Building PQR Trees in Almost-Linear Time

Guilherme P. Telles [*] Joao Meidanis [†]

August 1, 2004

Abstract

In 1976, Booth and Leuker invented the PQ trees as a compact way of storing and manipulating all the permutations on n elements that keep consecutive the elements in certain given sets $C_1, C_2, \ldots, C_m$. Such permutations are called *valid*. This problem finds applications in DNA physical mapping, interval graph recognition, logic circuit optimization and data retrieval, among others. PQ trees construction time is linear on the size of the input sets. In 1995, Meidanis and Munuera created the PQR trees, a natural generalization of PQ trees. The difference between them is that PQR trees exist for every set collection, even when there are no valid permutations. The R nodes encapsulate subsets where the consecutive ones property fails. In this note we present an almost-linear time algorithm to build a PQR tree for an arbitrary set collection.

Keywords: Trees, analysis of algorithms

1 Introduction

Given a collection of m subsets $C_1, C_2, \ldots, C_m$ of a set U of n elements, the *consecutive ones problem* consists in answering whether there is a *valid permutation* of the elements in U, that is, a permutation that keeps the elements of each C_i consecutive. Such permutations are called *valid.*

[*]Institute of Mathematical and Computer Sciences, University of Sao Paulo, Po Box 668, 13560-970, Sao Carlos, Sao Paulo, Brazil. E-mail: gpt@icmc.usp.br

[†]Scylla Bioinformatics, R. James Clerk Maxwell, 360, 13069-380, Campinas, Sao Paulo, Brazil and Institute of Computing, University of Campinas, Po Box 6176, 13083-970, Campinas, Sao Paulo, Brazil. E-mail: meidanis@ic.unicamp.br

Many problems can be stated as the consecutive ones problem, such as DNA physical mapping [10], interval graphs recognition [5], logic circuit optimization [4] and data retrieval [6].

The consecutive ones problem was solved by a polynomial time algorithm devised by Fulkerson and Gross [5] in 1965. Booth and Leuker [2] invented the PQ trees in 1976, a data structure to solve the problem and compactly represent every valid permutation for U subject to $C_1, C_2, \ldots, C_m$. They also gave a linear time algorithm for building PQ trees. In 1995, Meidanis and Munuera [8] created PQR trees, a natural generalization of PQ trees, and gave a quadratic algorithm to build PQR trees. In a continuation of this work, Meidanis, Porto and Telles [9] extended the algebraic theory behind this problem. PQR trees can be used to solve all problems that PQ trees solve, with the additional advantage that when there is no valid permutation PQR trees will point out specific subcollections responsible for the failure of the consecutive ones property [9]. This feature is specially interesting in applications where experimental errors may occur and the problem to solve becomes NP-complete, as in DNA physical mapping [10, 7].

For a time it was unknown whether the extra R nodes add enough complexity into the structure as to make a linear time algorithm impossible. One major problem in this respect was related to the movement of uncolored ("white") nodes (see Section 4), which did not seem to be bounded by the number of black or gray nodes. Here we show how to implement the algorithm so as to never have to move uncolored nodes. Another important issue was to merge two Q or R nodes in time $O(1)$. We solved this issue by representing the children of such nodes as a union-find structure, where only one designated child point to the parent, and the other siblings point to this designated child directly or indirectly, composing a union-find structure. This leads to $O(1)$ merging, but the price to pay is $O(\alpha(r))$ to find the parent, and an overall almost-linear time bound, where r is the sum of sizes of the input sets.

The algorithm given here is not a simple extension of Booth and Leuker's algorithm, but relies on deeper properties of the trees uncovered by the new theory. It uses a fewer number of better organized patterns than the PQ algorithm. The correctness proof also draws heavily on the theory developed. We believe that this contributes significantly towards a better solution to this problem.

This article is organized as follows. Section 2 gives basic definitions and Section 3 introduces the basics on PQR theory. Our algorithm and analysis come in Sections 4 and 5. Our conclusions appear in Section 6.

