Computing optimal linear layouts of trees in linear time

Konstantin Skodinis

University of Passau,
94030 Passau, Germany,
e-mail: skodinis@fmi.uni-passau.de

Abstract. We present a linear time algorithm which, given a tree, computes a linear layout optimal with respect to vertex separation. As a consequence optimal edge search strategies, optimal node search strategies, and optimal interval augmentations can be computed also in $O(n)$ for trees. This improves the running time of former algorithms from $O(n \log n)$ to $O(n)$ and answers two related open questions raised in [7] and [15]. [1]

1 Introduction

The *vertex separation* of graphs has been introduced in [13] in the form of a *vertex separator game*. It is an important concept with theoretical and practical value. Its closer relation to the black-white pebble game of [6] constitutes the theoretical value in complexity theory. Applications in certain layout problems as Weinberger arrays, gate matrix layout, and PLA-folding constitute the practical value in VLSI, see [16]. The *vertex separation* of a graph can be defined using *linear layouts*, see [7]. A linear layout, or simply a layout, L of a graph G is a permutation of the vertices of G. Intuitively L describes how the vertices are to be laid out along a horizontal line. The i-th *vertex cut* of L defines how many of the first i vertices in L are adjacent to a vertex lying right to the i-th vertex in L. The vertex separation of L is the maximum over all vertex cuts of L. The vertex separation of a graph G, $vs(G)$, is the minimum vertex separation of its layouts. L is *optimal* if its vertex separation is $vs(G)$ and therefore as small as possible. Recently the vertex separation of a graph has received more attention due to its closer relation to several well known graph parameters as *edge search number*, *node search number*, *interval thickness*, and *path-width*.

Edge searching has been introduced in [18]. We regard the edges of a graph G as pipes contaminated by a gas. A team of searchers (clearers) has to clear all edges of G. To do so the team has to find an *edge search strategy* which is a sequence of the following activities: (i) place a searcher on a vertex as a guard, (ii) remove a searcher from a vertex, and (iii) slide a searcher along an edge. A contaminated edge becomes cleared if one of its endpoints is guarded and a searcher slides along the edge from the guarded endpoint to the other endpoint of the edge. In [12] it has been shown that recontamination can be disallowed without increasing the search number of any graph. The *edge search number* of G, $es(G)$, is the minimum number of the searchers needed

[1] It should be clear that there are some published papers in which similar results are claimed. Later it turned out that the suggested algorithms do not run in $O(n)$ as stated but in $O(n \log n)$.

to clear G. An edge search strategy of G is *optimal* if it needs the smallest possible number of searchers (this number is $es(G)$).

Node searching is another version of graph searching and has been introduced in [11]. In this slightly different version the third activity of the edge searching disappears. An edge is cleared once both endpoints are simultaneously guarded by searchers. The *node search number* of G is denoted by $ns(G)$.

Interval thickness has been introduced in [10]. Given a set of intervals I of the real line the *interval graph* of I is obtained by representing every interval by a vertex and every intersection of two intervals by an edge between their corresponding vertices. The interval thickness of a graph G, $\theta(G)$, is the smallest maximum clique over all interval supergraphs of G. An interval set I is an *optimal interval augmentation* of G if the interval graph H of I is a supergraph of G and the maximum clique of H has the smallest possible size $\theta(G)$.

Path-width has been introduced in [19, 20]. The *path-decomposition* of a graph G is a sequence $D = X_1, X_2, \ldots, X_r$ of vertex subsets of G, such that (i) every edge of G has both ends in some set X_i and (ii) if a vertex of G occurs in some sets X_i and X_j, $i < j$, then the same vertex occurs in all sets X_k with $i < k < j$. The *width* of D is the maximum number of vertices in any X_i minus 1. The *path-width* of G, $pw(G)$, is the minimum width over all path-decompositions of G. A path-decomposition of G is *optimal* if its width is $pw(G)$ which is the smallest possible.

Rather surprisingly all notions mentioned above are closely related to each other. We will explain this in the following discussion: In [7] it has been shown that for every graph G $vs(G) \leq es(G) \leq vs(G) + 2$. The authors have also shown that $es(G) = vs(G')$, where G' is the 2-expansion of G. G' is obtained from G by replacing every edge $\{u, v\}$ by the edges $\{u, x\}, \{x, y\}$, and $\{y, v\}$ where x and y are two new vertices. Furthermore an algorithm has been presented which (by using a suitable data structure) transforms a given optimal layout L' of G' into an optimal edge search strategy S of G in linear time. Since the size of L' and G' is $O(size(G))$ we imply that for every graph the computation of its optimal edge search strategy requires no more time than the computation of its optimal layout. In [11] it has been shown that $ns(G) = vs(G) + 1$. The authors have also presented an algorithm which (by using a suitable data structure) transforms a given optimal layout into an optimal node search strategy in linear time. In [10] it has been shown that $\theta(G) = ns(G)$ and that a given optimal node search strategy can be transformed into an optimal interval augmentation in linear time. In [9] it has been shown that $pw(G) = vs(G)$. Additionally an algorithm has been presented which (by using a suitable data structure) transforms a given optimal layout into an optimal path-decomposition in linear time in the output size. Here the transformation time is not necessarily linear in the input size since the size of an optimal path-decomposition may be $pw(G)$ times larger than the size of an optimal layout.
Summarizing we get: For every graph G the above parameters are almost the same. Moreover given an input graph G the computation of an optimal edge search strategy, an optimal node search strategy, and an optimal interval augmentation of G is possible in $O(t(n))$ time where $t(n)$ is the time required by the computation of an optimal layout of G. The computation of an optimal path-decomposition is possible in time $O(t(n) + d)$, where d is the size of the computed path-decomposition.

