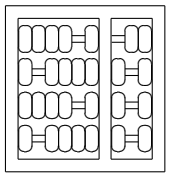


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Construção incremental de árvores PQR

J. P. P. Zanetti *J. Meidanis*

Technical Report - IC-10-31 - Relatório Técnico

October - 2010 - Outubro

Construção incremental de árvores PQR

João Paulo Pereira Zanetti* João Meidanis†

Resumo

As árvores PQ foram introduzidas por Booth e Lueker em 1976 [2]. Uma árvore PQ é uma estrutura de dados que representa permutações de um conjunto de elementos em que certos subconjuntos ocorrem consecutivamente. Aplicações incluem reconhecimento de grafos de intervalos, de grafos planares, e problemas envolvendo moléculas de DNA.

Apesar de sua grande popularidade em trabalhos teóricos, as árvores PQ são de difícil implementação. Neste trabalho, damos continuidade ao desenvolvimento de uma estrutura mais geral, a árvore PQR, introduzida por Meidanis, Porto e Telles [7], que tem boas possibilidades de se tornar uma alternativa mais fácil de implementar, além de dar mais informações sobre o problema.

Aqui detalhamos um algoritmo incremental (*online*) para a construção de árvores PQR [9], e provamos a corretude de cada operação.

1 Introdução

Uma estrutura de dados chamada árvore PQR é apresentada aqui como uma ferramenta para a solução de problemas relacionados a encontrar permutações válidas de um conjunto U. Estas permutações válidas são as em que os elementos de certos subconjuntos de U ocorrem consecutivamente (um exemplo será dado no próximo parágrafo). Estes subconjuntos podem ser chamados de **restrições** e uma árvore PQR representa de maneira compacta todas estas restrições dadas, mais todas as que derivam delas, de uma forma que ficará mais clara mais adiante, na Seção 2. Aplicações para este algoritmo geralmente envolvem testar a propriedade dos uns consecutivos em matrizes, por exemplo: reconhecer grafos de intervalos [4] e grafos planares [2], problemas envolvendo moléculas de DNA [8] e arqueologia [6].

Um exemplo é o de um sistema de recuperação de informação [5]. Um banco de dados contém um conjunto U de registros em disco. O sistema responde consultas sobre informações contidas nestes registros. Uma consulta consiste em um subconjunto $C \subset U$ de registros necessários para respondê-la. Este sistema visa ordenar seus registros em disco de forma que, para cada consulta C, todos os seus registros estejam armazenados consecutivamente, o que permite que cada consulta seja respondida com apenas um movimento da cabeça de leitura no disco. Esta estratégia diminui o tempo para a resposta às consultas.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro do CNPq, processo 132182/2010-6, e da FAPESP, processo 2010/04071-1.

†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

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

O problema pode ser descrito como segue: dados um conjunto U, de n elementos, e uma coleção $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ de m subconjuntos de U, o algoritmo deve decidir se existe uma permutação dos elementos de U em que os elementos de cada $C \in \mathcal{C}$ apareçam consecutivamente. Idealmente, o algoritmo determinaria também todas as permutações que satisfaçam estes critérios.

Existem duas maneiras de construir uma árvore PQR. Uma delas, dita *offline*, recebe todas as restrições de uma vez e retorna a árvore final, sem árvores intermediárias. O outro estilo, dito *online* e aqui chamado de incremental, parte de uma árvore universal e vai adicionando restrições a ela, de forma que sempre temos uma solução parcial, relativa às restrições já adicionadas. Doravante, trabalharemos apenas com a abordagem incremental.

Num algoritmo incremental, a cada novo subconjunto C cujos elementos devem aparecer consecutivamente, o número de permutações válidas é reduzido. Por isso, à operação correspondente na árvore PQR é dado o nome de **redução** em relação ao conjunto C. Um algoritmo eficiente para computar a redução de uma árvore PQR é apresentado aqui. Sua complexidade de tempo é praticamente linear (depende da função inversa de Ackermann, que cresce muito lentamente [3, Cap. 21]) em relação ao tamanho da entrada — tamanho de U, número de restrições e a soma de seus tamanhos.

Uma solução geral seria a do Algoritmo 1.

