos_para_o_LCA(T, w, v).
            fim
        nó R:
            Escolha v:
            nó P: Preparar_nó_P(T, v).
                  Transformar_nó_P_em_Q(T, v).
            nó Q: Mover_filhos_para_o_LCA(T, w, v).
            nó R: Mover_filhos_para_o_LCA(T, w, v).
            fim
        fim
```

Figura 4.4: Algoritmo para remover os nós cinzas de uma árvore PQR.

```
Função Ajustar(T, w)
    Se w é um nó P então
        Preparar_o_LCA(T, w).
    Se w é um nó Q e os filhos pretos de w não são consecutivos então
        Transformar_nó_Q_em_R(T, w).
```

Figura 4.5: Algoritmo para ajustar o LCA.

**Função** *Desmarcar*$(T, S)$
    fila $F = \emptyset$.
    Adicione os nós em $S$ a $F$.
    **Enquanto** $|F| > 0$ **faça**
        nó $v = retirar\_do\_início(F)$.
        nó $p = pai(v)$.
        **Se** $v.folhas\_pertinentes > 0$ **então**
            $v.caminhos\_pertinentes = 0$.
            $v.folhas\_pertinentes = 0$.
            $v.cor =$ branco.
            Esvazie a lista de filhos pretos de $v$.
            Esvazie a lista de filhos cinzas de $v$.
        **Se** $p.folhas\_pertinentes > 0$ **então**
            $adicionar\_ao\_fim(F, p)$.

Figura 4.6: Algoritmo para desmarcar uma árvore PQR.

## 4.2 Correção

Agora vamos mostrar que a função *Adicionar* modifica uma árvore PQR para refletir a adição de um conjunto $S$ corretamente. Mais formalmente, vamos mostrar que quando a função *Adicionar* recebe como parâmetros uma árvore $T'$ e um conjunto $S$, a árvore resultante $T''$ é tal que

$$\overline{Compl(T') \cup \{S\}} = Compl(T'').$$

**Teorema 30** *A árvore PQR $T''$ resultante da execução de Adicionar$(T', S)$ é tal que*

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

*Prova:* A função *Adicionar* executa quatro funções. A função *Marcar* não altera a estrutura da árvore. As funções *Adicionar* e *Ajustar*, que removem os nós cinzas e ajustam o LCA, são composições das operações de redução de árvores PQR marcadas que, como já foi demonstrado pelos Lemas 21, 22, 23, 25, 26, 27, 28 e 29, mantém inalterada a coleção $\overline{T'}$. A função *Desmarcar* também não altera a estrutura da árvore. $\square$

**Teorema 31** *A árvore PQR $T''$ resultante da execução de Adicionar$(T', S)$ é tal que*

$$S \in Compl(T'').$$

*Prova:* Vamos supor primeiro que após a marcação, $T'$ não tem nós cinzas e verificar quais transformações a função *Ajustar* provoca na árvore. Seja $w$ o LCA. Se $w$ é um nó preto então *Ajustar* não faz nada. Se $w$ é um nó branco, vamos considerar o seu tipo:

- Se $w$ é do tipo P, então os seus filhos pretos são agrupados em um novo nó P.
- Se $w$ é um nó Q e todos os seus filhos pretos são consecutivos, então a função *Ajustar* não faz nada. Caso contrário $w$ é transformado em um nó R.
- Se $w$ é um nó R então a função *Ajustar* não faz nada.

Em todos os casos $S \in Compl(T'')$ após as transformações.

Agora vamos supor que $T'$ tem nós cinzas após a marcação. Então por construção vemos que:

- Se $w$ é um nó P na primeira iteração da função *Reduzir*, então seu filho preto e seus filhos cinzas são transferidos para um filho $v$ de $w$, sendo que $v$ é cinza e do tipo Q ou R. O nó $v$ passa a ser o novo LCA e permanece nessa condição até o fim do algoritmo.
- Se $w$ é um nó Q ou R na primeira iteração da função *Reduzir*, então ele será o LCA até o fim do algoritmo.
- Exceto pelo caso em que $w$ é do tipo P na primeira iteração, em todas as iterações da repetição um nó cinza $v$ é removido da árvore e os filhos de $v$ são incorporados a $w$.

É fácil perceber que, ao final da função *Reduzir*, todo nó cinza filho de $w$ foi removido da árvore e $w$ é o pai de todos os nós pretos maximais que existiam na árvore inicial $T'$. Se $w$ é um nó Q e todos os seus filhos pretos são consecutivos, a função *Ajustar* não faz nada. Caso contrário, $w$ será transformado num nó R. Se $w$ é um nó R então *Ajustar* não faz nada. Em todos os casos $S \in Compl(T')$. $\square$

**Teorema 32** *A árvore PQR $T''$ resultante da execução de Adicionar$(T', S)$ é tal que*

$$\overline{Compl(T') \cup \{S\}} = Compl(T'').$$

*Prova:* Se $\overline{T'} = \overline{T''}$ então

$$\overline{Compl(T') \cup \{S\}} = \overline{Compl(T'') \cup \{S\}}.$$

Mas como $S \in Compl(T'')$ temos

$$\overline{Compl(T') \cup \{S\}} = \overline{T'} = \overline{T''} = \overline{Compl(T'') \cup \{S\}} = \overline{Compl(T'')} = Compl(T'')$$

$\square$

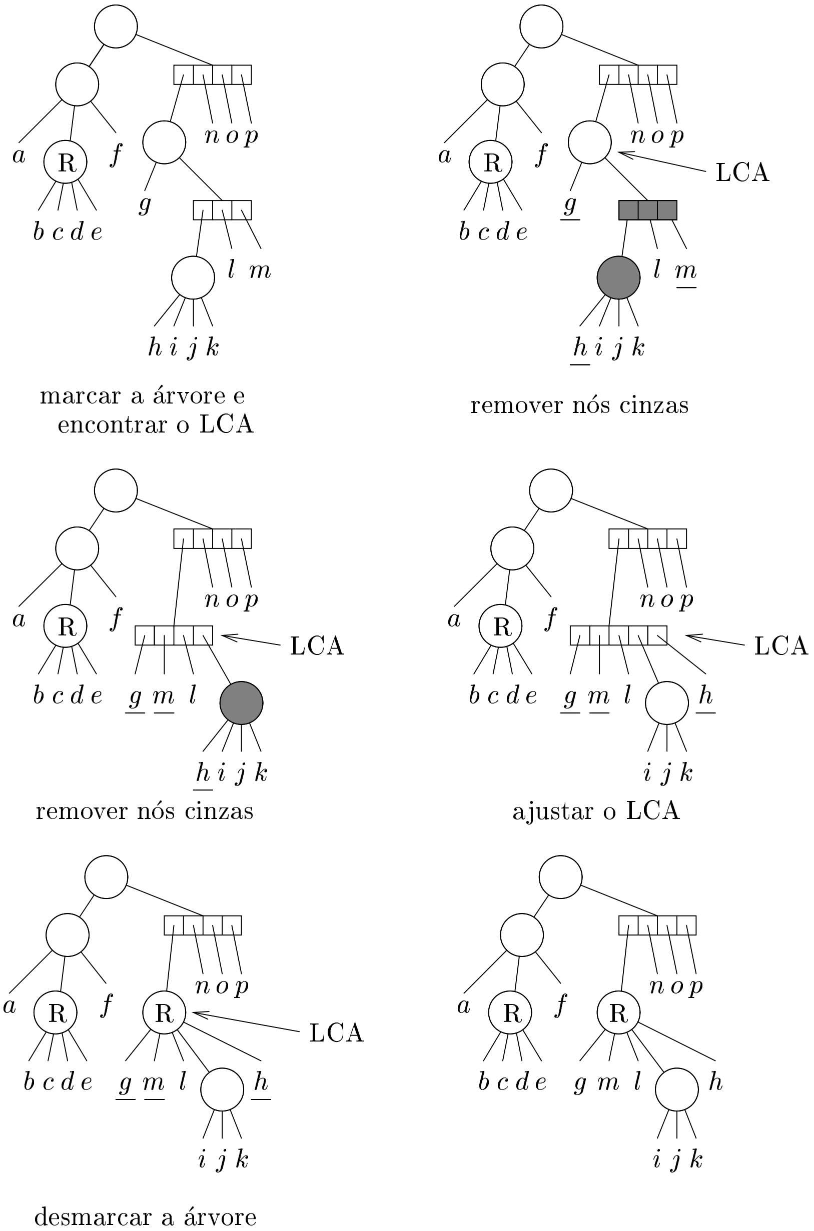



Figura 4.7: Adição do conjunto $S = \{g, h, m\}$ à arvore PQR que aparece no alto, à esquerda. Embaixo de cada árvore está a operação que foi aplicada a ela, transformando-a na próxima da seqüência.

## 4.3 Complexidade

Nesta seção vamos calcular a complexidade do nosso algoritmo através de uma análise amortizada pelo método da função potencial. Para simplificar as análises vamos considerar as quatro **operações de redução otimizadas** que aparecem na Figura 4.8. Essas operações são equivalentes aos casos P-P, P-Q/R, Q/R-P e Q/R-Q/R na função *Reduzir*. Observe que quando os filhos brancos de um nó $v$ do tipo P são isolados (casos P-P e Q/R-P), o nó que passa a ser o pai de tais filhos brancos é o próprio nó $v$, isto é, os nós pretos e cinzas são movimentados, mas os nós brancos não.

Durante a análise vamos considerar que a árvore PQR é representada da seguinte maneira:

1. A cada nó está associado um conjunto finito de variáveis. Essas variáveis armazenam informações que caracterizam o tipo e o estado de cada nó na árvore, e elas são criadas e inicializadas quando o nó é criado.

2. A cada nó está associado um apontador usado para encontrar o pai dele, de acordo com o seguinte regra:

    - Todos os filhos de nó P apontam diretamente para o seu pai.
    - Para nós Q e R uma estrutura de conjuntos disjuntos [9] é usada para implementar a relação "pai": um dos filhos de um nó Q ou R aponta para o seu pai e os demais apontam para um de seus irmãos.
    - Na raiz da árvore, esse apontador é nulo.

3. A cada nó está associada uma lista duplamente encadeada $L$ de apontadores para os seus filhos.

4. A cada nó estão associadas duas listas duplamente encadeadas: uma de apontadores para nós na lista $L$ que apontam para filhos pretos e outra de apontadores para nós em $L$ que apontam para filhos cinzas.

5. Cada nó $v$ mantém um apontador para o nó da lista $L$ no seu pai que aponta para ele mesmo ($v$).

Uma representação gráfica da estrutura aparece na Figura 4.9, onde o nó $w$ é pai dos nós $r, s, t, u$ e $v$ e filho do nó $x$. Na figura, o apontador do item 5 aparece rotulado como *índice*.

As listas em cada nó da árvore são duplamente encadeadas com duas cadeias de nós, como ilustrada na Figura 4.10. Uma posição da lista é composta por um nó na cadeia de

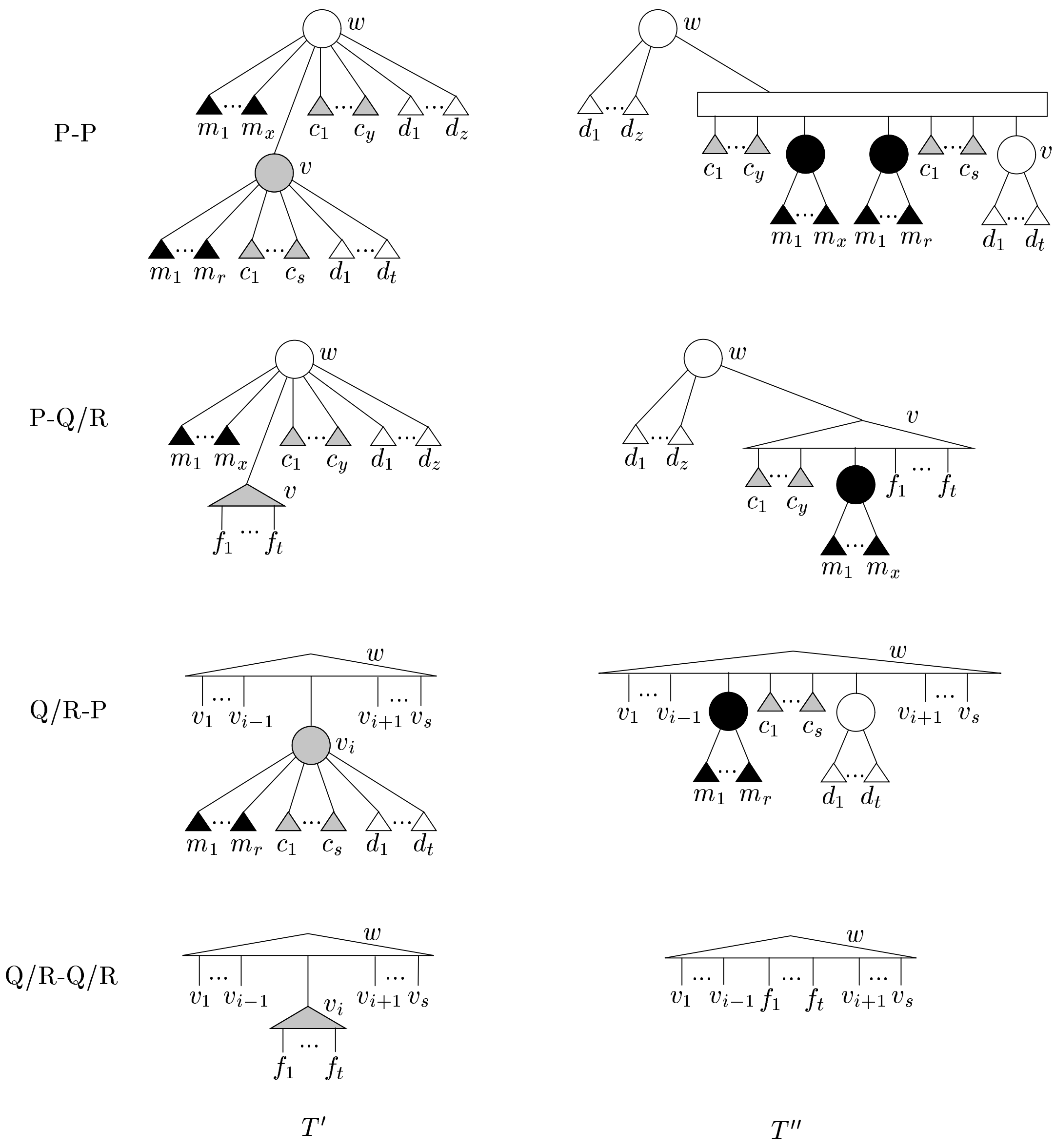



Figura 4.8: Operações de redução otimizadas. As árvores iniciais ($T'$) estão à esquerda e as árvores finais ($T''$) estão à direita.