Unfortunately given a graph G and an integer k the problems whether or not $vs(G) \leq k$ and $es(G) \leq k$ are both NP-complete, see [13] and [15], respectively. By the discussion above we obtain that the problems whether or not $ns(G) \leq k$, $\theta(G) \leq k$, and $pw(G) \leq k$ are also NP-complete. They remain NP-complete even for chordal graphs [8], starlike graphs [8], and for planar graphs with maximum degree three [17, 14]. For some special graphs the above problems are solvable in polynomial time. Examples are cographs [4], permutation graphs [3], and graphs of bounded tree-width [2]. Especially if k is fixed then the problems can be solved in $O(n)$ (but exponential in k) [1].

Here we consider trees. It has been shown in [7] that the vertex separation and in [15] that the edge search number can be computed in $O(n)$. Thus the node search number, the interval thickness and the path-width can also be computed in $O(n)$. However the computation of optimal layouts, optimal edge search strategies, optimal node search strategies, optimal interval augmentations, and optimal path-decompositions is more complicated. In [7] an algorithm is given for optimal layouts and in [15] for optimal edge strategies. These algorithms needing $O(n \log n)$ time were the fastest known so far. The authors have raised the question whether this time can be reduced to $O(n)$.

In this paper we establish an algorithm computing an optimal layout of an input tree in $O(n)$. As a consequence of this result and the discussion above the computation of an optimal edge search strategy, an optimal node search strategy, and an optimal interval augmentation of an input tree can be done in $O(n)$. Especially the computation of an optimal path-decomposition can be done in linear time in the output size.

2 Preliminaries

We consider undirected graphs $G = (V, E)$. For convenience loops and multiple edges are disallowed. A *(linear) layout* of G is a one to one mapping $L : V \to \{1, 2, \dots, |V|\}$. For an integer $1 \leq i \leq |V|$ the i-th cut of L, denoted by $cut_L(i)$, is the number of vertices which are mapped to integers less than or equal to i and adjacent to a vertex mapped to an integer larger than i. Formally $cut_L(i) = |\{ u \mid \exists (u, v) \in E \text{ with } L(u) \leq i \text{ and } L(v) > i \}|$. The *vertex separation* of G with respect to L, denoted by $vs_L(G)$, is the maximum over all cuts of L, i.e., $vs_L(G) = \max\limits_{1 \leq i \leq |V|} \{ cut_L(i) \}$. For convenience $vs_L(G)$ is also called the vertex separation of L. The *vertex separation* of G, denoted by $vs(G)$, is the minimum vertex separation over all layouts of G, i.e., $vs(G) = \min\limits_{L} \{vs_L(G)\}$. A layout L of G is *optimal* if $vs(G) = vs_L(G)$.

In this paper we deal with rooted trees. Let T be a tree and u a vertex of T. The *branches* of T at u are the tree components obtained by the removal of u from T. $T[u]$ denotes the subtree of T rooted at u and containing all descendants of u in T. The subtrees rooted at the sons of u are called *the subtrees of* u. Let $v_1, \dots, v_k$ be (arbitrary) vertices of a subtree $T[u]$. $T[u; v_1, \dots, v_k]$ denotes the tree obtained from $T[u]$ by removing $T[v_1], \dots, T[v_k]$. Notice that $T[u; v_1, \dots, v_k]$ is rooted at u.

In [7] the following lemma has been shown.

Lemma 1. *Let T be a tree and $k \geq 1$ an integer. Then $vs(T) \leq k$ iff for all vertices u of T at most two branches of T at u have vertex separation k and all other branches have vertex separation $\leq k - 1$.*

Corollary 1. *Let T be a tree with n vertices. Then $vs(T) \leq \log_3 n + 1 \leq \log n + 1$.*

Next we recall the definition of critical vertices and vertex labelling, see [7].

Definition 1. *Let T be a tree with $vs(T) = k$. A vertex v is k-critical in T if there are two sons v_1 and v_2 of v with $vs(T[v_1]) = vs(T[v_2]) = k$.*

Observe that by Lemma 1 the vertex v in the above definition can not have more than two sons with vertex separation k. By the same lemma we obtain:

Lemma 2. *Let T be a tree with $vs(T) = k$. Then at most one vertex in T is k-critical.*

The following labelling technique has been introduced in [7]. Similar methods were used in [21, 5, 15].