Algoritmo 1: Redução incremental

```
1 Π ← {π|π é uma permutação de U}
2 para cada C ∈ 𝒞 faça
3     Π ← Π ∩ {π|todos os elementos de C são consecutivos em π}
4 retorne Π
```

Este algoritmo é de fácil implementação se cada conjunto for apenas uma lista de seus elementos. Entretanto, nesta implementação inocente, o laço interno seria extremamente ineficiente, pois, a cada restrição, todas as permutações válidas até então que não a satisfaçam devem ser desconsideradas. Este trabalho detalha mecanismos que permitem executar este laço eficientemente.

A chave para o sucesso do algoritmo é o uso de uma estrutura de dados que permita representar toda a classe Π em um espaço reduzido, mas que ainda mantenha informação necessária para o processamento. A árvore PQR visa cumprir este papel. Ela é uma generalização da árvore PQ de Booth e Lueker, porém, enquanto para entradas para as quais não existe permutação válida não existe uma árvore PQ, existe uma árvore PQR para qualquer entrada, indicando a incompatibilidade. Além disso, as árvores PQR podem ser de implementação mais fácil que as árvores PQ, pois baseiam-se em cinco regras de transformação ao invés das mais de uma dezena de padrões de substituição para árvores

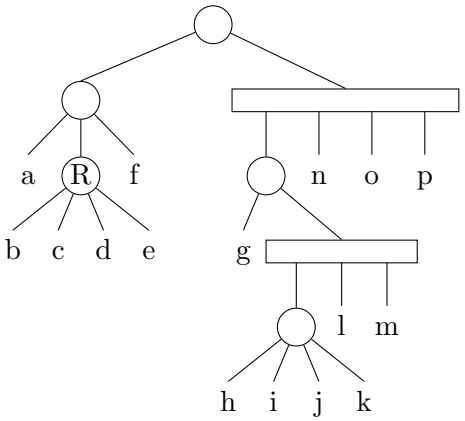


Figura 1: Exemplo de árvore PQR sobre $U = abcdefghijklmnop$ e com restrições $\mathcal{C} = \{abcdef, bc, cd, de, ce, ghijklmn, no, op, hijkl, lm\}$.

PQ [2].

2 Definições

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

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

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

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

Em geral representamos árvores PQR indicando o tipo do nó pelo seu formato: nó P por um círculo, nó Q por um retângulo e nó R por um círculo com a letra R. Um exemplo de árvore PQR pode ser visto na Figura 1.

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

Dizemos que dois conjuntos A e B são **ortogonais**, denotado por $A \perp B$, quando $A \subseteq B$, $B \subseteq A$, ou $A \cup B = \emptyset$.

A **fronteira** de uma árvore PQR é a permutação de U obtida lendo as folhas da árvore da esquerda para a direita.

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

A **união não-disjunta** de dois conjuntos A e B, denotada por $A \uplus B$, é igual a $A \cup B$, dado que $A \cap B \neq \emptyset$. A **diferença não-contida** de dois conjuntos A e B, denotada por $A \not\setminus B$, é igual a $A \setminus B$, dado que $B \not\subseteq A$. Estas operações, assim como a interseção de conjuntos, são sempre aplicadas da esquerda para a direita, a não ser que a prioridade seja alterada com o uso de parênteses.

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

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

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

- $\mathit{Trivial}(v)$, que contém o conjunto vazio, o conjunto $\hat{v}$ e todos os conjuntos $\{x\}$ tais que $x \in \hat{v}$.
- $\mathit{Consec}(v)$, que contém todos os conjuntos da forma $\bigcup_{v \in S} \hat{v}$ tais que S é um conjunto de filhos consecutivos de v.
- $\mathit{Pot}(v)$, que contém todos os conjuntos da forma $\bigcup_{v \in S} \hat{v}$ tais que S é um conjunto arbitrário de filhos de v.

Com estas definições, definimos o fecho $\overline{v}$ de um nó v como na Equação (1) e, então, definimos o fecho $\overline{T}$ de uma árvore PQR T como na Equação (2).

$$\overline{v} = \begin{cases} \mathit{Trivial}(v) & \text{se } v \text{ é do tipo P,} \\ \mathit{Trivial}(v) \cup \mathit{Consec}(v) & \text{se } v \text{ é do tipo Q,} \\ \mathit{Trivial}(v) \cup \mathit{Pot}(v) & \text{se } v \text{ é do tipo R.} \end{cases} \tag{1}$$

$$\overline{T} = \bigcup_{v \in T} \overline{v} \tag{2}$$

É importante notar que $\overline{T}$ é o que era chamado de $\mathit{Compl}(T)$ no artigo de Meidanis, Porto e Telles [7], apesar de ser definido de forma diferente.