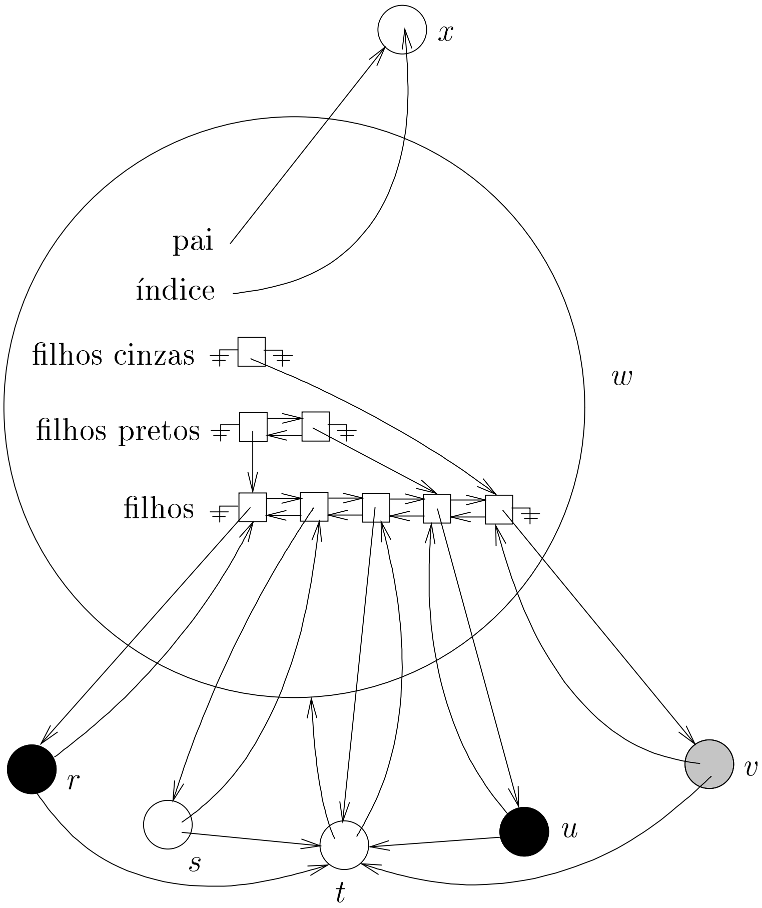



Figura 4.9: Representação das estruturas associadas a um nó interno da árvore PQR. O nó $w$ é filho do nó $x$ e pai dos nós $r, s, t, u$ e $v$, e está na mesma situação em relação a $x$ que o nó $t$ em relação a $w$, isto é, apontando diretamente para o seu pai.

cima e por um nó na cadeia de baixo. A cadeia de cima é a que é sempre referenciada, isto é, apontadores para posições da lista são referências a nós da cadeia de cima. A cadeia de cima é usada para avançar em direção ao fim e a cadeia de baixo é usada para avançar em direção ao início da lista. O dado armazenado em cada posição da lista pode estar no nó na cadeia de cima ou no nó da cadeia de baixo.

Usando estas estruturas somos capazes de:

- Substituir um dos filhos de um nó por outro nó em tempo constante.
- Reverter a ordem dos filhos de um nó em tempo constante.
- Unir as listas de filhos de dois nós em tempo constante.
- Implementar balanceamento e compressão de caminhos em conjuntos disjuntos [9]

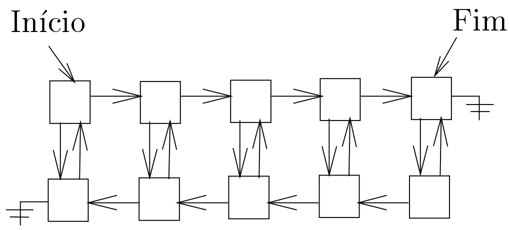



Figura 4.10: Lista duplamente encadeada com duas cadeias de nós.

(operações *Make-set, Union* e *Find*) para encontrar o pai de um nó e fundir dois nós em tempo médio $O(\alpha(|\mathcal{C}|, |U|))$, onde $\alpha$ é o inverso da função de Ackerman.

A partir deste ponto vamos calcular a complexidade do nosso algoritmo através dos seguintes passos:

1. Mostraremos que o número de operações realizadas pela função *Reduzir* é limitado assintoticamente pelo número de nós movimentados por ela (Teorema 33).

2. Depois mostraremos que o número total de nós movimentados pela função *Reduzir* é limitado assintoticamente pelo número de nós pretos e cinzas que descendem de nós em algum caminho de uma folha preta ao LCA (Teorema 34).

3. Em seguida vamos mostrar que o número de operações realizadas pela função $Adicionar(T', S)$ é limitado assintoticamente pelo número de nós movimentados por ela e pelo tamanho do conjunto $S$ (Teorema 35).

4. Então, através de uma função potencial adequada vamos mostrar que o número amortizado de nós movimentados pela operação $Adicionar(T', S)$ é limitado assintoticamente por $|S|$ (Definição 36 e Teoremas 37 e 38).

5. Finalmente, vamos calcular o custo total para construir uma árvore PQR (Teorema 39).

Antes das análises, vamos apresentar a notação que utilizamos. Vamos denotar o número de nós movimentados pela função *Reduzir* ao processar a árvore $T'$ durante a adição do conjunto $S$ como $mov(T', S)$. Para um nó $v$ de uma árvore PQR marcada, vamos denotar como $F_v$, $M_v$, $C_v$ e $D_v$, respectivamente, o conjunto de todos os filhos de $v$, o conjunto de filhos pretos de $v$, o conjunto de filhos cinzas de $v$ e o conjunto de filhos brancos de $v$.

**Teorema 33** *O número de operações realizado pela função Reduzir é $O(mov(T', S))$.*

*Prova:* As outras operações realizadas pela função são criação, destruição, reversão, marcação, desmarcação de nós e operações *Find* e *Union*.

- As folhas nunca são marcadas ou desmarcadas pela função.
- Nós internos são criados sem filhos. Então a criação de um nó sempre é seguida de movimentações para adicionar os filhos dele.
- Nós internos são destruídos apenas quando estão sem filhos. Então a destruição de um nó sempre é precedida de pelo menos uma movimentação para remover os filhos dele.
- Uma única reversão de um nó Q ou um par de reversões consecutivas sempre são seguidas de pelo menos uma movimentação.
- A marcação acontece quando um novo nó P recém-criado é pintado de preto.
- A desmarcação ocorre quando todos os filhos marcados de um nó P são movimentados, ou então quando um nó Q cinza é pintado de branco, caso precedido de pelo menos uma movimentação.
- A transformação de um nó $v$ do tipo P em Q obriga a um ajuste dos apontadores dos filhos, o que equivale a operações *Make-set*. Nessa situação o nó P tem no máximo um filho preto e no máximo um filho branco. Depois da transformação alguns nós podem ser movimentados para $v$ (PP) ou o nó $v$ pode ser unido ao seu pai (Q/R-P). O ajuste dos apontadores dos filhos cinzas $c_1, \ldots, c_y$ de $v$ serão sucedidos pelas movimentações dos filhos de $c_1, \ldots, c_y$ quando eles forem removidos.
- Toda operação *Find* é seguida de pelo menos uma movimentação.
- Uma operação *Union* precede a fusão de um nó Q ou R a um de seus filhos que também é do tipo Q ou R. Então toda operação *Union* é acompanhada de uma movimentação.

Então o número de operações realizado pela função *Reduzir* é $O(mov(T', S))$. □

**Teorema 34** *O número total de nós movimentados pela função Reduzir é tal que*

$$\begin{aligned} mov(T', S) \leq\ & |M_w| + |C_w| + 1 + \\ & \sum_{v \text{ nó P cinza}} (|M_v| + |C_v|) + \\ & \sum_{v \text{ nó Q/R cinza}} 1 \end{aligned}$$

*onde $w$ é o LCA.*

| Tipo do LCA | Tipo de $v$ | Movimentações |
|---|---|---|
| P | P | $\lvert M_w\rvert + \lvert C_w\rvert + \lvert M_v\rvert + \lvert C_v\rvert + 1$ |
| P | Q/R | $\lvert M_w\rvert + \lvert C_w\rvert + 1$ |
| Q/R | P | $\lvert M_v\rvert + \lvert C_v\rvert$ |
| Q/R | Q/R | 1 |

Tabela 4.1: Limites superiores no número de movimentações em cada combinação de tipo de LCA e nó cinza na função *Reduzir*.

*Prova:* Na Tabela 4.1 estão o número máximo de movimentações de nós para as quatro operações otimizadas. Ao final de cada operação, $v$ sempre deixa de existir ou é desmarcado. Então cada nó cinza é processado uma única vez. O LCA é do tipo P no máximo na primeira iteração e então entra no cálculo apenas uma vez. □

**Teorema 35** *O número total de operações realizado por Adicionar$(T', S)$ é igual a*

$$O(|S| + mov(T, S)).$$

*Prova:* Booth e Leuker [6] chamam de $prunned(T', S)$ a subárvore formada por todos os nós pretos, todos os nós cinzas e pelo LCA e mostram que para marcar a árvore e encontrar o LCA precisamos realizar $O(|prunned(T', S)|)$ operações. Como $|M_v| + |C_v| \geq 1$ para todo nó cinza, então $|prunned(T', S)| = O(mov(T', S))$. Para desmarcar a árvore são suficientes $O(|S|)$ visitas. Para reduzir os nós cinzas e ajustar o LCA são suficientes $O(|S|)$ operações. □

**Definição 36** *O* **potencial de uma árvore PQR** *$T$ é*

$$pot(T) = \sum_{v \text{ nó } P \text{ de } T} (|v| - 1) + \sum_{v \text{ nó } Q/R \text{ de } T} |F_v|.$$

**Teorema 37** *Dadas duas árvores PQR $T'$ e $T''$ tais que*

$$T'' = Adicionar(T', S),$$

*a diferença de potencial $\Delta pot(T', S) = pot(T'') - pot(T')$ satisfaz*

$$\begin{aligned} \Delta pot(T', S) \quad \leq \quad & |C_w| + 1 + \\ & \sum_{v \text{ nó } P \text{ cinza}} (|C_v| - \sum_{u \in C_v} |u|) - \\ & \sum_{v \text{ nó } Q/R \text{ cinza}} 1 \end{aligned}$$

*onde $w$ é o LCA.*

*Prova:* O LCA é do tipo P no máximo uma vez e nesse caso ele cede seus filhos cinzas e preto a um nó Q ou R. Então a variação de potencial causada pelo LCA do tipo P quando ele tem algum filho branco é

$$|C_w| + 1$$

ou, quando ele tem algum filho preto ou cinza, é

$$|C_w| + 1 - (\sum_{u \in F_w} |u| - 1).$$

Se o LCA é Q ou R ele não causa variação de potencial.

Todo nó P cinza $v$ cede seus filhos a um nó Q ou R depois de agrupar seus filhos pretos e brancos em nós P, então a variação de potencial causada por cada um deles é

$$|C_v| - \sum_{u \in C_v} |u|.$$

Todo nó Q ou R cinza $v$ cede seus filhos a um nó Q ou R e então a variação de potencial causada por cada um deles é

$$-1. \qquad \square$$

**Teorema 38** *Dada uma árvore PQR $T'$ e um conjunto $S \subseteq U$,*

$$mov(T', S) + \Delta pot(T', S) \leq 3|S| + 2.$$

*Prova:* Dos teoremas anteriores,

$$\begin{aligned}
mov(T,S) + \Delta pot(T,S) \leq\ & |M_w| + |C_w| + 1 + \\
& \sum_{v \text{ nó } P \text{ cinza}} (|M_v| + |C_v|) + \\
& \sum_{v \text{ nó } Q/R \text{ cinza}} 1 + |C_w| + 1 + \\
& \sum_{v \text{ nó } P \text{ cinza}} (|C_v| - \sum_{u \in C_v} |u|) - \\
& \sum_{v \text{ nó } Q/R \text{ cinza}} 1 \\
\leq\ & |M_w| + 2|C_w| + 2 + \\
& \sum_{v \text{ nó } P \text{ cinza}} (|M_v| + 2|C_v| - \sum_{u \in C_v} |u|) \\
\leq\ & |M_w| + 2|C_w| + 2 + \sum_{v \text{ nó } P \text{ cinza}} |M_v| \\
\leq\ & 3|S| + 2.
\end{aligned}$$

A última desigualdade se justifica pelas seguintes razões:

$$|M_w| + \sum_{v \text{ nó } P \text{ cinza}} |M_v| \leq |S|$$

já que $M_w \cup \bigcup_{v \text{ nó } P \text{ cinza}} M_v$ é uma união disjunta contida em $S$ e

$$\begin{aligned}
|C_w| &= \sum_{u \in C_w} 1 \\
&\leq \sum_{u \in C_w} |u \cap S| \\
&= |\bigcup_{u \in C_w} (u \cap S)| \\
&= |(\bigcup_{u \in C_w} u) \cap S| \\
&\leq |S|,
\end{aligned}$$

já que os conjuntos de $C_w$ são disjuntos. □

**Teorema 39** *O custo de $PQR(U, \mathcal{C})$ é*

$$custo(PQR(U, \mathcal{C})) = O(s\alpha(s, s)).$$

*onde $s = m + n + r$.*

*Prova:* Seja $T_0$ a árvore P-total e $T_i$ a árvore resultante da adição de $S_i$ a $T_{i-1}$. Do Teorema 38,

$$\sum_{i=1}^{m} mov(T_{i-1}, S_i) + \sum_{i=1}^{m} \Delta pot(T_{i-1}, S_i) \leq \sum_{i=1}^{m} (3|S_i| + 2).$$

Substituindo,

$$\begin{aligned}
\sum_{S \in \mathcal{C}} mov(T, S) &\leq -(pot(T_m) - pot(T_0)) + 3r + 2m \\
&\leq pot(T_0) + 3r + 2m \\
&\leq n + 3r + 2m
\end{aligned}$$

Construir $T_0$ custa $O(n)$. As únicas operações que não podem ser feitas em tempo constante são as operações sobre conjuntos disjuntos. Como o número máximo de operações desse tipo está limitado pelo número total de operações, então

$$\begin{aligned}\sum_{S \in \mathcal{C}} custo(T,S) &= O(n + \sum_{S \in \mathcal{C}} mov(T,S)\alpha(s,s)) \\ &= O(s\alpha(s,s))\end{aligned}$$

que é o custo de $PQR(U, \mathcal{C})$. $\square$

Considerando um computador com palavra de 64 bits, árvores com até $2^{64}$ nós podem ser processadas usando apontadores de uma palavra de tamanho. Neste caso o valor de $\alpha$ não passa de 4 e nosso algoritmo pode ser considerado linear em $s$. Árvores maiores possivelmente acarretarão o acréscimo de um fator $\log_2 s$, ou até mesmo uma potência desse fator, ao custo do nosso algoritmo, por conta do tamanho dos apontadores.