Definition 2. *Let T be a tree rooted at u with $vs(T) = k_1$. A list of integers $l = (k_1, k_2, \ldots, k_p)$ with $k_1 > k_2 > \ldots > k_p \geq 0$ is a label of u if there is a set of vertices $\{v_1, \ldots, v_{p-1}, v_p = u\}$ in T such that*

(i) For $1 \leq i < p$, $vs(T[u; v_1, \ldots, v_i]) = k_{i+1}$,
(ii) For $1 \leq i < p$, v_i is a k_i-critical vertex in $T[u; v_1, \ldots, v_{i-1}]$,
(iii) either v_p $(= u)$ is the k_p-critical vertex in $T[u; v_1, \ldots, v_{p-1}]$ or there is no k_p-critical vertex in $T[u; v_1, \ldots, v_{p-1}]$.

We call $v_1, \ldots, v_{p-1}, v_p (= u)$ the vertices corresponding to l.

Notice that the vertex separation of T is the largest integer occurring in l. If v_i is k_i-critical (k_i-noncritical) in $T[u; v_1, \ldots, v_{i-1}]$ then we say that k_i is critical (noncritical) in l. By definition all v_i are critical in l except possibly v_p $(= u)$. By Lemma 2 it is easy to see that the label l and the vertices corresponding to l are unique.

Lemma 3. *Let T be a tree rooted at u. The label l of u and the vertices corresponding to l are unique.*

Let T be a tree rooted at u. In [7] a linear algorithm is given computing the label l of u. Since the vertex separation is the largest integer occurring in l this algorithm also computes the vertex separation of T in linear time. The idea of the algorithm is to compute l bottom up. Let $u_1, \ldots, u_d$ be the sons of u and l_i the label of u_i in $T[u_i]$ for $1 \leq i \leq d$. The label l is computed by recursively computing the labels l_i and then by combining them. By definition the label of the leaves is (0). It is interesting to note that by the computation of l some of the old labels $l_1, \ldots, l_d$ must be destroyed. However the computation of an optimal layout of T is more complicated. The best known algorithm needs $O(n \log n)$ time [7] where it was stated as an open question whether or not there exists an $O(n)$ layout algorithm. In the following we give an affirmative answer.

3 Optimal layouts in linear time

We establish an $O(n)$ algorithm computing optimal layouts of trees. The idea is to compute the layout of a tree by first computing the layouts of the subtrees of its root and then by combining them. First we show how to compute an optimal layout of certain trees (we call them simple trees) using the optimal layouts of its subtrees. Then we extend this to arbitrary trees. For convenience we define a layout by a list of vertices and use the concatenation operator $\&$ of lists which makes our claims more readable.

3.1 Simple trees and their optimal layouts

Definition 3. *A tree is k-simple if the label of its root is $l = (k)$. Let T be a k-simple tree. T is called noncritical if k is noncritical in l. Otherwise T is called critical.*

Intuitively, T is noncritical k-simple if $vs(T) = k$ and T contains no k-critical vertex. T is critical k-simple if $vs(T) = k$ and the root of T is the (unique) k-critical vertex. Notice that every noncritical 0-simple tree is a singleton and that there are no critical 0-simple trees. The next two lemmas are more or less consequences of Definition 3.

Lemma 4. *Let T be a noncritical k-simple tree rooted at u. Then at most one of the subtrees of u in T is noncritical k-simple and all other subtrees have vertex separation $\leq k - 1$.*

Lemma 5. *Let T be a critical k-simple tree rooted at u. Then exactly two of the subtrees of u in T are noncritical k-simple and all other subtrees have vertex separation $\leq k-1$.*

We will construct an optimal layout of a tree by efficiently combining the optimal layouts of the subtrees. In order to do so the optimal layouts of the subtrees must have a suitable form. This leads to the introduction of left extendable, right extendable, and sparse layouts. For a tree T let T^w be the tree obtained from T by adding a new vertex w and an edge between w and the root of T.

Definition 4. *Let L be a layout of a tree T. If T is a singleton then L is both left and right extendable. Otherwise L is left extendable if $vs_{(w)\&L}(T^w) = vs_L(T)$ and right extendable if $vs_{L\&(w)}(T^w) = vs_L(T)$.*

Intuitively, L is left (right) extendable if the maximal vertex cut of L does not increase when we add an edge between root u and a new vertex w left (right) to L.

Definition 5. *Let L be a layout of a tree T rooted at u. L is sparse if no vertices v and w in T with $L(v) < L(u) < L(w)$ are adjacent in T.*

Intuitively, L is sparse if no vertex left to u is adjacent to a vertex right to u in L.

Lemma 6. *Every noncritical k-simple tree T has both an optimal left extendable layout lL and an optimal right extendable layout rL.*

Proof. Let u be the root of T and $u_1, \ldots, u_d$ the sons of u. Let L_i be an optimal layout of the subtree $T[u_i]$ for $1 \leq i \leq d$. The proof is by induction on the height h of T.

For $h = 0$ the tree T is a singleton and by Definition 4 the claim is clear.