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

É fácil verificar a seguinte importante propriedade dos fechos de nós equivalentes:

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

3 Algoritmo incremental

O algoritmo apresentado por Telles e Meidanis [9], partes do qual são reproduzidas a seguir, nos Algoritmos 2 a 10, descreve em alto nível as operações necessárias para a construção de uma árvore PQR. O Algoritmo 2 é um algoritmo incremental, que começa com uma árvore universal e adiciona a esta os conjuntos de $\mathcal{C}$ um a um, através do Algoritmo 3.

Um nó é **negro** quando $\hat{v} \subset C$, **cinza** quando $\hat{v} \not\perp C$ e **branco** nos outros casos, isto é, quando $\hat{v} \cap C = \emptyset$ ou $C \subseteq \hat{v}$. Estas cores são representadas pelos números 0, 1 e 2, correspondentes às cores branca, cinza e negra, respectivamente. Os conjuntos de filhos negros, cinzas e brancos de um dado nó v são chamados de $B(v)$, $G(v)$ e $W(v)$, respectivamente.

Lembramos que um dos resultados da teoria desenvolvida por Telles e Meidanis é que os nós de uma árvore PQR correspondem a conjuntos ortogonais à coleção inicial.

Portanto, temos que os nós negros são nós que já são compatíveis com a nova restrição e os nós brancos são nós que não são afetados por ela. Assim, chegamos ao objetivo do algoritmo, que é reparar os nós cinzas, seja eliminando-os, seja modificando-os de forma a torná-los brancos. Isto é feito usando as operações que serão descritas nas próximas seções.

Algoritmo 2: Redução de árvores PQR

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

Algoritmo 3: Algoritmo para adicionar um conjunto C a uma árvore T.

```
1 Colorir as folhas correspondentes a C
2 Colorir a árvore e encontrar a raiz da subárvore pertinente
3 Reestruturar a árvore, reparando nós cinzas (Algoritmo 4)
4 Ajustar a raiz
5 Descolorir a árvore
```

Algoritmo 4: Algoritmo para reestruturar a árvore, reparando nós cinzas. (Passo 3 do Algoritmo 3)

```
1 enquanto existe um filho cinza v da raiz r faça
2     preparar_raiz(r)
3     preparar_filho(r, v)
4     se r é do tipo P então
5         mover filhos para fora da raiz
6     senão
7         se v é do tipo Q então
8             reverter condicionalmente nó cinza
9         fundir com a raiz
```

Algoritmo 5: `preparar_raiz(`r`)`

```
1 se r é do tipo P então
2     unir filhos negros
```

Algoritmo 6: `preparar_filho(`r`, `v`)`

```
1 se v é do tipo P então
2     transformar nó P em nó Q
```

Nesta seção, serão detalhadas as seguintes operações:

1. Transformar nó P em nó Q
2. Unir filhos negros
3. Reverter condicionalmente nó cinza
4. Fundir com a raiz
5. Mover para fora da raiz

Para a descrição dessas operações, assume-se que um nó seja capaz de realizar as seguintes operações básicas:

- Criar nó
- Destruir nó
- `remover_filho(`p`, `f`)`, que consiste de remover de todas as listas de filhos do nó p um nó filho f e retorná-lo
- `inserir_filho_no_fim(`p`, `f`)`, que insere um nó f no fim da lista de filhos do nó p
- `inserir_filho_após(`p`, `f`, `i`)`, que insere um nó f depois do nó filho i nas listas de filhos do nó p (desnecessária nos nós tipos P e R, pois a ordem dos filhos só é relevante em nós do tipo Q)
- Reverter ordem dos filhos (também só necessária em nós tipo Q)
- Obter tipo (P, Q ou R)
- Alterar tipo
- Obter cor
- Alterar cor

As outras operações necessárias podem ser definidas a partir destas operações básicas.

Assim, por exemplo, mover um nó significa removê-lo de seu nó pai e inseri-lo (no fim ou em uma posição específica) entre os filhos de um outro nó. Quando não especificado o filho após o qual o nó será inserido, este é inserido no fim dos filhos do novo nó pai.