## 4.4 Implementação

Nosso algoritmo foi implementado em C++, com a estrutura de classes que aparece no diagrama UML na Figura 4.11. A lista de filhos foi implementada como um *template*.

## 4.5 Histórico

Nossas primeiras tentativas para obter um algoritmo linear para as árvores PQR foram na direção de construir um grafo de sobreposições estritas, como sugere o algoritmo de Meidanis e Munuera, mais eficientemente.

Uma dessas tentativas foi o *grafo de pisar*. O grafo de pisar é construído processando os conjuntos de uma coleção em ordem crescente de tamanho, acrescentando arestas orientadas para indicar quais conjuntos *pisam* (isto é, têm sobreposição) nos outros. Depois de construir o grafo de pisar podemos obter o grafo de sobreposições estritas para a coleção. Por exemplo, consideremos a seguinte coleção $\mathcal{C}$ de subconjuntos de $U = \{a, b, c, d, e, f, g, h, i\}$ ordenada por tamanho:

1 $\{d, g\}$
2 $\{a, b, c\}$
3 $\{e, f, g\}$
4 $\{d, h, i\}$
5 $\{d, g, h, i\}$
6 $\{d, e, f, g, h, i\}$

Usando cada conjunto da coleção para pisar sobre os elementos de $U$ na ordem em que estão numerados, obteríamos a marcação abaixo e o grafo na Figura 4.12.

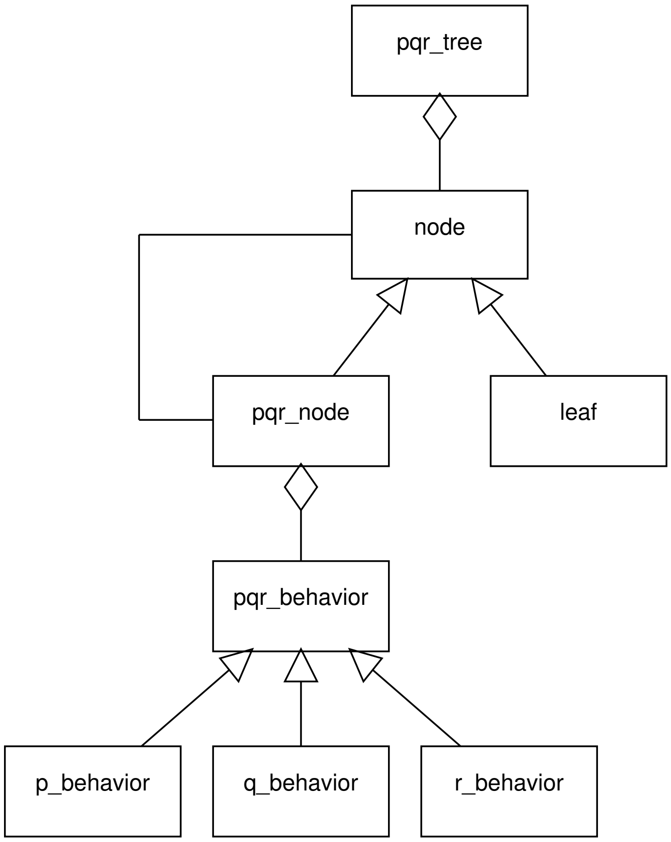



Figura 4.11: Modelo de classes da implementação das árvores PQR. No diagrama, a linha ligando duas classes significa associação, a seta significa herança e o losango significa agregação.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 6 | | | 6 | | | |
| | | | 5 | | | 5 | 6 | 6 | |
| | | | 4 | 6 | 6 | 3 | 5 | 5 | $\uparrow$ ordem das pisadas |
| 2 | 2 | 2 | 1 | 3 | 3 | 1 | 4 | 4 | |
| $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ | $h$ | $i$ | |

O grafo de pisar não foi uma boa solução porque não podíamos diferenciar entre conjuntos contidos e sobrepostos estritamente, e as idéias para contornar esse problema, acrescentando mais informação ao grafo de pisar ou construindo outros grafos auxiliares, eram muito caras.

Uma outra estrutura que idealizamos foi a que chamamos de *árvore de componentes*. Essa árvore representava as componentes conexas do grafo de sobreposições estritas em seus nós, que eram modificados pelos conjuntos em $\mathcal{C}$. Depois que todos os conjuntos eram processados, a árvore de componentes era transformada em uma árvore PQR.

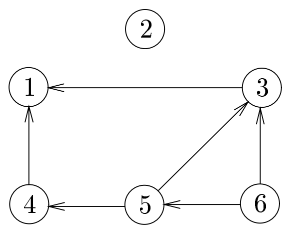


Figura 4.12: Grafo de pisar para a coleção $\mathcal{C}$.

Algumas propriedades dessa estrutura foram enunciadas, mas o algoritmo não deu nenhuma evidência de que era linear porque a conversão entre os tipos de árvores não era fácil. Para ilustrar, a Figura 4.13 mostra uma operação de compressão de uma árvore de componentes. A operação se assemelha àquelas feitas no algoritmo que apresentamos nas seções anteriores, com a diferença que os elementos na árvore são componentes conexas do grafo de sobreposições estritas para $\mathcal{C}$.

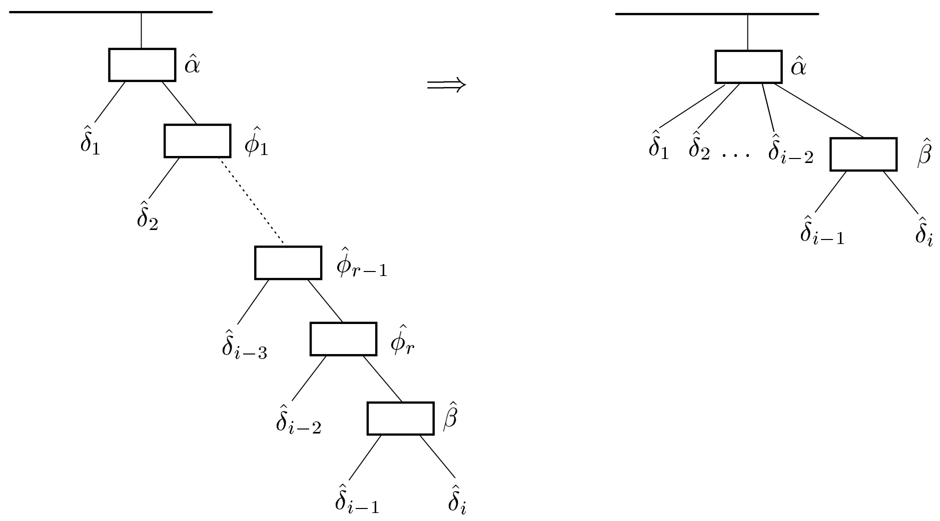


Figura 4.13: Compressão da componente $\hat{\beta}$ em relação a $\hat{\alpha}$.

A idéia seguinte foi a do algoritmo que chamamos de *algoritmo de compressão*. Ele marca a árvore da forma como marcamos agora, mas o processamento começa nas folhas e avança até o LCA. O problema da abordagem é que o número de casos é muito grande e alguns deles são complicados porque têm que levar em conta os tipos e cores de vários nós simultaneamente.

Mesmo com essas limitações tentamos mostrar que esse algoritmo era correto através das operações que chamamos de *transformações atômicas*. Uma transformação atômica foi definida como uma modificação de uma árvore $T$ para uma árvore $T'$ com as seguintes propriedades:

1. $Compl(T) \subseteq Compl(T')$
2. $Compl(T) \neq Compl(T')$
3. Se existe $T''$ tal que $Compl(T) \subseteq Compl(T'') \subseteq Compl(T')$ então $T'' = T$ ou $T'' = T'$.

Alguns exemplos de transformações atômicas aparecem na Figura 4.14.

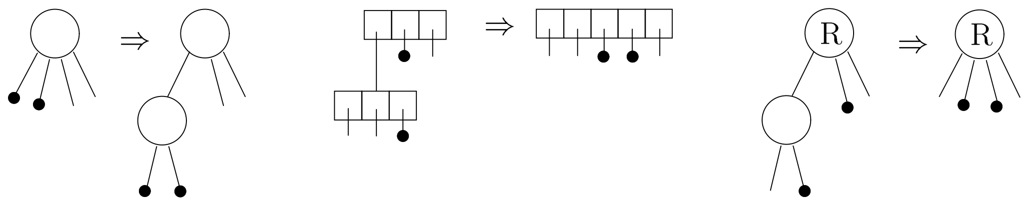



Figura 4.14: Exemplos de transformações atômicas em árvores PQR.

Imaginamos ser possível obter um algoritmo para construir árvores PQR compondo operações atômicas, mas avaliamos que calcular a complexidade de tal algoritmo não seria fácil.

Da junção do algoritmo de compressão e com as operações atômicas construímos o algoritmo que apresentamos nesta tese.

# Capítulo 5

# *Clustering* de Seqüências Expressas de Cana-de-Açúcar

Seqüências expressas ou ESTs (do inglês *Expressed Sequence Tags*) são amostras de moléculas geradoras de proteínas extraídas das células de um organismo. Estas amostras são seqüências das letras A, C, G e T que representam DNA. No Projeto SUCEST (*Sugarcane EST Project*), que produziu quase 300.000 ESTs de cana-de-açúcar, havia a necessidade de formar grupos de seqüências expressas semelhantes para avaliar a redundância do conjunto e para obter seqüências mais longas com base nas sobreposições entre elas. Nossa tarefa foi reformular o esquema de *clustering* do projeto durante seu andamento, para que atendesse melhor às necessidades dos biólogos que usam os dados como fonte para pesquisas sobre a planta.

Neste capítulo apresentamos uma síntese sobre *clustering* em geral, um processo de categorização de objetos no qual não há conhecimento prévio do número nem da composição das categorias onde os objetos serão colocados. Esse problema aparece em várias áreas, como medicina, arqueologia, biologia e outras, e já foi bastante estudado. Textos como o de Anderberg [3], o de Sokal e Sneath [32] e o de Everitt [11] exploram o assunto em mais profundidade, com exemplos de algoritmos e aplicações.

Apresentamos ainda alguns conceitos de biologia molecular que serão necessários para compreender o Capítulo 6, que apresenta o esquema de *clustering* adotado no projeto SUCEST. Tais conceitos são apresentados de forma simplificada e omitindo muitos detalhes que estão além do escopo desta tese. Todos eles são discutidos em profundidade em vários textos, como Lehninger [26], o de Voet e Voet [40] e o de Lewin [20]. Revisamos ainda alguns esquemas de *clustering* para ESTs que aparecem na literatura, para dar uma visão geral sobre as abordagens que já foram usadas para atacar o problema na prática.

## 5.1 Categorização

De acordo com Anderberg [3], classificação, reconhecimento de padrões e *clustering* são processos de atribuir objetos com um ou mais atributos a uma categoria. O que diferencia esses processos é o conhecimento acerca das categorias:

- Na classificação as categorias são estabelecidas e seus atributos essenciais são conhecidos antecipadamente.
- No reconhecimento de padrões a estrutura das categorias não é bem conhecida, mas conhece-se amostras de objetos de cada uma delas. Essas amostras serão usadas para estimar a informação que falta sobre as categorias.
- No *clustering* o número de categorias e a composição das categorias são desconhecidos. As categorias serão determinadas a partir dos atributos dos objetos.

Outros nomes, como identificação, discriminação, aprendizado supervisionado, taxonomia numérica, análise de *clusters*, aprendizado não-supervisionado, particionamento e botriologia também são usados para esses processos. Nomes diferentes aparecem em áreas de aplicação diferentes, que incluem biologia, medicina, psicologia, arqueologia, marketing e biologia molecular [19, 3, 11].

Apesar de ser de língua estrangeira, o termo *clustering* será usado nesta tese tanto para referenciar o processo quanto o resultado do processo de categorização. Usaremos os termos *categoria* e *grupo* indistintamente para nos referirmos aos agrupamentos de objetos produzidos pelo *clustering*.

### 5.1.1 *Clustering*

Métodos de *clustering* tentam categorizar um conjunto de objetos de forma que a recuperação de informação seja eficiente e que possamos generalizar e inferir características dos objetos a partir da categoria onde ele foi colocado. Sendo assim, objetivos diferentes podem levar a categorizações diferentes. Os objetos podem, também, não admitir particionamento porque são muito distintos uns dos outros, ou podem formar uma única categoria porque são indistinguíveis. Em poucas palavras: "qualquer classificação é a divisão de pessoas, objetos ou indivíduos em grupos baseada num conjunto de regras – ela não é verdadeira ou falsa e deve ser julgada principalmente pela utilidade dos seus resultados" [3].

Os métodos numéricos de *clustering* surgiram para tentar tornar a tarefa de categorização mais objetiva. Não obstante, métodos subjetivos e *ad-hoc* podem ser usados tanto para uma análise preliminar dos dados como para categorizá-los. Um exemplo do emprego da visualização é o trabalho de Gilbert *et alii* [16].