2 Definitions

The term **collection** is used here as a synonym for set of sets. Hereafter, collections will always be denoted by calligraph capital letters, such as $\mathcal{C}$. To simplify notation, we sometimes write a set as a list of its elements in any order. For example, $A = \{k, l, m, n\}$ can be written as $A = lnkm$. A **permutation** of a finite set U with $|U| = n$ is a one-to-one mapping $\alpha : \{1, 2, \ldots, n\} \mapsto U$. Permutations are also denoted as lists of elements, for instance, we write $\alpha = efcbda$ meaning $\alpha(1) = e$, $\alpha(2) = f$ and so on. The power set of a set U is denoted $\mathcal{P}(U)$. A **trivial** subset of U is a set with 0, 1, or n elements.

Given a permutation α of the elements of U, and a subset A of U, we say that A is **consecutive** in α when the elements of A appear consecutively in α. For example, if $U = abcdef$ and $\alpha = efcbda$, the subset $A = cdb$ is consecutive in α, while $B = efa$ is not. Given a pair $(U, \mathcal{C})$, with $\mathcal{C} \subseteq \mathcal{P}(U)$, we say that a permutation α of U is **valid** (with respect to $\mathcal{C}$) if all sets $A \in \mathcal{C}$ are consecutive in α. The pair $(U, \mathcal{C})$ has the **consecutive ones property** (C1P) if there is at least one valid permutation.

A **PQR tree over a set** U is a rooted tree with four types of nodes – P, Q, R and leaves – subject to the following restrictions:

- The leaves are in one-to-one correspondence with the elements of U.
- Every P node has at least two children.
- Every Q node has at least three children.
- Every R node has at least three children.

A node v in a PQR tree is identified with the set of leaves having v as ancestor. We use v to denote either the node or the set of its descending leaves (elements of U) interchangeably.

An example of a PQR tree over the set $U = abcdefghijklmnop$ and the collection $\mathcal{C} = \{abcdef, bc, cd, de, ce, ghijklmn, no, op, hijkl, lm\}$ appears in Figure 1. The notion of a PQR tree for a collection $\mathcal{C} \subseteq \mathcal{P}(U)$ is formalized in Section 3.

Two PQR trees are **equivalent** when one can be obtained from the other by zero or more of the following operations:

- arbitrary permutations of the children of a P node,
- reversal of the children of a Q node,
- arbitrary permutations of the children of an R node.

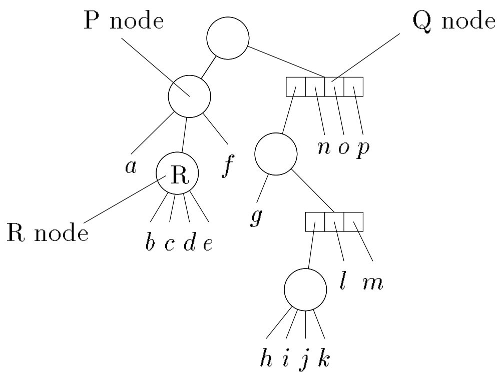

Figure 1: An example of a PQR tree.

This is an equivalence relation. We denote by $\overline{T}$ the equivalence class of a tree T.

A PQ tree is a PQR tree without R nodes [9]. PQ trees are a compact way of representing all valid permutations for a collection with the C1P [2]. In general, regardless of whether the tree has the C1P, a PQR tree is a compact way of representing its completion (see Section 3).

3 PQR Theory

Meidanis and Munuera [8] created a theory, later consolidated by Meidanis, Porto, and Telles [9], that greatly helps reasoning about the problem and lies at the heart of the algorithm we present here. In this section we recall basic findings of this theory that will be used in the sequel.

The main points of the theory are:

- the realization that the set operations *intersection*, *noncontained difference*, and *nondisjoint union* produce consecutive sets from consecutive sets and can be used to enrich the input set collection $\mathcal{C}$ into a normalized, complete collection $\overline{\mathcal{C}}$. The PQR tree associated to $\mathcal{C}$ is unique up to equivalence and depends only on $\overline{\mathcal{C}}$.

- the realization that the concept of *orthogonality* between sets can be used to characterize the nodes that appear in a PQR tree for $\mathcal{C}$, through the collection $\mathcal{C}^{\perp}$, which holds every set orthogonal to $\mathcal{C}$.

Formally, given two sets A and B, we define the following notation for operations provided that A and B satisfy some precondition:

- The **nondisjoint union** of A and B, denoted $A \uplus B$, is equal to $A \cup B$ provided that $A \cap B \neq \emptyset$.
- The **noncontained difference** of A and B, denoted $A \setminus\!\!\!\!\!\! \wedge\, B$, is equal to $A \setminus B$ provided that $B \not\subseteq A$.

