Optimal Edge Ranking of Trees in Linear Time

Tak Wah Lam* Fung Ling Yue†

Abstract

Finding an optimal node ranking and an edge ranking of a tree are interesting computational problems. The former problem already has a linear time algorithm in the literature. For the latter, only recently polynomial time algorithms have been revealed, and the best known algorithm takes $O(n^2 \log n)$ time. In this paper, we present a new approach for finding an optimal edge ranking in $O(n)$ time, showing that the optimal edge ranking problem is no more difficult than the node counterpart.

1 Introduction

Let G be an undirected graph. A node ranking of G is a labeling of its nodes with positive integers such that every path between two nodes with the same label i contains an intermediate node with label $j > i$. A node ranking is optimal if it uses the least number of distinct labels among all possible node rankings. An edge ranking of G is a labeling of its edges satisfying an analogous condition, i.e., every path between two edges with the same label i contains an intermediate edge with label $j > i$. Figure 1 illustrates an optimal edge ranking. The problems of finding an optimal node ranking and an edge ranking of a graph have been well studied as they find applications in different context [1, 5, 9, 10]. Both problems are now known to be NP-hard [11, 8, 6]. Nonetheless, in most applications, the graphs in concern are restricted to trees only. This initiates the study of node ranking and edge ranking of trees.

Figure 1: An optimal edge ranking of a tree.

With respect to trees, the node ranking problem seems easier than the edge ranking problem. There is already a linear time algorithm for finding an optimal node ranking of a tree [12]. For edge ranking, after the pioneering work of Iyer, Ratliff, and Vijayan [5], de la Torre, Greenlaw and Schäffer [3] devised the first polynomial time algorithm (more precisely, $O(n^3 \log n)$ time) for finding an optimal edge ranking of a tree. Then Zhou and Nishizeki [13] improved their algorithm to run in $O(n^2 \log \Delta)$ time, where n is the number of nodes and Δ is the maximum degree of the tree.

In this paper, we present an $O(n)$ time algorithm for finding an optimal edge ranking of a tree, matching the optimal result of the node counterpart.

An optimal edge ranking of a tree may not be unique. In fact, the optimal edge ranking computed by our algorithm may not be the same as those by previous algorithms [3, 13]. From a conceptual viewpoint, the improvement achieved by our algorithm is rooted at a broader class of optimal edge rankings. Such a class leads to a more ambitious approach for labeling the edges—the new approach can label a number of edges at a time without optimizing for individual edges. Previous algorithms are actually based on a "guess-and-test" approach, where determining a right label for an edge usually requires many guesses, each followed by a time-consuming testing. Our approach reduces the number of guesses per edge to no more than two on average and does not need to perform an explicit testing to ensure the optimality of the ranking.

To make our new approach fully effective, we need some novel data structure techniques. As a warm-up, we first give an $O(n \log n)$ time algorithm, which is based on a compact representation of labels, supplemented by a conservative merging process that avoids redundant processing of most of the labels. Then we further enhance the algorithm to run in $O(n)$ time. The improvement stems from a tight analysis of the usage of labels.

The remainder of this paper is organized as follows: Section 2 gives the basic definitions. Section 3 presents a new approach for finding an edge ranking of a tree, and Section 4 proves its correctness. Section 5 shows how the new approach can be executed in $O(n \log n)$ time. The last section shows how to improve the time complexity to linear.

*Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, twlam@cs.hku.hk

†Management Information Section, City University of Hong Kong, Kowloon, Hong Kong, opflyue@cityu.edu.hk

2 Concepts and notations

Following previous work [3, 13], we consider the input tree to be rooted at an arbitrary node. Note that rooting a tree does not change the definition of edge ranking, but it suggests a natural way to decompose the computation.

Let φ be an edge ranking of a tree T with root r. An edge e of T is said to be *visible* if all the labels on the path from e to r are smaller than or equal to the label of e. Take an example, all edges incident to r are visible. It follows from the definition of ranking that all visible edges must have distinct labels. A label ℓ is said to be *visible* if there is a visible edge labeled with ℓ. Denote by $L(\varphi)$ the set of visible labels of T under φ. Figure 2 shows an edge ranking φ of a tree T, where $L(\varphi) = \{5,4,3,1\}$.

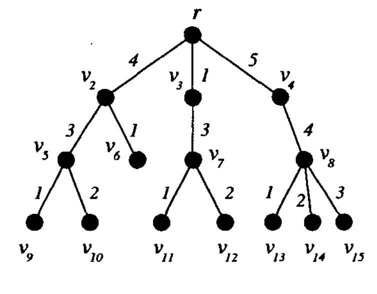

Figure 2: A tree with an edge ranking. The visible labels are {5,4,3,1}.

We determine the lexicographical order of two sets of labels by examining the labels in decreasing order. For instance, both $\{5,3,1\}$ and $\{3\}$ are considered to be smaller than $\{5,3,2\}$. An edge ranking φ of T is said to be *critical* if $L(\varphi)$ is lexicographically smaller than or equal to $L(\varphi')$ for any edge ranking φ' of T. A critical edge ranking is an optimal edge ranking, but the reverse may not be true. All critical edge rankings of T have the same set of visible labels. The *critical set of visible labels* of T, denoted by $L(T)$, is the set of visible labels of a critical edge ranking of T. The algorithm presented in this paper finds a critical edge ranking of a tree. The work of Torre, Greenlaw and Schäffer [3] has an important implication regarding the computation of critical edge rankings. Suppose the root of T has d children, numbered from 1 to d. Denote the subtree of T rooted at the i-th child of the root and the edge between the root and this child as T_i and *branch* i, respectively.

LEMMA 2.1. (see [3]) A critical edge ranking of T can be formed by any critical edge rankings of $T_1, T_2, \cdots, T_d$ together with a suitable labeling of the branches.

In light of Lemma 2.1, we can compute a critical edge ranking of T using a bottom-up approach:

(1) find a critical edge ranking for every subtree T_i;

(2) compute a label β_i for every *branch* i so that this labeling together with the critical edge rankings found in (1) form a critical edge ranking of T.

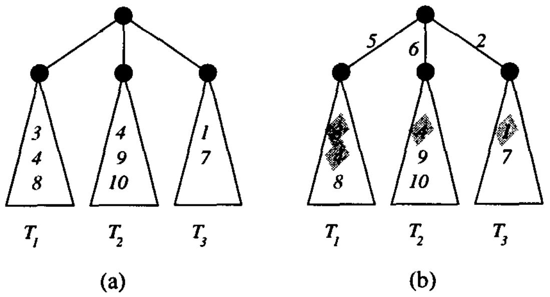

Figure 3: (a) $L_1 = \{8,4,3\}$, $L_2 = \{10,9,4\}$, and $L_3 = \{7,1\}$. (b) $\mathcal{B} = (5,6,2)$ is a valid labeling. The shaded labels are no longer visible; $\hat{L}_1 = \{8\}$, $\hat{L}_2 = \{10,9\}$ and $\hat{L}_3 = \{7\}$.