For the induction step consider T of height $h + 1$. By Lemma 4 at most one subtree of u is noncritical k-simple and all other subtrees have vertex separation at most $k-1$. If no subtree of u is noncritical k-simple then the layout of T in Fig. 1 (a) is both optimal left and optimal right extendable. If a subtree of u, say $T[u_1]$, is noncritical k-simple then let lL_1 be an optimal left and rL_1 an optimal right extendable layout of $T[u_1]$. Both exist by the induction hypothesis. Then the layout lL and rL of T in Fig. 1 (b) and 1 (c) is optimal left extendable and optimal right extendable, respectively.

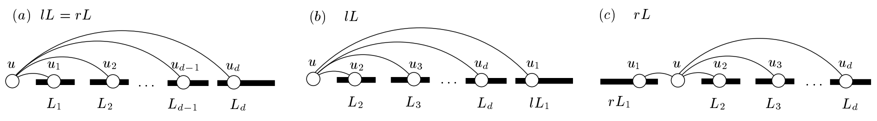

Fig. 1. Optimal extendable layouts of noncritical simple trees. The thick lines represent layouts.

Lemma 7. *Every critical k-simple tree T has an optimal sparse layout sL.*

Proof. Let u be the root of T and $u_1, \ldots, u_d$ the sons of u. By Lemma 5 exactly two subtrees of u, say $T[u_1]$ and $T[u_d]$, are noncritical k-simple and all other subtrees have vertex separation $\leq k-1$. By Lemma 6, $T[u_1]$ has an optimal right extendable layout rL_1 and $T[u_d]$ has an optimal left extendable layout lL_d. Let L_i be an optimal layout of the subtree $T[u_i]$ for $2 \leq i \leq d-1$. Then the layout sL in Fig. 2 is an optimal sparse layout of T.

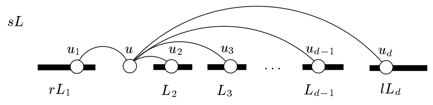

Fig. 2. Optimal sparse layouts of critical simple trees. The thick lines represent layouts.

Remark 1. The proofs of the last two lemmas provide construction schemes for optimal extendable layouts of noncritical simple trees and for optimal sparse layouts of critical simple trees. For the constructions only the optimal extendable layouts of the noncritical simple subtrees and arbitrary optimal layouts of the remaining subtrees are needed.

3.2 Arbitrary trees and their optimal layouts

The main idea is to partition a tree T into smaller simple trees and to define the coarse layout of T as the list of the optimal (extendable or sparse) layouts of the smaller trees.

Definition 6. *Let T be a tree rooted at u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. Let $v_1, \ldots, v_p$ be the vertices corresponding to l, see Definition 2. The parts $\widetilde{T}_{k_i}$, $1 \leq i \leq p$, of T are defined as follows, see Fig. 3.*

(i) $\widetilde{T}_{k_1}$ *is the subtree $T[v_1]$ and*
(ii) $\widetilde{T}_{k_i}$ *is the subtree of $T[u; v_1, \ldots, v_{i-1}]$ rooted at v_i for $2 \leq i \leq p$.*

Lemma 8. *Let T be a tree rooted at u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. Then every part $\widetilde{T}_{k_i}$, $1 \leq i \leq p$, is k_i-simple. Especially, all parts are critical except possibly $\widetilde{T}_{k_p}$ which may be noncritical.*

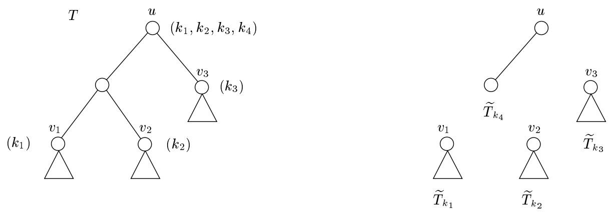

Fig. 3. The parts $\widetilde{T}_{k_i}$ of T

Notation 1 *Let T be a tree and $\widetilde{T}_{k_i}$ a k_i-simple part of T. If $\widetilde{T}_{k_i}$ is noncritical then $l\widetilde{L}_{k_i}$ denotes an optimal left extendable and $r\widetilde{L}_{k_i}$ an optimal right extendable layout of $\widetilde{T}_{k_i}$. If $\widetilde{T}_{k_i}$ is critical then $s\widetilde{L}_{k_i}$ denotes an optimal sparse layout of $\widetilde{T}_{k_i}$. Remember that all these layouts exist since Lemmas 6 and 7.*

Definition 7. *Let T be a tree rooted at u and $l = (k_1, k_2, \dots, k_p)$ the label of u. The coarse layout of T is defined as*

$$C = \begin{cases} (s\widetilde{L}_{k_1}, s\widetilde{L}_{k_2}, \dots, s\widetilde{L}_{k_p}) & \textit{if } k_p \textit{ is critical in } l \\ (s\widetilde{L}_{k_1}, s\widetilde{L}_{k_2}, \dots, s\widetilde{L}_{k_{p-1}}, (l\widetilde{L}_{k_p}, r\widetilde{L}_{k_p})) & \textit{otherwise.} \end{cases}$$

The coarse layout is a flexible representation of an optimal layout. In fact, an optimal layout can be computed from the coarse layout very efficiently. In order to show this we need some new notions: The *focus* of a layout L is a distinguished vertex of L partitioning L into two sublists $pre(L)$ and $suf(L)$. The prefix of L, $pre(L)$, consists of the focus and the vertices of L which are left to the focus. The suffix of L, $suf(L)$, consists of the vertices of L which are right to the focus. Given two layouts L and L' and their focuses, the layout $L \oplus L'$ is defined by $L \oplus L' = pre(L)\&L'\&suf(L)$. The focus of $L \oplus L'$ is the focus of L'. Notice that operation $\oplus$ is associative in the computation of both layouts and their focuses. We will apply $\oplus$ only on layouts of simple trees. The focus of a layout of a simple tree is defined as the root of the simple tree.