Sneath e Sokal [32] argumentam que, de um modo geral, as operações que compõem um processo de *clustering* são as seguintes:

1. Objetos e suas características são selecionados e armazenados.

2. Opcionalmente as similaridades ou dissimilaridades entre objetos são calculadas.

3. Um método é escolhido e categorias são construídas.

4. Generalizações são feitas sobre os dados.

Algumas precauções devem ser tomadas para a seleção dos objetos e para o cálculo das similaridades entre eles. Em primeiro lugar, se eles são a amostra de uma população, a escolha deve ser cuidadosa para que grupos pequenos na população sejam representados. Depois, suas características, também chamadas de variáveis, devem ser selecionadas com o cuidado de evitar aquelas com grande poder discriminativo mas pouco relevantes aos objetivos do *clustering*. O tipo e a escala das variáveis devem ser levados em conta e ajustados se necessário.

Feita a seleção dos objetos, registradas as características que interessam e calculadas as semelhanças entre eles, teremos a entrada para o *clustering*. São essencialmente dois os tipos de entrada:

1. Uma matriz de características, $n \times p$, onde as linhas representam $n$ objetos, cada um deles com $p$ características (variáveis).

2. Uma matriz de similaridades (ou dissimilaridades), $n \times n$, onde $S_{ij}$ contém o valor da similaridade entre os objetos $i$ e $j$.

Uma matriz de características pode ser construída a partir de uma matriz de similaridades através de escalonamento multidimensional [19]. Uma matriz de similaridades pode ser obtida a partir de uma matriz de características aplicando-se equações que relacionem as variáveis [3].

### 5.1.2 Métodos de *Clustering*

Um método de *clustering* é um procedimento que vai receber uma matriz de características ou de similaridades como entrada e vai devolver uma categorização dos objetos. Algumas escolhas vão influenciar a definição do problema, a forma da resposta e a implementação do algoritmo. Algumas dessas escolhas são:

- Admitir ou não sobreposição das categorias.

- Produzir ou não uma hierarquia de categorias que refinam outras do nível anterior.

- Produzir categorias dos objetos, das variáveis ou de ambos.

- Processar os objetos um a um ou em conjunto.

- Processar as variáveis uma a uma ou em conjunto.

- Ser ou não adaptativo, isto é, ser capaz ou não de adaptar parâmetros e algoritmos com base em resultados obtidos em uma ou mais aplicações anteriores do método.

Na literatura há vários tipos de métodos de aplicabilidade geral para *clustering*. A maioria são particionantes. Neste caso o problema é encontrar uma partição dos objetos, isto é, uma divisão de todos os objetos em grupos sem sobreposição. O número de partições de um conjunto de $n$ objetos em $g$ grupos pode ser calculado pela fórmula [11]

$$\frac{1}{g!}\sum_{i=0}^{g}(-1)^{g-i}\binom{g}{i}i^n.$$

Nas seções seguintes vamos falar brevemente sobre os tipos principais de métodos. Everitt [11] apresenta exemplos de aplicação de vários deles.

**Métodos Gulosos** Um exemplo de algoritmo guloso que recebe como entrada uma matriz de similaridades é o seguinte [19]: O algoritmo escolhe um objeto $i$ e cria um grupo contendo esse objeto. Depois acrescenta todo objeto que não pertence a nenhum grupo e que tem similaridade com $i$ maior que um certo limiar $t$ ao grupo que contém $i$. Esses passos são repetidos até que todos os objetos sejam colocados em algum grupo.

Este algoritmo produz uma partição dos objetos. Relaxando a condição de não pertencer a nenhum grupo, obtemos uma versão do algoritmo que produz categorias com sobreposição.

**Métodos Hierárquicos** Um método hierárquico constrói partições sucessivas dos $n$ objetos da entrada. Pode ser aglomerativo ou divisivo:

- Métodos aglomerativos: Um método aglomerativo produz partições $P_n, P_{n-1}, \ldots, P_1$ em seqüência, onde $P_n$ consiste de $n$ grupos unitários e $P_1$ consiste de um único grupo com $n$ elementos. A cada passo do algoritmo, o par de grupos mais similar é unido. Há várias maneiras de calcular a similaridade entre dois grupos, por exemplo, a *ligação simples*, que é a maior similaridade entre um par de objetos tomados de dois grupos distintos, e a *ligação média do grupo*, onde a similaridade entre dois grupos é definida como a média das similaridades entre todos os pares formados com um elemento de cada grupo.

- Métodos divisivos: Um método divisivo produz partições na ordem inversa de um método aglomerativo: $P_1, \ldots, P_{n-1}, P_n$. A cada passo do algoritmo, um grupo é dividido em duas partes mais homogêneas possível. Há definições de medidas de homogeneidade sobre as variáveis da matriz de características e sobre matrizes de similaridade.

Normalmente uma representação gráfica é construída para a hierarquia de partições. Essa representação, chamada dendrograma, é parecida com uma árvore binária enraizada onde as folhas representam as $n$ partições unitárias, cada nó interno uma fusão ou fissão de um grupo e a raiz representa a partição com um único grupo.

Uma dificuldade nesse método é determinar o número de grupos que é o resultado final da classificação, exceto quando o que se deseja é a hierarquia completa. Há métodos numéricos para tentar determinar o número ideal de grupos. Outra opção é determiná-lo visualmente, com a ajuda do dendrograma.

### Métodos Otimizantes

A idéia básica desses métodos é:

1. Definir uma função que mapeia cada partição de $n$ elementos em $g$ grupos em um número real que indique a qualidade dessa partição.

2. Maximizar essa função.

Como o número de partições de $n$ objetos em $g$ grupos é grande, um algoritmo de otimização é usado para buscar a solução ótima. Um algoritmo simples é o que, a partir de uma partição inicial, escolhe a melhor troca de um único objeto de uma partição para outra e executa essa troca para melhorar o valor da função até que não seja mais possível melhorar.

Nesses métodos normalmente é preciso escolher uma partição inicial, que pode afetar a partição final [19]. O número de grupos na partição inicial também deve ser escolhido, e novamente, isso pode ser feito de maneira subjetiva ou usando algum critério numérico [11].

### Métodos probabilísticos

Esses métodos supõem que os objetos são a amostra de uma população. Eles são adequados quando considera-se que os dados contém $g$ grupos, e que para uma dada característica, cada grupo tem uma função de densidade condicional, que não é conhecida. Conhece-se apenas a função de densidade para a amostra, que é uma superposição das funções de cada grupo, chamada de mistura de distribuições de densidades. As distribuições normais são as mais usadas.

Os grupos são formados com base nos valores máximos de probabilidade estimados de que cada indivíduo pertença a um grupo. Para isso, tanto o número de grupos como os parâmetros das densidades de cada um devem ser estimados.

**Outros Métodos**

Além dos métodos citados até agora, há outros dos quais destacamos: a busca de densidade, a análise modal e o *clustering* fuzzy.

A busca de densidade vê os objetos como pontos no espaço métrico, procura por regiões de alta densidade e considera cada região como um grupo.

Na análise modal imagina-se esferas de raio $R$ ao redor de cada ponto do espaço. As esferas são classificadas como densas ou não-densas, dependendo do número de pontos que contém. O raio $R$ é aumentado gradativamente e uma esfera densa é unida a outra em função de alguma métrica, como distância entre os centros.

No *clustering* fuzzy, depois que o número de grupos é determinado, a cada objeto são atribuídos graus de pertinência a cada um dos grupos. Um particionamento pode ser obtida a partir de um *clustering* fuzzy atribuindo cada objeto ao grupo a que ele tem maior grau de pertinência.

### 5.1.3 Validação de *Clustering*

Validação de *clustering* é a avaliação dos resultados da aplicação de um método de *clustering* [19, 41]. Consiste na escolha de um critério de validação e na avaliação de índices. Os três tipos de estruturas produzidos por um método de *clustering* podem ser avaliados: hierarquias de partições, partições e grupos.

Os critérios de validação, isto é, as estratégias de validação dessas estruturas podem ser de três tipos:

- Critérios externos: A estrutura é comparada com algum conhecimento prévio sobre os dados, por exemplo, um particionamento pré-existente.
- Critérios internos: A estrutura é avaliada usando apenas os próprios dados, por exemplo, medindo a similaridade média dos grupos obtidos.
- Critérios relativos: Duas estruturas são comparadas para determinar qual delas é melhor em algum sentido.

Um índice para validação mede a adequação da estrutura obtida por um método de *clustering*, isto é, se a estrutura fornece informação verdadeira sobre os dados e se reflete características reais dos dados. Uma forma de definir índices para validação de *clustering* é através de testes de hipóteses estatísticas. Além de testes estatísticos podem ser empregados procedimentos *ad hoc*, bem como verificações visuais.

## 5.2 Expressed Sequence Tags

Proteínas são moléculas que têm várias funções e estão envolvidas em quase todos os processos dentro de uma célula, como reprodução, respiração, defesa e movimentação. Por isso, cientistas que estudam os processos celulares têm um grande interesse nessas moléculas.

Uma **proteína** é uma cadeia de moléculas chamadas aminoácidos. Existem 20 aminoácidos diferentes. Embora toda proteína tenha uma estrutura tri-dimensional e sua função seja determinada em parte por essa estrutura, no escopo deste texto podemos pensar em uma proteína como uma cadeia linear das letras A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W e Y.

Uma célula é capaz de produzir proteínas usando o seu DNA como molde. O conjunto de todo o DNA de um organismo é chamado de **genoma**. O **DNA** é um agrupamento linear de moléculas chamadas de **nucleotídeos** ou **bases**. As quatro bases que compõem um DNA normal são adenina, citosina, guanina e timina. As extremidades livres do DNA são diferenciáveis; uma delas chamada de 5' e a outra de 3'. Podemos pensar então no DNA como uma cadeia linear das letras A, C, G e T, como no exemplo abaixo.

```
5'  ACCTGGATCAGCGCATAGATAACGAG  3'
```

Em estado natural, o DNA forma uma estrutura chamada de **fita dupla**: duas cadeias de bases complementares e invertidas se mantém juntas através de pontes de hidrogênio. Na fita dupla, a extremidade 5' de uma fita emparelha com a extremidade 3' da outra, uma base A emparelha com uma base T e uma base C emparelha com uma base G, como no exemplo abaixo.

```
5'  ACTGAACATTCGTTCGAGG  3'
3'  TGACTTGTAAGCAAGCTCC  5'
```

Uma fita pode ser obtida a partir da outra complementando-se as bases e invertendo-se a fita. Técnicas laboratoriais permitem separar as duas fitas de um DNA, obtendo DNA **fita simples**, e permitem obter DNA fita dupla a partir de DNA fita simples.

Para construir uma proteína, estruturas especializadas na célula são capazes de usar o DNA como receita, no processo chamado **tradução**. Um esquema, chamado **código**

**genético**, associa cada seqüência de três bases a um aminoácido, como no seguinte exemplo:

$$\begin{array}{rl} \text{DNA} & \underbrace{\texttt{AGT}}\underbrace{\texttt{ATT}}\underbrace{\texttt{GGC}}\underbrace{\texttt{AAT}}\underbrace{\texttt{CGA}}\underbrace{\texttt{TGC}}\underbrace{\texttt{AAA}} \\ \text{proteína} & \;\;\texttt{S}\quad\;\; \texttt{I}\quad\;\; \texttt{G}\quad\;\; \texttt{N}\quad\;\; \texttt{R}\quad\;\; \texttt{C}\quad\;\; \texttt{K} \end{array}$$

Nem todo o DNA em uma célula codifica proteínas, apenas os trechos chamados de **genes**. Os genes de eucariotos (organismos superiores, cujas células têm núcleo) normalmente têm a seguinte estrutura: (i) primeiro vêm seqüências de bases que permitem que a maquinaria celular reconheça o gene e regule sua atividade, é a região promotora, (ii) depois se alternam seqüências de bases que são usados na codificação da proteína, os **éxons**, e seqüências de bases que não usados na codificação da proteína, os **íntrons** e (iii) depois vem uma região que marca o fim do gene.

Para construir uma proteína, a maquinaria celular faz os seguintes passos no interior do núcleo da célula (Figura 5.1):

- Transcrição: Copia o gene desde o primeiro até o último éxon, em uma molécula de RNA chamada de RNA mensageiro (mRNA), no processo chamado **transcrição**. O RNA é similar ao DNA em estrutura, mas no lugar da base timina (T) aparece a uracila (U) e naturalmente ele é uma fita simples, não uma fita dupla como o DNA.

- Adição de poli-A: Acrescenta entre 100 e 250 bases A ao final de um ou mais éxons no mRNA. Essa região é chamada **poli-A**.

- Remoção de íntrons e éxons: Remove os íntrons e alguns éxons do RNA. Em conjunto com as regiões poli-A (o primeiro éxon que contém poli-A normalmente é o último que permanece no mRNA) a remoção de éxons permite que um gene dê origem a proteínas diferentes, mecanismo chamado expressão diferencial (*alternative splicing*).

Depois desses passos, o mRNA deixa o núcleo da célula e é usado como fonte para produzir uma proteína, no processo chamado **tradução**. Um mRNA pronto para ser traduzido em proteína é chamado de **transcrito** ou **seqüência expressa**.

## 5.3 Projetos EST

Para conhecer as proteínas produzidas em uma célula podem ser usados dois métodos. O primeiro é obter a seqüência de todo o DNA da célula e depois encontrar os genes nessa seqüência diretamente. As duas dificuldades principais desse método são, primeiro, o tamanho dos genomas de eucariotos, o que ainda representa um desafio para a tecnologia

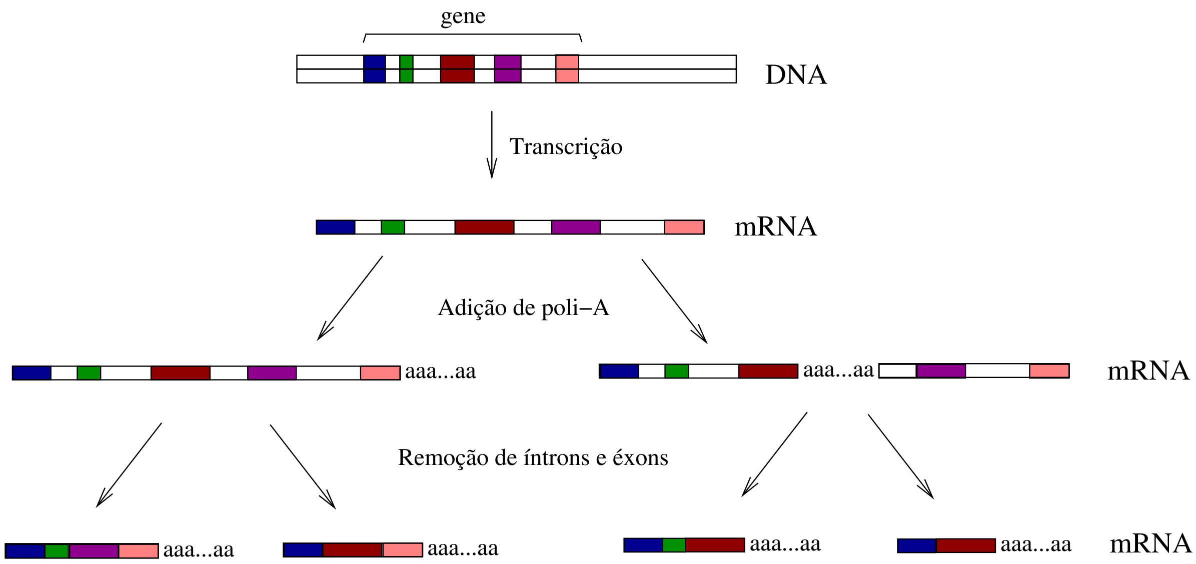



Figura 5.1: Seqüência de transformações, no interior da célula, para a construção de mRNAs a partir de um gene. A figura ilustra como um mesmo gene pode ser transcrito várias vezes e dar origem a vários mRNAs diferentes.