2.1 Valid labelings and optimal labelings

Suppose the critical edge rankings of the subtrees $T_1, T_2, \cdots, T_d$ and the critical sets of visible labels $L_1, L_2, \cdots, L_d$ have been computed. Let us focus on labeling the branches. Obviously, not any arbitrary labeling $\mathcal{B} = (\beta_1, \beta_2, \cdots, \beta_d)$ can form an edge ranking, let alone a critical edge ranking. Let $\hat{L}_i$ denote the set $\{\ell \in L_i \mid \ell > \beta_i\}$. $\mathcal{B}$ is said to be a *valid* labeling if for all branches i,

(i) $\beta_i \notin L_i$, and

(ii) $(\hat{L}_i \cup \{\beta_i\}) \cap (\hat{L}_j \cup \{\beta_j\}) = \phi$ for any branch $j \neq i$.

It is easy to verify that a valid labeling forms an edge ranking of T. Figure 3 gives an example. A valid labeling that forms a critical edge ranking of T is called an *optimal* labeling. We devote Sections 3 through 6 to showing how to find an optimal labeling. This optimal labeling may not be the same labeling as found by previous algorithms [3, 13]; nevertheless, they all use the same set of labels.

2.2 Partial labelings and conflicts

To compute a labeling of the branches of T, our algorithm initially labels every branch of T with zero, then it increases the labels stage by stage until they form a valid labeling. We require that at the end of every stage those branches that have already received positive labels satisfy a condition similar to that of a valid labeling.

Let $\mathcal{B} = (\beta_1, \beta_2, \ldots, \beta_d)$ be a labeling of the branches of T, in which some β_i's may be equal to zero. $\mathcal{B}$ is said to be a *partial* labeling if for all branches i with $\beta_i > 0$, **(i)** $\beta_i \notin L_i$, and **(ii)** $(\hat{L}_i \cup \{\beta_i\}) \cap (\hat{L}_j \cup \{\beta_j\}) = \phi$ for any branch $j \neq i$. A partial labeling in which all branches are assigned positive labels is a valid labeling.

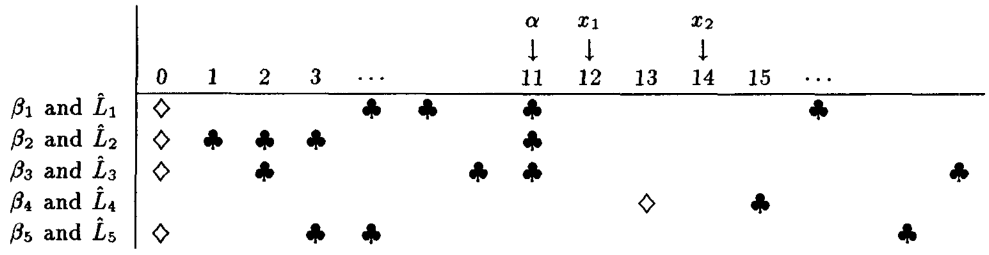
α
x_1
x_2
0 1 2 3 ··· 11 12 13 14 15 ···
β_1 and $\hat{L}_1$
β_2 and $\hat{L}_2$
β_3 and $\hat{L}_3$
β_4 and $\hat{L}_4$
β_5 and $\hat{L}_5$

Figure 4: In this example, there are five branches. $\beta_1 = \beta_2 = \beta_3 = \beta_5 = 0$, and $\beta_4 = 13$. The content of each $\hat{L}_i$ is represented by a number of ♣'s on the same row. There is a conflict at $\alpha = 11$. $K = \{1, 2, 3\}$. The smallest two free labels bigger than α are 12 and 14. Assigning 12 and 14 to branches 1 and 3 (in any order) resolves the conflict, while minimizing lexicographically the visible labels left.

With respect to a partial labeling $\mathcal{B}$, we say that there is a *conflict* at a label $\ell > 0$ if there exist two visible edges labeled with ℓ. The definition of partial labeling guarantees that a conflict, if present, must be due to two (or more) visible edges each residing in a distinct subtree T_i such that $\beta_i = 0$.

If $\mathcal{B}$ is a valid labeling, there is no conflict at any label $\ell > 0$. Thus, to produce a valid labeling eventually, our algorithm attempts to resolve all conflicts in the current partial labeling. To resolve a conflict at ℓ where $\ell \in \hat{L}_i \cap \hat{L}_j$ for some $i \neq j$, we can increase the label on one of the branches i and j to some value greater than ℓ. The new label assigned to the branch i (or j) should be distinct from all the currently visible labels of T; otherwise, another conflict is generated. That means, we should assign a branch with a *free* label, which is a positive integer not equal to any of the currently visible labels.

3 Algorithmic framework

Assume that the critical sets of visible labels $L_1, L_2, \cdots, L_d$ have been computed. In this section, we describe the framework of a new algorithm for finding a valid labeling of the branches of T. The discussion focuses on algorithmic aspects only. In Section 4, we give a characterization of this labeling and prove that it is an optimal labeling. In Sections 5 and 6, we describe some novel data structure techniques for executing this framework efficiently.

The algorithm starts off with a partial labeling in which every label is zero. The computation is divided into a number of iterations. Each iteration increases the labels of one or more branches, producing another partial labeling. When all branches have been assigned positive labels, the partial labeling obtained is a valid labeling and the algorithm stops.

In one iteration, the computation is as follows: We figure out the maximum label α at which the current partial labeling gives rise a conflict. Let $K = \{i \mid \alpha \in \hat{L}_i\}$, and let $k = |K|$. Our goal is to identify $k-1$ free labels greater than α and assign them to any $k-1$ branches in K, thus resolving the conflict at α. To attain an optimal labeling eventually, the conflict must be resolved *optimally*, that is, the labels to be put on the branches and the visible labels left in all subtrees are minimized lexicographically.

When we choose free labels to resolve the conflict, a natural attempt is to pick the smallest $k-1$ free labels starting from $\alpha + 1$, say, $x_1, x_2, \cdots, x_{k-1}$. Assigning these labels to some $k-1$ branches in K can resolve the conflict immediately. Yet, in some cases, this does not resolve the conflict optimally, and it is actually better to assign some x_i to branches outside K. In Section 3.1, we state a condition under which assigning the free labels $x_1, x_2, \cdots, x_{k-1}$ to the branches in K is always the best move. We show how to handle other cases in Sections 3.2 and 3.3.

3.1 Resolving the conflict immediately Consider the interval $[\alpha + 1, x_{k-1}]$. Suppose the labels in $\hat{L}_1, \hat{L}_2, \cdots, \hat{L}_d$ do not fall into $[\alpha+1, x_{k-1}]$. In this case, for any $i \notin K$, the labels in $\hat{L}_i$ are either bigger than $x_k - 1$ or smaller than α. Thus, assigning $x_1, \cdots, x_{k-1}$ to branches in K is more beneficial as we can eliminate α from $\hat{L}_i$ for every $i \in K$. More interestingly, a simple strategy suffices to choose which $k-1$ branches in K to receive the free labels $x_1, \cdots, x_{k-1}$.

> *Simple Strategy:* Define Cover$(\alpha, \hat{L}_i) = \{\ell \in \hat{L}_i \mid \ell \leq \alpha\}$. Let i_0 be a branch in K such that the set Cover$(\alpha, \hat{L}_{i_0})$ is lexicographically smallest among all branches in K. The free labels $x_0, x_1, \cdots, x_{k-1}$ are put, in any order, on the branches in K excluding i_0.