Lemma 9. *Let T be a tree and $l = (k_1, k_2, \dots, k_p)$ the label of its root.*

(i) If k_p is critical in l then $L = s\widetilde{L}_{k_1} \oplus s\widetilde{L}_{k_2} \oplus \dots \oplus s\widetilde{L}_{k_p}$ is an optimal layout of T,
(ii) If k_p is not critical in l then $L = s\widetilde{L}_{k_1} \oplus \dots \oplus s\widetilde{L}_{k_{p-1}} \oplus l\widetilde{L}_{k_p}$ is an optimal layout of T.

Now we are going to show how to compute a coarse layout of a tree using the coarse layouts of its subtrees. The following lemma is important.

Lemma 10. *Let T be a tree rooted at u, $l = (k_1, k_2, \dots, k_p)$ the label of u, and $u_1, \dots, u_d$ the sons of u.*

(i) The simple parts $\widetilde{T}_{k_1}, \widetilde{T}_{k_2}, \dots, \widetilde{T}_{k_{p-1}}$ of T are exactly the simple parts of $T[u_1]$, $\dots, T[u_d]$ with vertex separation larger than k_p,

(ii) No subtree $T[u_i]$, $1 \leq i \leq d$, has a critical k_p-simple part.

From Lemma 10 we directly obtain:

Lemma 11. *Let T be a tree rooted at u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. Let $C_1, \ldots, C_d$ be the coarse layouts of the subtrees of u. The components of $C_1, \ldots, C_d$ with vertex separation larger than k_p can be adopted as the first $p-1$ components $s\widetilde{L}_{k_1}, s\widetilde{L}_{k_2}, \ldots, s\widetilde{L}_{k_{p-1}}$ of a coarse layout of T.*

Given the layouts $s\widetilde{L}_{k_1}, s\widetilde{L}_{k_2}, \ldots, s\widetilde{L}_{k_{p-1}}$ they must be sorted according to the integers $k_1, \ldots, k_{p-1}$ for the construction of a coarse layout of T. In order to sort as less elements as possible we first find the subtree of u with the largest vertex separation, say $T[u_a]$. Then we find a large prefix of the coarse layout of $T[u_a]$ which is also a prefix of the coarse layout of T. The layouts of this prefix need not be sorted because they are already sorted in the coarse layout of $T[u_a]$. The next lemma shows how to find such a prefix which is large enough for our later complexity considerations.

Lemma 12. *Let T be a tree rooted at u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. Let $u_1, \ldots, u_d$ be the sons of u and l_i be the label of u_i in $T[u_i]$ for $1 \leq i \leq d$. Let $l_a = (m_1, m_2, \ldots, m_r)$ be the largest and $l_b = (n_1, n_2, \ldots, n_s)$ the second largest label over all l_i, where largest means containing the largest element. Then $(k_1, k_2, \ldots, k_j) = (m_1, m_2, \ldots, m_j)$ where j is the largest index $1 \leq j \leq r-1$ with $m_j > \max\{m_{j+1}, n_1\} + 1$. Hence the first j components of the coarse layout of $T[u_a]$ can be adopted as the first j components of the coarse layout of T.*

Next we show how to compute the last component $s\widetilde{L}_{k_p}$ or $(l\widetilde{L}_{k_p}, r\widetilde{L}_{k_p})$ of the coarse layout of T.

Lemma 13. *Let T be a tree rooted at u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. If the last component k_p of l, the information whether k_p is critical in l, and the coarse layouts of the subtrees of u are given then the last component of the coarse layout of T is computable.*

Proof. Let C be the coarse layout of T, $u_1, \ldots, u_d$ the sons of u, and C_i the coarse layouts of the subtrees $T[u_i]$ for $1 \leq i \leq d$. Let C_i' be the list obtained from C_i by removing all layout components with vertex separation larger than k_p. By Lemma 10 (i) the lists C_i' are the coarse layouts of the subtrees of u in $\widetilde{T}_{k_p}$. There are two cases:
Case 1: $\widetilde{T}_{k_p}$ is noncritical k_p-simple. We have to compute $l\widetilde{L}_{k_p}$ and $r\widetilde{L}_{k_p}$. Assume that u has a noncritical k_p-simple subtree in $\widetilde{T}_{k_p}$, say $\widetilde{T}_{k_p}[u_1]$ (otherwise the proof is similar). By Remark 1 we need the optimal (left and right) extendable layouts of $\widetilde{T}_{k_p}[u_1]$ and the optimal layouts of the other subtrees $\widetilde{T}_{k_p}[u_j]$, $2 \leq j \leq d$. The optimal extendable layouts are in the (only) component of C_1' and the optimal layouts of the other subtrees can be computed from C_j', $1 \leq j \leq d$, using Lemma 9.
Case 2: $\widetilde{T}_{k_p}$ is critical k_p-simple. We have to compute an optimal sparse layout $s\widetilde{L}_{k_p}$ of $\widetilde{T}_{k_p}$. Without loss of generality let $\widetilde{T}_{k_p}[u_1]$ and $\widetilde{T}_{k_p}[u_d]$ be the noncritical k_p-simple subtrees of u in $\widetilde{T}_{k_p}$. By Remark 1 we need the optimal extendable layouts of $\widetilde{T}_{k_p}[u_1]$