de seqüenciamento atual. O humano, por exemplo, tem 3 bilhões de bases. O da cana-de-açúcar, 930 milhões de bases. Outra dificuldade é identificar os genes, seus éxons e íntrons a partir da seqüência de bases de todo o DNA de um organismo superior.

O segundo método é coletar moléculas de mRNA e seqüenciá-las diretamente. A seqüência de um mRNA obtida através desse método é chamada de **EST** [1], do inglês *expressed sequence tag*. Aplicando o código genético ao EST obtém-se a seqüência da proteína que é produzida pela célula. Uma vantagem desse método é a rapidez com que os dados são obtidos. Uma desvantagem desse método é que, como a coleta é aleatória, a probabilidade de que mRNAs mais escassos sejam amostrados é pequena e mRNAs abundantes provavelmente serão amostrados várias vezes.

Um projeto EST é um tipo de projeto genoma que tem como objetivo amostrar vários mRNAs de células de um organismo de interesse e produzir ESTs a partir deles. Um projeto EST típico produz entre 50 mil e alguns milhões de seqüências. A produção de ESTs passa por várias fases em laboratório:

- Inicialmente as células são preparadas em cultura.

- As células são rompidas, o material resultante passa por filtragens e lavagens e boa parte do RNA que tem poli-A presente nas células é separado. Além de mRNA, existem na célula outros tipos de RNA, como **RNA ribossomal** (rRNA) e RNA de transporte (tRNA), que não são usados como moldes de proteínas.

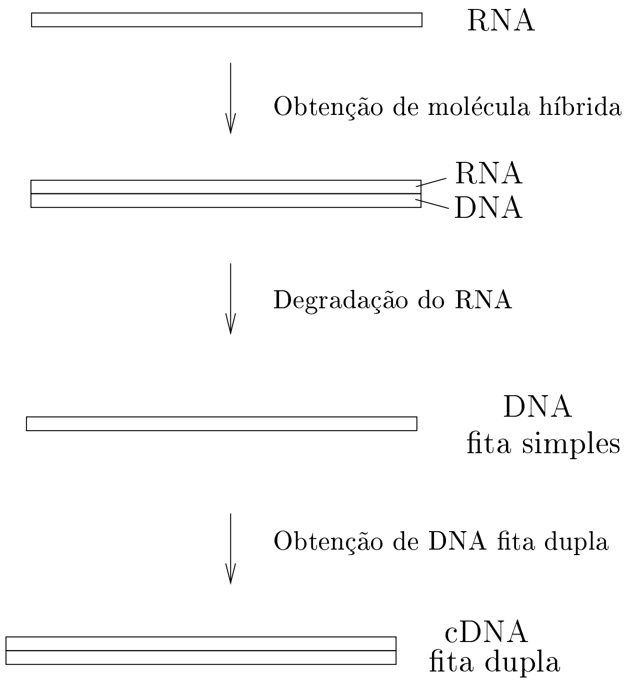



Figura 5.2: Obtenção de DNA fita dupla a partir de RNA.

- Os RNAs são processados por reagentes e enzimas e obtém-se uma molécula híbrida que tem uma fita de DNA e outra de RNA. Após algumas outras reações que degradam o RNA, obtém-se DNA fita simples e depois DNA fita dupla. Esse DNA é chamado de **DNA complementar** (cDNA) e tem seqüência equivalente à do mRNA (Figura 5.2). Neste ponto do processo tem-se um grande número de moléculas de DNA fita dupla diferentes.

- Uma reação adiciona **seqüências adaptadoras** às duas pontas de cada cDNA, como ilustrado na Figura 5.3. Estas seqüências são curtas e possuem um sítio de restrição (veja nos ítens abaixo). O cDNA nesse ponto possui um sítio de restrição diferente em cada ponta.

- As moléculas de DNA com seqüências adaptadoras são colocados em solução juntamente com **vetores de clonagem**. Um vetor de clonagem é uma molécula de DNA que contém, além de outros aparatos, um ou mais genes que conferem resistência a certos antibióticos e sítios de restrição (normalmente os mesmos sítios que os das seqüencias adaptadoras).

- Duas enzimas de restrição são adicionadas à solução contendo os cDNAs e os vetores de clonagem. Uma enzima de restrição corta moléculas de DNA sempre em locais

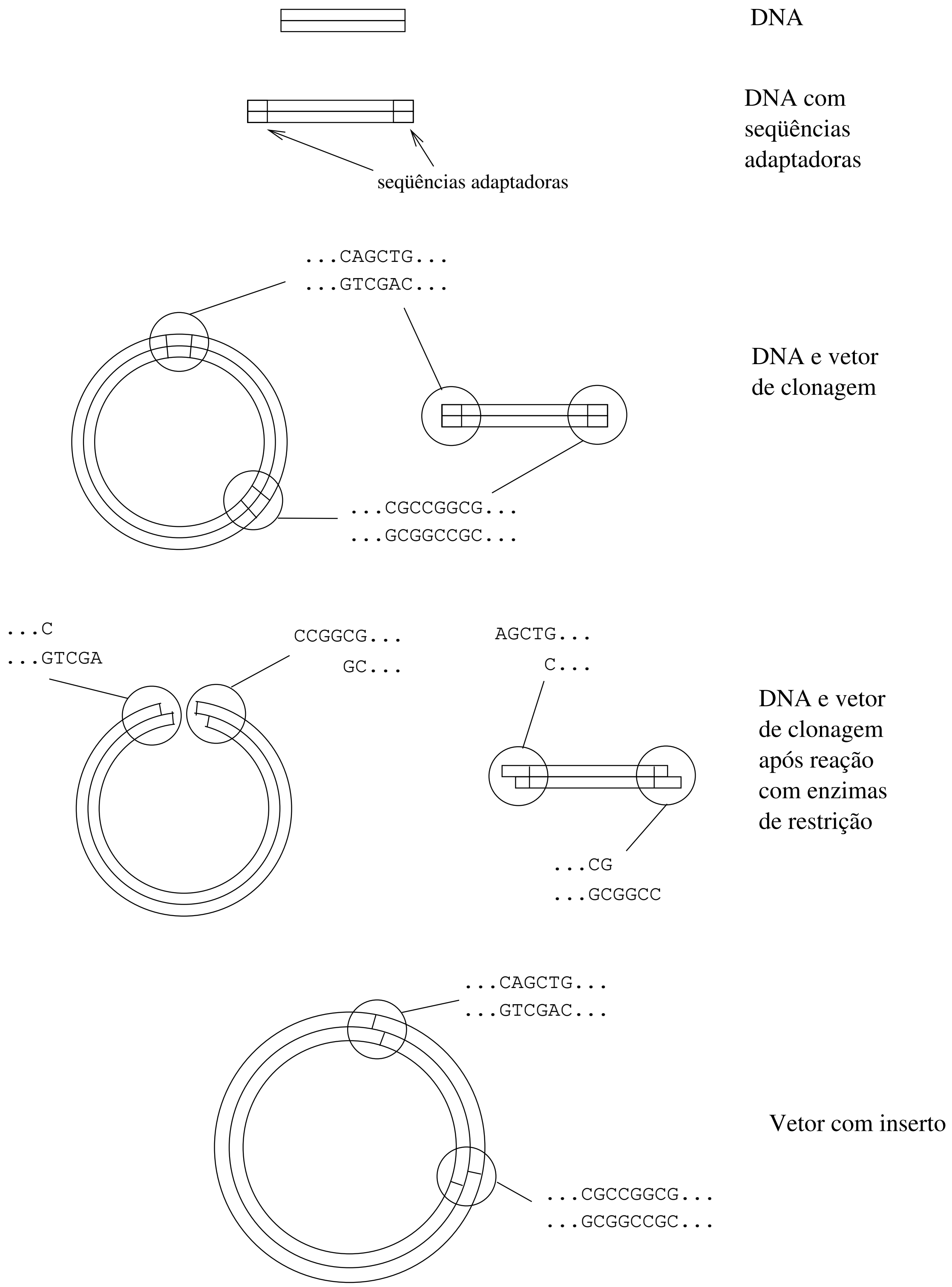



Figura 5.3: Passos para inserir um fragmento de DNA em um vetor de clonagem. Os sítios de restrição ilustrados são das enzimas NotI e SalI.

onde exista uma seqüência fixa, específica da enzima. Por exemplo, a enzima EcoRI corta a seqüência

```
...GAATTC...
...CTTAAG...
```

criando duas pontas:

```
...G          AATTC...
...CTTAA          G...
```

Um **sítio de restrição** é um ponto no DNA que tem a seqüência de bases que uma enzima de restrição corta.

- Uma reação faz com que as pontas livres dos DNAs e dos vetores de clonagem (recém cortadas pela enzima de restrição) se juntem. Essa reação produz DNAs agregados a vetores. Chamamos um DNA nessa situação de **inserto**. Uma parcela dos vetores se une novamente sem nenhum inserto. Apesar dos sítios de restrição diferentes nas duas pontas, alguns insertos entram invertidos no vetor e alguns vetores recebem mais de um inserto. Nesse caso chamamos este inserto de **inserto quimérico**.

- O vetor é inserido em uma bactéria. As bactérias que contém o vetor de clonagem são selecionadas pela aplicação de um ou mais antibióticos e separadas individualmente. Neste ponto tem-se então várias bactérias individualizadas, cada uma delas contendo um vetor de clonagem.

- Cada bactéria selecionada é colocada em meio de cultura e se reproduz. Ao se reproduzir, ela copia o vetor de clonagem como se fizesse parte do seu próprio material genético. Tem-se então várias colônia de bactérias, onde todos os milhões de indivíduos de cada colônia tem uma cópia do mesmo vetor de clonagem.

- As bactérias de cada colônia são rompidas, lavadas e filtradas separadamente. As milhões de cópias dos vetores de clonagem são isoladas.

- Cada conjunto de milhões de cópias de um inserto é preparado e seqüenciado uma única vez. Normalmente o seqüenciamento começa um pouco antes do início do inserto. O seqüenciamento pode ser feito na direção 5' ou na direção 3'.

O seqüenciamento produz um arquivo especial chamado **cromatograma** para cada inserto. Cada arquivo é processado e a interpretação do arquivo gera uma seqüência que tem entre 300 e 1.000 bases. É comum que a interpretação do cromatograma forneça também, para cada base, uma medida da precisão da com que ela foi lida, chamada **qualidade da base**. Phred [12] é um dos programas que interpreta um cromatograma e produz as seqüências de bases e suas qualidades. A **qualidade phred** $q$ de uma base é definida como

$$q = -10 \times \log_{10}(p),$$

onde $p$ é a probabilidade de erro estimada para a base. Assim, uma base com probabilidade de $10^{-3}$ de estar errada recebe um valor de qualidade 30.

Em um projeto EST são produzidos vários ESTs de células de um organismo. Normalmente são escolhidas células de tecidos diferentes e em condições variadas para a produção de ESTs, porque células de tecidos diferentes e células do mesmo tecido mas em condições diferentes expressam conjuntos de genes diferentes. Dá-se o nome de **biblioteca** a uma combinação tecido/condição que originou os ESTs de um projeto. Por exemplo, em um projeto EST de guaraná pode ser produzida uma biblioteca de raiz cultivada em solo ácido, ou uma biblioteca de caule sadio e outra de caule quando a planta está infestada por uma doença.

Por conta das técnicas de laboratório atuais, os ESTs possuem certas características que fazem com que seja importante tomar cuidado ao processá-los e utilizá-los como fonte de informação:

- ESTs normalmente têm entre 300 e 1.000 bases. As bases têm qualidade variável ao longo da sua extensão. Normalmente as pontas têm qualidade mais baixa.

- ESTs têm poli-A ou poli-T, muitas vezes longos e de alta qualidade.

- ESTs têm trechos das seqüências adaptadoras e do vetor que foram usados na clonagem.

- Pode ter havido contaminação em alguma etapa do processo, e então podem ser produzidos ESTs que não representam seqüências expressas do organismo que está sendo investigado.

- Pode haver seqüências de RNA ribossômico no conjunto de ESTs.

- ESTs podem ser sofrer derrapagem. Este é um fenômeno de seqüenciamento que acontece principalmente quando há poli-T ou poli-A longos na molécula de mRNA. O resultado é que a seqüência produzida apresenta vários trechos com bases ecoadas (veja o Capítulo 6).

- Pode haver seqüências originadas do seqüenciamento de insertos quiméricos.

Vale ainda reforçar que o conhecimento atual sobre os processos celulares é incompleto; vários mecanismos ainda são desconhecidos ou pouco compreendidos. A experiência prática demonstra também que quase sempre não é possível enquadrar processos biológicos em modelos exatos e que devemos sempre contar com exceções, falsos positivos e falsos negativos em nossas análises e algoritmos.

## 5.4 *Clustering* de ESTs

Em um projeto EST podem ser vários os objetos para o *clustering*:

1. Agrupar ESTs que vieram do mesmo transcrito.
2. Agrupar ESTs que vieram do mesmo alelo (Alelos são genes do mesmo organismo que têm seqüências muito próximas, isto é, que diferem em poucas bases, e que teoricamente deveriam dar origem à mesma proteína.)