A collection is **complete** when it contains the trivial sets and is closed under the operations *intersection*, *nondisjoint union* and *noncontained difference*. Given a collection $\mathcal{C} \subseteq \mathcal{P}(U)$, its completion $\overline{\mathcal{C}}$ is the smallest complete collection that contains $\mathcal{C}$. Analogously, the completion of a PQR tree T, denoted by $Compl(T)$, is defined as the following collection:

1. The empty set, the set of all leaves in T and the sets $\{x\}$ such that x is a leaf of T are contained in $Compl(T)$,
2. $\bigcup_{v \in S} v$ is in $Compl(T)$ if S is the set of all children of a P node of T,
3. $\bigcup_{v \in S} v$ is in $Compl(T)$ if S is a set of consecutive children of a Q node of T,
4. $\bigcup_{v \in S} v$ is in $Compl(T)$ if S is an arbitrary set of children of an R node of T,
5. No other sets are in $Compl(T)$.

Definition 1 *We say that T is a PQR tree for $\mathcal{C}$ when $\overline{\mathcal{C}} = Compl(T)$.*

In general there are several PQR trees for the same collection $\mathcal{C}$, and therefore we cannot speak of **the** PQR tree for $\mathcal{C}$, However, all these trees are equivalent, hence we can unambiguously define an equivalence class of trees for a given collection $\mathcal{C}$.

Definition 2 *We denote by $PQR(\mathcal{C})$ the equivalence class of all PQR trees for $\mathcal{C}$.*

Two sets A and B are **orthogonal**, denoted $A \perp B$, if either $A \subseteq B$, or $B \subseteq A$, or else $A \cap B = \emptyset$. Similarly, a collection $\mathcal{C}$ is orthogonal to a set A if every set in $\mathcal{C}$ is orthogonal to A. Given a collection $\mathcal{C} \subseteq \mathcal{P}(U)$, the collection $\mathcal{C}^{\perp}$ is the collection of every set in $\mathcal{P}(U)$ that is orthogonal to $\mathcal{C}$.

The following results from an earlier work [9] are important here. The first one states that for every collection there is a PQR tree, the second characterizes the nodes in a PQR tree and the third relates the C1P to the PQR trees.

Theorem 1 (Th. 38 from Porto et al [9]) *For any collection $\mathcal{C}$ over a set U, there is a PQR tree such that*

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

Theorem 2 (Lemma 35 from Porto et al [9]) *If T is a PQR tree for a collection $\mathcal{C}$ then the nodes of T correspond exactly to the sets in $\overline{\mathcal{C}} \cap \mathcal{C}^{\perp}$.*

Theorem 3 (Th. 15 and 31 from Porto et al [9]) *If T is a PQR tree for a collection $\mathcal{C}$ and T has no R nodes, then the set of all permutations of U valid with respect to $\mathcal{C}$ can be obtained reading the leaves of every tree equivalent to T from the left to the right.*

4 An Almost-Linear Time Algorithm

The algorithm for PQR tree construction in this section is an on-line algorithm, in the sense that it builds a PQR tree T for the collection $\mathcal{C}$ adding one set from $\mathcal{C}$ at a time. We start with a so called universal tree, which is a tree with a single internal node of type P with all leaves as its children. This tree corresponds to the empty set collection.

The procedure of adding S can be explained using the diagram in Figure 2. In this diagram, $\mathcal{C}$ is a collection and $\mathcal{T}$ is $PQR(\mathcal{C})$. We use the notation $\mathcal{C}+S$ meaning $\mathcal{C} \cup \{S\}$ and $\mathcal{T} + S$ meaning

$$\mathcal{T} + S = \{Y \mid \exists\, T \in \mathcal{T} \text{ with } \overline{Compl(T) \cup \{S\}} = Compl(Y)\}.$$

Based on the fact that two trees are equivalent if and only if they have the same completion [9, Theorem 36], it is easy to see that $\mathcal{T} + S$ is also an equivalence class of PQR trees. Furthermore, this class is $PQR(\mathcal{C} + S)$. The addition routine takes any tree from $\mathcal{T}$ and produces a tree in $\mathcal{T} + S$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\text{PQR}"] \arrow[d, "\text{Add } S"'] & T \arrow[d, "\text{Add } S"] \\
\mathcal{C} + S \arrow[r, "\text{PQR}"] & T + S
\end{tikzcd}$$

Figure 2: Diagram explaining one iteration of the main loop of the algorithm.