and $\widetilde{T}_{k_p}[u_d]$ and the optimal layouts of the other subtrees $\widetilde{T}_{k_p}[u_j]$, $2 \leq j \leq d-1$. The optimal extendable layouts are in the components of C'_1 and C'_2. The optimal layouts of the other subtrees can be computed from C'_j, $2 \leq j \leq d-1$, using Lemma 9.

3.3 Linear time layout Algorithm

We assume that every vertex u of T is marked by the last element k_p of the label $l = (k_1, \ldots, k_p)$ of u in $T[u]$ and by an indicator whether or not k_p is critical in l. Our layout algorithm first recursively computes the coarse layouts of the subtrees of u. Then using the mark of u and the coarse layouts of the subtrees of u it computes the coarse layout of $T[u]$. At the end an optimal layout of T is computed from the coarse layout of T. To achieve linear time we need a suitable representation of the coarse layouts based on the following representation of the labels given in [7]. A label is a list of blocks, where a block contains consecutive integers and is represented by an interval consisting of the endpoints of the block in decreasing order. For example the label $l = (9, 8, 7, 6, 4, 3, 2, 0)$ is represented by $l = ((9, 6), (4, 2), (0, 0))$.
Let $l = ((m_1, n_1), (m_2, n_2), \ldots, (m_{r-1}, n_{r-1}), (m_r, n_r))$ be the representation of the label of u in $T[u]$ where $m_r = n_r$; otherwise we split the last interval (m_r, n_r) into two intervals $(m_r, n_r - 1)$ and (n_r, n_r). The coarse layout of $T[u]$ is represented by a list of r records, one for every interval (m_i, n_i).

The first $r-1$ records consist of four fields. Two fields are of type integer and two are pointers. For $1 \leq i \leq r-1$ the first field of the i-th record is reserved for m_i, the second for n_i, the third for a pointer to the layout $s\widetilde{L}_{m_i,n_i} = s\widetilde{L}_{m_i} \oplus s\widetilde{L}_{m_i-1} \oplus \cdots \oplus s\widetilde{L}_{n_i}$, and the fourth for a pointer to the focus f_i of $s\widetilde{L}_{m_i,n_i}$. For the last r-th record there are two cases. If m_r is critical in l then the r-th record has the same structure as before, but now both integer fields are reserved for m_r since in this case $m_r = n_r$. If m_r is not critical in l then the r-th record has two additional fields which are pointers. Here the first two integer fields are reserved for m_r, the third field is reserved for a pointer to $l\widetilde{L}_{m_r}$, the fourth for a pointer to the focus of $l\widetilde{L}_{m_r}$, the fifth for a pointer to $r\widetilde{L}_{m_r}$ and the sixth for a pointer to the focus of $r\widetilde{L}_{m_r}$. This data structure allows the computation of the prefix and suffix of a layout in constant time.

Now we present our layout algorithm.

```
**function** $optimal\_layout(T : tree, u : root) : layout$;
 $C := compute\_coarse\_layout(T, u)$;
 Compute optimal layout $L$ from $C$ according to Lemma 9
 **return** $L$;

**function** $compute\_coarse\_layout(T : tree, u : root) : coarse\ layout$;
   {Let $u$ be the root of $T$}
   **if** ($T$ contains only $u$) **then** $C := ((0, 0, (u), u, (u), u))$;
                                   **return** $C$
                              **else** {Let $u_1, \ldots, u_d$ be the sons of $u$ in $T$};
                                   **for all** $u_i$ **do** $C_i := compute\_coarse\_layout(T[u_i], u_i)$;
                                   **return** $combine\_coarse\_layouts(u, C_1, \ldots, C_d)$;
```

```
function $combine\_coarse\_layouts(u : vertex; C_1, \ldots, C_d : coarse\ layout)$
                                                                    $: coarse\ layout;$
  {Let $C_t$ be the coarse layout of the subtree of $u$ with the largest vertex separation}
 1: Excise prefix $\bar{C}_t$ from $C_t$ according to Lemma 12;
  {Let $k_p$ be the mark of $u$}
 2: $R :=$ list containing all records $(m, n, X, Y)$ of $C_1, \ldots, C_d$ with $m > k_p$;
 3: Sort all records of $R$ according to their first integer fields;
 4: Compact $R$ by replacing all consecutive records $(m_k, n_k, X_k, Y_k), (m_{k+1}, n_{k+1},$
    $X_{k+1}, Y_{k+1}), \ldots, (m_r, n_r, X_r, Y_r)$ with $n_i = m_{i+1} + 1$ for $k \leq i < r$ by the
    record $(m_k, n_r, X, Y)$, where $X = X_k \oplus X_{k+1} \oplus \cdots \oplus X_r$ and $Y = Y_r$;
 5: Compute the last component $c$ of $C$ according to Lemma 13, using the mark of $u$;
 6: $C := \bar{C}_t \& R \& c$;
 return $C$;
```