3. Agrupar ESTs que vieram do mesmo gene.
4. Agrupar ESTs de genes da mesma família.

Agrupar ESTs do mesmo transcrito permite avaliar a redundância durante o andamento de um projeto EST e decidir quando parar a produção de ESTs de cada biblioteca ou finalizar o projeto. Também é possível produzir uma montagem dos ESTs em um grupo (veja abaixo), obtendo uma seqüência mais longa e de melhor qualidade, e eventualmente, a seqüência completa do mRNA. Agrupar os ESTs que vieram do mesmo gene permite identificar genes do organismo e os transcritos que foram produzidos a partir dele.

Na literatura há descrições de vários esquemas de *clustering* de ESTs. Os autores defendem as vantagens e desvantagens de cada método, às vezes comparando-os com outros usados por grandes institutos de pesquisa, como NCBI (www.ncbi.nlm.nih.gov) e TIGR (www.tigr.org). Antes de descrever brevemente alguns deles, vamos definir alguns termos.

**Programas de montagem** ou **montadores** são programas que recebem como entrada um conjunto de seqüências de DNA e, em função das sobreposições entre elas, produzem **contigs**, isto é, grupos de seqüências que têm sobreposição, e uma seqüência consenso para cada contig, que é uma superseqüência de todas as que fazem parte do contig. Exemplos desses programas são CAP3 [18], phrap [29] e TIGR assembly [33]. Normalmente esses programas permitem definir parâmetros para o seu funcionamento, como o número mínimo de bases em uma sobreposição aceitável de duas seqüências, a quantidade aceitável de bases divergentes em uma sobreposição e outros. Em geral montadores são feitos para montar seqüências do genoma de um organismo, e não ESTs.

Usados para agrupar ESTs, existe uma tendência de que eles produzam contigs quiméricos. Por outro lado, é desejável ter uma seqüência consenso para cada grupo porque isso facilita a realização de buscas baseadas em similaridade.

Um **alinhamento local** entre duas seqüências $r$ e $s$ é formado por uma subcadeia de $r$ e uma subcadeia de $s$ que, depois de receber a inserção de zero ou mais buracos em posições arbitrárias, quando colocadas uma sobre a outra têm o mesmo tamanho. Para a comparação de seqüências o que interessa são os alinhamentos locais em que as colunas coincidem bastante, ou seja, subcadeias de $r$ e $s$ que são bastante semelhantes. O algoritmo exato para encontrar alinhamentos locais entre duas seqüências $r$ e $s$ tem complexidade $O(|r|.|s|)$. O programa SWAT [36] implementa o algoritmo exato e o programa cross_match [36] implementa algumas otimizações para torná-lo mais rápido. Para um grande volume de seqüências, são usadas as heurísticas BLAST [2] e FASTA [28], mais rápidas que o algoritmo exato e bastante sensíveis.

TIGR Gene Indices [30] é o esquema de *clustering* usado pelo instituto TIGR. É um esquema incremental. Primeiro, os novos ESTs passam por uma limpeza para remoção de vetores, contaminantes e seqüências de baixa qualidade. Em seguida, os consensos dos blocos da versão anterior são adicionados ao conjunto. Depois, os mRNAs (não provenientes de ESTs) e trechos codificantes de genes completos depositados no banco de dados de seqüências Genbank (www.?) são adicionados ao conjunto. Todas as seqüências são comparadas entre si usando BLAST. Seqüências suficientemente similares são colocadas no mesmo grupo. As seqüências que formaram os consensos da versão anterior são adicionadas aos respectivos grupos. Cada grupo é montado pelo montador CAP3. Os consensos obtidos são montados em conjunto para eliminar redundâncias. Ao final, cada grupo é formado pelos contigs obtidos na montagem.

Unigene [5] é o esquema de *clustering* usado pelo NCBI. O primeiro passo é uma limpeza dos ESTs para remover vetores, contaminantes, elementos repetitivos, RNA mitocondrial e RNA ribossomal. Depois os mRNAs (não provenientes de ESTs) e trechos codificantes de genes completos depositados no banco de dados de seqüências do NCBI são comparados entre si. As que são suficientemente similares formam os grupos iniciais. O conjunto de ESTs é comparado com os genes usando uma variação do algoritmo BLAST chamada MegaBLAST (www.ncbi.nlm.nih.gov) e pares suficientemente similares são adicionados aos grupos. Qualquer EST que ligue dois grupos iniciais são descartados. Depois, qualquer grupo que não contenha pelo menos uma seqüência com poli-A ou pelo menos dois ESTs seqüenciados na extremidade 3' são descartados. Em seguida os ESTs que não pertencem a nenhum grupo são comparados novamente com os grupos, com critérios mais relaxados. Finalmente, os grupos unitários são comparados com os outros grupos e colocados no grupo que contém a seqüência mais similar.

O algoritmo JESAM [27], usa indexação para construir os grupos. Primeiro ele constrói uma tabela de hash usando todas as subseqüências de 12 letras que não se sobrepõem de todos os ESTs. Depois remove as entradas de baixa complexidade (por exemplo, poli-A, poli-C, poli-G e poli-T) e entradas muito abundantes. Depois, para cada EST, desliza uma janela de tamanho 12 do início até o seu fim, em passos de uma base, e recupera as posições de todas as ocorrências das janelas nos outros ESTs. Em seguida ele constrói um alinhamento para as seqüências. Depois os alinhamentos são pontuados e os que atingem boa pontuação são armazenados. Finalmente os grupos são formados por todas as seqüências que estão juntas em algum alinhamento.

O algoritmo d2_cluster [7] é uma variante de *clustering* hierárquico aglomerativo. A entrada para o algoritmo é uma lista de $n$ seqüências rotuladas de 0 a $n-1$. No início há $n$ blocos unitários. O algoritmo percorre a lista de seqüências de 0 a $n-1$ e compara, na $i$-ésima iteração, a seqüência $i$ com todas as de rótulo $j < i$. Se a distância entre as seqüências $i$ e $j$ é menor que um certo limiar, os blocos que as contém são unidos. A distância entre duas seqüências é definida como o tamanho da maior subseqüência comum com pelo menos $I\%$ de identidade entre as bases.

O método CRAW [8] particiona as seqüências usando um método guloso similar ao que citamos na Seção 5.1.2. Usa a semelhança entre as extremidades de um EST e o consenso de um grupo para decidir se ela será adicionado ao grupo ou não.

O método STACK [25] é usado para categorizar ESTs humanos disponíveis em bancos de dados públicos. Primeiro, os ESTs são divididos em conjuntos de acordo com os tecidos de onde se originaram. Depois as partes dos ESTs que têm semelhança com contaminantes conhecidos são mascaradas, isto é, têm suas bases substituídas por "x". O algoritmo d2_cluster é usado em cada conjunto de ESTs. Cada grupo obtido pelo d2_cluster é montado pelo montador phrap e o melhor consenso resultante de cada montagem (com base no número de ESTs e nas colunas do alinhamento) é escolhido como representante do grupo. Por fim, os representantes e os ESTs de grupos unitários são ligados se são originados do mesmo inserto.

No projeto da cana-de-açúcar usamos apenas o montador CAP3 para produzir o *clustering*, como veremos no próximo capítulo, onde consideraremos também algumas limitações dessa abordagem.

# Capítulo 6

# Trimming and clustering sugarcane ESTs *

Guilherme P. Telles and Felipe R. da Silva

**Abstract**

The original clustering procedure adopted in the Sugarcane Expressed Sequence Tag project (SUCEST) had many problems, for instance too many clusters, the presence of ribosomal sequences, etc. We therefore redesigned the clustering procedure entirely, including a much more careful initial trimming of the reads. In this paper the new trimming and clustering strategies are described in detail and we give the new official figures for the project, 237,954 expressed sequence tags and 43,141 clusters.

* ***Publicado em Genetics and Molecular Biology, volume 24, págs. 17–23, 2001***

## 6.1 Introduction

The Sugarcane EST project (SUCEST) produced 291,689 expressed sequence tags (ESTs) [1]. In the pipeline of the project it was important to cluster together sequences from the same transcript molecule and to obtain a representative sequence for each group. Clustering was important to evaluate the redundancy of the set of ESTs during library production and sequencing, and at the end of the project. Clustering also produces a smaller set of sequences which facilitates investigation of the data by biologists and computer scientists [35].

As in any other EST project, the raw SUCEST sequences sometimes contained unwanted segments like polyadenylation (poly-A), regions with low base quality, fragments from vectors and adapters, and slippage. Some reads may also came from ribosomal RNA or contaminant DNA. Such segments are unwanted because they introduce similarity between ESTs that has no relevance for clustering, and removal of such segments is essential to cluster correctly.

Trimming and clustering procedures were established at the beginning of the project in July 1999, but the amount of data grew each day and it soon became clear that the trimming and clustering procedures were both not good enough. SUCEST data-users were pointing out many problems when we designed and implemented new trimming and clustering procedures.

A trimming procedure is essentially the task of searching ESTs for unwanted regions, identifying them and then deciding whether to remove the unwanted region or to discard the entire EST. Trimming has already been described for UniGene (www.ncbi.nlm.nih.gov /UniGene), TIGR Gene Indices [30] and STACK [25].

In the SUCEST project, clustering was always performed using a fragment assembler for the whole set of ESTs. This is different from the procedure used by Unigene, TIGR Gene Indices, JESAM [27] and STACK which use some kind of pairwise comparison to estimate distance between ESTs, build clusters and then, if ever, assemble the clusters separately. In its first version, SUCEST clustering scheme produced 81,223 clusters (41,582 singletons) while the current version has 43,141 clusters (16,838 singletons).

In this paper we describe trimming in detail, because it had a major influence on the work performed by the assembler at the clustering stage. We have also compared the results of different assemblers for our set of ESTs before we decide in favor of the CAP3 program [18]. Although we had confidence in the fragment assemblers comparison performed by Liang et al. [21], three issues motivated us to produce our own comparison routines. Firstly, we wanted to examine the assembly results for our particular set of ESTs, secondly, we were using ESTs quality data and, thirdly, we used parameters for the assemblers that differ from the default ones. We also introduce the trimming and clustering procedures early in the project. Our intention in this paper is not to emphasize our improved results but to show the remarkable effect that 'noise' (i.e. unwanted sequences) can have on clustering.

## 6.2 Methodology and Results

Clone libraries were prepared as described by Vettore et al. [38] and sequenced by ABI 377 (Applied Biosystems) machines. After being processed by the phred base-calling program (version 0.980904.e, www.phrap.org) and by the phd2fasta program (version 0.990622.d, www.phrap.org), ESTs were stored as fasta and quality files in the 5' to 3' orientation. These files contained 291,689 sequences with an average length of 864.5 $\pm$186.3 bases. The average number of bases with a phred quality value greater than 20 per read was 399.5 $\pm$151.3. The programs were run on an 8 GB RAM AlphaServer ES40 (Compaq) with 2 processors at 667 MHz executing the OSF1 operating system (version 4.0G).

### 6.2.1 Trimming

An EST set may contain unwanted sequences made up of poly-A fragments, vector and adapter fragments, low quality ends, ribosomal RNA, contaminant DNA and slipped sequences. When clustering the sequences to produce groups of transcripts, these unwanted sequence introduce irrelevant relationships between reads. Trimming is the removal of such regions from ESTs or the removal of entire ESTs from the set.

Trimming refined the reads in several steps, using the blastall program (version 10/31/2000, www.ncbi.nlm.nih.gov) that implements the BLAST algorithm [2], the cross_match program (version 0.990319, www.phrap.org), the SWAT program (version 0.990319, www.phrap.org) and ad hoc pattern-matching programs written in Perl (version 5.6.0, www.cpan.org). Parsers (programs that do some kind of interpretation on data based on its syntactical structure) for the output of these programs were written in Perl, and bash (version 2.04.0(1), www.gnu.org) scripts were used to filter, build histograms and summarize data. Some regions, like poly-A, were searched several times, each time with a different recognition criterion. Trimming was tuned to keep as much as possible from each sequence.

The trimming scheme is summarized in Figure 6.1. The first step was the removal of ribosomal RNA sequences, and for this the ESTs were compared against 18S rRNA from Zea mays (GenBank AF168884), 5.8S rRNA from Platanus occidentalis (GenBank AF162215) and 26S rRNA from Lambertia inermis (GenBank AF274652) using the BLAST program. The choice of these rRNA sources was based on the similarity between them and sugarcane rRNA. A match with an e-value less than $10^{-10}$ was the threshold to discard a read, a total of 8,473 reads being removed in this step.

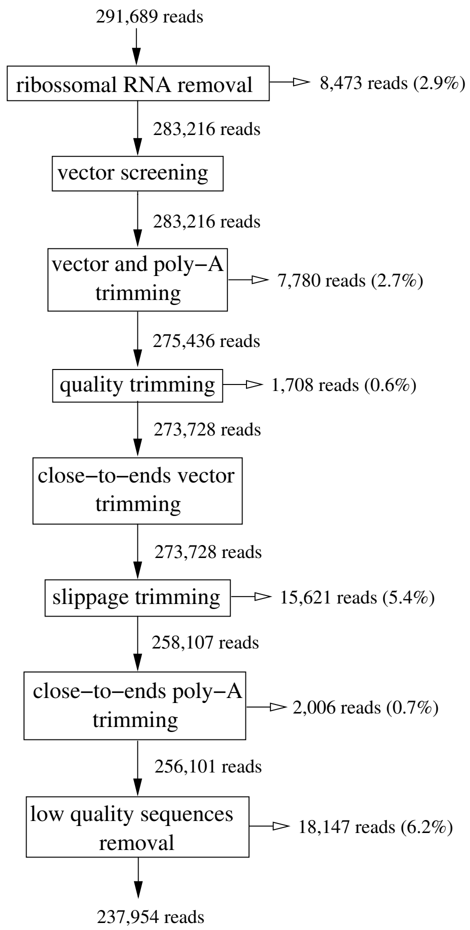



Figure 6.1: Overview of trimming procedure. White-headed arrows indicate the number of reads discarded in each step, with the percentage of total shown in parenthesis.

The next step was vector and adapter sequence masking, using the cross_match program that replaced bases with an X if they were very similar to vector and adapter sequences used in the clone libraries. This was followed by removing the vector and adapter sequences themselves from the reads by deleting the regions marked with an X. The actual treatment given to these ambiguous regions depended on where the X-regions were found and how many there were, an X-region being a contiguous masked sub-sequence in a read.