The algorithm for adding a set S to a tree T appears below. It first collors the tree using colors 0, 1, 2, with 0=white, 1=gray and 2=black, and finds the least common ancestor node of all leaves in S, refereed to as LCA. In the sequel gray nodes are repeatedly killed, merged, or uncolored. At the end, all nodes are back to their original color (white).

---

1. color the leaves corresponding to S
2. color the tree and find the LCA
3. restructure the tree, eliminating gray nodes
4. adjust the LCA
5. uncolor the tree

A better explanation of each step follows.

Step 1 Leaves corresponding to the elements of S are colored black.

Step 2 Coloring of internal nodes of T is done as follows:

- an internal node v is colored black when $v \subset S$,
- an internal node v is colored gray when $v \not\perp S$,
- an internal node v is left uncolored ("white") when either $v \cap S = \emptyset$ or $S \subseteq v$,

This coloring can be accomplished simultaneously with finding the LCA of all black leaves as described in the original Booth and Leuker paper [2]. It is important to guarantee that the LCA will always be an internal node. When S is a singleton, there is only one black leaf, and in this case we take its parent as the LCA.

Notice that the nodes colored gray will not be nodes of any tree $T + S$, since they are not orthogonal to S. This motivates the next step of the algorithm, which restructures the tree until no gray node remains.

Step 3 This step repeatedly kills, merges, or uncolors gray nodes, sometimes also creating new nodes when needed, until no gray nodes remain in the tree. This step is therefore the iteration of a simpler task, which is the processing of one gray node.

```
while there is a gray child v of the LCA r do
    prepare_LCA(r, v)
    prepare_child(r, v)
    if r is of type P then
        move children away from the LCA
    else
        merge into the LCA

subroutine prepare_LCA(r, v)
    case type of r of:
        P: join black children
        Q: conditionally reverse the LCA r
        R: do nothing

subroutine prepare_child(r, v)
    case type of r of:
        P: transform P node into Q node
        Q: conditionally reverse gray node v
        R: do nothing
```

The operations mentioned in the code fragment are illustrated in Figures 3 to 6, where a triangle represents a node whose type is either Q or R, r is the label for the LCA and v is the label for the gray child being processed.

The conditional reversal of a node is as follows. The LCA is reversed when the color of its leftmost child is less than the color of its rightmost child. A gray node v is reversed when the color of v's left neighbor is less than the color of v's right neighbor. If v has only one neighbor (i.e. v is the leftmost or rightmost node), apply the same criterion assuming that the color of its missing neighbor is $1/2$.

Booth and Lueker's algorithm for PQ trees construction consists of patterns that are to be matched and replaced within the tree. Our algorithm replaces this entire pattern matching scheme by simpler code, which depends only on the types of r and v, and is much easier to implement.

Step 4 After Step 3, there are no gray nodes left in the tree. Therefore, all maximal black nodes are children of the LCA. We have to adjust the LCA as follows, according to its type.

- the LCA is a P node: if the LCA has two or more black children, and at least one white child, create a new child of type P of the LCA and move all black maximal nodes to this new node.
- the LCA is a Q node: if all maximal black nodes are consecutive, do nothing; otherwise change the type of the LCA to R.

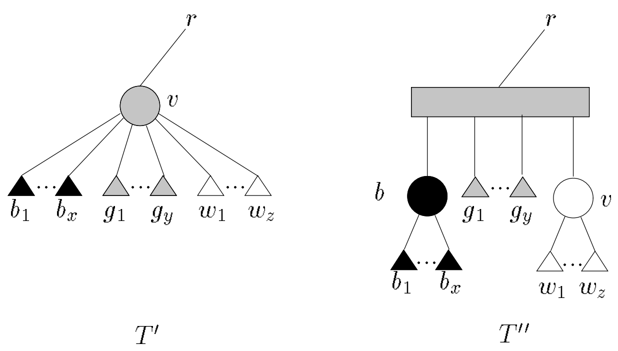

Figure 3: Operation "transform P node into Q node". This operation produces a Q node with children ordered in a way that does not require reversal, that is, in decreasing order of color.