An example is depicted in Figure 4. Notice that permuting the free labels among the $k-1$ branches chosen has no effect on the visible labels left in the corresponding subtrees; thus, we do not need to consider these $k-1$ branches separately. Assuming all previous conflicts have been resolved optimally, we can prove that the above strategy minimizes lexicographically both the labels put on the branches and the visible labels left in the subtrees. The formal proof is given in the Section 4.

3.2 Freeing better labels Should there be a visible label in some $\hat{L}_i$ lying in the range $[\alpha+1, x_{k-1}]$, applying the simple strategy directly may not resolve the conflict optimally. Figure 5 shows an example.

Intuitively, if there is a visible label ℓ in some $\hat{L}_i$ falling in the range $[\alpha+1, x_{k-1}]$, it is more advantageous to assign the free label just greater than ℓ to branch i instead of a branch in K because this assignment eliminates a visible label greater than α. This motivates us to handle such a case as follows:

> Let $x \leq x_{k-1}$ be the smallest free label bigger than α such that there exists a label in some $\hat{L}_i$ lying in the range $[\alpha+1, x]$. Let j be the branch such that $\hat{L}_j$ contains the biggest label smaller than x. We increase the label of branch j to x.

Note that j may or may not be a branch in K. After labeling branch j, we get at least one additional free label less than x. Intuitively, the smallest $k-1$ free labels move closer to α.

Now we examine the conflict at α again. If it has not been resolved, we recompute the set K and the smallest $|K|-1$ free labels bigger than α. If the visible labels left in the subtrees fall outside the range spanned by these $|K|-1$ free labels, we can apply the simple strategy immediately; otherwise, we repeat the procedure above.

3.3 Tackling the boundary case Eventually we will come to the case where there is no conflict under the current labeling but one or more branches still get a label zero. In this case, our aim is to identify a free label for each of these branches. We set $\alpha = 0$ and compute $K = \{i \mid \beta_i = 0\}$. If the currently visible labels in the subtrees all fall outside the range spanned by the $|K|$ (instead of $|K|-1$) smallest free labels, we apply the simple strategy to assign the $|K|$ smallest free labels to the branches in K as in Section 3.1, getting a valid labeling. Otherwise, we execute the procedure in Section 3.2 to label a branch not necessary in K freeing more small labels.

3.4 The Algorithm The computation described in previous sections is put together as a procedure, called LABELING.

Procedure LABELING(T)
Input: $L[1], L[2], \cdots, L[d]$
Output: $\beta[1], \cdots, \beta[d]$ (the branch labels) and L, the critical set of visible labels of T.

```
o // Step I - Initialization
  for $i = 1, 2, \ldots, d$ do $\beta[i] := 0$;   $\hat{L}[i] := L[i]$;
  $zero\text{-}label\text{-}count := d$;
o while $zero\text{-}label\text{-}count > 0$ do
  • // Step II - Locating the maximum conflict
    if there is a conflict then
      $\alpha := \max\{\ell \mid \ell \in \hat{L}[i] \cap \hat{L}[j]$ for some $i \neq j\}$;
      $K := \{i \mid \alpha \in \hat{L}[i]\}$;   $k := |K|$;
    else
      $\alpha := 0$;   $K := \{i \mid \beta[i] = 0\}$;   $k := |K|$;
  • // Step III - Counting free labels
    $\gamma :=$ the smallest label in $\bigcup_{i=1}^{d} \hat{L}[i]$ bigger than $\alpha$;
    $w :=$ the number of free labels bigger than $\alpha$ but smaller than $\gamma$;
    {In case no labels in $\bigcup_{i=1}^{d} \hat{L}[i]$ is bigger than $\alpha$, set $w := k$;}
  • if ($\alpha > 0$ and $w < k-1$) or ($\alpha = 0$ and $w < k$)
    then // I.e., not enough free labels
      // Step IV - Producing more free labels
      $x :=$ the smallest free label bigger than $\gamma$;
      $j :=$ the branch such that among all branches, $\hat{L}[j]$ contains the biggest label which is smaller than $x$;
      if $\beta[j] = 0$ then $zero\text{-}label\text{-}count := zero\text{-}label\text{-}count - 1$;
      $\beta[j] := x$;
    else if $\alpha > 0$ and $w \geq k-1$ then
      // Step Va Resolving the conflict at $\alpha > 0$
      compute the branch $i_0 \in K$ such that for all $i \in K$, Cover($\alpha, \hat{L}[i_0]$) $\preceq$ Cover($\alpha, \hat{L}[i]$);
      for each $i \in K - \{i_0\}$ do
        $\beta[i] :=$ the next smallest free label bigger than $\alpha$;
        $zero\text{-}label\text{-}count := zero\text{-}label\text{-}count - 1$;
    else if $\alpha = 0$ and $w \geq k$ then
      // Step Vb - Resolving the conflict at 0
      for each $i \in K$ do
        $\beta[i] :=$ the next smallest free label;
        $zero\text{-}label\text{-}count := zero\text{-}label\text{-}count - 1$;
  • // Step VI - Removing no-longer visible labels
    for each $i$ such that $\beta[i]$ gets a new value do
      delete all the labels in $\hat{L}[i]$, that are smaller than $\beta[i]$;
o // Step VII - Assembling visible labels
  $L := \cup_{i=1}^{d}(\hat{L}[i] \cup \{\beta[i]\})$;
```

The main body of the procedure LABELING is a **while** loop. In one iteration of the **while** loop, we locate the maximum conflict; if there are enough suitable free labels available, the conflict is resolved immediately; otherwise, we attempt to free more free labels. At the end of each iteration, the labels $\beta[1], \beta[2], \cdots, \beta[d]$ define a partial labeling. The positive labels in use always form a subset of the labels used by an optimal labeling (see Section 4). That means, the procedure never uses a "wrong" label. On exit from the **while** loop, the labels $\beta[1], \beta[2], \cdots, \beta[d]$ are all positive and form a valid labeling. In Section 4, we give a characterization of this valid labeling and prove that it is actually optimal.

Based on the procedure LABELING, we can construct an algorithm for computing a critical edge ranking of a rooted tree R. For each node v in R, let R_v denote the subtree in R rooted at v. We execute LABELING(R_v) for each internal node v of R in a bottom-up manner. The input to LABELING(R_v) consists of the critical sets of visible labels of the subtrees rooted at the children of v. This simple algorithm is referred to as EDGE-RANKING.

4 Proof of Optimality

Let T be a rooted tree. Denote the subtrees rooted at the children of T and their critical sets of visible labels as $T_1, T_2, \cdots, T_d$ and $L_1, L_2, \cdots, L_d$, respectively. In this section, we prove that the procedure LABELING(T) computes an optimal labeling for the branches of T.

In Section 4.1, we define a class of valid labelings, which characterizes the labeling computed by our procedure. In Section 4.2, we show that all intermediate partial labelings, as well as the final valid labeling, computed by our procedure are kept to minimum, thus leading to an optimal labeling.

The discussion below uses a notation $\langle \mathcal{B} \rangle$ to refer to the set of non-zero labels in a partial or valid labeling $\mathcal{B}$ of the branches of T.