Classes were devised based on the analysis of histograms of the lengths of X-regions, distance of the X-region from the 5' and 3' ends and on the analysis of the number of ESTs falling into each class. These classes were as follows:

1. Class 1. There were two distinct X-regions in the read, this being what is to be

expected as the result of sequencing a clone with a small insert. In this case only the sequence between the X-regions was kept.

2. Class 2. There were more than two X-regions in the read, probably because of a low-quality vector. In this case we did not change the read.

3. Class 3. There was only one X-region of no more than 300 bases that was less than 50 bases away from the 5' end. This was the case when the region from the X-region down to the 5' end probably consisted of vector sequences extending from the sequencing priming site to the cloning site. In this case we removed the X region together with the 5' end.

4. Class 4. There was only one X-region with more than 300 bases that was less than 50 bases away from the 5' end. In this case the clone probably had no insert so we discarded the whole read.

5. Class 5. There was only one X-region of at most 300 bases that was 51 to 300 bases from the 5' end. In this case it was hard to decide what the insert was so the read was not changed.

6. Class 6. There was only one X-region with more than 300 bases that was 51 to 300 bases from the 5' end. This probably occurred when the X-region and the 3' end consisted of a vector sequence after the cloning site. In this case we removed both the X-region and the 3' end.

7. Class 7. There was only one X-region of any length and it was at least 300 bases away from the 5' end. In this case we again removed both the X-region and 3' end because the deleted region probably consisted of a post cloning-site vector sequence.

While removing X-regions any poly-A fragment close to them was also removed. A poly-A fragment was considered to be any region that scored at least 8 when aligned with a probe sequence of As (adenines) only. The scoring scheme added 1 for a match and -2 for a mismatch, gaps were given a high penalty (-8) because they should not occur. The poly-A had to be at most 10 bases away from X-regions. Alignments were performed using the SWAT program. Depending on the reading direction a poly-A can be read as poly-T, so a poly-T probe was used as well. The removal of X-regions discarded 7,780 sequences.

The next step was quality-trimming, for which a window of 20 bases was slid over every sequence in the set. Starting at the 3' end, the window was slid one base at a time, dropping the extreme base until 12 or less bases in the window had a quality value below 10, the process being repeated for the 5' end. After quality-trimming, X-regions not further than 10 bases away from an end were removed. Quality-trimming removed 1,708 sequences from the set.

The quality-trimming thresholds were chosen as follows. A subset of 10,000 SUCEST sequences was randomly selected on the basis of (i) high similarity (BLASTX e-value below $10^{-20}$) with protein sequences in the NCBI nr database (www.ncbi.nlm.nih.gov), (ii) the length of the matching nr sequence was enough to cover the EST and (iii) the region of similarity did not extend to the end of the EST. By using these criteria we had matches showing a region of similarity that could, potentially, extend to the end of an EST. Cases where the region of similarity did not extend to the end of the EST may have been due to the low quality of the EST sequence.

The exact point where the region of similarity ended, the 'BLAST hit end' (BHE), was recorded for each EST in the set and then the set went through the quality-trimming procedure with varying values for the length of the window, quality threshold and number of bases below threshold. Obviously, high quality thresholds and low numbers of bases produced shorter reads. The difference between the trimmed position (TP) and the BHE (TP-BHE) was calculated and averaged. The results for a 20-base quality window are shown in Figure 6.2. The square in the figure indicates the selected threshold values and shows that, on average, 43 bases after the BLAST hit end were kept.

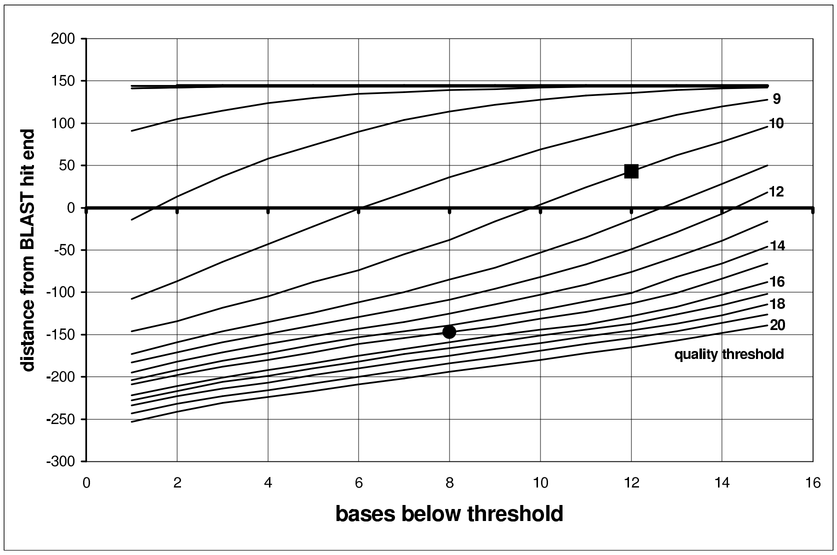



Figure 6.2: Distribution of the number of bases kept at the 3' end with a quality window size of 20, with respect to the best BLAST hits against nr (see text). The square and the bullet indicate the values used in the new and old trimming procedures, respectively.

The next step was slippage-trimming, slippage being a sequencing artifact [4] which produces 'echoed' bases in sequences, i.e. for one occurrence of a nucleotide in the template the chromatogram shows several peaks (q.v. Figure 6.3). Although bases sometimes appeared with high 'background noise' (e.g. bases 215-230), generally the intensity of the echoed peak was such that the base caller incorrectly assigned a high quality value for the fake bases (e.g. bases 175-205) and this prevented quality-trimming of these artifacts.

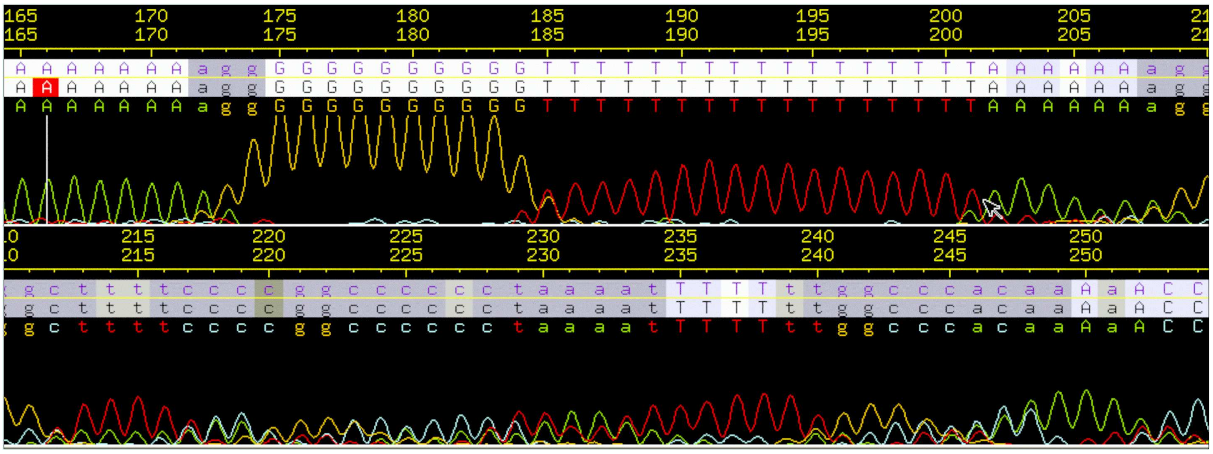



Figure 6.3: Consed (www.phrap.org) trace window of a slipped read. The background of the base letters indicates their phred quality. Darker colors correspond to lower qualities. The numbers above the letters show their position in the read.

A method to identify slipped reads based on the sequence of the read was devised, this method being able to find reads having many regions with repetitive bases (echoed regions). The product of echoed regions lengths (with at least 5 bases) was evaluated for each sequence. Echoed regions larger than 10 bases contributed 10 to the product only. Sequences with a product greater than $10^8$ and echoed regions covering more than 20% of its length were discarded completely. This was the procedure adopted in most cases when slippage was caused by a long poly-A sequence at the 5' end of the read. But when a long poly-A at the 3' end was increasing the product only the poly-A (together with the remaining 3' sequence) was discarded. The threshold for poly-A identification in this situation was an alignment with a score of at least 160. These thresholds were determined by varying the parameters for echoed region recognition, evaluating the products, and looking at several chromatograms in many product ranges. Slippage-trimming removed 15,621 reads.

The next step in the trimming procedure was another poly-A/T removal round, where poly-A/T scoring at 280 and over was removed from sequences. Smaller poly-A/T, scoring at least 30 and less than 20 bases away from one of the ends was also removed. This step removed 2,006 reads.

The final step was to remove any read with less than 100 bases or with less than 50 bases having phred quality greater or equal to 20. A total of 18,147 reads fell in this case.

At the end of all the steps described above, 237,954 reads were left with an average length of 641.6 ±139.8 bases (152.5 Mbp in total). The average number of bases with a phred quality greater or equal to 20 per read was 397.8 ±120.1.

In contrast, the trimming method formerly used in the SUCEST project was simpler. That method started with only one round of very restricted poly-A removal, searching for 12 or more consecutive As adjacent to the vector. The final step was quality-trimming using the same scheme as above with a window length of 20, quality equal to 15 and the number of bases equal to 8. For the reads used in the quality window experiment this combination of thresholds discarded 137 bases from the reads on average (relative to the BHE) as shown in Figure 6.2. This method applied to the original set of SUCEST reads resulted in 261,609 reads with average length of 512.1 ±114.8 bases. The average number of bases with a phred quality greater or equal to 20 per read was 392.4 ±128.3.

BLAST was used to compare the ESTs from the original set of reads in the SUCEST database with the genomes of *Xylella fastidiosa*, *Xanthomonas citri*, *Escherichia coli* and other potential laboratory contaminants that could have been present in the libraries. A match of at least 100 bases and more than 90% identity resulted in the read being marked as probably resulting from contamination. A total of 114 ESTs were thus marked. Because there were so few matches, and the difficulty of deciding whether or not marked ESTs really were the result of contamination, these ESTs were not removed by either of the trimming procedures.

### 6.2.2 Clustering

For the SUCEST project it was necessary to estimate the redundancy of the clone libraries as they were sequenced, which could be achieved by joining similar transcripts into clusters. Clustering results allowed project coordinators to decide when to stop sequencing any particular library.

Fragment assemblers were used for clustering. A fragment assembler is a program that takes a set of reads and their qualities as input, builds groups based on the overlaps of reads and creates a consensus sequence for the reads in each group.

Reads processed by the old trimmer were assembled using the phrap program(version 0.990319, www.phrap.org) with the arguments set to predetermined values (penalty -15, bandwidth 14, minscore 100, shatter_greedy) which made it more stringent and with quality data. This assembly, called 'old-trim', produced 81,223 clusters (41,582 singletons).

To cluster the reads trimmed by the new procedure, three different assemblies were performed and compared. Phrap was used with two sets of arguments, the default arguments (phrap-d assembly) and the more stringent arguments listed above (phrap-hs assembly). The CAP3 program was used with its default arguments. Quality data was

| cluster size | phrap-hs | X | phrap-d | X | CAP3 | X phrap-hs | common | old-trim |
|---|---|---|---|---|---|---|---|---|
| 1 | 32202 | 13731 | 18535 | 11634 | 16838 | 14296 | 10744 | 41582 |
| 2 | 12440 | 5617 | 9207 | 4869 | 7665 | 4852 | 3792 | 13619 |
| 3 | 6752 | 2402 | 5192 | 2151 | 4193 | 1984 | 1441 | 7421 |
| 4 | 4225 | 1239 | 3329 | 1145 | 2709 | 992 | 697 | 4482 |
| 5 | 2856 | 676 | 2360 | 700 | 1872 | 521 | 344 | 3110 |
| 6 | 2098 | 442 | 1806 | 482 | 1452 | 354 | 231 | 2151 |
| 7 | 1582 | 288 | 1362 | 317 | 1115 | 220 | 144 | 1582 |
| 8 | 1245 | 202 | 1091 | 242 | 862 | 153 | 99 | 1219 |
| 9 | 974 | 156 | 913 | 186 | 720 | 113 | 72 | 964 |
| 10 | 776 | 105 | 752 | 143 | 634 | 74 | 44 | 809 |
| 11 | 639 | 76 | 607 | 99 | 511 | 54 | 30 | 641 |
| 12 | 492 | 71 | 547 | 99 | 429 | 46 | 32 | 490 |
| 13 | 437 | 47 | 454 | 90 | 400 | 40 | 25 | 430 |
| 14 | 366 | 42 | 391 | 40 | 341 | 26 | 13 | 366 |
| 15 | 306 | 31 | 390 | 50 | 295 | 18 | 11 | 312 |
| 16 | 273 | 25 | 279 | 35 | 275 | 18 | 8 | 257 |
| 17 | 225 | 15 | 273 | 23 | 235 | 11 | 4 | 206 |
| 18 | 177 | 11 | 227 | 15 | 191 | 5 | 2 | 183 |
| 19 | 124 | 6 | 177 | 18 | 176 | 5 | 3 | 153 |
| 20 | 143 | 10 | 149 | 13 | 179 | 6 | 3 | 136 |
| 21 | 113 | 6 | 130 | 5 | 133 | 2 | 0 | 106 |
| 22 | 105 | 3 | 130 | 5 | 117 | 2 | 1 | 98 |
| 23 | 92 | 4 | 100 | 9 | 140 | 3 | 2 | 79 |
| 24 | 80 | 4 | 99 | 6 | 122 | 2 | 1 | 82 |
| 25 | 69 | 3 | 109 | 6 | 86 | 5 | 2 | 60 |
| 26 | 56 | 2 | 108 | 9 | 72 | 1 | 1 | 59 |
| 27 | 51 | 2 | 59 | 4 | 78 | 1 | 1 | 49 |
| 28 | 44 | 1 | 73 | 5 | 74 | 1 | 1 | 46 |
| >28 | 439 | 5 | 857 | 25 | 1227 | 0 | 0 | 577 |
| total | 69381 | 25222 | 49706 | 22425 | 43141 | 23805 | 17748 | 81223 |

Table 6.1: Cluster sizes distribution for CAP3, phrap-d and phrap-hs assemblies by the new trimming procedure. The 'X' columns indicate the number of equal clusters between two assemblies, while the 'common' column shows the number of clusters equal in the three assemblies. The number of clusters obtained with the original trimming procedure are shown in the 'Old-trim' column. Cluster sizes represent the number of expressed sequence tags (ESTs) in a cluster.