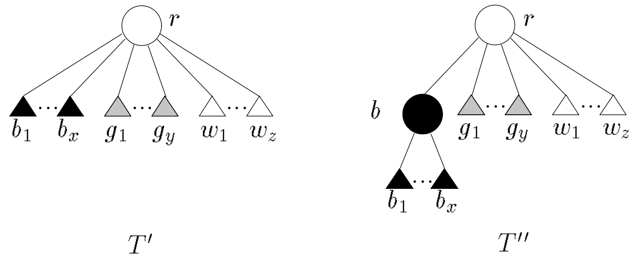

Figure 4: Operation "join black children".

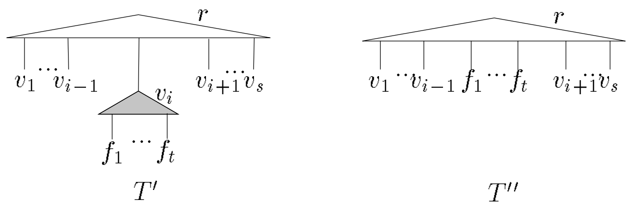

Figure 5: Operation "merge into the LCA".

- the LCA is a R node: do nothing.

We note that checking whether black children of a Q node are consecutive can be accomplished in time proportional to the number of black children.

Step 5 Uncolor all black nodes.

Figure 7 shows the execution of the algorithm for the tree shown and $S = ghm$.

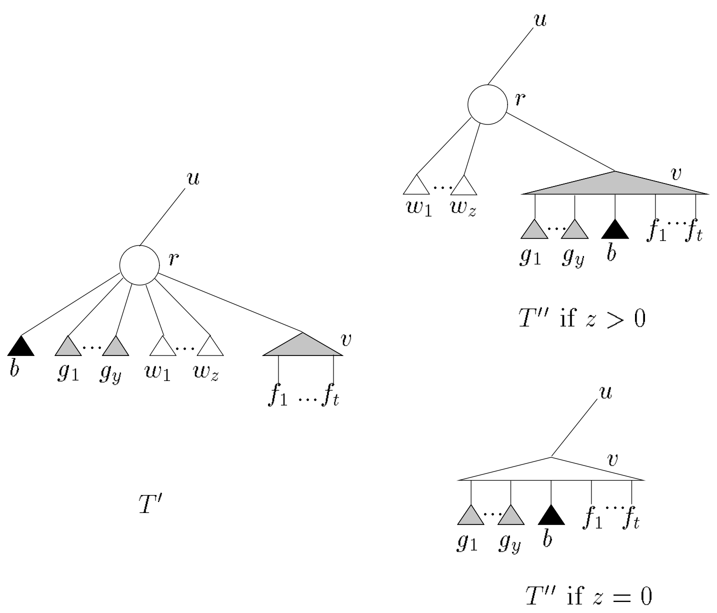

Figure 6: Operation "move children away from the LCA".

5 Correctness and complexity

5.1 Correctness

To prove that the algorithm is correct it is necessary to show that, if T' and T'' are the uncolored trees before and after any given step in the algorithm, then

$$\overline{Compl(T') \cup \{S\}} = \overline{Compl(T'') \cup \{S\}} \tag{1}$$

This is easy to see for Steps 1, 2, and 5. For Step 3, we need to show the invariant (1) holds for all the operations on PQR trees described in the previous section. For Step 4, when the LCA is a P node we just do the same as in the "join black chilcren" operation. When the LCA is a Q node we simply have another operation not present in Step 3: "transform Q node into R node".

To illustrate the proof method for operations, we will show how it applies to operation "join black children" (Figure 4). The nodes in T' all appear also in T'', with the same types. Therefore $\overline{Compl(T') \cup \{S\}} \subseteq \overline{Compl(T'') \cup \{S\}}$. On the other hand, we can write

$$b = (r \cap S) \obslash g_1 \obslash \ldots \obslash g_y,$$

which shows that $b \in \overline{Compl(T') \cup \{S\}}$ and allows us to conclude that $\overline{Compl(T'') \cup \{S\}} \subseteq \overline{Compl(T') \cup \{S\}}$, completing the proof.

If T and Y are the trees before and after the iteration that adds set S, we conclude that

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

Noticing that after Step 4 we have $S \in Compl(Y)$, we have actually

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

We conclude that

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

showing that tree Y is in fact in $\overline{T} + S$.