4.1 The relaxed greedy-cover labeling

It is known that all optimal labelings of the branches of T must use the same set of labels [3]. Let us denote Σ as this set. Two optimal labelings differ only in the order the labels of Σ are assigned to the branches. The set of labels used by any valid labeling must be lexicographically bigger than or equal to Σ. Figure 6 depicts an example.

Not every valid labeling using Σ is optimal. The edge ranking algorithm of Torre, Greenlaw and Schäffer [3] is based on the existence of a greedily-constructed labeling which is guaranteed to be optimal. The greediness of such a labeling is captured by the following notion.

Definition: Let $\mathcal{B} = \{\beta_1, \beta_2, \cdots \beta_d\}$ be a valid

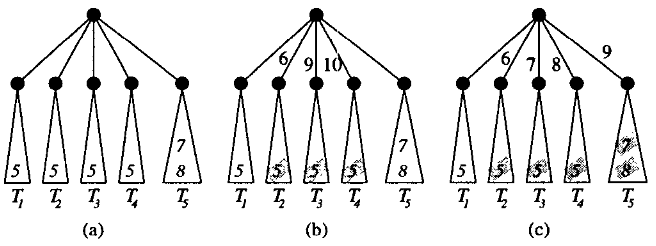

Figure 5: (a) A rooted tree T where $L_1 = L_2 = L_3 = L_4 = \{5\}$ and $L_5 = \{7, 8\}$. A conflict occurs at $\alpha = 5$; $K = \{1, 2, 3, 4\}$. The smallest $k-1$ free labels bigger than α are $\{6, 9, 10\}$. (b) Applying the simple strategy to resolve the conflict would label the branches 2, 3, 4 with 6,9,10, respectively. Such an assignment cannot form a critical edge ranking. (c) A better way to resolve the conflict is to first label branch 5 with 9.

labeling. A branch i is said to satisfy the *greedy-cover* (abbreviated as *gc*) property if for all branches j such that $\beta_j < \beta_i$, $(\text{Cover}(z_i, L_i), i) \succ (\text{Cover}(z_i, L_j), j)$.[1] $\mathcal{B}$ is said to be a *gc* labeling if every branch satisfies the *gc* property.

Intuitively, a gc labeling assigns the biggest label to a branch so as to cover the lexicographically biggest set of labels. The branch indices are used to break tie. The labeling shown in Figure 6(b) is a gc labeling. Notice that, given a fixed set of labels (say, Σ), a gc labeling is unique. Among all labelings using Σ, a gc labeling causes the smallest set of visible labels to be left in the subtrees, so it is optimal.

The optimal labeling computed by our algorithm does not follow the rule imposed by a gc labeling. We observe that with respect to a gc labeling $Z = (z_1, z_2, \cdots z_d)$, if there are branches i, j such that both L_i and L_j contain no labels between z_i and z_j inclusive, then z_i and z_j can be swapped without affecting the optimality of the labeling. See Figure 6(b) and (c) for an example. Below, we relax the definition of a gc labeling to capture the above observation.

Definition: Consider any valid labeling $\mathcal{B} = (\beta_1, \beta_2, \cdots, \beta_d)$. A branch i is said to satisfy the *relaxed greedy-cover* (abbreviated as *rgc*) property if for all branches j such that $\beta_j < \beta_i$, $(\text{Cover}(\beta_i, L_i), i) \succ (\text{Cover}(\beta_i, L_j), j)$, or both L_i and L_j contain no labels in the interval $[\beta_j, \beta_i]$. $\mathcal{B}$ is called a *rgc* labeling if every branch satisfies the *rgc* property.

A gc labeling is also a rgc labeling, but the reverse is not true. Also, unlike a gc labeling, a rgc labeling using Σ may not be unique. Given a rgc labeling $\mathcal{B}$,

[1] For any two sets of labels C and C' and any two distinct integers i and i', $(C, i) \succ (C', i')$ if and only if $C \succ C'$ or ($C = C'$ and $i > i'$).

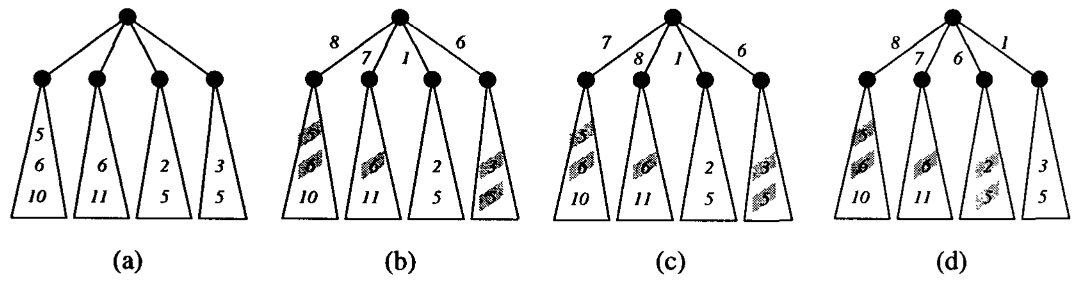

Figure 6: (a) A tree of which any optimal labeling must use the labels $\{1,6,7,8\}$. The labelings given in (b) and (c) are optimal while the one in (d) is not.

one can always permute the labels of the branches to produce a unique gc labeling $\bar{\mathcal{B}}$ which is *equivalent* to $\mathcal{B}$ in the sense that $\langle\mathcal{B}\rangle = \langle\bar{\mathcal{B}}\rangle$ and $\bar{\mathcal{B}}$ causes the same set of visible labels left in every subtree T_i as $\mathcal{B}$. Therefore, any rgc labeling using Σ, being equivalent to the gc labeling using Σ, must be optimal.

Definition: Consider any partial labeling $\mathcal{P} = (\beta_1, \beta_2, \cdots, \beta_d)$. **(i)** $\mathcal{P}$ is a *gc-partial labeling* if for any branch i such that $\beta_i > 0$, for any branch j such that $\beta_j < \beta_i$, $(\text{Cover}(\beta_i, L_i), i) \succ (\text{Cover}(\beta_i, L_j), j)$. **(ii)** $\mathcal{P}$ is a *rgc-partial labeling* if for any branch i such that $\beta_i > 0$, branch i satisfies the rgc property.

Given a rgc-partial labeling $\mathcal{P}$, one can always permute the labels on the branches to produce a unique gc-partial labeling $\bar{\mathcal{P}}$ which is equivalent to $\mathcal{P}$ in the sense that $\langle\mathcal{P}\rangle = \langle\bar{\mathcal{P}}\rangle$ and $\mathcal{P}$ and $\bar{\mathcal{P}}$ cause the same set of visible labels left in every subtree T_i. We call $\bar{\mathcal{P}}$ the *gc equivalent* of $\mathcal{P}$.

4.2 Invariant

A careful case analysis of the procedure LABELING can reveal that the partial labeling $\mathcal{P}$ produced at the end of every iteration of the **while** loop in the procedure is a rgc-partial labeling. Moreover, the following invariant is observed.

- Every positive label used by $\mathcal{P}$ is bigger than $\alpha(\mathcal{P})$, where $\alpha(\mathcal{P})$ denotes the maximum label at which there is a conflict.

In this section, we focus on a more important invariant concerning $\mathcal{P}$.

- $\langle\mathcal{P}\rangle \preceq \Sigma$.