Algoritmo 7: `inserir_filho_no_fim(`*p, f*`)`

```
1  inserir f no fim da lista de filhos de p
2  se p é do tipo P então
3      f.pai ← p
4  se p é do tipo Q ou R então
5      se r não tem filhos então
6          make_set(f)
7          p.representante ← f
8      senão
9          q ← p.representante
10         union(q,f)
```

Destruir um nó, por sua vez, é uma operação que requer mais cuidados, pois os nós, mesmo que removidos da árvore, ainda são necessários para manter a estrutura de union-find usada para descobrir o pai de um nó. Além disso, só se pode destruir um nó que não tenha filhos e que não seja filho de ninguém. Então, a operação de destruição de um nó v tem que incluir a remoção de v de seu nó pai. Já os filhos de v tem que ser movidos antes da destruição do nó, já que eles têm destinos específicos. Mais detalhes sobre a remoção de nós são discutidos na Seção 3.7.

Na implementação destas estruturas, é preciso fazer com que admitam um número variável de filhos. Além disso, os filhos de um nó dos tipos Q ou R são armazenados em uma lista simétrica [1], uma estrutura de dados similar a uma lista duplamente ligada que permite tanto a reversão de uma lista quanto a fusão de duas listas em tempo constante.

Como a árvore é colorida de baixo para cima (das folhas em direção à raiz), cada nó precisa conhecer seu pai. Como manter apontadores para o pai em todos os nós pode ser muito dispendioso, faz-se necessário para a eficiência do algoritmo armazenar esta informação em uma estrutura de union-find [3], onde só um nó representante entre os filhos tem um apontador válido para seu pai e todos os seus nós irmãos estão ligados a este representante através de uma árvore.

Em diversas das operações a seguir, é preciso percorrer todos os filhos cinzas, ou todos os filhos negros, de um nó. Para realizar estes percursos eficientemente, é preciso manter em cada nó, além da lista de todos os seus filhos, também a lista dos filhos cinzas e a lista dos filhos negros. Obviamente, ao mudar a cor de um nó, estas listas têm que ser atualizadas em seu pai.

3.1 Complexidade e corretude

A análise de complexidade se concentra no número de movimentações de nós, pois, conforme o Teorema 5 no trabalho de Telles e Meidanis [9], o número total de operações é dominado

pelo número de movimentações. Nesta análise são utilizados os conjuntos de filhos negros, cinzas e brancos de um nó v, chamados de $B(v)$, $G(v)$ e $W(v)$, respectivamente, conforme definido no início da Seção 3.

Cada operação do algoritmo deve realizar um número de movimentações de nós linear em relação ao número de nós negros e cinzas, mas constante em relação ao número do nós brancos.

Já a corretude dos algoritmos é provada seguindo a técnica apresentada na Seção 5.1 do trabalho de Telles e Meidanis [9]. Esta técnica consiste de provar que, se T' e T'' são as árvores antes e depois cada passo do algoritmo e C é o conjunto sendo adicionado, então

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

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

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

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

Vamos agora às operações.

3.2 Transformar nó P em nó Q

Nós P de cor cinza vão se transformar em nós Q antes de desaparecer ou se tornar brancos, e portanto a ordem de seus filhos torna-se importante. Esta transformação, porém, é ilusória, isto é, parece transformar o nó de P em Q, mas, na prática, cria-se um novo nó Q enquanto o nó P é apenas movido. A transformação de P em Q é ilustrada na Figura 2 e executa-se segundo o Algoritmo 8.

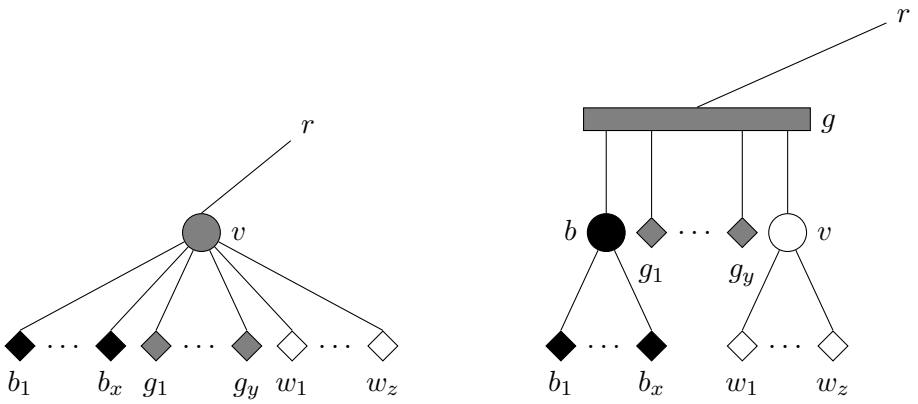


Figura 2: Operação "transformar nó P em nó Q".