Lemma 14. *Given an input tree T $compute\_coarse\_layout$ correctly computes the representation of a coarse layout of T.*

Proof. Let u be the root of T, $u_1, \ldots, u_d$ the sons of u and $l = (k_1, k_2, \ldots, k_p)$ the label of u. The proof is by induction on the height h of T.

For $h = 0, T$ consists only of the root u. In this case $compute\_coarse\_layout(T, u)$ results in $C := ((0, 0, (u), u, (u), u))$ and the claim trivially holds.

Now let T be of height $h+1$. By the induction hypothesis $compute\_coarse\_layout$ $(T[u_i], u_i)$ correctly computes the representations C_i of the coarse layouts of $T[u_i]$. We show that $C := \bar{C}_t \& R \& c$ computed in step 6 is a correct representation of the coarse layout of T. By Lemma 12, $\bar{C}_t$ computed in step 1 can be adopted as the prefix of C. Now consider the list R computed in step 2. First observe that no C_i has a record (m, n, X, Y) with $m > k_p \geq n$, otherwise subtree $T[u_i]$ would have a critical k_p-simple part which contradicts to Lemma 10 (ii). That means the records of R include exactly those layout components of the coarse layouts of the subtrees which have vertex separation larger than k_p. Now, by Lemma 11 and the associativity of $\oplus$, R after sorting in step 3, and compacting in step 4 can be adopted from C. Finally, by Lemma 13 and the associativity of $\oplus$, the record c computed in step 5 can be adopted as the last record of C. Hence $C := \bar{C}_t \& R \& c$ is a correct representation of the coarse layout of T.

By Lemma 14, Lemma 9 and the associativity of $\oplus$ we conclude:

Theorem 2. *The function $optimal\_layout$ correctly computes an optimal layout of an input tree.*

Next we show that the time complexity of our algorithm is linear.

Lemma 15. *Let T be a tree rooted at u and $u_1, \ldots, u_d$ the sons of u. Let C_i, $1 \leq i \leq d$, be the representation of a coarse layout of $T[u_i]$. Let s_i be the ith largest vertex separation of the subtrees of u. The time required by $combine\_coarse\_layouts(u, C_1, \ldots,$ $C_d)$ to compute the representation of the coarse layout of T is $O(d+2s_2+s_3+\ldots+s_d)$, where s_1 does not appear and s_2 appears twice.*

Proof. First observe that $vs(T[u_i])$ is the integer in the first field of the first record of C_i. The time required in step 1 is $O(s_2 + 1)$. We only need to search C_t from the right to the left until we find the first consecutive records (m_k, n_k, X_k, Y_k) and $(m_{k+1}, n_{k+1}, X_{k+1}, Y_{k+1})$ with $n_k > \max\{m_{k+1}, s_2\} + 1$, see also Lemma 12. Step 2 takes time linear in the number of the records in $C_1, \ldots, C_d$, i.e., time $O(d + 2s_2 + s_3 + \ldots + s_d)$ (note that $\bar{C}_t$ is removed from C_t). In step 3 the sorting of the records of R according to the integers in their first fields can be done in $O(s_2 + 1)$ using bucket sort. To see this observe that these integers are (i) all $\leq s_2$ except possibly one (this would occur in the first field of the first record of C_t) and (ii) all pairwise different since Lemma 10(i). The time required in step 4 is $O(s_2 + 1)$ since R contains at most $s_2 + 1$ records. Step 5 clearly can be done in $O(d + 2s_2 + s_3 + \ldots + s_d)$. Finally step 6 takes constant time. Hence the execution time of $combine\_coarse\_layouts(u, C_1, \ldots, C_d)$ is $O(d + 2s_2 + s_3 + \ldots + s_d)$.

Let T be a tree, u a vertex in T, and j an integer. Let $q_{u,j}$ be the number of the vertices v in $T[u]$ having a brother v' such that $vs(T[v]) = vs(T[v']) \geq j$. In a similar way as in the proof of Lemma 3.1 from [7] we can prove the next lemma.

Lemma 16. *Let T be a tree rooted at u. The time required by $compute\_coarse\_layout$ (T, u) is $\leq c(n + \sum_{j=0}^{vs(T)} q_{u,j})$ where c is a constant and n the size of T.*

Proof. Let $u_1, \ldots, u_d$ be the sons of u. The proof is by induction on the height of T.

For $h = 0$, $compute\_coarse\_layout(T, u)$ needs time c_1 where c_1 is the time needed by $C := ((0, 0, (u), u, (u), u))$. Hence the claim holds for every $c \geq c_1$.