In other words, when the procedure stops, we obtain a valid and rgc labeling $\mathcal{B}$ such that $\langle\mathcal{B}\rangle \preceq \Sigma$. Any valid labeling admits a set of labels lexicographically no smaller than Σ. Thus, $\langle\mathcal{B}\rangle = \Sigma$ and $\mathcal{B}$ is an optimal labeling.

The above two invariants seem a bit simple, yet we can easily derive from them two stronger properties concerning the quality of $\mathcal{P}$ (the proof is left in the full paper):

Property 1. $\langle\mathcal{P}\rangle \subseteq \Sigma$; and

Property 2. Let $\bar{\mathcal{P}} = (\bar{\beta}_1, \bar{\beta}_2, \cdots, \bar{\beta}_d)$ be the gc equivalent of $\mathcal{P}$. Let $Z = (z_1, z_2, \cdots, z_d)$ be the gc labeling using Σ. Then $\bar{\beta}_i \leq z_i$ for every branch i.

Below we give an inductive proof of the invariant that $\langle\mathcal{P}\rangle \preceq \Sigma$. Initially, the procedure LABELING assigns every branch a label zero, so the invariant holds. Suppose that after a number of iterations of the **while** loop, the rgc-partial labeling $\mathcal{P}$ produced satisfied the invariant. In the next iteration, one or more branches will receive bigger labels. Let $\mathcal{Q}$ be the new rgc-partial labeling. We are going to show that $\langle\mathcal{P}\rangle \prec \langle\mathcal{Q}\rangle \preceq \Sigma$.

Let us have a close look of the iteration that produces $\mathcal{Q}$ at the end. Steps II and III computes the values of α, K, k, γ, and w with respect to $\mathcal{P}$. Then, new labels are assigned to the branches in one of the Steps IV, Va, and Vb. Lemmas 4.1, 4.3, and 4.4 consider these three steps separately.

LEMMA 4.1. (for Step Vb) Suppose $\alpha = 0$ and $k \leq w$. Let W be the set of the k smallest free labels. Let $\mathcal{Q}$ be the labeling formed by assigning the labels in W to the k branches in K. Then $\langle\mathcal{Q}\rangle \preceq \Sigma$.

Proof. With respect to $\mathcal{P}$, the label of every branch i in K is zero. Thus, $\mathcal{Q}$ has exactly k positive labels more than P and $\langle\mathcal{Q}\rangle = \langle P\rangle \cup W$.

Let $\bar{\mathcal{P}} = (\bar{\beta}_1, \bar{\beta}_2, \cdots, \bar{\beta}_d)$ be the gc equivalent of $\mathcal{P}$. $\langle\mathcal{Q}\rangle = \langle\mathcal{P}\rangle \cup W = \langle\bar{\mathcal{P}}\rangle \cup W$. We want to show that $\Sigma \succeq \langle\bar{\mathcal{P}}\rangle \cup W$, then the lemma follows. Let $Z = (z_1, z_2, \cdots, z_d)$ be the gc labeling using Σ. Let $\Sigma_1 = \{z_i \mid \bar{\beta}_i > 0\}$. Property 2 of $\mathcal{P}$ states that $z_i \geq \bar{\beta}_i$ for every branch i. Thus, $\Sigma_1 \succeq \{\bar{\beta}_i \mid \bar{\beta}_i > 0\} = \langle\bar{P}\rangle$. Let Σ_2 be the $(d - |\Sigma_1|)$ smallest positive labels distinct from Σ_1. Obviously, $\Sigma \succeq \Sigma_1 \cup \Sigma_2$. Because $\Sigma_1 \succeq \langle\bar{\mathcal{P}}\rangle$, we further conclude that $\Sigma_1 \cup \Sigma_2 \succeq \langle\bar{\mathcal{P}}\rangle \cup W$.

LEMMA 4.2. *Suppose $\alpha > 0$. Let $\bar{\mathcal{P}} = (\bar{\beta}_1, \bar{\beta}_2, \cdots, \bar{\beta}_d)$ be the gc equivalent of $\mathcal{P}$. For any branch $i \in K$, $\bar{\beta}_i = \beta_i = 0$.*

Proof. Recall that with respect to a partial labeling, a conflict arises from branches with label zero. Thus, $\beta_i = 0$ for all $i \in K$.

Suppose on the contrary that there exists $i \in K$ such that $\bar{\beta}_i > 0$. Since every positive label in $\mathcal{P}$ is greater than α, $\bar{\beta}_i$ must be greater than α, too. Note that α is a positive label in L_i. $\bar{\beta}_i$ causes the label α in L_i to become invisible, though $\beta_i = 0$ does not. This contradicts the second property of a gc equivalent. Therefore, $\bar{\beta}_i = 0$ for all $i \in K$.

LEMMA 4.3. (for Step Va) Suppose $\alpha > 0$ and $k - 1 \le w$. Let W be the set of the $k-1$ smallest free labels bigger than α. Let $\mathcal{Q}$ be the partial labeling formed by assigning the labels in W to any $k-1$ branches in K. Then $\langle \mathcal{Q} \rangle \preceq \Sigma$.

Proof. Let $\bar{\mathcal{P}} = (\bar{\beta}_1, \bar{\beta}_2, \cdots, \bar{\beta}_d)$ be the gc equivalent of $\mathcal{P}$. Again, $\langle \mathcal{Q} \rangle = \langle \mathcal{P} \rangle \cup W = \langle \bar{\mathcal{P}} \rangle \cup W$. We are going to show that $\Sigma \succeq \langle \bar{\mathcal{P}} \rangle \cup W$.

Let $Z = (z_1, z_2, \cdots, z_d)$ be the gc labeling using Σ. Let $\Sigma_1 = \{z_i \mid i \notin K\}$. Since $z_i \ge \bar{\beta}_i$ for every branch i, $\Sigma_1 \succeq \{\bar{\beta}_i \mid i \notin K\}$. Lemma 4.2 implies that $\{\bar{\beta}_i \mid i \notin K\} \succeq \langle \bar{\mathcal{P}} \rangle$. In short, $\Sigma_1 \succeq \langle \bar{\mathcal{P}} \rangle$. Let Σ_2 be the set of $k-1$ smallest labels that are bigger than α and distinct from Σ_1. Note that Z must assign at least $k-1$ branches in K with label $> \alpha$ (otherwise, Z is not a valid labeling). Therefore, $\Sigma = \langle Z \rangle \succeq \Sigma_1 \cup \Sigma_2 \succeq \langle \bar{\mathcal{P}} \rangle \cup W$.

LEMMA 4.4. (for Step IV) Suppose ($\alpha = 0$ and $k > w$) or ($\alpha > 0$ and $k - 1 > w$). Let x be the smallest free label bigger than γ. Let j be the branch such that for all $i = 1, 2, \cdots, d$, $\text{Cover}(x, \hat{L}[j]) \succeq \text{Cover}(x, \hat{L}[i])$. Let $\mathcal{Q}$ be the partial labeling formed by relabeling the branch j to the value x in $\mathcal{P}$. Then $\langle \mathcal{Q} \rangle \preceq \Sigma$.