Algoritmo 8: Transformar nó P em nó Q

```
1  criar nó cinza g do tipo Q filho de r após v
2  se v tem 2 ou mais filhos negros então
3      criar nó negro b do tipo P filho de g
4      para cada filho negro n de v faça
5          mover n para b
6  senão
7      para cada filho negro n de v faça
8          mover n para g
9  para cada filho cinza c de v faça
10     mover c para g
11 se v tem 2 ou mais filhos então
12     mover v para g
13     pintar v de branco
14 senão
15     se v tem um filho w então
16         mover w para g
17     destruir v
```

Nas linhas de 2 a 8, são movidos os filhos negros de v. Os filhos cinzas são movidos no laço da linha 9.

A partir da linha 11, sabe-se que todos os filhos de v, se houver, são brancos, pois os filhos de outras cores já foram movidos. Assim, os nós brancos não são movidos individualmente. Na verdade, seu pai é movido e com ele vão todos os filhos, em uma única operação.

Para a complexidade, observe que, durante esta operação, são movidos $|B(v)|+|G(v)|+1$ nós.

Nós em T' sem equivalentes em T'': v'. Note que r' é equivalente a r'', visto que $\hat{v}' = \hat{g}''$,

e isto também garante que $\hat{v}' \in \overline{T''}$.

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

Para os dois primeiros, temos

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

e

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

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

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

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

No segundo caso, escrevemos:

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

No terceiro caso, escrevemos:

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

E no quarto caso:

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

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

3.3 Unir filhos negros

Esta operação agrupa todos os filhos negros da raiz sob um único nó do tipo P, como ilustrado na Figura 3. A operação é executada conforme o Algoritmo 9.

Algoritmo 9: Unir filhos negros

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

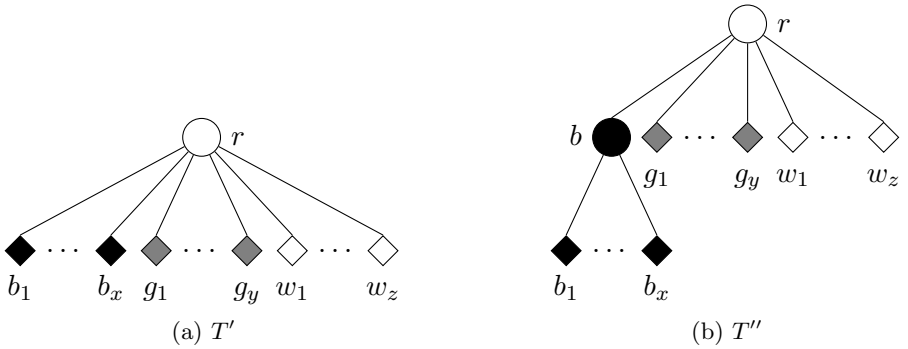


Figura 3: Operação "unir filhos negros".

Neste algoritmo, são movidos somente os filhos negros de r. Logo, são movidos $|B(r)|$ nós.

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

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

3.4 Reverter condicionalmente nó cinza

Para entender a necessidade desta operação, começamos com um lema.

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

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

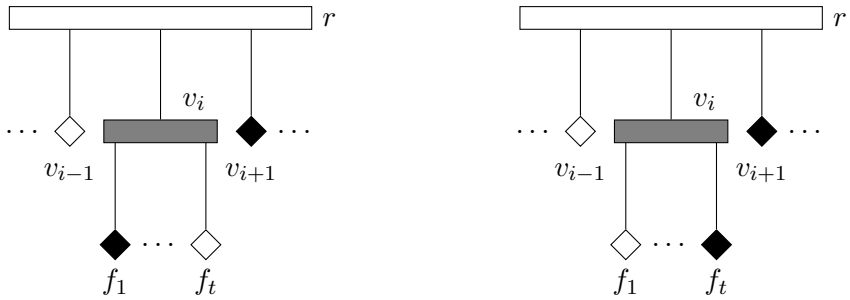
Se v tem um filho branco f_a e um filho negro f_b, temos que $\hat{f}_a \not\subseteq C$ e $\hat{f}_b \cap C \neq \emptyset$, concluindo a prova.