used for every assembly. Table 1 shows the cluster size (number of ESTs in a cluster) distribution for the assemblies, as well as the number of equal clusters between them. Equal clusters are those with the same reads.

Two tests were performed for the assemblies. The first verified 'internal consistency' by checking every cluster with two or more reads for discrepant reads. To be discrepant, a read base must both disagree with the consensus base and have less than a 2% probability of being miscalled by the phred program. An x% discrepant read is a read with at least x% discrepant bases. Figure 6.4 shows the proportion of x% discrepant reads in each assembly, for values of x varying from 30 to 90 in steps of 10.

The second test verified the 'external consistency' of the assemblies by comparing the consensi produced by a given assembly to each other using BLAST. Percentage identity

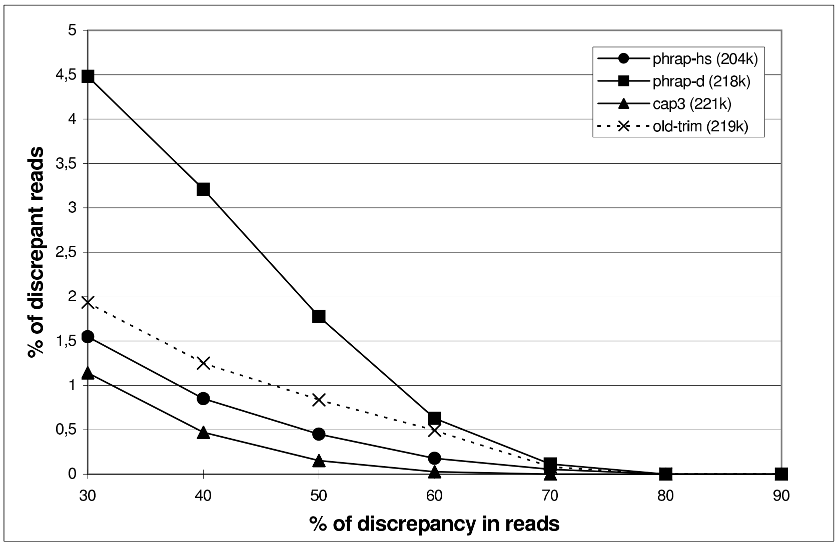



Figure 6.4: Distribution of discrepant reads among the assemblies. As discrepant reads can only be calculated for clusters of two or more reads the number of reads belonging to such clusters in each assembly is shown in parentheses in the legend.

was evaluated for end-overlaps of 200 or more bases found between two clusters, and Figure 6.5 shows a plot of the percentage of clusters having an identity of more than 75% with other clusters in a given assembly with respect to the total of possible overlaps within that set of clusters.

## 6.3 Discussion

The trimming procedure described in this paper discarded 53,735 SUCEST reads, 18.4% of the total. In spite of this large number, it is worth noting that 16% of the discarded sequences were ribosomal RNA and 34% were smaller than 100 bases. We cannot exclude the possibility of that we have discarded useful reads with this procedure, but we tried to avoid this as much as possible. It is also obvious that not every artifact has been removed. For example, counting how many reads have a sub-sequence of at least 30 consecutive adenines in the output of the trimming procedure found 711 reads. Moreover, trimming is not a light computational task, taking 8.3 hours to process all the SUCEST reads.

Nevertheless, the influence of the quality of trimming on the final clustering is remarkable. For instance, it is hard to accept that the number of singletons in the old assembly are uniquely expressed sugarcane genes, and 81,223 was an unreasonably large number of clusters. Good trimming also shortened the CPU time required for clustering, the phrap program took 9.2 hours to build the phrap-d assembly and 6.5 hours for

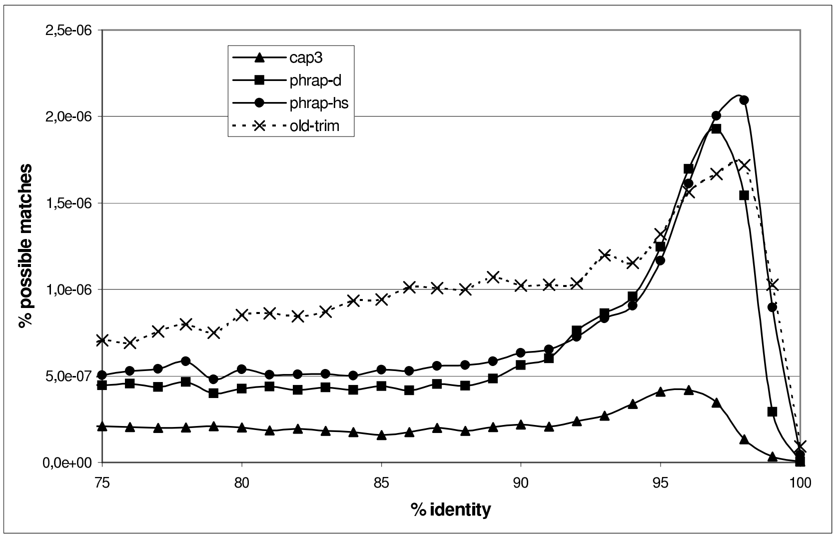



Figure 6.5: Plot of external consistency test results. For a given assembly with n clusters the number of overlaps detected was divided by $n(n-1)/2$, which is the maximum number of possible overlaps for n clusters.

phrap-hs assembly, while the CAP3 program took 77.1 hours. To assemble the old set of trimmed reads, phrap took 5 times more time than it spent to produce phrap-hs, while CAP3 ended abnormally when fed with that data-set.

We have used a fragment assembler for the whole set of ESTs in the SUCEST database and, consequently, the biological definition of 'one cluster, one gene' cannot be used. A SUCEST cluster can be better defined as 'a set of very similar transcripts'.

Building consensus sequences for clusters is useful in several respects. Firstly, electing a representative sequence for each cluster results in a smaller set of sequences to work with. Secondly, the portions of representative sequences covered by more than one read are more accurate than the reads themselves. Thirdly, representative sequences may be longer than individual reads, increasing their usefulness. This third point was confirmed by the fact that 33% of representative sequences with homologous genes in other organisms were actually full-length sequences [38].

However, chimeras may result from assembling ESTs and a further problem is that using a fragment assembler for clustering will put alternatively spliced forms of genes into different clusters. But in a dodecaploid organism like sugarcane it is especially difficult to distinguish alleles of genes from very conserved multigene families based on similarity.

The assembly produced by the CAP3 program was taken as the 'official' clustering for the SUCEST project. This decision was based on the result of the internal and external consistency tests, where the CAP3 assembly outperformed both the phrap-hs and phrap-d assemblies. Internal consistency shows that the CAP3 assembly has a lower incidence of discrepant reads in clusters when compared to the other assemblies. External consistency reveals that the CAP3 program produces fewer redundant clusters, i.e. two or more clusters that probably should be condensed to a single cluster. Unfortunately, we performed no comparisons of our results with those that would be produced using some other method described in the literature. This is an interesting investigation to perform in the future.

The trimming and clustering procedures described in this paper hide a large amount of computational time and human work spent looking at the data, testing insights, adjusting parameters, and designing the pipeline. There are no 'magic numbers'. We believe that these guidelines may be used in some other EST projects, although using these procedures with different data sets may require some adjustments. The need for many cycles of adjustment and testing is a natural consequence of the nature of the noise present in ESTs, the limitations posed by technological issues and the lack of a complete understanding of the biological processes occurring within cells.

## 6.4 Acknowledgments

This work was supported by the Brazilian agencies FAPESP and CNPq.

# Capítulo 7

# Conclusões

Nesta tese apresentamos nossas contribuições na forma de duas ferramentas que podem ser usadas em projetos genoma.

Apresentamos um algoritmo quase-linear para construir árvores PQR. Nosso algoritmo não é uma simples extensão do de Booth e Leuker: ele está fundamentado em propriedades teóricas sólidas das árvores e usa um conjunto menor de padrões, melhor organizados que os padrões das árvores PQ. A prova de correção também está fundamentada na teoria e a nossa análise amortizada usa uma função potencial simples. Acreditamos que nosso algoritmo contribui significativamente em direção a uma solução definitiva para o problema dos uns consecutivos.

Apresentamos também um esquema para o *clustering* de ESTs de cana-de-açúcar. O esquema inclui uma limpeza inicial intensiva dos ESTs, que permitiu usar um montador de genomas para todo o conjunto de seqüências. Cada grupo que foi gerado pelo nosso esquema tem uma seqüência consenso e potencialmente representa um transcrito. Acreditamos que esse esquema possa ser usado em outros projetos do mesmo tipo, depois de um ajuste adequado dos parâmetros.

# Bibliografia


[1] M.D. Adams, J.M. Kelley, J.D. Gocayne, et al. Complementary DNA sequencing: expressed sequence tags and the human genome project. *Science*, 252:1651–1656, 1991.

[2] S.F. Altschul, T.L. Madden, A.A. Schäffer, et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[3] M.R. Anderberg. *Cluster analysis for applications*. Academic Press, 1973.

[4] Applied Biosystems. *Chemistry Guide for Automated DNA Sequencing*, 1998.

[5] M.S. Boguski and G.D. Schuler. ESTablishing a human transcript map. *Nature Genetics*, 10:369–371, 1995.

[6] 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 and System Sciences*, 13(3):335–379, 1976.

[7] J. Burke, D. Davison, and W. Hide. d2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9:1135–1142, 1999.

[8] J. Burke, H. Wang, W. Hide, and D.B. Davison. Alternative gene form discovery and candidate gene selection from gene indexing projects. *Genome Research*, 8(8):276–290, 1998.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. McGraw-Hill, Inc., 1991.

[10] K.P. Eswaran. Faithful representation of a family of sets by a set of intervals. *SIAM Journal On Computing*, 4(1):56–68, 1975.

[11] B.S. Everitt. *Cluster Analysis*. Edward Arnold, 3rd edition, 1993.

[12] B. Ewing and P. Green. Base-calling of automated sequencer traces using Phred. ii. error probabilities. *Genome Research*, 8:186–194, 1998.

[13] A.G. Ferreira and S.W. Song. Achieving optimality for gate matrix layout and PLA folding: a graph theoretic approach. In I. Simon, editor, *Latin'92*, volume 583 of *Lecture Notes in Computer Science*, pages 139–153, São Paulo, Brasil, 1992. Springer-Verlag.

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

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

[16] D. Gilbert, M. Schroeder, and J. van Helden. Interactive visualisation and exploration of biological data. In *Proc. of Intern. Workshop on Biomolecular Informatics*, 2000.

[17] H. Heeb and W. Fichtner. A module generator based on the PQ-tree algorithm. *IEEE Transactions on Computer-Aided Design*, 11(7):876–884, 1992.

[18] X. Huang and A. Madan. CAP3: a DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[19] A.K. Jain and R.C. Dubes. *Algorithms for clustering data.* Prentice Hall, 1988.

[20] B. Lewin. *Genes VI.* Oxford, 1997.

[21] F. Liang, I. Holt, G. Pertea, S. Karamycheva, et al. An optimized protocol for analysis of EST sequences. *Nucleic Acids Research*, 28(18):3657–3665, 2000.

[22] G.S. Lueker and K.S. Booth. A linear time algorithm for deciding interval graph isomorphism. *Journal of the ACM*, 26(2):183–195, 1979.

[23] J. Meidanis and E.G. Munuera. A theory for the consecutive ones property. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proc. of the III South American Workshop on String Processing*, pages 194–202, Recife, Brasil, 1996.

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

[25] R.T. Miller, A.G. Christoffels, C. Gopalakrishnan, et al. A comprehensive approach to clustering of expressed human gene sequence. *Genome Research*, 9:1143–1155, 1999.

[26] D.L. Nelson and M.M. Cox. *Lehninger Principles of Biochemistry.* Macmillan Press/Worth Publishers, 3 edition, 2000.

[27] J.D. Parsons and P. Rodriguez-Tome. JESAM: CORBA software components to create and publish EST alignments and clusters. *Bioinformatics*, 4(16):313–325, 2000.

[28] W.R. Pearson, T. Wood, Z. Zhang, and W. Miller. Comparison of DNA sequences with protein sequences. *Genomics*, 46:24–36, 1997.

[29] Documentation for phrap and cross-match. http://bozeman.mbt.washington.edu/phrap.docs/phrap.html.

[30] J. Quackenbush, F. Liang, I. Holt, et al. The TIGR Gene Indices. *Nucleic Acids Research*, 28(1):141–145, 2000.

[31] J.C. Setubal and J. Meidanis. *Introduction to computational molecular biology.* PWS Publishing Co., 1997.

[32] P.H.A. Sneath and R.R. Sokal. *Numerical Taxonomy.* W.H. Freeman and Company, 1973.

[33] G.G. Sutton, O. White, M.D. Adams, and A.R. Kerlavage. TIGR assembler: a new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19, 1995.

[34] G.P. Telles. Propriedade dos uns consecutivos e árvores PQR. Master's thesis, Instituto de Computação – Universidade Estadual de Campinas, Campinas-SP-Brasil, 1997.

[35] G.P. Telles, M.D.V. Braga, Z. Dias, et al. Bioinformatics of the Sugarcane EST Project. *Genetics and Molecular Biology*, 24(1–4):9–15, 2001.

[36] The Phred/Phrap/Consed System Home Page. www.phrap.org.

[37] A. Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory*, 12(B):153–162, 1972.

[38] A.L. Vettore, F.R. da Silva, E.L. Kemper, and P. Arruda. The libraries that made SUCEST, the Sugarcane EST Project. *GMB*, 406:151–157, 2001.

[39] A.L. Vettore, F.R. da Silva, E.L. Kemper, et al. Analysis and functional annotation of an expressed sequence tag collection for tropical crop sugarcane. Submitted., 2002.

[40] D. Voet and J.G. Voet. *Biochemistry.* John Wiley & Sons, 2nd edition, 1995.

[41] K.Y. Yeung, D.R. Haynor, and W.L. Ruzzo. Validating clustering for gene expression data. Technical Report UW-CSE-00-01-01, University of Washington, Department of Computer Science and Engeneering, 2000.