Proof. Let $\bar{\mathcal{P}} = (\bar{\beta}_1, \bar{\beta}_2, \cdots, \bar{\beta}_d)$ be the gc equivalent of $\mathcal{P}$. $\langle \mathcal{Q} \rangle = (\langle \bar{\mathcal{P}} \rangle - \{\bar{\beta}_j\}) \cup \{x\}$. Due to Property 1 of $\mathcal{P}$, we have $\Sigma \supseteq \langle \bar{\mathcal{P}} \rangle$. Below we further show that Σ contains at least one label $\ell \ge x$ such that $\ell \notin \langle \bar{\mathcal{P}} \rangle$. Then $\Sigma \succeq \langle \bar{\mathcal{P}} \rangle \cup \{x\} \succeq \langle \mathcal{Q} \rangle$.

To prove the existence of such an ℓ, we prove the following two statements. Let $p = |\langle \mathcal{P} \rangle|$ (i.e., the number of positive labels used by $\mathcal{P}$). Recall that w denotes the number of free labels in the interval $[\alpha + 1, \gamma - 1]$ with respect to $\mathcal{P}$.

(I) Σ contains more than $p + w$ labels bigger than α.

(II) If Σ and $\langle \bar{\mathcal{P}} \rangle$ contain the same set of labels $\ge x$, then Σ contains at most $p + w$ labels bigger than α.

(I) and (II) together imply that Σ must contain a label $\ell \ge x$ such that $\ell \notin \langle \bar{\mathcal{P}} \rangle$.

Proof of (I): If $\alpha = 0$ and $k > w$, every label in Σ is greater than $\alpha = 0$, and $|\Sigma| = d = p + k > p + w$. It remains to consider the case where $\alpha > 0$ and $k - 1 > w$. Let $Z = (z_1, \cdots, z_d)$ be the gc labeling using Σ. For each i such that $\bar{\beta}_i > 0$, i is not in K (due to Lemma 4.2), and $z_i \ge \bar{\beta}_i > \alpha$. On the other hand, Z must assign at least $k-1$ branches in K with label bigger than α. Thus, Z uses at least $p + k - 1 > p + w$ labels greater than α.

Proof of (II): Suppose $\bar{\mathcal{P}}$ and Σ have the same set of labels $\ge x$. Let $Z = (z_1, \cdots, z_d)$ be the gc labeling using Σ. Since both $\bar{\mathcal{P}}$ and Z are gc labelings, for each label $h \in \langle \bar{\mathcal{P}} \rangle$ such that $h \ge x$, h lies on the same branch under both $\bar{\mathcal{P}}$ and Z. This can be proved using an induction starting with the largest label.

Furthermore, $\bar{\mathcal{P}}$ and Z must have the same set of labels in the range $[\gamma, x-1]$. More precisely, for any label $h \in [\gamma, x-1]$, if h is used by $\bar{\mathcal{P}}$ then h is also used by Z and h labels the same branch w.r.t. to both $\bar{\mathcal{P}}$ and Z; otherwise, h is the label of a visible edge residing in the same subtree T_i w.r.t. to both $\bar{\mathcal{P}}$ and Z. This can be proven again by a backward induction starting with $h = x - 1$.

In summary, $\bar{\mathcal{P}}$ and Z use the same set of labels $\ge \gamma$. With respect to $\mathcal{P}$ (or $\bar{\mathcal{P}}$), let p_1 be the number of labels $\ge \gamma$, and let p_2 be the number of labels in the interval $[\alpha + 1, \gamma - 1]$. Note that $p = p_1 + p_2$. Z uses exactly p_1 labels $\ge \gamma$. Recall that w.r.t. $\mathcal{P}$, w denotes the number of free labels in the interval $[\alpha + 1, \gamma - 1]$, and no visible labels residing in any subtrees fall into this interval. Thus, $p_2 + w = \gamma - 1 - \alpha$. The number of labels that are bigger than α in $\langle Z \rangle$ is no more than $p_1 + (\gamma - 1 - \alpha) = p_1 + p_2 + w = p + w$.

In conclusion, based on Lemmas 4.1 - 4.4, we can prove inductively that the partial labeling $\mathcal{P}$ computed in every iteration satisfies the invariant $\langle \mathcal{P} \rangle \preceq \Sigma$.

5 $O(n \log n)$ time algorithm

Given a tree R with n nodes, the algorithm EDGE-RANKING invokes the procedure LABELING for every subtree R_v rooted at an internal node v of R. A brute-force implementation of LABELING would enable EDGE-RANKING to run in $O(n^2)$ time. In this section, we give a more efficient implementation of LABELING, improving the time complexity of EDGE-RANKING to $O(n \log n)$. This implementation is based on two novel ideas, namely, a compact representation of the critical set of visible labels and a conservative merging process that avoids redundant processing of most of the labels.

5.1 Segments For any integers $\ell \leq r$, denote by $[\ell, r]$ the segment of consecutive integers spanning from ℓ to r. Let L be a set of labels. L can be considered as a union of segments $[\ell_1, r_1], [\ell_2, r_2], \cdots, [\ell_s, r_s]$ where $r_i < \ell_{i+1} - 1$ for $i = 1, 2, \cdots, s-1$. The number of segments in L may be as many as the number of distinct labels in L. Yet, if L is the critical set of visible labels of R, there are at most $\sqrt{n}$ disjoint segments.

LEMMA 5.1. Suppose $L(R) = \cup_{i=1}^{s}[\ell_i, r_i]$ where $r_i < \ell_{i+1} - 1$. Then $\ell_1 + \ell_2 + \cdots + \ell_s < n$ and $s < \sqrt{n}$.

Proof. We will prove that R contains s disjoint subtrees having at least ℓ_1, ℓ_2, $\cdots$, ℓ_s edges. Thus, $\ell_1 + \ell_2 + \cdots + \ell_s < n$. Because $r_i < l_{i+1} - 1$, $\sum_{i=1}^{s} \ell_i \geq 1 + 3 + \cdots + (2s-1) = s^2$. It follows that $s < \sqrt{n}$.

Let ψ be a critical edge ranking of R such that the restriction of ψ to every subtree of R is also critical. That is, $L(\psi) = L(T) = \cup_{i=1}^{s}[\ell_i, r_i]$. Consider the visible edge that is assigned with the label ℓ_s under ψ. Denote this edge by $e_s = (u, v)$ where u is the parent of v in R. We first prove that the subtree R_v contains at least $\ell_s - 1$ edges.

If $\ell_s = 1$, R_v obviously contains at least $\ell_s - 1$ edges. In the following, we assume that $\ell_s > 1$. Note that the label $(\ell_s - 1)$ is not visible in R. We consider the following two cases:

- **The label $(\ell_s - 1)$ is visible in R_v:** The restriction of ψ to R_v is critical. Since a critical edge ranking of R_v does not use a label greater than the number of edges in R_v, there must exist at least $(\ell_s - 1)$ edges in R_v.
- **The label $(\ell_s - 1)$ is not visible in R_v:** The following argument shows that this case cannot happen. The label $(\ell_s - 1)$ is not visible in R, and every label on the path from u to the root of R is at most $\ell_s - 2$. It follows that $\ell_s - 1$ is also not visible in R_u. Let us relabel the edge e_s with $(\ell_s - 1)$. The we obtain an edge ranking of R with a visible set of labels that is lexicographically smaller than $L(\psi)$. A contradiction occurs.