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

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

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

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



(a) T' (b) T''

Figura 4: Operação "reverter condicionalmente nó cinza".

Como esta operação deve ser realizada em tempo constante, a implementamos como segue. Um nó cinza v_i, do tipo Q, é revertido quando as cores de seus filhos extremos f_1 e f_t não estão alinhadas com as cores dos vizinhos de v_i. Se v_i só tem um vizinho, v_i é revertido se seu filho mais próximo do vizinho existente é mais claro que o outro filho extremo de v_i. Uma das situações em que isso ocorre é ilustrada na Figura 4 e todas as condições para a reversão podem ser vistas no Algoritmo 10, lembrando que as cores têm as seguintes representações numéricas: 0=branco, 1=cinza e 2=negro.

Algoritmo 10: Reverter condicionalmente nó cinza

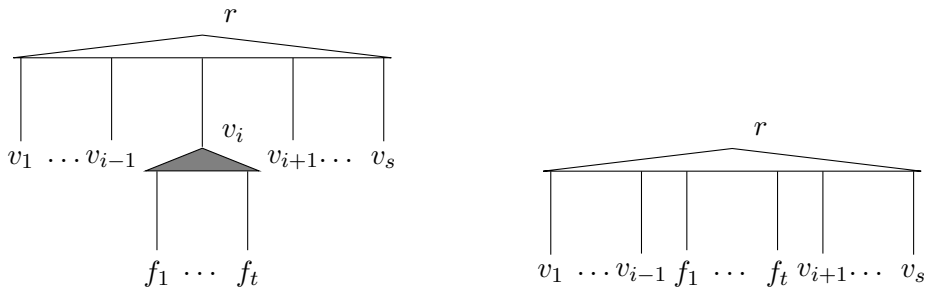
```
1 se v_{i-1} não existe e f_1.cor > f_t.cor então
2     reverta v
3 senão se v_{i+1} não existe e f_1.cor < f_t.cor então
4     reverta v
5 senão se (v_{i-1}.cor − v_{i+1}.cor)(f_1.cor − f_t.cor) < 0 então
6     reverta v
```

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

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

O nó v_i também não é revertido se v_{i-1} e v_{i+1} têm a mesma cor. Se v_{i-1} e v_{i+1} são ambos negros ou cinzas, qualquer um deles pode ser escolhido como v_d e, portanto, qualquer orientação de v_i é satisfatória. Se v_{i-1} e v_{i+1} são brancos, a árvore já está comprometida e não será possível manter juntos todos os nós escuros, então não adianta reverter. Este é o único caso em que a árvore após a execução do Algoritmo 10 não necessariamente satisfaz $(a - b)(i - d) \geq 0$.

Portanto, depois desta operação, com certeza existem nós f_a, f_b e v_d como descritos tais



(a) T' (b) T''

Figura 5: Operação "fundir com a raiz".

que $(a - b)(i - d) \geq 0$ a não ser que v_{i-1} e v_{i+1} sejam brancos. Estes nós serão usados na próxima operação, fundir com a raiz.

Nenhum nó é movido e um número constante de operações é realizado neste algoritmo, todas em tempo constante; portanto, a complexidade assintótica do algoritmo é constante. Consideraremos então que esta operação corresponde a uma movimentação de nó.

Como a única transformação feita nesta operação é uma transformação de equivalência, temos

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

e portanto

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

3.5 Fundir com a raiz

Antes desta operação, os nós r e v_i são do tipo R ou Q.

Algoritmo 11: Fundir com a raiz

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

Aqui, são movidos todos os filhos de v_i, mas como as listas simétricas podem ser fundidas em $O(1)$, a operação é realizada em tempo constante. Consideraremos portanto que esta operação corresponde a uma movimentação de nó.

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

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

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

Aqui precisamos definir mais dois conjuntos pertencentes a $\overline{T'}$ que serão úteis:

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

e

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

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

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

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

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

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

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

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

Vamos agora à parte principal da prova. Se v_{i-1} e v_{i+1} não são ambos brancos, a operação de reverter condicionalmente nó cinza orientou o nó v_i de forma que temos nós f_a e f_b, filhos de v_i, e v_d, irmão de v_i, com f_a não negro, f_b e v_d não brancos e f_b mais próximo de v_d que f_a (veja a Figura 4).