5.2 Complexity

In this section we will assume that we are given a collection $C_1, C_2, \ldots, C_m$ of subsets of $U = \{1, 2, \ldots, n\}$ to construct their PQR tree, and that $r = \sum_j |C_j|$.

The main operations involved in the algorithm are: node creation, node killing, node reversal, node moving, node merging, node coloring, and node uncoloring. We will consider first the operations done in Step 3, which is the most intricate step.

Each node must know its parent because of the way coloring is done (bottom to top). To guarantee the almost-linear time bound, we must implement the set of children of a Q or R node as a union-find structure[3]. One designated child will hold a pointer to their parent, and all other siblings will point to other siblings which are closer to the designated child. This permits execution of the "Merge into the LCA" operation in $O(1)$ time, but the price to pay is an $O(\alpha(r))$ bound for finding the parent of a node. Moreover, to perform the operations that compose the templates every node must know its children. Another drawback on using a union-find structure is that Q or R nodes cannot be actually deleted, but marked only as deleted. We have to be able to replace a child, revert a child list and merge child lists in $O(1)$ time. A data structure called symmetric list or symlist [1], based on ideas Sleator and Tarjan [11] can be used for this purpose

We will show that the algorithm performs $O(r)$ operations, with cost $O(\alpha(r))$ each. Each time a set S is added, work proportional to $|S|$ is allowed. It turns out that sometimes a small set is added and the tree changes significantly, meaning that we cannot guarantee that adding S can be done within $O(|S|)$ time. However, an amortized $O(|S|)$ bound holds. We will show this by exhibiting a tree potential function that records the part of the $O(|S|)$ not actually used for work yet and which is available for future operations.

In what follows, a *m*ove operation inlcudes also the case where a Q or R is merged with its Q or R parent. From this point on, let us denote the set of black, gray and white children of a node v respectively by $B(v)$, $G(v)$ and $W(v)$.

Theorem 4 *The amount of operations involved in Step 3 is dominated by the move operations.*

Proof: Leaves are never created or killed. Internal nodes are created with zero children, and only internal nodes with at most one child are killed. Therefore, to kill a node its children must be moved elsewhere first. Likewise, the creation of a node is always followed by movements to bring children in. The reversal of a node is always followed by a movement as well.

Node coloring occurs when a new node is colored black. Therefore, coloring is dominated by creation. Uncoloring occurs when all colored children of a P node are moved elsewhere. Therefore, uncoloring is dominated by movement. □

Theorem 5 *The total number of moves in Step 3 for adding a set S to a tree T is bounded by:*

$$\begin{aligned} m_3(T,S) \leq \; & |B(r)| + |G(r)| + \\ & \sum_{v \text{ gray P node}} (|B(v)| + |G(v)|) + \\ & \sum_{v \text{ gray Q/R node}} 1, \end{aligned}$$

where r is the LCA.

Proof: Table 1 shows the number of moves for each combination of LCA type and gray child type. Notice that in every case, the gray child being processed is destroyed or becomes uncolored. Therefore, each gray node is processed exactly once. Notice also that the LCA is of type P at most once for each set added (in the first iteration of the loop in Step 3), and therefore the term $|B(r)| + |G(r)|$ enters the sum at most once.

□

type of LCA	type of v	movements
P	P	$\lvert B(r)\rvert + \lvert G(r)\rvert + \lvert B(v)\rvert + \lvert G(v)\rvert$
P	Q/R	$\lvert B(r)\rvert + \lvert G(r)\rvert$
Q/R	P	$\lvert B(v)\rvert + \lvert G(v)\rvert$
Q/R	Q/R	1

Table 1: Upper bound on number of movements in each case of Step 3.

The main result of this section relies on the concept of *tree potential.* The potential of a tree T is defined as follows.

$$pot(T) = \sum_{v \text{ P node of } T} |v| + \sum_{v \text{ Q/R node of } T} 1.$$

Recall that $|v|$ is the cardinality of v as a set, which is the set of all its leaf descendants.

Theorem 6 *Let T be a PQR tree and Y the tree obtained from T by adding S as described in the algorithm of Section 4. The gain in potential $\Delta pot(T,S) = pot(Y) - pot(T)$ satisfies*

$$\Delta pot(T,S) \quad \leq \quad \sum_{v \in B(r)} |v| - \sum_{v \text{ gray P node}} \sum_{u \in G(v)} |u| - \sum_{v \text{ gray Q/R node}} 1,$$

where r is the LCA.