In short, the subtree rooted at the edge e_s (i.e. R_s plus the edge e_s) contains at least ℓ_s edges.

Next, we show that R contains another disjoint subtree with ℓ_{s-1} edges. Let T' be the subtree of R formed by deleting all the subtrees rooted at a visible edge with label $\geq \ell_s$. The labels inherited from ψ still forms a critical ranking of T', and $L(T') = \cup_{i=1}^{s-1}[\ell_i, r_i]$. Let e_{s-1} be the visible edge with the label ℓ_{s-1} in T'. Again, we can argue that the subtree rooted at the edge e_{s-1} in T' contains at least ℓ_{s-1} edges.

Repeating the argument above, we can show that there are s disjoint subtrees in R having $\ell_s, \ell_{s-1}, \cdots, \ell_1$ edges.

5.2 Data structures To attain a sub-quadratic time implementation of EDGE-RANKING(R), we would like to manipulate each individual critical set of visible labels segment by segment instead of label by label. We represent the critical set of visible labels of every subtree R_v (i.e. $L(R_v)$) as a search tree X. X is composed of disjoint segments, which partition the labels from 1 to n. Every segment $[\ell, r]$ in X is associated with a flag $f \in \{0, 1, 2\}$; the search key is the ordered pair (f, r). Segments in $L(R_v)$ are put into X as flag-1 segments. The free labels (i.e. labels not in $L(R_v)$) form the flag-0 segments of X.. Flag-2 segments are present only in the process of computing $L(R_v)$).

Operations to be performed on X include: insert a segment, delete a segment, search for the existence (or the predecessor and successor) of a segment, delete the minimum segment, and count the number of free labels between two consecutive flag-1 segments. We require the three operations each to be done in $O(\log n)$ time, and the delete minimum takes $O(1)$ time only. We maintain a count of such free labels explicitly in every pair of consecutive flag-1 segments.

5.3 Implementation of LABELING(R_v) Consider an internal node v of R. Let $u_1, u_2, \cdots, u_d$ be the children of v. Without loss of generality, we assume that the subtree R_{u_1} contains the largest number of nodes among all subtress rooted at the children of v. The input to the procedure LABELING(R_v) consists of d search trees, $X[1]$, $X[2]$, $\cdots$, $X[d]$, representing $L(R_{u_1})$, $L(R_{u_2})$, $\cdots$, $L(R_{u_d})$. Regarding the output, the procedure produces an array $\beta[1..d]$ representing the branch labels and a search tree X representing $L(R_v)$. Note that X inherits all the labels from $X[1], X[2], \cdots, X[d]$ that remain visible from the viewpoint of R_v. To save time, we do not build X starting from scratch. Instead we use $X[1]$ as a basis and merge segments from other input search trees to $X[1]$. At the end, we return $X[1]$ as the output for R_v.

R_{u_1} is the biggest subtree and possibly contains a large number of segments. If we avoid copying the content of $X[1]$, we potentially save a lot of time. Our goal is to ensure the time for executing LABELING(R_v) does not depend on the total number of segments or labels in the input trees. Instead we want the total time to be charged to the following number of tree operations.

- s—the total number of segments in $L(R_{u_2}), \cdots, L(R_{u_d})$;
- h—the total number of segments which are

in $L(R_{u_1}), L(R_{u_2}), \cdots, L(R_{u_d})$ but which are no longer visible when LABELING(R_v) stops.

- b—the number of segments formed by the branch labels computed by LABELING(R_v).

The following lemma captures the main result of our implementation.

LEMMA 5.2. It takes $O(d + (s + h + b) \log n)$ time to execute LABELING(R_v).

Proof. See Appendix.

THEOREM 5.1. It takes $O(n \log n)$ time to execute EDGE-RANKING(R), where n is the number of nodes in R.

Proof. To ease our discussion, we add a subscript v to the variables d, s, h, b from now on. Lemma 5.2 implies that the algorithm EDGE-RANKING on input R runs in $O(\sum_{v \in R} d_v + \sum_{v \in R}(s_v + h_v + b_v) \log n)$ time. Obviously, $\sum_{v \in R} d_v$ is bounded by n, and $b_v \leq d_v$. As regards h_v, we observe that once an edge becomes not visible in the subtree R_v, it will remain not visible in any subtree rooted at an ancestor of v. Thus, $\sum_{v \in R} h_v$ is also bounded by n. It remains to show that $\sum_{v \in R} s_v$ equals $O(n)$. Suppose the subtrees rooted at the children of a node v contain $n_1, n_2, \cdots, n_{d_v}$ nodes. W.L.O.G., assume $n_1 = \max\{n_1, n_2, \cdots, n_{d_v}\}$. By Lemma 5.1, $s_v \leq \sqrt{n_2} + \cdots + \sqrt{n_{d_v}}$. We can further show that $\sum_{v \in R}(\sqrt{n_2} + \cdots + \sqrt{n_{d_v}})$ is at most $4n - 4\sqrt{n}$. Therefore, $\sum_{v \in R} s_v = O(n)$.

6 $O(n)$ time algorithm

In the previous section, we show that the algorithm EDGE-RANKING can run in time

$$O\left(\sum_{v \in R} d_v + \sum_{v \in R}(s_v + h_v + b_v) \log n\right).$$

Note that $\sum_{v \in R} s_v$, as well as $\sum_{v \in R} h_v$ and $\sum_{v \in R} b_v$, is equal to $\Theta(n)$. The main observation that leads to a linear time algorithm is that most of the segments dealt with by the algorithm contain only small labels. More specifically, a segment $[\ell, r]$ is said to be *small* if $r < \log^2 n$; otherwise $[\ell, r]$ is said to be *big*. We decompose the value s_v into two values, say, s'_v and s''_v, corresponding to the number of small and big segments. Similarly, h_v can be decomposed into h'_v and h''_v, and b_v into b'_v and b''_v.

We use data structures other than simple search trees so that small segments can be handled more efficiently. We devise a hybrid representation that uses different data structures for small segments and big segments. This hybrid representation is able to support search, insert and delete operations in $O(1)$ time for small segments, and in $O(\log n)$ time for big segments. With such a hybrid representation, the time complexity of the algorithm EDGE-RANKING can be improved to

$$O\left(\sum_{v \in R}(d_v + s'_v + h'_v + b'_v) + \sum_{v \in R}(s''_v + h''_v + b''_v) \log n\right).$$

Since s'_v, h'_v, and b'_v are bounded by s_v, h_v, and b_v, respectively, $\sum_{v \in R}(s'_v + h'_v + b'_v)$ equals $O(n)$. The values of $\sum_{v \in R} s''_v$ and $\sum_{v \in R} h''_v$ have a simple relationship with the value of $\sum_{v \in R} b''_v$. We are able to show that $\sum_{v \in R} s''_v \leq \log n \sum_{v \in R} b''_v$, while $\sum_{v \in R} h''_v \leq \sum_{v \in R} b''_v$. Moreover, $\sum_{v \in R} b''_v = O(\frac{n}{\log^2 n})$. Therefore, $\sum_{v \in R}(s''_v + h''_v + b''_v)$ equals $O(\frac{n}{\log n})$. Thus, we can obtain a linear time implementation of the algorithm EDGE-RANKING.