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

$$F_{[b,t]} \cup V_{[i+1,d]} = C \cap V_{[i,d]} \oint F_{[1,b-1]} \uplus V_{[i+1,d]} \uplus F_{[b,t]}. \tag{3}$$

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

$$F_{[1,b]} \cup V_{[d,i-1]} = C \cap V_{[d,i]} \oint F_{[b+1,t]} \uplus V_{[d,i+1]} \uplus F_{[1,b]}. \tag{4}$$

Se v_{i-1} e v_{i+1} são brancos, mas f_1 e f_t têm cores iguais, as fórmulas (3) e (4) continuam válidas, com escolhas aproriadas de f_a e f_b, conforme descrito na seção anterior. Portanto, só precisamos nos preocupar com o caso em que v_{i-1} e v_{i+1} são brancos e f_1 e f_t têm cores diferentes. Se f_1 é o mais claro dos dois, ou seja, f_1 não é negro, temos:

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

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

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

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

Se v_i' é do tipo R, temos

$$\hat{f}_1 \cup \hat{f}_t \in \overline{\overline{T'} \cup C}$$

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

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

$$Pot(r'') \subseteq \overline{\overline{T'} \cup C}$$

Já se r' é do tipo R, temos

$$\hat{v}_1 \cup \hat{v}_s \in \overline{\overline{T'} \cup C}$$

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

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

$$Pot(r'') \subseteq \overline{\overline{T'} \cup C}$$

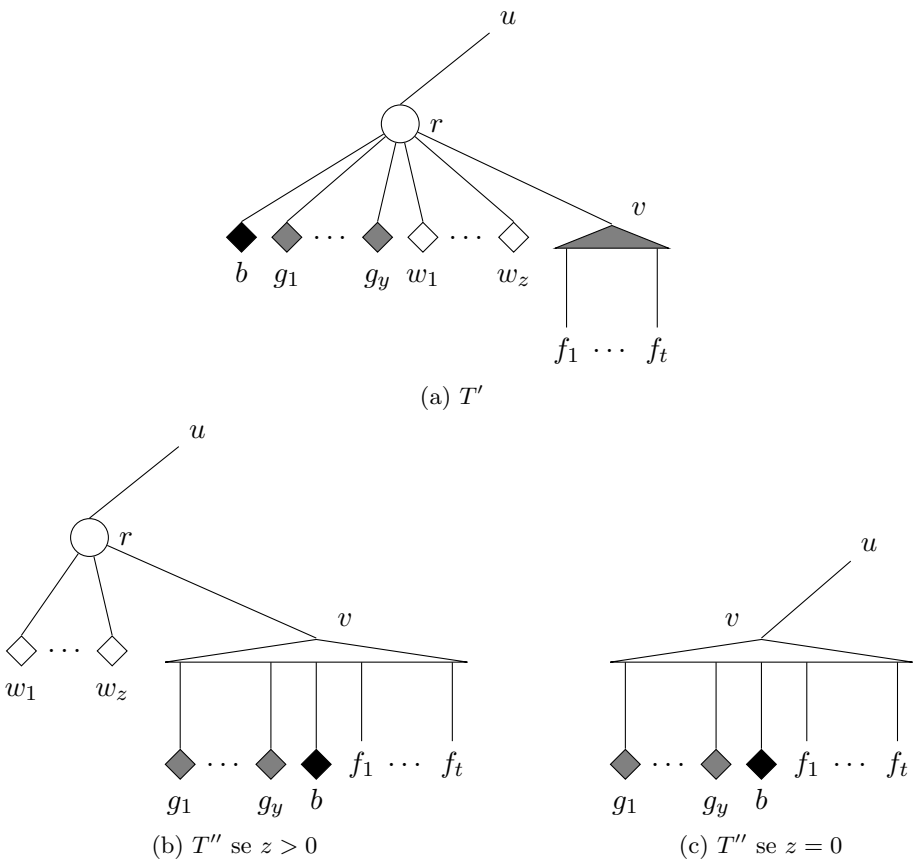
3.6 Mover para fora da raiz

Esta operação move os filhos negros e cinzas de r, do tipo P, para seu filho v, do tipo Q ou R.