Proof: The nodes in Y that are not in T are (at most):

- P node with all black children of r (due to operation "Join black children")
- For each gray P node, a P node with all its black children (due to operation "Transform P node into Q node")
- For each gray P node, a P node with all its white children (due to operation "Transform P node into Q node")
- A Q node child of an LCA of type P (due to operation "Move children away from the LCA"). Notice that physical node v remains, but a new logical node (as a set) may be created.

On the other hand, the nodes in T that are not in Y are exactly the gray nodes. Therefore,

$$\begin{aligned}\Delta pot(T,S) &\leq \sum_{v\in B(r)} |v| + \sum_{v \text{ gray P node}} \sum_{u\in B(v)\cup W(v)} |u| + 1 - \\ &\quad - \sum_{v \text{ gray P node}} |v| - \sum_{v \text{ gray Q/R node}} 1 \\ &\leq \sum_{v\in B(r)} |v| - \sum_{v \text{ gray P node}} \sum_{u\in G(v)} |u| - \sum_{v \text{ gray Q/R node}} 1.\end{aligned}$$

□

Theorem 7 *The total number of operations work(T,S) involved in adding S to T satisfies*

$$work(T,S) = O(|S| + m_3),$$

where $m_3 = m_3(T,S)$ is as in Theorem 5.

Proof: Steps 1, 4 and 5 are $O(|S|)$. Step 2 is $O(|S| + m_3)$, as proved by Booth and Leuker [2]. In their paper they call $prunned(T,S)$ the subtree of all gray nodes plus the LCA, and show that Step 2 is $O(|prunned(T,S)|)$. It turns out that $|prunned(T,S)| = O(m_3)$, because every gray node v satisfies $|B(v)| + |G(v)| \geq 1$. □

Theorem 8 *There is an $O(|S|)$ amortized bound for adding a set S to tree T, namely*

$$work(T,S) + \Delta pot(T,S) = O(|S|).$$

Proof: In view of Theorem 7, it suffices to show that

$$m_3 + \Delta pot(T,S) = O(|S|).$$

From previous formulas, by straightforward algebraic manipulation,

$$\begin{aligned}m_3 + \Delta pot(T,S) &\leq |B(r)| + |G(r)| + \sum_{v\in B(r)} |v| + \sum_{v \text{ gray P node}} |B(v)| \\ &\leq 3|S|.\end{aligned}$$

This last inequality is due to the following reasons:

$$|B(r)| + \sum_{v \text{ gray P node}} |B(v)| \leq |S|$$

since $B(r) \cup \bigcup_{v \text{ gray P node}} B(v)$ is a disjoint union contained in S;

$$|G(r)| \;\leq\; |S|,$$

since the sets of $G(r)$ are disjoint; finally,

$$\begin{aligned} \sum_{v \in B(r)} |v| &= \left| \bigcup_{v \in B(r)} v \right| \\ &\leq |S|, \end{aligned}$$

since the sets of $B(r)$ are disjoint and contained in S. □

From this last theorem we can assess the total work needed to add sets $C_1, C_2, \ldots, C_m$ to a universal tree T_0 (a tree with just one P internal node with all leaves as its children), which results in a PQR tree T for this collection of input sets. Summing up the number of operations for adding all the sets, we end up with:

$$\sum_{i=0}^{m-1} work(T_i, C_{i+1}) + pot(T) - pot(T_0) = O(r + m),$$

or

$$\sum_{i=0}^{m-1} work(T_i, C_{i+1}) = O(r + m + n),$$

since $pot(T) \geq 0$, and $pot(T_0) = n$. As we saw, each operation can be performed in $O(\alpha(r))$, and therefore the almost-linear bound follows.

6 Conclusions

This work represents a significant contribution towards a cleaner and more general solution to the consecutive ones problem. Based on the theory developed by Meidanis and Munuera [8], later extended by Meidanis, Porto, and Telles [9], we propose a new algorithm to build PQR trees corresponding to a set collection, in time almost proportional to the collection's size.

This algorithm is more intuitive than the original PQ tree algorithm proposed by Booth and Lueker, and, in addition, constructs trees for all input collections, regardless of their consecutive ones status. The resulting tree will point out specific subcollections responsible for the failure of the consecutive ones property if that is the case.