Now consider T of height $h+1$. The time needed by $compute\_coarse\_layout(T, u)$ is the time t_1 needed by $combine\_coarse\_layouts(u, C_1, \ldots, C_d)$ plus the time t_2 needed by the calls $compute\_coarse\_layout(T[u_i], u_i)$, $1 \leq i \leq d$. By Lemma 15 and the induction hypothesis we have $t_1 + t_2 \leq c_2(d + 2s_2 + s_3 + \ldots + s_d) + \sum_{i=1}^{d}(cn_i + \sum_{j=0}^{vs(T[u_i])} q_{u_i,j})$, where n_i is the size of $T[u_i]$. Moreover it is easy to verify that

$2s_2 + s_3 + \ldots + s_d + \sum_{i=1}^{d} \sum_{j=0}^{vs(T[u_i])} q_{u_i,j} = \sum_{j=0}^{vs(T)} q_{u,j}$. Defining $c := \max\{c_1, c_2\}$ we get

$t_1 + t_2 \leq c(d + \sum_{i=1}^{d} n_i) + c(2s_2 + s_3 + \ldots + s_d + \sum_{i=1}^{d} \sum_{j=0}^{vs(T[u_i])} q_{u_i,j}) \leq cn + c \sum_{j=0}^{vs(T)} q_{u,j} \leq$

$c(n + \sum_{j=0}^{vs(T)} q_{u,j})$ and the proof is complete.

The following has been proved in [7], page 77.

Lemma 17. *Let T be a tree rooted at u. Then $\sum_{j=0}^{vs(T)} q_{u,j} \leq 4n$ where n is the size of T.*

By Lemmas 16 and 17 we obtain:

Lemma 18. *The time required by $compute\_coarse\_layout(T, u)$ is linear in the size of T.*

By Corollary 1 a coarse layout of T contains at most $\log n$ records, where n is the size of T. Hence given a coarse layout the computation of an optimal layout according to Lemma 9 takes at most $O(\log n)$ time. This and Lemma 18 imply our last theorem.

Theorem 3. *The time complexity of the function $optimal\_layout$ is linear in the size of the input tree.*

References

[1] H. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[2] H. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21:358–402, 1996.

[3] H. Bodlaender, T. Kloks, and D. Kratsch. Treewidth and pathwidth of permutation graphs. *SIAM J. Disc. Math.*, 8:606–616, 1995.

[4] H. Bodlaender and R. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Disc. Math.*, 6:181–188, 1993.

[5] M. Chung, F. Makedon, I. Sudborough, and J. Turner. Polynomial algorithms for the min-cut linear arrangement problem on degree restricted trees. *SIAM J. Comput.*, 14(1):158–177, 1985.

[6] S. Cook and R. Sethi. Storage requirements for deterministic polynomial finite recognizable languages. *J. Comput. System Sci.*, 13:25–37, 1976.

[7] J. Ellis, I. Sudborough, and J. Turner. The vertex separation and search number of a graph. *Inform. Comput.*, 113:50–79, 1994.

[8] J. Gustedt. On the pathwidth of chordal graphs. *Disc. Appl. Math.*, 45:233–248, 1993.

[9] N. Kinnersley. The vertex separation number of a graph equals its path-width. *Inform. Process. Lett.*, 142:345–350, 1992.

[10] L. Kirousis and C. Papadimitriou. Interval graphs and searching. *Disc. Math.*, 55:181–184, 1985.

[11] L. Kirousis and C. Papadimitriou. Searching and pebbling. *Theor. Comp. Science*, 47:205–218, 1986.

[12] A. LaPaugh. Recontamination does not help to search a graph. *J. Assoc. Comput. Mach.*, 40(2):243–245, 1993.

[13] T. Lengauer. Black-white pebbles and graph separation. *Acta Informat.*, 16:465–475, 1981.

[14] F. Makedon and I. Sudborough. On minimizing width in linear layouts. *Disc. Appl. Math.*, 23:243–265, 1989.

[15] N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriou. The complexity of searching a graph. *J. Assoc. Comput. Mach.*, 35(1):18–44, 1988.

[16] R. H. Möhring. Graph problems related to gate matrix layout and PLA folding. In E. Mayr, H. Noltmeier, and M. Syslo, editors, *Computational Graph Theory*, pages 17–51. Springer Verlag, 1990.

[17] F. Monien and I. Sudborough. Min cut is NP-complete for edge weighted trees. *Theor. Comp. Science*, 58:209–229, 1988.

[18] T. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. Lick, editors, *Theory and Applications of Graphs*, pages 426–441. Springer-Verlag, New York/Berlin, 1976.

[19] N. Robertson and P. Seymour. Graph minors. I. Excluding a forest. *J. Comb. Theory Series B*, 35:39–61, 1983.

[20] N. Robertson and P. Seymour. Disjoint paths–A servey. *SIAM J. Alg. Disc. Meth.*, 6:300–305, 1985.

[21] M. Yannakakis. A polynomial algorithm for the min-cut linear arrangement of trees. *J. Assoc. Comput. Mach.*, 32(4):950–988, 1985.