THEOREM 6.1. An optimal edge ranking of a tree with n nodes can be computed in $O(n)$ time.

References

[1] H. L. Bodlaender, J. S. Deogun, K. Jansen, T. Kloks, D. Kratsch, H. Müller, and Zs. Tuza, *Rankings of graphs*, SIAM J. Discrete Math. To appear.

[2] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks, *Approximating treewidth, pathwidth, frontsize, and shortest elimination tree*, J. Algorithms, 18 (1995), pp. 238-255.

[3] P. de la Torre, R. Greenlaw, and A. A. Schäffer, *Optimal Edge Ranking of Trees in Polynomial Time*, Algorithmica, 13 (1995), pp. 529-618.

[4] A. V. Iyer, H. D. Ratliff, and G. Vijayan, *Optimal node ranking of trees*, Inform. Process. Lett., 28 (1988), pp. 225-229.

[5] A. V. Iyer, H. D. Ratliff, and G. Vijayan, *On an edge ranking problem of trees and graphs*, Discrete Appl. Math., 30 (1991), pp. 43-52.

[6] T. W. Lam and F. L. Yue, *The NP-completeness of Edge Ranking*, In Proc. of International Conference on Algorithms, International Computer Symposium'96, Taiwan, 1996, pp. 43-50.

[7] M. Katchalski, W. McCuaig, and S. Seager, *Ordered colourings*, Discrete Math., 142 (1995), pp. 141-154.

[8] D. C. Llewellyn, C. Tovey, and M. Trick, *Local optimization on graphs*, Discrete Appl. Math., 23 (1989), pp. 157-178.

[9] C. E. Leiserson, *Area efficient graph layouts for VLSI*, ACM Doctor. Diss. Awards, MIT Press, Cambridge, Massachusetts, 1983.

[10] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal of Matrix Analysis and Applications, 11 (1990), pages 134-172.

[11] A. Pothen, *The Complexity of Optimal Elimination Trees*, Technical Report CS-88-13, The Pennsylvania State University, 1988.

[12] A. A. Schäffer, ***Optimal node ranking of trees in linear time***, Inform. Process. Lett., 33 (1989/90), pp. 91-96.

[13] X. Zhou and T. Nishizeki, ***Finding optimal edge-rankings of trees***, in Proc. of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 122-131. A correct algorithm was given in Advances in Computing Techniques, Algorithms, Databases and Parallel Processing, World Scientific, pp. 23-35.

Appendix

This section gives the details of the procedure LABELING(R_v) so as to attain the time complexity stated in Lemma 5.2.

It is worth mentioning that the procedure gradually puts some segments of labels from $X[2], X[3], \cdots, X[d]$ into $X[1]$. More precisely, Just before the current conflict, say, α, is resolved, labels originating from $L(R_{u_2}), \cdots, L(R_{u_d})$ that are greater than α and that are still visible from the viewpoint of R_v must have been inserted into $X[1]$. Also, $X[1]$ is overloaded to include all labels that are currently assigned to the branches; these labels are represented by segments with flag 2.

Procedure LABELING(R_v)

Input: $X[1], X[2], \cdots, X[d]$

Output: The branch labels $\beta[1], \cdots, \beta[d]$ and X

- *Step I: Initialization.* We obtain a sorted list S of segments with flat equal to 1 from $X[2], X[3], \cdots, X[d]$ in descreasing order of the right limits of the segments (tie is broken with respect to the branch indices). In order to differentiate segments from different search trees, segments originating from the the same tree, say, $X[i]$, are linked together in S and each store the index i.

  Next, we initialize all entries in $\beta[1..d]$ to zero.

- **while** there exists $\beta[i] = 0$ **do**

  - *Step II*: Locating the maximum conflict.

    Case 1: S is not empty. Let $[\ell, r]$ be the segment in S with the biggest key. To locate the maximum conflict, we examine whether $[\ell, r]$ overlaps with the next segment in S or a segment in $X[1]$. If so, let α be the maximum label where the conflict occurs, and let K be all the branches i that give rise to the conflicts. Note that $\alpha \leq r$. If $\alpha < r$, the segment $[\alpha + 1, r]$ is currently visible from the viewpoint of R_v but not R_{u_1}. To maintain the invariant of $X[1]$, we split $[\ell, r]$ into two segments: $[\ell, \alpha]$ remains in S and $[\alpha + 1, r]$ is inserted into $X[1]$ as a flag-1 segment.

    If $[\ell, r]$ does not overlap with any segment, the maximum conflict occurs at a point less than ℓ. We remove $[\ell, r]$ from S, insert it to $X[1]$ as a flag-1 segment. Then we repeat Step II again.

    Case 2: S is empty. Set $\alpha = 0$, and set K to the set of branches i for which $\beta[i] = 0$.

  - *Step III: Counting suitable free labels.* If all flag-1 segments stored in $X[1]$ have right limits less than or equal to α, set w to k. Otherwise, locate the smallest flag-1 segment $[x, y]$ in $X[1]$ such that $y \geq \alpha$. Let $\gamma = \max(x, \alpha + 1)$. If $\gamma = \alpha + 1$, then set w to 0; otherwise, compute the number of free labels w in the interval $[\alpha + 1, \gamma - 1]$.

  - **if** ($\alpha > 0$ and $w < k - 1$) or ($\alpha = 0$ and $w < k$) **then**

    Step IV: Freeing more free labels. Let x be the smallest free label in $X[1]$ bigger than γ. Search $X[1]$ to locate the flag-1 segment $[a, b]$ with the largest right limit smaller than x. If $[a, b]$ originates from $X[j]$, set $\beta[j]$ to x.

    else

    Steps Va and Vb: Resolving conflicts. If $\alpha = 0$, we delete the k smallest free labels bigger than α from $X[1]$ and assign them to the branches in K. If $\alpha > 0$, we need to determine the branch i_0 in K that minimizes the set Cover($\alpha, \hat{L}_{i_0}$) lexicographically. Here, we use a brute force way. We examine S and $X[1]$ to compare, for all $i \in K$, the flat-1 segment originating from $X[i]$ that contain the label α; i_0 should correspond to the segment with the largest left limit. In case there is a tie, we further examine the next smaller segments until i_0 can be determined. Then we delete the $k - 1$ smallest free labels bigger than α from $X[1]$ and assign them to the $k - 1$ edges in K excluding i_0.

  - *Step VI: Removing labels no longer visible.* For each i such that $\beta[i]$ has just received a new value in Steps IV, Va or Vb, we update $X[1]$ or S to delete all flag-1 segments $[a, b]$ with labels $< \beta[i]$ and branch index i.

    If i is in K, we delete i from K. If K still contains two or more indices, the conflict at α has not been resolved competely and we can jump to Step III direct for another attempt.

- *Step VII: Assembling visible labels.* Now all labels in the critical set of R_v are represented by segments with a flag equal to 1 or 2 in $X[1]$. For each flag-1 segment originating from $X[2], \cdots, X[d]$ and are currently found in $X[1]$, if it comes in adjacent with other flag-1 segments in $X[1]$, merge them into one. For every flag-2 segment, change its flag to 1 and merge the segment with other flag-1 segments that comes in adjacent, if any. Return $X[1]$.