Several practical applications modeled by the consecutive ones problem require the ability to deal with input data with errors. It is important to explore the possibility of using PQR trees to address this issue. We hope our contribution, besides being an interesting theoretical result, leads to practical algorithms for dealing with noisy data.

Acknowledgments

We acknowledge the financial support of Brazilian agencies FAPESP and CNPq.

References

[1] C. Bachmaier and M. Raitner. Improved symmetric lists. http://infosun.fmi.uni-passau.de/~raitner/, 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 and System Sciences*, 13(3):335–379, 1976.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[4] 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.

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

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

[7] Wen-Lian Hsu. On physical mapping algorithms: an error tolerant test for the consecutive ones property. In *Proceeding of the 3rd Int. Conf. on Computing and Combinatorics*, pages 242–250, 1997.

[8] J. Meidanis and E.G. Munuera. A simple linear time algorithm for binary phylogeny. In N. Ziviani, J. Piquer, B. Ribeiro, and R. Baeza-Yates, editors, *Proceedings of the XV Intern. Conf. of the Chilean Computer Science Society*, pages 275–283, Arica, Chile, 1995.

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

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

[11] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(1):362–391, 1983.

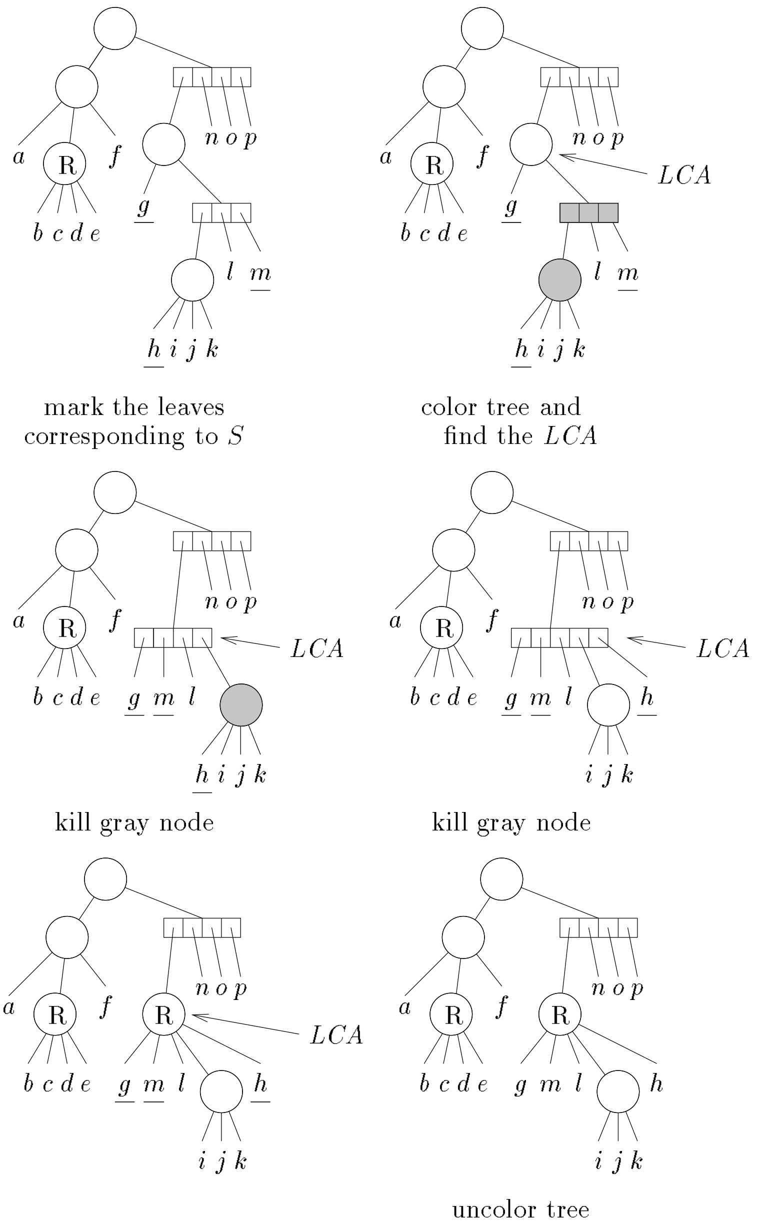

Figure 7: Execution of the algorithm for the tree shown and $S = ghm$.