Algoritmo 12: Mover para fora da raiz

```
1  se f1.cor < ft.cor então
2      reverter v
3  se r tem um filho negro b então
4      mover b para o início de v
5  para cada filho cinza c de r diferente de v faça
6      mover c para o início de v
7  se r tem apenas um filho então
8      se v é a raiz da árvore então
9          r torna-se a raiz da árvore
10     senão
11         u ← r.pai
12         mover v para u
13     pintar v de branco
14     destruir r
```

O primeiro comando condicional busca permitir que os nós escuros fiquem juntos. O segundo comando condicional testa por apenas um filho negro porque o nó r já passou pela operação "unir filhos negros". Assim, nas linhas 3 a 6, todos os filhos não-brancos de r são movidos para v. Note que, na linha 7, se r tem apenas um filho, este único filho é o nó v e, portanto, r deve ser removido.



(a) T'

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

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

Figura 6: Operação "mover para fora da raiz". A situação anterior à operação é ilustrada em (a), enquanto (b) e (c) são as duas possibilidades para a situação posterior.

Nesta operação, são movidos no máximo $|G(r)| + 1$ nós.

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

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

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

Nós em T'' sem equivalentes em T': r'' e v''. Entretanto, r'' existe apenas quando $z > 0$ e, neste caso, temos $\hat{r}'' = \hat{r}'$. Em relação ao nó v'', os conjuntos em $Consec(v'') \setminus Consec(v')$ são os de forma $\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1 \cup \ldots \cup \hat{f}_t$, com $1 \leq i \leq y + 1$. Estes conjuntos podem ser escritos como

$$\hat{g}_i \cup \ldots \cup \hat{g}_y \cup \hat{b} \cup \hat{f}_1 \cup \ldots \cup \hat{f}_t = (\hat{v}' \uplus C) \diamond \hat{g}_1 \diamond \ldots \diamond \hat{g}_{i-1} \uplus \hat{g}_i \ldots \uplus \hat{g}_y$$

3.7 Remoção virtual de nós

Os nós de árvore PQR podem ser removidos durante a execução do algoritmo. Porém, eles ainda são necessários na estrutura de union-find utilizada para encontrar o pai de um nó. Esses nós são removidos da árvore e marcados como tal, mas são mantidos na union-find.

Portanto, a estrutura de union-find pode conter não apenas nós atuais da árvore, mas também outros nós criados durante a execução do algoritmo. Isto poderia fazer com que a complexidade de tempo do algoritmo fosse maior que a esperada. Entretanto, de acordo com os Teoremas 4 e 8 do artigo de Telles e Meidanis [9], o número de nós criados é menor que o número de nós movidos, e, como este é linear em relação a entrada, a estrutura de union-find tem $O(r)$ nós, o que nos dá uma complexidade de $O(\alpha(r))$ para a busca.

Apesar da possibilidade de movimentação de nós, o uso da estrutura de union-find é possível, pois, em todas as movimentações temos que filhos de nós P tornam-se filhos de nós Q ou R, ou filhos de nós Q tornam-se filhos de nós R, isto é, os nós movem-se para pais de mesmo tipo ou seguindo a progressão P $\rightarrow$ Q $\rightarrow$ R. Ademais, uma vez que dois nós são irmãos e filhos de um nó do tipo Q ou R, eles não deixam de ser irmãos. Como consequência, um nó que não estava na estrutura union-find pode ser inserido nela; porém, uma vez que está na estrutura union-find, nunca é preciso sair dela.

4 Conclusões

Este trabalho consolida o conhecimento sobre o algoritmo incremental encontrado em trabalhos anteriores de Telles e Meidanis. Atenção especial foi dada em relação a: especificação mais cuidadosa das operações, provas de corretude e maiores detalhes sobre o uso de estrutura union-find, incluindo argumentos de por que ela funciona, sem necessitar pagar o preço da retirada de elementos.

Os próximos temas a serem estudados são: uma prova de que a complexidade do algoritmo apresentado neste relatório é o limitante inferior para o problema e um algoritmo offline de complexidade de tempo linear.

Referências

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

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

[3] T. H. Cormen, R. L. Rivest, C. E. Leiserson, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2^{nd} edition, 2001.

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

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

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

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

[8] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997. ISBN: 0-534-95262-3.

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